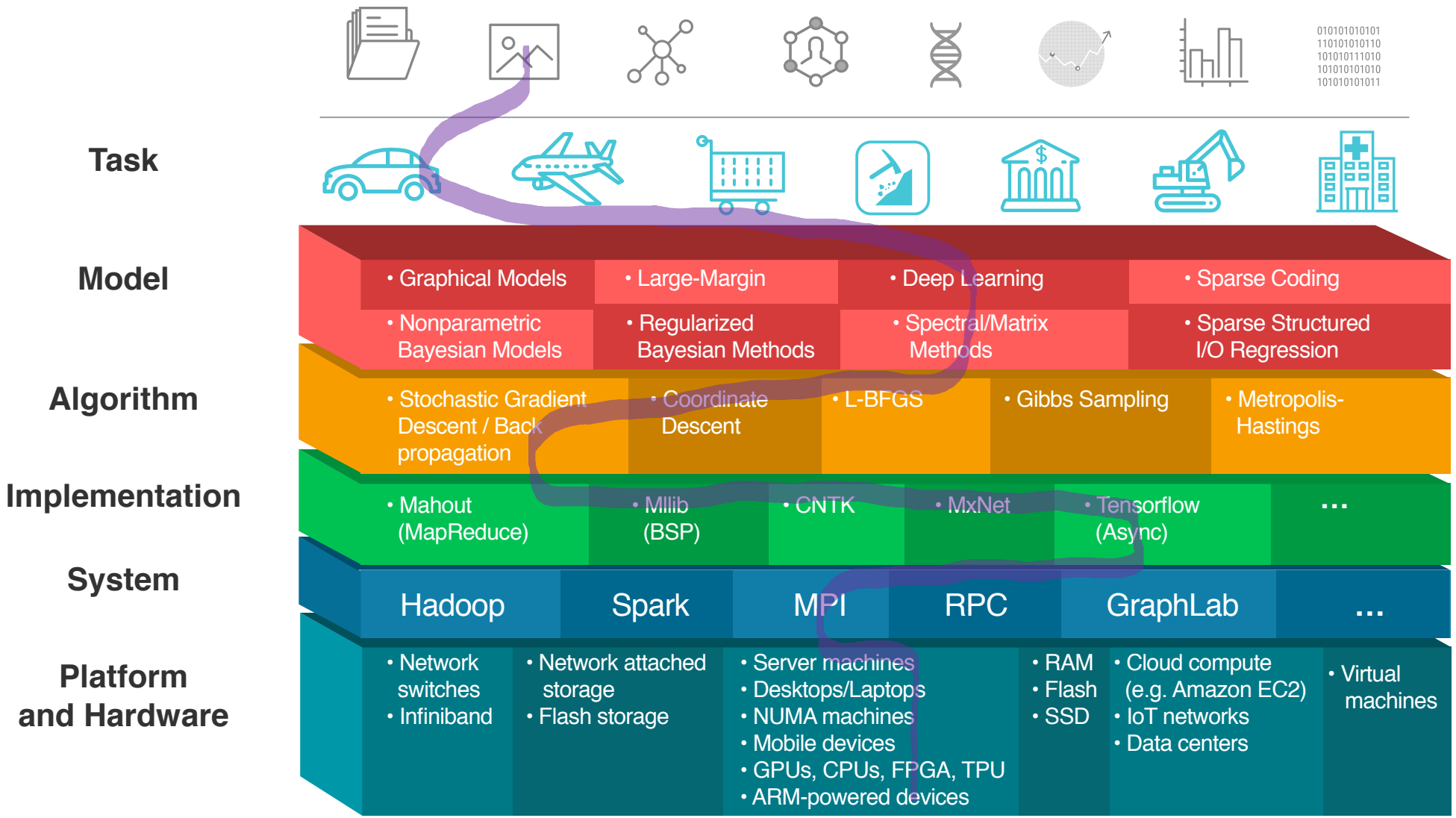
PETUUM

A Statistical Machine Learning Perspective of Deep Learning: Algorithm, Theory, Scalable Computing

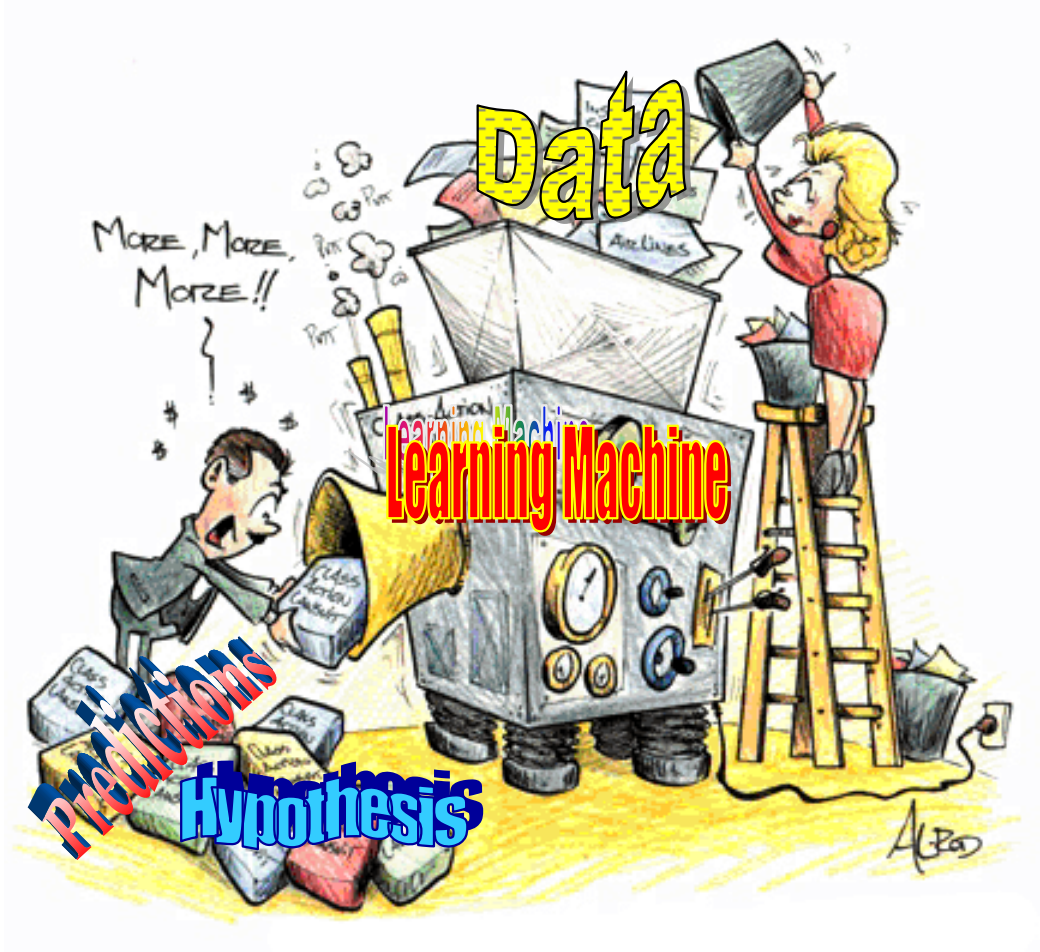
Maruan Al-Shedivat, Zhiting Hu, Hao Zhang, and Eric Xing

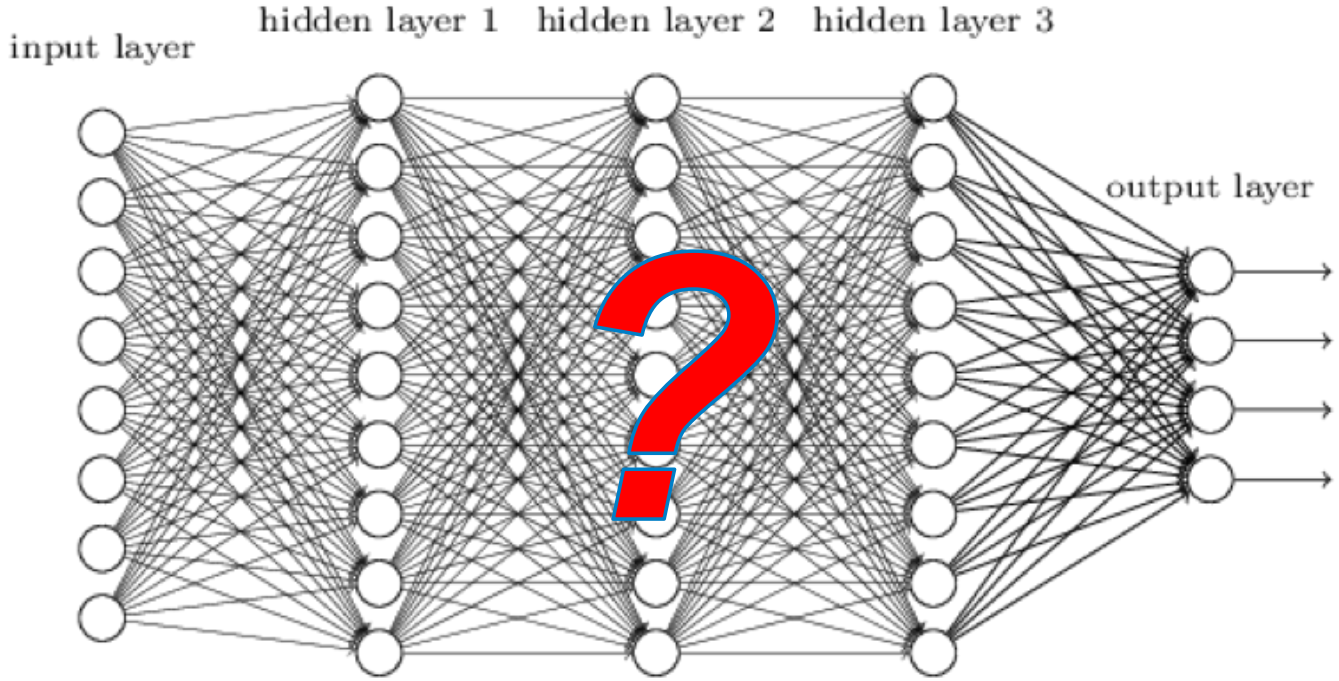
Petuum Inc
&
Carnegie Mellon University

Element of AI/Machine Learning



ML vs DL





Plan

- Statistical And Algorithmic Foundation and Insight of Deep Learning
- On Unified Framework of Deep Generative Models
- Computational Mechanisms: Programming Platforms and Distributed Deep Learning Architectures

PETUUM

Part-I
Statistical And Algorithmic Foundation and Insight of Deep Learning

Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
  - Historical remarks: early days of neural networks
  - Modern building blocks: units, layers, activations functions, loss functions, etc.
  - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
  - Graphical models vs. computational graphs
  - Sigmoid Belief Networks as graphical models
  - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
  - Using outputs of NNs as inputs to GMs
  - GMs with potential functions represented by NNs
  - NNs with structured outputs
- Bayesian Learning of NNs
  - Bayesian learning of NN parameters
  - Deep kernel learning

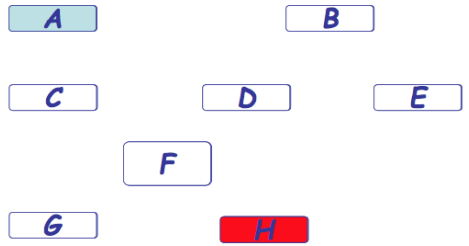
Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
  - Historical remarks: early days of neural networks
  - Modern building blocks: units, layers, activations functions, loss functions, etc.
  - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
  - Graphical models vs. computational graphs
  - Sigmoid Belief Networks as graphical models
  - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
  - Using outputs of NNs as inputs to GMs
  - GMs with potential functions represented by NNs
  - NNs with structured outputs
- Bayesian Learning of NNs
  - Bayesian learning of NN parameters
  - Deep kernel learning

Fundamental questions of probabilistic modeling

- **Representation:** what is the joint probability distr. on multiple variables?

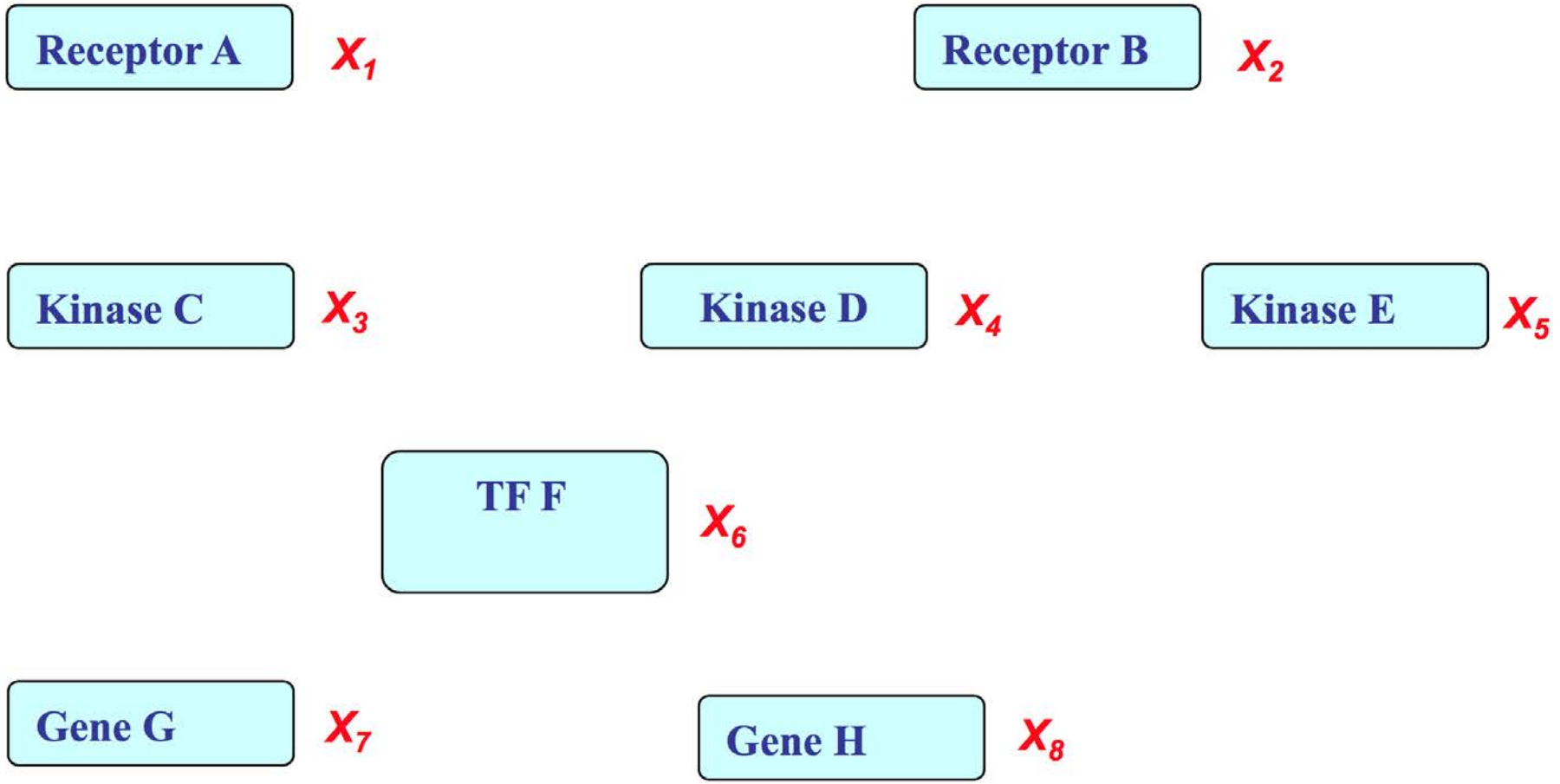
$$P(X_1, X_2, X_3, \ldots, X_d)$$



  - How many state configurations are there?
  - Do they all need to be represented?
  - Can we incorporate any domain-specific insights into the representation?
- **Learning:** where do we get the probabilities from?
  - Maximum likelihood estimation? How much data do we need?
  - Are there any other established principles?
- **Inference:** if not all variables are observable, how to compute the conditional distribution of latent variables given evidence?
  - Computing $P(H|A)$ would require summing over 2^6 configurations of the unobserved variables

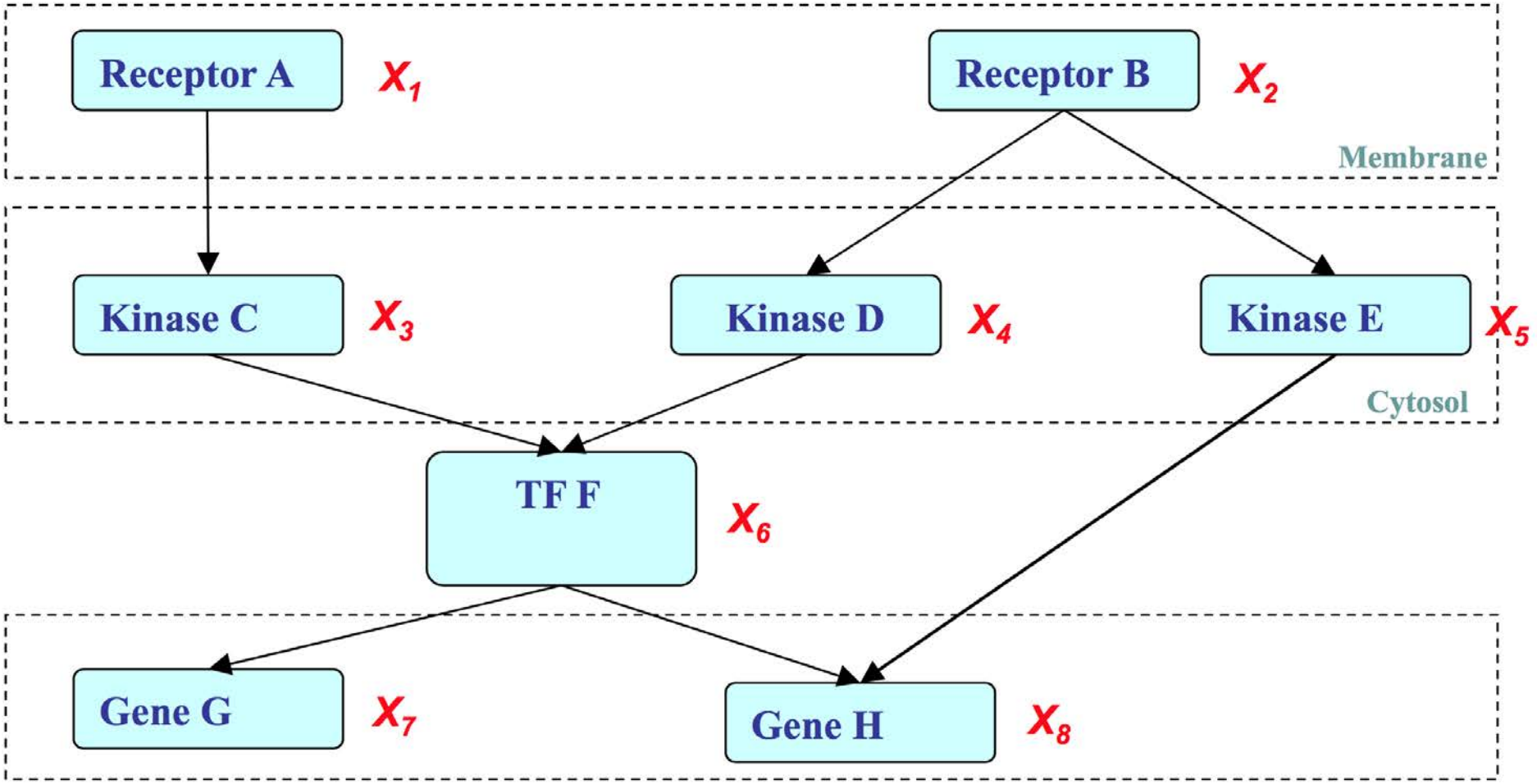
What is a graphical model?

- A possible world of cellular signal transduction



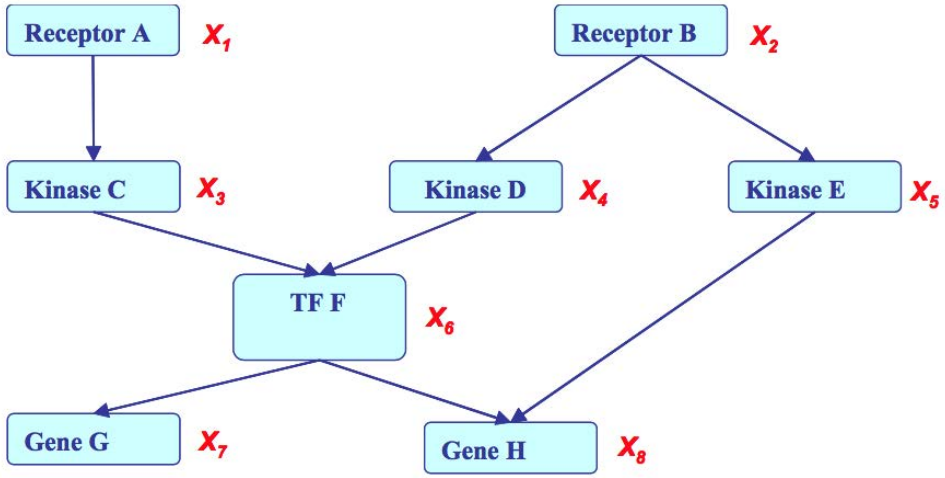
GM: structure simplifies representation

- A possible world of cellular signal transduction



Probabilistic Graphical Models

- If X_i's are **conditionally independent** (as described by a **PGM**), then the joint can be factored into a product of simpler terms



$$P(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) = P(X_1)P(X_2)P(X_3|X_1)P(X_4|X_2)P(X_5|X_2) P(X_6|X_3, X_4)P(X_7|X_6)P(X_8|X_5, X_6)$$

- Why we may favor a PGM?
  - Easy to incorporate domain knowledge and causal (logical) structures
  - Significant reduction in representation cost (2^8 reduced down to 18)

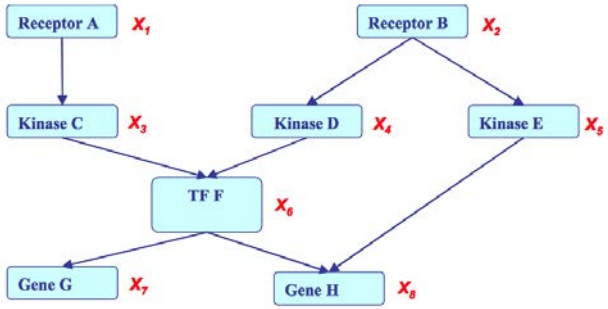
The two types of GMs

$$P(H|V)$$

$$\theta = \text{argmax}_\theta P_\theta(V)$$

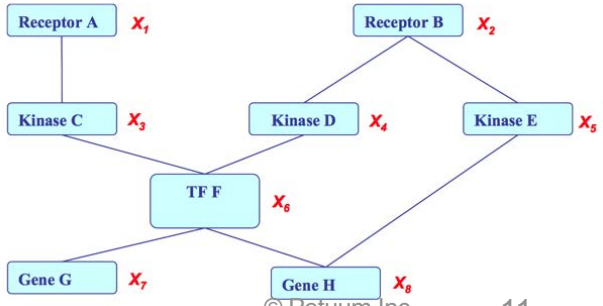
- **Directed edges** assign causal meaning to the relationships (Bayesian Networks or Directed Graphical Models)

$$P(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) =$$
$$P(X_1)P(X_2)P(X_3|X_1)P(X_4|X_2)P(X_5|X_2)$$
$$P(X_6|X_3, X_4)P(X_7|X_6)P(X_8|X_5, X_6)$$



- **Undirected edges** represent correlations between the variables (Markov Random Field or Undirected Graphical Models)

$$P(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) =$$
$$\frac{1}{Z}\exp\{E(X_1) + E(X_2) + E(X_1, X_3) + E(X_2, X_4) + E(X_5, X_2) +$$
$$E(X_3, X_4, X_6) + E(X_6, X_7) + E(X_5, X_6, X_8)\}$$

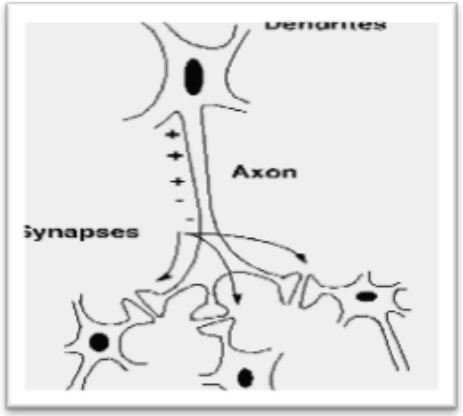


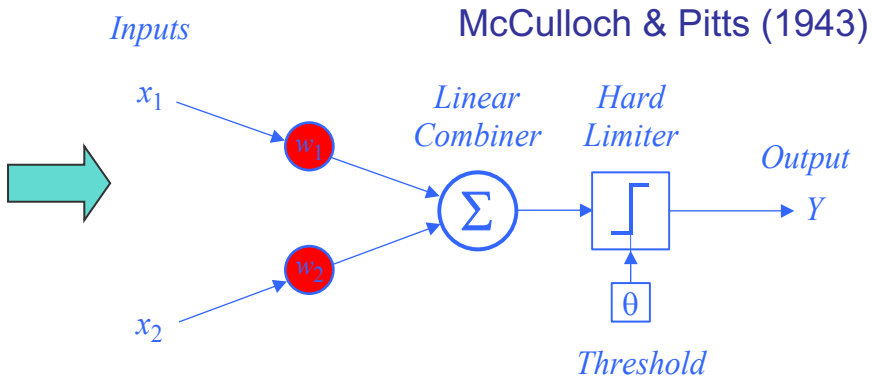
Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
  - Historical remarks: early days of neural networks
  - Modern building blocks: units, layers, activations functions, loss functions, etc.
  - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
  - Graphical models vs. computational graphs
  - Sigmoid Belief Networks as graphical models
  - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
  - Using outputs of NNs as inputs to GMs
  - GMs with potential functions represented by NNs
  - NNs with structured outputs
- Bayesian Learning of NNs
  - Bayesian learning of NN parameters
  - Deep kernel learning

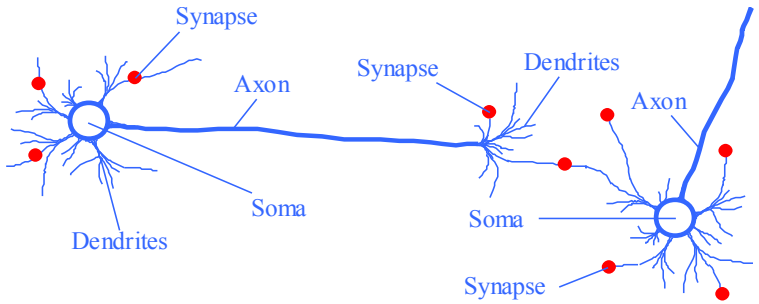
Perceptron and Neural Nets

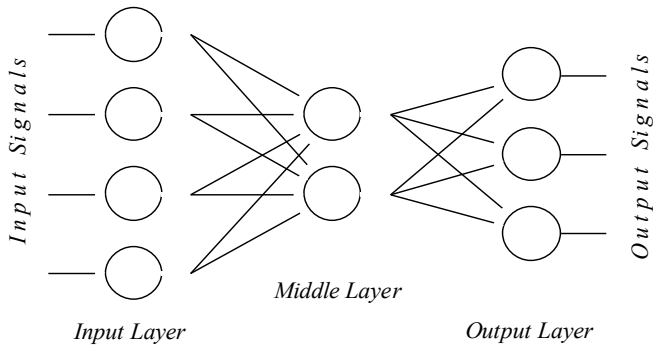
- From biological neuron to artificial neuron (perceptron)



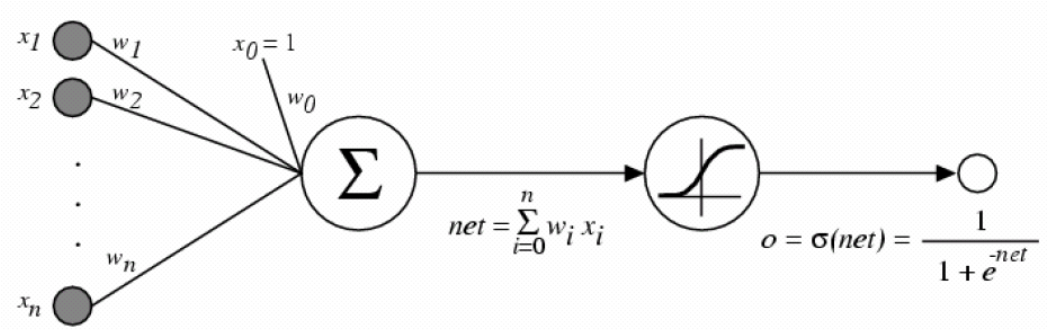


- From biological neuron network to artificial neuron networks





The perceptron learning algorithm



- Recall the nice property of sigmoid functic $\frac{d\sigma}{dt} = \sigma(1 - \sigma)$
- Consider regression problem f: X→Y, for scalar $y = f(x) + \epsilon$
- We used to maximize the conditional data likelihood
- Here …

$$\vec{w} = \arg\max_{\vec{w}} \ln \prod_i P(y_i | x_i; \vec{w})$$

$$\vec{w} = \arg\min_{\vec{w}} \sum_i \frac{1}{2} (y_i - \hat{f}(x_i; \vec{w}))^2$$

The perceptron learning algorithm

$$\begin{aligned}
\frac{\partial E_D[\vec{w}])}{\partial w_j} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_i (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial net_i}\frac{\partial net_d}{\partial w_i} \\
&= -\sum_d (t_d - o_d) o_d(1 - o_d) x_d^i
\end{aligned}$$

x_d = input

t_d = target output

o_d = observed output

w_i = weight i

Batch mode:

Do until converge:

1. compute gradient $\nabla E_D[w]$
2. $\vec{w} = \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode:

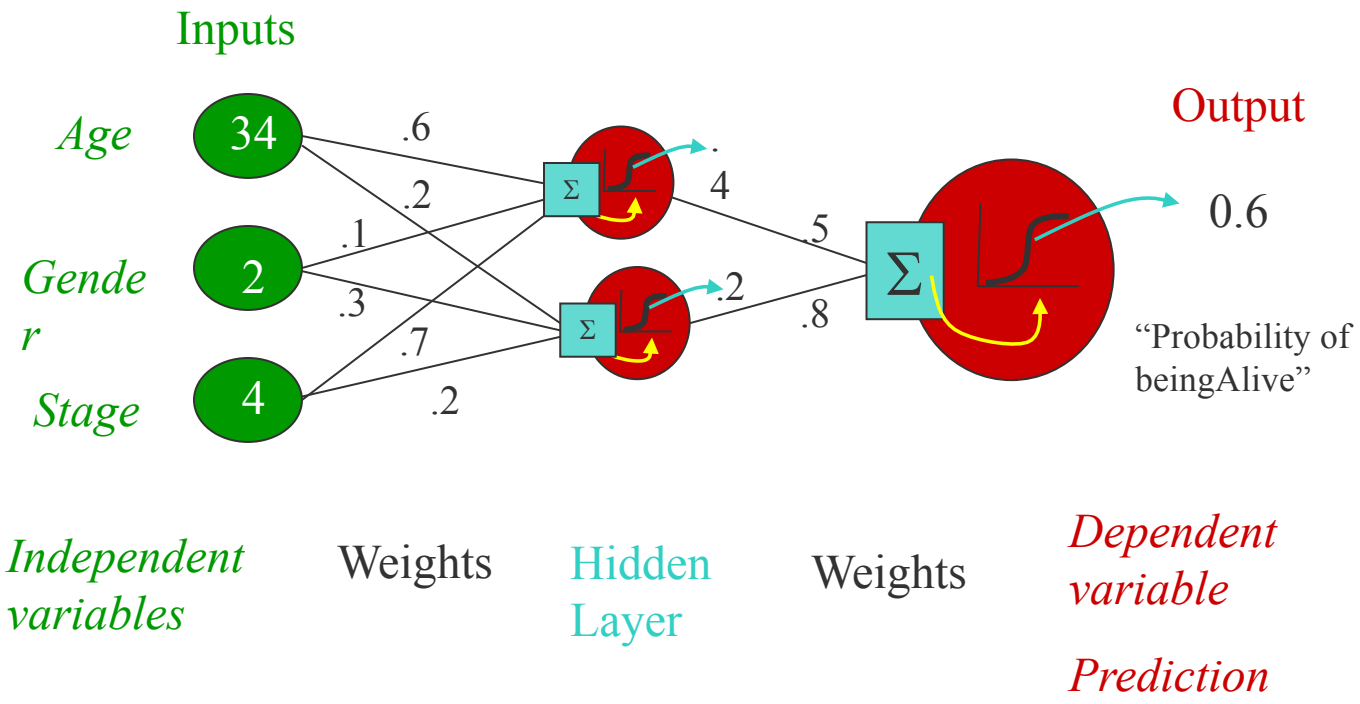
Do until converge:

- For each training example *d* in *D*
  1. compute gradient $\nabla E_d[w]$
  2. $\vec{w} = \vec{w} - \eta \nabla E_d[\vec{w}]$

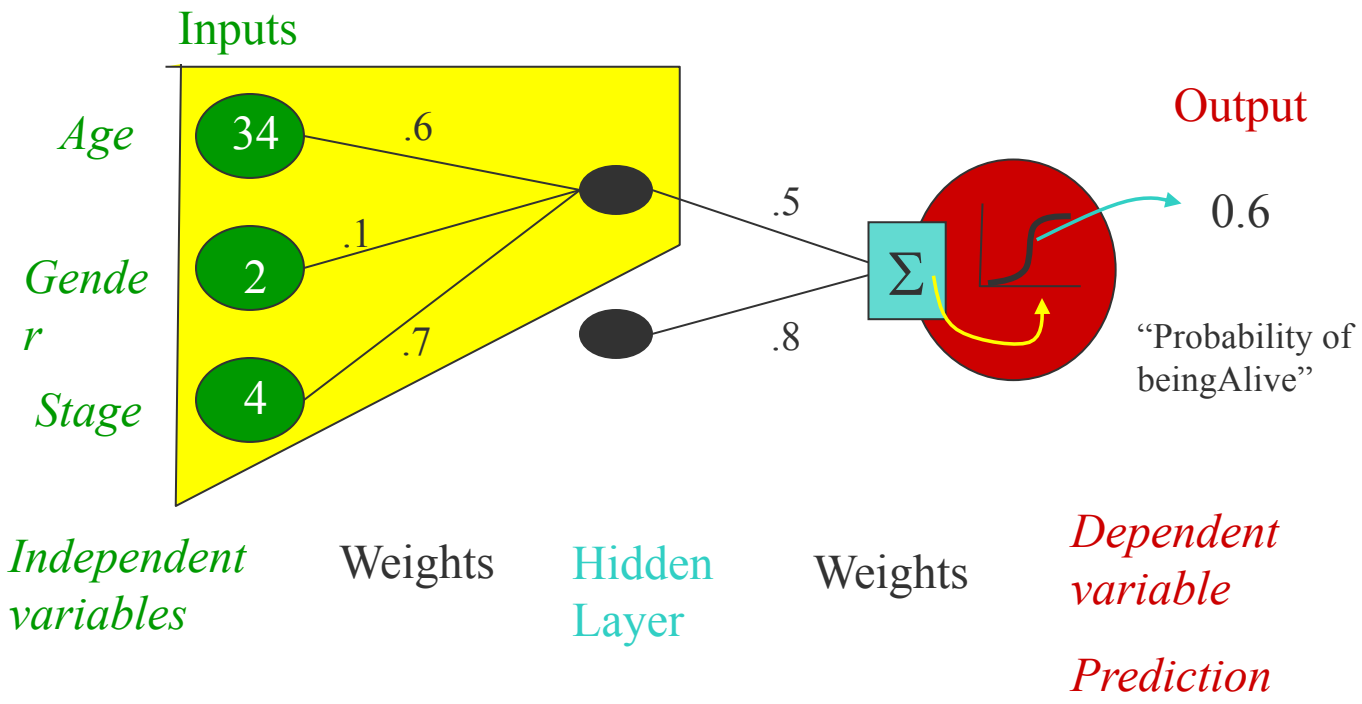
  where

  $\nabla E_d[\vec{w}] = -(t_d - o_d) o_d(1 - o_d)\vec{x}_d$

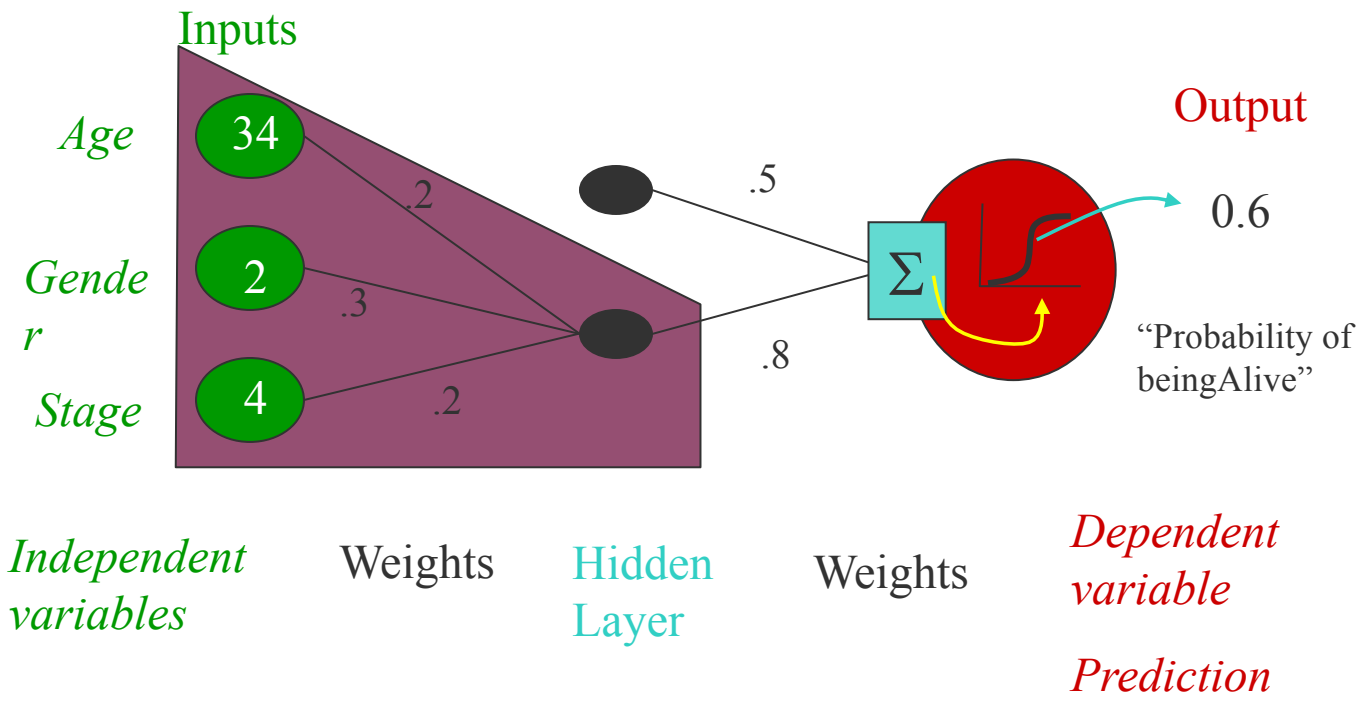
Neural Network Model



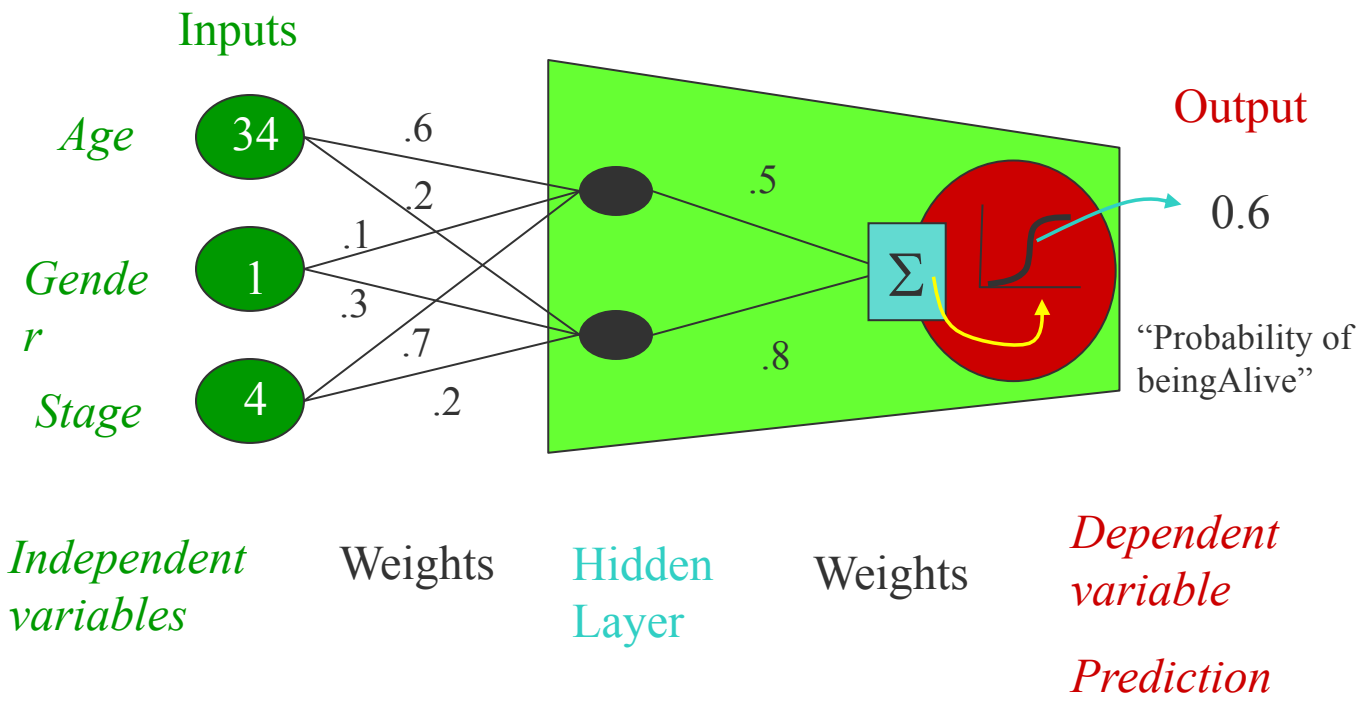
“Combined logistic models”



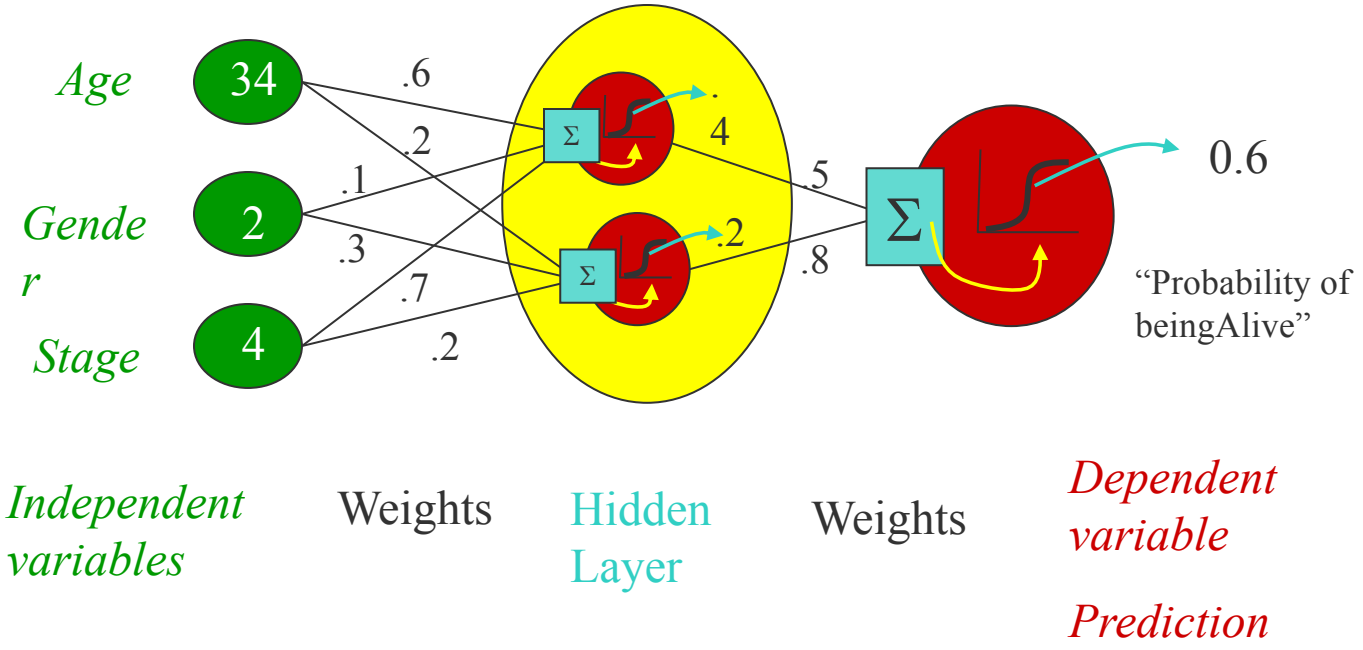
“Combined logistic models”



“Combined logistic models”

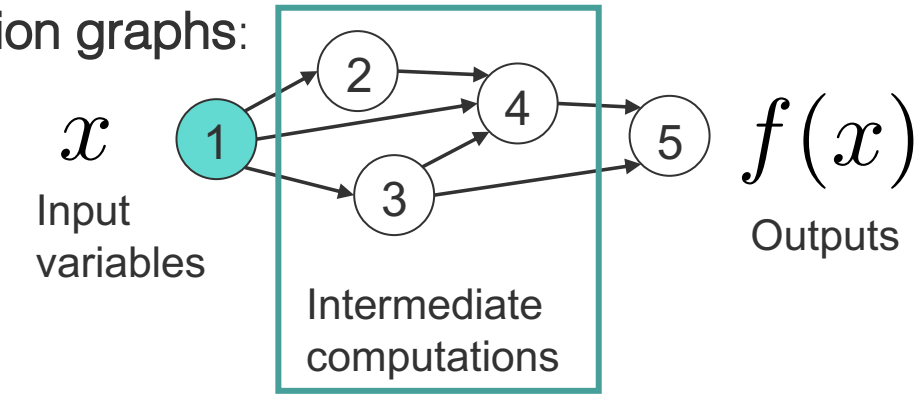


Not really, no target for hidden units...



Backpropagation: Reverse-mode differentiation

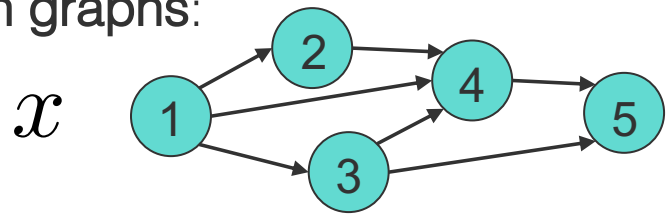
- Artificial neural networks are nothing more than complex functional compositions that can be represented by **computation graphs**:



$$\frac{\partial f_n}{\partial x} =$$

Backpropagation: Reverse-mode differentiation

- Artificial neural networks are nothing more than complex functional compositions that can be represented by **computation graphs**:



$$\frac{\partial f_n}{\partial x} =$$

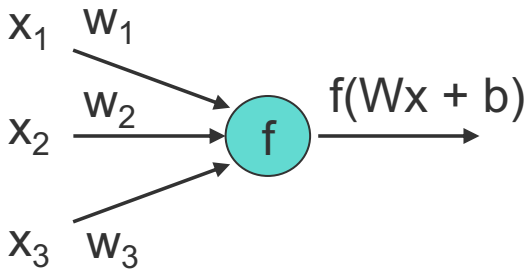
- By applying the chain rule and using reverse accumulation, we get

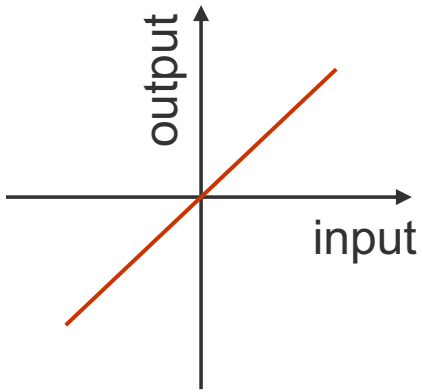
$$\frac{\partial f_n}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \frac{\partial f_{i_1}}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \sum_{i_2 \in \pi(i_1)} \frac{\partial f_{i_1}}{\partial f_{i_2}} \frac{\partial f_{i_1}}{\partial x} = \dots$$

- The algorithm is commonly known as backpropagation
- What if some of the functions are stochastic?
- Then use stochastic backpropagation!
  (to be covered in the next part)
- Modern packages can do this *automatically* (more later)

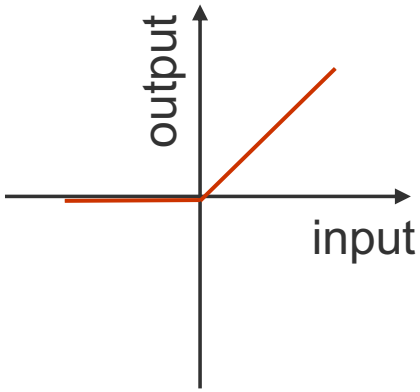
Modern building blocks of deep networks

- Activation functions
  - Linear and ReLU
  - Sigmoid and tanh
  - Etc.





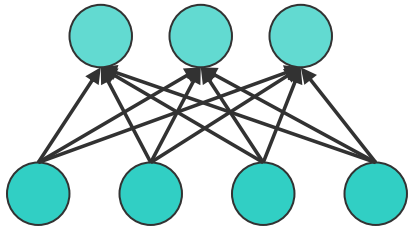
Linear



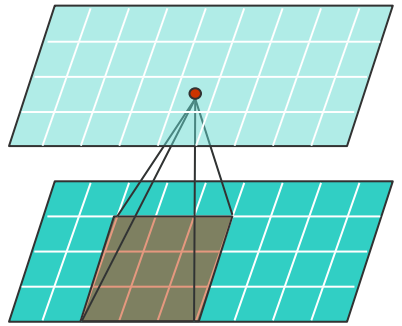
Rectified linear (ReLU)

Modern building blocks of deep networks

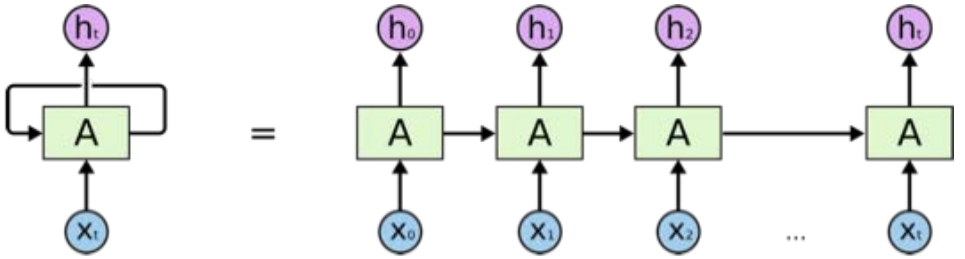
- Activation functions
  - Linear and ReLU
  - Sigmoid and tanh
  - Etc.
- Layers
  - Fully connected
  - Convolutional & pooling
  - Recurrent
  - ResNets
  - Etc.



fully connected

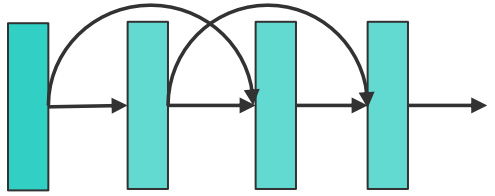


convolutional



recurrent

source: colah.github.io

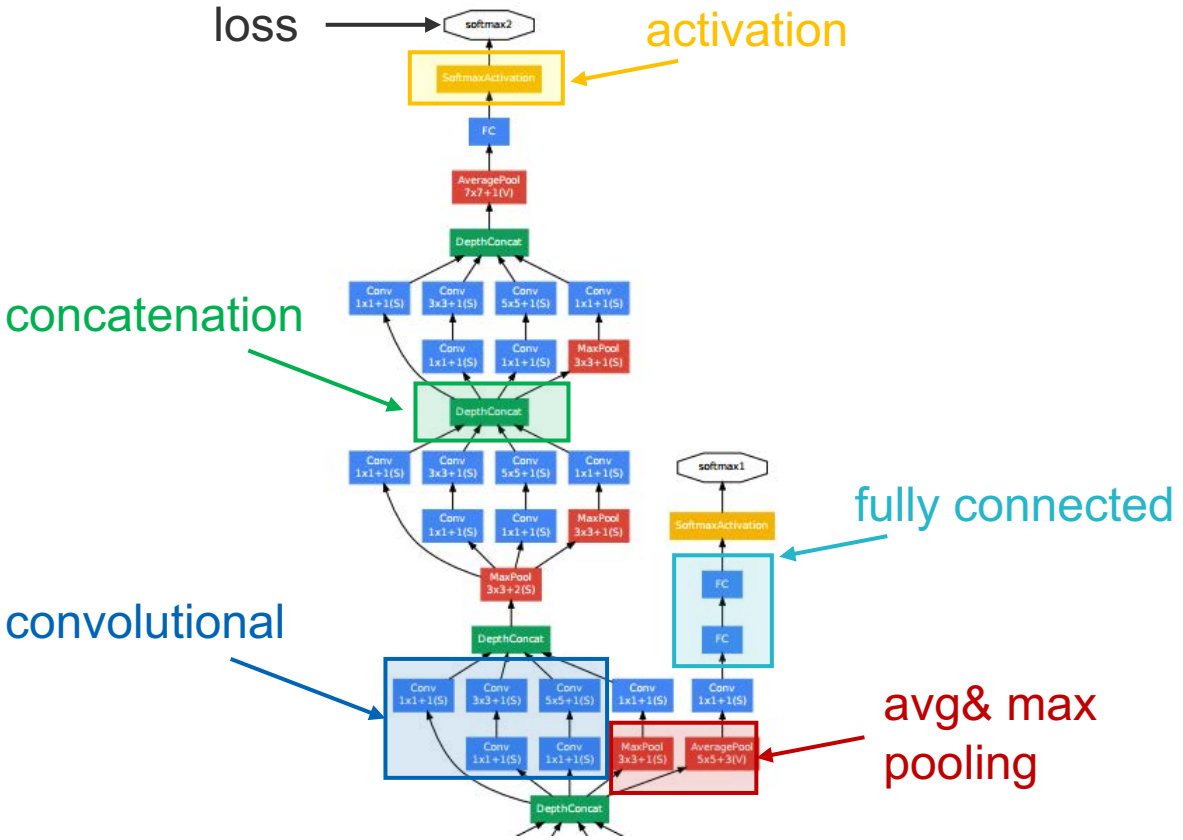


blocks with residual connections

Modern building blocks of deep networks

- Activation functions
  - Linear and ReLU
  - Sigmoid and tanh
  - Etc.
- Layers
  - Fully connected
  - Convolutional & pooling
  - Recurrent
  - ResNets
  - Etc.
- Loss functions
  - Cross-entropy loss
  - Mean squared error
  - Etc.

Putting things together:

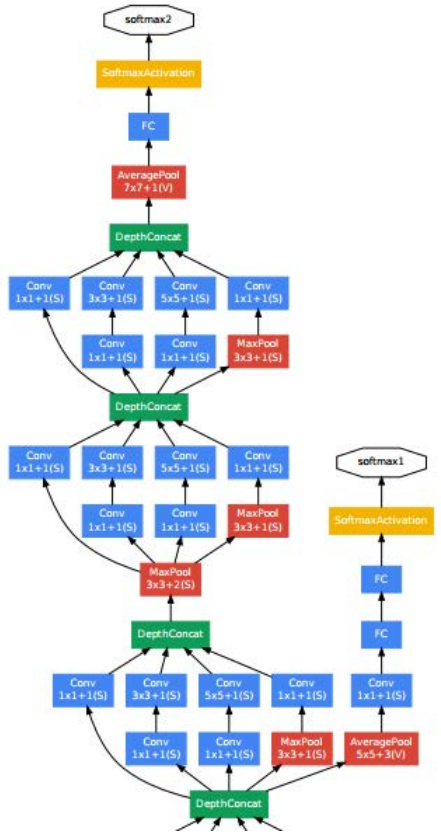


(a part of GoogleNet)

Modern building blocks of deep networks

- Activation functions
  - Linear and ReLU
  - Sigmoid and tanh
  - Etc.
- Layers
  - Fully connected
  - Convolutional & pooling
  - Recurrent
  - ResNets
  - Etc.
- Loss functions
  - Cross-entropy loss
  - Mean squared error
  - Etc.

Putting things together:



(a part of GoogleNet)

- Arbitrary combinations of the basic building blocks
- Multiple loss functions – multi-target prediction, transfer learning, and more
- Given enough data, deeper architectures just keep improving
- Representation learning: the networks learn increasingly more abstract representations of the data that are “disentangled,” i.e., amenable to linear separation.

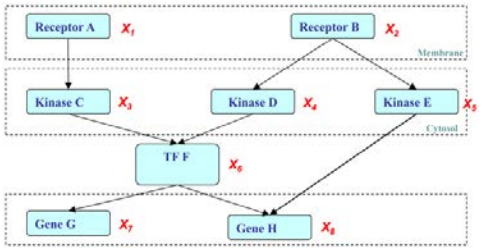
Outline

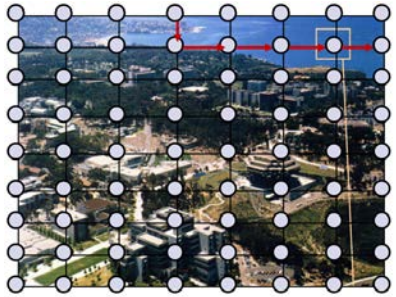
- Probabilistic Graphical Models: Basics
- An overview of the DL components
  - Historical remarks: early days of neural networks
  - Modern building blocks: units, layers, activations functions, loss functions, etc.
  - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
  - Graphical models vs. computational graphs
  - Sigmoid Belief Networks as graphical models
  - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
  - Using outputs of NNs as inputs to GMs
  - GMs with potential functions represented by NNs
  - NNs with structured outputs
- Bayesian Learning of NNs
  - Bayesian learning of NN parameters
  - Deep kernel learning

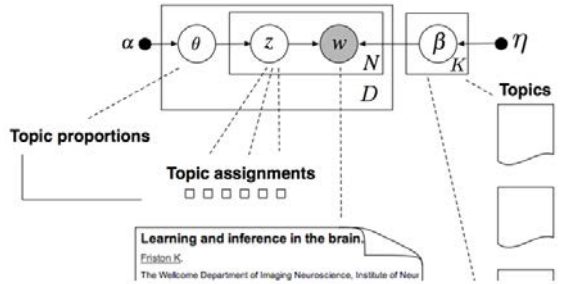
Graphical models vs. Deep nets

Graphical models

- Representation for encoding meaningful knowledge and the associated uncertainty in a graphical form

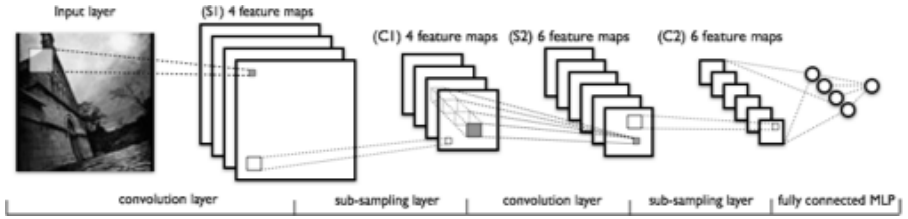






Deep neural networks

- Learn representations that facilitate computation and performance on the end-metric (intermediate representations are not guaranteed to be meaningful)



Graphical models vs. Deep nets

Graphical models

- Representation for encoding meaningful knowledge and the associated uncertainty in a graphical form
- Learning and inference are based on a rich toolbox of well-studied (structure-dependent) techniques (e.g., EM, message passing, VI, MCMC, etc.)
- Graphs represent models

Deep neural networks

- Learn representations that facilitate computation and performance on the end-metric (intermediate representations are not guaranteed to be meaningful)
- Learning is predominantly based on the gradient descent method (aka backpropagation); Inference is often trivial and done via a "forward pass"
- Graphs represent computation

Graphical models vs. Deep nets

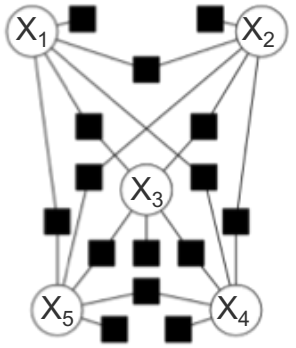
Graphical models

Utility of the graph

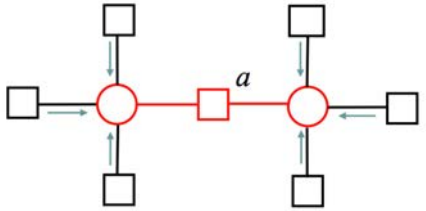
- A vehicle for synthesizing a global loss function from local structure
  - potential function, feature function, etc.
- A vehicle for designing sound and efficient inference algorithms
  - Sum-product, mean-field, etc.
- A vehicle to inspire approximation and penalization
  - Structured MF, Tree-approximation, etc.
- A vehicle for monitoring theoretical and empirical behavior and accuracy of inference

Utility of the loss function

- A major measure of quality of the learning algorithm and the model



$$\log P(X) = \sum_i \log \phi(x_i) + \sum_{i,j} \log \psi(x_i, x_j)$$

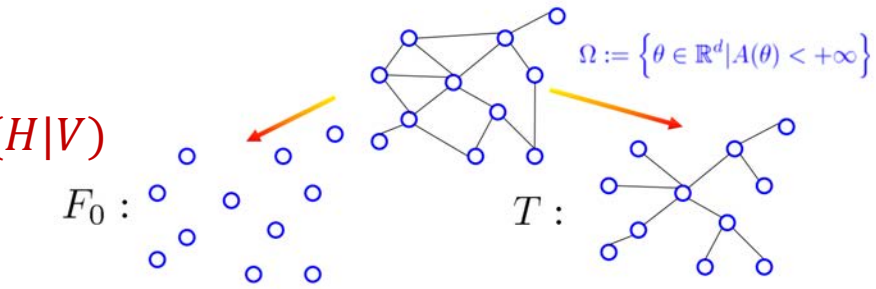


$$m_{i\to a}(x_i) = \prod_{c\in N(i)\backslash a} m_{c\to i}(x_i)$$

$$b_a(X_a) \propto f_a(X_a) \prod_{i\in N(a)} m_{i\to a}(x_i)$$

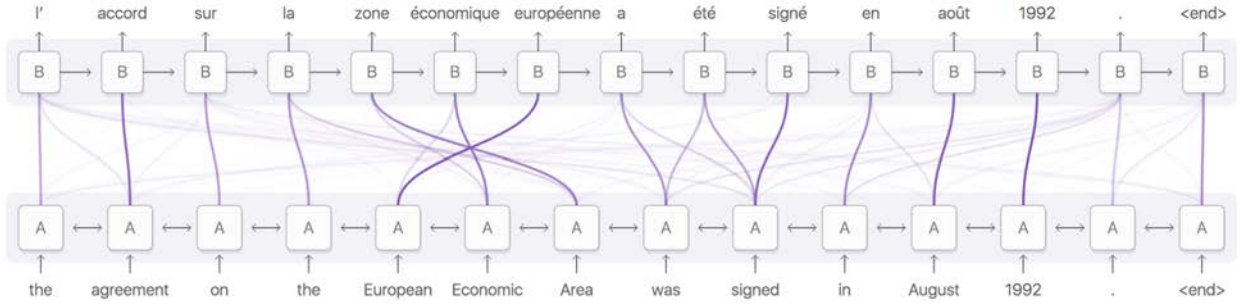
$$m_{a\to i}(x_i) = \sum_{X_a\backslash x_i} f_a(X_a) \prod_{j\in N(a)\backslash i} m_{j\to a}(x_j)$$

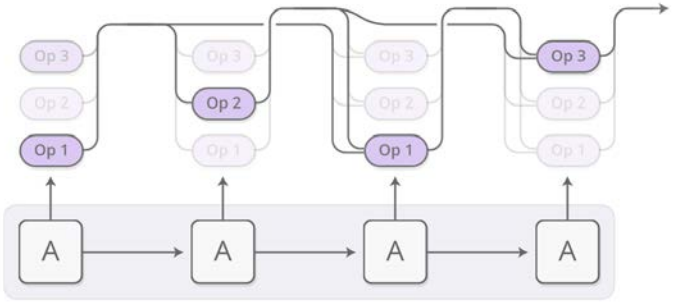
$Q(H) \sim P(H|V)$



$\theta = \text{argmax}_\theta P_\theta(V)$

Graphical models vs. Deep nets





Images from Distill.pub

Deep neural networks

Utility of the network

- A vehicle to conceptually synthesize complex decision hypothesis
  - stage-wise projection and aggregation
- A vehicle for organizing computational operations
  - stage-wise update of latent states
- A vehicle for designing processing steps and computing modules
  - Layer-wise parallelization
- No obvious utility in evaluating DL inference algorithms

Utility of the Loss Function

- Global loss? Well it is complex and non-convex...

Graphical models vs. Deep nets

Graphical models

Utility of the graph

- A vehicle for synthesizing a global loss function from local structure
  - potential function, feature function, etc.
- A vehicle for designing sound and efficient inference algorithms
  - Sum-product, mean-field, etc.
- A vehicle to inspire approximation and penalization
  - Structured MF, Tree-approximation, etc.
- A vehicle for monitoring theoretical and empirical behavior and accuracy of inference

Utility of the loss function

- A major measure of quality of the learning algorithm and the model

Deep neural networks

Utility of the network

- A vehicle to conceptually synthesize complex decision hypothesis
  - stage-wise projection and aggregation
- A vehicle for organizing computational operations
  - stage-wise update of latent states
- A vehicle for designing processing steps and computing modules
  - Layer-wise parallelization
- No obvious utility in evaluating DL inference algorithms

Utility of the Loss Function

- Global loss? Well it is complex and non-convex...

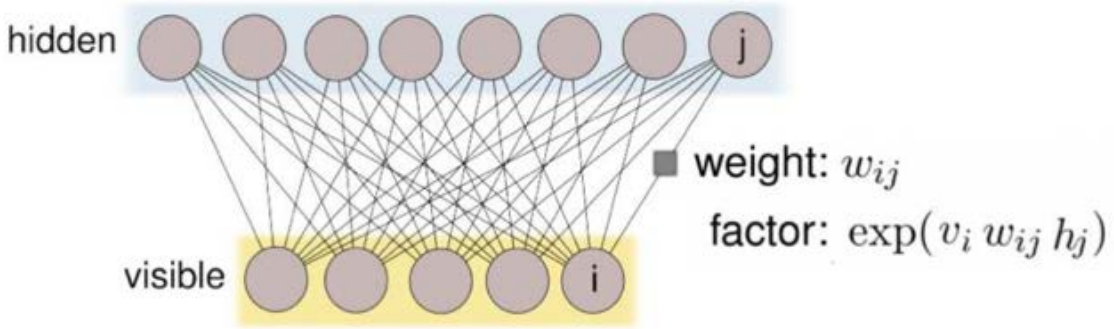
	DL	≡ ?	ML (e.g., GM)
Empirical goal:	e.g., classification, feature learning		e.g., latent variable inference, transfer learning
Structure:	Graphical		Graphical
Objective:	Something aggregated from local functions		Something aggregated from local functions
Vocabulary:	Neuron, activation function, …		Variable, potential function, …
Algorithm:	A single, unchallenged, inference algorithm – Backpropagation (BP)		A major focus of open research, many algorithms, and more to come
Evaluation:	On a black-box score – end performance		On almost every intermediate quantity
Implementation:	Many tricks		More or less standardized
Experiments:	Massive, real data (GT unknown)		Modest, often simulated data (GT known)

Graphical Models vs. Deep Nets

- So far:
  - Graphical models are representations of probability distributions
  - Neural networks are function approximators (with no probabilistic meaning)
- Some of the neural nets are in fact proper graphical models (i.e., units/neurons represent random variables):
  - Boltzmann machines (Hinton & Sejnowsky, 1983)
  - Restricted Boltzmann machines (Smolensky, 1986)
  - Learning and Inference in sigmoid belief networks (Neal, 1992)
  - Fast learning in deep belief networks (Hinton, Osindero, Teh, 2006)
  - Deep Boltzmann machines (Salakhutdinov and Hinton, 2009)
- Let's go through these models one-by-one

I: Restricted Boltzmann Machines

- RBM is a Markov random field represented with a bi-partite graph
- All nodes in one layer/part of the graph are connected to all in the other; no inter-layer connections



- Joint distribution

$$P(v,h) = \frac{1}{Z}\exp\left\{\sum_{i,j} w_{ij}v_i h_i + \sum_i b_i v_i + \sum_j c_j h_j\right\}$$

Images from Marcus Frean, MLSS Tutorial 2010

I: Restricted Boltzmann Machines

- Log-likelihood of a single data point (unobservables marginalized out):

$$\log L(v) = \log \sum_h \exp\left\{ \sum_{i,j} w_{ij} v_i h_i + \sum_i b_i v_i + \sum_j c_j h_j - \log(Z) \right\}$$

- Gradient of the log-likelihood w.r.t. the model parameters:

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \sum_h P(h|v) \frac{\partial}{\partial w_{ij}} P(v,h) - \sum_{v,h} P(v,h) \frac{\partial}{\partial w_{ij}} P(v,h)$$

  - where we have averaging over the posterior and over the joint.

Images from Marcus Frean, MLSS Tutorial 2010

I: Restricted Boltzmann Machines

- Gradient of the log-likelihood w.r.t. the parameters (alternative form):

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \mathbb{E}_{P(h|v)}\left[\frac{\partial}{\partial w_{ij}} P(v,h)\right] - \mathbb{E}_{P(v,h)}\left[\frac{\partial}{\partial w_{ij}} P(v,h)\right]$$

  - Both expectations can be approximated via sampling
  - Sampling from the posterior is exact (RBM factorizes over h given v)
  - Sampling from the joint is done via MCMC (e.g., Gibbs sampling)
  - In the neural networks literature:
    - computing the first term is called **the clamped / wake / positive phase**
      (the network is "awake" since it conditions on the visible variables)
    - Computing the second term is called **the unclamped / sleep / free / negative phase**
      (the network is "asleep" since it samples the visible variables from the joint;
      metaphorically, it is "dreaming" the visible inputs)

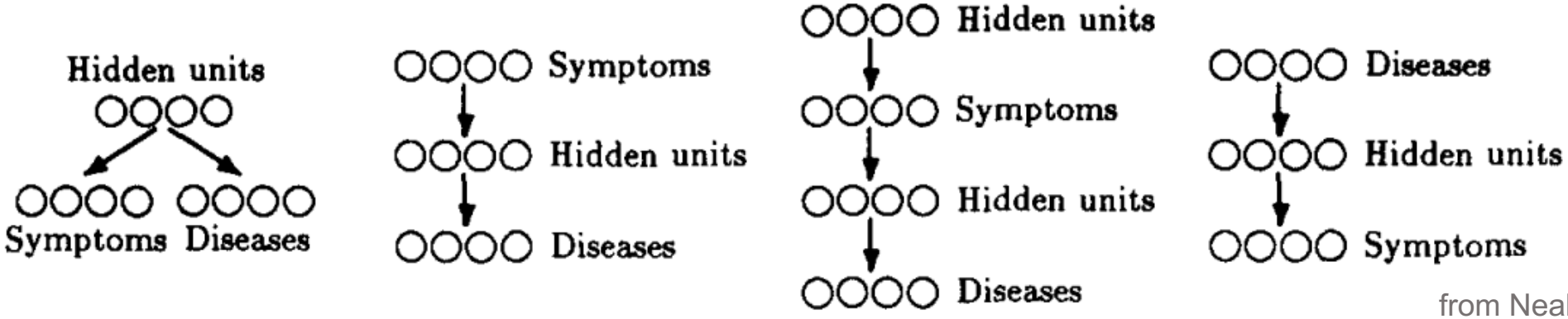
I: Restricted Boltzmann Machines

- Gradient of the log-likelihood w.r.t. the parameters (alternative form):

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \mathbb{E}_{P(h|v)}\left[\frac{\partial}{\partial w_{ij}} P(v,h)\right] - \mathbb{E}_{P(v,h)}\left[\frac{\partial}{\partial w_{ij}} P(v,h)\right]$$

- Learning is done by optimizing the log-likelihood of the model for a given data via stochastic gradient descent (SGD)
- Estimation of the second term (the negative phase) heavily relies on the mixing properties of the Markov chain
- This often causes slow convergence and requires extra computation

II: Sigmoid Belief Networks

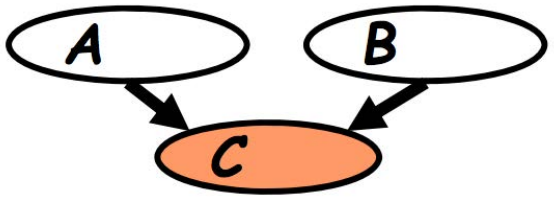


from Neal, 1992

- Sigmoid belief nets are simply Bayesian networks over binary variables with conditional probabilities represented by sigmoid functions:

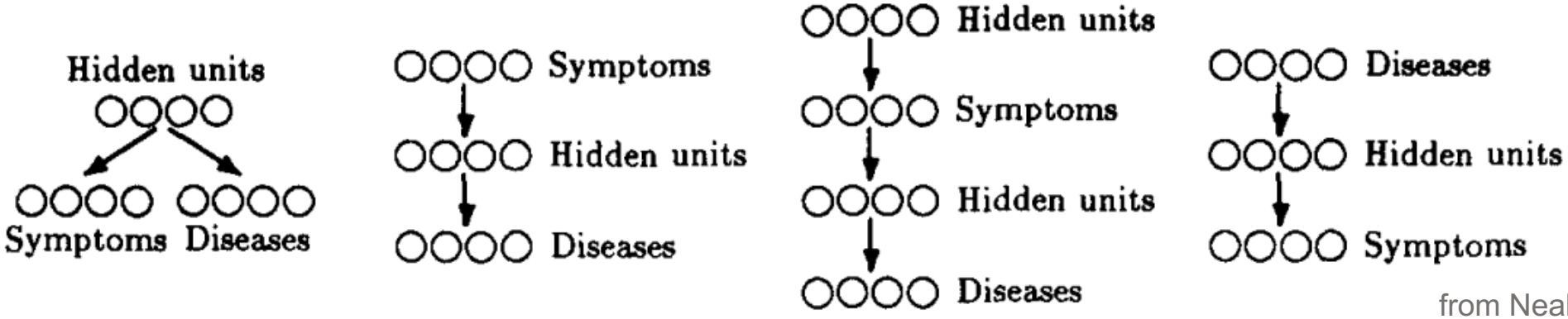
$$P(x_i|\pi(x_i)) = \sigma\left(x_i \sum_{x_j \in \pi(x_i)} w_{ij}x_j\right)$$

- Bayesian networks exhibit a phenomenon called "explain away effect"



If A correlates with C, then the chance of B correlating with C decreases. ⇒ A and B become correlated given C.

II: Sigmoid Belief Networks

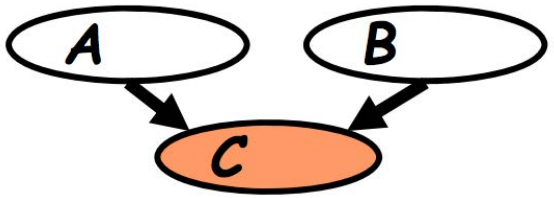


from Neal, 1992

- Sigmoid belief nets are simply Bayesian networks over binary variables with conditional probabilities represented by sigmoid functions:

$$P(x_i|\pi(x_i)) = \sigma\left(x_i \sum_{x_j \in \pi(x_i)} w_{ij}x_j\right)$$

- Bayesian networks exhibit a phenomenon called "explain away effect"



Note:
Due to the "explain away effect," when we condition on the visible layer in belief networks, hidden variables all become dependent.

Sigmoid Belief Networks: Learning and Inference

- Neal proposed Monte Carlo methods for learning and inference (Neal, 1992):

Approximated with Gibbs sampling

- Conditional distributions:

$$P(S_i = x \mid S_j = s_j : j \neq i) \propto \sigma\left(x^* \sum_{j<i} s_j w_{ij}\right) \prod_{j>i} \sigma\left(s_j^* \left(x w_{ji} + \sum_{k<j, k\neq i} s_k w_{jk}\right)\right)$$

- No negative phase as in RBM!
- Convergence is very slow, especially for large belief nets, due to the intricate "explain-away" effects…

$$\begin{aligned}
\frac{\partial L}{\partial w_{ij}} &= \sum_{\tilde{v}\in\mathcal{T}} \frac{1}{P(\tilde{V}=\tilde{v})} \frac{\partial P(\tilde{V}=\tilde{v})}{\partial w_{ij}} && \text{log derivative} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \frac{1}{P(\tilde{V}=\tilde{v})} \sum_{\tilde{h}} \frac{\partial P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle)}{\partial w_{ij}} && \text{prob. of the visibles via marginalization} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{h}} P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle \mid \tilde{V}=\tilde{v}) \cdot \frac{1}{P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle)} \frac{\partial P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle)}{\partial w_{ij}} && \text{Bayes rule + rearrange sums} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{s}} P(\tilde{S}=\tilde{s} \mid \tilde{V}=\tilde{v}) \frac{1}{P(\tilde{S}=\tilde{s})} \frac{\partial P(\tilde{S}=\tilde{s})}{\partial w_{ij}} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{s}} P(\tilde{S}=\tilde{s} \mid \tilde{V}=\tilde{v}) \cdot \frac{1}{\sigma\left(s_i^* \sum_{k<i} s_k w_{ik}\right)} \frac{\partial \sigma\left(s_i^* \sum_{k<i} s_k w_{ik}\right)}{\partial w_{ij}} && \text{Plug-in the actual sigmoid form of the conditional prob.} \\
&= \boxed{\sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{s}} P(\tilde{S}=\tilde{s} \mid \tilde{V}=\tilde{v})\, s_i^* s_j \sigma\left(-s_i^* \sum_{k<i} s_k w_{ik}\right).}
\end{aligned}$$

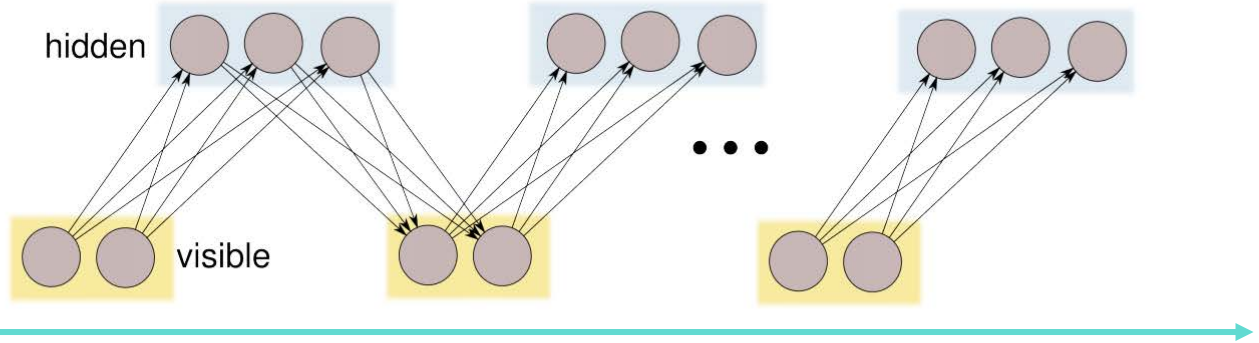
Equations from Neal, 1992

RBMs are infinite belief networks

- Recall the expression for the gradient of the log likelihood for RBM:

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \mathbb{E}_{P(h|v)}\left[\frac{\partial}{\partial w_{ij}} P(v,h)\right] - \mathbb{E}_{P(v,h)}\left[\frac{\partial}{\partial w_{ij}} P(v,h)\right]$$

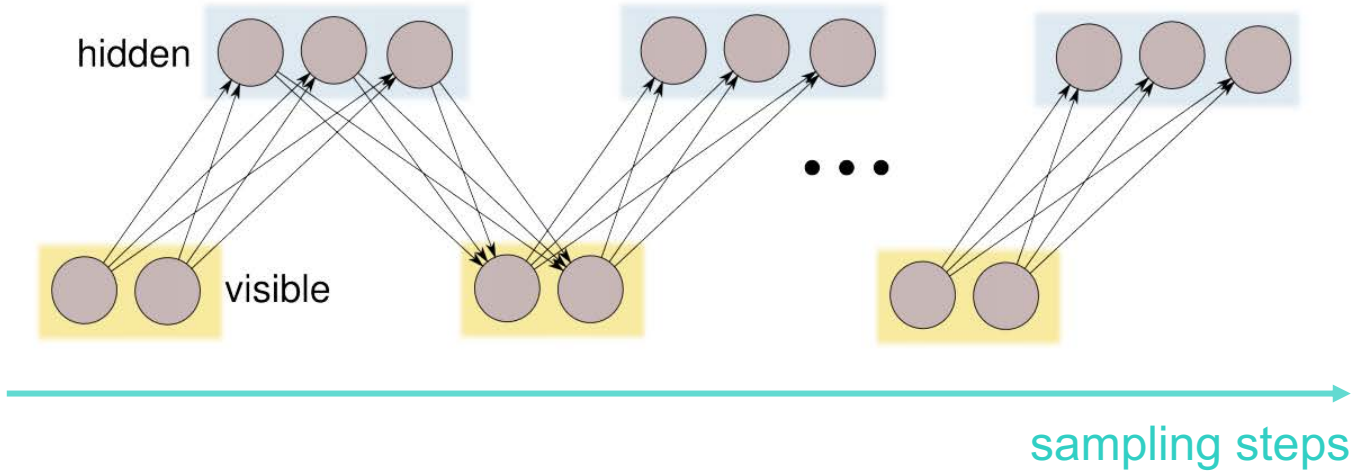
- To make a gradient update of the model parameters, we need compute the expectations via sampling.
  - We can sample exactly from the posterior in the first term
  - We run block Gibbs sampling to approximately sample from the joint distribution



images from Marcus Frean, MLSS Tutorial 2010

RBMs are infinite belief networks

- Gibbs sampling: alternate between sampling hidden and visible variables

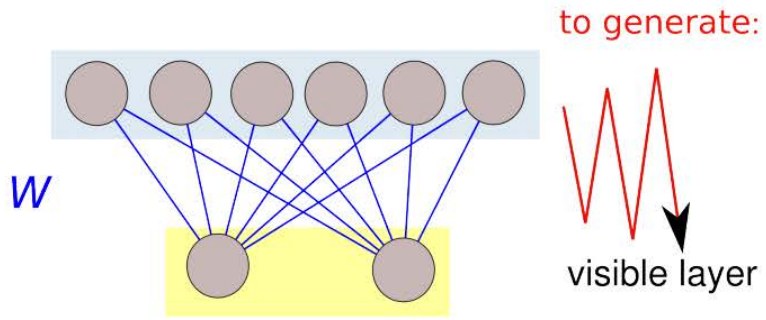


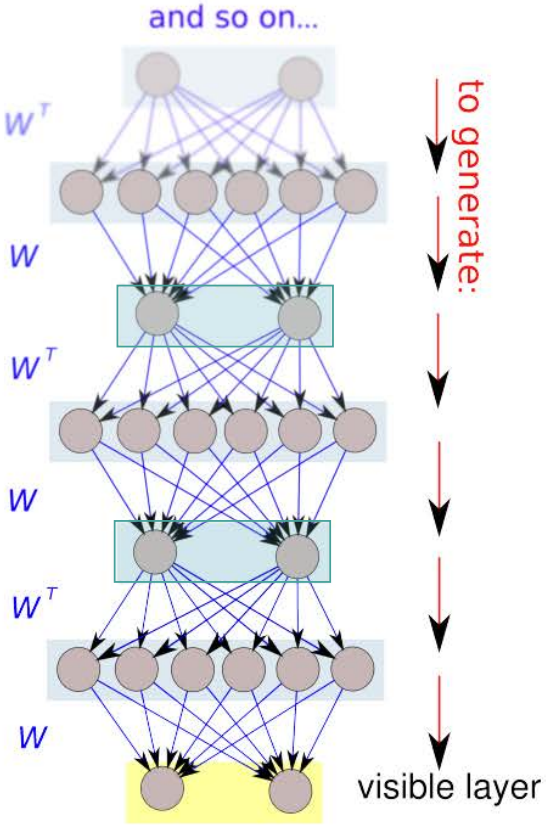
- Conditional distributions $P(v|h)$ and $P(h|v)$ are represented by sigmoids
- Thus, we can think of Gibbs sampling from the joint distribution represented by an RBM as a top-down propagation in an infinitely deep sigmoid belief network!

images from Marcus Frean, MLSS Tutorial 2010

RBMs are infinite belief networks

- RBMs are equivalent to infinitely deep belief networks



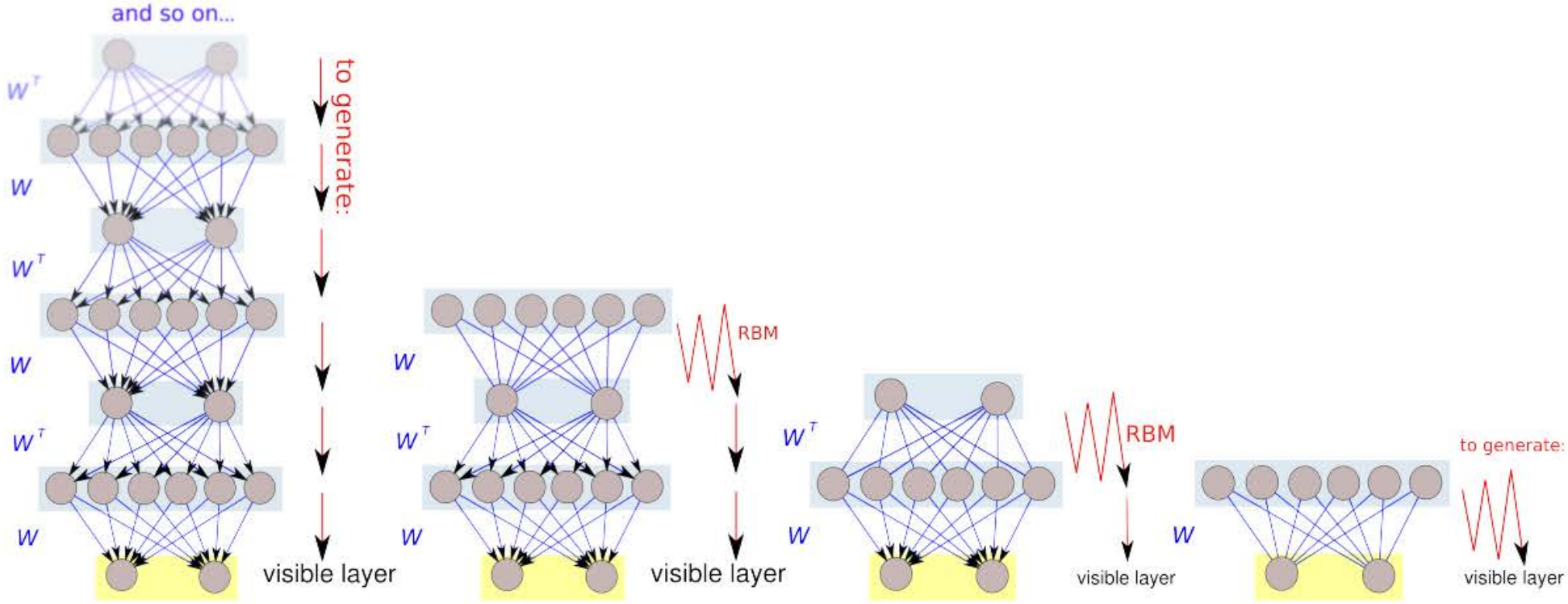


- Sampling from this is the same as sampling from the network on the right

images from Marcus Frean, MLSS Tutorial 2010

RBMs are infinite belief networks

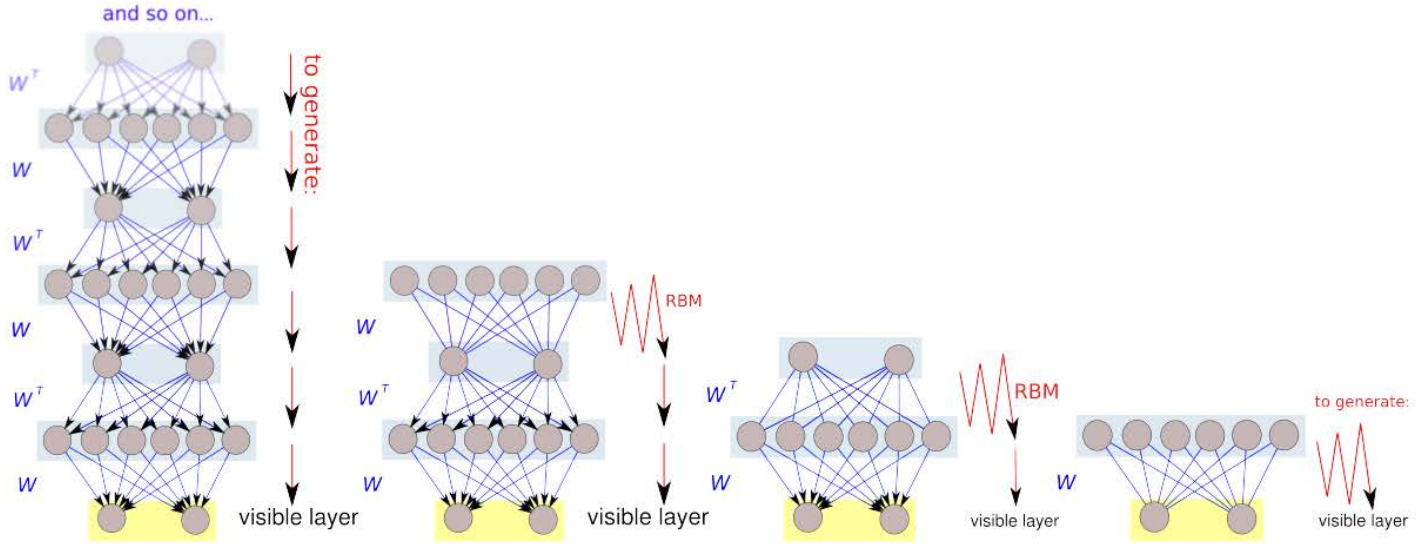
- RBMs are equivalent to infinitely deep belief networks



images from Marcus Frean, MLSS Tutorial 2010

RBMs are infinite belief networks

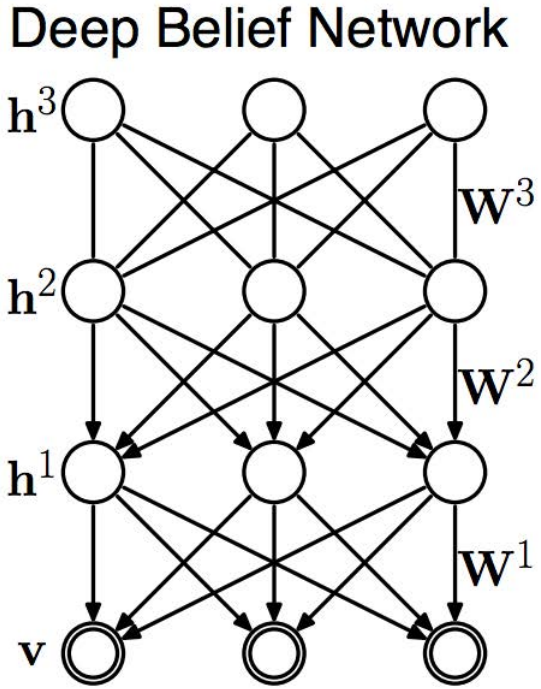
- RBMs are equivalent to infinitely deep belief networks



- When we train an RBM, we are really training an infinitely deep brief net!
- It is just that the weights of all layers are tied.
- If the weights are “untied” to some extent, we get a Deep Belief Network.

images from Marcus Frean, MLSS Tutorial 2010

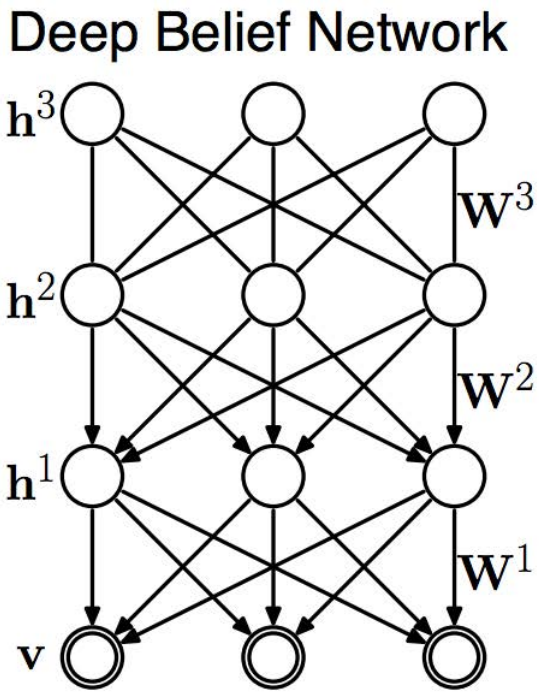
III: Deep Belief Nets



Now weights are untied!

- DBNs are hybrid graphical models (chain graphs):
  - Exact inference in DBNs is problematic due to explaining away effect
  - Training: greedy pre-training + ad-hoc fine-tuning; no proper joint training
  - Approximate inference is feed-forward

Deep Belief Networks



- DBNs represent a joint probability distribution

$$P(v, h^1, h^2, h^3) = P(h^2, h^3)P(h^1|h^2)P(v|h^1)$$

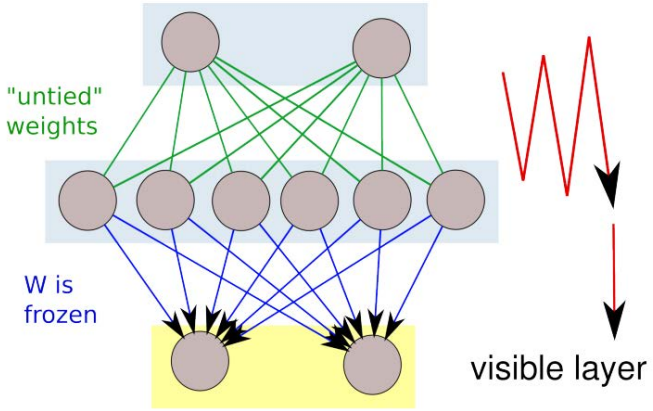
- Note that $P(h^2, h^3)$ is an RBM and the conditionals $P(h^1|h^2)$ and $P(v|h^1)$ are represented in the sigmoid form
- The model is trained by optimizing the log likelihood for a given data $\log P(v)$

Challenges:

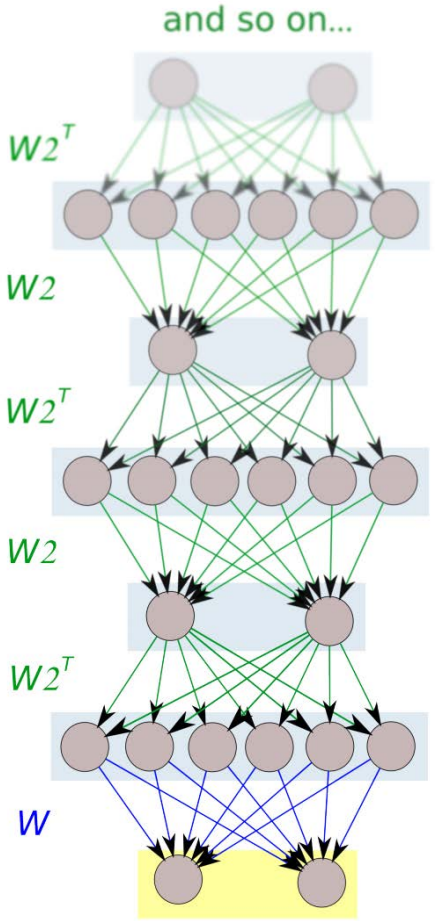
- Exact inference in DBNs is problematic due to explain away effect
- Training is done in two stages:
  - greedy pre-training + ad-hoc fine-tuning; no proper joint training
- Approximate inference is feed-forward (bottom-up)

DBN: Layer-wise pre-training

- Pre-train and freeze the 1st RBM
- Stack another RBM on top and train it



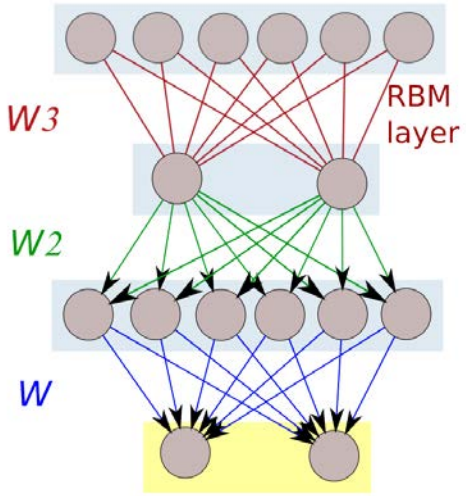
- The weights weights 2+ layers remain tied
- We repeat this procedure: pre-train and untie the weights layer-by-layer…

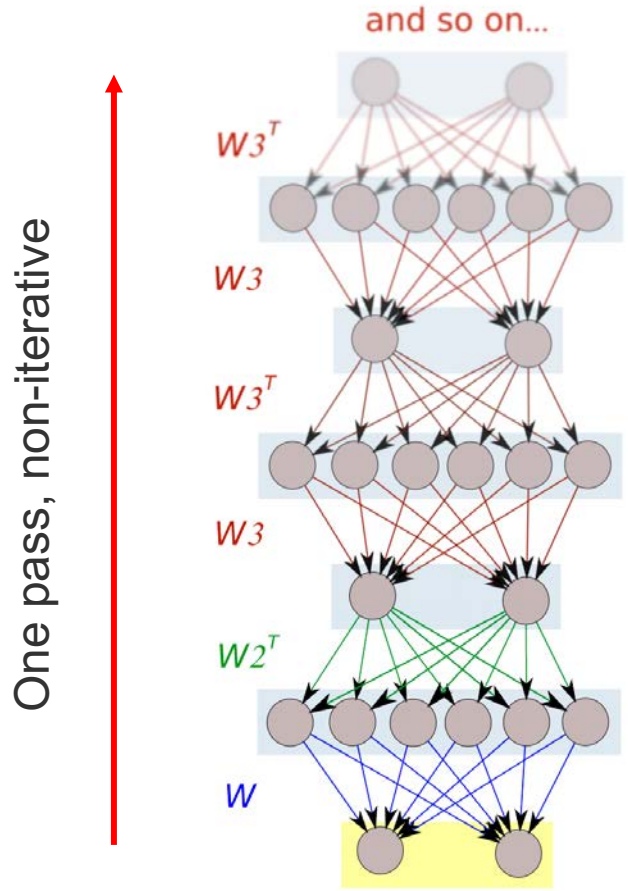


images from Marcus Frean, MLSS Tutorial 2010

DBN: Layer-wise pre-training

- We repeat this procedure: pre-train and untie the weights layer-by-layer:
- The weights of 3+ layers remain tied





- and so forth
- *From the optimization perspective, this procedure loosely corresponds to an approximate block-coordinate accent on the log-likelihood*

images from Marcus Frean, MLSS Tutorial 2010

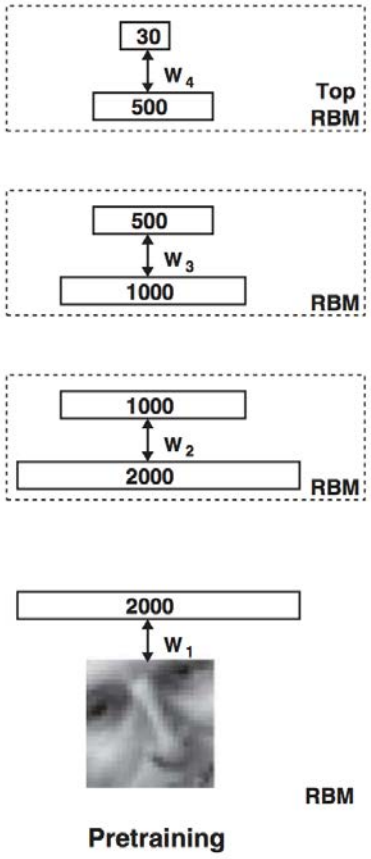
DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!

- We can further "fine-tune" a pre-trained DBN for some other task

Setting A: Unsupervised learning (DBN → autoencoder)

1. **Pre-train a stack of RBMs in a greedy layer-wise fashion**
2. "Unroll" the RBMs to create an autoencoder
3. Fine-tune the parameters by optimizing the reconstruction error



images from Hinton & Salakhutdinov, 2006

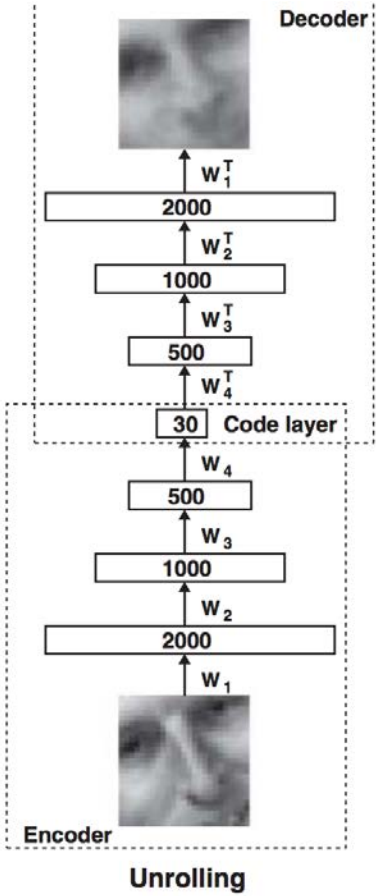
DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!

- We can further "fine-tune" a pre-trained DBN for some other task

Setting A: Unsupervised learning (DBN → autoencoder)

1. Pre-train a stack of RBMs in a greedy layer-wise fashion
2. **"Unroll" the RBMs to create an autoencoder**
3. Fine-tune the parameters by optimizing the reconstruction error



images from Hinton & Salakhutdinov, 2006

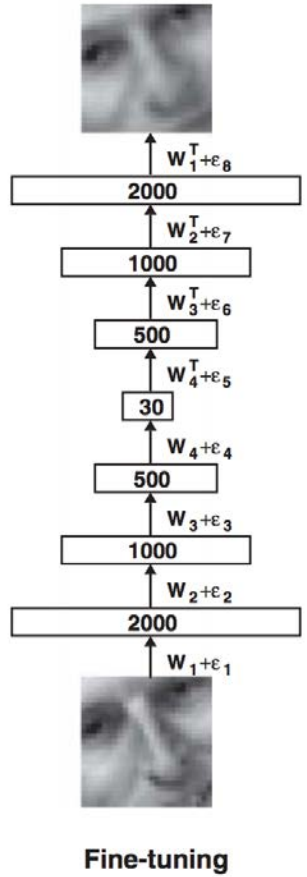
DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!

- We can further "fine-tune" a pre-trained DBN for some other task

Setting A: Unsupervised learning (DBN → autoencoder)

1. Pre-train a stack of RBMs in a greedy layer-wise fashion
2. "Unroll" the RBMs to create an autoencoder
3. **Fine-tune the parameters by optimizing the reconstruction error**



Fine-tuning

images from Hinton & Salakhutdinov, 2006

DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!

- We can further "fine-tune" a pre-trained DBN for some other task

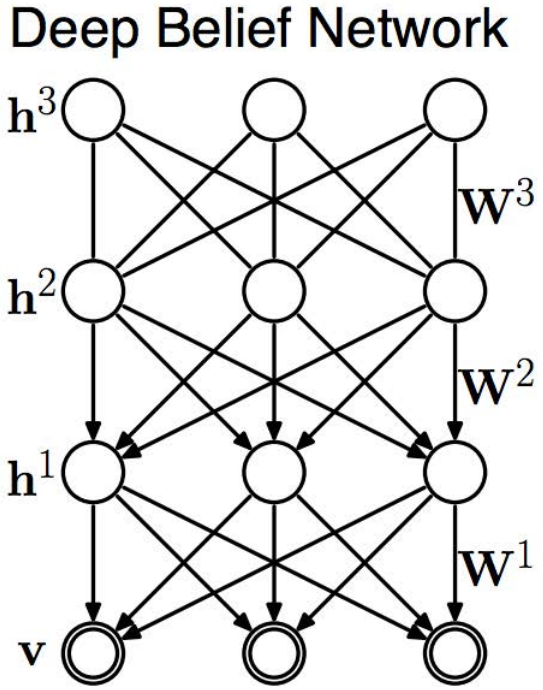
Setting B: Supervised learning (DBN → classifier)

1. Pre-train a stack of RBMs in a greedy layer-wise fashion
2. "Unroll" the RBMs to create a feedforward classifier
3. Fine-tune the parameters by optimizing the reconstruction error

Some intuitions about how pre-training works:

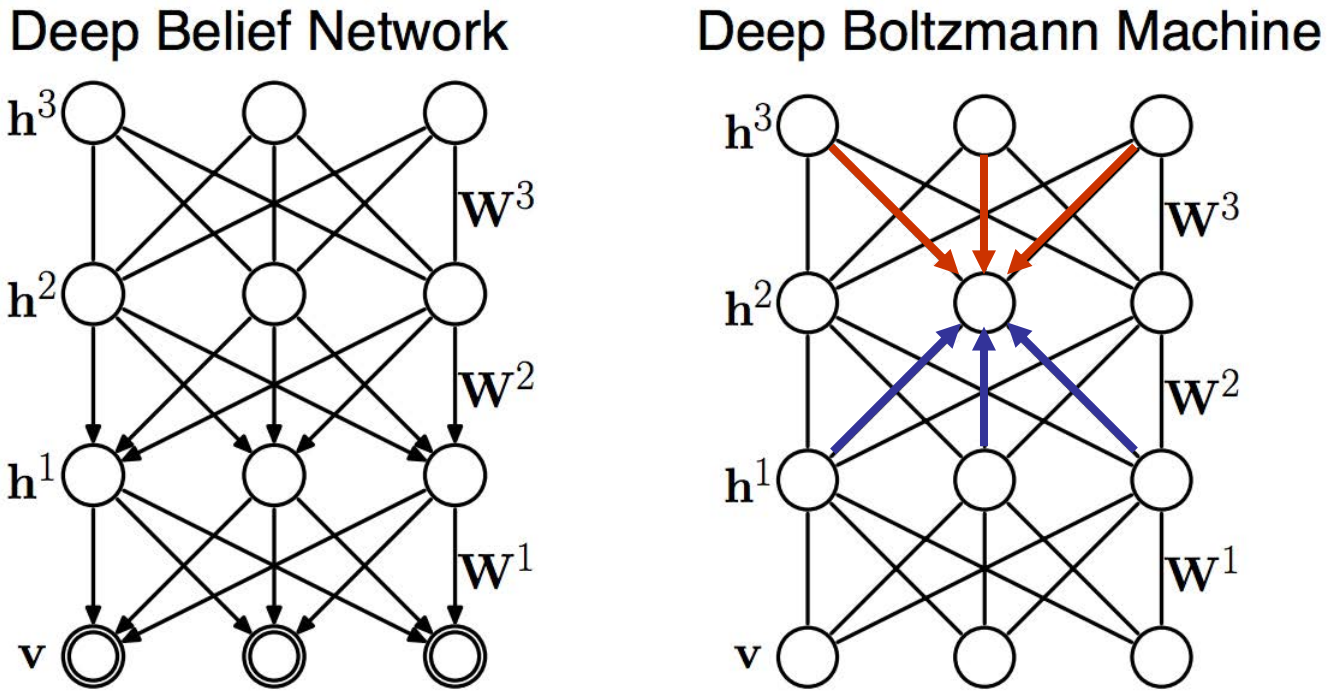
Erhan et al.: Why Does Unsupervised Pre-training Help Deep Learning? JMLR, 2010

Deep Belief Nets and Boltzmann Machines



- DBNs are hybrid graphical models (chain graphs):
  - Inference in DBNs is problematic due to explaining away effect
  - Training: greedy pre-training + ad-hoc fine-tuning; no proper joint training
  - Approximate inference is feed-forward

Deep Belief Nets and Boltzmann Machines



- DBMs are fully un-directed models (Markov random fields):
  - Can be trained similarly as RBMs via MCMC (Hinton & Sejnowski, 1983)
  - Use a variational approximation of the data distribution for faster training (Salakhutdinov & Hinton, 2009)
  - Similarly, can be used to initialize other networks for downstream tasks

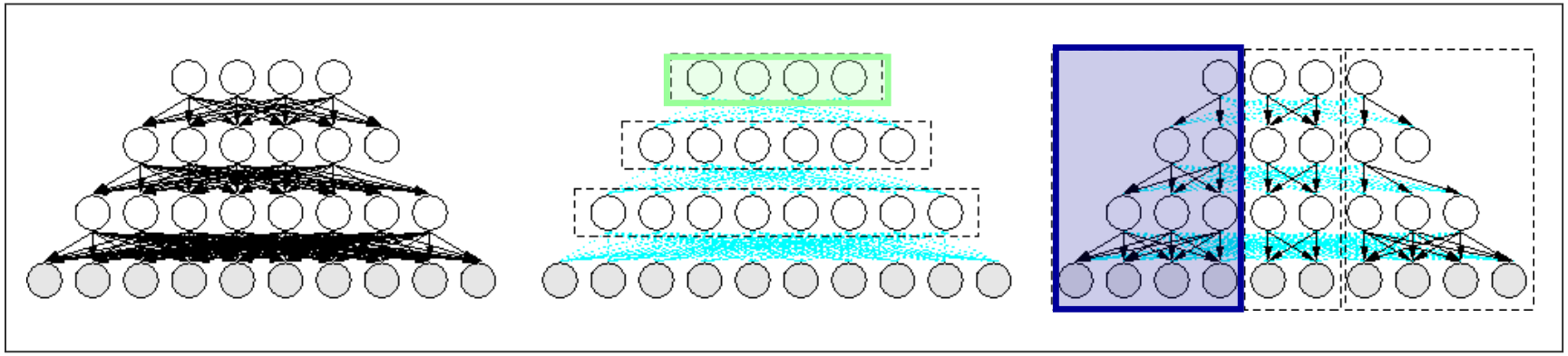
Graphical models vs. Deep networks

- A few critical points to note about all these models:
  - The primary goal of deep generative models is to represent the distribution of the observable variables. Adding layers of hidden variables allows to represent increasingly more complex distributions.
  - Hidden variables are secondary (auxiliary) elements used to facilitate learning of complex dependencies between the observables.
  - Training of the model is ad-hoc, but what matters is the quality of learned hidden representations.
  - Representations are judged by their usefulness on a downstream task (the probabilistic meaning of the model is often discarded at the end).
- In contrast, classical graphical models are often concerned with the correctness of learning and inference of all variables

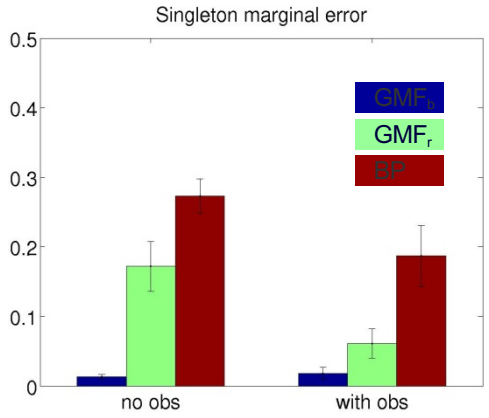
An old study of belief networks from the GM standpoint

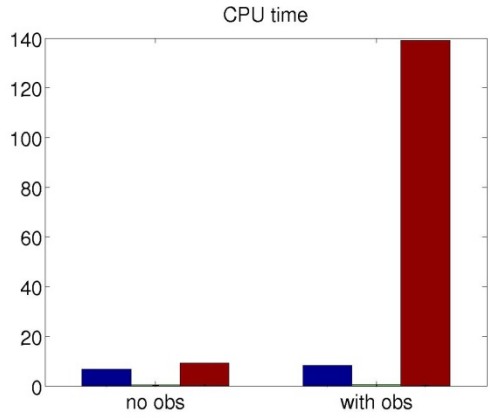
[Xing, Russell, Jordan, UAI 2003]

Mean-field partitions of a sigmoid belief network for subsequent GMF inference



Study focused on only inference/learning accuracy, speed, and partition





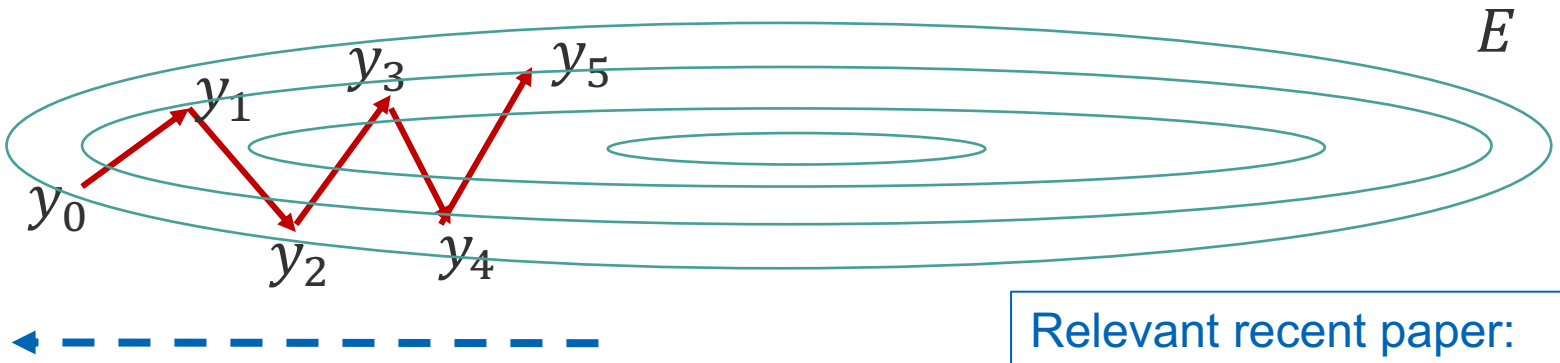
"Optimize" how to optimize via truncation & re-opt

- Energy-based modeling of the structured output (CRF)

$$\mathbf{y}^*(\mathbf{x};\mathbf{w}) := \arg\min_{\mathbf{y}} E(\mathbf{y},\mathbf{x};\mathbf{w})$$

- Unroll the optimization algorithm for a fixed number of steps (Domke, 2012)

$$\mathbf{y}^*(\mathbf{x};\mathbf{w}) = \text{opt-alg}_{\mathbf{y}} E(\mathbf{y},\mathbf{x};\mathbf{w})$$



We can backprop through the optimization steps since they are just a sequence of computations

Relevant recent paper:
Anrychowicz et al.: *Learning to learn by gradient descent by gradient descent*. 2016.

Dealing with structured prediction

- Energy-based modeling of the structured output (CRF)

$$\mathbf{y}^*(\mathbf{x};\mathbf{w}) := \arg\min_{\mathbf{y}} E(\mathbf{y},\mathbf{x};\mathbf{w})$$

- Unroll the optimization algorithm for a fixed number of steps (Domke, 2012)

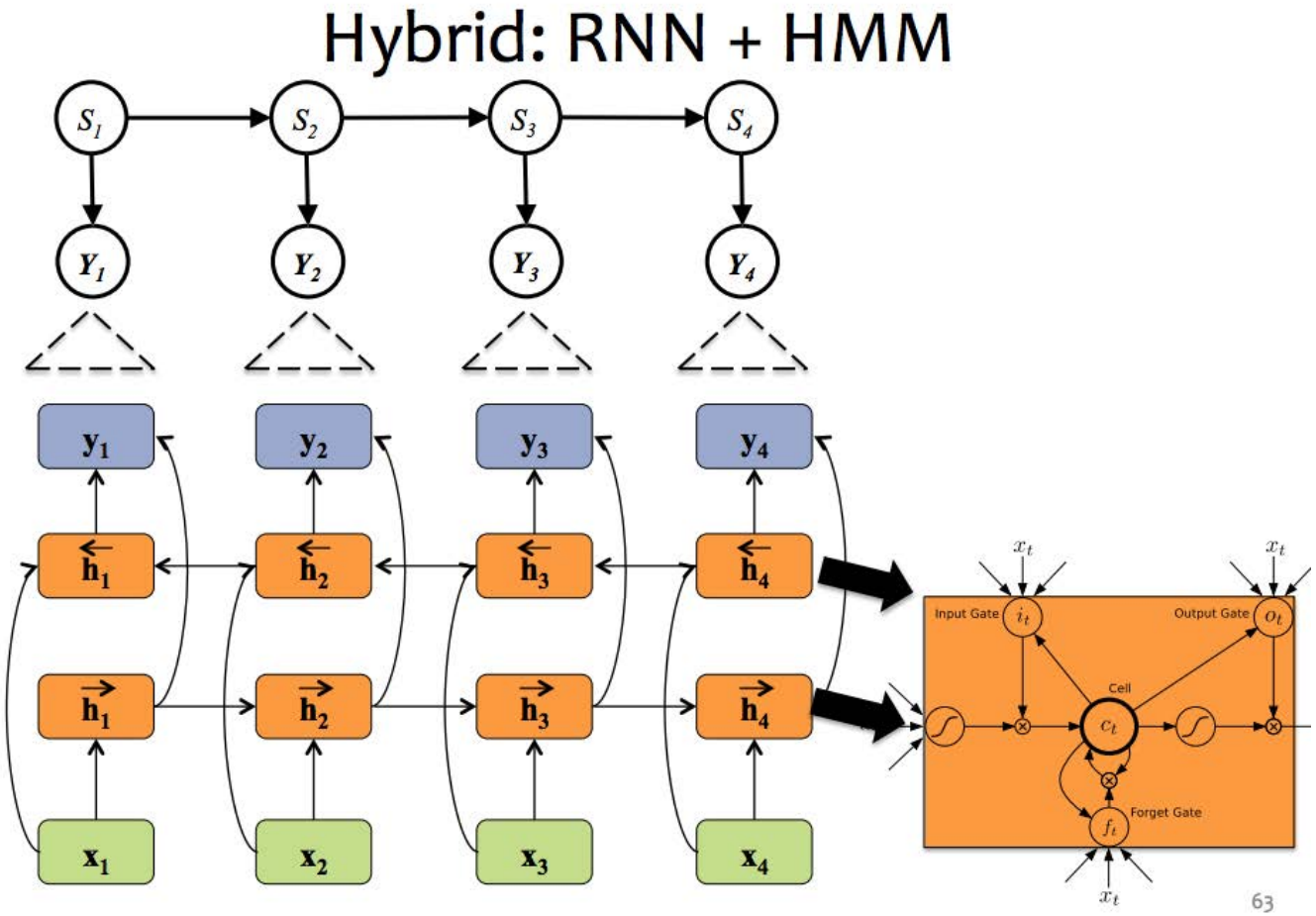
$$\mathbf{y}^*(\mathbf{x};\mathbf{w}) = \text{opt-alg}_{\mathbf{y}} E(\mathbf{y},\mathbf{x};\mathbf{w})$$

- We can think of y* as some non-linear differentiable function of the inputs and weights → impose some loss and optimize it as any other standard computation graph using backprop!
- Similarly, message passing based inference algorithms can be truncated and converted into computational graphs (Domke, 2011; Stoyanov et al., 2011)

Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
  - Historical remarks: early days of neural networks
  - Modern building blocks: units, layers, activations functions, loss functions, etc.
  - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
  - Graphical models vs. computational graphs
  - Sigmoid Belief Networks as graphical models
  - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
  - Using outputs of NNs as inputs to GMs
  - GMs with potential functions represented by NNs
  - NNs with structured outputs
- Bayesian Learning of NNs
  - Bayesian learning of NN parameters
  - Deep kernel learning

Combining sequential NNs and GMs



slide courtesy: Matt Gormley

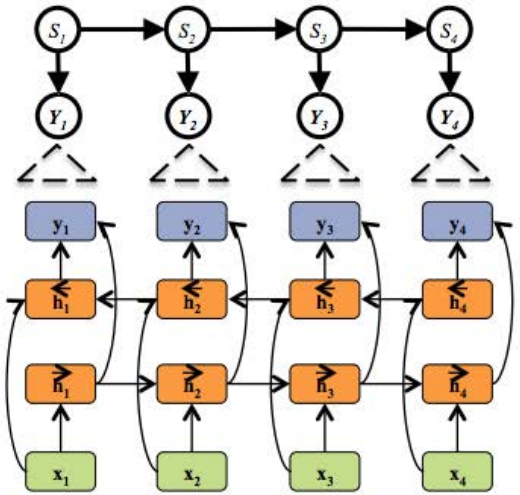
Combining sequential NNs and GMs

Hybrid: RNN + HMM

(Graves et al., 2013)

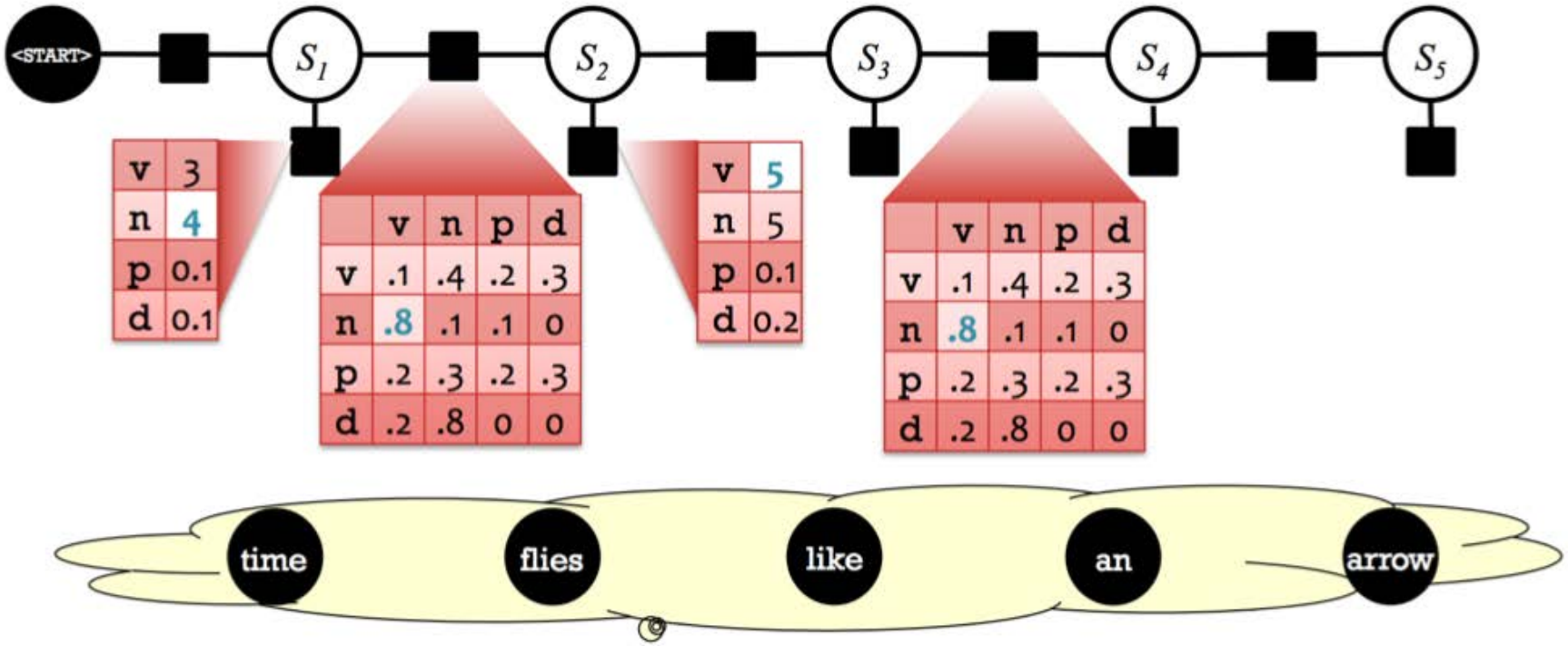
The model, inference, and learning can be **analogous** to our NN + HMM hybrid

- **Objective:** log-likelihood
- **Model:** HMM/Gaussian emissions
- **Inference:** forward-backward algorithm
- **Learning:** SGD with gradient by backpropagation



slide courtesy: Matt Gormley

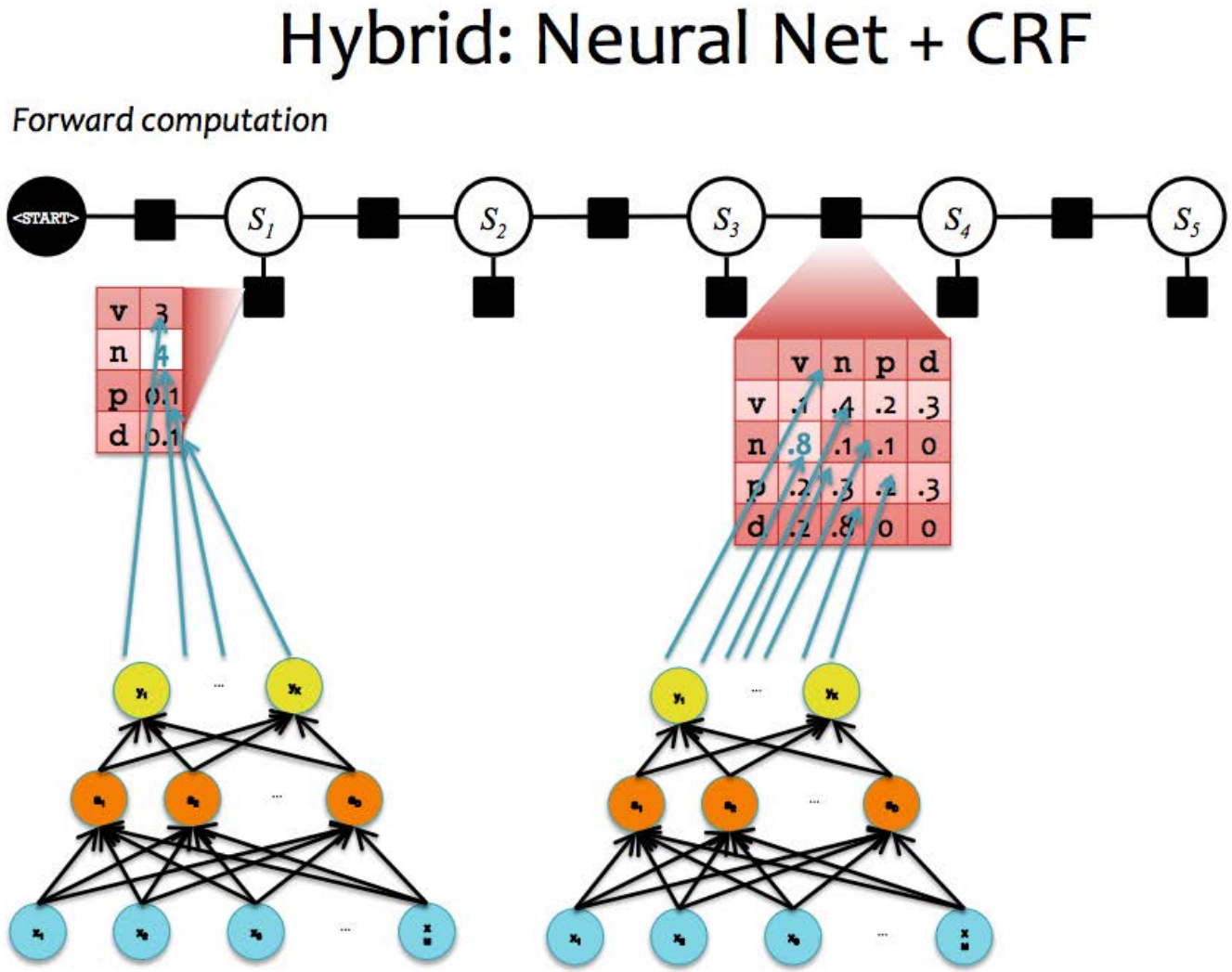
Hybrid NNs + conditional GMs



- In a standard CRF, each of the factor cells is a parameter.
- In a hybrid model, these values are computed by a neural network.

slide courtesy: Matt Gormley

Hybrid NNs + conditional GMs



71

slide courtesy: Matt Gormley

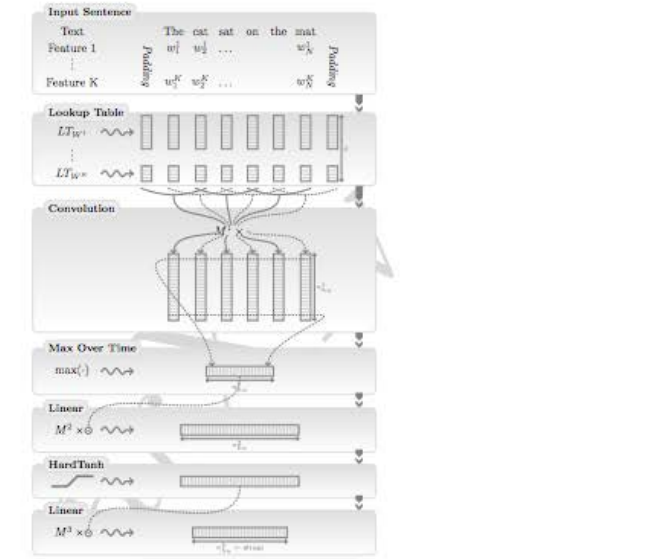
Hybrid NNs + conditional GMs

(Collobert & Weston, 2011)

Hybrid: CNN + CRF

"NN + SLL"

- Model: Convolutional Neural Network (CNN) with **linear-chain CRF**
- Training objective: maximize **sentence-level likelihood** (SLL)



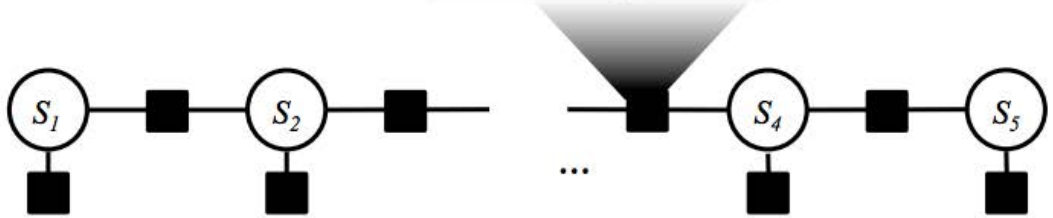


Figure from (Collobert & Weston, 2011)

slide courtesy: Matt Gormley

Using GMs as Prediction Explanations

Satellite imagery

Meaningful attributes

01 Nightlight intensity	09 Avg. vegetation dec.
02 Is urban	10 Avg. dist. to market
03 Has electricity	11 Avg. dist. to road
04 Has generator	12 Num. of rooms
05 Avg. temperature	13 Dist. to water src.
06 Avg. percipitation	14 Water usage p/ day
07 Vegetation	15 Is water payed
08 Avg. vegetation inc.	16 HH type: BQ

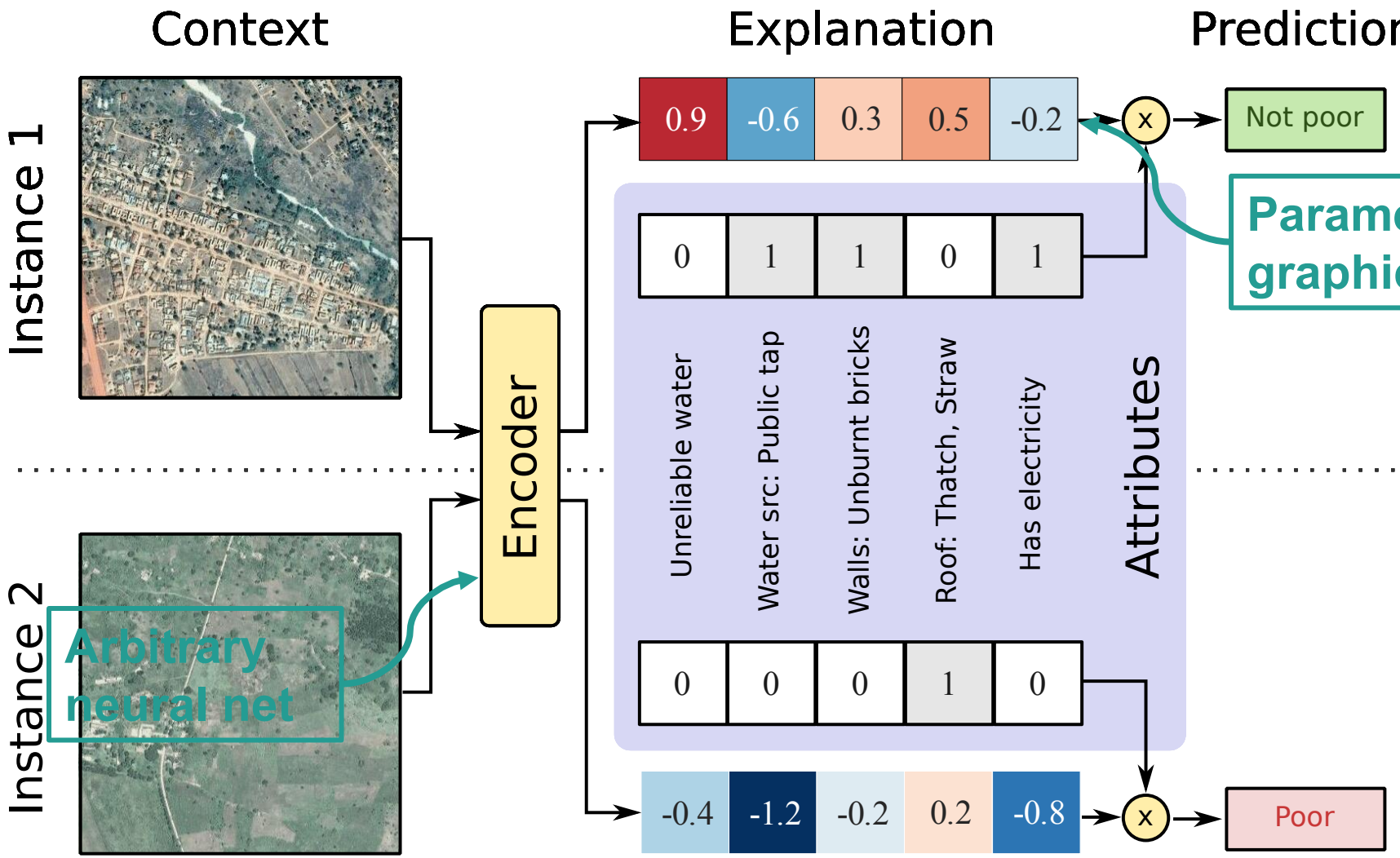
Al-Shedivat, Dubey, Xing, arXiv, 2017

Using GMs as Prediction Explanations

How do we build a powerful predictive model whose predictions we can interpret in terms of semantically meaningful features?

Al-Shedivat, Dubey, Xing, arXiv, 2017

Contextual Explanation Networks (CENs)

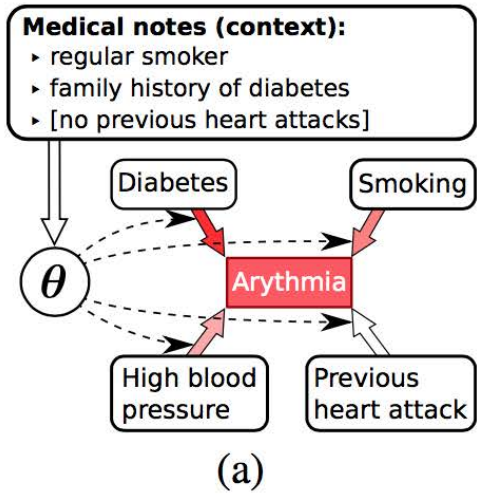


Notes:

- The final prediction is made by a linear GM.
- Each coefficient assigns a weight to a meaningful attribute.
- Allows us to judge predictions in terms of GMs produced by the context encoder.

Al-Shedivat, Dubey, Xing, arXiv, 2017

Contextual Explanation Networks (CENs)

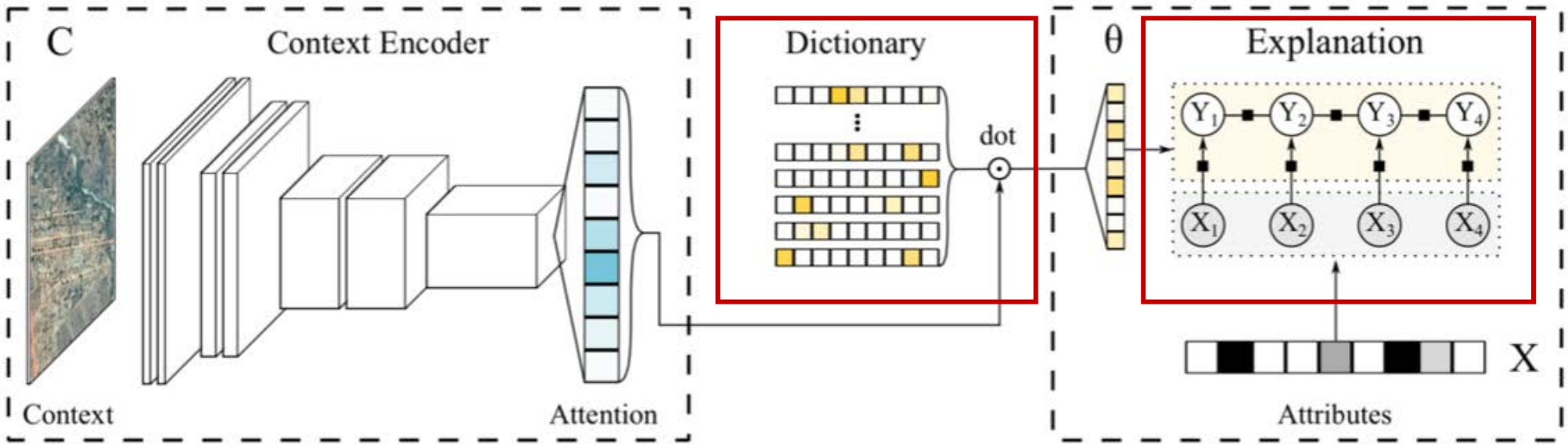


(a)

- General idea: Use deep neural nets to **generate parameters** for graphical models applicable in a given context (e.g., for a given patient).
- Produced GMs are used to make the final prediction ⇒ 100% fidelity and consistency.
- GMs are built on top of semantically meaningful variables (not deep embeddings!) and can be used as **explanations** for each prediction.

Al-Shedivat, Dubey, Xing, arXiv, 2017

CEN: Implementation Details

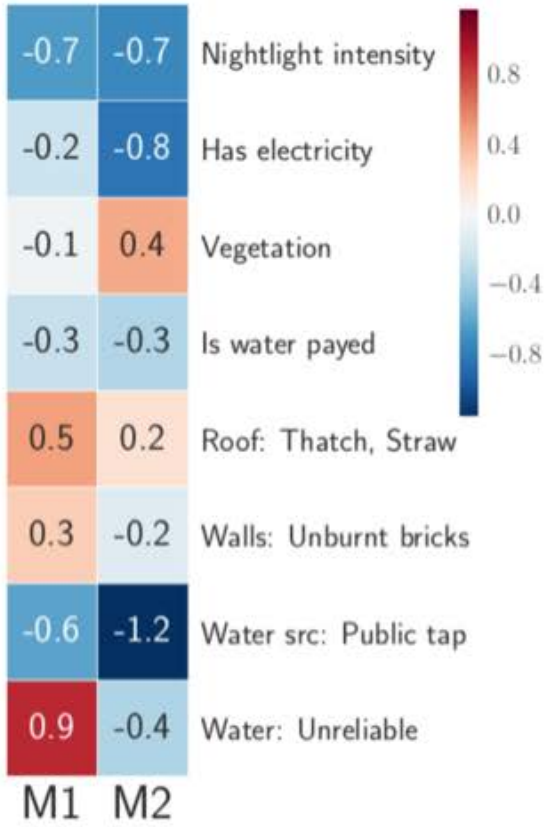


Workflow:

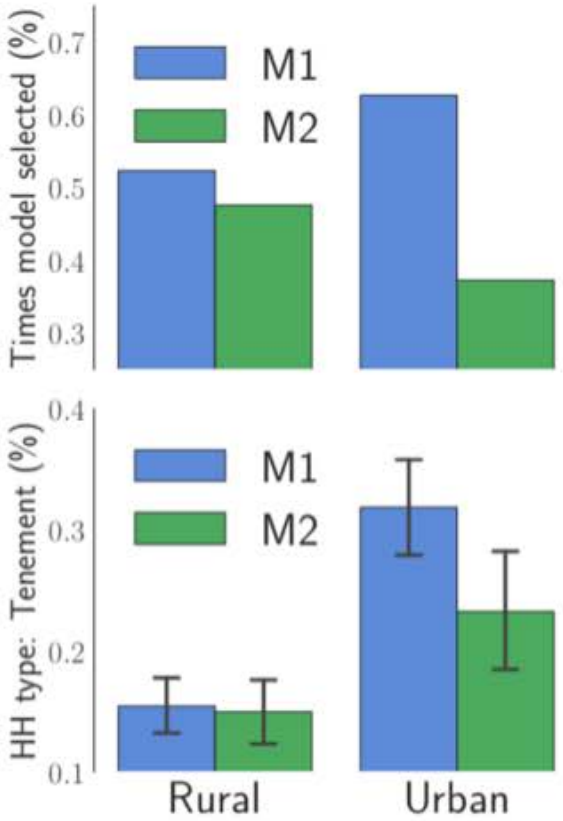
- Maintain a (sparse) dictionary of GM parameters.
- Process complex inputs (images, text, time series, etc.) using deep nets; use soft attention to either select or combine models from the dictionary.
- Use constructed GMs (e.g., CRFs) to make predictions.
- Inspect GM parameters to understand the reasoning behind predictions.

Al-Shedivat, Dubey, Xing, arXiv, 2017

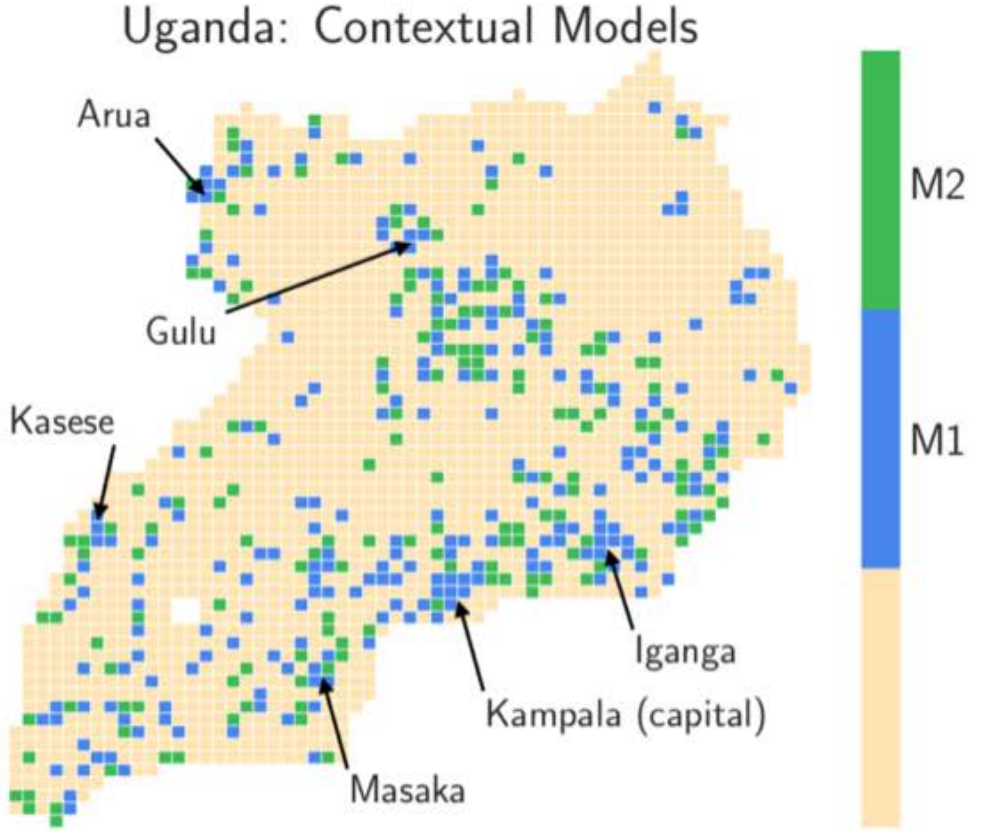
Results: imagery as context



(a)



(b)



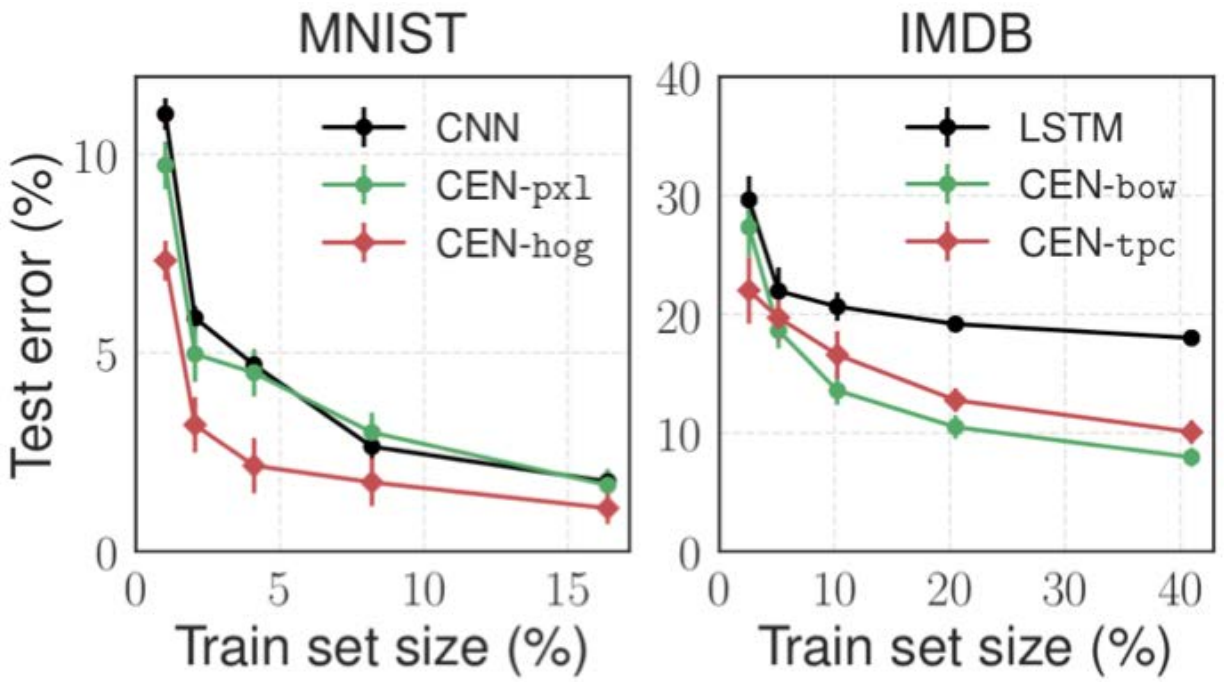
(c)

Based on the imagery, CEN learns to select different models for urban and rural areas

Al-Shedivat, Dubey, Xing, arXiv, 2017

Results: classical image & text datasets

MNIST		CIFAR10	
Model	**Err (%)**	**Model**	**Err (%)**
LR_{pxl}	8.00	LR_{pxl}	60.1
LR_{hog}	2.98	LR_{hog}	48.6
CNN	**0.75**	VGG	9.4
MoE_{pxl}	1.23	MoE_{pxl}	13.0
MoE_{hog}	1.10	MoE_{hog}	11.7
CEN_{pxl}	**0.76**	CEN_{pxl}	9.6
CEN_{hog}	**0.73**	CEN_{hog}	**9.2**



Same performance as vanilla deep networks; no compute overhead.

Predicting via explanation regularizes the model when there is not enough data.

Al-Shedivat, Dubey, Xing, arXiv, 2017

Results: classical image & text datasets

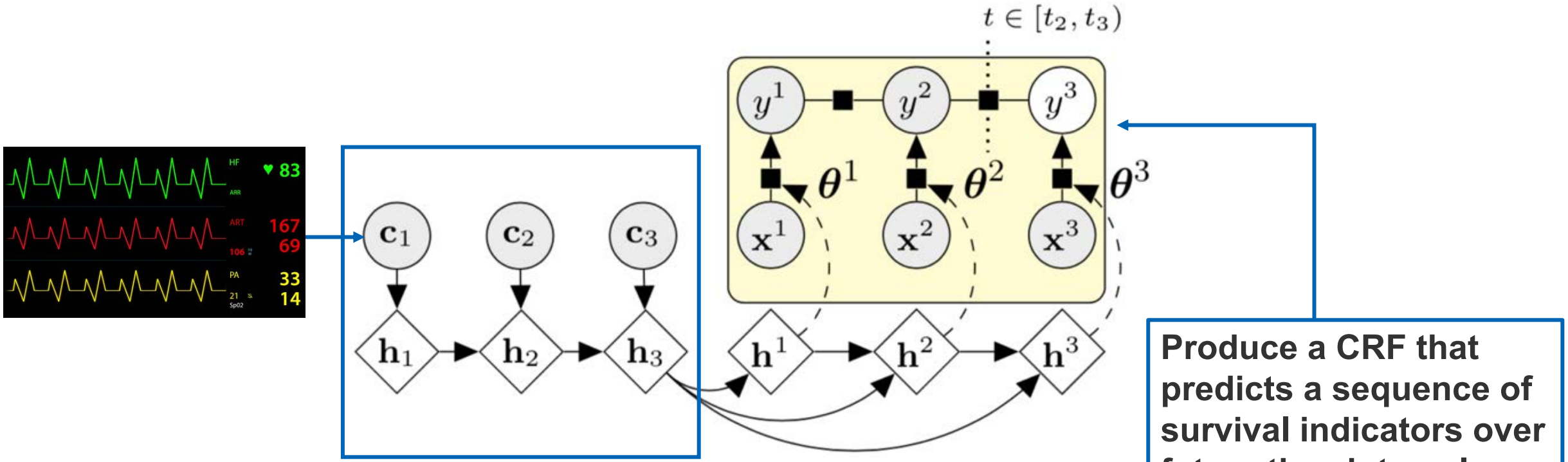
Method	Error
Paragraph Vector (Le and Mikolov, 2014)	7.42%
SA-LSTM with joint training (Dai and Le, 2015)	14.70%
LSTM with tuning and dropout (Dai and Le, 2015)	13.50%
LSTM initialized with word2vec embeddings (Dai and Le, 2015)	10.00%
SA-LSTM with linear gain (Dai and Le, 2015)	9.17%
LM-TM (Dai and Le, 2015)	7.64%
SA-LSTM (Dai and Le, 2015)	7.24%
Virtual Adversarial (Miyato et al., 2016)	**5.94 ± 0.12%**
TopicRNN (Dieng et al., 2017)	6.28 %
CEN-bow	**5.92 ± 0.05 %**
CEN-topic	**6.25 ± 0.09 %**

Semi-supervised training

Only supervised training (!!)

Al-Shedivat, Dubey, Xing, arXiv, 2017

CEN architectures for survival analysis



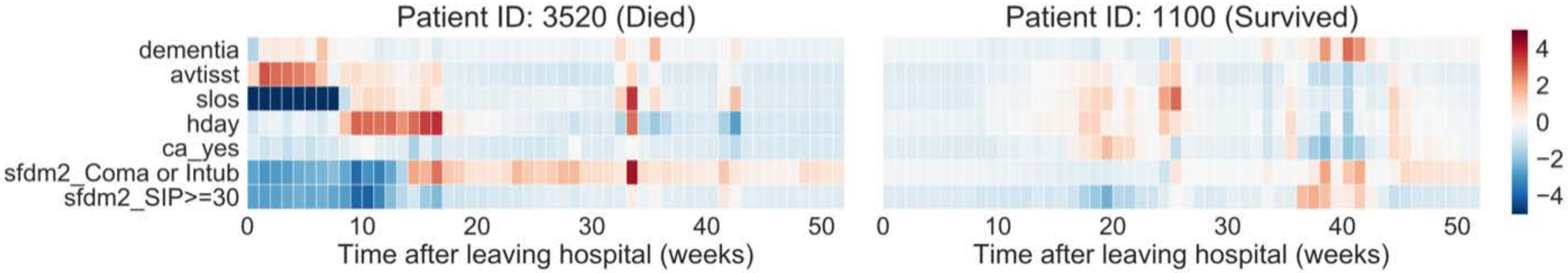
Produce a CRF that predicts a sequence of survival indicators over future time intervals.

Encode a sequence of observations for a patient (e.g., vitals/tests measured in ICU).

Al-Shedivat, Dubey, Xing, arXiv, 2017

Results: survival analysis

SUPPORT2					PhysioNet Challenge 2012				
Model	Acc@25	Acc@50	Acc@75	RAE	Model	Acc@25	Acc@50	Acc@75	RAE
Cox	84.1	73.7	47.6	0.90	Cox	93.0	69.6	49.1	0.24
Aalen	87.1	66.2	45.8	0.98	Aalen	93.3	78.7	57.1	0.31
CRF	84.4	89.3	79.2	0.59	CRF	93.2	85.1	65.6	0.14
MLP-CRF	**87.7**	89.6	80.1	0.62	LSTM-CRF	93.9	86.3	68.1	**0.11**
MLP-CEN	85.5	**90.8**	**81.9**	**0.56**	LSTM-CEN	**94.8**	**87.5**	**70.1**	**0.09**



Outline

- An overview of the DL components
  - Historical remarks: early days of neural networks
  - Modern building blocks: units, layers, activations functions, loss functions, etc.
  - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
  - Graphical models vs. computational graphs
  - Sigmoid Belief Networks as graphical models
  - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
  - Using outputs of NNs as inputs to GMs
  - GMs with potential functions represented by NNs
  - NNs with structured outputs
- Bayesian Learning of NNs
  - Bayesian learning of NN parameters
  - Deep kernel learning

Bayesian learning of NNs

- A neural network as a probabilistic model:
  - Likelihood: $p(y|x, \boldsymbol{\theta})$
    - Categorical distribution for classification ⇒ cross-entropy loss
    - Gaussian distribution for regression ⇒ squared loss
  - Prior on parameters: $p(\boldsymbol{\theta})$
- Maximum a posteriori (MAP) solution:
  - $\boldsymbol{\theta}^{MAP} = \text{argmax}_\theta \log p(y|x, \boldsymbol{\theta}) p(\boldsymbol{\theta})$
  - Gaussian prior ⇒ L2 regularization
  - Laplace prior ⇒ L1 regularization
- Bayesian learning [MacKay 1992, Neal 1996, de Freitas 2003]
  - Posterior: $p(\boldsymbol{\theta}|x, y)$
  - Variational inference with approximate posterior $q(\boldsymbol{\theta})$

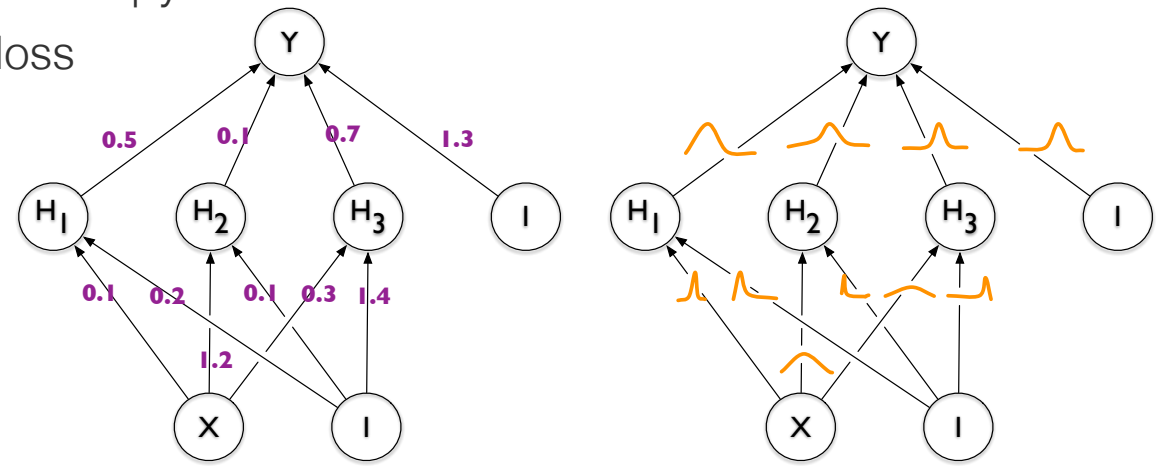


Figure courtesy: Blundell et al, 2016

Bayesian learning of NNs

- Variational inference (in a nutshell):

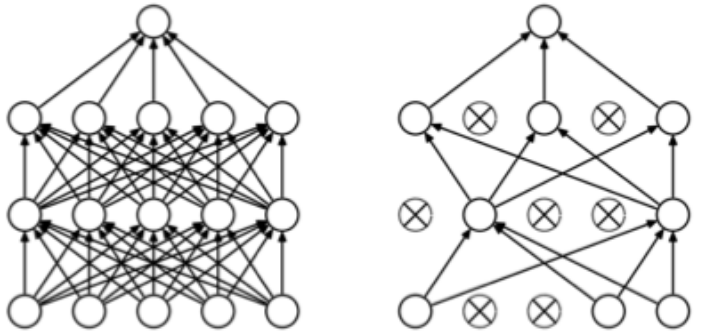
$$\min_q F(D, \boldsymbol{\theta}) = \mathrm{KL}\big(q(\boldsymbol{\theta}) || \, p(\boldsymbol{\theta}|D)\big) - \mathrm{E}_{q(\boldsymbol{\theta})}[\log p(D|\boldsymbol{\theta})]$$

$$\min_q F(D, \boldsymbol{\theta}) = \mathrm{KL}\big(q(\boldsymbol{\theta}) || \, p(\boldsymbol{\theta}|D)\big) - \sum_i \log p(D|\boldsymbol{\theta}_i)$$

  where $\boldsymbol{\theta_i} \sim q(\boldsymbol{\theta})$; KL term can be approximated similarly

- We can define $q(\boldsymbol{\theta})$ as a diagonal Gaussian or full-covariance Gaussian
- Alternatively, $q(\boldsymbol{\theta})$ can be defined implicitly, e.g. via dropout [Gal & Ghahramani, 2016]

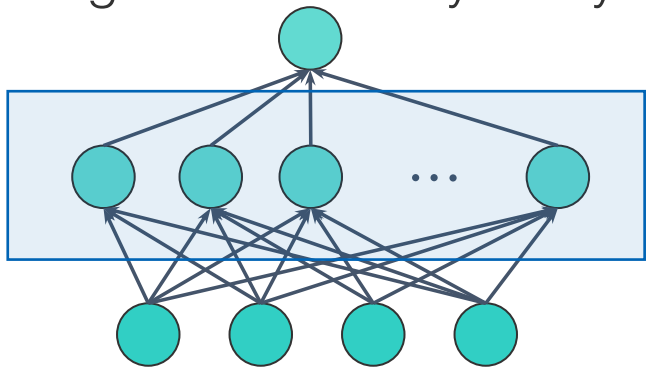
$$\boldsymbol{\theta} = \boldsymbol{M} \cdot \mathrm{diag}(\boldsymbol{z}),$$
$$\boldsymbol{z} \sim \mathrm{Bernoulli}(p)$$



- Dropping out neurons is equivalent to zeroing out columns of the parameter matrices (i.e., weights)
- $z_i = 0$ corresponds to i-th column of $\boldsymbol{M}$ being dropped out
  $\Rightarrow$ the procedure is equivalent to dropout of unit i [Hinton et al., 2012]
- Variational parameters are $\{\boldsymbol{M}, \boldsymbol{p}\}$

“Infinitely Wide” Deep Models

- We have seen that an ”infinitely deep” network can be explained by a proper GM, How about an “infinitely wide” one?
- Consider a neural network with a Gaussian prior on its weights an infinitely many hidden neurons in the intermediate layer.
- Turns out, if we have a certain Gaussian prior on the weights of such infinite network, it will be equivalent to a Gaussian process [Neal 1996].



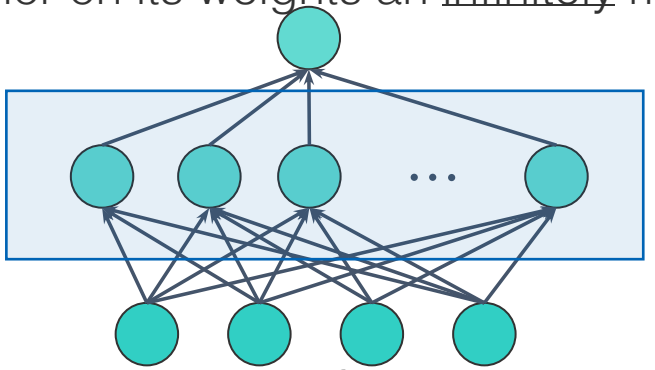
- Gaussian process (GP) is a distribution over functions:

$$
\begin{aligned}
m(\mathbf{x}) &= \mathbb{E}[f(\mathbf{x})], \\
k(\mathbf{x}, \mathbf{x}') &= \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))], \\
f(\mathbf{x}) &\sim \mathcal{GP}\big(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')\big).
\end{aligned}
$$

- When used for prediction, GPs account for correlations between the data points and can output well-calibrated predictive uncertainty estimates.

Gaussian Process and Deep Kernel Learning

- Consider a neural network with a Gaussian prior on its weights an infinitely many hidden neurons in the intermediate layer.



Infinitely many hidden units

- Certain classes of Gaussian priors for neural networks with infinitely many hidden units converge to Gaussian processes [Neal 1996]
- Deep kernel [Wilson et al., 2016]
  - Combines the inductive biases of deep model architectures with the non-parametric flexibility of Gaussian processes

$$k(x_i, x_j|\phi) \rightarrow k(g(x_i, \theta), g(x_j, \theta)|\phi, \theta) \quad \text{where } K_{ij} = k(x_i, x_j)$$

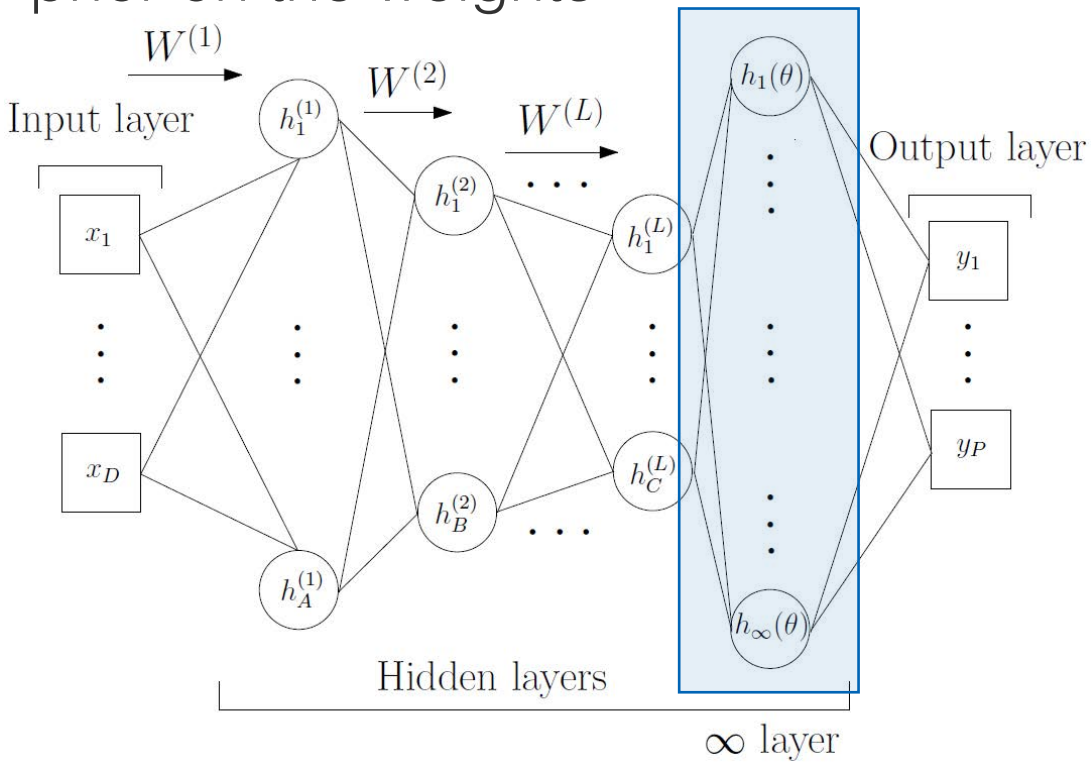
  - Starting from a base kernel $k(x_i, x_j|\phi)$, transform the inputs x as

$$p(f|\phi) = \mathcal{N}(f|m(x), K)$$
$$p(y|f) = \mathcal{N}(y|f, \beta^{-1})$$

  - Learn both kernel and neural parameters $\{\phi, \theta\}$ jointly by optimizing marginal log-likelihood (or its variational lower-bound).
  - Fast learning and inference with local kernel interpolation, structured inducing points, and Monte Carlo approximations

Gaussian Process and Deep Kernel Learning

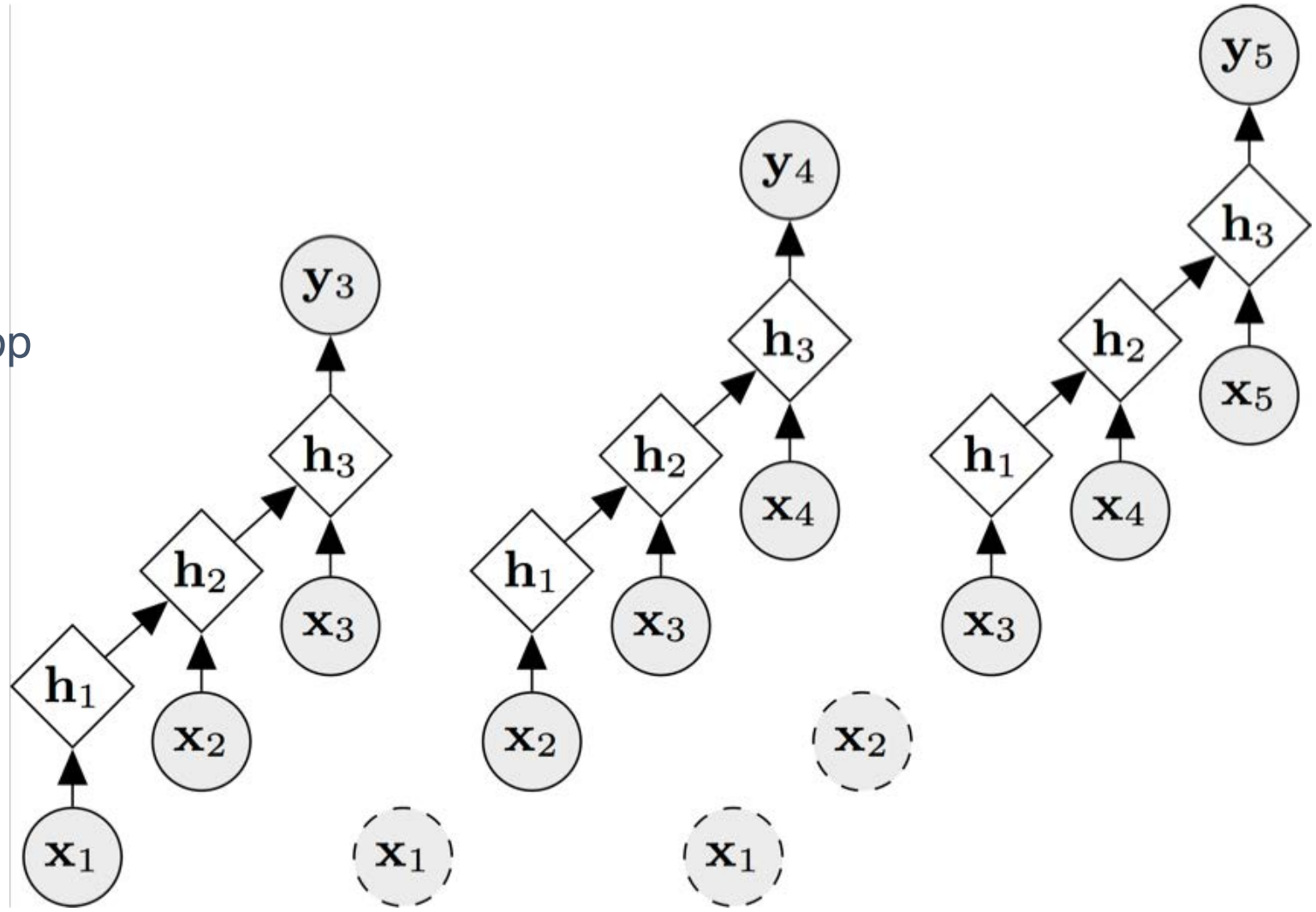
- By adding GP as a layer to a deep neural net, we can think of it as adding an infinite hidden layer with a particular prior on the weights
- Deep kernel learning [Wilson et al., 2016]
  - Combines the inductive biases of deep models with the non-parametric flexibility of Gaussian processes
  - GPs add powerful regularization to the network
  - Additionally, they provide predictive uncertainty estimates



Deep kernel learning on sequential data

What if we have data of sequential nature?

Can we still apply the same reasoning and build rich nonparametric models on top recurrent nets?

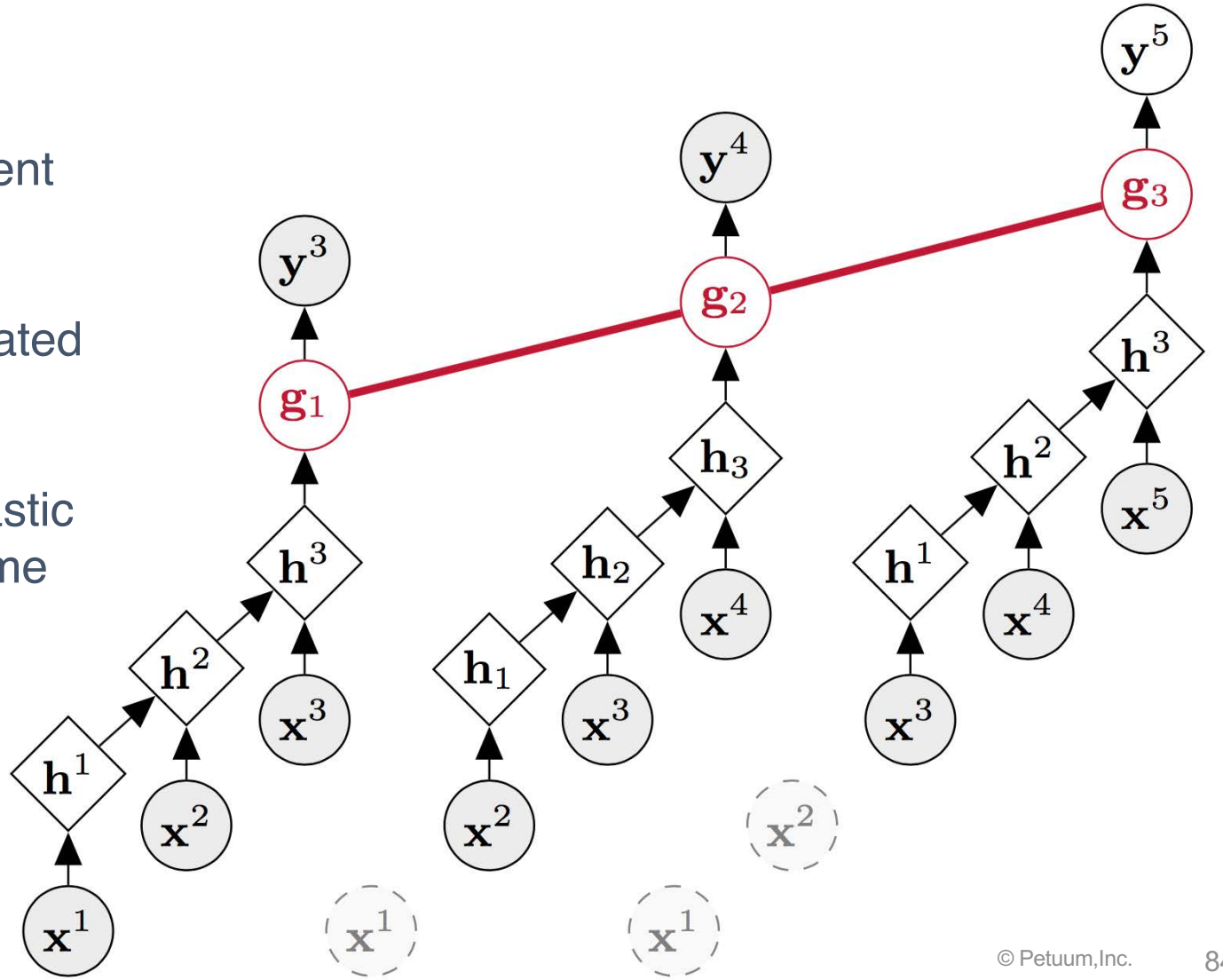


Deep kernel learning on sequential data

The answer is YES!

By adding a GP layer to a recurrent network, we effectively correlate samples across time and get predictions along with well calibrated uncertainty estimates.

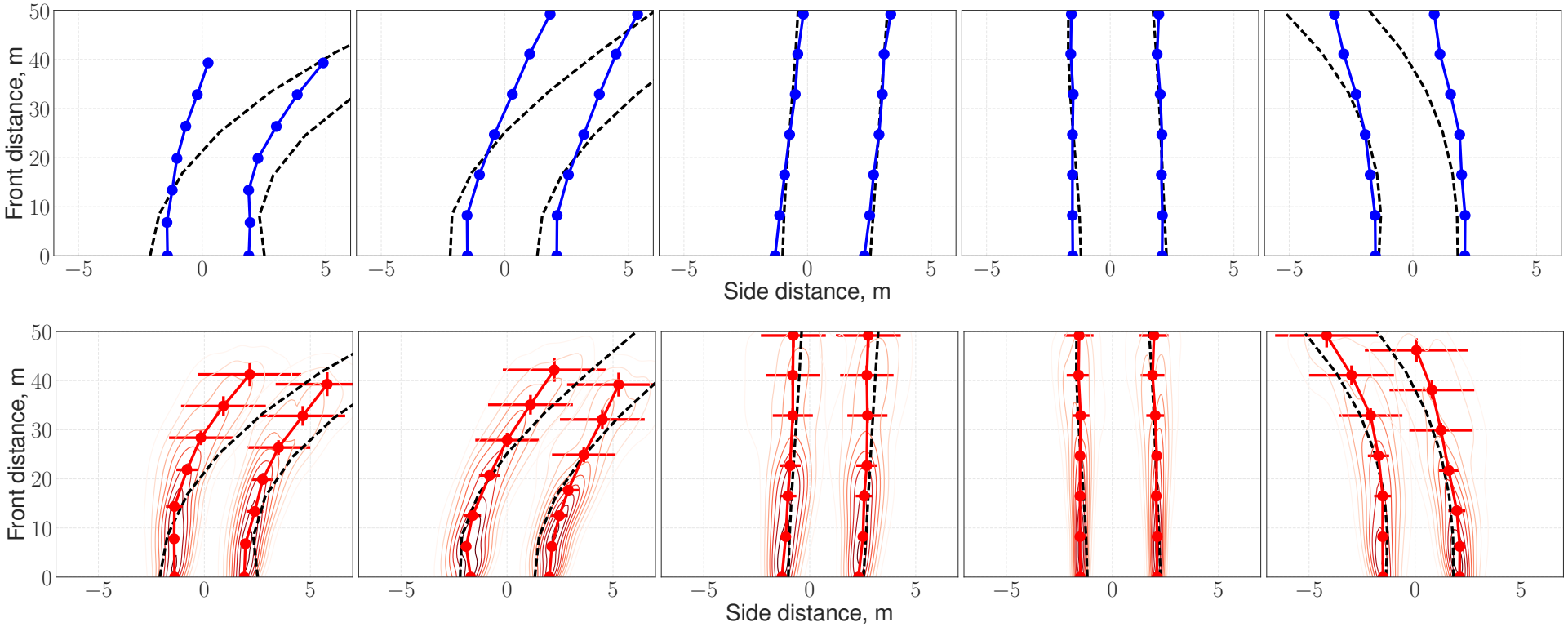
To train such model using stochastic techniques however requires some additional care (see our paper).



Al-Shedivat et al., JMLR, 2017

Deep kernel learning on sequential data

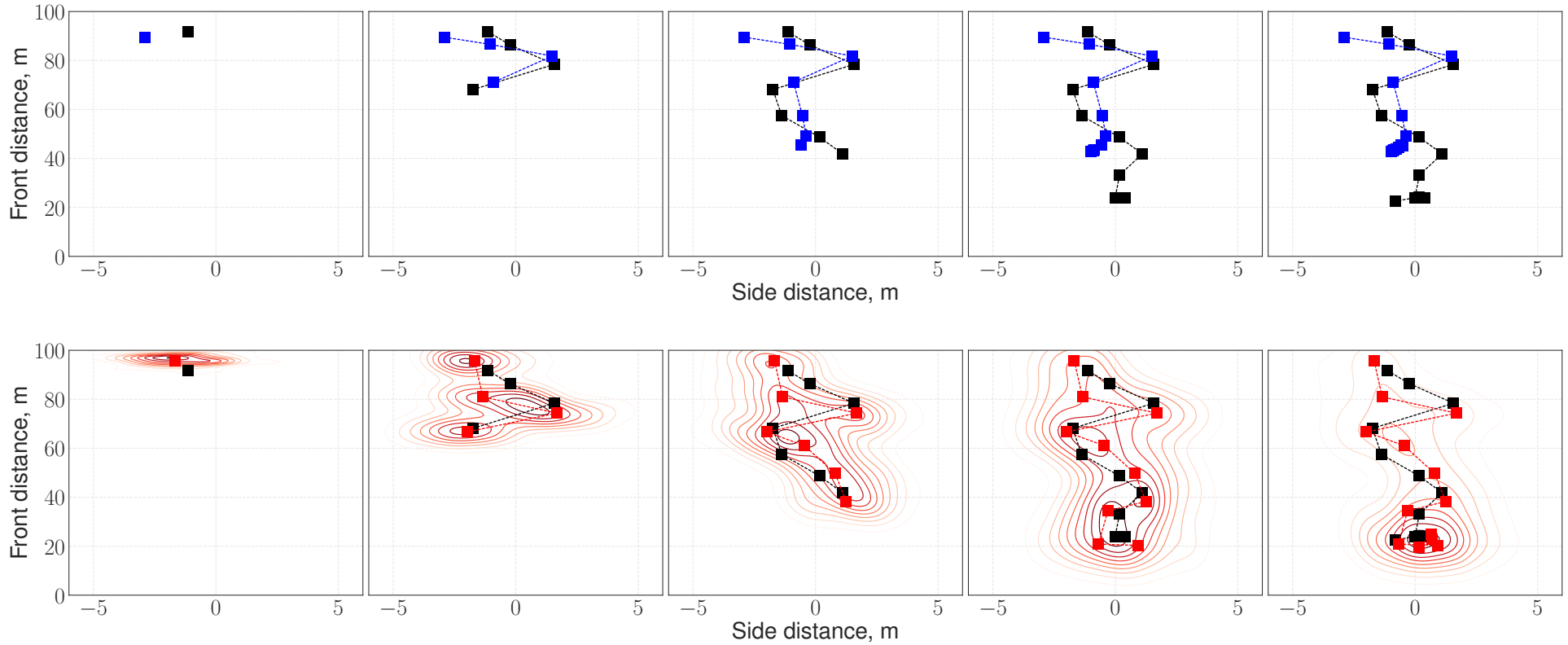
Lane prediction: LSTM vs GP-LSTM



Al-Shedivat et al., JMLR, 2017

Deep kernel learning on sequential data

Lead vehicle prediction: LSTM vs GP-LSTM



Al-Shedivat et al., JMLR, 2017

Conclusion

- DL & GM: the fields are similar in the beginning (structure, energy, etc.), and then diverge to their own signature pipelines
- DL: most effort is directed to comparing different architectures and their components (models are driven by evaluating empirical performance on a downstream tasks)
  - DL models are good at learning robust hierarchical representations from the data and suitable for simple reasoning (call it “low-level cognition”)
- GM: the effort is directed towards improving inference accuracy and convergence speed
  - GMs are best for provably correct inference and suitable for high-level complex reasoning tasks (call it “high-level cognition”)
- Convergence of both fields is very promising!
  - Next part: a unified view of deep generative models in the GM interpretation

PETUUM

Part-II
A Unified Framework of Deep Generative Models

Plan

- Statistical And Algorithmic Foundation and Insight of Deep Learning
- A Unified Framework of Deep Generative Models
- Computational Mechanisms: Programming Platforms and Distributed Deep Learning Architectures

Outline

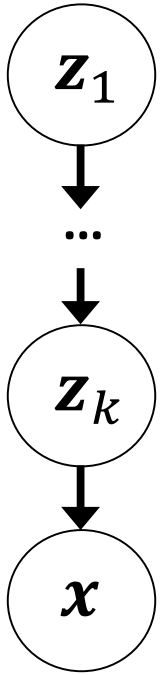
- Overview of advances in deep generative models
- Theoretical Basis of deep generative models
  - Wake sleep algorithm
  - Variational autoencoders
  - Generative adversarial networks
- A unified view of deep generative models
  - new formulations of deep generative models
  - Symmetric modeling of latent and visible variables

Outline

- Overview of advances in deep generative models
- Theoretical Basis of deep generative models
  - Wake sleep algorithm
  - Variational autoencoders
  - Generative adversarial networks
- A unified view of deep generative models
  - new formulations of deep generative models
  - Symmetric modeling of latent and visible variables

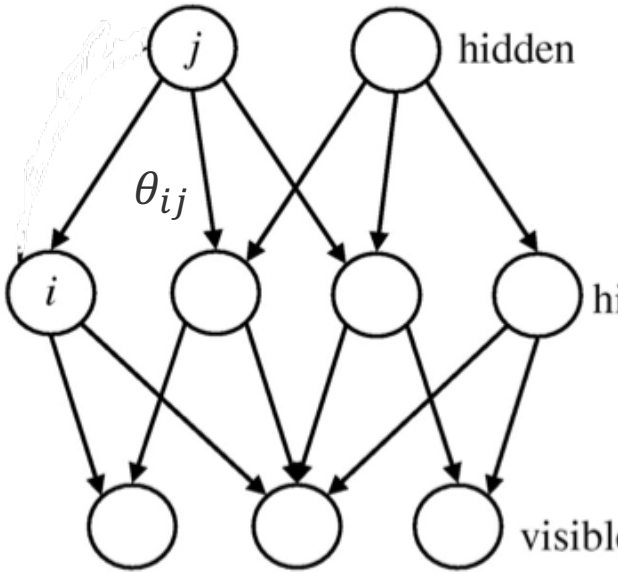
Deep generative models

- Define probabilistic distributions over a set of variables
- "Deep" means multiple layers of hidden variables!



Early forms of deep generative models

- Hierarchical Bayesian models
  - Sigmoid brief nets [Neal 1992]



$$\boldsymbol{z}_n^{(2)} = \{0,1\}^J$$

$$\boldsymbol{z}_n^{(1)} = \{0,1\}^I$$

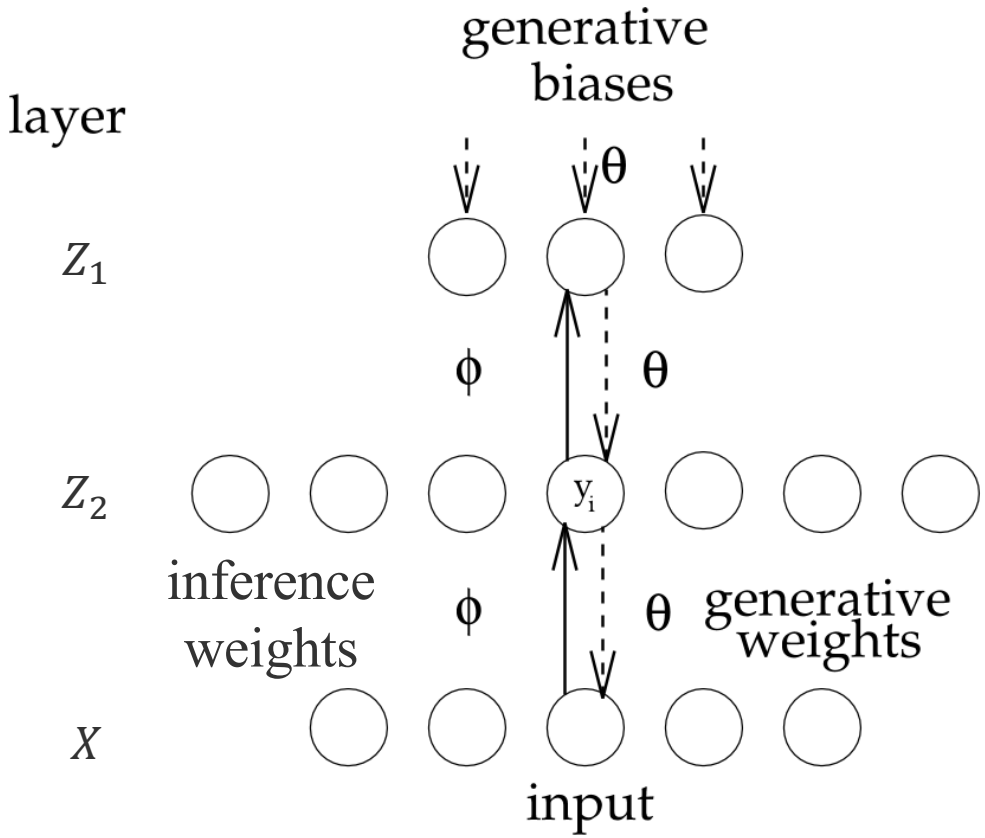
$$\boldsymbol{x}_n = \{0,1\}^K$$

$$p\left(x_{kn} = 1 \middle| \boldsymbol{\theta}_k, \boldsymbol{z}_n^{(1)}\right) = \sigma\left(\boldsymbol{\theta}_k^T \boldsymbol{z}_n^{(1)}\right)$$

$$p\left(z_{in}^{(1)} = 1 \middle| \boldsymbol{\theta}_i, \boldsymbol{z}_n^{(2)}\right) = \sigma\left(\boldsymbol{\theta}_i^T \boldsymbol{z}_n^{(2)}\right)$$

Early forms of deep generative models

- Hierarchical Bayesian models
  - Sigmoid brief nets [Neal 1992]
- Neural network models
  - Helmholtz machines [Dayan et al.,1995]



[Dayan et al. 1995]

Early forms of deep generative models

- Hierarchical Bayesian models
  - Sigmoid brief nets [Neal 1992]
- Neural network models
  - Helmholtz machines [Dayan et al.,1995]
  - Predictability minimization [Schmidhuber 1995]

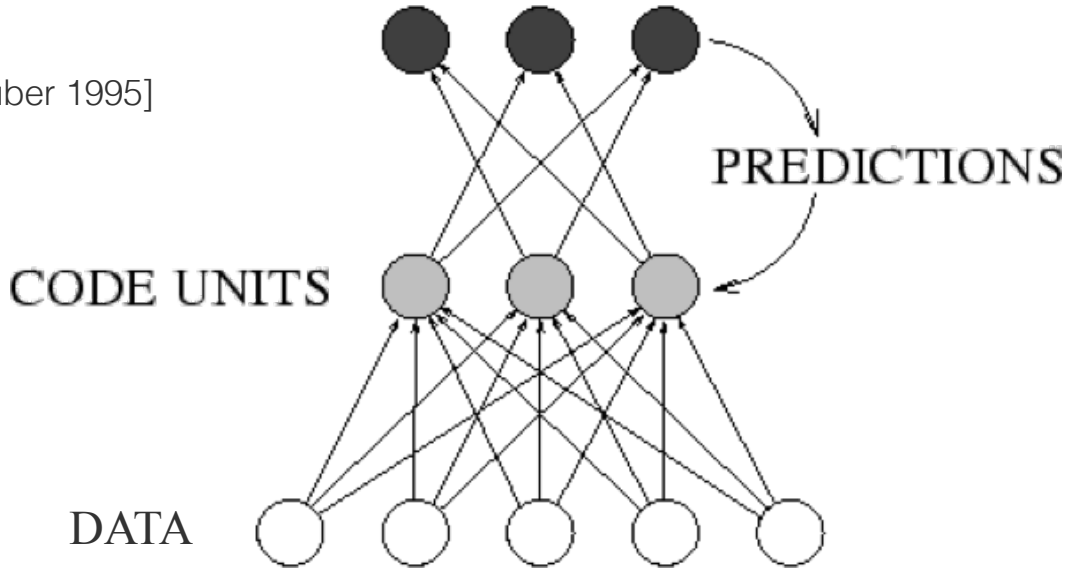


Figure courtesy: Schmidhuber 1996

Early forms of deep generative models

- Training of DGMs via an EM style framework
  - Sampling / data augmentation

    $\boldsymbol{z} = \{\boldsymbol{z}_1, \boldsymbol{z}_2\}$

    $\boldsymbol{z}_1^{new} \sim p(\boldsymbol{z}_1 | \boldsymbol{z}_2, \boldsymbol{x})$

    $\boldsymbol{z}_2^{new} \sim p(\boldsymbol{z}_2 | \boldsymbol{z}_1^{new}, \boldsymbol{x})$
  - Variational inference

    $\log p(\boldsymbol{x}) \geq \mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})] - \mathrm{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p(\boldsymbol{z})) \coloneqq \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$

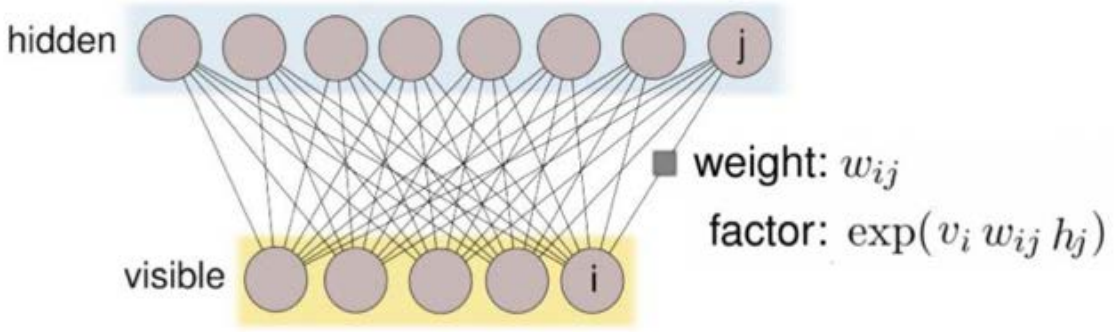
    $\max_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$
  - Wake sleep

    Wake: $\min_{\theta} \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\boldsymbol{\theta}}(x|z)]$

    Sleep: $\min_{\phi} \mathbb{E}_{p_{\boldsymbol{\theta}}(x|z)}[\log q_{\boldsymbol{\phi}}(z|x)]$

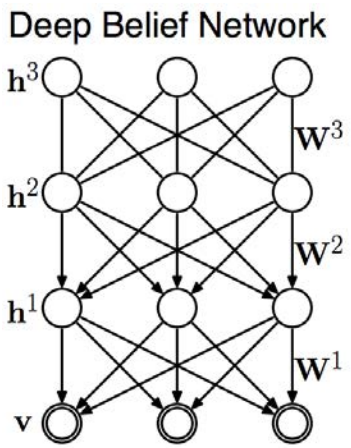
Resurgence of deep generative models

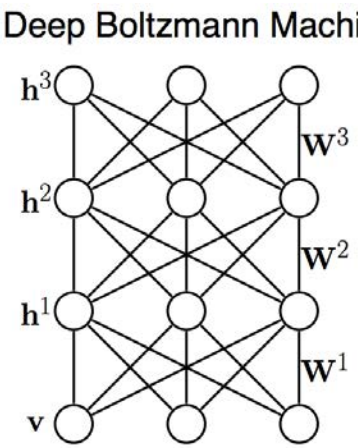
- Restricted Boltzmann machines (RBMs) [Smolensky, 1986]
  - Building blocks of deep probabilistic models



Resurgence of deep generative models

- Restricted Boltzmann machines (RBMs) [Smolensky, 1986]
  - Building blocks of deep probabilistic models
- Deep belief networks (DBNs) [Hinton et al., 2006]
  - Hybrid graphical model
  - Inference in DBNs is problematic due to explaining away
- Deep Boltzmann Machines (DBMs) [Salakhutdinov & Hinton, 2009]
  - Undirected model





Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
  / Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]

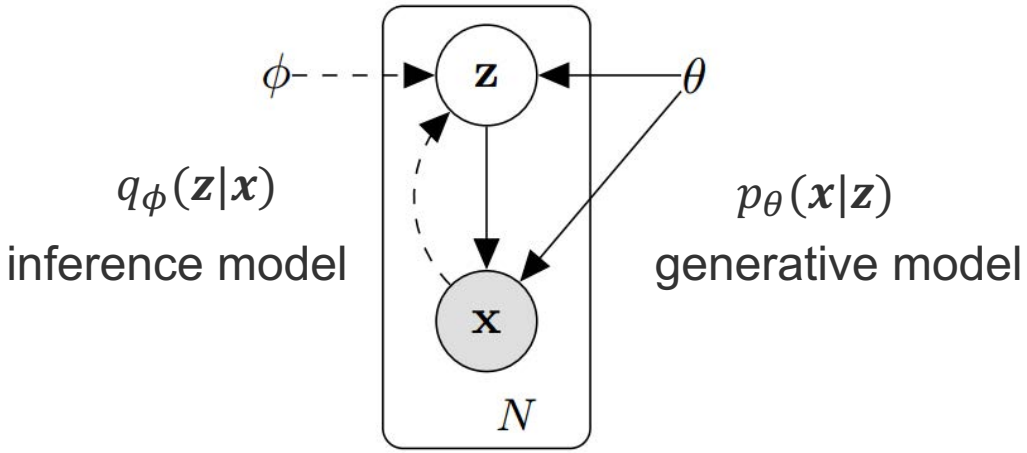
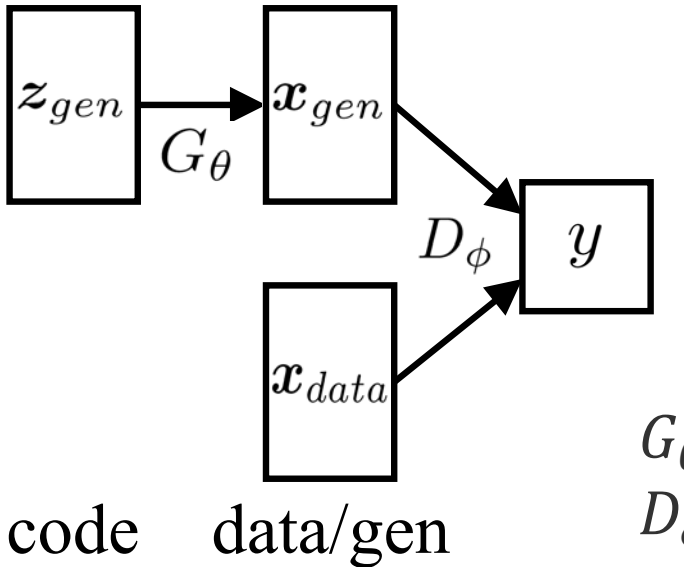


Figure courtesy: Kingma & Welling, 2014

Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
  / Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]
- Generative adversarial networks (GANs) [Goodfellow et al,. 2014]



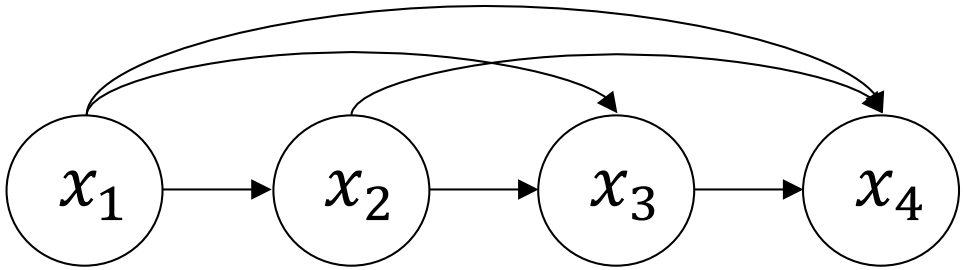
G_θ: generative model ?
D_ϕ: discriminator

Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
  / Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]
- Generative adversarial networks (GANs) [Goodfellow et al,. 2014]
- Generative moment matching networks (GMMNs) [Li et al., 2015; Dziugaite et al., 2015]

Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
  / Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]
- Generative adversarial networks (GANs) [Goodfellow et al,. 2014]
- Generative moment matching networks (GMMNs) [Li et al., 2015; Dziugaite et al., 2015]
- Autoregressive neural networks



Outline

- Overview of advances in deep generative models
- Theoretical Basis of deep generative models
  - Wake sleep algorithm
  - Variational autoencoders
  - Generative adversarial networks
- A unified view of deep generative models
  - new formulations of deep generative models
  - Symmetric modeling of latent and visible variables

Synonyms in the literature

- Posterior Distribution -> Inference model
  - Variational approximation
  - Recognition model
  - Inference network (if parameterized as neural networks)
  - Recognition network (if parameterized as neural networks)
  - (Probabilistic) encoder
- "The Model" (prior + conditional, or joint) -> Generative model
  - The (data) likelihood model
  - Generative network (if parameterized as neural networks)
  - Generator
  - (Probabilistic) decoder

Recap: Variational Inference

- Consider a generative model $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, and prior $p(\boldsymbol{z})$
  - Joint distribution: $p_\theta(\boldsymbol{x},\boldsymbol{z}) = p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$
- Assume variational distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$
- Objective: Maximize lower bound for log likelihood

$$\begin{aligned}
&\log p(\boldsymbol{x}) \\
&= KL\left(q_\phi(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right) + \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} \\
&\geq \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} \\
&:= \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x})
\end{aligned}$$

  - Equivalently, minimize free energy

$$F(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(q_\phi(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$$

Recap: Variational Inference

Maximize the variational lower bound:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL(q_{\boldsymbol{\phi}}(z|x)||p(z))$$

- E-step: maximize $\mathcal{L}$ wrt. $\boldsymbol{\phi}$, with $\boldsymbol{\theta}$ fixed

$$\max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$$

  - If closed form solutions exist:

$$q_{\boldsymbol{\phi}}^{*}(z|x) \propto \exp[\log p_{\theta}(x, z)]$$

- M-step: maximize $\mathcal{L}$ wrt. $\boldsymbol{\theta}$, with $\boldsymbol{\phi}$ fixed

$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$$

Recap: Amortized Variational Inference

- Variational distribution as an inference model $q_{\phi}(\boldsymbol{z}|\boldsymbol{x})$ with parameters $\boldsymbol{\phi}$ (which was traditionally factored over samples)
- Amortize the cost of inference by learning a **single** data-dependent inference model
- The trained inference model can be used for quick inference on new data
- Maximize the variational lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$
  - E-step: maximize $\mathcal{L}$ wrt. $\boldsymbol{\phi}$ with $\boldsymbol{\theta}$ fixed
  - M-step: maximize $\mathcal{L}$ wrt. $\boldsymbol{\theta}$ with $\boldsymbol{\phi}$ fixed

Deep generative models with amortized inference

- Helmholtz machines
- Variational autoencoders (VAEs) / Neural Variational Inference and Learning (NVIL)
- We will see later that *adversarial approaches* are also included in the list
  - Predictability minimization (PM)
  - Generative adversarial networks (GANs)

Wake Sleep Algorithm

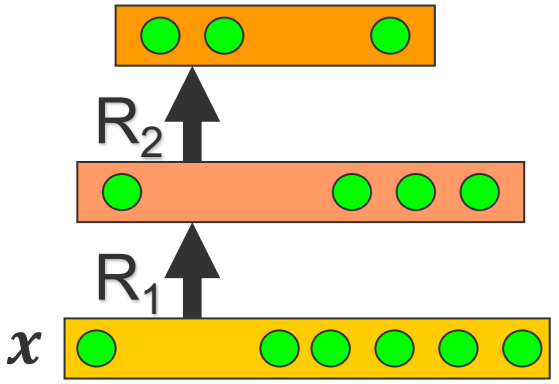
- [Hinton et al., Science 1995]
- Train a separate inference model along with the generative model
  - Generally applicable to a wide range of generative models, e.g., Helmholtz machines
- Consider a generative model $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ and prior $p(\boldsymbol{z})$
  - Joint distribution $p_\theta(\boldsymbol{x}, \boldsymbol{z}) = p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$
  - E.g., multi-layer brief nets
- Inference model $q_\phi(\boldsymbol{z}|\boldsymbol{x})$
- Maximize data log-likelihood with two steps of loss relaxation:
  - Maximize the variational lower bound of log-likelihood, or equivalently, minimize the free energy

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(q_\phi(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_\theta(\boldsymbol{z}|\boldsymbol{x}))$$

  - Minimize a different objective (reversed KLD) wrt ϕ to ease the optimization
    - Disconnect to the original variational lower bound loss

$$F'(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(p_\theta(\boldsymbol{z}|\boldsymbol{x}) \,||\, q_\phi(\boldsymbol{z}|\boldsymbol{x}))$$

Wake Sleep Algorithm



- Free energy:

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$$

- Minimize the free energy wrt. $\boldsymbol{\theta}$ of p_θ → *wake* phase

$$\max_{\boldsymbol{\theta}} \mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} [\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})]$$

  - Get samples from $q_{\boldsymbol{\phi}}(z|x)$ through inference on hidden variables
  - Use the samples as targets for updating the generative model $p_\theta(\boldsymbol{z}|\boldsymbol{x})$
  - Correspond to the variational M step

[Figure courtesy: Maei's slides]

Wake Sleep Algorithm

- Free energy:

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$$

- Minimize the free energy wrt. $\boldsymbol{\phi}$ of $q_{\phi}(\boldsymbol{z}|\boldsymbol{x})$
  - Correspond to the variational E step
  - Difficulties:
    - Optimal $q^*_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \frac{\boldsymbol{p_\theta}(\boldsymbol{z}, \boldsymbol{x})}{\int \boldsymbol{p_\theta}(\boldsymbol{z}, \boldsymbol{x})\, \boldsymbol{dz}}$ intractable
    - High variance of direct gradient estimate $\nabla_\phi F(\theta, \phi; x) = \cdots + \nabla_\phi \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(z, x)] + \cdots$
      - Gradient estimate with the log-derivative trick:

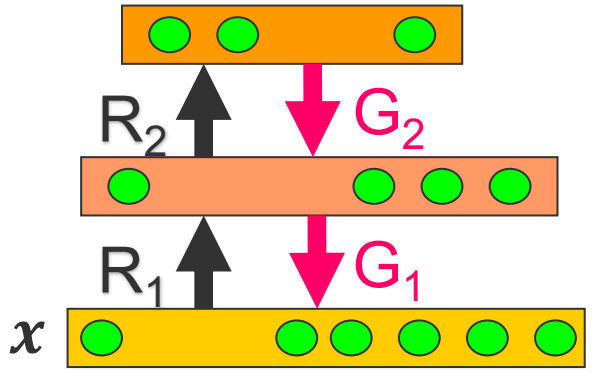
$$\nabla_\phi \mathbb{E}_{q_\phi}[\log p_\theta] = \int \nabla_\phi q_\phi \log p_\theta = \int q_\phi \log p_\theta \, \nabla_\phi \log q_\phi = \mathbb{E}_{q_\phi}[\log p_\theta \, \nabla_\phi \log q_\phi]$$

        - Monte Carlo estimation:

$$\nabla_\phi \mathbb{E}_{q_\phi}[\log p_\theta] \approx \mathbb{E}_{z_i \sim q_\phi}[\log p_\theta(x, z_i) \, \nabla_\phi q_\phi(z_i|x)]$$

        - The scale factor $\log p_\theta(x, z_i)$ can have arbitrary large magnitude

Wake Sleep Algorithm



- Free energy:

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$$

- WS works around the difficulties with the sleep phase approximation
- Minimize the following objective → *sleep* phase

$$F'(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(p_{\theta}(\boldsymbol{z}|\boldsymbol{x}) \,||\, q_{\phi}(\boldsymbol{z}|\boldsymbol{x}))$$

$$\max_{\boldsymbol{\phi}} \mathrm{E}_{p_{\theta}(\boldsymbol{z},\boldsymbol{x})} [\log q_{\phi}(\boldsymbol{z}|\boldsymbol{x})] \qquad \sout{\max_{\boldsymbol{\phi}} \mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} [\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})]}$$

  - "Dreaming" up samples from $p_{\theta}(\boldsymbol{x}|\boldsymbol{z})$ through top-down pass
  - Use the samples as targets for updating the inference model
- (Recent approaches other than sleep phase are developed to reduce the variance of gradient estimate: slides later)

[Figure courtesy: Maei's slides]

Wake Sleep Algorithm

Wake sleep

- Parametrized inference model $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$
- Wake phase:
  - minimize $KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$ wrt. θ
  - $\mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\nabla_\theta \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]$
- Sleep phase:
  - minimize $KL(p_\theta(\boldsymbol{z}|\boldsymbol{x}) \,||\, q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}))$ wrt. ϕ
  - $\mathrm{E}_{p_\theta(\boldsymbol{z},\boldsymbol{x})}\left[\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})\right]$
  - low variance
  - Learning with generated samples of $\boldsymbol{x}$
- Two objective, not guaranteed to converge

Variational EM

- Variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$
- Variational M step:
  - minimize $KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$ wrt. θ
  - $\mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\nabla_\theta \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]$
- Variational E step:
  - minimize $KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$ wrt. ϕ
  - $q^*_{\boldsymbol{\phi}} \propto \exp[\log p_\theta]$ if with closed-form
  - $\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}}[\log p_\theta(z,x)]$
    - need variance-reduce in practice
  - Learning with real data $\boldsymbol{x}$
- Single objective, guaranteed to converge

Variational Autoencoders (VAEs)

- [Kingma & Welling, 2014]
- Use variational inference with an inference model
  - Enjoy similar applicability with wake-sleep algorithm
- Generative model $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, and prior $p(\boldsymbol{z})$
  - Joint distribution $p_\theta(\boldsymbol{x},\boldsymbol{z}) = p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$
- Inference model $q_\phi(\boldsymbol{z}|\boldsymbol{x})$

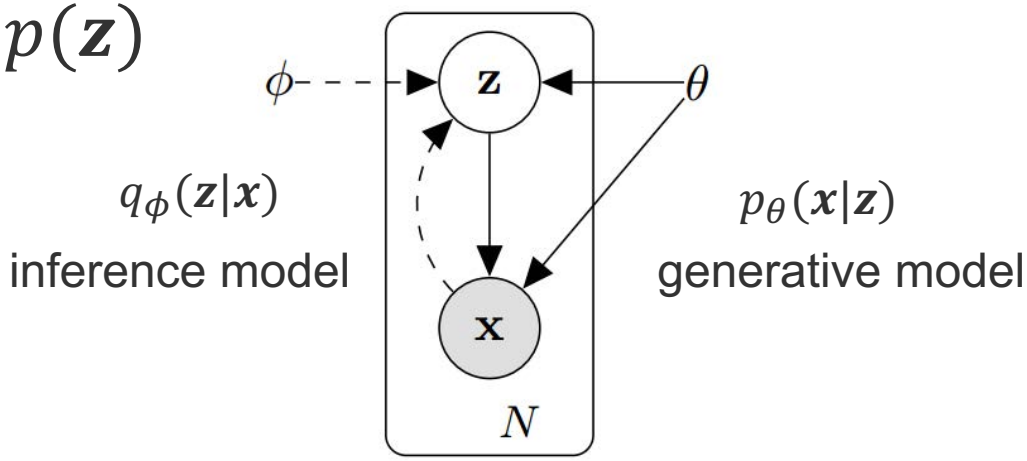


Figure courtesy: Kingma & Welling, 2014

Variational Autoencoders (VAEs)

- Variational lower bound

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = \mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\theta}(\boldsymbol{x}, \boldsymbol{z})] - \mathrm{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p(\boldsymbol{z}))$$

- Optimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$ wrt. θ of $p_{\theta}(\boldsymbol{x}|\boldsymbol{z})$
  - The same with the wake phase
- Optimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$ wrt. ϕ of $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$

$$\nabla_{\phi}\mathcal{L}(\theta, \phi; x) = \cdots + \nabla_{\phi}\mathbb{E}_{q_{\phi}(z|x)}[\log p_{\theta}(x|z)] + \cdots$$

  - Use *reparameterization trick* to reduce variance
  - Alternatives: use control variates as in reinforcement learning [Mnih & Gregor, 2014; Paisley et al., 2012]

Reparametrized gradient

- Optimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$ wrt. ϕ of $q_{\phi}(\boldsymbol{z}|\boldsymbol{x})$
  - Recap: gradient estimate with log-derivative trick:

$$\nabla_{\phi} \mathbb{E}_{q_{\phi}}[\log p_{\theta}(\boldsymbol{x}, \boldsymbol{z})] = \mathbb{E}_{q_{\phi}}[\log p_{\theta}(\boldsymbol{x}, \boldsymbol{z}) \nabla_{\phi} \log q_{\phi}]$$

    - High variance: $\nabla_{\phi} \mathbb{E}_{q_{\phi}}[\log p_{\theta}] \approx \mathbb{E}_{z_i \sim q_{\phi}}[\log p_{\theta}(x, z_i) \nabla_{\phi} q_{\phi}(z_i|x)]$
      - The scale factor $\log p_{\theta}(x, z_i)$ can have arbitrary large magnitude
  - gradient estimate with *reparameterization trick*

$$\boldsymbol{z} \sim q_{\phi}(\boldsymbol{z}|\boldsymbol{x}) \quad \Leftrightarrow \quad \mathbf{z} = \mathrm{g}_{\phi}(\boldsymbol{\epsilon}, \boldsymbol{x}), \qquad \boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$$

$$\nabla_{\phi} \mathrm{E}_{q_{\phi}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\theta}(\boldsymbol{x}, \boldsymbol{z})] = \mathrm{E}_{\boldsymbol{\epsilon} \sim p(\epsilon)}\left[\nabla_{\phi} \log p_{\theta}\left(\boldsymbol{x}, \boldsymbol{z}_{\phi}(\boldsymbol{\epsilon})\right)\right]$$

    - (Empirically) lower variance of the gradient estimate
    - E.g., $\boldsymbol{z} \sim N\left(\boldsymbol{\mu}(\boldsymbol{x}), \boldsymbol{L}(\boldsymbol{x})\boldsymbol{L}(\boldsymbol{x})^{T}\right) \Leftrightarrow \boldsymbol{\epsilon} \sim N(0,1), \ \boldsymbol{z} = \boldsymbol{\mu}(\boldsymbol{x}) + \boldsymbol{L}(\boldsymbol{x})\boldsymbol{\epsilon}$

VAEs: algorithm

Algorithm 1 Minibatch version of the Auto-Encoding VB (AEVB) algorithm. Either of the two SGVB estimators in section 2.3 can be used. We use settings $M = 100$ and $L = 1$ in experiments.

$\boldsymbol{\theta}, \boldsymbol{\phi} \leftarrow$ Initialize parameters
repeat
  $\mathbf{X}^M \leftarrow$ Random minibatch of M datapoints (drawn from full dataset)
  $\boldsymbol{\epsilon} \leftarrow$ Random samples from noise distribution $p(\boldsymbol{\epsilon})$
  $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \widetilde{\mathcal{L}}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{X}^M, \boldsymbol{\epsilon})$ (Gradients of minibatch estimator (8))
  $\boldsymbol{\theta}, \boldsymbol{\phi} \leftarrow$ Update parameters using gradients $\mathbf{g}$ (e.g. SGD or Adagrad [DHS10])
until convergence of parameters $(\boldsymbol{\theta}, \boldsymbol{\phi})$
return $\boldsymbol{\theta}, \boldsymbol{\phi}$

[Kingma & Welling, 2014]

VAEs: example results

- VAEs tend to generate blurred images due to the mode covering behavior (more later)

Celebrity faces [Radford 2015]

- Latent code interpolation and sentences generation from VAEs [Bowman et al., 2015].

“ i want to talk to you . ”
“i want to be with you . ”
“i do n’t want to be with you . ”
i do n’t want to be with you .
she did n’t want to be with him .

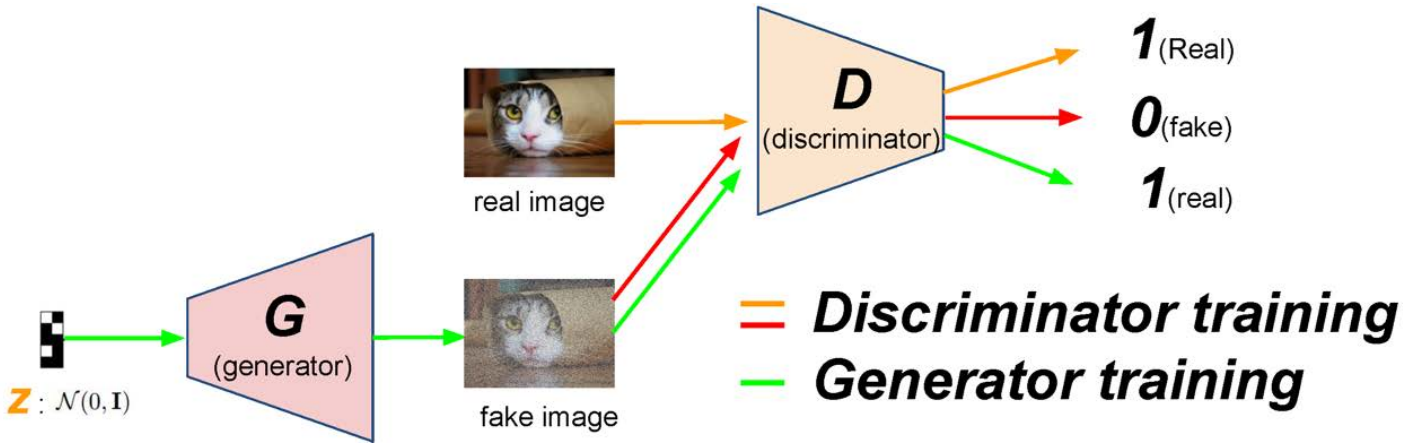
Generative Adversarial Nets (GANs)

- [Goodfellow et al., 2014]
- Generative model $\boldsymbol{x} = G_\theta(\boldsymbol{z}), \quad \boldsymbol{z} \sim p(\boldsymbol{z})$
  - Map noise variable $\boldsymbol{z}$ to data space $\boldsymbol{x}$
  - Define an implicit distribution over $\boldsymbol{x}$: $p_{g_\theta}(\boldsymbol{x})$
    - a stochastic process to simulate data $\boldsymbol{x}$
    - Intractable to evaluate likelihood
- Discriminator $D_\phi(\boldsymbol{x})$
  - Output the probability that $\boldsymbol{x}$ came from the data rather than the generator
- No explicit inference model
- No obvious connection to previous models with inference networks like VAEs
  - We will build formal connections between GANs and VAEs later

Generative Adversarial Nets (GANs)

- Learning
  - A minimax game between the generator and the discriminator
  - Train D to maximize the probability of assigning the correct label to both training examples and generated samples
  - Train G to fool the discriminator

$$\max_D \mathcal{L}_D = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}\left[\log D(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z})}\left[\log(1-D(\boldsymbol{x}))\right]$$
$$\min_G \mathcal{L}_G = \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z})}\left[\log(1-D(\boldsymbol{x}))\right].$$

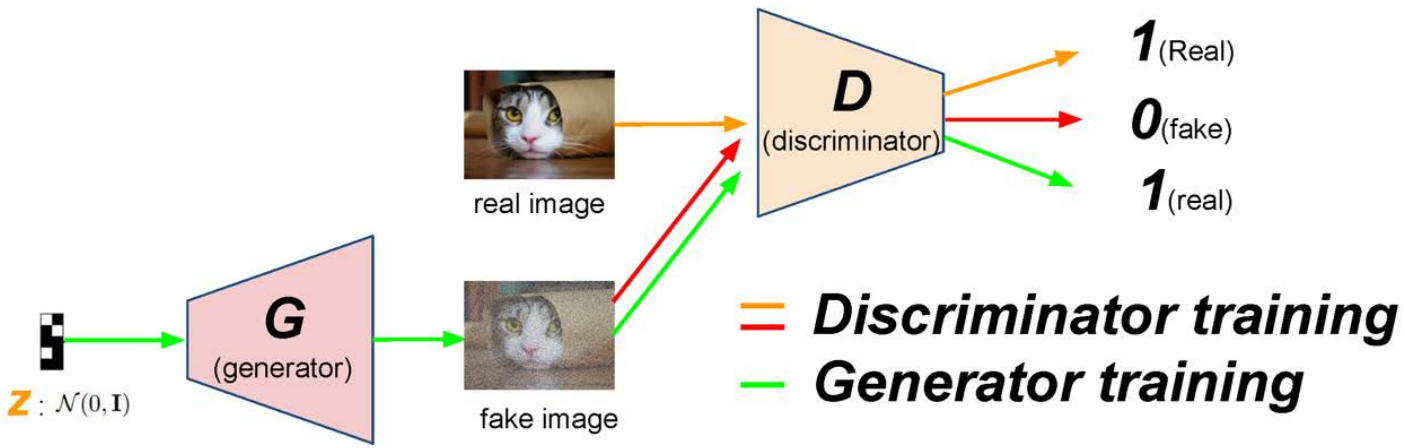


[Figure courtesy: Kim's slides]

Generative Adversarial Nets (GANs)

- Learning
  - Train G to fool the discriminator
    - The original loss suffers from vanishing gradients when D is too strong
    - Instead use the following in practice

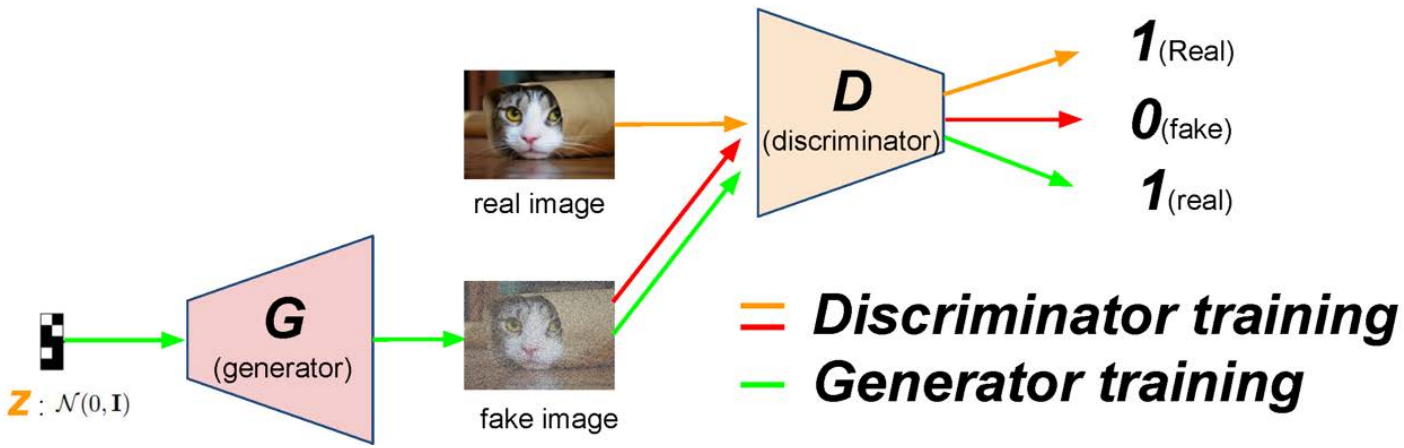
$$\max_G \mathcal{L}_G = \mathbb{E}_{\boldsymbol{x} \sim G(\boldsymbol{z}), \boldsymbol{z} \sim p(\boldsymbol{z})} [\log D(\boldsymbol{x})]$$



[Figure courtesy: Kim's slides]

Generative Adversarial Nets (GANs)

- Learning
  - Aim to achieve equilibrium of the game
  - Optimal state:
    - $p_g(\boldsymbol{x}) = p_{data}(x)$
    - $D(\boldsymbol{x}) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)} = \frac{1}{2}$



[Figure courtesy: Kim's slides]

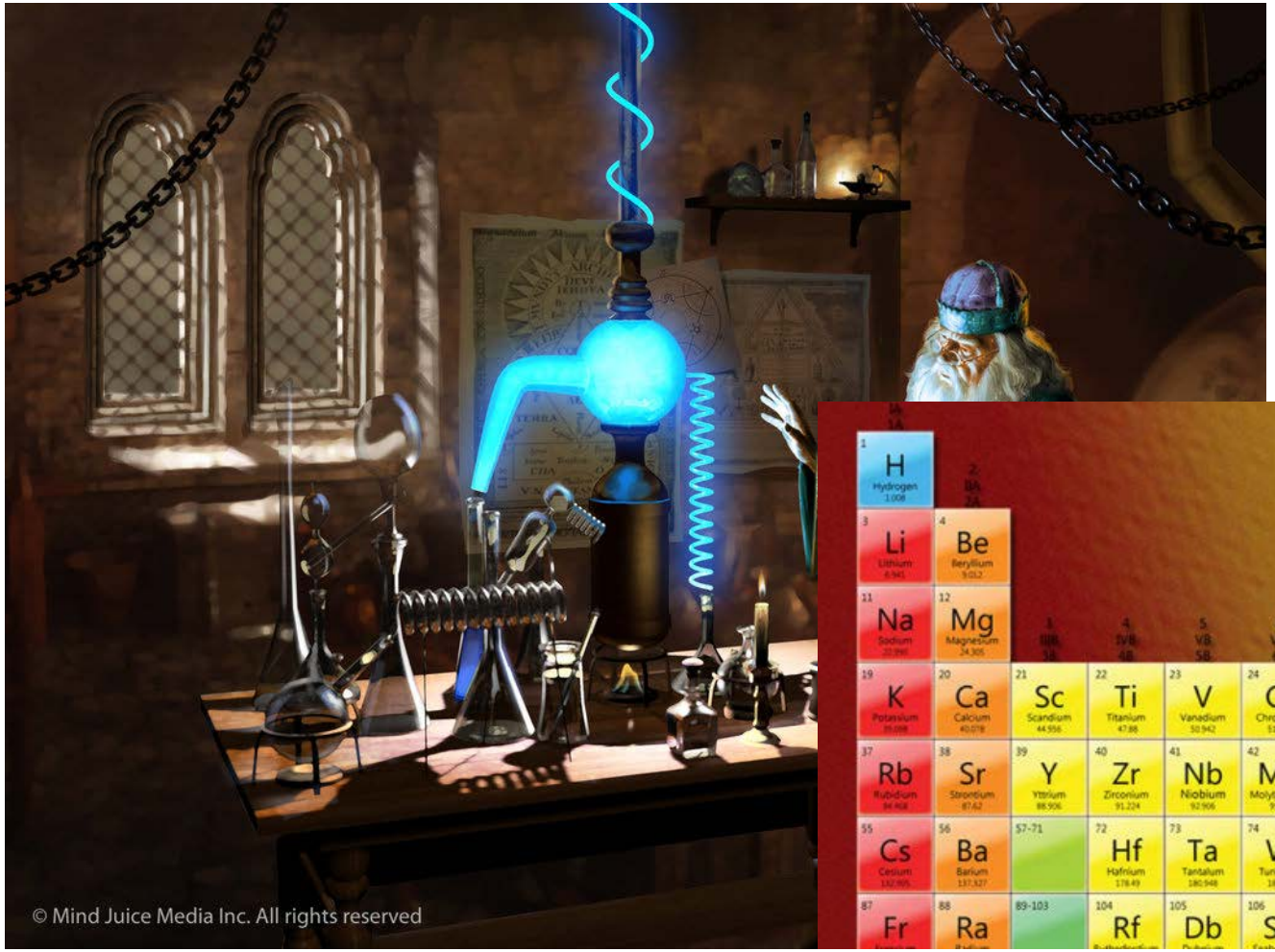
GANs: example results

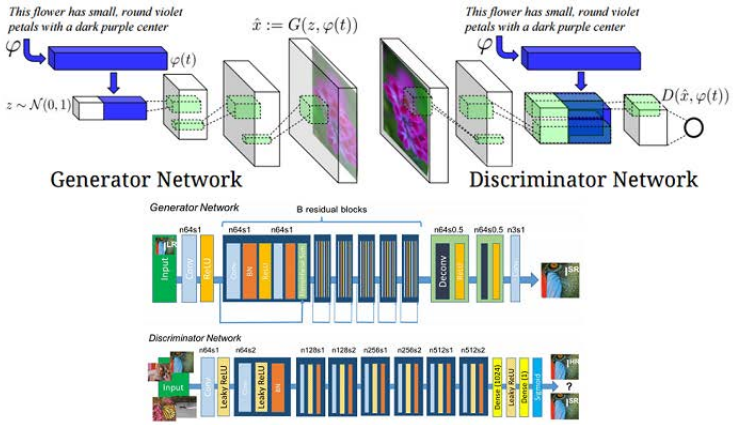
Generated bedrooms [Radford et al., 2016]

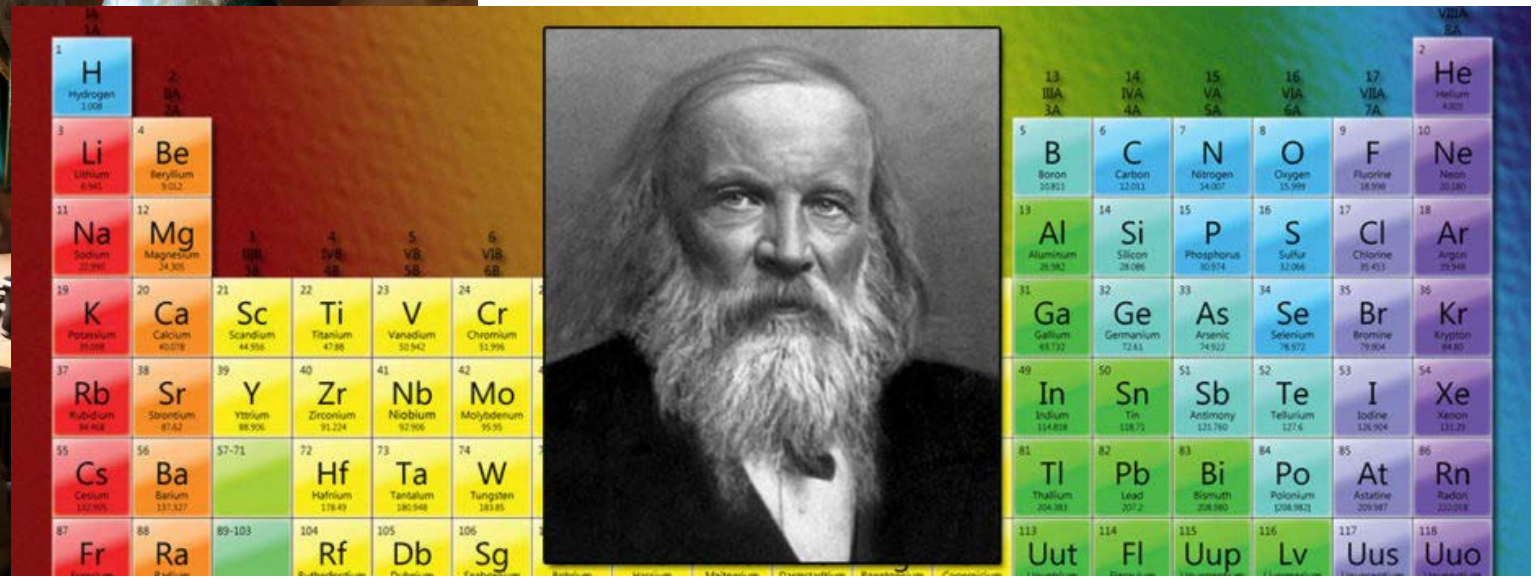
The Zoo of DGMs

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
  - Adversarial autoencoder [Makhzani et al., 2015]
  - Importance weighted autoencoder [Burda et al., 2015]
  - Implicit variational autoencoder [Mescheder., 2017]
- Generative adversarial networks (GANs) [Goodfellos et al., 2014]
  - InfoGAN [Chen et al., 2016]
  - CycleGAN [Zhu et al., 2017]
  - Wasserstein GAN [Arjovsky et al., 2017]
- Autoregressive neural networks
  - PixelRNN / PixelCNN [Oord et al., 2016]
  - RNN (e.g., for language modeling)
- Generative moment matching networks (GMMNs) [Li et al., 2015; Dziugaite et al., 2015]
- Retricted Boltzmann Machines (RBMs) [Smolensky, 1986]

Alchemy Vs Modern Chemistry







Outline

- Overview of advances in deep generative models
- Theoretical backgrounds of deep generative models
  - Wake sleep algorithm
  - Variational autoencoders
  - Generative adversarial networks
- A unified view of deep generative models
  - new formulations of deep generative models
  - Symmetric modeling of latent and visible variables

Z Hu, Z YANG, R Salakhutdinov, E Xing,
"**On Unifying Deep Generative Models**", arxiv 1706.00550

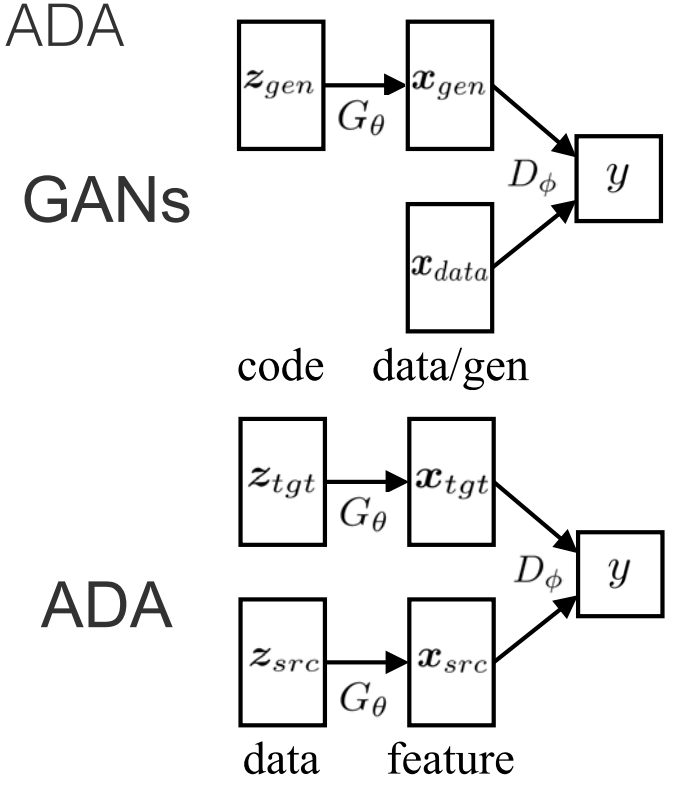
A unified view of deep generative models

- Literatures have viewed these DGM approaches as distinct model training paradigms
  - GANs: achieve an equilibrium between generator and discriminator
  - VAEs: maximize lower bound of the data likelihood
- Let's study a new formulation for DGMs
  - Connects GANs, VAEs, and other variants, under a unified view
  - Links them back to inference and learning of Graphical Models, and the wake-sleep heuristic that approximates this
  - Provides a tool to analyze many GAN-/VAE-based algorithms
  - Encourages mutual exchange of ideas from each individual class of models

Adversarial domain adaptation (ADA)

- Let's start from ADA
  - The application of adversarial approach on domain adaptation
  - We then show GANs can be seen as a special case of ADA
  - Correspondence of elements:

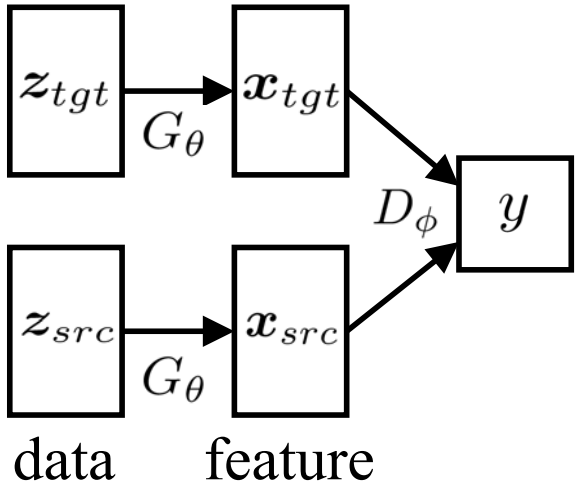
Elements	GANs	ADA
$\boldsymbol{x}$	data/generation	features
$\boldsymbol{z}$	code vector	Data from source/target domains
y	Real/fake indicator	Source/target domain indicator



Adversarial domain adaptation (ADA)

Is this "inference" or "generation"?

- Data z from two domains indicated by $y \in \{0,1\}$
  - Source domain $(y = 1)$
  - Target domain $(y = 0)$
- ADA transfers prediction knowledge learned from the source domain to the target domain
  - Learn a feature extractor G_θ: $\boldsymbol{x} = G_\theta(\boldsymbol{z})$
  - Wants $\boldsymbol{x}$ to be indistinguishable by a domain discriminator: $D_\phi(\boldsymbol{x})$
- Application in classification
  - E.g., we have labels of the source domain data
  - Train classifier over $\boldsymbol{x}$ of source domain data to predict the labels
  - $\boldsymbol{x}$ is domain invariant ⇒ $\boldsymbol{x}$ is predictive for target domain data



ADA: conventional formulation

- Train $D_{\boldsymbol{\phi}}$ to distinguish between domains

$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi} = \mathbb{E}_{\boldsymbol{x}=G_{\theta}(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=1)} \left[\log D_{\phi}(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}=G_{\theta}(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=0)} \left[\log(1 - D_{\phi}(\boldsymbol{x}))\right]$$

- Train $G_{\boldsymbol{\theta}}$ to fool $D_{\boldsymbol{\phi}}$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\theta} = \mathbb{E}_{\boldsymbol{x}=G_{\theta}(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=1)} \left[\log(1 - D_{\phi}(\boldsymbol{x}))\right] + \mathbb{E}_{\boldsymbol{x}=G_{\theta}(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=0)} \left[\log D_{\phi}(\boldsymbol{x})\right]$$

ADA: new formulation

- To reveal the connections to conventional variational approaches, let's rewrite the objectives in a format that resembles variational EM
- Implicit distribution over $\boldsymbol{x} \sim p_\theta(\boldsymbol{x}|y)$

$$\boldsymbol{x} = G_\theta(\boldsymbol{z}),\ \boldsymbol{z} \sim p(\boldsymbol{z}|y)$$

- Discriminator distribution $q_{\boldsymbol{\phi}}(y|\boldsymbol{x})$

$$q^r_{\boldsymbol{\phi}}(y|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(1-y|\boldsymbol{x})$$

- Rewrite the objective in the new form (up to constant scale factor)

$$\max_{\boldsymbol{\phi}} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)}\left[\log q_\phi(y|\boldsymbol{x})\right]$$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_\theta = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)}\left[\log q^r_\phi(y|\boldsymbol{x})\right]$$

  - $\boldsymbol{z}$ is encapsulated in the implicit distribution $p_\theta(\boldsymbol{x}|y)$

$$\begin{aligned}\max_{\boldsymbol{\phi}} \mathcal{L}_\phi &= \mathbb{E}_{p_\theta(\boldsymbol{x}|y=0)p(y=0)}\left[\log q_\phi(y=0|\boldsymbol{x})\right] + \mathbb{E}_{p_\theta(\boldsymbol{x}|y=1)p(y=1)}\left[\log q_\phi(y=1|\boldsymbol{x})\right] \\ &= \frac{1}{2}\mathbb{E}_{\boldsymbol{x}=G_\theta(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=0)}\left[\log(1-D_\phi(\boldsymbol{x}))\right] + \frac{1}{2}\mathbb{E}_{\boldsymbol{x}=G_\theta(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=1)}\left[\log D_\phi(\boldsymbol{x})\right]\end{aligned}$$

    - (Ignore the constant scale factor 1/2)

ADA: new formulation

- New formulation

$$\max_{\boldsymbol{\phi}} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q_\phi(y|\boldsymbol{x})\right]$$
$$\max_{\boldsymbol{\theta}} \mathcal{L}_\theta = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q_\phi^r(y|\boldsymbol{x})\right]$$

  - The only difference between optimizing θ and ϕ: q vs. q^r
  - This is where the adversarial mechanism comes about

ADA vs. Variational EM

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\phi}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\phi}(z|x)||p(z)\right)$$

$$\max_{\theta} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\phi}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\phi}(z|x)||p(z)\right)$$

  - Single objective for both θ and ϕ
  - Extra prior regularization by $p(z)$

ADA

- Objectives

$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q_{\phi}(y|\boldsymbol{x})\right]$$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q^{r}_{\phi}(y|\boldsymbol{x})\right]$$

  - Two objectives
  - Have global optimal state in the game theoretic view

ADA vs. Variational EM

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\boldsymbol{\phi}}(z|x)||p(z)\right)$$

$$\max_{\theta} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\boldsymbol{\phi}}(z|x)||p(z)\right)$$

  - Single objective for both θ and ϕ
  - Extra prior regularization by $p(z)$

- The reconstruction term: maximize the conditional log-likelihood of x with the generative distribution $p_{\theta}(x|z)$ conditioning on the latent code z inferred by $q_{\boldsymbol{\phi}}(z|x)$

- $p_{\theta}(x|z)$ is the generative model
- $q_{\boldsymbol{\phi}}(z|x)$ is the inference model

ADA

- Objectives

$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q_{\phi}(y|\boldsymbol{x})\right]$$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q^{r}_{\phi}(y|\boldsymbol{x})\right]$$

  - Two objectives
  - Have global optimal state in the game theoretic view

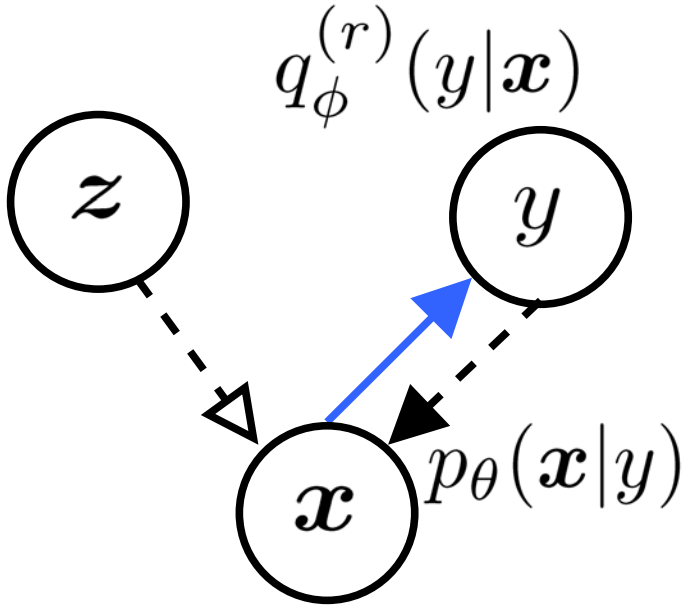
- The objectives: maximize the conditional log-likelihood of y (or $1-y$) with the distribution $q_{\phi}(y|x)$ conditioning on latent feature x inferred by $p_{\theta}(x|y)$

- Interpret $q_{\phi}(y|x)$ as the generative model
- Interpret $p_{\theta}(x|y)$ as the inference model

ADA: graphical model

Define:

- Solid-line arrows ($x \to y$):
  - generative process
- Dashed-line arrows ($\mathrm{y}, \mathrm{z} \dashrightarrow x$):
  - inference
- Hollow arrows ($\mathrm{z} \twoheadrightarrow x$):
  - deterministic transformation
  - leading to implicit distributions
- Blue arrows ($x \to y$):
  - adversarial mechanism
  - involves both $q_{\phi}(y|\boldsymbol{x})$ and $q_{\phi}^{r}(y|\boldsymbol{x})$

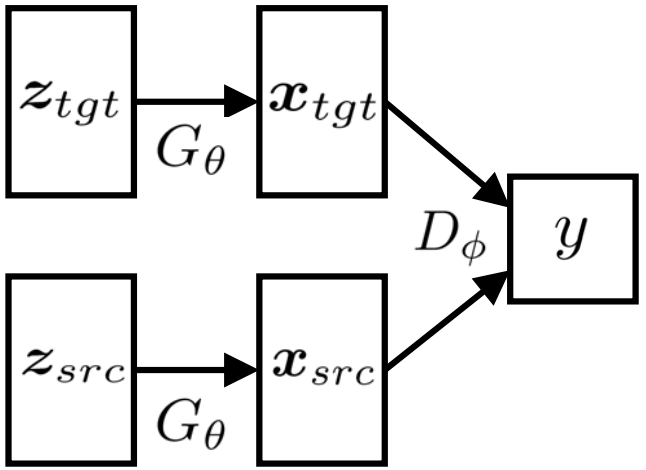


$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)} \left[\log q_{\phi}(y|\boldsymbol{x})\right]$$

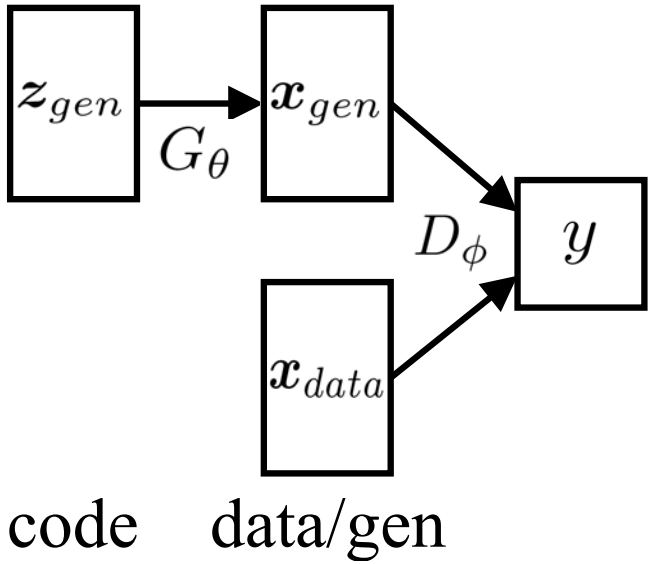
$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)} \left[\log q_{\phi}^{r}(y|\boldsymbol{x})\right]$$

GANs: a variant of ADA

- Transfer the properties of source domain to target domain
  - Source domain: e.g. real image, $y = 1$
  - Target domain: e.g. generated image, $y = 0$



ADA



GANs

GANs: a variant of ADA

- Implicit distribution over $\boldsymbol{x} \sim p_\theta(\boldsymbol{x}|y)$

$$p_\theta(\boldsymbol{x}|y) = \begin{cases} p_{g_\theta}(\boldsymbol{x}) & y = 0 \\ p_{data}(\boldsymbol{x}) & y = 1. \end{cases}$$

(distribution of generated images)

(distribution of real images)

- $\boldsymbol{x} \sim p_{g_\theta}(\boldsymbol{x}) \Leftrightarrow \boldsymbol{x} = G_\theta(\boldsymbol{z}),\ \boldsymbol{z} \sim p(\boldsymbol{z}|y = 0)$
- $\boldsymbol{x} \sim p_{data}(\boldsymbol{x})$
  - the code space of $\boldsymbol{z}$ is degenerated
  - sample directly from data

GANs: new formulation

- Again, rewrite GAN objectives in the ”variational-EM” format
- Recap: conventional formulation:

$$\max_{\boldsymbol{\phi}} \mathcal{L}_\phi = \mathbb{E}_{\boldsymbol{x}=G_\theta(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=0)}\left[\log(1-D_\phi(\boldsymbol{x}))\right] + \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}\left[\log D_\phi(\boldsymbol{x})\right]$$

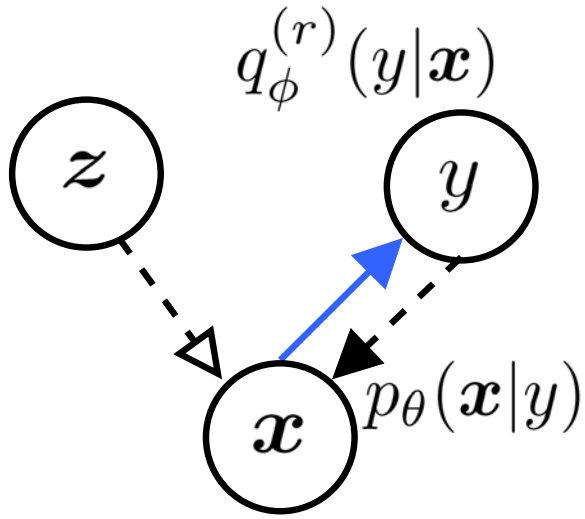
$$\max_{\boldsymbol{\theta}} \mathcal{L}_\theta = \mathbb{E}_{\boldsymbol{x}=G_\theta(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=0)}\left[\log D_\phi(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}\left[\log(1-D_\phi(\boldsymbol{x}))\right]$$

$$= \mathbb{E}_{\boldsymbol{x}=G_\theta(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z}|y=0)}\left[\log D_\phi(\boldsymbol{x})\right]$$

- Rewrite in the new form

$$\max_{\boldsymbol{\phi}} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)}\left[\log q_\phi(y|\boldsymbol{x})\right]$$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_\theta = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)}\left[\log q^r_\phi(y|\boldsymbol{x})\right]$$



- Exact the same with ADA !
- The same correspondence to variational EM !

GANs vs. Variational EM

Variational EM

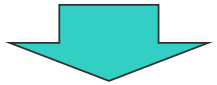
- Objectives

$$\max_{\phi} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\boldsymbol{\phi}}(z|x)||p(z)\right)$$

$$\max_{\theta} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\boldsymbol{\phi}}(z|x)||p(z)\right)$$

  - Single objective for both θ and ϕ
  - Extra prior regularization by $p(z)$

- The reconstruction term: maximize the conditional log-likelihood of x with the generative distribution $p_{\theta}(x|z)$ conditioning on the latent code z inferred by $q_{\boldsymbol{\phi}}(z|x)$



- $p_{\theta}(x|z)$ is the generative model
- $q_{\boldsymbol{\phi}}(z|x)$ is the inference model

GAN

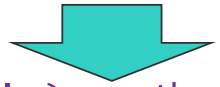
- Objectives

$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q_{\phi}(y|\boldsymbol{x})\right]$$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q^{r}_{\phi}(y|\boldsymbol{x})\right]$$

  - Two objectives
  - Have global optimal state in the game theoretic view

- The objectives: maximize the conditional log-likelihood of y (or $1-y$) with the distribution $q_{\boldsymbol{\phi}}(y|x)$ conditioning on data/generation x inferred by $p_{\boldsymbol{\theta}}(x|y)$



- Interpret $q_{\boldsymbol{\phi}}(y|x)$ as the generative model
- Interpret $p_{\boldsymbol{\theta}}(x|y)$ as the inference model

GANs vs. Variational EM

- Interpret x as latent variables
- Interpret generation of x as performing inference over latent

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\phi}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\phi}(z|x)||p(z)\right)$$

$$\max_{\theta} \mathcal{L}_{\boldsymbol{\phi},\boldsymbol{\theta}} = \mathbb{E}_{q_{\phi}(z|x)}[\log p_{\theta}(x|z)] + KL\left(q_{\phi}(z|x)||p(z)\right)$$

  - Single objective for both θ and ϕ
  - Extra prior regularization by $p(z)$

- The reconstruction term: maximize the conditional log-likelihood of x with the generative distribution $p_{\theta}(x|z)$ conditioning on the latent code z inferred by $q_{\phi}(z|x)$

- $p_{\theta}(x|z)$ is the generative model
- $q_{\phi}(z|x)$ is the inference model

GAN

- Objectives

$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q_{\phi}(y|\boldsymbol{x})\right]$$

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\boldsymbol{x}|y)p(y)}\left[\log q^{r}_{\phi}(y|\boldsymbol{x})\right]$$

  - Two objectives
  - Have global optimal state in the game theoretic view

- The objectives: maximize the conditional log-likelihood of y (or $1-y$) with the distribution $q_{\phi}(y|x)$ conditioning on data/generation x inferred by $p_{\theta}(x|y)$

- Interpret $q_{\phi}(y|x)$ as the generative model
- Interpret $p_{\theta}(x|y)$ as the inference model

GANs: minimizing KLD

- As in Variational EM, we can further rewrite in the form of minimizing KLD to reveal more insights into the optimization problem
- For each optimization step of $p_\theta(\boldsymbol{x}|y)$ at point $(\theta = \theta_0, \phi = \phi_0)$, let
  - $p(y)$: a uniform distribution
  - $p_{\theta=\theta_0}(\boldsymbol{x}) = \mathrm{E}_{p(y)}\left[p_{\theta=\theta_0}(\boldsymbol{x}|y)\right]$: prior distribution over $\boldsymbol{x}$
  - $q^r(\boldsymbol{x}|y) \propto q^r_{\phi=\phi_0}(y|\boldsymbol{x})p_{\theta=\theta_0}(\boldsymbol{x})$: posterior distribution over $\boldsymbol{x}$
- ***Lemma 1***: The updates of $\boldsymbol{\theta}$ at $\boldsymbol{\theta_0}$ have

$$\nabla_\theta \left[-\mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q^r_{\phi=\phi_0}(y|\boldsymbol{x})\right] \right]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} =$$

$$\nabla_\theta \left[\mathbb{E}_{p(y)} \left[KL\left(p_\theta(\boldsymbol{x}|y)\|q^r(\boldsymbol{x}|y)\right)\right] - JSD\left(p_\theta(\boldsymbol{x}|y=0)\|p_\theta(\boldsymbol{x}|y=1)\right) \right]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$$

  - KL: KL divergence
  - JSD: Jensen-shannon divergence

GANs: minimizing KLD

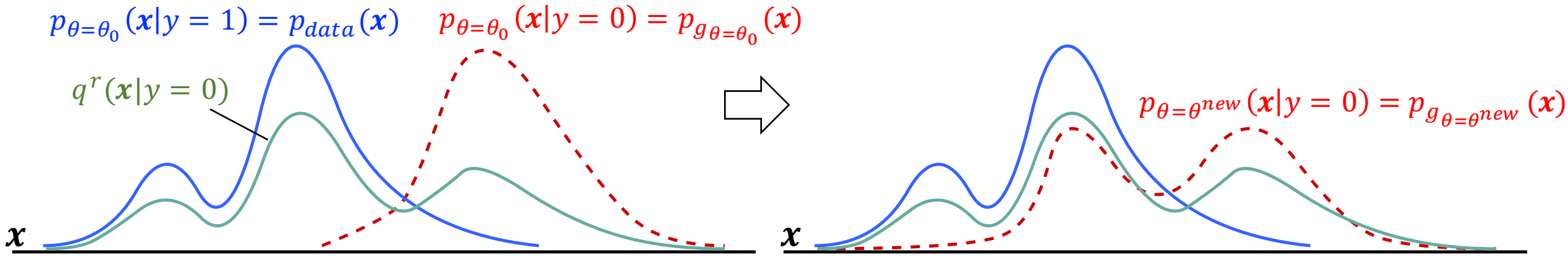
- *Lemma 1*: The updates of $\boldsymbol{\theta}$ at $\boldsymbol{\theta}_0$ have

$$\nabla_\theta \Big[-\mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q^r_{\phi=\phi_0}(y|\boldsymbol{x}) \right] \Big] \Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} =$$

$$\nabla_\theta \Big[\mathbb{E}_{p(y)} \left[\mathrm{KL}\left(p_\theta(\boldsymbol{x}|y) \| q^r(\boldsymbol{x}|y)\right) \right] - \mathrm{JSD}\left(p_\theta(\boldsymbol{x}|y=0) \| p_\theta(\boldsymbol{x}|y=1)\right) \Big] \Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$$

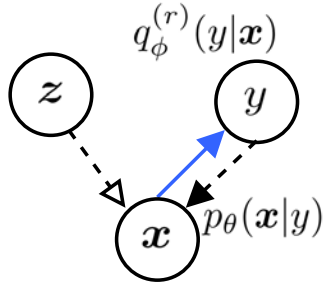
- Connection to variational inference
  - See $\boldsymbol{x}$ as latent variables, y as visible
  - $p_{\theta=\theta_0}(\boldsymbol{x})$: prior distribution
  - $q^r(\boldsymbol{x}|y) \propto q^r_{\phi=\phi_0}(y|\boldsymbol{x}) p_{\theta=\theta_0}(\boldsymbol{x})$: posterior distribution
  - $p_\theta(\boldsymbol{x}|y)$: variational distribution
    - Amortized inference: updates model parameter $\boldsymbol{\theta}$
- Suggests relations to VAEs, as we will explore shortly

GANs: minimizing KLD

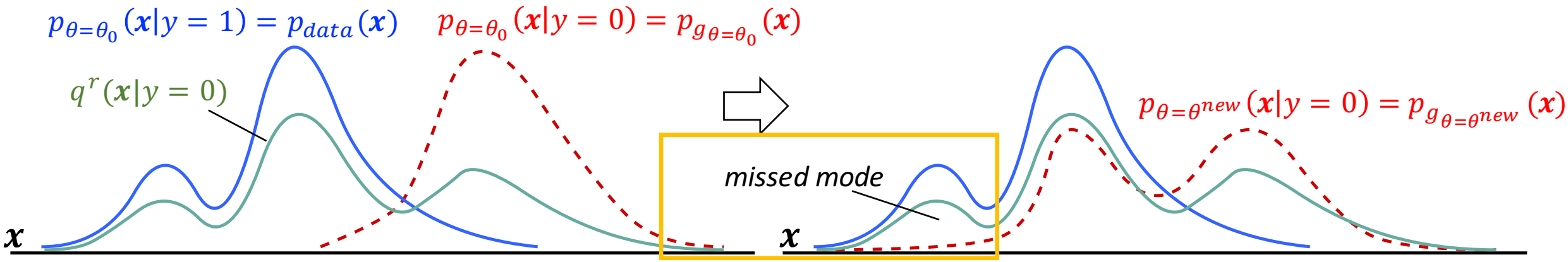


- Minimizing the KLD drives $p_{g_\theta}(\boldsymbol{x})$ to $p_{data}(\boldsymbol{x})$
  - By definition: $p_{\theta=\theta_0}(\boldsymbol{x}) = \mathrm{E}_{p(y)}\left[p_{\theta=\theta_0}(\boldsymbol{x}|y)\right] = \left(p_{g_{\theta=\theta_0}}(\boldsymbol{x}) + p_{data}(\boldsymbol{x})\right)/2$
  - $\mathrm{KL}\left(p_\theta(x|y=1)||q^r(x|y=1)\right) = \mathrm{KL}\left(p_{data}(x)||q^r(x|y=1)\right)$: constant, no free parameters
  - $\mathrm{KL}\left(p_\theta(x|y=0)||q^r(x|y=0)\right) = \mathrm{KL}\left(p_{g_\theta}(x)||q^r(x|y=0)\right)$: parameter θ to optimize
    - $q^r(\boldsymbol{x}|y=0) \propto q^r_{\phi=\phi_0}(y=0|\boldsymbol{x})p_{\theta=\theta_0}(\boldsymbol{x})$
      - seen as a mixture of $p_{g_{\theta=\theta_0}}(\boldsymbol{x})$ and $p_{data}(\boldsymbol{x})$
      - mixing weights induced from $q^r_{\phi=\phi_0}(y=0|\boldsymbol{x})$
    - Drives $p_{g_\theta}(\boldsymbol{x}|y)$ to mixture of $p_{g_{\theta=\theta_0}}(\boldsymbol{x})$ and $p_{data}(\boldsymbol{x})$

      ⇒ Drives $p_{g_\theta}(\boldsymbol{x})$ to $p_{data}(\boldsymbol{x})$



GANs: minimizing KLD



- Missing mode phenomena of GANs
  - Asymmetry of KLD
    - Concentrates $p_\theta(\boldsymbol{x}|y=0)$ to large modes of $q^r(\boldsymbol{x}|y)$

      $\Rightarrow p_{g_\theta}(\boldsymbol{x})$ misses modes of $p_{data}(\boldsymbol{x})$
  - Symmetry of JSD
    - Does not affect the behavior of mode missing

$$\mathrm{KL}\left(p_{g_\theta}(x)||q^r(x|y=0)\right)$$
$$=\int p_{g_\theta}(x)\log\frac{p_{g_\theta}(x)}{q^r(x|y=0)}dx$$

- Large positive contribution to the KLD in the regions of x space where $q^r(x|y=0)$ is small, unless $p_{g_\theta}(x)$ is also small
- $\Rightarrow p_{g_\theta}(x)$ tends to avoid regions where $q^r(x|y=0)$ is small

GANs: minimizing KLD

- *Lemma 1*: The updates of $\boldsymbol{\theta}$ at $\boldsymbol{\theta}_0$ have

$$\nabla_\theta \Big[-\mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q^r_{\phi_0}(y|\boldsymbol{x})\right] \Big]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} =$$

$$\nabla_\theta \Big[\mathbb{E}_{p(y)} \left[KL\left(p_\theta(\boldsymbol{x}|y)\|q^r(\boldsymbol{x}|y)\right)\right] - JSD\left(p_\theta(\boldsymbol{x}|y=0)\|p_\theta(\boldsymbol{x}|y=1)\right) \Big]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$$

- No assumption on optimal discriminator $q^r_{\boldsymbol{\phi}_0}(y|\boldsymbol{x})$
  - Previous results usually rely on (near) optimal discriminator
    - $q^*(y=1|\boldsymbol{x}) = p_{data}(\boldsymbol{x})/(p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x}))$
  - Optimality assumption is impractical: limited expressiveness of $D_{\boldsymbol{\phi}}$ [Arora et al 2017]
  - Our result is a generalization of the previous theorem [Arjovsky & Bottou 2017]
    - Plug the optimal discriminator into the above equation, we recover the theorem

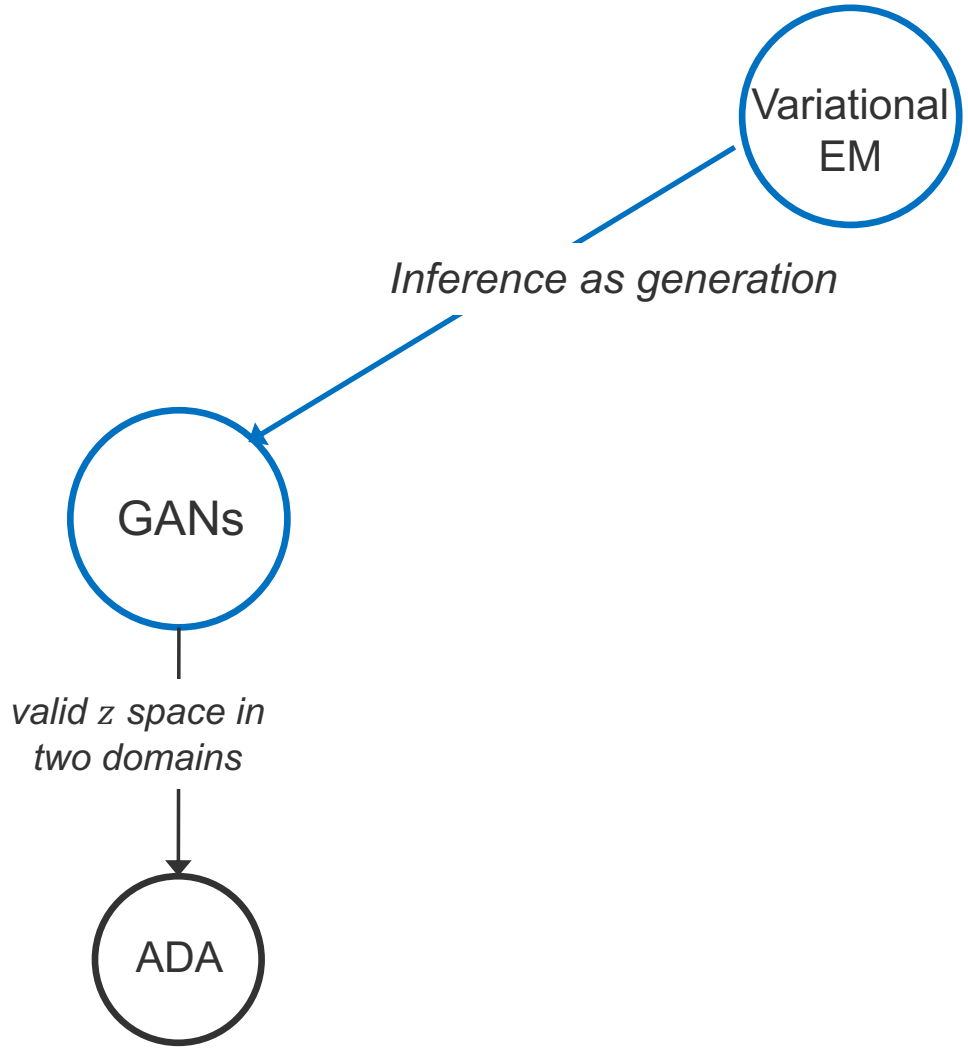
$$\nabla_\theta \Big[-\mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q^r_{\phi_0}(y|\boldsymbol{x})\right] \Big]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} = \nabla_\theta \left[\frac{1}{2}\mathrm{KL}\left(p_{g_\theta}\|p_{data}\right) - \mathrm{JSD}\left(p_{g_\theta}\|p_{data}\right)\right]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$$

    - Give insights on the generator training when discriminator is optimal

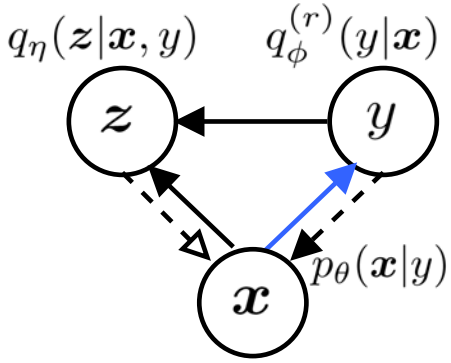
GANs: minimizing KLD

In summary:

- Reveal connection to variational inference
  - Build connections to VAEs (slides soon)
  - Inspire new model variants based on the connections
- Offer insights into the generator training
  - Formal explanation of the missing mode behavior of GANs
  - Still hold when the discriminator does not achieve its optimum at each iteration


Variational EM
Inference as generation
GANs
valid z space in two domains
ADA

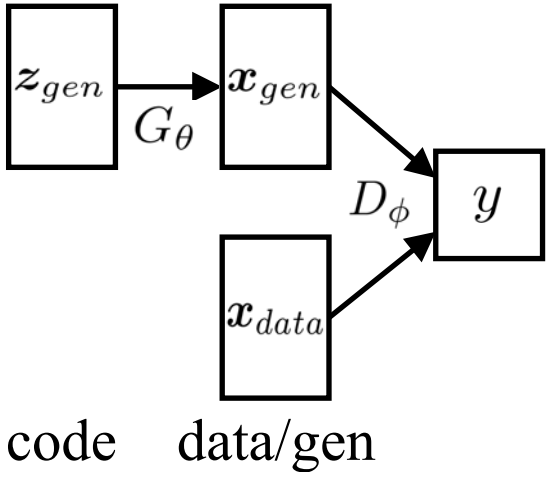
Variant of GAN: InfoGAN



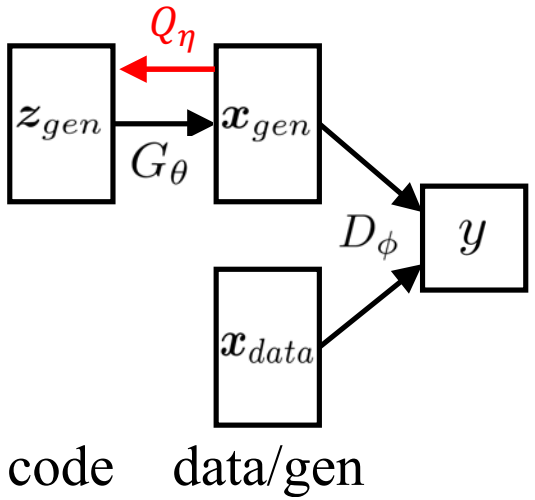
- GANs don't offer the functionality of inferring code $\boldsymbol{z}$ given data
- InfoGAN [Chen et al., 2016]
  - Introduce inference model $Q_\eta(\boldsymbol{z}|\boldsymbol{x})$ with parameters $\boldsymbol{\eta}$
  - Augment the objectives of GANs by additionally inferring $\boldsymbol{z}$

$$\max_D \mathcal{L}_D = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1-D(\boldsymbol{x}))],$$
$$\max_{G,Q} \mathcal{L}_{G,Q} = \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z}),\boldsymbol{z}\sim p(\boldsymbol{z})}[\log D(\boldsymbol{x}) + \log Q(\boldsymbol{z}|\boldsymbol{x})].$$

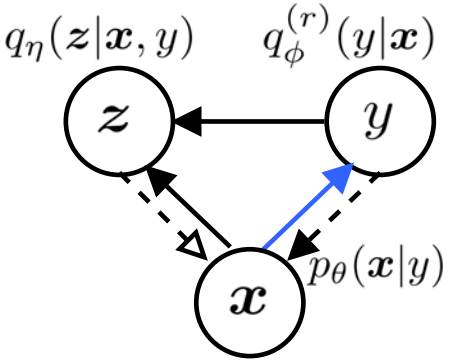
GANs



InfoGAN



InfoGAN: new formulation

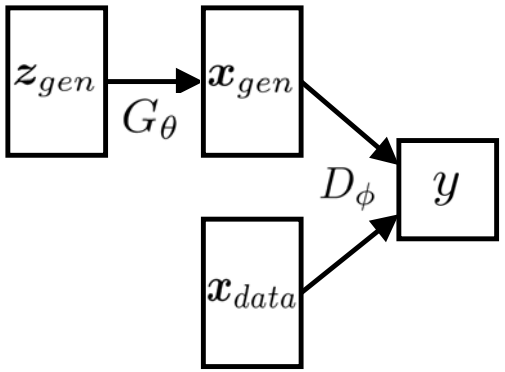


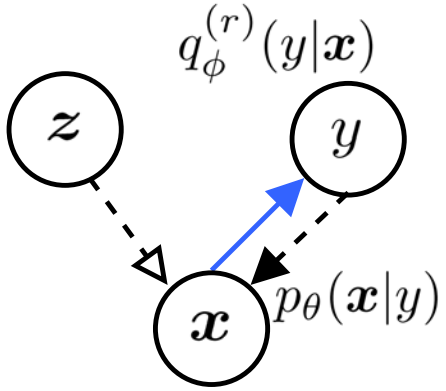
- Defines conditional $q_\eta(\boldsymbol{z}|\boldsymbol{x},y)$
  - $q_\eta(\boldsymbol{z}|\boldsymbol{x},y=1)$ is fixed without free parameters to learn
    - As GANs assume the code space of real data is degenerated
  - Parameters η are only associated with $q_\eta(\boldsymbol{z}|\boldsymbol{x},y=0)$
- Rewrite in the new form:

$$\max_{\boldsymbol{\phi}} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)}\left[\log q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q_\phi(y|\boldsymbol{x})\right]$$

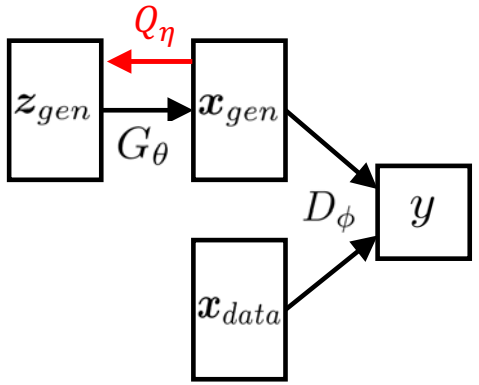
$$\max_{\boldsymbol{\theta},\boldsymbol{\eta}} \mathcal{L}_{\theta,\eta} = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)}\left[\log q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q_\phi^r(y|\boldsymbol{x})\right]$$

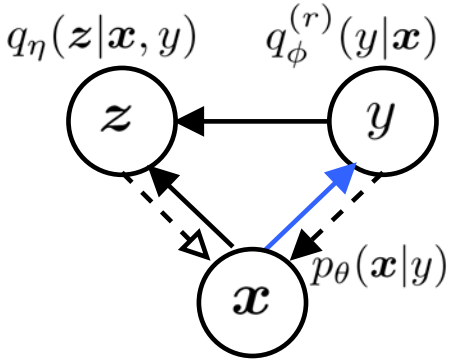
GANs vs InfoGAN





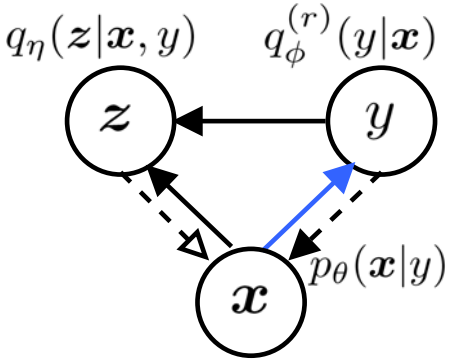
$$\max_{\phi} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(x|y)p(y)} [\log q_\phi(y|x)]$$
$$\max_{\theta} \mathcal{L}_\theta = \mathbb{E}_{p_\theta(x|y)p(y)} [\log q_\phi^r(y|x)]$$





$$\max_{\phi} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(x|y)p(y)} [\log q_\eta(z|x,y) q_\phi(y|x)]$$
$$\max_{\theta,\eta} \mathcal{L}_{\theta,\eta} = \mathbb{E}_{p_\theta(x|y)p(y)} [\log q_\eta(z|x,y) q_\phi^r(y|x)]$$

InfoGAN: new formulation

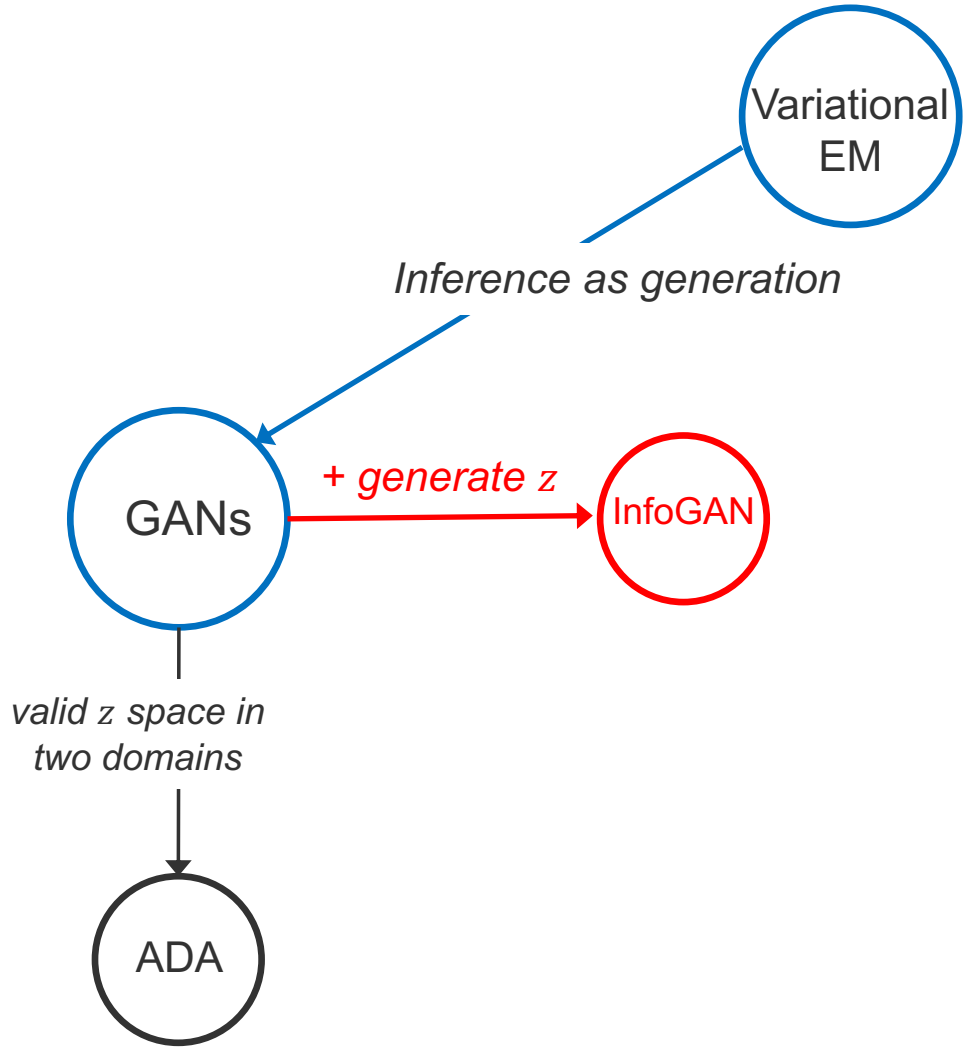


- Similar results as in GANs hold:
  - Let $q^r(\boldsymbol{x}|\boldsymbol{z},y) \propto q_{\eta=\eta_0}(\boldsymbol{z}|\boldsymbol{x},y)q^r_{\phi=\phi_0}(y|\boldsymbol{x})p_{\theta=\theta_0}(\boldsymbol{x})$
  - We have:

$$\nabla_\theta \Big[-\mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q_{\eta_0}(\boldsymbol{z}|\boldsymbol{x},y)q^r_{\phi_0}(y|\boldsymbol{x})\right] \Big]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} =$$

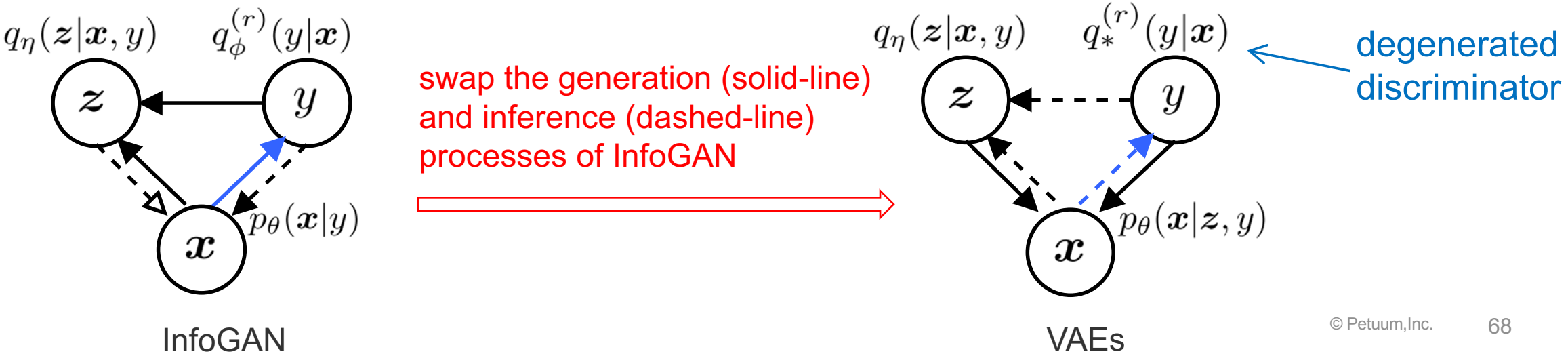
$$\nabla_\theta \Big[\mathbb{E}_{p(y)} \left[\mathrm{KL}\left(p_\theta(\boldsymbol{x}|y)\|q^r(\boldsymbol{x}|\boldsymbol{z},y)\right)\right] - \mathrm{JSD}\left(p_\theta(\boldsymbol{x}|y=0)\|p_\theta(\boldsymbol{x}|y=1)\right) \Big]\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$$

- Next we show correspondences between GANs/InfoGAN and VAEs


Variational EM
Inference as generation
GANs
+ generate z
InfoGAN
valid z space in two domains
ADA

Relates VAEs with GANs

- Resemblance of GAN generator learning to variational inference
  - Suggest strong relations between VAEs and GANs
- Indeed, VAEs are basically minimizing KLD with an opposite direction, and with a degenerated adversarial discriminator



Recap: conventional formulation of VAEs

- Objective:

$$\max_{\boldsymbol{\theta},\boldsymbol{\eta}} \mathcal{L}^{\text{vae}}_{\theta,\eta} = \mathbb{E}_{p_{data}(\boldsymbol{x})}\left[\mathbb{E}_{\tilde{q}_\eta(\boldsymbol{z}|\boldsymbol{x})}\left[\log \tilde{p}_\theta(\boldsymbol{x}|\boldsymbol{z})\right] - \mathrm{KL}(\tilde{q}_\eta(\boldsymbol{z}|\boldsymbol{x})\|\tilde{p}(\boldsymbol{z}))\right]$$

  - $\tilde{p}(\boldsymbol{z})$: prior over $\boldsymbol{z}$
  - $\tilde{p}_\theta(\boldsymbol{x}|\boldsymbol{z})$: generative model
  - $\tilde{q}_\eta(\boldsymbol{z}|\boldsymbol{x})$: inference model
  - Only uses real examples from $p_{data}(\boldsymbol{x})$, lacks adversarial mechanism

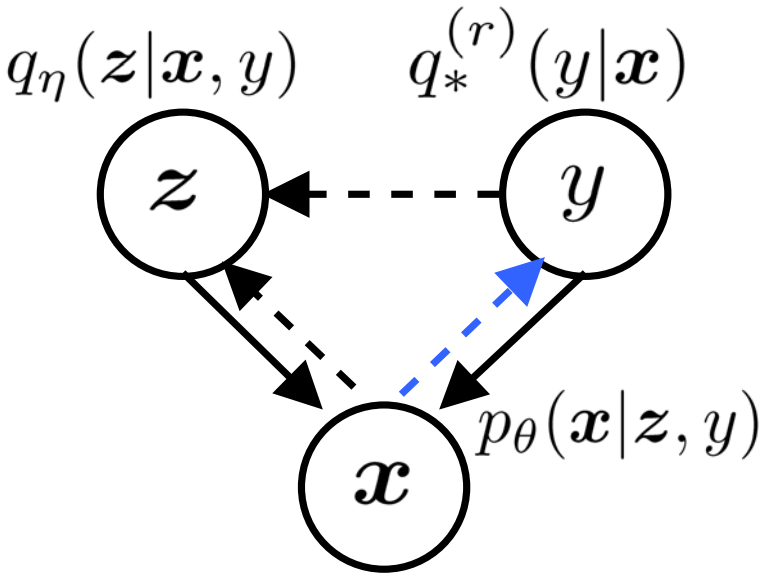
- To align with GANs, let's introduce the real/fake indicator y and adversarial discriminator

VAEs: new formulation

- Assume a *perfect* discriminator $q_*(y|\boldsymbol{x})$
  - $q_*(y=1|\boldsymbol{x})=1$ if $\boldsymbol{x}$ is real examples
  - $q_*(y=0|\boldsymbol{x})=1$ if $\boldsymbol{x}$ is generated samples
  - $q_*^r(y|\boldsymbol{x}) := q_*(1-y|\boldsymbol{x})$
- Generative distribution

$$p_\theta(\boldsymbol{x}|\boldsymbol{z},y)=\begin{cases}p_\theta(\boldsymbol{x}|\boldsymbol{z}) & y=0\\ p_{data}(\boldsymbol{x}) & y=1.\end{cases}$$

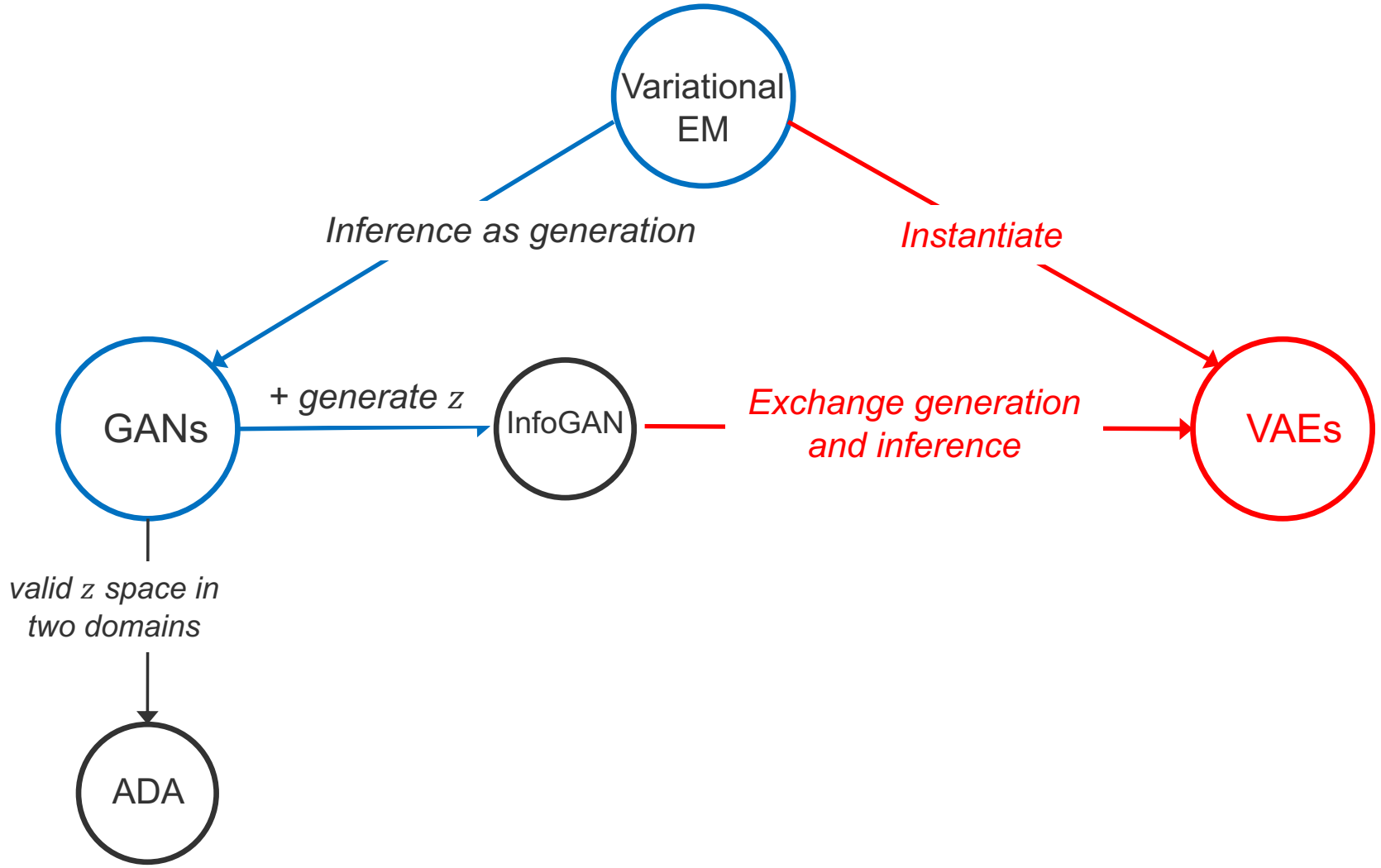
- Let $p_\theta(\boldsymbol{z},y|\boldsymbol{x}) \propto p_\theta(\boldsymbol{x}|\boldsymbol{z},y)p(\boldsymbol{z}|y)p(y)$
- *Lemma 2*



$$\begin{aligned}\mathcal{L}^{vae}_{\theta,\eta} &= 2\cdot\mathbb{E}_{p_{\theta_0}(\boldsymbol{x})}\left[\mathbb{E}_{q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q_*^r(y|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z},y)\right]-KL(q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q_*^r(y|\boldsymbol{x})\|p(\boldsymbol{z}|y)p(y))\right]\\ &= 2\cdot\mathbb{E}_{p_{\theta_0}(\boldsymbol{x})}\left[-KL\left(q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q_*^r(y|\boldsymbol{x})\|p_\theta(\boldsymbol{z},y|\boldsymbol{x})\right)\right].\end{aligned}$$

GANs vs VAEs side by side

	GANs (InfoGAN)	VAEs
Generative distribution	$p_\theta(\boldsymbol{x}\|y) = \begin{cases} p_{g_\theta}(\boldsymbol{x}) & y=0 \\ p_{data}(\boldsymbol{x}) & y=1. \end{cases}$	$p_\theta(\boldsymbol{x}\|\boldsymbol{z},y) = \begin{cases} p_\theta(\boldsymbol{x}\|\boldsymbol{z}) & y=0 \\ p_{data}(\boldsymbol{x}) & y=1. \end{cases}$
Discriminator distribution	$q_\phi(y\|\boldsymbol{x})$	$q_*(y\|\boldsymbol{x})$, perfect, degenerated
$\boldsymbol{z}$-inference model	$q_\eta(\boldsymbol{z}\|\boldsymbol{x},y)$ of InfoGAN	$q_\eta(\boldsymbol{z}\|\boldsymbol{x},y)$
KLD to minimize	$\min_\theta \mathrm{KL}\left(p_\theta(\boldsymbol{x}\|y) \,\|\| \, q^r(\boldsymbol{x}\|\boldsymbol{z},y)\right)$ $\sim \min_\theta \mathrm{KL}(P_\theta \,\|\| \, Q)$	$\min_\theta \mathrm{KL}\left(q_\eta(\boldsymbol{z}\|\boldsymbol{x},y)q^r_*(y\|\boldsymbol{x}) \,\|\| \, p_\theta(\boldsymbol{z},y\|\boldsymbol{x})\right)$ $\sim \min_\theta \mathrm{KL}(Q \,\|\| \, P_\theta)$


Variational EM
Inference as generation
Instantiate
GANs
+ generate z
InfoGAN
Exchange generation and inference
VAEs
valid z space in two domains
ADA

Link back to wake sleep algorithm

- Denote
  - Latent variables $\boldsymbol{h}$
  - Parameters $\boldsymbol{\lambda}$
- Recap: wake sleep algorithm

$$\text{Wake}: \quad \max_{\boldsymbol{\theta}} \mathbb{E}_{q_\lambda(\boldsymbol{h}|\boldsymbol{x})p_{data}(\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{h})\right]$$

$$\text{Sleep}: \quad \max_{\boldsymbol{\lambda}} \mathbb{E}_{p_\theta(\boldsymbol{x}|\boldsymbol{h})p(\boldsymbol{h})}\left[\log q_\lambda(\boldsymbol{h}|\boldsymbol{x})\right]$$

VAEs vs. Wake-sleep

- Wake sleep algorithm

$$\text{Wake}: \quad \max_{\boldsymbol{\theta}} \mathbb{E}_{q_\lambda(\boldsymbol{h}|\boldsymbol{x})p_{data}(\boldsymbol{x})} [\log p_\theta(\boldsymbol{x}|\boldsymbol{h})]$$

$$\text{Sleep}: \quad \max_{\boldsymbol{\lambda}} \mathbb{E}_{p_\theta(\boldsymbol{x}|\boldsymbol{h})p(\boldsymbol{h})} [\log q_\lambda(\boldsymbol{h}|\boldsymbol{x})]$$

- Let $\boldsymbol{h}$ be $\boldsymbol{z}$, and $\boldsymbol{\lambda}$ be $\boldsymbol{\eta}$

$$\Rightarrow \max_{\boldsymbol{\theta}} \mathbb{E}_{q_\eta(\boldsymbol{z}|\boldsymbol{x})p_{data}(\boldsymbol{x})} [\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] ,$$ recovers VAE objective of optimizing $\boldsymbol{\theta}$

- VAEs extend wake phase by also learning the inference model (η)

$$\max_{\boldsymbol{\theta},\boldsymbol{\eta}} \mathcal{L}^{\text{vae}}_{\theta,\eta} = \mathbb{E}_{q_\eta(\boldsymbol{z}|\boldsymbol{x})p_{data}(\boldsymbol{x})} [\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - \mathbb{E}_{p_{data}(\boldsymbol{x})} [\mathrm{KL}(q_\eta(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z}))]$$

  - Minimize the KLD in the original variational free energy wrt. $\boldsymbol{\eta}$
  - Stick to minimizing the wake-phase KLD wrt. both $\boldsymbol{\theta}$ and $\boldsymbol{\eta}$
  - Do not involve sleep-phase objective
  - Recall: sleep phase minimizes the *reverse* KLD in the variational free energy

GANs vs. Wake-sleep

- Wake sleep algorithm

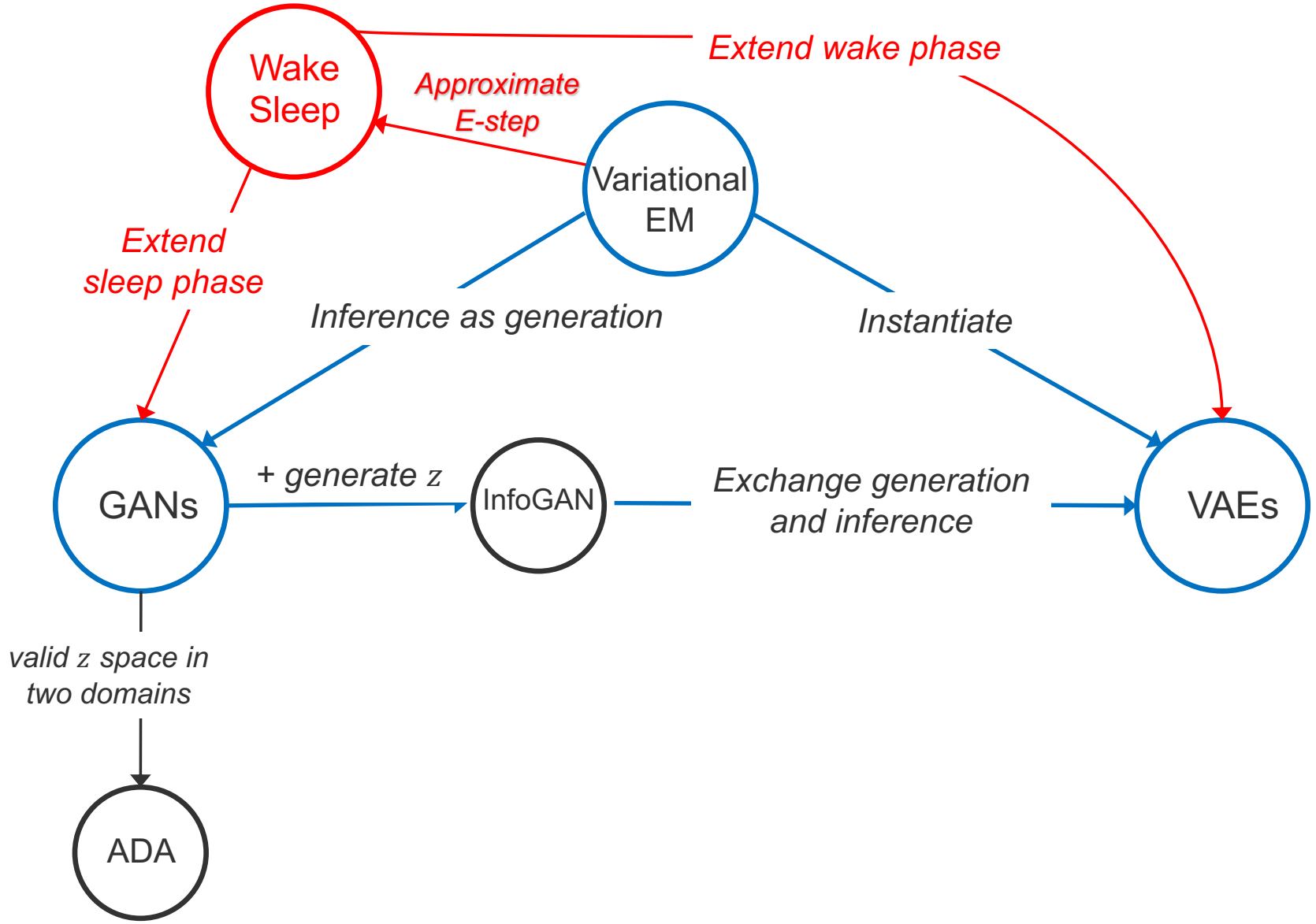
$$\text{Wake}: \quad \max_{\boldsymbol{\theta}} \mathbb{E}_{q_\lambda(\boldsymbol{h}|\boldsymbol{x})p_{data}(\boldsymbol{x})} [\log p_\theta(\boldsymbol{x}|\boldsymbol{h})]$$

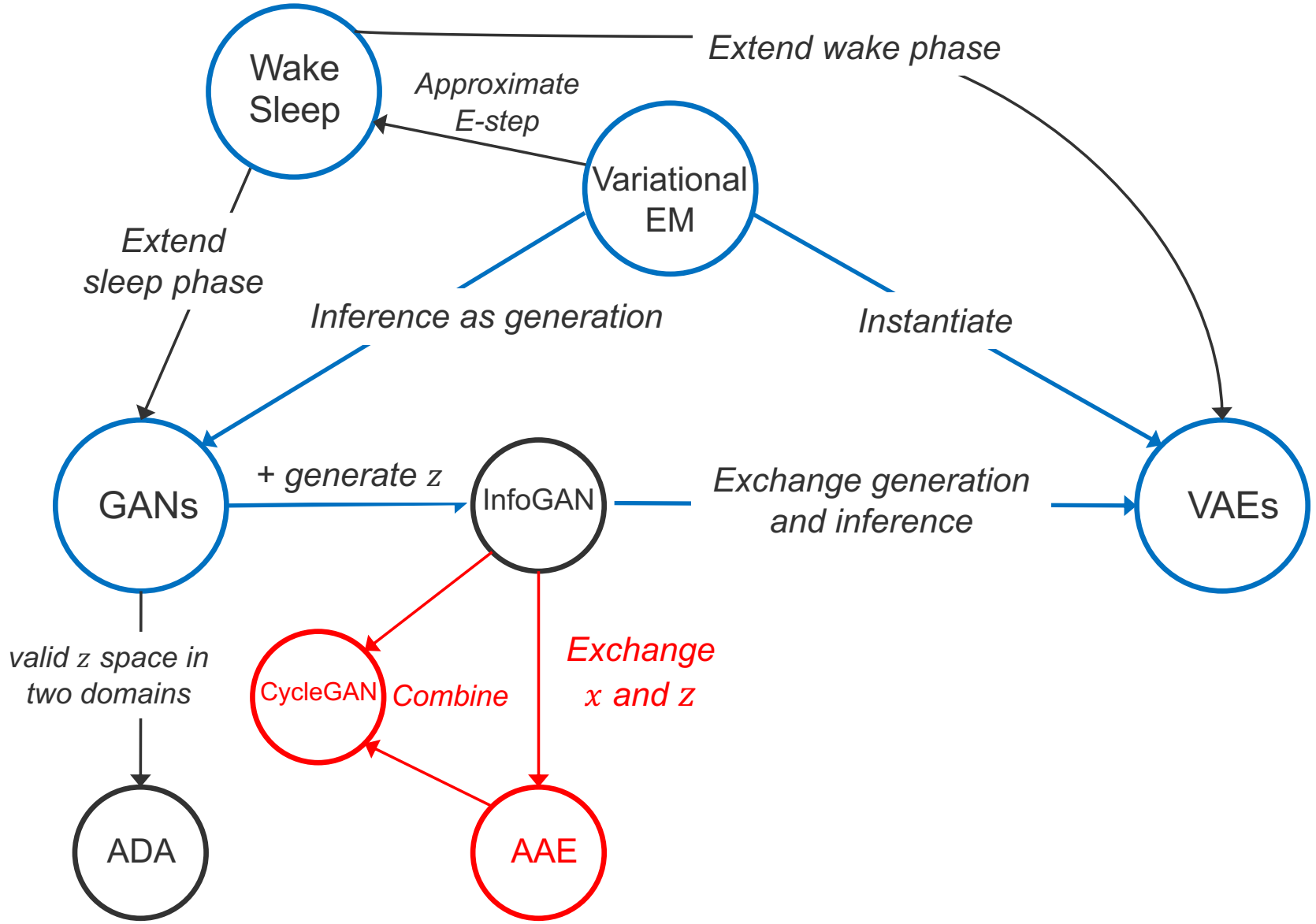
$$\text{Sleep}: \quad \max_{\boldsymbol{\lambda}} \mathbb{E}_{p_\theta(\boldsymbol{x}|\boldsymbol{h})p(\boldsymbol{h})} [\log q_\lambda(\boldsymbol{h}|\boldsymbol{x})]$$

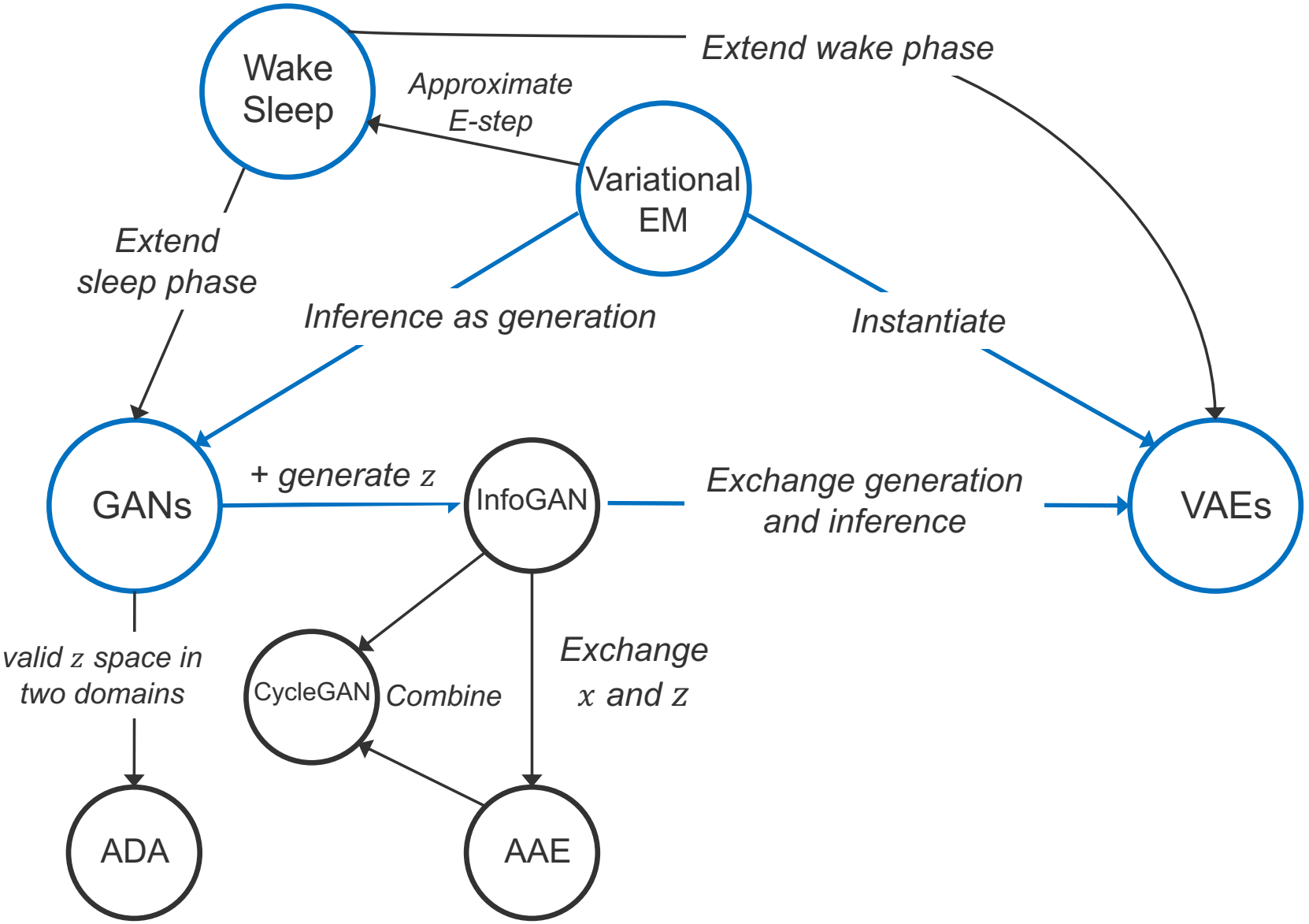
- Let $\boldsymbol{h}$ be y, and $\boldsymbol{\lambda}$ be $\boldsymbol{\phi}$

$$\Rightarrow \quad \max_{\boldsymbol{\phi}} \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} [\log q_\phi(y|\boldsymbol{x})],$$ recovers GAN objective of optimizing $\boldsymbol{\phi}$

- GANs extend sleep phase by also learning the generative model ($\boldsymbol{\theta}$)
  - Directly extending sleep phase: $\max_{\boldsymbol{\phi}} \mathcal{L}_\phi = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} [\log q_\phi(y|\boldsymbol{x})]$
  - GANs: $\max_{\boldsymbol{\theta}} \mathcal{L}_\theta = \mathbb{E}_{p_\theta(\boldsymbol{x}|y)p(y)} [\log q^r_\phi(y|\boldsymbol{x})]$
  - The only difference is replacing $q_{\boldsymbol{\phi}}$ with $q^r_{\boldsymbol{\phi}}$
  - This is where adversarial mechanism come about !
  - GANs stick to minimizing the sleep-phase KLD
  - Do not involve wake-phase objective


Wake Sleep
Extend wake phase
Approximate E-step
Variational EM
Extend sleep phase
Inference as generation
Instantiate
GANs
+ generate z
InfoGAN
Exchange generation and inference
VAEs
valid z space in two domains
ADA


Wake Sleep
Extend wake phase
Approximate E-step
Variational EM
Extend sleep phase
Inference as generation
Instantiate
GANs
+ generate z
InfoGAN
Exchange generation and inference
VAEs
valid z space in two domains
CycleGAN
Combine
Exchange x and z
ADA
AAE



Combining GANs and VAEs

VAE/GAN Joint Models

Implicit inference

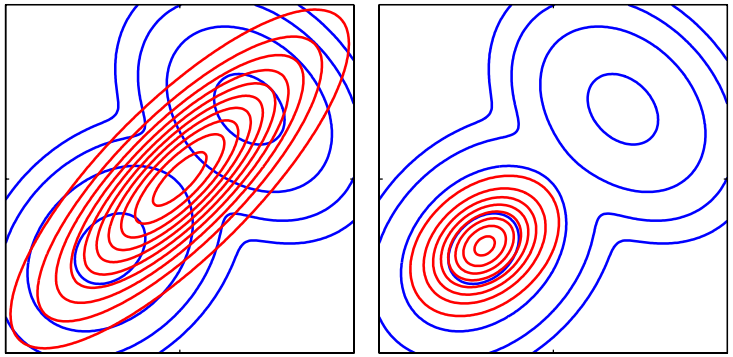
IW-GAN

AA-VAE

Mutual exchanges of ideas: augment the loss functions

	GANs (InfoGAN)	VAEs
KLD to minimize	$\min_\theta \mathrm{KL}\left(p_\theta(\boldsymbol{x}\|y) \,\|\|\, q^r(\boldsymbol{x}\|\boldsymbol{z}, y)\right)$ $\sim \min_\theta \mathrm{KL}(P_\theta \,\|\|\, Q)$	$\min_\theta \mathrm{KL}(q_\eta(\boldsymbol{z}\|\boldsymbol{x}, y) q^r_*(y\|\boldsymbol{x}) \,\|\|\, p_\theta(\boldsymbol{z}, y\|\boldsymbol{x}))$ $\sim \min_\theta \mathrm{KL}(Q \,\|\|\, P_\theta)$

- Asymmetry of KLDs inspires combination of GANs and VAEs
  - GANs: $\min_\theta \mathrm{KL}(P_\theta \| Q)$ tends to missing mode
  - VAEs: $\min_\theta \mathrm{KL}(Q \| P_\theta)$ tends to cover regions with small values of p_{data}



Mode covering Mode missing

[Figure courtesy: PRML]

Mutual exchanges of ideas: augment the loss functions

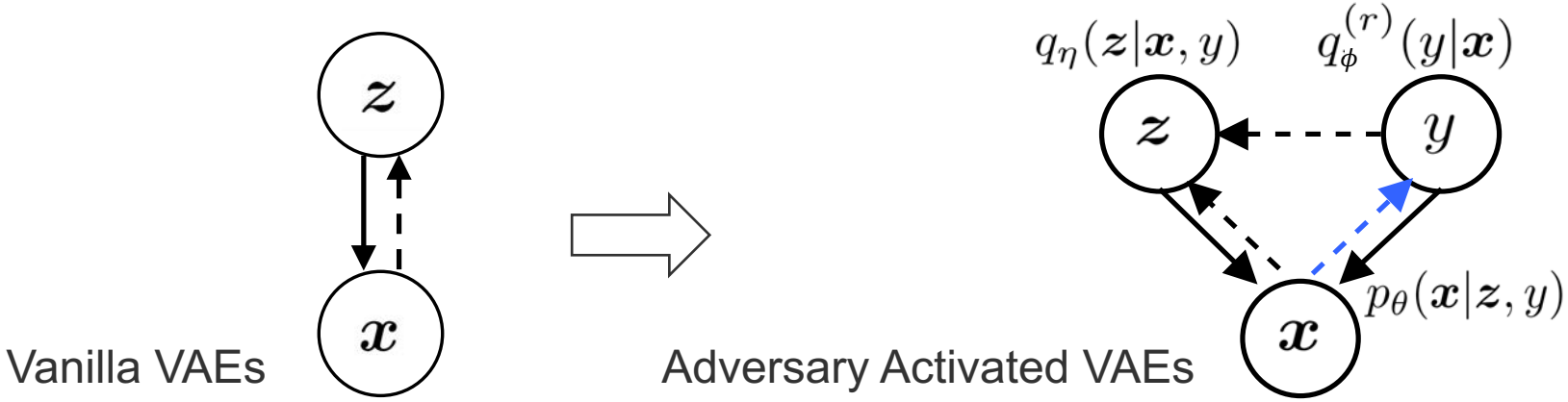
	GANs (InfoGAN)	VAEs
KLD to minimize	$\min_\theta \mathrm{KL}\left(p_\theta(\boldsymbol{x}\|y) \,\|\|\, q^r(\boldsymbol{x}\|\boldsymbol{z}, y)\right)$ $\sim \min_\theta \mathrm{KL}(P_\theta \,\|\|\, Q)$	$\min_\theta \mathrm{KL}(q_\eta(\boldsymbol{z}\|\boldsymbol{x}, y) q^r_*(y\|\boldsymbol{x}) \,\|\|\, p_\theta(\boldsymbol{z}, y\|\boldsymbol{x}))$ $\sim \min_\theta \mathrm{KL}(Q \,\|\|\, P_\theta)$

- Asymmetry of KLDs inspires combination of GANs and VAEs
  - GANs: $\min_\theta \mathrm{KL}(P_\theta \| Q)$ tends to missing mode
  - VAEs: $\min_\theta \mathrm{KL}(Q \| P_\theta)$ tends to cover regions with small values of p_{data}
  - Augment VAEs with GAN loss [Larsen et al., 2016]
    - Alleviate the mode covering issue of VAEs
    - Improve the sharpness of VAE generated images
  - Augment GANs with VAE loss [Che et al., 2017]
    - Alleviate the mode missing issue of GANs

Mutual exchanges of ideas: augment the graphical model

	GANs (InfoGAN)	VAEs
Discriminator distribution	$q_\phi(y\|\boldsymbol{x})$	$q_*(y\|\boldsymbol{x})$, perfect, degenerated

- Activate the adversarial mechanism in VAEs
  - Enable adaptive incorporation of fake samples for learning
  - Straightforward derivation by making symbolic analog to GANs



Vanilla VAEs

Adversary Activated VAEs

Adversary Activated VAEs (AAVAE)

- Vanilla VAEs:

$$\max_{\boldsymbol{\theta},\boldsymbol{\eta}} \mathcal{L}^{\text{vae}}_{\theta,\eta} = \mathbb{E}_{p_{\theta_0}(\boldsymbol{x})}\left[\mathbb{E}_{q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q^r_*(y|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z},y)\right] - \mathrm{KL}(q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q^r_*(y|\boldsymbol{x})\|p(\boldsymbol{z}|y)p(y))\right]$$

- Replace $q_*(y|\boldsymbol{x})$ with learnable one $q_{\boldsymbol{\phi}}(y|\boldsymbol{x})$ with parameters $\boldsymbol{\phi}$
  - As usual, denote reversed distribution $q^r_{\boldsymbol{\phi}}(y|x) = q_{\boldsymbol{\phi}}(y|x)$

$$\max_{\boldsymbol{\theta},\boldsymbol{\eta}} \mathcal{L}^{\text{aavae}}_{\theta,\eta} = \mathbb{E}_{p_{\theta_0}(\boldsymbol{x})}\left[\mathbb{E}_{q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q^r_{\boldsymbol{\phi}}(y|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z},y)\right] - \mathrm{KL}(q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q^r_{\boldsymbol{\phi}}(y|\boldsymbol{x})\|p(\boldsymbol{z}|y)p(y))\right]$$

AAVAE: adaptive data selection

$$\max_{\boldsymbol{\theta},\boldsymbol{\eta}} \mathcal{L}^{\text{aavae}}_{\theta,\eta} = \mathbb{E}_{p_{\theta_0}(\boldsymbol{x})}\left[\mathbb{E}_{q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q^r_\phi(y|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z},y)\right] - \mathrm{KL}(q_\eta(\boldsymbol{z}|\boldsymbol{x},y)q^r_\phi(y|\boldsymbol{x})\|p(\boldsymbol{z}|y)p(y))\right]$$

- An effective data selection mechanism:
  - Both generated samples and real examples are weighted by $q^r_\phi(y=0|\boldsymbol{x}) = q_\phi(y=1|\boldsymbol{x})$
  - Only samples that resembles real data and fool the discriminator will be used for training
  - A real example receiving large weight $q^r_\phi(y|\boldsymbol{x})$

    ⇒ Easily recognized by the discriminator as real

    ⇒ Hard to be simulated from the generator

    ⇒ Hard examples get larger weights

AAVAE: discriminator learning

- Use the binary classification objective as in GAN

$$\max_{\boldsymbol{\phi}} \mathcal{L}_{\phi}^{\text{aavae}} = \mathbb{E}_{p_\theta(\boldsymbol{x}|\boldsymbol{z},y)p(\boldsymbol{z}|y)p(y)} \left[\log q_\phi(y|\boldsymbol{x})\right]$$

AAVAE: empirical results

- Applied the adversary activating method on
  - vanilla VAEs
  - class-conditional VAEs (CVAE)
  - semi-supervised VAEs (SVAE)

AAVAE: empirical results

- Evaluated test-set variational lower bound on MNIST
  - The higher the better

Train Data Size	VAE	AA-VAE	CVAE	AA-CVAE	SVAE	AA-SVAE
1%	-122.89	**-122.15**	-125.44	**-122.88**	-108.22	**-107.61**
10%	-104.49	**-103.05**	-102.63	**-101.63**	-99.44	**-98.81**
100%	-92.53	**-92.42**	-93.16	**-92.75**	—	—

- X-axis: the ratio of training data for learning: (1%, 10%, 100%)
- Y-axis: value of test-set lower bound

AAVAE: empirical results

- Evaluated classification accuracy of SVAE and AA-SVAE

	1%	10%
SVAE	0.9412±.0039	0.9768±.0009
AASVAE	**0.9425±.0045**	**0.9797±.0010**

- Used 1% and 10% data labels in MNIST

Mutual exchanges of ideas

- AAVAE enhances VAEs with ideas from GANs
- We can also enhance GANs with ideas from VAEs
- VAEs maximize a variational lower bound of log likelihood
- Importance weighted VAE (IWAE) [Burda et al., 2016]
  - Maximizes a tighter lower bound through importance sampling
- The variational inference interpretation of GANs allows the importance weighting method to be straightforwardly applied to GANs
  - Just copy the derivations of IWAE side by side with little adaptions!

Importance weighted GANs (IWGAN)

- Generator learning in vanilla GANs

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x}\sim p_\theta(\boldsymbol{x}|y)p(y)} \left[\log q^r_{\phi_0}(y|\boldsymbol{x})\right]$$

- Generator learning in IWGAN

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_k\sim p_\theta(\boldsymbol{x}|y)p(y)} \left[\sum_{i=1}^{k} \frac{q^r_{\phi_0}(y|\boldsymbol{x}_i)}{q_{\phi_0}(y|\boldsymbol{x}_i)} \log q^r_{\phi_0}(y|\boldsymbol{x}_i)\right]$$

  - Assigns higher weights to samples that are more realistic and fool the discriminator better

IWGAN: empirical results

- Applied the importance weighting method to
  - vanilla GANs
  - class-conditional GANs (CGAN)
    - CGAN adds one dimension to code z to represent the class label
    - The derivations of the IW extension remain the same as in vanilla GANs

IWGAN: empirical results

- Evaluated on MNIST and SVHN
- Used pretrained NN to evaluate:
  - Inception scores of samples from GANs and IW-GAN
    - Confidence of a pre-trained classifier on generated samples + diversity of generated samples

	MNIST	SVHN
GAN	8.34±.03	5.18±.03
IWGAN	**8.45±.04**	**5.34±.03**

  - Classification accuracy of samples from CGAN and IW-CGAN

	MNIST	SVHN
CGAN	0.985±.002	0.797±.005
IWCGAN	**0.987±.002**	**0.798±.006**

Recap: Variational Inference

Maximize the variational lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$, or equivalently, minimize free energy

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -\log p(\boldsymbol{x}) + KL(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$$

- E-step: maximize $\mathcal{L}$ wrt. $\boldsymbol{\phi}$ with $\boldsymbol{\theta}$ fixed

$$\max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL(q_{\boldsymbol{\phi}}(z|x)||p(z))$$

  - If with closed form solutions

$$q^*_{\boldsymbol{\phi}}(z|x) \propto \exp[\log p_{\theta}(x, z)]$$

- M-step: maximize $\mathcal{L}$ wrt. $\boldsymbol{\theta}$ with $\boldsymbol{\phi}$ fixed

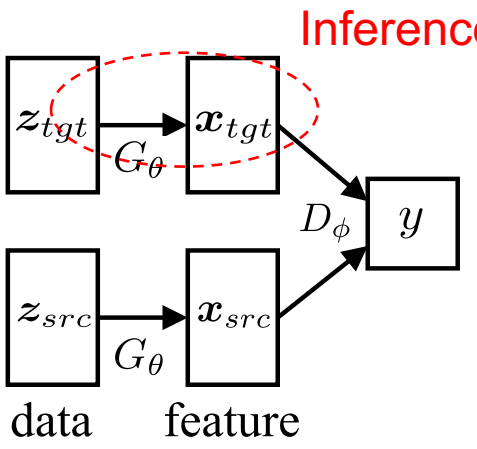
$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|x)}[\log p_{\theta}(x|z)] + KL(q_{\boldsymbol{\phi}}(z|x)||p(z))$$

Discussion: Modeling latent vs. visible variables

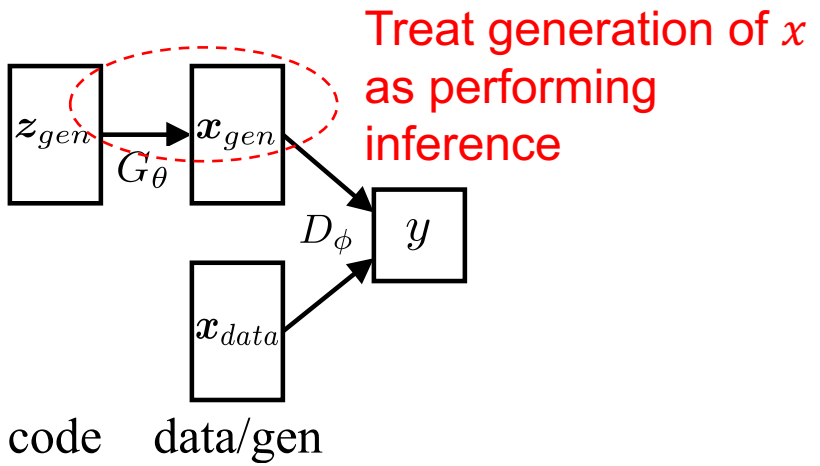
- Latent and visible variables are traditionally distinguished clearly and modeled in very different ways
- A key thought in the new formulation:
  - Not necessary to make clear boundary between latent and visible variables,
  - And between inference and generation
  - Instead treat them as a symmetric pair

Symmetric modeling of latent & visible variables

- Help with modeling and understanding:
  - Treating the generation space $\boldsymbol{x}$ in GANs as latent
    - reveals the connection between GANs and ADA
    - provides an variational inference interpretation of generation



ADA



GANs

Symmetric modeling of latent & visible variables

- Help with modeling and understanding:
  - Treating the generation space $\boldsymbol{x}$ in GANs as latent
    - reveals the connection between GANs and ADA
    - provides an variational inference interpretation of generation
  - Wake sleep algorithm
    - wake phase reconstructs visible variables based on latents
    - sleep phase reconstructs latent variables based on visibles
    - latent and visible variables are treated in a completely symmetric way

Wake: $\max_{\boldsymbol{\theta}} \mathrm{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} [\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})]$

Sleep: $\max_{\boldsymbol{\phi}} \mathrm{E}_{p_{\theta}(\boldsymbol{z},\boldsymbol{x})} [\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})]$

Symmetric modeling of latent & visible variables

- New modeling approaches narrow the gap

Empirical distributions over visible variables

- Impossible to be explicit distribution
  - The only information we have is the observe data examples
  - Do not know the true parametric form of data distribution
- Naturally an implicit distribution
  - Easy to sample from, hard to evaluate likelihood

Prior distributions over latent variables

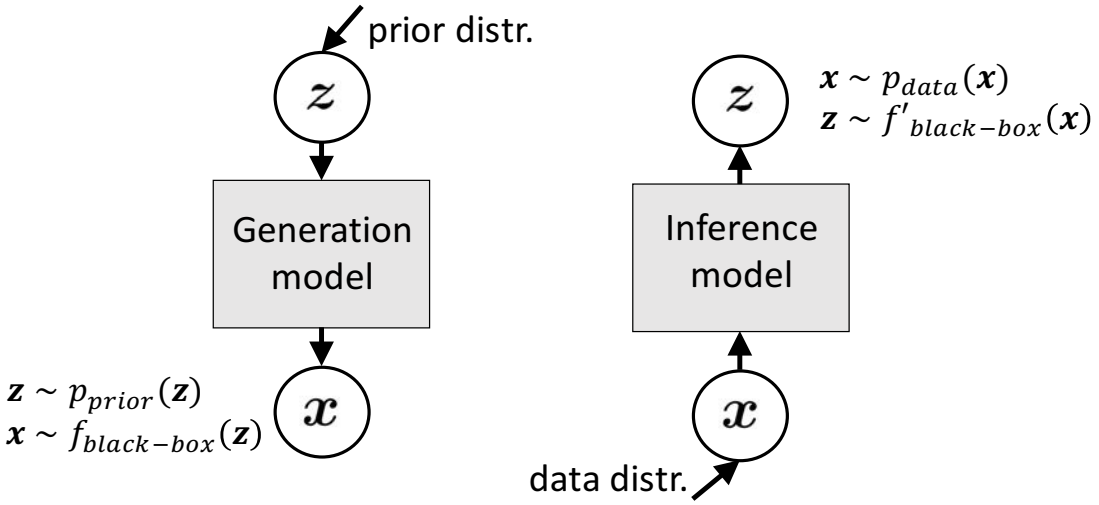
- Traditionally defined as explicit distributions, e.g., Gaussian prior distribution
  - Amiable for likelihood evaluation
  - We can assume the parametric form according to our prior knowledge
- New tools to allow implicit priors and models
  - GANs, density ratio estimation, approximate Bayesian computations
  - E.g., adversarial autoencoder [Makhzani et al., 2015] replaces the Gaussian prior of vanilla VAEs with implicit priors

Symmetric modeling of latent & visible variables

- No difference in terms of formulations
  - with implicit distributions and black-box NN models
  - just swap the symbols x and z

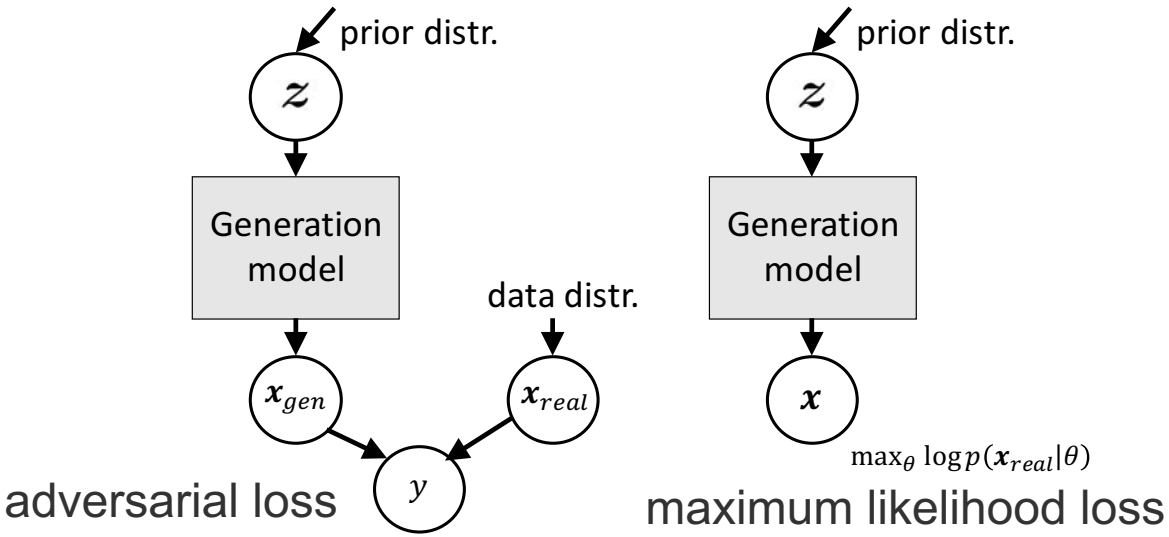
$$\boldsymbol{z} \sim p_{prior}(\boldsymbol{z})$$
$$\boldsymbol{x} \sim f_{black-box}(\boldsymbol{z})$$

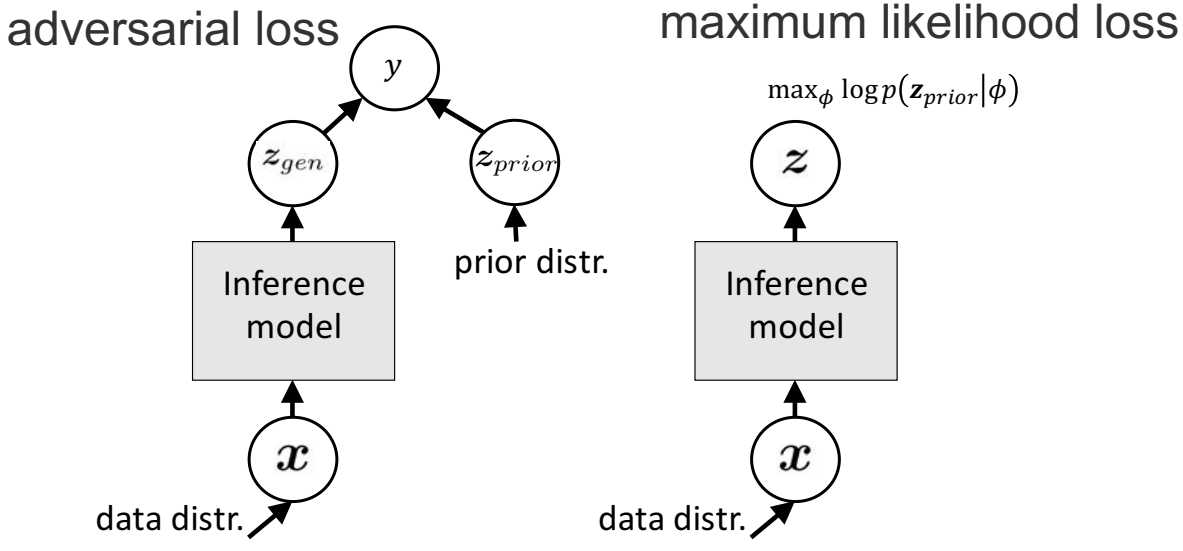
$$\boldsymbol{x} \sim p_{data}(\boldsymbol{x})$$
$$\boldsymbol{z} \sim f'_{black-box}(\boldsymbol{x})$$



Symmetric modeling of latent & visible variables

- No difference in terms of formulations
  - with implicit distributions and black-box NN models
- Difference in terms of space complexity
  - depend on the problem at hand
  - choose appropriate tools:
    - implicit/explicit distribution, adversarial/maximum-likelihood optimization, …



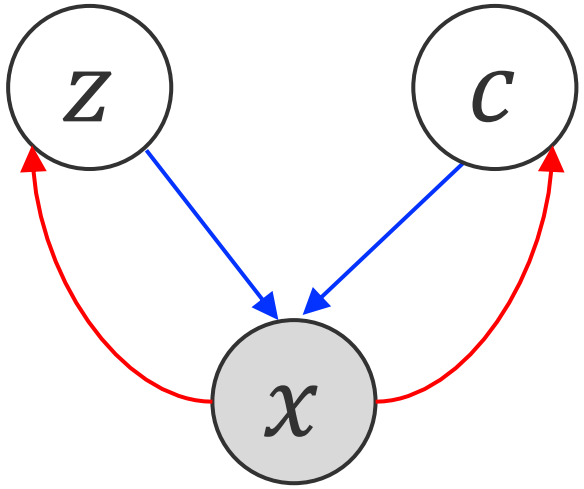


Examples of symmetric modeling: Controlled generation of text and image

- Goals:
  - Generation of realistic sentences / images
  - Control of user-specified attributes
- Conditional generative models:
  - Control-gen [1] for controlled text generation
  - SGAN [2] for controlled image generation
  - Structured code c with designated semantics
    - Text: sentiment, tense, …
    - Image: object, ...
  - Unstructured code z:
    - Captures all other aspects of the sample

Generation: $p_\theta(\boldsymbol{x}|\boldsymbol{z},\boldsymbol{c})$

Inference: $q_\phi(\boldsymbol{z}|\boldsymbol{x}), q_\phi(\boldsymbol{c}|\boldsymbol{x})$

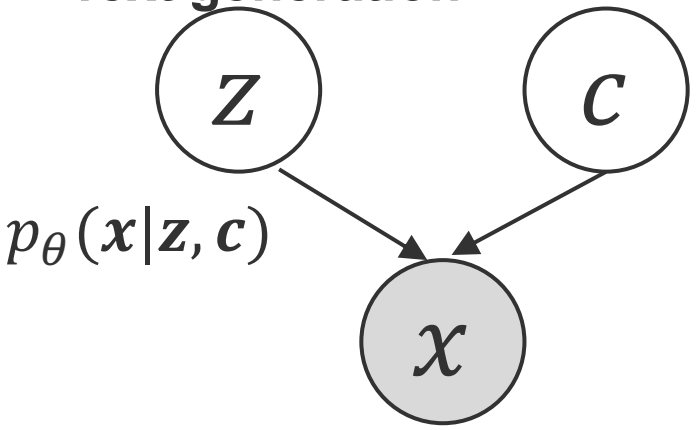


[1] Hu et al., “Towards Controlled Generation of Text”, ICML 2017
[2] Zhang et al., “Structured GANs”, NIPS 2017

Losses in generation space

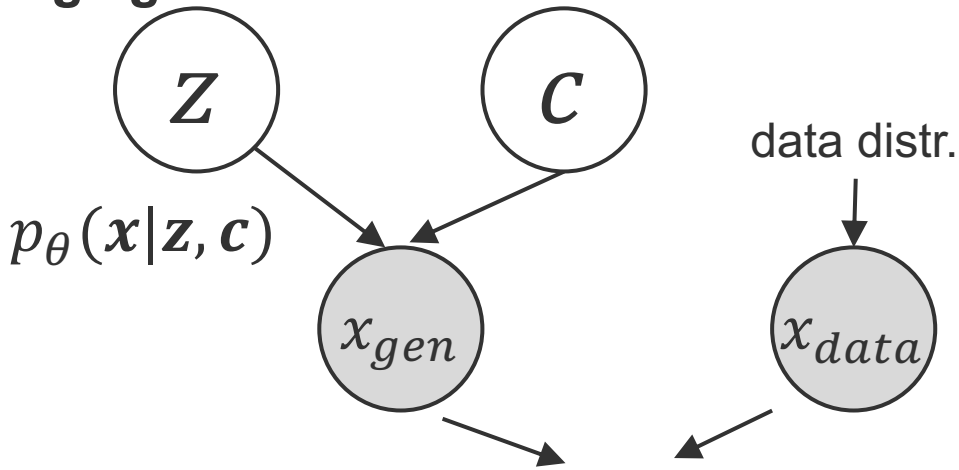
- Generation of realistic sentences / images
  - => **Generation distribution** stays "close" to the **real data distribution**
  - Control-gen for text: maximum likelihood loss in $\boldsymbol{x}$ space
  - SGAN for image: adversarial loss in joint $(\boldsymbol{x}, \boldsymbol{z})$ and $(\boldsymbol{x}, \boldsymbol{c})$ space

Text generation



maximum likelihood loss

Image generation



adversarial loss

Losses in latent space

- Control of user-specified attributes
  - => **Inference distributions** stay close to **true posterior distributions**
    & **Inference distributions** $p_\phi(\boldsymbol{c}|\boldsymbol{x})$ stay close to **data label distributions**
- Control-gen for text: maximum likelihood loss in $\boldsymbol{z}$ and $\boldsymbol{c}$ space
- SGAN for image: adversarial loss in joint $(\boldsymbol{x}, \boldsymbol{z})$ and $(\boldsymbol{x}, \boldsymbol{c})$ space
- Both: maximum likelihood loss in $\boldsymbol{c}$ space

Text generation

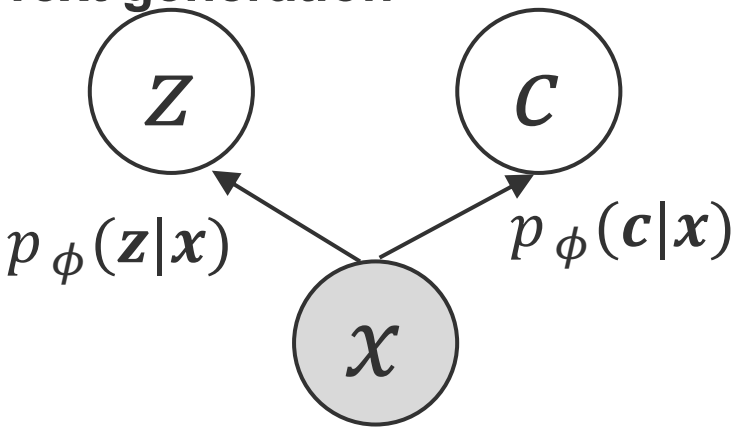
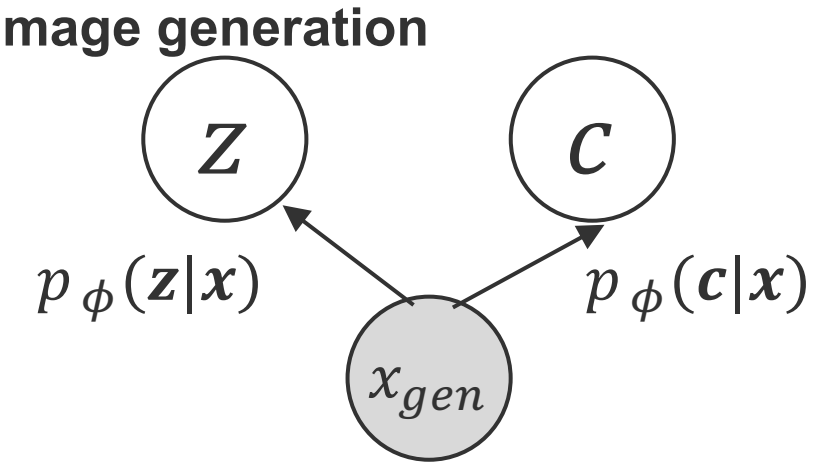


Image generation



Example results

- Control-gen for text

Generated sentence pairs with opposite sentiment *(text style transfer)*

the film is strictly routine !
the film is full of imagination .

after watching this movie , i felt that disappointed .
after seeing this film , i 'm a fan .

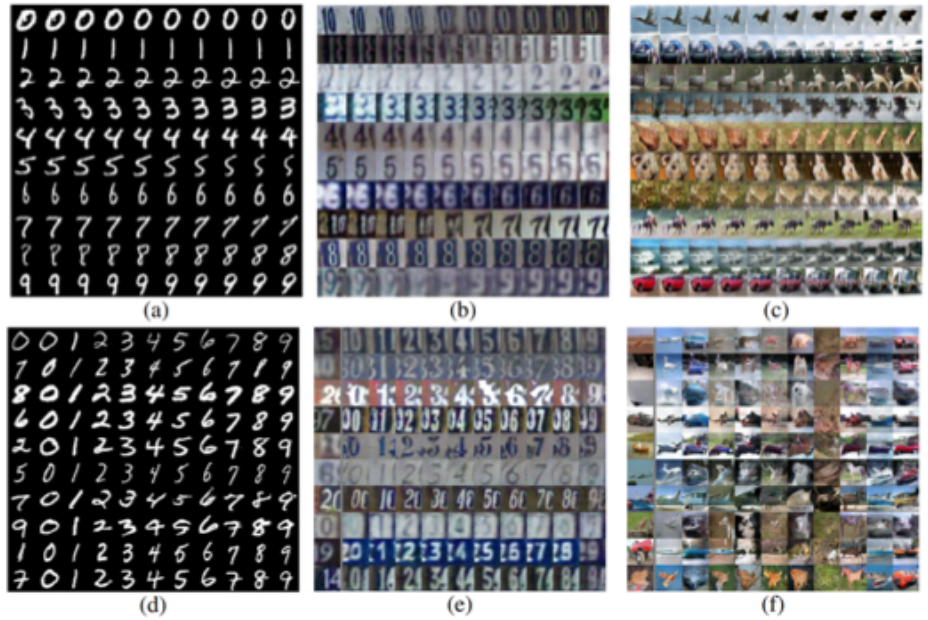
the acting is uniformly bad either .
the performances are uniformly good .

this is just awful .
this is pure genius .

- SGAN for image

Image progression

Style transfer



Part-II: Conclusions

Z Hu, Z YANG, R Salakhutdinov, E Xing,
"**On Unifying Deep Generative Models**", arxiv 1706.00550

- Deep generative models research have a long history
  - Deep blief nets / Helmholtz machines / Predictability Minimization / …
- Unification of deep generative models
  - GANs and VAEs are essentially minimizing KLD in opposite directions
    - Extends two phases of classic wake sleep algorithm, respectively
  - A general formulation framework useful for
    - Analyzing broad class of existing DGM and variants: ADA/InfoGAN/Joint-models/…
    - Inspiring new models and algorithms by borrowing ideas across research fields
- Symmetric view of latent/visible variables
  - No difference in formulation with implicit prior distributions and black-box NN transformations
  - Difference in space complexity: choose appropriate tools
- Structured modeling of latent space

Plan

- Statistical And Algorithmic Foundation and Insight of Deep Learning
- On Unified Framework of Deep Generative Models
- Computational Mechanisms: Distributed Deep Learning Architectures

PETUUM

Part-III (1)
Programming Interface and Software Frameworks

Outline

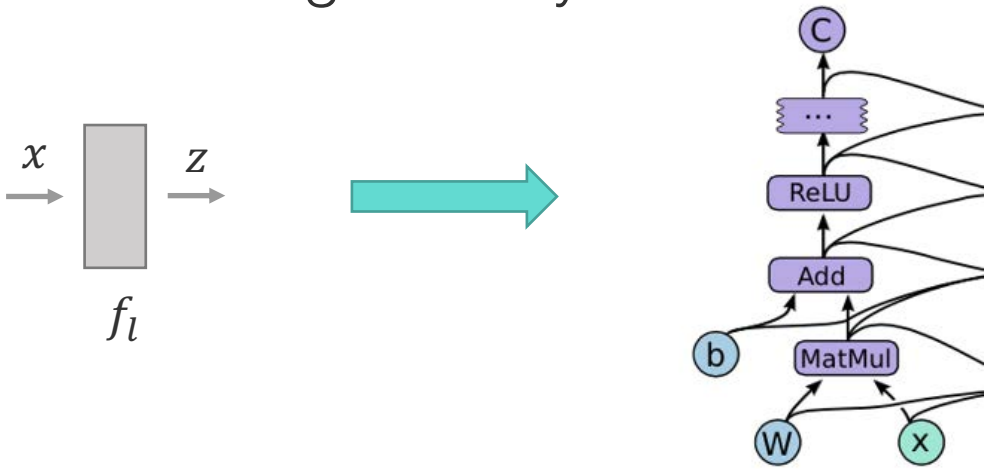
- Deep Learning as Dataflow Graphs
- Auto-differential Libraries
- Static and Dynamic Neural Networks
- DyNet
- Cavs

Outline

- Deep Learning as Dataflow Graphs
- Auto-differential Libraries
- Static and Dynamic Neural Networks
- DyNet
- Cavs

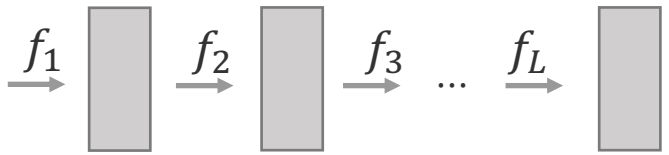
A Computational Layer in DL

- A layer in a neural network is composed of a few finer computational operations
  - A layer l has input x and output z, and transforms x into z following: $y = Wx + b, z = ReLU(y)$
  - Denote the transformation of layer l as f_l, which can be represented as a dataflow graphs: the input x flow though the layer



From Layers to Networks

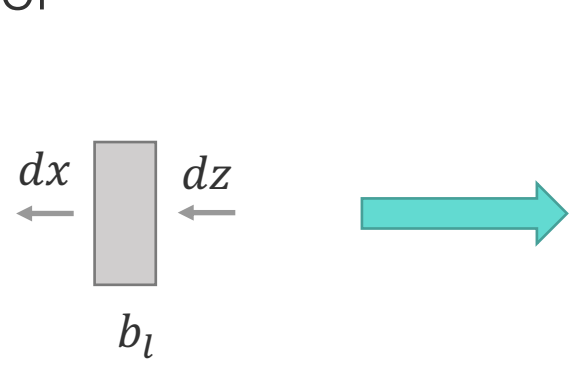
- A neural network is thus a few stacked layers $l = 1, \ldots, L$, where every layer represents a function transform f_l
  - The forward computation proceeds by sequentially executing $f_1, f_2, f_3, \ldots, f_L$

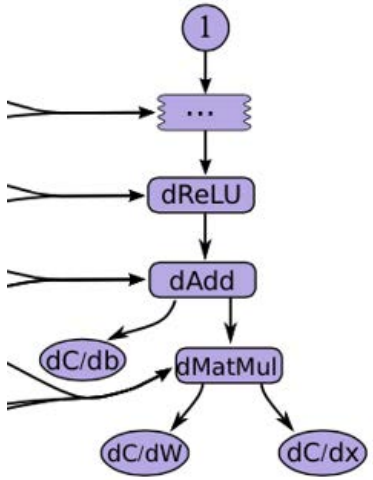


  - Training the neural network involves deriving the gradient of its parameters with a backward pass (next slides)

A Computational Layer in DL

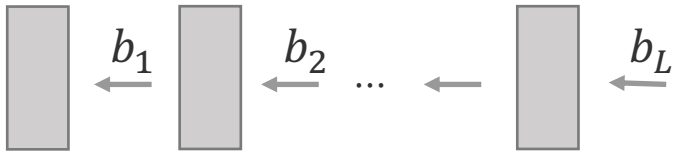
- Denote the backward pass through a layer l as b_l
  - b_l derives the gradients of the input $x(\mathrm{d}x)$,given the gradient of z as $\mathrm{d}z$, as well as the gradients of the parameters $\mathbf{W}, \mathbf{b}$
  - $\mathrm{d}x$ will be the backward input of its previous layer $l-1$
  - Backward pass can be thought as a backward dataflow where the gradient flow through the layer





Backpropagation through a NN

- The backward computation proceeds by sequentially executing $b_L, b_{L-1}, b_{L-2}, \ldots, b_1$



A Layer as a Dataflow Graph

- Give the forward computation flow, gradients can be computed by auto differentiation
  - Automatically derive the backward gradient flow graph from the forward dataflow graph

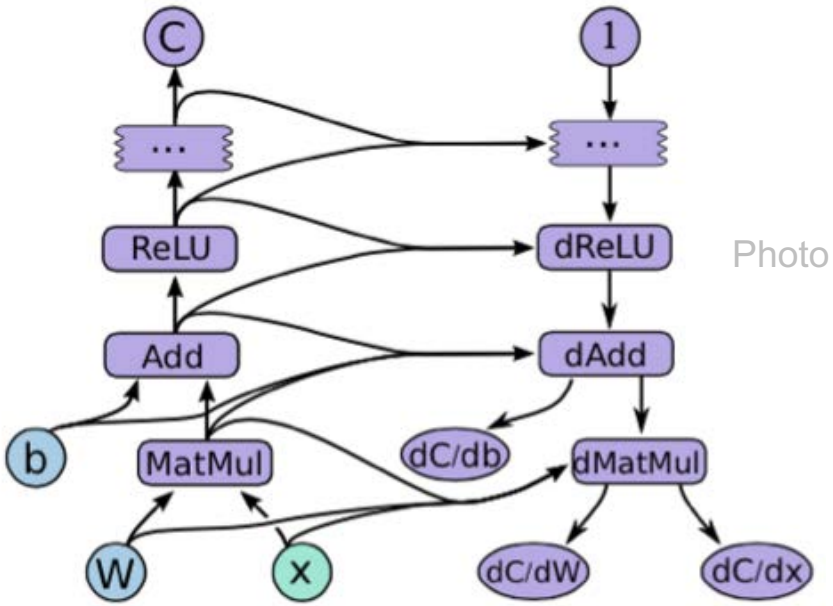
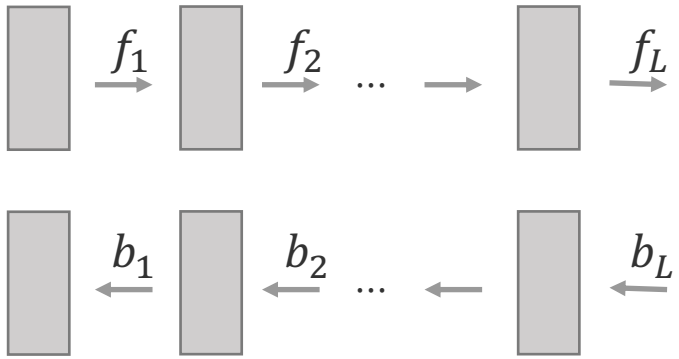


Photo from TensorFlow website

A Network as a Dataflow Graph

- Gradients can be computed by auto differentiation
  - Automatically derive the gradient flow graph from the forward dataflow graph



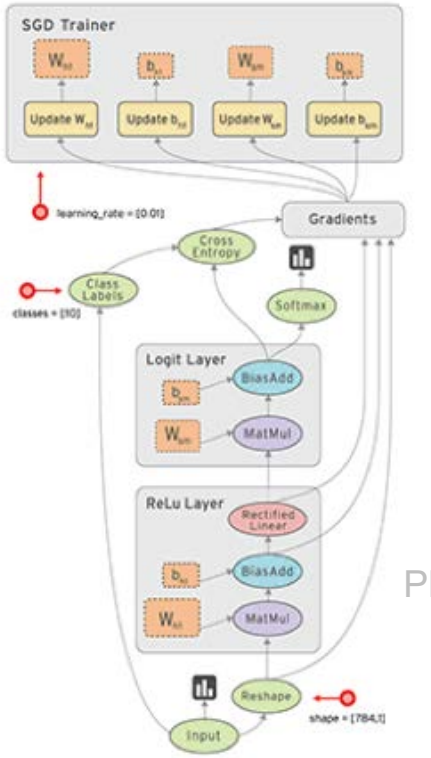
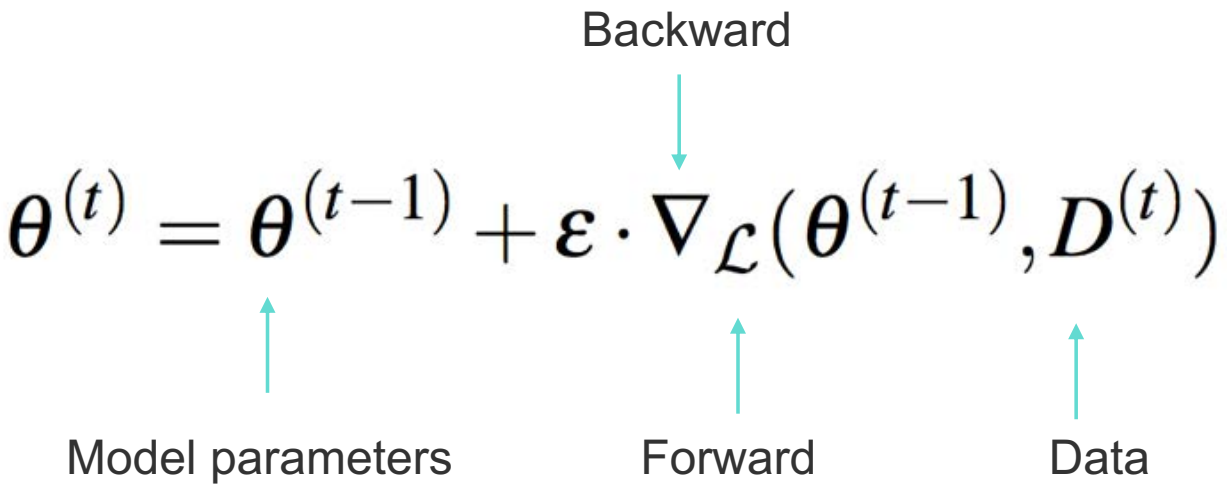


Photo from TensorFlow website

Gradient Descent via Backpropagation

- The computational workflow of deep learning
  - Forward, which we usually also call inference: forward dataflow
  - Backward, which derives the gradients: backward gradient flow
  - Apply/update gradients and repeat

- Mathematically,



Outline

- Deep Learning as Dataflow Graphs
- **Auto-differential Libraries**
- Static and Dynamic Neural Networks
- DyNet
- Cavs

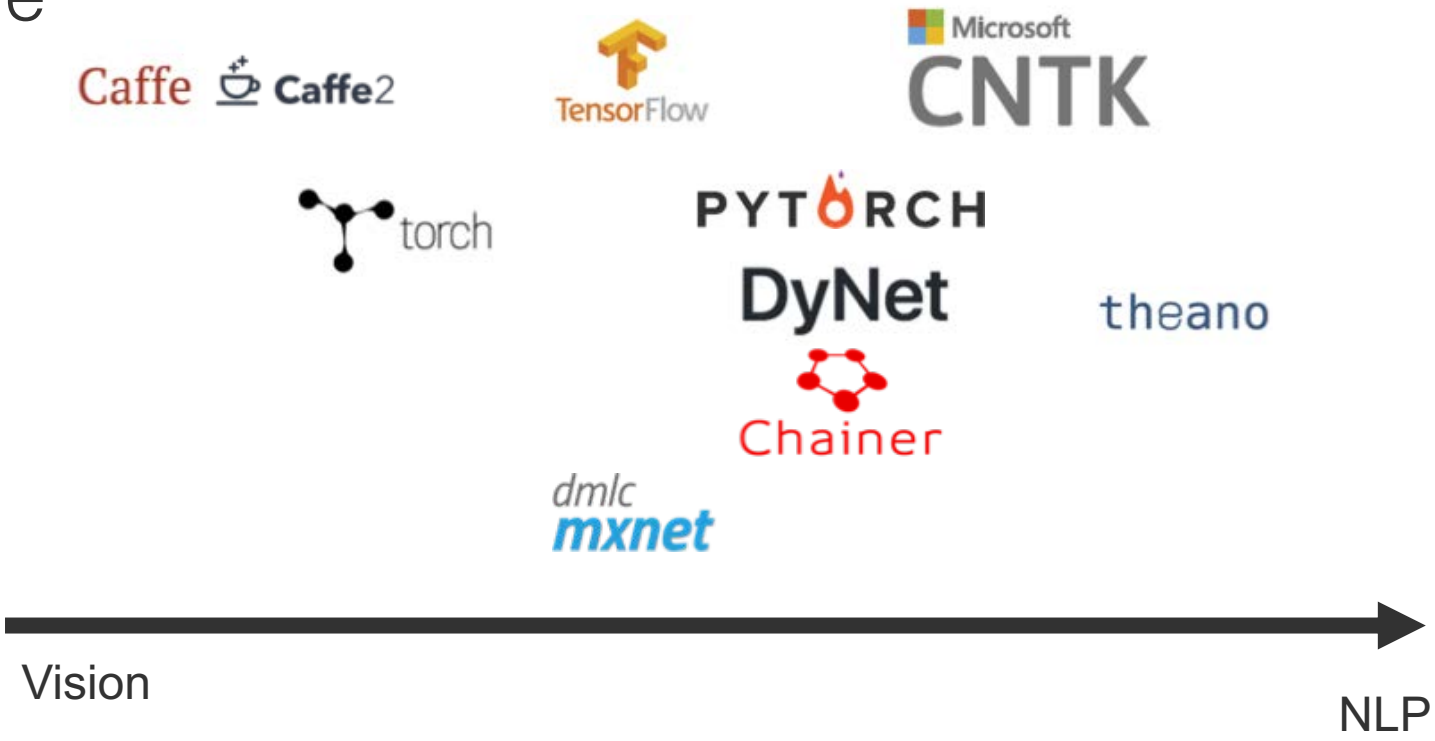
Auto-differential Libraries

- Auto-diff libraries automatically derive the gradients following the back-propagation rule.
- A lot of auto-differentiation libraries have been developed:
- So-called Deep Learning toolkits

Caffe Caffe2 TensorFlow Microsoft CNTK torch PYTORCH DyNet theano Chainer dmlc mxnet

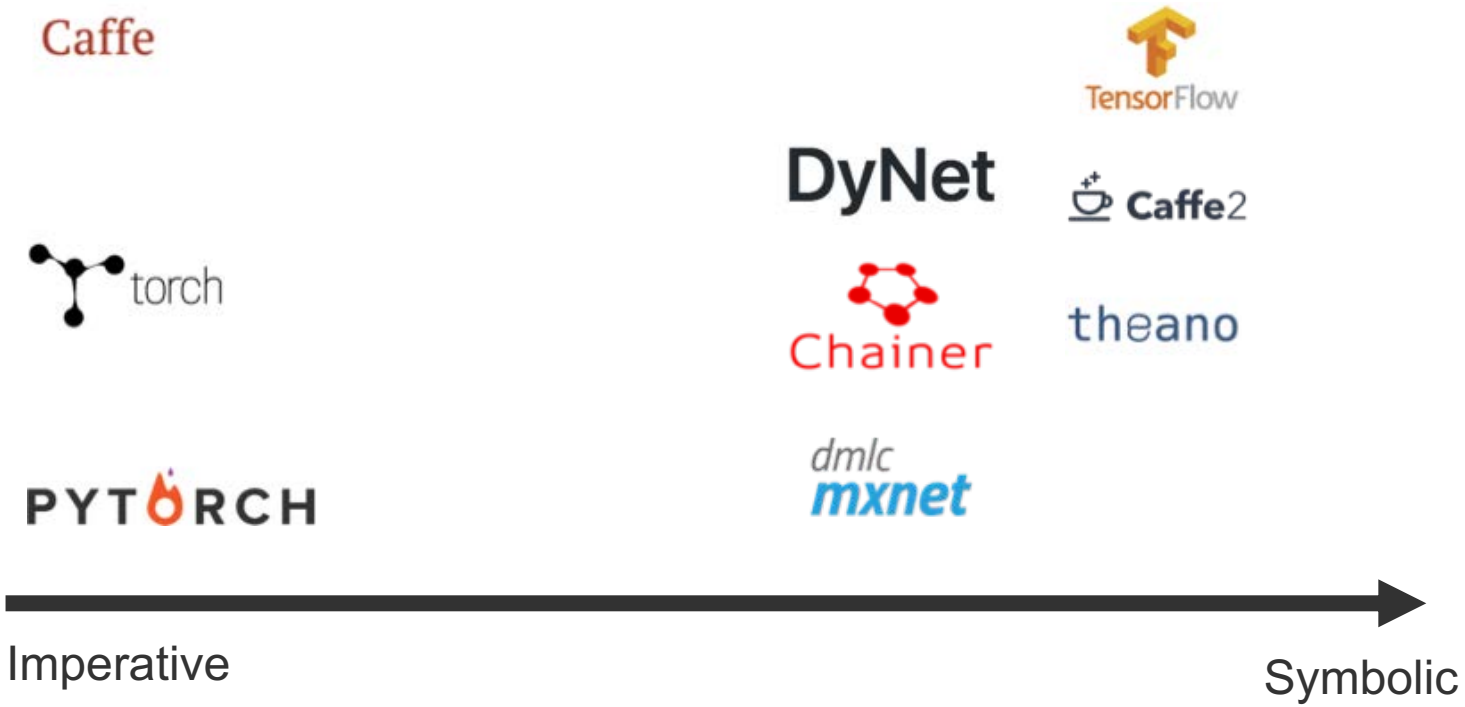
Deep Learning Toolkits

- They are adopted differently in different domains
- For example



Deep Learning Toolkits

- They are also designed differently
  - Symbolic vs. imperative programming



Deep Learning Toolkits: Symbolic and Imperative

- Symbolic vs. imperative programming
  - Symbolic: write symbols to assemble the networks first, evaluate later
  - Imperative: immediate evaluation

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```

Symbolic

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

Imperative

Deep Learning Toolkits

- Symbolic
  - Good
    - easy to optimize (e.g. batching, parallelization, kernel fusion) for developers
    - More efficient
  - Bad
    - The way of programming might be counter-intuitive
    - Harder to debug for user programs
    - Less flexible: you need to write symbols before actually doing anything
- Imperative:
  - Good
    - More flexible: write one line, evaluate/execute one line
    - Easy to program and easy to debug: because it matches the way we use in most programming languages
  - Bad
    - Less efficient
    - More difficult to optimize

Outline

- Deep Learning as Dataflow Graphs
- Auto-differential Libraries
- **Static and Dynamic Neural Networks**
- Dynamic Declaration and DyNet
- Cavs

Program an NN = Assemble a Dataflow Graph?

- Define a neural network
  - Define operations and layers: fully-connected? Convolution? Recurrent?
  - Define the data I/O: what data to read? Where?
  - Define a loss function/optimization objective: L2 loss? Softmax? Ranking Loss?
  - Define an optimization operator: SGD, Momentum, Adam, etc.
  - Connect operations, data I/O, loss functions and optimizer as a full dataflow graph, which is the representation of the neural network

Auto-differential Libraries then take over

- Build forward dataflow graph and backward gradient flow graphs automatically
- Perform training and apply updates

Summary: Static Declaration

- Users build a dataflow graph
- Frameworks analyze and optimize the graph
  - Automatically derive the backward graph
  - Incorporate some optimization if desired
- Perform training iteratively (via SGD)

Perform graph-level optimization over D (optionally)

```
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph 𝒟.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  batched computation: 𝒟({x_i^t}_{i=1}^K).
```

Summary: Static Declaration

- Static Declaration *seems to be* a dominant choice for programming deep learning models
  - Good for **static** workflows: define once, run for arbitrary batches/data
  - Easy to parallelize/batching for a fixed graph
  - Easy to optimize: a lot of off-the-shelf optimization techniques for graph

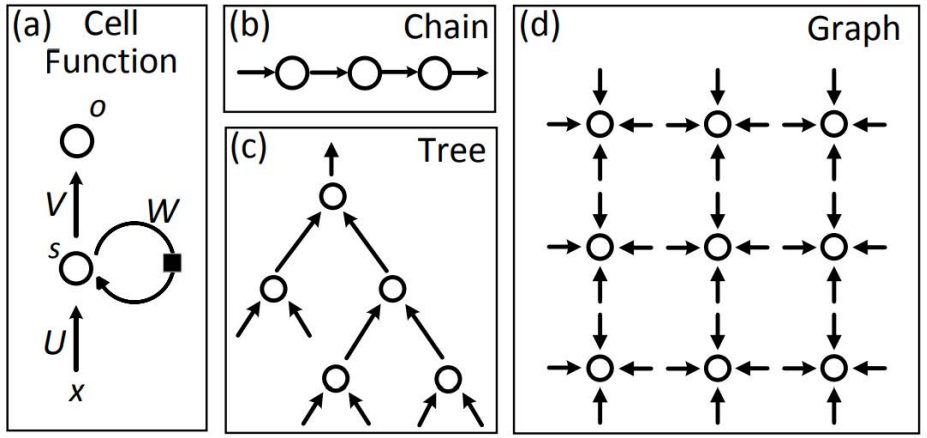
Static Declaration: Advantages and Assumptions

- The dataflow graph D only needs to be declared (and optimized) once, therefore
  - Ease of programming: users declare graph once, works for all samples
  - Constant graph construction/optimization overhead
  - All samples compute over one graph, therefore the computation can be "***by-nature" batched*** – by leveraging GPU and other advanced matrix-computing libs (CUDA, etc.)

```
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph 𝒟.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  batched computation: 𝒟({x_i^t}_{i=1}^K).
```

Introduction to Dynamic Neural Networks

- Deep Learning has been applied on more structured data
- The neural network compute following a data-dependent structure, in order to encode the structure information
  - Sequences, Trees, Graphs
- E.g. Recurrent Neural Networks and their variants
  - Sequence RNN in machine translation, video understanding
  - Tree RNN in sentence parsing and sentiment analysis
  - GraphRNN in social network/image segmentation



Dynamic Neural Networks

- The dynamics of the NNs come from multiple dimensions
  - Variably sized inputs
  - Variably sized outputs
    - e.g. in machine translation, input/output sentences have different length across samples
  - Variably structured inputs/outputs
    - e.g. in image segmentation/social network, input and output are differently structured graphs.
  - Nontrivial inference algorithms
    - e.g. in graph NNs, the inference path is dependent on the structure of the input data
- The NN architecture used to handle structured data would change with the input sample
  - i.e. *Dynamic Neural networks*

Static Declaration for Dynamic Dataflow Graphs

- Can we handle *dynamic* dataflow graphs using *static* declaration?
- Which sounds contradicting, but there do exist some techniques that only work for sequential data:
  - *Static unroll*: preprocessing all inputs to have the same length
  - *Bucketing*: put inputs into different buckets, one bucket one NN
  - *Dynamic unroll* (in TensorFlow): based on control flow operations
- Observations
  - At the core of the above tricks is to find a way to **batch the computation** in order to make the computation more efficient

Static Declaration for Dynamic Dataflow Graphs

- They are very commonly adopted, but are they good?
  - Unable to express structures beyond sequences
  - Usually yield unnecessary (extra) computation, which wastes computational resources
- More problems
  - Difficulty in expressing complex flow-control logic
  - Complexity of the computation graph implementation
  - Difficulty in debugging

Outline

- Deep Learning as Dataflow Graphs
- Auto-differential Libraries
- Static Declaration and Dynamic Neural Networks
- **Dynamic Declaration and DyNet**
- Cavs

An Extended Model: Dynamic Declaration

- From Static Declaration to Dynamic Declaration
- **Key idea**: declare and construct a dataflow graph *for each* input sample

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
      declare a data flow graph D_i^t for x_i^t.
      single-instance computation: D_i^t(x_i^t).
```

Dynamic Declaration

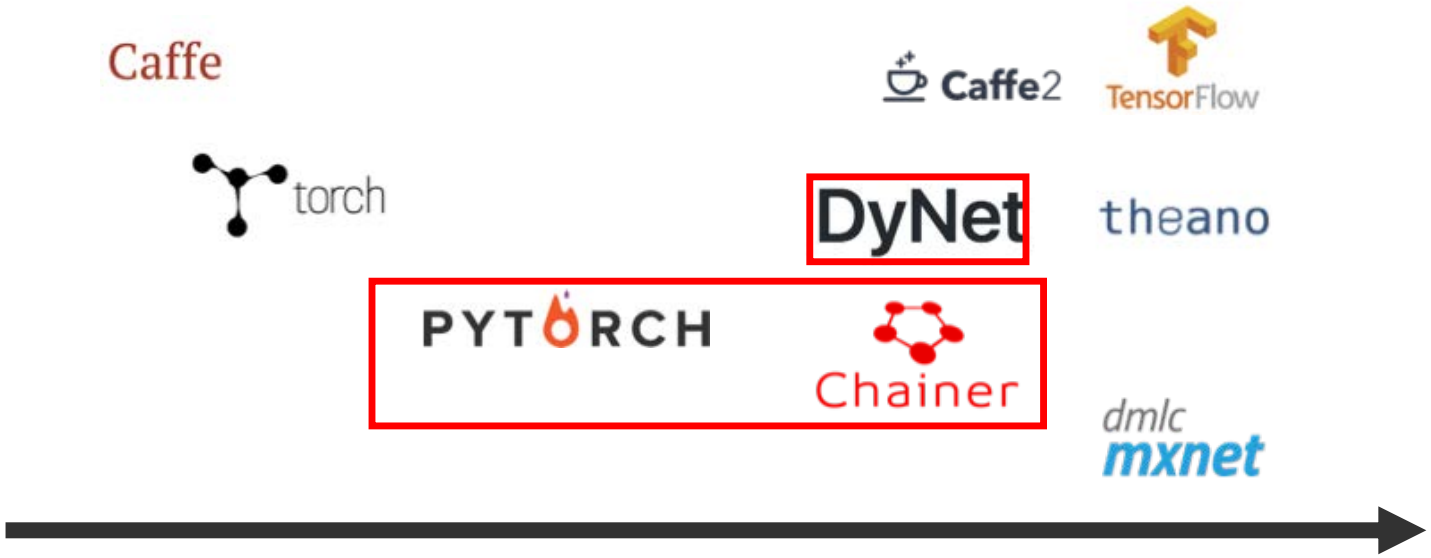
- Static Declaration vs. Dynamic Declaration
  - Move the graph declaration and construction (and optimization) from outside of the loop to inside the loop
  - Perform single instance training because it is hard to batch

```
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph 𝒟.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  batched computation: 𝒟({x_i^t}_{i=1}^K).
```

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
    declare a data flow graph 𝒟_i^t for x_i^t.
    single-instance computation: 𝒟_i^t(x_i^t).
```

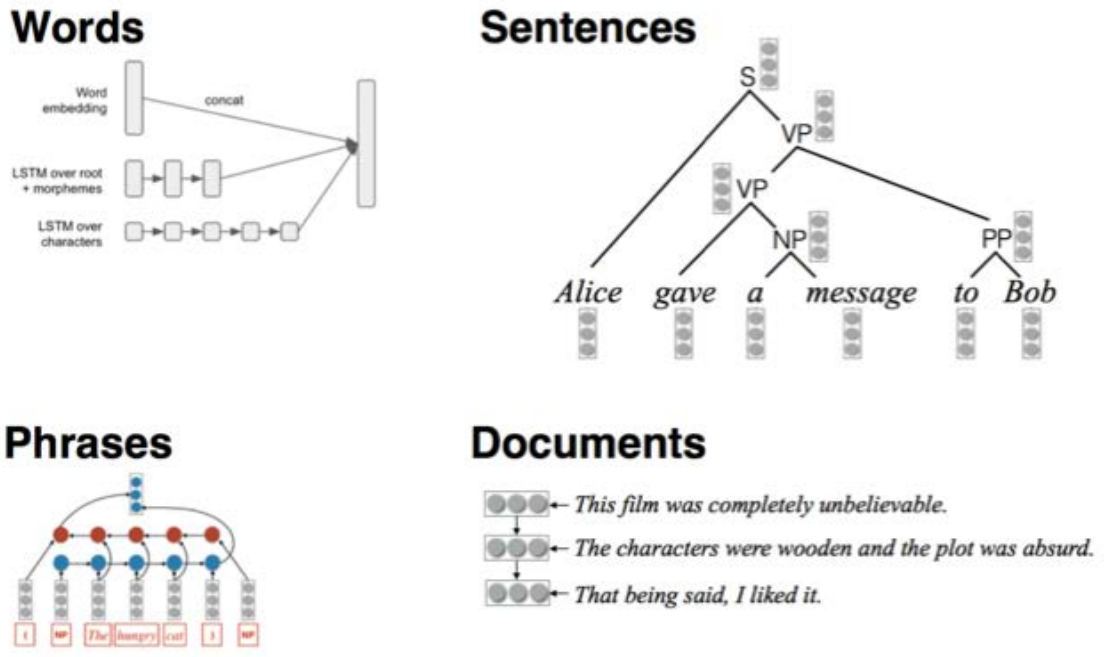
Dynamic Declaration: Existing Frameworks

- Dynamic Declaration
  - Imperative (instance evaluation)
    - Chainer (Preferred Networks Inc.), PyTorch (Facebook)
  - Symbolic (Lazy evaluation): **DyNet (Petuum Inc.)**



Introducing DyNet

- Designed for dynamic deep learning workflow, e.g.



DyNet is Optimized for Dynamic Declaration

- Key Ingredients
  - Separate parameter declaration and graph construction
  - Declare trainable parameters and construct models first
  - Construct computation graphs
  - Conclusion: Define parameter once, but define graphs dynamically depending on inputs → therefore making the graph construction lighter-weight

```
model = dy.Model()

pW = model.add_parameters((20,4))
pb = model.add_parameters(20)

dy.renew_cg()
x = dy.inputVector([1,2,3,4])
W = dy.parameter(pW) # convert params to expression
b = dy.parameter(pb) # and add to the graph

y = W * x + b
```

Key Ingredients in DyNet

- A visual comparison: implement a TreeRNN

```
class TreeRNNBuilder(object):
    def __init__(self, model, word_vocab, hdim):
        self.W = model.add_parameters((hdim, 2*hdim))
        self.E = model.add_lookup_parameters((len(word_vocab),hdim))
        self.w2i = word_vocab

    def encode(self, tree):
        if tree.isleaf():
          return self.E[self.w2i.get(tree.label,0)]
        elif len(tree.children) == 1: # unary node, skip
          expr = self.encode(tree.children[0])
          return expr
        else:
          assert(len(tree.children) == 2)
          e1 = self.encode(tree.children[0])
          e2 = self.encode(tree.children[1])
          W = dy.parameter(self.W)
          expr = dy.tanh(W*dy.concatenate([e1,e2]))
          return expr

model = dy.Model()
U_p = model.add_parameters((2,50))
tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
trainer = dy.AdamTrainer(model)
for epoch in xrange(10):
  for in_tree, out_label in read_examples():
    dy.renew_cg()
    U = dy.parameter(U_p)
    loss = dy.pickneglogsoftmax(U*tree_builder.encode(in_tree), out_label)
    loss.forward()
    loss.backward()
    trainer.update()
```

DyNet TreeLSTM (30 LoC)

TensorFlow TreeLSTM (200 LoC)

Outline

- Deep Learning as Dataflow Graphs
- Auto-differential Libraries
- Static Declaration and Dynamic Neural Networks
- Dynamic Declaration and DyNet
- **Cavs**

Dynamic Declaration: Advantages

- Dynamic declaration has one major advantages
  - Flexibility: it can express ***arbitrarily dynamically structured*** networks by declaring as many as dataflow graphs as the number of training data

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
      declare a data flow graph D_i^t for x_i^t.
      single-instance computation: D_i^t(x_i^t).
```

Dynamic Declaration: Problems

- Dynamic declaration scarifies efficiency for flexibility
  - Graph construction overhead grows linearly with # of samples
  - Only single-instance computation can be performed – no batching!
  - Hard to incorporate graph-level optimization

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
      declare a data flow graph D_i^t for x_i^t.
      single-instance computation: D_i^t(x_i^t).
```

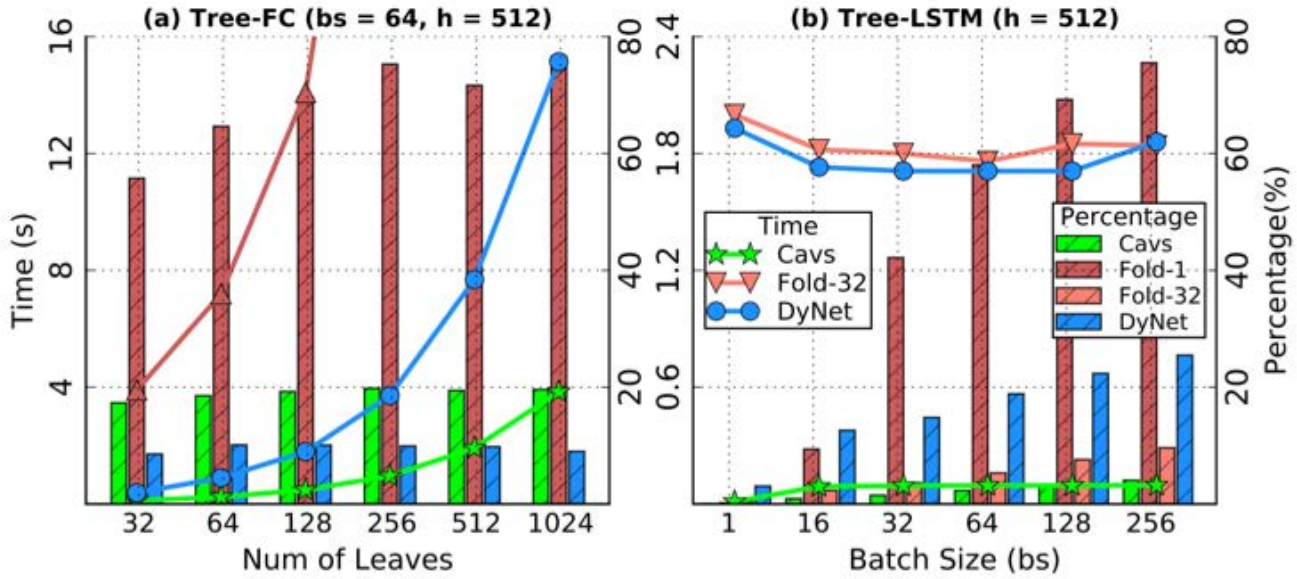
Problem #1: Graph Construction and Optimization

```
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph 𝒟.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  batched computation: 𝒟({x_i^t}_{i=1}^K).
```

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
      declare a data flow graph 𝒟_i^t for x_i^t.
      single-instance computation: 𝒟_i^t(x_i^t).
```

Problem #1: Graph Construction and Optimization

- Graph construction literally takes 80% of time in TensorFlow Fold
- Curve (left axis): absolute time; bar (right): percentage time



Problem #2: Batching will be Very Hard

```
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph 𝒟.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  batched computation: 𝒟({x_i^t}_{i=1}^K).
```

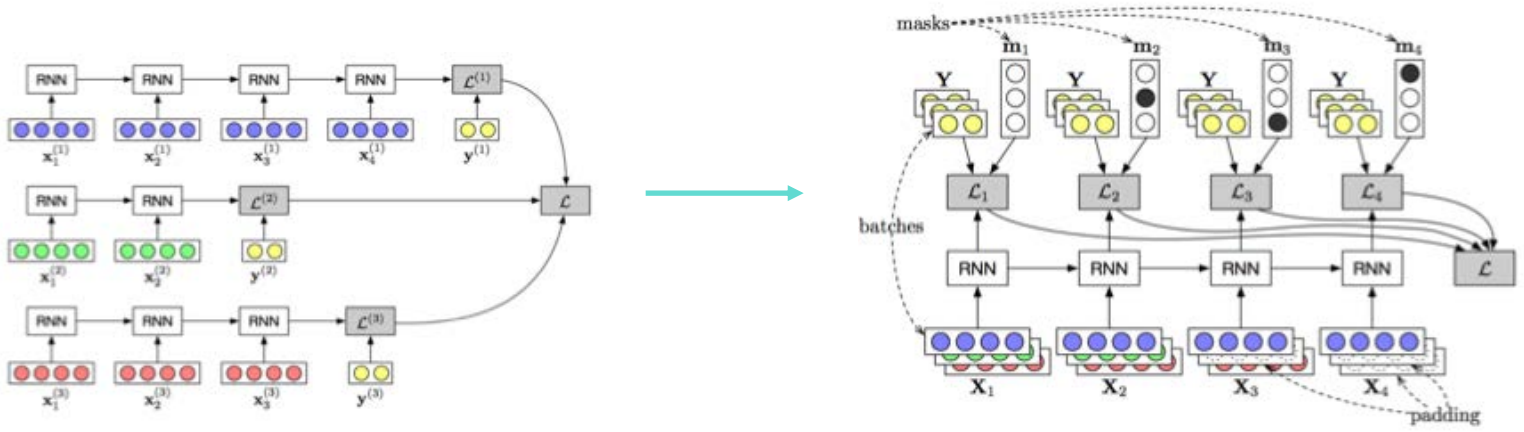
In static declaration: batching is natural

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
     declare a data flow graph 𝒟_i^t for x_i^t.
     single-instance computation: 𝒟_i^t(x_i^t).
```

In dynamic declaration: batching is difficult

Problem #2: Batching will be Very Hard

- Manual batching of dynamic graphs?
- Users have to write code to do batching by themselves
  - In fact, until 2017, the famous tree-LSTM (a typical dynamic NN) model is trained with batchsize=1
- And it is not always available!



Problem #3: Unavailable to Graph Optimizations

- Graph optimization itself has a overhead (growing with the size of the graph)
- In static declaration, we optimize the graph only once, and apply for all input data points – graph optimization overhead is constant, and the gain is mostly positive
- In dynamic declaration, if we want to incorporate these optimization, we need to optimize for each declared graph
- Therefore, linear graph optimization overhead
- As a result: the optimization overhead might cost more than gained

```
/* (b) dynamic declaration */
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  for k = 1 → K:
     declare a data flow graph D_i^t for x_i^t.
     single-instance computation: D_i^t(x_i^t).
```

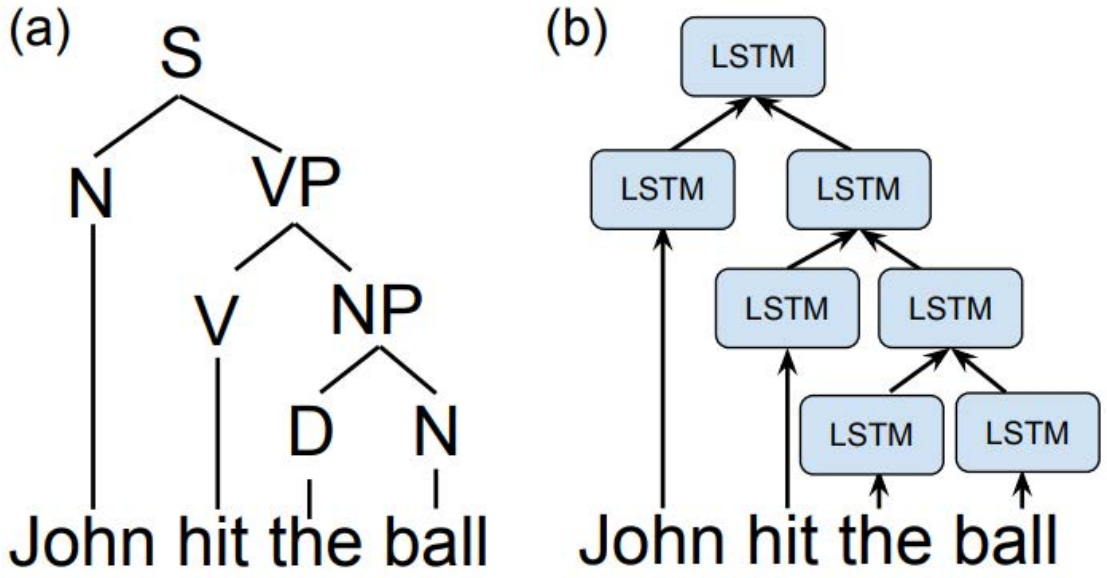
Graph optimization happens here: inside the loops!

Introducing Cavs: Design Goals

- Simple Interface, rich expressiveness
  - Keep the flexibility of symbolic programming and dynamic declaration
- At the same time, address the three aforementioned problems:
  - Minimize graph construction overhead
  - Allow for efficient computation and batching
  - (Re-)open the opportunities for graph optimization techniques

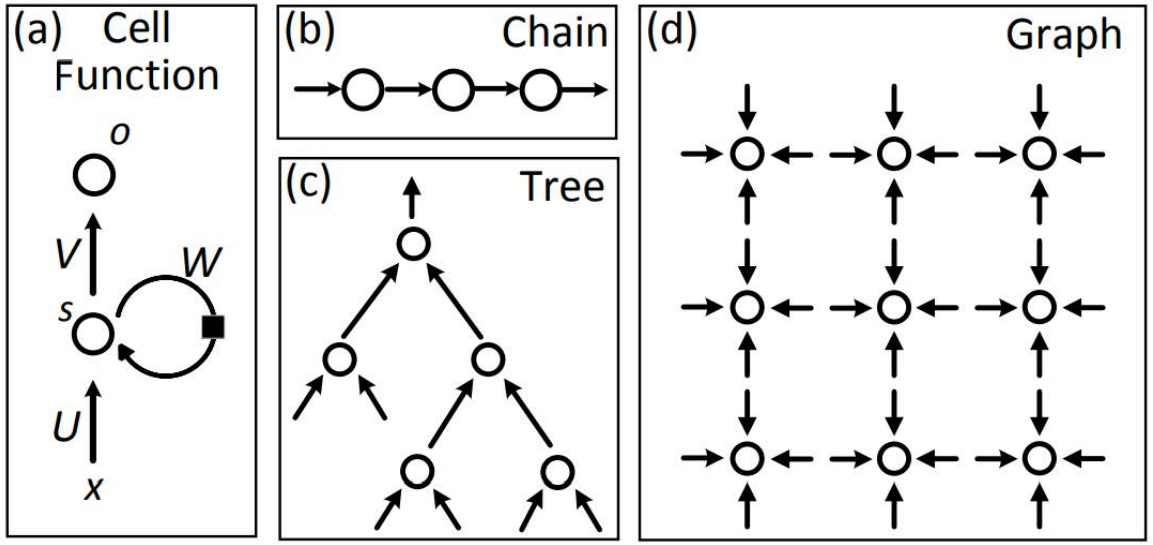
Cavs: Motivation

- An example of a dynamic NN
  - (a) a constituency parsing tree
  - (b) the corresponding Tree-LSTM network.
  - We use the following abbreviations in (a): S for sentence, N for noun, VP for verb phrase, NP for noun phrase, D for determiner, and V for verb.



Cavs: Motivation

- Observation: Most dynamic NNs has *recurrent/recursive* structures
  - The dynamics come from the sample-dependent structure
  - Not the ”neural network” itself

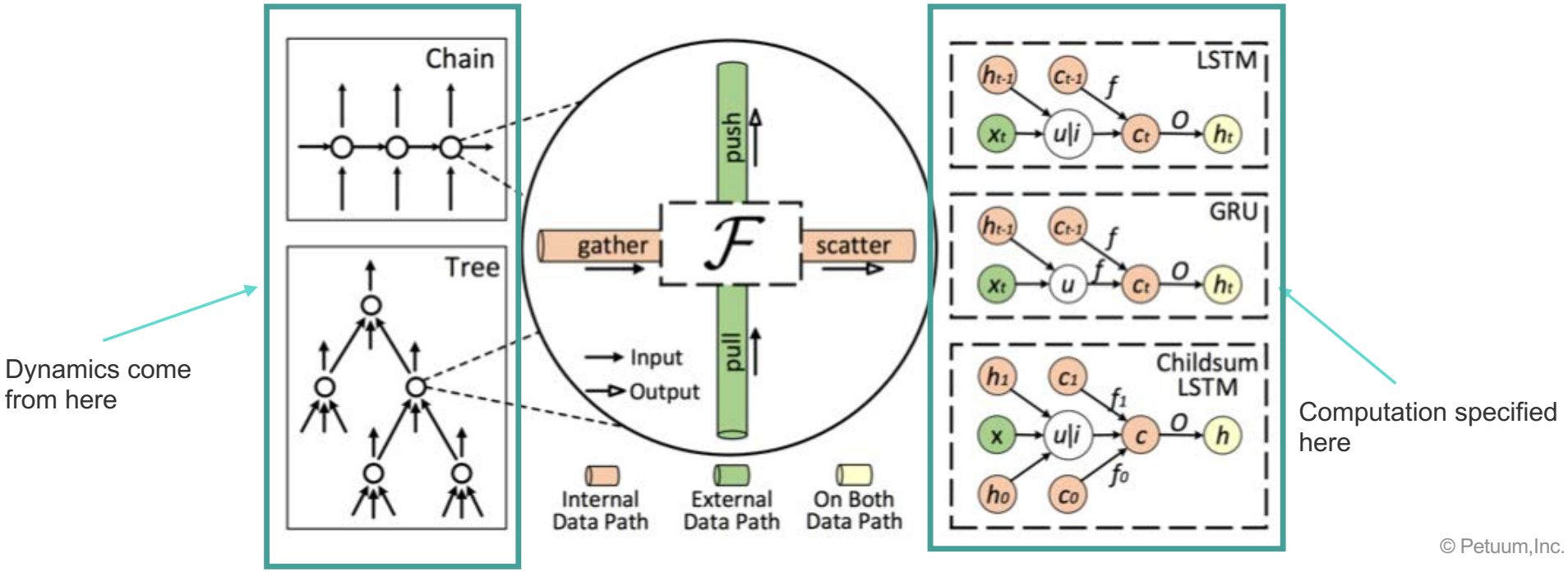


Think Like a Vertex

- "Think like a vertex"
  - Originally from the graph computing community
- Key idea: ***express "global" through "local"***
  - User implements a vertex function, specifying how a node will interact with its neighboring nodes
  - The system will compile the local vertex function and figure out the overall computing pattern over the whole graph

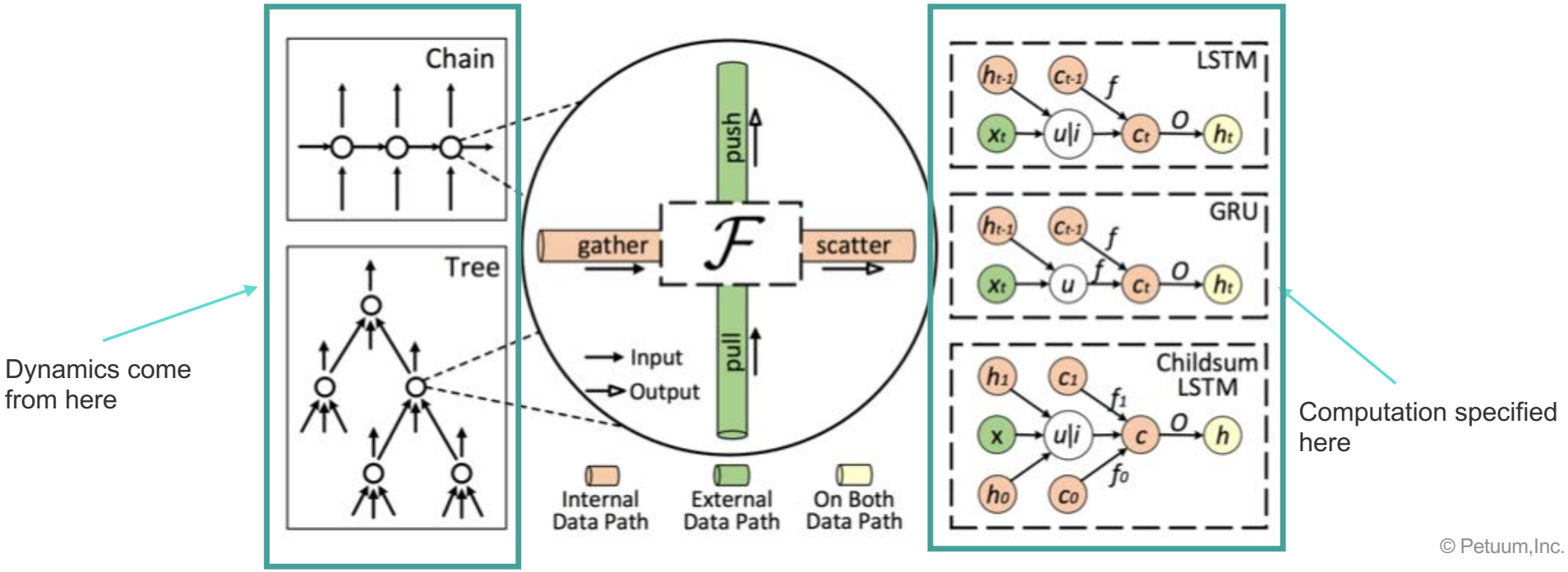
Cavs: A Vertex-centric Representation

- Cavs introduces vertex-centric representation for DL, and decompose a dynamic NN as two modules
  - A vertex function F, which is static;
  - An input graph G, which is data-dependent and dynamic;



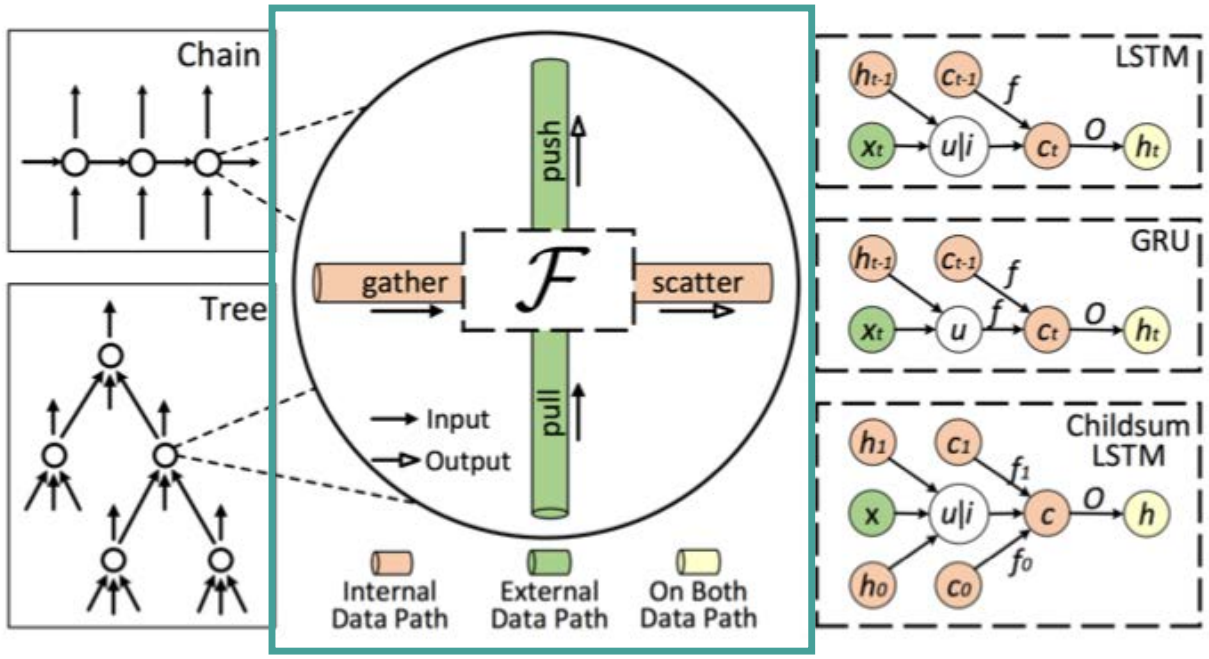
Cavs: A Vertex-centric Representation

- The key idea: separate out static ML model declaration from the data-dependent dynamics of input samples



Cavs: Four APIs

- *Gather & Scatter* for internal data path
- *Pull & Push* for external data path



Cavs: Four APIs

- An example: expressing Tree-LSTM using the four APIs

```
def F():
  S = gather()                    # gather states of child vertices
  for k in range(N):
    c_k, h_k = split(S[k], 2)     # get hidden states c and h
  x = pull({0})                   # pull the first external input x

  # specify the computation
  h = sum_{k=0}^{N-1} h_k
  i = sigmoid(W^(i) × x + U^(i) × h + b^(i))
  for k in range(N):
    f_k = sigmoid(W^(f) × x + U^(f) × h_k + b^(f))
  o = sigmoid(W^(o) × x + U^(o) × h + b^(o))
  u = tanh(W^(u) × x + U^(u) × h + b^(u))
  c = i ⊗ u + sum_{k=0}^{N-1} f_k ⊗ c_k
  h = o ⊗ tanh(c)

  scatter(concat([c, h], 1))      # scatter c, h to parent vertices
  push(h)                         # push to external connectors
```

Expressing Backpropagation

- It transforms the forward-backward computation on batch of dataflow graphs as
  - Forward: schedule the execution of the vertex function F through a batch of input graphs
  - Backward: schedule the execution of ∂F through the same batch of input graphs, in a reverse order

```
/* (c) our proposed vertex-centric model */
declare a symbolic vertex function F.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  read their associated graphs {G_i^t}_{i=1}^K.
  compute F over {G_i^t}_{i=1}^K with inputs {x_i^t}_{i=1}^K.
```

Cavs Bypass Graph Construction Overhead

- No repeated graph construction overhead!
  - The graph construction overhead is constant – we only need to construction F, which is usually a small-scale dataflow graph
  - Bypass the repeated dataflow graph construction
  - Instead, read the input graph G, which could be achieved by an I/O function

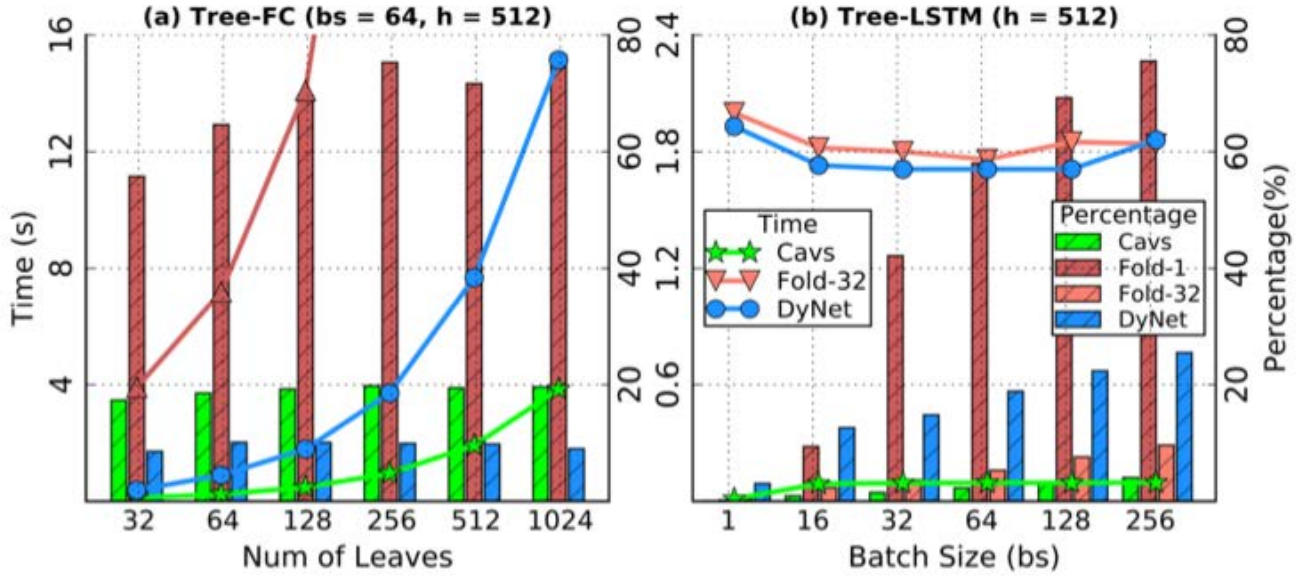
Declare only once → constant graph construction cost

```
/* (c) our proposed vertex-centric model */
declare a symbolic vertex function F.
for t = 1 → T:
  read the tth data batch {x_i^t}_{i=1}^K.
  read their associated graphs {G_i^t}_{i=1}^K.
  compute F over {G_i^t}_{i=1}^K with inputs {x_i^t}_{i=1}^K.
```

Read through I/O, no graph construction involved any more.

Empirical Results: Graph Construction Cost

- Cavs has constant graph construction time
- Curve (left axis): absolute time; bar (right): percentage time
- Cavs outperforms TensorFlow-Fold by a large margin



Cavs Enables Batched Computation

- Batched computation is key to deep learning
- Recall the Dynamic Declaration problem #2
- Batched computation on dynamic graphs are difficult:
  - Difficult to find batching opportunities
    - need to analyze the graph
    - need to manually write code to do extra processing
  - Given batching opportunities, different to re-order memory contents
    - For the batched computation to be efficient, their input/output need to coalesce on memory
    - How to efficiently re-arrange memory layout to guarantee continuity?

Cavs Enables Batched Computation

- Batched computation is natural and automatic in Cavs
  - Cavs transforms the backpropagation as **evaluating F on nodes of a batch of input graphs**, see the figure below.

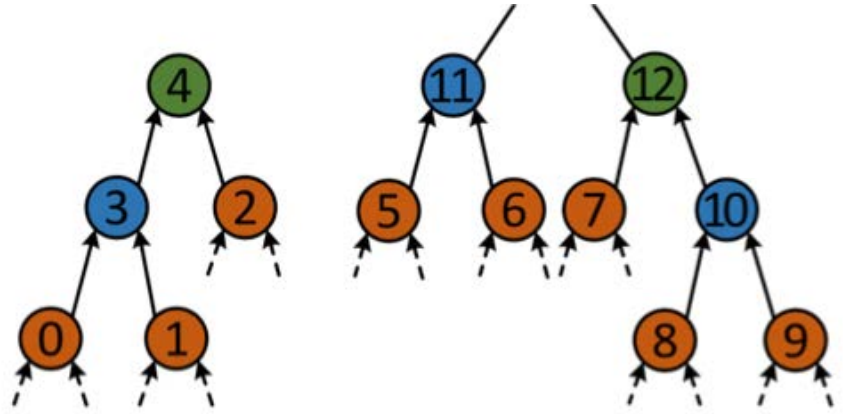
- Then, batched computation can be realized by
  - Figure out a set of vertices that we are ready to evaluate F on
  - Batch the evaluation of F on this set of vertices
  - Pass the output of F to their parent vertices

Cavs Enables Batched Computation

- A simple policy enables efficient batched computation
  - Vertices with same colors are batched evaluated.

Algorithm 1 Backpropagation with the batching policy.

function FORWARD($\{x_k\}_{k=1}^K, \{\mathcal{G}_k\}_{k=1}^K, \mathcal{F}$)
  set task ID $t \leftarrow 0$, task stack $S \leftarrow \emptyset$.
  while NOT all vertices in $\{\mathcal{G}_k\}_{k=1}^K$ are *evaluated* **do**
    figure out all activated vertices in $\{\mathcal{G}_k\}_{k=1}^K$ as a set V_t.
    push V_t into $\mathcal{S}$.
    evaluate $\mathcal{F}$ on V_t: `GraphExecute`$(V_t, \mathcal{F})$ (see §3.5).
    set the status of all vertices in V_t as *evaluated*.
    set $t \leftarrow t + 1$.
  end while
  return $\mathcal{S}$.
end function
function BACKWARD($\mathcal{S}, \{\mathcal{G}_k\}_{k=1}^K, \partial\mathcal{F}$)
  set t as the size of $\mathcal{S}$.
  while $\mathcal{S}$ is not empty **do**
    pop the top element of $\mathcal{S}$ as V_t.
    Evaluate $\partial\mathcal{F}$ on V_t: `GraphExecute`$(V_t, \partial\mathcal{F})$ (§3.5).
    set $t \leftarrow t - 1$.
  end while
end function



Tree 1 Tree 2

Dynamic Batching: Memory Management Challenge

- Batched computational kernels on CPU/CPUs requires the inputs to a batched computation kernel locate continuously on memory
  - In Dynamic Declaration, this is usually not the case due to the dynamic-varying input structures.
  - To achieve memory continuity, one has to frequently re-arrange memory layouts (memcpy) of the inputs to each batched operation.
- Cavs proposes a new data structure, DynamicTensor, to **ensure memory continuity**, at the same time **minimize memory movement overhead**

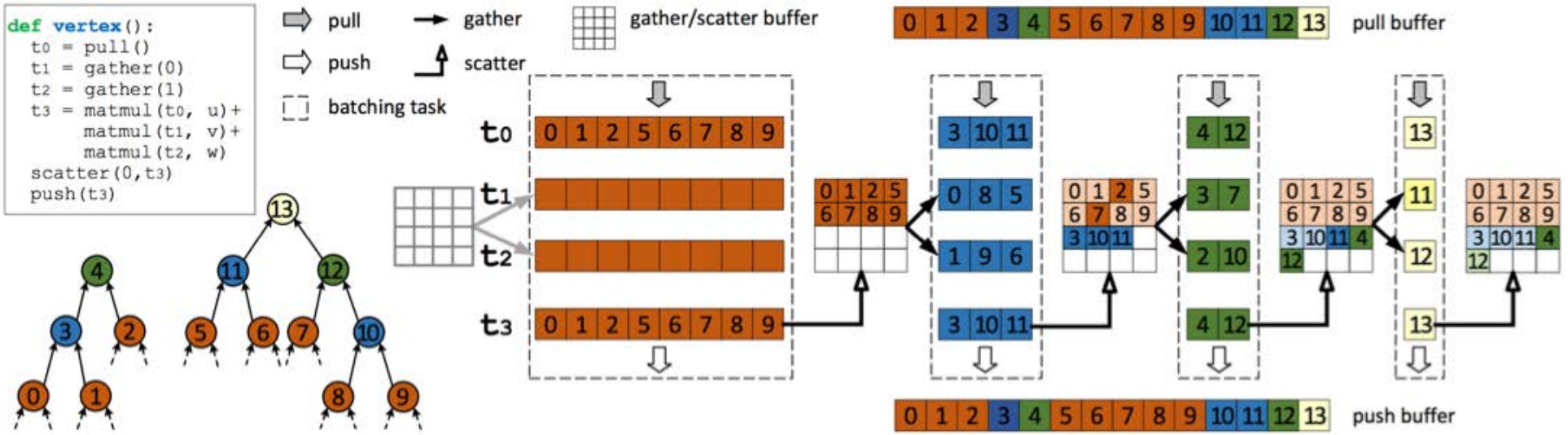
Cavs: Dynamic Tensor

- Cavs proposes a new data structure, DynamicTensor, to ensure memory continuity.
  - **shape**: tensorshape except the batch dimension
  - **bs**: a placeholder for batch size, determined and filled at runtime
  - **offset**: records where to read from
  - **p:** pointer to the allocated memory, which can be extended whenever needed.
- Hence, dynamic tensor is more flexible for dynamically-varying batch size

```
struct DynamicTensor {
  vector<int> shape;
  int bs;
  int offset;
  void* p; };
```

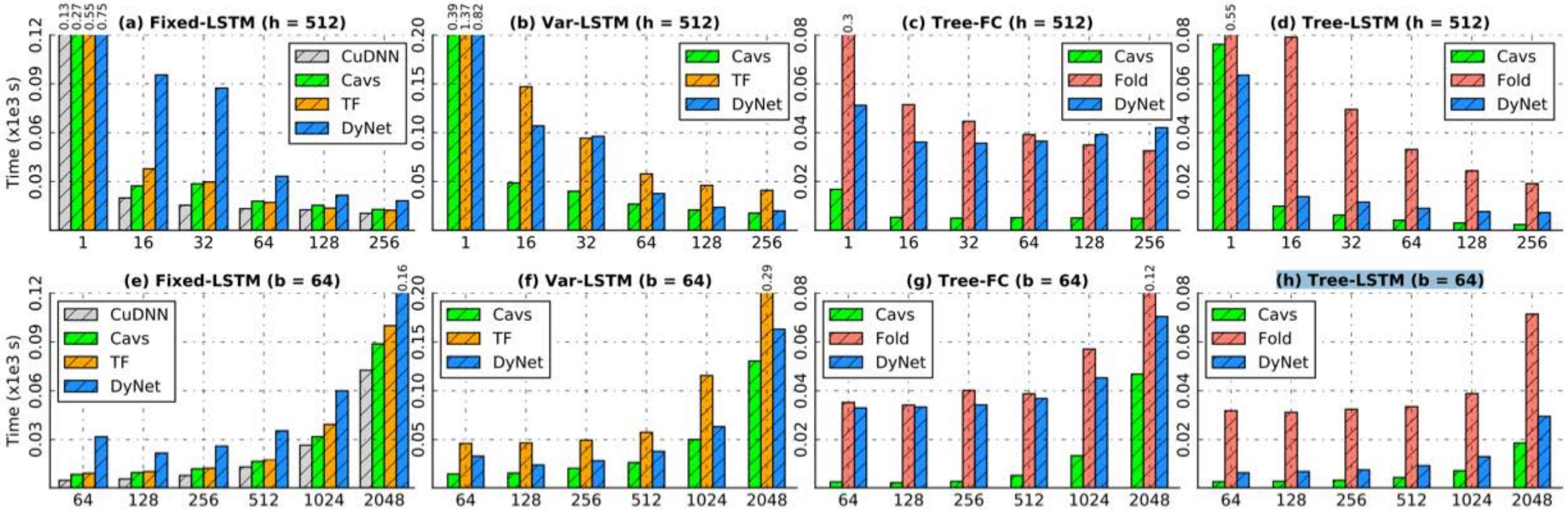
Cavs: Advanced Memory Management – Dynamic Tensor

- With dynamic tensors, Cavs designs a memory management mechanism to guarantee the coalesce of input contents of batched operations on memory



Overall Performance

- Overall, Cavs is 1 – 2 orders of magnitude faster than state-of-the-art systems such as DyNet and TensorFlow-Fold.



Cavs: Improvement on Computation

- In terms of compution-only, Cavs shows maximally 5.4x/9.7x and 7.2x/2.4x speedups over Fold/DyNet on Tree-FC and Tree-LSTM, respectively.

# leaves	time (s)	Speedup	*bs*	time (s)	Speedup
32	**0.6** / 3.1 / 4.1	**5.4** / 7.1	1	76 / 550 / **62**	**7.2** / 0.8
64	**1.1** / 3.9 / 8.0	3.7 / 7.5	16	**9.8** / 69 / 12	7.0 / 1.2
128	**2** / 6.2 / 16	3.0 / 7.9	32	**6.2** / 43 / 9.9	7.0 / 1.6
256	**4** / 10.6 / 33.7	2.7 / 8.7	64	**4.1** / 29 / 7.4	**7.2** / 1.8
512	**8** / 18.5 / 70.6	2.3 / 8.9	128	**2.9** / 20.5 / 5.9	7.1 / 2.0
1024	**16** / 32 / 153	2.1 / **9.7**	256	**2.3** / 15.8 / 5.4	7.0 / **2.4**

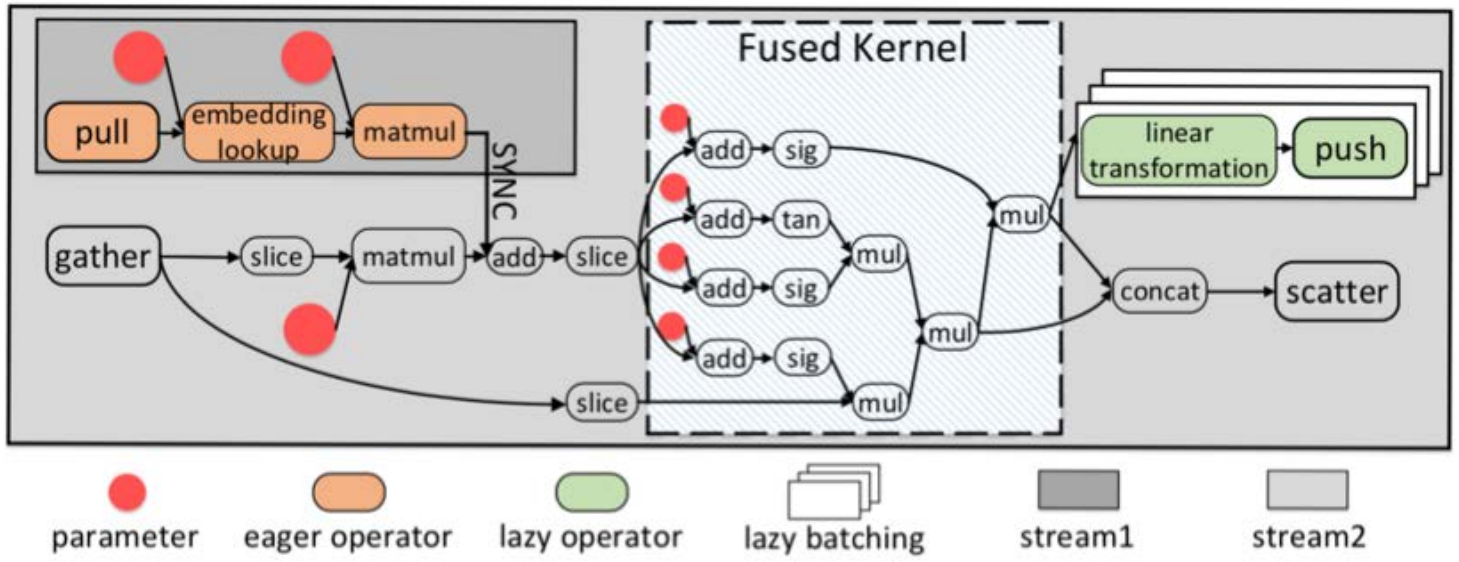
Cavs: Improvement on Memory Management

- The improvement is significant (2x - 3x) at larger batch size, c comparing to DyNet.

bs	Memory operations (s) (*Cavs / DyNet*)		Computation (s) (*Cavs / DyNet*)	
	Train	Inference	Train	Inference
16	**1.14** / 1.33	**0.6** / 1.33	**9.8** / 12	**2.9** / 8.53
32	**0.67** / 0.87	**0.35** / 0.87	**6.1** / 9.8	**1.9** / 5.35
64	**0.39** / 0.6	**0.21** / 0.6	**4.0** / 7.4	**1.3** / 3.48
128	**0.25** / 0.44	**0.13** / 0.44	**2.9** / 5.9	**0.97** / 2.52
256	**0.17** / 0.44	**0.09** / 0.44	**2.3** / 5.4	**0.77** / 2.58

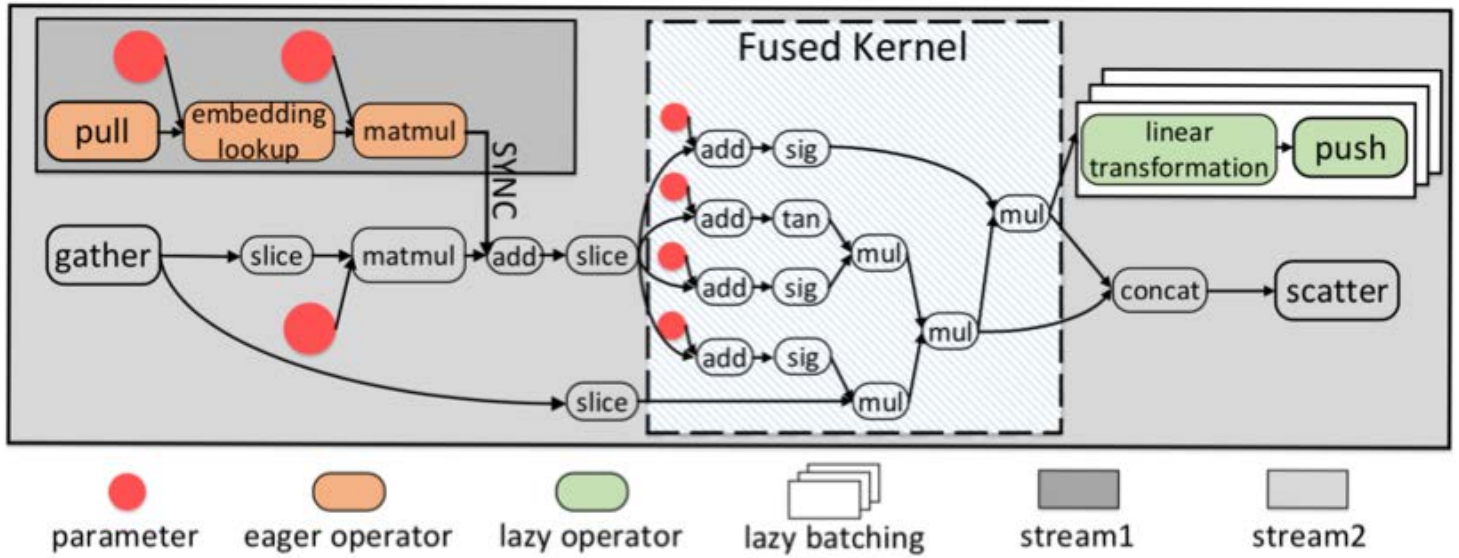
Cavs is Open to Graph Optimization

- Incorporating graph-level optimization in Cavs is the same as it in static declaration
  - Optimize the static vertex function F
  - F will be evaluated at each vertex of the input structure
  - Optimize once, benefit elsewhere



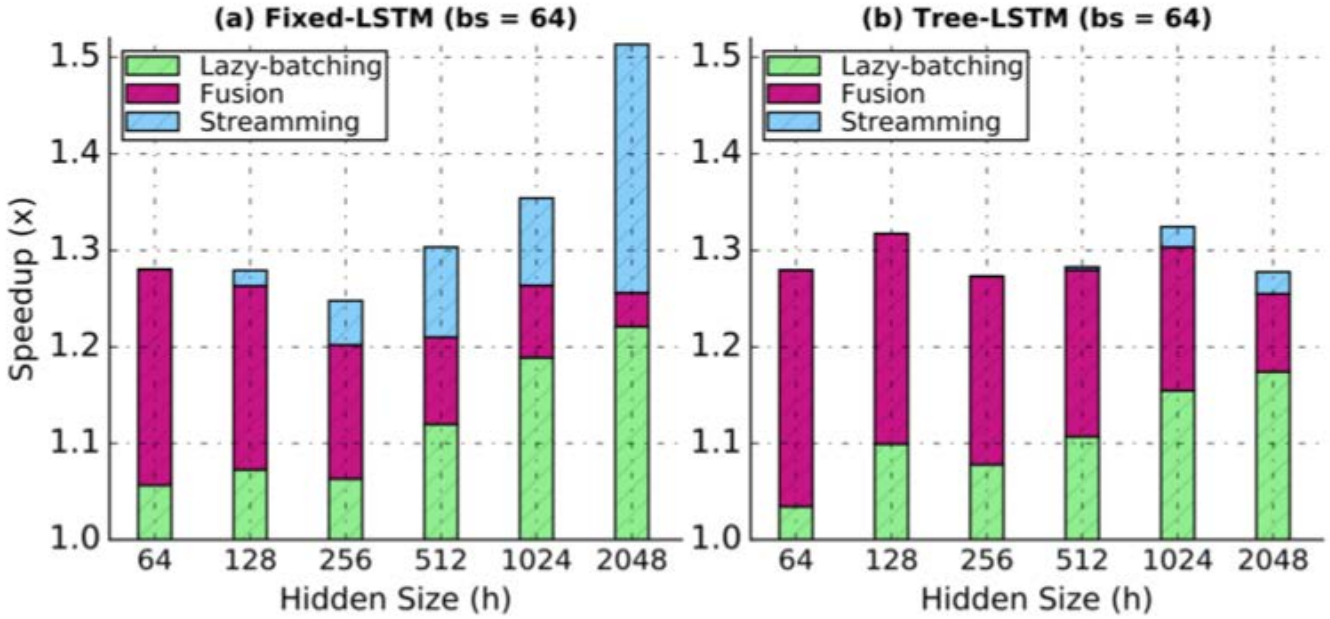
Cavs Exposes Opportunities for Graph Optimization

- Cavs propose three graph-level optimization strategies
  - Lazy batching
  - Streaming
  - Automatic kernel fusion



How Important is Graph Optimization?

- Together they result in another 1.5 times speedup



Summary: Comparing Software Frameworks for Dynamic Neural Networks

Model	Frameworks	Expressiveness	Batching	Graph Cons. Overhead	Graph Exec. Optimization
static declaration	Caffe, Theano, TensorFlow, MxNet	×	×	low	beneficial
dynamic declaration (instant evaluation)	PyTorch, Chainer	√	×	N/A	unavailable
dynamic declaration (lazy evaluation)	DyNet	√	√	high	not beneficial
Fold	TensorFlow-Fold	√	√	high	unknown
Vertex-centric	Cavs	√	√	low	beneficial

Cavs will be in

- More descriptions about Cavs:

Cavs: A Vertex-centric Programming Interface for Dynamic Neural Networks. *Hao Zhang, Shizhen Xu, Graham Neubig, Wei Dai, Qirong Ho, Guangwen Yang, Eric P. Xing. ATC 2018.*

- Code will be released soon, check out at: https://github.com/petuum-inc

PETUUM

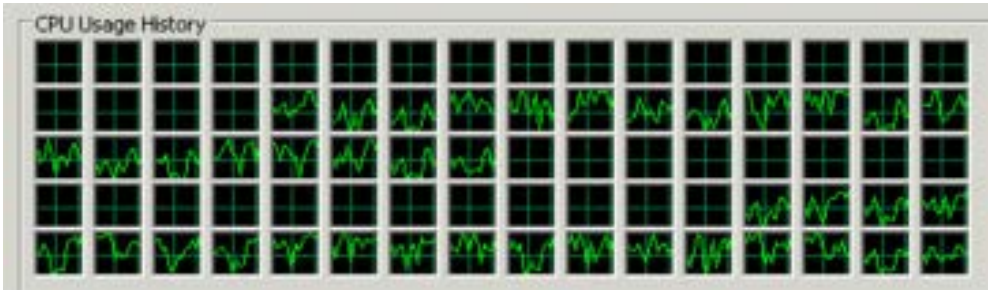
Part-III (2)
Distributed Deep Learning

Outline

- Overview: Distributed Deep Learning on GPUs
- Challenges 1: Addressing the communication bottleneck
- Challenges 2: Handling the limited GPU memory

Review – DL toolkits on single machine

- Using GPU is a must
  - A small number of GPU-equipped machines could achieve satisfactory speedup compared to CPU clusters with thousands of cores

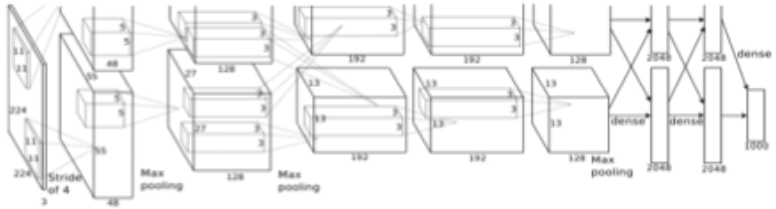


- A cluster of 8 GPU-equipped machines
- A cluster of 2000 CPU cores

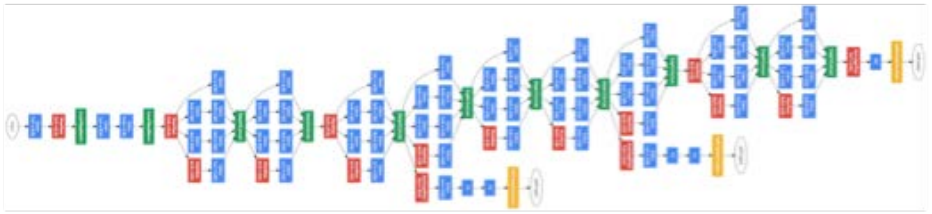
More readily available to researchers

Review – DL toolkits on single machine

- However, using a single GPU is far from sufficient
  - average-sized deep networks can take days to train on a single GPU when faced with 100s of GBs to TBs of data
  - Demand faster training of neural networks on ever-larger datasets



AlexNet, 5 – 7 days



GoogLeNet, 10+ days

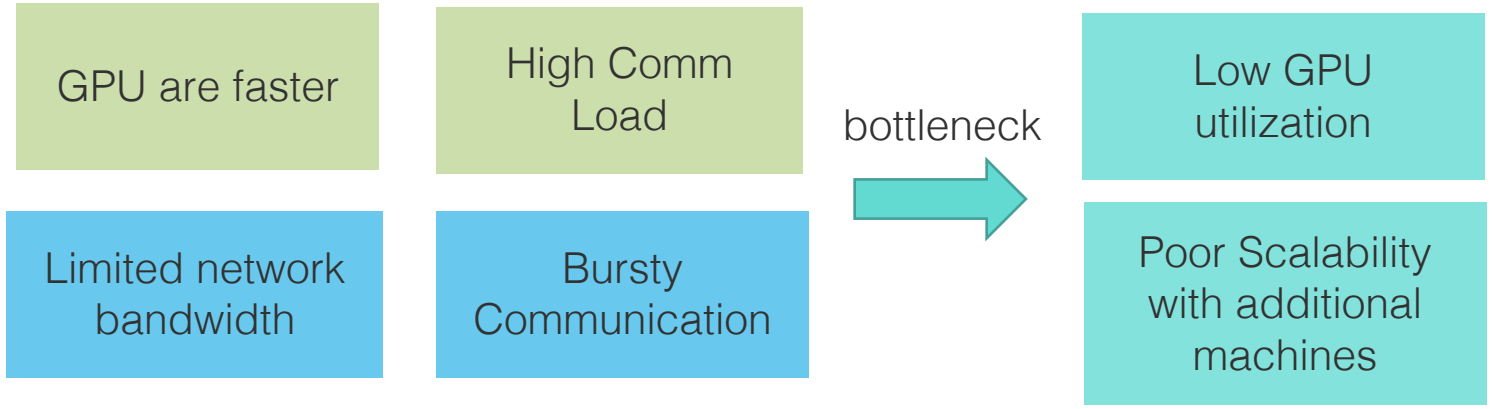
- However, current distributed DL implementations (e.g. in TensorFlow) can scale poorly due to substantial parameter synchronization over the network (we will show later)

Outline

- Overview: Distributed Deep Learning on GPUs
- **Challenges 1: Addressing the communication bottleneck**
- Challenges 2: Handling the limited GPU memory

Challenges

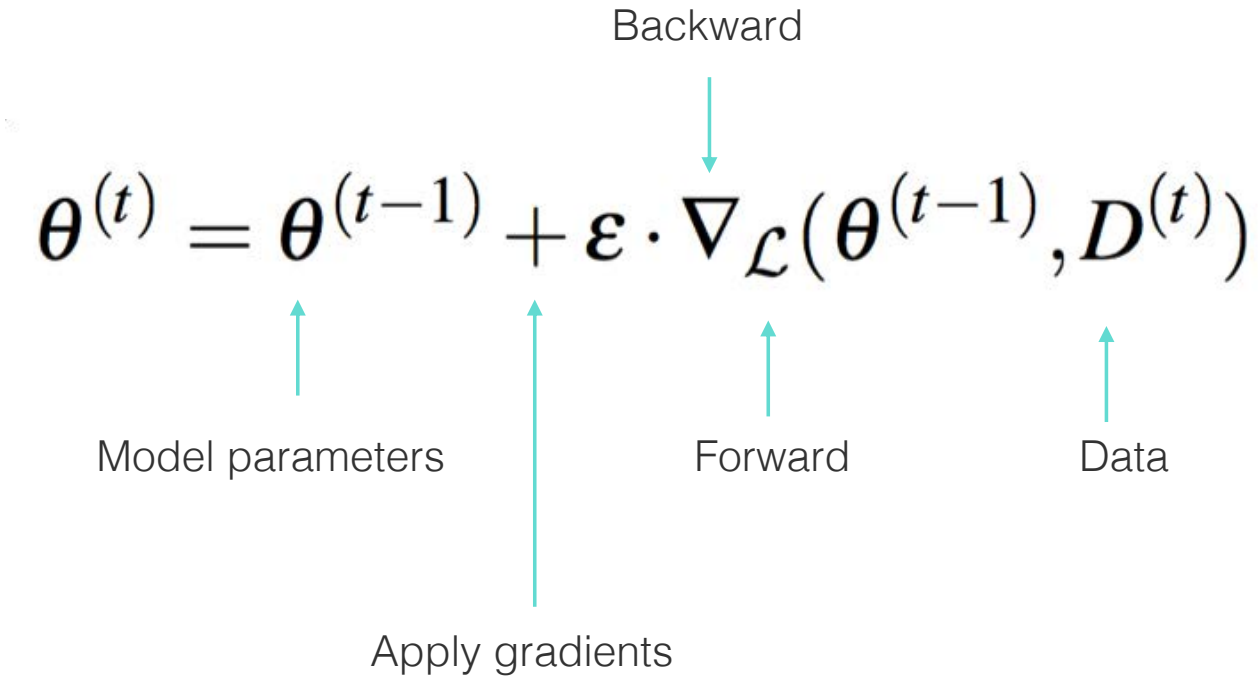
- Communication challenges
  - GPUs are at least one order of magnitude faster than CPUs



  - High communication load raises the network communication as the main bottleneck given limited bandwidth of commodity Ethernet
  - Managing the computation and communication in a distributed GPU cluster often complicates the algorithm design

Let's see what causes the problem

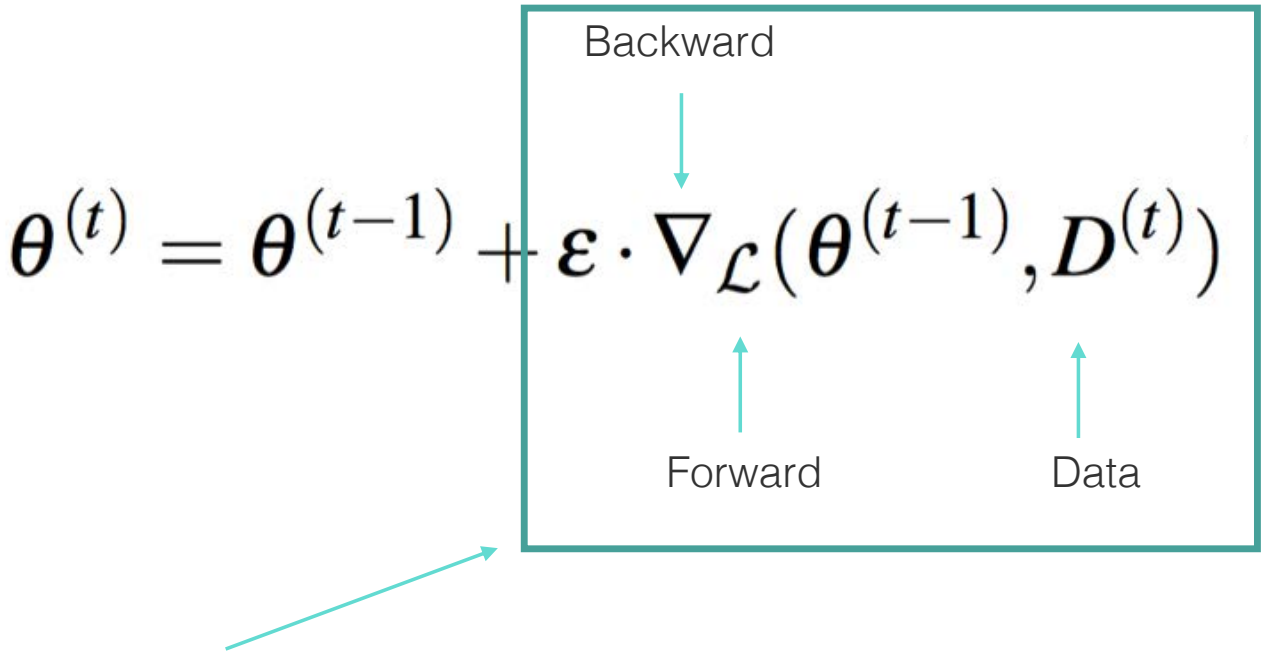
- Deep Learning on a single node – an iterative-convergent formulation



Zhang et al., 2017

Let's see what causes the problem

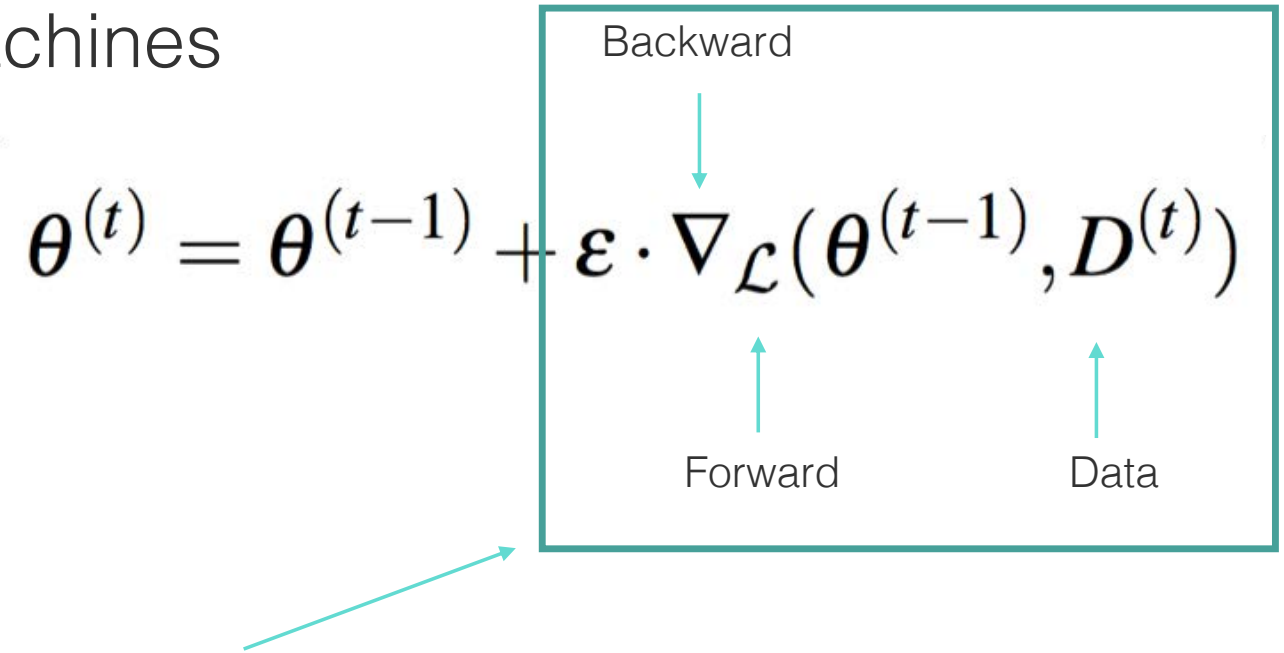
- Deep Learning on a single node – an iterative-convergent formulation



Forward and backward are the main computation (99%) workload of deep learning programs.

Distributed Deep Learning

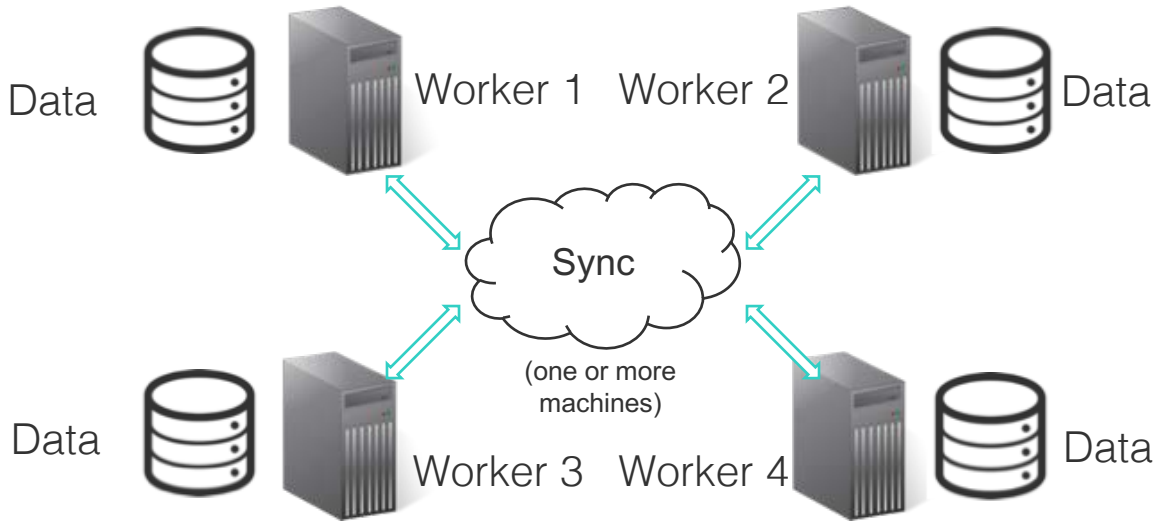
- Distributed DL: parallelize DL training using multiple machines.
- i.e. we want to accelerate the heaviest workload (in the box) to multiple machines



Forward and backward are the main computation (99%) workload of deep learning programs.

Data parallelism with stochastic gradient descent

- We usually seek a parallelization strategy called data parallelism, based on SGD
  - We partition data into different parts
  - Let different machines compute the gradient updates on different data partitions
  - Then aggregate/sync.



Data Parallel SGD

- Data parallel stochastic gradient descent
- Data-parallelism requires every worker to have read and write access to the shared model parameters θ, which causes communication among workers;

In total P workers

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

Data partition p

Collect and aggregate before application, where communication is required

Happening locally on each worker

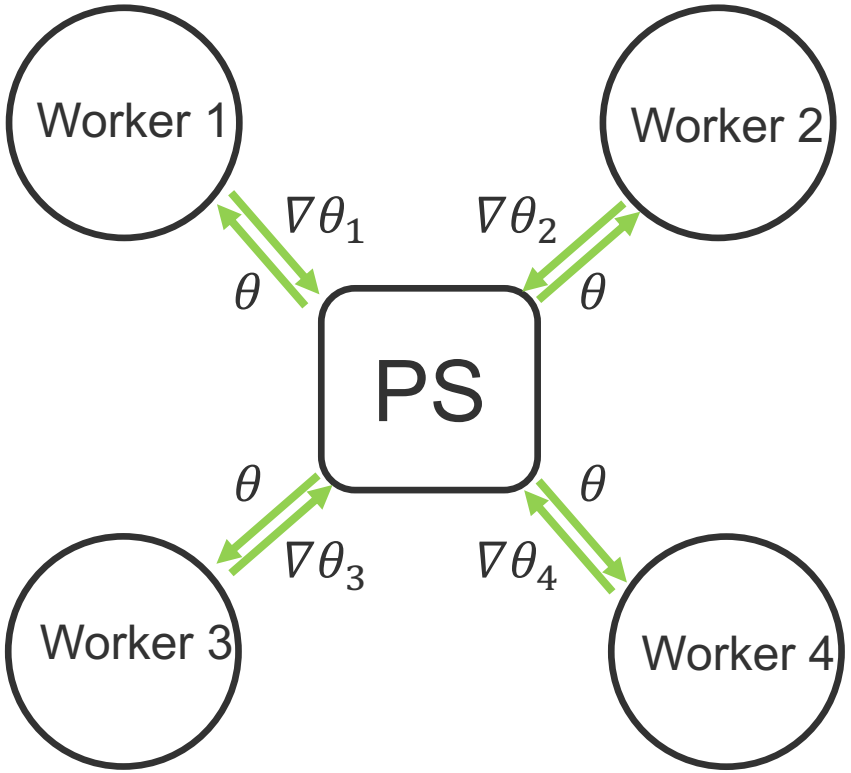
Zhang et al., 2015, Zhang et al. 2017

How to communicate

- Parameter server, e.g. Bosen, SSP
  - A parameter server (PS) is a shared memory system that provides a shared access for the global model parameters θ
- Deep learning can be trivially data-parallelized over distributed workers using PS by 3 steps:
  - Each worker computes the gradients ($\nabla \mathrm{L}$) on their own data partition (D_p) and send them to remote servers;
  - servers receive the updates and apply (+) them on globally shared parameters;
  - Each worker pulls back the updated parameters (θ_t)

Ho et al., 2013, Wei et al. 2015

How PS works



Ho et al., 2013, Wei et al. 2015, Zhang et al., 2015, Zhang et al. 2017

Parameter Server

- Parameter server has been successful for CPU-based deep learning
  - Google Distbelief, Dean et al. 2012
    - Scale up to thousands of CPU machines and 16000 CPU cores
  - SSPTable, Ho et al, 2013
    - Stale-synchronous parallel consistency model
  - Microsoft Adam, Chilimbi et al. 2014
    - 63 machines, state-of-art results on ImageNet 22K
  - Bosen, Wei et al. 2015
    - Managed communication

Parameter Server on GPUs

- Directly applying parameter server for GPU-based distributed deep learning will underperform (as will show later).
  - GPU is too fast
  - Ethernet bandwidth is limited, and has latency
- For example
  - AlexNet: 61.5M float parameters, 0.25s/iteration on Geforce Titan X (batchsize = 256)
    - Gradient generation rate: 240M float/(s*GPU)
  - Parallelize it over 8 machines each w/ one GPU using PS.
  - To ensure the computation not blocked on GPU (i.e. linear speed-up with additional nodes)
    - As a worker: send 240M floats/s and pull back 240M floats/s (at least)
    - As a server: receive 240M * 8 floats/s and send back 240M * 8/s (at least)

Zhang et al., 2015, Zhang et al. 2017

Parameter Server on GPUs

- Let's see where we are

This is what the GPU workstation in you lab has

Ethernet standards

Ethernet	Rate(GBit/s)	Rate (Mb/s)	Rate (# floats/s)
1 GbE	1	125	31.25M
10 GbE	10	1250	312.5M
Infiband	40	5000	1250M

One of the most expensive instances AWS could provide you (18$/h?)

Specialized hardware! Non-commodity anymore, inaffordable

Parameter Server on GPUs

The problem is more severe than described above

- We only use 8 nodes (which is small). How about 32,128, or even 256?
- We haven't considered other issues (which might be also troublesome), e.g.
  - Memory copy between DRAM and GPU will have a non-trivial cost
  - The Ethernet might be shared with other tasks, i.e. available bandwidth is even less.
  - Burst communication happens very often on GPUs (which will explain later).

Address the Communication Bottleneck

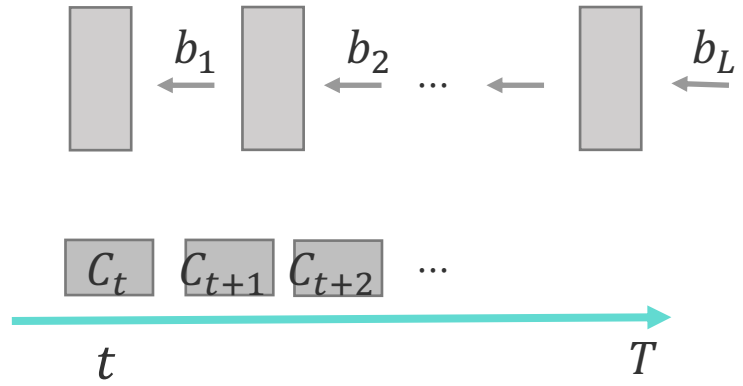
- A simple fact:
  - Communication time may be reduced, but cannot be eliminated (of course)
- Therefore, **possible** ideas to address the communication bottleneck
  - Hide the communication time by overlapping it with the computation time
  - Reduce the size of messages needed to be communications

Address the Communication Bottleneck

- A simple fact:
  - Communication time may be reduced, but cannot be eliminated (of course).
- Therefore, possible ideas to address the communication bottleneck
  - **Hide the communication time by overlapping it with the computation time**
  - Reduce the size of messages needed to be communications

Overlap Computation and Communication

- Revisit on a single node the computation flow of BP
  - b_l: backpropagation computational through layer l
  - C_t: forward and backward computation at iteration t



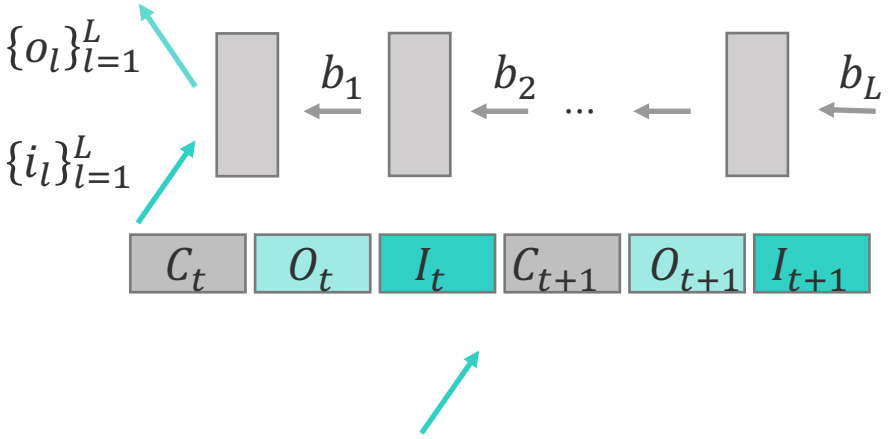
$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$

Zhang et al., 2015, Zhang et al. 2017

Overlap Computation and Communication

- On multiple nodes, when communication is involved
- Introduce two communication operations
  - o_l: send out the gradients in layer l to the remote
  - i_l: pull back the globally shared parameters of layer l from the remote
  - O_t: the set $\{o_l\}_{l=1}^{L}$ at iteration t
  - I_t: the set $\{i_l\}_{l=1}^{L}$ at iteration t

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$



Computation and communication happen sequentially!

Zhang et al., 2015, Zhang et al. 2017

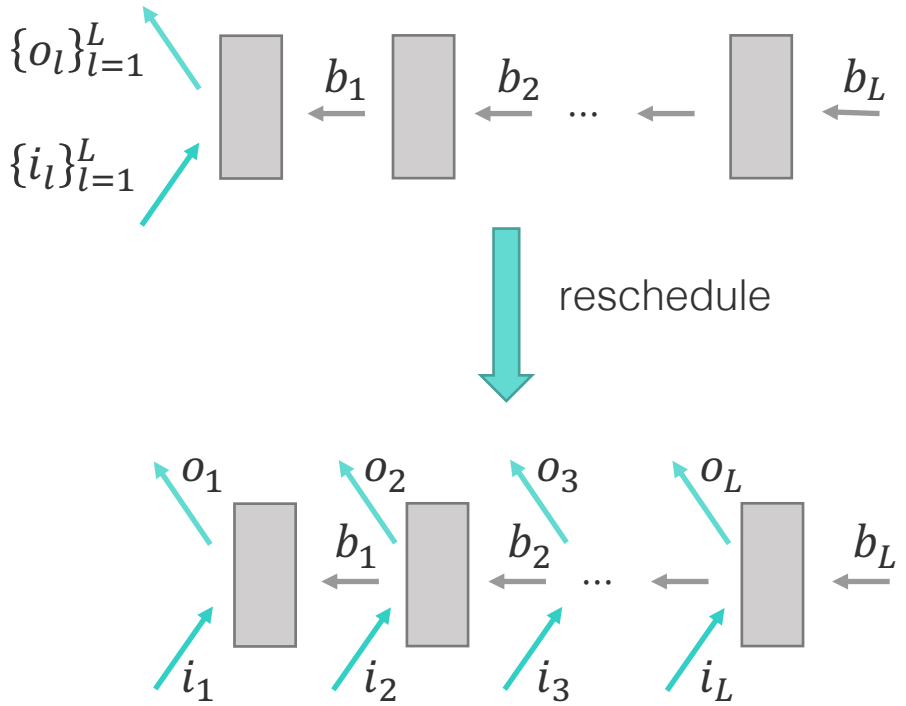
Overlap Computation and Communication

- Note the following independency
  - The send-out operation o_l is independent of backward operations
  - The read-in operation i_l could update the layer parameters as long as b_l was finished, without blocking the subsequent backward operations b_i $(i < l)$
- Idea: overlap computation and communication by utilizing concurrency
  - Pipelining the updates and computation operations

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$

WFBP: Wait-free backpropagation

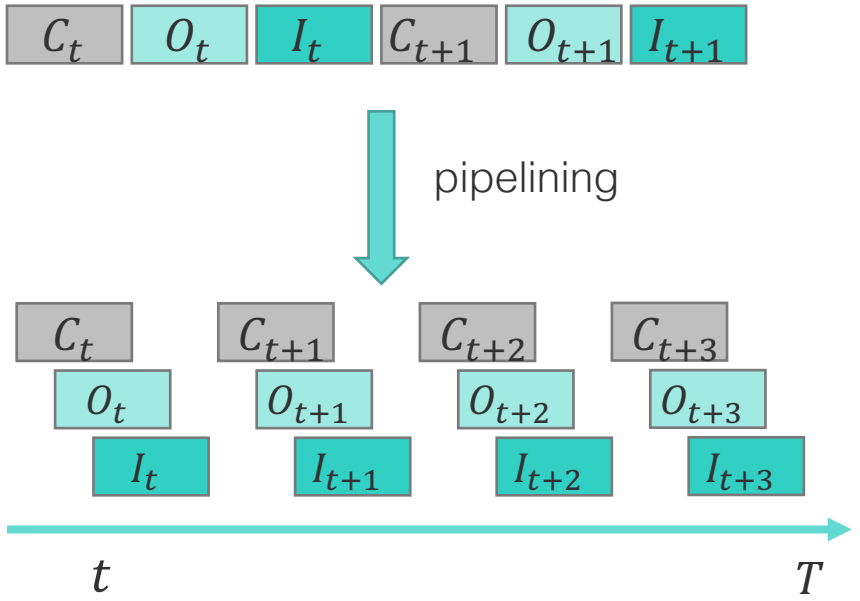
- Idea: overlap computation and communication by utilizing concurrency
  - Pipelining the updates and computation operations



Zhang et al., 2015, Zhang et al. 2017

WFBP: Wait-free backpropagation

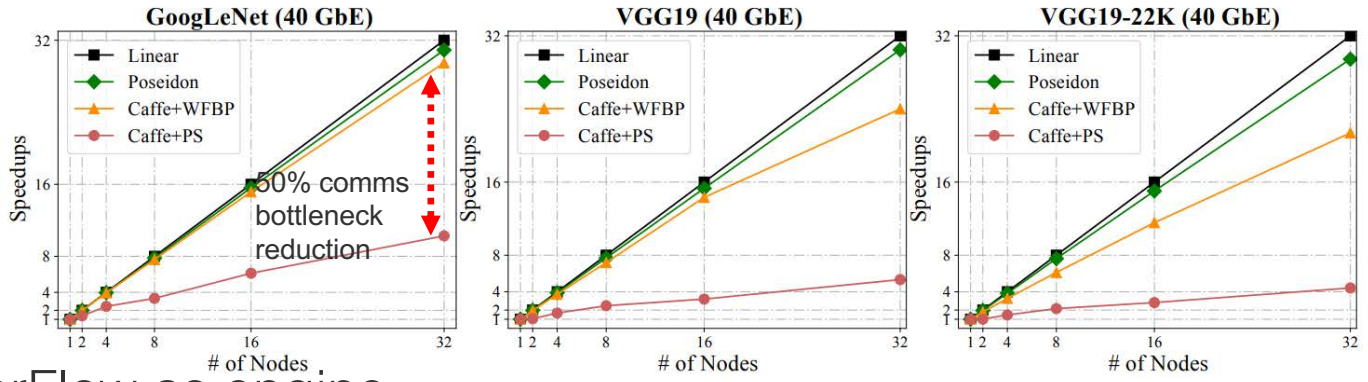
- Idea: overlap computation and communication by utilizing concurrency
  - Communication overhead is hidden under computation
  - Results: more computations in unit time



Zhang et al., 2015, Zhang et al. 2017

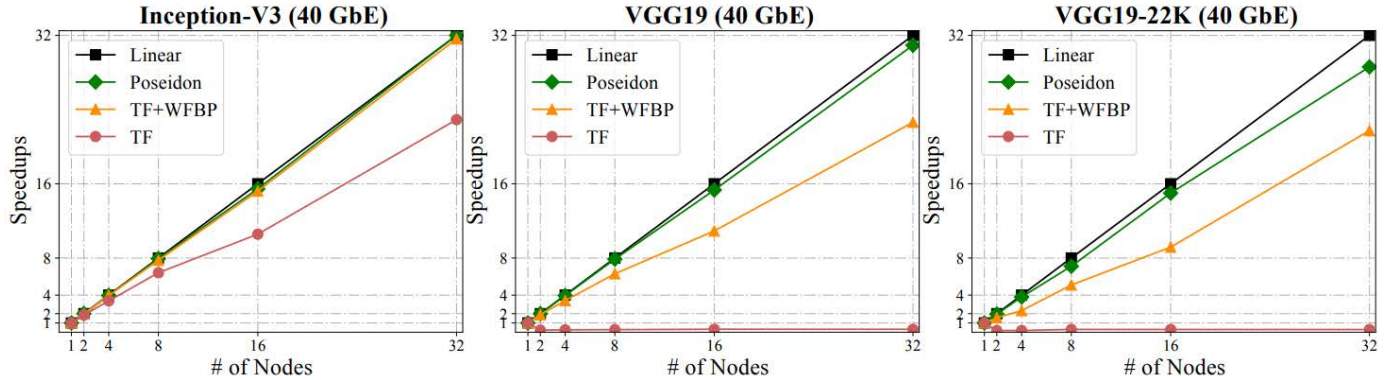
WFBP: Distributed Wait-free backpropagation

- How does WFBP perform?
  - Using Caffe as an engine:



Zhang et al. 2017

  - Using TensorFlow as engine:



WFBP: Distributed Wait-free backpropagation

- Observation: Why DWBP would be effective
  - More statistics of modern CNNs

Params/FLOP distribution of modern CNNs

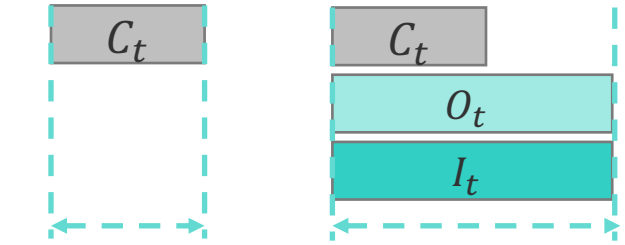
Parameters	CONV Layers (#/%)	FC Layers (#/%)
AlexNet	2.3M / 3.75	59M / 96.25
VGG-16	7.15M / 5.58	121.1M / 94.42
FLOPs	**CONV Layers (#/%)**	**FC Layers (#/%)**
AlexNet	1,352M / 92.0	117M / 8.0
VGG-16	10,937M / 91.3	121.1M / 8.7

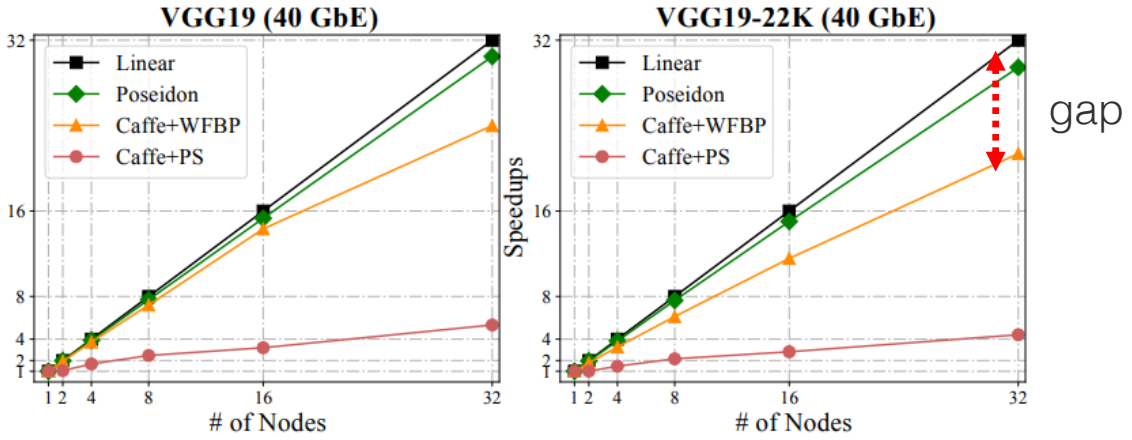
- 90% computation happens at bottom layers
- 90% communication happens at top layers
- WFBP overlaps 90% and 90%

Zhang et al., 2015, Zhang et al. 2017

WFBP: Wait-free Backpropagation

- Does overlapping communication and computation solve all the problems?
  - When communication time is longer than computation, no (see the figure below).
  - Say, if communication and computation are perfectly overlapped, how many scalability we can achieve?





gap

Zhang et al., 2015, Zhang et al. 2017

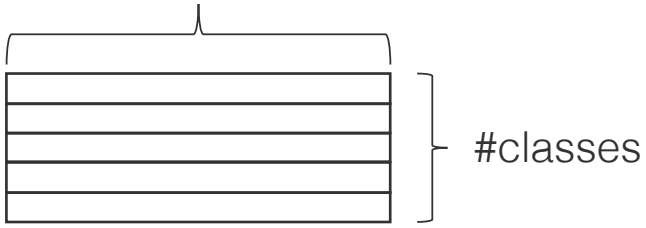
Address the communication bottleneck

- Note a simple fact:
  - Communication time may be reduced, but cannot be eliminated (of course).
- Therefore, possible ideas to address the communication bottleneck
  - Hide the communication time by overlapping it with the computation time – which we have described before.
  - **Reduce the size of messages needed to be communications**
    - **While without compromising statistical convergence**

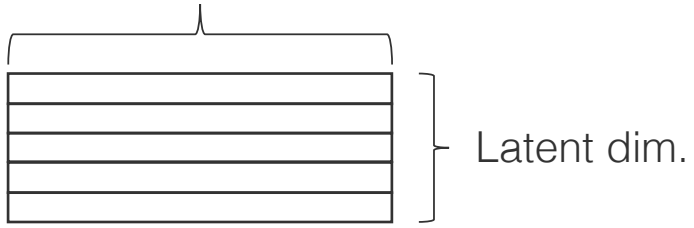
Introducing Sufficient Factor Broadcasting

- Matrix-parametrized models

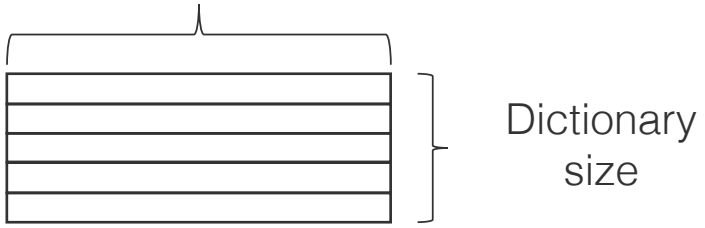
Multiclass Logistic Regression



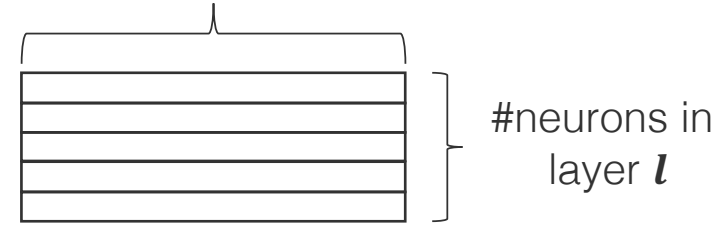
Distance Metric Learning



Sparse Coding

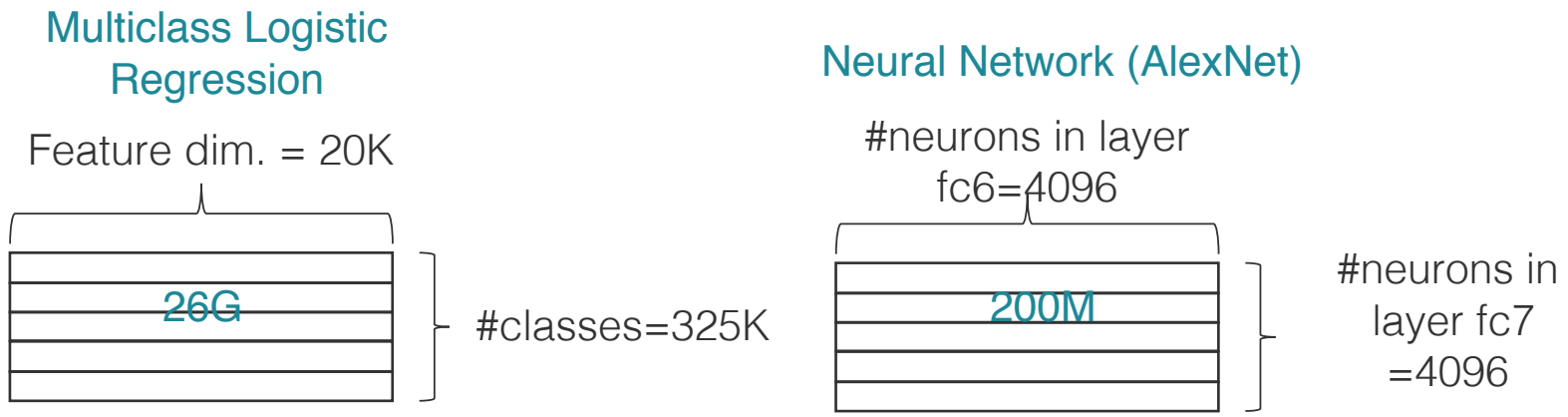


Neural Network



Distributed Learning of MPMs

- Learning MPMs by communicating parameter matrices between server and workers
  - Dean and Ghemawat, 2008; Dean et al, 2012; Sindhwani and Ghoting, 2012; Gopal and Yang, 2013; Chilimbi et al, 2014, Li et al, 2015
- High communication cost and large synchronization delays



Contents:
Sufficient Factor (SF) Updates

Full parameter matrix update ΔW can be computed as outer product of two vectors uv^T (called sufficient factors)

- Example: Primal stochastic gradient descent (SGD)

$$\min_W \frac{1}{N}\sum_{i=1}^{N} f_i(Wa_i; b_i) + h(W)$$

$$\Delta W = uv^{\mathrm{T}} \quad u = \frac{\partial f(Wa_i, b_i)}{\partial (Wa_i)} \quad v = a_i$$

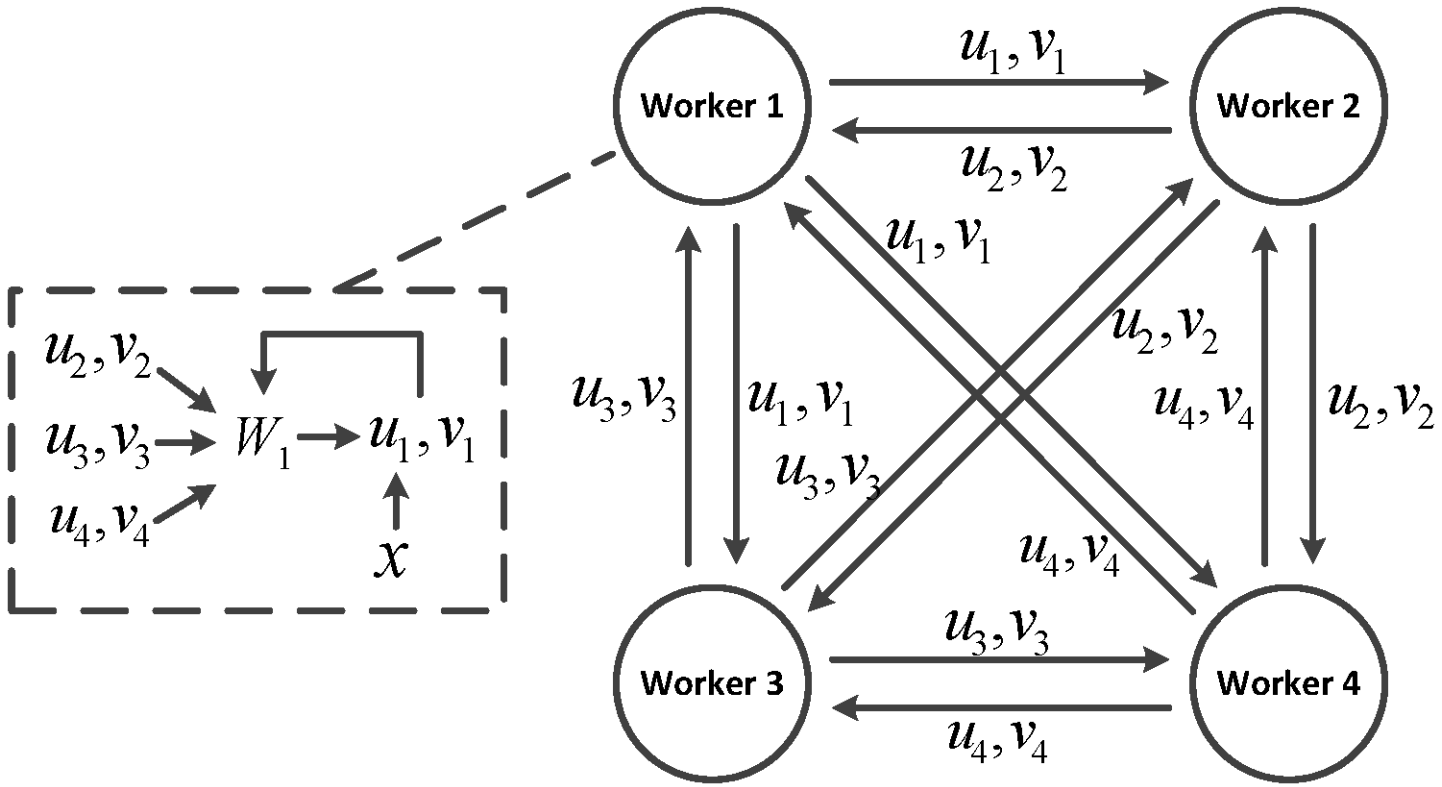
- Example: Stochastic dual coordinate ascent (SDCA)

$$\min_Z \frac{1}{N}\sum_{i=1}^{N} f_i^*(-z_i) + h^*\left(\frac{1}{N} ZA^{\mathrm{T}}\right)$$

$$\Delta W = uv^{\mathrm{T}} \quad u = \Delta z_i \quad v = a_i$$

Send lightweight SF updates (u,v), instead of expensive full-matrix ΔW updates!

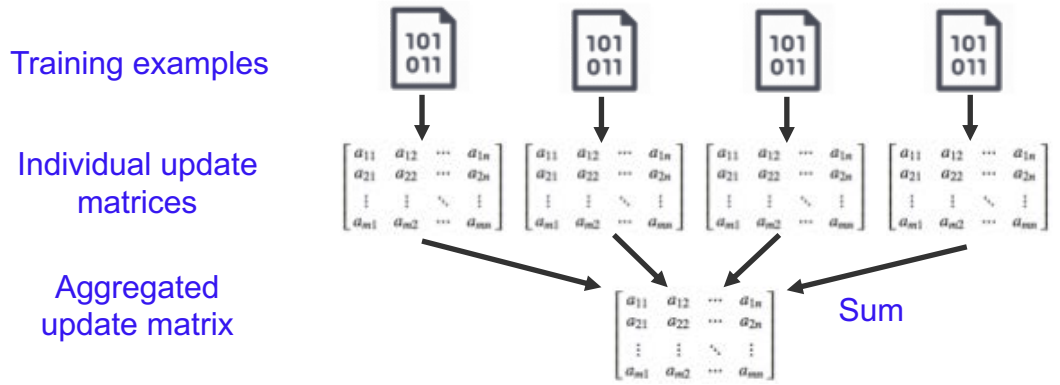
Sufficient Factor Broadcasting: P2P Topology + SF Updates



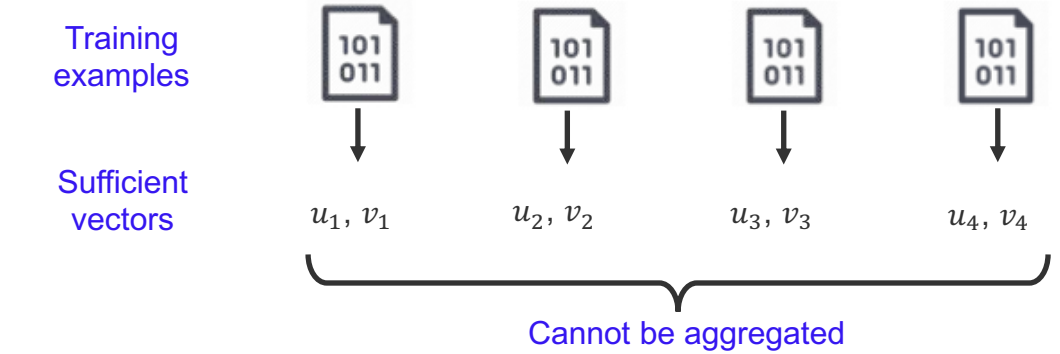
Xie et al., 2015

A computing & communication tradeoff

- Full update:



- Pre-update

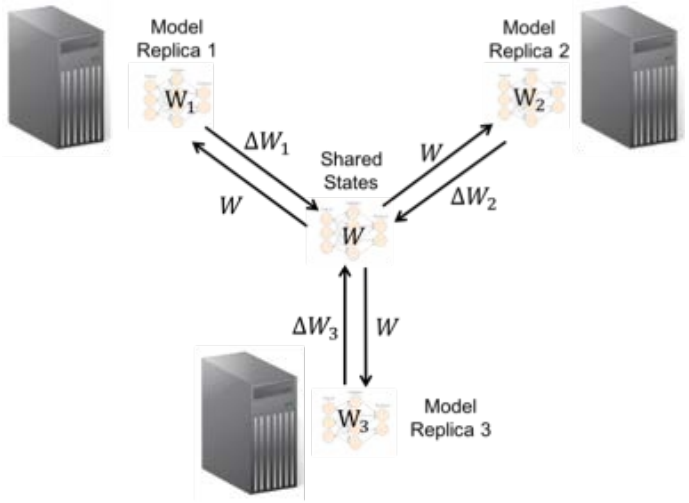


- Stochastic algorithms
  - Mini-batch: C samples

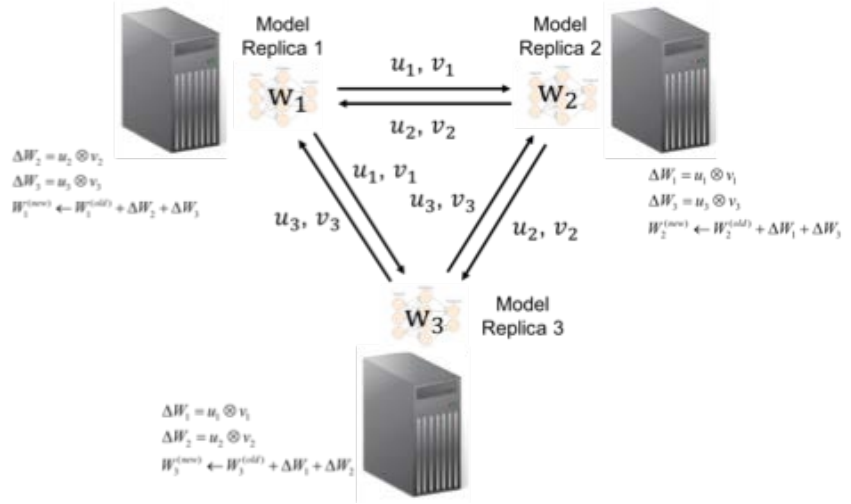
Matrix Representation	$O(JK)$
SV Representation	$O((J+K)C)$

Synchronization of Parameter Replicas

parameter server



Transfer SVs instead of ΔW



- A Cost Comparison

	Size of one message	Number of messages	Network Traffic
P2P SV-Transfer	$O(J+K)$	$O(P^2)$	$O((J+K)P^2)$
Parameter Server	$O(JK)$	$O(P)$	$O(JKP)$

Convergence Speedup

Multiclass Logistic Regression (MLR)

Distance Metric Learning (DML)

Sparse Coding (SC)

- 3 Benchmark ML Programs
  - Big parameter matrices with 6.5-8.6b entries (30+GB), running on 12- & 28-machine clusters
- 28-machine SFB finished in 2-7 hours
  - Up to 5.6x faster than 28-machine PS, 12.3x faster than 28-machine Spark
- PS cannot support SF communication, which requires decentralized storage

Xie et al., 2015

Convergence Guarantee

- Assumptions
  - Bridging model
    - Staleness Synchronous Parallel (SSP) with staleness parameter s
    - Bulk Synchronous Parallel is a special case of SSP when $s = 0$
  - Communication methods
    - Partial broadcast (PB): sending messages to a subset of Q ($Q < P - 1$) machines
    - Full broadcast is a special case of PB when $Q = P - 1$
  - **Assumption 1.** *(1) For all j, f_j is continuously differentiable and F is bounded from below; (2) ∇F, ∇F_p are Lipschitz continuous with constants L_F and L_p, respectively, and let $L = \sum_{p=1}^{P} L_p$; (3) There exists G, σ^2 such that for all p and c, we have (almost surely) $\|U_p(\mathbf{W}_p^c, I_p^c)\| \leq G\eta$ and $\mathbb{E}\| \, |S_p| \sum_{j \in I_p} \nabla f_j(\mathbf{W}) - \nabla F_p(\mathbf{W}) \, \|_2^2 \leq \sigma^2$.*

Convergence Guarantee

- Results

Theorem 1. *Let Assumption 1 hold, and let* $\{\mathbf{W}_p^c\}$, $p = 1, \ldots, P$, $\{\mathbf{W}^c\}$ *be the local sequences and the auxiliary sequence, respectively.*

Under full broadcasting (i.e., $Q = P - 1$*) and set the learning rate* $\eta := \eta_c = O(\sqrt{\frac{1}{L\sigma^2 Psc}})$*, we have*

- $\liminf_{c\to\infty} \mathbb{E}\|\nabla F(\mathbf{W}^c)\| = 0$*, hence there exists a subsequence of* $\nabla F(\mathbf{W}^c)$ *that almost surely vanishes;*
- $\lim_{c\to\infty} \max_p \|\mathbf{W}^c - \mathbf{W}_p^c\| = 0$*, i.e., the maximal disagreement between all local sequences and the auxiliary sequence converges to 0 (almost surely);*
- *There exists a common subsequence of* $\{\mathbf{W}_p^c\}$ *and* $\{\mathbf{W}^c\}$ *that converges almost surely to a stationary point of* F*, with the rate* $\min_{c\le C} \mathbb{E}\| \sum_{p=1}^P \nabla F_p(\mathbf{W}_p^c)\|_2^2 \le O\left(\sqrt{\frac{L\sigma^2 Ps}{C}}\right)$

Under partial broadcasting (i.e., $Q < P - 1$*) and set a constant learning rate* $\eta = \frac{1}{CLG(P-Q)}$*, where* C *is the total number of iterations. Then we have*

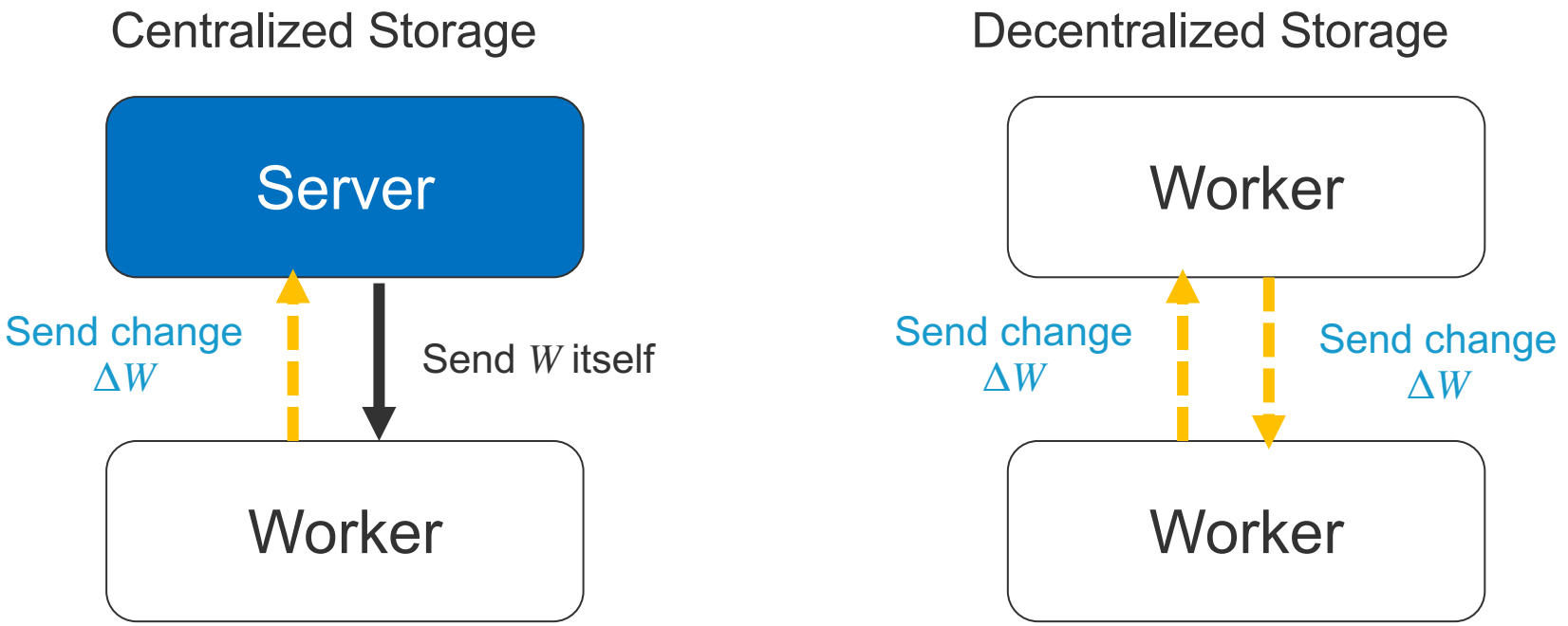
$$\min_{c\le C} \mathbb{E}\left[\| \sum_{p=1}^P \nabla F_p(\mathbf{W}_p^c)\|_2^2\right] \le O\left(LG(P-Q) + \frac{P(sG+\sigma^2)}{CG(P-Q)}\right).$$

Hence, the algorithm converges to a $O(LG(P-Q))$ *neighbourhood if* $C \to \infty$.

Convergence Guarantee

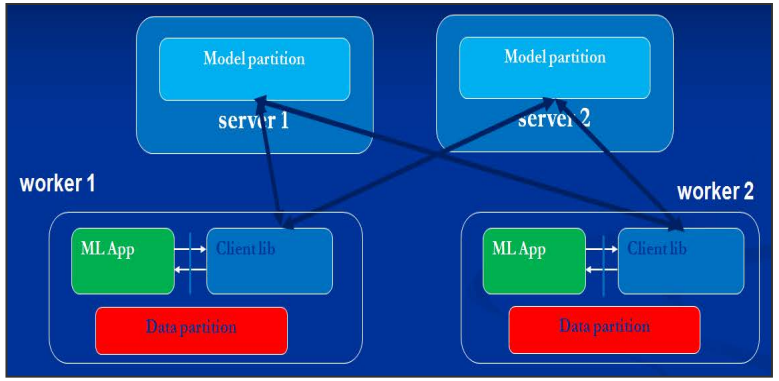
- Take-home message:
  - Under full broadcasting, given a properly-chosen learning rate, all local worker parameters W_p^c eventually converge to stationary points (i.e. local minima) of the objective function, despite the fact that SV transmission can be delayed by up to s iterations.
  - Under partial broadcasting, the algorithm converges to a $O(LG(P-Q))$ neighbourhood if $C \longrightarrow \infty$.

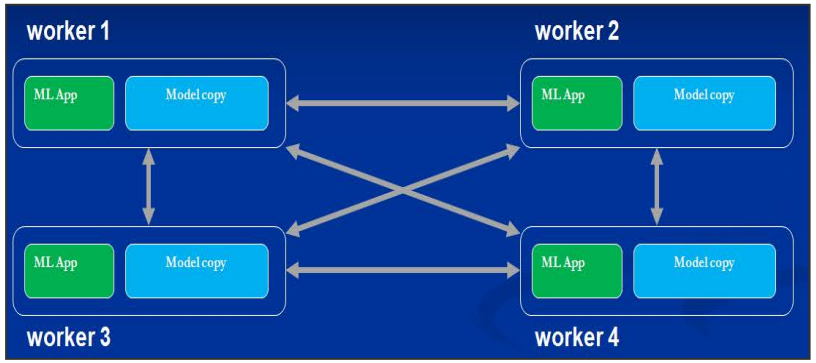
Parameter Storage and Communication Paradigms



- **Centralized:** send parameter W itself from server to worker
  - Advantage: allows compact comms topology, e.g. bipartite
- **Decentralized:** always send changes ΔW between workers
  - Advantage: more robust, homogeneous code, low communication (?)

Topologies: Master-Slave versus P2P?





Master-slave

- Used with **centralized storage** paradigm
- **Disadvantage:** need to code/manage clients and servers separately
- **Advantage:** bipartite topology is comms-efficient
- Popular for Parameter Servers: Yahoo LDA, Google DistBelief, Petuum PS, Project Adam, Li&Smola PS, …

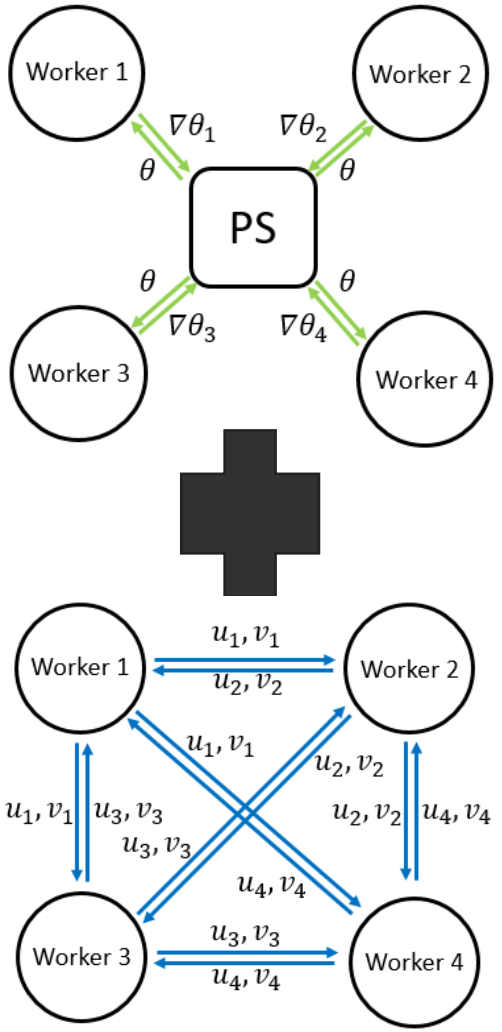
P2P

- Used with **decentralized storage**
- **Disadvantage (?):** high comms volume for large # of workers
- **Advantage:** same code for all workers; no single point of failure, high elasticity to resource adjustment
- Less well-explored due to perception of high communication overhead?

Hybrid Updates: PS + SFB

- Hybrid communications: Parameter Server + Sufficient Factor Broadcasting
  - Parameter Server: Master-Slave topology
  - Sufficient factor broadcasting: P2P topology

- For problems with a mix of large and small matrices,
  - Send small matrices via PS
  - Send large matrices via SFB



Zhang et al., 2015, Zhang et al. 2017

Hybrid example: CNN

Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, Eric P. Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. USENIX ATC 2016.

- Example: AlexNet CNN model
  - Final layers = 4096 * 30000 matrix (120M parameters)
  - Use SFB to communicate
    - 1. Decouple into two 4096 vectors: u, v
    - 2. Transmit two vectors
    - 3. Reconstruct the gradient matrix

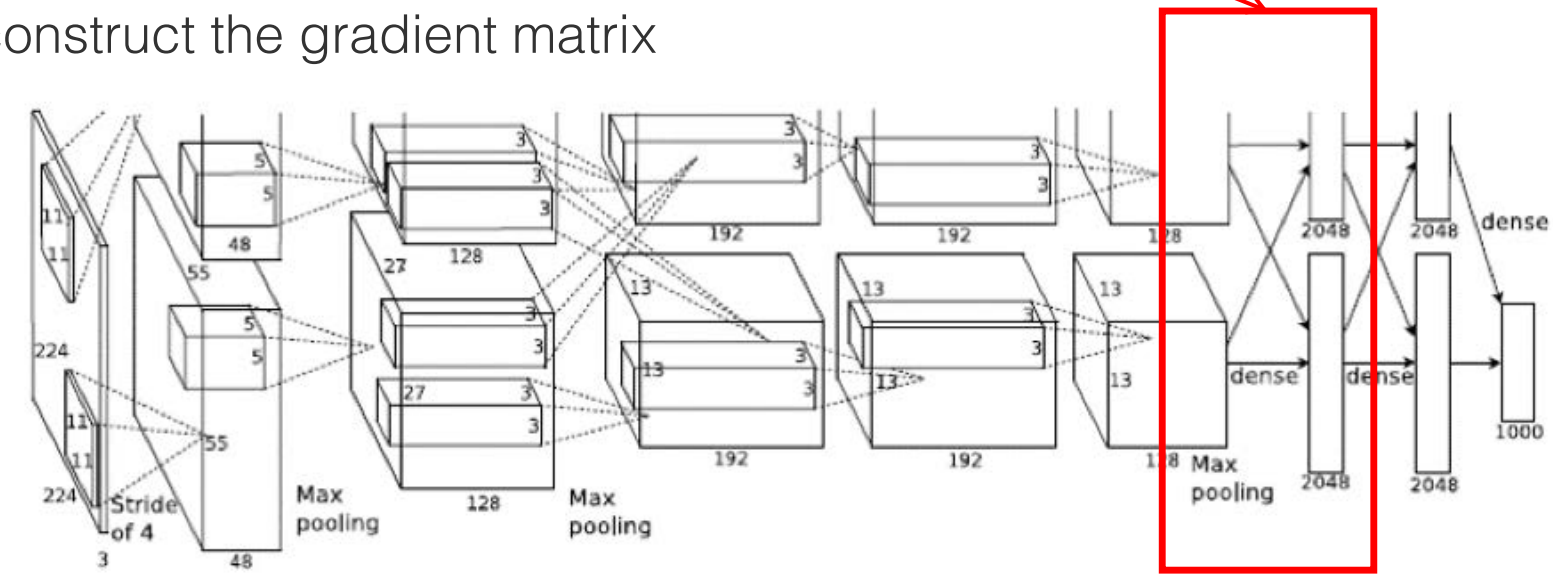


Figure from Krizhevsky et al. 2012

Zhang et al., 2015, Zhang et al. 2017

Hybrid example: CNN

Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, Eric P. Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. USENIX ATC 2016.

- Example: AlexNet CNN model
  - Convolutional layers = e.g. 11 * 11 matrix (121 parameters)
  - Use Full-matrix updates to communicate
    - 1. Send/receive using Master-Slave PS topology

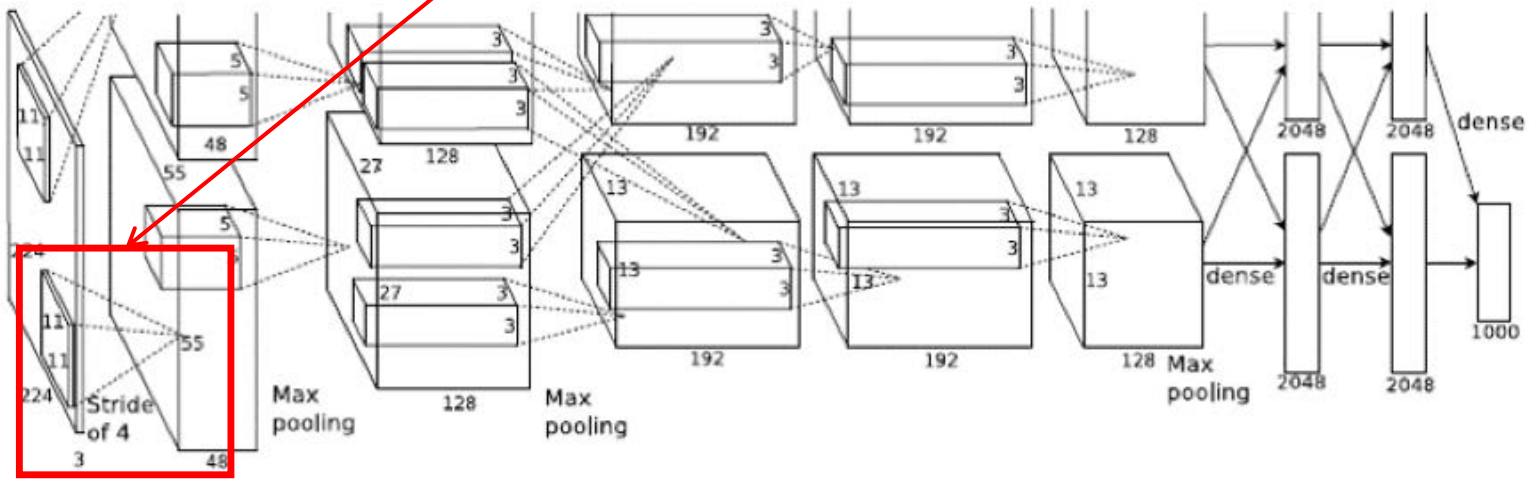
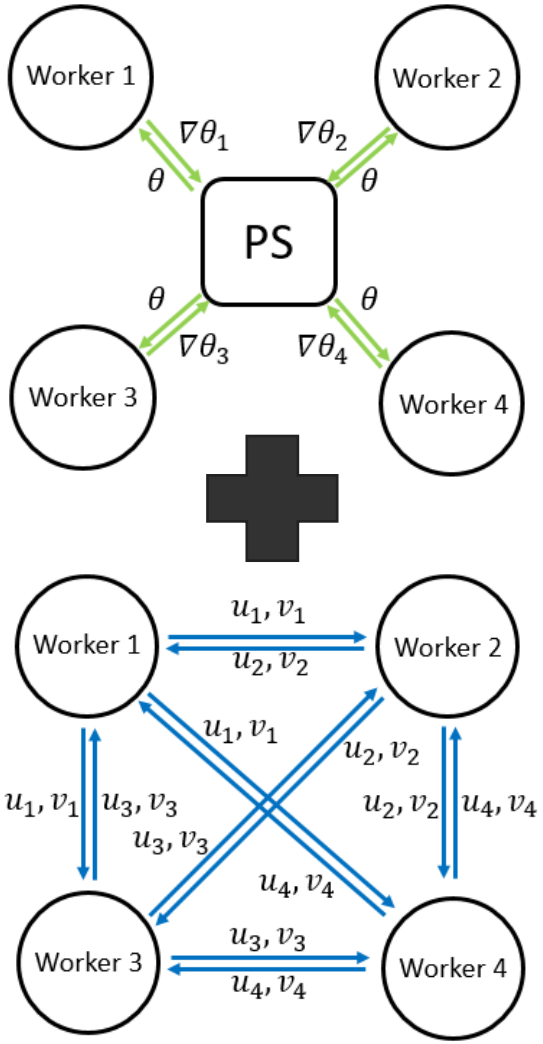


Figure from Krizhevsky et al. 2012

Zhang et al., 2015, Zhang et al. 2017

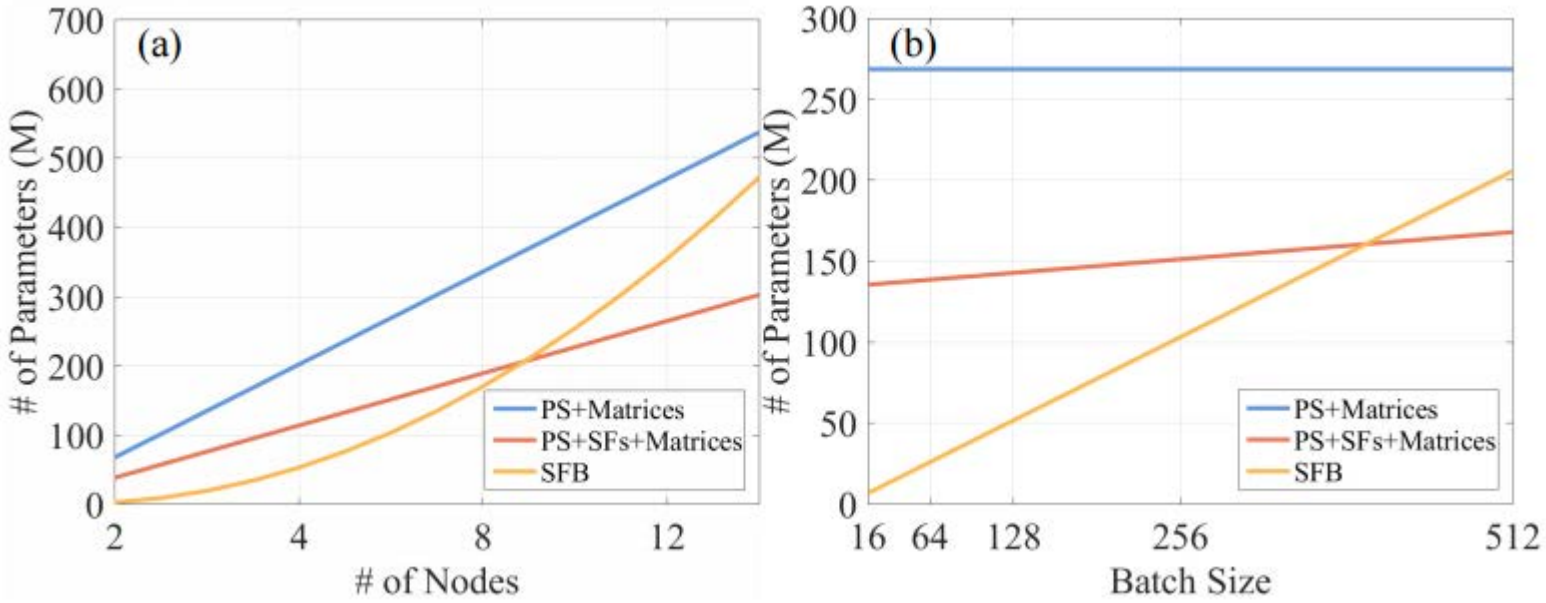
Hybrid Communication

- Idea
  - Sync FC layers using SFB
  - Sync Conv layer using PS
- Effectiveness
  - It directly reduces the size of messages in many situations
- Is SFB always optimal?
  - No, its communication load increases quadratically
  - The right strategy: choose PS whenever it results in less communication



Hybrid Communication

- A best of both worlds strategy
- For example, AlexNet parameters between FC6 and FC7
- Tradeoff between PS and SFB communication



Zhang et al., 2015

Hybrid Communication

- How to choose? Where is the threshold?
- Determine the best strategy depending on
  - Layer type: CONV or FC?
  - Layer size
  - Batch size
  - # of Cluster nodes

Method	Server	Worker	Server & Worker
PS	$2P_1MN/P_2$	$2MN$	$2MN(P_1+P_2-2)/P_2$
SFB	**N/A**	$2K(P_1-1)(M+N)$	**N/A**
Adam (max)	$P_1MN+P_1K(M+N)$	$K(M+N)+MN$	$(P_1-1)(MN+KM+KN)$

Table 1: Estimated communication cost of PS, SFB and Adam for synchrnizing the parameters of a $M \times N$ FC layer on a cluster with P_1 workers and P_2 servers, when batchsize is K.

Zhang et al., 2015, Zhang et al. 2017

Hybrid Communication

- Hybrid communication algorithm

Determine the best strategy depending on

- Layer type: CONV or FC?
- Layer size: M, N
- Batch size: K
- # of Cluster nodes: P_1, P_2

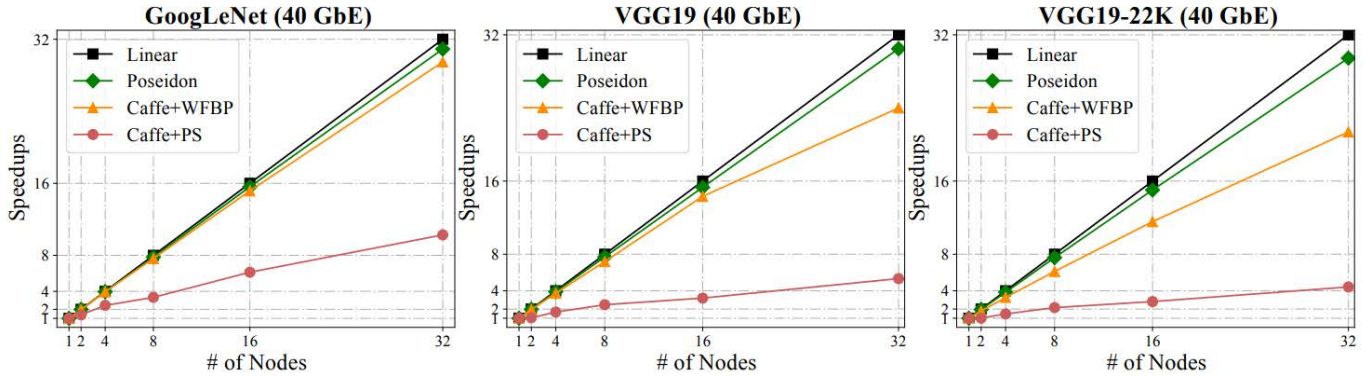
Algorithm 1 Get the best comm method of layer l

```
function BESTSCHEME(l)
    layer_property = Query(l.name)
    P1, P2, K = Query('n_worker', 'n_server', 'batchsize')
    if layer_property.type == 'FC' then
        M = layer_property.width
        N = layer_property.height
        if 2K(P1 − 1)(M + N) ≤ 2MN(P1+P2−2)/P2 then
            return 'SFB'
        end if
    end if
    return 'PS'
end function
```

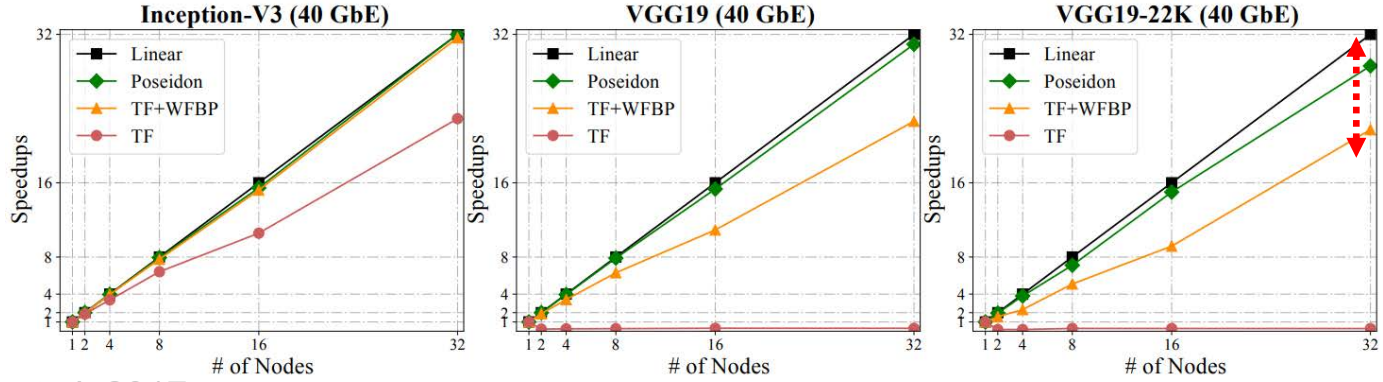
Zhang et al., 2015, Zhang et al. 2017

Hybrid Communication

- Results: achieve linear scalability across different models/data with 40GbE bandwidth
  - Using Caffe as an engine:



  - Using TensorFlow as engine

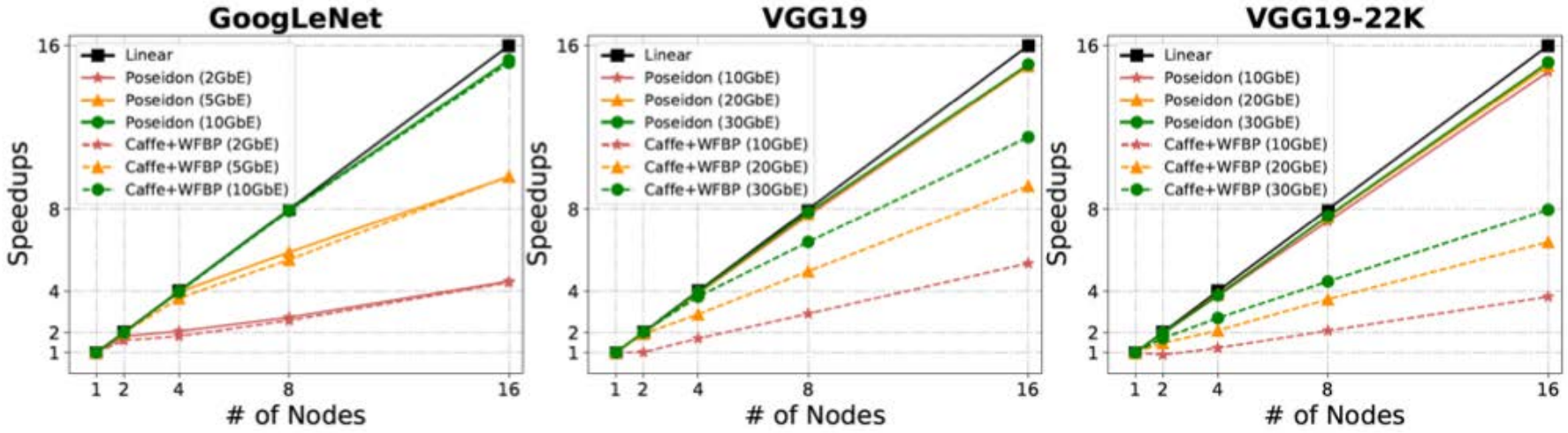


Improve over WFBP

Zhang et al., 2015, Zhang et al. 2017

Hybrid Communication

- Linear scalability on throughput, even with limited bandwidth!
  - Make distributed deep learning affordable



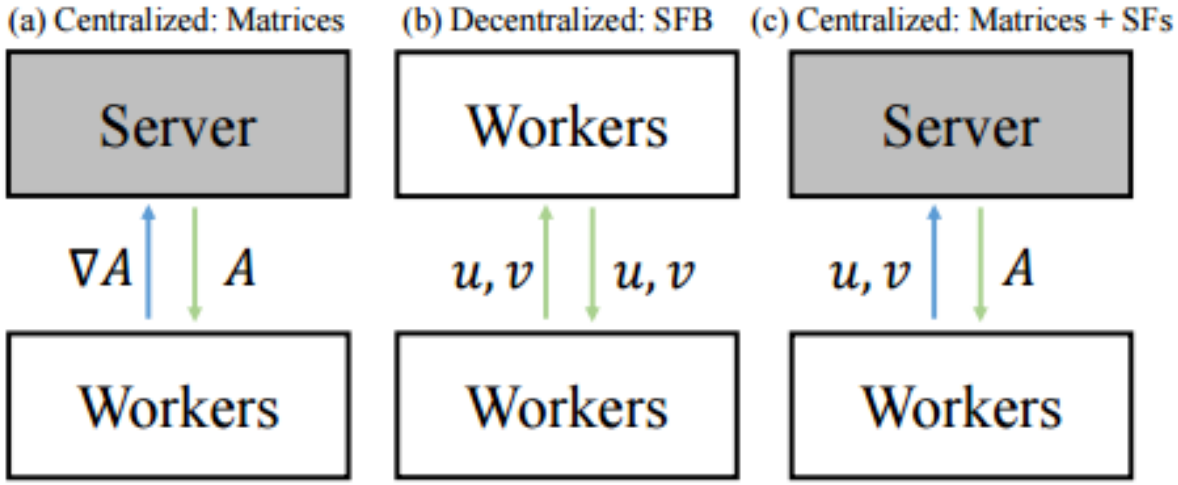
parameters 5M 143M 229M

Ethernet	Rate(GBit/s)	Rate (Mb/s)	Rate (# floats/s)
1 GbE	1	125	31.25M
10 GbE	10	1250	312.5M
Infiband	40	5000	1250M

Zhang et al., 2015, Zhang et al. 2017

Hybrid Communication

- Discussion: Utilizing SFs is not a new idea, actually
  - Microsoft Adam uses the third strategy (c)



Hybrid Communication

- Adam's strategy leads to communication bottleneck
  - Pushing SFs to server is fine
  - Pulling full matrices back will create a bottleneck on the server node.

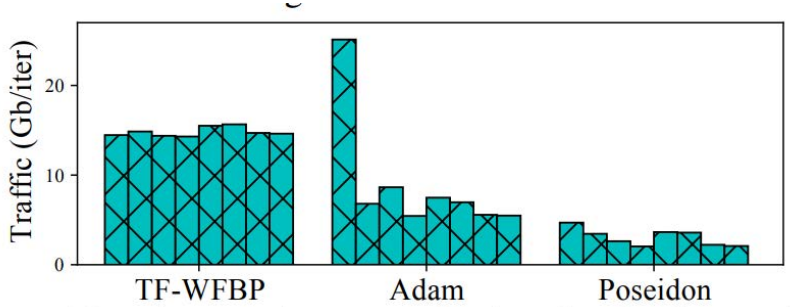
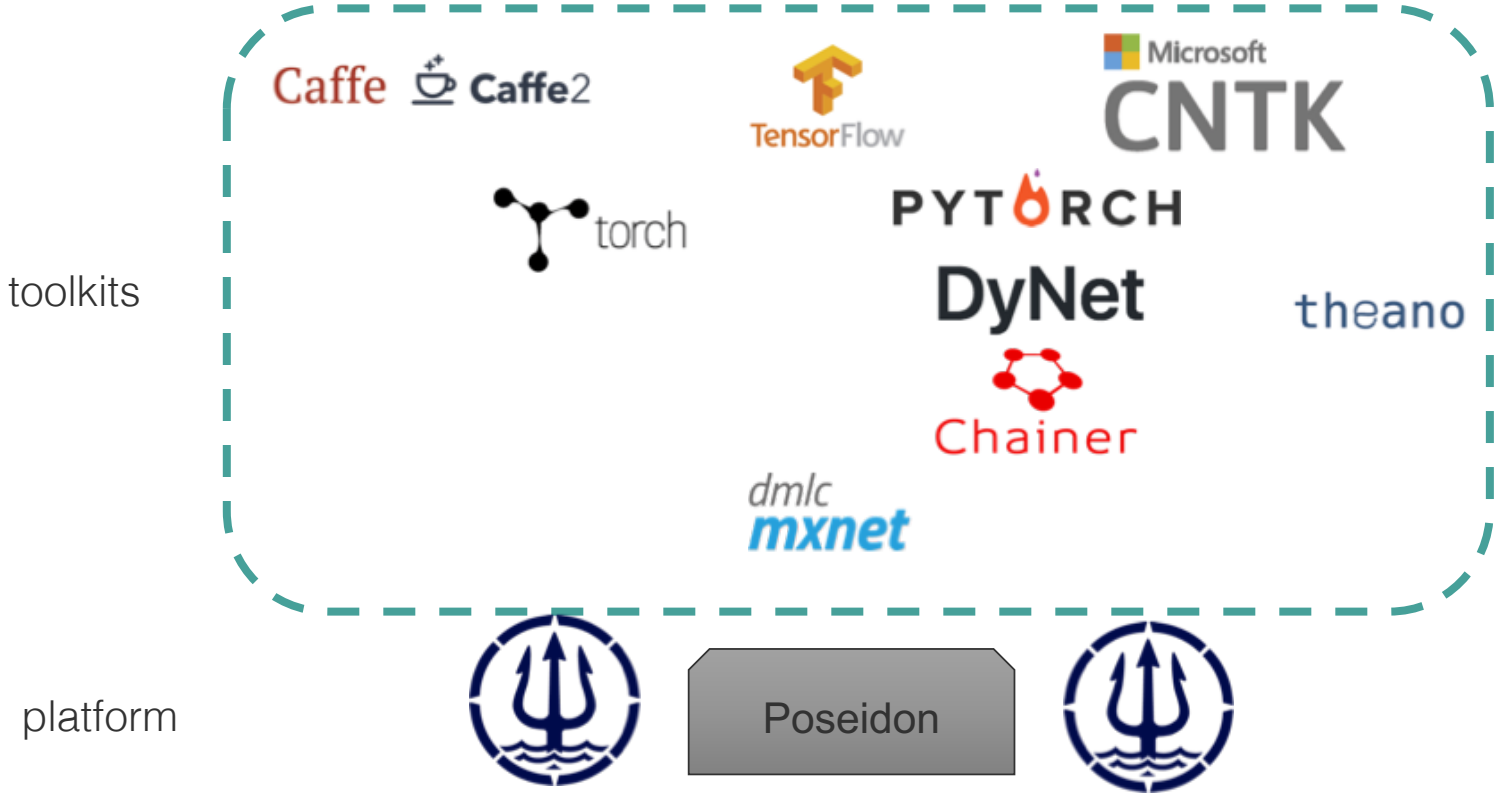


Figure 10: Averaged communication load when training VGG19 using *TF-WFBP*, *Adam* and *Poseidon* with TensorFlow engine. Each bar represents the network traffic on a node.

- Hybrid communication yields communication load balancing
  - Which is important to address the problem of burst communication.

Introducing Poseidon

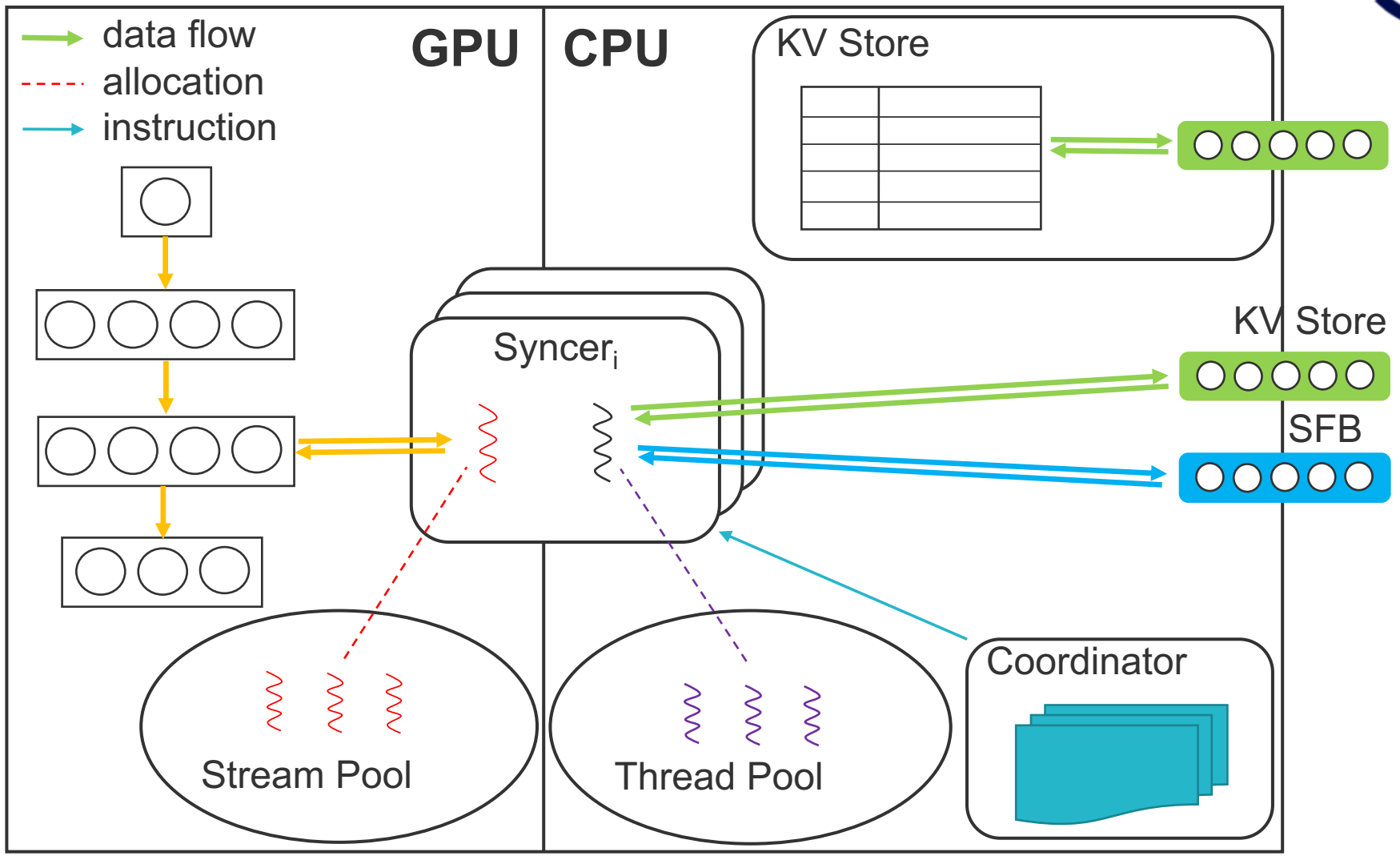
- Poseidon: An efficient communication architecture
  - A distributed platform to amplify existing DL toolkits



Poseidon's position

- Design principles
  - Efficient distributed platform for amplifying any DL toolkits
  - Preserve the programming interface for any high-level toolkits
    - i.e. distribute the DL program without changing any line of code
  - Easy deployment, easy adoption.

Poseidon System Architecture



Zhang et al., 2015, Zhang et al. 2017

Poseidon APIs

- KV Store, Syncer and Coordinator
- Standard APIs similar to parameter server
  - *Push/Pull* API for parameter synchronization
  - *BestScheme* method to return the best communication method

Method	Owner	Arguments	Description
`BestScheme`	Coordinator	A layer name or index	Get the best communication scheme of a layer
`Query`	Coordinator	A list of property names	Query information from coordinators' information book
`Send`	Syncer	None	Send out the parameter updates of the corresponding layer
`Receive`	Syncer	None	Receive parameter updates from either parameter server or peer workers
`Move`	Syncer	A GPU stream and an indicator of move direction	Move contents between GPU and CPU, do transformations and application of updates if needed
`Send`	KV store	updated parameters	Send out the updated parameters
`Receive`	KV store	parameter buffer of KV stores	Receive gradient updates from workers

Zhang et al., 2015, Zhang et al. 2017

Amplify DL toolboxes Using Poseidon

- For developers: plug Poseidon API into the backpropagation code, all you need to do is:
  - Back propagate through layer l
  - Sync parameters of layer l
  - Wait for finishing
- Amplifying Google TensorFlow
  - 250 line of code
- Amplifying Caffe
  - 150 line of code

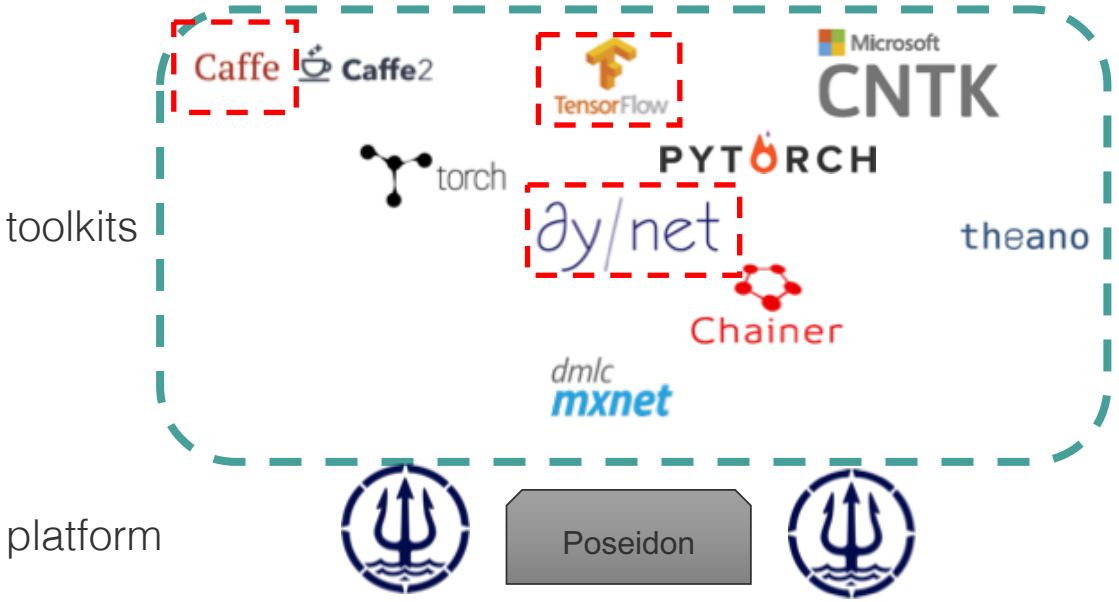
Algorithm 2 Parallelize a DL library using Poseidon

```
1:  function TRAIN(net)
2:      for iter = 1 → T do
3:          sync_count = 0
4:          net.Forward()
5:          for l = L → 1 do
6:              net.BackwardThrough(l)
7:              thread_pool.Schedule(sync(l))
8:          end for
9:          wait_until(sync_count == net.num_layers)
10:     end for
11: end function
12: function SYNC(l)
13:     stream = stream_pool.Allocate()
14:     syncers[l].Move(stream, GPU2CPU)
15:     syncers[l].method = coordinator.BestScheme(l)
16:     syncers[l].Send()
17:     syncers[l].Receive()
18:     syncers[l].Move(stream, CPU2GPU)
19:     sync_count++
20: end function
```

Zhang et al., 2015, Zhang et al. 2017

Using Poseidon

- Poseidon: An efficient communication architecture
  - Preserve the programming interface for any high-level toolkits
    - i.e. distribute the DL program **without changing any line of application code**



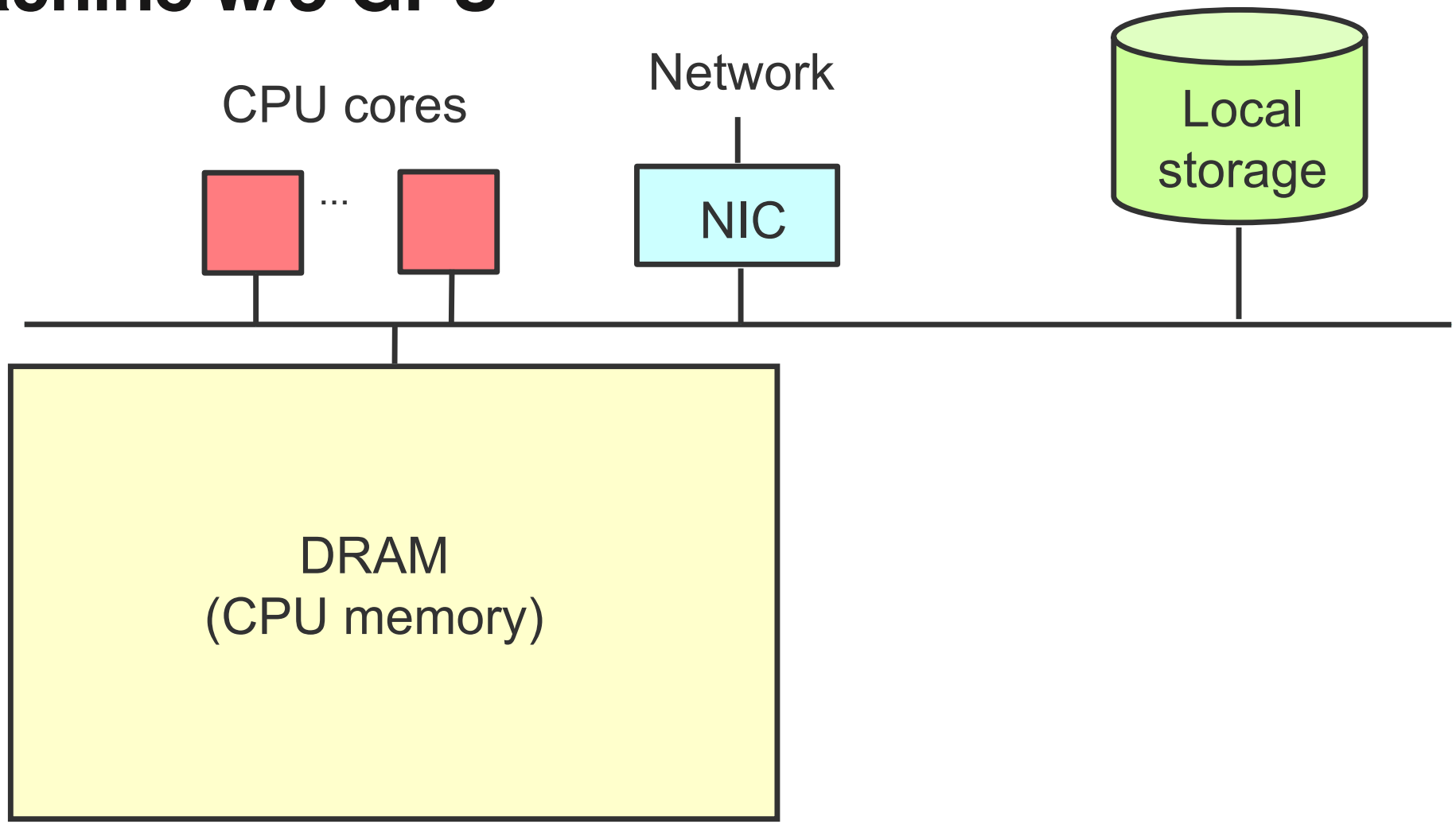
Outline

- Overview: Distributed Deep Learning on GPUs
- Challenges 1: Addressing the communication bottleneck
- **Challenges 2: Handling the limited GPU memory**

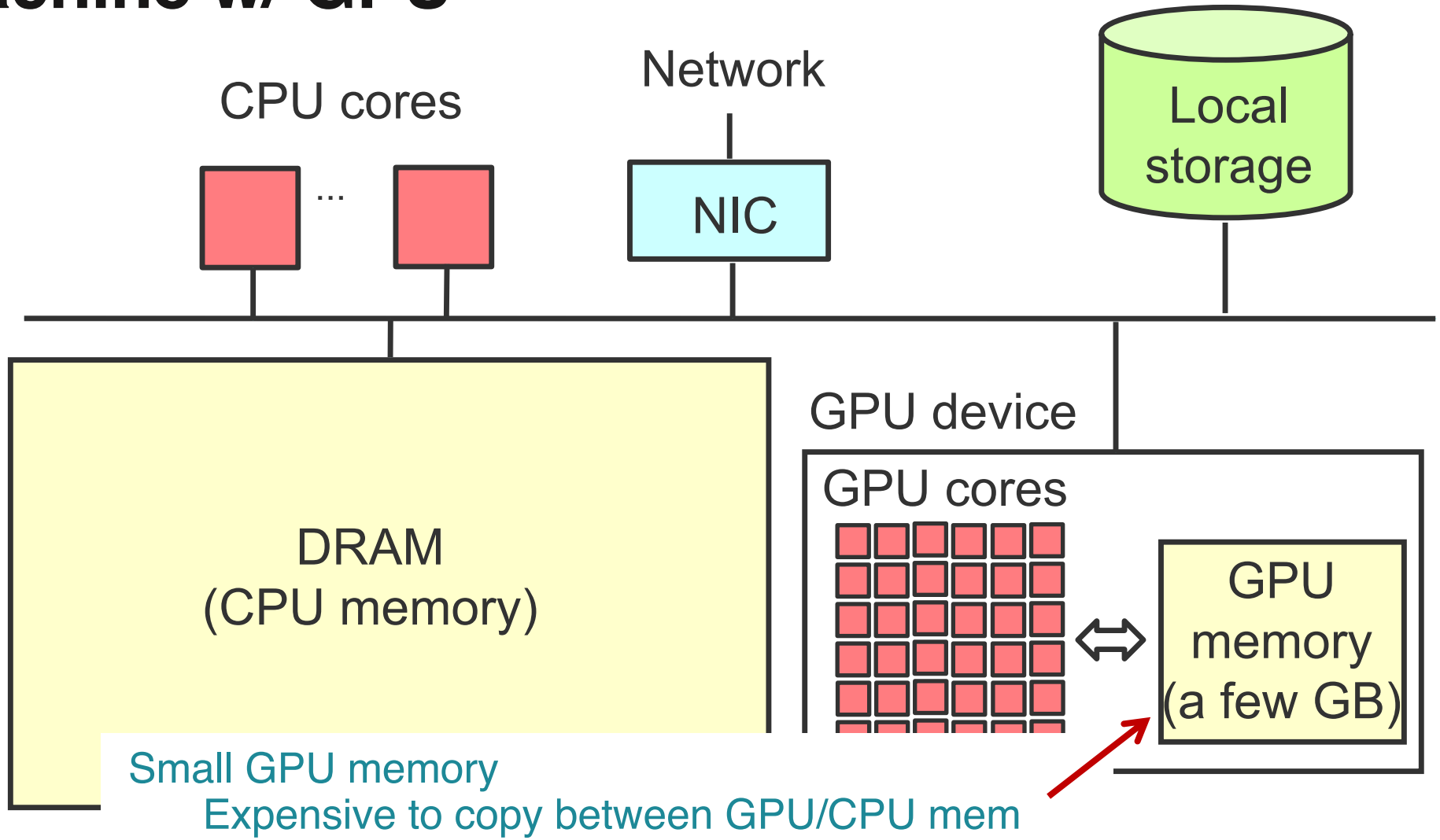
What is the Issue

- Memory
  - GPUs have dedicate memory
  - For a DL training program to be efficient, its data must be placed on GPU memory
  - GPU memory is limited, compared to CPU, e.g. maximally 12Gb
  - Memcpy between CPU and GPU is expensive – a memcpy takes the same time as launching a GPU computation kernel
- Problems to be answered
  - How to Avoid memcpy overhead between CPU and GPU?
  - How to proceed the training of a gigantic network with very limited available memory?

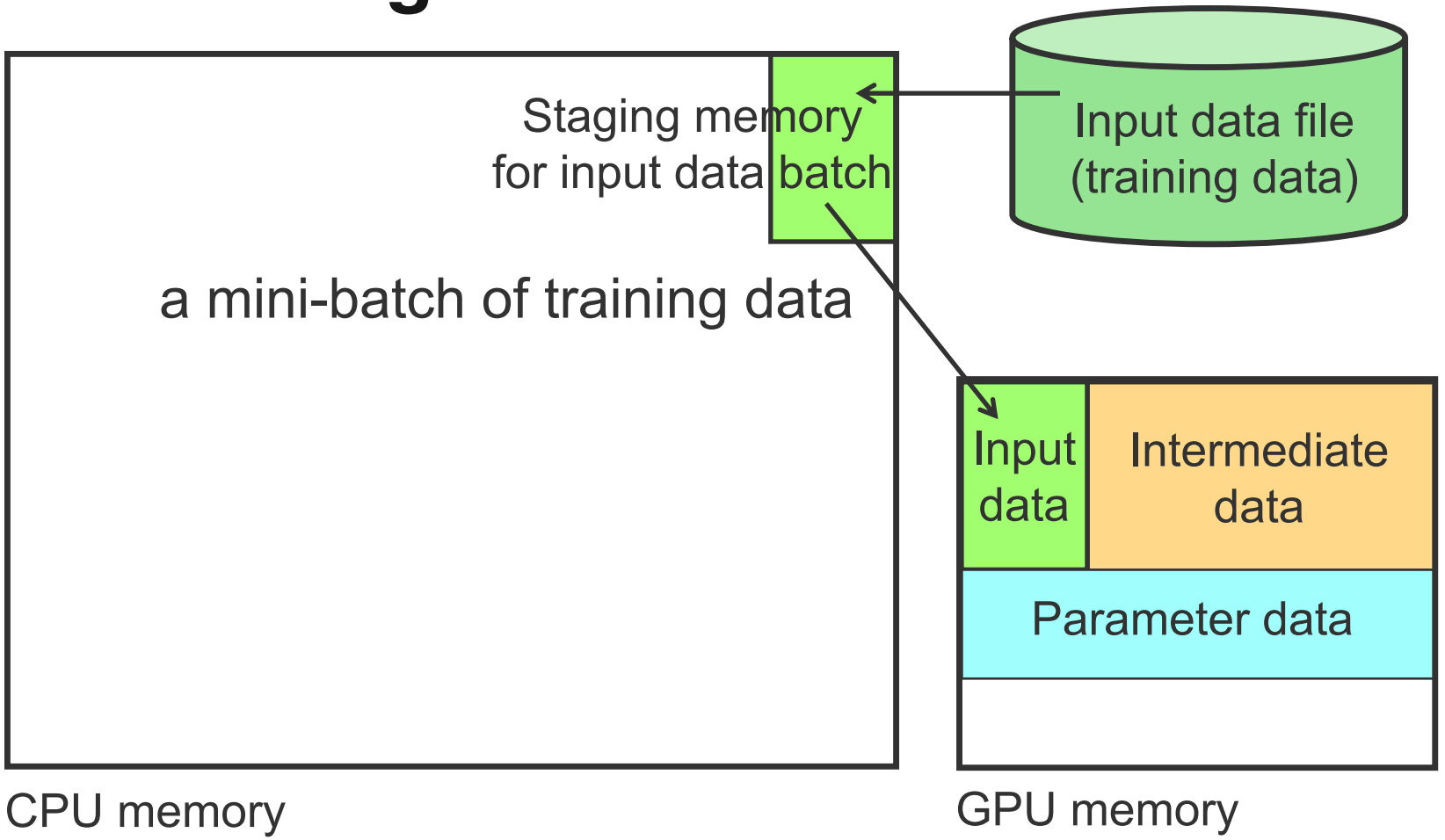
A Machine w/o GPU



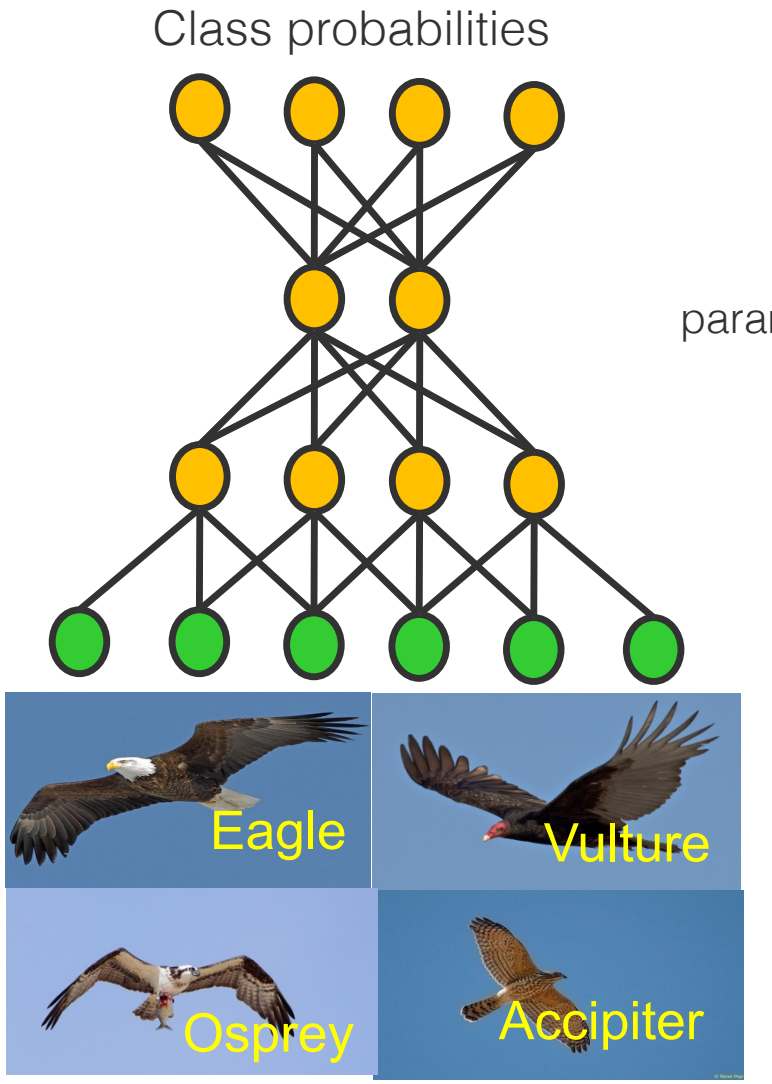
A Machine w/ GPU

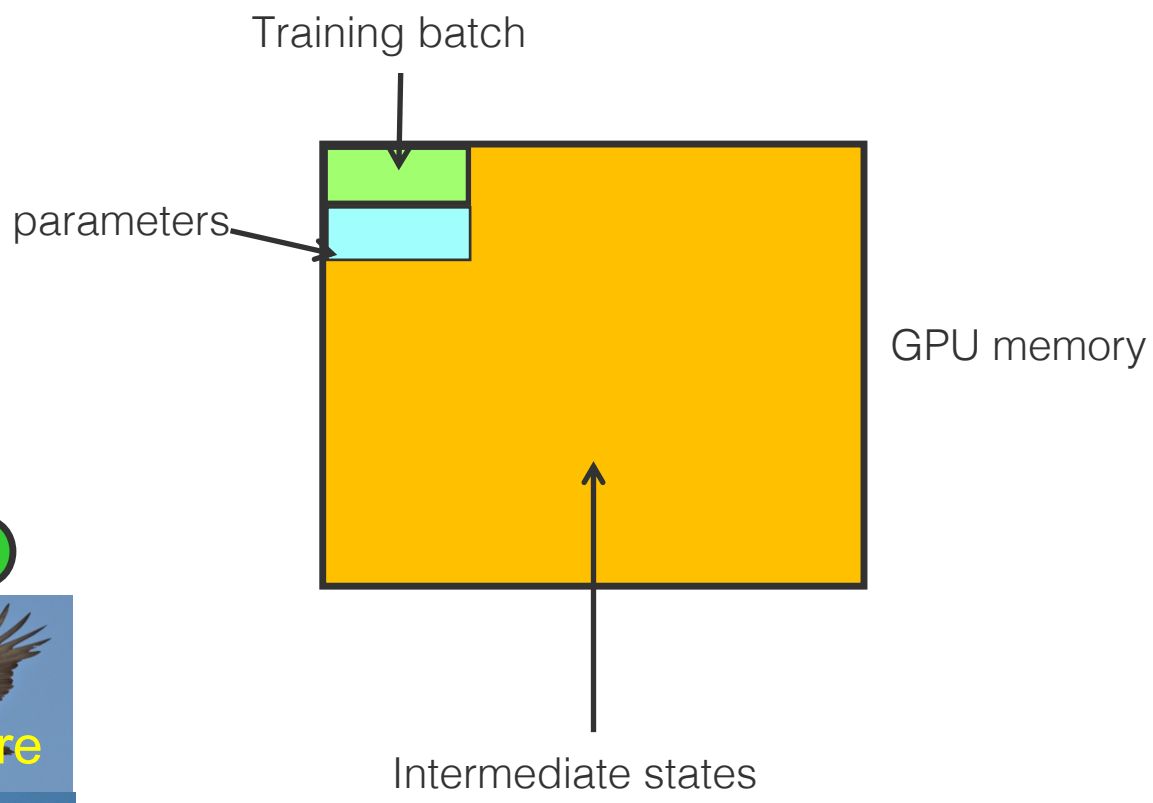


Machine Learning on GPU

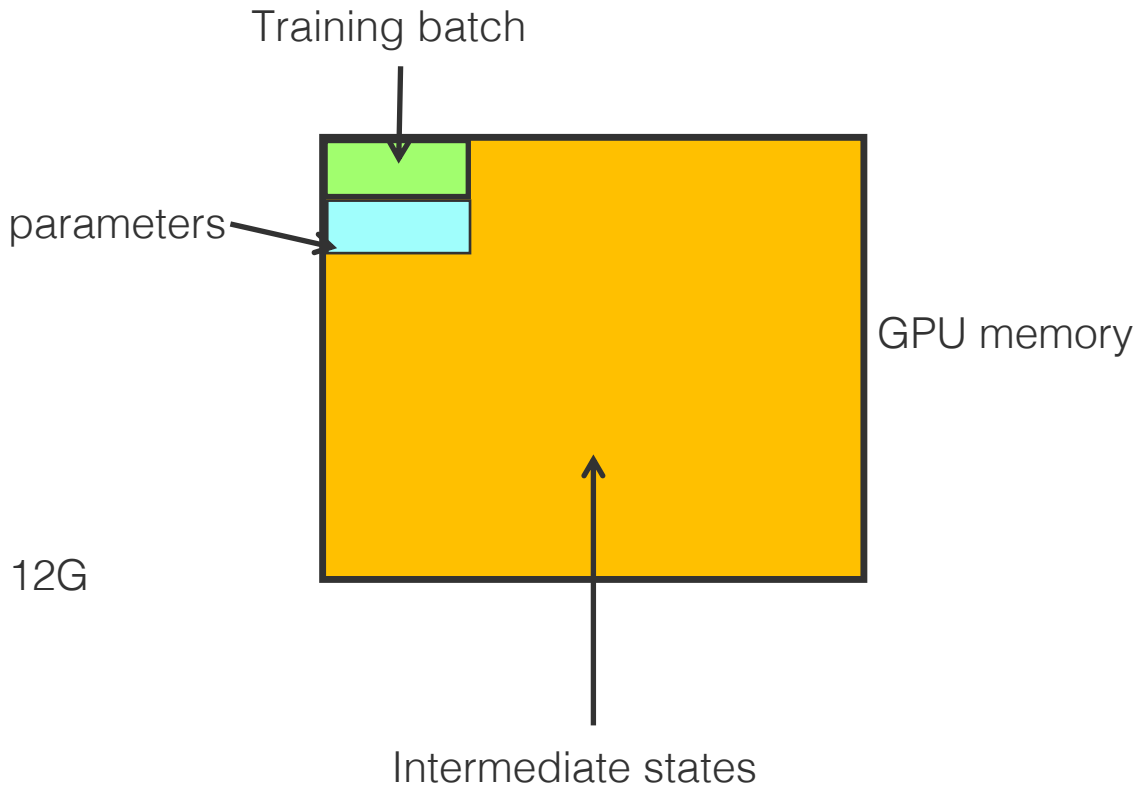


Deep Learning on GPU





Numbers



Max available GPU memory: 12G

Network	Batch size	Input size	Parameters + grads	Intermediate states
AlexNet	256	150MB	<500M	4.5G
GoogLeNet	64	19MB	<40M	10G
VGG19	16	10MB	<1.2G	10.8G

Why Memory is an Issue?

- Intermediate states occupy 90% of the GPU memory
- Intermediate states is proportional to input batch size

- However,
  - If you want high throughput, you must have large batch size (because of the SIMD nature of GPUs)
  - If you have large batch size, your GPU will be occupied by intermediate states, which thereby limits your model size/depth

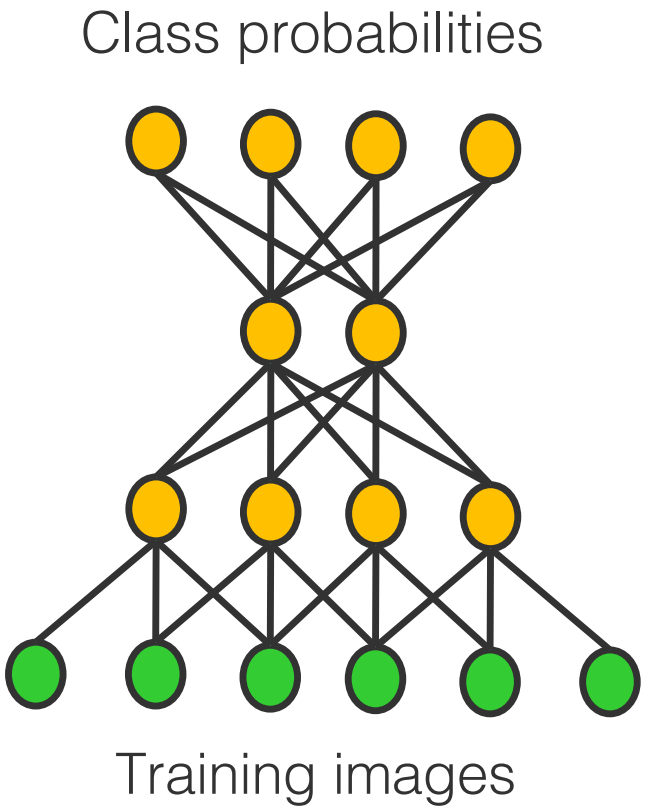
Saving Memory: A Simple Trick

- Basic idea
  - The fact: intermediate states are proportional to the batch size K
  - Idea: achieve large batch size by accumulating gradients generated by smaller batch sizes which are affordable in the GPU memory
- Solution:
    - Parition K into M parts, every part has K/M samples
    - For iter = 1:M
      - Train with mini-batchsize K/M
      - Accumulate the gradient on GPU w/o updating model parameters
    - Update the model parameter all together when all M parts finished
- Drawbacks
  - What if the GPU still cannot afford the intermediate states even if K=1?
  - Small batch size usually leads to insufficient use of GPUs' computational capability

Memory Management using CPU Memory

- Core ideas
  - If the memory is limited, trade something for memory
    - Trade extra computations for memory
    - Trade other cost (e.g. memory exchange) for more available memory
  - If the memory is limited, then get more
    - model parallel
    - CPU memory

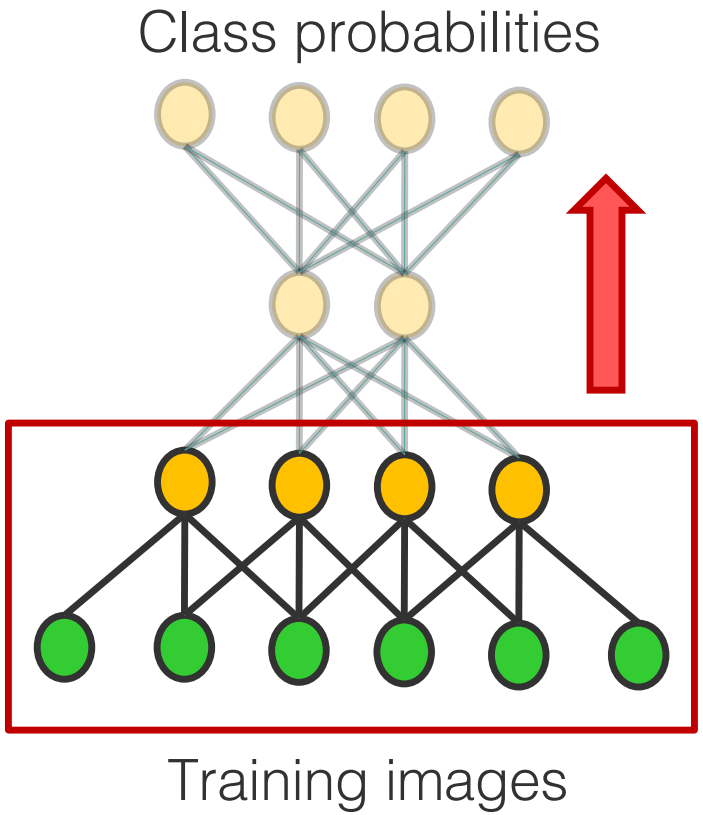
Memory Management using CPU Memory



- For each iteration (mini-batch)
  - A forward pass
  - Then a backward pass
- Each time only data of two layers are used

Cui et al., 2016

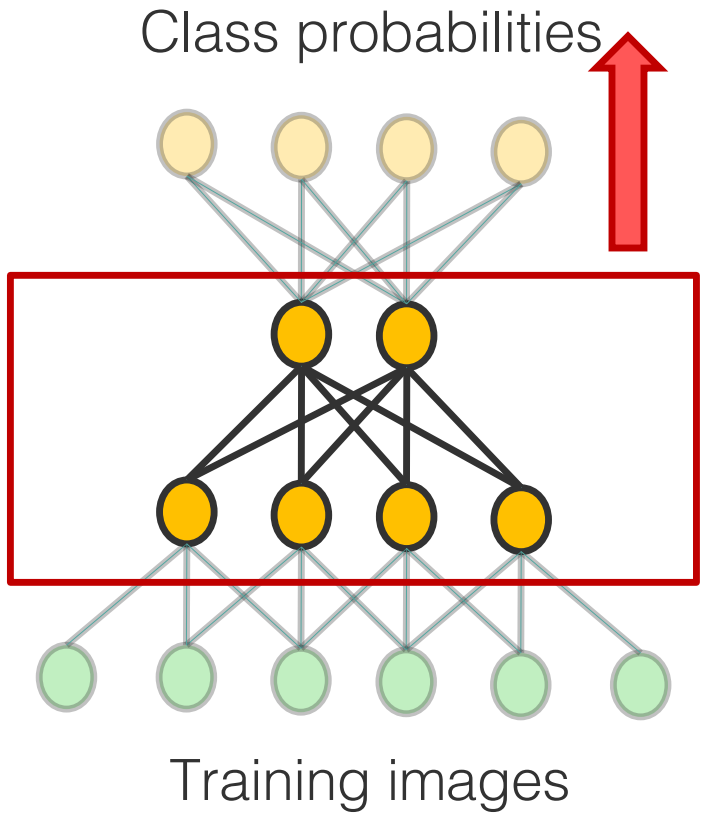
Memory Management using CPU Memory



- For each iteration (mini-batch)
  - A forward pass
  - Then a backward pass
- Each time only data of two layers are used

Cui et al., 2016

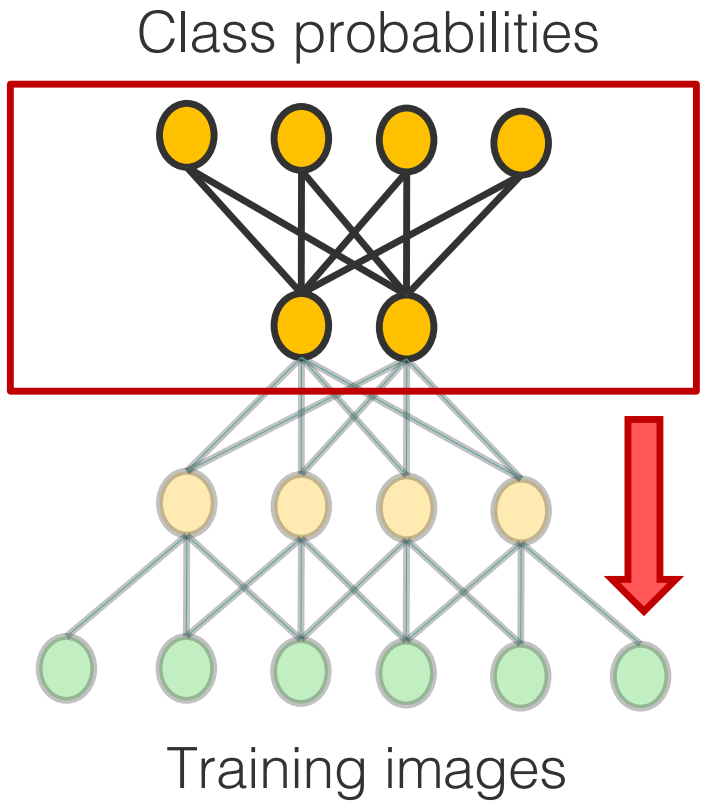
Memory Management using CPU Memory



- For each iteration (mini-batch)
  - A forward pass
  - Then a backward pass
- Each time only data of two layers are used

Cui et al., 2016

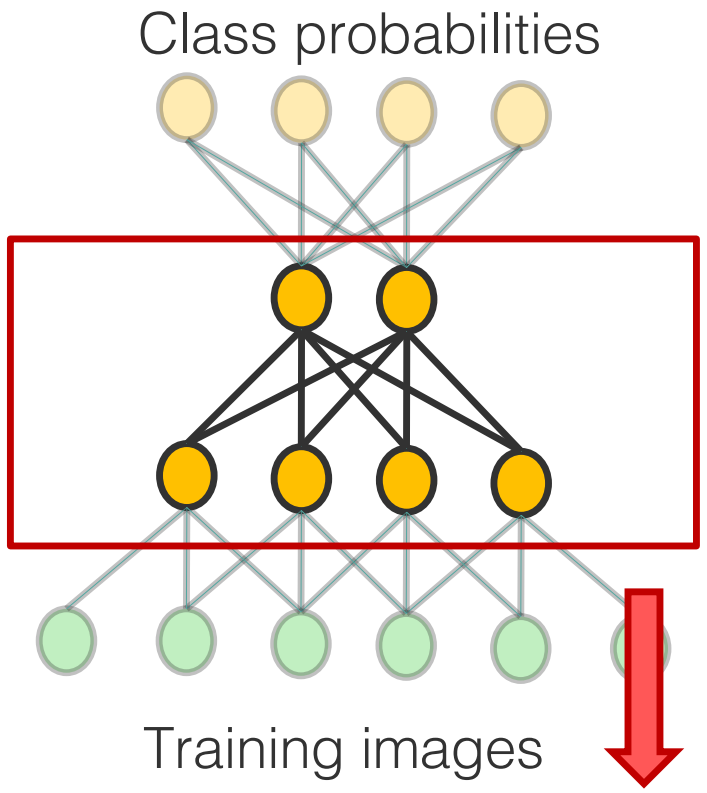
Memory Management using CPU Memory



- For each iteration (mini-batch)
  - A forward pass
  - Then a backward pass
- Each time only data of two layers are used

Cui et al., 2016

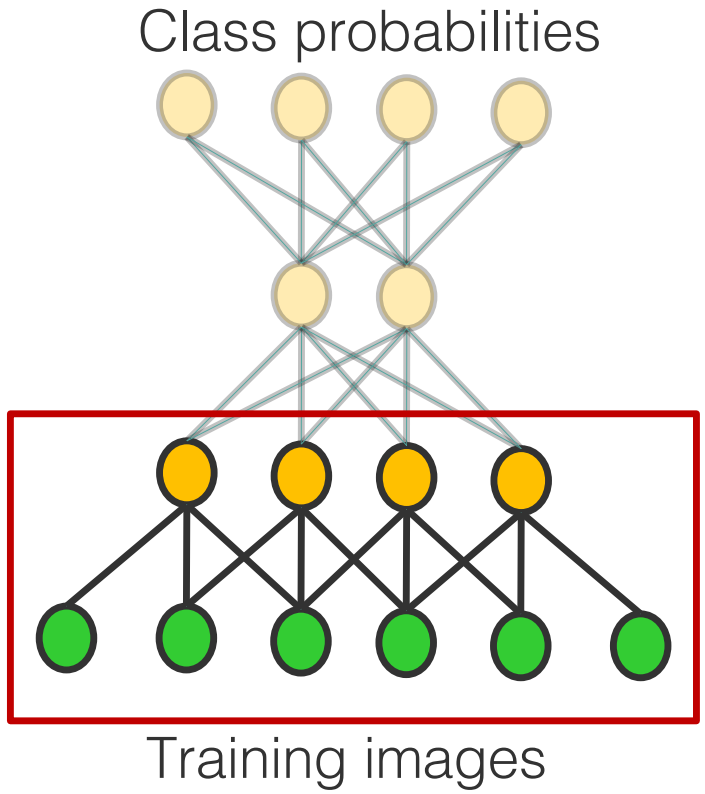
Memory Management using CPU Memory



- For each iteration (mini-batch)
  - A forward pass
  - Then a backward pass
- Each time only data of two layers are used

Cui et al., 2016

Memory Management using CPU Memory

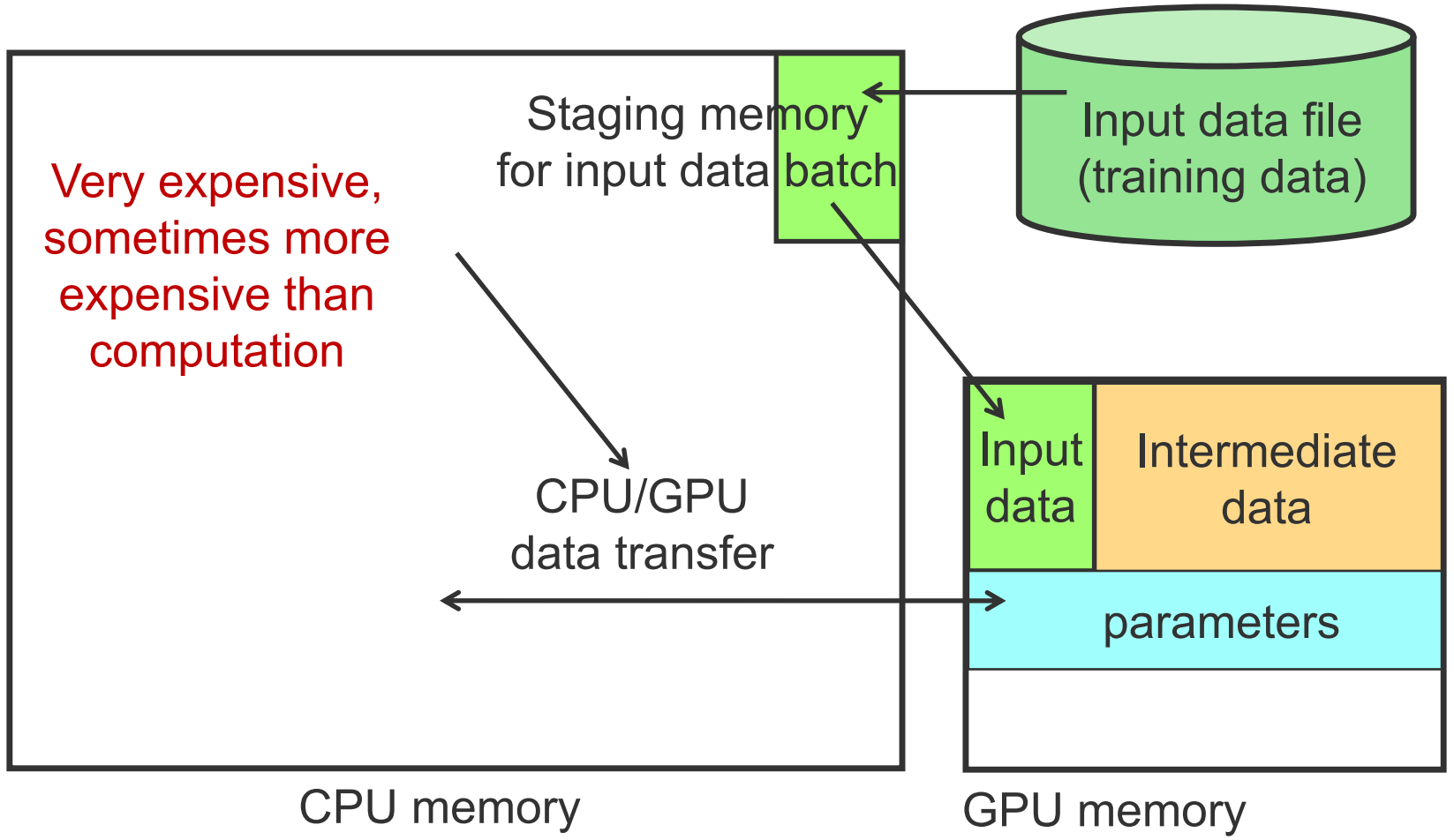


- For each iteration (mini-batch)
  - A forward pass
  - Then a backward pass
- Each time only data of two layers are used

The idea
- Use GPU mem as a cache to keep actively used data
- Store the remaining in CPU memory

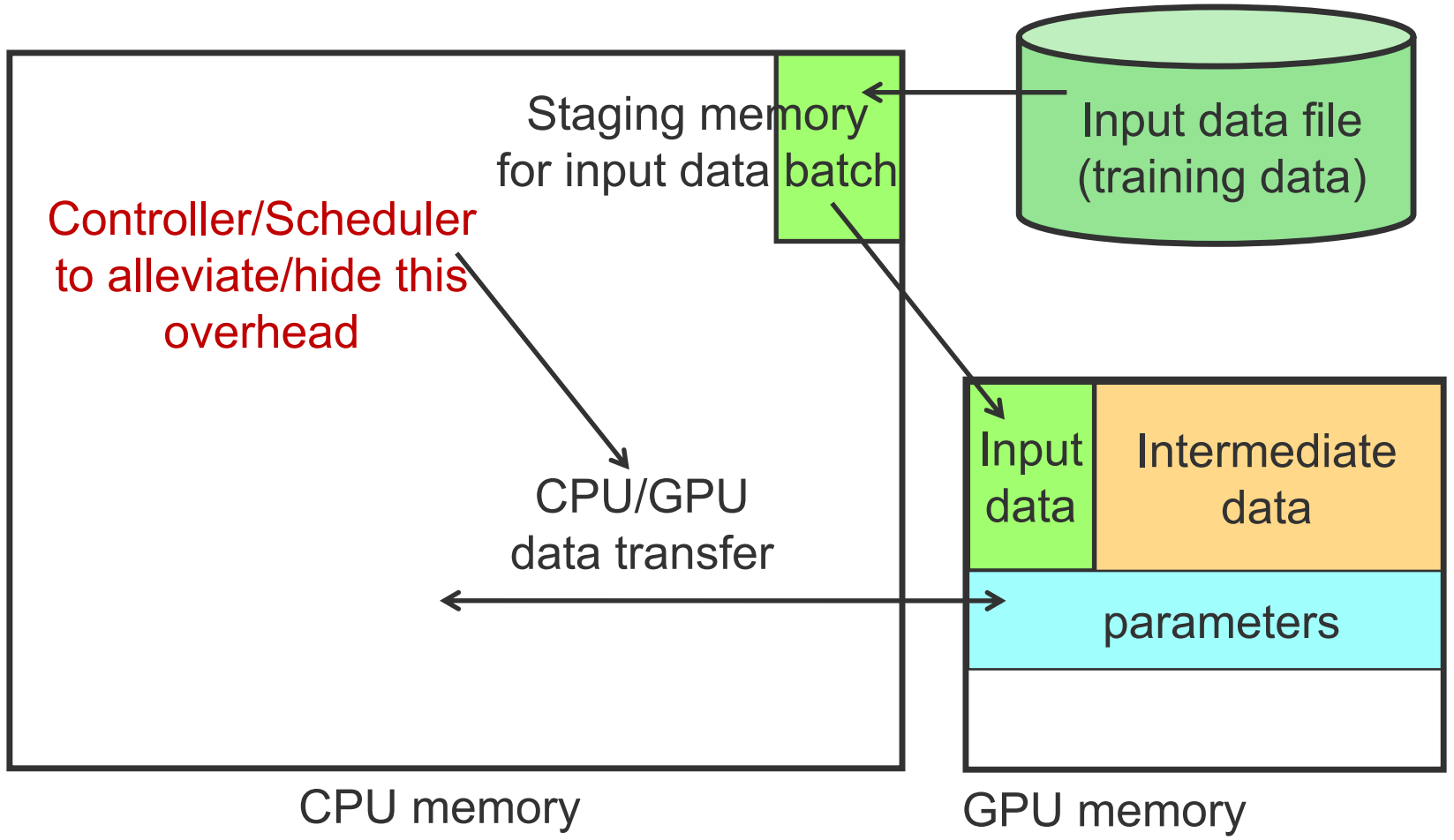
Cui et al., 2016

Memory Management using CPU Memory



Cui et al., 2016

Memory Management using CPU Memory



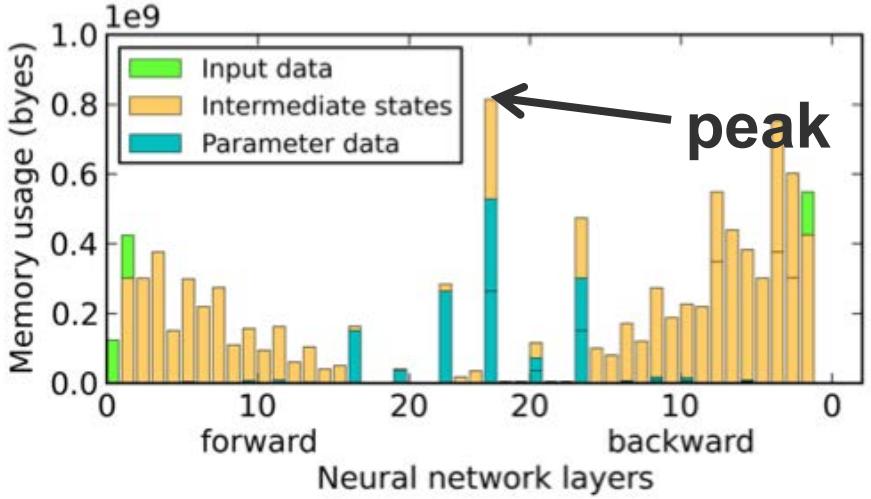
Cui et al., 2016

Memory Management using CPU Memory

- Controller
  - The fact: the memory access order is deterministic and can be exactly known by a single forward and backward pass
  - Idea:
    - Obtain the memory access order by a virtual iteration
    - Pre-fetch memory blocks from CPU to GPU
    - Overlap memory swap overhead with computation

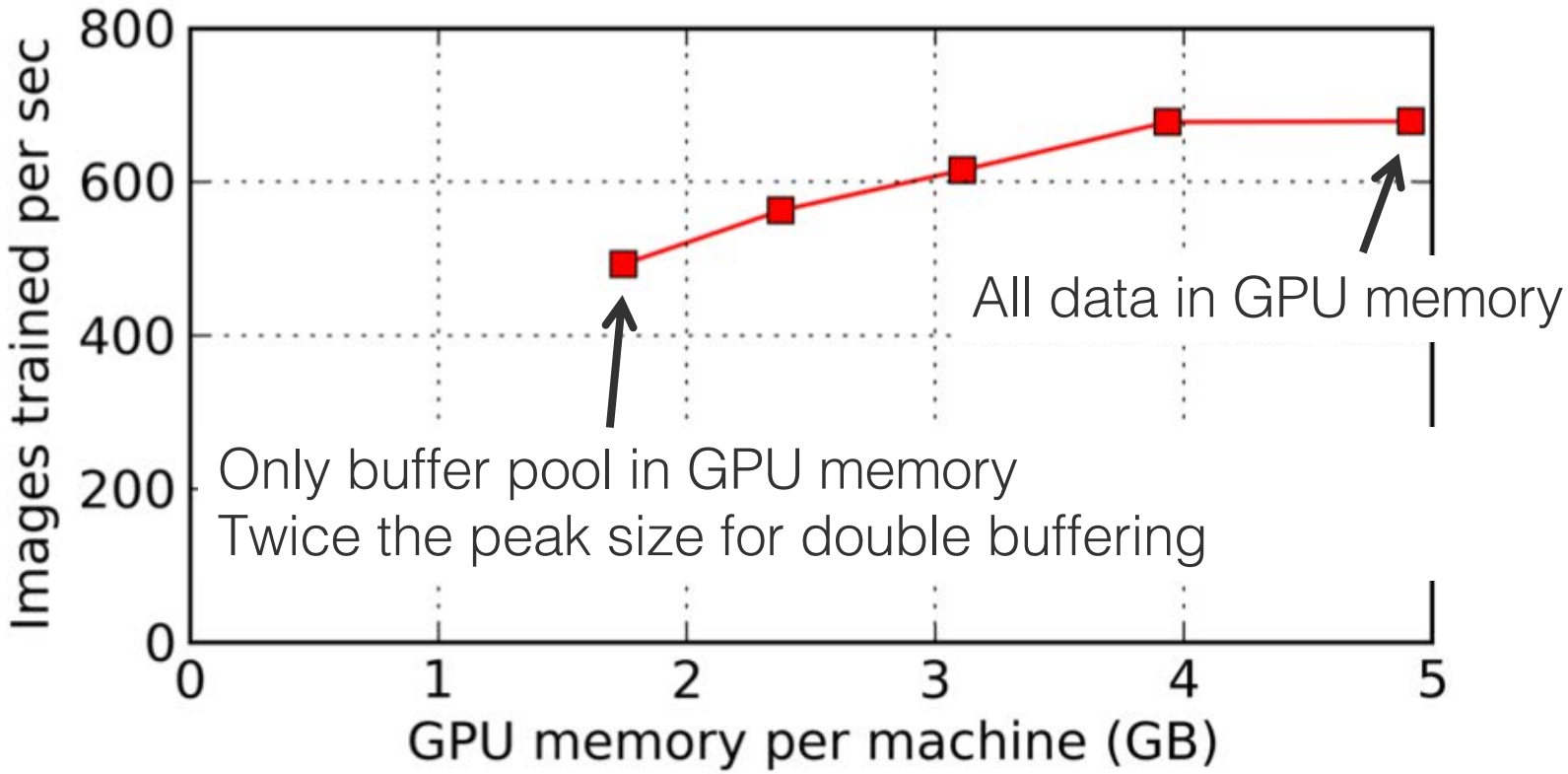
Memory Management using CPU Memory

- What's the best we can do with this strategy
  - We only need 3 memory blocks (peak size) on GPU for:
    - Input, Parameters, Output
  - The whole training can process with ONLY these three blocks by
    - Scheduling memcpy between CPU and GPU to be overlapped with computation
    - Move in and out for each layer's computation as training proceeds



Cui et al., 2016

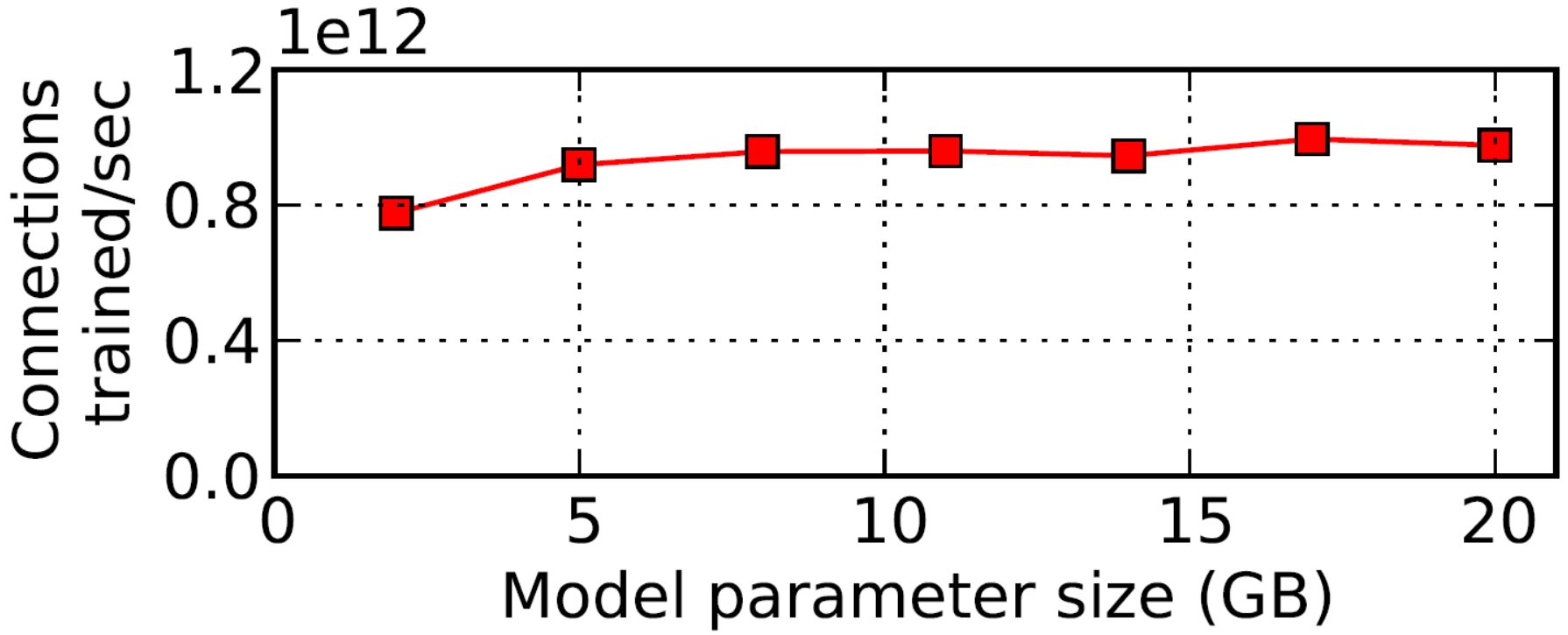
Throughput vs. memory budget



- Only 27% reduction in throughput with 35% memory
- Can do 3x bigger problems with little overhead

Cui et al., 2016

Larger models



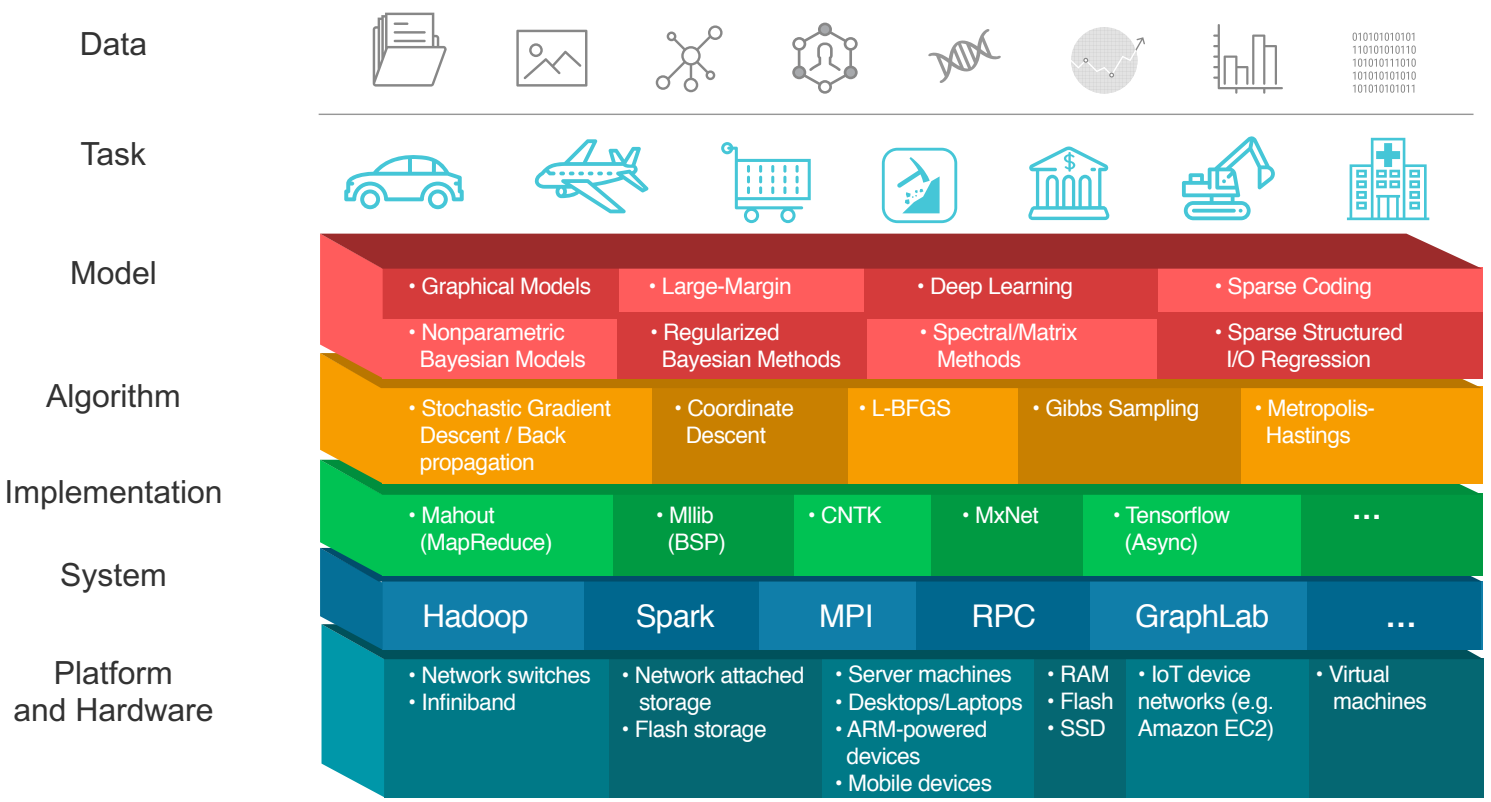
- Models up to 20 GB

Cui et al., 2016

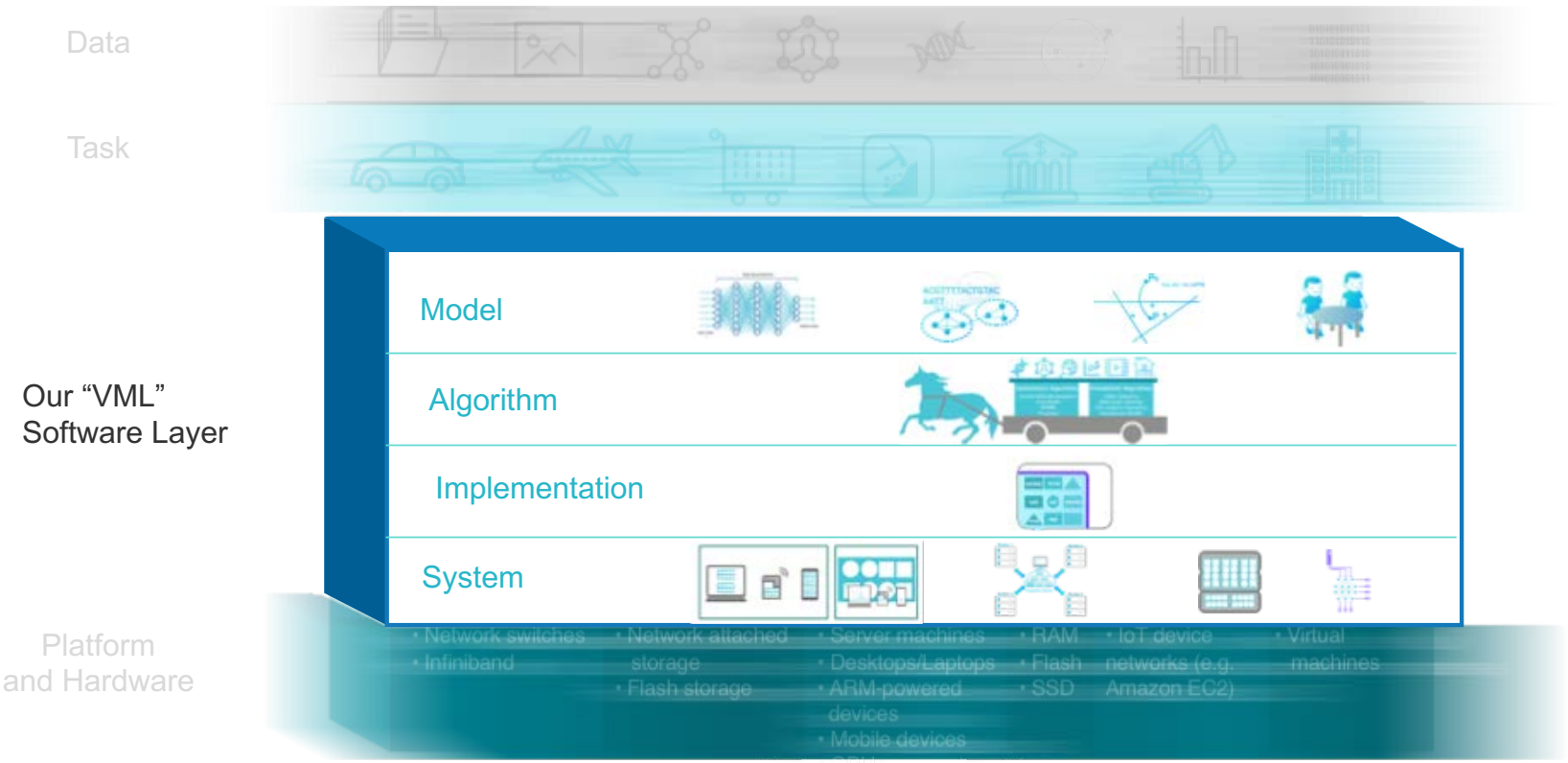
Summary

- Deep learning as dataflow graphs
- A lot of auto-differentiation libraries have been developed to train NNs
  - Different adoption, advantages, disadvantages
  - DyNet is a new framework for next-wave dynamic NNs
- Difficulties arise when scaling up DL using distributed GPUs
  - Communication bottleneck
  - Memory limit
- Poseidon as a platform to support and amplify different kinds of DL toolboxes

Elements of Modern AI



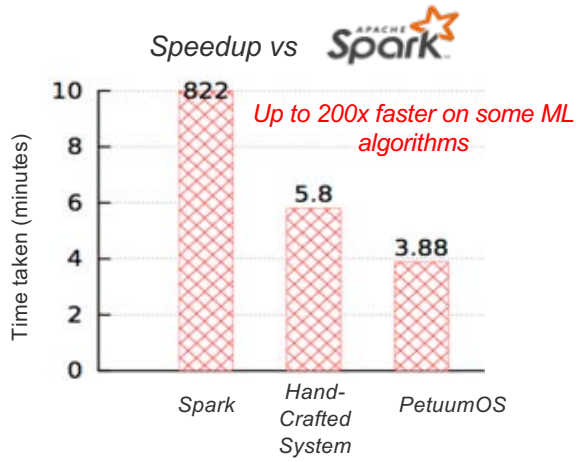
Sys-Alg Co-design Inside!

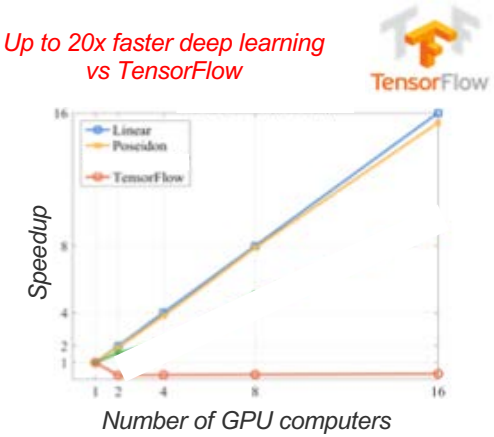


Our “VML” Software Layer

Better Performance

- Fast and Real-Time
  - Orders of magnitude faster than Spark and TensorFlow
  - As fast as hand-crafted systems
- Any Scale
  - Perfect straight-line speedup with more computing devices
  - Spark, TensorFlow can slow down with more devices
- Low Resource
  - Turning a regular cluster into a super computer:
    - Achieve AI results with much more data, but using fewer computing devices
    - Google brain uses ~1000 machines whereas Petuum uses ~10 for the same job





A Petuum Vision