CSCE 421: Machine Learning

Lecture 21: Optimizing for Neural Networks

Midterm Results

Review

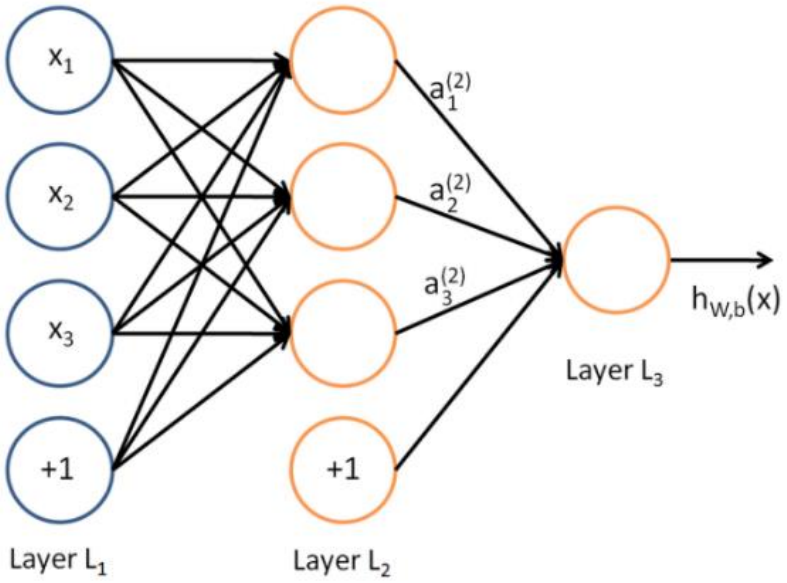
1. How can a single layer perceptron solve the xor problem?
2. What are the advantages/disadvantages of the activation functions?
3. How do we train neural networks?
4. What are the six optimization algorithms we went over last class?
5. What is dropout? Why and how do we use it?
6. What is batch normalization?
7. What are the hyperparameters associated with neural networks?
8. How can we speed up neural networks?

Outline

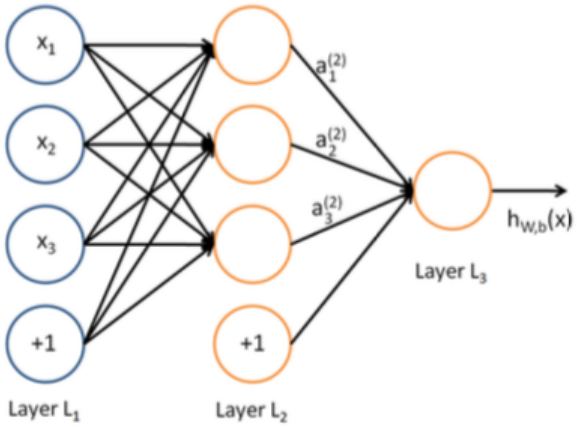
- Activation Function
- Backpropogation
- Optimization
- Neural Network Training and Design

Multilayer Perceptron

- Type of feedforward neural network
- Can model non-linear associations
- "Multi-level combination" of many perceptrons



Multilayer Perceptron: Representation



$$\alpha_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$

$$\alpha_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$

$$\alpha_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \alpha_1^{(3)} = f(W_{11}^{(2)}\alpha_1^{(2)} + W_{12}^{(2)}\alpha_2^{(2)} + W_{13}^{(2)}\alpha_3^{(2)} + b_1^{(2)})$$

Terminology

$W_{ij}^{(l)}$: connection between unit j in layer l to unit i in layer $l+1$

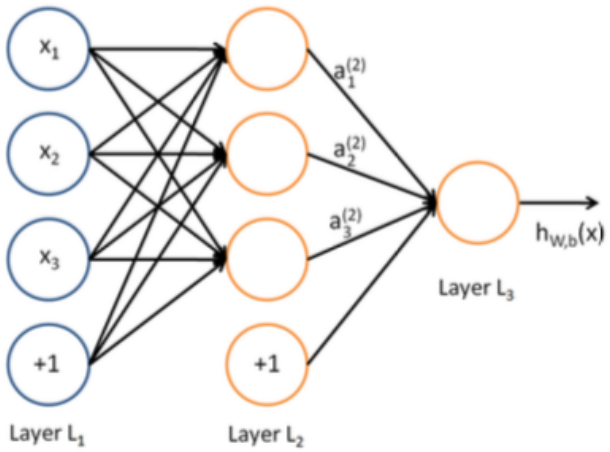
$\alpha_i^{(l)}$: activation of unit i in layer l

$b_i^{(l)}$: bias connected with unit i in layer $l+1$

Forward propagation: The process of propagating the input to the output through the activation of inputs and hidden units to each node

Multilayer Perceptron: Representation

Matrix notation



$$\alpha^{(2)} = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \alpha^{(3)} = f(\mathbf{W}^{(2)}\alpha^{(2)} + \mathbf{b}^{(2)})$$

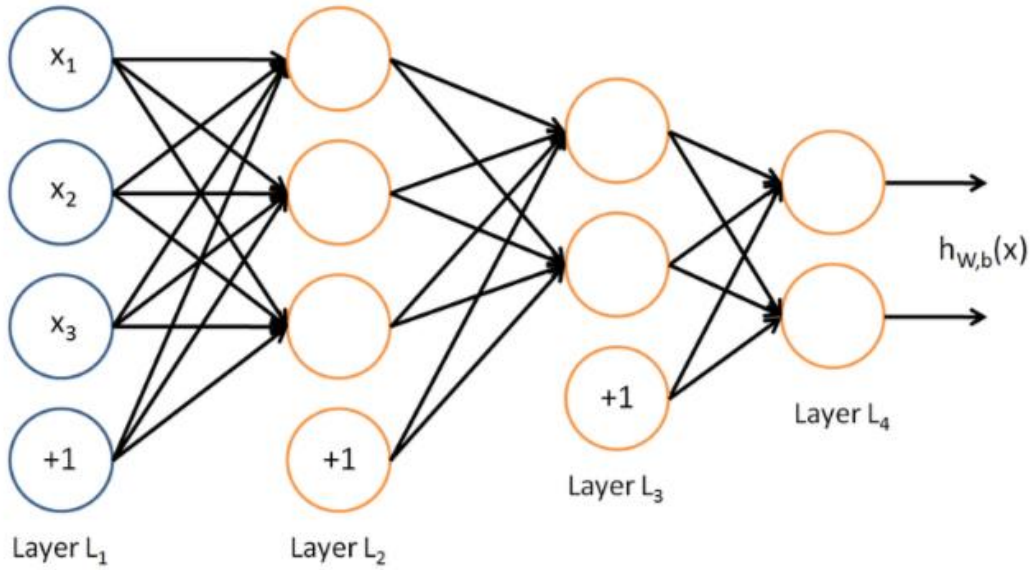
$$\mathbf{W}^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} & W_{33}^{(1)} \end{bmatrix}, \mathbf{b}^{(1)} = [b_1^{(1)}\ b_2^{(1)}\ b_3^{(1)}], \text{ etc.}$$

Multilayer Perceptron: Representation

Alternative architectures

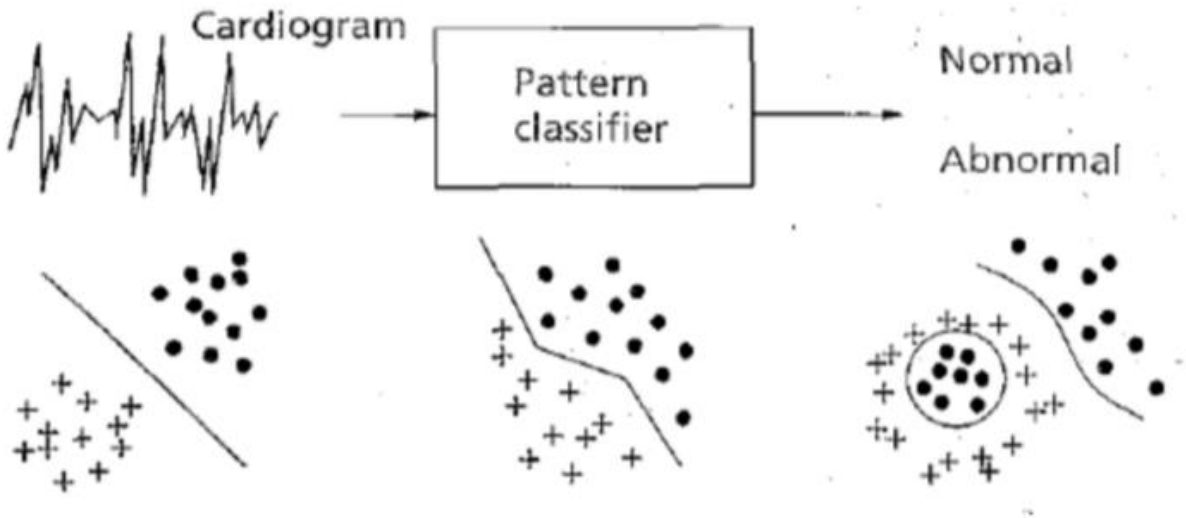
2 hidden layers, multiple output units

e.g. medical diagnosis: different outputs might indicate presence or absence of different diseases



Multilayer Perceptron

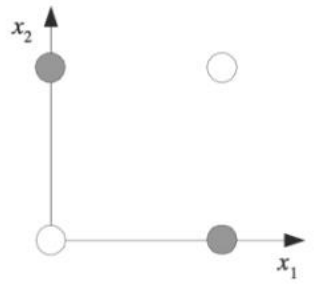
Non-linear feature learning

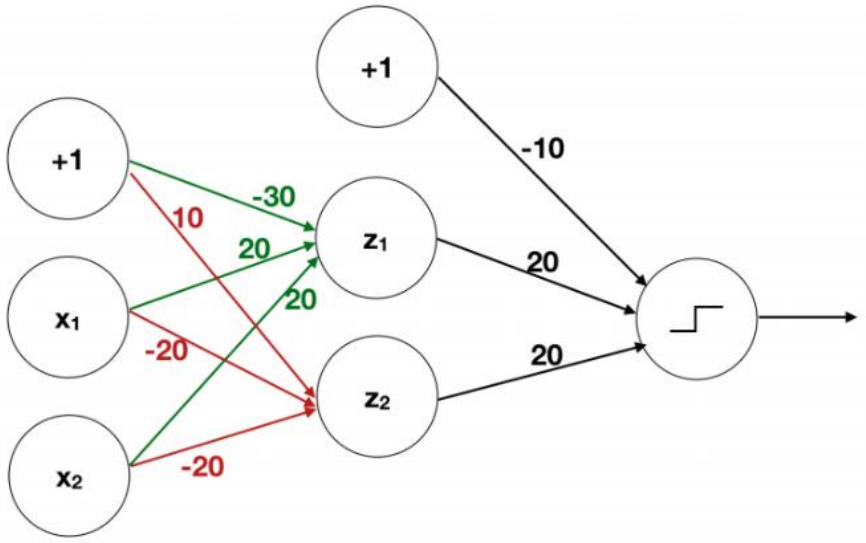


Multilayer Perceptron: Approximating non-linear functions

Example: Boolean XNOR multilayer perceptrons

x_1	x_2	z_1	z_2	r
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1





Activation Function

Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it.

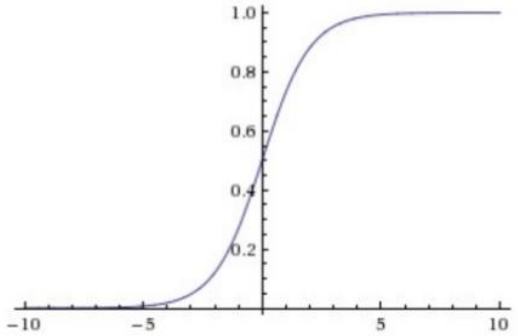
Purpose: introduce non-linearity into the output of a neuron.

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent: $s(x) = \tanh(x) = 2\sigma(2x) - 1$
- Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$
- Leaky ReLU: $f(x) = (ax) \cdot \mathbb{I}(x < 0) + (x) \cdot \mathbb{I}(x \geq 0)$ (e.g. $a = 0.01$)

Activation Function

Sigmoid: $s(x) = \frac{1}{1+e^{-x}}$

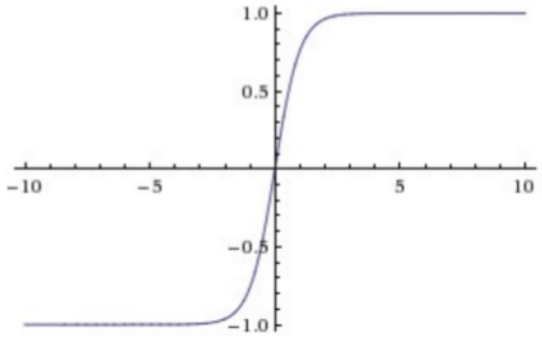
- Transforms a real-valued number between 0 and 1
- Large negative numbers become 0 (not firing at all)
- Large positive numbers become 1 (fully-saturated firing)
- Used historically because of its nice interpretation
- Saturates gradients: The gradient at either extremes (0 or 1) is almost zero, "killing" the signal will flow
- Non-zero centered output: Can be problematic during training, since it can bias outputs toward being always positive or always negative, causing unnecessary oscillations during the optimization



Activation Function

Hyperbolic tangent: $s(x) = \tanh(x) = 2\sigma(2x) - 1$

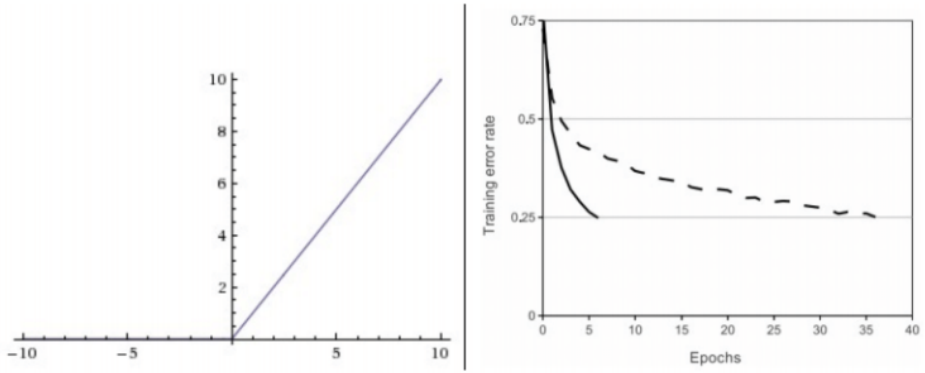
- Scaled version of sigmoid
- Transforms a real-valued number between -1 and 1
- Saturates gradients: Similar to sigmoid
- Output is zero-centered, avoiding some oscillation issues



Activation Function

Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

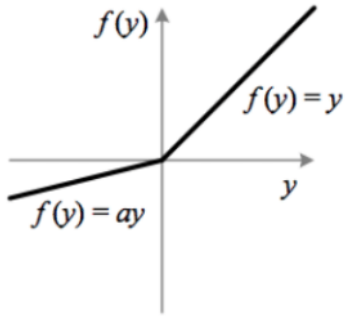
- Activation simply thresholded at zero
- Very popular during the last years
- Accelerates convergence (e.g. a factor of 6, see bellow) compared to the sigmoid/tanh (due to its linear, non-saturating form)
- Cheap implementation by simply thresholding at zero
- Activation can "die": a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again, proper adjustment of learning rate can mitigate that



Activation Function

Leaky ReLU: $f(x) = (ax) \cdot \mathbb{I}(x < 0) + (x) \cdot \mathbb{I}(x \geq 0)$

- Instead of the function being zero when $x < 0$, leaky ReLU will have a small negative slope (e.g. $a = 0.01$)
- Some successful results, but not always consistent



Backpropagation

Multilayer Perceptron: Representation

- Input: $\mathbf{x} \in \mathbb{R}^D$
- Output:
  $y \in \{0, 1\}$ or $y \in \{1, \ldots, K\}$ (classification)
  $y \in \mathbb{R}$ or $y \in \mathbb{R}^K$ (regression)
- Training data: $\mathcal{D}^{train} = \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_N}, y_N)\}$
- Model: $h_{\mathbf{W},\mathbf{b}}(\mathbf{x})$
  represented through forward propagation (see previous slides)
- Model parameters: weights $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$ and biases $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}$

Multilayer Perceptron: Evaluation criterion

$J(\mathbf{W}, \mathbf{b}, \mathcal{D}^{train}) = \frac{1}{2} \| h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) - y \|_2^2$ (regression)

$J(\mathbf{W}, \mathbf{b}, \mathcal{D}^{train}) = y \log h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) + (1 - y) \log(1 - h_{\mathbf{W},\mathbf{b}}(\mathbf{x}))$ (classification)

Backpropagation

Multilayer Perceptron: Evaluation criterion

Regression

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}\frac{1}{2}\|h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) - y_n\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

Classification

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}(y_n \log h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) + (1-y_n)\log(1-h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n})))$$

$$+ \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

We will perform gradient descent

Backpropagation

Gradient descent for regression

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}\frac{1}{2}\|h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) - y_n\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

$$W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$$
$$b_i^{(l)} := b_i^{(l)} - \alpha\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$$

Note: Initialize the parameters randomly $\rightarrow$ symmetry breaking

Use backpropagation to compute partial derivatives $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$ and $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$

Backpropagation

Implementation

- For each node i in output layer L
  - $\delta_i^{(L)} = (\alpha_i^{(L)} - y_n) f'(z_i^{(L)})$
- For each node i in layer $l = L-1, L-2, \ldots, 2$
  - Hidden nodes: $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$
- Compute the desired partial derivatives as:
  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}} = \alpha_j^{(l)} \delta_i^{(l+1)}$
  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}} = \delta_i^{(l+1)}$
- Update the weights as:
  $W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$
  $b_i^{(l)} := b_i^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$

Optimization

- Gradient Descent (GD)

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Compute gradient estimate over N examples:

3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$

4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

5: **end while**

  - Positive: Gradient estimates are stable
  - Negative: Need to compute gradients over the entire training for one update

Optimization

- Stochastic Gradient Descent (SGD)

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**
2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3: Compute gradient estimate:
4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
5: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
6: **end while**

- ϵ_k is learning rate at step k
- Sufficient condition to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

Optimization

- GD versus SGD

Batch Gradient Descent:

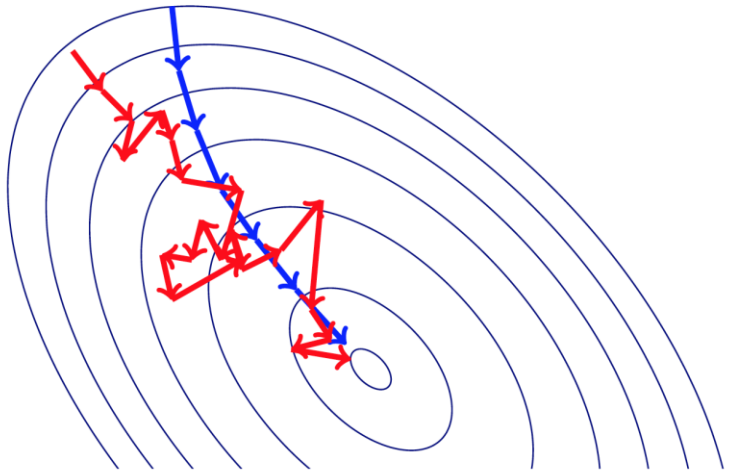
$$\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$$

$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$

SGD:

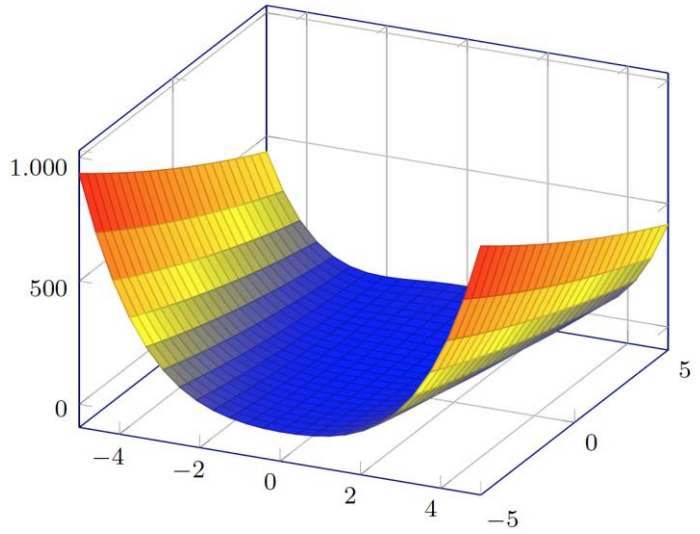
$$\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$$

$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$



Optimization

- Momentum
  - The Momentum method is a method to accelerate learning using SGD.
  - In particular SGD suffers in the following scenarios:
    - Error surface has high curvature
    - Small but consistent gradients
    - Noisy gradients



- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

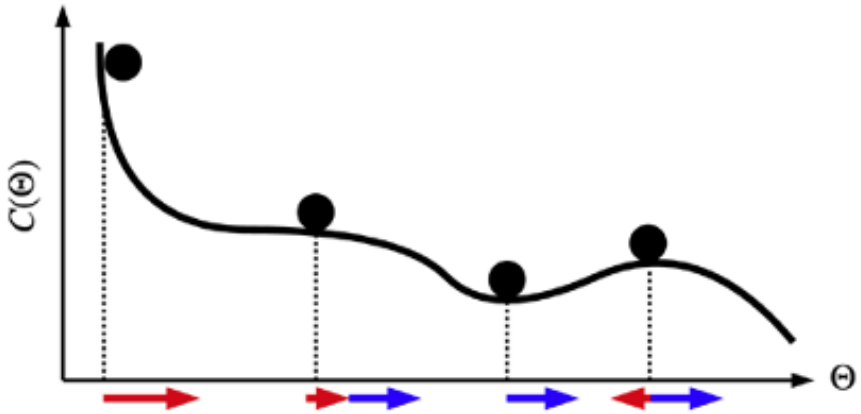
Optimization

Update rule in SGD:

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta g^{(t)}$$

where $g^{(t)} = \nabla_{\Theta} C(\Theta^{(t)})$

- Gets stuck in local minima or saddle points

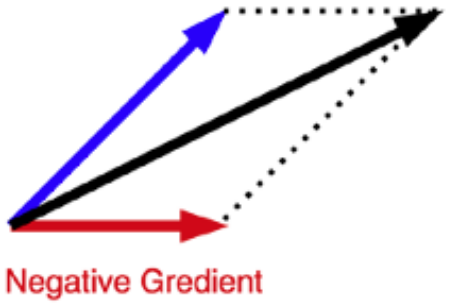


Momentum: make the same movement $v^{(t)}$ in the last iteration, corrected by negative gradient:

$$v^{(t+1)} \leftarrow \lambda v^{(t)} - (1-\lambda) g^{(t)}$$

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} + \eta v^{(t+1)}$$

$v^{(t)}$ is a moving average of $-g^{(t)}$



Optimization

- Popular Solver Examples: AdGrad, RMSProp, Adam

SGD: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

Momentum: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \mathbf{v}$

Nesterov: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$ then $\theta \leftarrow \theta + \mathbf{v}$

AdaGrad: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ then $\Delta\theta - \leftarrow \frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ then $\theta \leftarrow \theta + \Delta\theta$

RMSProp: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ then $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \Delta\theta$

Adam: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$ then $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ then $\theta \leftarrow \theta + \Delta\theta$

Optimization

- AdaGrad
  - **Idea**: Downscale a model parameter by square-root of sum of squares of all its historical values
  - Parameters that have large partial derivative of the loss -> learning rates for them are rapidly declined
  - Some interesting theoretical properties

Algorithm 4 AdaGrad

Require: Global Learning rate ϵ, Initial Parameter θ, δ

Initialize $\mathbf{r} = 0$

while stopping criteria not met **do**
  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
  Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  Accumulate: $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
  Apply Update: $\theta \leftarrow \theta + \Delta\theta$
end while

Optimization

- RMSProp
  - AdaGrad might shrink the learning rate too aggressively, we can adapt it to perform better by accumulating an exponentially decaying average of the gradient

Algorithm 5 RMSProp

Require: Global Learning rate ϵ, decay parameter ρ, δ

Initialize $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**
2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
4: Accumulate: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
5: Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
6: Apply Update: $\theta \leftarrow \theta + \Delta\theta$
7: **end while**

Optimization

- Adam
  - Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

Algorithm 7 RMSProp with Nesterov

Require: ϵ (set to 0.0001), decay rates ρ_1 (set to 0.9), ρ_2 (set to 0.9), θ, δ

Initialize moments variables $\mathbf{s} = 0$ and $\mathbf{r} = 0$, time step t $= 0$

while stopping criteria not met **do**
  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
  Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  $t \leftarrow t + 1$
  Update: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\hat{\mathbf{g}}$
  Update: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  Correct Biases: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
  Compute Update: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$
  Apply Update: $\theta \leftarrow \theta + \Delta\theta$
end while

Neural Network Training

Minibatch

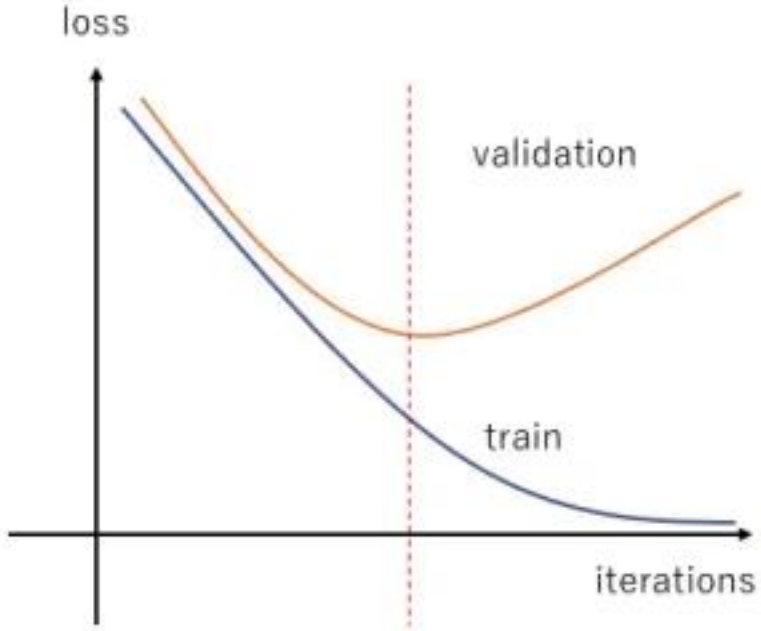
- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches (In theory, growingly larger)

- Advantage: Computation time per update does not depend on number of training examples.
- This allows convergence on extremely large datasets

- “Large Scale Learning with Stochastic Gradient Descent”, Leon Bottou.

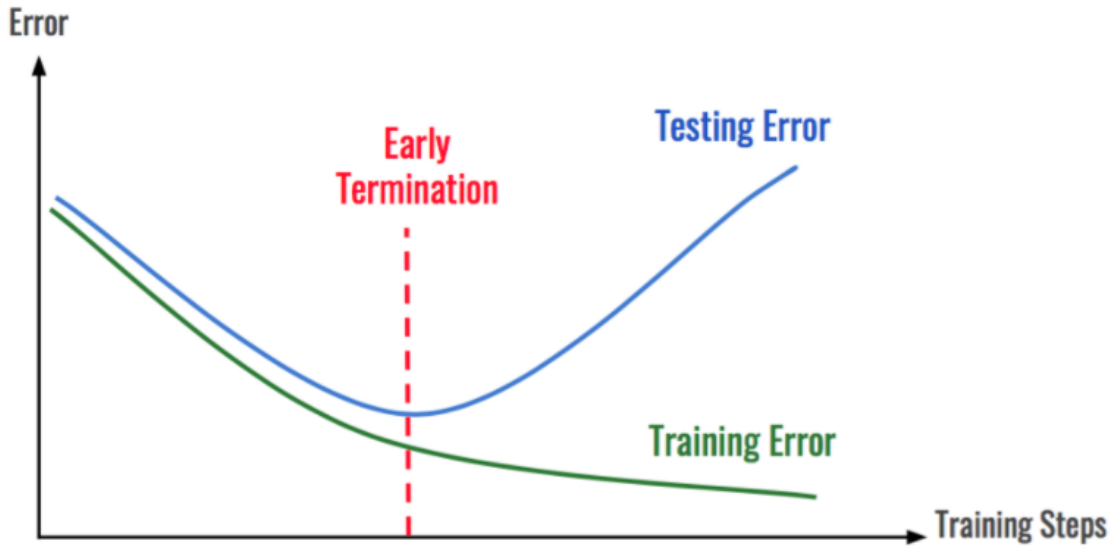
Neural Network Training

Challenge: Overfitting



Neural Network Training

- Early stop



Regularization in Deep Learning

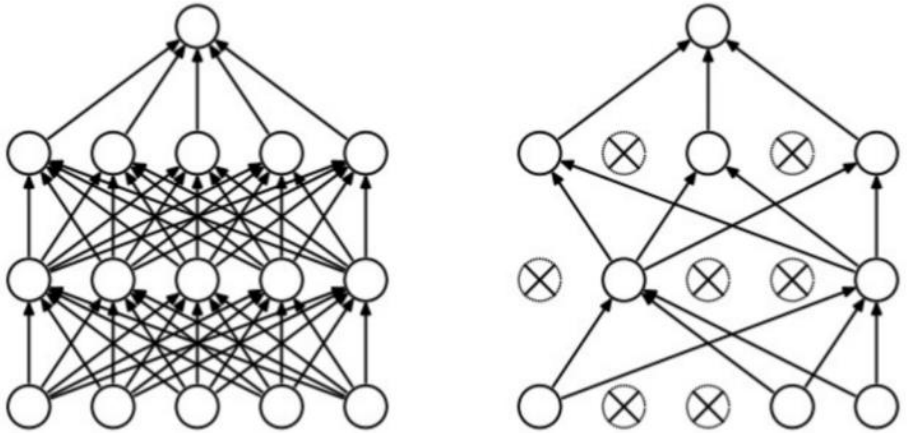
- Norm penalty/weight decay: l_2, l_1
- Add stochastic noise
- Data Augmentation
- Early Stopping
- Random Pruning: Dropout, Dropconnect, etc.
- Batch Normalization

Neural Network Training

Dropout

How to avoid overfitting

- An alternative method that complements the above is dropout
- While training, dropout keeps a neuron active with some probability p (a hyperparameter), or sets it to zero otherwise

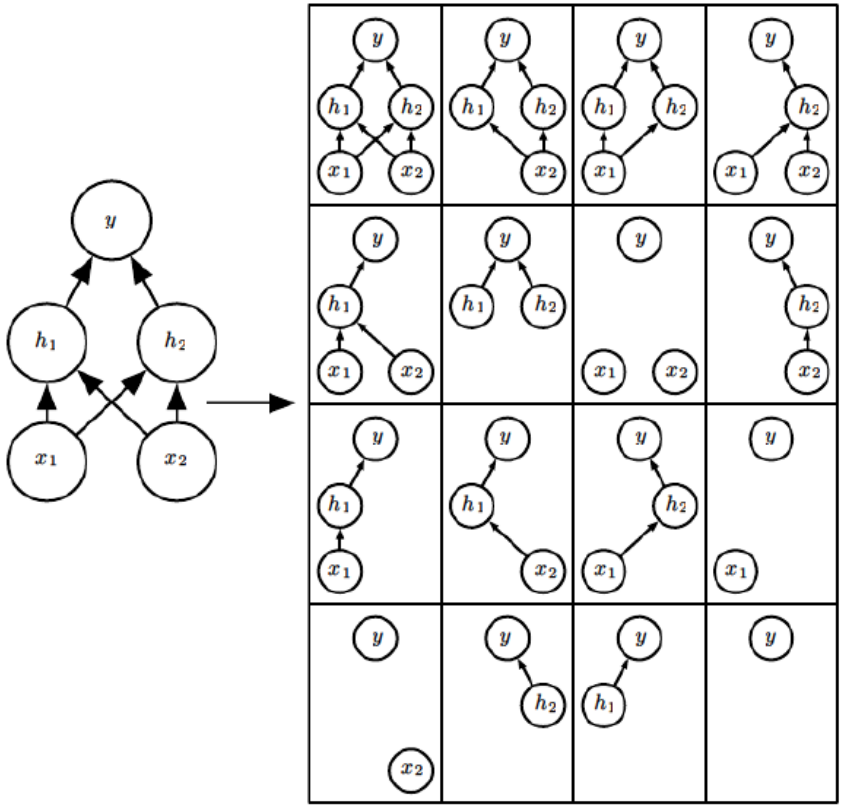


(a) Standard Neural Net

(b) After applying dropout.

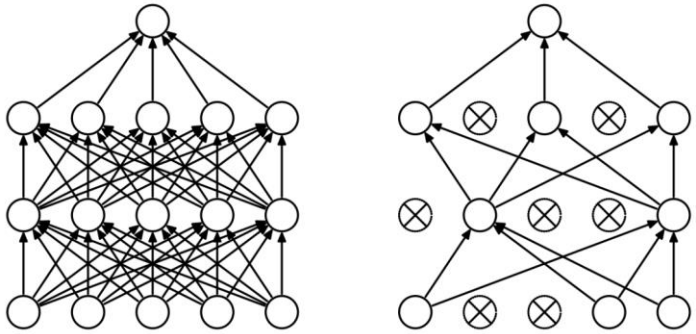
Dropout

- Randomly select weights to update
  - In each update step, randomly sample a different binary mask to all the input and hidden units
  - Multiple the mask bits with the units and do the update as usual
  - Typical dropout probability: 0.2 for input and 0.5 for hidden units
  - Very useful for FC layers, less for conv layers, not useful in RNNs



Dropout: A Stochastic Ensemble

- ***Dropout***: a feature-based bagging
  - Resamples input as well as ***latent*** features
  - With ***parameter sharing*** among voters

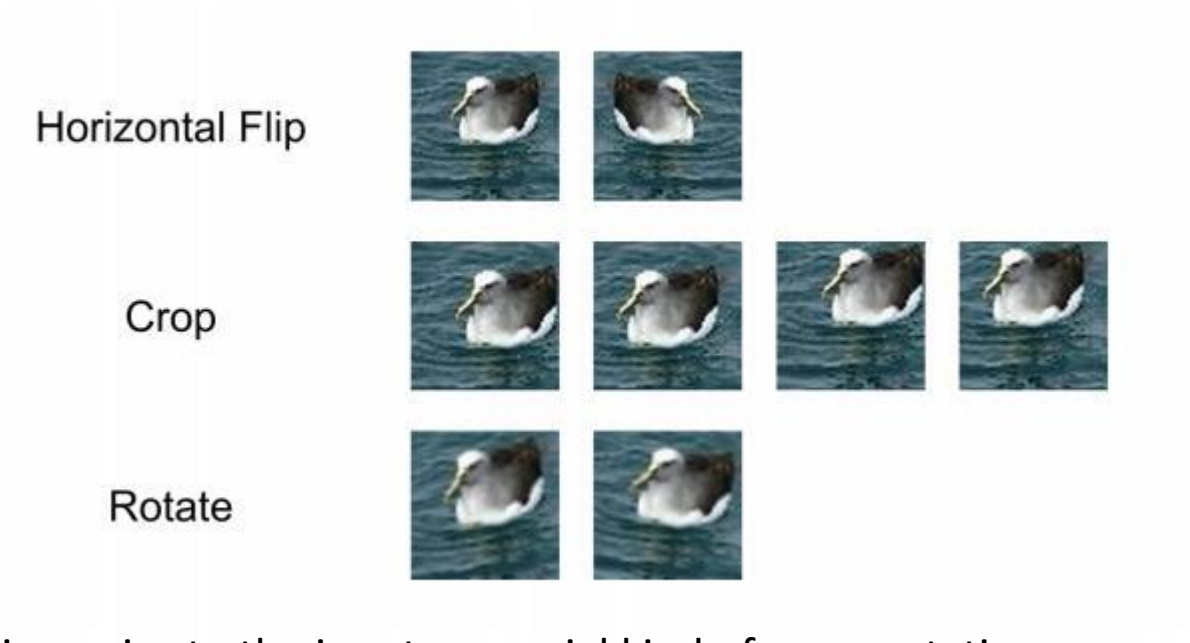


- SGD training: each time loading a minibatch, randomly sample a binary mask to apply to all input and hidden units
  - Each unit has probability α to be included (a hyperparameter)
  - Typically, 0.8 for input units and 0.5 for hidden units
- Different minibatches are used to train different parts of the NN
  - Similar to bagging, but much more efficient
  - No need to retrain unmasked units
  - Exponential number of voters

Dropout: A Stochastic Ensemble

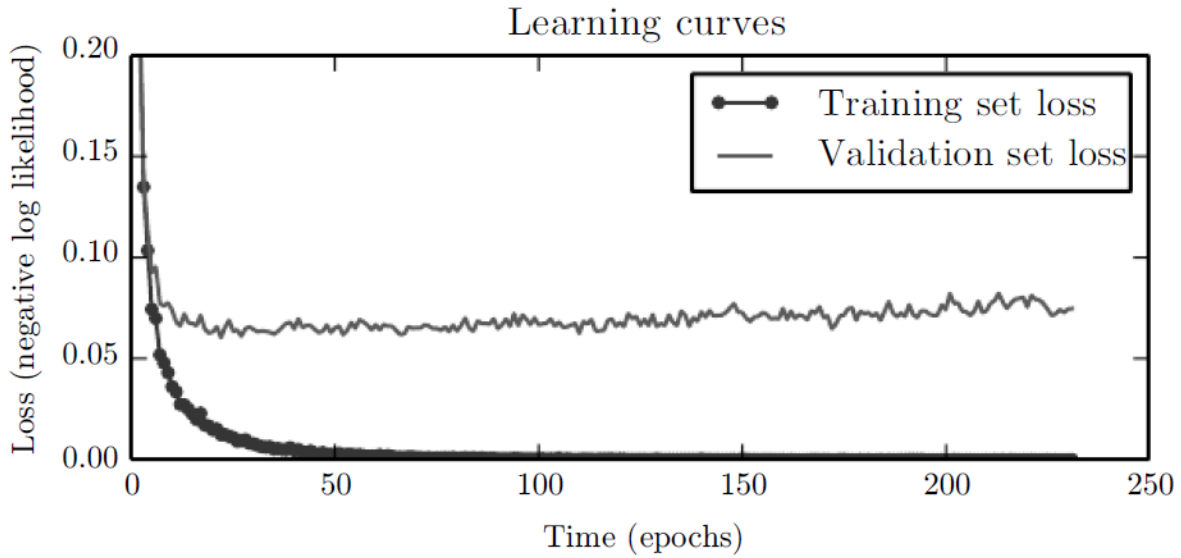
- How to vote to make a final prediction?
- Mask sampling:
  1. Randomly sample some (typically, $10 \sim 20$) masks
  2. For each mask, apply it to the trained NN and get a prediction
  3. Average the predictions
- Weigh scaling:
  - Make a single prediction using the NN with all units
  - But weights going out from a unit is multiplied by α
  - Heuristic: each unit outputs the same expected amount of weight as in training
- The better one is problem dependent

Data Augmentation



- Adding noise to the input: a special kind of augmentation
- Be careful about the transformation applied -> label preserving
  - **Example:** classifying ‘b’ and ‘d’; ‘6’ and ‘9’

Early Stopping



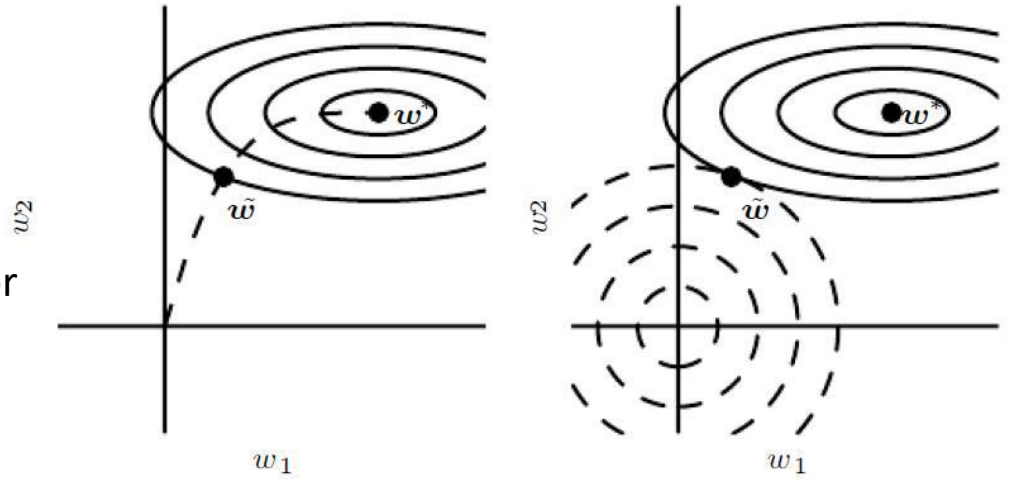
- Idea: don't train the network to too small training error
- Recall overfitting: Larger the hypothesis class, easier to find a hypothesis that fits the difference between the two
- Prevent overfitting: use validation error to decide when to stop

Early Stopping

In Practice:

- When training, also output validation error
- Every time validation error improved, store a copy of the weights
- When validation error not improved for some time, stop
- Return the copy of the weights stored
- Training step is the hyperparameter

Early Stopping as Regularization



Batch Normalization

- In ML, we assume future data will be drawn from same probability distribution as training data
- For a hidden layer, after training, the earlier layers have new weights and hence may generate a new distribution for the next hidden layer
- We want to reduce this internal covariate shift for the benefit of later layers

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

- First three steps are just like standardization of input data, but with respect to only the data in mini-batch.
- We can take derivative and incorporate the learning of last step parameters into backpropagation.
- Note last step can completely un-do previous 3 steps
- But even if so, this un-doing is driven by the later layers, not the earlier layers; later layers get to “choose” whether they want standard normal inputs or not

Neural Network Design

How to chose the number of layers and nodes

- No general rule of thumb, this depends on:
  - Amount of training data available
  - Complexity of the function that is trying to be learned
  - Number of input and output nodes
- If data is linearly separable, you don't need any hidden layers at all
- Start with one layer and hidden nodes proportional to input size
- Gradually increase

Hyperparameter Tuning

- Learning rate: how much to update the weight during optimization
- Number of epochs: number of times the entire training set pass through the neural network
- Batch size: the number of times the entire training set pass through the neural network
- Activation function: the function that introduces non-linearity to the model (e.g. sigmoid, tanh, ReLU, etc.)
- Number of hidden layers and units
- Weight initialization: Uniform distribution usually works well
- Dropout for regularization: probability of dropping a unit

We can perform grid or randomized search over all parameters

Challenge

High memory requirements

- Memory is used to store input data, weight parameters and activations as an input propagates through the network
- Activations from a forward pass must be retained until they can be used to calculate the error gradients in the backwards pass
- Example: 50-layer neural network
  - 26 million weight parameters, 16 million activations in the forward pass
  - 168MB memory (assuming 32-bit float)

Parallelize computations with GPU (graphics processing units)

Challenge

Backpropagation does not work well

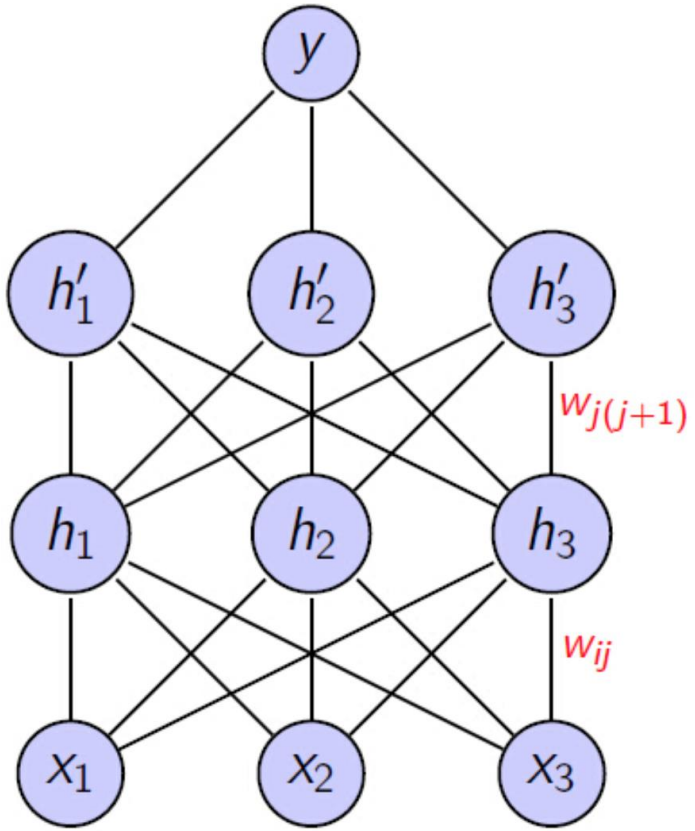
- Deep networks trained with backpropagation (without unsupervised pretraining) perform worse than shallow networks
- Gradient is progressively getting more dilute
  - Weight correction is minimal after moving back a couple of layers
- High risk of getting "stuck" to local minima
- In practice, a small portion of data is labelled

Perform pretraining to mitigate this issue

	train.	valid.	test
DBN, unsupervised pre-training	0%	1.2%	1.2%
Deep net, auto-associator pre-training	0%	1.4%	1.4%
Deep net, supervised pre-training	0%	1.7%	2.0%
Deep net, no pre-training	.004%	2.1%	2.4%
Shallow net, no pre-training	.004%	1.8%	1.9%

(Bengio et al., NIPS 2007)

Why are deep nets hard to train?



- Vanishing gradient problem in backward propagation

$$\frac{\partial Loss}{\partial w_{ij}} = \frac{\partial Loss}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} = \delta_j x_i$$

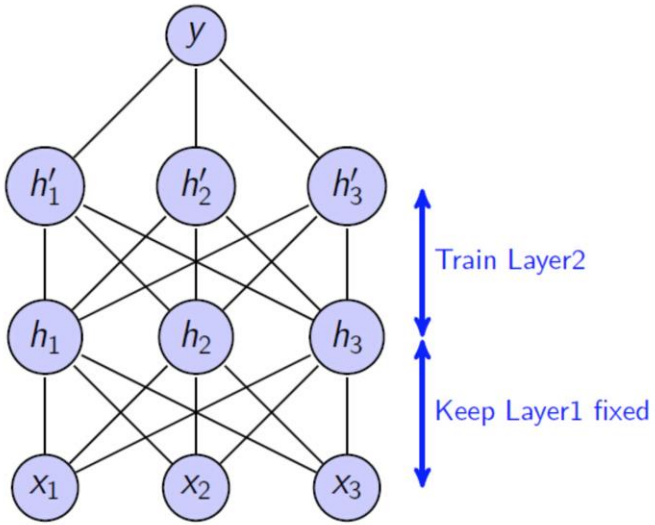
$$\delta_j = \left[\sum_{j+1} \delta_{j+1} w_{j(j+1)}\right] \sigma'(in_j)$$

δ_j may vanish after repeated multiplication

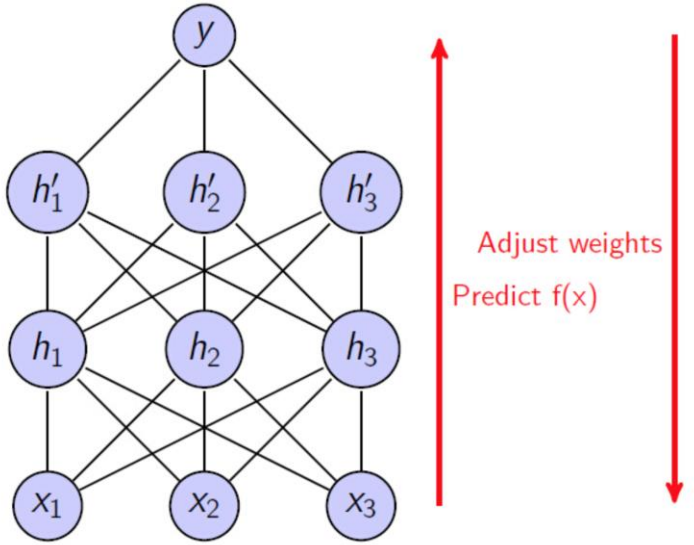
- Insufficient training data

Layer-wise Pre-training [Hinton et al. 2006]

First, train one layer at a time, optimizing data-likelihood objective $P(x)$



Finally, fine-tune labeled objective $P(y|x)$ by Backpropagation

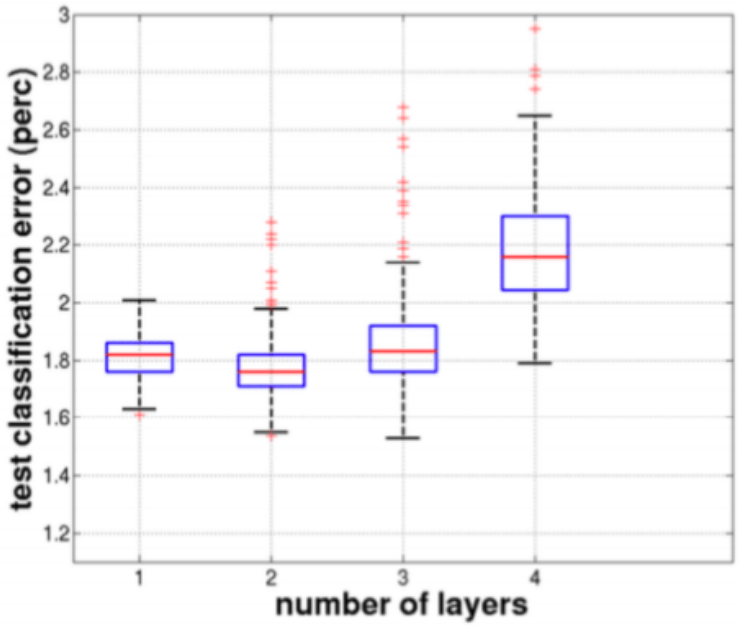


Unsupervised Pretraining

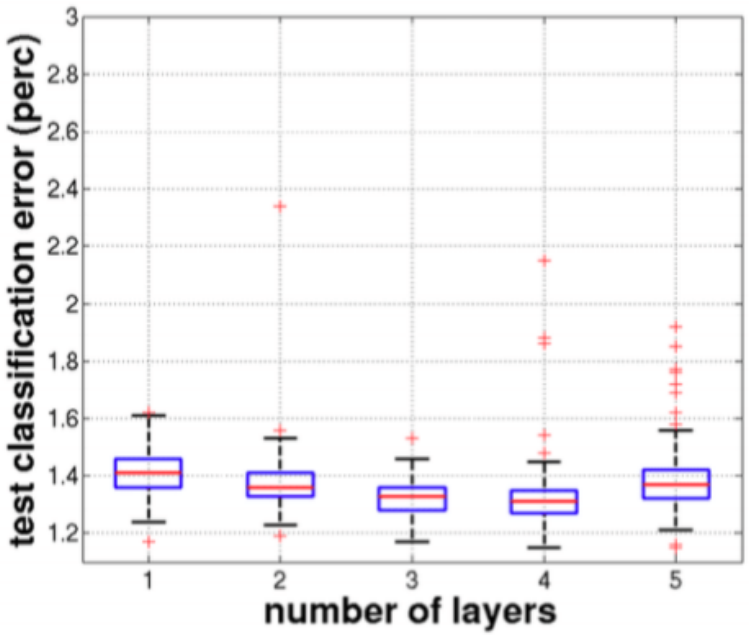
- This idea came into play when research studies found that a DNN trained on a particular task (e.g. object recognition) can be applied on another domain (e.g. object subcategorization) giving state-of-the-art results
- **1st part: Greedy layer-wise unsupervised pre-training**
  - Each layer is pre-trained with an unsupervised learning algorithm
  - Learning a nonlinear transformation that captures the main variations in its input (the output of the previous layer)
- **2nd part: Supervised fine-tuning**
  - The deep architecture is fine-tuned with respect to a supervised training criterion with gradient-based optimization
- We will examine the deep belief networks and stacked autoencoders

Unusual form of regularization: minimizing variance and introducing bias towards configurations of the parameter space that are useful for unsupervised learning

Unsupervised Pretraining



Without pre-training

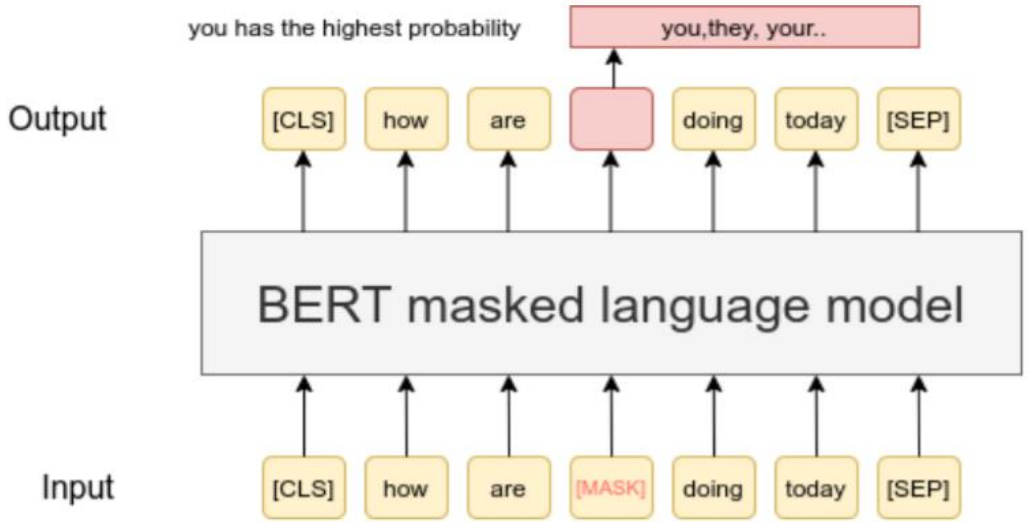


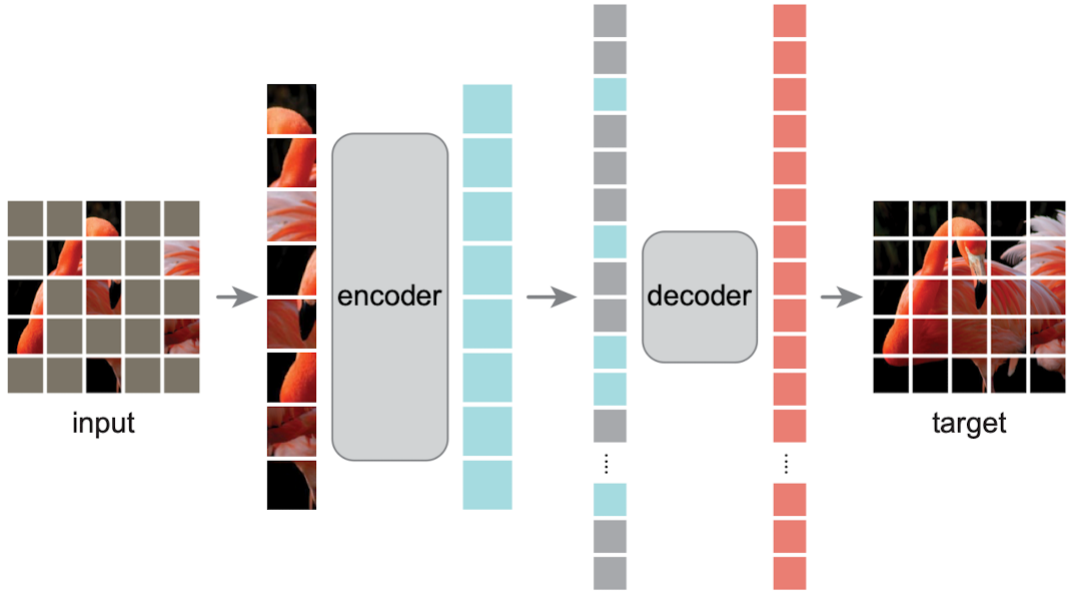
With pre-training

[Source: Erhan et al., 2010]

Unsupervised Pretraining

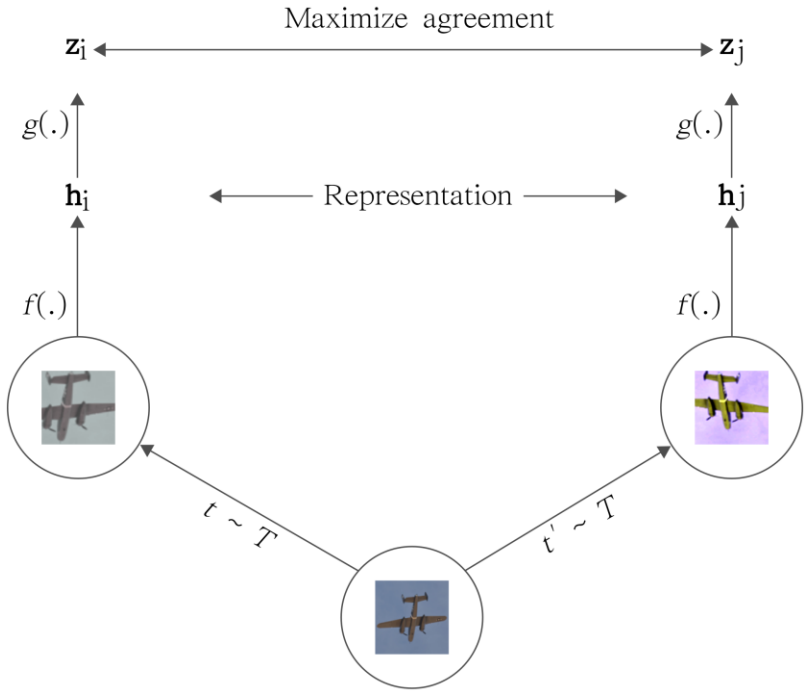
- Masked Pretraining
  - Mask some part of the input
  - Get the model to predict the masked portion





Unsupervised Pretraining

- Joint-Embedding Self-Supervised Pretraining
  - Create two different views of an instance
  - Train a model to maximize the agreement between the views



Deep learning made practical

- (More) tips for deep learning training
- Distributed Implementation of deep learning
- Compressing deep model size
- Energy-efficient deep learning

Review: Training Techniques We Learned

- Different solvers: naïve SGD, AdaGrad, RMSProp, Adam …
- Pre-training
- Norm penalty/weight decay: l_2, l_1
- Data Augmentation
- Early Stopping
- Random Pruning: Dropout, Dropconnect, etc.
- Batch Normalization

Data Pre-processing

- **Simplest: zero-center** the data, and then **normalize** them
  - It only makes sense to apply this pre-processing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm.
  - In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional pre-processing step.

- **PCA Whitening**: the data is first centered, and then projected into the eigen basis to decorrelate the data, followed by dividing every dimension by the corresponding eigenvalue to normalize the scale.

- Never forget to **shuffle** your data each epoch for SGD input!!

Monitor Your Training Curve

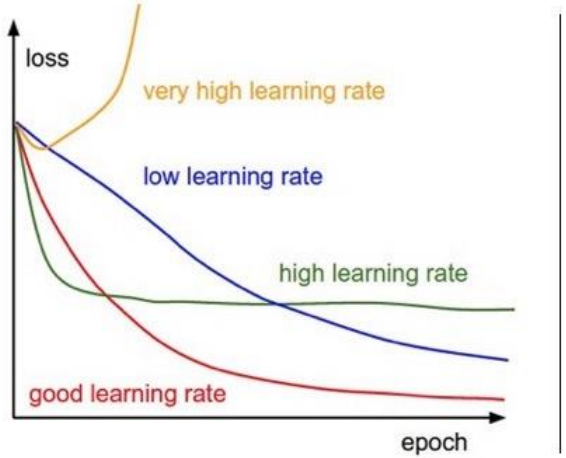


Figure 1

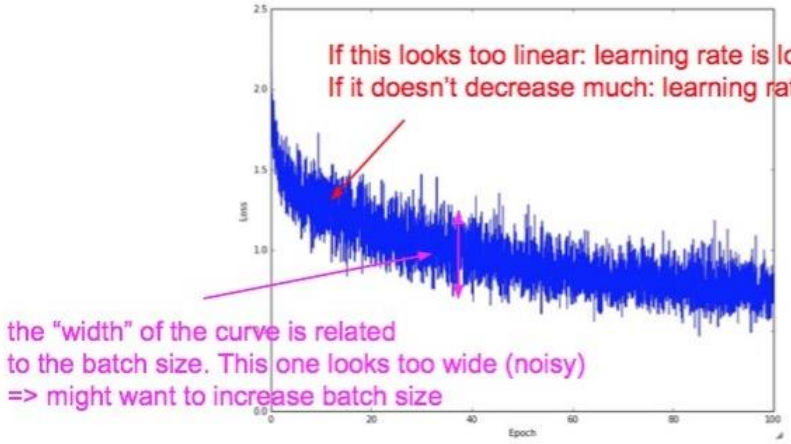


Figure 2

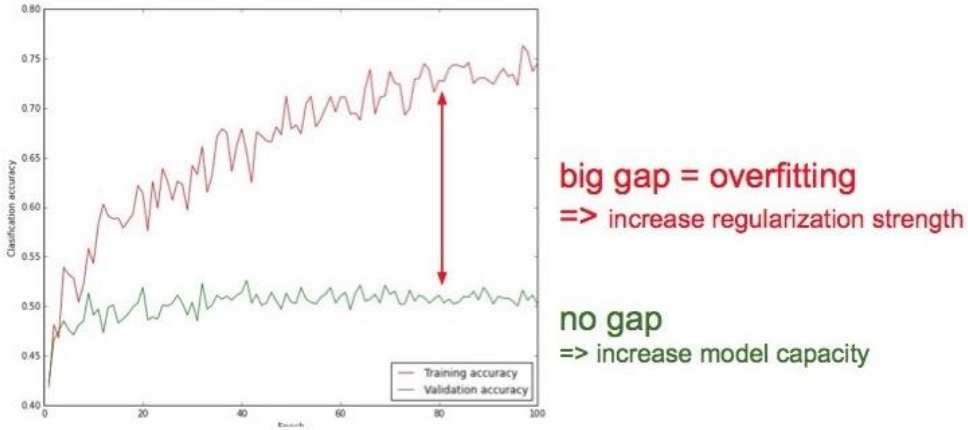
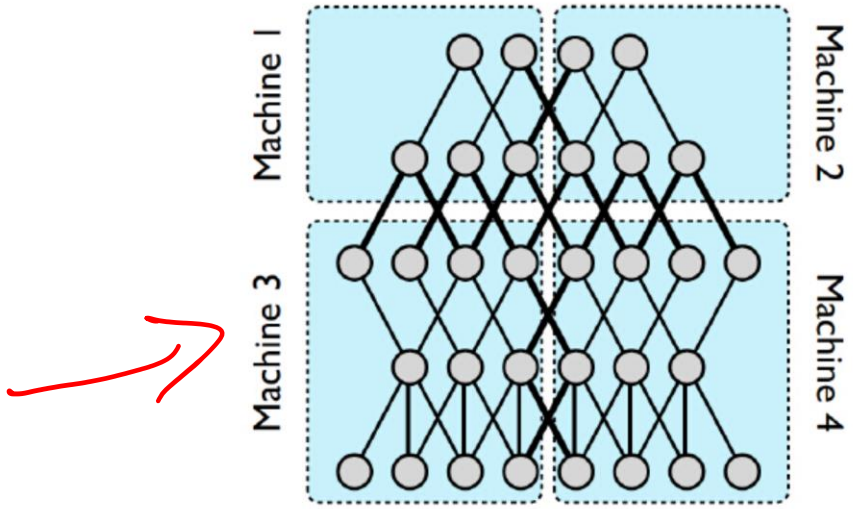


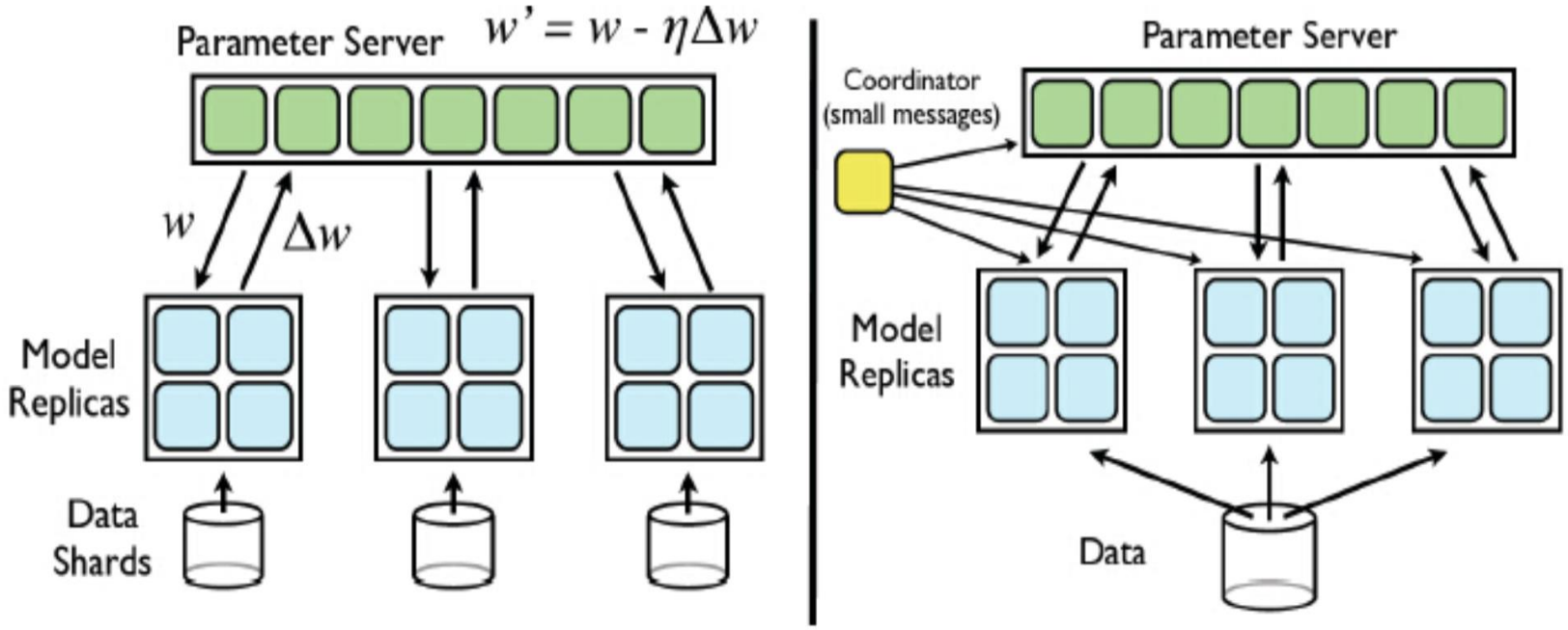
Figure 3

Model Parallelism



- Deep net is stored and processed on multiple cores (multi-thread) or machines (message passing)
- Performance benefit depends on connectivity structure vs. computational demand

Data Parallelism



Asynchronous SGD

Distributed L-BFGS

Next Time

- Deep Learning