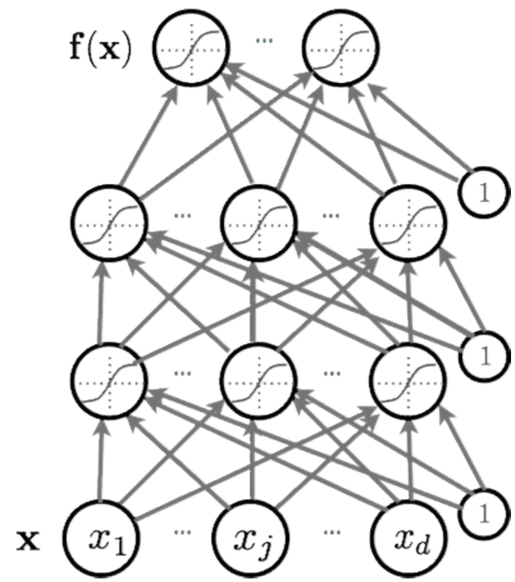
How to Train Neural Networks

Vahid Tarokh

ECE 685D, Fall 2025

Feedforward Neural Networks

- How neural networks predict f(x) given an input x:
  - Forward propagation
  - Types of units
  - Capacity of neural networks
- How to train neural nets:
  - Loss function
  - Back-propagation with gradient descent
- More recent techniques:
  - Dropout
  - Batch normalization
  - Unsupervised Pre-training



Training

- Empirical Risk Minimization:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_t \underbrace{l(f(\mathbf{x}^{(t)};\boldsymbol{\theta}), y^{(t)})}_{\text{Loss function}} + \underbrace{\lambda\Omega(\boldsymbol{\theta})}_{\text{Regularizer}}$$

- Learning is cast as optimization.

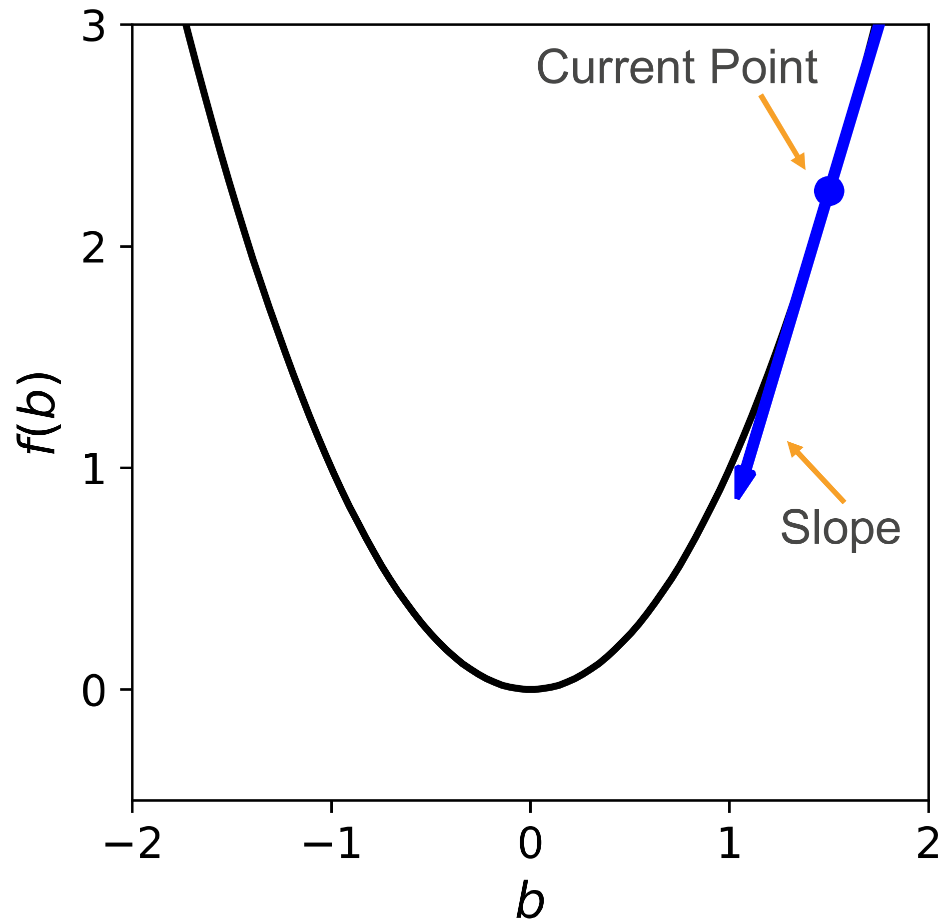
  - For classification problems, we would like to minimize classification error.

  - Loss function can sometimes be viewed as a surrogate for what we want to optimize (e.g. upper bound)

GRADIENT DESCENT

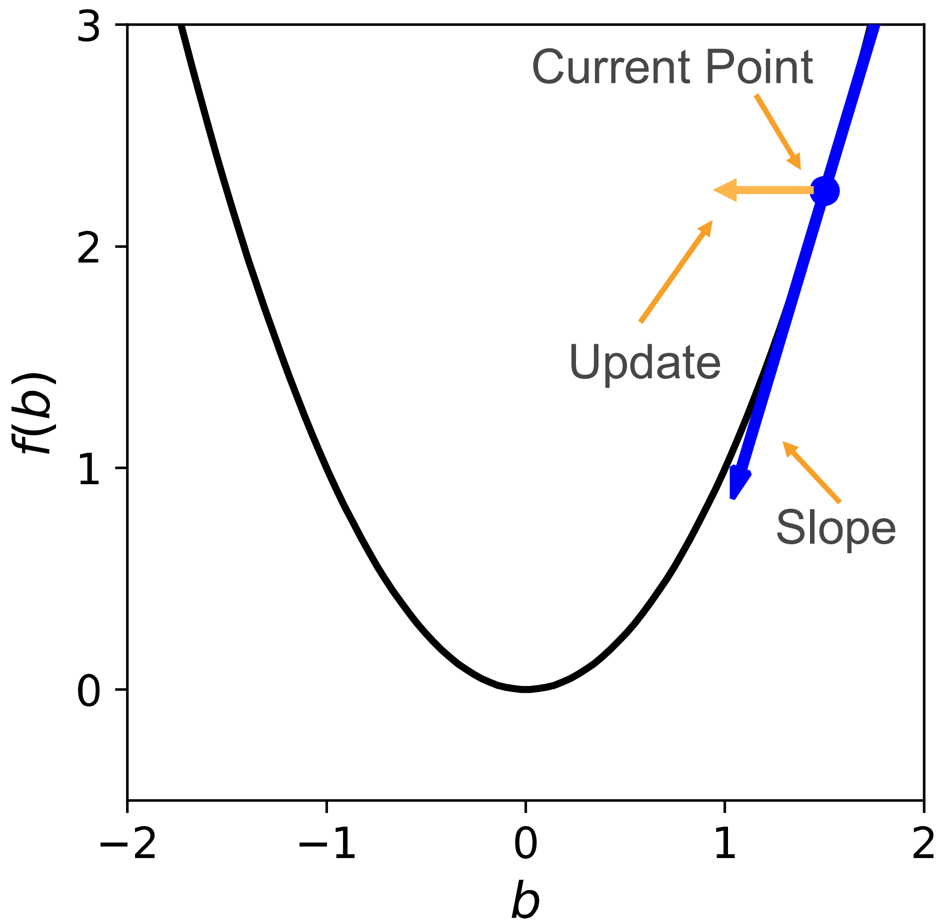
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
  1. Find the direction pointing "down the hill" (towards a smaller value)
  2. Move a bit in that direction
  3. Repeat 1-2 until satisfied



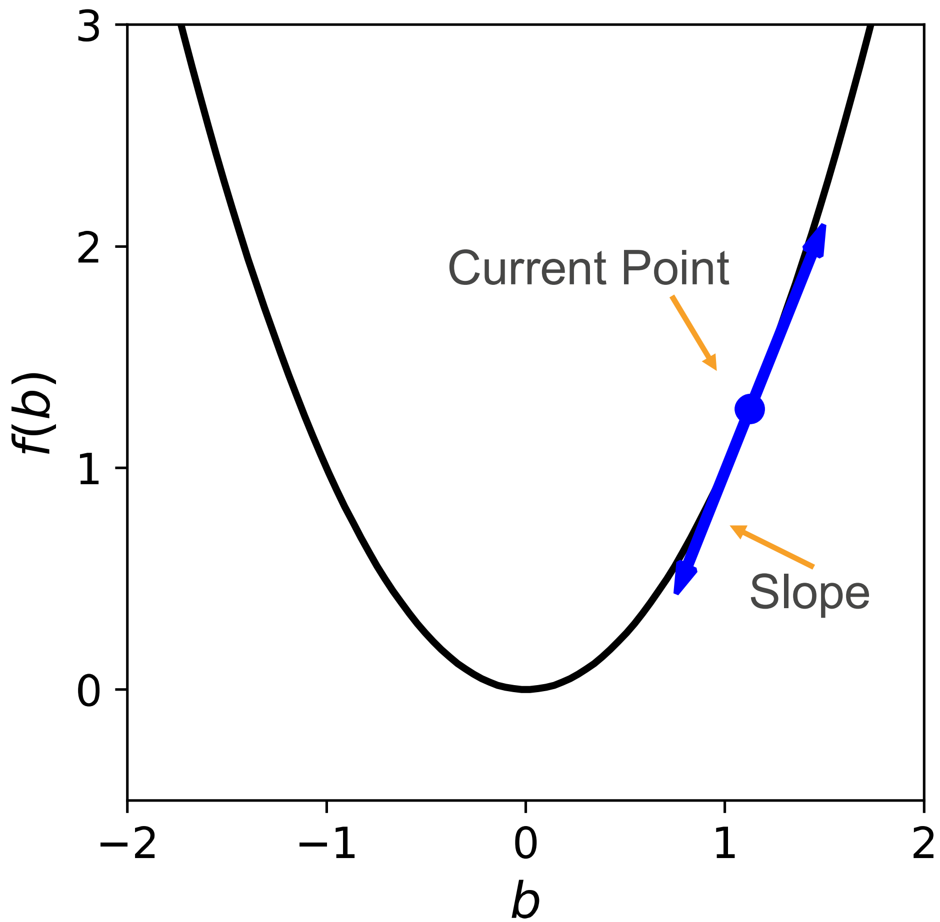
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
    1. Find the direction pointing "down the hill" (towards a smaller value)
    2. Move a bit in that direction
    3. Repeat 1-2 until satisfied



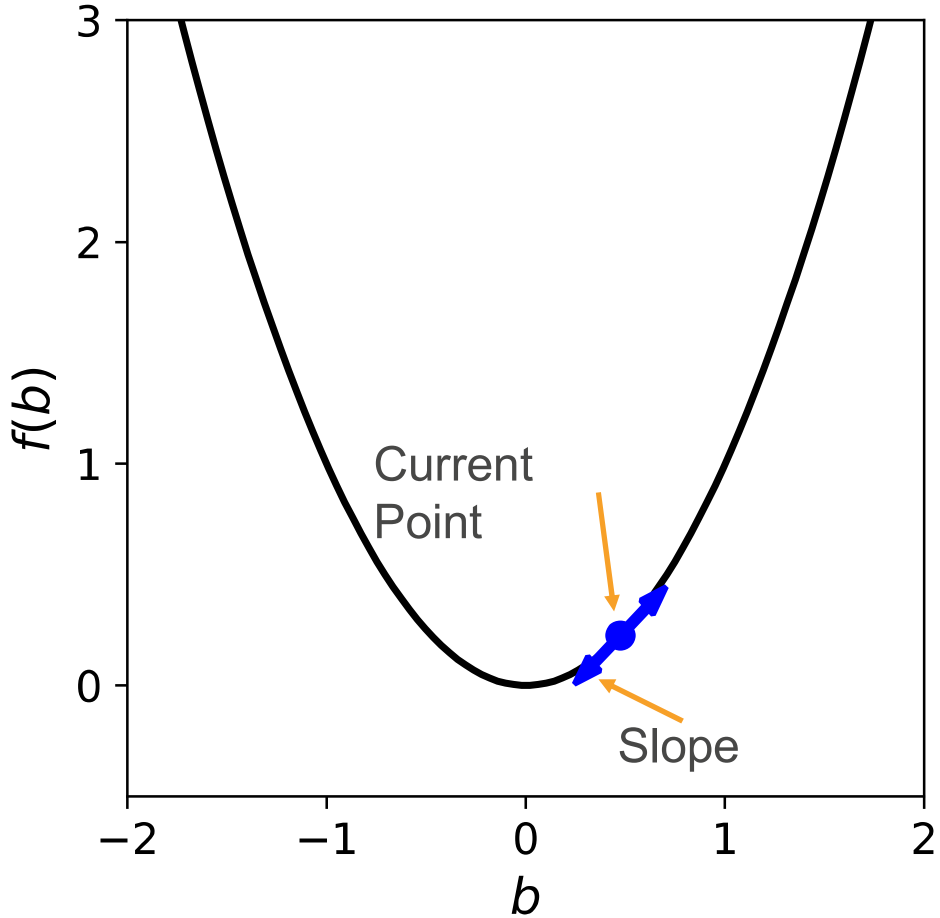
Visualization of Optimization Method

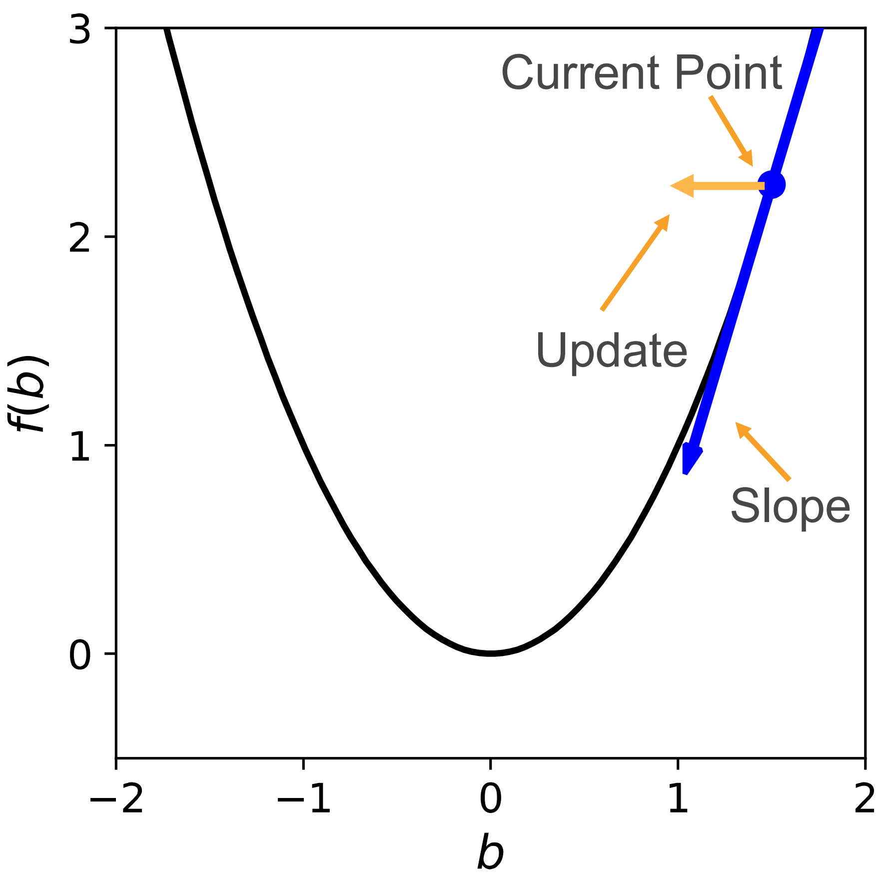
- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
    1. Find the direction pointing "down the hill" (towards a smaller value)
    2. Move a bit in that direction
    3. Repeat 1-2 until satisfied
- This shows the **first** update



Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
  1. Find the direction pointing "down the hill" (towards a smaller value)
  2. Move a bit in that direction
  3. Repeat 1-2 until satisfied
- This shows the **fourth** update





Mathematical Description of Gradient Descent

- We want to minimize a function
$$b^* = \arg\min_b f(b)$$
- Start at an initial value b^0
- We will run a series of updates to move from b^k to b^{k+1} (i.e. from b^0 to b^1)
- Iteratively run the procedure:
    1. Calculate the slope at the current point (For one parameter, this is the derivative. For multiple parameters, this is the *gradient*.): $\nabla f\left(b^k\right)$,
    ∇ means gradient or multidimensional slope
    2. Move in the direction of the negative gradient with *step size* α^k:
    $$\mathrm{b}^{\mathrm{k}+1} = \mathrm{b}^{\mathrm{k}} - \alpha^k \nabla \mathrm{f}(\mathrm{b}^{\mathrm{k}})$$
    3. Repeat 1-2 until converged

STOCHASTICS GRADIENT DESCENT

Stochastic Gradient Descent

- Perform updates after seeing each example:

  - Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

  - For t=1:T

    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\, \Delta$$

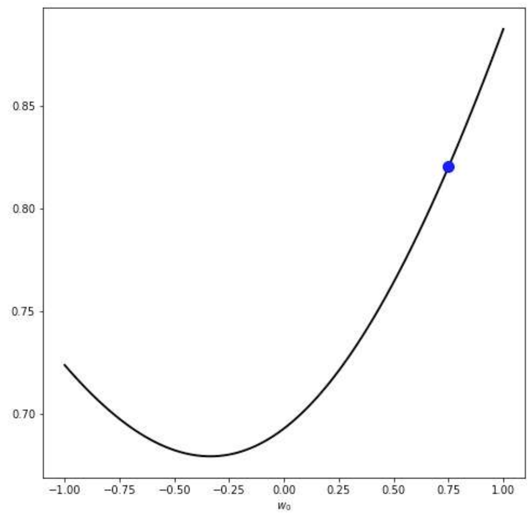
Training epoch
=
Iteration of all examples

- To train a neural net, we need:

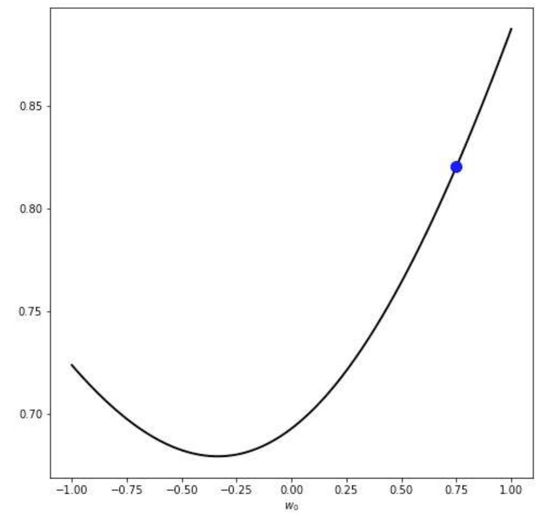
  - Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - Regularizer and its gradient: $\Omega(\boldsymbol{\theta})$, $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

Videos for Visualization

Gradient Descent



Stochastic Gradient Descent



Comments on SGD

Stochastic Gradient Descent can update *many more times* than Gradient Descent

Gets ***near*** the solution very quickly

Allows scaling to *big data* (update time doesn't increase with the data size)

In practice, we often use a minibatch, which uses a few data examples to estimate the gradient

Loss Function

- Let us start by considering a classification problem with a softmax output layer.

- We need to estimate: $f(\mathbf{x})_c = p(y = c|\mathbf{x})$

  - We can maximize the log-probability of the correct class given an input: $\log p(y^{(t)} = c|x^{(t)})$

- Alternatively, we can minimize the negative log-likelihood:

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y$$

- This is also known as a cross-entropy entropy function for multi-class classification problem (will be discussed more later on).

Stochastic Gradient Descent

- Perform updates after seeing each example:
  - Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
  - For t=1:T
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:
  - ➢ Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - ➢ A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - ➢ Regularizer and its gradient: $\Omega(\boldsymbol{\theta})$, $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

Multilayer Neural Net: Reminder

- Consider a network with L hidden layers.

  - layer pre-activation for k>0

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

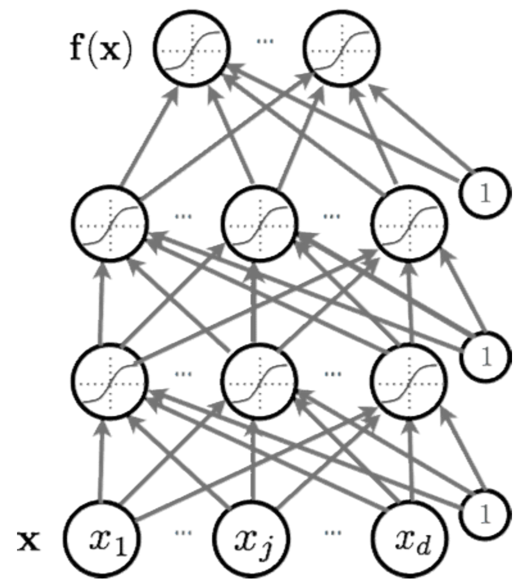
  - hidden layer activation from 1 to L:

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

  - output layer activation (k=L+1):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

Softmax activation function



$$(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$$

Gradient Computation

- Loss gradient at output

  - Partial derivative:

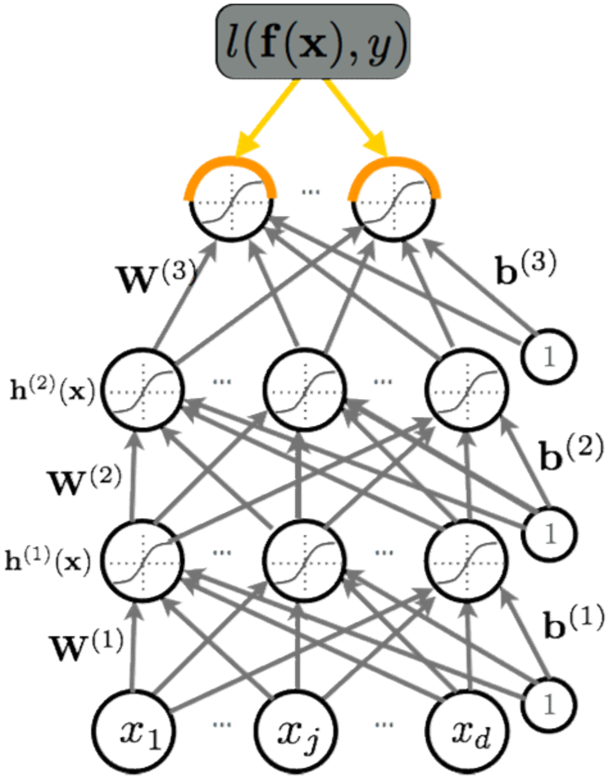
$$\frac{\partial}{\partial f(\mathbf{x})_c} - \log f(\mathbf{x})_y \quad = \quad \frac{-1_{(y=c)}}{f(\mathbf{x})_y}$$

  - Gradient:

$$\begin{aligned} & \nabla_{\mathbf{f}(\mathbf{x})} - \log f(\mathbf{x})_y \\ = \quad & \frac{-1}{f(\mathbf{x})_y} \begin{bmatrix} 1_{(y=0)} \\ \vdots \\ 1_{(y=C-1)} \end{bmatrix} \\ = \quad & \frac{-\mathbf{e}(y)}{f(\mathbf{x})_y} \end{aligned}$$

Indicator function

Remember: $f(\mathbf{x})_c = p(y = c|\mathbf{x})$



Gradient Computation

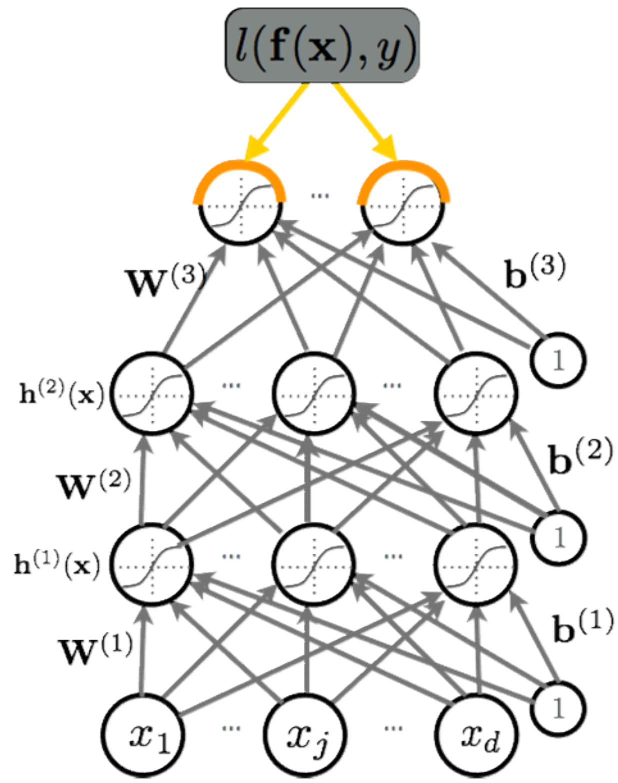
- Loss gradient at output pre-activation
  - Partial derivative:

$$\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y$$
$$= \quad -\left(1_{(y=c)} - f(\mathbf{x})_c\right)$$

  - Gradient:

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y$$
$$= \quad -\left(\mathbf{e}(y) - \mathbf{f}(\mathbf{x})\right)$$

Indicator function



Derivation

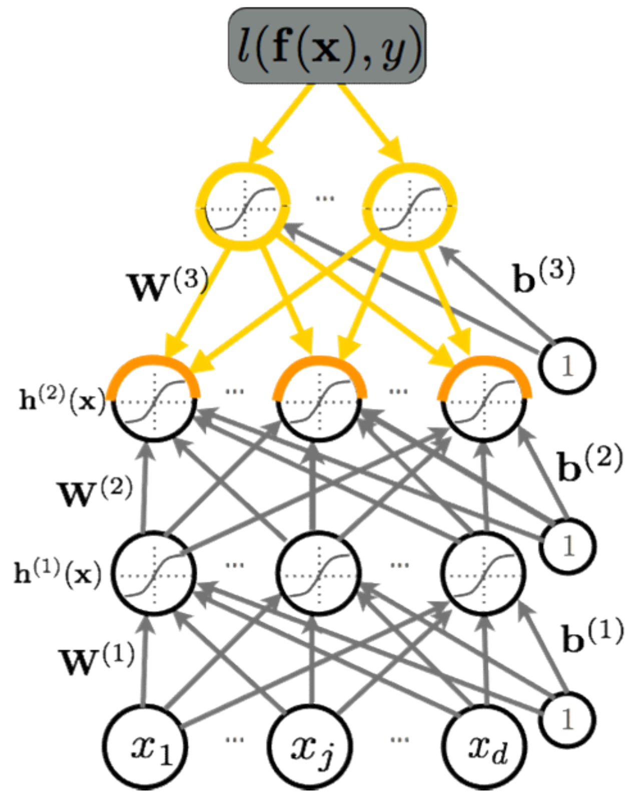
$$
\begin{aligned}
& \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y \\
= \; & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} f(\mathbf{x})_y \\
= \; & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \mathrm{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \\
= \; & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \\
= \; & \frac{-1}{f(\mathbf{x})_y} \left(\frac{\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y) \left(\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)}{\left(\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)^2} \right) \\
= \; & \frac{-1}{f(\mathbf{x})_y} \left(\frac{1_{(y=c)} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \frac{\exp(a^{(L+1)}(\mathbf{x})_c)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \right) \\
= \; & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} \mathrm{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y - \mathrm{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \, \mathrm{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_c \right) \\
= \; & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} f(\mathbf{x})_y - f(\mathbf{x})_y \, f(\mathbf{x})_c \right) \\
= \; & -\left(1_{(y=c)} - f(\mathbf{x})_c \right)
\end{aligned}
$$

$$
\boxed{\frac{\partial \frac{g(x)}{h(x)}}{\partial x} = \frac{\partial g(x)}{\partial x} \frac{1}{h(x)} - \frac{g(x)}{h(x)^2} \frac{\partial h(x)}{\partial x}}
$$

Gradient Computation

- Loss gradient for hidden layers

  - This is getting complicated!

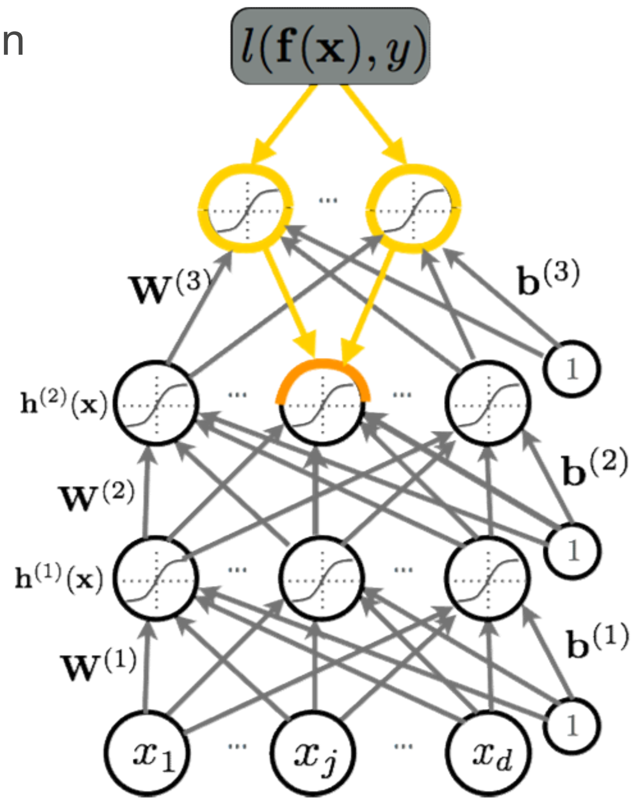


Gradient Computation

- Chain Rule: Assume that a function $p(a)$ can be written as a function of intermediate results $q_i(a)$, then:

$$\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$$

- We can invoke it by setting:
  - a be a hidden unit
  - $q_i(a)$ be a pre-activation in the layer above
  - $p(a)$ be the loss function



Gradient Computation

- Loss gradient at hidden layers

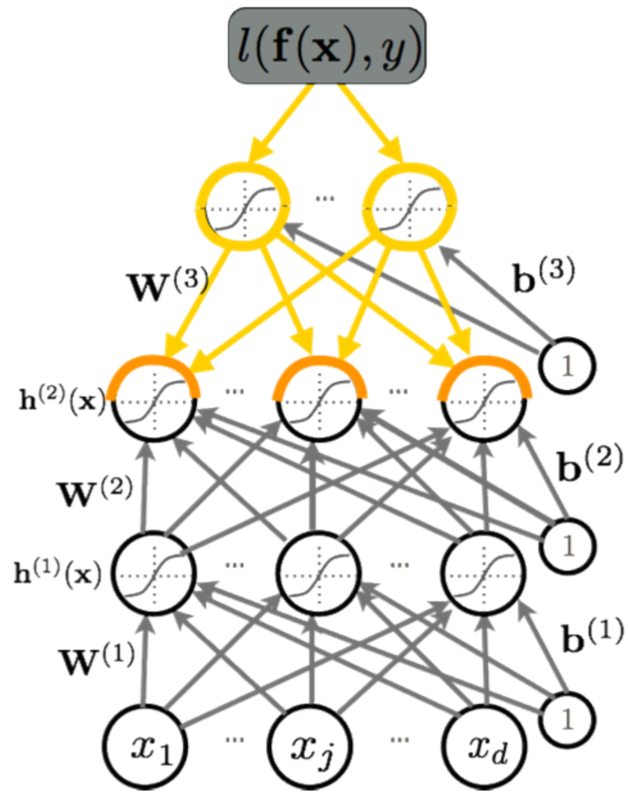
  - Partial derivative:

$$\frac{\partial}{\partial h^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$

$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} \frac{\partial a^{(k+1)}(\mathbf{x})_i}{\partial h^{(k)}(\mathbf{x})_j}$$

$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} W_{i,j}^{(k+1)}$$

Remember:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$



Gradient Computation

- Loss gradient at hidden layers
  - Gradient

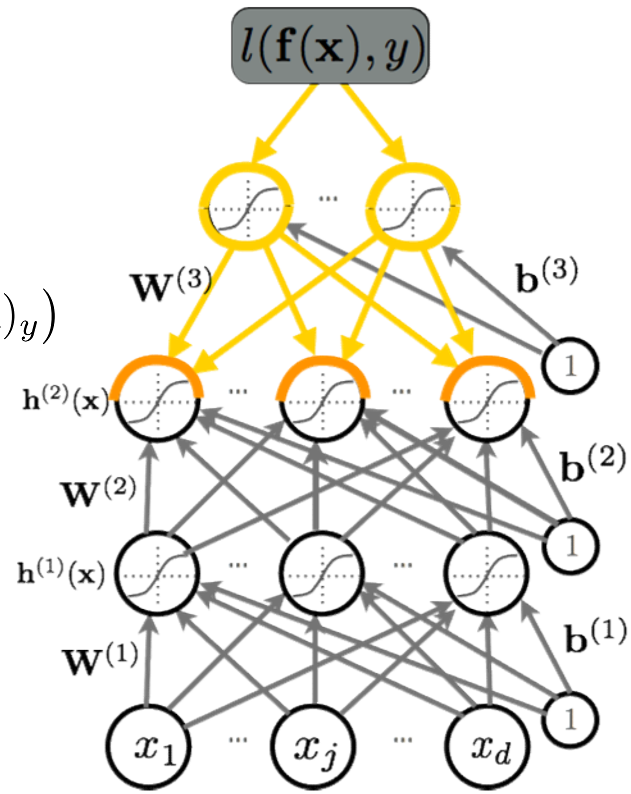
$$\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= \mathbf{W}^{(k+1)^{\top}} \underbrace{\left(\nabla_{\mathbf{a}^{(k+1)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)}_{\text{We already know how to compute that}}$$

We already
know how to
compute that

Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$



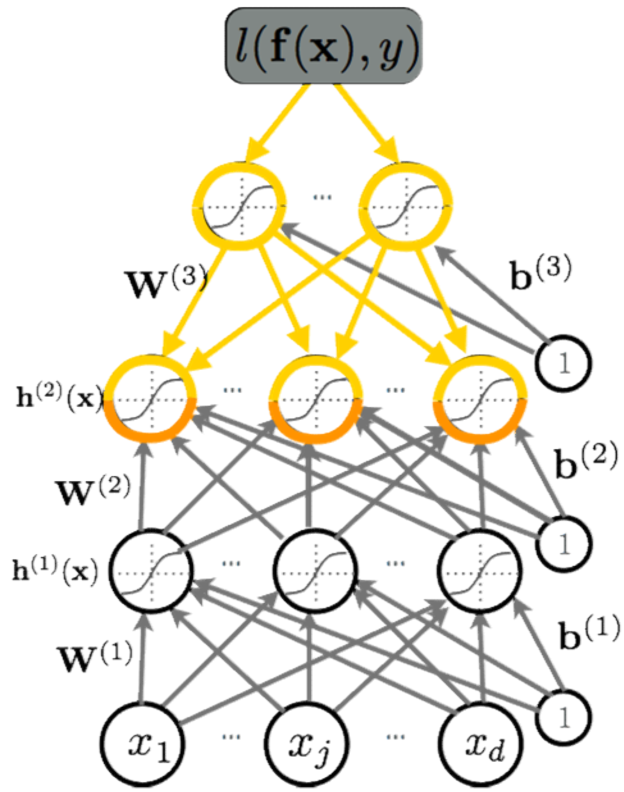
Gradient Computation

- Loss gradient at hidden layers (pre-activation)

  - Partial derivative:

$$\frac{\partial}{\partial a^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$
$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} \frac{\partial h^{(k)}(\mathbf{x})_j}{\partial a^{(k)}(\mathbf{x})_j}$$
$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} g'(a^{(k)}(\mathbf{x})_j)$$

Remember:
$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$



Gradient Computation

- Loss gradient at hidden layers (pre-activation)

  - Gradient:

$$
\begin{aligned}
& \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\
= \quad & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)^\top \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\
= \quad & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k)}(\mathbf{x})_j), \ldots]
\end{aligned}
$$

Let's look at the gradients of activation functions.

Gradient of the activation function

Remember:
$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$

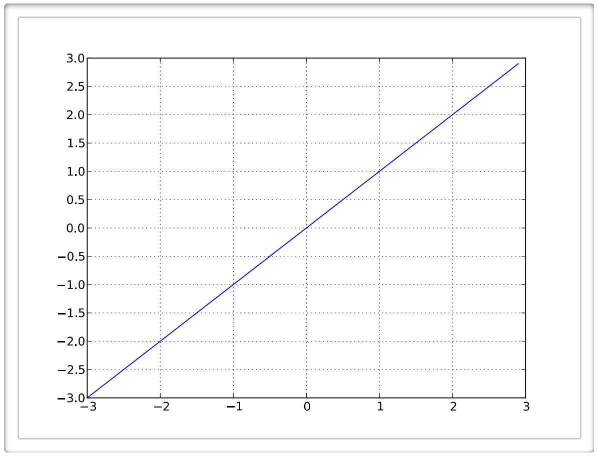
Linear Activation Function Gradient

• Linear activation function:

$$g(a) = a$$

- Partial derivative

$$g'(a) = 1$$



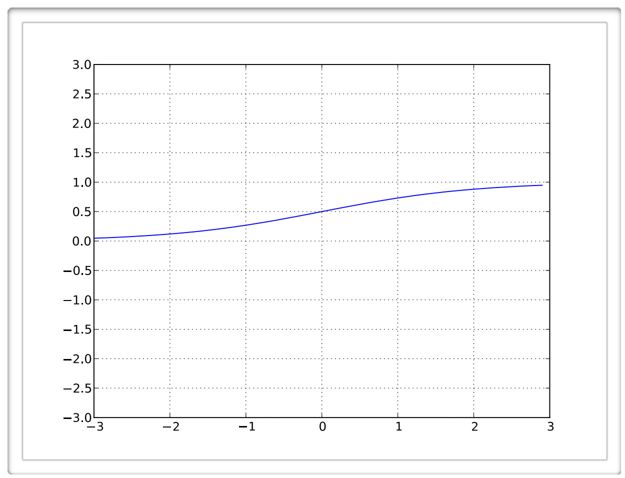
Sigmoid Activation Function Gradient

- Sigmoid activation function:

$$g(a) = \mathrm{sigm}(a) = \frac{1}{1+\exp(-a)}$$

  - Partial derivative

$$g'(a) = g(a)(1 - g(a))$$



Tanh Activation Function Gradient

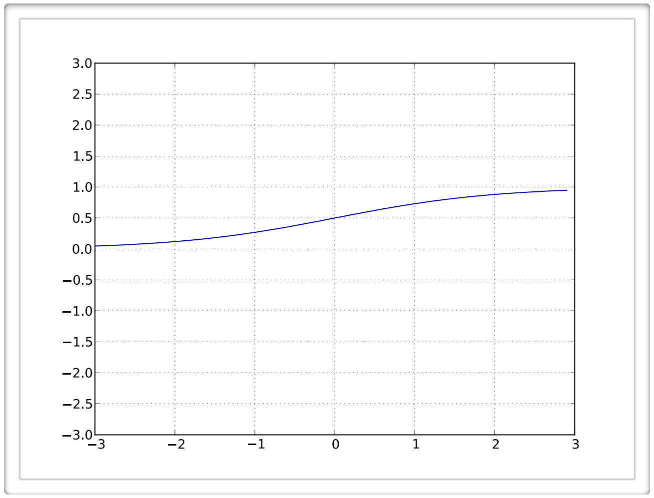
- Hyperbolic tangent ("tanh") activation function:

$$g(a) = \tanh(a) = $$

$$= \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = \frac{\exp(2a)-1}{\exp(2a)+1}$$

- Partial derivative

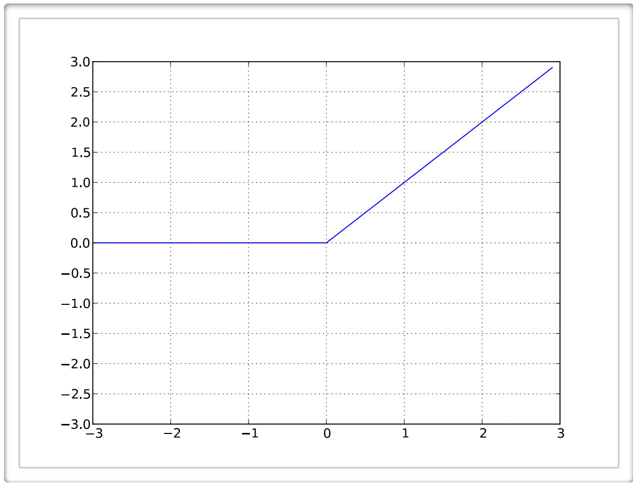
$$g'(a) = 1 - g(a)^2$$



Tanh Activation Function Gradient

- Rectified linear (ReLU) activation function:

$$g(a) = \mathrm{reclin}(a) = \max(0, a)$$



- Partial derivative

$$g'(a) = 1_{a>0}$$

Stochastic Gradient Descent

- Perform updates after seeing each example:
  - Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
  - For t=1:T
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\, \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:
  - Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - Regularizer and its gradient: $\Omega(\boldsymbol{\theta})$, $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

Gradient Computation

- Loss gradient of parameters

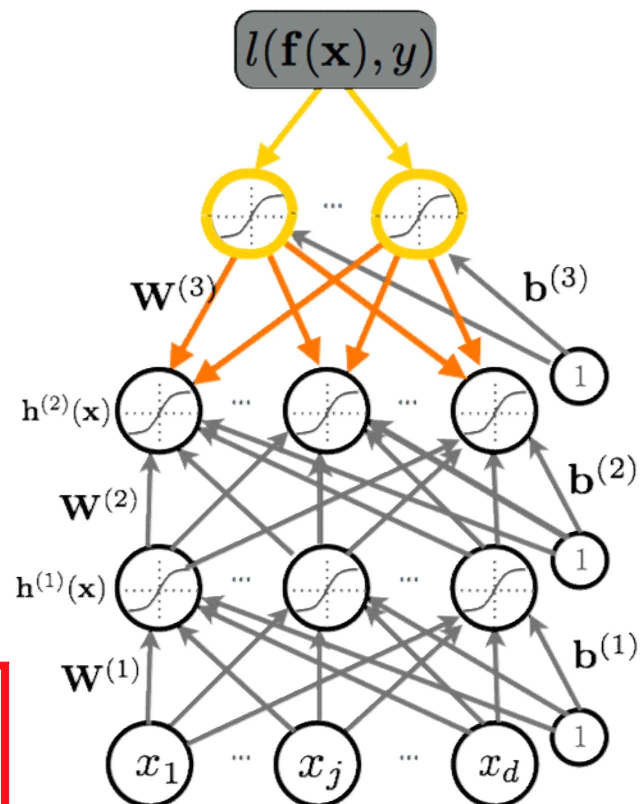
  - Partial derivative (weights):

$$\frac{\partial}{\partial W_{i,j}^{(k)}} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial W_{i,j}^{(k)}}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} h_j^{(k-1)}(\mathbf{x})$$

Remember:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

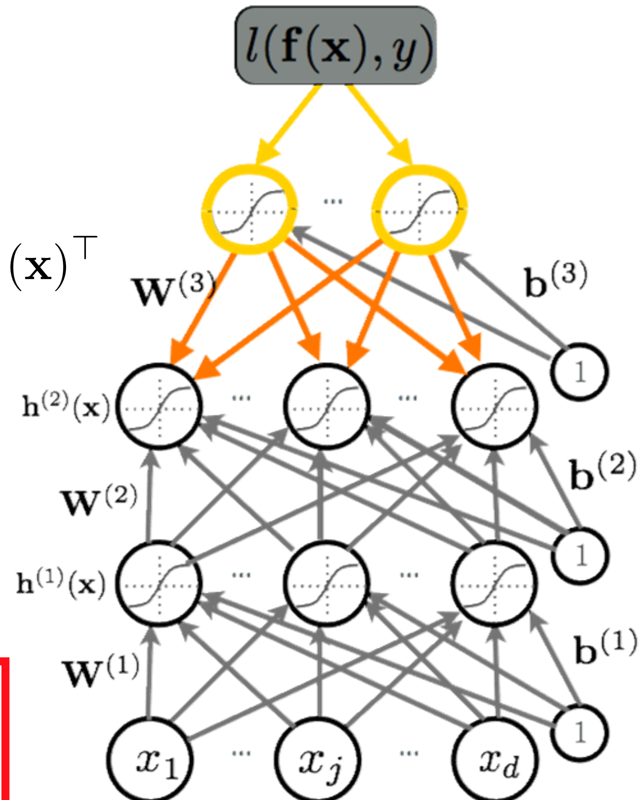


Gradient Computation

- Loss gradient of parameters

  - Gradient (weights):

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y$$
$$= \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \; \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$



Remember:
$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

Gradient Computation

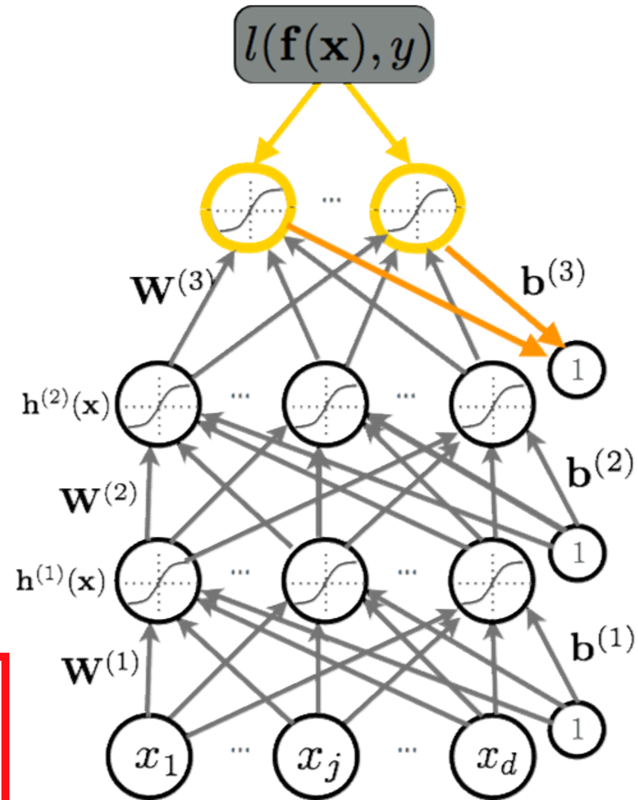
- Loss gradient of parameters
  - Partial derivative (biases):

$$\frac{\partial}{\partial b_i^{(k)}} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial b_i^{(k)}}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i}$$

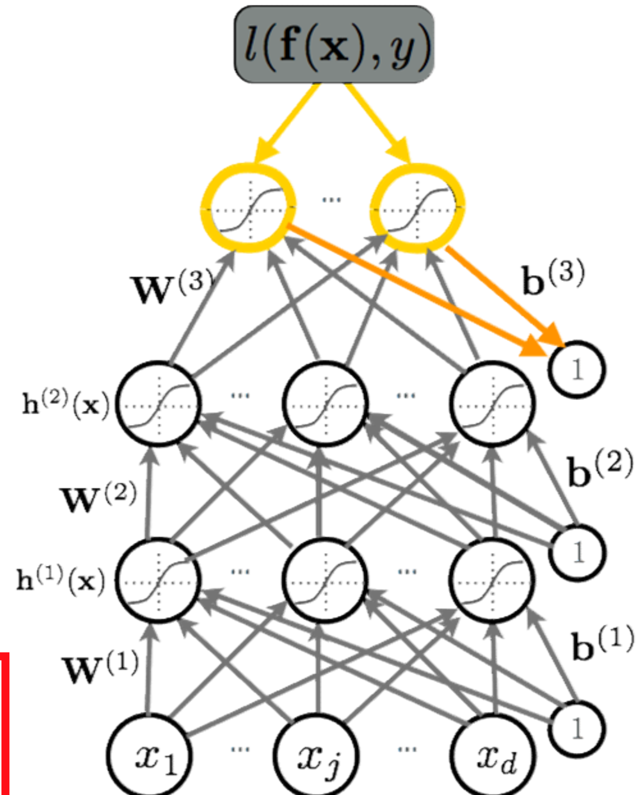
Remember:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$



Gradient Computation

- Loss gradient of parameters
  - Gradient (biases):

$$
\begin{aligned}
& \nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \\
= \quad & \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y
\end{aligned}
$$



Remember:
$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

Backpropagation Algorithm

- Perform forward propagation
- Compute output gradient (before activation):

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad -\left(\mathbf{e}(y) - \mathbf{f}(\mathbf{x})\right)$$

- For k=L+1 to 1
  - Compute gradients w.r.t. the hidden layer parameters:

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \; \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

  - Compute gradients w.r.t. the hidden layer below:

$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \mathbf{W}^{(k)^\top} \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)$$

  - Compute gradients w.r.t. the hidden layer below (before activation):

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \left(\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot \left[\ldots, g'(a^{(k-1)}(\mathbf{x})_j), \ldots\right]$$

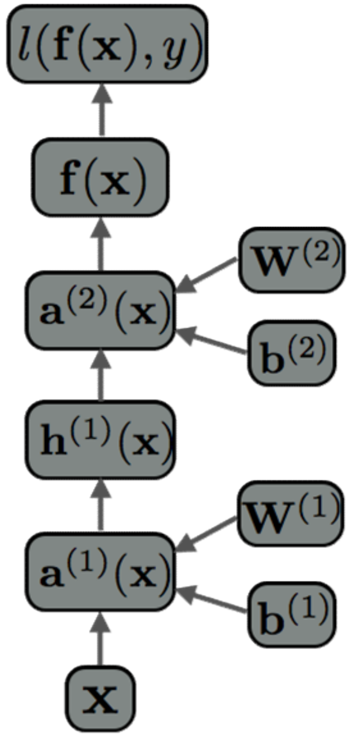
Computational Flow Graph

- Forward propagation can be represented as an acyclic flow graph

- Forward propagation can be implemented in a modular way:

  - Each box can be an object with an fprop method, that computes the value of the box given its children

  - Calling the fprop method of each box in the right order yields forward propagation



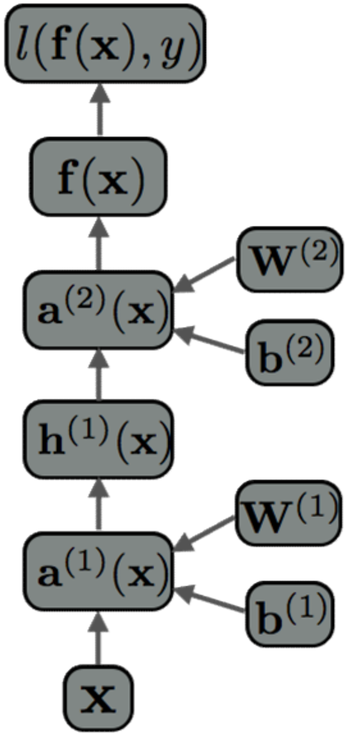
Computational Flow Graph

- Each object also has a bprop method

  - it computes the gradient of the loss with respect to each child box.

  - fprop depends on the fprop output of box's children, while bprop depends on the bprop of box's parents

- By calling bprop in the reverse order, we obtain backpropagation



Stochastic Gradient Descent

- Perform updates after seeing each example:

  - Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
  - For t=1:T
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\, \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:

  - Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - Regularizer and its gradient: $\Omega(\boldsymbol{\theta})$,$\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

Weight Decay

- L^2 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)}\right)^2 = \sum_k ||\mathbf{W}^{(k)}||_F^2$$

- Gradient:

$$\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = 2\mathbf{W}^{(k)}$$

- Only applies to weights, not biases (weigh decay)

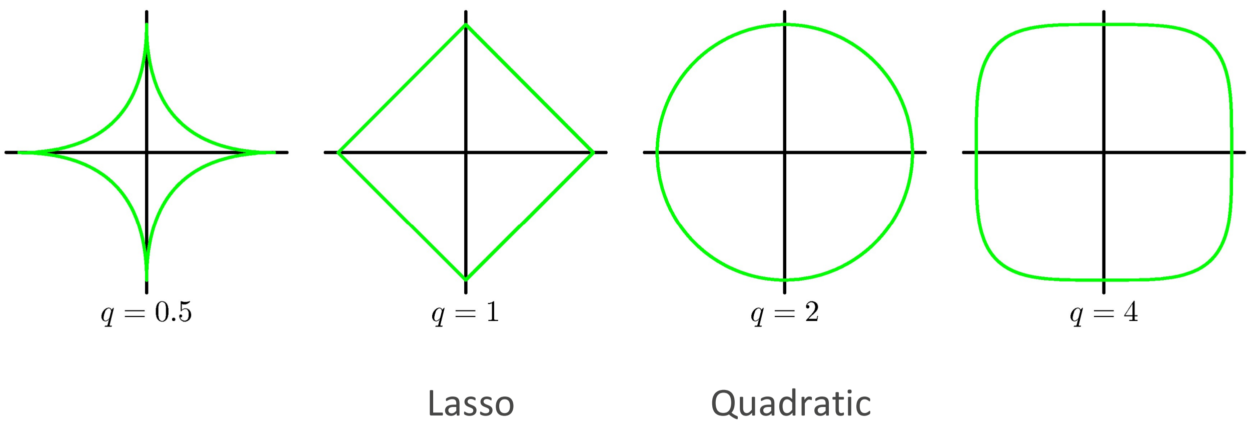
- Can be interpreted as having a Gaussian prior over the weights, while performing MAP estimation.

- We will later look at Bayesian methods.

Other Regularizers

- Using a more general regularizer, we get:

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\sum_{j=1}^{M}|w_j|^q$$



L^1 Regularization

- L^1 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

- Gradient:

$$\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \mathrm{sign}(\mathbf{W}^{(k)})$$

$$\mathrm{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$$

- Only applies to weights, not biases (weigh decay)

- Can be interpreted as having a Laplace prior over the weights, while performing MAP estimation.

- Unlike L2, L1 will push some weights to be exactly 0.

Initialization

- Initialize biases to 0
- For weights
  - Can not initialize weights to 0 with tanh activation
    - All gradients would be zero (saddle point)
  - Can not initialize all weights to the same value
    - All hidden units in a layer will always behave the same
    - Need to break symmetry
  - Sample $\mathbf{W}_{i,j}^{(k)}$ from $U\left[-b, b\right]$, where

$$b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$$

Sample around 0 and break symmetry

Size of $\mathbf{h}^{(k)}(\mathbf{x})$

Model Selection

- Training Protocol:

  - Train your model on the Training Set $\mathcal{D}^{\mathrm{train}}$

  - For model selection, use Validation Set $\mathcal{D}^{\mathrm{valid}}$

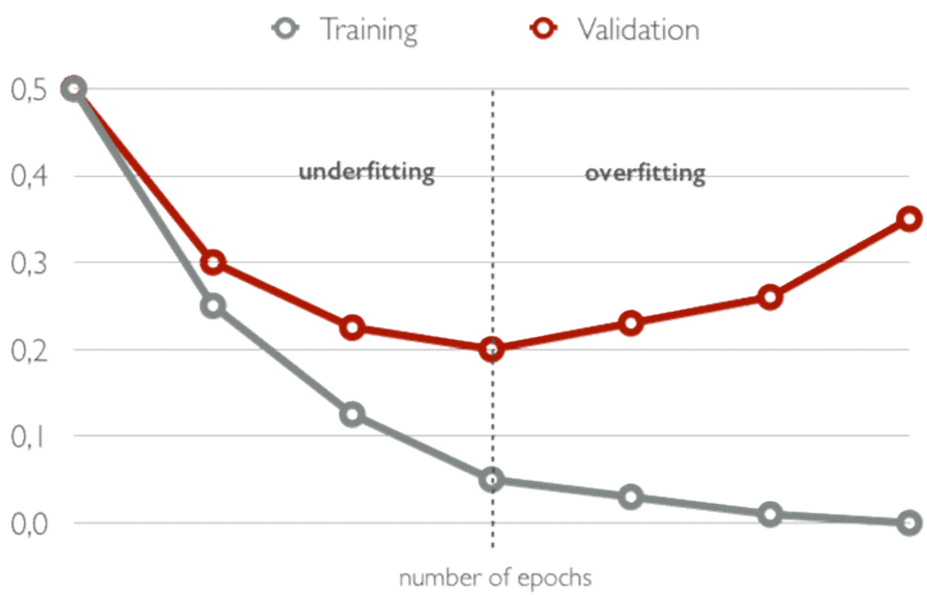
    - Hyper-parameter search: hidden layer size, learning rate, number of iterations/epochs, etc.

  - Estimate generalization performance using the Test Set $\mathcal{D}^{\mathrm{test}}$

- Remember: Generalization is the behavior of the model on **unseen examples**.

Early Stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead).



Tricks of the Trade:

- Normalizing your (real-valued) data:
  - for each dimension x_i subtract its training set mean
  - divide each dimension x_i by its training set standard deviation
  - this can speed up training

- Decreasing the learning rate: As we get closer to the optimum, take smaller update steps:

  i. start with large learning rate (e.g. 0.1)
  ii. maintain until validation error stops improving
  iii. divide learning rate by 2 and go back to (ii)

Gradient Checking

- To debug your implementation of fprop/bprop, you can compare with a finite-difference approximation of the gradient:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$$

- $f(x)$ would be the loss
- x would be a parameter
- $f(x+\epsilon)$ would be the loss if you add ϵ to the parameter
- $f(x-\epsilon)$ would be the loss if you subtract ϵ to the parameter

Debugging on Small Dataset

- If not, investigate the following situations:
  - ➢ Are some of the units saturated, even before the first update?
    - scale down the initialization of your parameters for these units
    - properly normalize the inputs
  - ➢ Is the training error bouncing up and down?
    - decrease the learning rate

- This does not mean that you have computed gradients correctly:
  - ➢ You could still overfit with some of the gradients being wrong