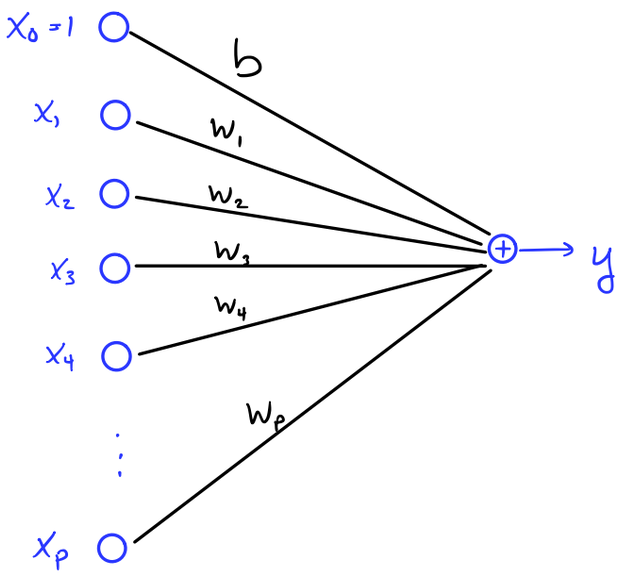
Neural Networks Part I

STAT 37710 / CMSC 35400
Rebecca Willett and Yuxin Chen

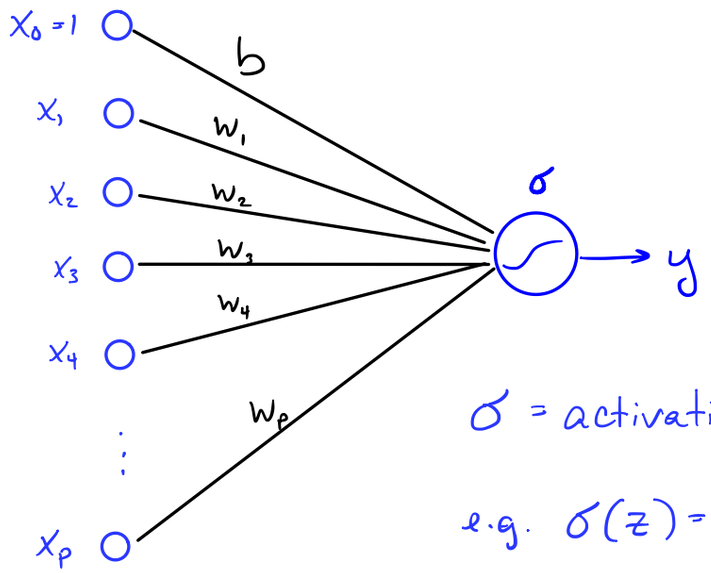
Introduction

A Simple Neural Network

$$y = \underline{x}^T \underline{w} + b$$



$$y = \sigma(\underline{x}^T \underline{w} + b)$$



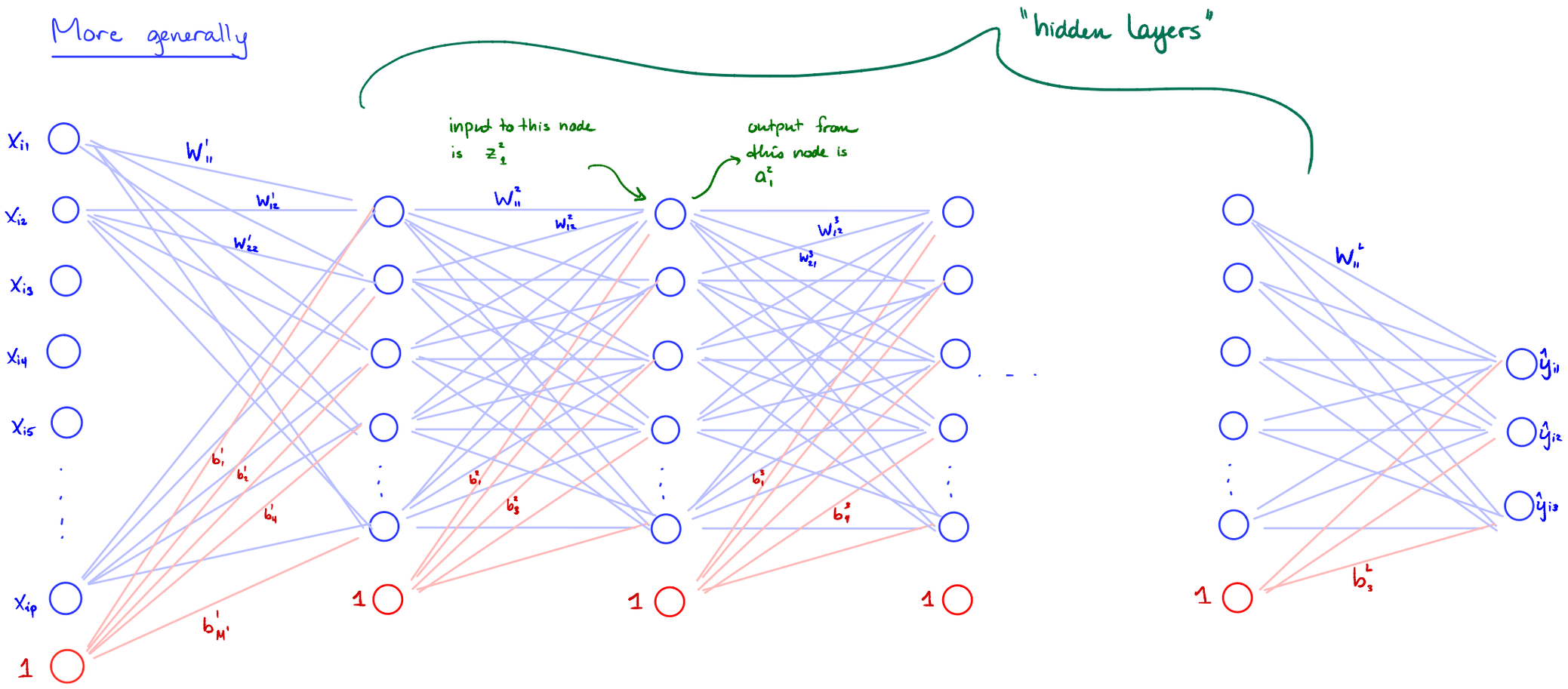
σ = activation function

e.g. $\sigma(z) = \max(0, z) = \text{ReLU}$

$$\sigma(z) = \frac{1}{1+e^{-z}} = \text{logistic}$$

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \tanh(z)$$

Multi-Layered Networks



$\underline{X}$ W^1, b^1 W^2, b^2 W^3, b^3 ... W^L, b^L

$= a^0$

$z^1 = W^1 a^0 + b^1$ $z^2 = W^2 a^1 + b^2$ $z^3 = W^3 a^2 + b^3$ $z^L = W^L a^{L-1} + b^L = \hat{y}$

$a^1 = \sigma(z^1)$ $a^2 = \sigma(z^2)$ $a^3 = \sigma(z^3)$

Let M^l = # of blue nodes in layer l

$W^l \sim M^l \times M^{l-1}$,

$\underline{b}^l \sim M^l \times 1$,

$\underline{a}^l, \underline{z}^l \sim M^l \times 1$,

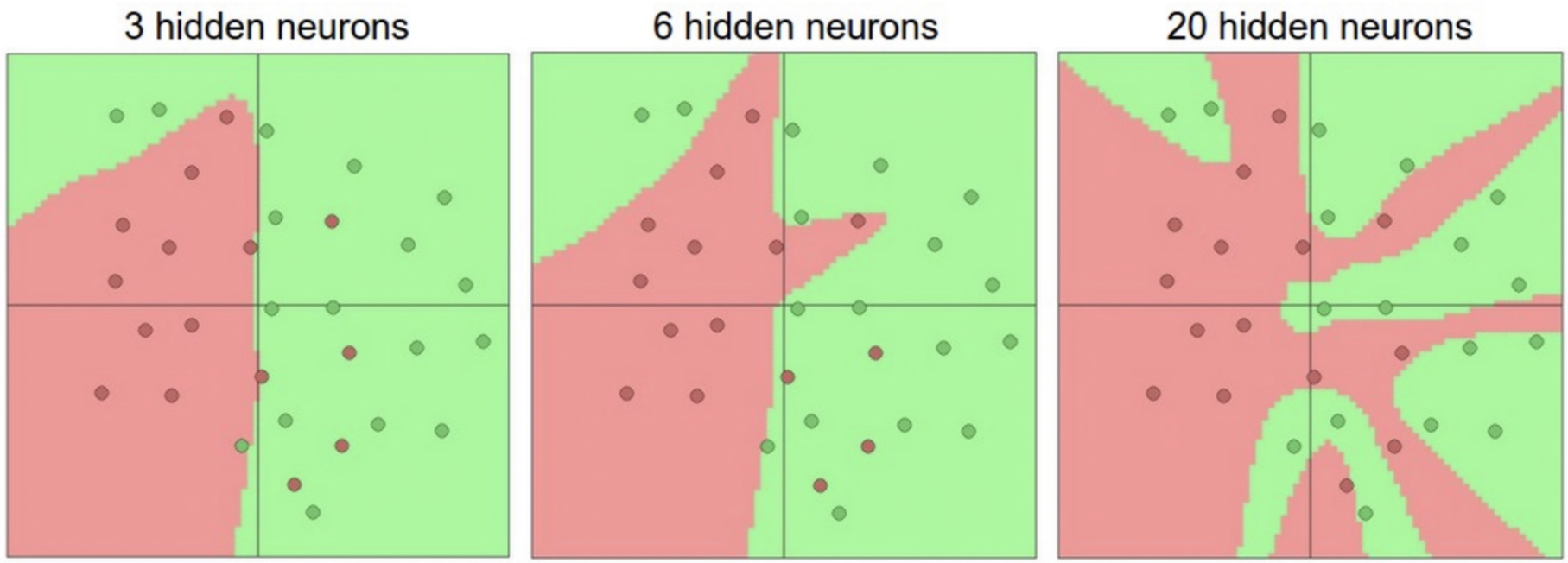
$\underline{b}^l \sim M^l \times 1$

$$\hat{y}_{i,k} = W_k^L \sigma\left(W_k^{L-1} \sigma\left(\cdots \sigma\left(W^2 \sigma\left(W^1 x_i + b^1\right) + b^2\right) + \cdots\right) + b^{L-1}\right) + b^L$$

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper



- The capacity of the network increases with more hidden units and more hidden layers

Training

Let $\theta = (W^1, \dots, W^L, b^1, \dots, b^L)$ denote all the weights in the neural network.

Our task in training is to use the training samples (x_i, y_i), $i = 1, \dots, n$, to select a $\hat{\theta}$ that minimizes the training loss.

That is, let $f_\theta(x)$ be the output of the network with weights θ and input x:

$$f_\theta(x) = W_k^L \sigma\left(W_k^{L-1}\left(\cdots \sigma\left(W^2 \sigma\left(W^1 x + b^1\right) + b^2\right) + \cdots\right) + b^{L-1}\right) + b^L$$

We want to find $\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, f_\theta(x_i))$, where

$L(\hat{y}, y)$ might be $\cdot \|y - \hat{y}\|^2$ or $\exp(-y\hat{y})$ for binary classification.

For K-class classification, map $y_i \to z_i \in \{0,1\}^K$, where $z_{ik} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{otherwise} \end{cases} = [0 \;\; 0 \cdots \underset{\uparrow}{1} \cdots 0]^T$ (k^{th} position)

Now we want to predict z_i for each x_i.

Let $\hat{y}_{ik}$ be k^{th} output of network with input x_i, and say $\mathbb{P}(\text{class} = k | x_i, \theta) = \dfrac{\exp(\hat{y}_{ik})}{\sum_{j=1}^{K} \exp(\hat{y}_{ij})}$

Then $L_i(\theta) = -\sum_{k=1}^{K} z_{ik} \log \mathbb{P}(\text{class} = k | x_i, \theta)$

"cross-entropy loss"

Stochastic Gradient Descent

Optimizing network weights

Write $L_i(\theta) := L(y_i, f_\theta(x_i))$

and $\hat{R}(\theta) := \frac{1}{n}\sum_{i=1}^{n} L_i(\theta)$.

Then $\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\ \hat{R}(\theta)$

Our basic tool for this is

stochastic gradient descent.

Recall gradient descent:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\ \hat{R}(\theta)$$

$$\theta^{(t+1)} = \theta^{(t)} - \tau \nabla \hat{R}(\theta^{(t)})$$

Recall $\hat{R}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta)$

Then Gradient Descent =

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\tau}{n}\sum_{i=1}^{n} \nabla L_i(\theta^{(t)})$$

Stochastic Gradient Descent:

@ iteration t, choose $i_t \in \{1, 2, \ldots, n\}$

$$\theta^{(t+1)} = \theta^{(t)} - \tau \nabla L_{i_t}(\theta^{(t)})$$

- each iteration easier/faster to compute
- need more iterations
- choose i_t uniformly at random

Ex: $\hat{R}(\theta) = \frac{1}{n}\sum_{i=1}^{n} (y_i - \langle \underline{x}_i, \theta \rangle)^2 + \lambda \|\theta\|_2^2$

$$L_i(\theta) = (y_i - \langle \underline{x}_i, \theta \rangle)^2 + \lambda \|\theta\|_2^2$$

check: $\frac{1}{n}\sum_{i=1}^{n} L_i(\theta) = \hat{R}(\theta)$

$$\nabla L_i(\theta) = -2(y_i - \langle \underline{x}_i, \theta \rangle)\underline{x}_i + 2\lambda\theta$$

SGD: $\theta^{(t+1)} = \theta^{(t)} + 2\tau(y_{i_t} - \langle \underline{x}_{i_t}, \theta^{(t)} \rangle)\underline{x}_{i_t} - 2\tau\lambda\theta^{(t)}$

SGD on simple network

for $\hat{y} = \sigma(\underline{x}^T \underline{w} + b)$ and $\sigma(z) = \frac{1}{1+e^{-z}}$, how do we learn weights?

loss $\hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

$= \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \sigma(\underbrace{\langle \underline{x}_i, \underline{w} \rangle + b}_{z_i}) \right)^2$

$\hat{y}_i = \sigma(\underline{x}_i^T \underline{w} + b) = \sigma(z_i)$

$z_i = \underline{x}_i^T \underline{w} + b$

aside: $\frac{d\sigma}{dz} = \sigma'(z)$

$= \sigma(z)(1 - \sigma(z))$

Learn $\underline{w}, b$ via Stochastic Gradient Descent!

@ iteration t

- choose i_t uniformly at random
- set $\underline{w}^{(t+1)} = \underline{w}^{(t)} - \tau \nabla L_{i_t}(\underline{w}^{(t)})$

What is $\nabla L_i(w^{(t)})$?

$$\left.\frac{dL_i}{dw_j}\right|_{\underline{w}^{(t)}} = \frac{dL_i}{d\hat{y}_i} \cdot \frac{d\hat{y}_i}{dz_i} \cdot \left.\frac{dz_i}{dw_j}\right|_{\underline{w}^{(t)}}$$

$$= 2(\hat{y}_i - y_i) \cdot \sigma'(z_i) \cdot x_{ij} \Big|_{\underline{w}^{(t)}}$$

$$= 2(\hat{y}_i - y_i) \cdot \sigma(z_i)(1 - \sigma(z_i)) \, x_{ij} \Big|_{\underline{w}^{(t)}}$$

$= \underbrace{2(\hat{y}_i - y_i)\hat{y}_i(1 - \hat{y}_i)}_{\text{scalar, independent of } j} x_{ij} \Rightarrow \nabla L_i(\underline{w}^{(t)}) = \delta_i \underline{x}_i$

call $\delta_i = \delta_i(w^{(t)})$

$b^{(t+1)} = b^{(t)} - \tau \nabla L_{i_t}(b^{(t)})$

What is $\nabla_b L_{i_t}(b^{(t)}) = \left.\frac{dL_{i_t}}{db}\right|_{b^{(t)}}$?

$$\left.\frac{dL_i}{db}\right|_{b^{(t)}} = \frac{dL_i}{d\hat{y}_i} \cdot \frac{d\hat{y}_i}{dz_i} \cdot \left.\frac{dz_i}{db}\right|_{b^{(t)}}$$

$= \delta_i \cdot 1$

SGD: $\underline{w}^{(t+1)} = w^{(t)} - \tau \delta_{i_t} \underline{x}_{i_t}$, $b^{(t+1)} = b^{(t)} - \tau \delta_{i_t}$

Backpropagation

To train deeper networks (i.e. networks with one or more hidden layers), we use

BACKPROPAGATION: an efficient way of computing gradients needed for SGD in a multilayer network

Basic backpropagation structure:

Loop until convergence
  for each sample $i \in \{1, \ldots, n\}$

1. given feature x_i, "propagate activity forward" (i.e. calculate network output with input x_i and current estimate of network weights) $\Rightarrow$ "FORWARD PASS"
2. Propagate gradients backward $\Rightarrow$ "BACKWARD PASS"
3. Update each weight (via stochastic gradient descent)

Hidden activities

We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity

- Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
- Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
- We can compute error derivatives for all the hidden units efficiently
- Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit

This is just the chain rule!

https://www.cs.toronto.edu/~urtasun/courses/CSC411_Fall16/10_nn1.pdf

General weight updates

Weight updates for more complex network

z^ℓ is input to nodes in layer ℓ, a^ℓ is output of those nodes

$W^\ell_{j,k}$ = weight in layer ℓ, applied to layer $\ell-1$ output $(a^{\ell-1}_k)$, feeding into next layer (z^ℓ_j)

To update W^ℓ, need to compute $\nabla L_i(W^\ell)$

$$\frac{dL_i}{dW^\ell_{j,k}} = \frac{dL_i}{dz^\ell_j} \cdot \frac{dz^\ell_j}{dW^\ell_{j,k}} = \delta^\ell_j \cdot a^{\ell-1}_k \Longrightarrow \nabla L_i(W^\ell) = \underline{\delta}^\ell \left(\underline{a}^{\ell-1}\right)^T$$

$$\delta^\ell_j := \frac{dL_i}{dz^\ell_j} = \begin{cases} \frac{dL_i}{da^\ell_j} \cdot \frac{da^\ell_j}{dz^\ell_j} = \left[\left(W^{\ell+1}\right)^T \delta^{\ell+1}\right]_j \cdot \sigma'(z^\ell_j) & \text{for } \ell < L \\ \frac{dL_i}{d\hat{y}_{ij}} \frac{d\hat{y}_{ij}}{dz^L_j} & \text{for } \ell = L \end{cases}$$

$$\underline{\delta}^\ell = \begin{cases} \left(\left(W^{\ell+1}\right)^T \underline{\delta}^{\ell+1}\right) \odot \sigma'(\underline{z}^\ell) & \text{for } \ell < L \\ \frac{dL_i}{d\hat{y}_{ij}} & \text{for } \ell = L \end{cases}$$

$\odot$: element-wise multiplication

$$\frac{dL_i}{da^\ell_j} = \sum_{k=1}^{M^{\ell+1}} \frac{dL_i}{dz^{\ell+1}_k} \frac{dz^{\ell+1}_k}{da^\ell_j} = \sum_{k=1}^{M^{\ell+1}} \delta^{\ell+1}_k W^{\ell+1}_{kj} = \left[\left(W^{\ell+1}\right)^T \delta^{\ell+1}\right]_j$$

$$\frac{da^\ell_j}{dz^\ell_j} = \sigma'(z^\ell_j)$$

$$\frac{dz^\ell_j}{dW^\ell_{j,k}} = a^{\ell-1}_k \quad \text{because} \quad z^\ell_j = \left(W^\ell a^{\ell-1} + b^\ell\right)_j = \sum_k W^\ell_{jk} a^{\ell-1}_k + b^\ell_j$$

$$\frac{dL_i}{db^\ell_j} = \frac{dL_i}{dz^\ell_j} \cdot \frac{dz^\ell_j}{db^\ell_j} = \delta^\ell_j \cdot 1 \Rightarrow \nabla_{b^\ell} L_i = \delta^\ell$$

$$\Longrightarrow \quad \nabla L_i(W^\ell) = \underline{\delta}^\ell \left(\underline{a}^{\ell-1}\right)^T$$

$$\nabla L_i(b^\ell) = \underline{\delta}^\ell$$

Backpropagation Algorithm

Backpropagation Algorithm

for $t = 1, 2, 3, \ldots$

select $i_t \sim \text{unif}(1, 2, \ldots, n)$

\# forward pass:

$a^0 = x_{i_t}$

for $l = 1, 2, \ldots, L$

$z^l = W^{l(t)} a^{l-1} + b^{l(t)}$

$a^l = \sigma(z^l)$

end

\# backprop

$s^L = \nabla L_{i_t}(\hat{y}_{i_t})$

for $l = L-1, L-2, \ldots, 1$

$s^l = \left[(W^{l+1\,t})^T s^{l+1} \right] \odot \sigma'(z^l)$

$\nabla L_{i_t}(W^{lt}) = s^l (a^{l-1})^T$

$\nabla L_{i_t}(b^{lt}) = s^l$

$W^{l\,t+1} = W^{lt} - \tau \nabla L_{i_t}(W^{lt})$

$b^{l\,t+1} = b^{lt} - \tau \nabla L_{i_t}(b^{lt})$

end

Initialization and Standardization

Initializing weights:

Usually initial weights are chosen to be random values near zero.

In this regime, is σ is a sigmoid (e.g. $\sigma(z) = \frac{1}{1+e^{-z}}$), then for each node, the corresponding z is approximately 0, which in turn suggests $\sigma(z) \approx \frac{1}{2} + \frac{z}{4}$

In other words, when all weights ≈ 0, each neural network node behaves approximately LINEARLY.

Thus the model starts out nearly linear, and as training moves some weights away from 0, the model becomes more nonlinear where needed.

(Starting with weights exactly equal to zero leads to "vanishing gradients" (i.e. derivatives = 0), so training cannot move the weights and we never learn a good model.)

Scaling inputs (training data)

Rule of thumb: standardize all inputs to have zero mean and standard deviation = 1

$\Rightarrow$ makes it easier to choose a good range of initial weights and easier to regularize (discussed later)

With standard inputs, can take initial weights by drawing from $\text{Unif}([-0.7, +0.7])$

Minibatch and Step Size

How often to update weights:

- after computing gradients for single training sample (stochastic gradient descent)

$$\theta^{(t+1)} = \theta^{(t)} - \tau \nabla L_{i_t}(\theta^{(t)})$$

- after a full sweep through the training data:

$$\theta^{(t+1)} = \theta^{(t)} - \tau \nabla \hat{R}(\theta^{(t)}) = \theta^{(t)} - \frac{\tau}{n} \sum_{i=1}^{n} \nabla L_i(\theta^{(t)})$$

- after a mini-batch of training cases (e.g. a random sample of $m<n$ training samples)

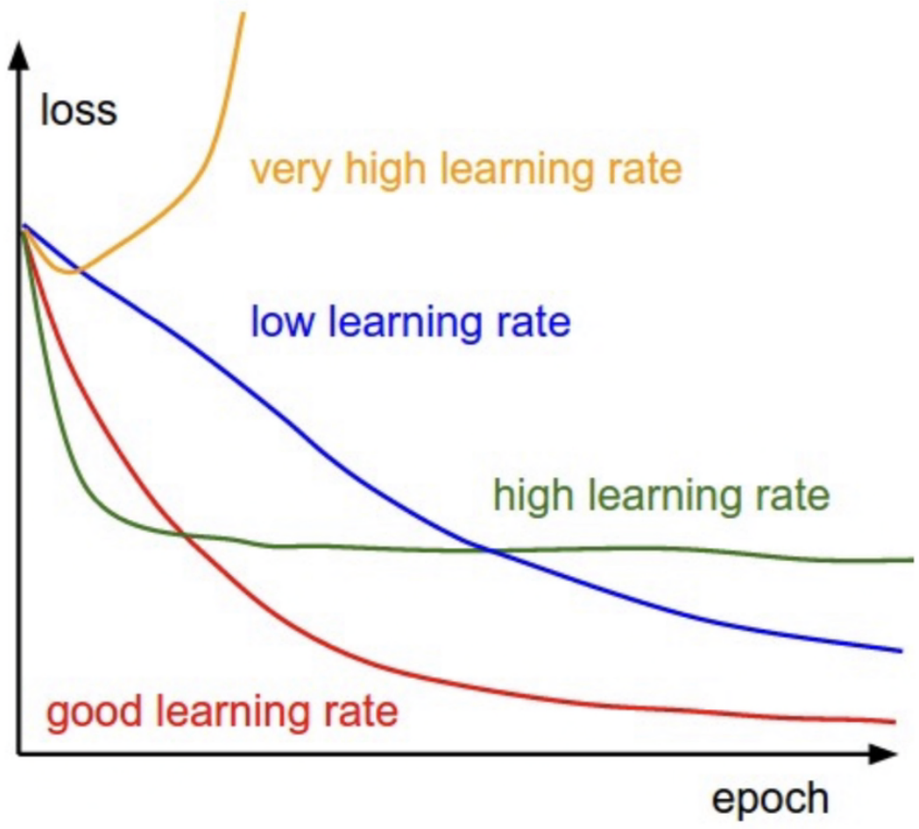
How much to update

- τ is a fixed constant ("fixed learning rate")
- τ is changing over time, e.g. $\tau_t \propto 1/t$
- add "momentum": $\theta^{(t+1)} = \theta^{(t)} - v^{(t+1)}$

$$v^{(t+1)} = \gamma v^{(t)} + \tau \nabla \hat{R}(\theta^{(t)})$$

Learning Rate Effects

- Check how your loss behaves during training, to spot wrong hyperparameters, bugs, etc



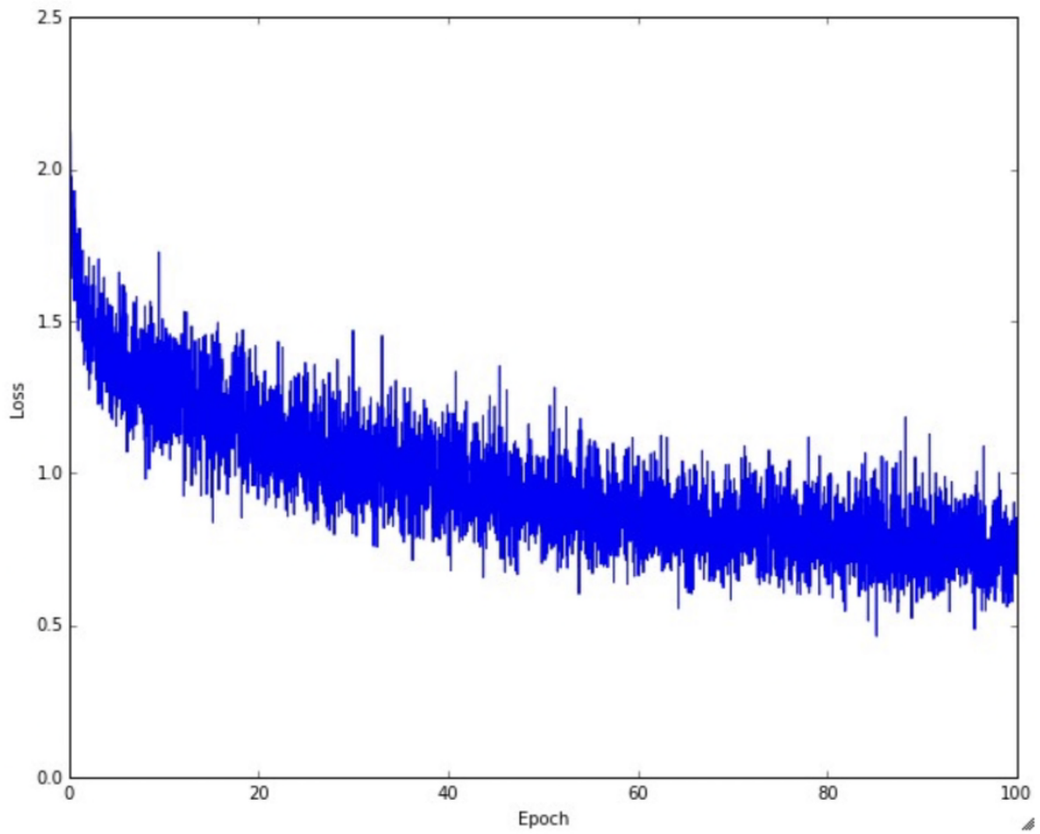


Figure : **Left:** Good vs bad parameter choices, **Right:** How a real loss might look like during training. What are the bumps caused by? How could we get a more smooth loss?