

BACKPROPAGATION AND DROPOUT



BATCHING

FULL BATCH GRADIENT DESCENT

So far in this class, we've passed the full training dataset into our model for each training step

Pro:

We get an exact derivative for the training set

Con:

Right now—it's really slow. Soon, it won't be computationally feasible

What can we do?

MINI-BATCHING

Idea:

Estimate derivative with a sample of training data

Pro:

Much faster to compute

Con:

Don't get the exact derivative

Side effect:

Can potentially get out of local minima, if loss function has them

STOCHASTIC GRADIENT DESCENT (SGD)

At an extreme, use a single example to estimate derivative

- Mini-batch with a size of 1

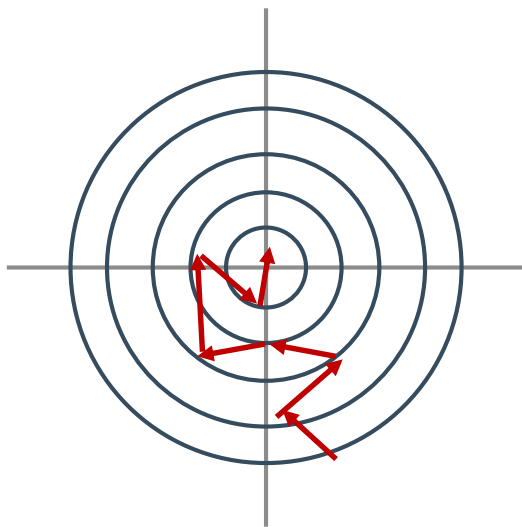
Also known as online training

Movement along the error curve is much more sporadic

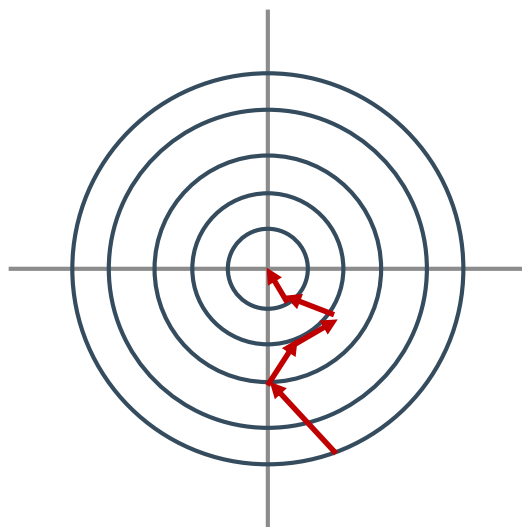
But also much easier to compute

WHAT STEPS MIGHT LOOK LIKE FOR DIFFERENT BATCH SIZES

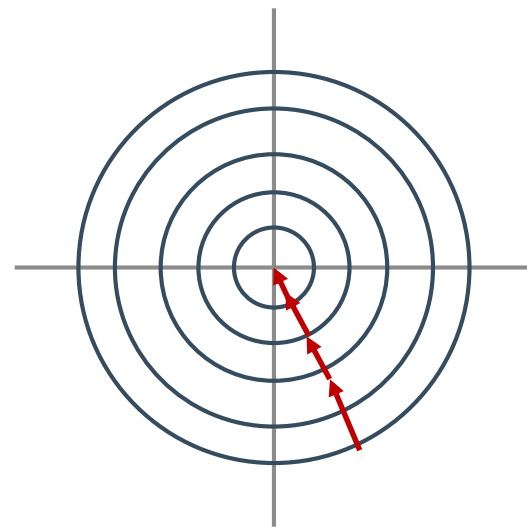
Stochastic



Mini-batch



Full batch



1

Batch size

N

Faster, less accurate step

Slower, more accurate step

BATCHING TERMINOLOGY CHEAT-SHEET

Full-batch gradient descent

- Use all of training data per step

Mini-batch gradient descent

- Use small portion of training data per step

Stochastic gradient descent (SGD)

- Use single example per step
- Occasionally refers to mini-batch gradient descent

EPOCHS

***An epoch* refers to one entire pass through the dataset**

The number of times a model has seen each training example

SHUFFLING

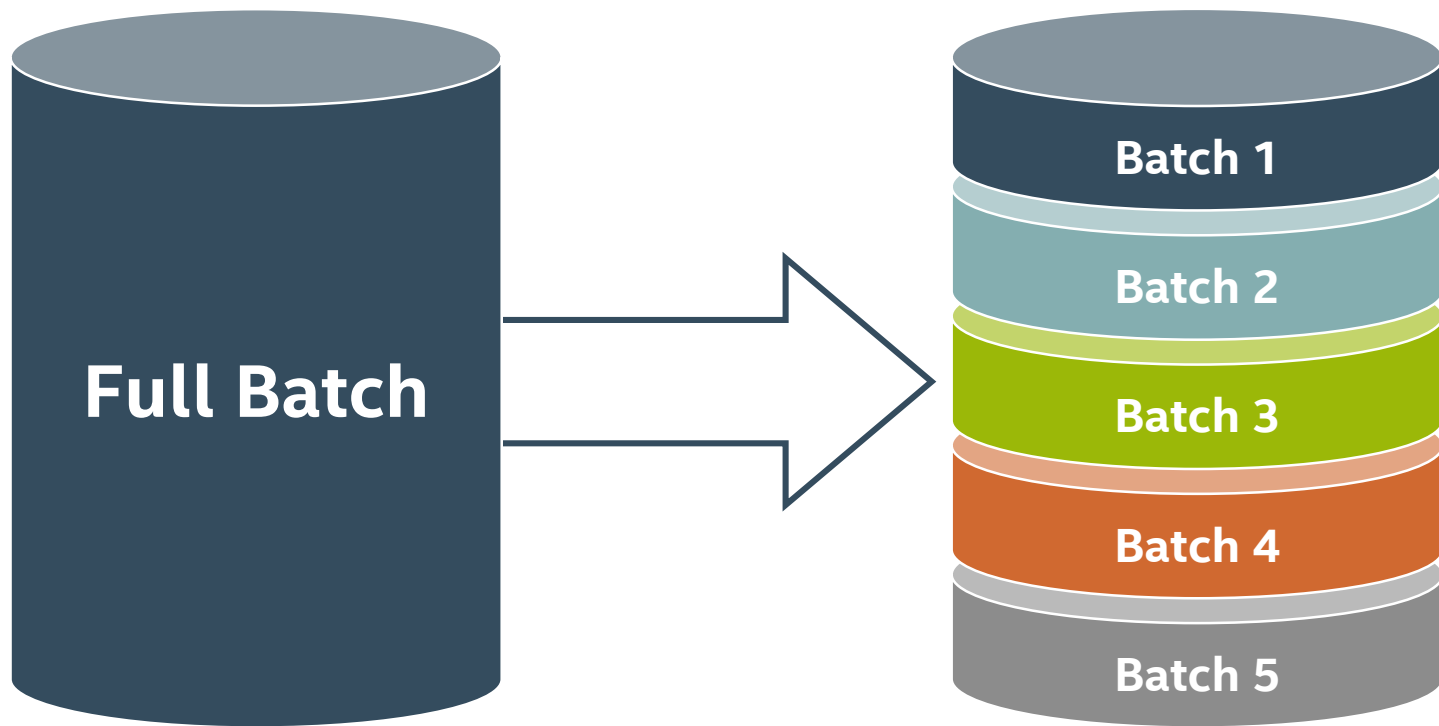
If we are using our data piecemeal, we must be careful:

- Don't want to oversample
- Avoid reusing same mini-batch over and over

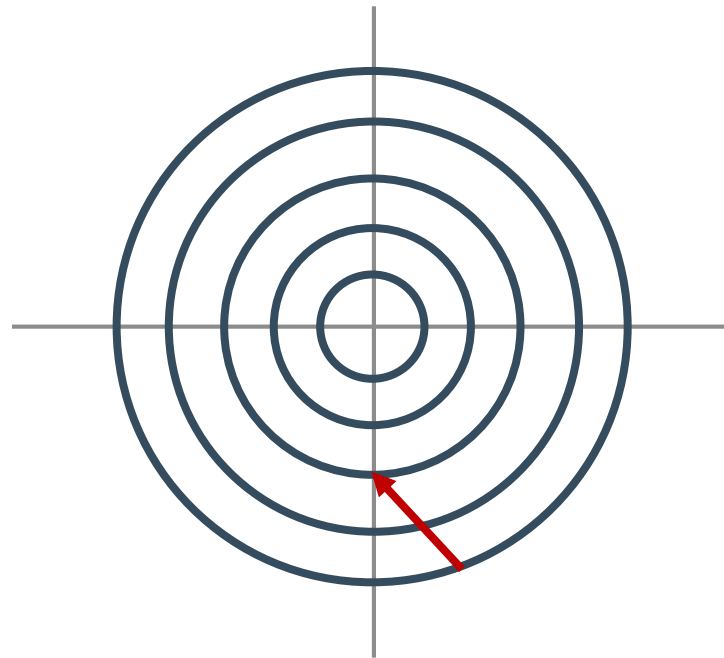
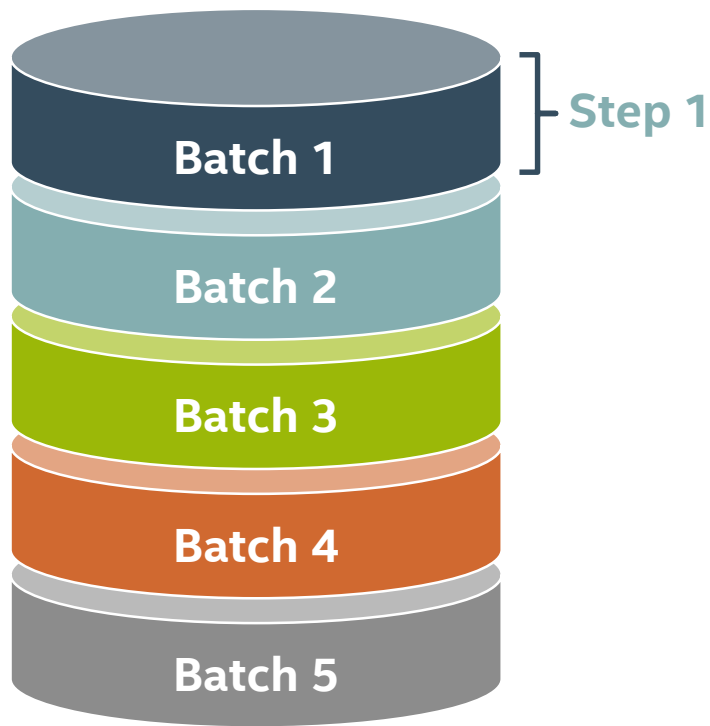
Solution: shuffle training data after each epoch

- Shuffle, make batches, repeat

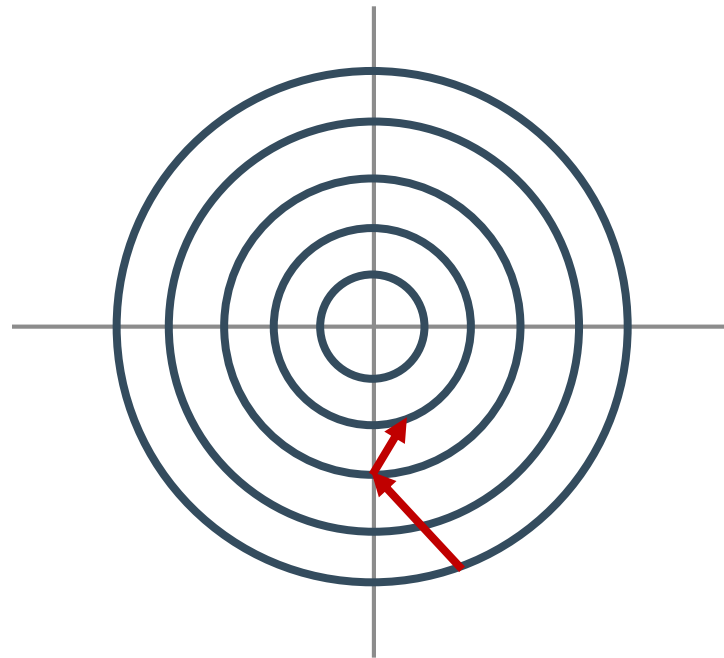
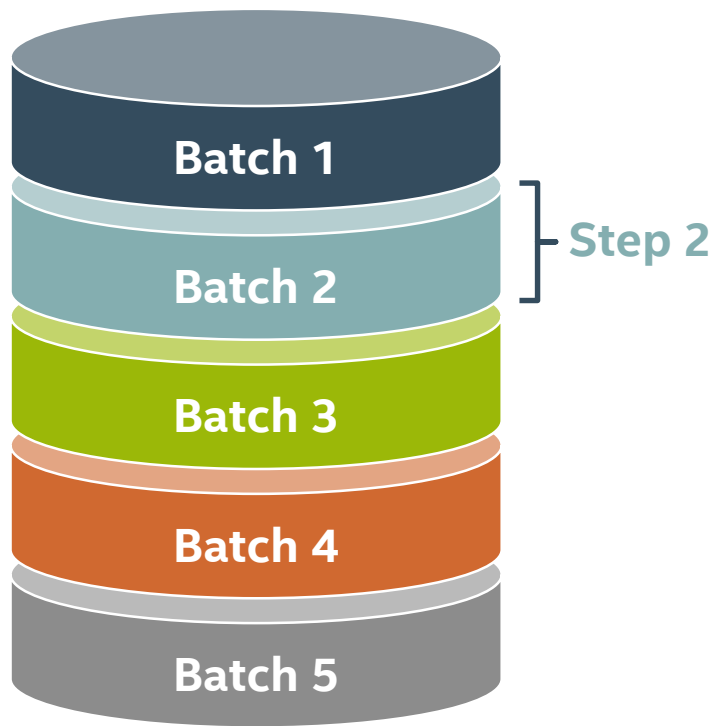
SPLITTING DATA UP INTO BATCHES



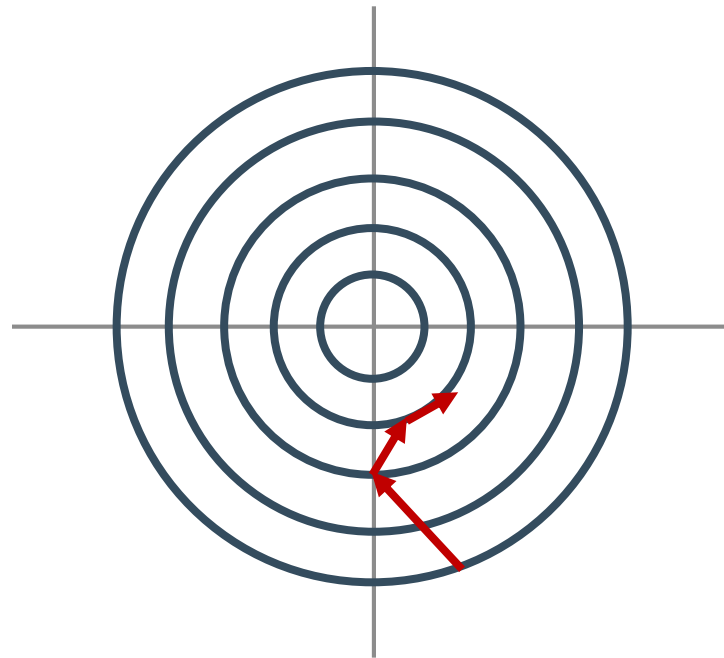
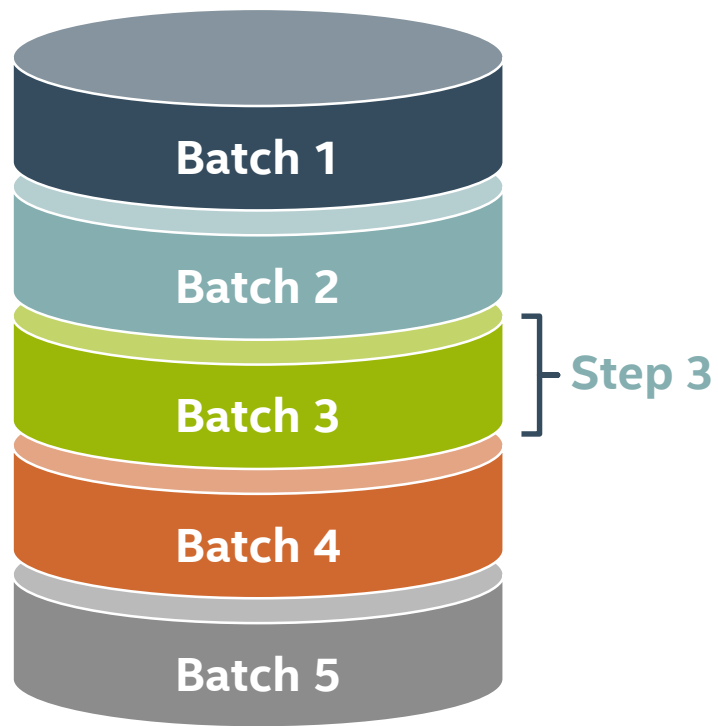
SPLITTING DATA UP INTO BATCHES



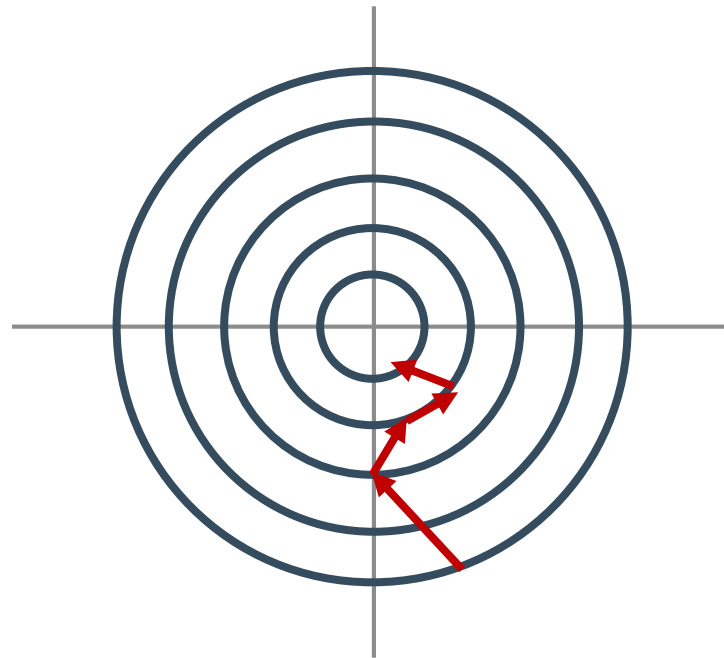
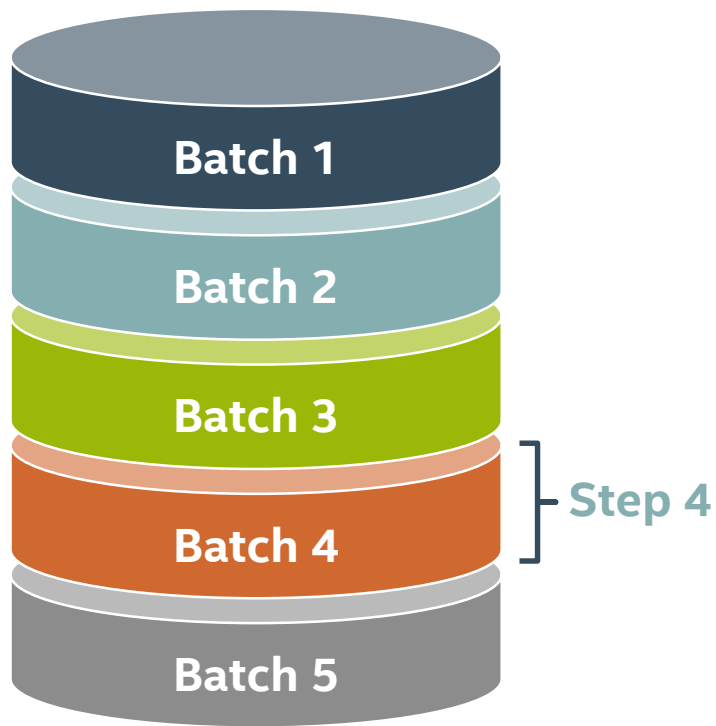
SPLITTING DATA UP INTO BATCHES



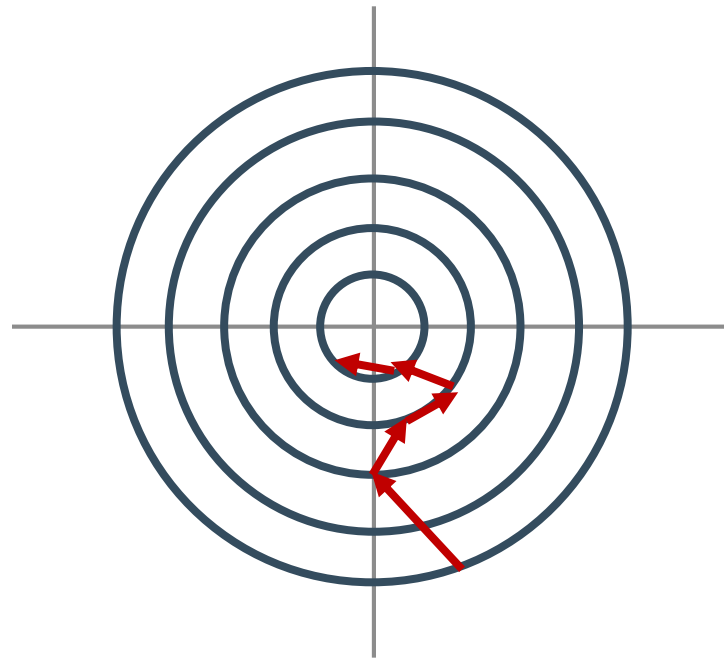
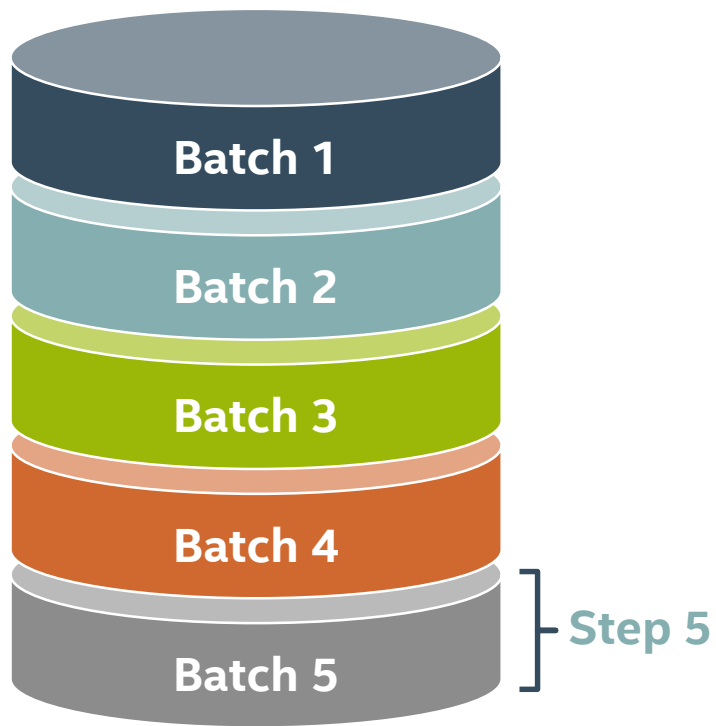
SPLITTING DATA UP INTO BATCHES



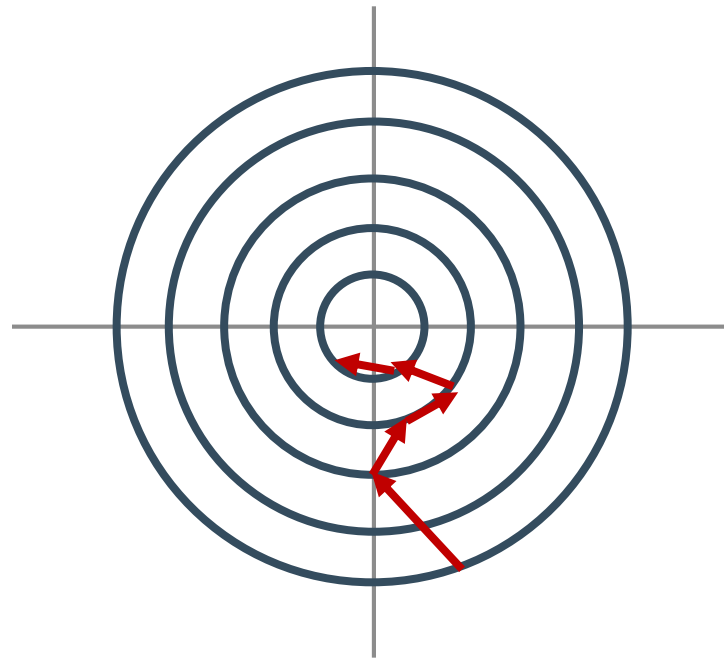
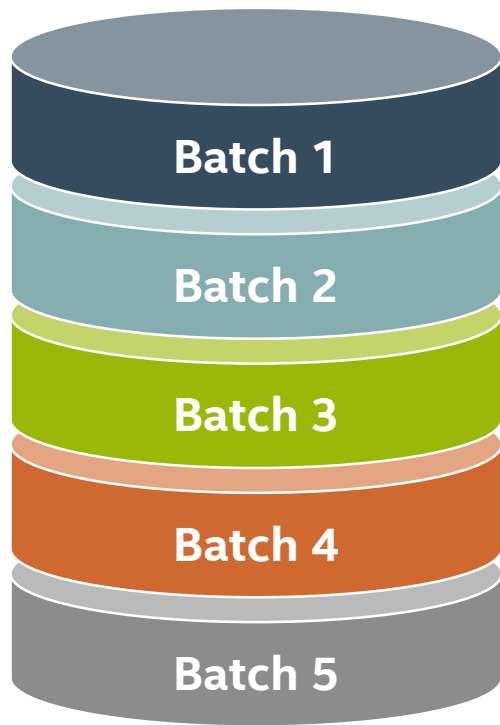
SPLITTING DATA UP INTO BATCHES



SPLITTING DATA UP INTO BATCHES



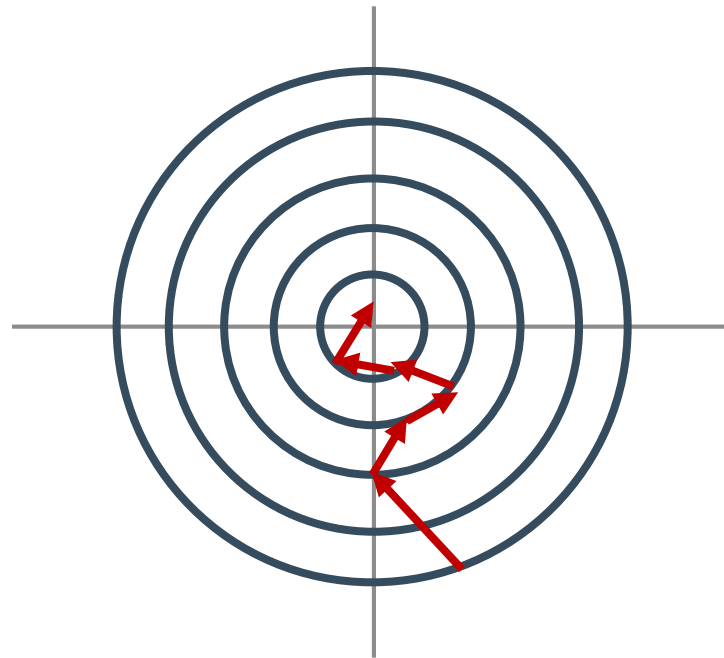
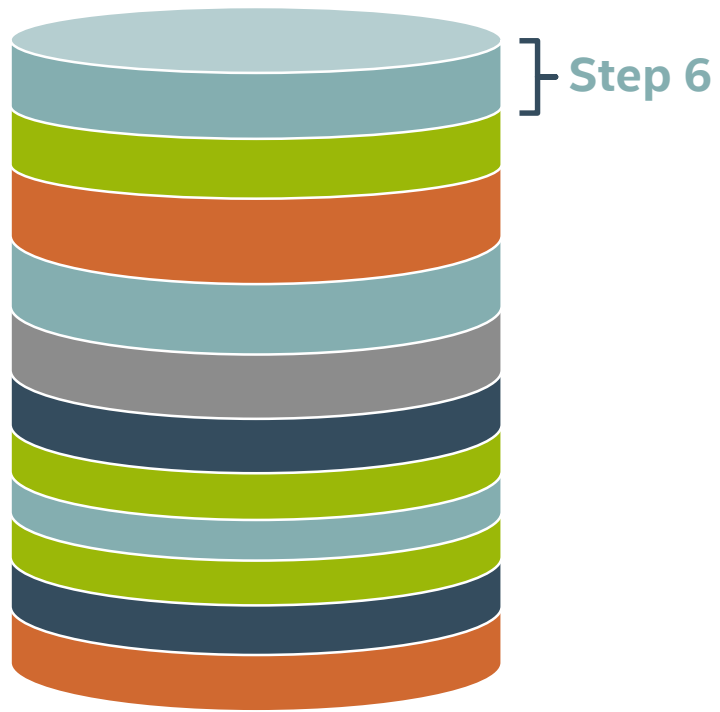
FIRST EPOCH COMPLETED



SHUFFLE THE DATA!



CONTINUE TRAINING



MULTI-CLASS CLASSIFICATION

ONE-HOT VECTORS

A vector that usually represents a single choice from among many options

One entry is 1, the rest are 0

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Cat

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Dog

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Toaster

MULTI-CLASS CLASSIFICATION

Goal: We want our model to be able to predict from one of many discrete classes

- E.g., Cat, dog, toaster, etc.

Idea: train model to output some sort of predictive vector

- Each entry in the vector corresponds to a single class's score

How?

THOUGHT 1: JUST USE A REGULAR VECTOR

Create an output layer that has the same size as number of classes

- Use the z values ($z=aW+b$) for that layer as the score

Problems:

- Super high variance in output
- Loss function isn't clean
 - The label vector is 1 at the index corresponding to the correct class, 0 elsewhere
 - How to relate binary data $\{0, 1\}$ to continuous values from $(-\infty, \infty)$?

A BETTER WAY: SOFTMAX

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}$$

Converts z to a probability vector

- Each z_i is mapped to $\{0, 1\}$
- Sum of softmaxes is equal to 1
- We call z *logits*

Can compare directly with our labels vector

LOSS FUNCTION: CROSS ENTROPY

$$C.E. = - \sum_{i=1}^n y_i \log(\hat{y}_i)$$

Taken from information theory

- Is high when \hat{y}_i and y_i are different
- Is low when they are similar

DERIVATIVE OF CROSS-ENTROPY WITH SOFTMAX INPUT

$$\frac{\partial C.E.}{\partial softmax} \cdot \frac{\partial softmax}{\partial z_i} = \hat{y}_i - y_i$$

Super easy to compute!

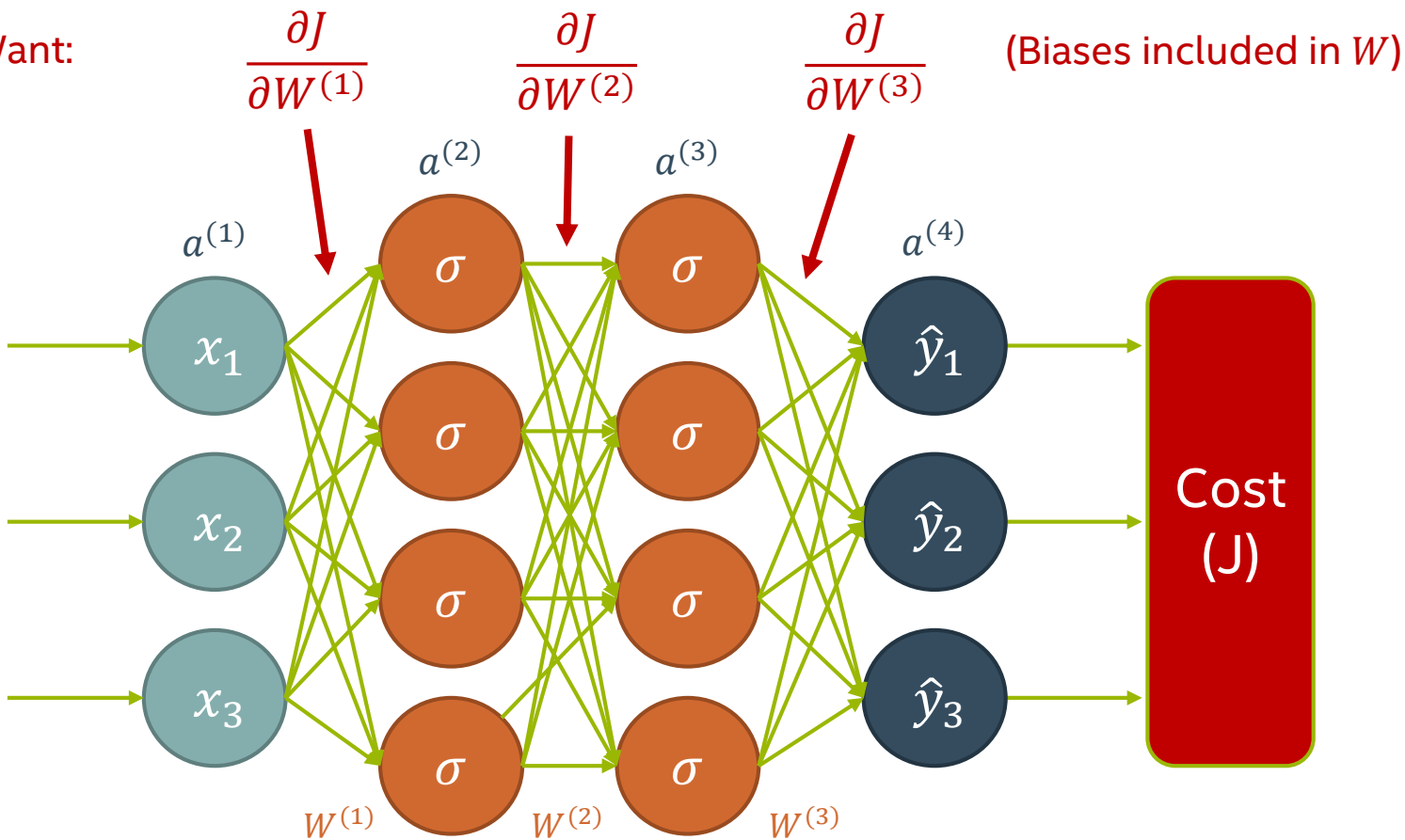
BACKPROPAGATION

BACKPROPAGATION

Idea: use the chain rule to find derivative of cost function with respect to each set of weights

Works basically the same as the automatic differentiation example we went over last week

Want:



EQUATIONS FOR THE MODEL

Layer 1: inputs, X

$$J = C.E.(\hat{y}, y)$$

Layer 2: first hidden layer

$$z^{(2)} = XW^{(1)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

Layer 3: second hidden layer

$$z^{(3)} = a^{(2)}W^{(2)}$$

$$a^{(3)} = \sigma(z^{(3)})$$

Layer 4: output

$$z^{(4)} = a^{(3)}W^{(3)}$$

$$\hat{y} = \text{softmax}(z^{(4)})$$

GETTING PARTIAL DERIVATIVES

$$z^{(2)} = XW^{(1)}$$

$$z^{(3)} = a^{(2)}W^{(2)}$$

$$z^{(4)} = a^{(3)}W^{(3)}$$

$$J = C.E.(\hat{y}, y)$$

$$a^{(2)} = \sigma(z^{(2)})$$

$$a^{(3)} = \sigma(z^{(3)})$$

$$\hat{y} = softmax(z^{(4)})$$

GETTING PARTIAL DERIVATIVES

$$J = C.E.(\hat{y}, y)$$

$$z^{(2)} = XW^{(1)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)}$$

$$a^{(3)} = \sigma(z^{(3)})$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

GETTING PARTIAL DERIVATIVES

$$z^{(2)} = XW^{(1)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)}$$

$$a^{(3)} = \sigma(z^{(3)})$$

$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}; \quad \frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

GETTING PARTIAL DERIVATIVES

$$z^{(2)} = XW^{(1)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)}$$

$$\frac{\partial a^{(3)}}{\partial z^{(3)}} = \sigma'(z^{(3)})$$

$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}; \quad \frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

GETTING PARTIAL DERIVATIVES

$$z^{(2)} = XW^{(1)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = a^{(2)}; \frac{\partial z^{(3)}}{\partial a^{(2)}} = W^{(2)}$$

$$\frac{\partial a^{(3)}}{\partial z^{(3)}} = \sigma'(z^{(3)})$$

$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}; \frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

GETTING PARTIAL DERIVATIVES

$$z^{(2)} = XW^{(1)}$$

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = a^{(2)}; \quad \frac{\partial z^{(3)}}{\partial a^{(2)}} = W^{(2)}$$

$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}; \quad \frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = \sigma'(z^{(2)})$$

$$\frac{\partial a^{(3)}}{\partial z^{(3)}} = \sigma'(z^{(3)})$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

GETTING PARTIAL DERIVATIVES

$$\frac{\partial z^{(2)}}{\partial W^{(1)}} = X$$

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = \sigma'(z^{(2)})$$

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = a^{(2)}; \quad \frac{\partial z^{(3)}}{\partial a^{(2)}} = W^{(2)}$$

$$\frac{\partial a^{(3)}}{\partial z^{(3)}} = \sigma'(z^{(3)})$$

$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}; \quad \frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

CAN NOW USE THE CHAIN RULE!

$$\frac{\partial z^{(2)}}{\partial W^{(1)}} = X$$

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = \sigma'(z^{(2)})$$

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = a^{(2)}; \quad \frac{\partial z^{(3)}}{\partial a^{(2)}} = W^{(2)}$$

$$\frac{\partial a^{(3)}}{\partial z^{(3)}} = \sigma'(z^{(3)})$$

$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}; \quad \frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial J}{\partial z^{(4)}} = \hat{y} - y$$

CAN NOW USE THE CHAIN RULE!

$$\frac{\partial J}{\partial W^{(3)}} = \frac{\partial J}{\partial z^{(4)}} \cdot \frac{\partial z^{(4)}}{\partial W^{(3)}}$$

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial J}{\partial z^{(4)}} \cdot \frac{\partial z^{(4)}}{\partial a^{(3)}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial W^{(2)}}$$

$$\frac{\partial J}{\partial W^{(1)}} = \frac{\partial J}{\partial z^{(4)}} \cdot \frac{\partial z^{(4)}}{\partial a^{(3)}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial W^{(1)}}$$

FILL IN FOR SPECIFIC VALUES

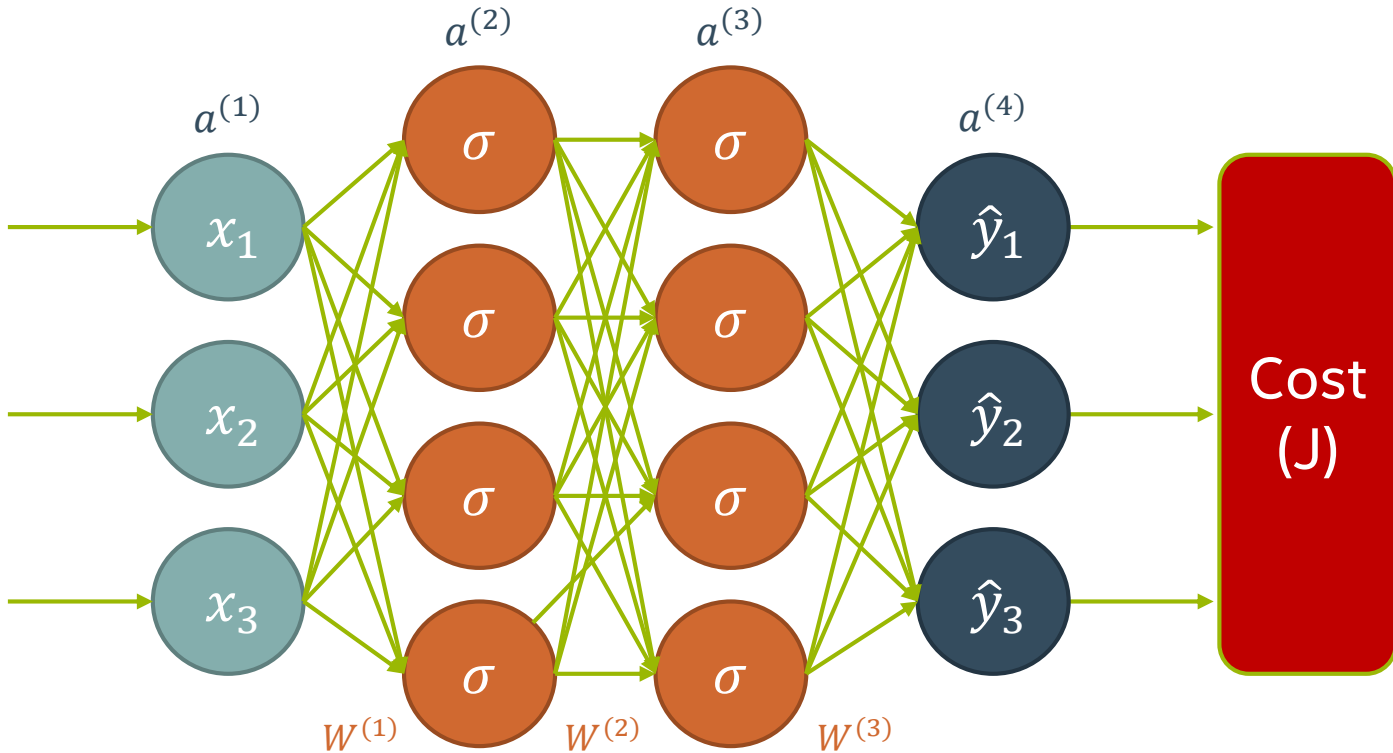
$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3) \cdot a^{(2)}$$

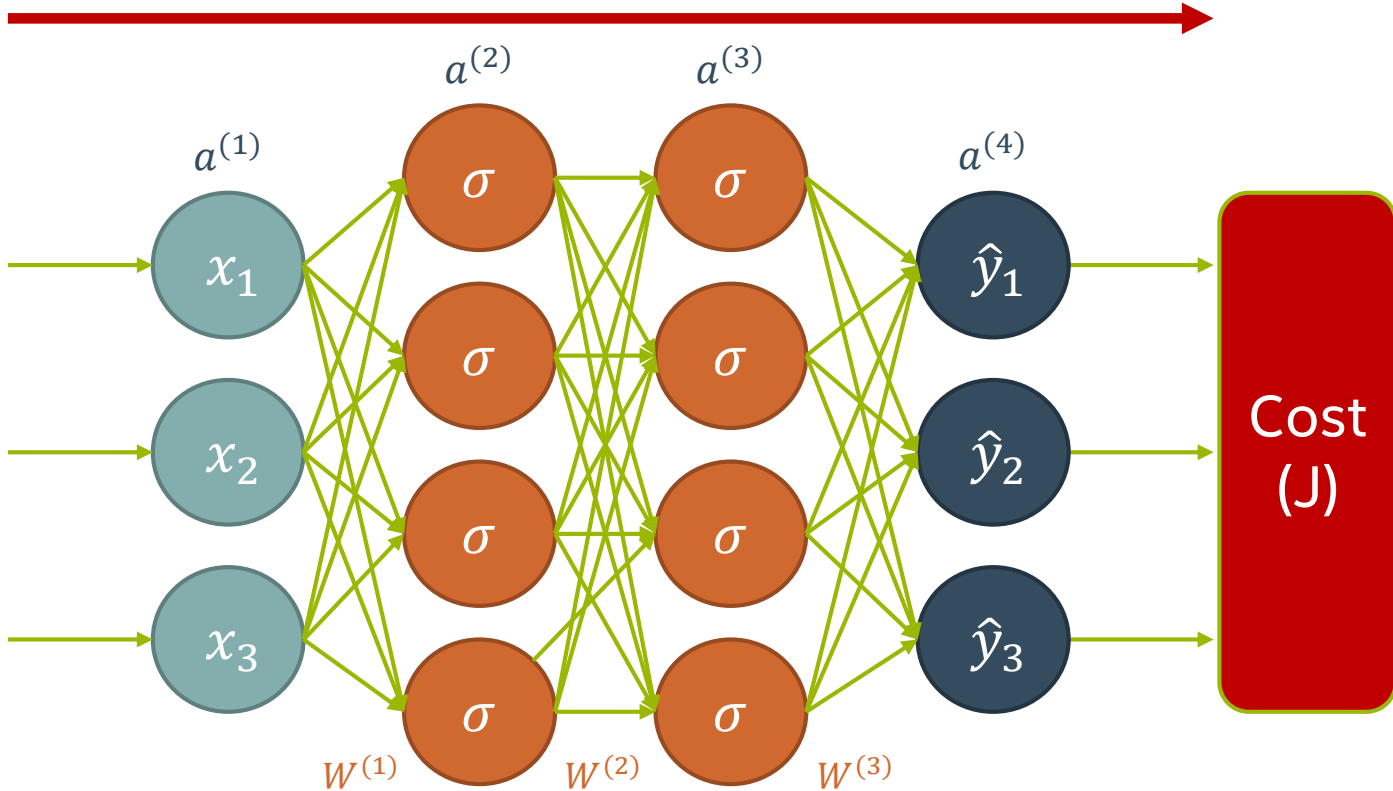
$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3) \cdot W^{(2)} \cdot \sigma'(z^{(2)}) \cdot X$$

VISUAL INTUITION

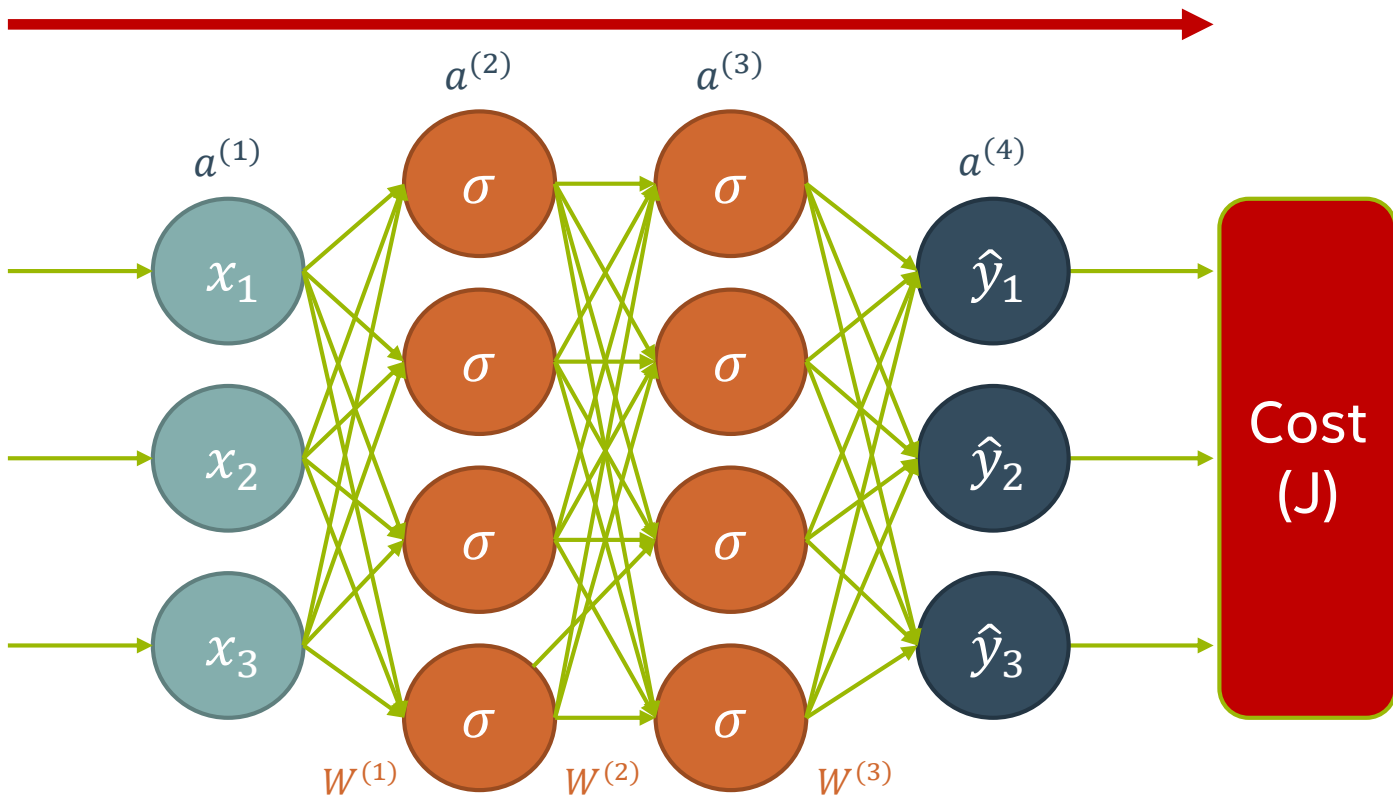
VISUALLY, WHAT WE ARE DOING IS ATTRIBUTING ERROR TO SPECIFIC WEIGHTS



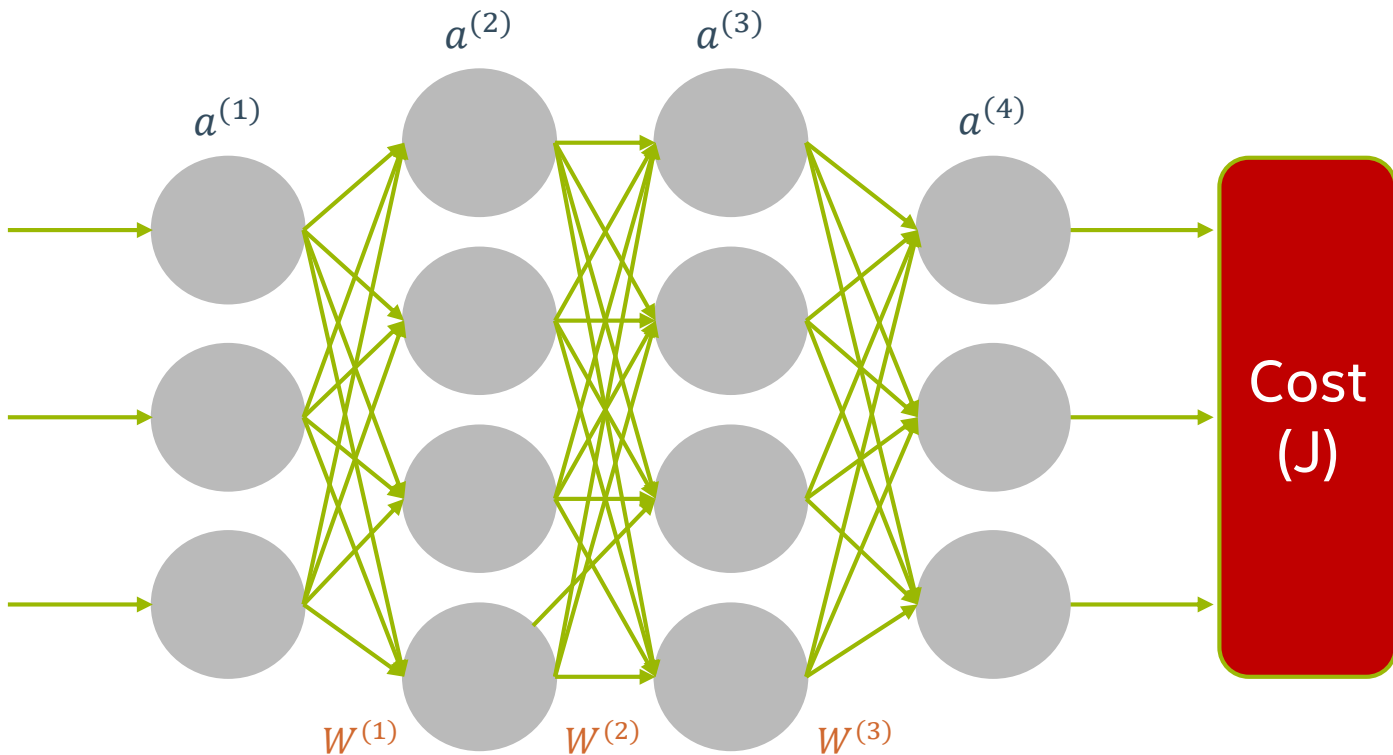
FIRST, WE PASS DATA INTO THE MODEL TO GET A PREDICTION



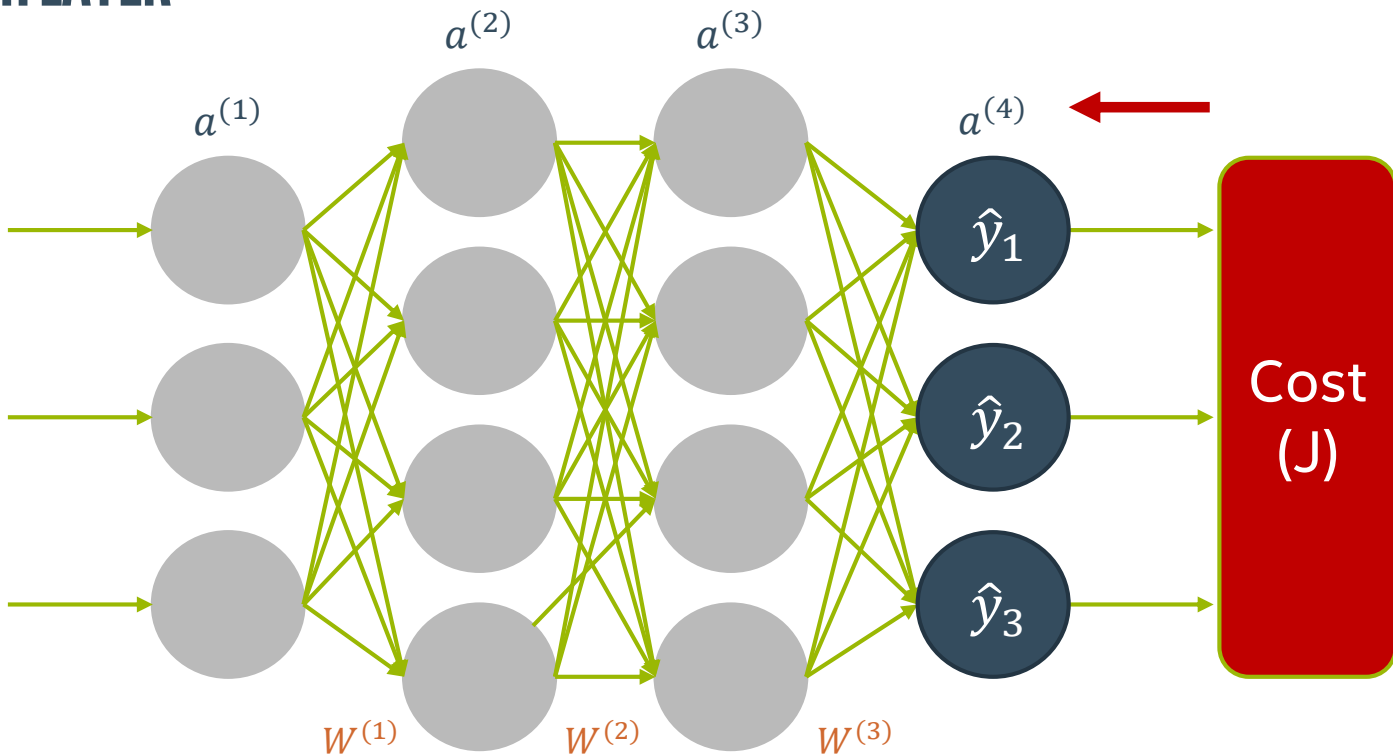
THIS IS KNOWN AS FORWARD PROPAGATION, OR FORWARD-PROP



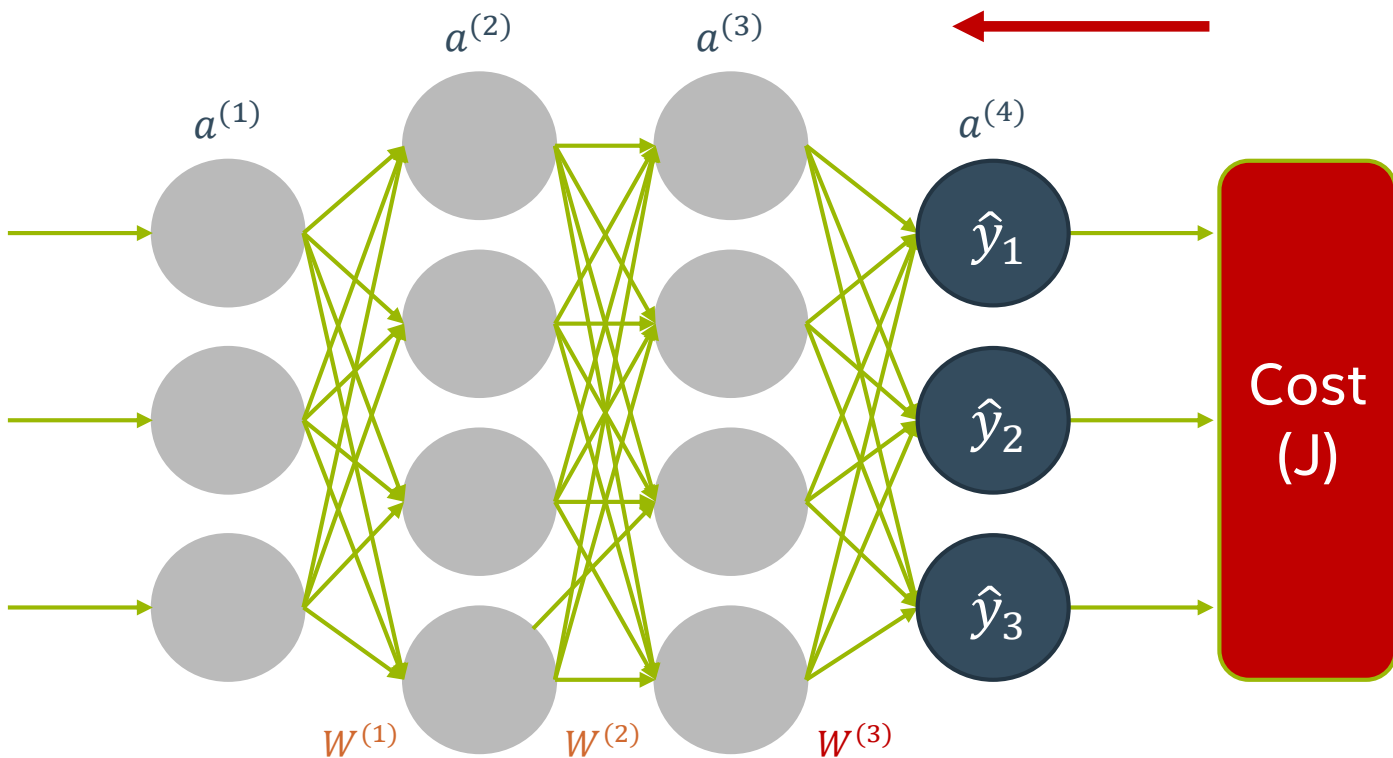
THIS LEAVES US WITH THE COST FUNCTION



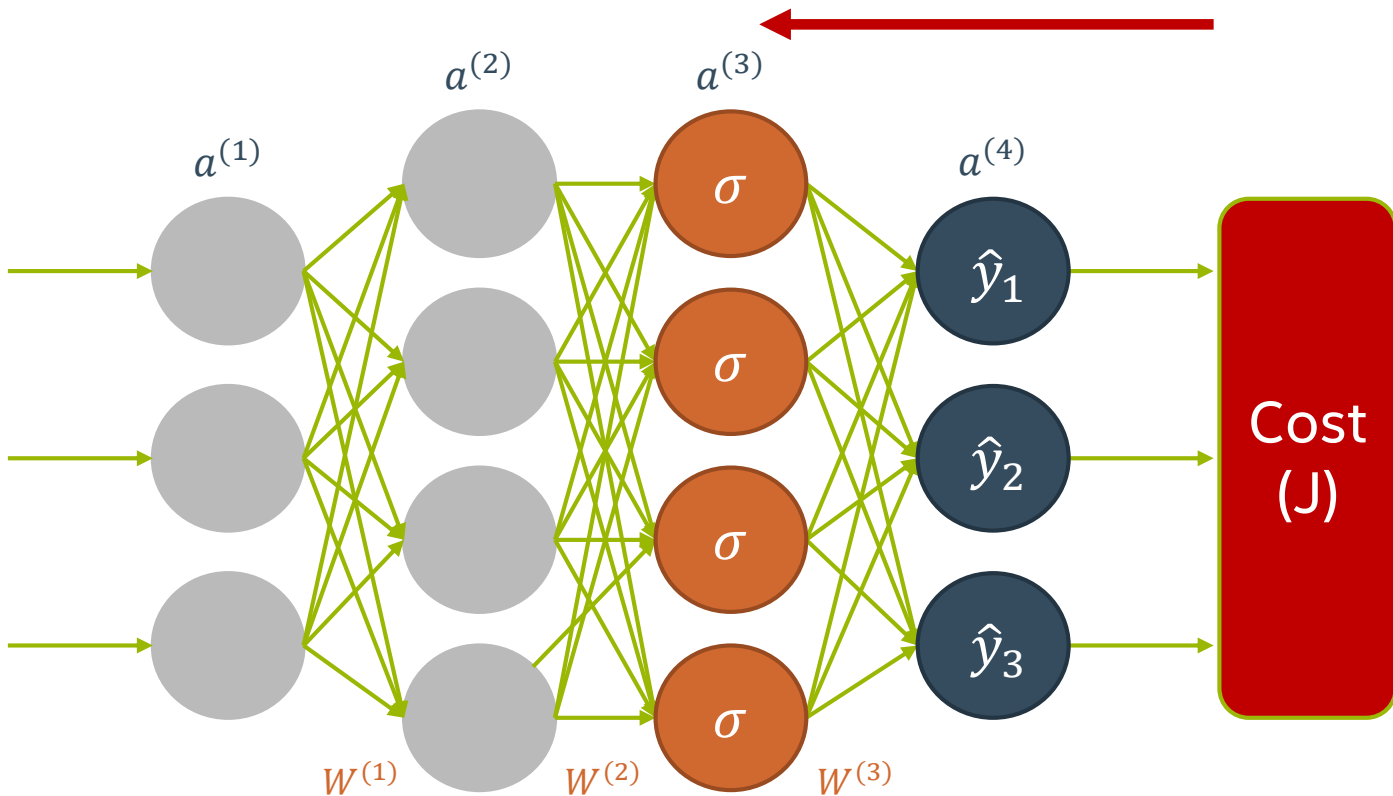
USE THE CHAIN RULE, WORK OUR WAY BACK, APPLY LOCAL DERIVATIVES AT EACH LAYER



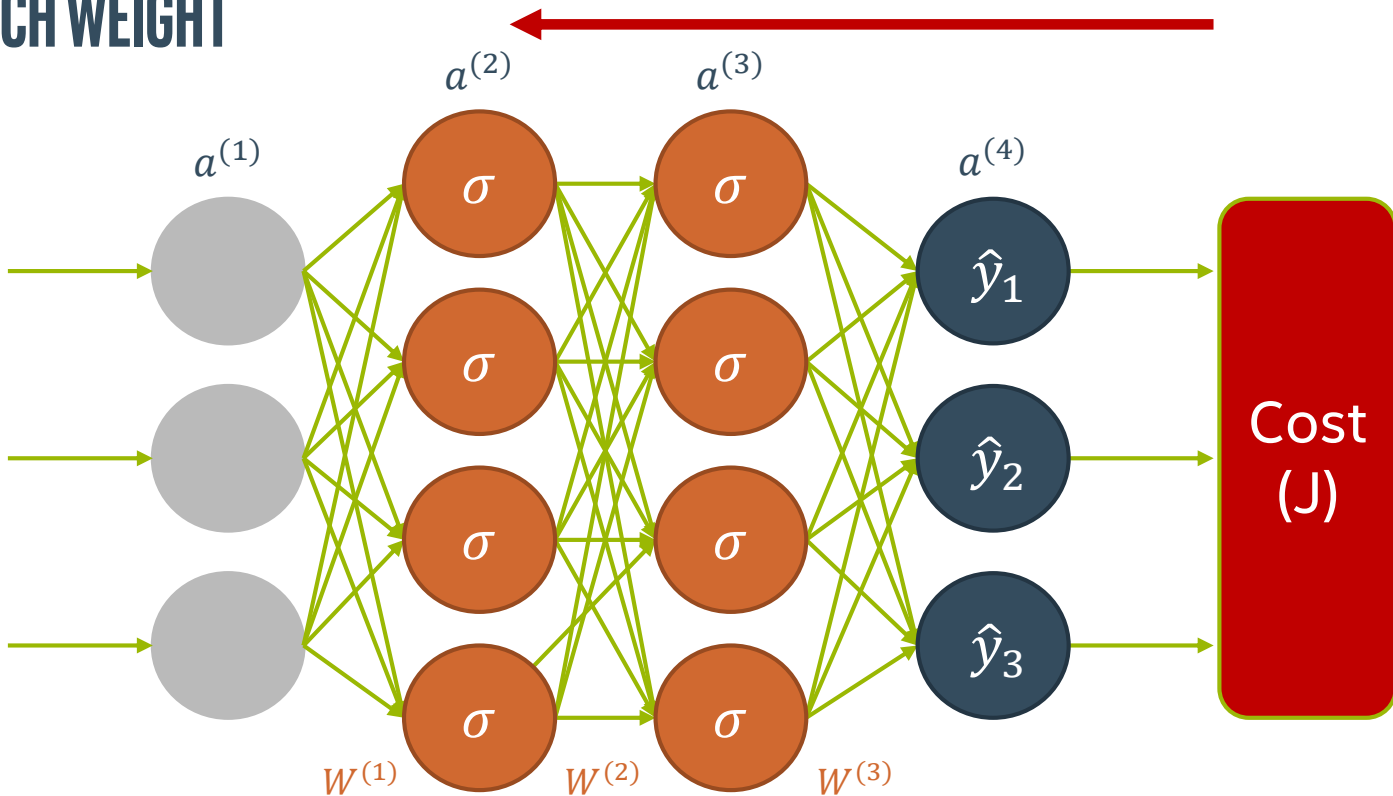
AT THIS POINT WE CAN TAKE THE DERIVATIVE W.R.T $W^{(3)}$



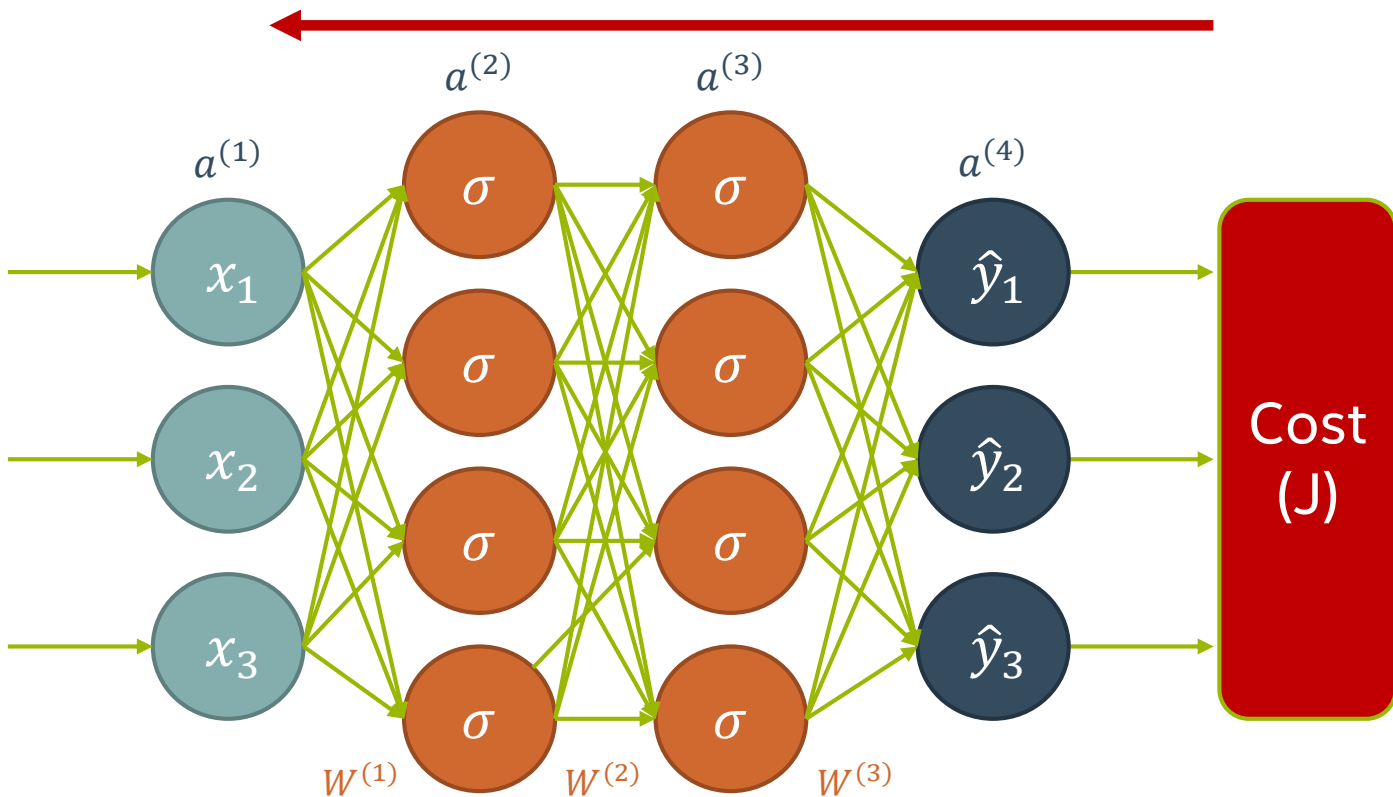
THEN W.R.T $a^{(3)}$ TO CONTINUE THE CHAIN



THIS PROCESS CONTINUES AS WE ATTEMPT TO GET THE DERIVATIVE FOR EACH WEIGHT



UNTIL WE HAVE FOUND ALL THE DERIVATIVES WE NEED



THE ERROR SIGNAL: δ

Looking at these equations, we may notice a pattern:

$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3) \cdot a^{(2)}$$

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3) \cdot W^{(2)} \cdot \sigma'(z^{(2)}) \cdot X$$

THE ERROR SIGNAL: δ

Let's call the error signal to a layer, $\delta^{(l)}$, the derivative of the cost function w.r.t. net inputs $z^{(l)}$

$$\delta^{(4)} = \frac{\partial J}{\partial z^{(4)}} = (\hat{y} - y)$$

We can use the error signal to compute the derivative w.r.t the previous layer's weight and *error signal*

$$\frac{\partial J}{\partial W^{(l)}} = \delta^{(l+1)} \cdot a^{(l)}$$

$$\delta^{(l)} = \delta^{(l+1)} \cdot W^{(l)} \cdot \sigma'(z^{(l)})$$

$$\frac{\partial J}{\partial W^{(l)}} = \delta^{(l+1)} \cdot a^{(l)}$$

$$\delta^{(l)} = \delta^{(l+1)} \cdot W^{(l)} \cdot \sigma'(z^{(l)})$$

$$\frac{\partial J}{\partial W^{(3)}} = \overbrace{(\hat{y} - y)}^{\delta^{(4)}} \cdot a^{(3)}$$

$$\frac{\partial J}{\partial W^{(2)}} = \overbrace{(\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3)}^{\delta^{(3)}} \cdot a^{(2)}$$

$$\frac{\partial J}{\partial W^{(1)}} = \overbrace{(\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3) \cdot W^{(2)} \cdot \sigma'(z^{(2)})}^{\delta^{(2)}} \cdot X$$

ERROR SIGNAL IN PRACTICES

Allows you to easily code a repeatable pattern when implementing backpropagation by hand

That said, you probably won't be thinking about the error signal when you use TensorFlow

May come up in papers you read or other resources you learn from in the future

MENTAL BREAK

A POTENTIAL PROBLEM

Take a look at this formula for the gradient w.r.t. $W^{(1)}$:

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'(z^3) \cdot W^{(2)} \cdot \sigma'(z^{(2)}) \cdot X$$

We're going to want a deep network!

- More layers allows more complex representations

Is there something that might get screwy due to the functions we're using?

RANGE OF $\sigma'(z)$?


Recall derivative of the sigmoid function:

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

Range of $\sigma'(z) = (0, 0.25]$. Consequences?

WHAT HAPPENS TO OUR GRADIENT, AS LAYERS INCREASE?

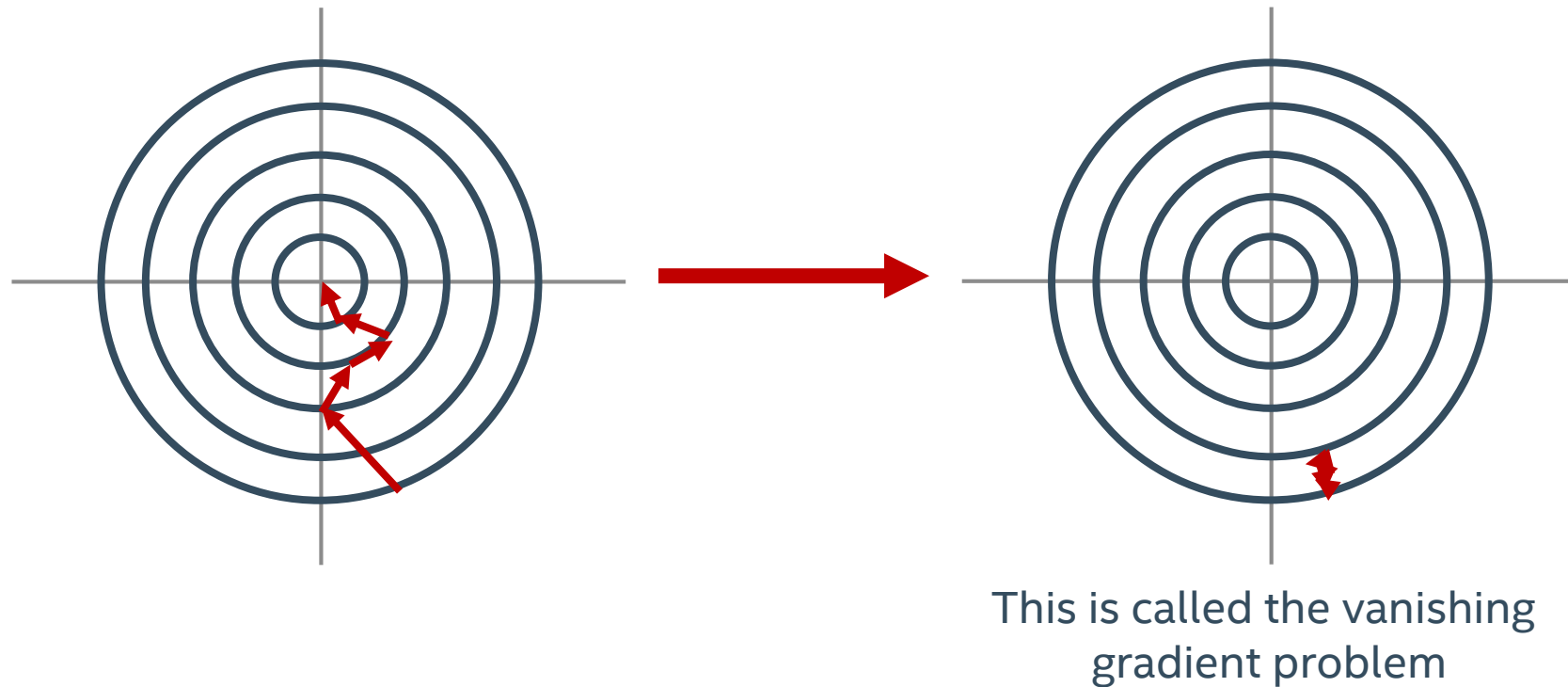
Assume we have many layers

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(L-1)} \cdot \sigma'(z^{L-2}) \cdot W^{(L-2)} \cdot \sigma'(z^{L-2}) \cdot \dots \cdot W^{(2)} \cdot \sigma'(z^{(2)}) \cdot X$$


Gradient gets multiplied by numbers strictly between $(0, 0.25]$ repeatedly

And, the gradient goes to zero! (Ugh.)

WHAT HAPPENS IF GRADIENT IS CLOSE TO ZERO?



NOT TRAINING IS BAD

So we can't make a model too deep right now.

- But we want a deep network for better models!

If the sigmoid is the problem, can we change it?

- Alternative activation functions

SUITE OF ACTIVATION FUNCTIONS

NON-LINEARITIES FOR DEEP LEARNING

These functions are the most commonly used in deep learning

All of them have easy derivatives

SIGMOID (LOGISTIC)

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

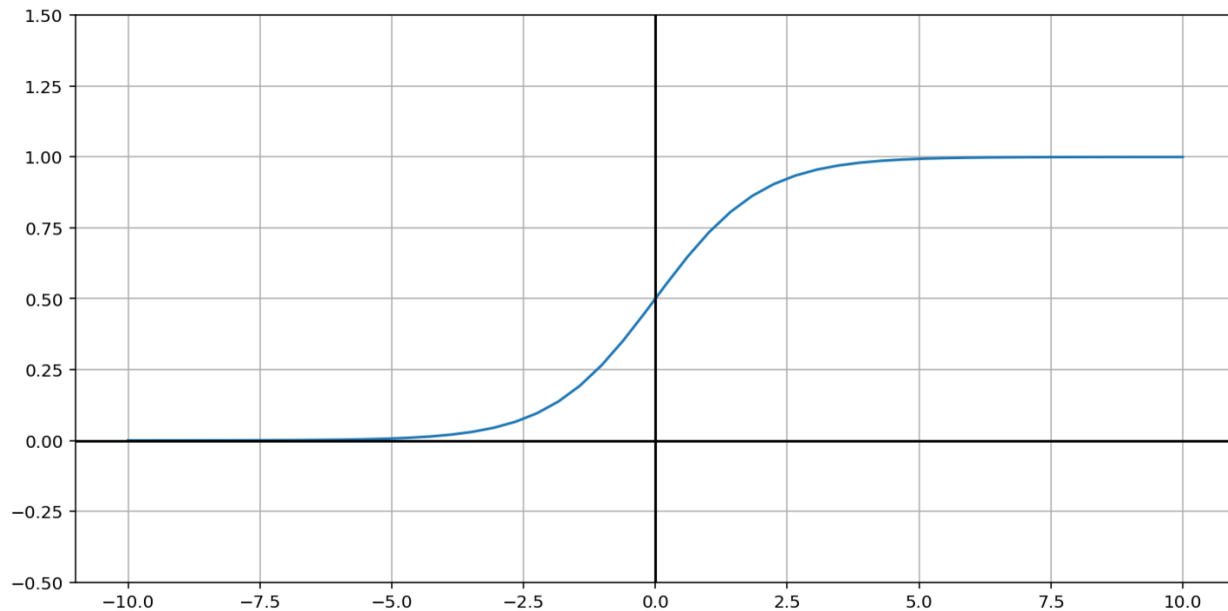
Value at $z \ll 0$? ≈ 0

Value at $z = 0$? $= 0.5$

Value at $z \gg 0$? ≈ 1

SIGMOID (LOGISTIC)

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



HYPERBOLIC TANGENT (TANH)

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

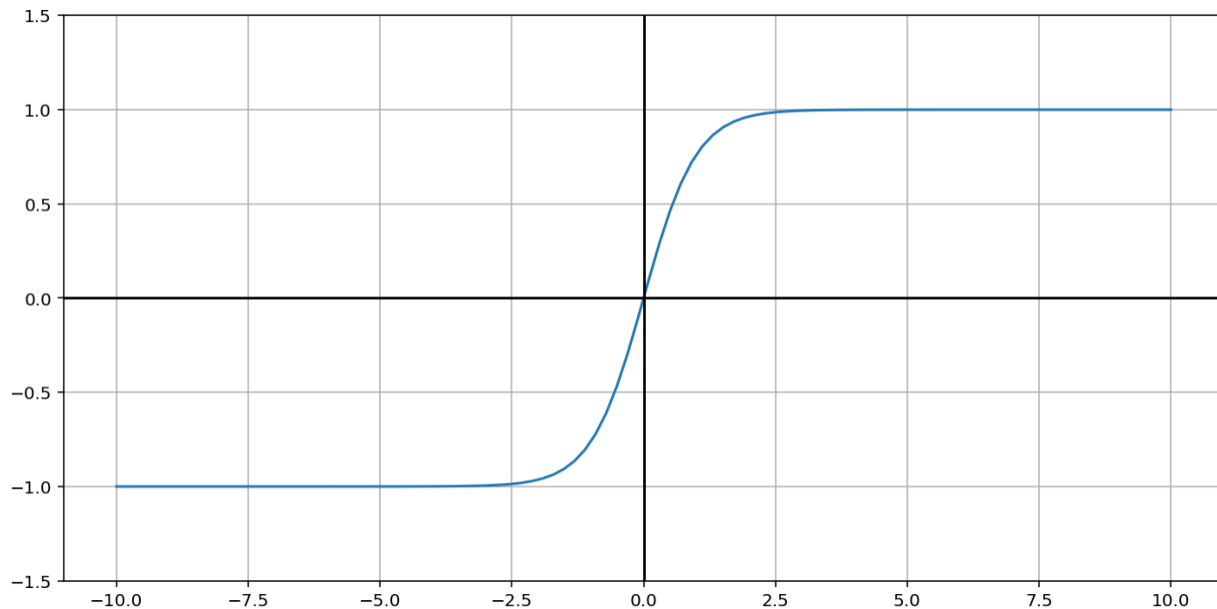
Value at $z \ll 0$? ≈ -1

Value at $z = 0$? $= 0.$

Value at $z \gg 0$? ≈ 1

HYPERBOLIC TANGENT

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x} - 1}{e^{2x} + 1}$$



RECTIFIED LINEAR UNIT (RELU)

$$\text{ReLU}(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$
$$= \max(0, z)$$

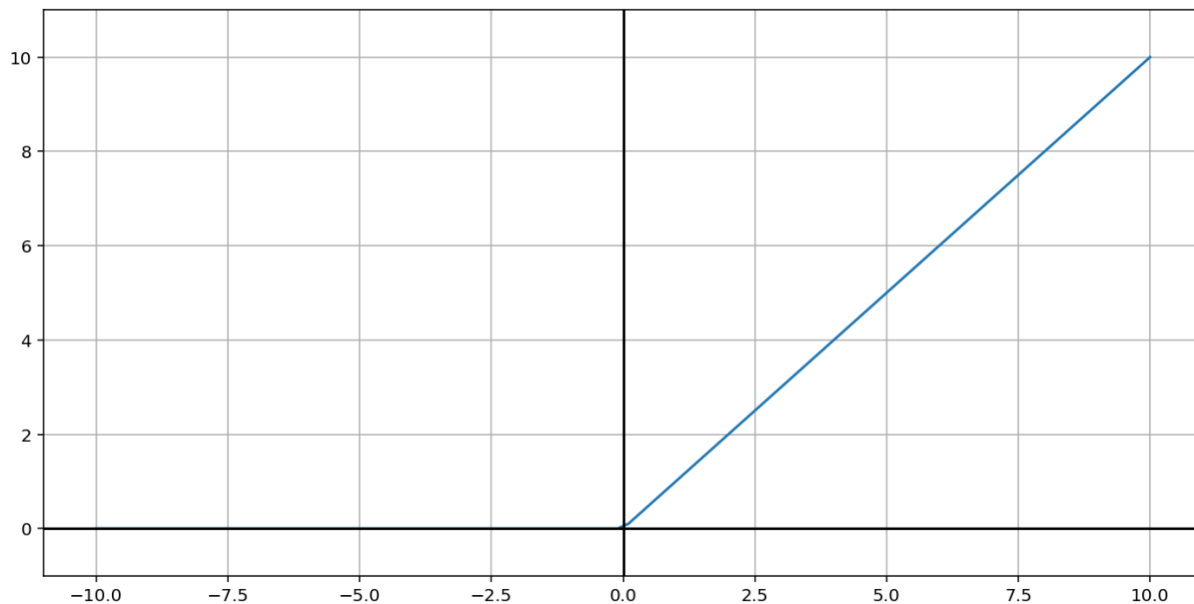
Value at $z \ll 0$? $= 0$

Value at $z = 0$? $= 0.$

Value at $z \gg 0$? $= z$

RECTIFIED LINEAR UNIT

$$\text{ReLU}(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$



LEAKY RELU

$$LReLU(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$
$$= \max(\alpha z, z)$$

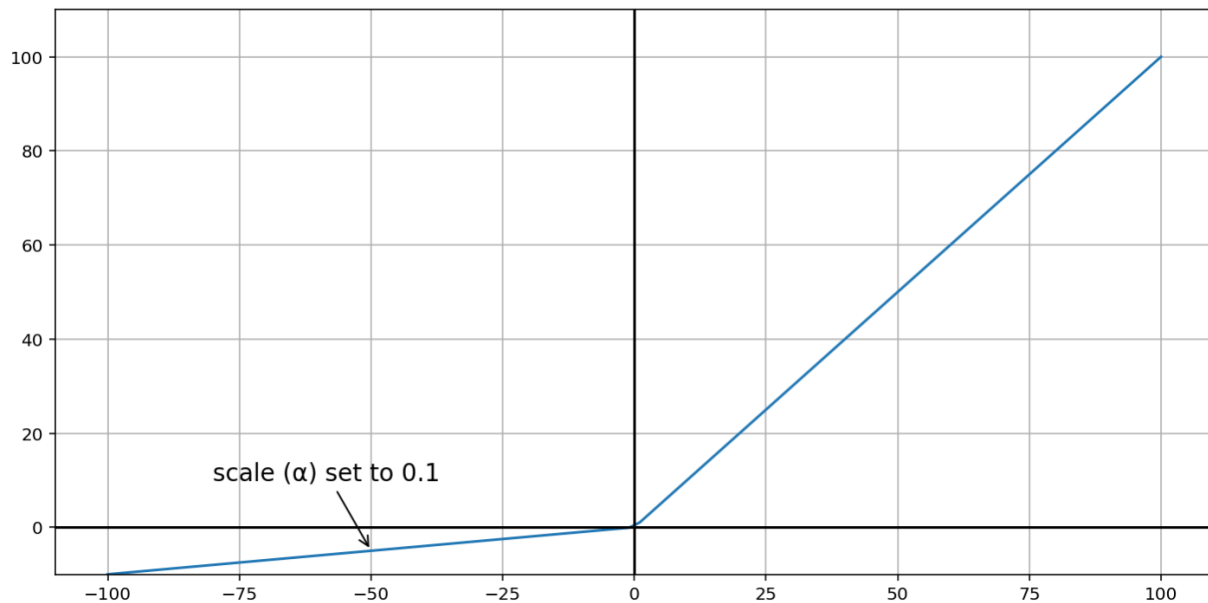
Value at $z \ll 0$? $= \alpha z$

Value at $z = 0$? $= 0.$

Value at $z \gg 0$? $= z$

LEAKY RELU

$$LReLU(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$



DROPOUT

Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov, 2014

DROPOUT OVERVIEW

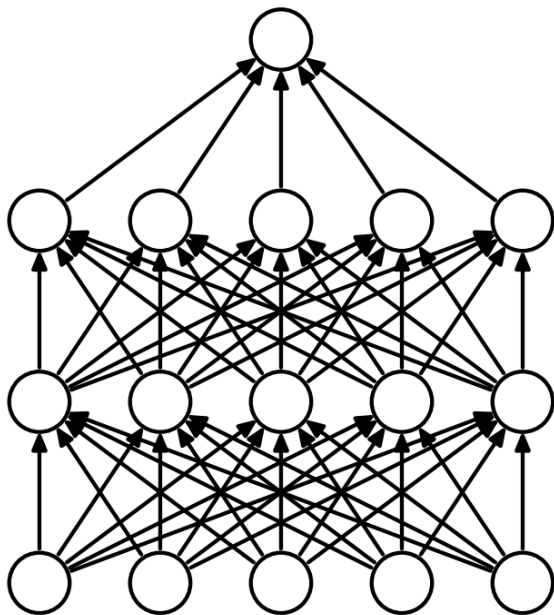
Neural networks can represent extremely complex data

- Very large number of parameters allows NNs to memorize a dataset

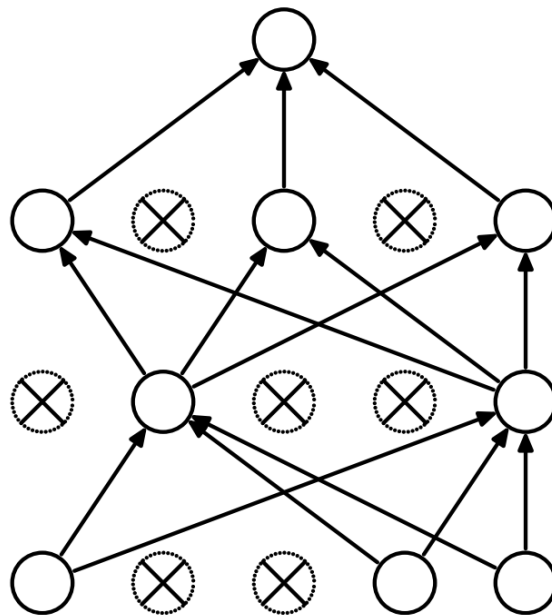
We want to regularize (smooth) their solution

- Prevent single neurons from dominating
- Require other neurons to be more flexible
- Randomly zero the output of neurons during training
 - Other neurons have to adapt

DROPOUT MODEL

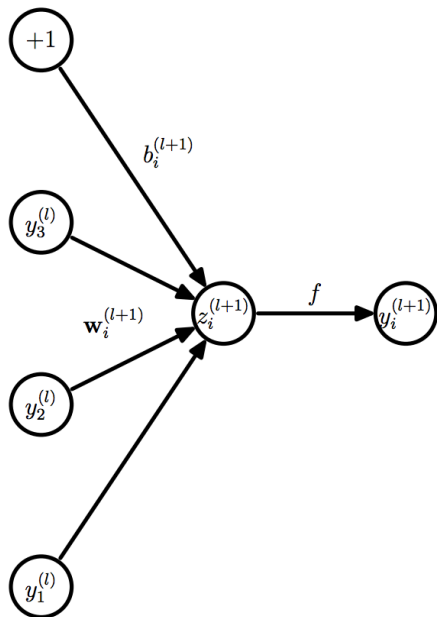


(a) Standard Neural Net

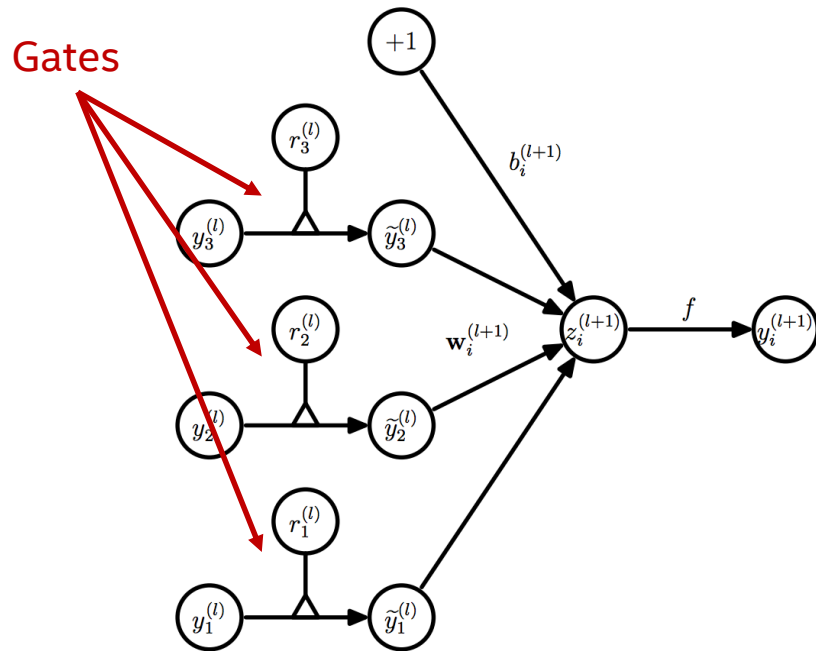


(b) After applying dropout.

DROPOUT LAYER

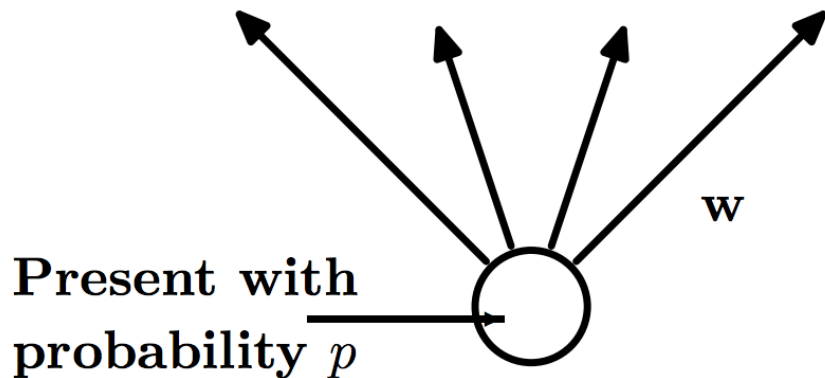


(a) Standard network

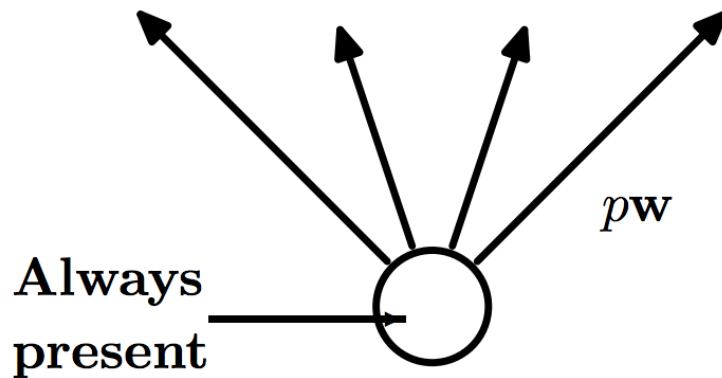


(b) Dropout network

KNOCKING OUT AND RESCALING NEURONS



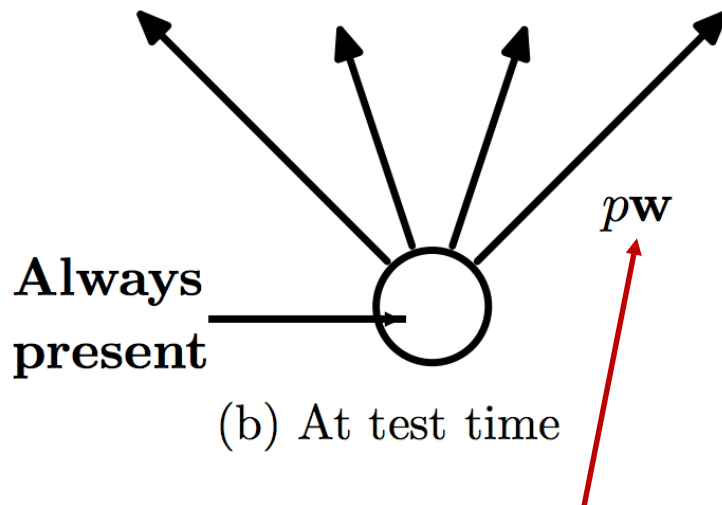
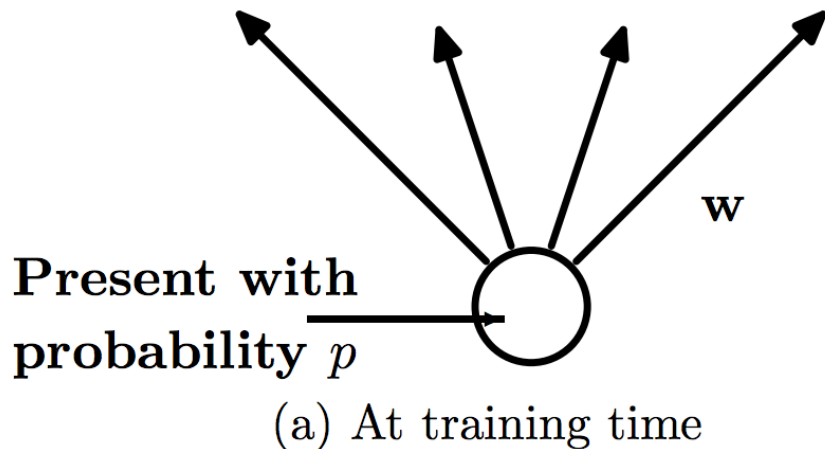
(a) At training time



(b) At test time

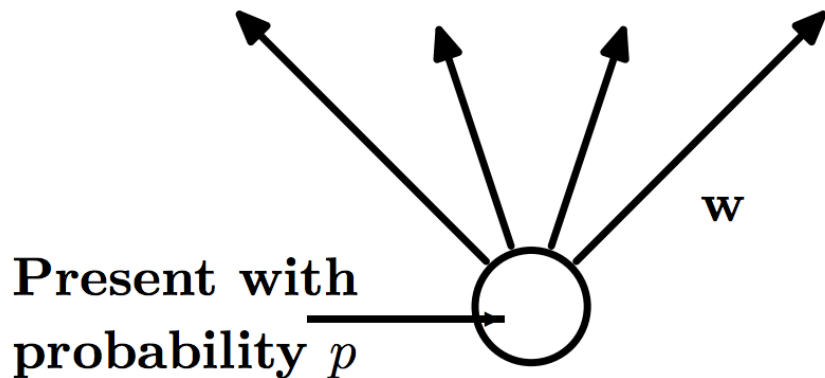
During training, we randomly drop each neuron with probability $1 - p$

KNOCKING OUT AND RESCALING NEURONS

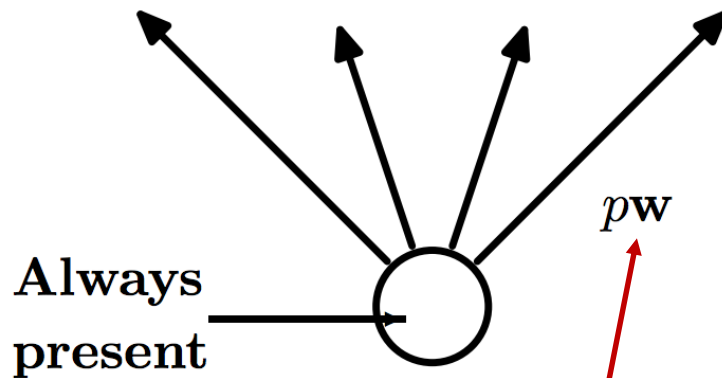


During training, we randomly drop each neuron with probability $1 - p$

KNOCKING OUT AND RESCALING NEURONS



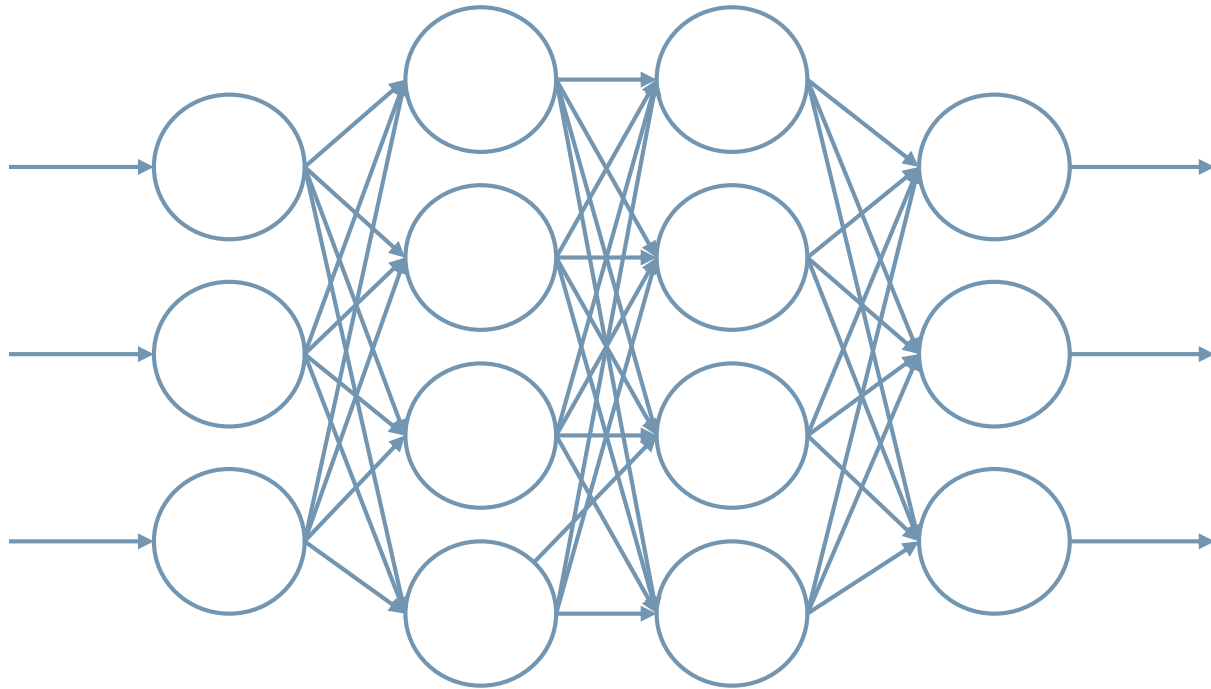
(a) At training time



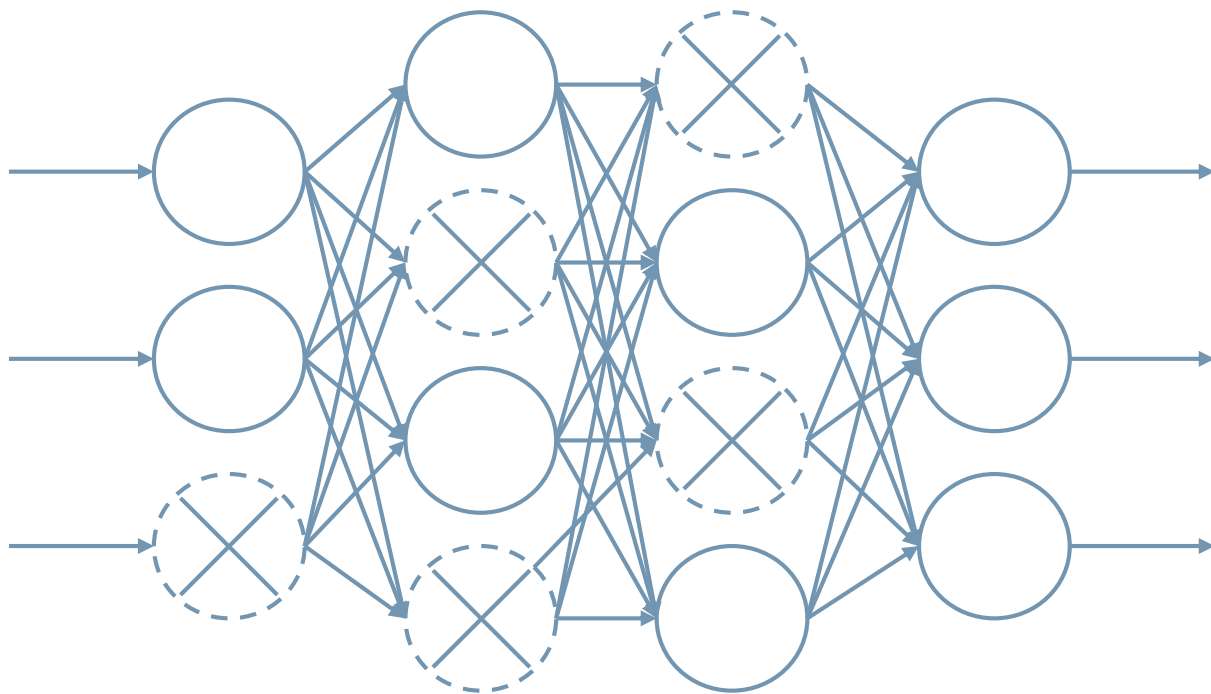
(b) At test time

This ensures that the *expected* value of the weights stays the same at run time

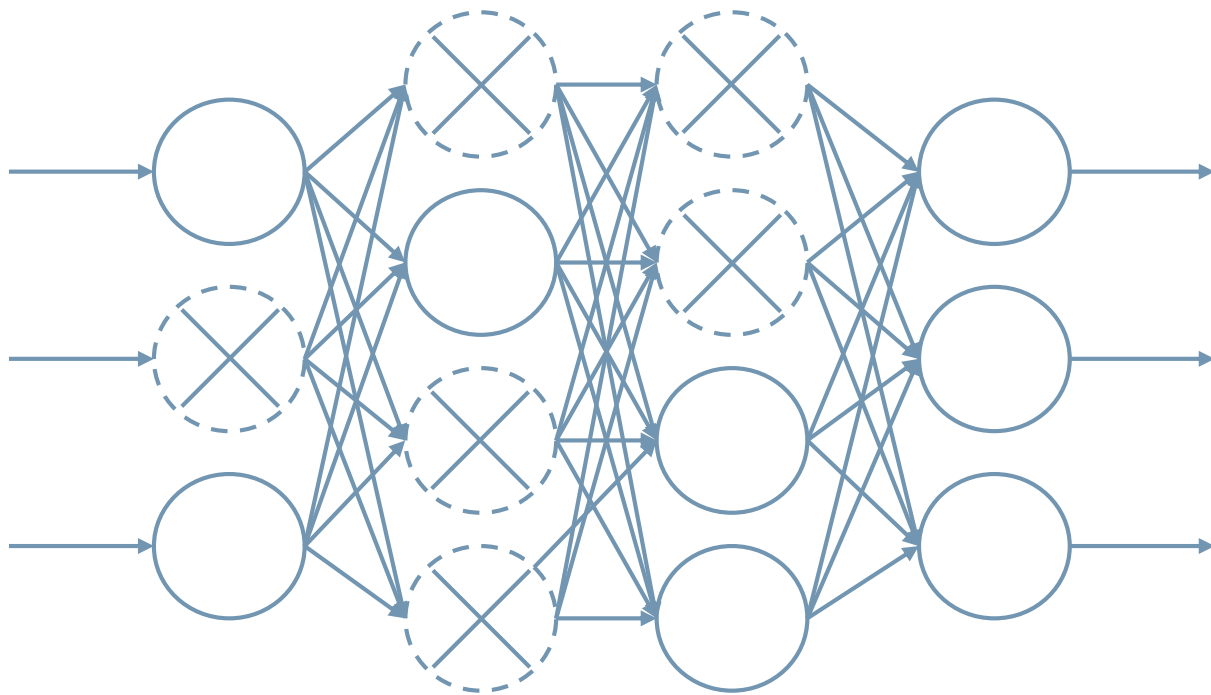
CONCEPT OF A “PSEUDO-ENSEMBLE”



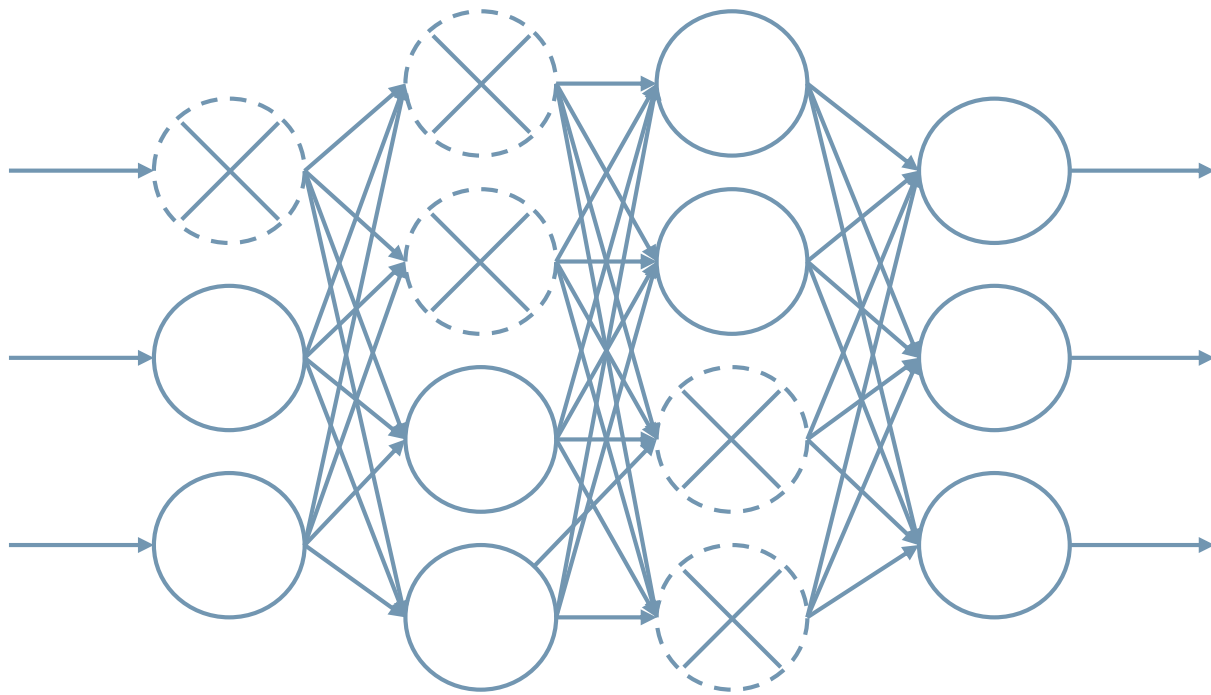
MODEL 1



MODEL 2



MODEL 3 ETC.



DROPOUT: BOTTOM LINE

- Empirically shown to be useful for neural networks
- Relatively easy to manually implement
- Automatically included with TensorFlow!

```
tf.nn.dropout(inputs, keep_prob)
```

Scales weights for you

- Uses keep probability instead of drop probability
- $keep_prob = p$

NEURAL NET VS. MNIST DATASET

MNIST: OUR FIRST “REAL” DATASET

MNIST:

- Labeled hand-written digits from 0-9
- Each digit picture is grayscale
- 28x28 pixels

Game plan:

1. Convert 2D matrices of pixels into vectors
2. Pass into our neural network

