

# Intro to Neural Nets

Session 2: Mathematical Building Blocks &  
Working with Keras API

# Session Agenda

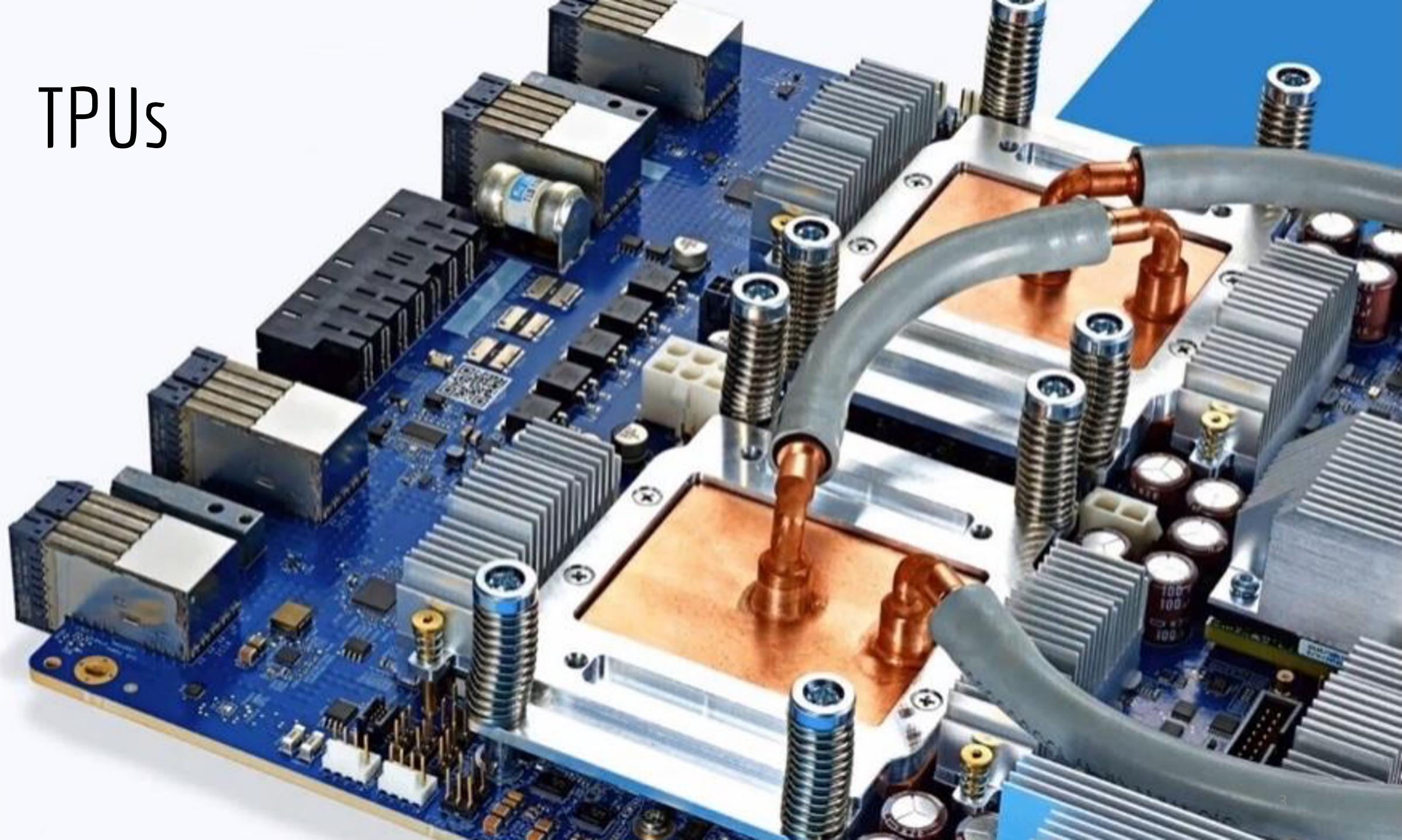
## 1. Building Blocks of NNs

- Tensors (and relevant mathematical operations)
- Loss Functions
- Backpropagation: Derivatives, Gradients & the Chain Rule (quick examples)
- Optimizers

## 2. Building a Linear Classifier

- Overview of Keras and Tensorflow.
- Implementing a linear classifier in Keras (now that we know the components).

# TPUs



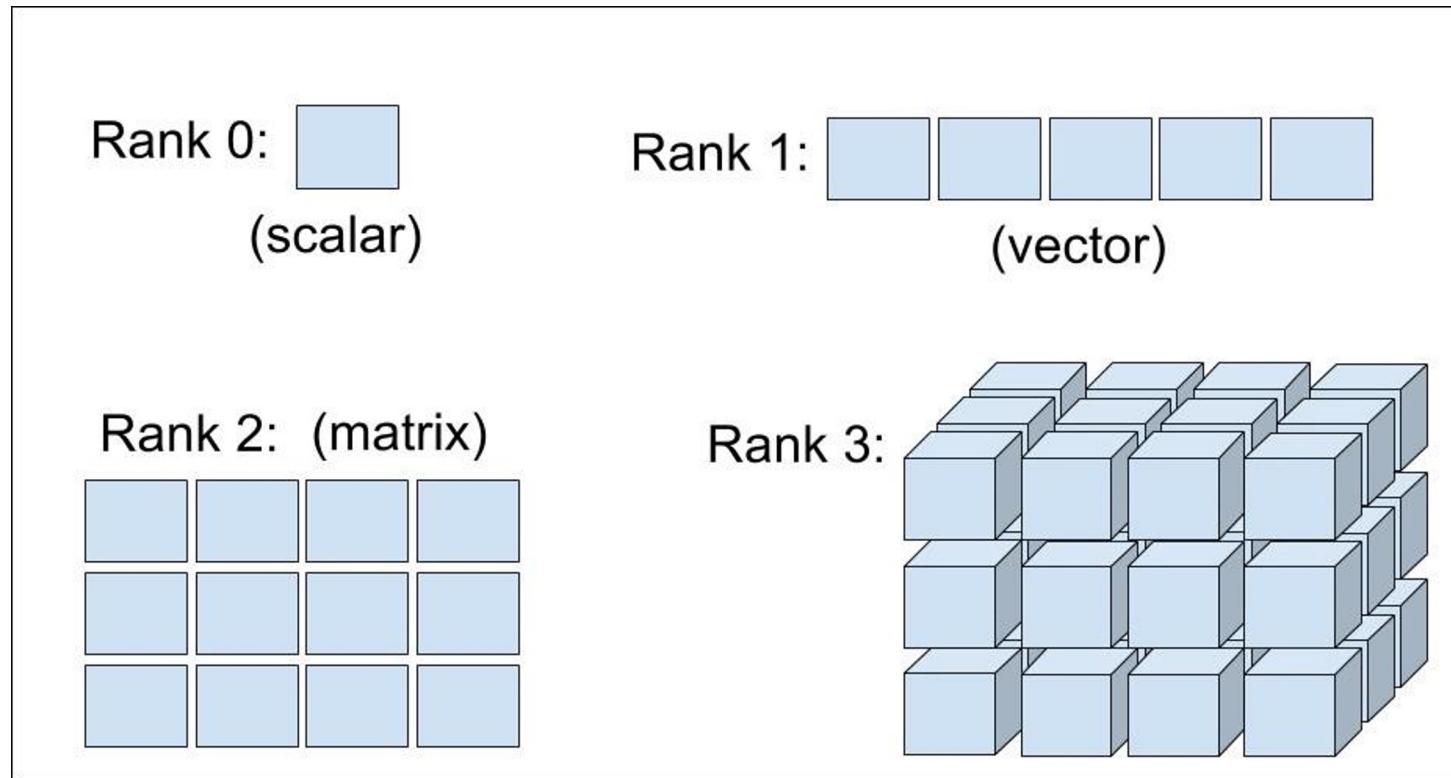
# Abu Dhabi in talks to invest in OpenAI chip venture

State-backed group MGX in discussions over ambitious plans by Sam Altman to build semiconductor business



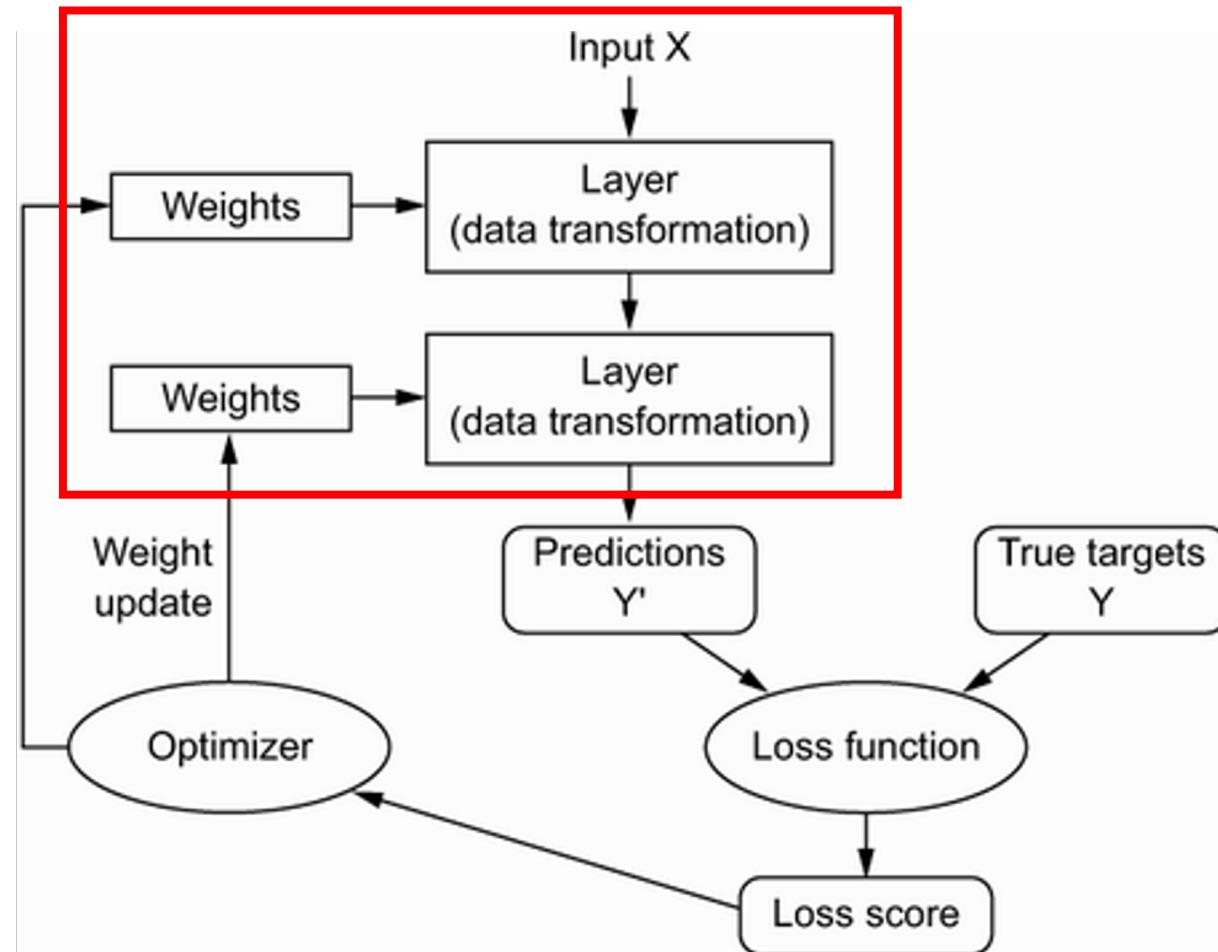
Sheikh Tahnoon bin Zayed al-Nahyan, left, chairs MGX, which is in early stage discussions over a funding deal with Sam Altman's OpenAI © FT montage/UAE Presidential Court via Reuters/AFP via Getty Images

# Tensors

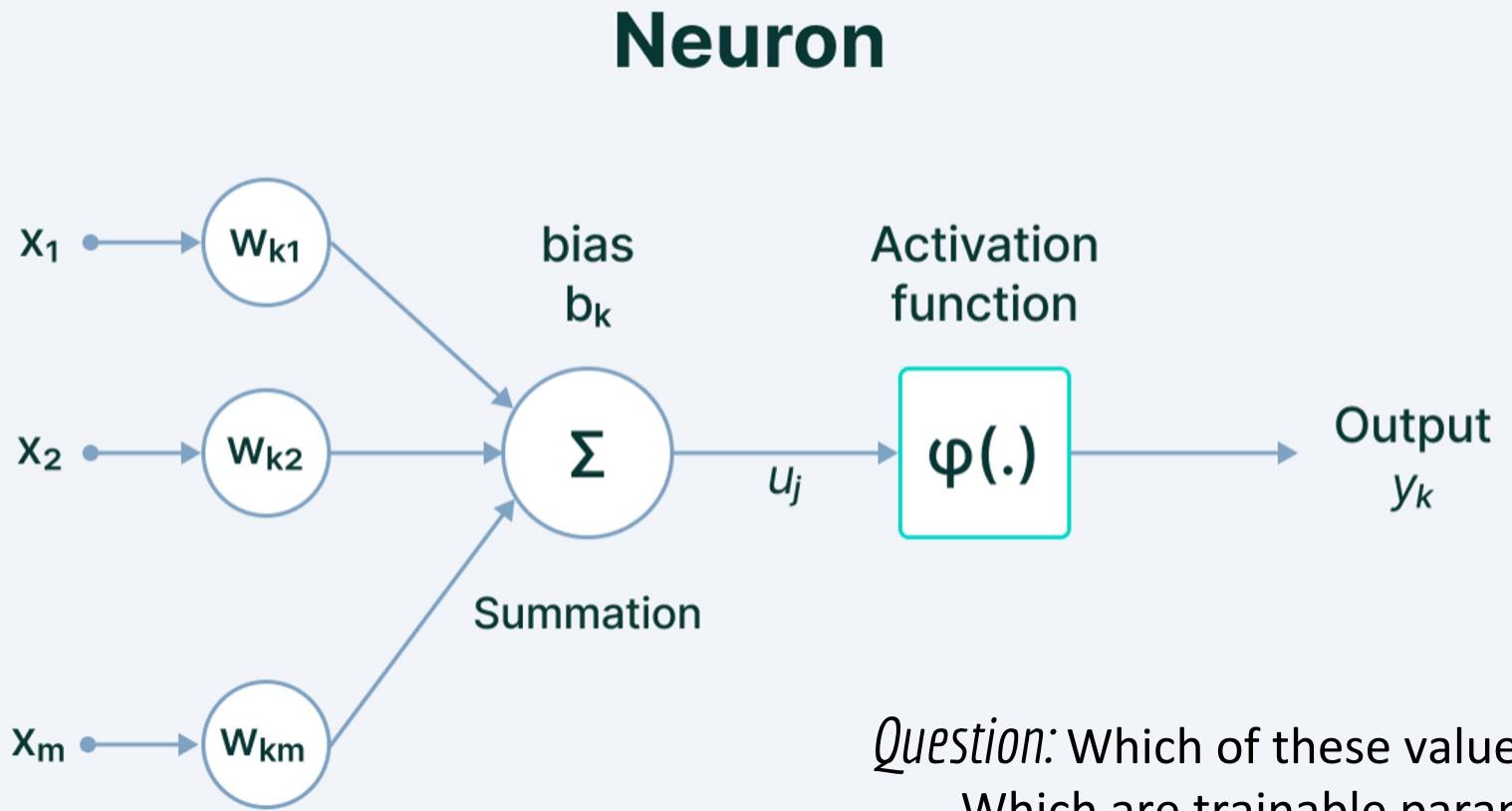


*Question:* What sort of data (give an example) would be stored in a rank-3 tensor? How about a rank-4 tensor?

# Forward Pass

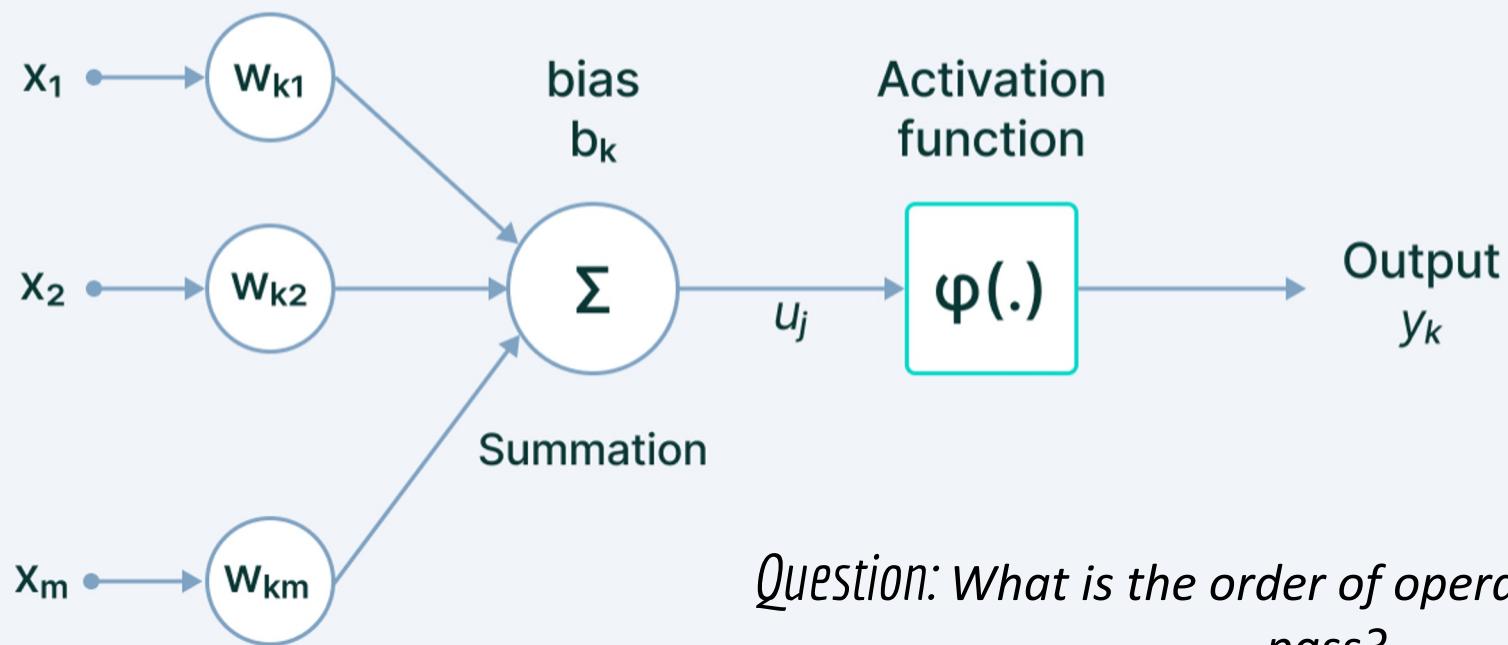


# Neuron / Network Components



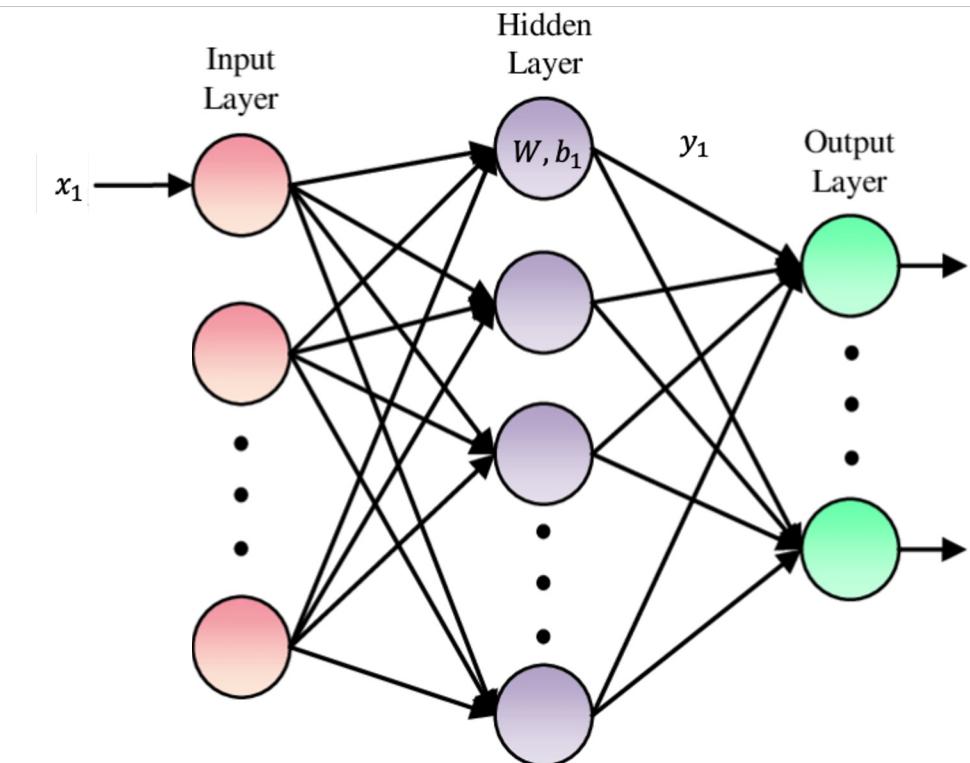
# Neuron / Network Components

## Neuron



# Multiple Neurons Execute in Parallel

- Recall...  $x$  is our flattened digit image, a vector of 784 elements. Matrix  $X$ , which contains our 60,000 images, is thus  $784 \times 60,000$ .
- For each hidden node, we need 784 weights (one weight per input value). We have 512 nodes, so matrix  $W$  is  $784 \times 512$ .
- We have 512 bias terms, one per hidden node, in vector  $B$ .



# Matrix (Tensor) Multiplication

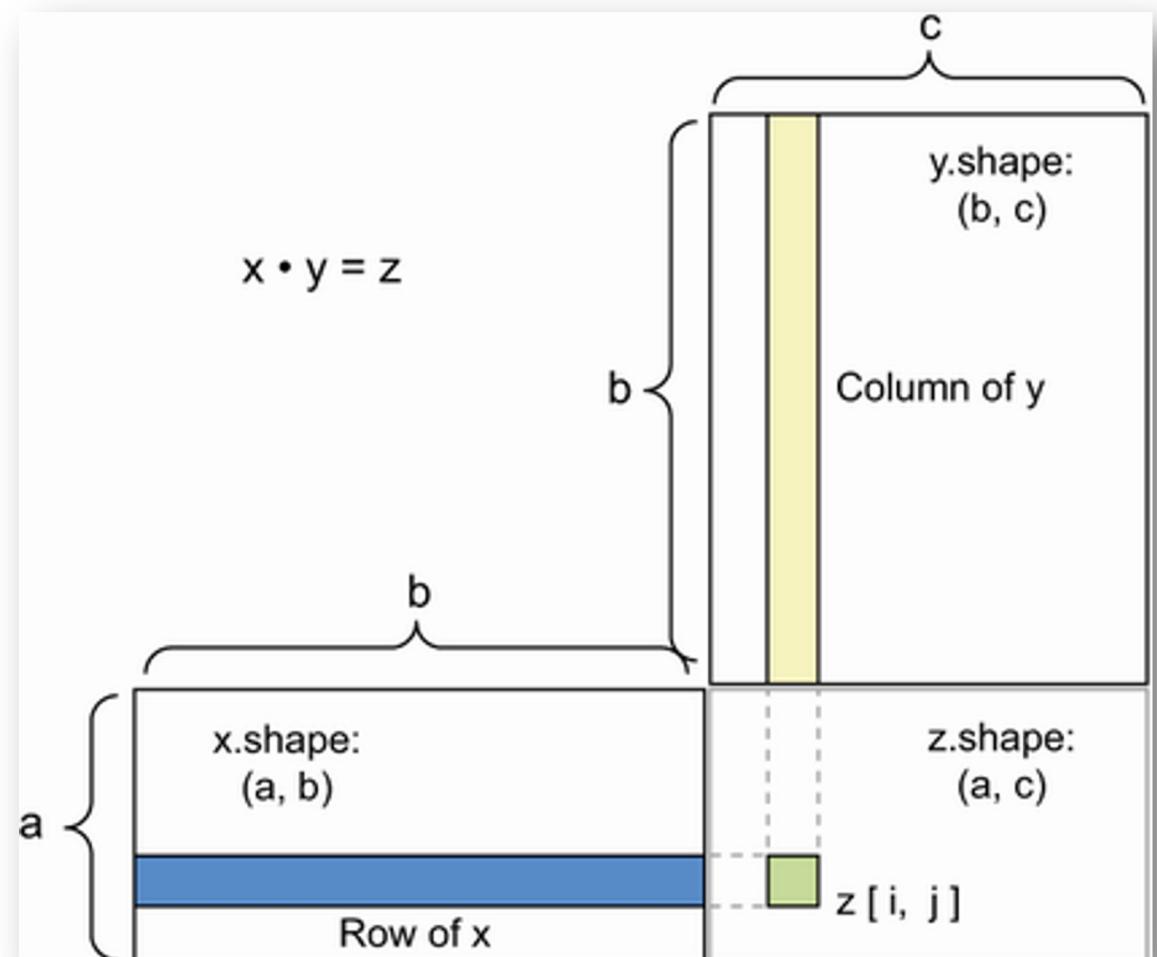
$$y_1 = \varphi (\mathbf{x}_1 \cdot \mathbf{w}_1 + b_1)$$

We calculate all  $\mathbf{x}$ 's \*  $\mathbf{w}$ 's at once via matrix multiplication, i.e.,  $\mathbf{W}^T \cdot \mathbf{X}$

Elements of the Resulting Matrix are the Dot Products of  $\mathbf{X}$ 's Rows and  $\mathbf{Y}$ 's Columns

- $\mathbf{Y}[2,2] = \mathbf{W}^T[2,:] \cdot \mathbf{X}[:,2]$

If you get a cryptic error message about shapes not conforming, it is referring to this.



# Parallel Addition Requires Broadcasting

$$y_1 = \varphi(x_1 \cdot w_1 + b_1)$$

## Shape of the Two Tensors Needs to Conform

- $A + B$  will only work if  $\text{len}(A)$  is cleanly divisible by  $\text{len}(B)$ .

## Broadcasting

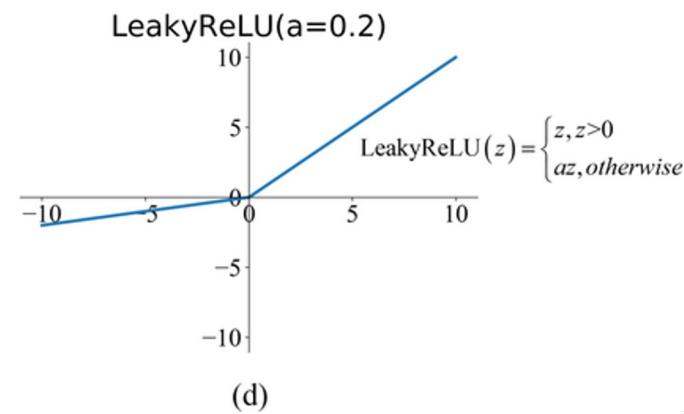
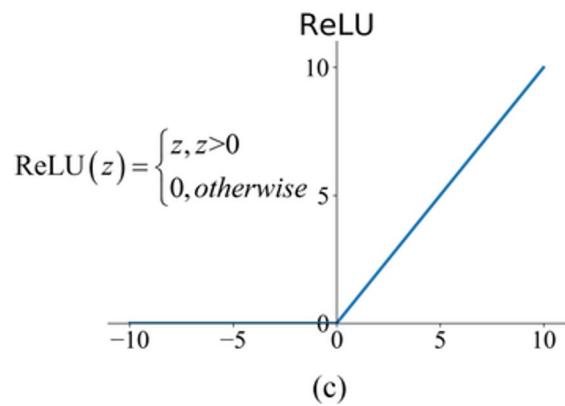
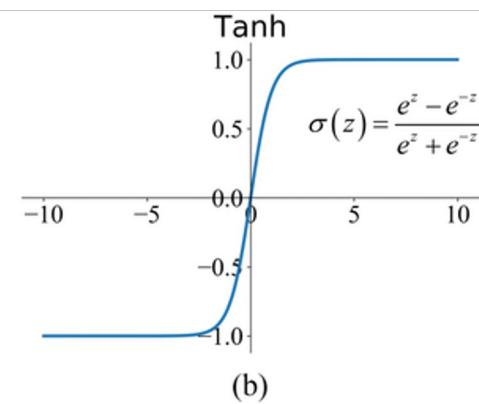
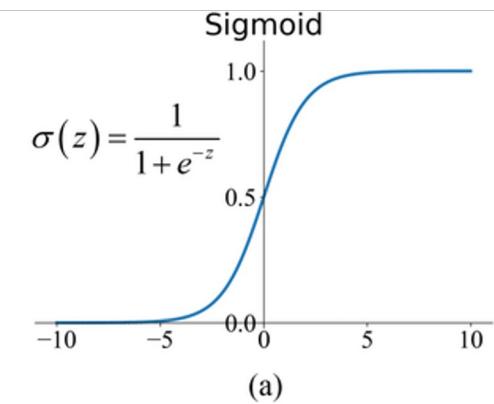
- Duplicates B until it matches A's dimensions, then performs element-wise operation.

**Code might run but broadcast in a different direction than you expect, yielding a wrong answer!**

$$\begin{array}{c} \text{np.arange(3)} + 5 \\ \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline \end{array} \quad + \quad \begin{array}{|c|c|c|} \hline 5 & 5 & 5 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|c|} \hline 5 & 6 & 7 \\ \hline \end{array} \\ \text{np.ones((3, 3))} + \text{np.arange(3)} \\ \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \quad + \quad \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline \end{array} \\ \text{np.arange(3).reshape((3, 1))} + \text{np.arange(3)} \\ \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline \end{array} \quad + \quad \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 1 & 2 & 3 \\ \hline 2 & 3 & 4 \\ \hline \end{array} \end{array}$$

# Activation Applied Element-wise

$$y_1 = \varphi(x_1 \cdot w_1 + b_1)$$



$$y_1 = \varphi(x_1 \cdot w_1 + b_1)$$

# Softmax Activation

## Multi-class, Single Label:

This is a generalization of sigmoid activation. Whereas sigmoid (logit) is for a single binary outcome (e.g., fraud vs. not), softmax (multinomial logit) deals with a single multi-class outcome (e.g., red vs. green vs. blue).

**We only use softmax activation in the output layer.**

```
[ ] predictions = model.predict(test_images)
predictions[1:5]

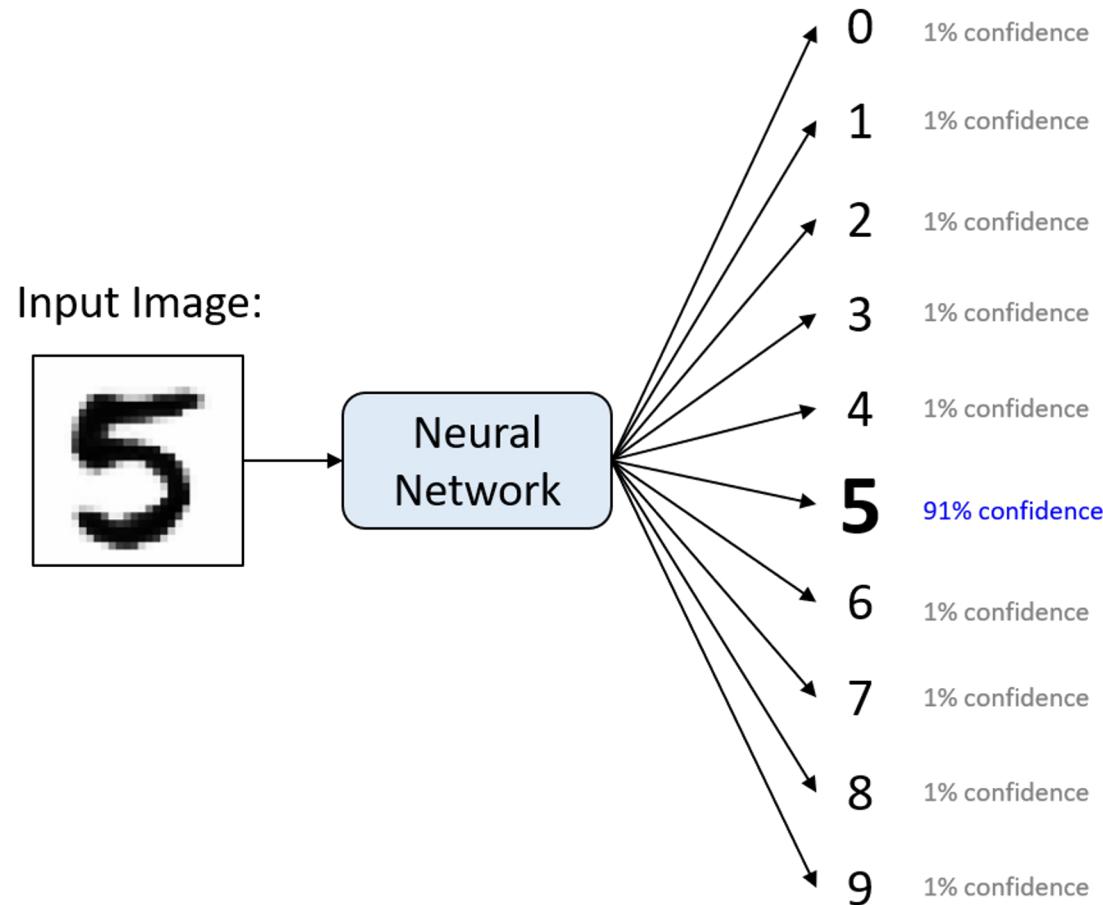
array([[3.6144251e-11, 1.9087817e-07, 9.9999964e-01, 7.4062939e-08,
       1.8678016e-17, 4.8710058e-09, 1.2722286e-11, 7.0804596e-17,
       1.8209119e-09, 1.5724455e-16],
      [1.4256794e-07, 9.9912840e-01, 1.6355875e-04, 1.5004550e-05,
       7.7388759e-06, 3.4624031e-06, 4.7226295e-06, 1.8777854e-04,
       4.8707443e-04, 2.1639225e-06],
      [9.9995363e-01, 8.8815255e-10, 2.9017941e-05, 1.2894120e-07,
       1.0134034e-08, 2.1255494e-06, 1.3782760e-06, 3.7380698e-06,
       5.5121645e-09, 9.9050267e-06],
      [9.0073827e-06, 4.1446402e-09, 8.8214925e-05, 6.3586896e-07,
       9.4997090e-01, 6.3517414e-06, 2.1397573e-06, 4.1880834e-04,
       4.4119348e-05, 4.9459830e-02]], dtype=float32)
```

$$y_1 = \varphi(x_1 \cdot w_1 + b_1)$$

# Multi-Class, Single-Label

## Multi-class, Single Label:

As with sigmoid, we can loosely interpret the values output by softmax activation as probabilities.

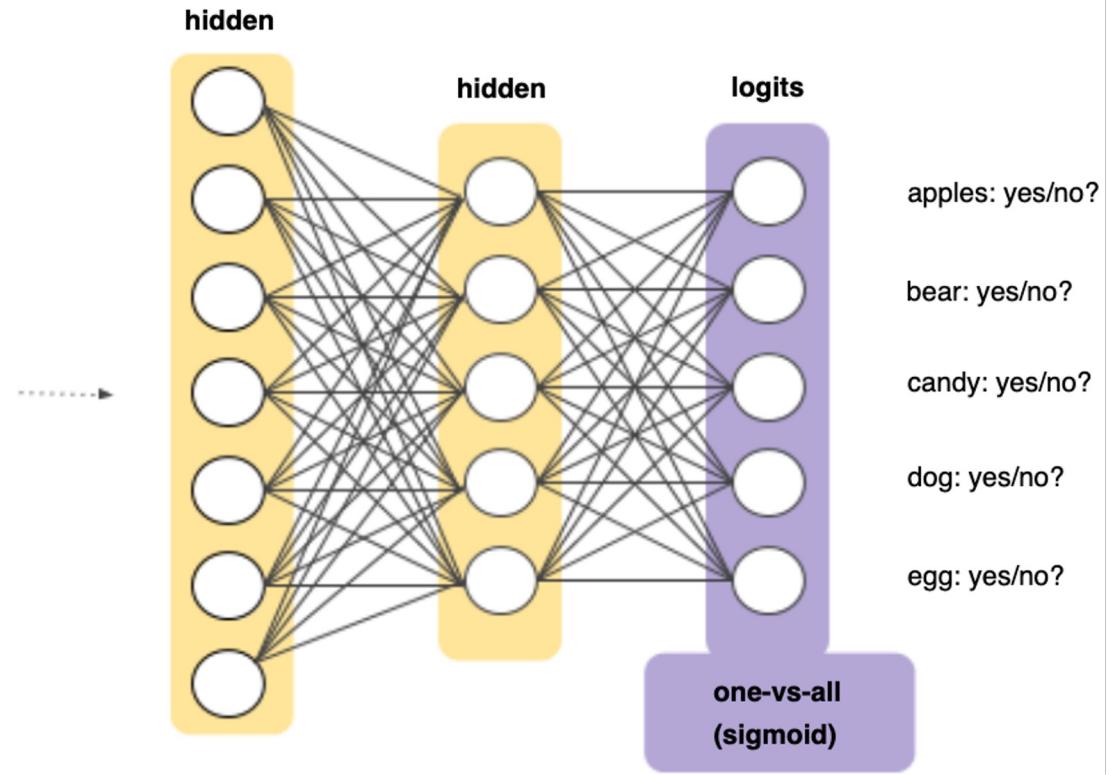


# Aside: Multi-Class, Multi-Label

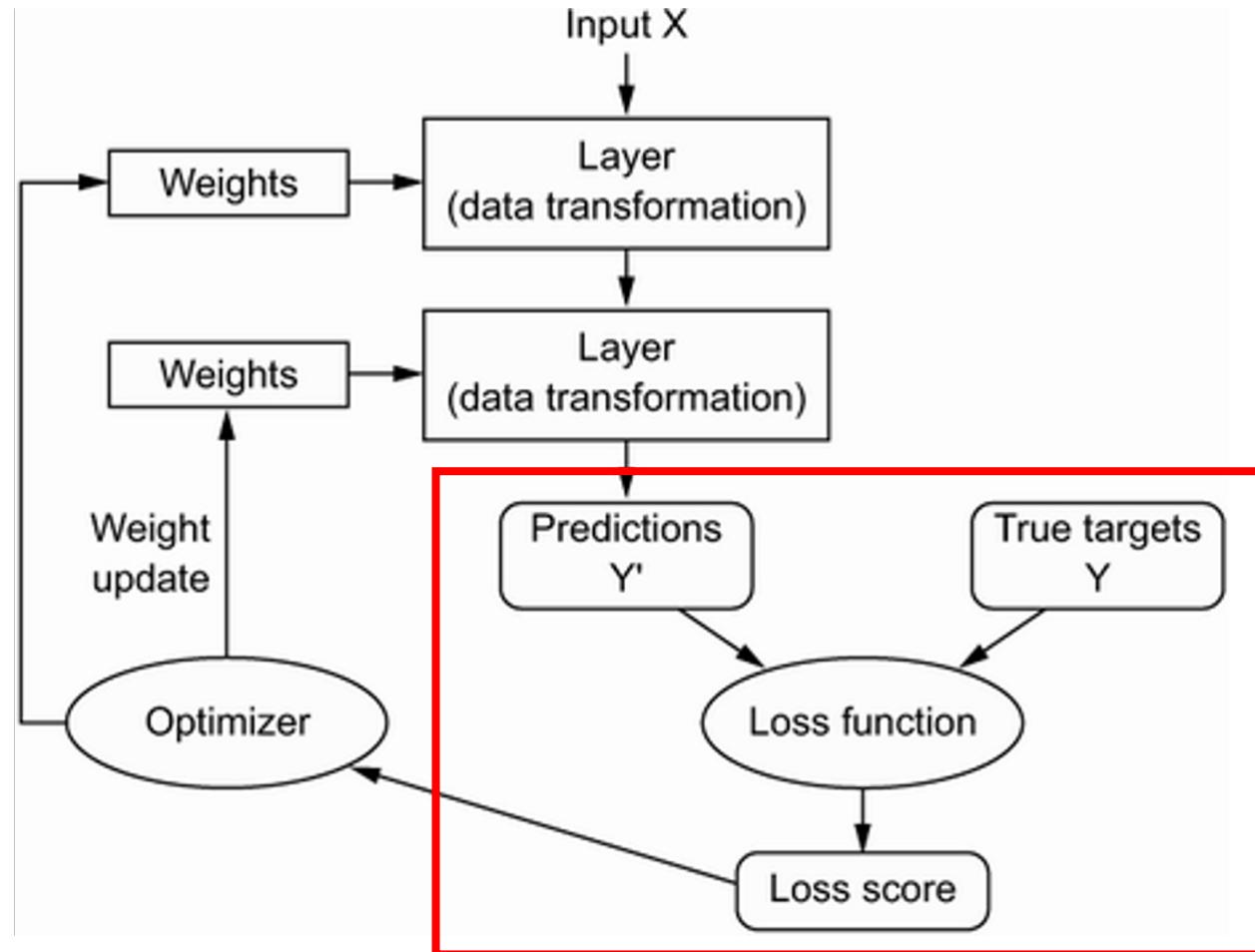
## Many Binary Choices vs. One Multi-category Choice

- We might also have an output layer comprised of many binary labels in some cases (e.g., a cat, a guitar, and a tree may be present in a photo all at the same time).

**Many sigmoids versus one softmax depends on whether labels are mutually exclusive or not.**



# Calculate Loss



# Loss Functions

## Cross-Entropy / Log-Loss

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

- Typical for binary outcomes.  
Value grows exponentially larger as the predicted probability moves away from the true 0,1 label.
- Multi-category outcomes have an analogous loss function known as categorical cross-entropy.

$$CE = - \sum_i^C t_i \log(s_i)$$

## MAE / L1 Loss

$$MAE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{n}$$

- Typical for continuous outcomes.  
Errors are penalized homogenously, in magnitude and direction. This should look familiar!

## MSE / Quadratic / L2 Loss

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

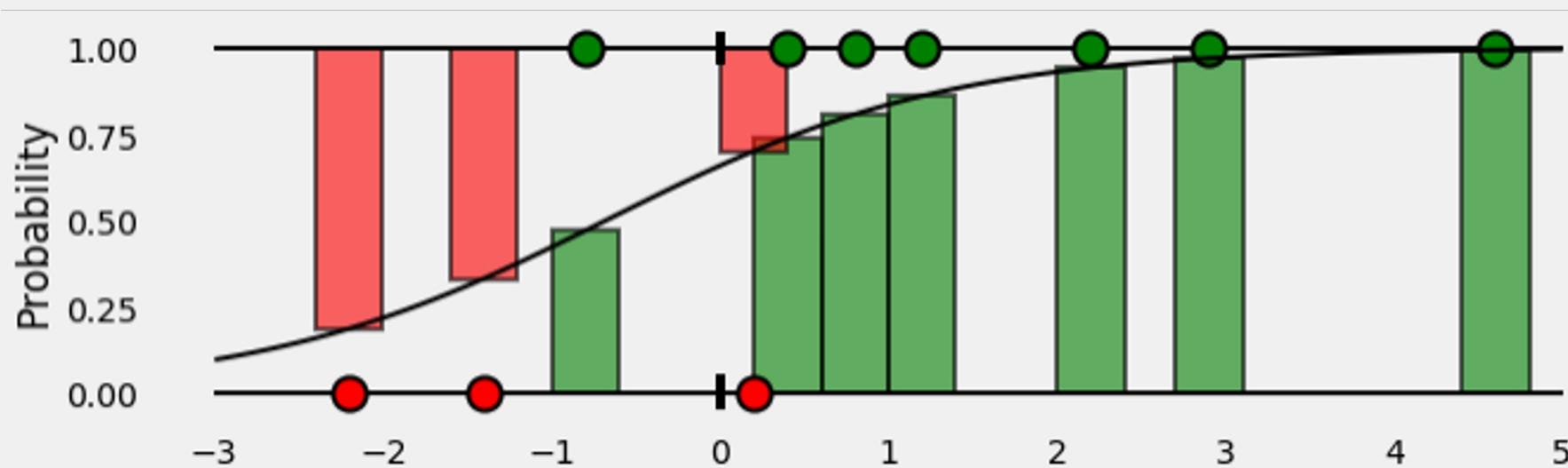
- Typical for continuous outcomes, larger errors penalized exponentially more. This should look familiar!

# Binary Cross-Entropy Loss

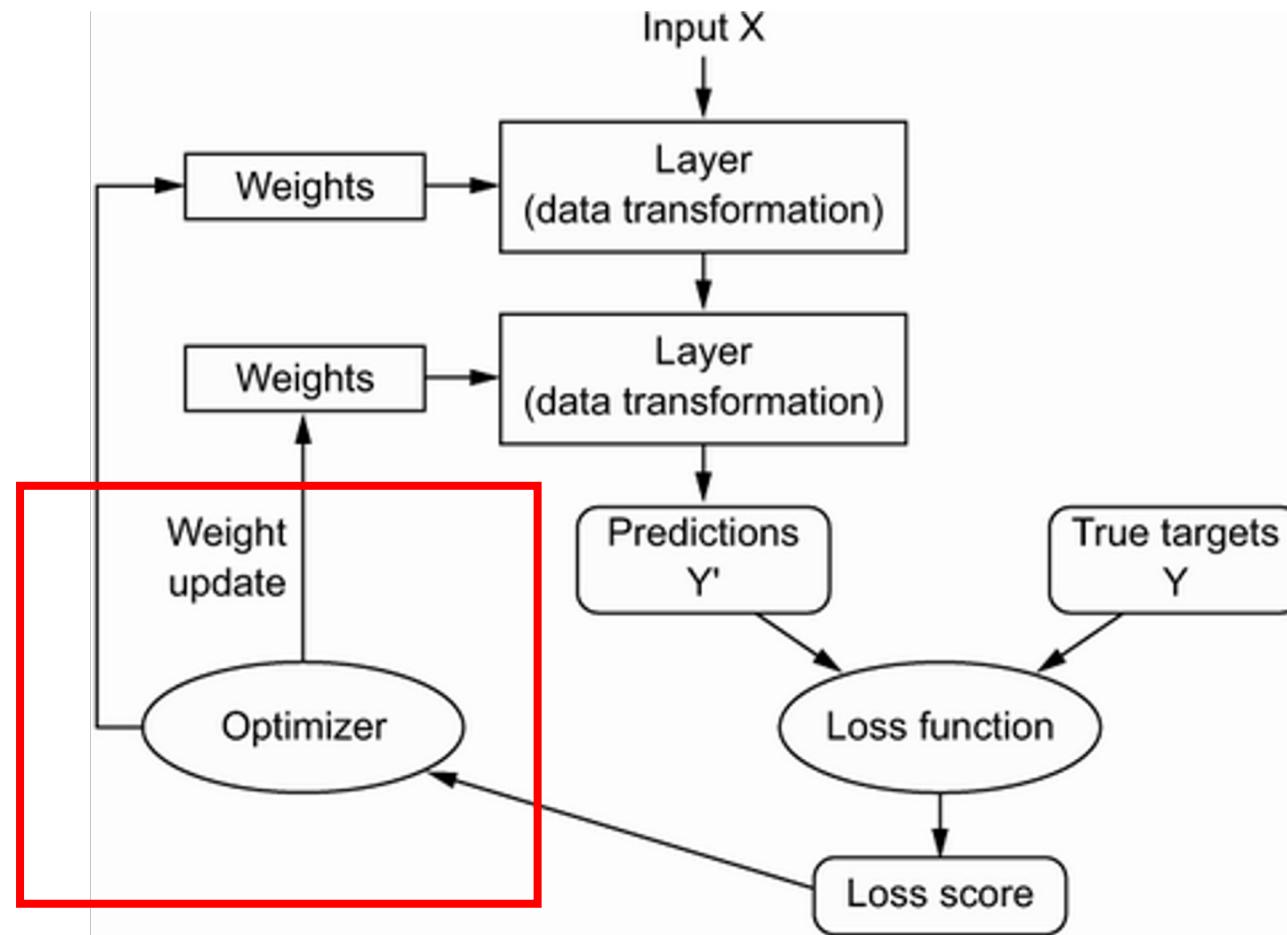
$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

## Piecemeal Function:

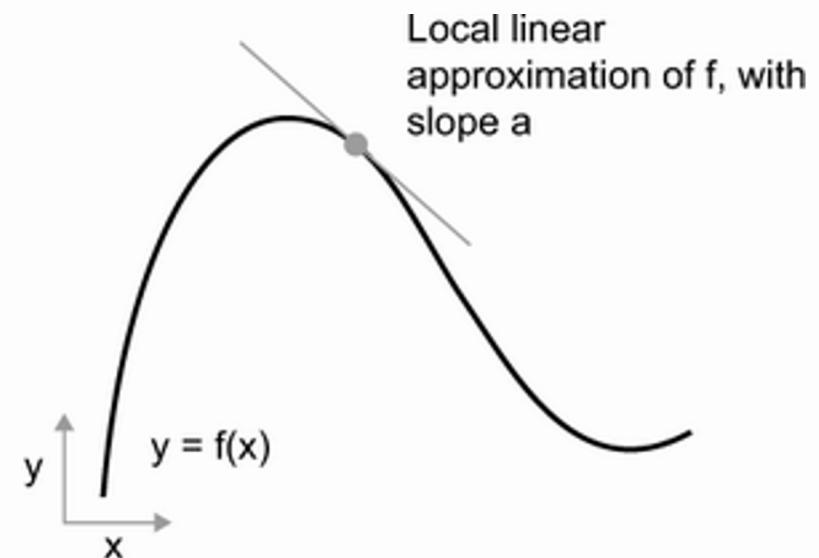
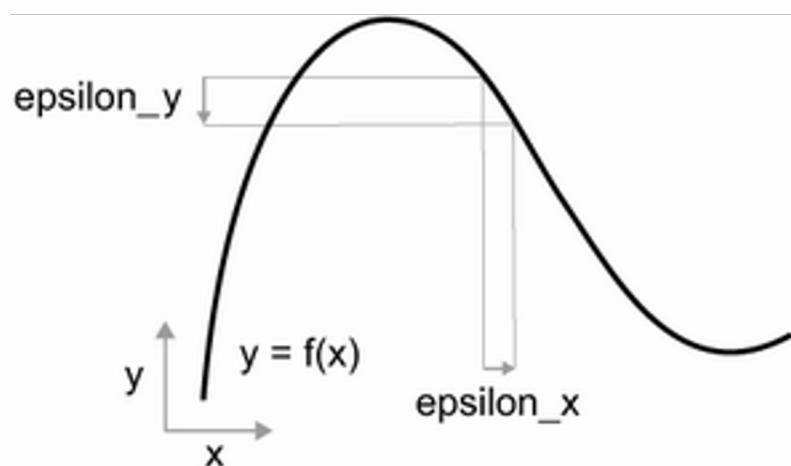
- If ground truth is 1, then loss is  $-1 * \log(p)$ . As prediction approaches 1, loss approaches 0. As prediction approaches 0, loss grows exponentially.
- If ground truth is 0, then loss is  $-1 * \log(1-p)$ . As prediction approaches 1, loss rises exponentially. As prediction approaches 0, loss approaches 0.



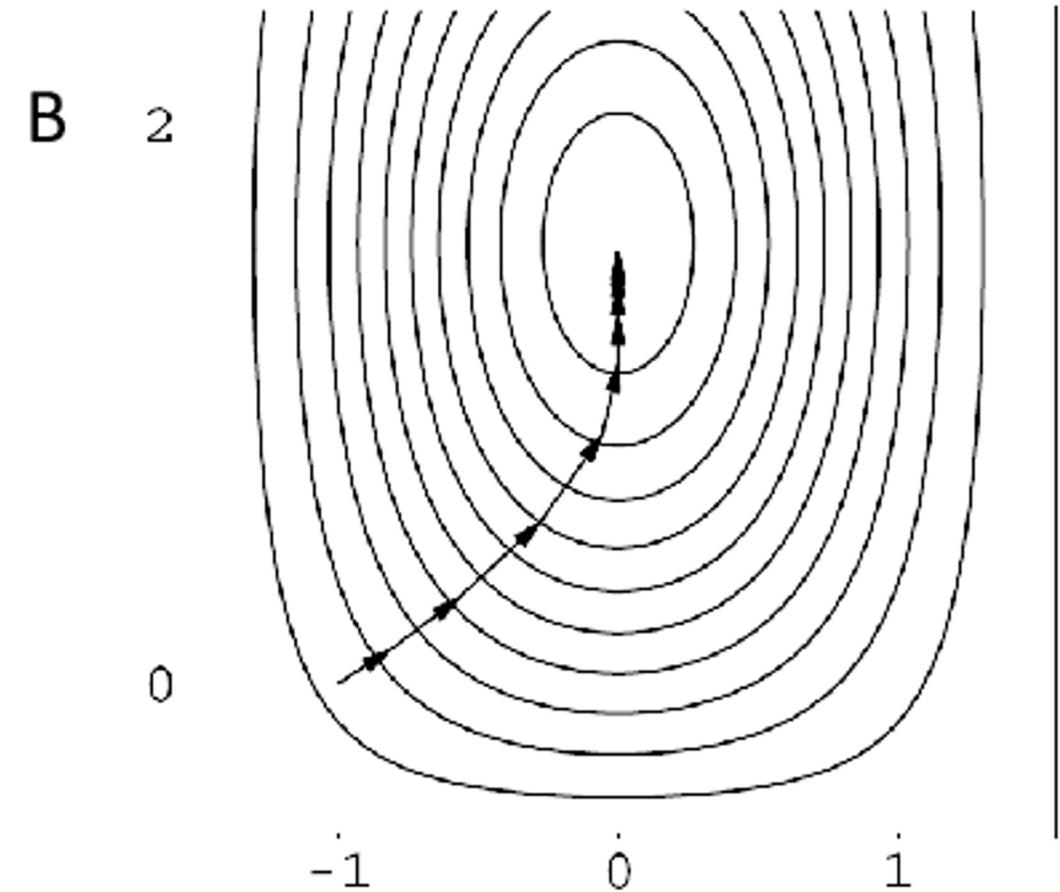
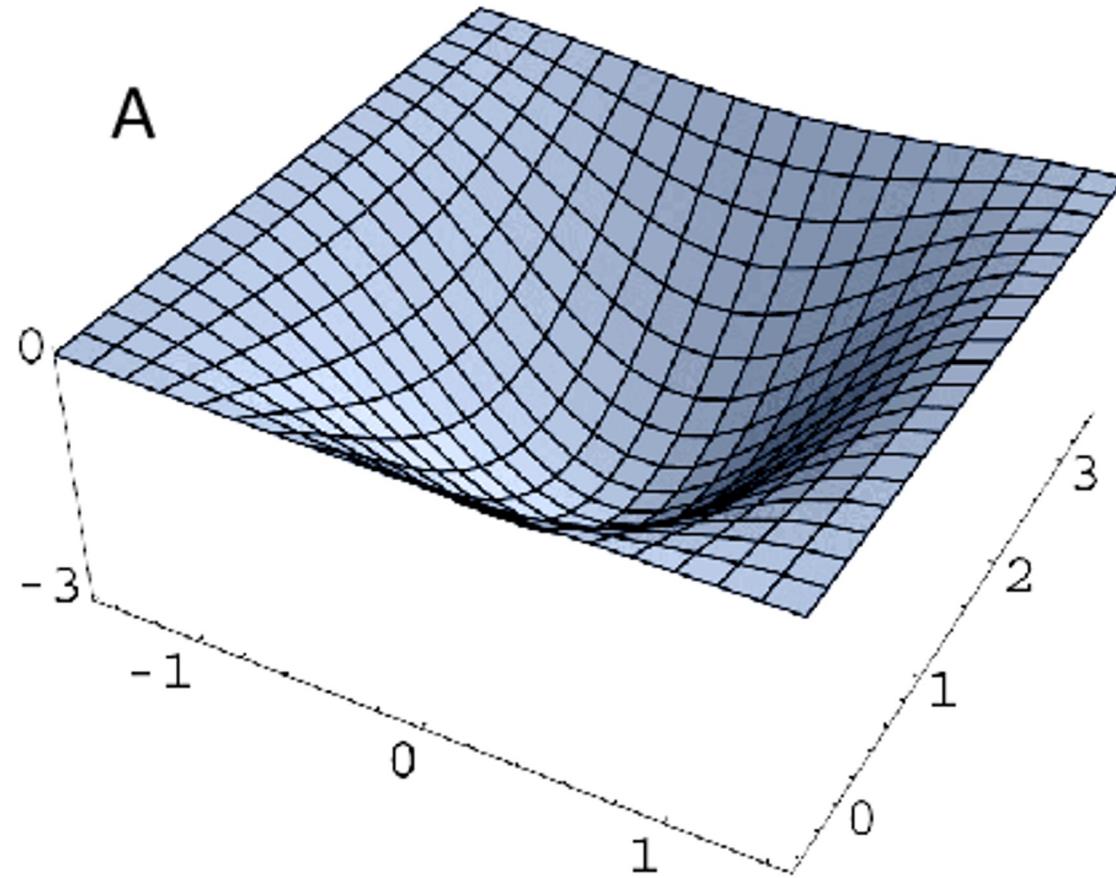
# Backpropagation



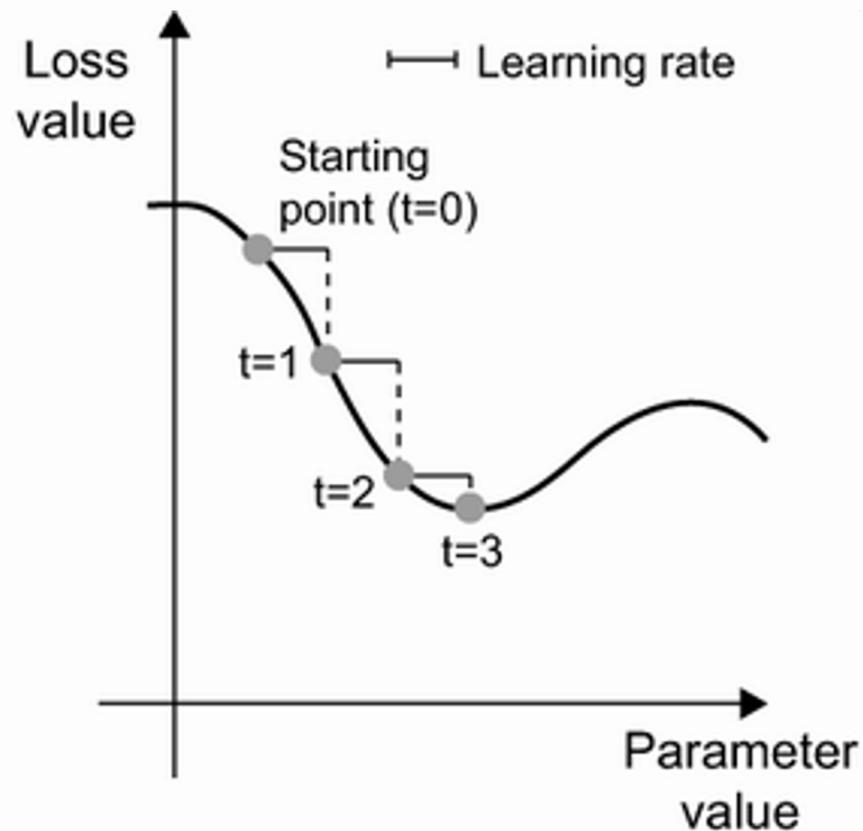
# Derivative = Rate of Change



# Gradient = Derivative in Multiple Dimensions



# Gradient Descent



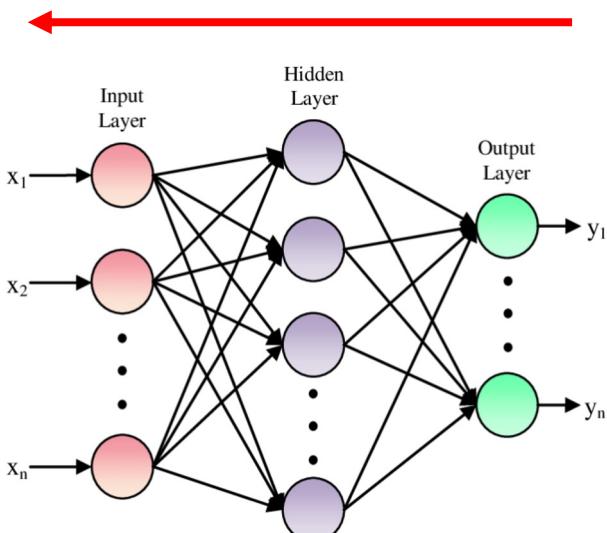
# Derivatives of LOSS w.r.t ALL Model Parameters

Each Node's Output Can be Expressed as a Function of all Prior Nodes' Outputs

$$y_1 = \varphi(x_1 \cdot w_{1,1} + x_2 \cdot w_{1,2} + \dots + b_1)$$

$$y_2 = \varphi(x_1 \cdot w_{2,1} + x_2 \cdot w_{2,2} + \dots + b_2)$$

...

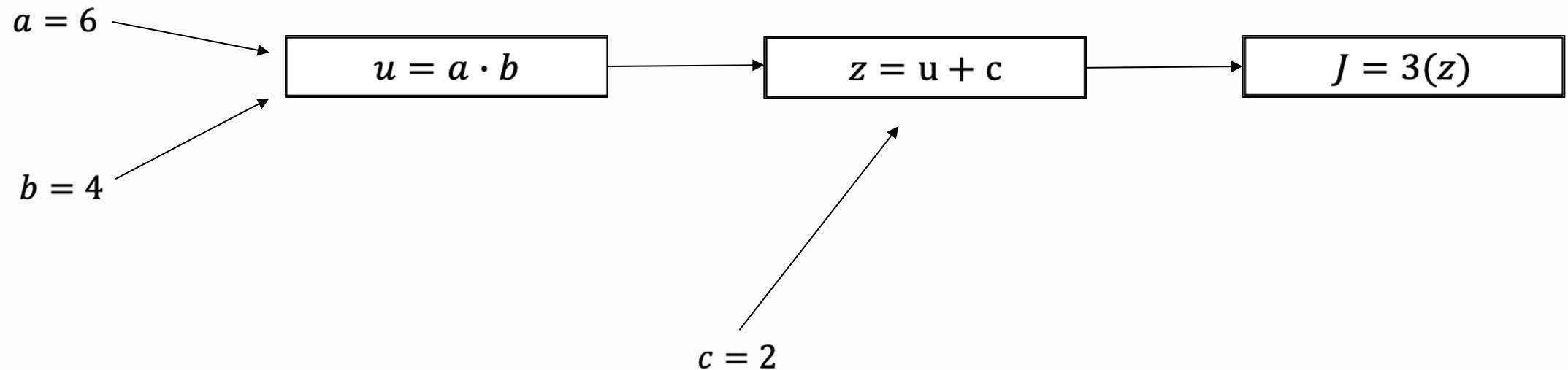


Start at the final nodes in the network and work backwards

- We calculate partial derivatives that explain how Loss will change if we modify a particular weight.
- We use that understanding to decide how we modify each weight for the next iteration.
- So, how can we figure out the derivative of Loss w.r.t. a particular weight in the neural network? **Backpropagation!**

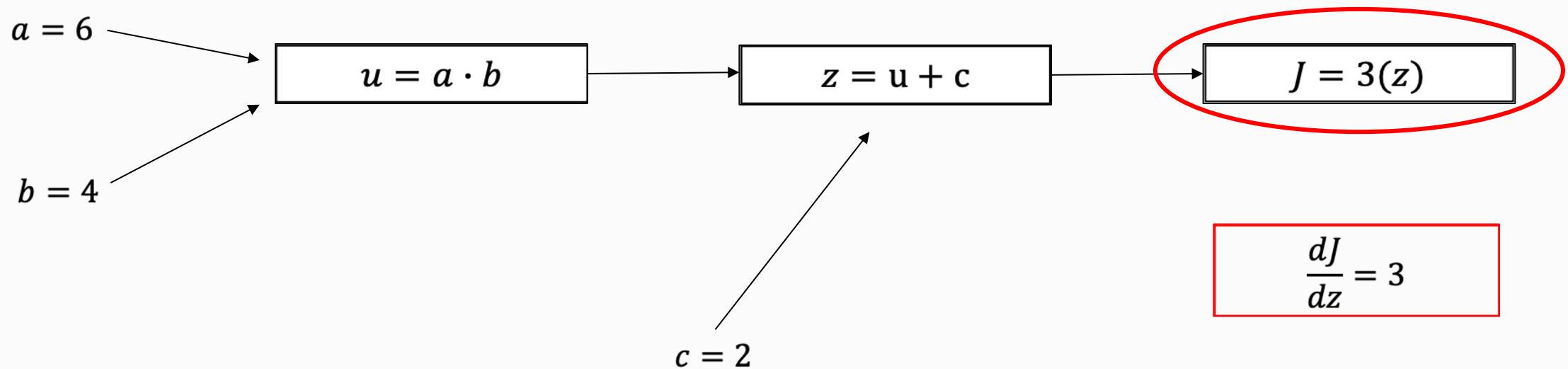
# Simplifying Gradients: Computation Graph

$$J = 3(a \cdot b + c)$$



# Backpropagation = Working Backwards

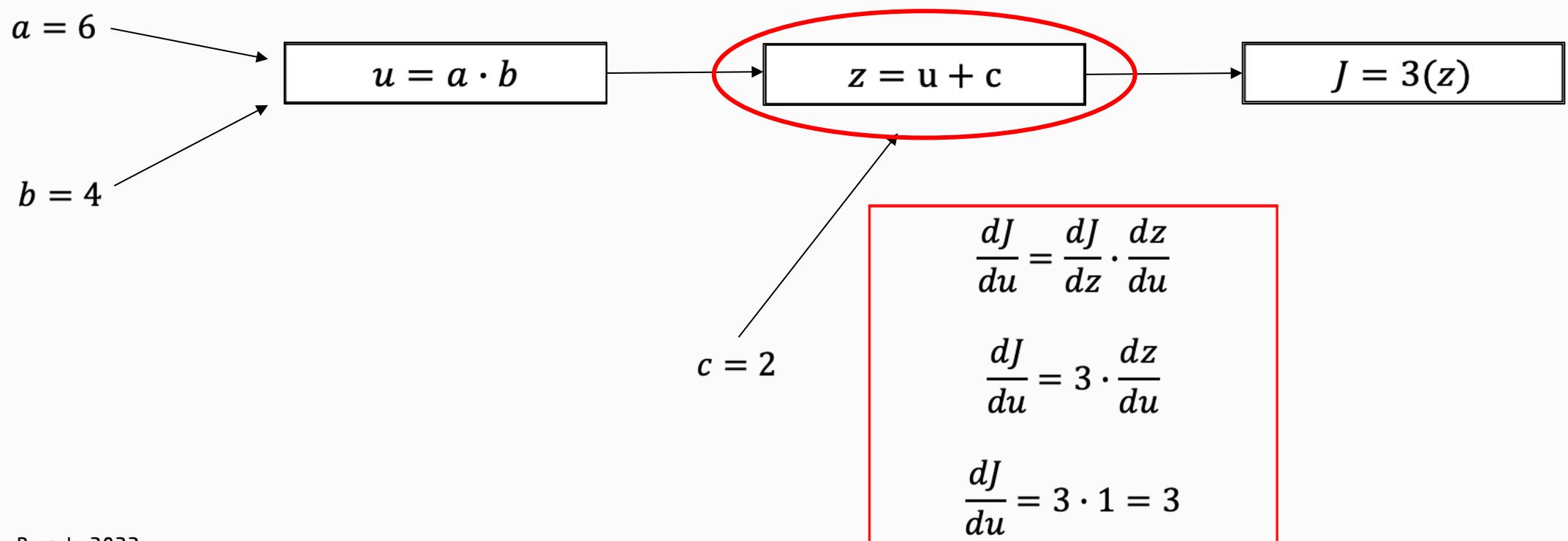
$$J = 3(a \cdot b + c)$$



# Backpropagation = Work Backwards

$$\frac{dJ}{dz} = 3$$

$$J = 3(a \cdot b + c)$$

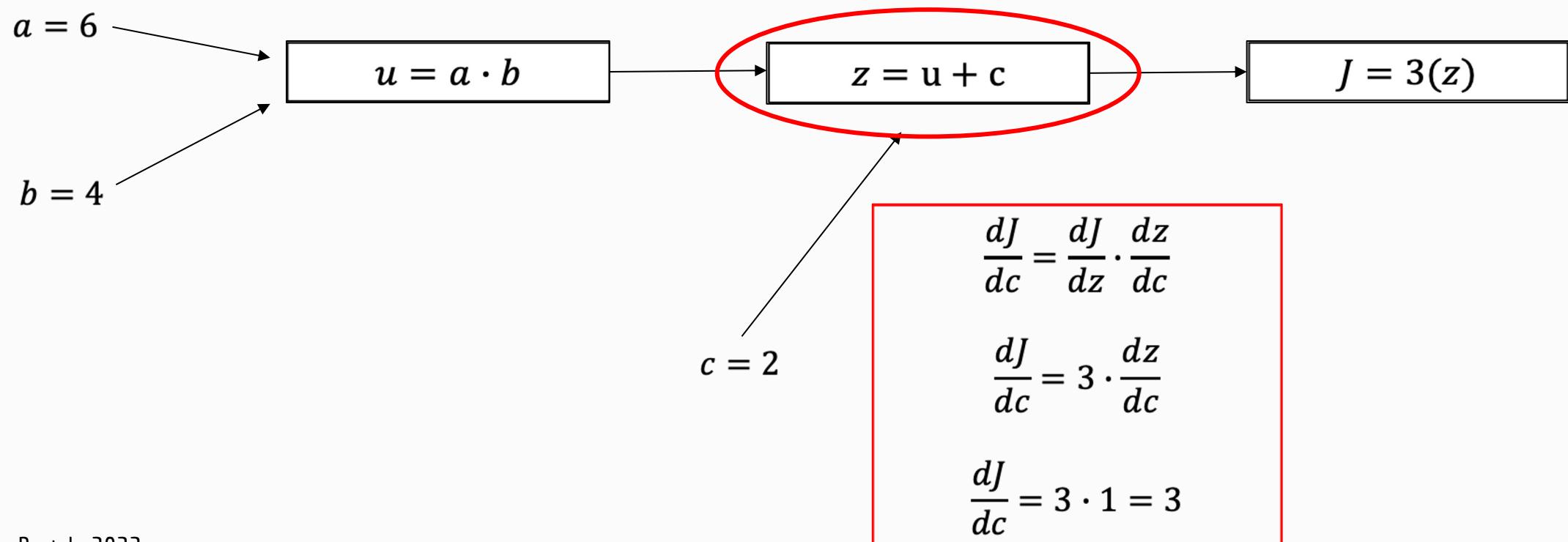


# Backpropagation = Work Backwards

$$\frac{dJ}{dz} = 3$$

$$J = 3(a \cdot b + c)$$

$$\frac{dJ}{du} = 3$$



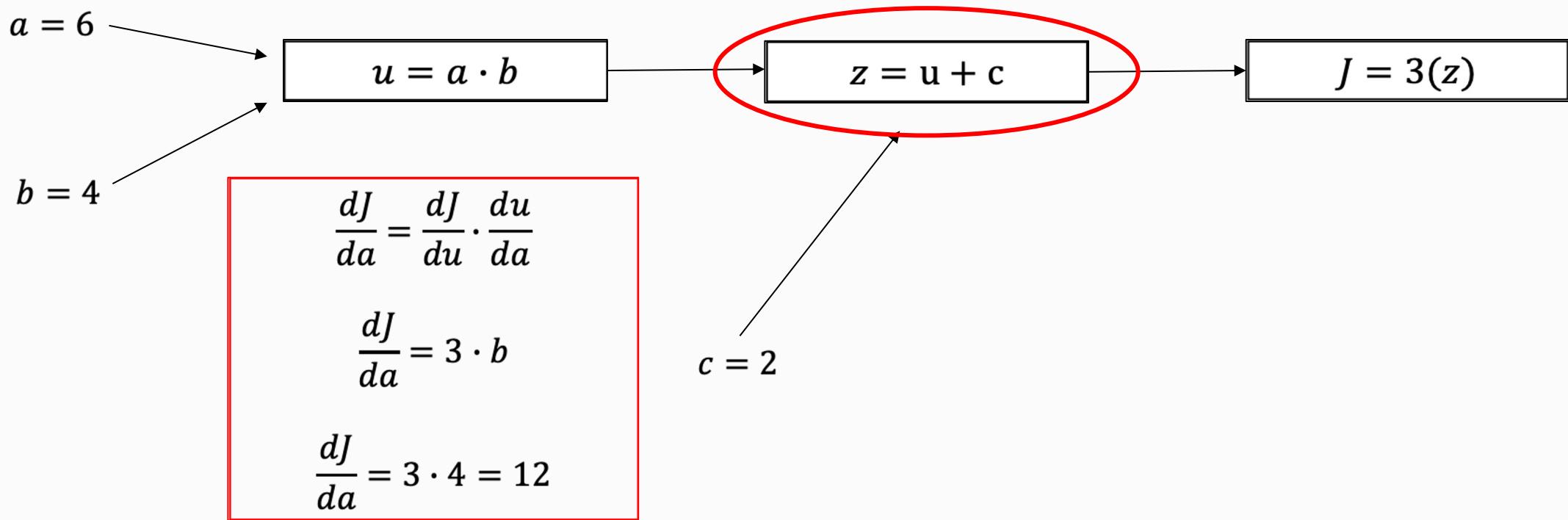
# Backpropagation = Work Backwards

$$\frac{dJ}{dz} = 3$$

$$J = 3(a \cdot b + c)$$

$$\frac{dJ}{du} = 3$$

$$\frac{dJ}{dc} = 3$$



# Backpropagation = Work Backwards

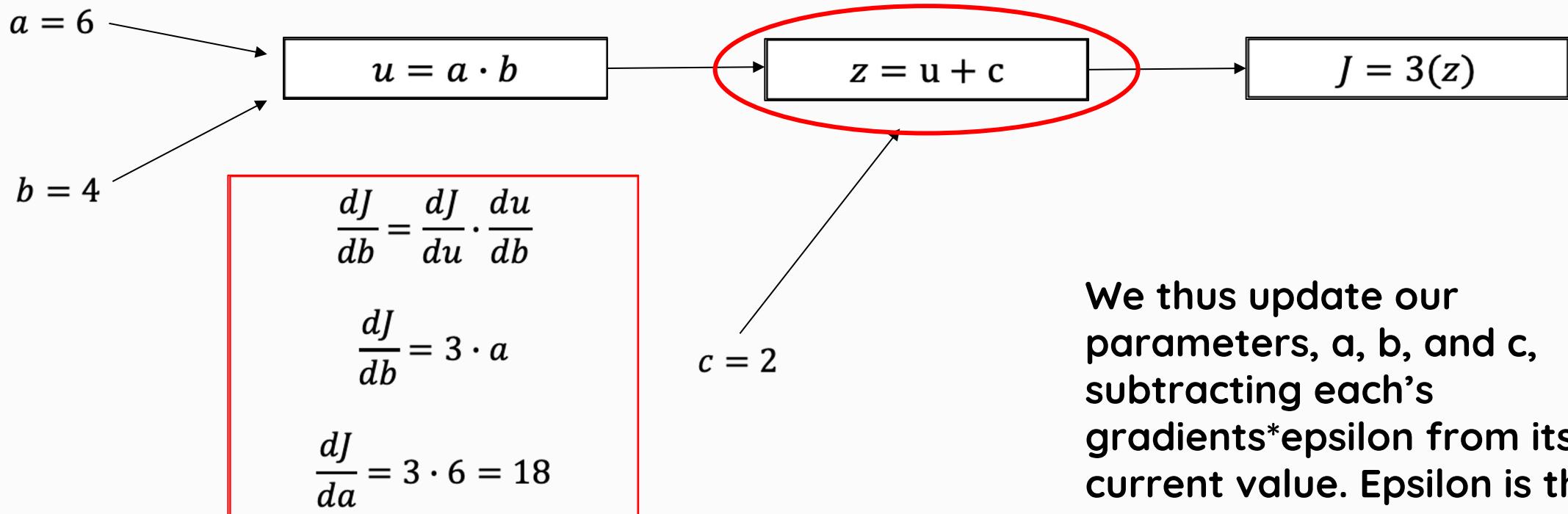
$$\frac{dJ}{dz} = 3$$

$$J = 3(a \cdot b + c)$$

$$\frac{dJ}{da} = 12$$

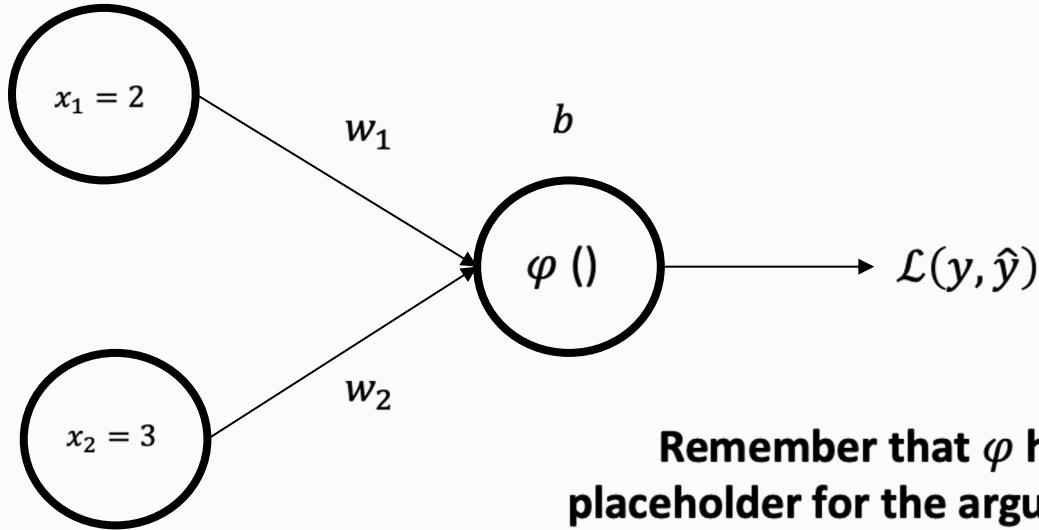
$$\frac{dJ}{du} = 3$$

$$\frac{dJ}{dc} = 3$$



We thus update our parameters,  $a$ ,  $b$ , and  $c$ , subtracting each's gradients\*epsilon from its current value. Epsilon is the learning rate.

# Single Node with Sigmoid & Cross-Entropy Loss (i.e., Logistic Regression)



**Remember that  $\varphi$  here is just a placeholder for the argument to the loss function. It happens to be a sigmoid transformation of ‘something’, i.e.,  $\varphi(wx+b)$ , but it doesn’t really matter. We just represent it with some variable name and calculate an expression for the derivative.**

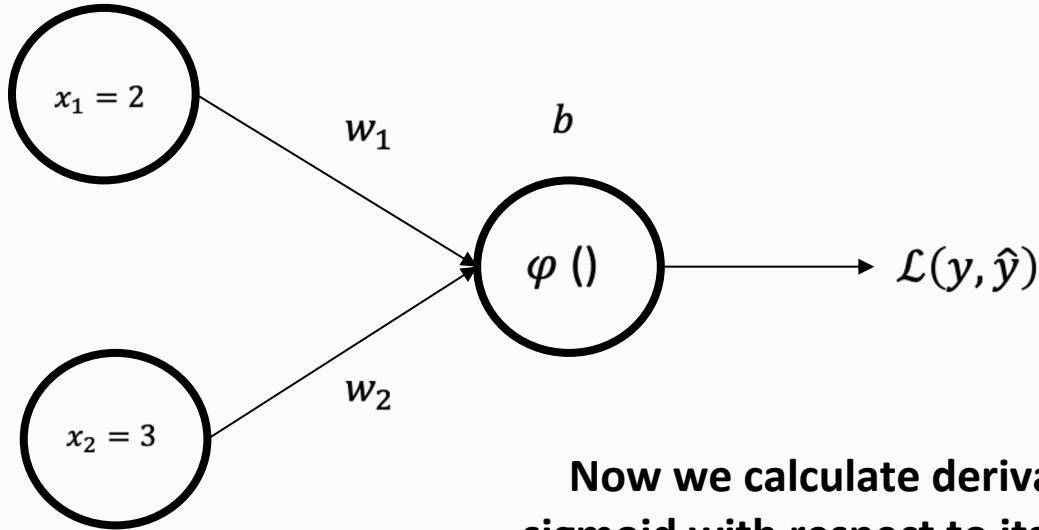
$$\frac{d\mathcal{L}}{d\varphi} = -\frac{y}{\varphi} + \frac{1-y}{1-\varphi}$$

$$\frac{d\mathcal{L}}{d\varphi} = \frac{\varphi(1-y) - y(1-\varphi)}{\varphi(1-\varphi)}$$

$$\frac{d\mathcal{L}}{d\varphi} = \frac{\varphi - \varphi y - y + \varphi y}{\varphi(1-\varphi)}$$

$$\frac{d\mathcal{L}}{d\varphi} = \frac{\varphi - y}{\varphi(1-\varphi)}$$

# Single Node with Sigmoid & Cross-Entropy Loss (i.e., Logistic Regression)



Now we calculate derivative of the sigmoid with respect to its argument,  $z$ .

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

$$\varphi'(z) = -1 \cdot (1 + e^{-z})^{-2} \cdot (0 + e^{-z} \cdot -1)$$

$$\varphi'(z) = (1 + e^{-z})^{-2} \cdot e^{-z}$$

$$\varphi'(z) = \varphi(z) \cdot (1 - \varphi(z))$$

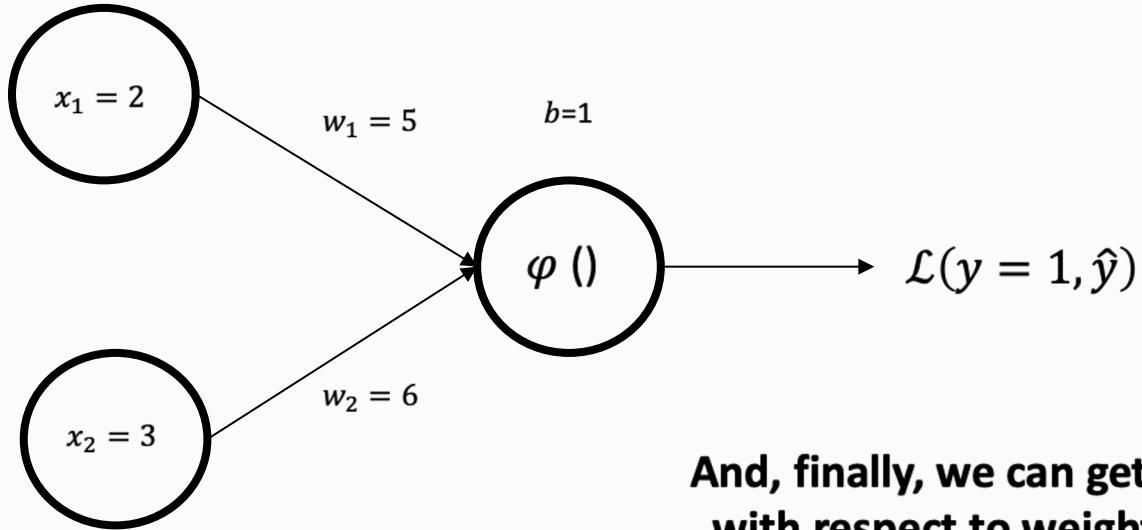
$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{d\varphi} \cdot \frac{d\varphi}{dz}$$

$$\frac{d\mathcal{L}}{dz} = \frac{\varphi - y}{\varphi(1 - \varphi)} \cdot \frac{d\varphi}{dz}$$

$$\frac{d\mathcal{L}}{dz} = \frac{\varphi - y}{\varphi(1 - \varphi)} \cdot \varphi(1 - \varphi)$$

$$\frac{d\mathcal{L}}{dz} = \varphi - y$$

# Single Node with Sigmoid & Cross-Entropy Loss (i.e., Logistic Regression)



**And, finally, we can get gradient of loss with respect to weights and bias. For example, for the first weight...**

**Evaluate  $\varphi$  based on current values of parameters and the data.**

**Finally, update the weights...**

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dz} \cdot \frac{dz}{dw_1}$$

$$\frac{d\mathcal{L}}{dw_1} = (\varphi - y) \cdot x_1$$

$$w_{1,new} = w_{1,old} - \left( \frac{d\mathcal{L}}{dw_{1,old}} \cdot \varepsilon \right)$$

# Tensorflow GradientTape: AutoDiff

## 1. Gradient Tape

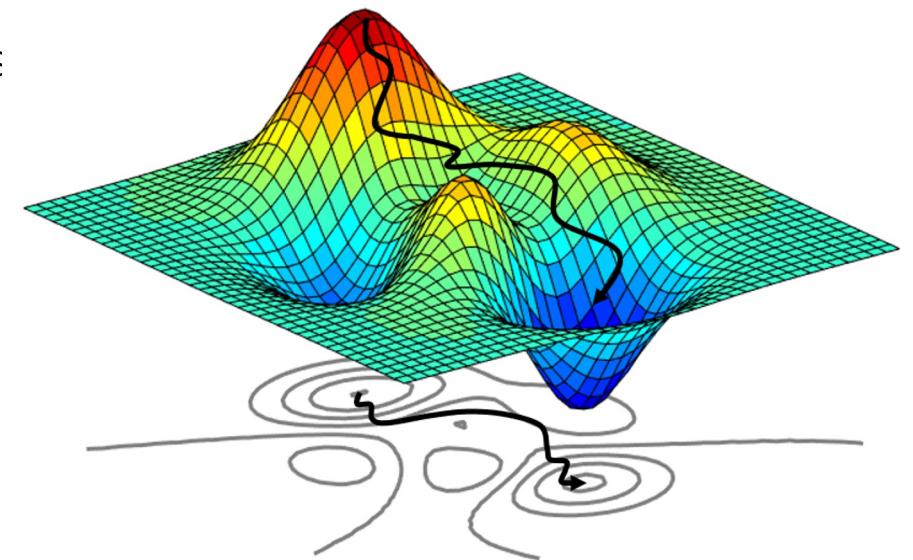
- A Tensorflow function that automates the calculation of derivatives.
- It constructs a computation graph in the background and implements codified rules for calculating derivatives of functions.
- You could technically use gradient tape to implement a gradient descent algorithm for many optimization problems.



# Optimizers

## Keras Supports 8 Optimizers

- SGD = Stochastic Gradient Descent
- Momentum
- Ftrl (2010) = Follow the Regularized Leader
- Adagrad and Adadelta (2012) = Adaptive Gradient Descent
- RMSprop (~2012) = Root Mean Squared propagation
- Adam (2015) = Adadelta / RMSProp with Momentum.
  - Adamax, Nadam are extensions to Adam.



# SGD: Gradient Descent

## Types of GD

- Batch GD = Use all the available training data in each pass.
  - Works well if the loss surface is smooth and lacks any saddle points / valleys.
- Stochastic GD = Mini-batch with batch size = 1.
  - If troughs / saddles exist, we move past them as our exploration of gradients for the model will vary with a given observation that we are considering in an iteration.
  - Computationally quite burdensome but performs well on non-linear problems (eventually).
- Mini-batch GD = What we have been doing so far (randomly split the data in each epoch, into folds, and then cycle over the folds for training).
  - This is a happy-medium between batch and stochastic GD.

## Role of Batch Size

- Empirically has been observed that smaller batches yield less overfitting (because of implicit noise in the training process – variance of the gradients obtained will go up).

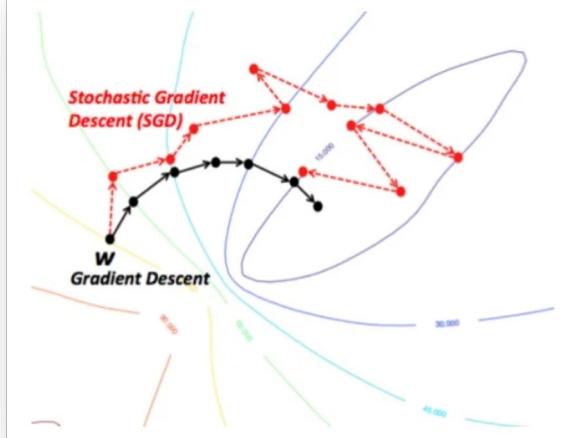
# Batch (All) vs. Stochastic (1)

## Same Convergence

- If you have a convex surface, either approach will converge to the global optimum (no guarantee your problem is convex of course). Always converges at least to a local minimum.

## Tradeoffs

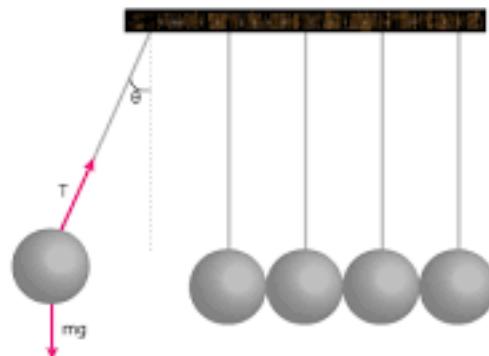
- Batch, each step is slower, more computationally burdensome, but convergence with fewer iterations; Need to be able to hold the entire dataset in memory.
- SGD makes noisier updates, and requires more iterations to converge, but a single iteration is quick. Only need one observation in memory, at a time.



# Momentum

## Getting Past Local Minima

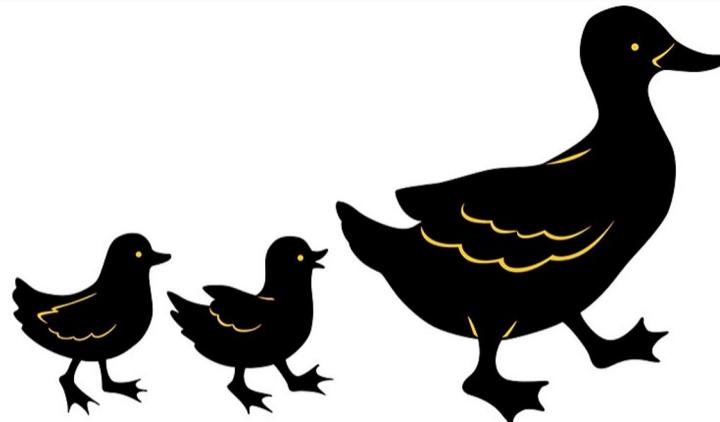
- SGD gets stuck in local minima; the idea of momentum is to make updates be a function of current gradient\*learning rate, as well as some fraction (decay) of the update you made last iteration.
- This reduces updates to parameters where the gradients are flipping sign and amplifies updates to gradients that are going in a consistent direction (steeply descending).



# FTRL

## Google Developed in 2010...

- This is an optimization technique that is used in “online” learning; it’s typically used in situations where your model training is happening continuously as new data arrives, and where drift might therefore happen.
- It works well in situations where you have a ton of sparse features.
- Was originally used for predicting conversion in online advertising systems.



*Follow The Leader*

videogamegenius

# Adagrad & Adadelta (RMS Prop)

## Adaptive Gradient Descent (Variable Learning Rate)

- We implicitly apply a high learning rate for features we have been updating very little so far (speed up movement through saddle points, for example).
- We implicitly apply a low learning rate for features we have been updating a lot so far.
- Technically learning rate is removed from the process, every update is a function of past updates.

## Adadelta

- Same idea but we use a sliding window of previous updates to determine magnitude of current updates (rather than all prior updates).
- RMSProp is conceptually very similar but was independently developed (around the same time).

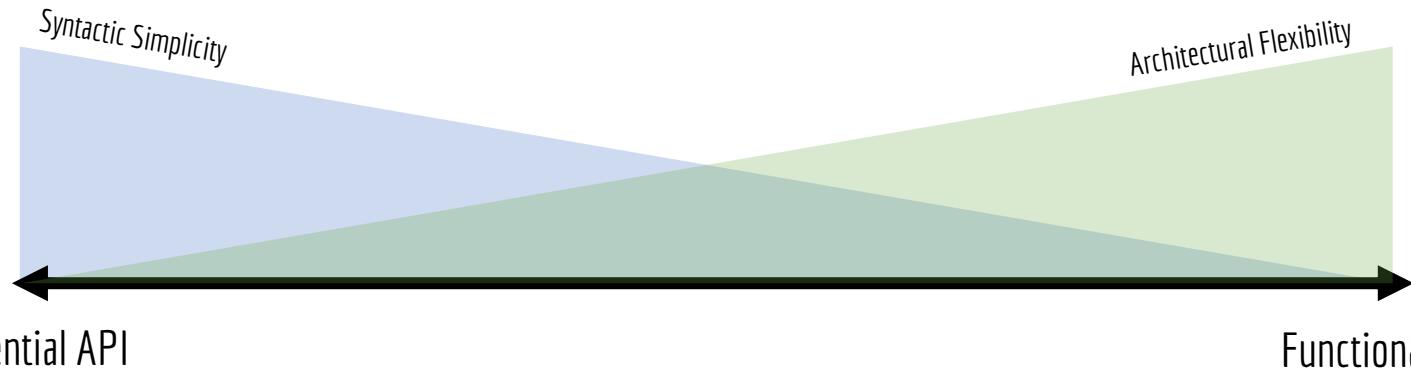
# Note: Sequential vs. Functional API

## We Have Only Seen Sequential API So Far

- Sequential is easy to work with but is also very inflexible. Can only really handle basic feed-forward networks. It automatically figures out the shape of each layer's output tensor and specifies the next layer's input shape accordingly.

## Functional API Let's You Construct Any Topology You Want

- We will see the difference in how each API is used, syntactically.



# Recap

## Building Blocks of NNs

- Tensors and Tensor Operations
- Activation Functions
- Loss Functions
- Backpropagation: Derivatives, Gradients & the Chain Rule

## Procedure of Minibatch Stochastic Gradient Descent

- Grab a batch of observations (samples)
- Predict their labels using current weights / bias terms.
- Calculate loss value.
- Calculate gradient of loss w.r.t. all weight / bias terms.
- Update each weight by subtracting its gradient\*learning rate
- Cycle over the whole training dataset (each cycle is an epoch) repeatedly, until loss is small.