Intro to Neural Nets

Mathematical Building Blocks &
Working with Keras API

Today’s / Next Week’s Agenda

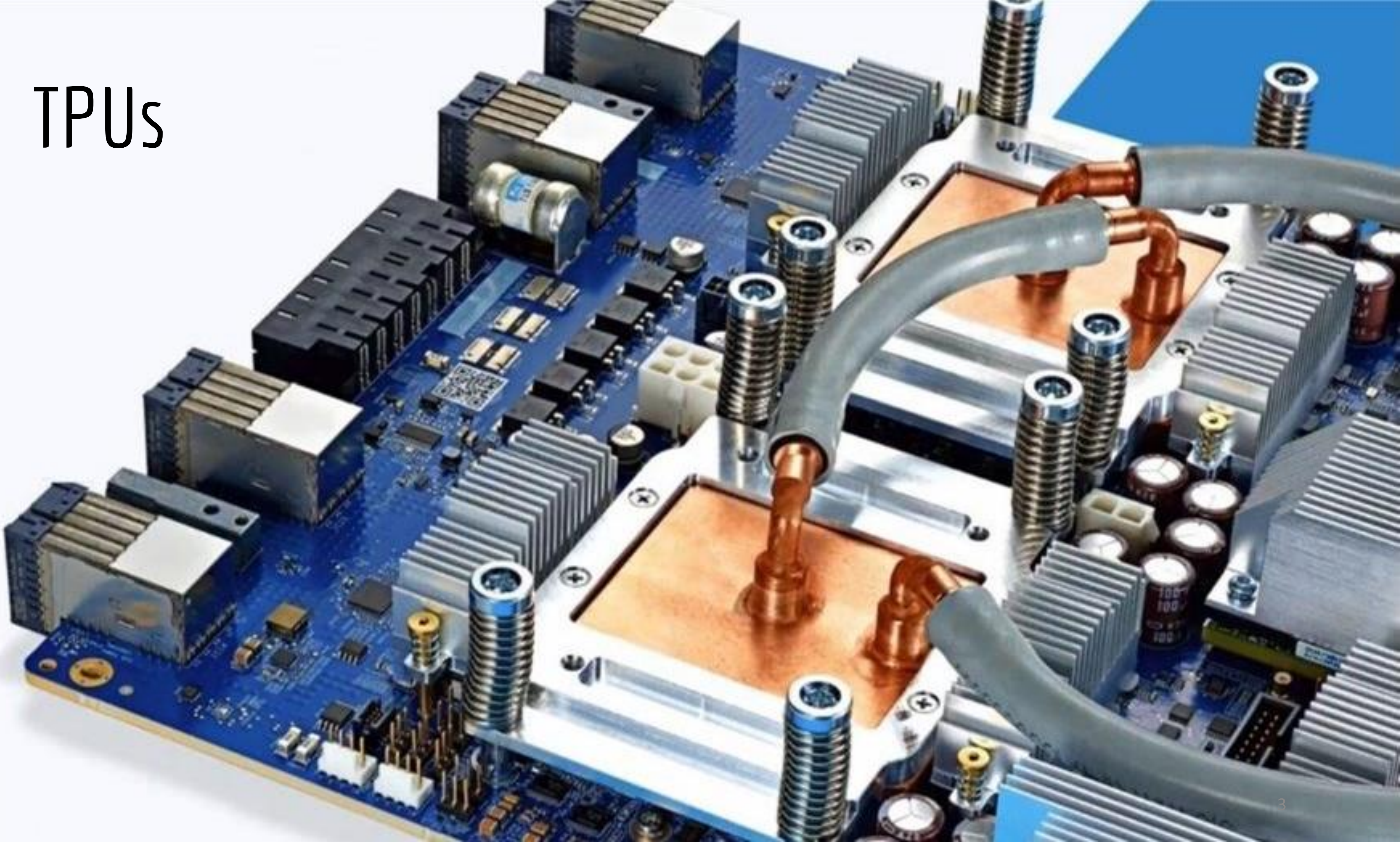
1. Building Blocks of NNs

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

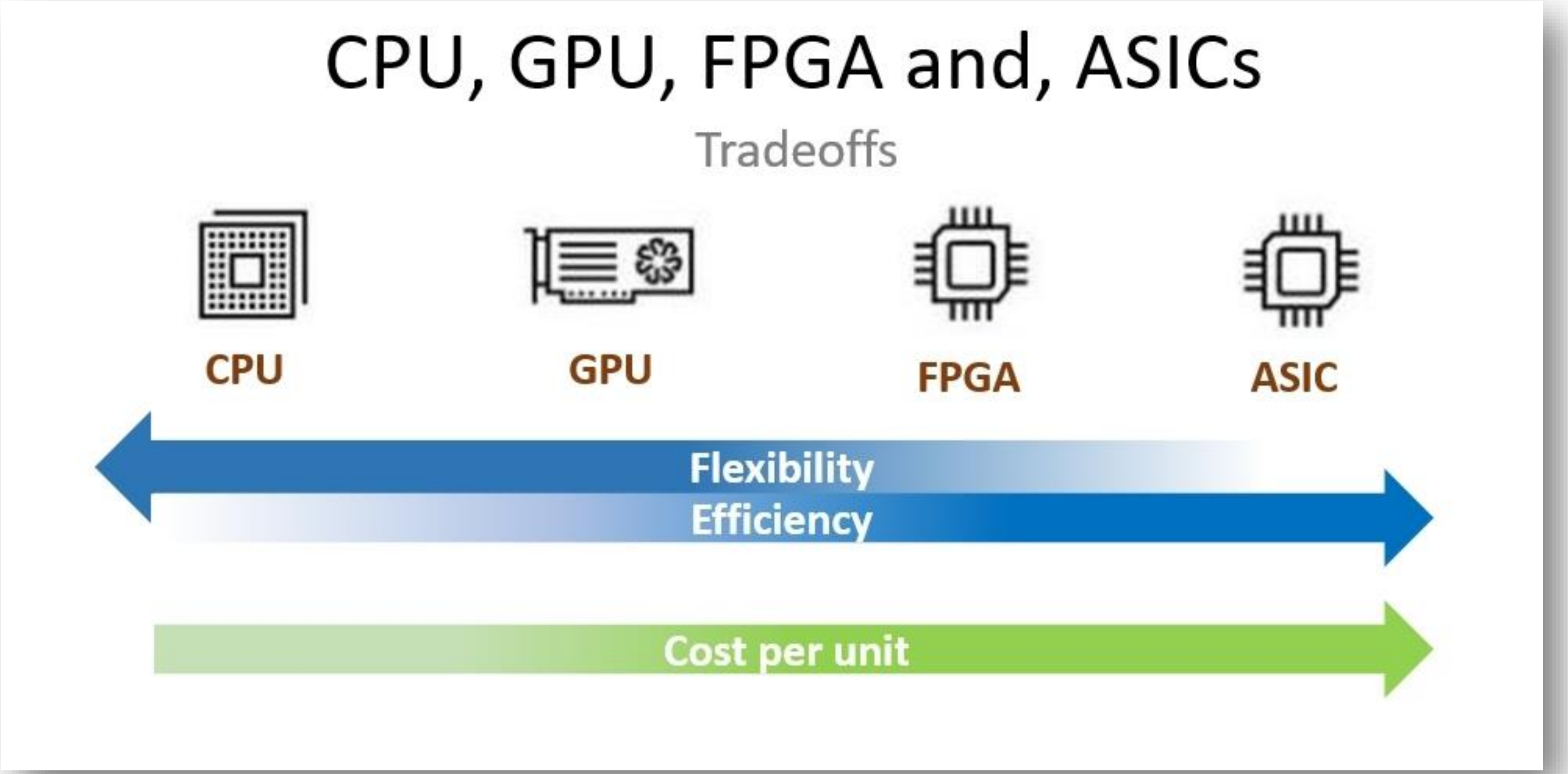
2. Building a Linear Classifier

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

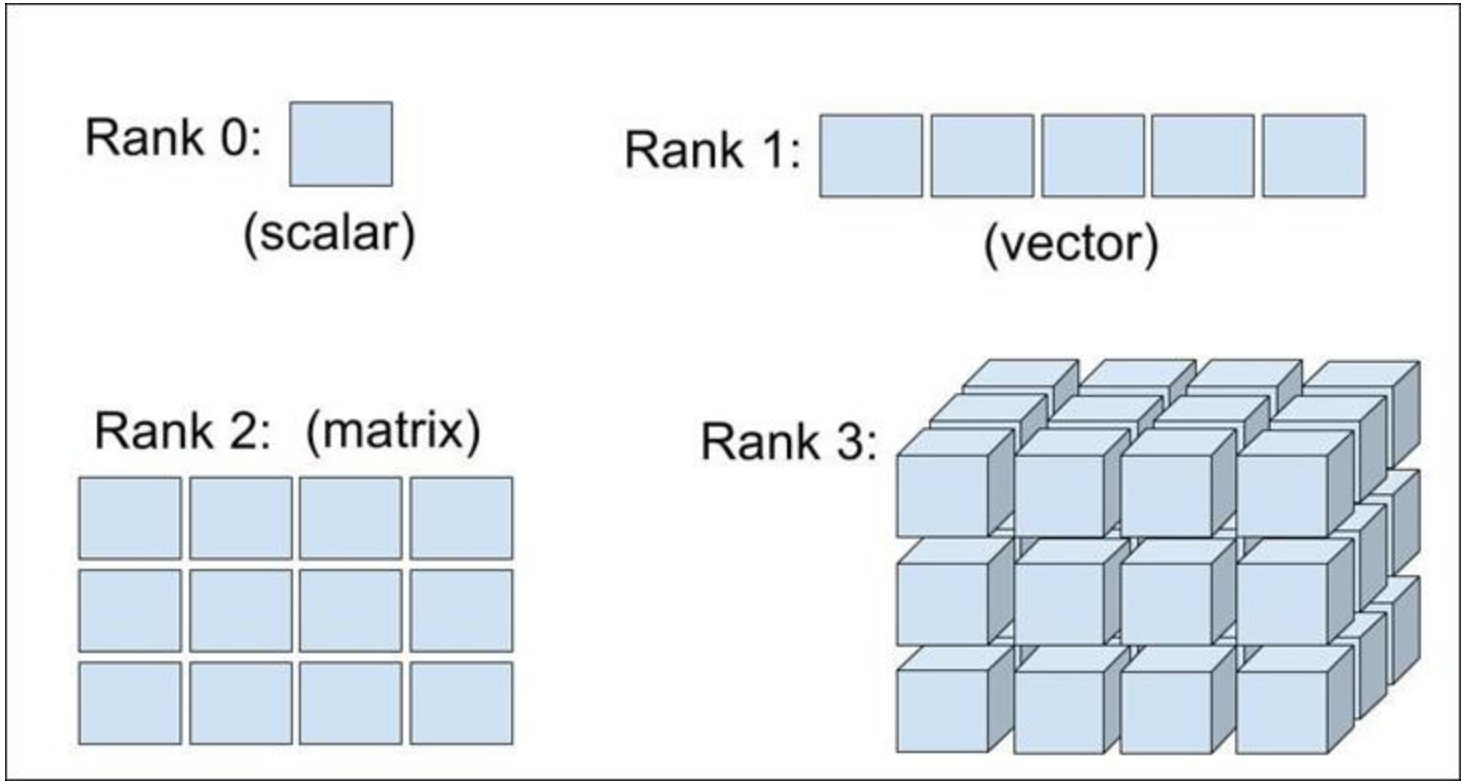
TPUs



An Aside: GPU vs. ASIC

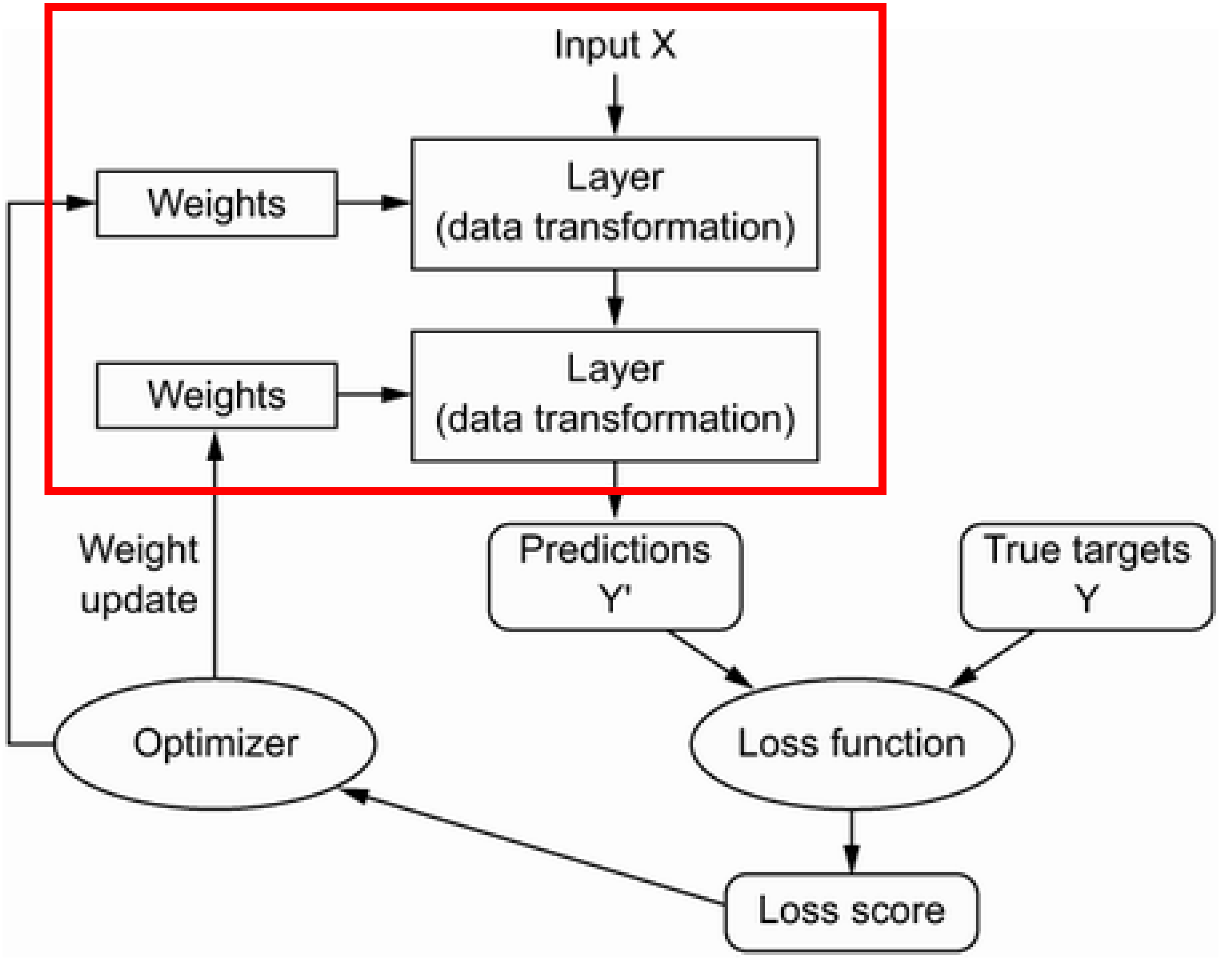


Tensors

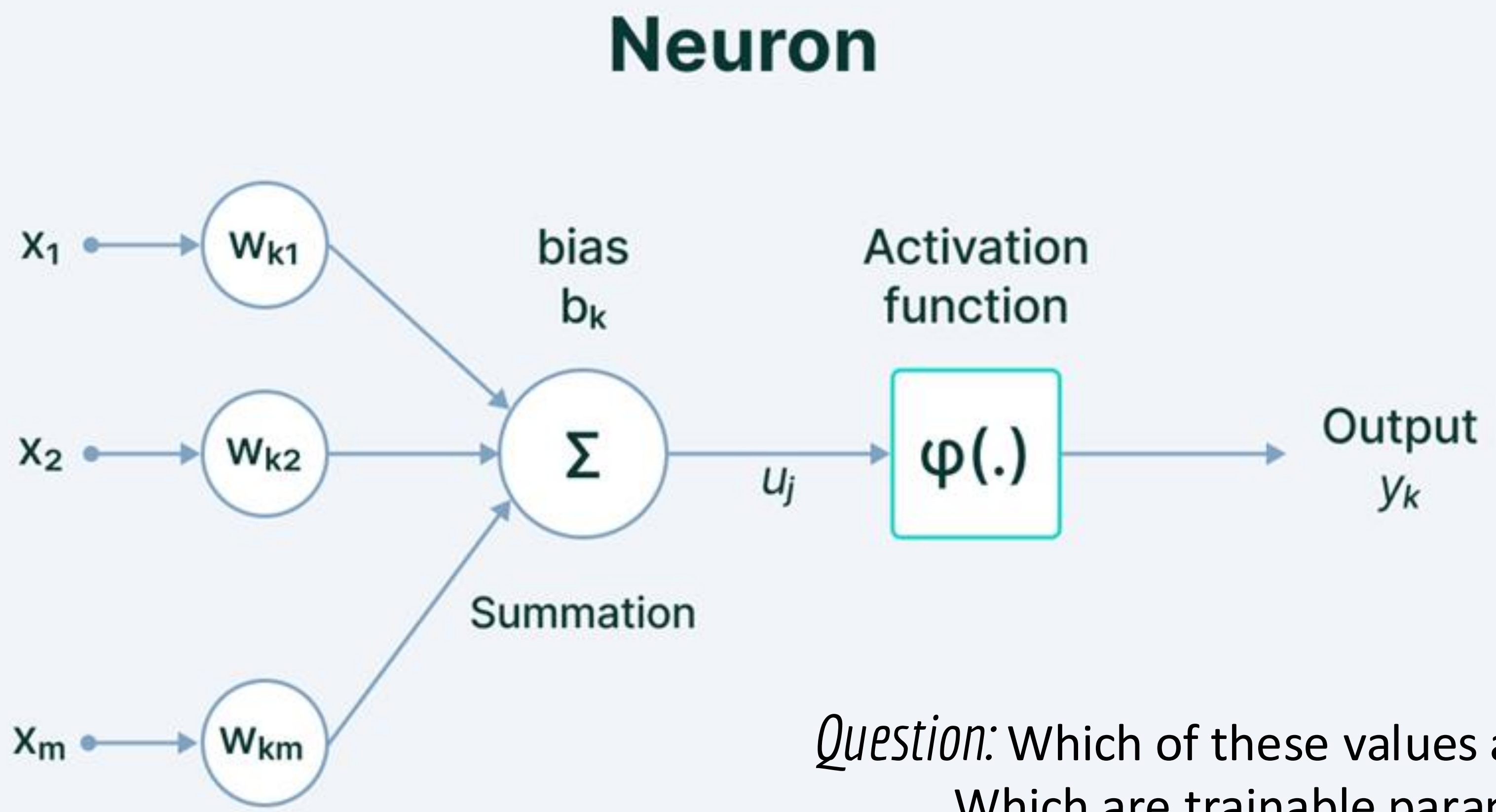


Question: What sort of data (give an example) would be stored in a rank-3 tensor? How about a rank-4 tensor?

Forward Pass

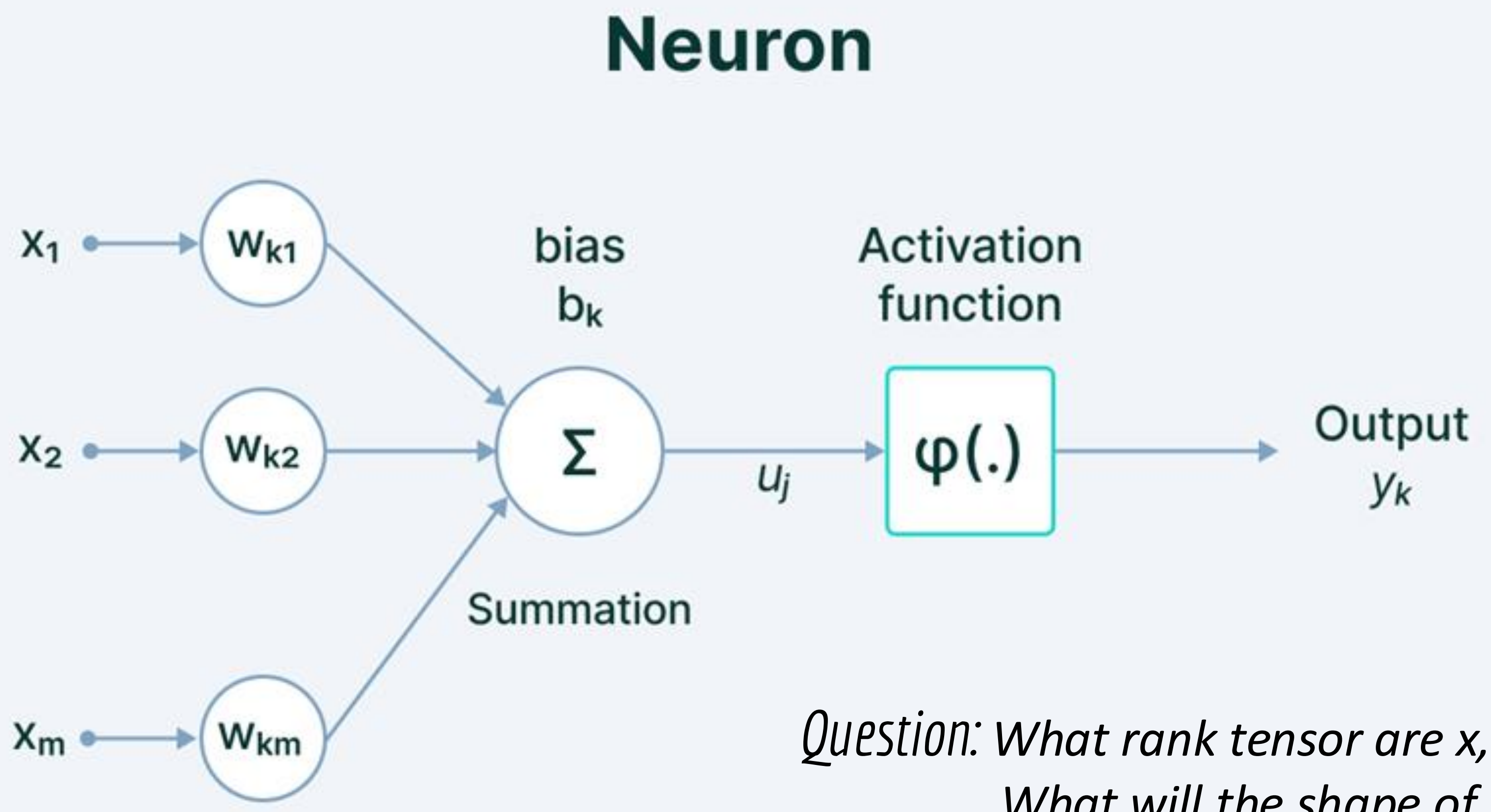


Neuron / Network Components



Question: Which of these values are constants?
Which are trainable parameters?

Neuron / Network Components



Question: What rank tensor are x, w and b here?
What will the shape of y be?
What is the order of operations in a forward pass?

Multiplication

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

Conformity of Shapes

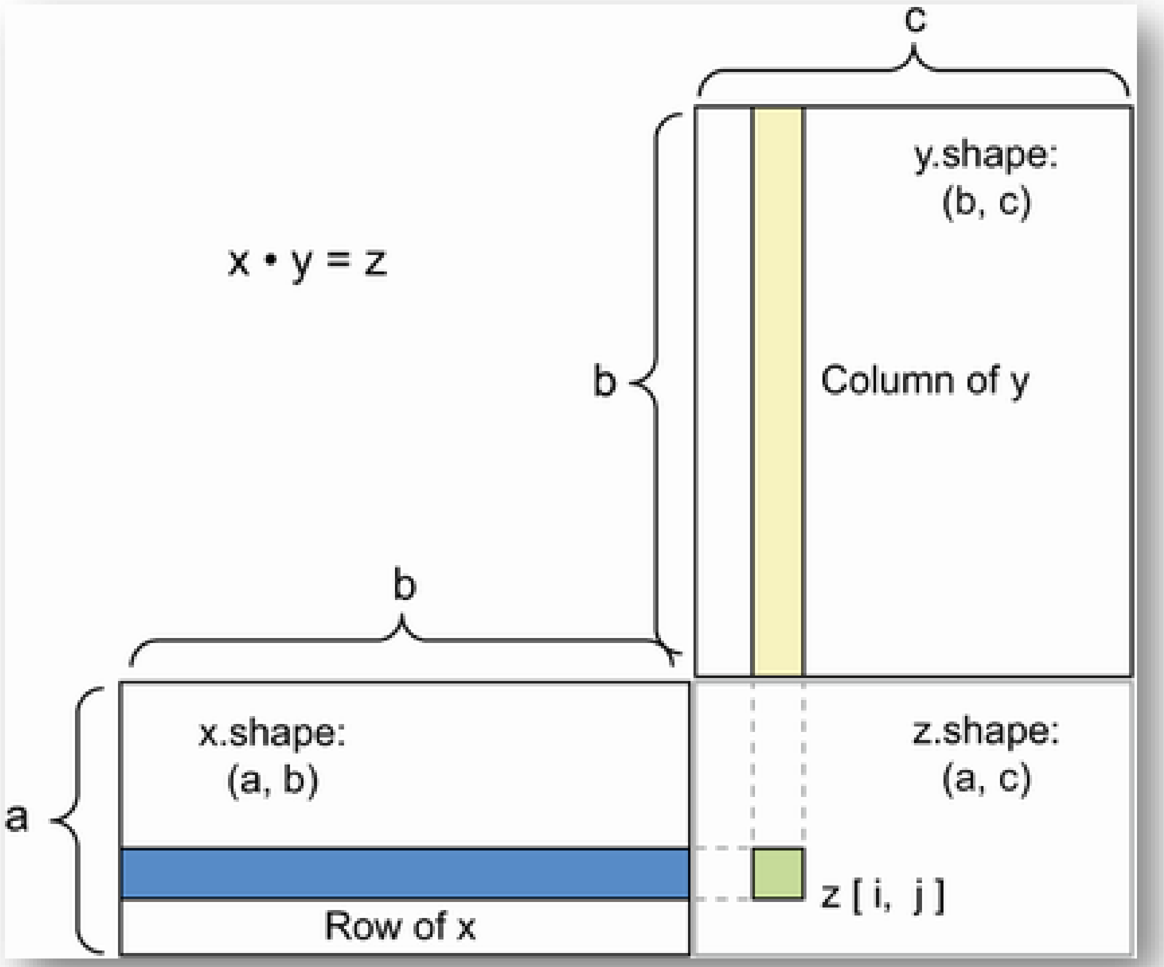
- NCOL(X) == NROW(W)

Elements of Resulting Tensor are the Dot Product of X's Rows and Y's Columns

- Z[2,2] = X[2,:] · Y[:,2]

We Use This for Multiplication Step

- x*w calculations.



Matrix Addition (Broadcast)

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

Shape of the Two Tensors Needs to Conform

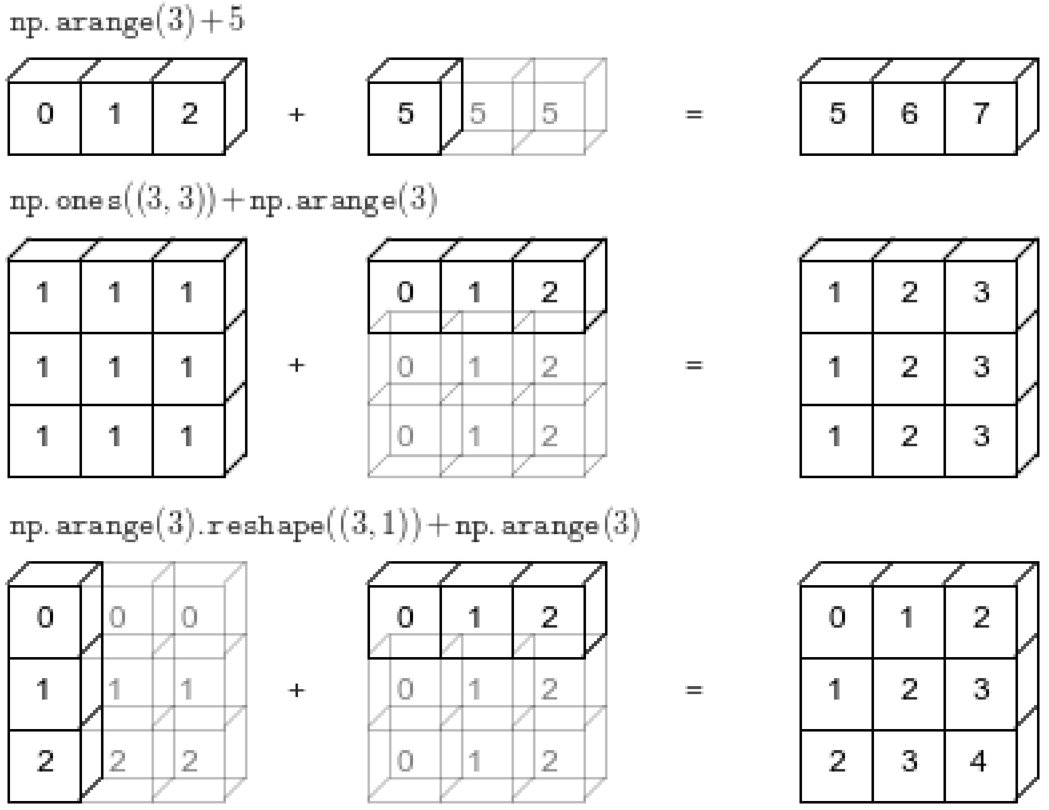
- A + B will only work if A is cleanly divisible by B (or vice versa)

Sum Element-wise

- Replicate B until it matches A’s dimensions, then perform element-wise addition.

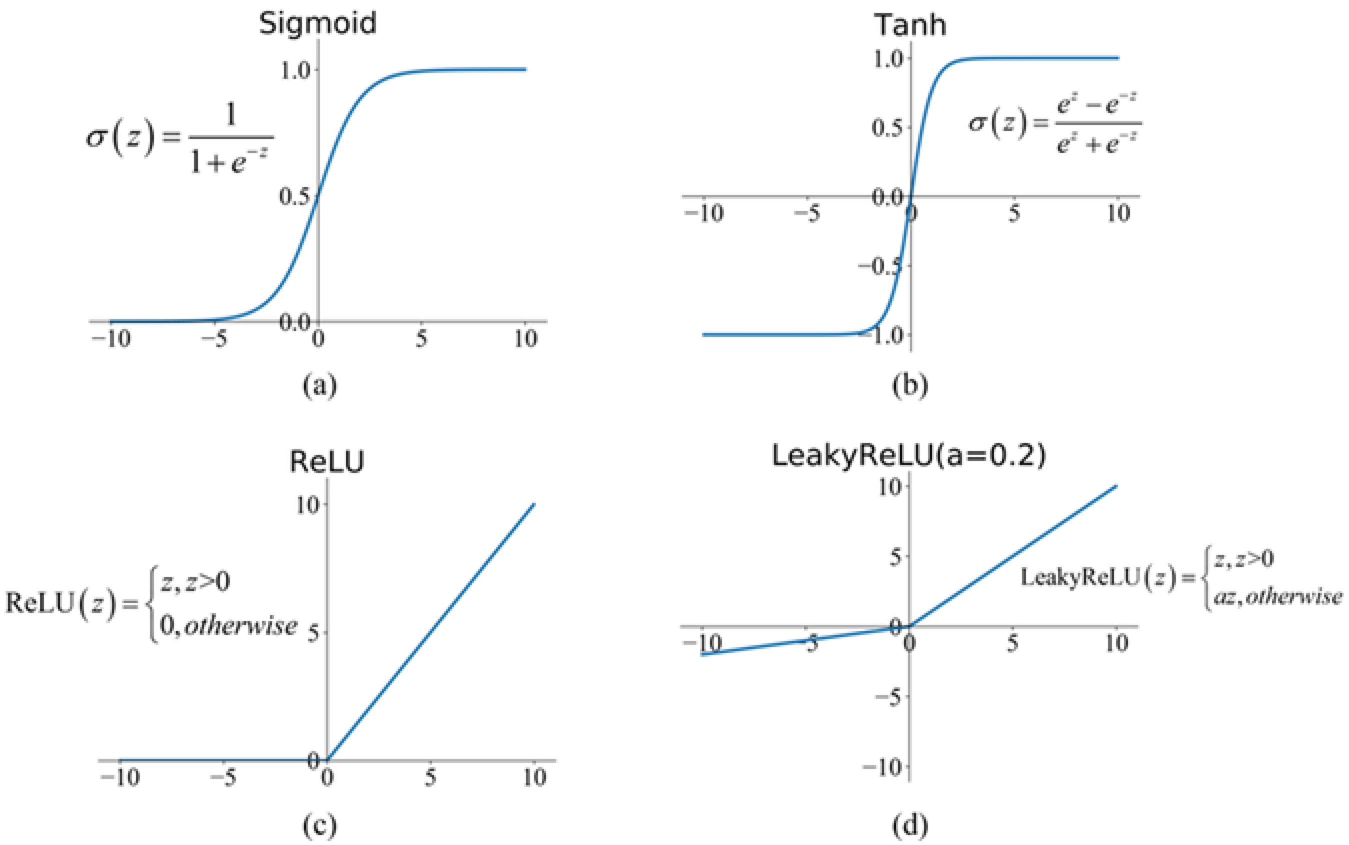
We Use This for the Addition Step

- Add x*w and b (bias)



Activation Functions

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



Multi-Class, Single-Label

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

Softmax (MLOGIT):

We have D inputs (x's).
We have k outputs (classes).

So, W is a (D,k) matrix and X is a (D,1) matrix.

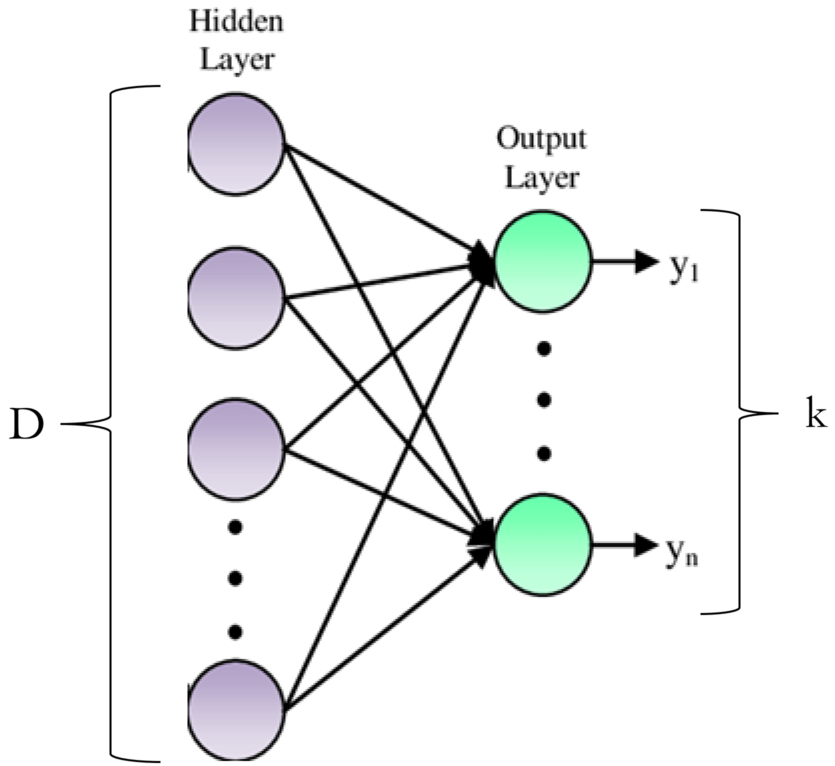
That means, A is a (k,1) matrix.

That means Y is also a (k,1) matrix.

$$A = W^T X,$$

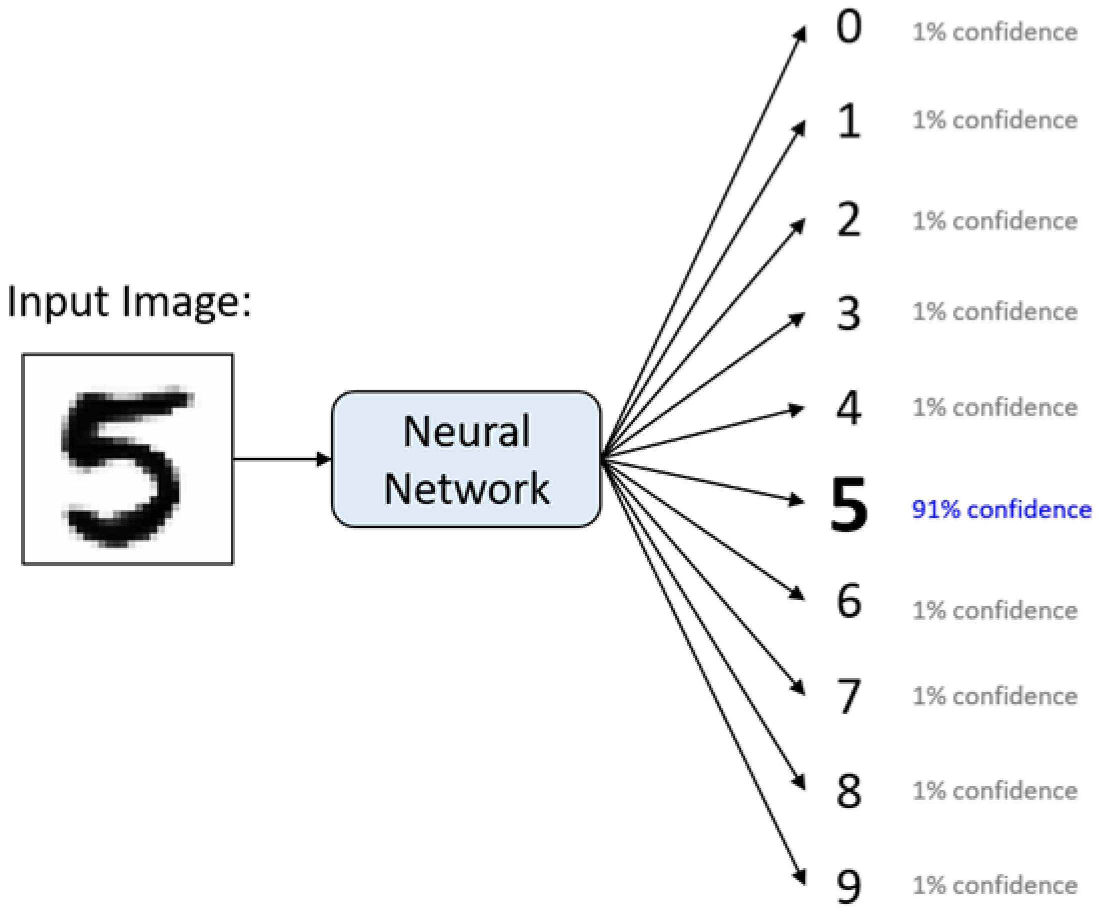
$$Y = \text{softmax}(A),$$

$$Y_i = \frac{e^{A_i}}{\sum_{j=1}^{k} e^{A_j}}.$$



Multi-Class, Single-Label

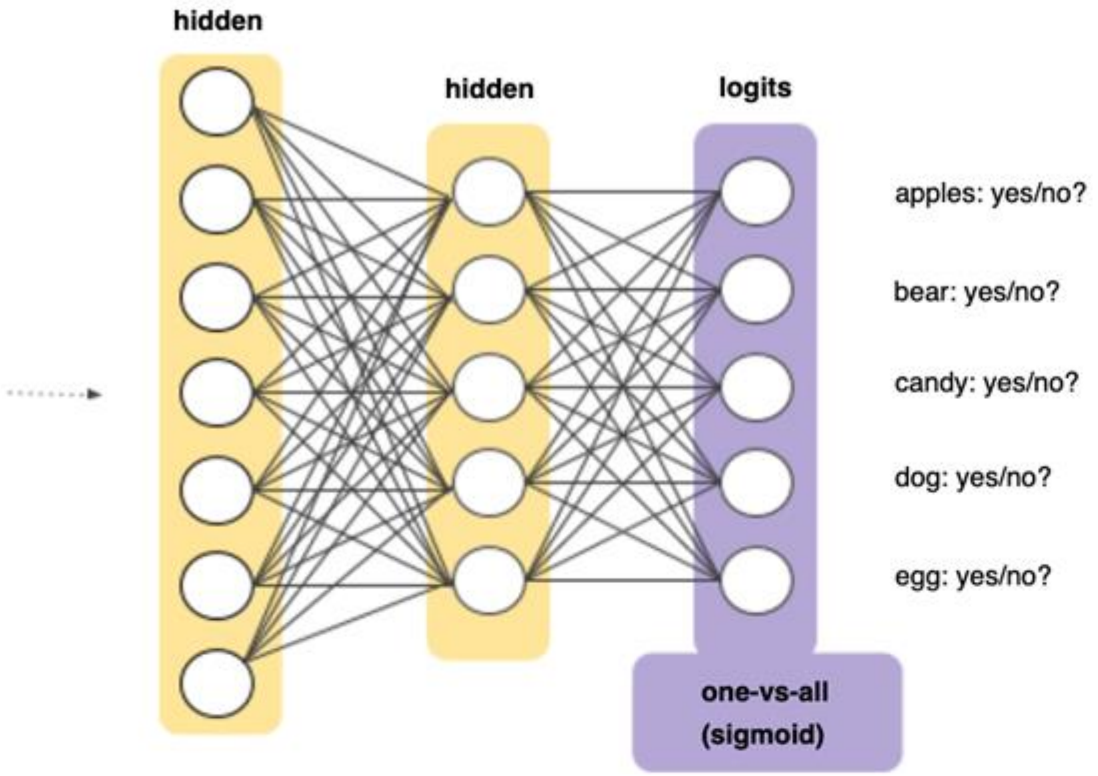
$y_1 = \varphi\,(x_1 \cdot w_1 + b_1)$



Multi-Class, Multi-Label

Many Non-Exclusive Labels

- We would create a sigmoid output layer with one output for each class we are predicting.
- Train on all labels together.



We Know Enough for a Forward Pass

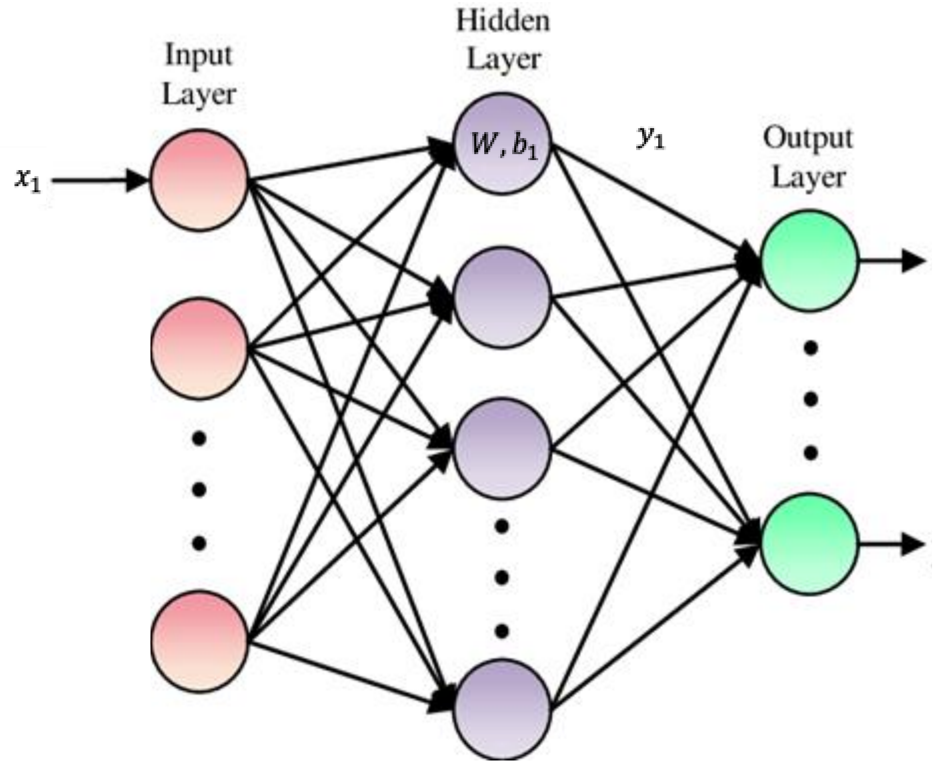
Calculate Output of Each Node Sequentially

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

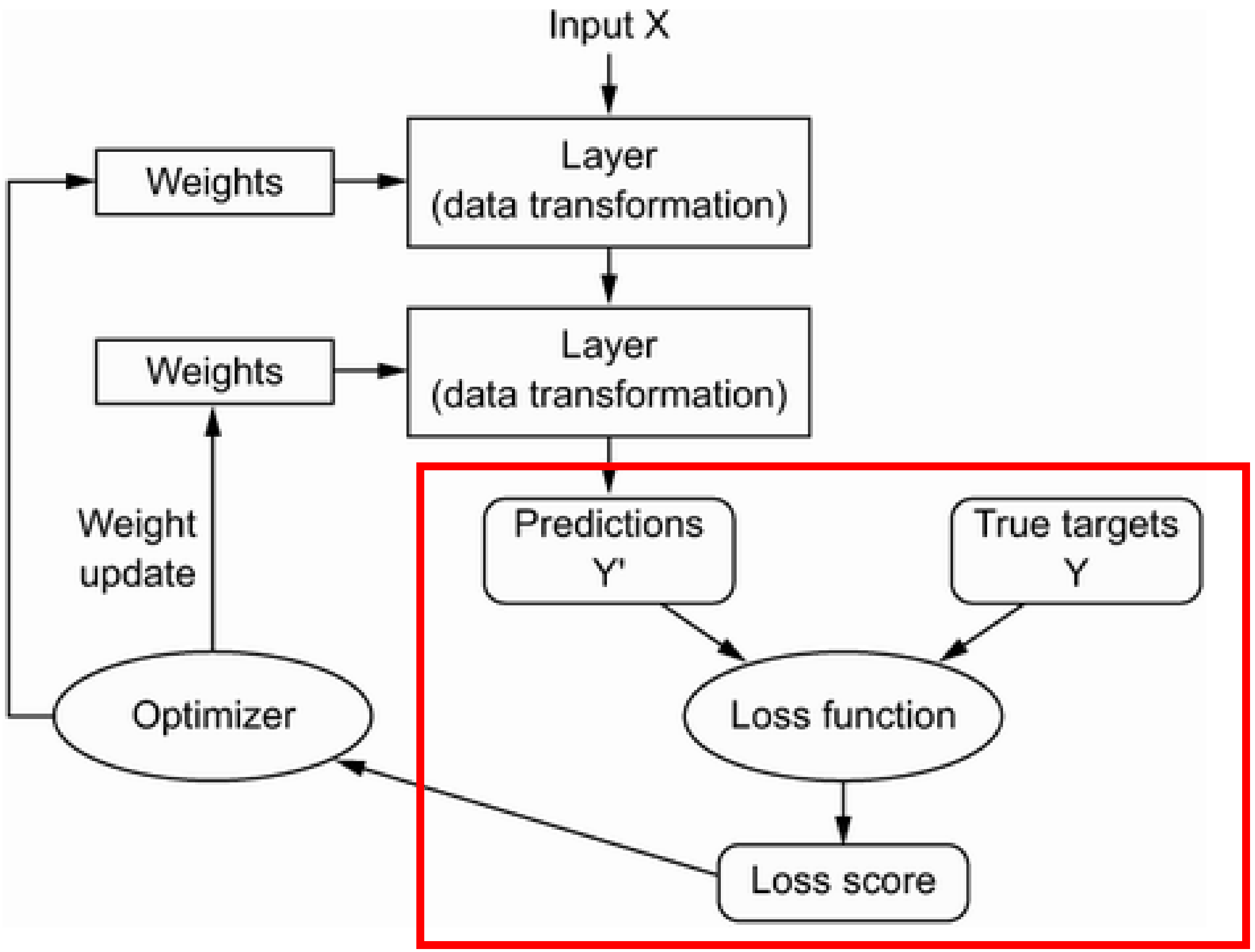
$$y_2 = \varphi\left(x_1 \cdot w_{2,1} + x_2 \cdot w_{2,2} + \cdots + b_2\right)$$

…

Eventually We Obtain Model's Predictions



Calculate Loss



Loss Functions

<u>Cross-Entropy / Log-Loss</u>

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

- 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.

$$CE = -\sum_{i}^{C} t_i log(s_i)$$

<u>MAE / L1 Loss</u>

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

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

<u>MSE / Quadratic / L2 Loss</u>

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

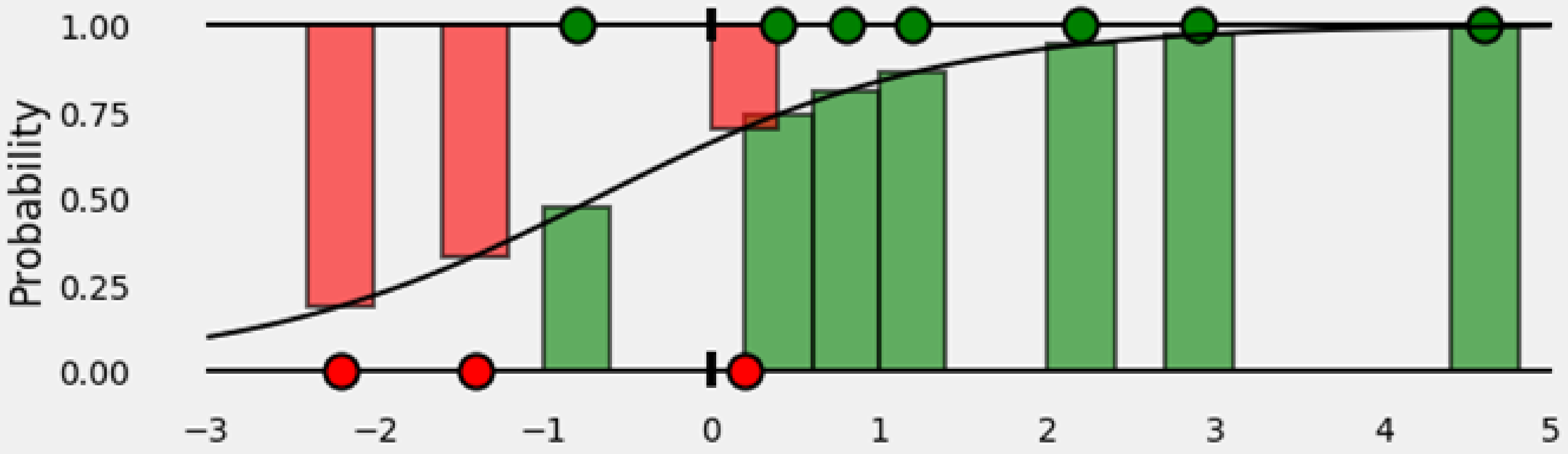
- Typical for continuous outcomes, larger errors penalized exponentially more. This should look familiar!

Binary Cross-Entropy Loss

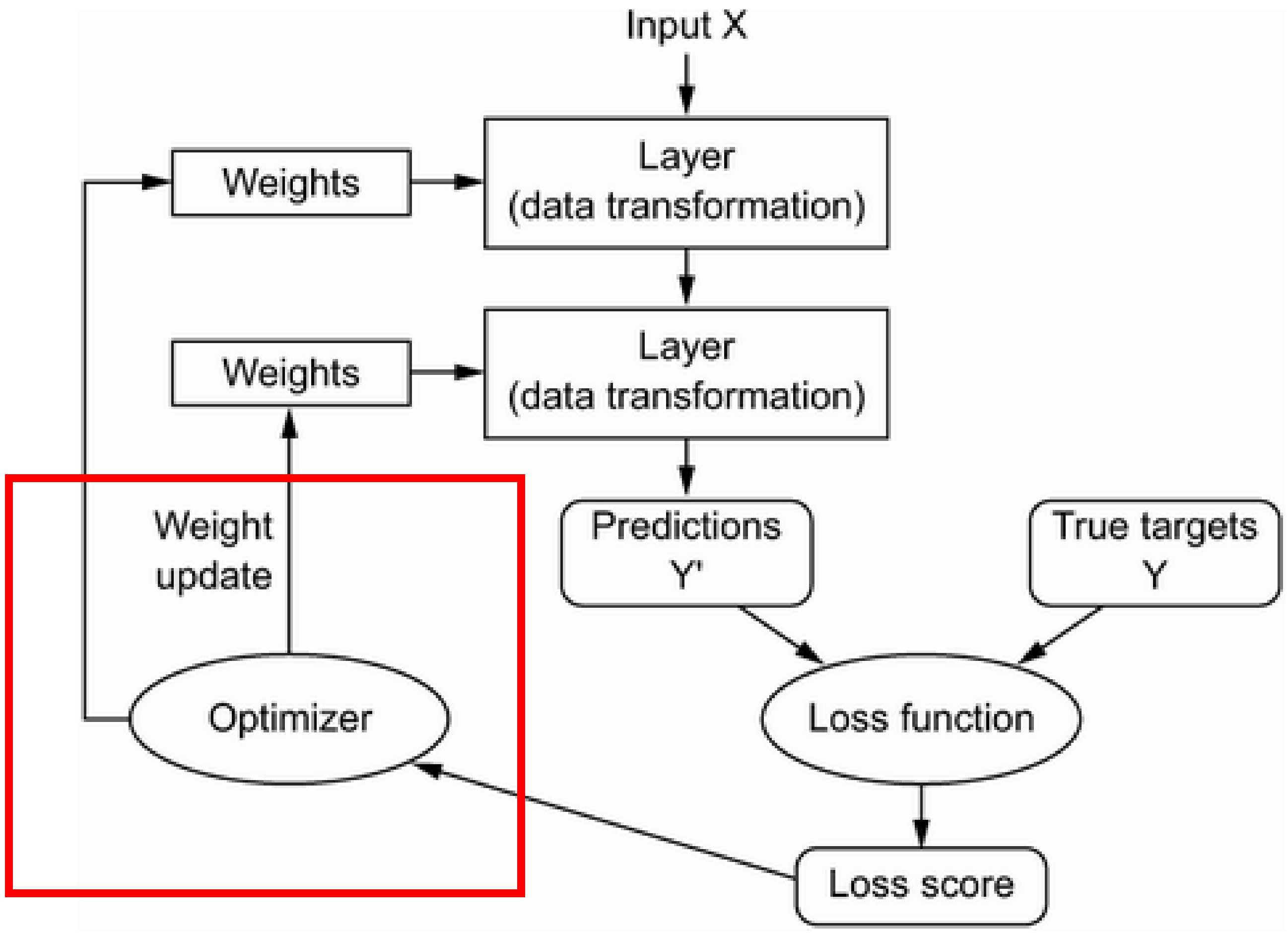
$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Piecemeal Function:

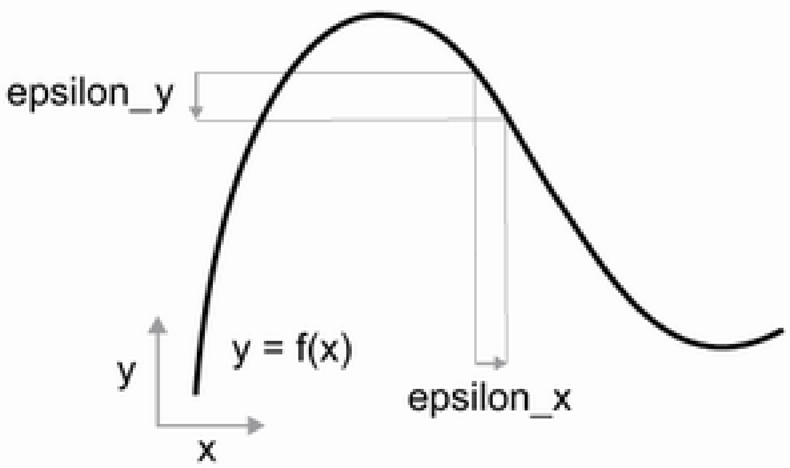
- If ground truth is 1, then loss is -1*log(*p*). As prediction approaches 1, loss approaches 0. As prediction approaches 0, loss grows exponentially.
- If ground truth is 0, then loss is -1*log(1-*p*). As prediction approaches 1, loss rises exponentially. As prediction approaches 0, loss approaches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

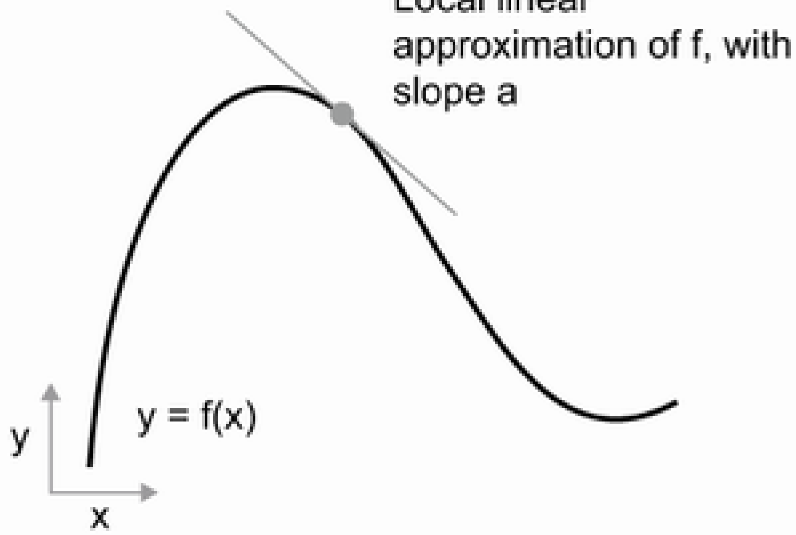


Backpropagation

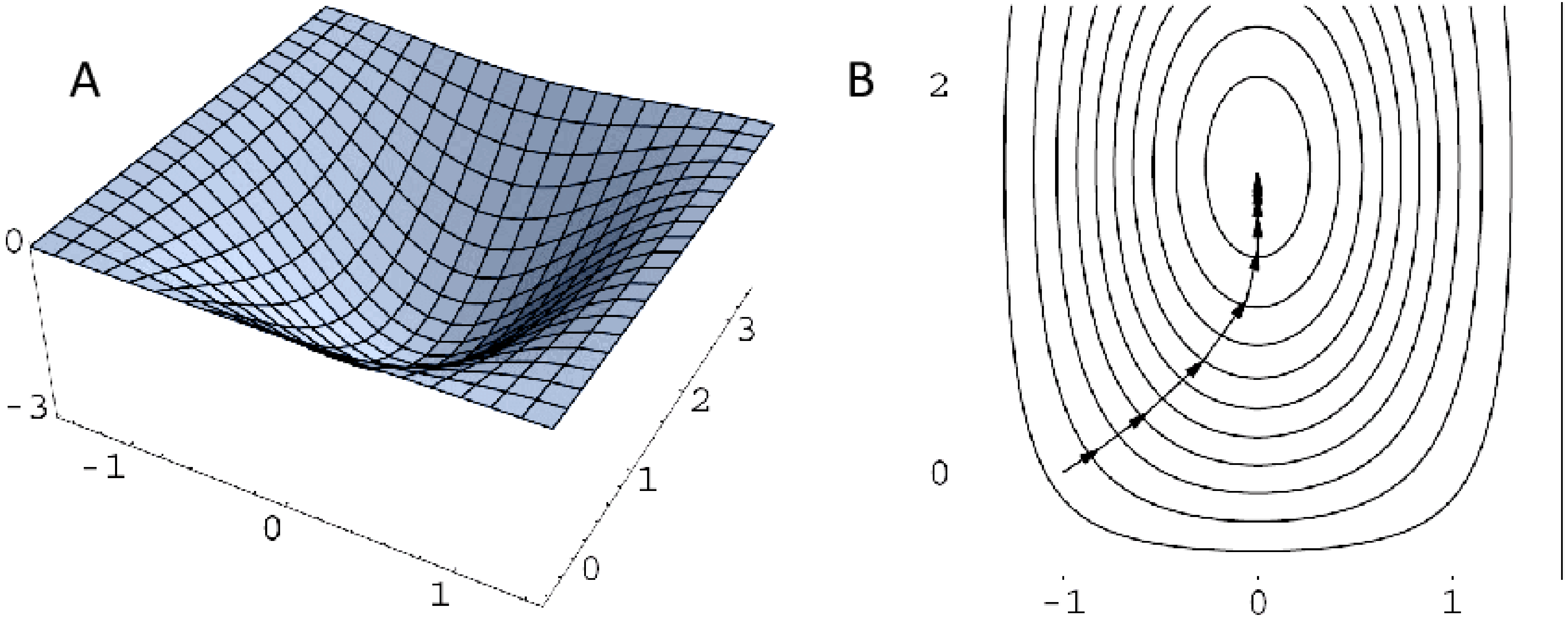


Derivative = “Rate” of Change

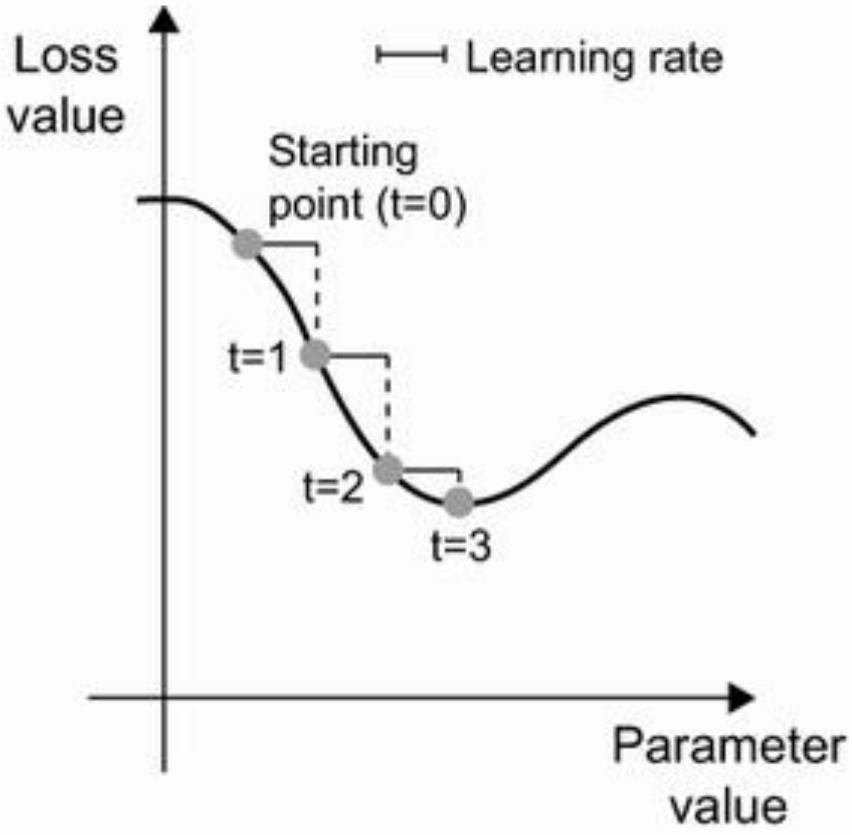




Gradient = Derivative in Multiple Dimensions



Gradient Descent



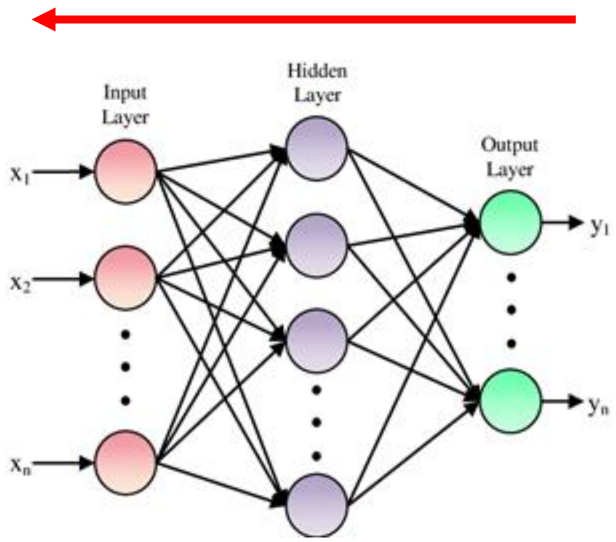
Derivatives of Loss w.r.t All Parameters

Recall that Each Node's Output Can be Expressed as a Function of the Prior Nodes' Outputs

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

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

$$\cdots$$

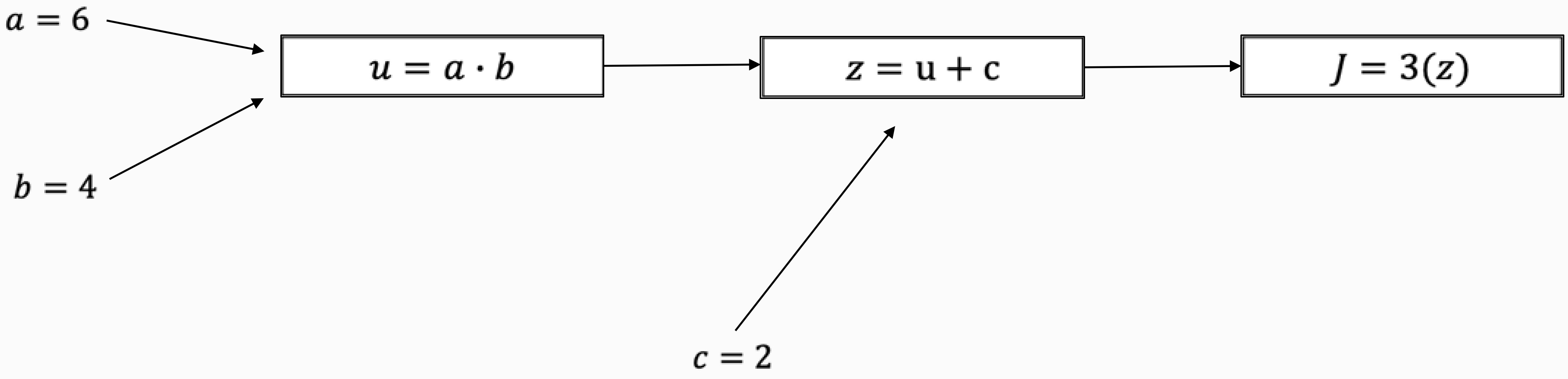


Start at the final nodes in the network and work backwards

- We calculate partial derivatives w.r.t. their inputs / weights.
- Then, use those partial derivatives and work backward into earlier layers to get partial derivatives w.r.t. *their* inputs / weights, and so on.

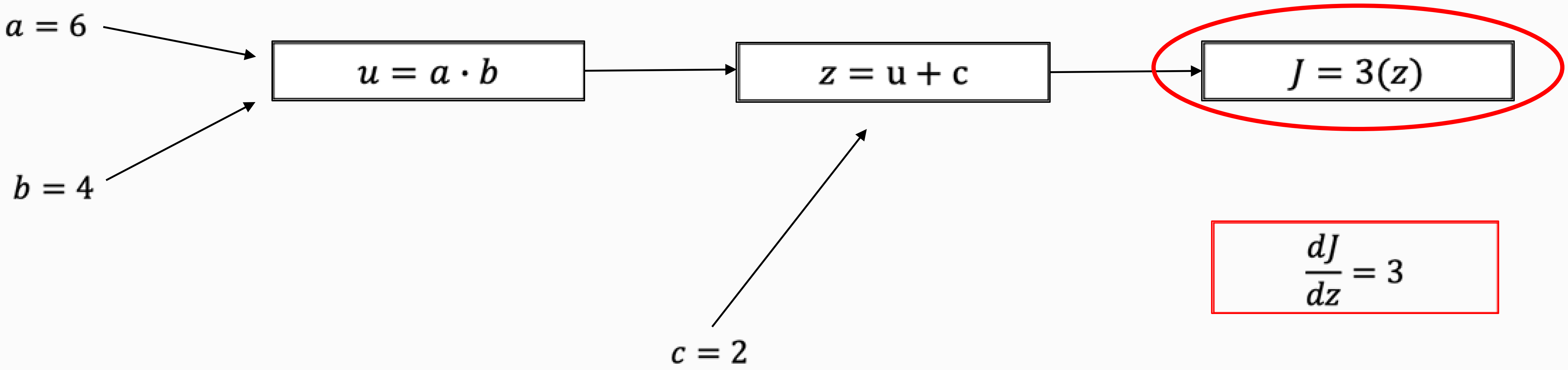
Simplifying Gradients: Computation Graph

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



Backpropagation = Working Backwards

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

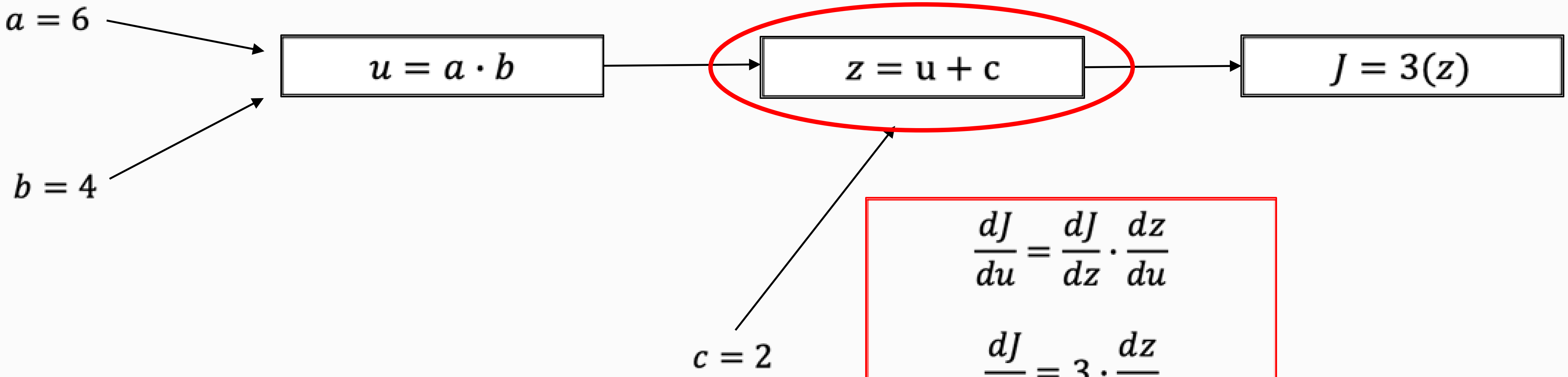


$$\boxed{\frac{dJ}{dz} = 3}$$

Backpropagation = Work Backwards

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

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



$$\frac{dJ}{du} = \frac{dJ}{dz} \cdot \frac{dz}{du}$$

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

$$\frac{dJ}{du} = 3 \cdot 1 = 3$$

Backpropagation = Work Backwards

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

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

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

$$\begin{array}{l} a = 6 \\ b = 4 \end{array} \longrightarrow \boxed{u = a \cdot b} \longrightarrow \boxed{z = \mathrm{u} + \mathrm{c}} \longrightarrow \boxed{J = 3(\mathrm{z})}$$

$$c = 2 \longrightarrow \boxed{z = \mathrm{u} + \mathrm{c}}$$

$$\frac{dJ}{dc} = \frac{dJ}{dz} \cdot \frac{dz}{dc}$$

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

$$\frac{dJ}{dc} = 3 \cdot 1 = 3$$

Backpropagation = Work Backwards

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

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

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

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

$a = 6$

$b = 4$

$$u = a \cdot b \rightarrow z = \mathrm{u} + \mathrm{c} \rightarrow J = 3(\mathrm{z})$$

$c = 2$

$$\frac{dJ}{da} = \frac{dJ}{du} \cdot \frac{du}{da}$$

$$\frac{dJ}{da} = 3 \cdot b$$

$$\frac{dJ}{da} = 3 \cdot 4 = 12$$

Backpropagation = Work Backwards

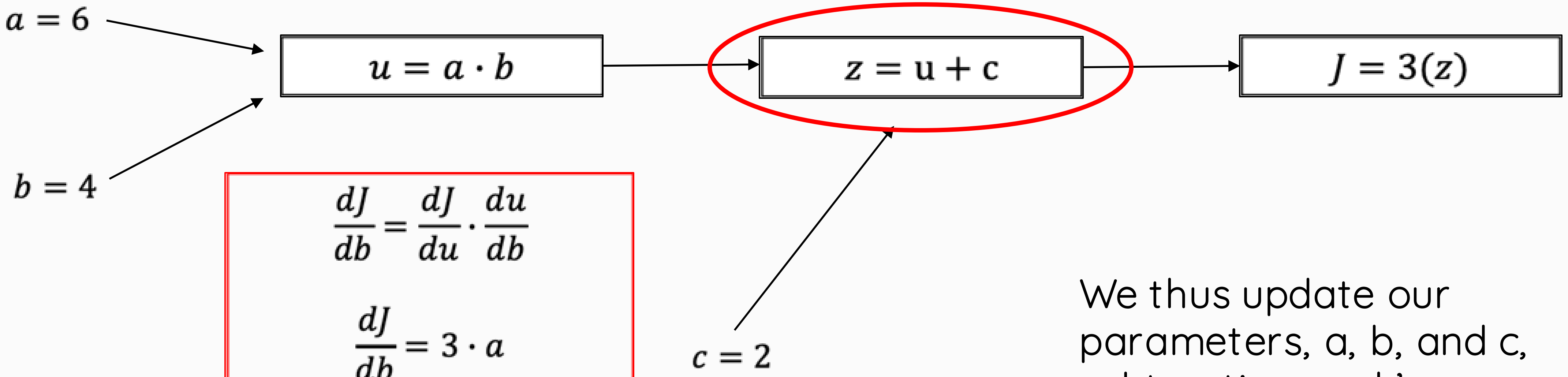
$$\frac{dJ}{dz} = 3$$

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

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

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

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



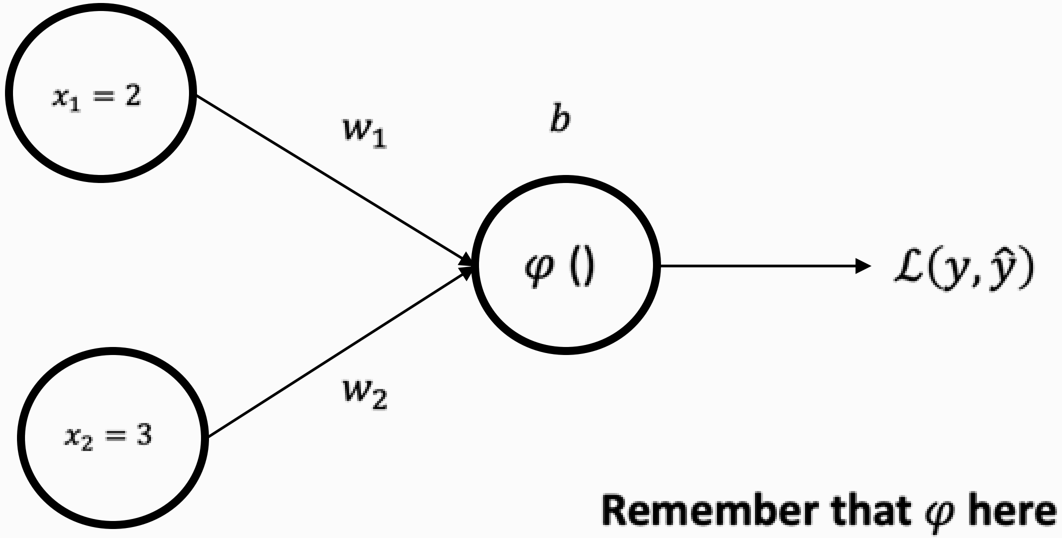
$$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db}$$

$$\frac{dJ}{db} = 3 \cdot a$$

$$\frac{dJ}{da} = 3 \cdot 6 = 18$$

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)



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

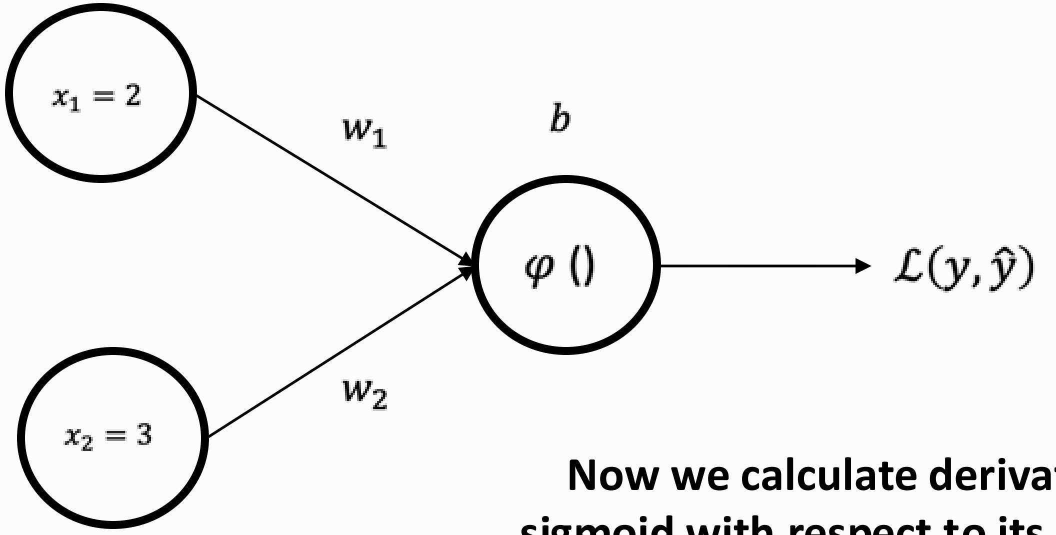
$$\frac{d\mathcal{L}}{d\varphi} = \frac{\varphi(1-y) - y(1-\varphi)}{\varphi(1-\varphi)}$$

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

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

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

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



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

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

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

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

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

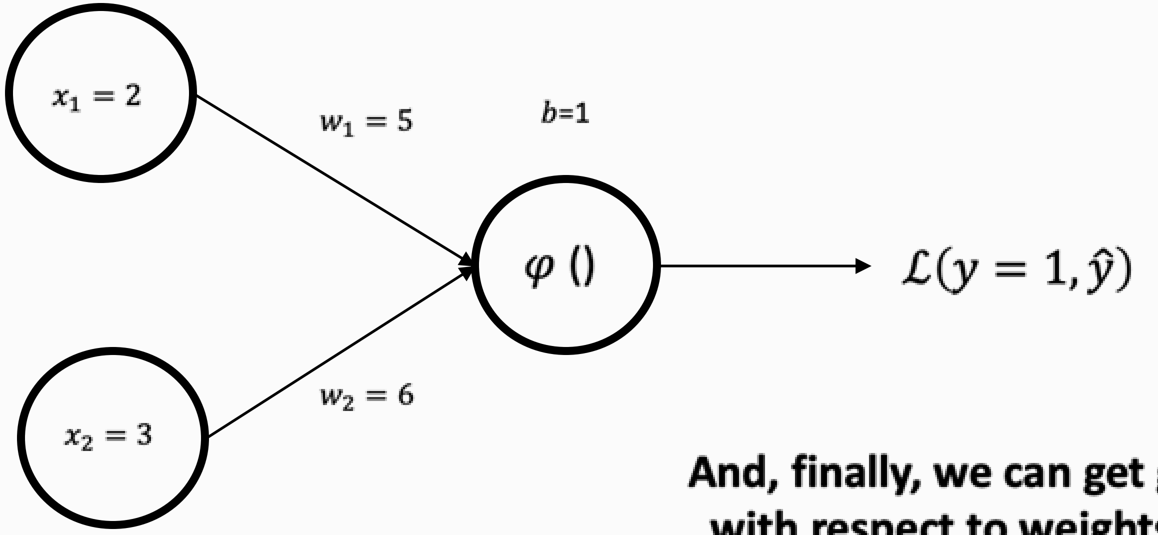
$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{d\varphi} \cdot \frac{d\varphi}{dz}$$

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

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

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

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 φ based on current values of parameters and the data.

Finally, update the weights…

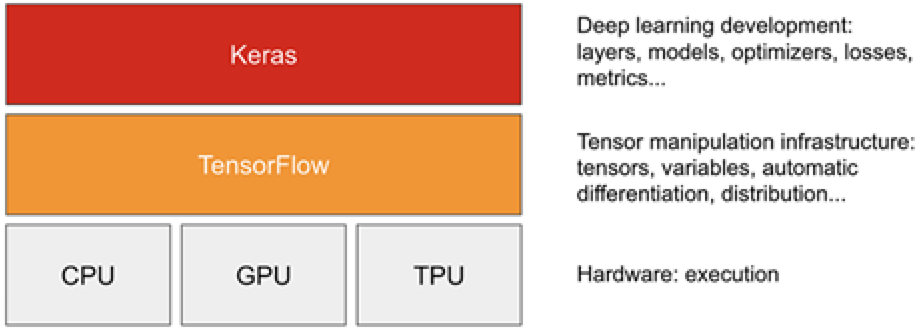
$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dz} \cdot \frac{dz}{dw_1}$$

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

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

Keras and Tensorflow

1. Tensorflow
    - A Python platform for working with tensors, implementing automatic differentiation, providing access to repositories of (well-known) pre-trained models.
2. Keras
    - A higher-level API that wraps common usage patterns with Tensorflow functions, pre-defined loss functions, optimization algorithms, etc.
    - Keras simplifies data scientists' interaction with Tensorflow.



Tensorflow GradientTape: AutoDiff

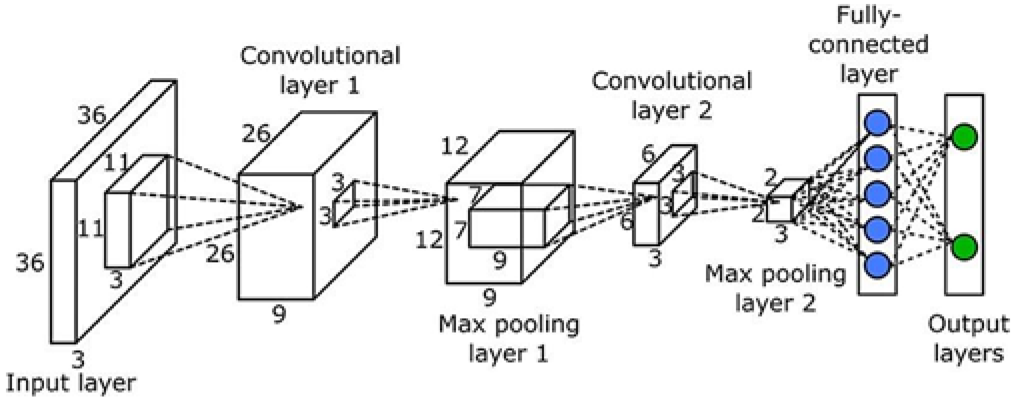
1. Gradient Tape

- 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.

The Layer

Layers are the Key Building Block of NNs in Keras

- There are a few subclasses of the Layers class: e.g., Dense is the one we have seen so far – layers.Dense(), but we also have convolutional layers, max-pooling layers, recurrent layers, and so on. There are many pre-defined layers in Keras. See: https://keras.io/api/layers/.
- These are different architectural components that can be mixed and matched in different ways to create different network topologies.
- It is also possible to construct custom layers.



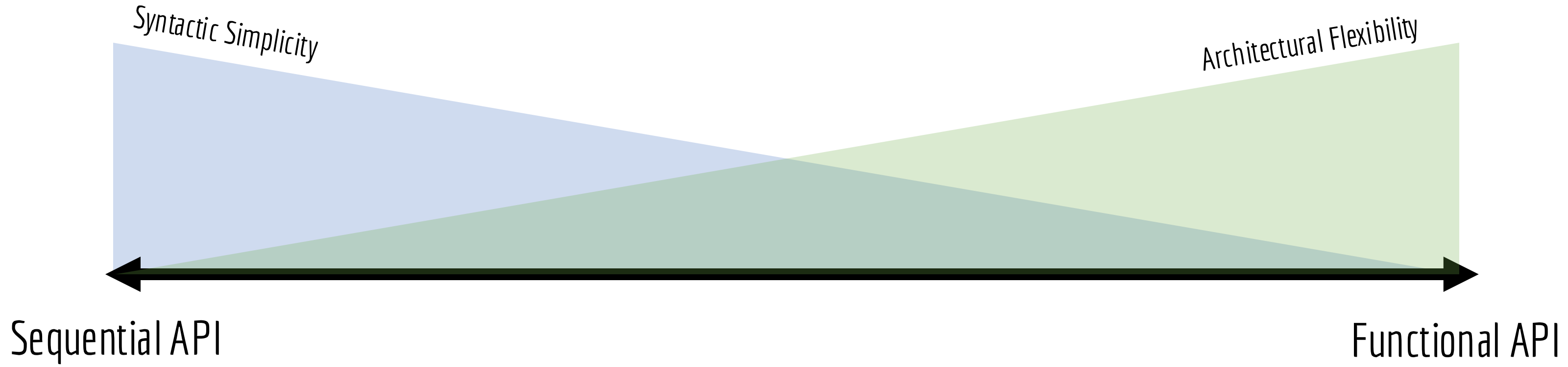
Sequential vs. Functional API

We Have Only Used 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

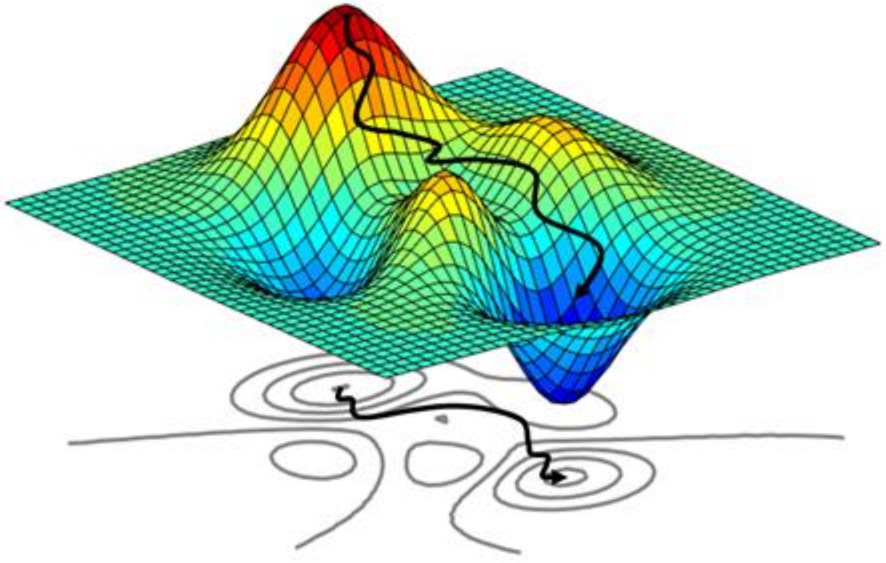
- But – we will look at the difference in how each API is used, syntactically.



Optimizers

Keras Supports 8 Optimizers

- SGD = Stochastic Gradient Descent
- Momentum
- Ftrl (2010) = Follow the Regularized Leader
- Adagrad and Adadelta (2012) = Adaptive Gradient Des
- 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 withe 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).

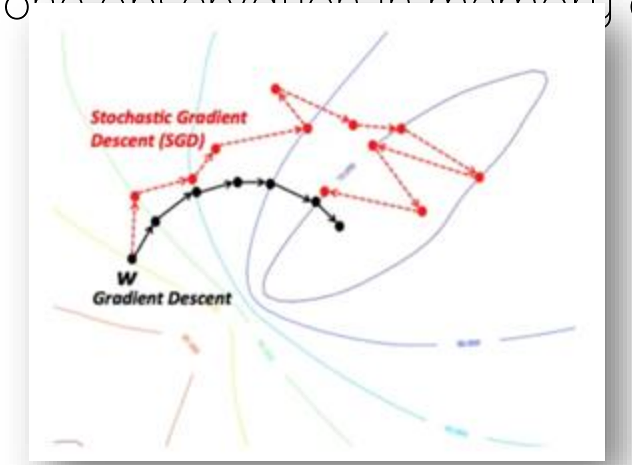
Batch (All) vs. Stochastic (1)

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 descer

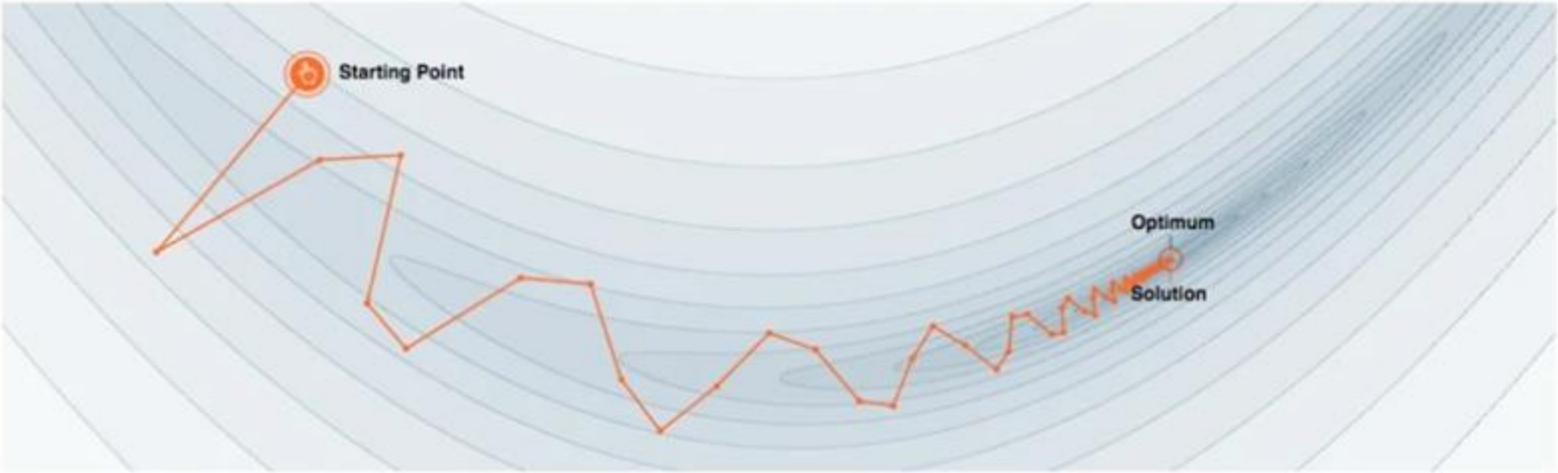


Figure: Optimization with momentum (Source: distill.pub)

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.

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).

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.