Intro to Neural Nets

Week 2: Mathematical Building Blocks &
Working with Keras API

Today's Agenda

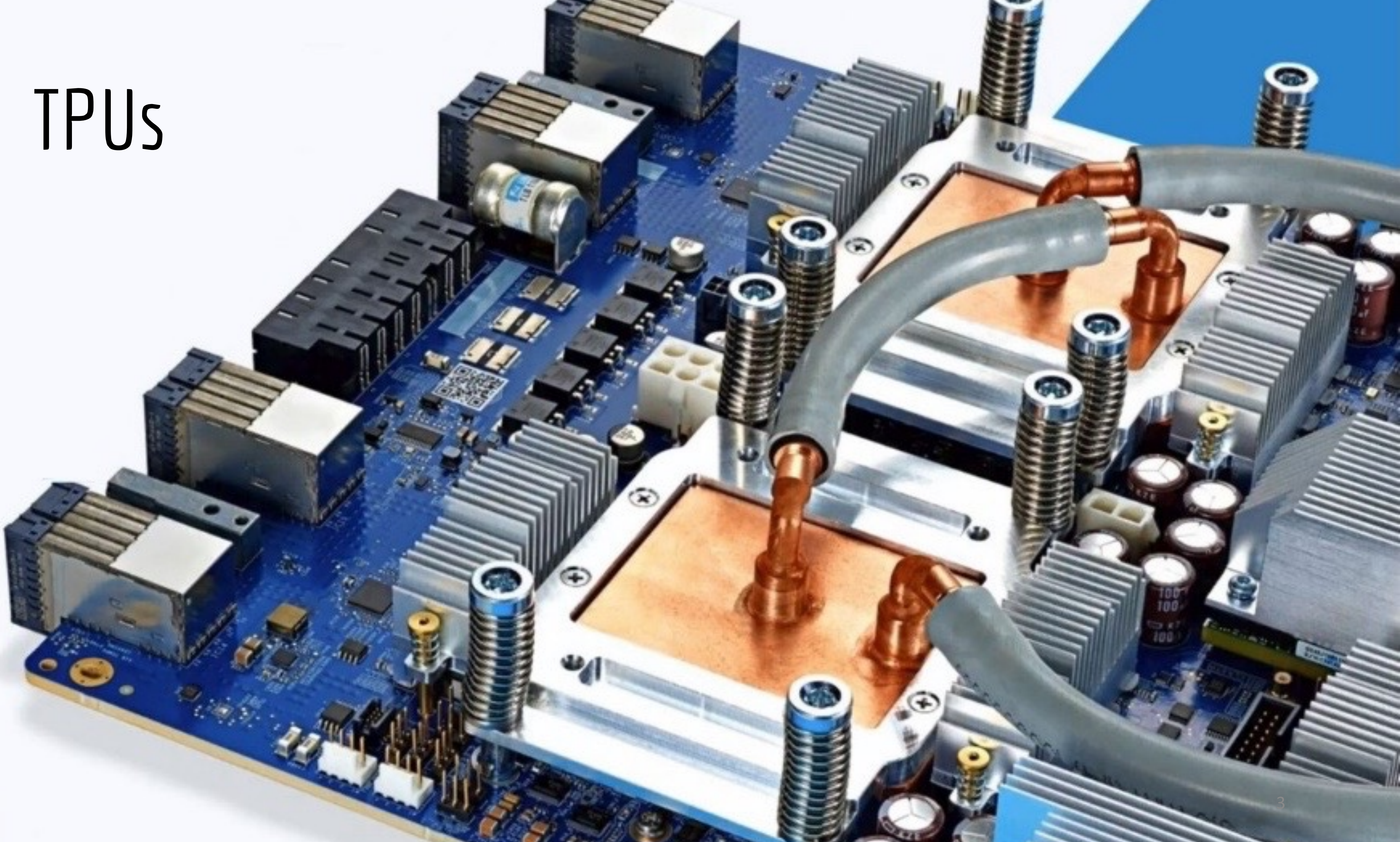
1. Building Blocks of NNs

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

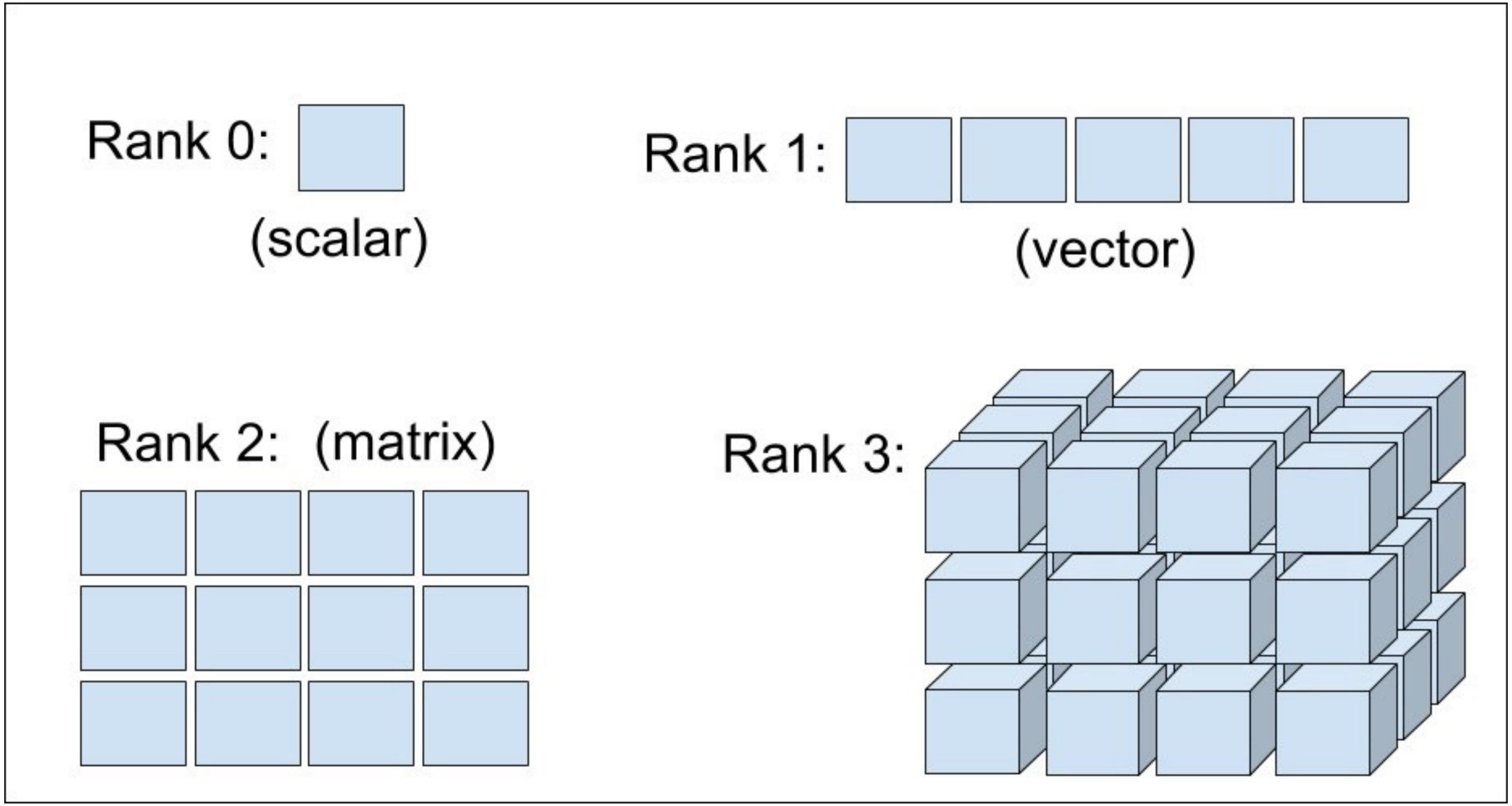
2. Building a Linear Classifier

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

TPUs



Tensors



Neuron / Network Components

X and Y are data

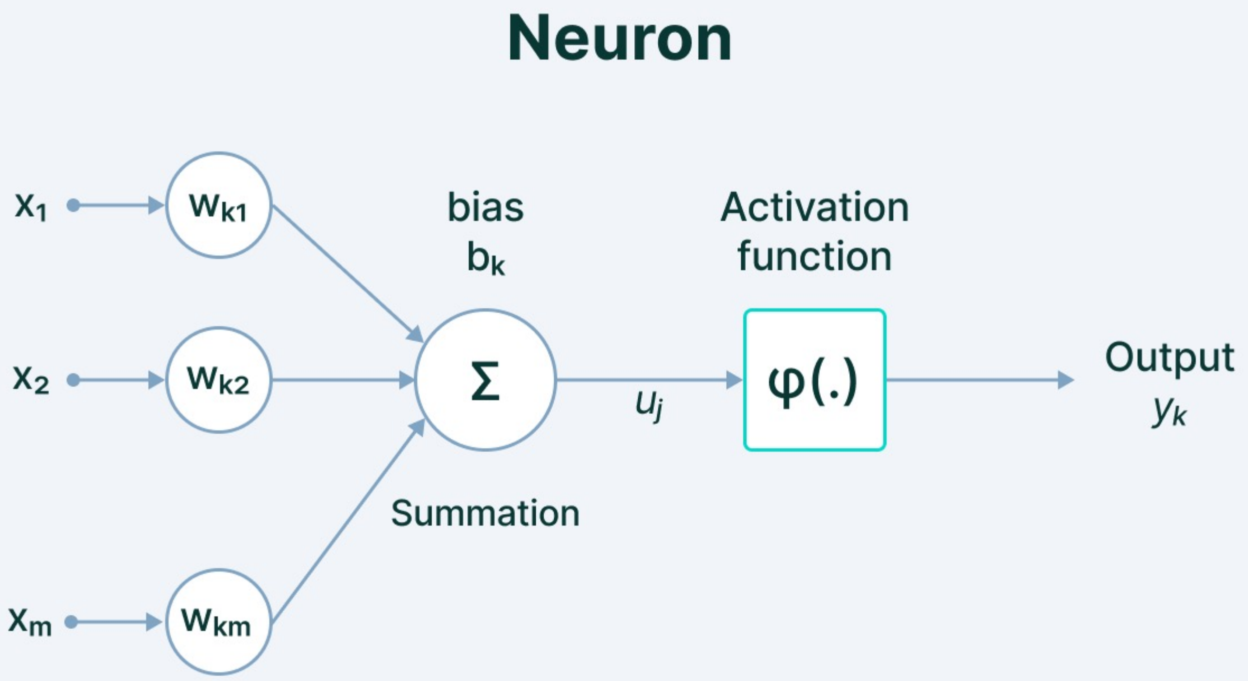
- These are input values and labels.

W and b are parameters

- These are values we 'learn' through the optimization process.

$\varphi(.)$ is a function we choose

- This is a hyper-parameter.



Neuron / Network Components

***x* and *w* are vectors for a node**

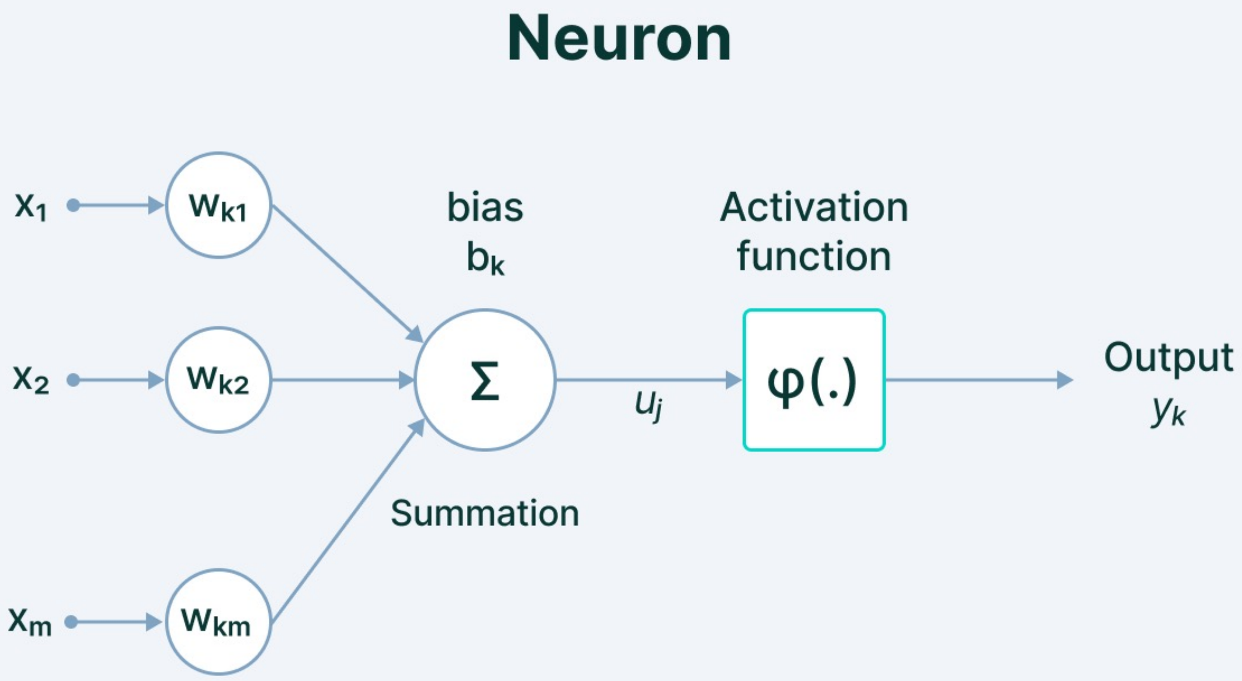
- These vectors comprise matrices that represent a layer of nodes in the network.

***b* and *y* are scalars for a node**

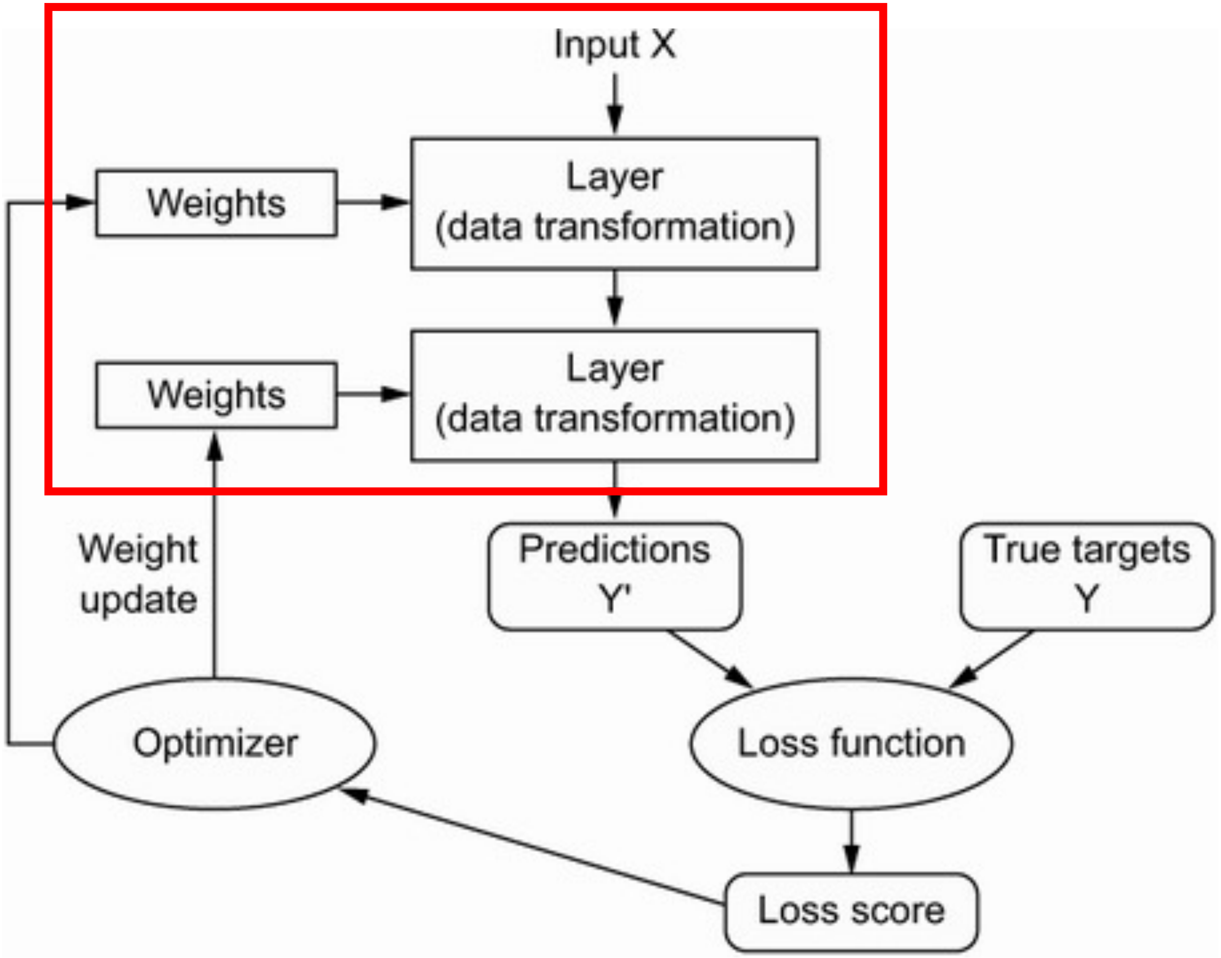
- The set of all b's in a layer becomes a vector.

Nature of output, i.e., *y*

- Depends on position in the network, and what we are predicting



Forward Pass



Multiplication

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

Conformity of Shapes

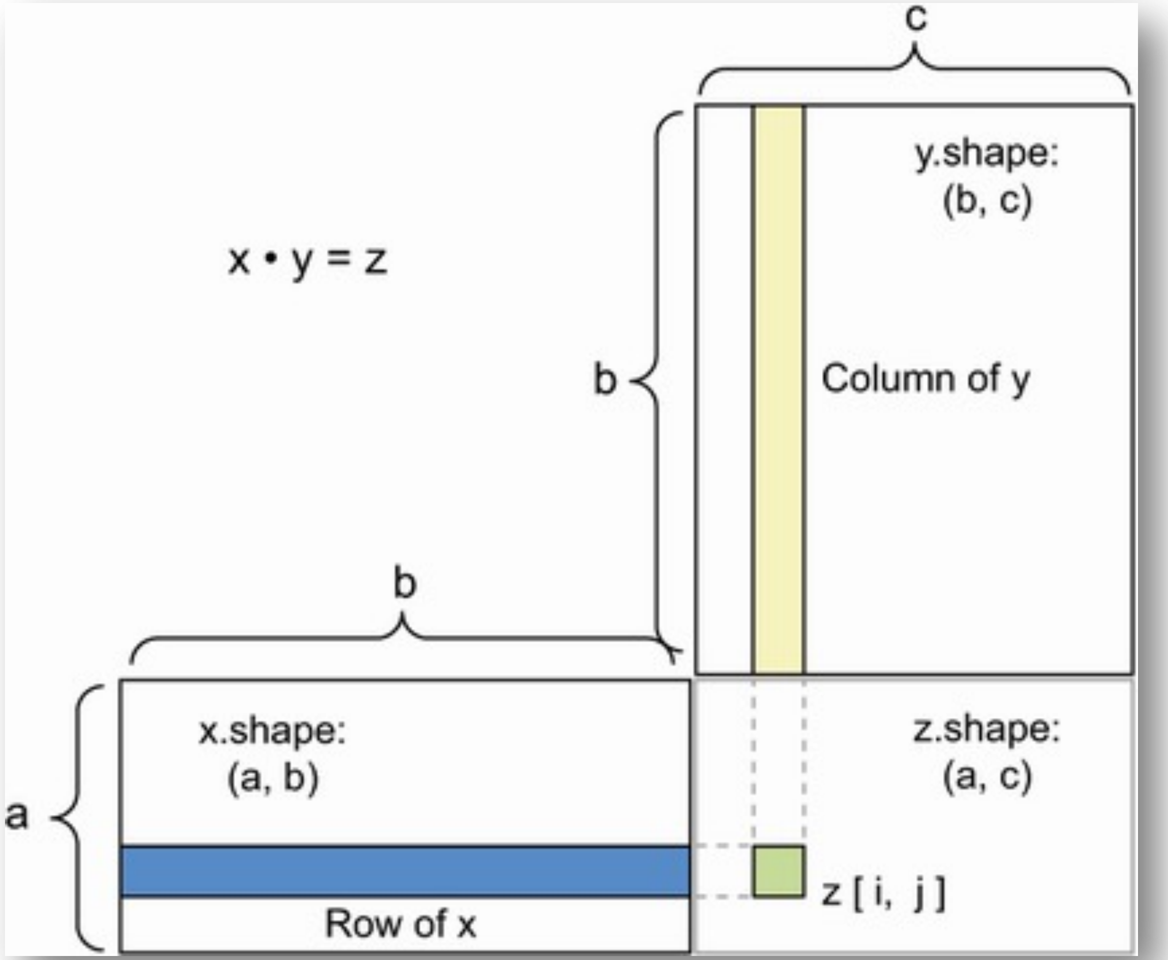
- NCOL(X) == NROW(W)

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.



Addition + Broadcast

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

Shape of the Two Tensors Needs to Conform

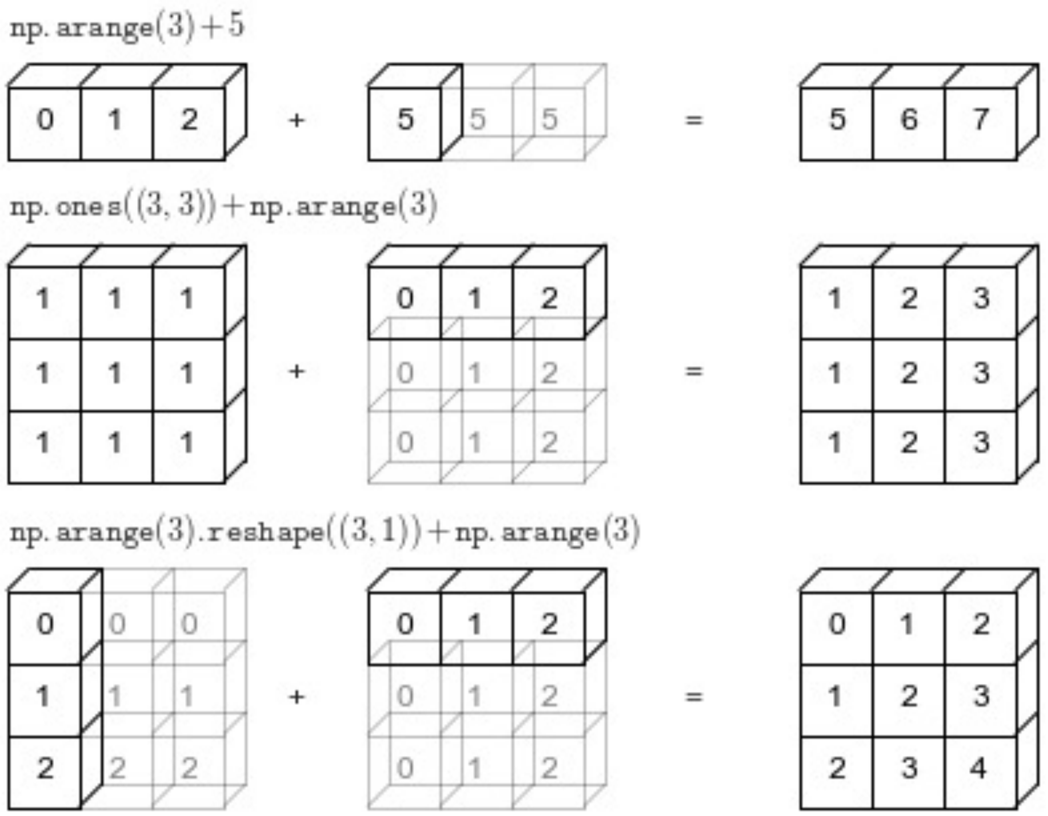
- A + B will only work if A is cleanly divisible by B (or vice versa)

Sum the Element-wise Products

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

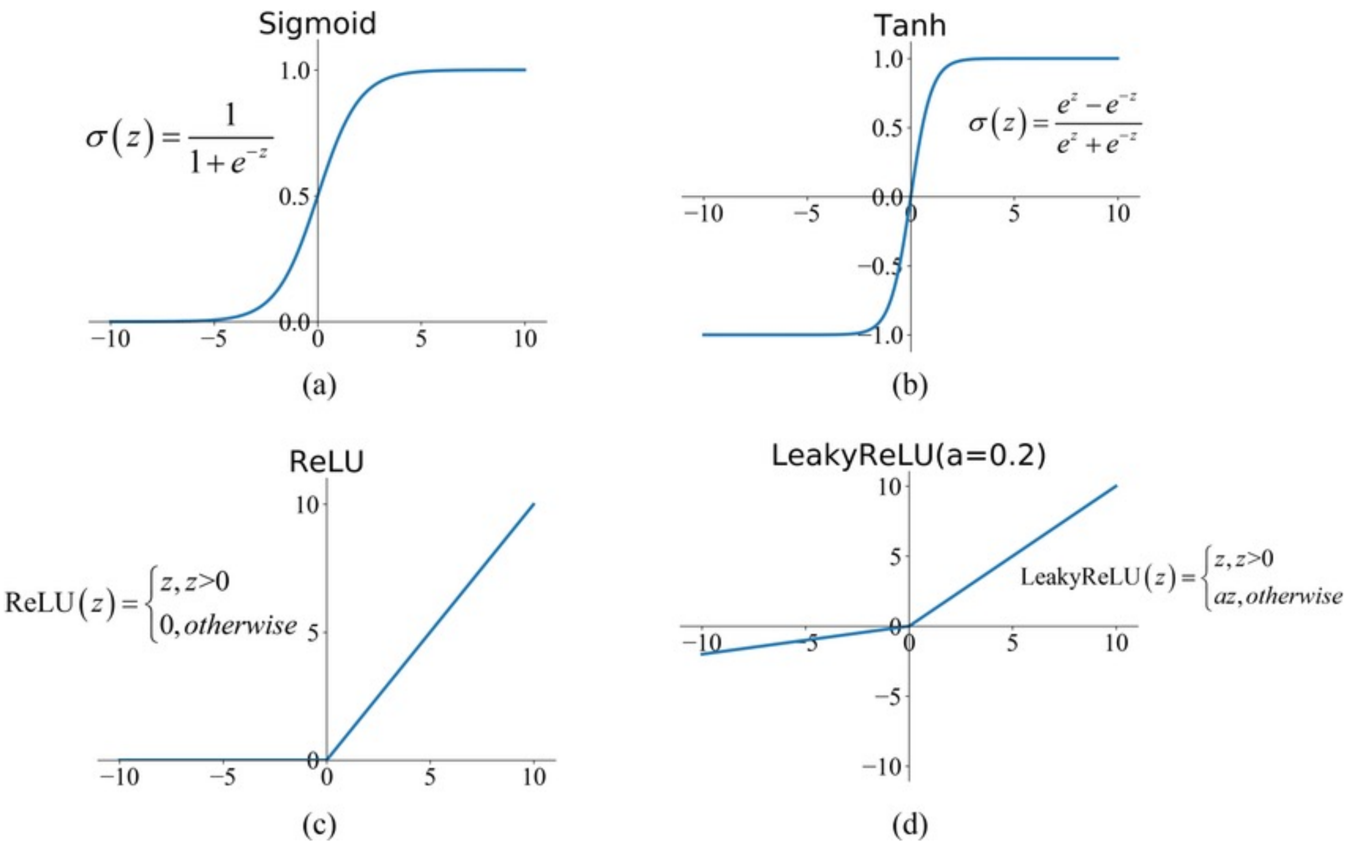
We Use This for the Addition Step

- Add x*w and b (bias)



Activation Functions

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



We Know Enough for a Forward Pass

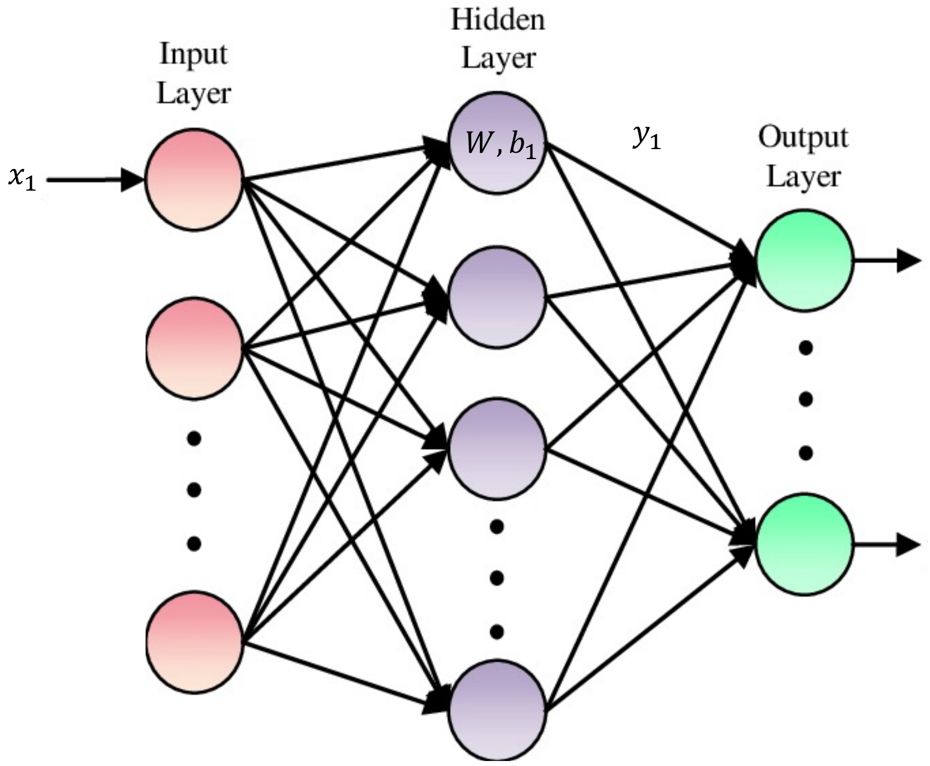
Calculate Output of Each Node Sequentially

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

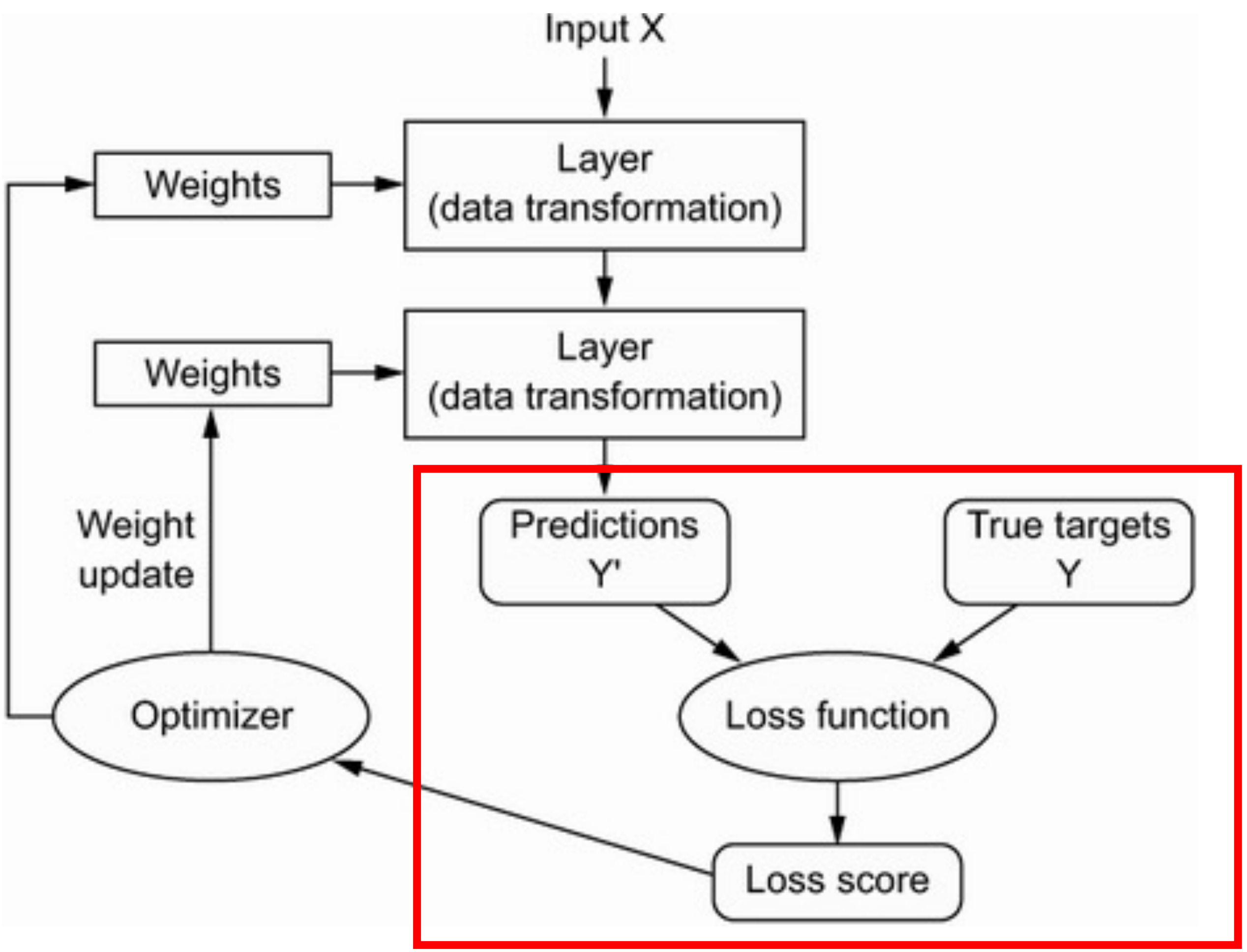
$$y_2 = \varphi\ \left(x_1 \cdot w_{2,1} + x_2 \cdot w_{2,2} + \cdots + b_2\right)$$

…

Eventually We Obtain Model's Predictions



Calculate Loss



Loss Functions

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 multi-class cross-entropy.

MAE / L1 Loss

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

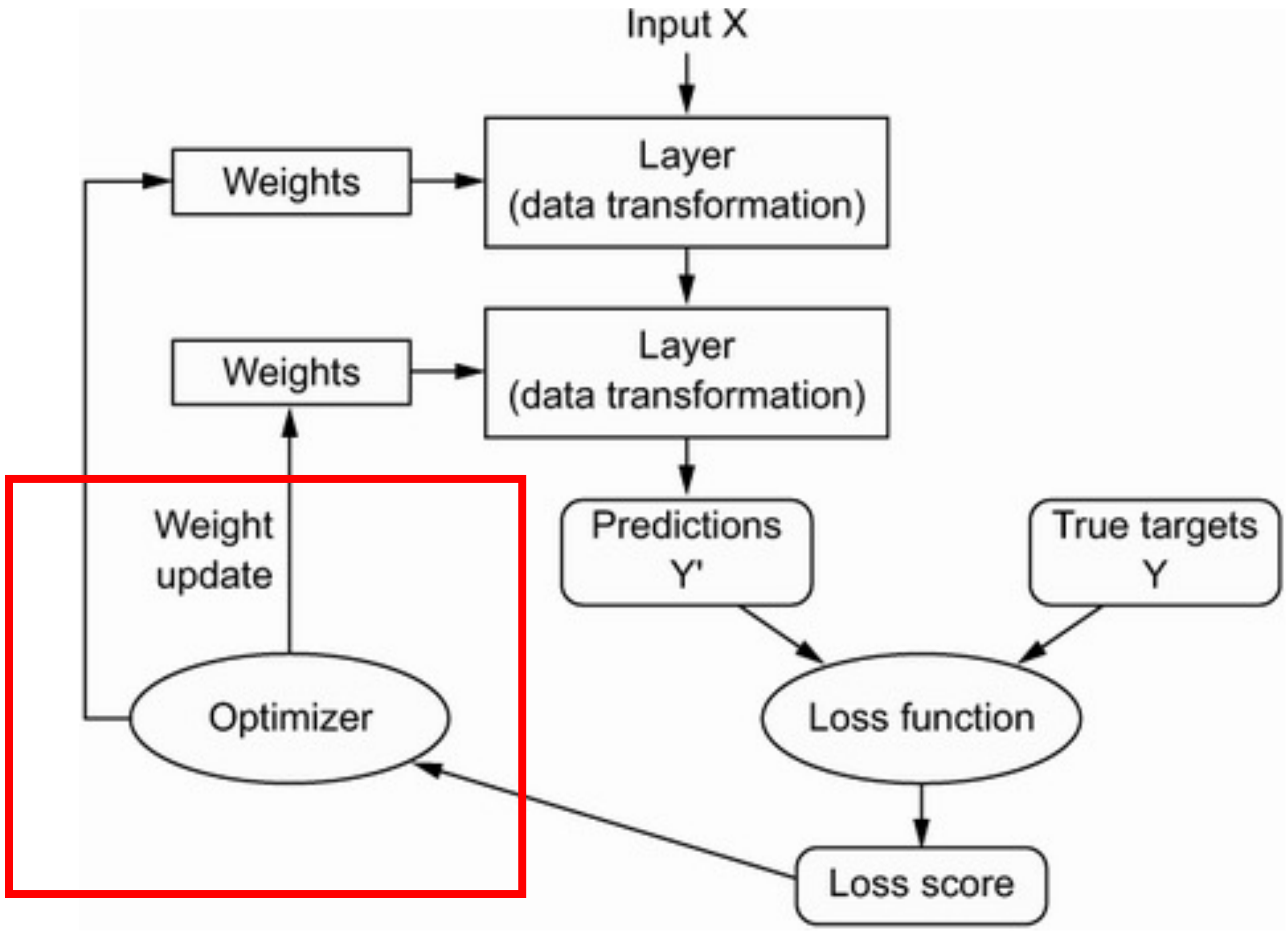
- Typical for continuous outcomes. Errors are penalized homogenously, in magnitude and direction. This should look familiar!

MSE / Quadratic / L2 Loss

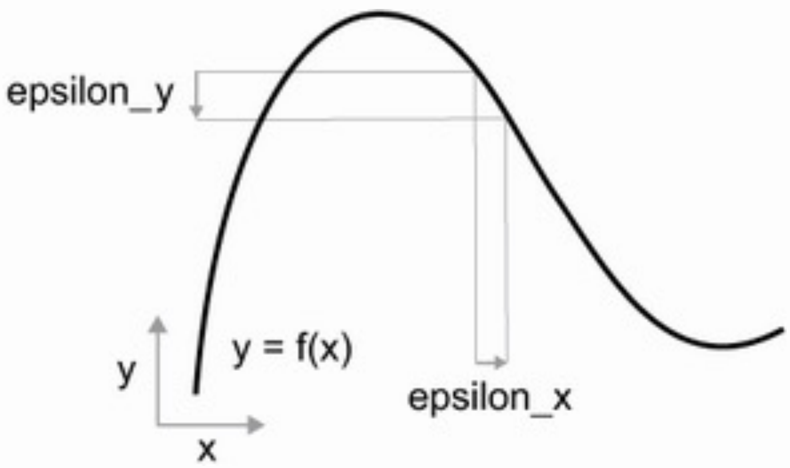
$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}$$

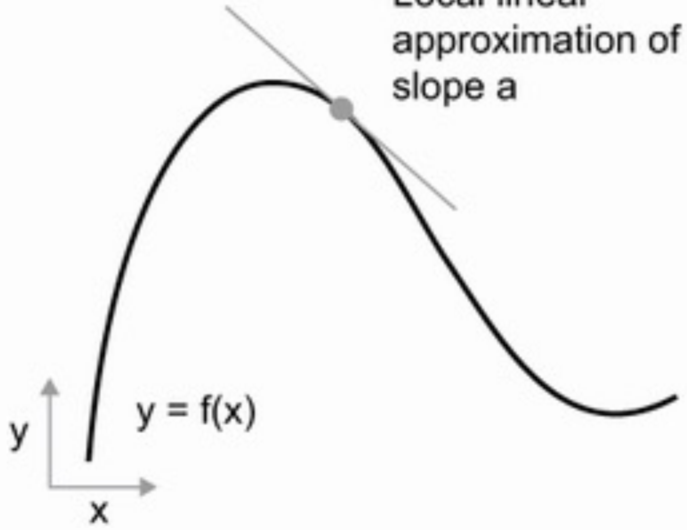
- Typical for continuous outcomes, larger errors penalized exponentially more. This should look familiar!

Backpropagation

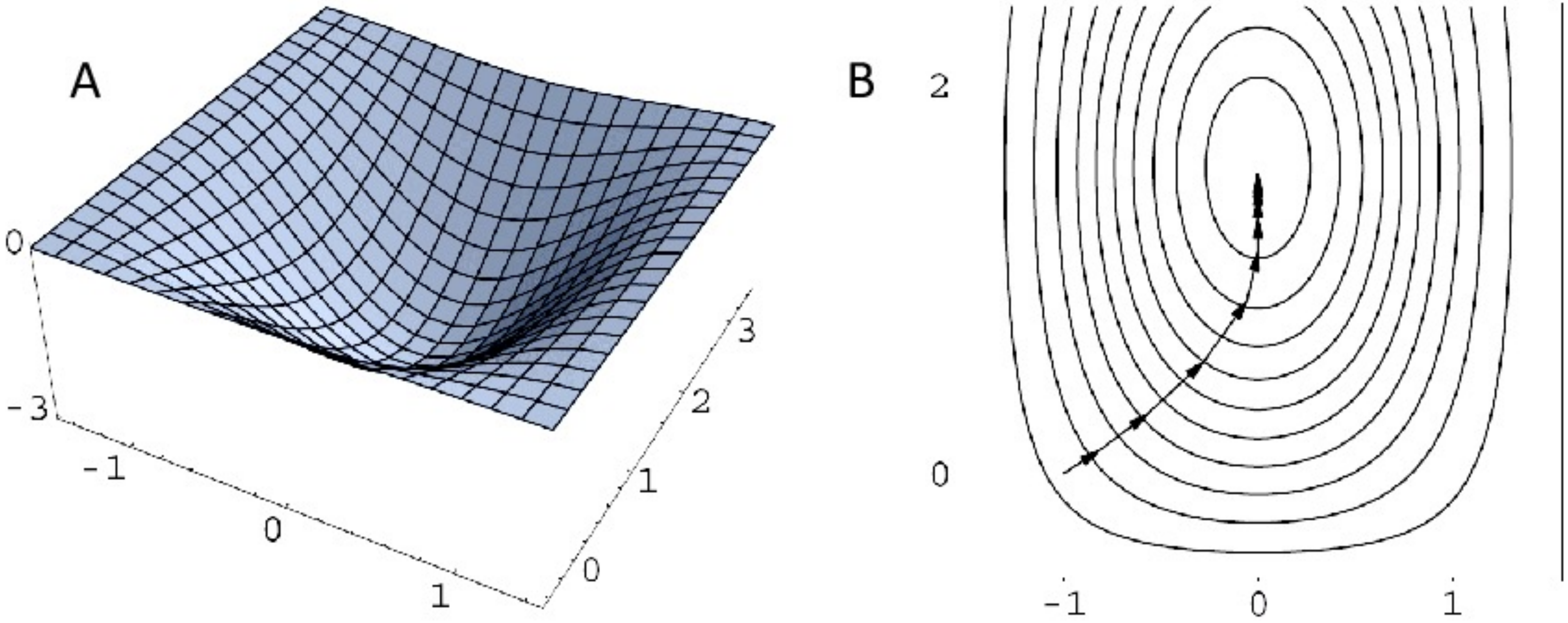


Derivative = “Rate” of Change

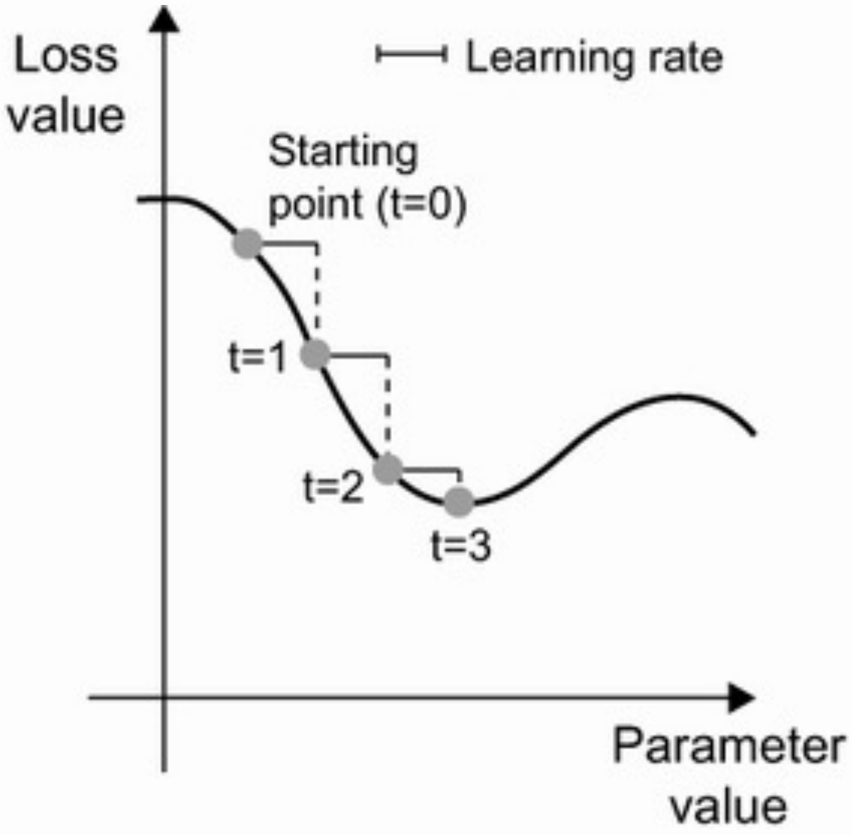




Gradient = Derivative in Multiple Dimensions



Gradient Descent



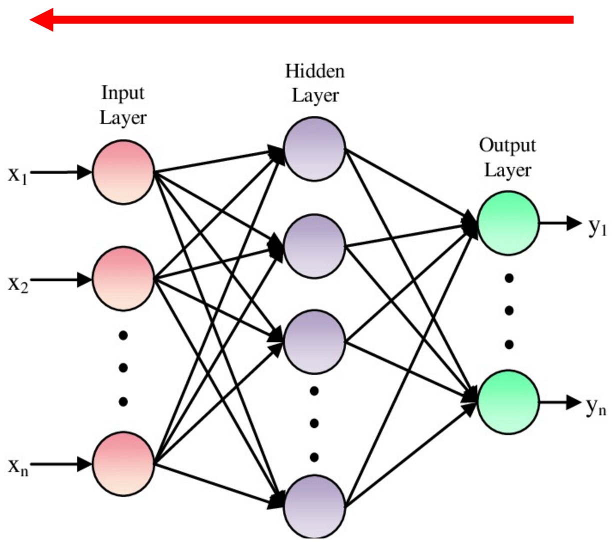
Derivatives of Loss w.r.t All Parameters

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

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

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

…

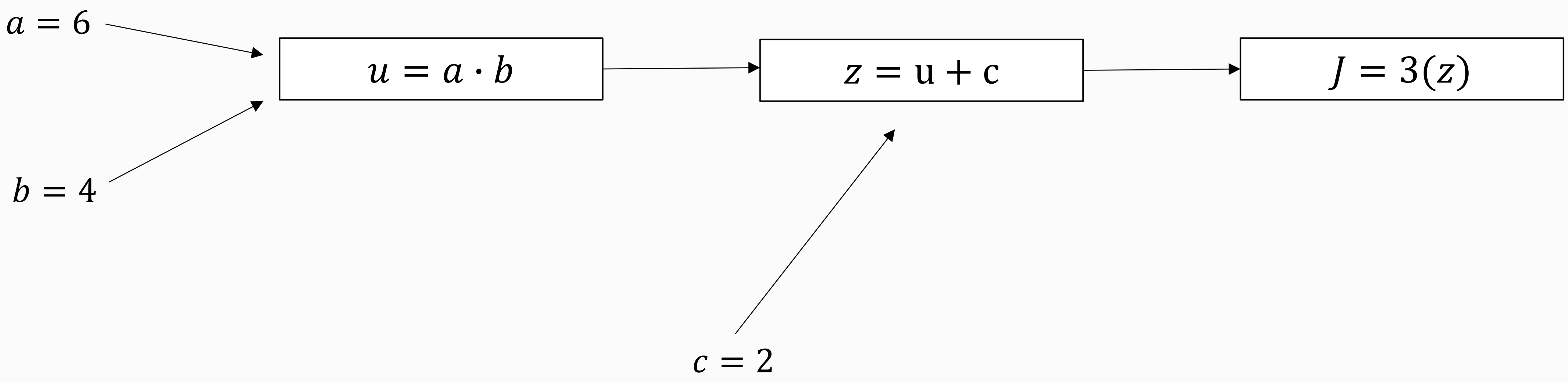


Start at the final nodes in the network and work backwards

- 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.

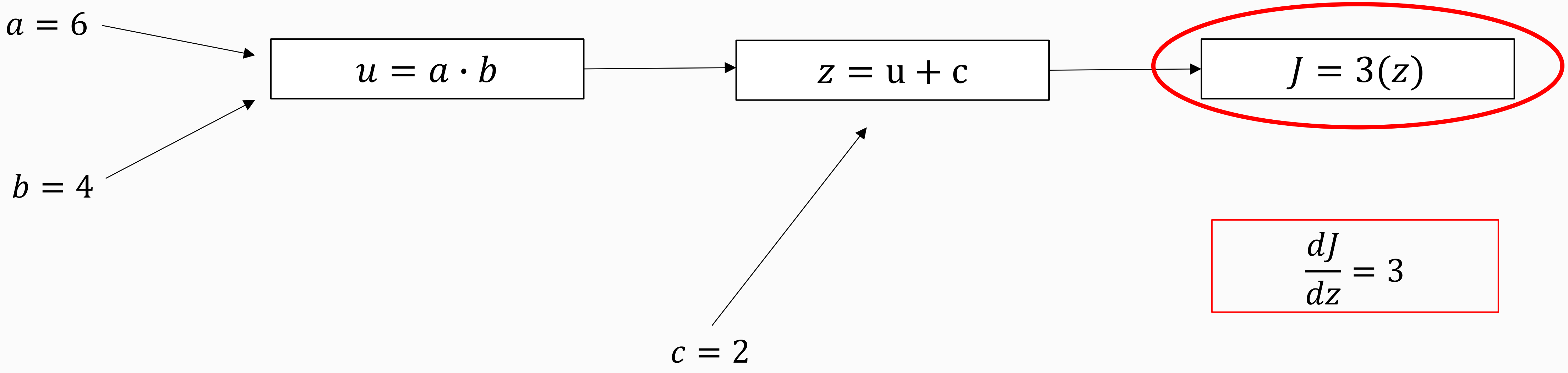
Computation Graphs

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



Backpropagation = Work Backwards

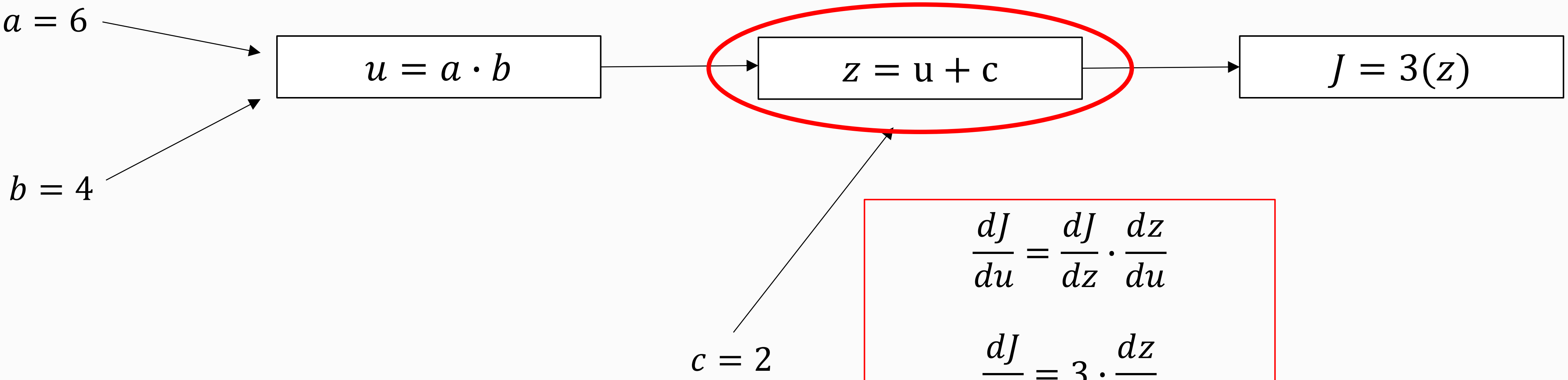
$J = 3(a \cdot b + c)$



Backpropagation = Work Backwards

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

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



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

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

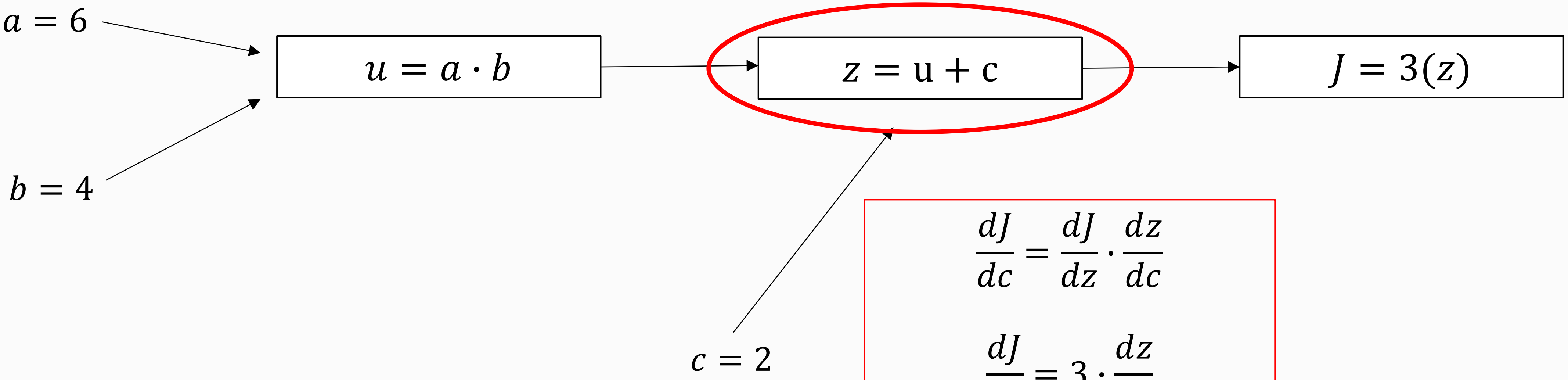
$$\frac{dJ}{du} = 3 \cdot 1 = 3$$

Backpropagation = Work Backwards

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

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

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



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

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

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

Backpropagation = Work Backwards

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

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

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

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

$a = 6$

$b = 4$

$$u = a \cdot b$$

$$z = \mathrm{u} + \mathrm{c}$$

$$J = 3(z)$$

$c = 2$

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

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

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

Backpropagation = Work Backwards

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

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

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

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

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

$a = 6$

$b = 4$

$$u = a \cdot b$$

$$z = \mathrm{u} + \mathrm{c}$$

$$J = 3(z)$$

$c = 2$

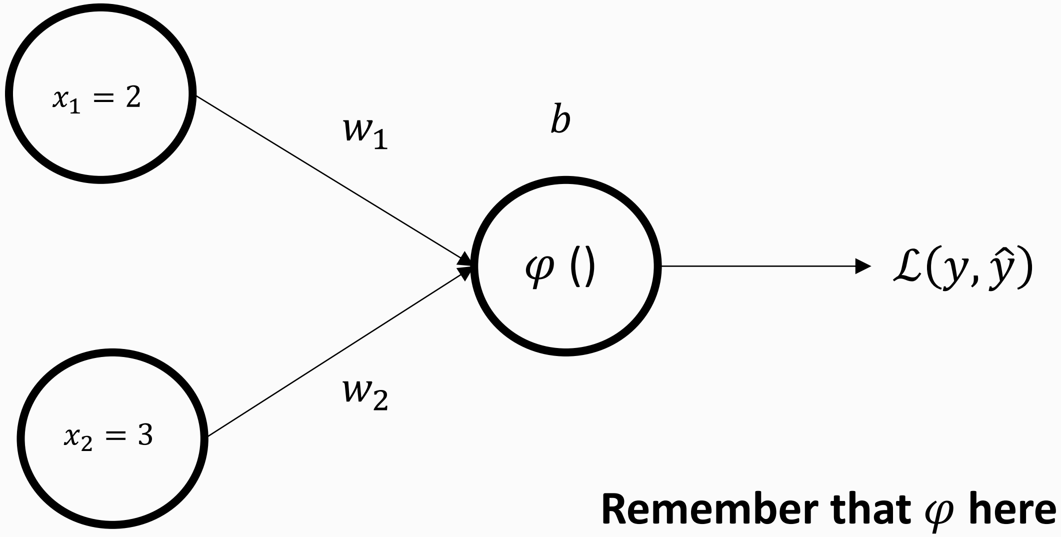
$$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db}$$

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

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

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)



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

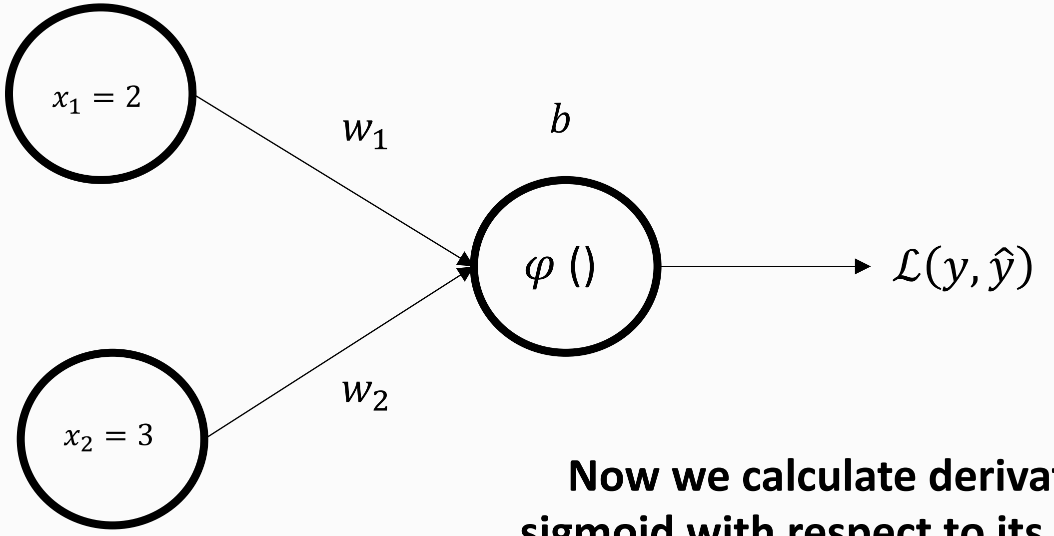
$$\frac{d\mathcal{L}}{d\varphi} = \frac{\varphi(1-y) - y(1-\varphi)}{\varphi(1-\varphi)}$$

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

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

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., $\boldsymbol{\varphi}$(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.

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

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

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

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

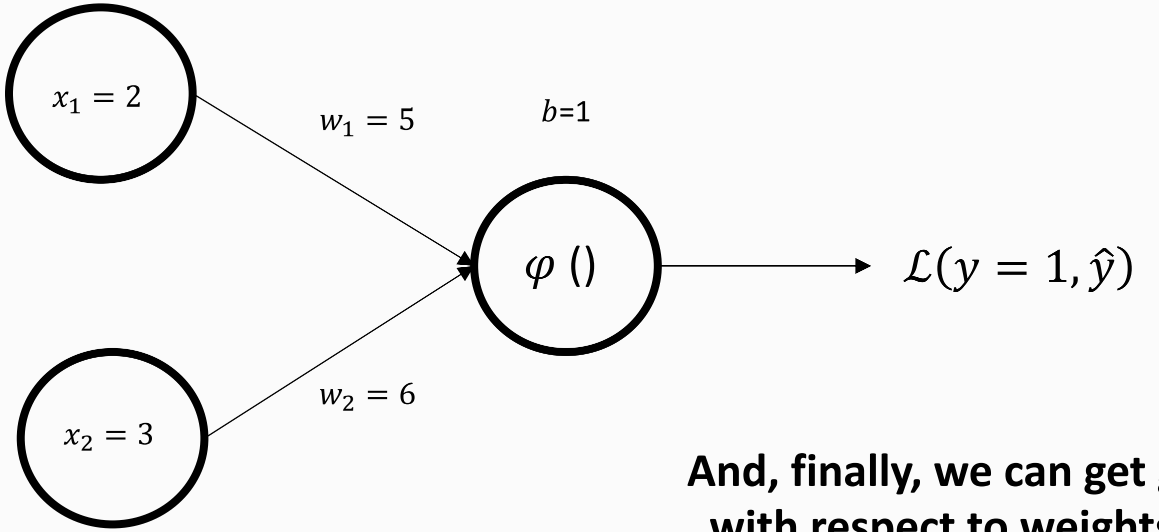
$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{d\varphi} \cdot \frac{d\varphi}{dz}$$

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

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

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

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



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

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

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

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…

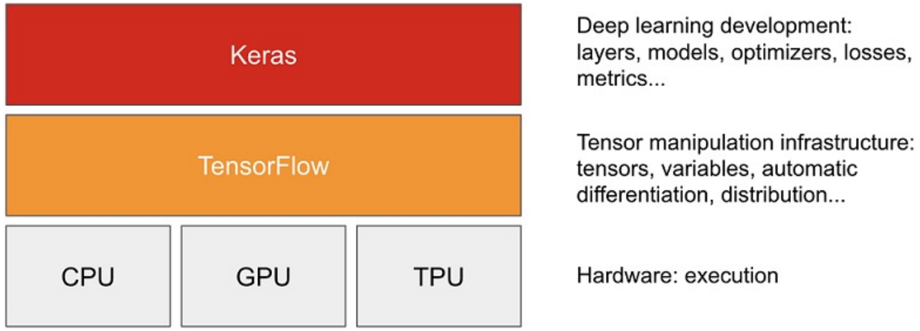
Keras and Tensorflow

1. Tensorflow

- A Python platform for working with tensors, implementing automatic differentiation, providing access to repositories of (well-known) pre-trained models.

2. Keras

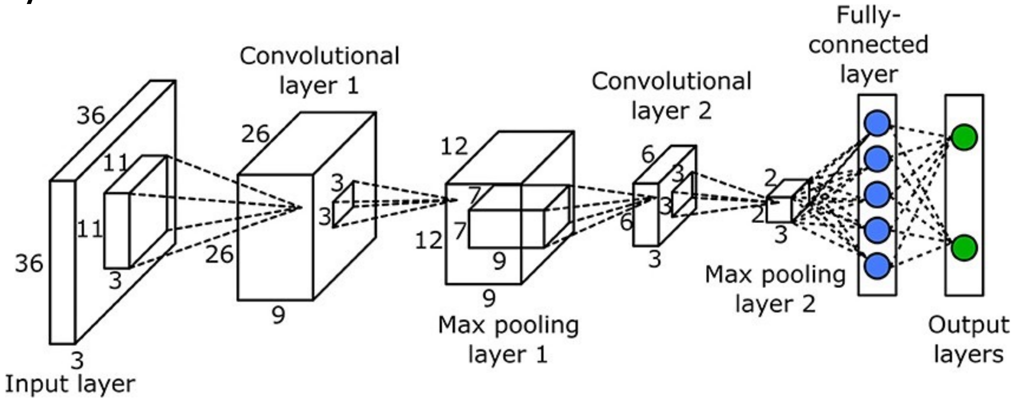
- A higher-level API that wraps common usage patterns with Tensorflow functions, pre-defined loss functions, optimization algorithms, etc.
- Keras simplifies data scientists’ interaction with Tensorflow.

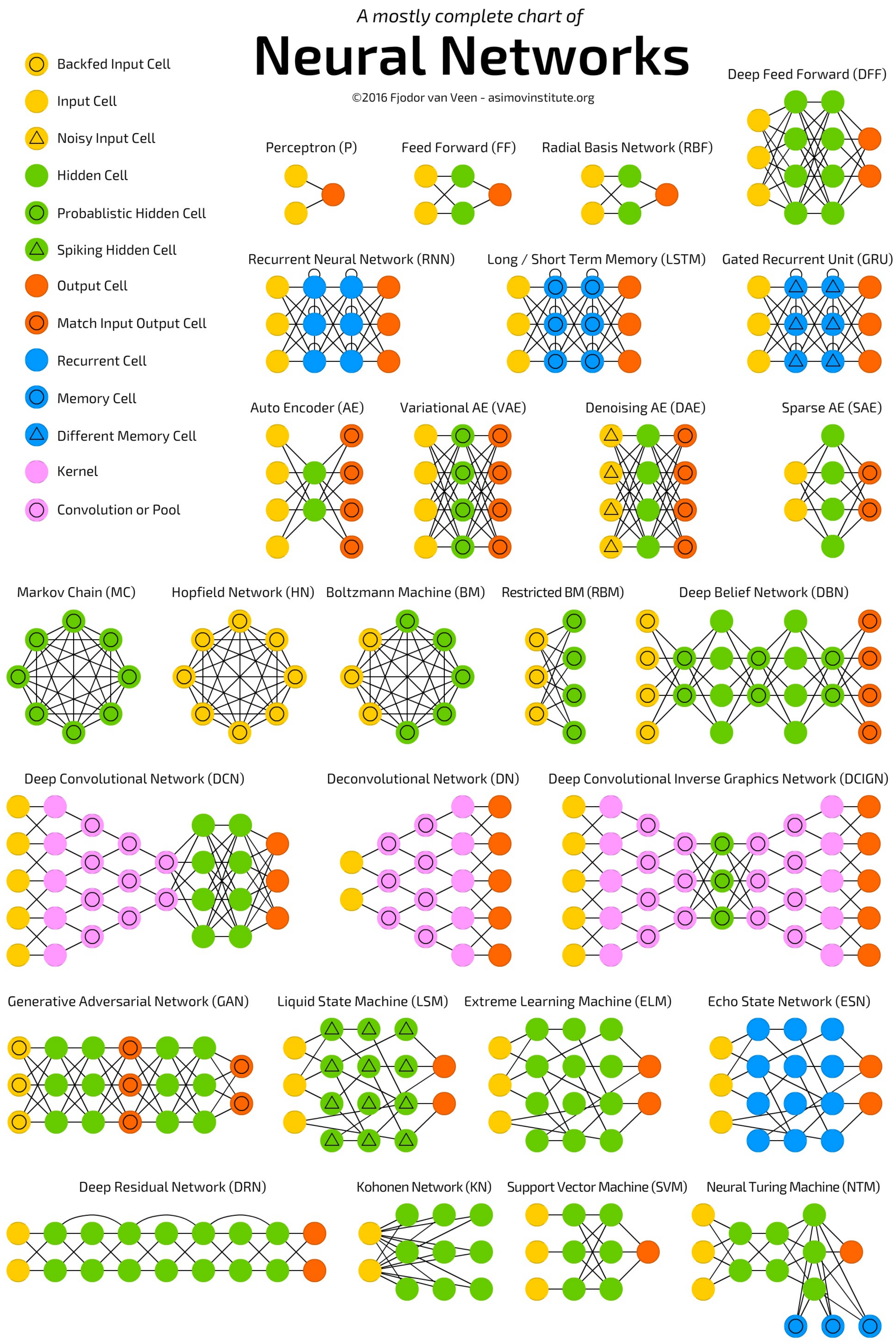


The Layer

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().
- There are many more, though. See: https://keras.io/api/layers/.
- Most notably, there are pre-processing layers, convolutional layers, attention layers, etc.
- These are different architectural components that can be mixed and matched to create different network topologies.
- It's possible to make custom layers, as is shown in the book, e.g., "SimpleDense()", but we won't need this immediately.




A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Deep Feed Forward (DFF)
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

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.