Lecture 02

Introduction to Neural Networks
Keras, an API for Deep Learning

cscis-89A Deep Learning for NLP

Summer 2019

Zoran B. Djordjević

Objectives

- Relevance of Neural Networks
- Biological Motivations
- Perceptor Model
- XOR problem briefly
- Loss Functions
- Back propagation
- Keras
- Simple regression model with Keras: Pima Indians Health Statistics
- Binary Classification: Movie Reviews, IMDB Dataset

References

- *Deep Learning with Pyt*hon by François Chollet,2018, Manning Publications Co., Chapter 3.
- *Deep Learning in Python* by Jason Brownlee, 2017, Machine Learning Mastery

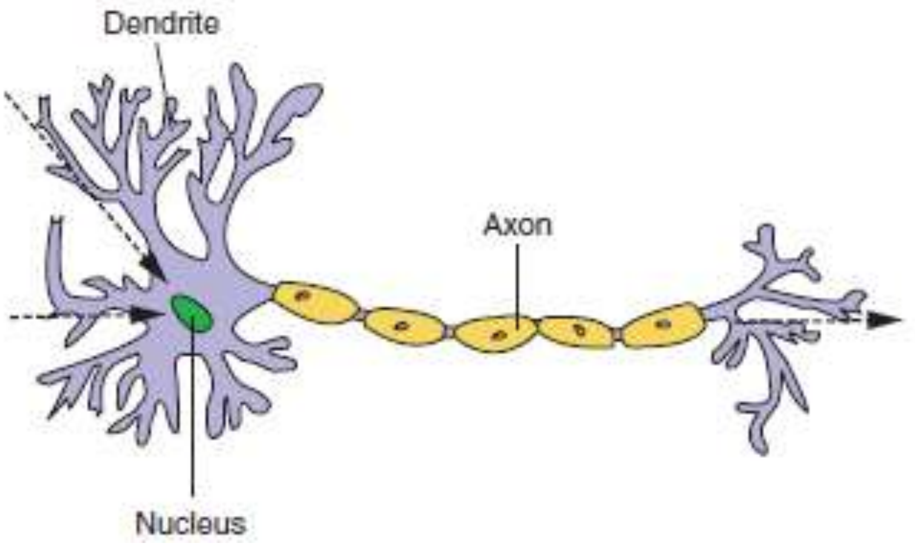
Deep Learning in NLP and Everything Else

- Neural Networks (NN) and their most recent incarnation Deep Learning (DL) Networks demonstrated ability to recognize and classify data of various form, transform speech into text, translate text and speech from one language to another and generate original textual and speech content.
- Deep Learning technology is also extensively used in areas where text and speech processing are not the primary concern. For example, DL is successfully used for image captioning, self-driving car navigation, recognition of objects in the environment, recognition of faces in security camera videos, impersonation of voices of specific individuals, and many other application.
- Deep neural networks are very powerful practical tools, and we are using them in many existing and most successful commercial applications and pipelines.
- Amazon Alexa, Apple Siri, Google Translate are all commercial products based Natural Language Processing ideas and Deep Learning technology.
- In what follows, we will introduce the basic neural network and deep learning concepts and soon after start applying those concepts in practical use case.

Biologic Motivations

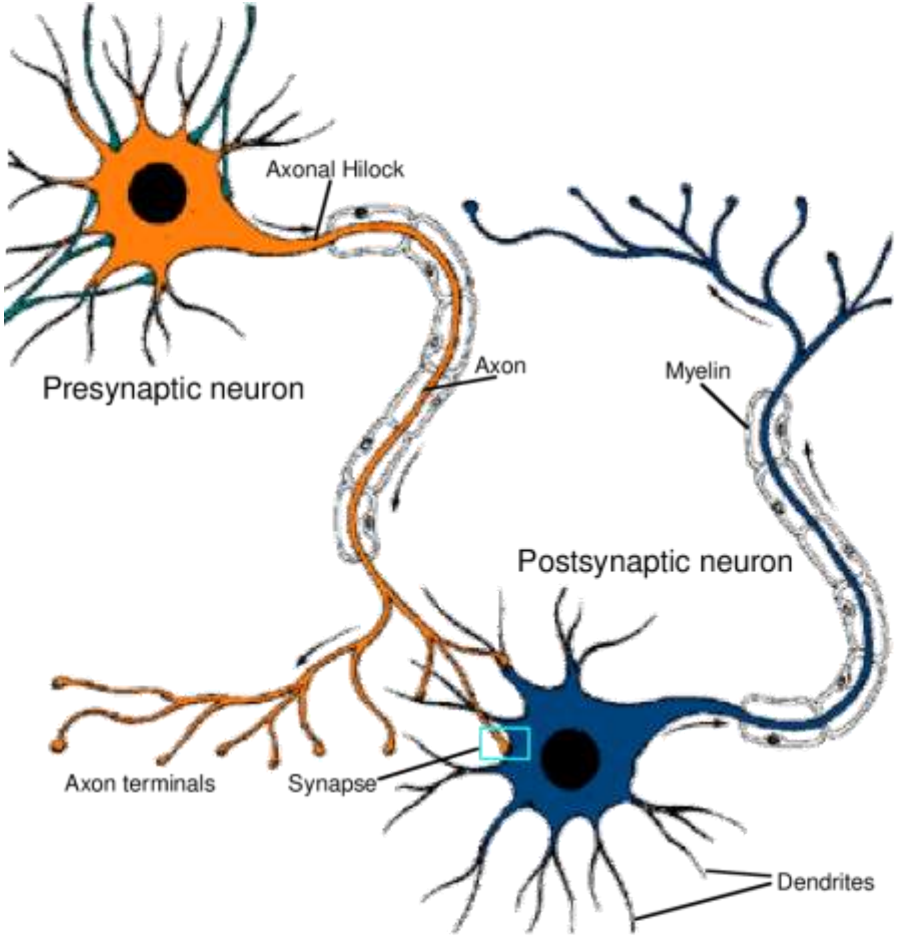
- Many computer scientist working on artificial neural networks find their motivation in the field of neuroscience.
- Cells in our brain are called neurons. As electrical signals flow through the *dendrites* into the nucleus, electric charge builds up. When the cell reaches a certain level of charge , a threshold, it *fires*, sending an electrical signal out through the *axon*. Axon terminals of any one neuron connect to dendrites of other neurons.
- The dendrites are not all created equal. The neuron could be more "sensitive" to signals through certain dendrites than others. It takes weaker signals in those paths to fire the axon.

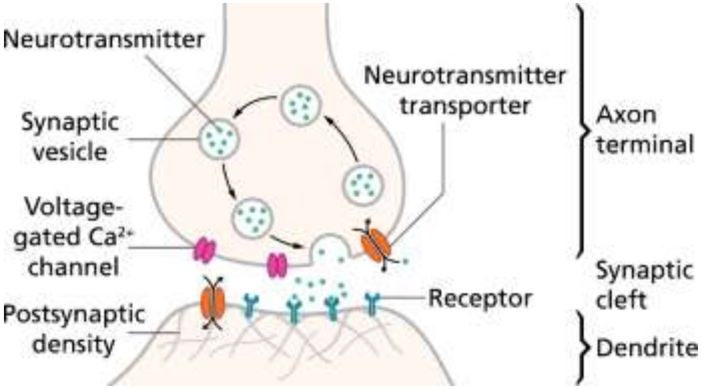
- This is a key observation neurologists made. Neurons *weight* different incoming signals differently when deciding when to fire.
- Every neuron will dynamically change those weights in the decision making process over the course of its life.
- Artificial neural networks (ANN) will mimic the process of assigning different weights to different dendrites (inputs).
- ANNs are usually static. They freeze their weights, once they determine them.



We need Network of Neurons to Think

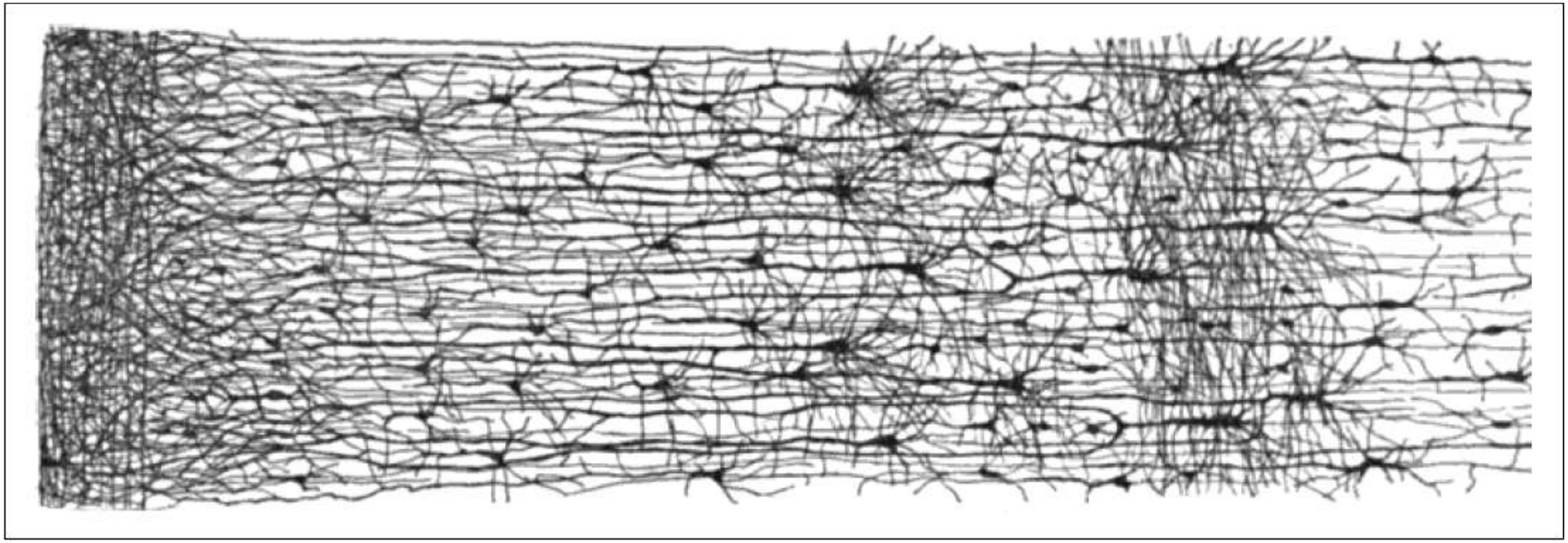
- Our brain is made of 100 billion neurons (10^{11}).
- Axon terminals establish connections with dendrites of other neurons. Those connections are call synapses.
- Each neuron has on average 7,000 **synaptic** connections to other neurons. It has been estimated that the **brain** of a three-year-old child has about 10^{15} **synapses.**
- Synapses appear to be semi-permanent or permanent and their creation involves chemical reactions.
- We remember things by creating synaptic networks of neurons.
- We think by generating waves through those networks.





One Learning Algorithm & Layers Hypothesis

- Results of many neurological studies suggest that brain is not made of very different structures serving different purposes but is rather made of one and the same basic unit, the neuron, which could adjust and perform different functions.
- Neurons appear to be organized in layers.
- A Spanish scientist Santiago Ramón y Cajal made extensive work & discoveries on neural structures in late 1800-s. Image below is from "*Texture of the Nervous System of Man and the Vertebrates*" by Santiago Ramón y Cajal (1899–1904). https://www.nytimes.com/2017/02/17/science/santiago-ramon-y-cajal-beautiful-brain.html
- This image suggests that neurons do appear organized in loose layers.



Rosenblatt’s Perceptor

- In 1958, at Cornell’s Aeronautical Laboratory, Frank Rosenblatt was working on a US Navy funded project with the objective to create a machine which could recognize images.
- The original perceptron was a conglomeration of photo-receptors and potentiometers, not a computer. This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.
- In a 1958 press conference organized by the US Navy, Rosenblatt made statements about the perceptron that caused a heated controversy among the fledgling AI community.
- Based on Rosenblatt's statements, *The New York Times* reported the perceptron to be "the embryo of an electronic computer that the Navy expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.“
- From today’s perspective, Rosenblatt and New York Times were right.

The first and other Ice Ages

- Although the perceptron initially seemed promising, it was quickly proven that single perceptron could not be trained to recognize many classes of patterns.
- This caused the field of neural network research to stagnate for many years.
- In 1969 a book entitled *Perceptrons* by Marvin Minsky and Seymour Papert showed that it was impossible for single perceptron to learn an XOR function.
- Single layer perceptrons are only capable of learning linearly separable patterns. At the time, many assumed (incorrectly) that Minsky and Papert also conjectured that a similar result would hold for a multi-layer perceptron network.
- This is not true, as both Minsky and Papert already knew. Multi-layer perceptrons were capable of producing an XOR and other logical functions. Nevertheless, the often-miscited Minsky/Papert text caused a significant decline in the interest and funding of neural network research.
- It took more than ten years until the artificial neural network research experienced a resurgence in the 1980s. The progress continued, though the public enthusiasm for the neural networks was raising and falling periodically.
- In 2012, the Deep Learning was "born" when a paper by the University of Toronto team (Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton) demonstrated that a deep convolutional neural network could perform image classification task with a precision considerably higher than achievable with other technique used at the time.

Formal Perceptron

- Inputs in most of experiments are represented by many (n) features. The individual features are denoted as x_i, where i is a reference integer. Those features could be organized as a vector X, which we represent as:

  $$X = [x_1, x_2, \dots, x_i, \dots, x_n]$$

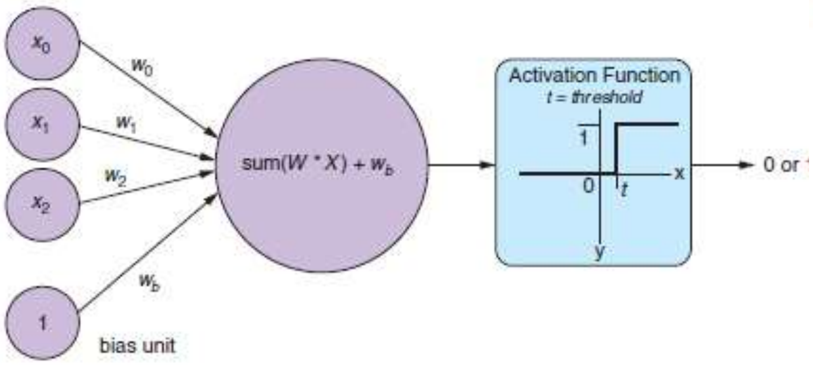
- similarly, you'll see the associate weights for each feature as w_i, where i corresponds to the index of feature x associated with that weight. The weights are generally represented as a vector W:

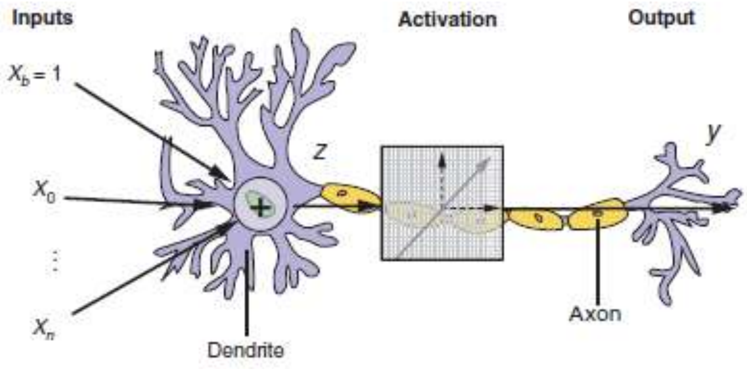
  $$W = [w_1, w_2, \dots, w_i, \dots, w_n]$$

- Neuron multiplies each feature (x_i) by the corresponding weight (w_i) and then sums those up:

  $$(x_1 \cdot w_1) + (x_2 \cdot w_2) + \dots + (x_i \cdot w_i) + \dots + (x_n \cdot w_n)$$

- Once the weighted sum exceeds a certain threshold, the perceptron outputs 1. Otherwise it outputs 0.





Unit Step Function

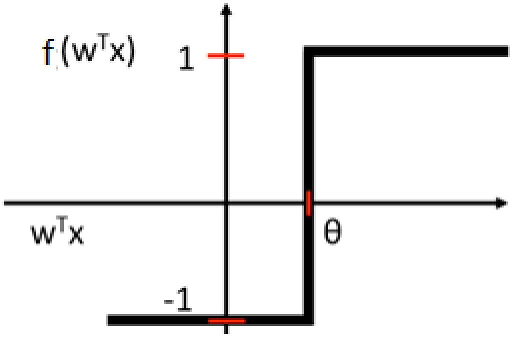
- This function used in original perceptron is called *threshold activation function*, *unit step function* or *Heaviside* function.
- If we are performing binary classification, we usually label the *positive* and *negative* class in our binary classification as "1" and "-1", or “1” and “0”, respectively.
- The activation function f(z) takes a linear combination of the input values x and weights w as input ($z = w_1x_1 + \cdots + w_mx_m$).
- If $f(z)$ is greater than a defined threshold θ, we predict 1 and -1 otherwise.

$$f(z) = 1 \; if \; z \geq \; \theta, \qquad -1 \; if \; z < \; \theta$$

- We can express z as a dot product of vectors w and x, $z = \; W^T x$
- W is the vector of weights
- X is an n-dimensional sample

  from the training dataset.
- The simple threshold activation function is

  practically never used.



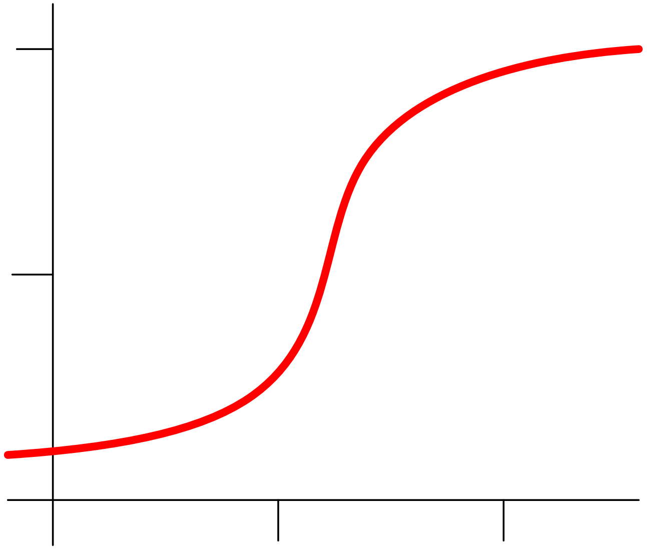
Linear Algebra & Modern Hardware

- The sum of the pairwise multiplications of the input vector (X) and the weight vector (W) is the dot product of the two vectors.
- Many processes inside Deep Learning networks could be expressed as either dot (vector) products or matrix-matrix multiplications. Those are the most basic operations of *Linear Algebra*. This course will not place an emphasis on mathematical details of this technology. Though, we will not hide them either.
- One side benefit of being able to represent processes in DL networks over matrix multiplication is that we can efficiently use: **Graphical Processing Units (GPUs), Field-Programmable Gate Arrays** (**FPGA**) and **Tensor Processing Units (TPUs).**
- **GPUs, FPGAs** and **TPUs** are super efficient at implementing neural networks due to their hardware and software optimization for linear algebra operations.

- In many books articles, the basic computational unit we are describing is persistently called “preceptor”. We will call it “neuron”, meaning an “*artificial* neuron”.

Sigmoid, Nonlinear Activation Function

- Eventually, computer scientist learned that Sigmoid function is much more easier to work with than the step function and it became the most frequently used neuron activation function.



Sigmoid

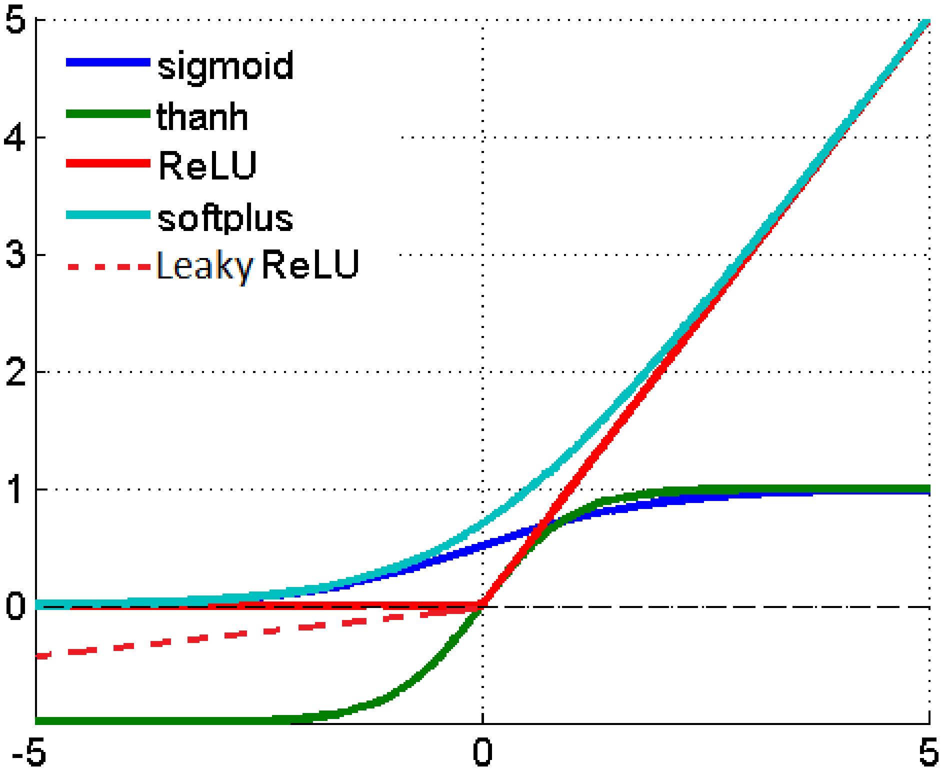
$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$

- Sigmoid function is “continuous” and everywhere “differentiable”. Its derivative is also “continuous” and everywhere differentiable.
- The derivative of the sigmoid function can conveniently be expressed over the function itself.

Activation functions

- Sigmoid, $\sigma(x)$ is not the only activation function. Again, as we will learn and justify, we will use other non-linear activation functions. Rectifier Linear Unite (ReLU) and Leaky ReLU are frequent choices:



$$ReLU(x) \;=\; \max(0, x)$$

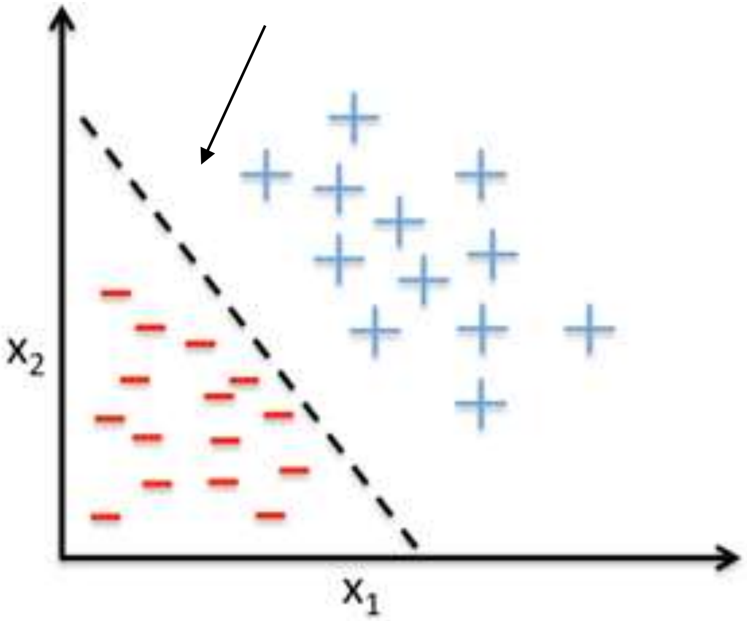
$$Leaky\ ReLU = \max(0, -\alpha x) + \max(0, x)$$

- Sigmod, $\sigma(x)$, has two flat regions left and right of zero and gradient descent propagation will be very, very slow if our inputs lead us to those regions.
- $ReLU(x)$ has a slope of 1 for $x > 0$ and propagation will not be interrupted for x > 0.
- We resort to the $Leaky\ ReLU$ when we cannot control input variables and are likely to end up left of the zero.

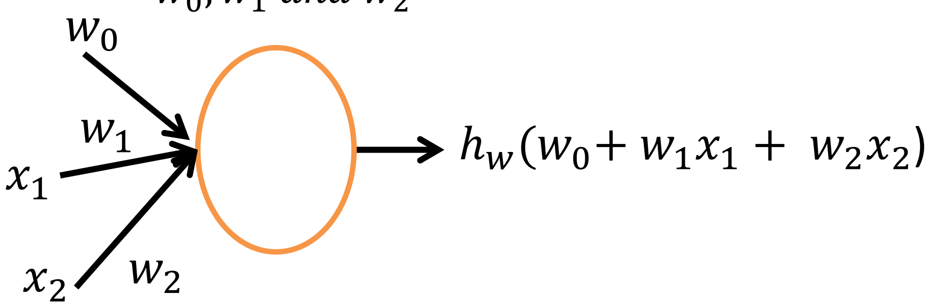
Perceptron as a Linear Binary Classifier

- In the context of pattern classification, perceptron could be useful to determine if a sample belongs to one of two classes (binary classification).

- Linear decision boundary for binary classification



- The task is to predict to which of two possible categories a certain data point belongs based on a set of input variables.
- As the diagram suggests we can readily do that when the decision boundary is straight line or a simple curve defined by a small number of parameters, e.g. w_0, w_1 and w_2

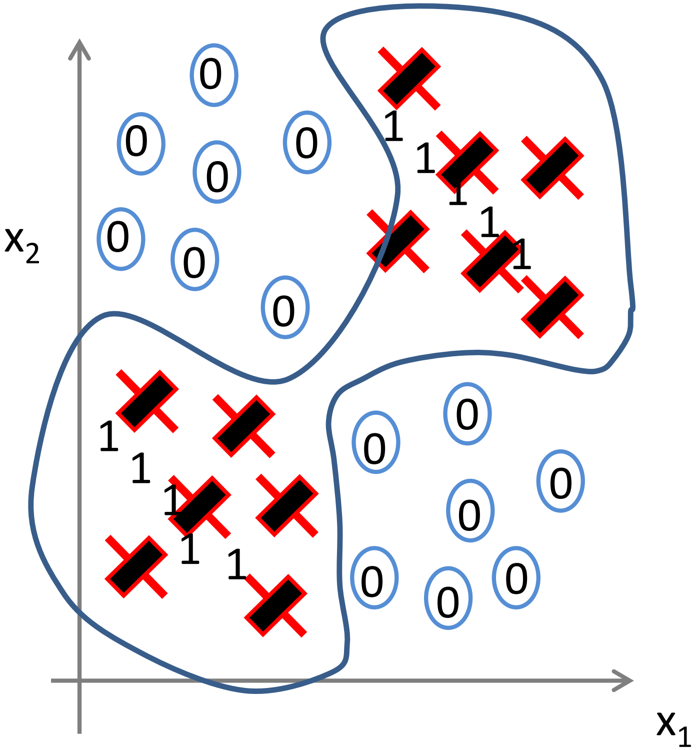


- h_w is a frequently used symbol for the activation function. It comes from the fact that the activation function is sometimes called the “hypothesis”.
- Minsky and Papert argued that this is where the usefulness of perceptron ends.

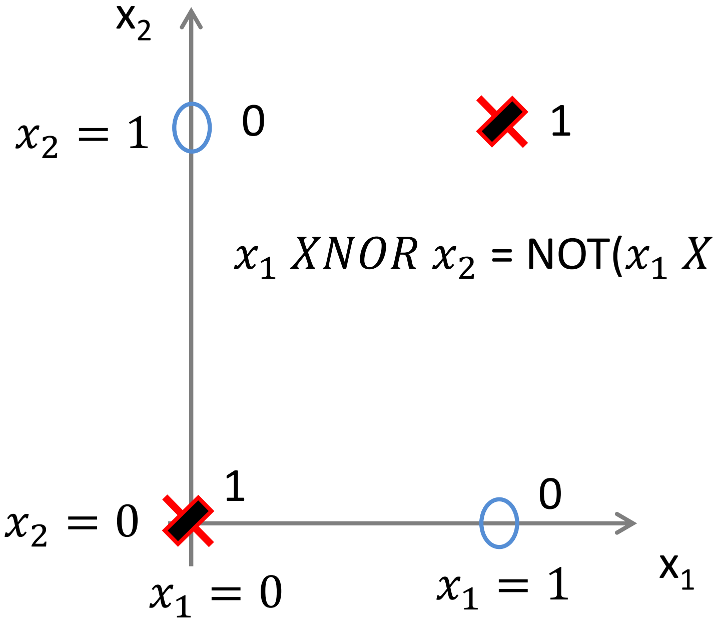
Non-linear classification example: XOR/XNOR

Consider two classes indicated by 0-s and X-s respectively, as presented below. We would like to find the decision boundary between these two classes. A single straight line would not do it.

We could roughly approximate the problem on the left with a simple logical problem: XNOR function.



x_1, x_2 are binary (0 or 1).



$x_1\ XNOR\ x_2$ = NOT($x_1\ XOR\ x_2$)

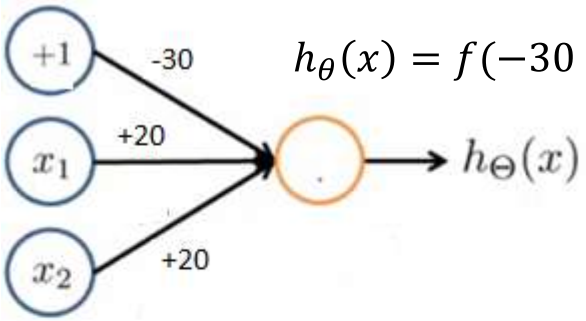
We will try to implement a neural network circuit for the "computation" on the right. We start by looking at even simpler calculations.

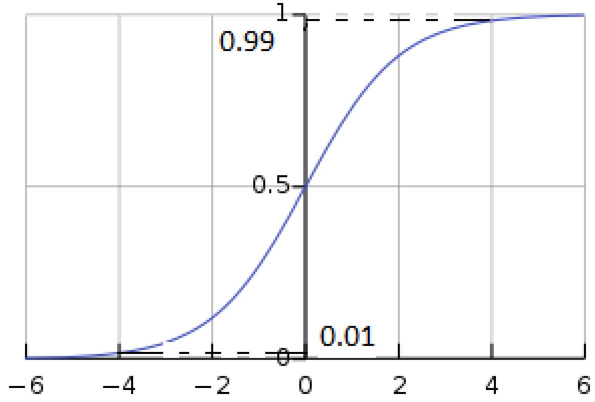
Andrew Ng

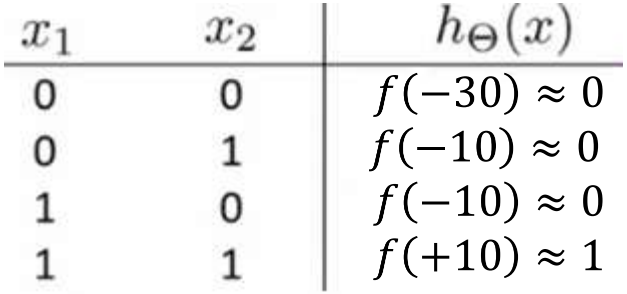
Neural network AND Logical Function

- Could we get a one-unit neural network to compute this logical AND function? Add a bias unit
  - Add some weights for the networks
- What are weights?
  - Weights are the parameter values which multiply into the input nodes (i.e. Θ)
- Sometimes it's convenient to add the weights into the diagram
  - These values are just the Θ parameters so
    - Θ_{10}^{1} = -30
    - Θ_{11}^{1} = 20
    - Θ_{12}^{1} = 20
- Look at the four input values.
- The table on the right is called “logical table”
- Sigmoid function at +/-4.6 is 0.01 = 1% away from its saturation value.

$$h_\theta(x) = f(-30 + 20x_1 + 20x_2)$$



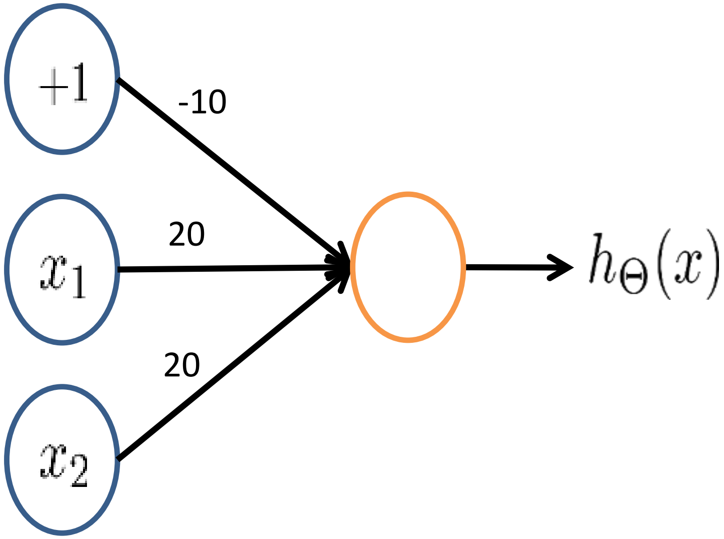




x_1	x_2	$h_\Theta(x)$
0	0	$f(-30) \approx 0$
0	1	$f(-10) \approx 0$
1	0	$f(-10) \approx 0$
1	1	$f(+10) \approx 1$

$$h_\theta(x) \approx x_1\ AND\ x_2$$

Neural network OR function

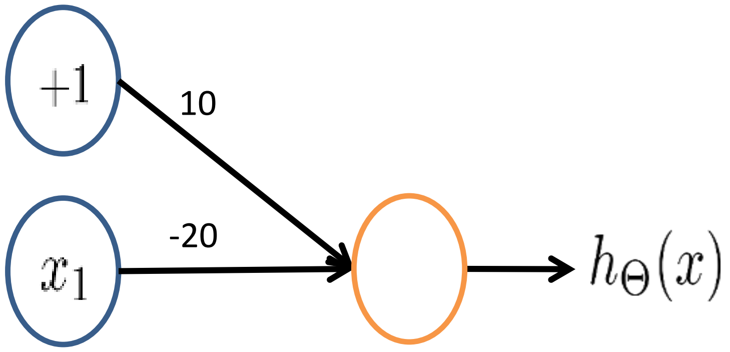


x_1	x_2	$h_\Theta(x)$
0	0	f(−10) ~ 0
0	1	f(10) ~ 1
1	0	~ 1
1	1	~1

f$(-10 + 20x_1 + 20\ x_2)$

Neural Network NOT function

- Negation is achieved by putting a large negative weight in front of the variable you want to negative



$$f(10\ -20x_1)$$

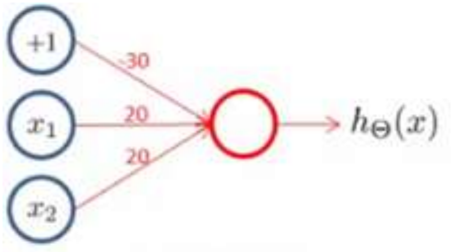
x_1	$h_\Theta(x)$
0	$f(10) \sim 1$
1	$f(-10) \sim 0$

- If you recall from your Computer Science courses, using AND, OR and NOT functions, one could build any logical function. At a conceptual level, our computers are collections of AND, OR and NOT gates.
- This is were our claim that Neural Networks could calculate anything is coming from.

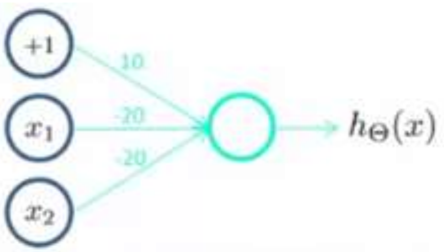
Neural Network XNOR function

- XNOR is short for NOT XOR
  - i.e. NOT an exclusive or, so either (1,1) or (0,0)
- So we want to structure this so the input which produce a positive output are
  - AND (i.e. both true)
    OR Neither (which we can shortcut by saying not only one being true)

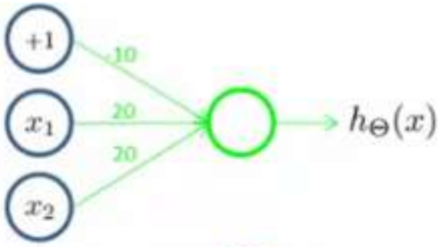
So we combine these into a neural network as shown below;



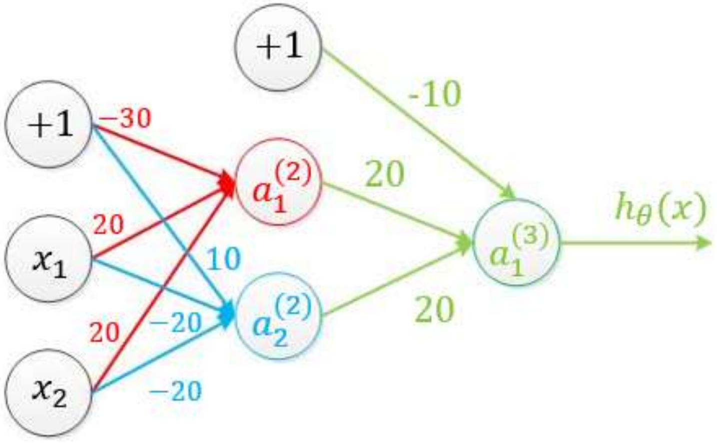
x_1 AND x_2

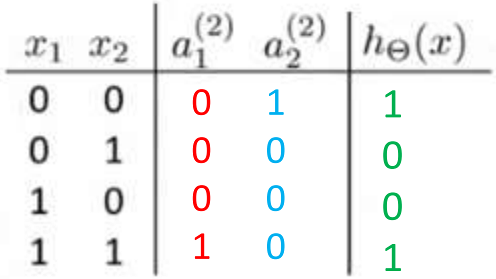


(NOT x_1) AND (NOT x_2)



x_1 OR x_2

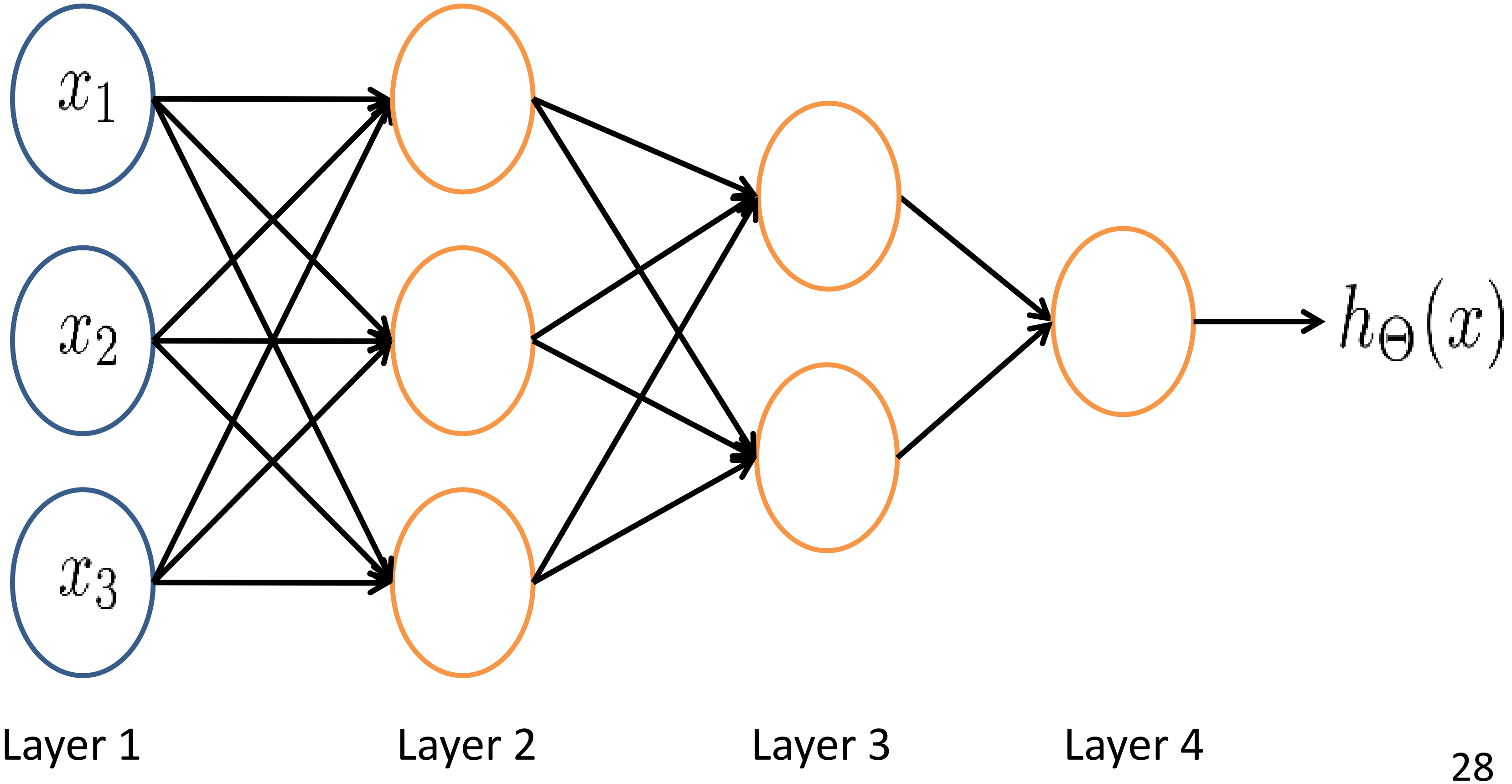




x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

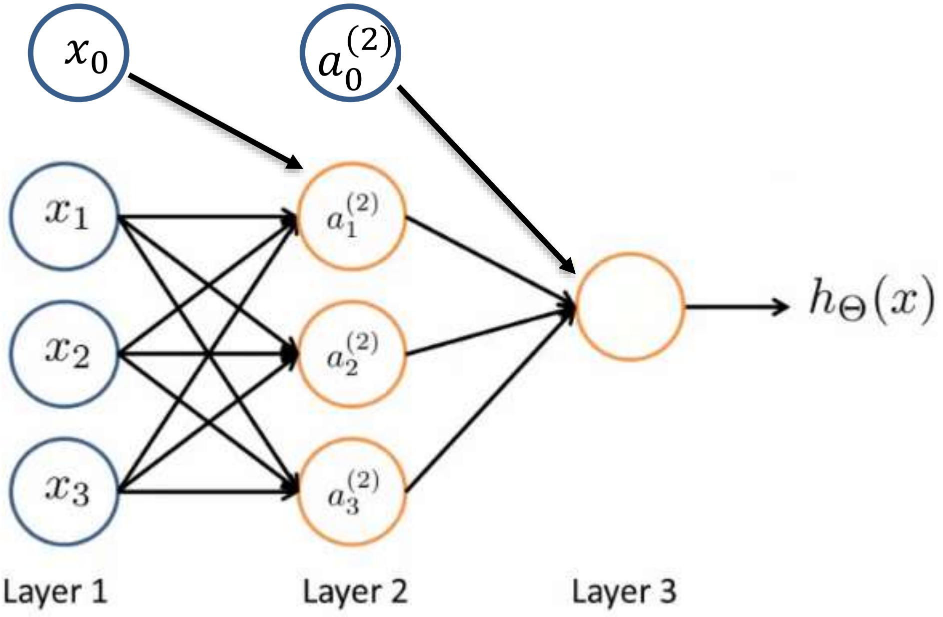
Neural Network Intuition

- From the previous and similar analyses we got an idea. Apparently, we need to build artificial neural networks as sequences of layers where outputs of one layer are inputs into the next layer.
- The network on the previous slide had 3 layers. We could have many, many more.



Notation and Terminology of Multi Layer Networks

- Below we have a group of neurons strung together



- Here, input are x_1, x_2 and x_3
- Sometimes we call inputs the activations on the first layer - i.e. (a_1^1, a_2^1 and a_3^1)
- There are three neurons in layer 2 (a_1^2, a_2^2 and a_3^2)
- The final fourth neuron produces the output. Which again we *could* call a_1^3
- First layer is the **input layer**
- Final layer is the **output layer** - produces value computed by a hypothesis
- Middle layer(s) are called the **hidden layers**
- You don't observe the values processed in the hidden layer
- Not a great name.
- Can have many hidden layers

$a_i^{(j)}$ - activation of unit *i* in layer *j*

So, a_1^2 - is the **activation** of the 1st unit in the second layer

By activation, we mean the value which is computed and placed as output by that node

$a_0^{(j)}$ bias term of layer j. The bias term is often presented as **$b_0^{(j)}$**

Andrew Ng

Multiple Output Networks: One-vs-all.

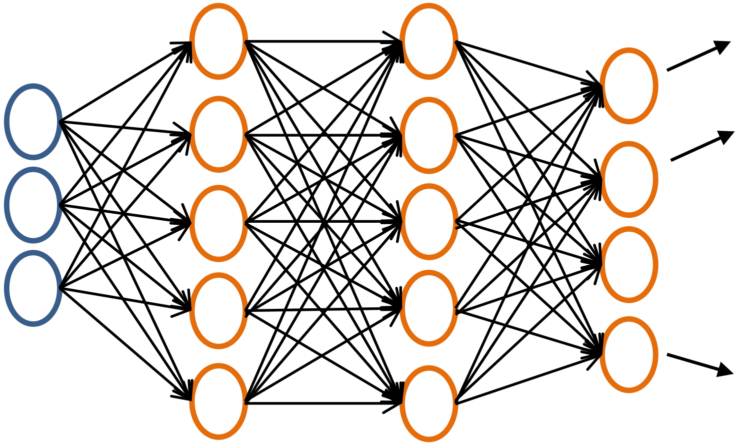
Pedestrian

Car

Motorcycle

Truck

Input:
x_j
Image of
an object



$h_\Theta(x) \in \mathbb{R}^4$

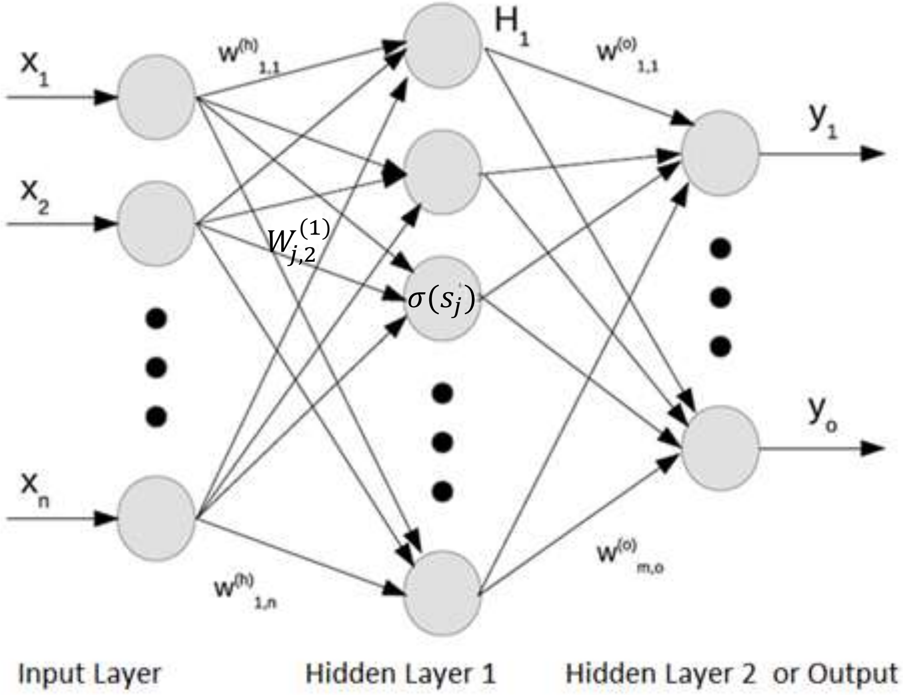
In the typical classification analysis we have that many output neurons as we have classes.

Want $h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, etc.

when pedestrian when car when motorcycle

Neural Network, Hidden Layers, Forward Propagation

- Typically, neurons are organized in (hidden) layers. Hidden layers are indexed.
- Every neuron within every layer is also indexed. For example, j indicates a neuron in the first layer and i specifies i-th dendrite (input) of that neuron.
- $(x_1, x_2, ..., x_n)$ on the left side are coordinates of an n-dimensional vector X representing an input sample (example). There could be N such samples.
- Every input x_i is multiplied by a weight $W_{j,i}^{(k)}$.
- The sum of weighted inputs is passed through the
- activation function
- Outputs of the first hidden layer could be fed into a "final" output layer or could be fed into another hidden layer. The number of hidden layers is relatively arbitrary and ranges from 1 to 150 or more.
- The activation functions of the last layer represent the outputs $\{y_i\}$.



- The objective is to build a network that would produce reasonably accurate outputs (e.g. classification indicators) $\{y_i\}$. In the classification problems the output vector Y typically has as many coordinates, C, as there are categories in the classification problem.
- Process of calculating output vector Y starting with input vector X is called the *Forward Propagation*

Optimization Problem, Loss Function

- Initially, we do not know proper values of parameters ($w_{ij}^{(k)} and\ b_j^{(k)}$) of any neuron in any layer. The training set of experiments $\{X^i\}$ contains proper (correct) labels $\{Y^i\}$ for all of its elements. Superscript i is sample/experiment index.
- Forward calculation for any one experiment X^i produced an output vector $Y_{predicted}^i$. The value of that vector does not have to match the known label Y^i.
- The sum over all N samples of the difference between known labels Y^i and the forward calculated values $Y_{predicted}^i$ is a measure of the quality of our network.
- Usually, we express that quality as the Cost or Loss function of the form:

$$L_2(\{W\},\{b\}) = \sum_{i=1}^{N}(Y^i - Y_{forward\ pass\ calculated}^i)^2$$

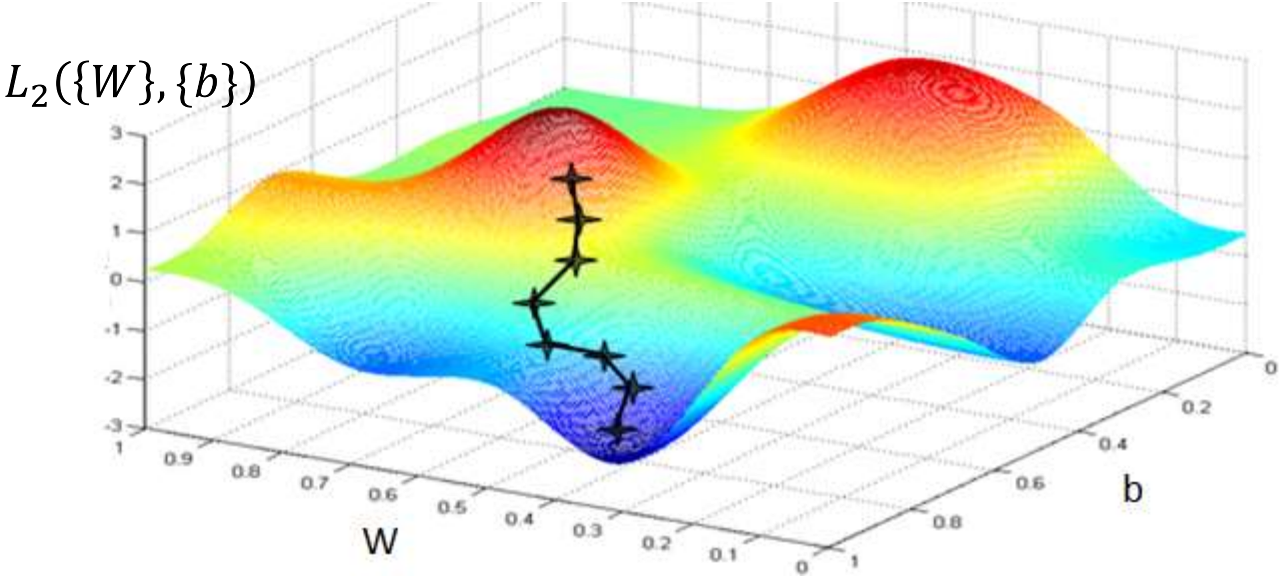
- which is the sum of squares of the errors the model makes at every forward pass. Notice that the loss function depends on values of all weights and biases of all neurons in all hidden layers.
- Finding the best collection of weights and biases which minimizes the loss function is an optimization problem.
- The input vectors could have very large dimensionality, 10s, 100s, 1000s, even more. We could have very large number of weights just in the first layer. Since we could have many hidden layers, the number of weights that we have to find could be truly large.
- We are dealing with an optimization in a space of a (very) large dimensionality.

Gradient Descent

- Input variables for the optimization, i.e. minimization of the Loss function L_2 are the model weights $\{W\}$ and biases $\{b\}$. The dataset features (inputs) $\{X^i\}$ are fixed and are not variables in the optimization process.
- We search for the minimum of L_2 by starting from a random point in space $\{W, b\}$ and by moving away from it in the direction opposite to the gradient:

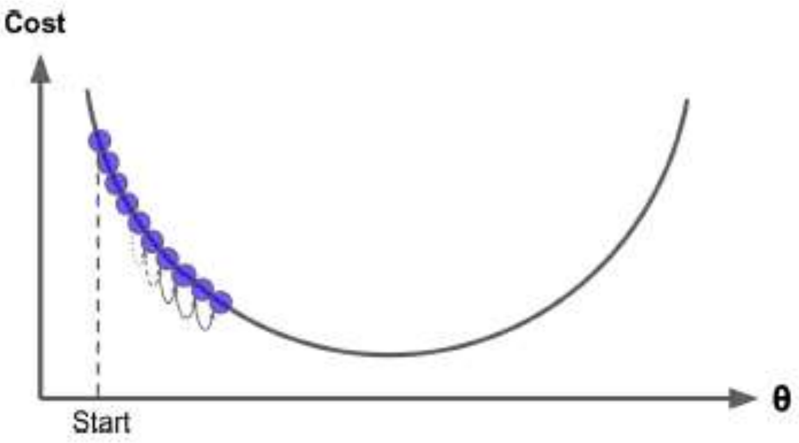
$$\nabla L_2 = \left(\frac{\partial L_2}{\partial W}, \frac{\partial L_2}{\partial b}\right)$$

- The gradient indicates the direction of maximum growth for the loss function. To find the minimum of L_2 we move in the direction opposite to the gradient.
- That is where the name "gradient descent" comes from.

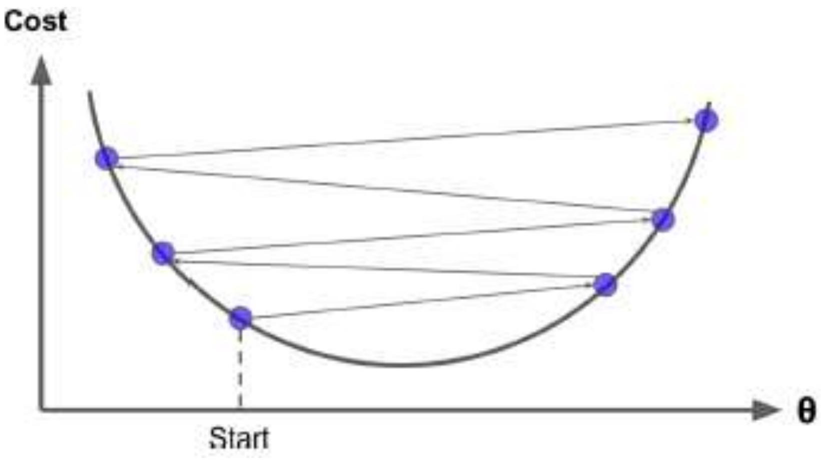


Importance of Learning Rate

- Choosing the step size, the learning rate, is one of the most important hyper-parameter settings in training a neural network. In our blindfolded hill-descent analogy, we feel the hill below our feet sloping in some direction, but the step length we should take is uncertain.
- If we choose our feet carefully we can expect to make consistent but very small progress.
- Conversely, we can choose to make a large, confident step in an attempt to descend faster, but this may not pay off.
- At some point taking a bigger step gives a higher loss as we "overstep".



Learning rate too small



Learning rate too large

Compute Gradients Analytically

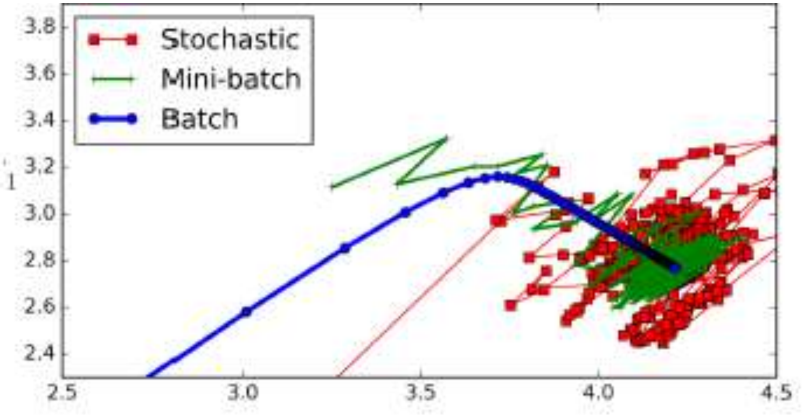
- The second way to compute the gradient is analytically using Calculus, which allows us to derive a direct formula for the gradient (no approximations).
- Analytic gradients are very fast to compute. For not so large networks analytic gradients are tractable.
- However, unlike the numerical gradient, implementations of analytical gradients could be more error prone. In practice, it is very common to compute the analytic gradient and compare it to the numerical gradient to verify the correctness of your implementation.
- If you are interested in this technique you should look at Autograd API:
- https://github.com/HIPS/autograd

Stochastic Gradient Descent

- The loss function is a sum over the entire batch of n samples. When n is small, and the dimension of $\{W, b\}$ space is not huge, we can perform that sum and such calculation is referred to as the *Batch Gradient Descent*.
- Typically, both the number of samples n and the dimension of the weight space are very large. Performing such sum at every point along the gradient descent curve is prohibitive.
- We could perform gradient descent by taking a single sample X^i and by calculating the gradient with of the single term $(Y^i - Y^i_{forward})^2$ with respect to parameters $\{W, b\}$. This is obviously an approximation but is a much faster procedure. To account for the influence of different samples, after every step we could choose a different sample in a random (stochastic) way. Such process is named *Stochastic Gradient Descent.*
- As we are changing samples, the gradients typically vary wildly.
- A somewhat better approach is to take a moderately small number of samples: 64, 128, or some other power of 2, and approximate the loss function with

$$L_2(\{W\},\{b\}) \approx \sum_{i}^{mini\_batch} (Y^i - Y^i_{forward})^2$$

Summation of errors over a mini-batch smooths the gradient and also speeds up the calculation since our hardware efficiently parallelizes such calculations.



The axes of the above plot are two weights W.

Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD) implemented with one sample at a time is rarely used.
- Due to vectorized code optimizations it is computationally much more efficient to evaluate the gradient for 128 examples in one go, than the gradient for 128 individual samples 128 times over.
- Even though SGD technically refers to calculate the gradient using a single example at a time, when we use the term SGD, we typically refer to mini-batch gradient descent. In literature, mentions of MGD for "Minibatch Gradient Descent", or BGD for "Batch gradient descent" are rare.
- The size of the mini-batch is a hyper-parameter but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or randomly set to some value, e.g. 32, 64 or 128. We use powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2.

- Over last two decades we developed proficiency in computing the gradient analytically using the chain rule, otherwise also referred to as backpropagation.

L1, L2 Loss Functions

- Two standard and most frequently used loss functions are the Absolute deviations, L1, and the Least square errors, L2.
- L1 loss function minimizes the **absolute differences** between the estimated values and the existing target values. Here, y_i is the target value, the corresponding estimated value $h(x_i)$ is the result of a forward pass. x_i denotes the feature set of a single sample.
- L1 is the sum of absolute differences for ‘n’ samples calculated as

$$L1 = \sum_{i=0}^{n} |y_i - h(x_i)|$$

- L2 loss function minimizes the **squared differences** between the estimated and existing target values.

$$L2 = \sum_{i=0}^{n} (y_i - h(x_i))^2$$

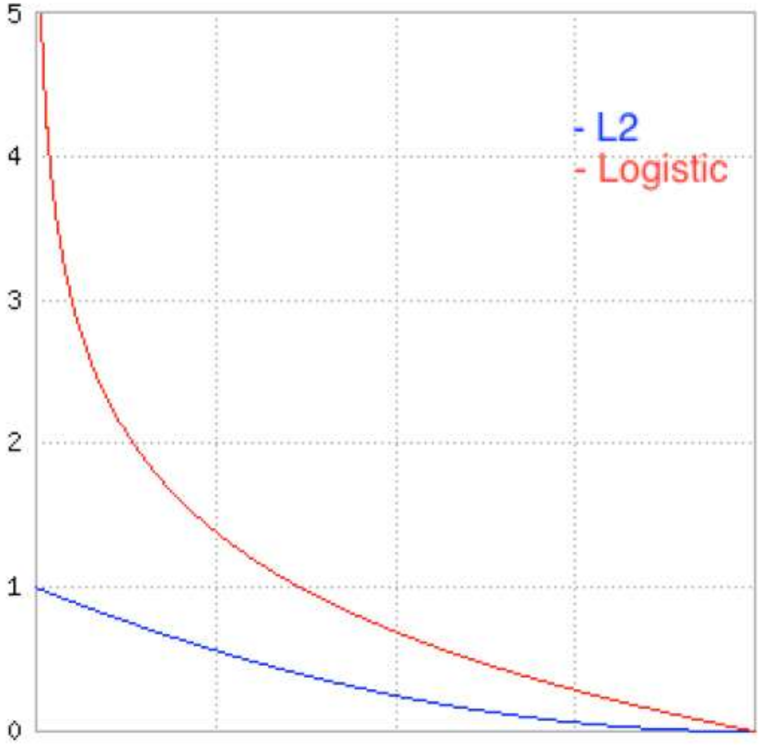
- L2 components are much larger in the case of outliers compared to L1. Square of the difference between an incorrectly predicted target value and the original target value emphasizes that difference.
- L1 loss function is more robust and is generally not affected by outliers. L2 loss function adjusts the model according to the outlier values, even on the expense of other samples. L2 loss function is more sensitive to outliers in the dataset.

Loss Function for Classifications

- The logistic function computes the probability of the answer being "yes." In the training set, a "yes" answer should represent 100% of probability, or the output value of 1.
- The loss for a sample with “yes” value should be calculated as the probability our model assigns in forward calculation minus 1 squared.
- A "no" answer represents 0 probability, hence the loss is any probability the model assigns for that example, and again squared.
- Consider an example where the expected answer is "yes" and the model is predicting a very low probability for it, close to 0. This means that it is close to 100% sure that the answer is "no."
- The squared error penalizes such a case with the same order of magnitude for the loss as if the probability would have been predicted as 20, 30, or even 50% for the "no" output. In other words, simple error-squared is not very sensitive loss function.

Logistic or Cross Entropy Loss Function

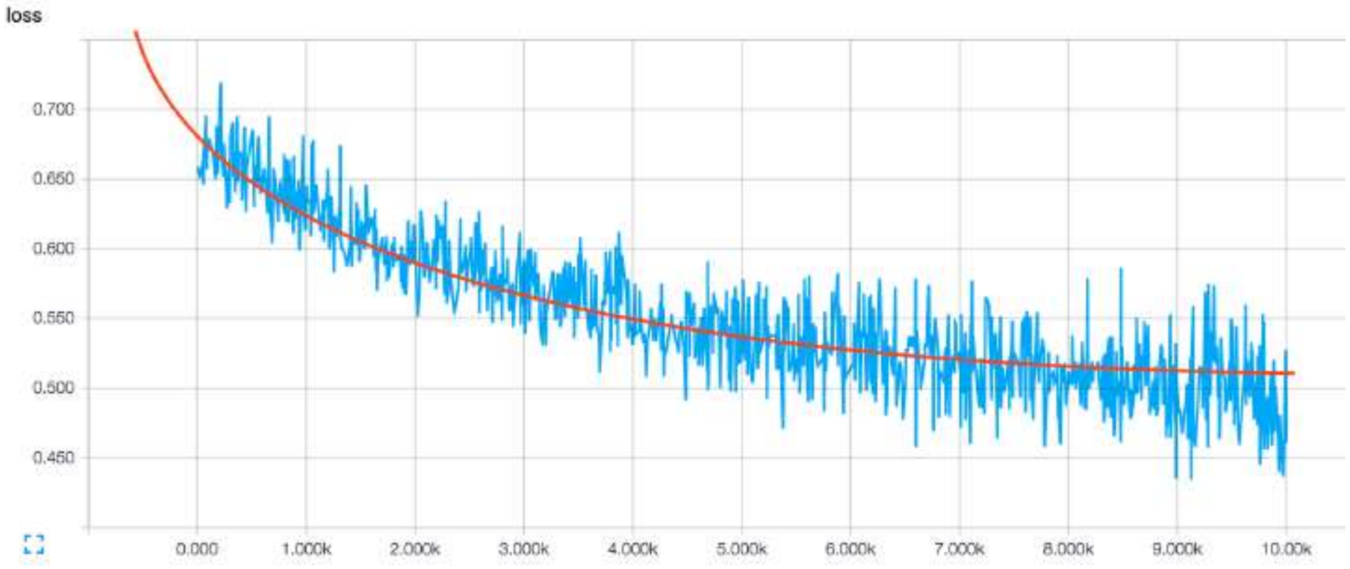
- Logistic or Cross entropy loss function is defined as:
- $L = \sum_{i=1}^{N}[y^i \log(y_{estimated}^i) + (1 - y^i) \log(1 - y_{estimated}^i)]$
- We can visualize this function for the predicted output for a "yes" as:



- Cross entropy outputs a much greater value ("penalty"), when the output is farther from what is expected.
- With cross entropy, as the predicted probability comes closer to 0 for the "yes" example, the penalty grows towards infinity.
- This makes it very expensive for the model to make that mis-prediction.
- Cross entropy is better suited as a loss function for classification models.

Loss Function over time

- In practice, we can’t really plot the loss function as a function of its many variables. However, we can look at the plot of the computed total loss thru time.
- This is how a well behaving loss should diminish through time, indicating a good learning rate:



- The blue line is the loss function over time, and the red line represents the tendency line.

Softmax Classification

- With logistic regression we were able to model the answer to the yes-no question.
- We want to be able to answer questions of multiple choice type, like: "Was a person born in Boston, London, or Sydney?"
- For that case we use the **Softmax** function, which is a generalization of the logistic regression for K possible different value$s$.
- Softmax function calculates K components of a K-dimensional vector representing corresponding probabilities for each output class. Softmax function represents the j-th component of that vector as:

$$f(x)_j = \frac{e^{-x_j}}{\sum_{i=0}^{K-1} e^{-x_i}} \quad , \quad \sum_{j=0}^{K-1} f(x)_j = 1$$

- Elements of vector $f(x)$ are probabilities, their sum is equal to 1. Every possible input sample must belong to one output class. All classes cover 100% of possible examples.
- If the sum would be less than 1, it would imply that there could be some hidden class alternative. If it would be more than 1, it would mean that each example could belong to more than one class.
- You can show that if the number of classes is 2, the resulting output probability is the same as a logistic regression model.

Probabilistic View

- Looking at the expression, we see that

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

- can be interpreted as the (normalized) probability . Softmax classifier interprets the scores inside the output vector as the unnormalized log probabilities.
- Exponentiating these quantities therefore gives the (unnormalized) probabilities, and the division performs the normalization so that the probabilities sum to one.
- In the probabilistic interpretation, we are therefore minimizing the negative log likelihood of the correct class, which can be interpreted as performing Maximum Likelihood Estimation (MLE).
- If you recall slide 28, the last 4 neurons will implement a Softmax layer, each providing the probability that the input image represents on of 4 analyzed objects.

Back Propagation

- We stated that neural networks can learn their weights and biases by optimizing a selected loss function by using the gradient descent algorithm.
- There is, however, a gap in our explanation.
- We did not discuss how we compute the gradients of the loss function.
- To calculate gradients we use a fast algorithm known as *back propagation*.
- The back propagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.
- That paper describes several neural networks where back propagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble.
- We will try to explain the mechanism using some simple computational graphs

Derivatives, Simple Rules

- Partial derivative, with $\frac{df}{dx}$ replaced with $\frac{\partial f}{\partial x}$, is used when f is a function of several variables, i.e. $f \; = \; f(x, y, z, \dots)$ and we are interested in the rate of change of f when we change one variable and keep other variables fixed.
- For example, if $f(x, y) = xy$, then $\frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$
- Similarly, if $f(x, y) = x + y$, then $\frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$

$$f(x, y) = \max(x, y) \rightarrow \frac{\partial f}{\partial x} = 1(x >= y) \quad \frac{\partial f}{\partial y} = 1(y >= x)$$

$$f(x) = \frac{1}{x} \qquad \rightarrow \qquad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \qquad \rightarrow \qquad \frac{df}{dx} = 1$$

$$f(x) = e^x \qquad \rightarrow \qquad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \qquad \rightarrow \qquad \frac{df}{dx} = a$$

Sigmoid Expression

- Sigmoid, which we frequently use , has a useful property. The derivative of the sigmoid function could be simply expressed over sigmoid function itself:

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

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right)$$

$$\frac{d\sigma(x)}{dx} = (1-\sigma(x))\,\sigma(x)$$

Chain Rule

- Chain rule is convenient mnemonics for writing derivatives of nested functions.
- For example if y is a result of nesting functions f, g and h, as

$$y = f\left(g\left(h(x)\right)\right),$$

then

$$\frac{dy}{dx} = \frac{df}{dg}\,\frac{dg}{dh}\,\frac{dh}{dx}$$

- You could prove the rule for yourself by looking at a simpler expression:

$$y \;=\; f(g(\mathrm{x}\,)), \quad \frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f\left(g(x+\Delta x)\right) - f(x)}{\Delta x}$$

$$\frac{\Delta y}{\Delta x} = \frac{f\left(g(x+\Delta x)\right) - f(g(x))}{\Delta x} = \frac{f\left(g(x) + \Delta x \frac{\Delta g}{\Delta x}\right) - f(g(x))}{\Delta x} = \frac{f\left(g(x)\right) + \frac{\Delta f}{\Delta g}\frac{\Delta g}{\Delta x}\Delta x - f(g(x))}{\Delta x} = \frac{\Delta f}{\Delta g}\,\frac{\Delta g}{\Delta x}$$

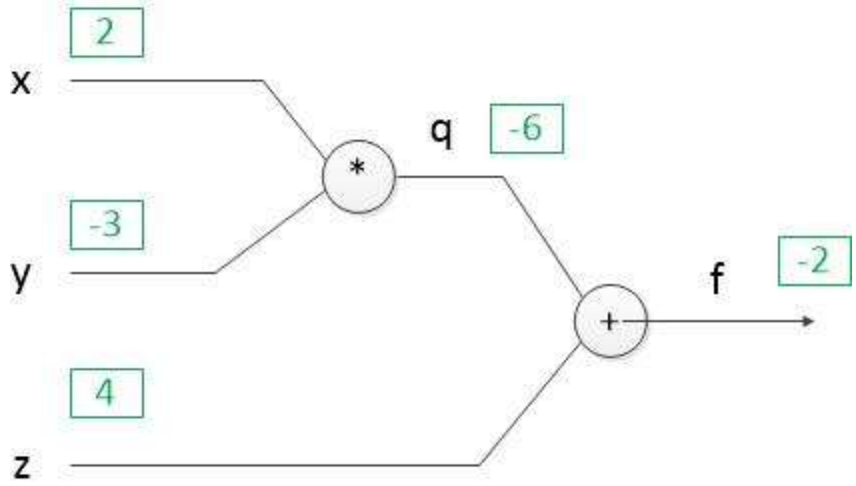
$$\frac{dy}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Backpropagation, Computational Graph

- Objective of backpropagation is to find the value of the gradient of the loss function L with respect to weights W and biases b.
- L is usually a very, very massive and complex function. Let us examine a much simpler function.

$$f = x * y + z$$

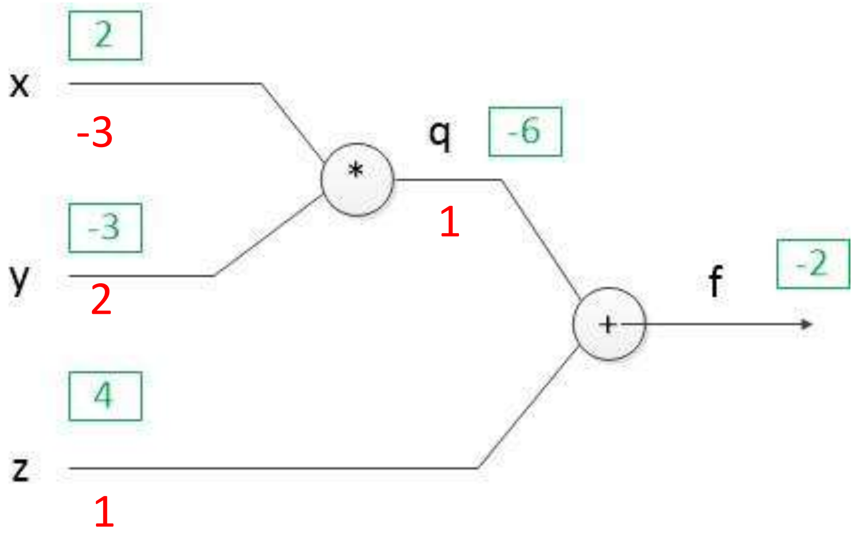
- Computational graph for this expression looks like the one below. Let us choose values of variables, $x = 2, y = -3$ and $z = 4$ and calculate every expression along the way. Those are forward values. Notice that we treat operations as nodes and also label all intermediary variables, like $q = x * y$



Rate of Change of function $f(x, y, z)$

- We want to find the rate of change of $f(x, y, z)$ with respect to all variables: x, y and z, for $f = x * y + z$ = q + z
- Notice: $\frac{\partial f}{\partial z} = 1$, let us place that 1 below the input value of z.
- $\frac{\partial f}{\partial q} = 1$, let us place that 1 below the forward value of q.
- $\frac{\partial q}{\partial x} = y$ = -3 and $\frac{\partial q}{\partial y} = \mathrm{x} = 2$

We also apply chain rule:



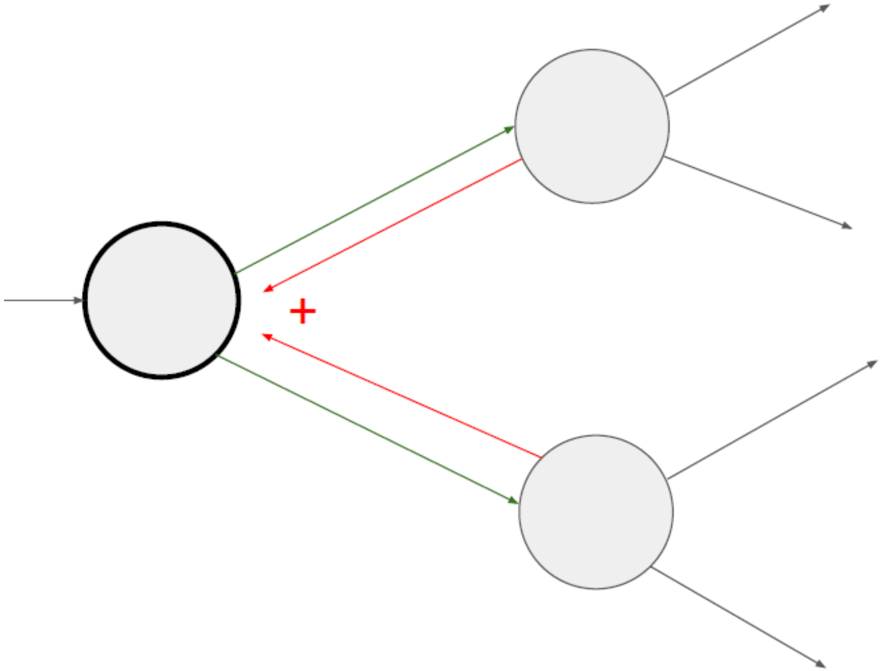
$\frac{\partial f}{\partial z}$ = 1 $\frac{\partial f}{\partial f}$ = 1

$\frac{\partial f}{\partial x}$ =$\frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$ = $1 \; * (-3)$= -3

$\frac{\partial f}{\partial y}$ =$\frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$ = $1 \; * (2)$ = 2

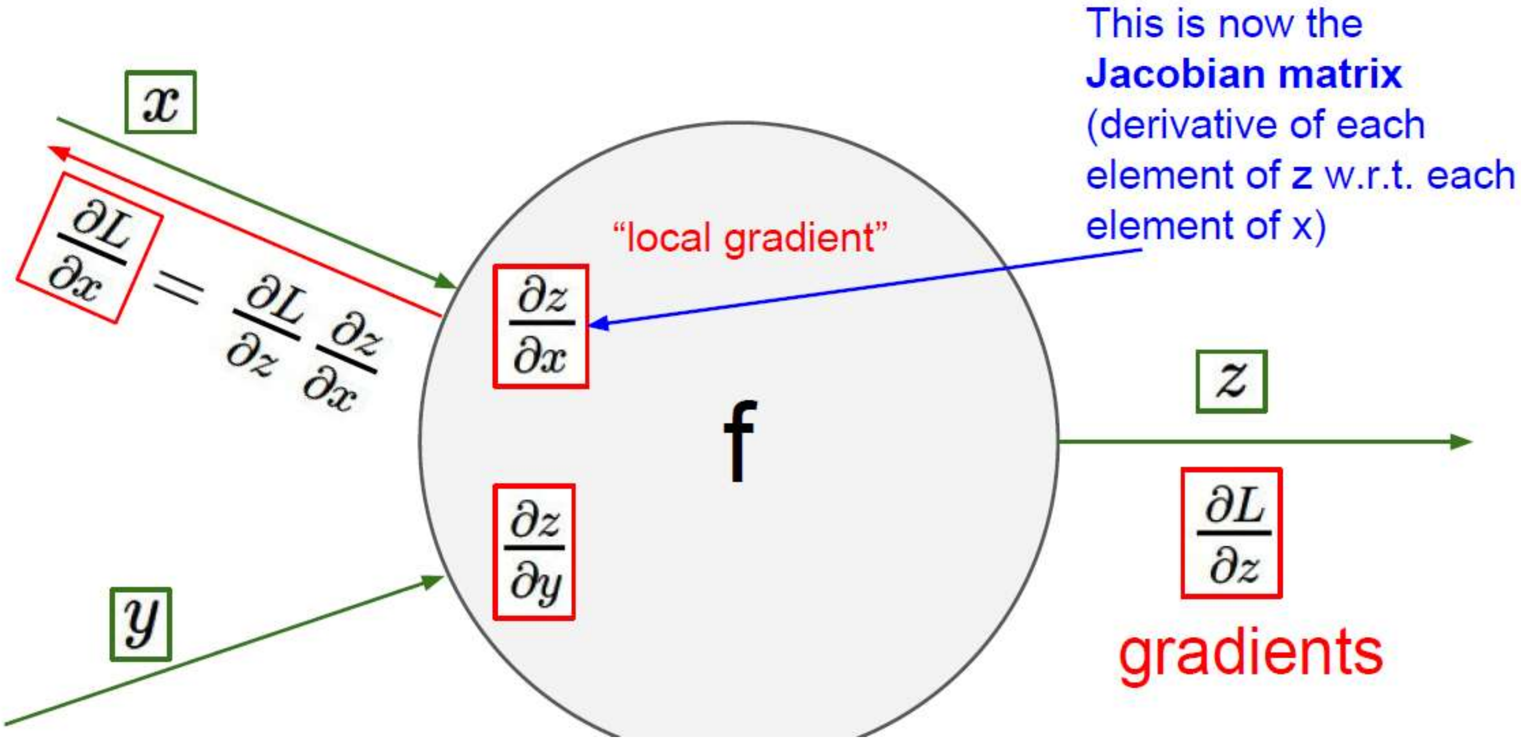
Patterns in Backward flow

- Addition operation: upstream gradient is passed through (multiplied by 1)
- Multiplication op: upstream gradient is multiplied by the other multiplicand
- $\max(x, y)$: upstream gradient is passed to the larger value. Zero is passed to the smaller value
- Branch point: gradients add



Gradients for vectorized code

- If our flow variables: z, y and z in the diagram below are vectors, partial derivatives are replaced with Jacobian matrices:



x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
y
This is now the **Jacobian matrix** (derivative of each element of **z** w.r.t. each element of x)
z
$\frac{\partial L}{\partial z}$
gradients

Pattern of Back Propagation

- You should notice a pattern. The derivative on each layer is the product of the derivatives of the layers after it and the output of the layer before.
- Back propagation algorithm is based on the chain rule. We go forward from the inputs calculating the outputs of each hidden layer up to the output layer.
- Then we start calculating derivatives going backwards through the hidden layers and by reusing all of the elements already calculated. That's the origin of the name back propagation.
- Back propagation is extremely efficient algorithm. It takes the same amount of computational operations to perform back propagation as it takes to perform the forward propagation.

- Unfortunately, you will not have to do any of this exciting mathematics and computations. Your APIs do it all for you.

Keras

- One objective of this lesson is to introduce Keras, a very popular API for Deep Learning.
- Material in this lecture follows fairly closely
  - Keras Documentation (https://keras.io) and
  - *Deep Learning with Pyt*hon by François Chollet, 2018, Manning Publications Co., chapter 3.

Keras

- Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.
- Keras was developed with a focus on enabling fast experimentation. *Being able to go from idea to result with the least possible delay is key to doing good research.*
- Use Keras if you need a deep learning library that:
  - Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
  - Supports both convolutional networks and recurrent networks, as well as combinations of the two.
  - Runs seamlessly on CPU and GPU.
- Keras is compatible with: **Python 2.7-3.6**.

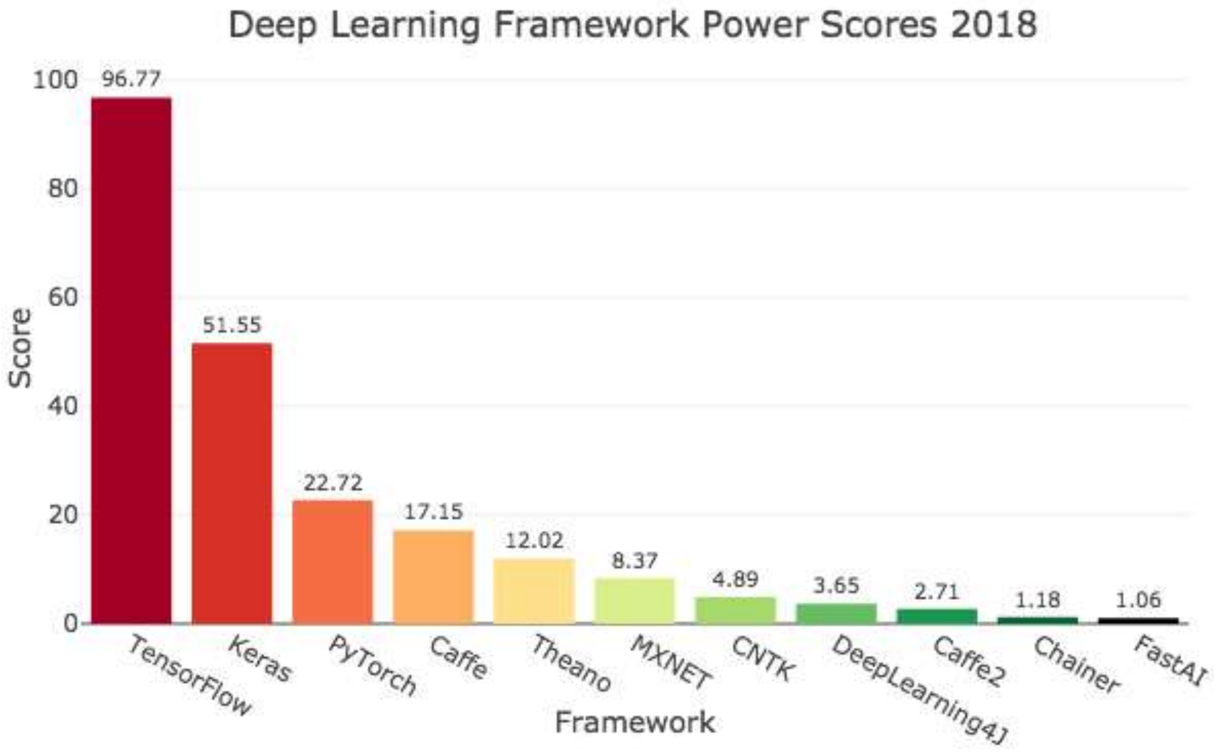
- Based on some statistics, citing the number of downloads, Keras is less popular than TensorFlow, but more popular than PyTorch.
- Keras API is the official front end of TensorFlow, via the `tf.keras` module

Popularity of Different Frameworks

- The following is from

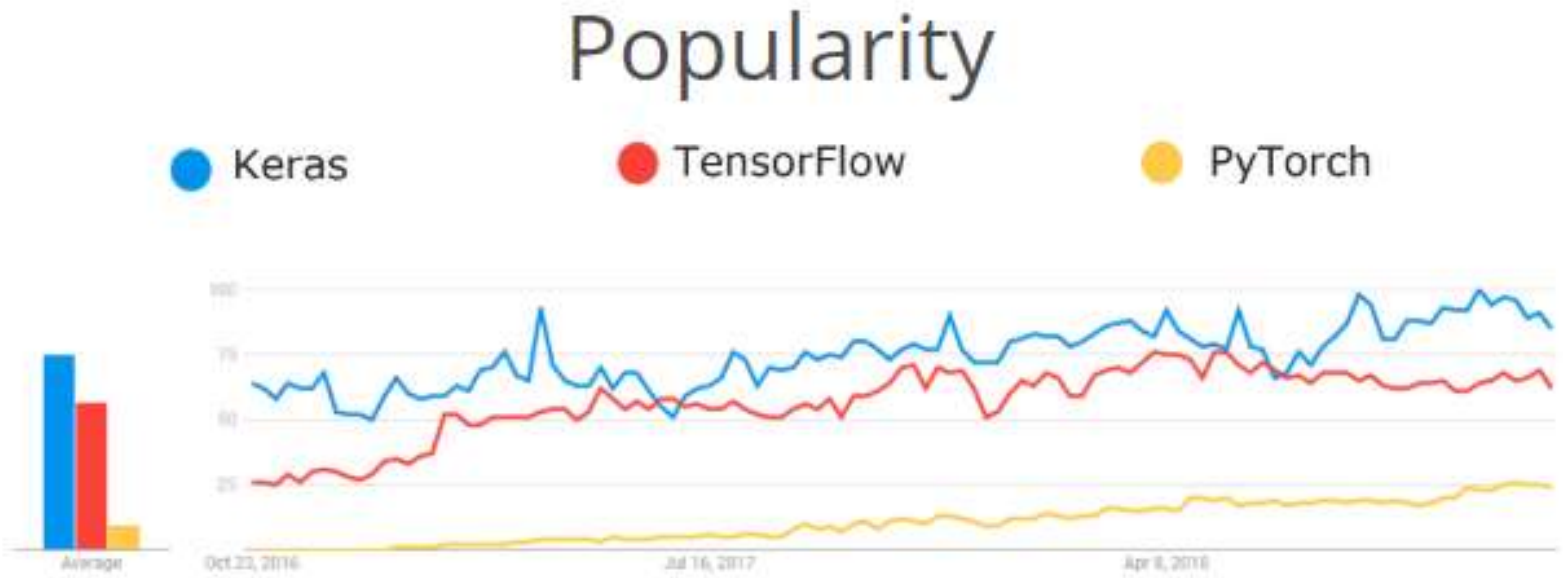
https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

- and represents survey of some 250,000 DL practitioners.



Popularity, again

- Blog: https://www.edureka.co/blog/keras-vs-tensorflow-vs-pytorch/, on May 2019, presents the following information. To me this looks like statistics from indeed.com



- This blog claims: “Keras is usually used for **small datasets** as it is comparatively slower. On the other hand, TensorFlow and PyTorch are used for **high performance** models and **large datasets** that require fast execution. … all the three frameworks have gained quite a lot of popularity. **Keras** tops the list followed by TensorFlow and PyTorch. It has gained immense popularity due to its **simplicity** when compared to the other two.”

Keras Principles

- **User friendliness.** Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- **Modularity.** A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
- **Easy extensibility.** New modules are simple to add (as new classes and functions), and existing modules provide ample examples. Keras lets you easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- **Work with Python**. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easy to debug, and easy to extend.

Keras Models are portable

- Keras models can be easily deployed across a great range of platforms:
  - On iOS, via Apple’s CoreML (Keras support officially provided by Apple).
  - On Android, via the TensorFlow Android runtime..
  - In the browser, via GPU-accelerated JavaScript runtimes such as Keras.js and WebDNN.
  - On Google Cloud, via TensorFlow-Serving.
  - In a Python webapp backend (such as a Flask app).
  - On the JVM, via DL4J model import provided by SkyMind.
  - On Raspberry Pi.

Installing Keras

- You need to have already installed TensorFlow, Theano or CNTK.
- TensorFlow installs on a system which has Python 3.6.x with

```
$ sudo pip install tensorflow
```

- On any system that has pip installed and TensorFlow installed you do:

```
$ sudo pip install keras          # Linux
C:\..> pip install keras          # Windows
```

- If you have a good reason, you could install Keras from the source.
- First, clone Keras using git:

```
git clone https://github.com/keras-team/keras.git
```

- Then, `cd` to `keras` folder and run the install command:

```
cd keras
sudo python setup.py install
```

- If you have TensorFlow installed, tf.Keras is already there as `tf.keras`
- You might need the following tools as well:
- `cuDNN` (recommended if you plan on running Keras on GPU).
- `IHDF5` and h5py (required if you plan on saving Keras models to disk).
- Install `graphviz` and `pydot` (used by visualization utilities to plot model graphs).

Building Blocks of NN

- Training a neural network revolves around the following objects:
  - *Layers*, which are combined into a *network* (or *model*)
  - The *input data* and corresponding *targets*
  - The *loss function*, which defines the feedback signal used for learning
  - The *optimizer*, which determines how learning proceeds

- The network, composed of layers that are chained together, maps the input data to predictions.
- The loss function compares these predictions to the targets, producing a loss value.
- Loss value is a measure of how well the network's predictions match what was expected.
- The optimizer uses this loss value to update the network's weights.

- As we will see in the following slides, Keras makes assembly of neural networks from the above building blocks very simple.

Layers

- The fundamental data structure in neural networks is the *layer.*
- A layer is a data-processing module that takes as input one or more tensors and outputs one or more tensors. Some layers are stateless, but many types of layers have a state. *The state is represented by the layer's weights*, one or several tensors learned with stochastic gradient descent, which together contain the network's *knowledge*.
- Different layers are appropriate for different tensor formats and different types of data processing.
- For instance, simple vector data, stored in 2D tensors of shape (`samples, features`), is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the `Dense` class in Keras).
- Sequence data, stored in 3D tensors of shape (`samples, timesteps, features`), is typically processed by *recurrent* layers such as an `LSTM` layer. Image data, stored in 4D tensors, is usually processed by 2D convolution layers (`Conv2D`).
- Keras treats layers as the LEGO bricks of deep learning. Lego blocks is a very appropriate metaphor for Keras layers.
- Deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.
- In Keras l*ayer compatibility* refers to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape.

Tensors

- In Keras and TensorFlow, tensors are glorified multidimensional matrixes.
- Unfortunately, tensors could cause a lot of grief.
- Keras comes with a large collection of functions (methods) that transform and manipulate tensors. Just a few are listed below:
- **reshape** keras.backend.reshape(x, shape) Reshapes a tensor to the specified shape. Arguments x : Tensor or variable. shape : Target shape tuple. Returns A tensor.
- **Ones k**eras.initializers.Ones() Initializer that generates tensors initialized to 1. [source]
- **Zeros** keras.initializers.Zeros() Initializer that generates tensors initialized to 0. [source]
- **set_value** keras.backend.set_value(x, value) Sets the value of a variable, from a Numpy array. Arguments x : Tensor to set to a new value. value : Value to set the tensor to, as a Numpy array (of the same shape)
- **log** keras.backend.log(x) Element-wise log. Arguments x : Tensor or variable. Returns A tensor.
- **constant** keras.initializers.Constant(value=0) Initializer that generates tensors initialized to a constant value. Arguments value : float; the value of the generator tensors. [source]
- **square** keras.backend.square(x) Element-wise square. Arguments x : Tensor or variable. Returns A tensor.
- **average** keras.layers.Average() Layer that averages a list of inputs. It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape). [source]
- **concatenate** keras.backend.concatenate(tensors, axis=-1) Concatenates a list of tensors alongside the specified axis. Arguments tensors : list of tensors to concatenate. axis : concatenation axis. Returns A tensor
- **flatten** keras.backend.flatten(x) Flatten a tensor. Arguments x : A tensor or variable. Returns A tensor, reshaped into 1-D
- **exp** keras.backend.exp(x) Element-wise exponential. Arguments x : Tensor or variable. Returns A tensor.
- **abs** keras.backend.abs(x) Element-wise absolute value. Arguments x : Tensor or variable. Returns A tensor.

Creating Layers

- We will create a layer that will only accept as input 2D tensors where the first dimension is 784. Axis 1, the batch dimension, is unspecified, and thus any number of samples could be accepted. You recognize this as standard MNIST example

```
from keras import layers
layer = layers.Dense(32, input_shape=(784,))
```

- This layer will return a tensor where the first dimension has been transformed to be 32. Thus this layer can only be connected to a downstream layer that expects 32-dimensional vectors as its input.
- When using Keras, we do not worry about compatibility, because the layers we add to our models are dynamically built to match the shape of the incoming layer.
- For instance, suppose you write the following:

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

- The second layer didn't receive an input shape argument—instead, it automatically inferred its input shape as being the output shape of the layer that came before.

Models, Network Architecture

- Object `model`, that we introduce on the previous slide is the key holder of network architecture. In Keras the `model` is a directed, acyclic graph of layers.
- The most common `model` is a linear stack of layers, mapping a single input tensor to a single output tensor.
- Keras can support a broad variety of network topologies, like
  - Sequential networks
  - Two-branch networks
  - Multihead networks
  - Inception blocks
- The topology of a network defines a *hypothesis space*. By choosing a network topology, you constrain your *space of possibilities* (hypothesis space) to a specific series of tensor operations, mapping input data to output data.
- Within every hypothesis space we will search for is an optimal set of values for the weight tensors of all layers.
- Picking the right network architecture is more an art than a science. There are some best practices and principles you can rely on.

Loss functions and Optimizers

- Once the network architecture is defined, you still have to choose :
  - *Loss function (cost or objective function)*—This is the quantity that will be minimized during training. It represents a measure of success of network's ability to predict values.
  - *Optimizer i*s a class which determines network weights based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).
- A neural network that has multiple outputs may have multiple loss functions (one per output). But the gradient-descent process must be based on a *single* scalar loss value; so, for multi-loss networks, all losses are combined (via averaging) into a single scalar quantity.
- Choosing the right loss function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss;
- When it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss function.
- We use
  - *binary cross entropy* for a *two-class classification problem*,
  - *categorical cross entropy* for a *many-class classification problem*,
  - *mean-squared-error* for a *regression problem*,
  - *connectionist temporal classification* (CTC) for a *sequence-learning problem*, and so on.
- Only when you work on truly new research problems, will you have to develop your own loss functions.

Usage of Optimizers

- An optimizer is one of the two arguments required for compiling a Keras model:

```
from keras import optimizers
model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('softmax'))
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

- You can instantiate an optimizer before passing it to `model.compile()`, as above, or you can call it by its name, when the default parameters for the optimizer will be used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd’)
```

- The parameters `clipnorm` and `clipvalue` can be used with all optimizers to control gradient clipping:

```
# All parameter gradients will be clipped to a maximum norm of 1.
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
from keras import optimizers
# All parameter gradients will be clipped to a max 0.5 and min of -
0.5.
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```

Supported Optimizers (https://keras.io/optimizers/)

- *Momentum (not directly offered by Keras) is a s*tochastic gradient descent algorithm which remembers the update Δ *w* at each iteration, and determines the next update as a linear combination of the gradient and the previous update
- *SGD (*Stochastic gradient descent) optimizer includes support for momentum, learning rate decay, and Nesterov momentum.
- *RMSProp* (for Root Mean Square Propagation) is an optimizer in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- *Adam* (short for Adaptive Moment Estimation) is an update to the *RMSProp* optimizer. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used.
- *AdaGrad* (for adaptive gradient algorithm) is a modified stochastic gradient descent with per-parameter learning rate. Informally, this increases the learning rate for more sparse parameters and decreases the learning rate for less sparse ones. This strategy often improves convergence performance over standard stochastic gradient descent in settings where data is sparse and sparse parameters are more informative.
- *Adadelta* is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelta you don't have to set an initial learning rate. In this version, initial learning rate and decay factor can be set, as in most other Keras optimizers. Recommended to leave the parameters at default values.
- *Adamx*: It is a variant of Adam based on the infinity norm.
- *Nadam:* Nesterov Adam optimizer extends Adam with Nestorov accelerated gradient algorithm.

Developing with Keras

- Typical Keras program requires the following steps:
  1. Define your training data: input tensors and target tensors.
  2. Define a network of layers (or *model*) that maps your inputs to your targets.
  3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
  4. Iterate on your training data by calling the `fit()` method of your model.

- There are two ways to define a model: using the
  - Sequential class (only for linear stacks of layers, which is the most common network architecture by far) or the
  - *Functional API* (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures).

Learning Process

- All of the following steps are the same for both Sequential model and the Functional API.
- The learning process is configured in the `compile()` step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you monitor during training.
- The following is an example with a single loss function, which is by far the most common case:

```
from keras import optimizers
model.compile(optimizer=optimizers.RMSprop(lr=0.001), loss='mse',
metrics=['accuracy’])
```

- Finally, the learning process consists of passing Numpy array of input data (and the corresponding target data) to the model via the `fit()` method, similar to what you would do in Scikit-Learn and several other machine-learning libraries:

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10
```

- Inference is performed with method `predict()`:

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None)
```

- The method generates output predictions for the input samples x.

Where to Run Deep Learning Jobs

- Toy NN jobs deliver results on your Windows PC, laptop or Mac machine.
- If you have NVIDIA GPU card in your machine you will get 5 to 10 increase of performance.
- Both TensorFlow and Keras and other DL frameworks appear to be developed on Linux machines.
- Linux VMs are good for testing. More serious runs ask for Linux workstations. You can create dual boot machines which have both Linux and Windows OS installed. Again, this will work for small jobs.
- More serious multi-layer networks require GPU machines of considerable power.
- You could build workstations with more powerful GPU cards (one, two or many)
- Another option is to work on Amazon AWS GPU machines, Google Cloud GPU machines, or MS Azure GPU machines.
- Google also offers: Google Colab - a free Jupyter Notebook environment with GPUs.

https://colab.research.google.com/notebooks/welcome.ipynb

Puma Indians

- Pima Indians onset of diabetes dataset is a standard machine learning dataset available for free download from the UC Irvine Machine Learning repository.
- It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within 5 years.
- This is a binary classification problem (onset of diabetes as 1 or not as 0).
- The input variables that describe each patient are numerical and have varying scales.
- The eight attributes for the dataset are:

1. Number of times pregnant.
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skin fold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index.
7. Diabetes pedigree function.
8. Age (years).
9. Class, onset of diabetes within 5 years.

Data

- Data come in a csv file: `pima-indians-diabetes.csv.` The first 8 rows of that data set are presented below:

6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	26	1
10	115	0	0	0	35.3	0.134	29	0

- The dataset has 768 rows (samples).

Pima Indians, Keras Binary Classification

```
# Create your first MLP in Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)# load Pima Indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu’))
model.add(Dense(8, activation='relu’))
model.add(Dense(1, activation='sigmoid’))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy’])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)print("\n%s: %.2f%%" % (model.metrics_names[1],
scores[1]*100)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Run the model

- Our code is contained in the file first_mlp.py. We run it as:

```
$ python first_mlp.py.
```

- The result is a table of the values of loss function and model accuracy for 150 epochs. An epoch is set of forward and back propagation passes which all available samples. The results of the last several epochs are presented below. Noticed that our simple model with a small dataset achieved an accuracy of ~ 80%

```
Epoch 143/150
768/768 [==============================] - 0s 423us/step - loss: 0.4753 - acc: 0.7721
Epoch 144/150
768/768 [==============================] - 0s 418us/step - loss: 0.4763 - acc: 0.7747
Epoch 145/150
768/768 [==============================] - 0s 414us/step - loss: 0.4867 - acc: 0.7682
Epoch 146/150
768/768 [==============================] - 0s 396us/step - loss: 0.4919 - acc: 0.7747
Epoch 147/150
768/768 [==============================] - 0s 397us/step - loss: 0.4826 - acc: 0.7839
Epoch 148/150
768/768 [==============================] - 0s 397us/step - loss: 0.4699 - acc: 0.7786
Epoch 149/150
768/768 [==============================] - 0s 410us/step - loss: 0.4762 - acc: 0.7643
Epoch 150/150
768/768 [==============================] - 0s 424us/step - loss: 0.4753 - acc: 0.7812
768/768 [==============================] - 0s 106us/step
```

Classifying Movie Reviews

- As a more complex example of use or Keras, we will examine "IMDB dataset", a set of 50,000 highly-polarized reviews from the Internet Movie Database.
- They are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.
- Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.
- The following code requires `numpy 1.16.1.` Downgrade numpy:

```
$ pip install numpy=1.16.1
import numpy
import keras
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels)=
imdb.load_data(num_words=10000)
Downloading data from https://s3.amazonaws.com/text-datasets/imdb.npz
17465344/17464789 [==============================] - 3s 0us/step
```

- The argument num_words=10000 means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.
- The variables train_data and test_data are lists of reviews, each review being a list of word indices (encoding a sequence of words). train_labels and test_labels are lists of 0s and 1s, where 0 stands for "negative" and 1 stands for "positive":

Movie Reviews

- To see the content of `train_data` and `train_labels`, we could ask for values in the first sample, with index 0:

```
train_data[0]
[1, 14, 22, 16, 43,

train_labels[0]
1
```

- We restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

```
max([max(sequence) for sequence in train_data])
9999
```

- To decode the review, see the text in English, we have to pay attention to the fact that indices are offset by 3. 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".

Decoded Review

- Here is how you can quickly decode one of these reviews back to English words

```
# word_index is a dictionary mapping words to an integer index
word_index = imdb.get_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])
# We decode the review; indices are offset by 3
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?')
   for i in train_data[0]]
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/imdb_word_index.json 1646592/1641221
[==============================] - 1s 1us/step
```

```
decoded_review
```

```
"? this film was just brilliant casting location scenery story direction everyone's really suited the part they
played and you could just imagine being there robert ? is an amazing actor and now the same being director ?
father came from the same scottish island as myself so i loved the fact there was a real connection with this
film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as
soon
```

Preparing the Data

- We have to turn our lists of number into tensors.
- There are two ways we could do that:
  - We could pad our lists so that they all have the same length, and turn them into an integer tensor of shape (samples, word_indices).
  - We could one-hot-encode our lists to turn them into vectors of 0s and 1s. Concretely, this would mean for instance turning the sequence [3, 5] into a 10,000-dimensional vector that would be all-zeros except for indices 3 and 5, which are set to ones. Then we could use as first layer in our network a Dense layer, capable of handling floating point vector data.
- We will go with the latter solution. We will vectorize our data
- Our original data look like this:

```
train_data
```

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838,
112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447,
4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, . .
. . . . ., 19, 178, 32]),
list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119,
954, 189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116,
9, 35, 8163, 4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 100 . . . . . ]),
```

One-hot-encoding

- The following will replace lists with vectors of 0-s and 1-s

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

- Every review (document) is now an array of floating point 0-s and 1-s.

```
x_train[0]
array([0., 1., 1., ..., 0., 0., 0.])
```

- We also vectorize labels

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Building the network

- Our input data is simple vectors, and our labels are scalars (1s and 0s).
- A type of network that performs well on such a problem is a simple stack of fully-connected (Dense) layers with `relu` activations: `Dense(16, activation='relu').`
- We choose a network with 3 layers. The first two layers have 16 neurons. The first layer accepts 10,000 units long vectors. The last layer is a single neuron with a simple sigmoid since we want to make a decision: a review is positive (1) or negative (0).

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- We need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss function. It is not the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you are dealing with models that output probabilities.

Optimizers

- Keras documentation is not very verbose about various optimizers Keras supports.
- You could take a look at this page:

https://keras.io/optimizers/

- There are many articles on the Internet and Wikipedia that clarify major features of most popular optimizers. For example

http://ruder.io/optimizing-gradient-descent/index.html#rmsprop

Loss function and Optimizer

- Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the binary_crossentropy loss. It isn't the only viable choice: you could use, for instance, mean_squared_error.
- But crossentropy is usually the best choice when you are dealing with models that output probabilities.

```
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
Keras supports many standard optimizers: sgd, adam, rmsprop
```

Testing Results, Validating the Approach

- In order to monitor during training the accuracy of the model on data that it has never seen before, we will create a "validation set" by setting apart 10,000 samples from the original training data:

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

- We will now train our model for 20 epochs (20 iterations over all samples in the x_train and y_train tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the validation_data argument:

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Train on 15000 samples, validate on 10000 samples Epoch 1/20 15000/15000
[==============================] - 5s 308us/step - loss: 0.5834 - acc: 0.7413 - val_loss: 0.4437 - val_acc:
0.8386 Epoch 2/20 15000/15000 [==============================] - 3s 212us/step - loss: 0.3364 - acc:
0.8907 - val_loss: 0.3136 - val_acc: 0.8817  . . . .

.   .   .   .   .

Epoch 20/20 15000/15000 [==============================] - 4s 289us/step - loss: 0.0043 - acc: 1.0000 -
val_loss: 0.5788 - val_acc: 0.8699
```

History Object

- The call to model.fit() returns a History object. This object has a member history, which is a dictionary containing data about everything that happened during training. Let us examine it:

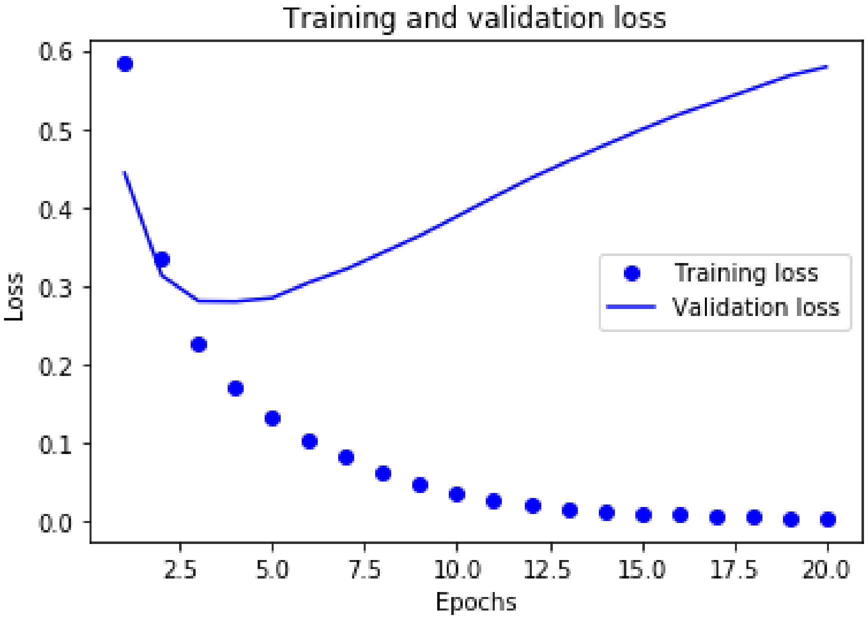
```
history_dict = history.history
history_dict.keys()
```

```
dict_keys(['val_acc', 'acc', 'val_loss', 'loss'])
```

- It contains 4 entries: one per metric that was being monitored, during training and during validation.
- We could use Matplotlib to plot the training and validation loss side by side, next slide, as well as the training and validation accuracy, on the following slide.

Training and Validation Loss

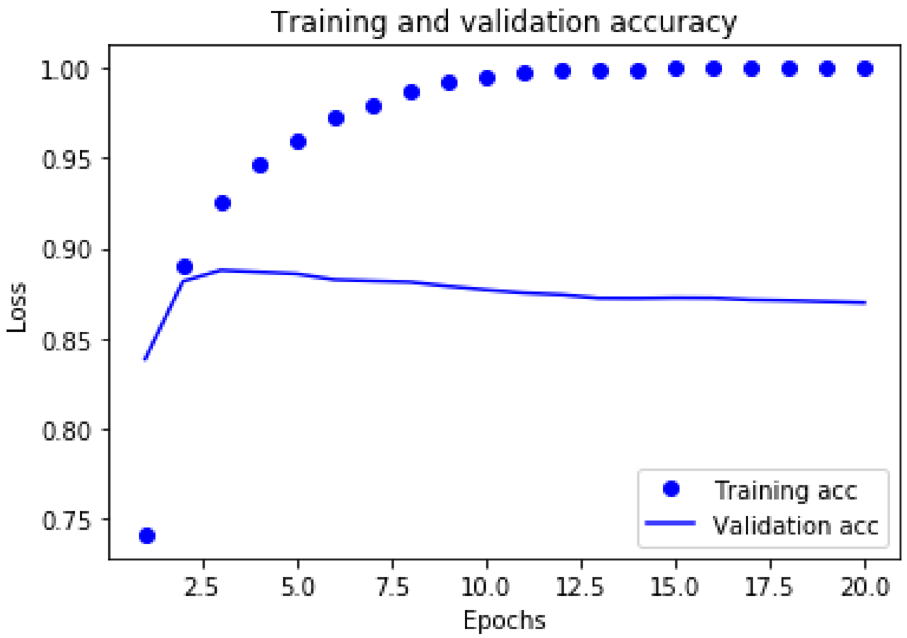
```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)
```



```
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Training and Validation Accuracy

```
plt.clf()   # clear figure
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



Overfitting

- As you can see, the training loss decreases with every epoch and the training accuracy increases with every epoch. That's what you would expect when running gradient descent optimization -- the quantity you are trying to minimize should get lower with every iteration.
- But that isn't the case for the validation loss and accuracy: they seem to peak at the third or fourth epoch.
- A model that performs well on the training data isn't necessarily a model that will do better on data it has never seen before.
- What you are seeing is "overfitting": after the second epoch, we are over-optimizing on the training data, and we ended up learning representations that are specific to the training data and do not generalize to data outside of the training set.
- In this case, to prevent overfitting, we could simply stop training after three epochs. In general, there is a range of techniques you can leverage to mitigate overfitting, most notably the regularization.

Train network on “optimal” number of epochs

- We will train the network for four epochs, then evaluate it on our test data:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
Epoch 1/4 25000/25000 [==============================] - 4s 165us/step - loss: 0.4749 - acc: 0.8217
Epoch 2/4 25000/25000 [==============================] - 3s 138us/step - loss: 0.2658 - acc: 0.9097
Epoch 3/4 25000/25000 [==============================] - 4s 143us/step - loss: 0.1982 - acc: 0.9299
Epoch 4/4 25000/25000 [==============================] - 4s 159us/step - loss: 0.1679 - acc: 0.9404
25000/25000 [==============================] - 4s 171us/step

Results
[0.3231498811721802, 0.87352]
```

- Our fairly naive approach achieves an accuracy of 87.35%. With state-of-the-art approaches, one should be able to get close to 95%.