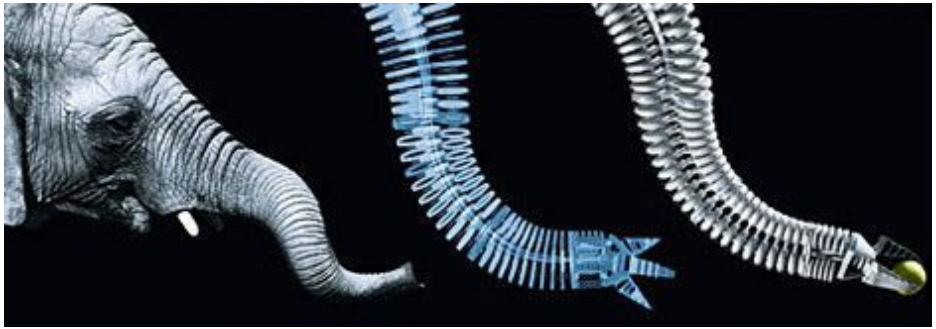
Artificial Neural Networks

Lecture 10

Today: Learning Objectives

1. Learn about the story behind the Artificial Neural Network
2. Know the Linear Threshold Unit and Perceptron
3. Expand to the Multilayer Perceptron (MLP)
4. Understand how backpropagation works
5. Finetune the neural networks

1. Motivations and History



Nature has inspired many inventions!

Brain’s architecture for an intelligent machine

- Key idea of artificial neural networks (ANNs)
- ANNs are at the very **core** of Deep Learning
- Versatile, powerful, scalable
  - Classifying billions of images (**Google** Images)
  - Powering speech recognition (**Apple** Siri)
  - Recommending the best videos to millions of user (**YouTube**)

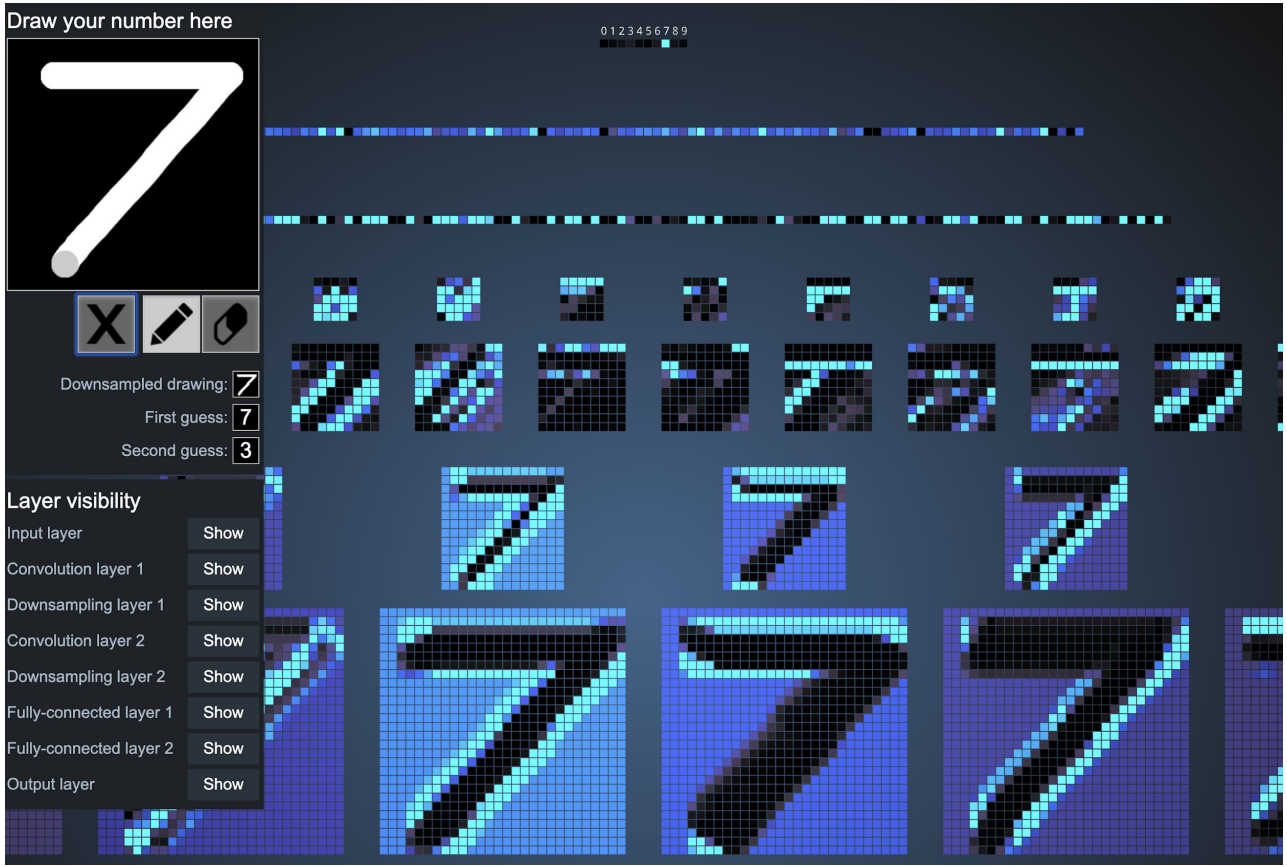
On ML by Sundar Pichai, CEO of Google

"Machine Learning is a core, transformative way by which we're rethinking how we're doing everything. We're thoughtfully applying it across all of our products, be it search, ads, YouTube, or Play. And we're in the early days, but you'll see us -- in a systematic way -- apply machine learning in all these areas" - Sundar Pichai, 2015

What neural networks has achieved so far?

- Near human-level image classification
- Near human-level speech recognition
- Near human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Near-human-level autonomous driving
- Improved ad targeting
- Improved search results
- Ability to answer natural language questions
- Super-human Go playing
- And much more...

Demo: Neural Networks on MNIST



From Biological to Artificial Neurons

ANNs have been around for a while (proposed by McCulloch and Pitts in 1943)

- Early success of ANNs led to belief of creating intelligent machines (**FAILS**)
- 1960s -1970s: **Long** dark era while this belief would go **unfulfilled**
- 1980s: **revival** of interest in ANNs as network architecture and training
- 1990s - 2000s: **SVMs** were favored by most researchers
- 2010s: another wave of **interest** in ANNs.

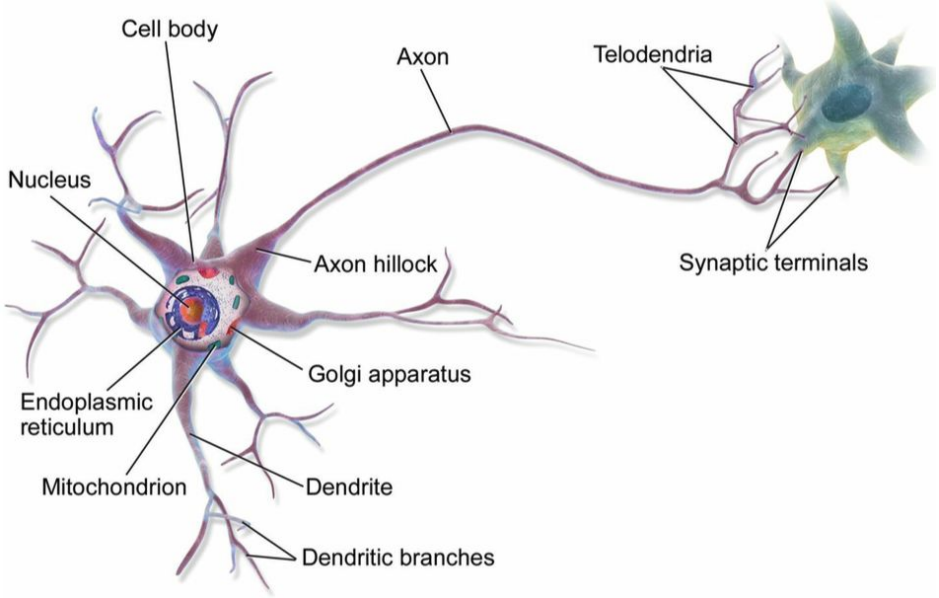
Would this one last?

A few reasons why this time will have impact

- Huge **quantity of data** available to train (large and complex problems)
- Tremendous increase in **computing power** (Thanks to Moore's Law)
- Improved training **algorithms** (tweaks and tunings)
- Some **theoretical limitation** of ANNs turns out to be benign in practice
- Seems to enter a virtuous circle of **funding and progress** (headline news, products)

Biological Neurons

- Unusual looking cells found in animal cerebral cortex (brain)
- Receive short electrical impulses (signals) from other neurons via synapses.
- When receives a **sufficient** number of signals, **fires its own signals**



Biological Neural Networks (BNN)

Individual neurons seems simple, but they are organized in a vast network of billions of neurons in consecutive layers, each neuron connected to thousands of other neurons.

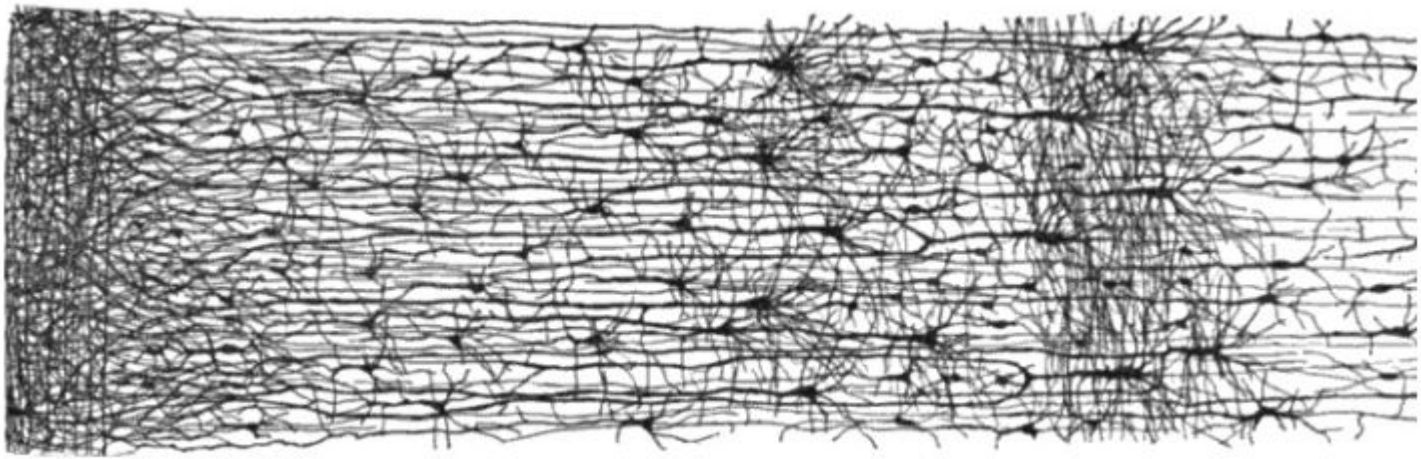


Figure 10-2. Multiple layers in a biological neural network (human cortex)[5]

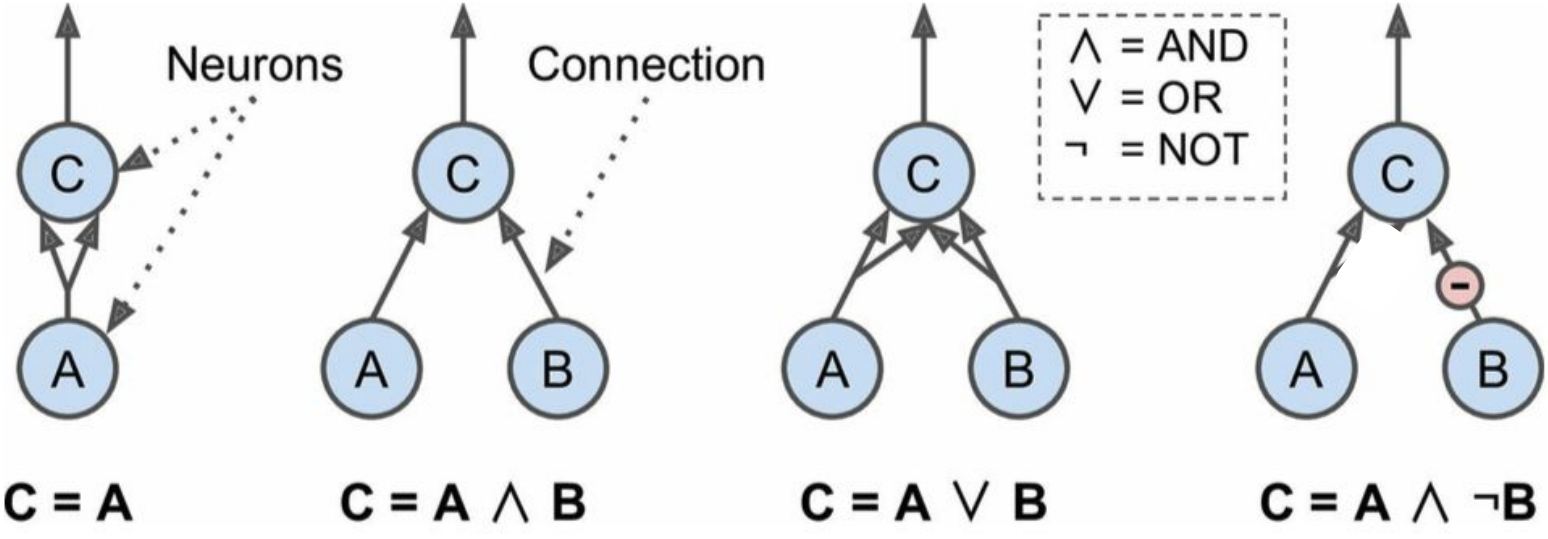
2. Perceptron

Logical Computations with Neurons

Artificial Neuron: has one or more binary (on/off) inputs and one binary output.

An ANN of a few neurons can perform various logical computations.

Assuming a neuron can be activated when both of its input are active.



The Linear Threshold Unit (LTU)

- LTU is based on an artificial neuron
- LTU has inputs and output of numbers (instead of binary), and each input connection is associated with a weight.
- LTU computes a weighted sum of its inputs:

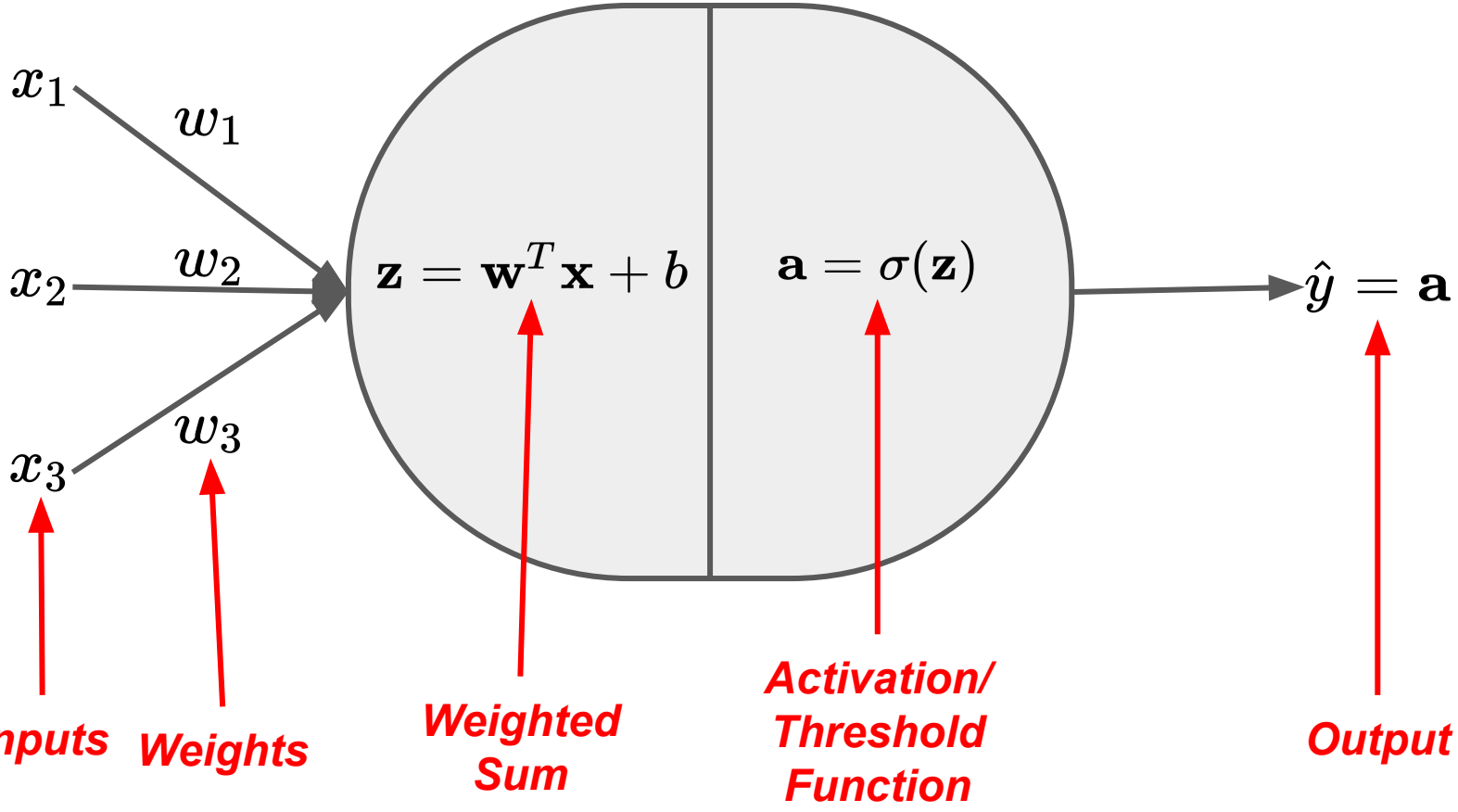
$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$$

- LTU then applied a step function and outputs:

$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

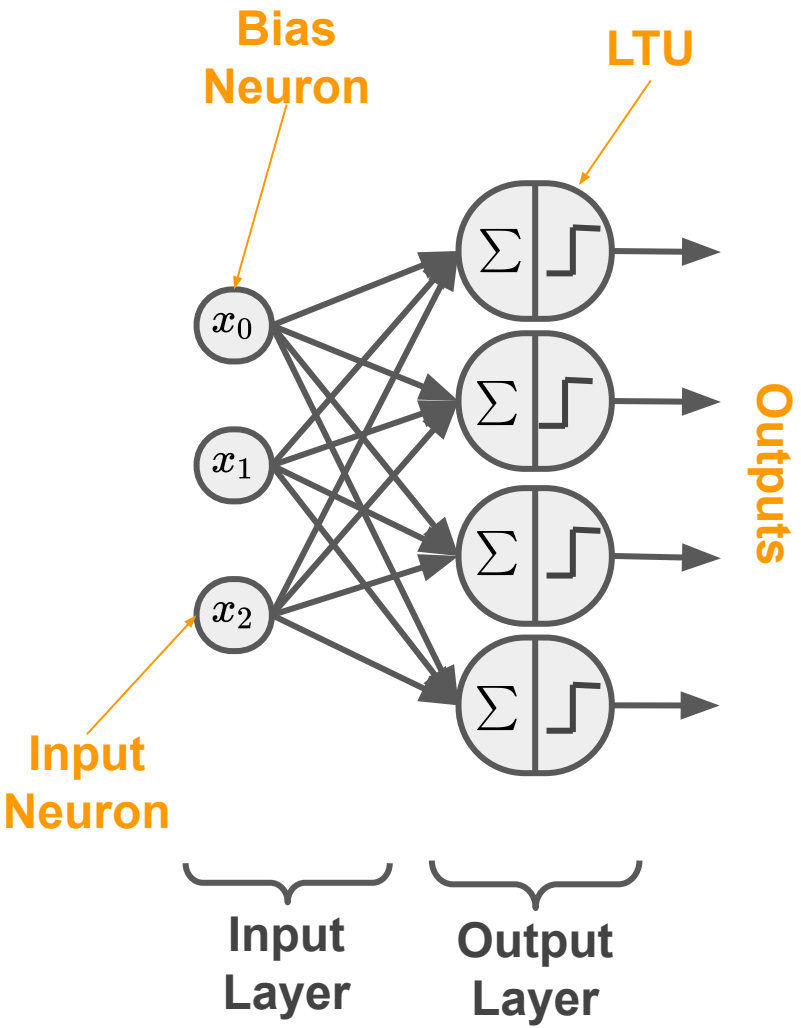
We've seen this already in the course!

Graphical representation of a LTU



The Perceptron

- One of the simplest ANN architecture (invented in 1957 by Frank Rosenblatt)
- Composed of a layer of LTUs, with each neurons connected to all the inputs
- An extra bias feature is generally added (x_0 =1) : bias neuron



Training a Perceptron

"Cells that fire together, wire together" - Hebb's rule

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

- $w_{i,j}$ is the connection weight between i^{th} input neuron and j^{th} output neuron
- x_i is the i^{th} input value of the current training sample
- $\hat{y}_j$ is the output of the j^{th} neuron
- y_j is the target output of j^{th} output neuron
- η is the learning rate

What does this learning algorithm remind you of?

Code example

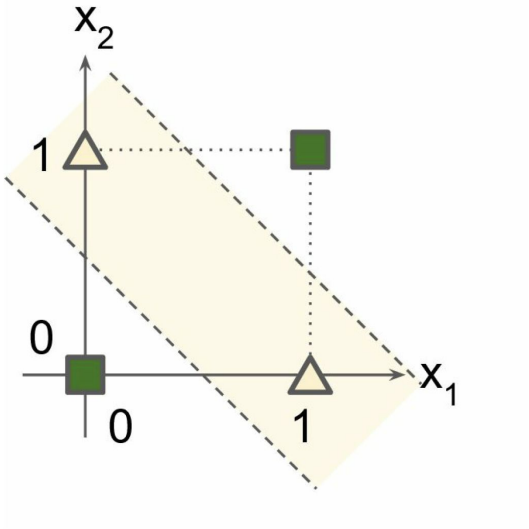
Perceptron makes predictions based on a hard threshold (not class probability)

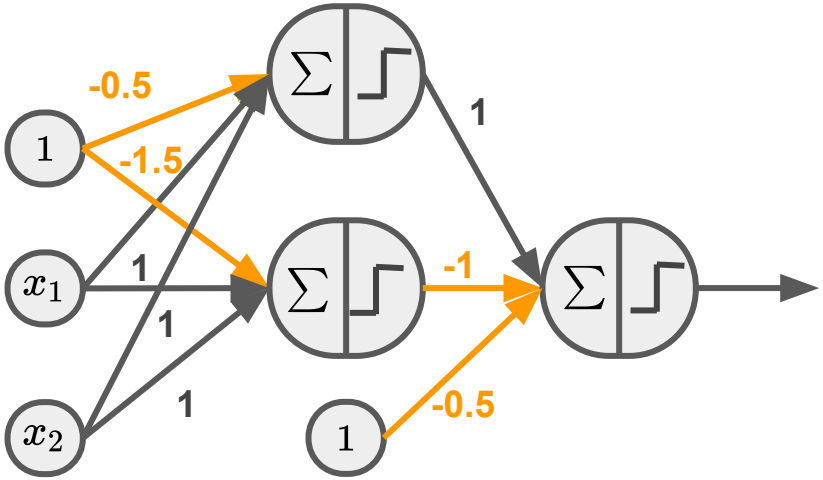
Also, it only works if the training samples are linearly separate.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

XOR Classification problem

INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0





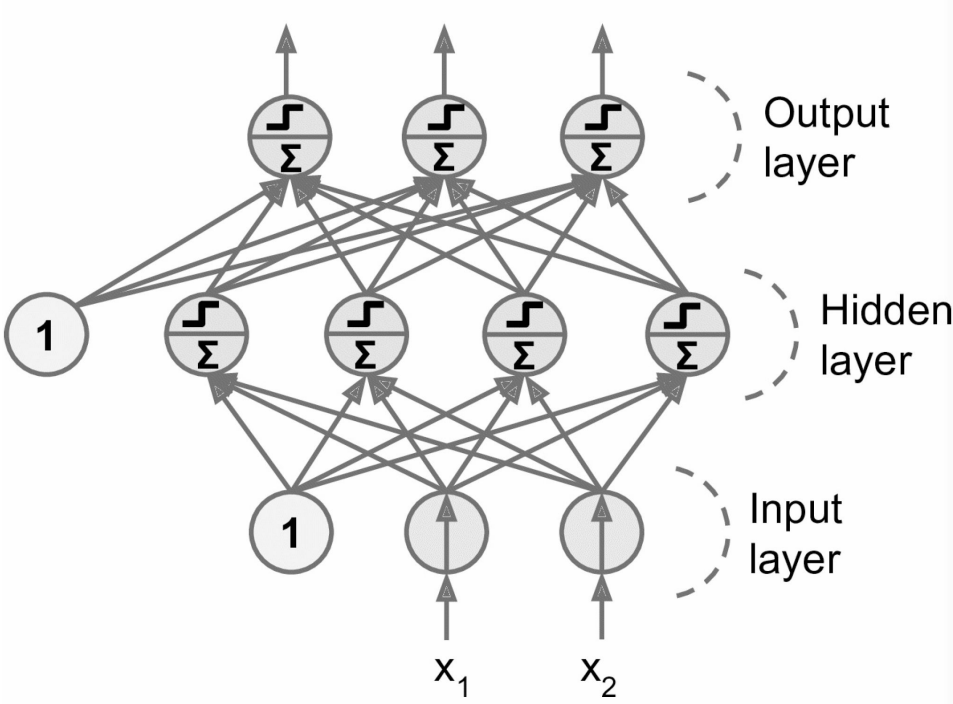
XOR is not linearly separable, cannot be solved with a single perceptron (LTU)

→ **stacking** multiple Perceptrons can overcome this limitation

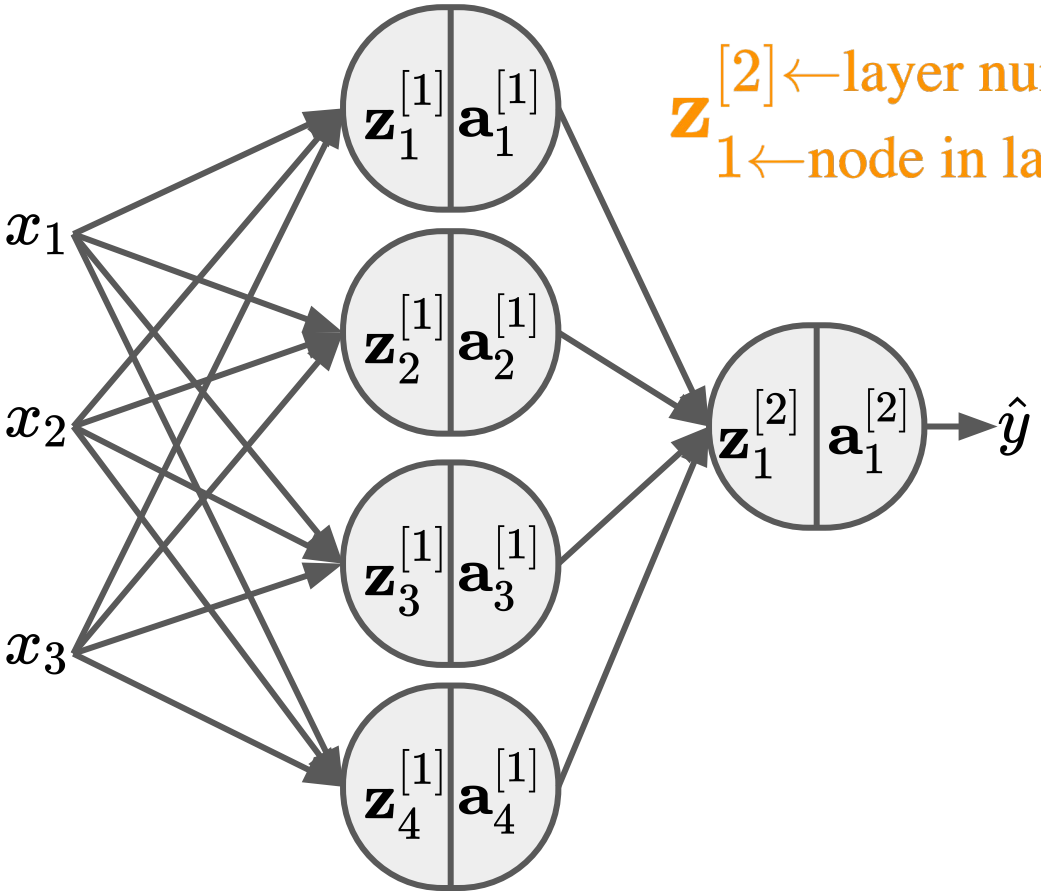
3. Multilayer Perceptron (MLP)

Multi-Layer Perceptron (MLP)

- Composed of:
  - 1 (passthrough) input layer
  - 1 or more layers of LTUs (called hidden layers)
  - 1 final layer of LTU (output layer).
- When an ANN has 2+ hidden layer, it is called Deep Neural Network (DNN)



Representation of a MLP



$$\mathbf{z}_1^{[1]} = \mathbf{w}_1^{[1]T}\mathbf{x} + b_1^{[1]}$$
$$\mathbf{a}_1^{[1]} = \sigma\left(\mathbf{z}_1^{[1]}\right)$$

$$\mathbf{z}_2^{[1]} = \mathbf{w}_2^{[1]T}\mathbf{x} + b_2^{[1]}$$
$$\mathbf{a}_2^{[1]} = \sigma\left(\mathbf{z}_2^{[1]}\right)$$

$$\mathbf{z}_3^{[1]} = \mathbf{w}_3^{[1]T}\mathbf{x} + b_3^{[1]}$$
$$\mathbf{a}_3^{[1]} = \sigma\left(\mathbf{z}_3^{[1]}\right)$$

$$\mathbf{z}_4^{[1]} = \mathbf{w}_4^{[1]T}\mathbf{x} + b_4^{[1]}$$
$$\mathbf{a}_4^{[1]} = \sigma\left(\mathbf{z}_4^{[1]}\right)$$

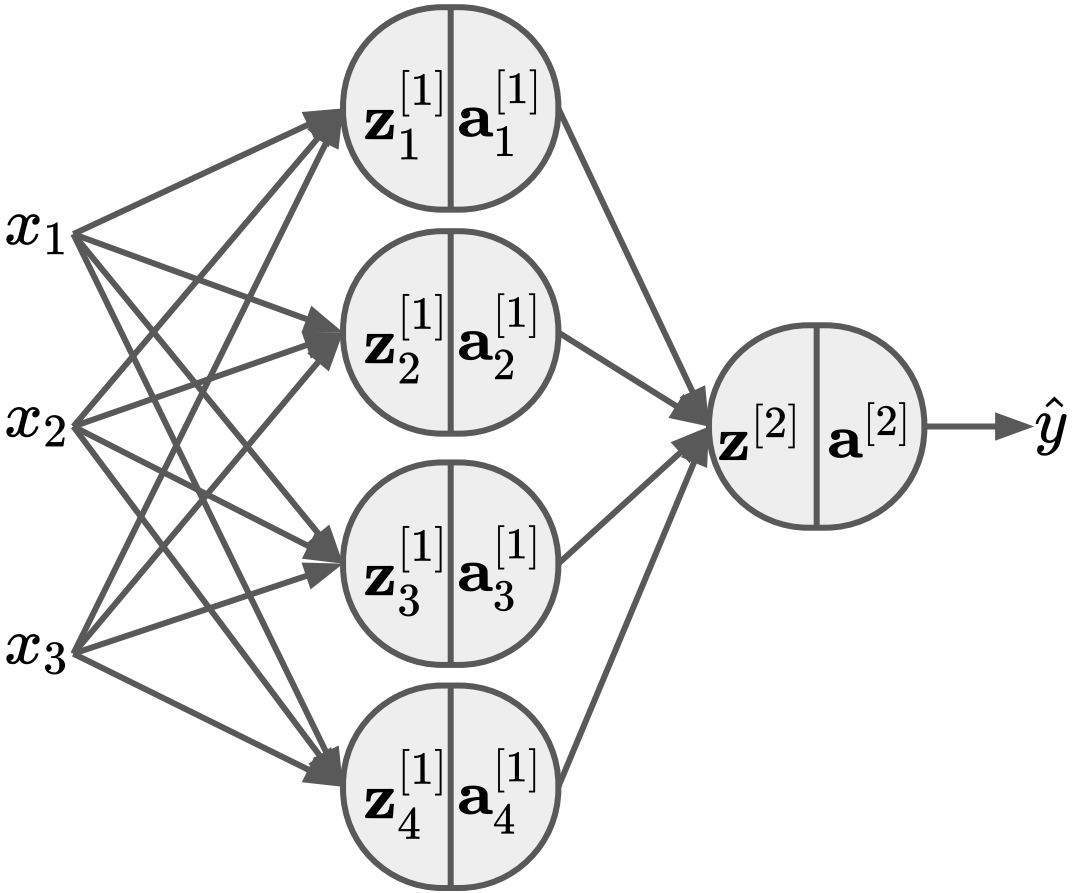
Vectorizing by stacking them vertically

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix} x + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \qquad \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \begin{bmatrix} \sigma(z_1^{[1]}) \\ \sigma(z_2^{[1]}) \\ \sigma(z_3^{[1]}) \\ \sigma(z_4^{[1]}) \end{bmatrix}$$

$z^{[1]}$ $W^{[1]}$ $b^{[1]}$ $a^{[1]}$ $\sigma(z^{[1]})$

$$z^{[1]} = W^{[1]} x + b^{[1]} \qquad a^{[1]} = \sigma(z^{[1]})$$

Dimensionality of vectorized components



$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$$

4x1 4x3 3x1 4x1

$$\mathbf{a}^{[1]} = \sigma\left(\mathbf{z}^{[1]}\right)$$

4x1 4x1

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$

1x1 1x4 4x1 1x1

$$\mathbf{a}^{[2]} = \sigma\left(\mathbf{z}^{[2]}\right)$$

1x1 1x1

Expanding for multiple examples

$$\mathbf{x} \longrightarrow \mathbf{a}^{[2]} = \hat{y}$$

$$\begin{aligned}
\mathbf{x}^{(1)} &\longrightarrow \mathbf{a}^{[2](1)} = \hat{y}^{(1)} \\
\mathbf{x}^{(2)} &\longrightarrow \mathbf{a}^{[2](2)} = \hat{y}^{(2)} \\
&\vdots \\
\mathbf{x}^{(m)} &\longrightarrow \mathbf{a}^{[2](m)} = \hat{y}^{(m)}
\end{aligned}$$

for $i = 1$ to m

$$\mathbf{z}^{[1](i)} = \mathbf{W}^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$

$$\mathbf{a}^{[1](i)} = \sigma\left(\mathbf{z}^{[1](i)}\right)$$

$$\mathbf{z}^{[2](i)} = \mathbf{W}^{[2]}\mathbf{a}^{[1](i)} + \mathbf{b}^{[2]}$$

$$\mathbf{a}^{[2](i)} = \sigma\left(\mathbf{z}^{[2](i)}\right)$$

k is # of nodes in the layer

Vectorizing for multiple examples

$$\underset{\text{nxm}}{\mathbf{X}} = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \quad \underset{\text{kxm}}{\mathbf{Z}^{[1]}} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix} \quad \underset{\text{kxm}}{\mathbf{A}^{[1]}} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ | & | & & | \end{bmatrix}$$

for $i = 1$ to m

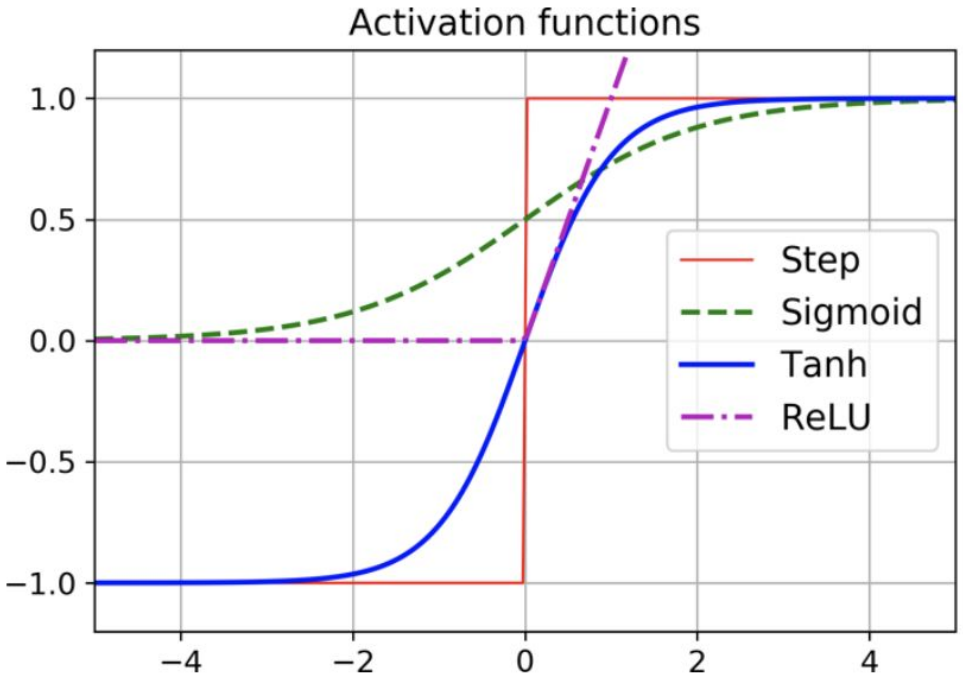
$$\mathbf{z}^{[1](i)} = \mathbf{W}^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1](i)} = \sigma(\mathbf{z}^{[1](i)})$$
$$\mathbf{z}^{[2](i)} = \mathbf{W}^{[2]}\mathbf{a}^{[1](i)} + \mathbf{b}^{[2]}$$
$$\mathbf{a}^{[2](i)} = \sigma(\mathbf{z}^{[2](i)})$$

$\Implies$

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]}$$
$$\mathbf{A}^{[1]} = \sigma(\mathbf{Z}^{[1]})$$
$$\mathbf{Z}^{[2]} = \mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{b}^{[2]}$$
$$\mathbf{A}^{[2]} = \sigma(\mathbf{Z}^{[2]})$$

Activation function a = σ(z)

There are a number of activation functions available:



Step: $a = 1 \text{ if } z > 0,$
$0 \text{ if } z < 0$

Sigmoid: $a = \frac{1}{1+e^{-z}}$

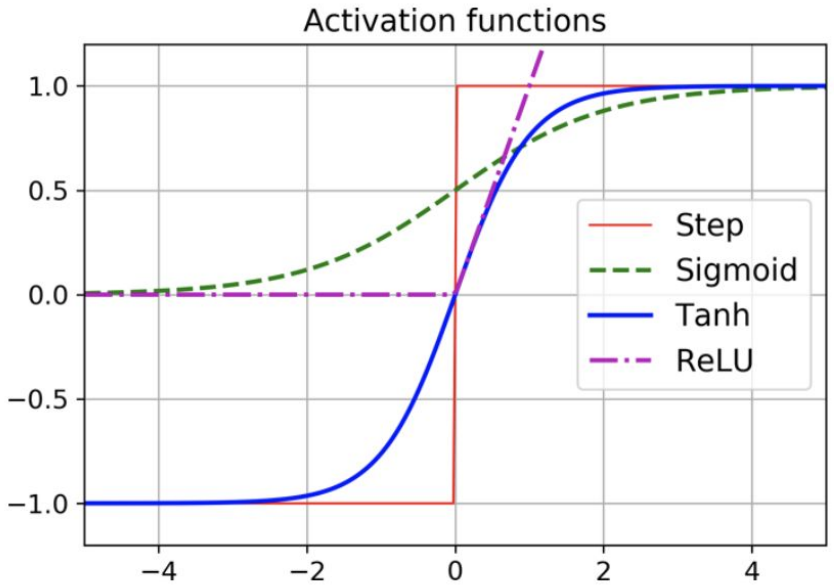
Tanh: $a = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

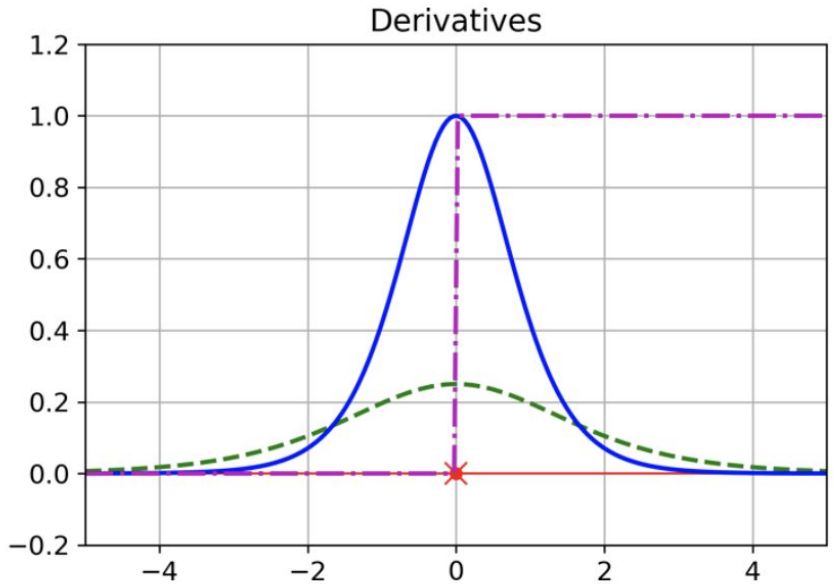
ReLU: $a = \max(0, z)$

Activation functions

Step Function has zero derivative → does not work with Gradient Descent

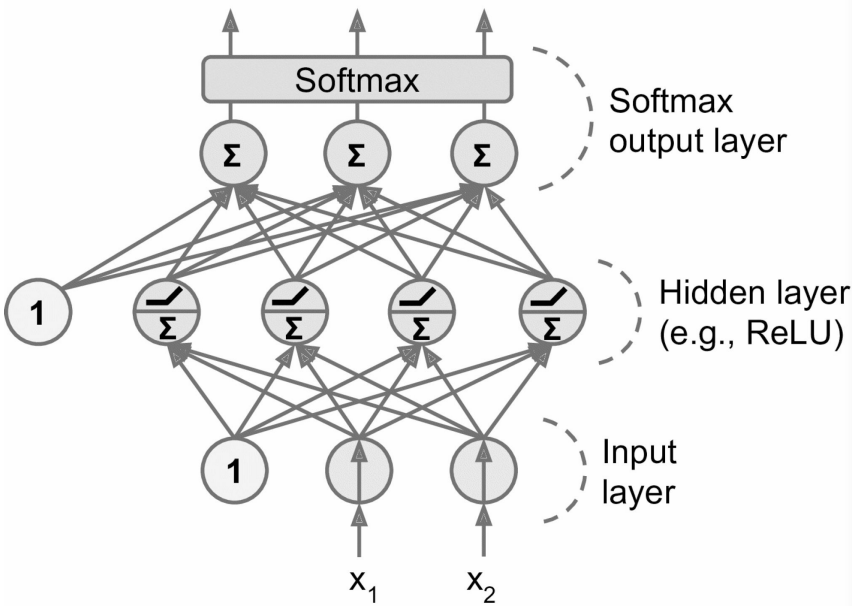
Use Sigmoid function (and other below) with well-defined non-zero derivative → allow Gradient Descent to make progress





Activation Function for classification

Use **softmax** function to produce the estimated probability of the corresponding class. Note that in this case, signal flows only in one direction (inputs → output) → feedforward neural network (FNN)

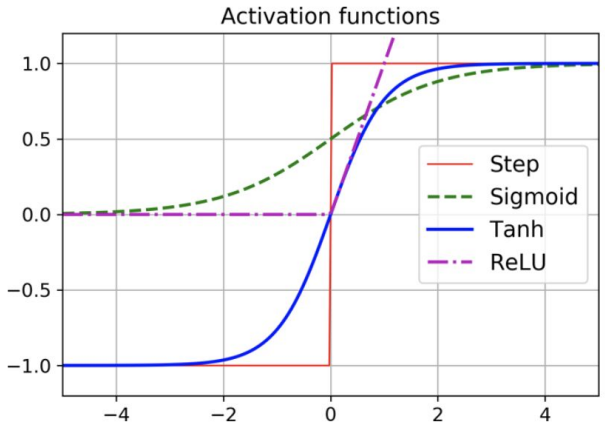


$$\hat{p}_k^{(i)} = P(y^{(i)} = k | \mathbf{x}^{(i)}; \theta)$$

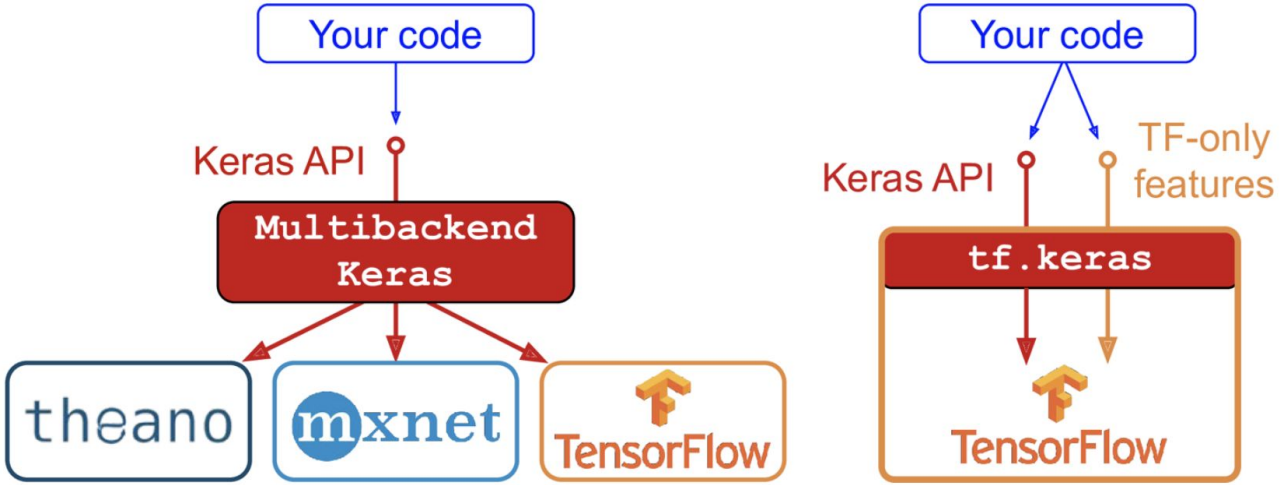
$$= \frac{\exp({\theta_{(k)}}^T \mathbf{x}^{(i)})}{\sum_{j=1}^{K} \exp({\theta_{(j)}}^T \mathbf{x}^{(i)})}$$

Tips on Activation Functions

- **ReLU** in hidden layers is faster to compute
- **Step** function does not work with Gradient Descent
- **Sigmoid (Logistic)** and **Hyperbolic Tangent** function saturate at 1
- For classification tasks, **Softmax** function is a good choice
- For regression tasks, **no need** for activation function



Getting started on Tensorflow and Keras




```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

Fashion MNIST Dataset

- Same format as MNIST (70,000 grayscale images of 28x28 pixels, with 10 classes)
- Images represent fashion items rather than handwritten digits
- More challenging than MNIST

Code Demo on Tensorflow

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
```

```
>>> model.evaluate(X_test, y_test)
10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

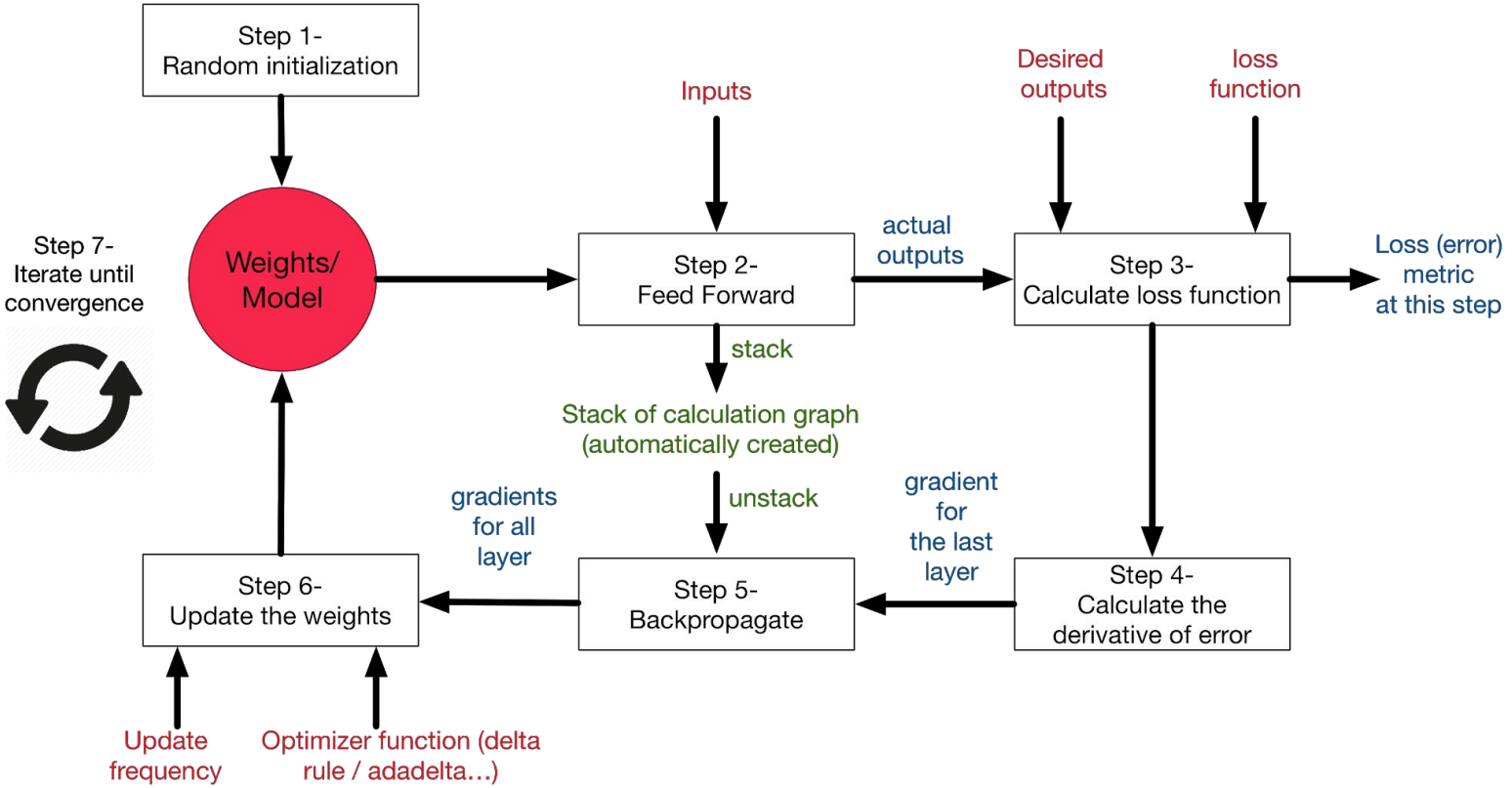
4. Backpropagation

Training MLPs

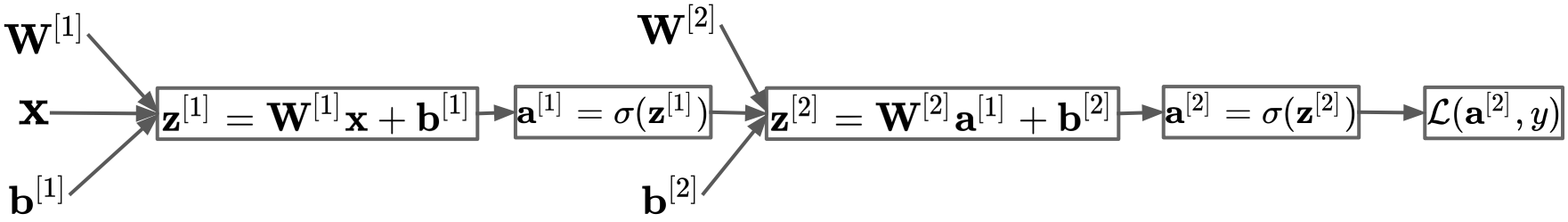
Using **backpropagation** (groundbreaking article by Rumelhart, 1986):

- **Forward Pass:** Feeds a training instance to the network and computes the output of every neuron
- **Backward Pass:** Measures the network error, and computes the error contributions came from each neuron in previous hidden layer
- **Gradient Descent:** Runs Gradient Descent on all connection weights using the measured error

Feedback mechanism for learning the weights



Forward pass (4 equations)



$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$$

$$\mathbf{a}^{[1]} = \sigma(\mathbf{z}^{[1]})$$

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$

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

Gradient Descent

Parameters: $\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \mathbf{b}^{[2]}$

$a^{[2]}$

Cost function: $\mathbf{J}(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \mathbf{b}^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathbf{L}(\hat{y}^{(i)}, y^{(i)})$

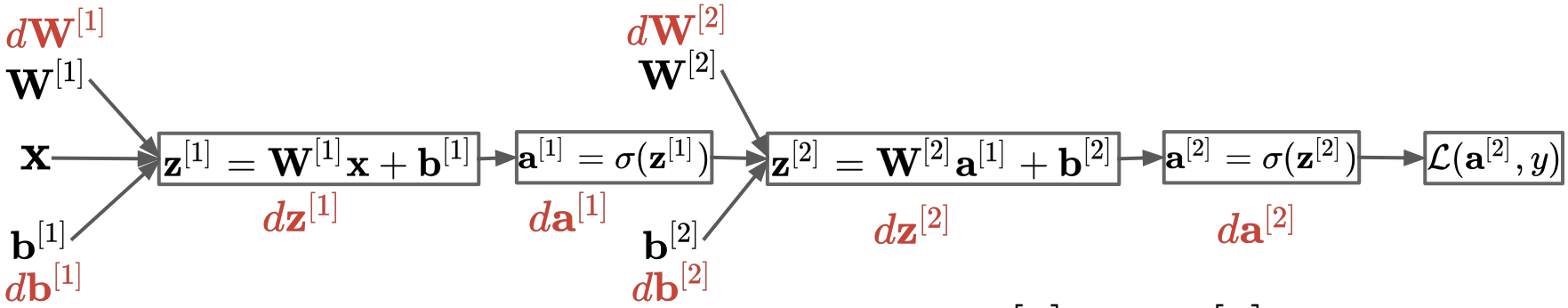
Gradient Descent: `repeat:`

`Compute prediction: y^(i), i = 1..m`

$$\mathbf{W}^{[1]} = \mathbf{W}^{[1]} - \alpha d\mathbf{W}^{[1]}$$

$$\mathbf{b}^{[1]} = \mathbf{b}^{[1]} - \alpha d\mathbf{b}^{[1]}$$

Backward Pass (8 equations)



$$d\mathbf{a}^{[1]} = \mathbf{W}^{[2]T} d\mathbf{z}^{[2]}$$

$$d\mathbf{z}^{[1]} = \mathbf{W}^{[2]T} d\mathbf{z}^{[2]} * \sigma'(\mathbf{z}^{[1]})$$

$$d\mathbf{W}^{[1]} = d\mathbf{z}^{[1]} \mathbf{x}^T$$

$$d\mathbf{b}^{[1]} = d\mathbf{z}^{[1]}$$

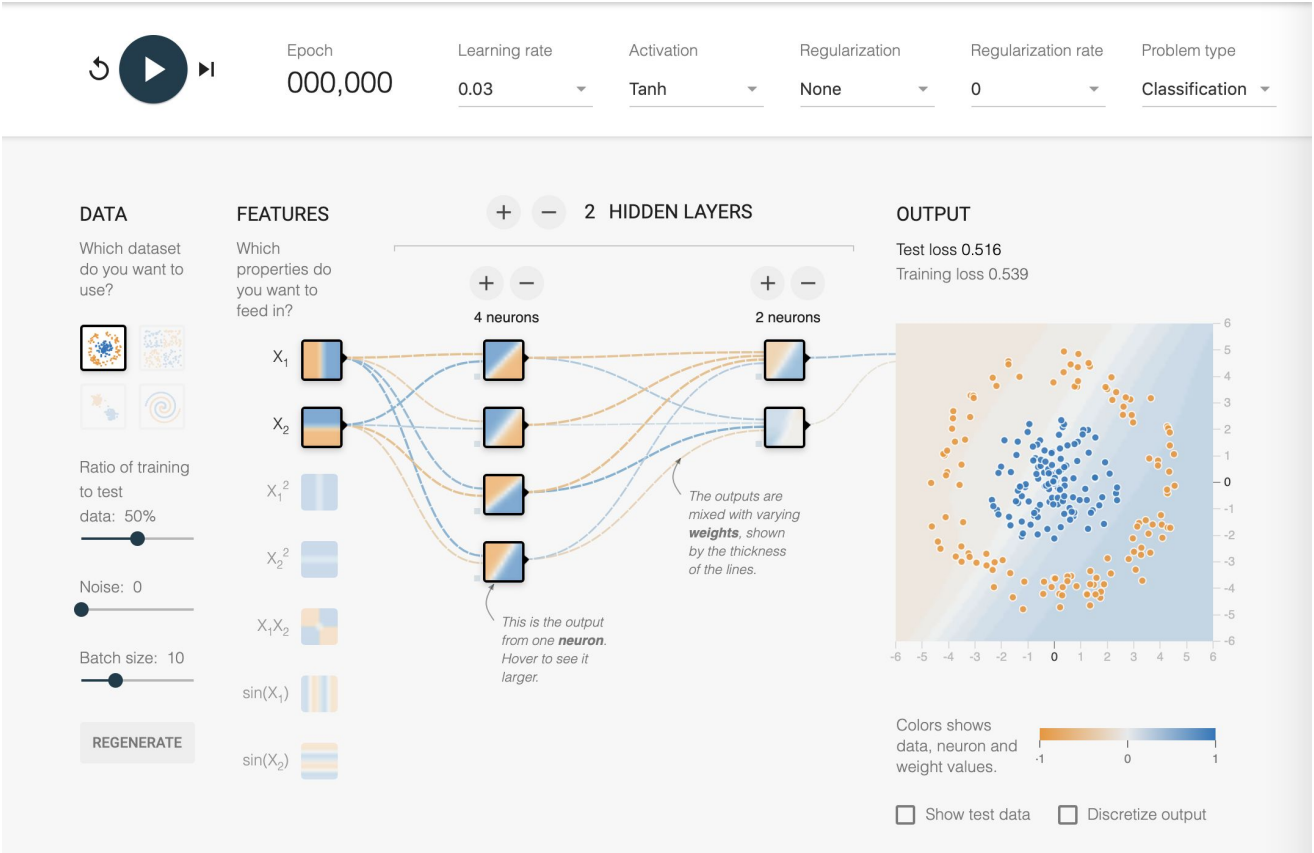
$$d\mathbf{a}^{[2]} = \mathbf{a}^{[2]} - \mathbf{y}$$

$$d\mathbf{z}^{[2]} = d\mathbf{a}^{[2]} * \sigma'(\mathbf{z}^{[2]})$$

$$d\mathbf{W}^{[2]} = d\mathbf{z}^{[2]} \mathbf{a}^{[1]T}$$

$$d\mathbf{b}^{[2]} = d\mathbf{z}^{[2]}$$

TensorFlow Playground



https://playground.tensorflow.org

5. Fine-tuning the Neural Networks

Fine-Tuning Network Hyperparameters

Many hyperparameters to tweak, still be considered as a **black art** to find the perfect hyperparameters:

- Number of layers
- Number of neurons per layer
- Type of activation function to use at each layer
- Weight initialization logic
- and much more

How do you know which combination of hyperparameters is the best for your task?

Number of Hidden Layers

- One hidden layer can model complex functions if it has enough neurons
- Deep networks can model complex functions using exponentially fewer neurons than shallow nets, making them faster to train
- Real-world data often structured in such a hierarchical way:
  - Lower hidden layers model low-level structures (line segment, shape and orientation)
  - Intermediate layers combine low-level structures to intermediate structure (squares, circles)
  - Highest hidden layers combine intermediate structure to model high-level (faces)
- Better generalization if only changes the high hidden layer (ie. hairstyles)
- Start with two hidden layers (98% accuracy on MNIST). Careful for overfitting!
- Use other pretrained state-of-the-art network to perform a similar task

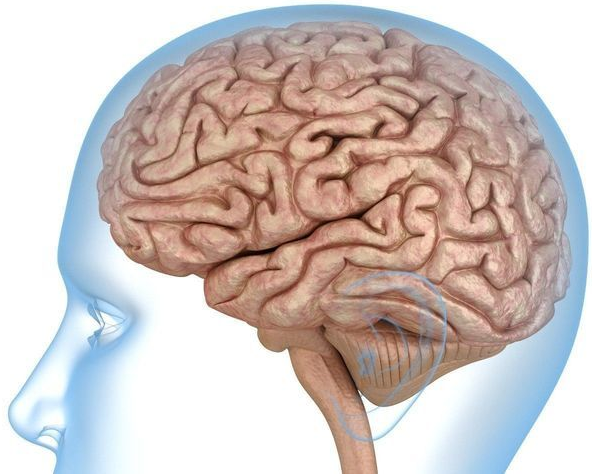
Number of Neurons per Layer

- Obviously, it depends on the type of input and output your task requires.
- MNIST task requires 28 x 28 = 784 input neuron and 10 output neurons
- For hidden layer, a common practice is to form a funnel, with fewer and fewer neurons at each layer
- Rationale: Many low-level features can coalesce into far fewer high-level features (for MNIST, 300 neurons on the 1st hidden layer and 100 on 2nd layer)
- Simpler approach: pick model with more layer and neuron that needed, than use early stopping.

Today: Learning Objectives

- ✓ Learn about the story behind the Artificial Neural Network
- ✓ Know the Linear Threshold Unit and Perceptron
- ✓ Expand to the Multilayer Perceptron (MLP)
- ✓ Understand how backpropagation works
- ✓ Finetune the neural networks

Next: Dive deeper (no pun intended) into Deep Neural Nets



Bonus Slides

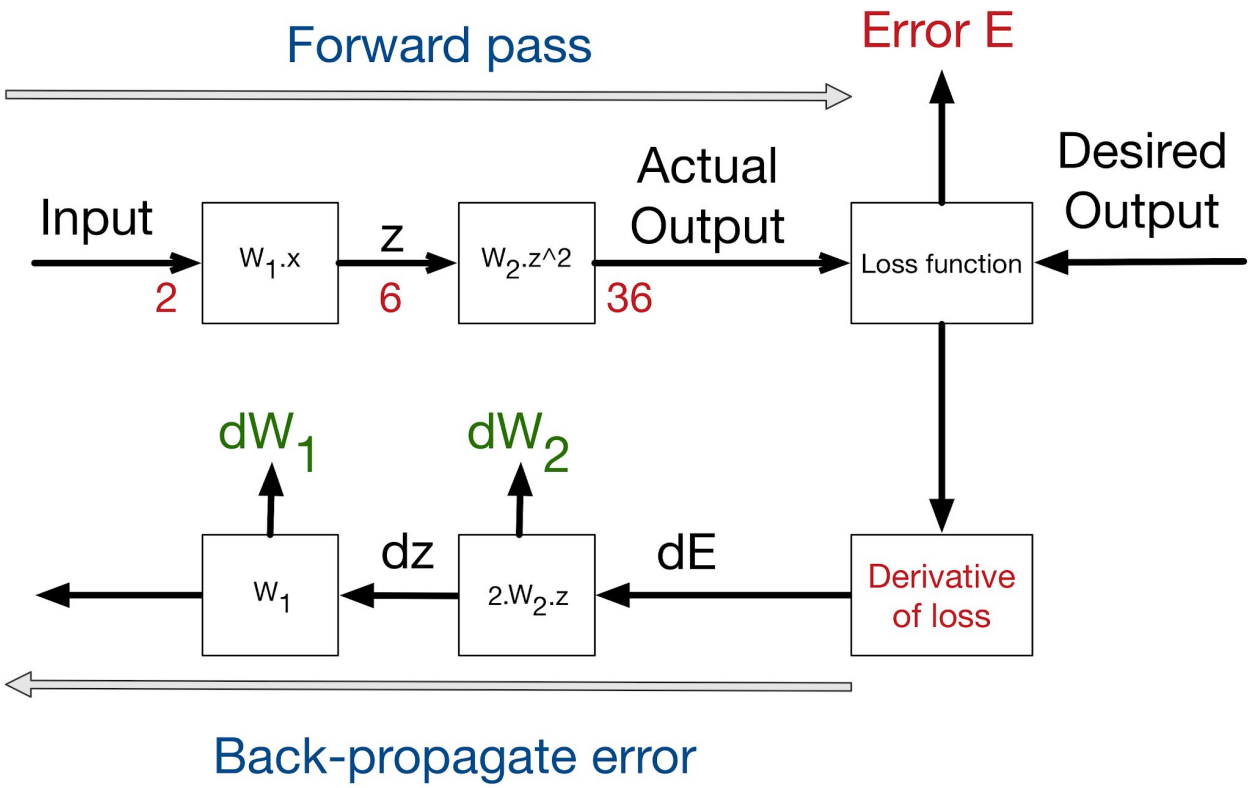
Typical MLP architecture for regression

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Typical MLP architecture for classification

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Feedback mechanism for learning the weights



Cool Animation