Neural Networks

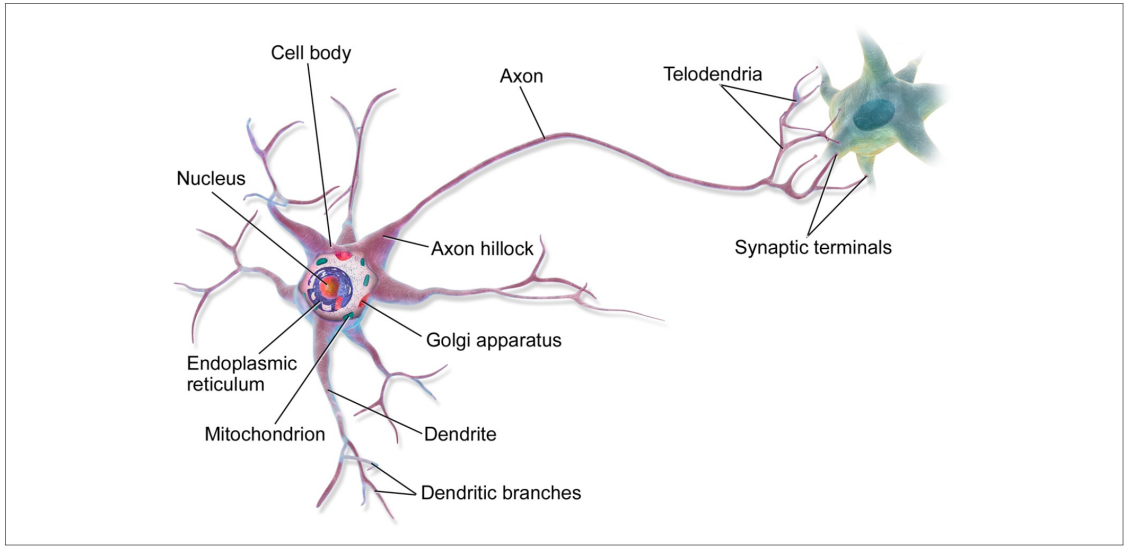
DS 8015

Outline

- Perceptron
- Multi Layer Perceptron (MLP)
- Implementing MLP with Keras

Perceptron

Biological Neuron



Biological neurons produce short electrical impulses called *action potentials* (AP) or signals which travel along the axons and make the synapses release chemical signals called *neurotransmitters*.

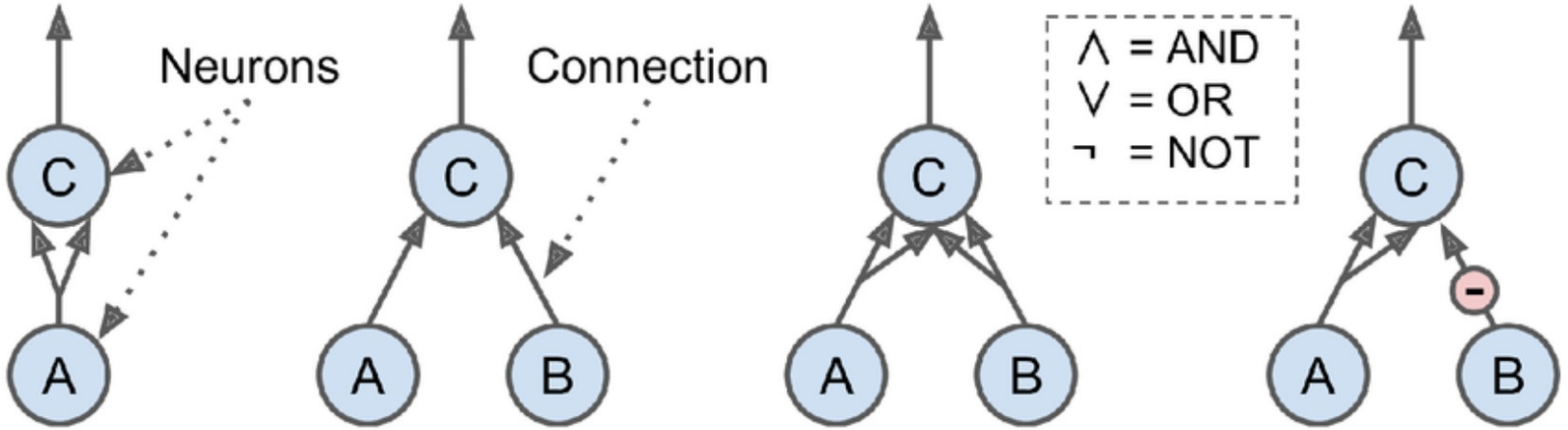
When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses

The architecture of biological neural networks (BNNs) is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, especially in the cerebral cortex (i.e., the outer layer of your brain)

A. Geron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*, O'Reilly, 2nd Edition, 2019

Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an artificial neuron: it has one or more binary (on/off) inputs and one binary output



C = A

Identity function

if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well

C = A ∧ B

Logical AND

Neuron C is activated only when both neurons A and B are activated

C = A ∨ B

Logical OR

Neuron C gets activated if either neuron A or neuron B is activated (or both)

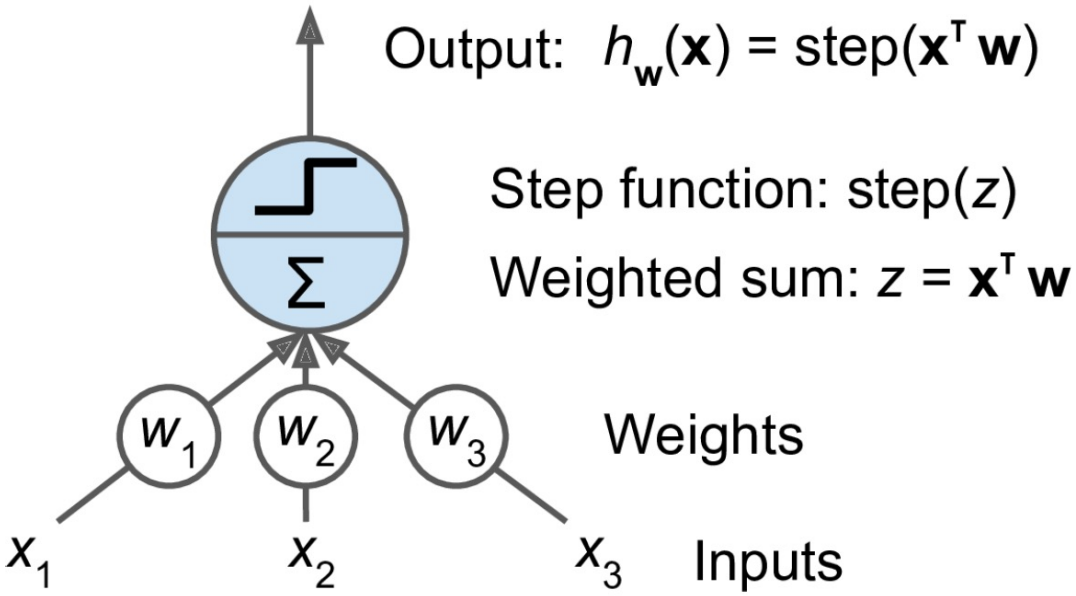
C = A ∧ ¬B

More Complex Functions

Neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

The artificial neuron activates its output when more than a certain number of its inputs are active

Threshold Logic Unit (TLU)



The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.

It is based on a slightly different artificial neuron called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU).

The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight.

The TLU computes a weighted sum of its inputs
($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = x^T w$),
then applies a step function to that sum and outputs the result:
$h_w(x) = \text{step}(z)$,
where $z = x^T w$.

A. Geron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*, O'Reilly, 2nd Edition, 2019

Step Functions

- The most common step functions are Heaviside Step Function and Sign Function

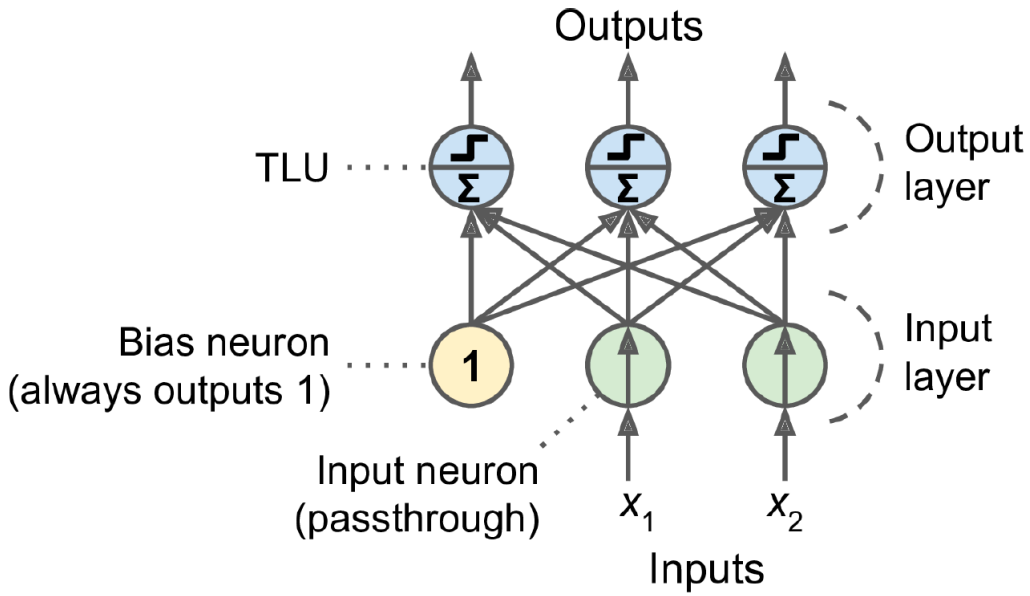
$$heaviside\,(z) = \begin{cases} 0\ if\ z < 0 \\ 1\ if\ z \geq 1 \end{cases}$$

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

A Single TLU

- A single TLU can be used for simple linear binary classification.
- It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise it outputs the negative class (just like a Logistic Regression or linear SVM classifier).
- A single TLU can be used to classify iris flowers based on petal length and width (also adding an extra bias feature $x_0 = 1$)
- Training a TLU in this case means finding the right values for w_0, w_1, and w_2

The Perceptron



A Perceptron with two inputs and three outputs

A **Perceptron** is composed of a single layer of TLUs, with each TLU connected to all the inputs.

When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a *fully connected layer*, or a *dense layer*.

The inputs of the Perceptron are fed to special passthrough neurons called *input neurons*: they output whatever input they are fed

All the input neurons form the input layer. An extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a *bias neuron*, which outputs 1 all the time.

This Perceptron can classify instances simultaneously into three different binary classes, which makes it a **multilabel classifier**

Computing the Outputs of a Fully Connected Layer

$$h_{W,b}(X) = \phi(XW + b)$$

- X represents the matrix of input features (one row per instance and one column per feature)
- W is the weight matrix containing all the connection weights except the ones from the bias neuron (one row per input neuron, one column per artificial neuron in the layer)
- b is the bias vector containing all the connection weights between the bias neuron and the artificial neurones (one bias term per artificial neuron)
- ϕ is the activation function: when the artificial neurons are TLUs then it's a step function

Training The Perceptron

- Perceptron training algorithm proposed by Rosenblatt (inspired by Hebb's rule)
- Hebb: "when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger"
- *Hebb's Rule* (a.k.a. *Hebbian Learning*): the connection weight between two neurons tends to increase when they fire simultaneously.
- Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the **Perceptron learning rule reinforces connections that help reduce the error**

Training The Perceptron

- The Perceptron is fed one training instance at a time, and for each instance it makes its predictions.
- For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction

Perceptron learning rule (weight update)

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

$w_{i,j}$ connection weight between ith input neuron and jth output neuron
x_i is the ith input value of the current training instance
$\hat{y}_j$ is the output of the jth output neuron for the current training instance
y_j is the target output of the jth output neuron for the current training instance
η is the learning rate

Decision Boundary of Perceptrons

- The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers).
- However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.
- This is called the *Perceptron convergence theorem*

Python Implementation

```
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(int) # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```

Perceptron Learning Algorithm

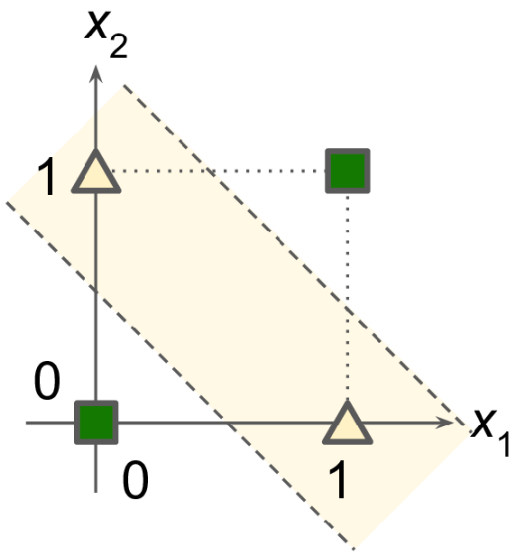
- The Perceptron's learning algorithm resembles Stochastic Gradient Descent
- Scikit-Learn's Perceptron class is equivalent to using an SGDClassifier with the following hyperparameters:
  - loss="perceptron"
  - learning_rate="constant"
  - eta0=1 (the learning rate)
  - penalty=None (no regularization)
- Contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they make predictions based on a hard threshold.
  - This is one reason to prefer Logistic Regression over Perceptrons

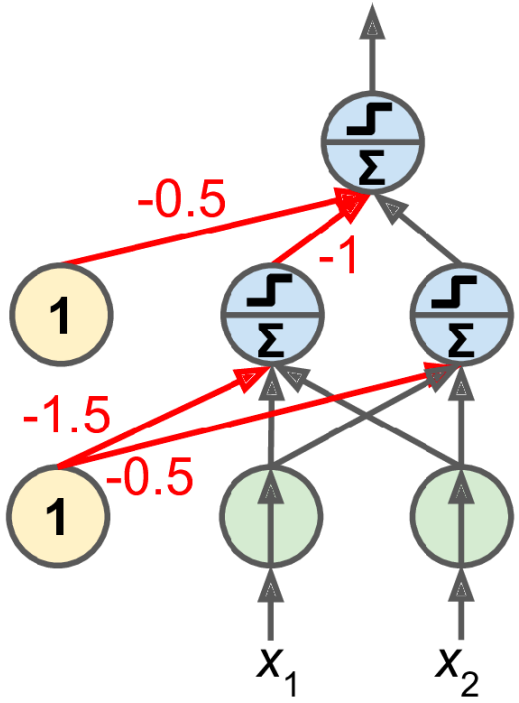
Multilayer Perceptron (MLP)

Multilayer Perceptron

- In their 1969 monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons
  - They are incapable of solving some trivial problems (e.g. XOR - Exclusive OR Classification Problem)
- The limitation were eliminated by stacking multiple Perceptrons
- The resulting ANN is called a *Multilayer Perceptron* (MLP)
- An MLP can solve the XOR problem

Solving XOR Classification Problem

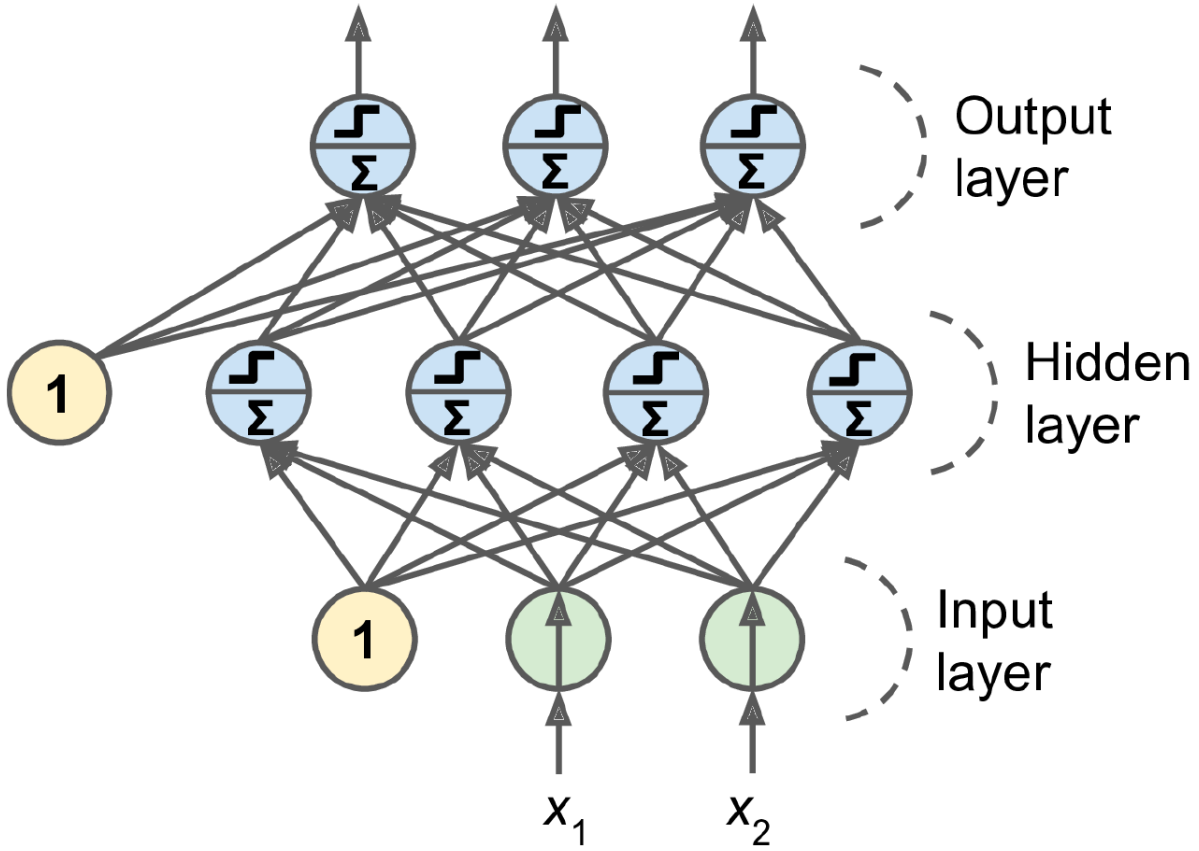




Inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.

All connections have a weight equal to 1, except the four connections where the weight is shown.

MLP



An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer*.

The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*.

Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN)

MLP and Backpropagation

- The algorithm used to train MLPs is called the backpropagation algorithm
- It is Gradient Descent using an efficient technique for computing the gradients automatically: in just two passes through the network (one forward, one backward)
- The backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter
- It can find out how each connection weight and each bias term should be tweaked in order to reduce the error
- Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

MLP and Backpropagation

- *Epoch*: The algorithm handles one mini-batch at a time (e.g., containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*.
- *Forward Pass*:
  - Each mini-batch is passed to the network's input layer, which sends it to the first hidden layer.
  - The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch).
  - The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until the output layer.
  - It is like making predictions, except all intermediate results are preserved since they are needed for the backward pass

MLP and Backpropagation (cont'd)

- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error)
- Then it computes how much each output connection contributed to the error. This is done analytically by applying the *chain rule* which makes this step fast and precise.
- Reverse Pass:
  - The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer
  - This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed

Backpropagation

- In order for Backpropagation work properly a key change in the MLP architecture is done: the step function is replaced with the logistic (sigmoid) function.
  - Step function contains only flat surface
  - Logistic function has a well-defined nonzero derivative everywhere allowing Gradient Descent to make some progress at every step
- Other activation functions that the backpropagating algorithm works well:
  - The hyperbolic tangent function: $tanh(z) = 2\sigma(2z) - 1$
  - The Rectified Linear Unit Function: $ReLU(z) = \max(0, z)$

Regression MLPs

- MLPs can be used for regression tasks. E.g. to predict a single value given multiple features: need a single output neuron (its output is the predicted value)
- Multivariate regression (to predict multiple values at once): one output neuron per output dimension. E.g.,
  - To locate the center of an object in an image: need to predict 2D coordinates, so 2 output neurons
  - Place a bounding box around the object: need 2 more numbers (the width and the height of the object)

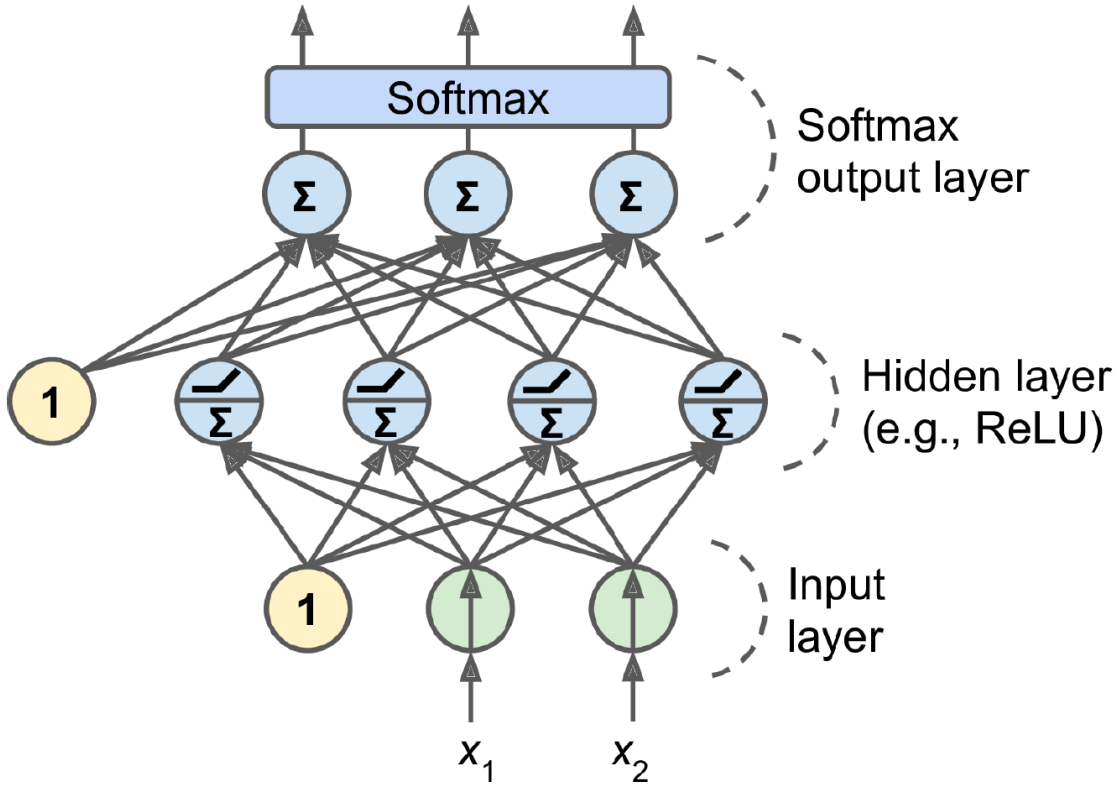
Regression MLP Architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g. 28x28 =784 for MNIST)
# hidden layers	Depends on the problem, typically 1 to 5
# neurons per hidden layer	Depends on the problem, typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLPs

- MLPs can be used for classification tasks.
  - For a binary classification problem: a single output neuron using the logistic activation function (the output will be a number between 0 and 1 that will be interpreted as the estimated probability of the positive class)
- MLPs can handle multilabel binary classification tasks
  - E.g., email classification to predict incoming email is ham or spam, and it is an urgent or nonurgent email. Use 2 output neurons: first outputs the probability that the email is spam; second outputs the probability that it is urgent

Classification MLPs – Multiclass Classification



If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the *softmax* activation function for the whole output layer

The softmax function will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive).

This is called *multiclass classification*.

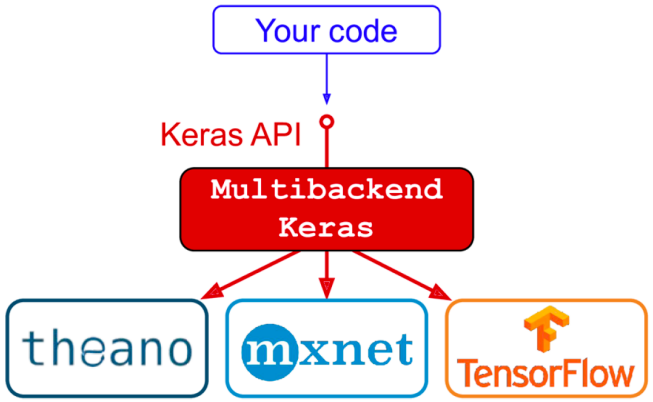
Classification MLP Architecture

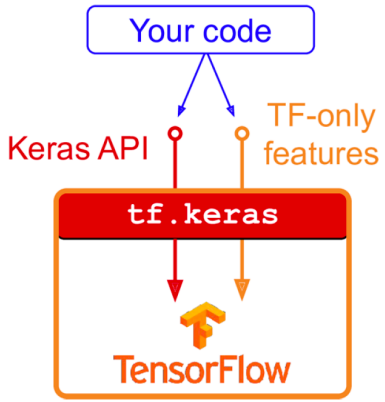
Hyperparameter	Binary Classification	Multilabel Binary Classification	Multiclass Classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# of output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy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

Implementing MLP with Keras

Keras

- Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks
- Documentation: https://keras.io/
- TensorFlow now comes bundled with its own Keras implementation: tf.keras. It only supports TensorFlow as the backend





You may need to install Keras in your local machines

If you're using Google Colab, no need to install

Building an Image Classifier

Fashion MNIST

70,000 grayscale images of 28 x 28 pixels each, with 10 classes

The images represent fashion items rather than handwritten digits (i.e. MNIST), so each class is more diverse

The problem is significantly more challenging than MNIST. E.g., a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST

Python Implementation

```
import tensorflow as tfx
from tensorflow import keras

# Loading Fashion MNIST
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

X_train_full.shape # (60000, 28, 28)
X_train_full.dtype # dtype('uint')

# Creating a validation set and scaling the pixel intensities down to 0..1
# by dividing them by 255.0
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.0

class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
"Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Python Implementation

```
class_names[y_train[0]]   # -> ”coat”

# Creating the model using the Sequential API
model = keras.models.Sequential() # Creates a sequential model
model.add(keras.layers.Flatten(input_shape=[28, 28])) # Converts input images to a 1D array
model.add(keras.layers.Dense(300, activation="relu")) # Hidden layer with 300 neurons with ReLU
model.add(keras.layers.Dense(100, activation="relu")) # Hidden layer with 100 neurons with ReLU
model.add(keras.layers.Dense(10, activation="softmax")) # Output layer. 1 output neuron per class
```

Alternative way

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

Python Implementation

```
model.summary() # Displays all layers of the model
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
dense (Dense)                (None, 300)               235500
dense_1 (Dense)              (None, 100)               30100
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Python Implementation

```
model.layers # Model's layers can be referred as a list, and can be fetched by name
```

```
[<keras.layers.core.flatten.Flatten at 0x7f7a1eaed250>,
 <keras.layers.core.dense.Dense at 0x7f7a1eaed950>,
 <keras.layers.core.dense.Dense at 0x7f7a1ea9a310>,
 <keras.layers.core.dense.Dense at 0x7f7a1ea9ab50>]
```

```
hidden1 = model.layers[1]
hidden1.name
```

```
'dense'
```

```
model.get_layer('dense') is hidden1
```

```
True
```

Python Implementation

Compiling the model (after the model is created it must be compiled). While compiling the model, the loss function and the optimised needs to be specified

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

Using `"sparse_categorical_crosentropy"`, because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive.

If we had one target probability per class for each instance (such as one-hot vectors, e.g. [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), then we would need to use the `"categorical_crosentropy"` loss instead.

If we were doing binary classification or multilabel binary classification, then we would use the "sigmoid" (i.e., logistic) activation function in the output layer instead of the "softmax" activation function, and we would use the `"binary_crosentropy"` loss.

`"sgd"` means that we will train the model using simple Stochastic Gradient Descent

Python Implementation

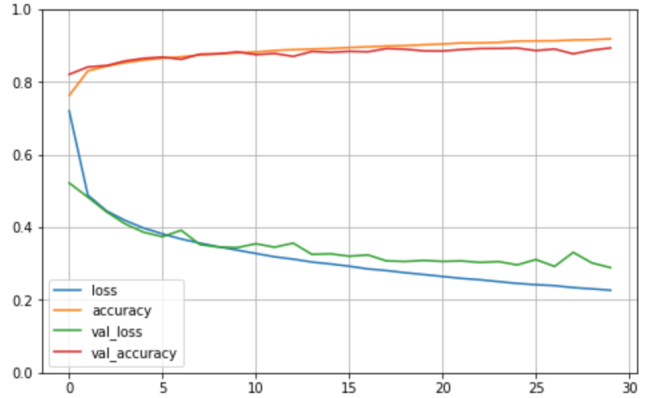
Training the model

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

Visualizing the learning curves

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```



Evaluating the model

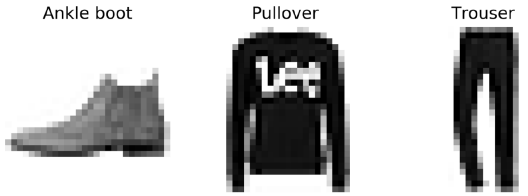
```
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3244 - accuracy: 0.8839
[0.3243924677371979, 0.883899986743927]
```

Python Implementation

Using the model to make predictions

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```




```
array([[0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.02, 0. , 0.98],
       [0. , 0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
dtype=float32)
```

Predicted images are: 9, 2, 1

```
import numpy as np
y_pred=[9, 2, 1]
np.array(class_names)[y_pred]
```

```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```