CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Week 9: Book Chapters 10

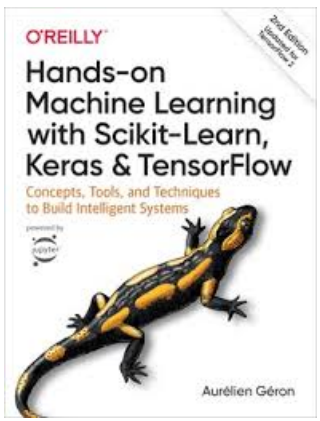
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapters 10.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Introduction to Artificial Neural Networks

Here are the main topics we will cover:

- Perceptrons and *threshold logic unit* (TLU)
- Training of a Perceptron and *Hebb's rule*
- Perceptrons and the XOR classification problem
- Multi-Layer Perceptrons (MLP) and backpropagation
- Commonly used activation functions
- Training a deep neural network (DNN) using the Sequential API
- Fine-tuning neural network hyperparameters

The Perceptron

- 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), where 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: $z = w_1 x_1 + \cdots + w_n x_n = \mathbf{x}^T \mathbf{w}$ then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{x}^T \mathbf{w})$.

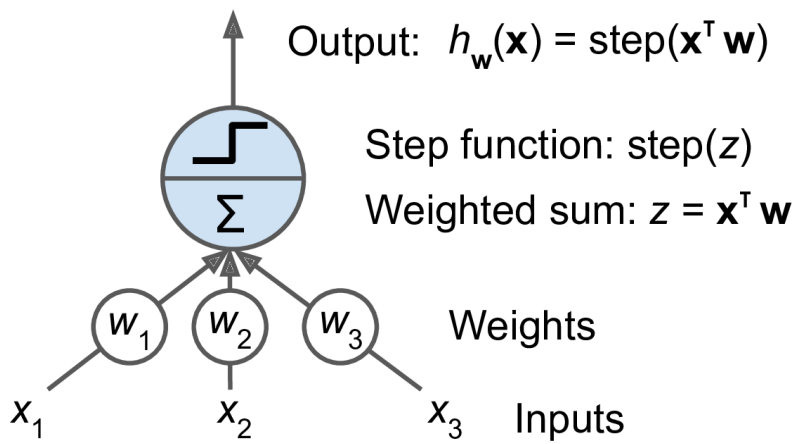


Figure 10-4.
Threshold logic unit

The Perception (cont.)

Step Functions

The most common step function used in Perceptrons is the *Heaviside step function*:

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

Sometimes the *sign function* is used instead:

$$\text{sign}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

The Perceptron (cont.)

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 or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM).

A Perceptron is simply composed of a single layer of TLUs, with

- special passthrough neurons called *input neurons*, which just output whatever input they are fed; and
- an extra bias feature ($x_0 = 1$), which is typically represented using a special type of neuron called a *bias neuron* (it outputs 1 all the time).

Perceptron: An example

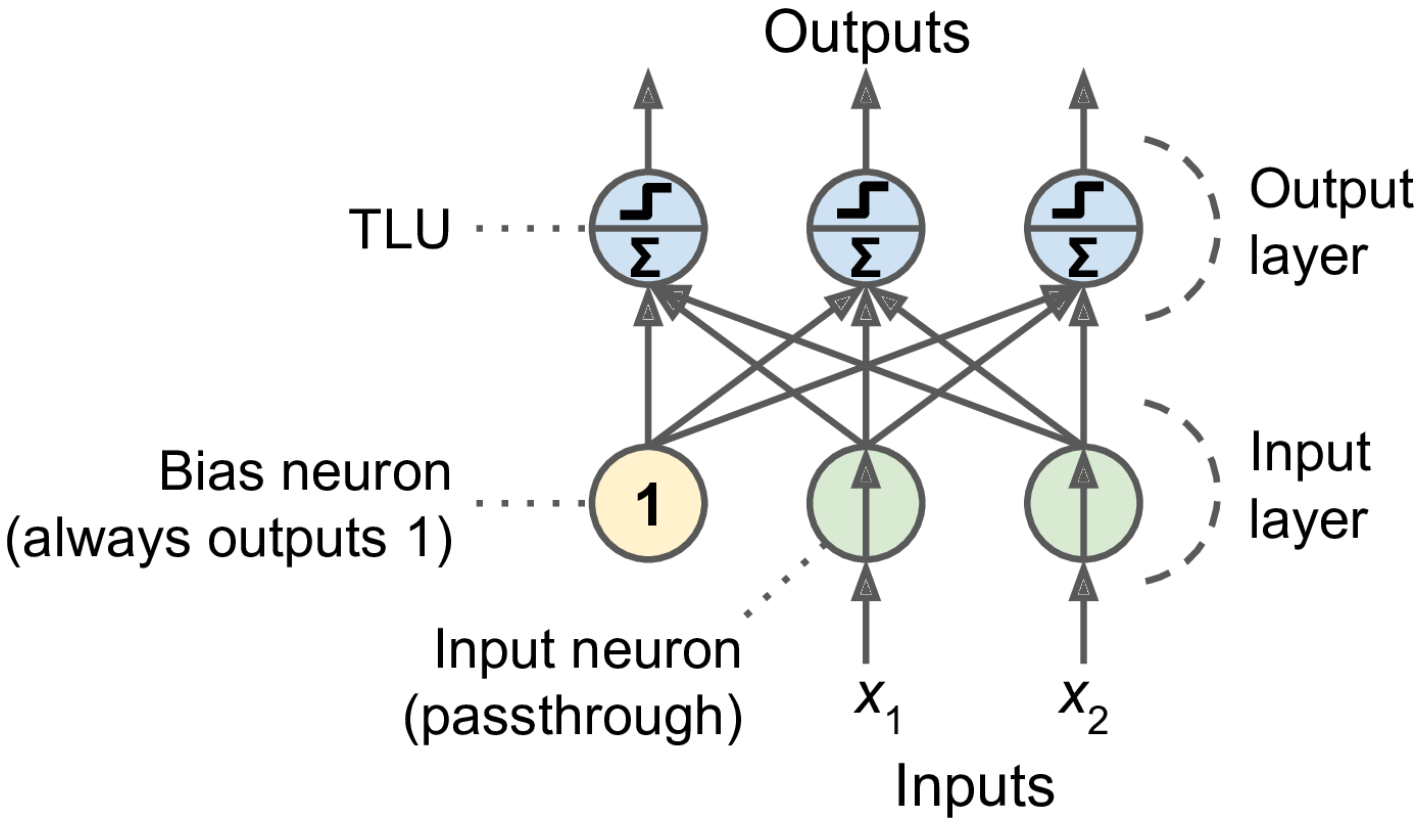
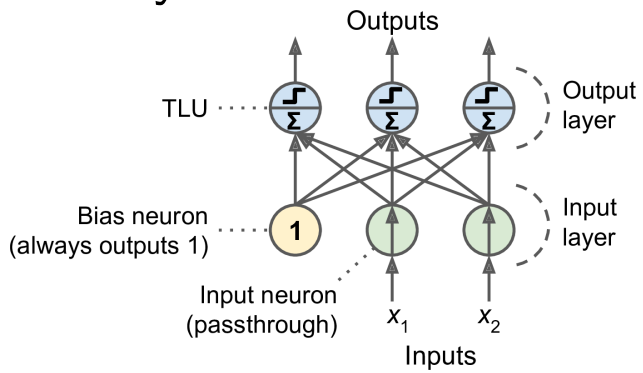


Figure 10-5. A Perceptron with two input, one bias, and three output neurons.

Perceptron: Computing the outputs

The magic of linear algebra makes it possible to efficiently compute the outputs of a fully connected layer:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \phi(\mathbf{x}^\top \mathbf{W} + \mathbf{b}^\top)$$



where

- $\mathbf{x}$ represents the input feature
- $\mathbf{W}$ represents the unknown weight matrix which contains all the connection weights except for the ones from the bias neuron
- $\mathbf{b}$ is the bias vector containing all the connection weights between the bias neuron and the output neurons.
- ϕ is called the *activation function*.

How is a Perceptron trained?

The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by *Hebb's rule* or *Hebbian learning* (name after Donald Hebb) which can be summarized as "cells that fire together, wire together".

More specifically, 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.

How is a Perceptron trained? (cont.)

Perceptron learning rule (weight update):

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

where

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

Perceptron: Decision Boundary

The decision boundary of each output neuron is linear, so Perceptrons (just like Logistic Regression classifiers) are incapable of learning complex patterns.

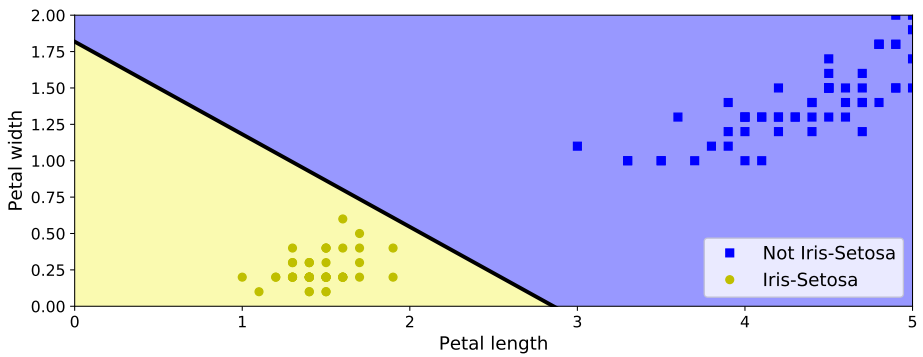
However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.[1] This is called the *Perceptron convergence theorem*.

[1] Note that this solution is generally not unique.

Implementing Perceptron in Scikit-Learn

Scikit-Learn provides a `Perceptron` class that implements a single TLU network:

```
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)   # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```



Decision boundary output by the code above.

Implementing Perceptron in Scikit-Learn (cont.)

The Perceptron learning algorithm strongly resembles Stochastic Gradient Descent. In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

- `loss="perceptron"`,
- `learning_rate="constant"`,
- `eta0=1` (learning rate), and
- `penalty=None` (no regularization).

Comparing Perceptron with Logistic Regression Classifiers

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they just make predictions based on a hard threshold.

This is one of the good reasons to prefer Logistic Regression over Perceptrons.

Weaknesses of Perceptrons

In their 1969 monograph titled "*Perceptrons*", Marvin Minsky and Seymour Papert highlighted some serious weaknesses of Perceptrons, in particular, their inability of solving some trivial problems (e.g., the *Exclusive OR* (XOR) classification problem).

Of course this is true of any other linear classification model as well (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons. As a result, many researchers dropped *connectionism* altogether.

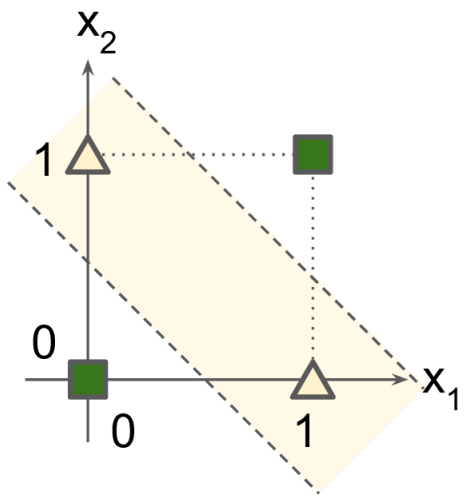


Figure 10-6a. XOR classification problem

Weaknesses of Perceptrons (cont.)

However, it turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multi-Layer Perceptron* (MLP).

In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP for each combination of inputs: with inputs (0, 0) or (1, 1) the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.

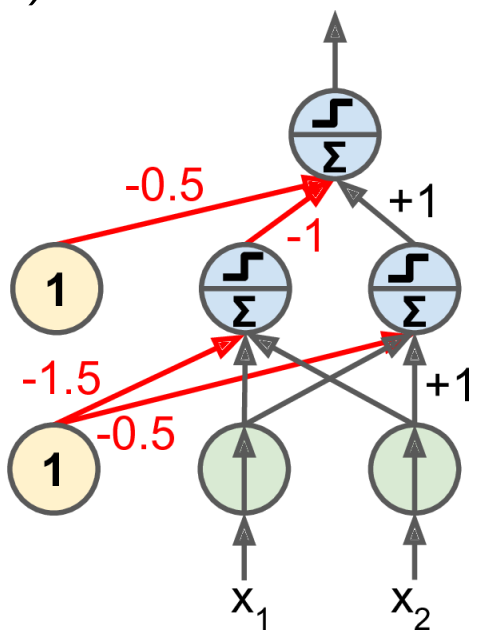


Figure 10-6b. An MLP that solves the XOR classification problem.

Multi-Layer Perceptron and Backpropagation

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*.

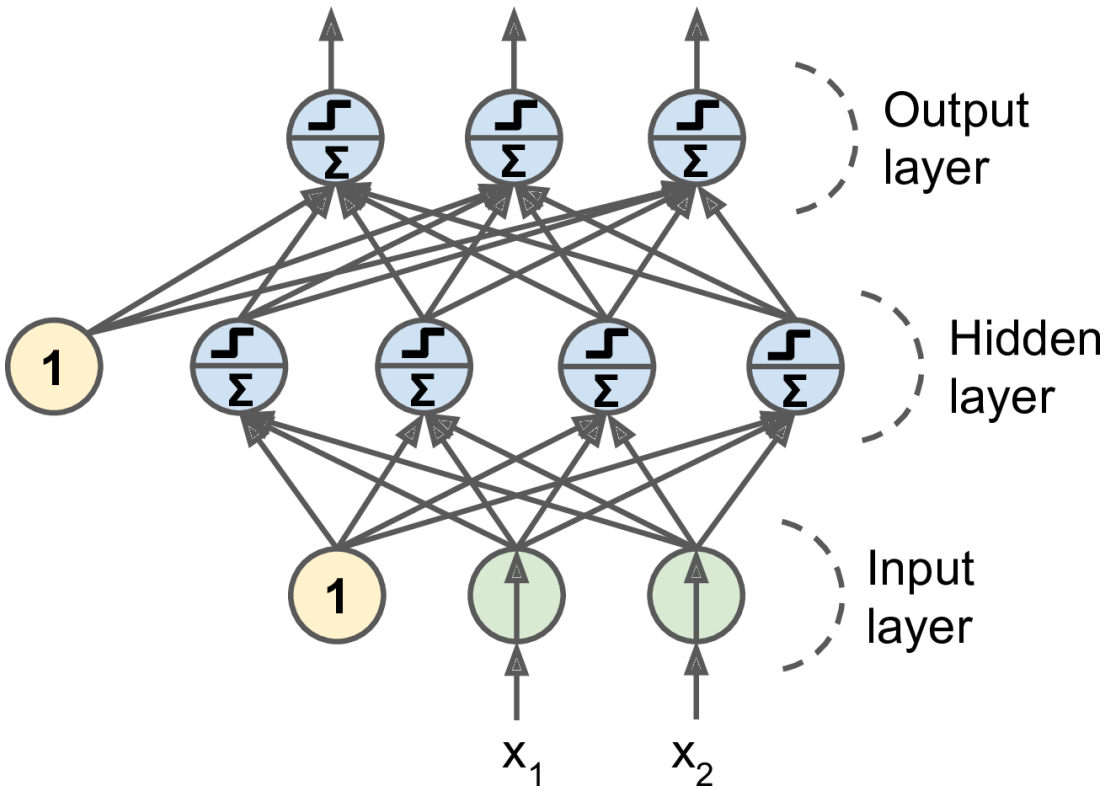


Figure 10-7. Multi-Layer Perceptron.

Multi-Layer Perceptron and Backpropagation (cont.)

We can describe the *backpropagation training algorithm*[2] as Gradient Descent using reverse-mode autodiff.
For each training instance, the algorithm

- feeds it to the network and computes the output of every neuron in each layer (this is the *forward pass*),
- measures the network's output error (i.e., the difference between the desired output and the actual output of the network),
- goes through each layer in reverse order to measure the error contribution from each connection (this is the *reverse pass*) until the input layer is reached, and
- finally slightly tweaks the connection weights to reduce the error (this is the *Gradient Descent step*).

[2] "Learning Internal Representations by Error Propagation," D. Rumelhart, G. Hinton, R. Williams (1986). This algorithm was actually invented several times, starting with P. Werbos in 1974.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

In order for the backpropagation training algorithm to work, Rumelhart et al made a key change to the MLP's architecture: they replaced the step function with the logistic function, $\sigma(z) = 1/(1 + \exp(-z))$. This was essential because the step function has no gradient to work with, while the logistic function has a well-defined nonzero derivative everywhere.

Such a function which defines the output of a neuron (or node) given an input or set of inputs is known as the *activation function*.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

Apart from the logistic function, two other popular activation functions are:

- *The hyperbolic tangent function*: $\tanh(z) = 2\sigma(2z) - 1$. This function has a similar S shape as the logistic function, is continuous and differentiable, but its values range from -1 to $+1$.
- *The ReLU function*: $\text{ReLU}(z) = \max(0, z)$. This function is continuous but unfortunately not differentiable everywhere. However, in practice it works very well and has the advantage of being fast to compute. Most importantly, the fact that it does not have a maximum output value also helps reduce some issues during Gradient Descent.

Activation functions

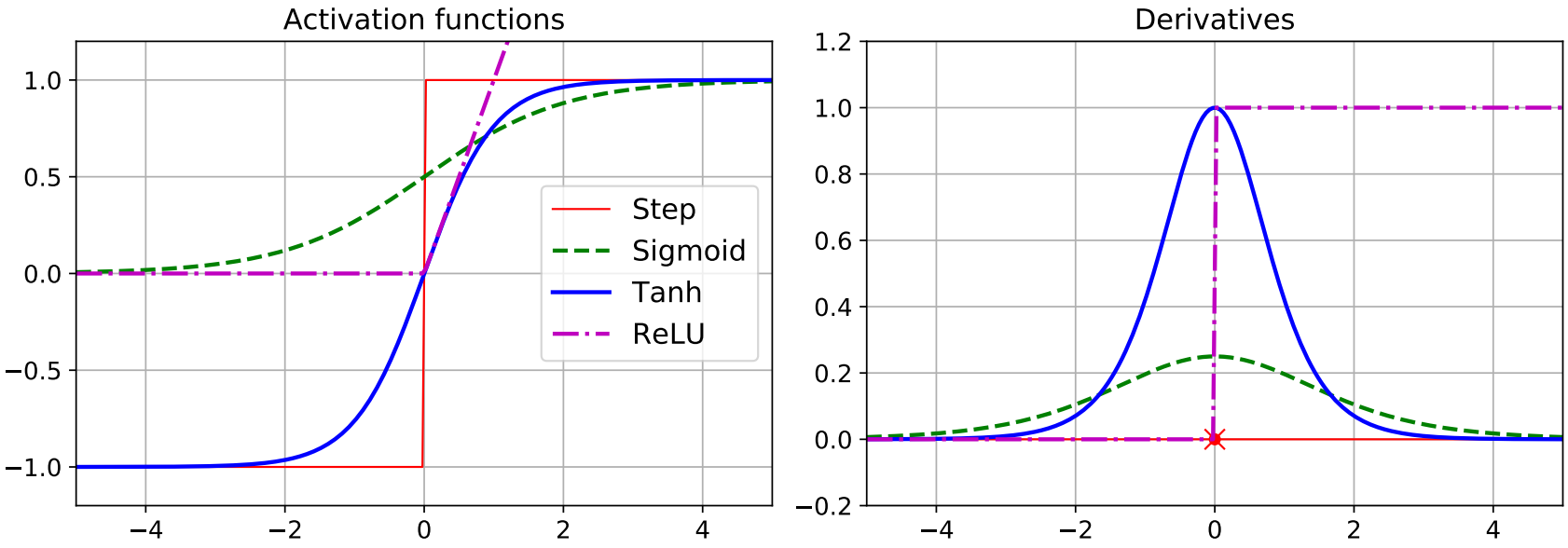


Figure 10-8. Activation functions and their derivatives

Regression MLPs

MLPs can be used for regression tasks. For example, to predict the price of a house given many of its features, you need one output neuron. For multivariate regression, you need multiple neurons.

For regression, the output neurons usually do not need an activation function. Only when the output z needs to be

- always positive, then use the ReLU or $\text{softplus}(z) = \log(1 + \exp(z))$ activation function;
- in a certain range of values, then use logistic or hyperbolic tangent and then scale the value to the required range.

Regression MLPs (cont.)

The loss function to use during training is typically the mean squared error or mean absolute error (the latter is more robust against outliers). Alternatively, the Huber loss, which is a combination of MSE and MAE, can also be used.

Typical regression MLP architecture:

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)

Classification MLPs

MLPs can also be used for classification tasks.

- For binary classification problems, one single output neuron with a logistic activation function would be sufficient.
- For multilabel classification problems, you can have multiple output neurons, each with a logistic activation function.
- For multiclass classification problems (e.g., MNIST dataset), you can have multiple output neurons, with the softmax activation function. This will ensure that all the output values are probabilities and they sum to 1.

For the loss function, since we are predicting probability distributions, the *cross-entropy* loss (also called the *log loss*) is generally a good choice:

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

where K denotes the total number of classes (e.g., $K = 10$ for MNIST).

Classification MLPs (cont.)

Typical classification MLP architecture:

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

Implementing MLPs with Keras

Keras (`https://keras.io/`) is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks.

It was developed by François Chollet and was released as an open source project in March 2015. It quickly gained popularity and, at present, is available in three popular open source Deep Learning libraries: *TensorFlow*, *Microsoft Cognitive Toolkit* (CNTK), and *Theano*. We will refer to this reference implementation as *multibackend Keras*.

Implementing MLPs with Keras (cont.)

TensorFlow now comes bundled with its own Keras implementation, `tf.keras`, which supports TensorFlow's Data API.

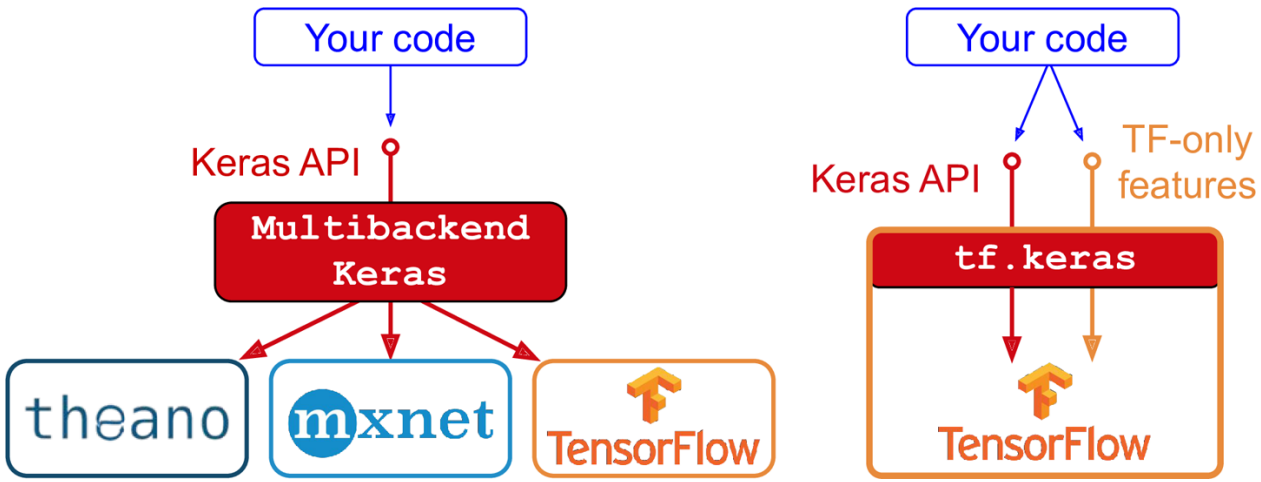


Figure 10-10. Two implementations of the Keras API: multibackend Keras (left) and tf.keras (right)

Another popular Deep Learning library, after Keras and TensorFlow, is Facebook's *PyTorch* library.

Implementing MLPs with Keras (cont.)

After installation, check that you have the correct version of TensorFlow:

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

Example: Building an Image Classifier Using the Sequential API

A linear model can reach about 92% on MNIST, but only about 83% on *Fashion MNIST*, which also has 10 classes with input being 28×28 images:

Figure 10-11. Samples from Fashion MNIST.

The 10 classes are: "T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", and "Ankle boot".

Example: Building an Image Classifier Using the Sequential API (cont.)

Loading the data and creating the model

```
import tensorflow as tf
from tensorflow import keras

fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()   #60,000 and 10,000
# split training set to form a validation set
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:]

# creating the model
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"))
```

The last few lines above can be replaced by

```
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")
])
```

Example: Building an Image Classifier Using the Sequential API (cont.)

Inspecting the model

```
>>> model.summary()
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

>>> hidden1 = model.layers[1]
>>> weights, biases = hidden1.get_weights()
.......
>>> weights.shape
(784, 300)
>>> biases.shape
(300,)
```

Example: Building an Image Classifier Using the Sequential API (cont.)

Compiling the model

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

Training and evaluating the model

```
>>> history = model.fit(X_train, y_train, epochs=30,
... validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218 - accuracy: 0.7660
- val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [======] - 2s 45us/sample - loss: 0.4840 - accuracy: 0.8327
- val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [======] - 3s 53us/sample - loss: 0.2252 - accuracy: 0.9192
- val_loss: 0.2999 - val_accuracy: 0.8926
```

Useful options to incorporate: `class_weight`, `sample_weight`, `batch_size` (default=32)

Example: Building an Image Classifier Using the Sequential API (cont.)

Inspecting the learning curves

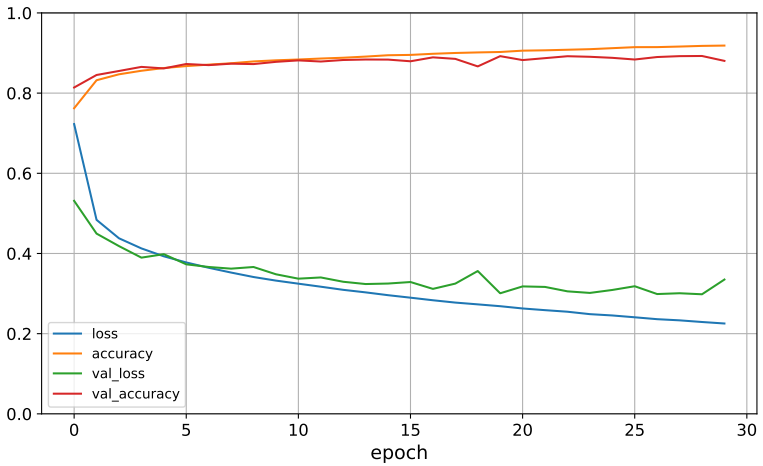


Figure 10-12. Learning curves.

Applying the trained model on the test set

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

Example: Building an Image Classifier Using the Sequential API (cont.)

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. , 0.03, 0. , 0.01, 0. , 0.96],
       [0. , 0. , 0.98, 0. , 0.02, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')

# compare with the ground truth
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

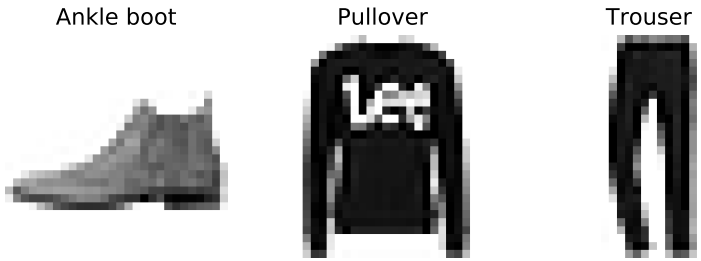


Figure 10-13. Correctly classified Fashion MNIST images.

Example: Building a Regression MLP Using the Sequential API

The code below is for the California housing problem.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data, housing.target, random_state=4
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full, y_train_full, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))

# evaluate the model on the test set
mse_test = model.evaluate(X_test, y_test)

X_new = X_test[:3]
y_pred = model.predict(X_new)
```

Example: Building a Regression MLP Using the Sequential API (cont.)

A close inspection on the output shows that this simple MLP does not produce very good predictions. Sometimes it is useful to build neural networks with more complex topologies, such as using the *Functional API*.

Both *Functional API* and *Subclassing API* are outside the scope of the unit.

Saving and Restoring a Model

For both the Sequential API and the Functional API, we start by declaring the layers we want to use and how they should be connected. The actual feeding of data for training is done after the neural network has been constructed. This has the advantages that the model can be saved, cloned, and shared.

Saving a trained Keras model is very simple:

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")   # HDF5 format
```

Restoring a model from disk is equally simple:

```
model = keras.models.load_model("my_keras_model.h5")
```

If your model takes a long time to train, it is useful to save checkpoints at regular intervals during training.

Saving and Restoring a Model (cont.)

Using Callbacks If your model takes a long time to train, it is useful to save checkpoints at regular intervals during training. To do so, we use *callbacks* to tell the `fit()` method to save checkpoints.

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid),
callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

By default, a model checkpoint is saved at the end of each epoch. Another way to implement early stopping is to simply use the `EarlyStopping` callback to interrupt training when no progress is observed on the validation set for a number of epochs (defined by the `patience` argument):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

Using TensorBoard for Visualization

TensorBoard is a great interactive visualization tool for viewing the learning curves during training, compare learning curves between multiple runs, etc. It comes with the installation of TensorFlow.

To use it, your program needs to output the data that you want to visualize to special binary log files called *event files*.
We need to specify a root log directory for storing our TensorBoard logs. We can also include extra information in the log directory name, such as hyperparameter values that you are testing for inspection in TensorBoard:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
  import time
  run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
  return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2020_04_17-15_15_22'
```

Using TensorBoard for Visualization (cont.)

```
[...] # Build and compile your model
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

To start TensorBoard:

```
$ tensorboard --logdir=./my_logs --port=6006
```

Once the server has been started, open a web browser and go to `http://localhost:6006`.

Alternatively, start TensorBoard inside Jupyter:

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```

Using TensorBoard for Visualization (cont.)

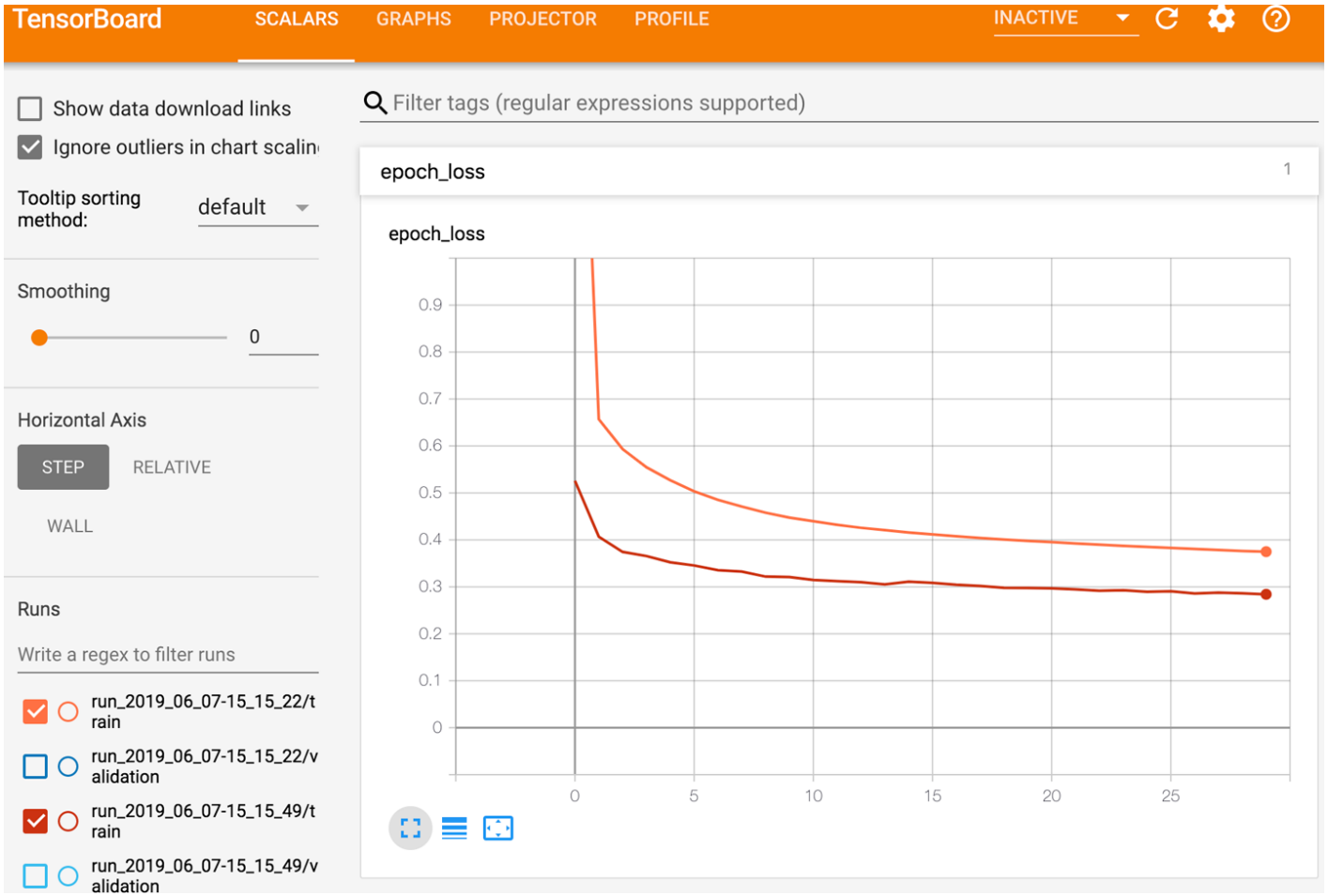


Figure 10-17. Visualizing learning curves with TensorBoard.

Using TensorBoard for Visualization (cont.)

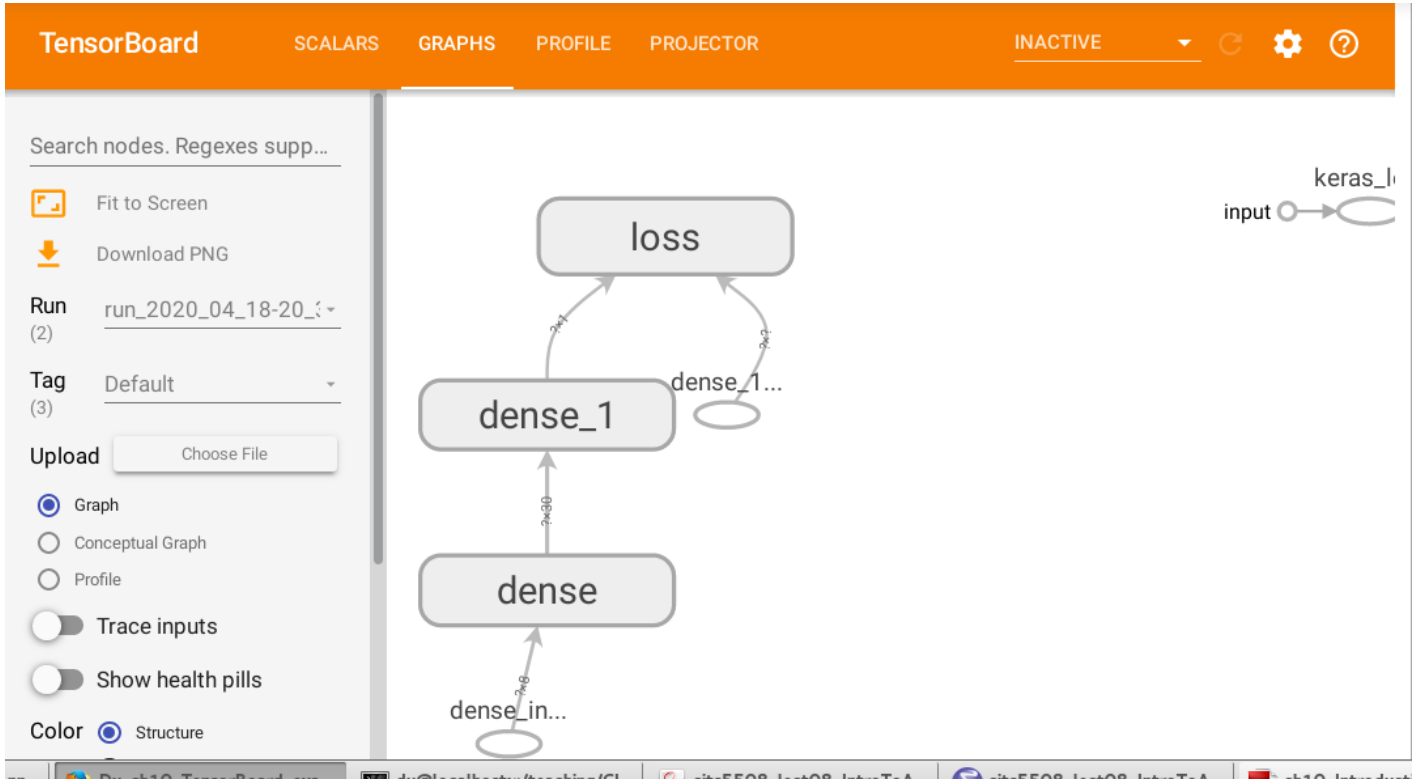


Figure 10-17. Visualizing the Network Graph with TensorBoard.

You can also visualize the learned weights (projected to 3D) and the profiling traces.

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. We can use `GridSearchCV` or `RandomizedSearchCV` to explore the hyperparameter space.

There are also many Python libraries you can use to optimize hyperparameters: Keras Tuner, Scikit-Optimize (skopt), Spearmint, etc.

Despite all this exciting progress and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter so that you can build a quick prototype and restrict the search space. The remaining slides provide guidelines for choosing some of the main hyperparameters.

Fine-Tuning Neural Network Hyperparameters (cont.)

Number of Hidden Layers

- For many problems you can start with just one or two hidden layers and the neural network will work just fine, e.g., 97% (or above 98%) accuracy can be reached on the MNIST dataset using just one (or two) hidden layer(s) with a few hundred neurons.
- For more complex problems, you can increase the number of hidden layers until you start overfitting the training set. Typically, for complex image recognition or speech recognition tasks, dozens (maybe hundreds) of hidden layers (but not fully connected) are required.
- Can also reduce the number hidden layers by training a network that has been pre-trained on a similar problem. This is known as *transfer learning*.

Fine-Tuning Neural Network Hyperparameters (cont.)

Number of Neurons per Hidden Layer

- Obviously the number of neurons in the input and output layers is determined by the type of input and output your task requires.
- As for the hidden layers, a common practice is to size them to form a pyramid, with fewer and fewer neurons at each layer – the rationale being that many low-level features can coalesce into far fewer high-level features. However, a common practice today is to have the same number of neurons in each hidden layer. This also helps to reduce the number of parameters to tune.
- Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting, then use early stopping and other regularization techniques to prevent it from overfitting.

Fine-Tuning Neural Network Hyperparameters (cont.)

Activation Functions

- In most cases you can use the ReLU activation function in the hidden layers (or one of its variants). It is a bit faster to compute than other activation functions, and Gradient Descent does not get stuck as much on plateaus (as opposed to the logistic function or the hyperbolic tangent function, which saturate at 1).
- For the output layer, the softmax activation function is generally a good choice for classification tasks when the classes are mutually exclusive. When they are not mutually exclusive (or when there are just two classes), you generally want to use the logistic function. For regression tasks, you can simply use no activation function at all for the output layer.

Fine-Tuning Neural Network Hyperparameters (cont.)

Optimizer

- Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will see several advanced optimizers in the next lecture.

Batch size

- The common recommendation is to use larger batch size that can fit in the GPU RAM; however, in practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. Currently, small batch sizes (2 to 32) are preferred by some researchers while large batch sizes are recommended by others. *Learning rate warmup* has been suggested to use with large batch sizes.

Fine-Tuning Neural Network Hyperparameters (cont.)

Number of iterations

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

Summary

- Understand the architecture of Perceptrons
- Know how to train a Perceptron and understand its weakness
- Understand multi-layer Perceptrons and backpropagation
- Understand what activation functions are and know a few commonly used activation functions
- Know how to train a DNN using Keras in TensorFlow
- Understand how to use TensorBoard
- Understand some of the important hyperparameters in a DNN and how to fine tune them; in particular, the number of hidden layers, the number of neurons per layer, and the activation functions.

For next week

Work through the lab sheet and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read up to Chapters 11&12 on Training Deep Neural Networks with TensorFlow

And that's all for this week's lecture.

Have a good week.