DEEP LEARNING TUTORIAL

This tutorial covers several major aspects of neural networks by providing working nets coded in Keras, a minimalist and efficient Python library for deep learning computations running on the top of either Google's TensorFlow or University of Montreal's Theano backend. So, let's star For the algorithms of deep learning the packages that are required are:

- TensorFlow 1.0.0 or higher
- Keras 2.0.2 or higher
- Matplotlib 1.5.3 or higher
- Scikit-learn 0.18.1 or higher
- NumPy 1.12.1 or higher

Some of the mos known arquitectures:

- Perceptron: is a model having one single linear layer
- Multilayer perceptron: is a model having multiple layers
- Activation functions
- Gradient descent
- Stochastic gradient descent
- Backpropagation

More in: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464 (https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464)

Optimization functions

- A common problem we all face when working on deep learning projects is choosing a learning rate and optimizer (the hyper-parameters).
- In keras we have 6 different optimizers: Gradient Descent, Adam, Adagrad, Adadelta, RMS Prop and Momentum.
- gradient + momentum: fuerza hacia lam isma dirección de decrecimiento, evita oscilaciones hacia arriba y hacia abajo.
- Rprop truncamientos hacia arriba y hacia abajo para evitar oscilaciones (método adaptativo)
- RMS prop: tiene un factor de memoria para recordar una mejora con respecto a R prop.
- Adam learns the fastest. Adam is more stable than the other optimizers, it doesn’t suffer any major decreases in accuracy.
- **Momentum vs. Learning rate tradeoff**
- **Learnign rate:**
- There is a valley shape for each optimizer: too low a learning rate never progresses, too high a learning rate causes instability and never converges. In between there is a band of “just right” learning rates that successfully train.
- There is no learning rate that works for all optimizers.
- Learning rate can affect training time by an order of magnitude.
- It’s crucial you choose the correct learning rate as otherwise your network will either fail to train, or take much longer to converge.

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1 (https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1)

Activation functions

1. **Linear Activation Function**: It is a simple linear function of the form f(x) = x. Basically, the input passes to the output without any modification.
2. **Non-Linear Activation Functions:** These functions are used to separate the data that is not linearly separable and are the most used activation functions. Few examples of different types of non-linear activation functions are sigmoid, tanh, relu, lrelu, prelu, swish, etc.
   A. **Sigmoid**: It is also known as Logistic Activation Function. It takes a real-valued number and squashes it into a range between 0 and 1. It is also used in the output layer where our end goal is to predict probability. It converts large negative numbers to 0 and large positive numbers to 1.

The three major drawbacks of sigmoid are:

- **Vanishing gradients:** Notice, the sigmoid function is flat near 0 and 1. In other words, the gradient of the sigmoid is 0 near 0 and 1. During backpropagation through the network with sigmoid activation, the gradients in neurons whose output is near 0 or 1 are nearly 0. These neurons are called saturated neurons. Thus, the weights in these neurons do not update. Not only that, the weights of neurons connected to such neurons are also slowly updated. This problem is also known as vanishing gradient. So, imagine if there was a large network comprising of sigmoid neurons in which many of them are in a saturated regime, then the network will not be able to backpropagate.

- **Not zero centered:** Sigmoid outputs are not zero-centered.
- **Computationally expensive:** The exp() function is computationally expensive compared with the other non-linear activation functions.

2. **Tanh** You can think of a tanh function as two sigmoids put together. In practice, tanh is preferable over sigmoid. The negative inputs considered as strongly negative, zero input values mapped near zero, and the positive inputs regarded as positive. The only drawback of tanh is that the tanh function also suffers from the vanishing gradient problem and therefore kills gradients when saturated.

To address the vanishing gradient problem, let us discuss another non-linear activation function known as the rectified linear unit (ReLU) which is a lot better than the previous two activation functions and is most widely used these days.

Rectified Linear Unit (ReLU)

ReLU is half-rectified from the bottom as you can see from the figure above. Mathematically, it is given by this simple expression

```
\[ f(x) = \max(0,x) \]
```

This means that when the input x < 0 the output is 0 and if x > 0 the output is x. This activation makes the network converge much faster. It does not saturate which means it is resistant to the vanishing gradient problem at least in the positive region (when x > 0), so the neurons do not backpropagate all zeros at least in half of their regions. ReLU is computationally very efficient because it is implemented using simple thresholding. But there are few drawbacks of ReLU neuron :

Not zero-centered: The outputs are not zero centered similar to the sigmoid activation function. The other issue with ReLU is that if x < 0 during the forward pass, the neuron remains inactive and it kills the gradient during the backward pass. Thus weights do not get updated, and the network does not learn. When x = 0 the slope is undefined at that point, but this problem is taken care of during implementation by picking either the left or the right gradient.

Leaky ReLU

his was an attempt to mitigate the dying ReLU problem. The function computes

```
\[ f(x) = max(0.1x, x) \]
```

The concept of leaky ReLU is when x < 0, it will have a small positive slope of 0.1. This function somewhat eliminates the dying ReLU problem, but the results achieved with it are not consistent. Though it has all the characteristics of a ReLU activation function, i.e., computationally efficient, converges much faster, does not saturate in positive region.

The idea of leaky ReLU can be extended even further. Instead of multiplying x with a constant term we can multiply it with a hyperparameter which seems to work better the leaky ReLU. This extension to leaky ReLU is known as Parametric ReLU.

Parametric ReLU

The PReLU function is given by

```
\[ f(x) = \max(\alpha x, x) \]
```

Where \alpha is a hyperparameter. The idea here was to introduce an arbitrary hyperparameter \alpha, and this \alpha can be learned since you can backpropagate into it. This gives the neurons the ability to choose what slope is best in the negative region, and with this ability, they can become a ReLU or a leaky ReLU.

In summary, it is better to use ReLU, but you can experiment with Leaky ReLU or Parametric ReLU to see if they give better results for your problem

SWISH

Also known as a self-gated activation function, has recently been released by researchers at Google. Mathematically it is represented as

```
\[ \sigma(x) = \frac{x}{1 + e^{-x}} \]
```

According to the paper, the SWISH activation function performs better than ReLU

In the negative region of the x-axis the shape of the tail is different from the ReLU activation function and because of this the output from the Swish activation function may decrease even when the input value increases. Most activation functions are monotonic, i.e., their value never decreases as the input increases. Swish has one-sided boundedness property at zero, it is smooth and is non-monotonic. It will be interesting to see how well it performs by changing just one line of code.

https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/ (https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/)

Softmax

The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems. The sigmoid function as we saw earlier was able to handle just two classes. What shall we do when we are trying to handle multiple classes. Just classifying yes or no for a single class would not help then. The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs. (convertir los valores en probabilidad)

Choosing the right Activation Function

- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoids and tanh functions are sometimes avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used in most cases these days
- If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
- Always keep in mind that ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn't provide with optimum results

Loss functions

Take care of the **objective function** (loss function). Choosing the right objective function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss; so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.

Fortunately, 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. For instance,

- you'll use **binary crossentropy** for a two-class classification problem,
- **categorical crossentropy** 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're working on truly new research problems will you have to develop your own objective functions.

Bag of words vs. Embbedings

https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a (https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a)

Data Augmentation

Deep learning algorithms often perform better with more data. If you can't reasonably get more data, you can invent more data.

- If your data are vectors of numbers, create randomly modified versions of existing vectors.
- If your data are images, create randomly modified versions of existing images.
- If your data are text, you get the idea…

Often this is called **data augmentation or data generation.** You can use a generative model. You can also use simple tricks. **For example:**

- with photograph image data, you can get big gains by randomly shifting and rotating existing images. It improves the generalization of the model to such transforms in the data if they are to be expected in new data.
- remenber the problem of optimal price for insurance

**ImageDataGenerator* *

Keras provides the ImageDataGenerator class that defines the configuration for image data preparation and augmentation. This includes capabilities such as:

- Sample-wise standardization.
- Feature-wise standardization.
- ZCA whitening.

- Random rotation, shifts, shear and flips.
- Dimension reordering.
- Save augmented images to disk.
- An augmented image generator can be created as follows:

```
datagen = ImageDataGenerator()
```

Rather than performing the operations on your entire image dataset in memory, the API is designed to be iterated by the deep learning model fitting process, creating augmented image data for you just-in-time. This reduces your memory overhead, but adds some additional time cost during model training.

https://machinelearningmastery.com/image-augmentation-deep-learning-keras/ (https://machinelearningmastery.com/image-augmentation-deep-learning-keras/)

TIPS

- Avoid bottlenecks, the hidden nodes have to be grater in number in comparison with the input nodes.
- Take in consideration the rate of learning and momentum.
- The neural networks consedire interactions between variables (for this reason more nodes are more interactions). Te problem is the interpretability because you never know which variables and which interactions are the better.
- Data augmentaton to fit overfittithg. Sometime the overfitting is because you do not show to the machine all the patters so, you have to create more data in order to have more information.

- **Rescale Your Data** to the bounds of your activation functions. If you are using sigmoid activation functions, rescale your data to values between 0-and-1. If you're using the Hyperbolic Tangent (tanh), rescale to values between -1 and 1. This applies to inputs (x) and outputs (y). For example, if you have a sigmoid on the output layer to predict binary values, normalize your y values to be binary. If you are using softmax, you can still get benefit from normalizing your y values. I would suggest that you create a few different versions of your training dataset as follows:
- Normalized to 0 to 1.
- Rescaled to -1 to 1.
- Standardized.

Then evaluate the performance of your model on each. Pick one, then double down. If you change your activation functions, repeat this little experiment. Big values accumulating in your network are not good. In addition, there are other methods for keeping numbers small in your network such as normalizing activation and weights, but we'll look at these techniques later.

- **Transform Your Data**

Related to rescaling suggested above, but more work. You must really get to know your data. Visualize it. Look for outliers. Guesstimate the univariate distribution of each column.

- Does a column look like a skewed Gaussian, consider adjusting the skew with a Box-Cox transform.
- Does a column look like an exponential distribution, consider a log transform.
- Does a column look like it has some features, but they are being clobbered by something obvious, try squaring, or square-rooting.
- Can you make a feature discrete or binned in some way to better emphasize some feature.

Lean on your intuition. Try things.

- Can you pre-process data with a projection method like PCA?
- Can you aggregate multiple attributes into a single value?
- Can you expose some interesting aspect of the problem with a new boolean flag?
- Can you explore temporal or other structure in some other way?
- Neural nets perform feature learning. They can do this stuff.

But they will also learn a problem much faster if you can better expose the structure of the problem to the network for learning. Spot-check lots of different transforms of your data or of specific attributes and see what works and what doesn't.

Code to scale data https://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/ (https://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/)

Other tips https://machinelearningmastery.com/improve-deep-learning-performance/ (https://machinelearningmastery.com/improve-deep-learning-performance/)

Hyperarameter optimization

https://towardsdatascience.com/hyperparameter-optimization-with-keras-b82e6364ca53 (https://towardsdatascience.com/hyperparameter-optimization-with-keras-b82e6364ca53)

In [1]:
```
# Load libraries

import os
import sys

from keras.layers import Activation, Dense
from keras.models import Sequential
model = Sequential()

#kernel_initializer (initializations of weights)
#kernel_initializer : random_uniform/uniform, random_normal, zero

#It means 8 input parameters, with 12 neurons in the FIRST hidden layer.
model.add(Dense(12, kernel_initializer="random_uniform", input_dim=8, activation="relu"))
```

```
Using TensorFlow backend.
```

HELP

In [2]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

KERAS

Keras is a deep-learning framework for Python that provides a convenient way to define and train almost any kind of deep-learning model. Keras was initially developed for researchers, with the aim of enabling fast experimentation.

Keras has the following key features:

- It allows the same code to run seamlessly on CPU or GPU .
- It has a user-friendly API that makes it easy to quickly prototype deep-learning models.
- It has built-in support for convolutional networks (for computer vision), recur- rent networks (for sequence processing), and any combination of both.
- It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial net- work to a neural Turing machine

Keras has well over 200,000 users, ranging from academic researchers and engi- neers at both startups and large companies to graduate students and hobbyists. Keras is used at Google, Netflix, Uber, CERN , Yelp, Square, and hundreds of startups work- ing on a wide range of problems. Keras is also a popular framework on Kaggle, the machine-learning competition website, where almost every recent deep-learning com- petition has been won using Keras models

Keras can be run with any of the backends: tHEANO, CNK & TENSORFLOW

MNIST

MNIST es un conjunto de 60000 imágenes de dígitos manuscritos recopilados por el National Institute of Standards and Technology (NIST) en los años 80. Se ha convertido en un conjunto de datos básico para comprobar el funcionamiento de cualquier algoritmo de Deep Learning. Esta base de datos también contiene 10000 imágenes para realizar los tests una vez entrenada la red neuronal. En este ejemplo veremos como construir una FFNN para resolver este problema usando la librería Keras.

- Separar los datos en conjuntos de entrenamiento y de comprobación o test.
- Escalar los datos y convertirlos en matrices o conjuntos de categorías.
- Diseñar la red neuronal eligiendo el número y tipo de capas y los filtros.
- Entrenar la red con "fit".
- Validar el modelo con el conjunto de datos reservado como test.

In [3]:
```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images.shape

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

TENSORS

Tha data is defined as a multidimensional Numpy arrays, also called **tensors**.

In general, all current machine-learning systems use tensors as their basic data structure. Tensors are fundamental to the field—so fundamental that Google's TensorFlow was named after them. So what's a tensor? At its core, a tensor is a container for data—almost always numerical data. So, it's a container for numbers. You may be already familiar with matrices, which are 2D ten- sors: tensors are a generalization of matrices to an arbitrary number of dimensions (note that in the context of tensors, a dimension is often called an axis)

Scalars (0D tensors)

A tensor that contains only one number is called a scalar (or scalar tensor, or 0-dimensional tensor, or 0D tensor). In Numpy, a float32 or float64 number is a scalar tensor (or scalar array). You can display the number of axes of a Numpy tensor via the ndim attribute; a sca- lar tensor has 0 axes (ndim == 0). The number of axes of a tensor is also called its rank.

Vectors (1D tensors)

An array of numbers is called a vector, or 1D tensor. A 1D tensor is said to have exactly one axis. This vector has five entries and so is called a 5-dimensional vector. Don't confuse a 5D vector with a 5D tensor! A 5D vector has only one axis and has five dimensions along its axis, whereas a 5D tensor has five axes (and may have any number of dimensions along each axis).

Matrices (2D tensors)

An array of vectors is a matrix, or 2D tensor. A matrix has two axes (often referred to rows and columns).

3D tensors and higher-dimensional tensors

If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers.

By packing 3D tensors in an array, you can create a 4D tensor, and so on. In deep learn- ing, you'll generally manipulate tensors that are 0D to 4D , although you may go up to 5D if you process video data.

Key attributes

A tensor is defined by three key attributes:

- Number of axes (rank)—For instance, a 3D tensor has three axes, and a matrix has two axes. This is also called the tensor's ndim in Python libraries such as Numpy.
- Shape—This is a tuple of integers that describes how many dimensions the ten- sor has along each axis. For instance, the previous matrix example has shape (3, 5) , and the 3D tensor example has shape (3, 3, 5) . A vector has a shape with a single element, such as (5,) , whereas a scalar has an empty shape, () .
- Data type (usually called dtype in Python libraries)—This is the type of the data contained in the tensor; for instance, a tensor's type could be float32 , uint8 , float64 , and so on. On rare occasions, you may see a char tensor. Note that string tensors don't exist in Numpy (or in most other libraries), because tensors live in preallocated, contiguous memory segments: and strings, being variable length, would preclude the use of this implementation.

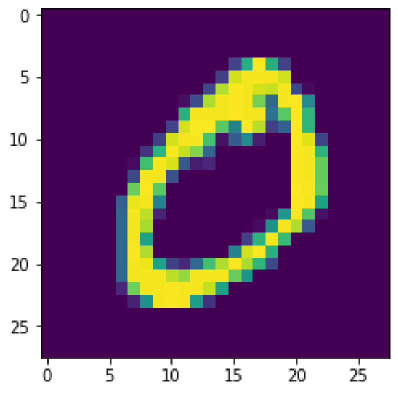
In [4]:
```
print(train_images.ndim)
print(train_images.shape)
print(train_images.dtype)
```

```
3
(60000, 28, 28)
uint8
```

In [5]:
```
import matplotlib.pyplot as plt
i = 1
img = train_images[i]
plt.imshow(img, cmap="Greys")
plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

In [6]:
```
import matplotlib.pyplot as plt
i = 1
img = train_images[i]
plt.imshow(img)
plt.show()
```



TRAINING A NET

- The core building block of neural networks is the layer, a data-processing module that you can think of as a filter for data. Some data goes in, and it comes out in a more use- ful form. Specifically, layers extract representations out of the data fed into them—hope- fully, representations that are more meaningful for the problem at hand. Most of deep learning consists of chaining together simple layers that will implement a form of progressive data distillation. A deep-learning model is like a sieve for data process- ing, made of a succession of increasingly refined data filters—the layers.
- Here, our network consists of a sequence of two Dense layers, which are densely connected (also called fully connected) neural layers. The second (and last) layer is a 10-way softmax layer, which means it will return an array of 10 probability scores (sum- ming to 1). Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

In [7]:
```
from keras import models
from keras import layers
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

To make the network ready for training, we need to pick three more things, as part of the compilation step:

- **A loss function**—How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direc- tion.
- **An optimizer**—The mechanism through which the network will update itself based on the data it sees and its loss function.
- **Metrics to monitor during training and testing** —Here, we'll only care about accu- racy (the fraction of the images that were correctly classified).

In [8]:
```
#Compilation step
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

Before training, we'll preprocess the data by reshaping it into the shape the network expects and scaling it so that all values are in the [0, 1] interval. Previously, our train- ing images, for instance, were stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval. We transform it into a float32 array of shape (60000, 28 * 28) with values between 0 and 1.

In [9]:
```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

We also need to categorically encode the labels

In [10]:
```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

We're now ready to train the network, which in Keras is done via a call to the network's fit method—we fit the model to its training data.

Here, he network will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an epoch). At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights

In [11]:
```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5
60000/60000 [==============================] - 19s 315us/step - loss: 0.2594 - acc: 0.92
50
Epoch 2/5
60000/60000 [==============================] - 12s 203us/step - loss: 0.1053 - acc: 0.96
89
Epoch 3/5
60000/60000 [==============================] - 12s 205us/step - loss: 0.0691 - acc: 0.97
95
Epoch 4/5
60000/60000 [==============================] - 13s 212us/step - loss: 0.0508 - acc: 0.98
44
Epoch 5/5
60000/60000 [==============================] - 12s 202us/step - loss: 0.0381 - acc: 0.98
85
```

Out[11]:
```
<keras.callbacks.History at 0x7fc02421af28>
```

We quickly reach an accuracy of 0.989 (98.9%) on the training data. Now let's check that the model performs well on the test set, too:

In [12]:
```
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
```

```
10000/10000 [==============================] - 2s 206us/step
test_acc: 0.9805
```

The test-set accuracy turns out to be 97.8%—that's quite a bit lower than the training set accuracy. This gap between training accuracy and test accuracy is an example of overfitting: the fact that machine-learning models tend to perform worse on new data than on their training data.

MANIPULATING TENSORS IN NUMPY

In [13]:
```
# equivalents
my_slice = train_images[10:100]
print(my_slice.shape)
#my_slice = train_images[10:100, :, :]
#print(my_slice.shape)
#my_slice = train_images[10:100, 0:28, 0:28]
#print(my_slice.shape)

#my_slice = train_images[:, 14:, 14:]
#print(my_slice.shape)
#my_slice = train_images[:, 7:-7, 7:-7]
#print(my_slice.shape)
```

```
(90, 784)
```

The notion of data batches

Deep-learning models don’t process an entire dataset at once; rather, they break the data into small batches. Concretely, here’s one batch of our MNIST digits, with batch size of 128:

```
In [14]: batch = train_images[:128]
         #And here’s the next batch:
         batch = train_images[128:256]
         #And the n th batch:
         n =1
         batch = train_images[128 * n:128 * (n + 1)]
```

Real-world examples of data tensors

Let’s make data tensors more concrete with a few examples similar to what you’ll encounter later. The data you’ll manipulate will almost always fall into one of the fol- lowing categories:

- Vector data— 2D tensors of shape (samples, features)
- Timeseries data or sequence data— 3D tensors of shape (samples, timesteps, features)
- Images— 4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)
- Video— 5D tensors of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Image data

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in 2D tensors, by convention image tensors are always 3D , with a one- dimensional color channel for grayscale images. A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape (128, 256, 256, 1) , and a batch of 128 color images could be stored in a tensor of shape (128, 256, 256, 3)

Tensor reshaping

A third type of tensor operation that’s essential to understand is tensor reshaping. Although it wasn’t used in the Dense layers in our first neural network example, we used it when we preprocessed the digits data before feeding it into our network: train_images = train_images.reshape((60000, 28 * 28)) Reshaping a tensor means rearranging its rows and columns to match a target shape. Naturally, the reshaped tensor has the same total number of coefficients as the initial tensor. Reshaping is best understood via simple examples:

```
In [15]: x = np.array([[0., 1.],
         [2., 3.],
         [4., 5.]])
         print(x.shape)

         x = x.reshape((6, 1))
         print(x.shape)

         x = x.reshape((2, 3))
         print(x.shape)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-8452973d4105> in <module>()
----> 1 x = np.array([[0., 1.],
      2 [2., 3.],
      3 [4., 5.]])
      4 print(x.shape)
      5

NameError: name 'np' is not defined
```

A special case of reshaping that’s commonly encountered is transposition. Transposing a matrix means exchanging its rows and its columns, so that x[i, :] becomes x[:, i] :

```
In [ ]: x = np.zeros((300, 20))
        x = np.transpose(x)
        print(x.shape)
```

Developing with Keras: a quick overview

You've already seen one example of a Keras model: the MNIST example. The typical Keras workflow looks just like that example:

- Define your training data: input tensors and target tensors.
- Define a network of layers (or model) that maps your inputs to your targets.
- Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
- 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 **func- tional API** (for directed acyclic graphs of layers, which lets you build completely arbi- trary architectures).

Once your model architecture is defined, it doesn't matter whether you used a Sequential model or the functional API . All of the following steps are the same.

Finally, the learning process consists of passing Numpy arrays 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)
```

OBJECTIVE FUNCTION

Take care of the **objective function** (loss function). Choosing the right objective function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss; so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.

Fortunately, 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. For instance,

- you'll use **binary crossentropy** for a two-class classification problem,
- **categorical crossentropy** 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're working on truly new research problems will you have to develop your own objective functions.

GPU

it's highly recommended, although not strictly necessary, that you run deep-learning code on a modern NVIDIA GPU

If you don't want to install a GPU on your machine, you can alternatively consider running your experi- ments on an **AWS EC2 GPU** instance or on Google Cloud Platform. But note that cloud GPU instances can become expensive over time

Use the official **EC2 Deep Learning AMI** (https://aws.amazon.com/amazon- (https://aws.amazon.com/amazon-) ai/amis), and run Keras experiments as Jupyter notebooks on EC2 . Do this if you don't already have a GPU on your local machine.

The IMDB dataset

- A set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.
- The reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

```
In [ ]: from keras.datasets import imdb
        help(imdb.load_data) # this load the numeric data. We prefer to see the process since the be
```

The function above, load the numeric data. We prefer to see the process since the beggining

Loading the IMDB dataset

The dataset is available at https://www.kaggle.com/c/word2vec-nlp-tutorial/data (https://www.kaggle.com/c/word2vec-nlp-tutorial/data)

In [ ]:

```

```

In [ ]:

```
import pandas as pd
import numpy as np
imdb_df = pd.read_csv('IMBD/labeledTrainData.tsv', sep = '\t')

# reproducibility
np.random.seed(42)
```

In [ ]:

```
pd.set_option('display.max_colwidth', 500)
imdb_df.head(5)
```

Data Tokenization

The text data need to be converted into vectors using either **bag of words or embeddings model**. We will first explore bag of words (BOW) model. In the BOW model, a sentence will be represented as a vector with the words (also called tokens) as dimensions of the vectors.

For the purpose of creating vectors, we need to tokenize the sentences first and find out all unique tokens (words) used across all sentences. The corpus of unquie words used could very large, so we can limit the corpus of tokens by using only the most popular (frequently used) words. In this example, we will use 10000 words.

TOKENIZER

Tokenizer provides 4 attributes that you can use to query what has been learned about your documents:

- word_counts: A dictionary of words and their counts.
- word_docs: A dictionary of words and how many documents each appeared in.
- word_index: A dictionary of words and their uniquely assigned integers.
- document_count:An integer count of the total number of documents that were used to fit the Tokenizer.

In [ ]:

```
from keras.preprocessing.text import Tokenizer
```

In [ ]:

```
all_tokenizer = Tokenizer()
all_tokenizer
all_tokenizer.fit_on_texts( imdb_df.review )
```

In [ ]:

```
type(all_tokenizer)
```

In [ ]:

```
all_tokenizer.document_count
```

```
There are 25000 documents (reviews) and 88582 unique words.
```

In [ ]:

```
len(all_tokenizer.word_counts)
```

In [ ]:

```
#high frequency words
list(all_tokenizer.word_counts.items())[0:10]
```

In [ ]:

```
#low frequency words
list(all_tokenizer.word_counts.items())[-10:]
```

We can assume the low frequencey words are rarely used to express sentiments as they have appeared only once across all reviews. And only choose to keep top N (for example 10000) words for our analysis. So, let's tokenize agains with a limit to number of words to 10000.

In [ ]:
```
num_words = 10000
tokenizer = Tokenizer(num_words = num_words)
tokenizer.fit_on_texts( imdb_df.review )
```

In [ ]:
```
#Checking first few words and their counts
import itertools

x = itertools.islice(tokenizer.word_counts.items(), 0, 5)

for key, value in x:
    print(key, value)
```

In [ ]:
```
#Checking words and their indexes
list(tokenizer.word_index.items())[-10:]
```

In [ ]:
```
list(tokenizer.word_index.items())[1:10]
```

Encoding

- Encoding a text using the dictionary of tokens
- Finding indexes of the words

In [ ]:
```
from collections import OrderedDict
words_by_sorted_index = sorted(tokenizer.word_index.items(),
                                           key=lambda idx: idx[1])
type(words_by_sorted_index)
words_by_sorted_index[0:10]
```

In [ ]:
```
tokenizer.word_index['the']
```

In [ ]:
```
tokenizer.word_index['a']
```

In [ ]:
```
tokenizer.texts_to_sequences( ["The movie gladiator is a brilliant movie"])
```

Encoding all the movie reviews

Now the documents (reviews) will be encoded as per the dictionary.

In [ ]:
```
%%time
sequences = tokenizer.texts_to_sequences(imdb_df.review)
```

In [ ]:
```
#Let's look at the words index sequences for a specific sentence.
imdb_df.review[10:11]
```

In [ ]:
```

np.array(sequences[10:11])
```

Encode Y Variable

In [ ]:
```

y = np.array(imdb_df.sentiment)
```

In [ ]:
```
y[0:5]
```

In [ ]:
```
#How many classes available?
imdb_df.sentiment.unique()
```

Truncate and Pad Sequences

One of the problem in dealing with sentences are they are not of same size. Some sentences will have more words and some will have fewer words. Neural networks take input of same lenghts for training a batch.

So, we need to choose a length or size of input. Larger sentences will have to be truncated and smaller ones need to be padded. But what size or lenght to consider?

We need to take the length which can cover most of the sentences. Only few need to be truncated or padded. For that we will look at the distribution of the word or token lengths.

In [ ]:
```
num_tokens = [len(tokens) for tokens in sequences]
num_tokens = np.array(num_tokens)
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
sn.distplot( num_tokens );
```

In [ ]:
```
mean_num_tokens = num_tokens.mean()
std_num_tokens = num_tokens.std()
print(mean_num_tokens)
print(std_num_tokens)

#if we assume that legnth chosen should address 95% of the sentences, then we can take 2 sta
#of the mean length.

max_review_length = int(mean_num_tokens + 2 * std_num_tokens)
max_review_length
```

In [ ]:
```
#How many sentences will not be truncated at all?

print(np.sum(num_tokens < max_review_length) / len(num_tokens))

#Almost 95%.
```

In [ ]:
```
#Now we will pad or truncate. But padding or truncating can be done at the beginning of the
#or at the end of the sentences. pre or post can be used to specify the padding and truncati
#or end of sentence.

from keras.preprocessing.sequence import pad_sequences
pad = 'pre'
X = pad_sequences(sequences,
                  max_review_length,
                  padding=pad,
                  truncating=pad)
X[0:1]
```

Split Datasets

In [ ]:
```

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = 0.2)

print(X_train.shape)
print(X_test.shape)

input_shape = X_train.shape
```

Bag Of Words Model

Model Architecture

(Bag of words) -> Dense Layer(1024) -> Dense Layer(256) -> Dense Layer(128) -> Dense Layer(64) -> Relu -> Dense Layer(1) -> Sigmoid

In [ ]:
```
from keras import backend as K
from keras.models import Sequential
from keras.layers import Flatten, Dense, Activation

np.random.seed(42)
K.clear_session()  # clear default graph


bow_model = Sequential()

bow_model.add(Dense(16, input_shape=(input_shape[1],)))
bow_model.add(Activation('relu'))
bow_model.add(layers.Dropout(0.5))
# An "activation" is just a non-linear function applied to the output
# of the layer above. Here, with a "rectified linear unit",
# we clamp all values below 0 to 0.
bow_model.add(Dense(16))
bow_model.add(Activation('relu'))
bow_model.add(layers.Dropout(0.5))
bow_model.add(Dense(16))
bow_model.add(Activation('relu'))
bow_model.add(layers.Dropout(0.5))
bow_model.add(Dense(1))
# This special "softmax" activation among other things,
# ensures the output is a valid probaility distribution, that is
# that its values are all non-negative and sum to 1.
bow_model.add(Activation('sigmoid'))
bow_model.summary()
```

In [ ]:
```
bow_model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
EPOCHS = 15 # INITIALLY 20, LATER YO SEE WHERE THE TRAINING AND TEST STAT TO SEPARATE A FIXE
BATCH_SIZE = 512
```

In [ ]:
```
%%time
# reproducibility
np.random.seed(42)
# fit model
bow_history = bow_model.fit(
    X_train,
    y_train,  # prepared data
    batch_size = BATCH_SIZE,
    epochs = EPOCHS,
    shuffle = True,
    verbose=1,
    validation_data = (X_test, y_test)
)
```

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

def plot_accuracy(hist):
    plt.plot(hist['acc'])
    plt.plot(hist['val_acc'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train',
                'test'],
               loc='upper left')
    plt.show()

def plot_loss(hist):
    plt.plot(hist['loss'])
    plt.plot(hist['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train',
                'test'],
               loc='upper left')
    plt.show()
```

In [ ]:
```
plot_accuracy( bow_history.history )
```

In [ ]:
```

plot_loss( bow_history.history )
```

In [ ]:
```
result = bow_model.evaluate(X_test, y_test)
print("Accuracy: {0:.2%}".format(result[1]))
```

In [ ]:
```
y_pred = bow_model.predict_classes(X_test[0:1000])

from sklearn import metrics
cm = metrics.confusion_matrix( y_test[0:1000],
                              y_pred, [1,0] )

sn.heatmap(cm, annot=True,
           fmt='.2f',
           xticklabels = ["Positive", "Negative"] ,
           yticklabels = ["Positive", "Negative"] )

plt.ylabel('True label')
plt.xlabel('Predicted label');
plt.title( 'Confusion Matrix for Sentiment Classification');
```

In [ ]:
```
from sklearn.metrics import classification_report
print( classification_report(y_test[0:1000],
                             y_pred))
```

ROC

In [ ]:
```
y_pred_probs = bow_model.predict(X_test[0:1000])

auc_score = metrics.roc_auc_score( y_test[0:1000],
                                   y_pred_probs  )

fpr, tpr, thresholds = metrics.roc_curve( y_test[0:1000],
                                          y_pred_probs,
                                          drop_intermediate = False )

plt.figure(figsize=(8, 6))
plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

Using Embeddings (Recurrent Neural Networks)

In Word embeddings, words are represented by a vector i.e. series of numbers (weights). The vectors represent words in a N dimension space, in which similar meaning words are places nearer to each other while the dissimilar words are kept far. The dimensions in the space represent some latent factors, by which the words could be defined. All words are assigned some weights in each each latent factors. Words that share some common meaning have similar weights across common factors.

The word embeddings weights can be estimated during the NN model building. There are also pre-built word embeddings are available, which can be used in the model. We will discuss about the pre-built word embeddings later in the tutorial.

Word embeddings are commonly used in many Natural Language Processing (NLP) tasks because they are found to be useful representations of words and often lead to better performance in the various tasks performed. Given its widespread use, this post seeks to introduce the concept of word embeddings to the prospective NLP practitioner.

Here are couple of good references to understand embeddings

https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a (https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a)

(Bag of words) -> Embeddings (8) -> Dense Layer(16) -> Relu -> Dense Layer(1) -> Sigmoid

- We will try another optimizers and applying regularization (dropouts).
- Add a dropout layer as a regularization layer for dealing with overfitting.
- We will also add callbacks for reducing LR and early stopping. And store tensorflow logs for monitoring.

In [ ]:
```
from keras_tqdm import TQDMNotebookCallback
from keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint
from keras.callbacks import TensorBoard

callbacks_list = [ReduceLROnPlateau(monitor='val_loss',
                                    factor=0.1,
                                    patience=3),
                  EarlyStopping(monitor='val_loss',
                                patience=4),
                  ModelCheckpoint(filepath='imdb_model.h5',
                                  monitor='val_loss',
                                  save_best_only=True),
                  TensorBoard("./imdb_logs"),
                  TQDMNotebookCallback(leave_inner=True,
                                       leave_outer=True)]
```

In [ ]:
```
from keras.layers import Dropout

K.clear_session()

emb_model = Sequential()
# We specify the maximum input length to our Embedding layer
# so we can later flatten the embedded inputs
emb_model.add(Embedding(10000,
                          8,
                          input_length=max_review_length,
                          name='layer_embedding'))
# After the Embedding layer,
# our activations have shape `(samples, maxlen, 8)`.

# We flatten the 3D tensor of embeddings
# into a 2D tensor of shape `(samples, maxlen * 8)`
emb_model.add(Flatten())

emb_model.add(Dense(16))
emb_model.add(Activation('relu'))

emb_model.add(Dropout(0.8))

# We add the classifier on top
emb_model.add(Dense(1))
emb_model.add(Activation('sigmoid'))
emb_model.compile(optimizer="adam",
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
emb_history = emb_model.fit(X_train,
                    y_train,
                    epochs=20,
                    batch_size=32,
                    callbacks = callbacks_list,
                    validation_split=0.3)
```

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

def plot_accuracy(hist):
    plt.plot(hist['acc'])
    plt.plot(hist['val_acc'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train',
                'test'],
               loc='upper left')
    plt.show()

def plot_loss(hist):
    plt.plot(hist['loss'])
    plt.plot(hist['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train',
                'test'],
               loc='upper left')
    plt.show()

plot_accuracy( emb_history.history )
```

In [ ]:
```
plot_loss( emb_history.history )
```

In [ ]:
```
result = emb_model.evaluate(X_test, y_test)
print("Accuracy: {0:.2%}".format(result[1]))
```

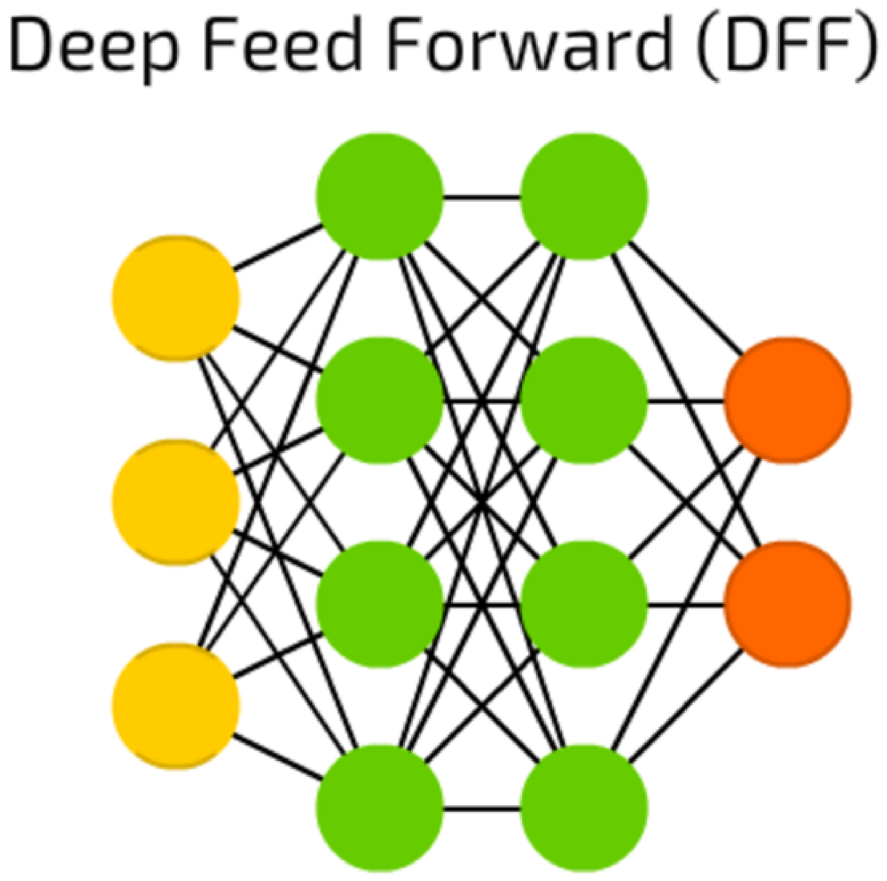
In [ ]:
```
y_pred = emb_model.predict_classes(X_test[0:1000])

from sklearn import metrics
cm = metrics.confusion_matrix( y_test[0:1000],
                            y_pred, [1,0] )

sn.heatmap(cm, annot=True,
           fmt='.2f',
           xticklabels = ["Positive", "Negative"] ,
           yticklabels = ["Positive", "Negative"] )

plt.ylabel('True label')
plt.xlabel('Predicted label');
plt.title( 'Confusion Matrix for Sentiment Classification');
```

In [ ]:
```
y_pred_probs = emb_model.predict(X_test[0:1000])

auc_score = metrics.roc_auc_score( y_test[0:1000],
                                   y_pred_probs  )

fpr, tpr, thresholds = metrics.roc_curve( y_test[0:1000],
                                          y_pred_probs,
                                          drop_intermediate = False )

plt.figure(figsize=(8, 6))
plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

TIPOS DE REDES

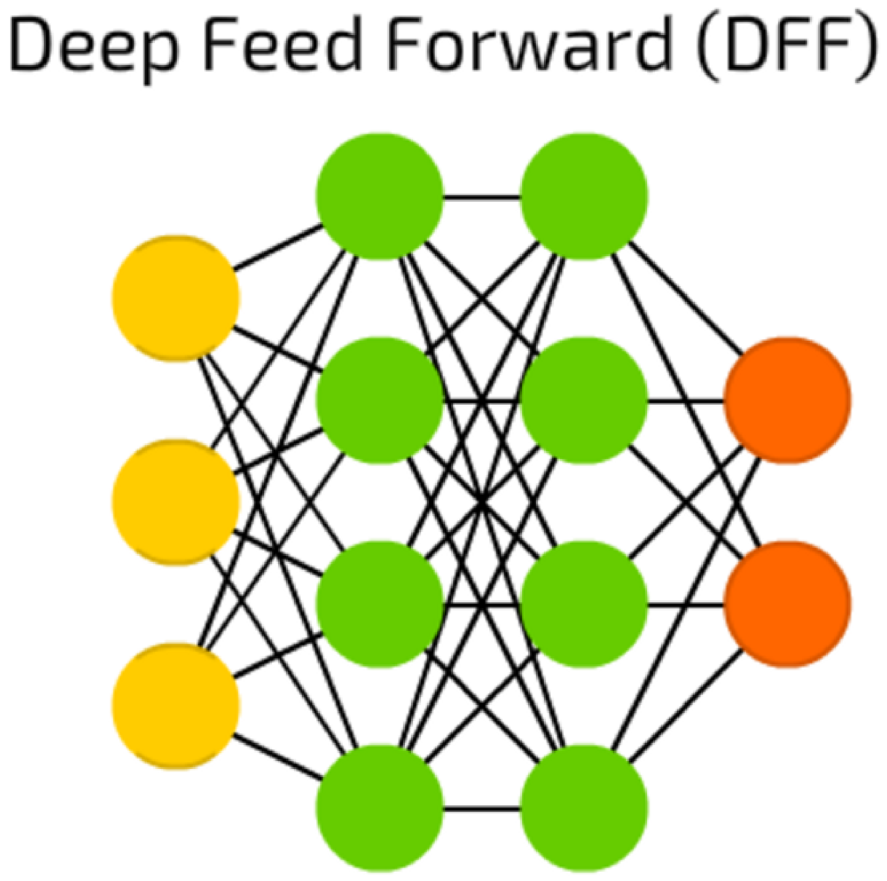
- There are a lot of aquitectures. The following link describe some of then. https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464 (https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464)
- The most important for us are:

Recurrent Neural Networks here each of hidden cell received it's own output with fixed delay—one or more iterations. Apart from that, it was like common FNN.This is used when decisions from past iterations or samples can influence current ones. The most common examples of such contexts are texts—a word can be analysed only in context of previous words or sentences.



Long-Short term memory (LSTM) This type introduces a memory cell, a special cell that can process data when data have time gaps (or lags). **RNNs can process texts by “keeping in mind” ten previous words**, and **LSTM networks can process video frame “keeping in mind” something that happened many frames ago**. LSTM networks are also widely used for writing and speech recognition.

Memory cells are actually composed of a couple of elements—called gates, that are recurrent and control how information is being remembered and forgotten (note that there are no activation functions between blocks).

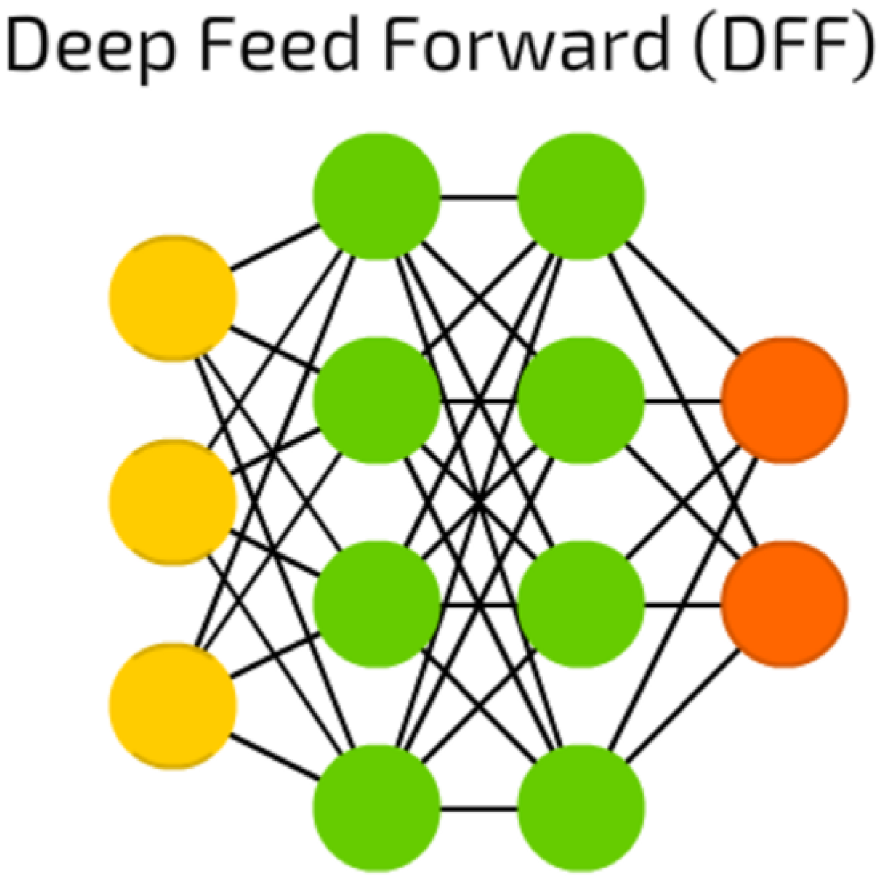


Deep Convolutional Networks

DCN nowadays are stars of artificial neural networks. They feature convolution cells (or pooling layers) and kernels, each serving a different purpose.

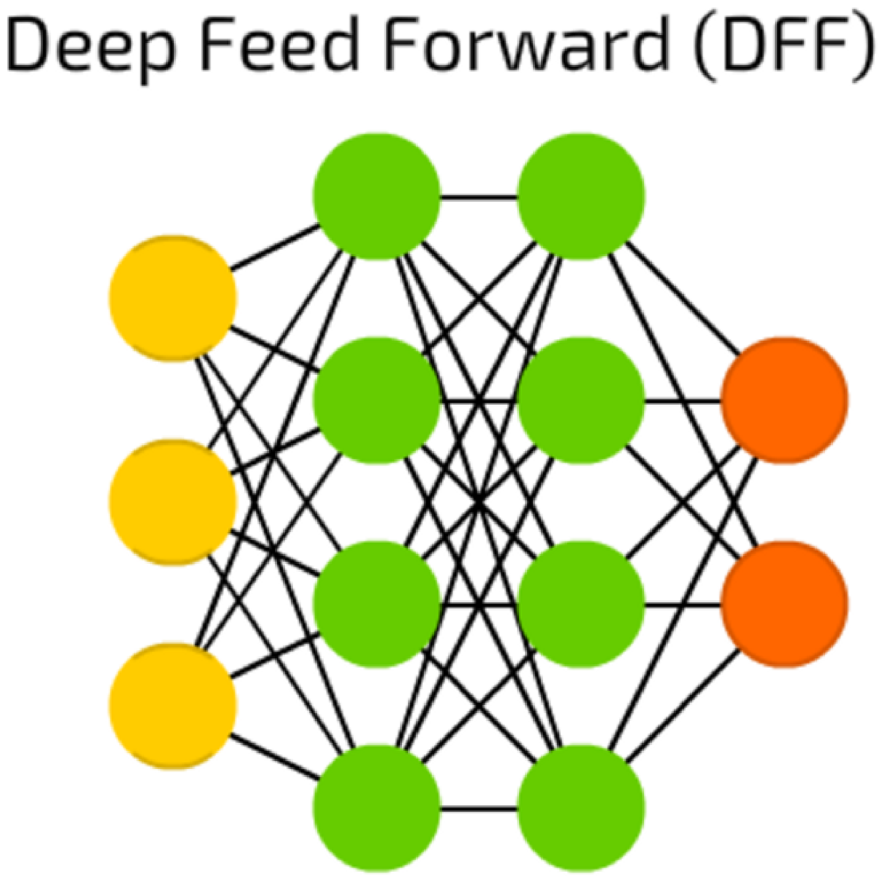
Convolution kernels actually process input data, and pooling layers simplify it (mostly using non-linear functions, like max), reducing unnecessary features.

Typically used for image recognition, they operate on small subset of image (something about 20x20 pixels). The input window is sliding along the image, pixel by pixel. The data is passed to convolution layers, that form a funnel (compressing detected features). From the terms of image recognition, first layer detects gradients, second lines, third shapes, and so on to the scale of particular objects. DFFs are commonly attached to the final convolutional layer for further data processing.



Deep Feed Forward

(Multilayer Perceptron)



REGULARIZATION

The processing of fighting overfitting this way is called **regularization**.

A simple model in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called weight regularization, and it's done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

- L1 regularization—The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).
- L2 regularization—The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks. Don't let the different name con- fuse you: weight decay is mathematically the same as L2 regularization.

Adding L2 weight regularization to the model

```
 model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

l2(0.001) means every coefficient in the weight matrix of the layer will add 0.001 * weight_coefficient_value to the total loss of the network. Note that because this penalty is only added at training time, the loss for this network will be much higher at training than at test time.

Different weight regularizers available in Keras

L1 regularization

```
 from keras import regularizers
regularizers.l1(0.001)
```

Simultaneous L1 and L2 regularization

```
 regularizers.l1_l2(l1=0.001, l2=0.001)
```

DROPOUT

Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training. It was developed by Geoff Hinton and his students at the Uni- versity of Toronto.

The creator says he was inspired by, among other things, a fraud-prevention mechanism used by banks. In his own words, "I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to suc- cessfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example **would prevent conspiracies** and thus reduce over- fitting. The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren't significant (what the autho refers to as con- spiracies), which the network will start memorizing if no noise is present.

To recap, these are the most common ways to prevent overfitting in neural networks:

- Get more training data.
- Reduce the capacity of the network.
- Add weight regularization.
- Add dropout.

```
In [ ]:
```

```
In [ ]:
```

In [ ]:

```

```

In [ ]:

```

```