Introduction To Keras

Installing keras:

The easiest way to install keras is by using Anconda (https://www.anaconda.com)

```
conda create -n nnProject python=3.7
conda activate nnProject
conda install keras [or] conda install keras-gpu
```

Tensors

Each layer in the neural network (including the first layer) works with tensors. A tensor is just a container for the data. It is actually a matrix on many dimensions. When using tensors, when we refer to dimensions we call them axis. A tensor with 0 dimension is called a scalar, A tensor with 1 dimensions is called a vector while a tensor with 2 dimensions is a simple matrix.

You can display the number of dimensions on a tensor (numpy array) by using *ndim*

```
import numpy as np
>>> x=np.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> print(x.ndim)
2
```

If you have a vector of matrices then you get a 3 dimensional tensor. If you have a vector of these 3 dimensional tensors you get a 4 dimensional tensor (and so on). Usually, tensors of 2, 3 and 4 dimensions are the one that are most used.

Ex:

- 2dimensional tensor: Vector data (samples, features), such as mnist dataset
- 3dimensional tensor: Time series data or sequence data (sample,timesteps, features), such as imdb dataset
- 4dimensional tensor: Images data (samples, height, width, channels)
- 5 dimensional tensor: Video data (samples, frames, height, width, channels)

Example, mmist dataset:

```
from keras.datasets import mnist
>>> (train,train_labels), (test, test_labels) = mnist.load_data()
>>> print(train.shape)
(6000,28,28)            ←A 3 dimensional tensor
```

The first dimension is usually the batch size of the tensor.

```
batch = train[:128]
```

```
batch = train[:128,: ,: ]
```

There are 2 ways to build a neural network using keras:

1. Using the *Sequential()* class (for the majority of the models)
2. Using the *functional API* (for arbitrary models)

The building block of the neural network is the layer. Each time we add a layer we add it to find more structure (features) in the data that will allow us to better train the network.
In order to train the neural network, you need:

1. A model (made of multiple layers)
   The call to Sequential() returns a model to which layers can pe added using the method add(). Each new layer is added on top of the previous. For the previous layer you must specify the input dimensions (except the batch size)

   ```
   network = models.Sequential()
   network.add(layers.Dense(512, activation='relu',
   input_shape=(28*28,)))
   network.add(layers.Dense(100, activation='relu'))
   network.add(layers.Dense(10, activation='softmax'))
   ```

   Other parameters to the Dense layer are the ones used for regularization:

   ```
   network.add(layers.Dense(512,
   activation='relu',
   input_shape=(28*28,),
   kernel_regularizer=regularizers.l2(0.001)))
   ```

   For dropout, just add a dropout layer after the layer you want to perform regularization:

   ```
   network.add(layers.Dense(100, activation='relu'))
   network.add(layers.Dropout(0.5))
   ```

2. An optimizer
   An optimizer is an algorithm that tries to minimize the loss function. Depending on what you choose you can increase the accuracy and the training speed. A good choice is rmsprop, adam or nadam. The optimizer is given as argument to the compile function.

   ```
   network.compile(optimizer='rmsprop',...
   ```

   Each optimizer has its own set of arguments. If you want to use the default one the above method works just fine. However, if you wish to adjust these arguments you can create an optimizer object and pass it to the compile function.

   ```
   from keras import optimizers
   myOptim = optimizers.RMSprop(lr=0.001)
   #myOptim = optimizers. SGD(learning_rate=0.01, momentum=0.0,
   nesterov=True)
   network.compile(optimizer=myOptim,...
   ```

3. A loss function

The loss function is specific to your problem and describes the cost of the training data when compared to the labels. Usually there are 3 main loss functions used (even though there are more available):

- binary_crossentropy : for 2 class classification
  `network.compile(...,loss='binary_crossentropy',.. )`
- categorical_crossentropy: for multiclass classification
  `network.compile(..,loss='categorical_crossentropy',.. )`
- mean_square_error: for regression
  `network.compile(..,loss='mse',.. )`

4. Metrics:
   These are not mandatory for training but provide good insight for how the training is evolving. The most used one is accuracy.
   `network.compile(..,metrics=['accuracy'],.. )`

After all of the above are set up the `fit()` method will train the network on the training data. The fit method expects:

- The training data (first argument or `x=`)
- The labels for the training data (second argument or `y=`)
- Number of epochs (`epochs=`)

It is recommended that you also provide a validation data, as well as the batch size:

- Validation Data (`validation_data=(test_data, test_labels)`). You can also use the validation_split argument which will use a fraction of the training data for validation (`validation_data = 0.1 )`
- Number of elements in a batch (`batch_size = )`. Default value is 32.

Example :
Training on mnist dataset

```
import keras
```

Using TensorFlow backend.

Load data

In [2]:

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) =  mnist.load_data()
#print the shapes of the training dataset. Just for checking.
train_images.shape, train_labels.shape
```

Out[2]:

((60000, 28, 28), (60000,))

Network Architecture

In [3]:

```
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'))

network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Preparing the image data

In [4]:

```
from keras.utils import to_categorical

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

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Train the network

In [5]:

```
network.fit(train_images, train_labels,
                validation_data=(test_images, test_labels),
                epochs=10,
                batch_size=128)
```

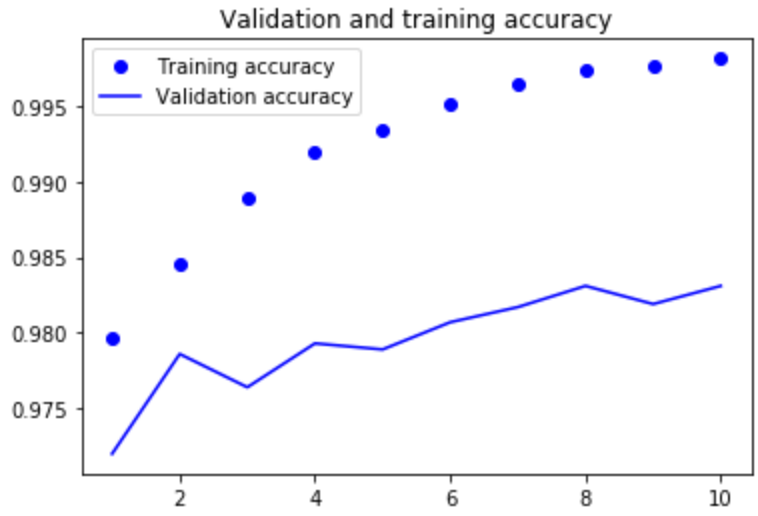
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 2s 31us/step - loss: 0.0673 - acc: 0.9797 - val_loss: 0.0907 - val_acc: 0.9720
Epoch 2/10

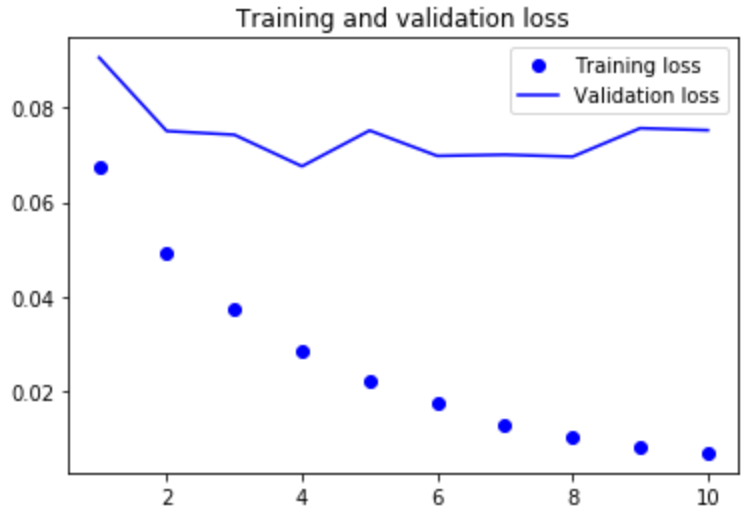
```
60000/60000 [==============================] - 2s 30us/step - loss: 0.0493 - acc: 0.9846 -
val_loss: 0.0751 - val_acc: 0.9786
Epoch 3/10
60000/60000 [==============================] - 2s 30us/step - loss: 0.0374 - acc: 0.9888 -
val_loss: 0.0743 - val_acc: 0.9764
Epoch 4/10
60000/60000 [==============================] - 2s 30us/step - loss: 0.0283 - acc: 0.9919 -
val_loss: 0.0676 - val_acc: 0.9793
Epoch 5/10
60000/60000 [==============================] - 2s 30us/step - loss: 0.0222 - acc: 0.9934 -
val_loss: 0.0752 - val_acc: 0.9789
Epoch 6/10
60000/60000 [==============================] - 2s 31us/step - loss: 0.0172 - acc: 0.9951 -
val_loss: 0.0698 - val_acc: 0.9807
Epoch 7/10
60000/60000 [==============================] - 2s 30us/step - loss: 0.0127 - acc: 0.9965 -
val_loss: 0.0700 - val_acc: 0.9817
Epoch 8/10
60000/60000 [==============================] - 2s 30us/step - loss: 0.0100 - acc: 0.9974 -
val_loss: 0.0696 - val_acc: 0.9831
Epoch 9/10
60000/60000 [==============================] - 2s 28us/step - loss: 0.0081 - acc: 0.9977 -
val_loss: 0.0757 - val_acc: 0.9819
Epoch 10/10
60000/60000 [==============================] - 2s 30us/step - loss: 0.0068 - acc: 0.9981 -
val_loss: 0.0752 - val_acc: 0.9831
```

Plot the results

```
%matplotlib inline
import matplotlib.pyplot as plt

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

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

plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Validation and training accuracy')
plt.legend()

plt.figure()
```

```
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





Bibliography: Deeplearning with Python by François Chollet