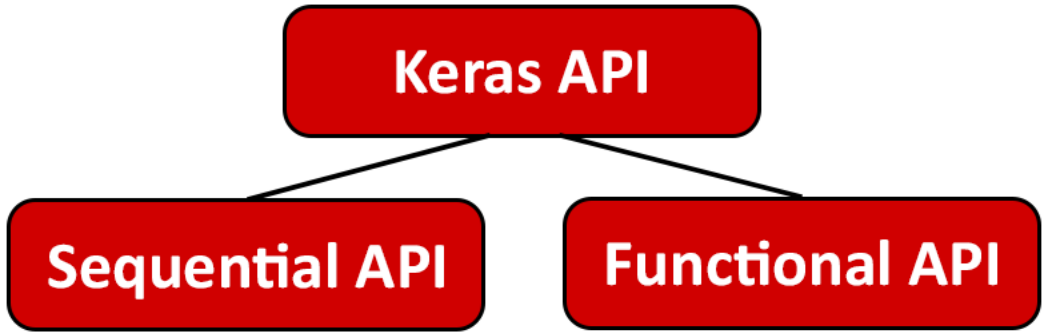
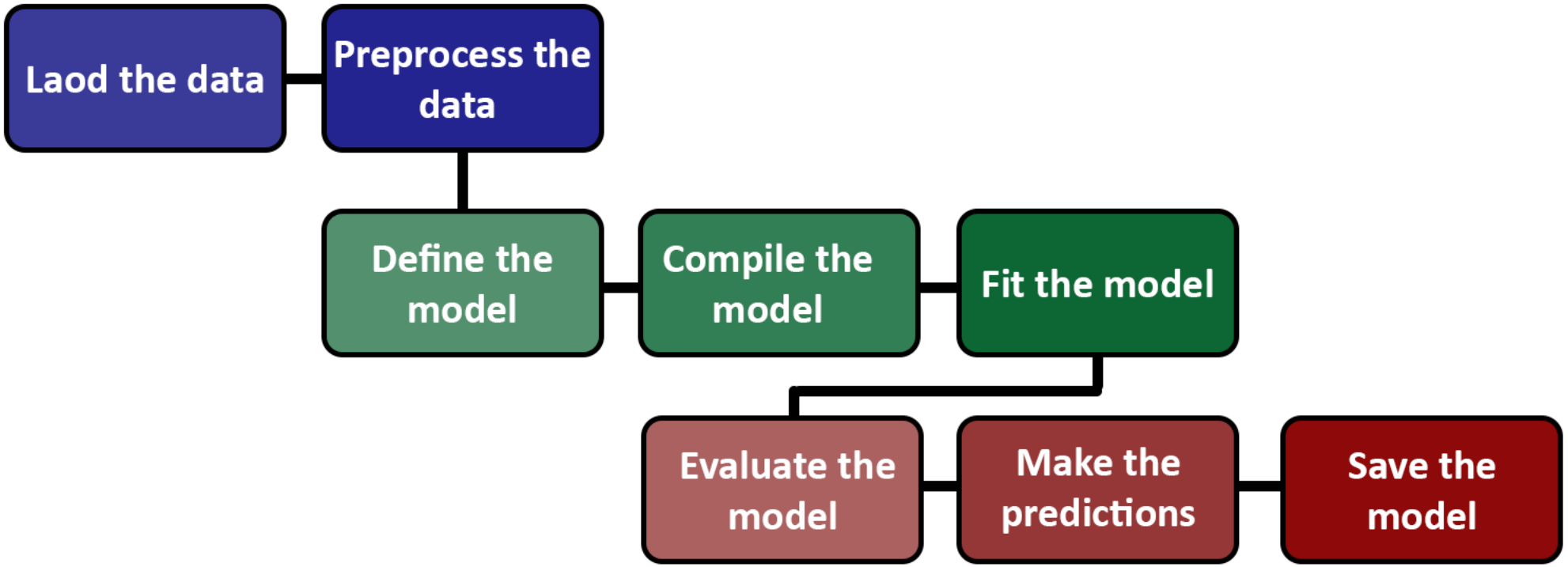
Understanding and Working with Keras

Keras is a compact and easy-to-learn high-level Python library for deeplearning that can run on top of TensorFlow. It allows developers to focus on the main concepts of deep learning, such as creating layers for neural networks, while taking care of the nitty-gritty details of tensors, their shapes, and their mathematical details.



The sequential API is based on the idea of a sequence of layers; this is the most common usage of Keras and the easiest part of Keras. The sequential model can be considered as a linear stack of layers.
In short, you create a sequential model where you can easily add layers, and each layer can have convolution, max pooling, activation, dropout, and batch normalization.

There are eight steps to the deep learning process in Keras:



1.Load Data

In this step, we should import the data which is going to be used to train and test our model. There are different methods to do that. In this example, we are going to import the CIFAR10 dataset. We are going to see some other methods to import the data into our project in different examples in the course.

Python code to import the data

```
# Importing modules
import numpy as np
import os
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import adam
from keras.utils import np_utils
```

```
#Load Data
np.random.seed(100) # for reproducibility
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

#cifar-10 has images of airplane, automobile, bird, cat,
# deer, dog, frog, horse, ship and truck ( 10 unique labels)
# For each image. width = 32, height =32, Number of channels(RGB) = 3
```

2.Preprocess the Data

Our data needs to go through different stages of preprocessing before it is ready to be fed into our model. Sometimes this could be as easy as normalizing and splitting the test data from the train data and sometimes it includes more complicated procedures.
In this example we are going to reshape each 32*32*3 image into a vector with 3072 values. Then we are going to normalize the input. In the end, we are going to use one hot encoding to prepare our labels. This changes 5 to [0 0 0 0 0 1 0 0 0 0].

Python example to preprocess the data

```
#Preprocess the data
#Flatten the data, MLP doesn't use the 2D structure of the data. 3072 = 3*32*32
X_train = X_train.reshape(50000, 3072) # 50,000 images for training
X_test = X_test.reshape(10000, 3072) # 10,000 images for test

# Gaussian Normalization( Z- score)
X_train = (X_train- np.mean(X_train))/np.std(X_train)
X_test = (X_test- np.mean(X_test))/np.std(X_test)


# Convert class vectors to binary class matrices (ie one-hot vectors)
labels = 10 #10 unique labels(0-9)
Y_train = np_utils.to_categorical(y_train, labels)
Y_test = np_utils.to_categorical(y_test, labels)
```

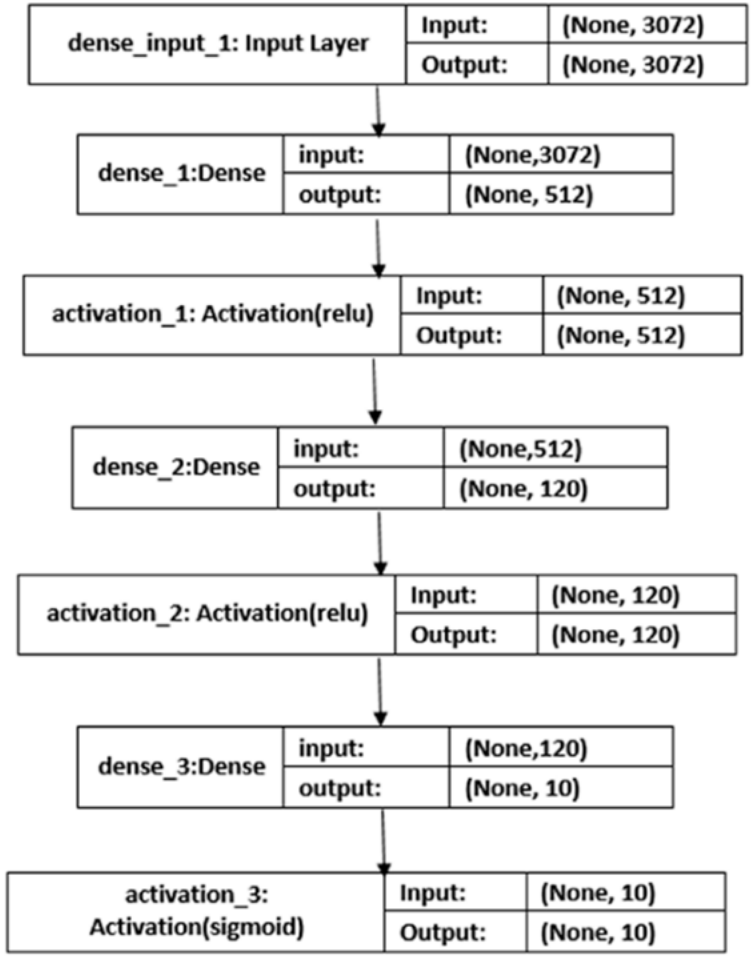
3.Define the Model

Sequential models in Keras are defined as a sequence of layers. We create a sequential model and then add layers. We need to ensure the input layer has the right number of inputs. In this example we have 3,072 input variables and we have 10 different output nodes. We can design the layers in between as we want as there is no answer to what the best structure is. We just need to experiment to find a structure which is complex enough to find the patterns and simple enough not to overfit the train data.

Example of defining a model

```
#Define the model achitecture
model = Sequential()
model.add(Dense(512, input_shape=(3072,))) # 3*32*32 = 3072
model.add(Activation('relu'))
model.add(Dropout(0.4)) # Regularization
model.add(Dense(120))
model.add(Activation('relu'))
model.add(Dropout(0.2))# Regularization
model.add(Dense(labels)) #Last layer with 10 outputs, each output per class
model.add(Activation('sigmoid'))
```

3.Define the Model



Example of defining a model

```
#Define the model achitecture
model = Sequential()
model.add(Dense(512, input_shape=(3072,))) # 3*32*32 = 3072
model.add(Activation('relu'))
model.add(Dropout(0.4)) # Regularization
model.add(Dense(120))
model.add(Activation('relu'))
model.add(Dropout(0.2))# Regularization
model.add(Dense(labels)) #Last layer with 10 outputs, each output per class
model.add(Activation('sigmoid'))
```

Here we can see the result of the code. we have a model with 3 dense layers with 3 activations. The first layer has 512 nodes and uses Relu as its activation. The second layer has 120 nodes with Relu activation. The lastt layer has 10 nodes (the same as the number of classes) and uses sigmoid as its activation function.

3.Define the Model

After defining the model we can see the structure of our model using the model.summary() method. this will give us a complete overview of our model.

Example of model summary

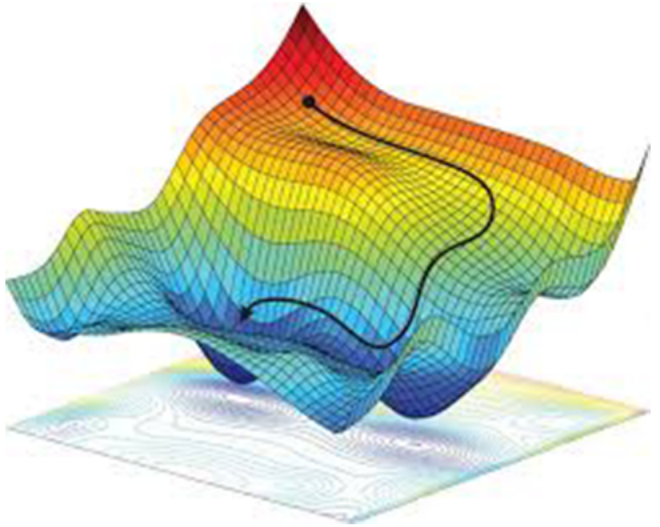
```
#Summary of the model
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 512)               1573376
_________________________________________________________________
activation_7 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 120)               61560
_________________________________________________________________
activation_8 (Activation)    (None, 120)               0
_________________________________________________________________
dropout_6 (Dropout)          (None, 120)               0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1210
_________________________________________________________________
activation_9 (Activation)    (None, 10)                0
=================================================================
Total params: 1,636,146
Trainable params: 1,636,146
Non-trainable params: 0
```

4.Compile the Model

Having defined the model in terms of layers, you need to declare the loss function, the optimizer, and the evaluation metrics. When the model is proposed, the initial weight and bias values are assumed to be 0 or 1, a random normally distributed number, or any other convenient numbers. But the initial values are not the best values for the model. This means the initial values of weight and bias are not able to explain the target/label in terms of predictors (Xs). So, you want to get the optimal value for the model. The journey from initial values to optimal values needs a motivation, which will minimize the cost function/loss function. The journey needs a path (change in each iteration), which is suggested by the optimizer. The journey also needs an evaluation measurement, or evaluation metrics.Popular loss functions are binary cross entropy, categorical cross entropy, mean_squared_logarithmic_error and hinge loss.

Popular optimizers are stochastic gradient descent (SGD), RMSProp, adam, adagrad, and adadelta. Popular evaluation metrics are accuracy, recall, and F1 score.



source: ai.plainenglish.io

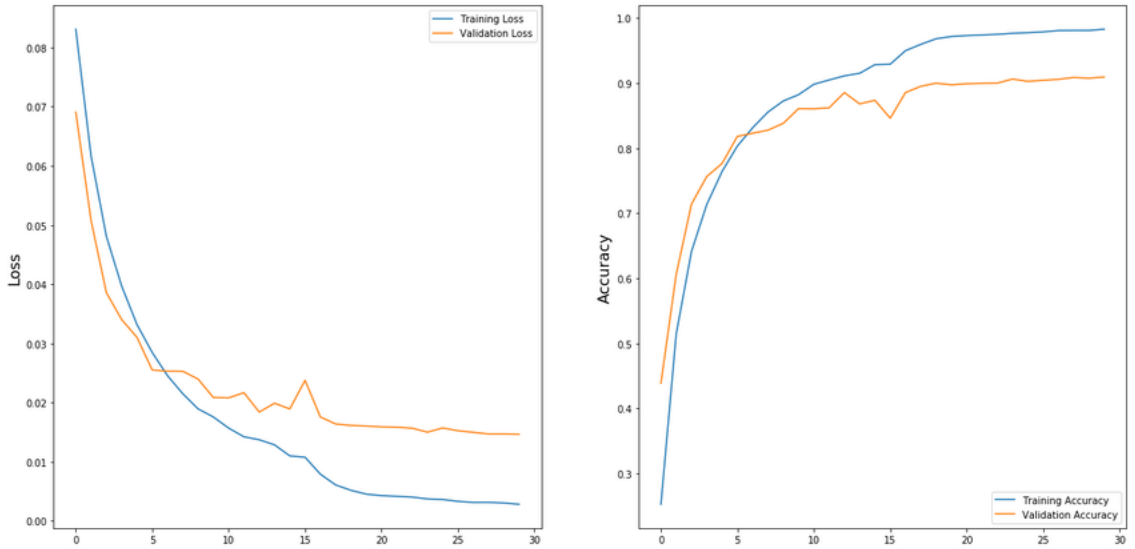
4.Compile the Model

```
# Compile the model
# Use adam as an optimizer
adam = adam(0.01)
# the cross entropy between the true label and the output(softmax) of the model
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=["accuracy"])
```

In this code, we first define an adam optimizer and set its learning rate to 0.01. then we compile our model using the optimizer we defined and the categorical cross entropy loss function. we are going to use accuracy as our metric as well.

5.Fit the Model

Having defined and compiled the model, now we need to train the model by feeding it some data. Here you need to specify the epochs; these are the number of iterations for the training process to run through the data set and the batch size, which is the number of instances that are evaluated before a weight update. For this problem, the program will run for a small number of epochs (10), and in each epoch, it will complete 50(=50,000/1,000) iterations where the batch size is 1,000 and the training data set has 50,000 instances/images. Again, there is no hard rule
to select the batch size. But it should not be very small, and it should be much less than the size of the training data set to consume less memory.



source: www.researchgate.net

5.Fit the Model

```
#Make the model learn ( Fit the model)
model.fit(X_train, Y_train,batch_size=1000, nb_epoch=10,validation_data=(X_test, Y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
 1000/50000 [..............................] - ETA: 6s - loss: 2.3028 - acc: 0.1060
C:\ProgramData\Anaconda3\lib\site-packages\keras\models.py:848: UserWarning: The `nb_epoch` argument in `fit` has been renamed
`epochs`.
  warnings.warn('The `nb_epoch` argument in `fit` '
50000/50000 [==============================] - 6s - loss: 2.3030 - acc: 0.0974 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 2/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.1012 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 3/10
50000/50000 [==============================] - 7s - loss: 2.3028 - acc: 0.0972 - val_loss: 2.3026 - val_acc: 0.1000
Epoch 4/10
50000/50000 [==============================] - 7s - loss: 2.3028 - acc: 0.0997 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 5/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0975 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 6/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0986 - val_loss: 2.3028 - val_acc: 0.1000
Epoch 7/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0995 - val_loss: 2.3028 - val_acc: 0.1000
Epoch 8/10
50000/50000 [==============================] - 7s - loss: 2.3028 - acc: 0.0983 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 9/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0998 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 10/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0972 - val_loss: 2.3027 - val_acc: 0.1000
<keras.callbacks.History at 0x2870136eef0>
```

In this part of the code, we will train our model using the x_train and y_train that we prepared in the previous stages. We are going to use batches of size 1000 to train our model. we are going to train the model for 10 epochs. we are going to use x_test and y_test as our validation data during the training. We can see the loss and the accuracy of our model both on the training data and the validation data during training in the output.

6.Evaluate Model

Having trained the neural networks on the training data sets, we need to evaluate the performance of the network. We can evaluate our model on the testing data set using the evaluation() function on our model and passing it the same input and output that we prepared to test our model. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics we have configured, such as accuracy. we can use other metrics like precision, recall, f1 score as well.

Example of model evaluation

```
#Evaluate how the model does on the test set
score = model.evaluate(X_test, Y_test, verbose=0)
#Accuracy Score
print('Test accuracy:', score[1])
```

7.Prediction

Once we have built and evaluated the model and we are content with the results, we can use the model to predict unknown data. For example, we could use a same sized picture of a car from anywhere and use the model to classify it. In this example, we are just going to use the testing data. We try to classify the pictures using the model. However, in real world use cases this is the stage in which we are going to use our model to make predictions on new data.

Example of prediction using the model

```
#Predict digit(0-9) for test Data
model.predict_classes(X_test)
```

```
 9888/10000 [============================>.] - ETA: 0s
```

```
array([3, 8, 8, ..., 3, 4, 7], dtype=int64)
```

8.Save and Reload the Model

Now that our model is ready it is time to save it so that we can use it later without the need to retrain the model from scratch. Saving the model means saving the weights, biases and all the parameters of the model in a file from which we can load them later into our programs. This process is fairly straight forward in Python and Keras.

Example of saving and reloading a mode

```
#Saving the model
model.save('model.h5')
jsonModel = model.to_json()
model.save_weights('modelWeight.h5')
```

```
#Load weight of the saved model
modelWt = model.load_weights('modelWeight.h5')
```

Final Review

Now that we are familiar with all the steps we need to take to build a model, let's review the procedure using a new example. Try to explain each section of the code to make sure you have learned the concepts profoundly.

```
# A TYPING DEEP LEARNING MODEL WITH KERAS
    import numpy as np
    from keras.models import Sequential
    from keras.layers import Dense

# Loading Data

    data = np.random.random((500,100))
    labels = np.random.randint(2,size=(500,1))


# Create model
        model = Sequential()
        model.add(Dense(12, input_dim=8, activation='relu'))
        model.add(Dense(8, activation='relu'))
       model.add(Dense(1, activation='sigmoid'))

# Compile model

     model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Fit the model

    model.fit(X[train], Y[train], epochs=150, batch_size=10, verbose=0)

# Evaluate the model

    scores = model.evaluate(X[test], Y[test], verbose=0)
   print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
  cvscores.append(scores[1] * 100) print("%.2f%% (+/- %.2f%%)" %
(numpy.mean(cvscores), numpy.std(cvscores)))

# Predict
    predictions = model.predict(data)
```