**Project -1 :: Identifying age group of an actor**

Have you ever wondered about the age group of a movie actor/actress just by looking at their face? Well, if you have but were not exactly able to figure out a way to make an approximately accurate prediction, do not worry, as we will do the same with the help of deep neural networks.

We are going to take a scenario of identifying the age group of various movie characters just by considering their facial attributes and in turn will try to understand the implementation of deep neural networks in python.

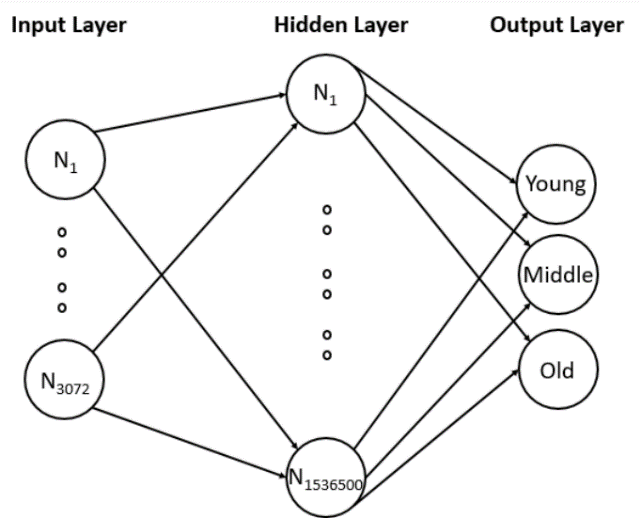
We will use the Indian Movie Face Database (IMFDB)\* created by Shankar Setty et.al. as a benchmark for facial recognition with wide variation. The database consists of thousands of images of 50+ actors taken from more than 100 videos. Since the database has been created manually by cropping the images from the video, there’s high variability in terms of pose, expression, illumination, resolution, etc. The original database provides many attributes including:

* Expressions: Anger, Happiness, Sadness, Surprise, Fear, Disgust
* Illumination: Bad, Medium, High
* Pose: Frontal, Left, Right, Up, Down
* Occlusion: Glasses, Beard, Ornaments, Hair, Hand, None, Others
* Age: Child, Young, Middle and Old
* Makeup: Partial makeup, Over-makeup
* Gender: Male, Female

In this scenario, we will use a cleaned and formatted data set with 26742 images split as 19906 train images and 6636 test images respectively. The target here is to use the images and predict the age of the actor/actress within the available classes i.e. young, middle and old making it a multi-class classification problem.

You can download the data sets from google drive(deep learning datasets). In each directory, you will find a folder consisting of images along with an excel file which has two columns, ID and Class. The ID column consists of image names like **352.jpg** and Class column holds the respective image character’s age like **Old**.

In the data set, images belong to different sizes and hence cannot be fed directly to the input layer. An input layer has a defined number of nodes which are not changed during the process of implementation of the network. Therefore, before proceeding with building the network, we need to ensure that all the images have equal width and height.

For our problem statement, we will resize all the images to 32 x 32 shape. All the images have red, blue and green color components, therefore, the final shape becomes 32 x 32 x 3 giving us a total of 3072 nodes for the input layer.

Next, we will choose one hidden layer to start with along with 500 nodes making a total of 1536500 (3072 x 500) connections between the input and the hidden layer. We will use the ReLU activation function in this layer.

Next, we have the output layer having only three classes and hence three nodes making a total of 1503 (500 x 3) connections between hidden and output layer. In this layer, we will use the Softmax activation function.

While building the model, we will split the training data into training and validation data set and will find the loss and accuracy for both the data sets.

Now that we have defined the structure of our neural network, let us start implementing the code in Keras.

Implementing ANN

Before we proceed, let us take a look at the current challenges of the given data set:

* Variations in shape: For example, one image has a shape of (66, 46) whereas another has a shape of (102, 87), there is no consistency
* Multiple viewpoints/ profiles: faces with different viewpoints/profiles may exist
* Brightness and contrast: It varies across images and can introduce discrepancy in few cases
* Quality: Some images are found to be too pixelated

In this resource, we are going to handle the above challenges by performing image preprocessing, as well as implement a basic neural network.

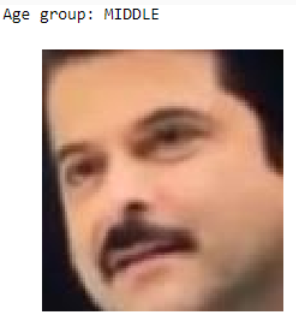
Let us first import all the necessary libraries and modules which will be used throughout the code:

1. *# Importing necessary libraries*
2. import os
3. import numpy as np
4. import pandas as pd
5. import matplotlib.pyplot as plt
6. %matplotlib inline
7. from sklearn.preprocessing import LabelEncoder
8. from tensorflow.python.keras import utils
9. from keras.models import Sequential
10. from keras.layers import Dense, Flatten, InputLayer
11. import keras
12. import imageio *# To read images*
13. from PIL import Image *# For image resizing*

Next, let us read the train and test data sets into separate pandas DataFrames as shown below:

1. *# Reading the data*
2. train = pd.read\_csv('age\_detection\_train/train.csv')
3. test = pd.read\_csv('age\_detection\_test/test.csv')

Once, both the data sets are read successfully, we can display any random movie character along with their age group to verify the ID against the Class value, as shown below:

1. np.random.seed(10)
2. idx = np.random.choice(train.index)
3. img\_name = train.ID[idx]
4. img = imageio.imread(os.path.join('age\_detection\_train/Train', img\_name))
5. print('Age group:', train.Class[idx])
6. plt.imshow(img)
7. plt.axis('off')
8. plt.show()

Next, we can start transforming the data sets to a one-dimensional array after reshaping all the images to a size of 32 x 32 x 3.

Let us reshape and transform the training data first, as shown below:

1. temp = []
2. for img\_name in train.ID:
3. img\_path = os.path.join('age\_detection\_train/Train', img\_name)
4. img = imageio.imread(img\_path)
5. img = np.array(Image.fromarray(img).resize((32, 32))).astype('float32')
6. temp.append(img)
7. train\_x = np.stack(temp)

Next, let us reshape and transform the testing data, as shown below:

1. temp = []
2. for img\_name in test.ID:
3. img\_path = os.path.join('age\_detection\_test/Test', img\_name)
4. img = imageio.imread(img\_path)
5. img = np.array(Image.fromarray(img).resize((32, 32))).astype('float32')
6. temp.append(img)
7. test\_x = np.stack(temp)

Next, let us normalize the values in both the data sets to feed it to the network. To normalize, we can divide each value by 255 as the image values lie in the range of 0-255.

1. *# Normalizing the images*
2. train\_x = train\_x / 255.
3. test\_x = test\_x / 255.

and label encodes the output classes to numerics:

1. *# Encoding the categorical variable to numeric*
2. lb = LabelEncoder()
3. train\_y = lb.fit\_transform(train.Class)
4. train\_y = utils.np\_utils.to\_categorical(train\_y)

Next, let us specify the network parameters to be used, as shown below:

1. *# Specifying all the parameters we will be using in our network*
2. input\_num\_units = (32, 32, 3)
3. hidden\_num\_units = 500
4. output\_num\_units = 3
5. epochs = 5
6. batch\_size = 128

Next, let us define a network with one input layer, one hidden layer, and one output layer, as shown below:

1. model = Sequential([
2. InputLayer(input\_shape=input\_num\_units),
3. Flatten(),
4. Dense(units=hidden\_num\_units, activation='relu'),
5. Dense(units=output\_num\_units, activation='softmax'),
6. ])

We can also use summary() method to visualize the connections between each layer, as shown below:

1. *# Printing model summary*
2. model.summary()

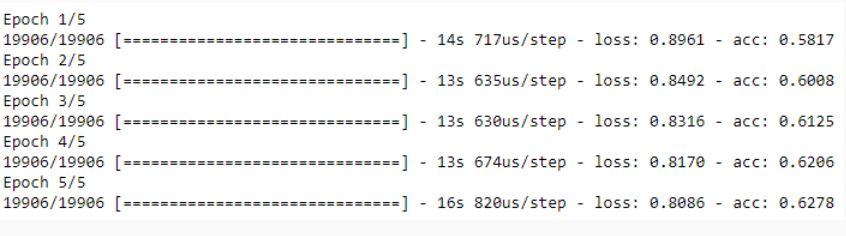
Next, let us compile our network with SGD optimizer and use accuracy as a metric:

1. *# Compiling and Training Network*
2. model.compile(optimizer='sgd', loss='categorical\_crossentropy', metrics=['accuracy'])

Now, let us build the model, using the fit() method:

1. model.fit(train\_x, train\_y, batch\_size=batch\_size, epochs=epochs, verbose=1)

This results in the following log:

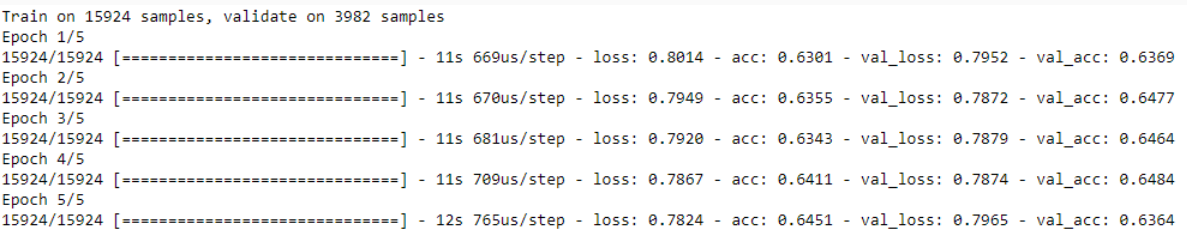


We can observe in the above results, that the final accuracy is 62.78%. However, it is recommended  that we use 20% to 30% of our training data as a validation data set to observe how the model works on unseen data.

The following code considers 20 percent of the training data as validation data set:

1. *# Training model along with validation data*
2. model.fit(train\_x, train\_y, batch\_size=batch\_size, epochs=epochs, verbose=1, validation\_split=0.2)

This results in the following log:



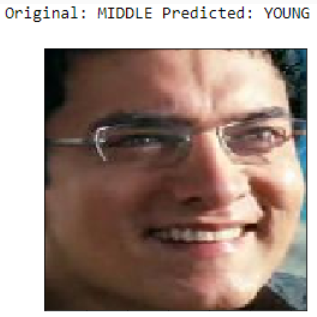
We can observe, that the training accuracy is 64.51% and validation accuracy is 63.64%. Since both the results are quite close we can conclude that there's no overfitting in the model. However, the accuracy itself is too low. The accuracy can be increased by overcoming the previously stated challenges and some difference can even be observed by tuning the hyper-parameters which we are going to observe in the next resource.

With our baseline neural network, we can now predict the age group of test data and save the results in an output file, as shown below:

1. *# Predicting and importing the result in a csv file*
2. pred = model.predict\_classes(test\_x)
3. pred = lb.inverse\_transform(pred)
4. test['Class'] = pred
5. test.to\_csv('out.csv', index=False)

We can also perform the visual inspection on any random image, as shown below:

1. *# Visual Inspection of predictions*
2. idx = 2481
3. img\_name = test.ID[idx]
4. img = imageio.imread(os.path.join('age\_detection\_test/Test', img\_name))
5. plt.imshow(np.array(Image.fromarray(img).resize((128, 128))))
6. pred = model.predict\_classes(test\_x)
7. print('Original:', train.Class[idx], 'Predicted:', lb.inverse\_transform(pred[idx]))

The network misidentified the current image from the middle age group as young. This could be due to the 64% accuracy of the model. Therefore, let us learn about hyper-parameter tuning and try to improve the results.

Hyperparameter Tuning

In the previous resource, we built a basic neural network model without selecting the optimum hyper parameters. In this resource, we will go through through the implementation of finding the best hyper parameters in a given neural network. Following are the hyper parameters which we are going to discuss:

* Optimization algorithms
* Activation function
* Learning rate
* Weights
* Bias
* Number of nodes in a layer
* L0, L1 and L2 regularization
* Number of epochs

For each of these, we will train the model for 100 epochs and provide the results in Tensor board for us to visualize. Once, we have the optimum value from each of these, then we will use them to improve our model to identify a Bollywood character's age group.

**Optimization algorithms:**

Most of the code in the above eight hyper parameters is same initially, which actually deals with loading necessary libraries, reading training data, converting it to a form suitable for the neural network, normalizing it and label encoding the output classes. Therefore, we provide this common code as shown below:

1. *# Necessary libraries*
2. import os
3. import re
4. import numpy as np
5. import pandas as pd
6. from keras.models import Sequential
7. from keras.layers import Dense, Flatten, InputLayer
8. from sklearn.preprocessing import LabelEncoder
9. from tensorflow.python.keras import utils
10. import keras
11. import imageio
12. from PIL import Image
13. *# Reading the data*
14. train = pd.read\_csv('age\_detection\_train/train.csv')
15. *# Image resizing of train data into single numpy array*
16. temp = []
17. for img\_name in train.ID:
18. img\_path = os.path.join('age\_detection\_train/Train', img\_name)
19. img = imageio.imread(img\_path)
20. img = np.array(Image.fromarray(img).resize((32, 32))).astype('float32')
21. temp.append(img)
22. train\_x = np.stack(temp)
23. *# Normalizing the images*
24. train\_x = train\_x / 255.
25. *# Encoding the categorical variable to numeric*
26. lb = LabelEncoder()
27. train\_y = lb.fit\_transform(train.Class)
28. train\_y = utils.np\_utils.to\_categorical(train\_y)
29. *# Specifying all the parameters we will be using in our network*
30. input\_num\_units = (32, 32, 3)
31. hidden\_num\_units = 500
32. output\_num\_units = 3
33. epochs = 100
34. batch\_size = 128

We are going to focus on the following five optimization algorithms:

* SGD
* Adagrad
* Adadelta
* RMSprop
* Adam

First, let us define the network, as shown below:

1. *# Defining the network*
2. model = Sequential([
3. InputLayer(input\_shape=input\_num\_units),
4. Flatten(),
5. Dense(units=hidden\_num\_units, activation='relu'),
6. Dense(units=output\_num\_units, activation='softmax'),
7. ])

Next, let us write a function to build a model for each optimizer and save the accuracy, loss, validation accuracy and validation loss for Tensor board visualization.

1. def models\_with\_different\_optimizers(list\_of\_optimizers):
3. for i in range(len(list\_of\_optimizers)):
4. model.compile(loss='categorical\_crossentropy',
5. optimizer=list\_of\_optimizers[i], *# Learning rate and momentum can be passed inside optimizer*
6. metrics=['accuracy'])
7. *# Traning the model and writing log files for TensorBoard in distinct directories*
8. val = re.search('optimizers\..\*\so', str(list\_of\_optimizers[i])).group(0)[11:][:-2] *# Fetching optimizer name*
9. logdir = r'optims\\' + val *# Each log file needs to be written in a distinct directory. (Mandatory)*
11. *# Writing graph will take time. Hence, keeping it False.*
12. cb = keras.callbacks.TensorBoard(log\_dir=logdir, write\_graph=False)
13. print('Building model using '+ val + ' optimizer')
14. history = model.fit(train\_x, train\_y, epochs=epochs,
15. validation\_split=0.2,
16. callbacks=[cb])
17. print('Model built sucessfully.')
18. print('')
19. *# Listing the optimizers*
20. optims = [keras.optimizers.Adam(), keras.optimizers.Adadelta(),
21. keras.optimizers.Adagrad(), keras.optimizers.RMSprop(),
22. keras.optimizers.SGD()]
23. *# Calling the function*
24. models\_with\_different\_optimizers(optims)

This will result in a folder named “**optims**” which consists of the evaluation metrics. We can visualize the output by running the following command in Anaconda prompt:

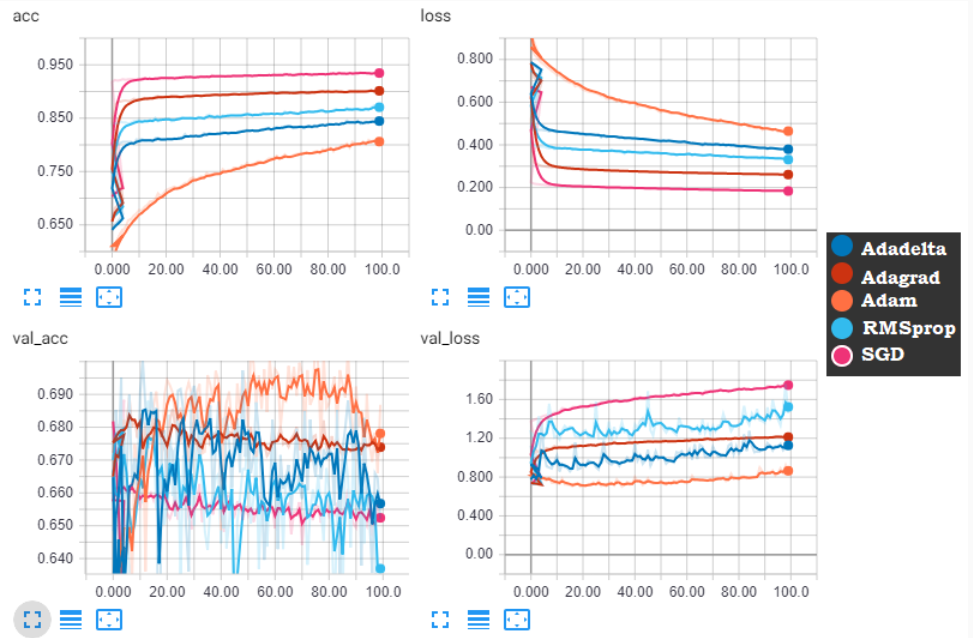
1. cd “go to the directory which holds optims directory”
2. tensorboard --logdir=optims

In case the above command doesn’t result in Tensor board, we can run the following command instead, to manually set the port:

1. tensorboard --logdir=optims/ --host localhost --port 8088

Use **.\*** under Tensorboard to visualize all four graphs in the first section. You can also change the Horizontal Axis from Step to Relative which describes the relative time taken by each algorithm to finish.

The result will be similar to the one given below:



In Tensorboard, when you hover over any of these images, you can see the value of each algorithm at a particular epoch. From the above figure, we can infer easily that as compared to others, Adam has high validation accuracy and low validation loss, however, with increasing epoch each of these algorithm tends to overfit which is an issue to handled by regularization and early stopping, to be discussed ahead.

Optims files are placed in Google drive with datasets

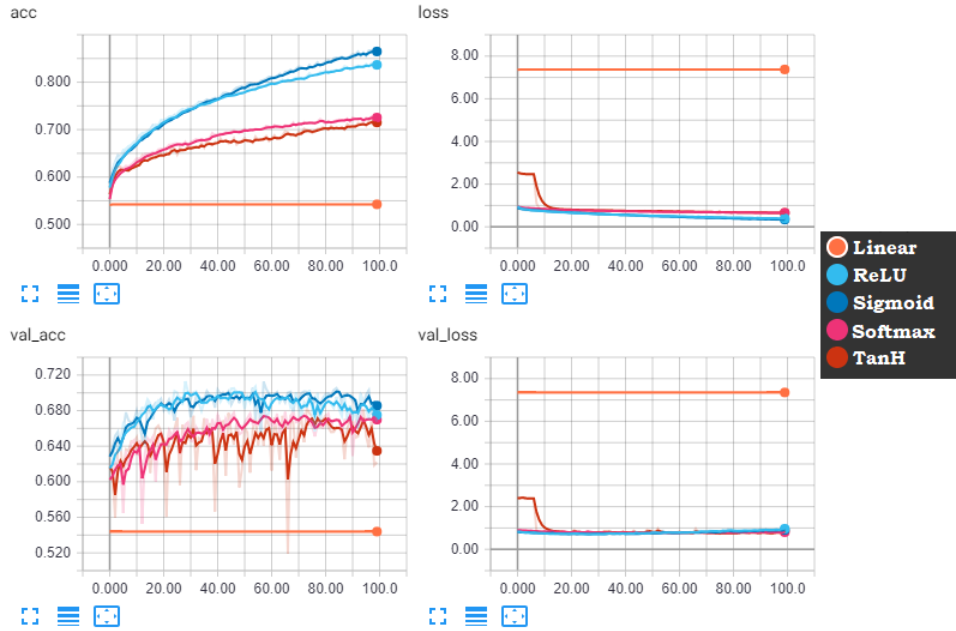
**Activation Functions:**

In this section, we are choosing '**linear**', '**sigmoid**', '**tanh**', '**relu**', and '**softmax**' as the competing activation functions for the hidden layer. The code to build a model with each of these activation functions and saving the evaluation metrics which results in a directory named ‘**activation**’, is as shown below:

1. def models\_with\_different\_activation\_fn(list\_of\_activation\_fn):
3. for i in range(len(list\_of\_activation\_fn)):
4. *# Defining the network*
5. model = Sequential([
6. InputLayer(input\_shape=input\_num\_units),
7. Flatten(),
8. Dense(units=hidden\_num\_units, activation=list\_of\_activation\_fn[i]),
9. Dense(units=output\_num\_units, activation='softmax'),
10. ])
11. model.compile(loss='categorical\_crossentropy',
12. optimizer=keras.optimizers.Adam(),
13. metrics=['accuracy'])
14. *# Traning the model and writing log files for TensorBoard in distinct directories*
15. logdir = r'activation\\' + list\_of\_activation\_fn[i] *# Each log file needs to be written in a distinct directory. (Mandatory)*
17. *# Writing graph will take time. Hence, keeping it False.*
18. cb = keras.callbacks.TensorBoard(log\_dir=logdir, write\_graph=False)
19. print('Building model using '+ list\_of\_activation\_fn[i] + ' activation function')
21. history = model.fit(train\_x, train\_y, epochs=epochs,
22. validation\_split=0.2,
23. callbacks=[cb])
24. print('Model built sucessfully.')
25. print('')
26. *# List of activation functions*
27. act = ['linear', 'sigmoid', 'tanh', 'relu', 'softmax']
28. *# Calling the function*
29. models\_with\_different\_activation\_fn(act)

This results in the below image after running the following command:

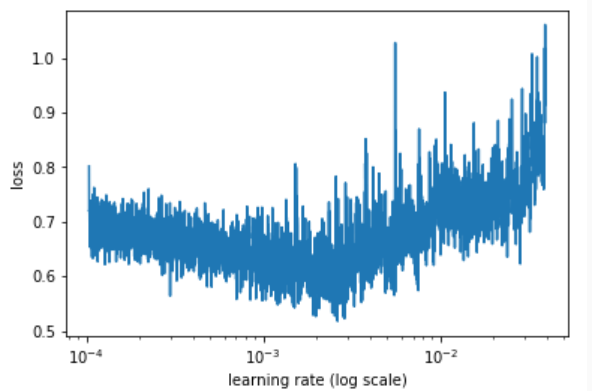
1. tensorboard --logdir=activation/ --host localhost --port 8088



We can observe that ReLU and sigmoid are always the top contenders with sigmoid having the least loss both in training and validation as compared to ReLU. So, for the final model, you can use any of these two.

**Learning Rate:**

To find the optimum learning rate, we have to use a new library, **keras\_lr\_finder**, and a python script [clr\_callback.py](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012776456297463808249/web-hosted/assets/clrcallback.py). First, we need to find the range of optimum learning rate which is extracted using the **keras\_lr\_finder** library by plotting the learning rate against the loss. The range where the learning rate decreases quite rapidly is chosen, as shown below:

1. *# For finding optimum learning rate [pip install keras\_lr\_finder]*
2. import keras\_lr\_finder as lr\_find
3. lr\_finder = lr\_find.LRFinder(model)
4. *# Training can stop abruptly if rate is set too high. In such cases, reduce the value of end\_lr.*
5. lr\_finder.find(train\_x, train\_y, start\_lr=0.0001, end\_lr=0.09, batch\_size=batch\_size, epochs=epochs)
6. *# Plot the loss, ignore 20 batches in the beginning and 5 in the end*
7. lr\_finder.plot\_loss(n\_skip\_beginning=20, n\_skip\_end=5)

We can observe from the above figure that the loss decreases between 1E-4 and 1E-3. So, we consider this range and feed it to **clr\_callback.py** script, this let the learning rate vary between only this range and doesn't let the gradient diminish during the model training by applying a particular pattern like triangular, sinusoidal, etc. Here, we apply the triangular wave on the given range of learning rate as shown below:

1. *# Importing cyclic learning rate module*
2. from clr\_callback import \*
3. *# Pass the values of base\_lr with the value of*
4. cb\_triangular = CyclicLR(base\_lr=0.0001, max\_lr=0.001, step\_size=2000., mode='triangular2') *# Setting callback for model*
5. *# Writing graph will take time. Hence, keeping it False.*
6. cb\_save = keras.callbacks.TensorBoard(log\_dir='learning\_rate', write\_graph=False)
7. model.fit(train\_x, train\_y, batch\_size=batch\_size, epochs=epochs,
8. validation\_split=0.2, callbacks=[cb\_triangular, cb\_save], verbose=1)
9. print('Model built sucessfully.')
10. plt.xlabel('Training Iterations')
11. plt.ylabel('Learning Rate')
12. plt.title("CLR - 'triangular2' Policy")
13. plt.plot(cb\_triangular.history['iterations'], cb\_triangular.history['lr'])
14. *# With more number of iterations the graph will correspond to that of a triangular wave*

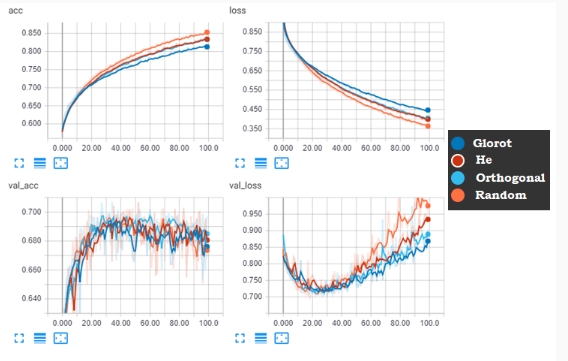


**Weights:**

Here, we are going to use four different variations in weights, '**Random**', '**Glorot**', '**He**', and '**Orthogonal**' and compare their performance:

1. def weight\_initializer(list\_of\_weights, init\_weight\_names):
2. for weight\_init in range(len(list\_of\_weights)):
3. *# Writing structure of model*
4. model = Sequential([
5. InputLayer(input\_shape=input\_num\_units),
6. Flatten(),
7. Dense(units=hidden\_num\_units, kernel\_initializer=list\_of\_weights[weight\_init], activation='relu'),
8. Dense(units=output\_num\_units, kernel\_initializer=list\_of\_weights[weight\_init], activation='softmax'),
9. ])
10. *# Defining parameters like optmizer, loss function and evaluating metric*
11. model.compile(loss='categorical\_crossentropy', *#*
12. optimizer=keras.optimizers.Adam(), *# Learning rate and momentum can be passed inside optimizer*
13. metrics=['accuracy'])
14. *# Traning the model and writing log file for TensorBoard*
16. logdir = r'weights\\' + init\_weight\_names[weight\_init] *# Each log file needs to be written in a distinct directory. (Mandatory)*
17. *# To store tensorboard graph logs*
18. cb = keras.callbacks.TensorBoard(log\_dir=logdir, write\_graph=False)
20. print('Model building using ' + init\_weight\_names[weight\_init] + ' Initializer')
21. history = model.fit(train\_x, train\_y, epochs=epochs,
22. validation\_split=0.2,
23. callbacks=[cb])
24. print('Built model successfully.')
25. print('')
26. *# Listing the weights*
27. list\_of\_weights = [keras.initializers.RandomNormal(), keras.initializers.glorot\_normal(),
28. keras.initializers.he\_normal(), keras.initializers.Orthogonal()]
29. init\_weight\_names = ['Random', 'Glorot', 'He', 'Orthogonal']
30. *# Calling the function*
31. weight\_initializer(list\_of\_weights, init\_weight\_names)

The above code saves the result in “**weights**” directory and the results can be visualized using the following code:

1. tensorboard --logdir=weights/ --host localhost --port 8088
2. We can't set the weights to zero so we need to follow any of these weight initializing technique. In the image above, the model clearly starts to overfit after the epoch range of 27-33. During this phase, the weights provided by “**He**” shows great improvement when compared to the rest.
3. Note that your graph might be different than the one shown above due to the model picking up random values for weights.

**Bias:**

Bias can be set to zero if weights are initialized as non-zero or if weights are small enough to break the symmetry. However, biases for a ReLU non-linearities can be initialized with a value more than 0 like 0.01 or 0.1 to avoid large saturation during initialization. Here, we will build the model for a bias value of **0**, **0.01**, and **0.1**,as shown below:

1. def bias\_initializer(list\_of\_biases, bias\_names):
2. for bias\_init in range(len(list\_of\_biases)):
3. *# Writing structure of model*
4. model = Sequential([
5. InputLayer(input\_shape=input\_num\_units),
6. Flatten(),
7. Dense(units=hidden\_num\_units, bias\_initializer=list\_of\_biases[bias\_init], activation='relu'),
8. Dense(units=output\_num\_units, activation='softmax'),
9. ])
10. *# Defining parameters like optmizer, loss function and evaluating metric*
11. model.compile(loss='categorical\_crossentropy', *#*
12. optimizer=keras.optimizers.Adam(), *# Learning rate and momentum can be passed inside optimizer*
13. metrics=['accuracy'])
14. *# Traning the model and writing log file for TensorBoard*
16. logdir = r'biases\\' + bias\_names[bias\_init] *# Each log file needs to be written in a distinct directory. (Mandatory)*
17. *# To store tensorboard graph logs*
18. cb = keras.callbacks.TensorBoard(log\_dir=logdir, write\_graph=False)
20. print('Model building using ' + bias\_names[bias\_init] + ' Bias Initializer')
21. history = model.fit(train\_x, train\_y, epochs=epochs,
22. validation\_split=0.2,
23. callbacks=[cb])
24. print('Model built successfully.')
25. print('')
26. *# Listing bias values*
27. list\_of\_biases = [keras.initializers.Zeros(), keras.initializers.Constant(value=0.01), keras.initializers.Constant(value=0.1)]
28. bias\_names = ['0', '0.01', '0.1']
29. *# Calling the function*
30. bias\_initializer(list\_of\_biases, bias\_names)

The results are stored in the ‘**biases**’ directory which can be fetched to the Tensorboard using the below command:

1. tensorboard --logdir=biases/ --host localhost --port 8088

**Number of nodes in a layer**:

Deciding on the number of nodes in a layer is a challenging task. We know the exact number of output layer nodes but the question remains open for all the hidden layers nodes to be kept in model building.

You can either start with say three hidden layers, the first layer having x nodes, the second layer having 2x nodes and the third layer having 3x nodes, or we can either follow a procedure where some percentage of a random number of nodes are dropped and the model is built with the rest.

Dropout regularization is one such procedure. We can decide on the percentage of nodes to be dropped randomly and then the model is built with the remaining. This helps to decide an optimum value for the number of nodes in a layer.

1. from keras.layers import Dropout
2. model = Sequential([
3. InputLayer(input\_shape=input\_num\_units),
4. Flatten(),
5. *# Dropout layer can also be added between input layer and first hidden layer.*
6. *# keras.layers.Dropout(0.30),*
8. *# According to dropout paper, the maximum norm of the weights should not exceed a value of 3.*
9. *# Hence, defining constraint maxnorm(3).*
10. Dense(units=hidden\_num\_units, activation='relu', kernel\_constraint=keras.constraints.maxnorm(3)),
11. Dropout(0.30), *# Dropout layer with 30% of input units of last layer to drop*
12. Dense(units=output\_num\_units, activation='softmax'),
13. ])
14. *# Defining parameters like optmizer, loss function and evaluating metric*
15. model.compile(loss='categorical\_crossentropy', *#*
16. optimizer=keras.optimizers.Adam(), *# Learning rate and momentum can be passed inside optimizer*
17. metrics=['accuracy'])
18. cb = keras.callbacks.TensorBoard(log\_dir='dropout', write\_graph=False)
20. history = model.fit(train\_x, train\_y, epochs=epochs, validation\_split=0.2, callbacks=[cb])

The above code results in the ‘**dropout**’ directory which consists of the evaluation metric to be visualized in Tensor board, as shown below:

1. tensorboard --logdir=dropout/ --host localhost --port 8088



With the **30%** dropout regularization on the hidden layer, we have a minute loss in the accuracy which is acceptable.

**Ridge, Lasso and Elastic-Net Regularization:**

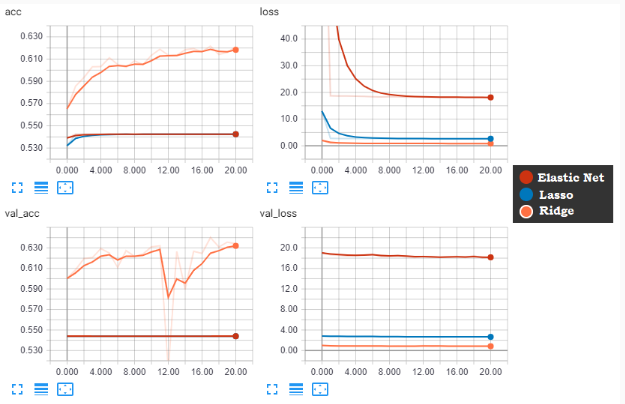
To tackle the overfitting problems, we can use either of the regularization techniques **Ridge**, **Lasso** or **Elastic-Net**. While building the model, lasso regularization can completely remove few of the features, ridge regularization can decrease the effects of few features but will not remove them completely, whereas elastic-net works by combining both of these regularization techniques.

Following is the implementation of all the three techniques:

1. epochs = 21
2. def regularization(list\_of\_regs, reg\_names):
3. for reg\_init in range(len(list\_of\_regs)):
4. *# Writing structure of model*
5. model = Sequential([
6. InputLayer(input\_shape=input\_num\_units),
7. Flatten(),
8. Dense(units=hidden\_num\_units, kernel\_regularizer=list\_of\_regs[reg\_init], activation='relu'),
9. Dense(units=output\_num\_units, activation='softmax'),
10. ])
11. *# Defining parameters like optmizer, loss function and evaluating metric*
12. model.compile(loss='categorical\_crossentropy', *#*
13. optimizer=keras.optimizers.Adam(), *# Learning rate and momentum can be passed inside optimizer*
14. metrics=['accuracy'])
15. *# Traning the model and writing log file for TensorBoard*
17. logdir = r'regularization\\' + reg\_names[reg\_init] *# Each log file needs to be written in a distinct directory. (Mandatory)*
18. *# To store tensorboard graph logs*
19. cb = keras.callbacks.TensorBoard(log\_dir=logdir, write\_graph=False)
21. print('Model building using ' + reg\_names[reg\_init] + ' Regularization')
22. history = model.fit(train\_x, train\_y, epochs=epochs,
23. validation\_split=0.2,
24. callbacks=[cb])
25. print('Model built successfully.')
26. print('')
27. *# Value of lambda can be calculated using cross-validation*
28. list\_of\_regs = [keras.regularizers.l2(), keras.regularizers.l1(), keras.regularizers.l1\_l2()]
29. reg\_names = ['Ridge.L2', 'Lasso.L1', 'Elastic.Net.L1.L2']
30. regularization(list\_of\_regs, reg\_names)

The above code produces the ‘**regularization**’ directory. We can use the command given below to open the metrics in the Tensorboard:

1. tensorboard --logdir=regularization/ --host localhost --port 8088



We can clearly infer from the above figure that **Ridge**regularization is better when compared to the other two. However, if we compare the current result with previous models, then we can observe that Ridge itself has decreased the accuracy as well as increased the loss. So, when building the model we will consider Ridge just to keep overfitting aside, with a trade-off of less accuracy.

**Number of epochs:**

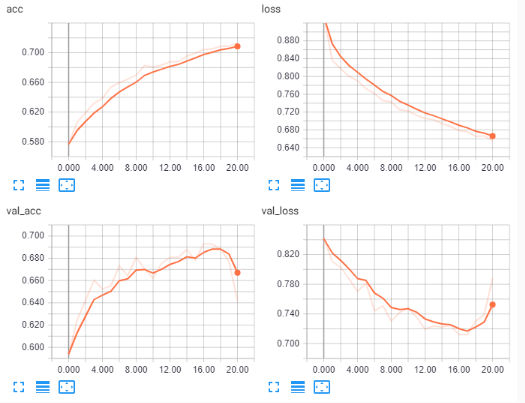
To decide on the number of epochs, we can follow the early stopping principle. With an increasing number of epochs, it can be seen that a consistently decreasing validation set error can rise again. In such cases, training needs to be stopped at the point in time with the lowest validation set error. A hyper-parameter named as patience is provided in early stopping which determines a threshold to the number of epochs after which if significant improvement is not seen in the monitored metric then training needs to be stopped.

Here, we will set patience equal to 8 epochs.

1. model = Sequential([
2. InputLayer(input\_shape=input\_num\_units),
3. Flatten(),
4. Dense(units=hidden\_num\_units, activation='relu'),
5. Dense(units=output\_num\_units, activation='softmax'),
6. ])
7. *# Defining parameters like optmizer, loss function and evaluating metric*
8. model.compile(loss='categorical\_crossentropy', *#*
9. optimizer=keras.optimizers.Adam(), *# Learning rate and momentum can be passed inside optimizer*
10. metrics=['accuracy'])
11. *# Defining early stopping callback.*
12. *# After epoch 50, training can be stopped if not improved in 'val\_loss' is seen*
13. cb\_early\_stop = keras.callbacks.EarlyStopping(monitor='val\_loss', patience=8, min\_delta=0.02)
14. cb = keras.callbacks.TensorBoard(log\_dir='early\_stop', write\_graph=False)
15. history = model.fit(train\_x, train\_y, epochs=epochs, validation\_split=0.2, callbacks=[cb, cb\_early\_stop])

The above code creates ‘**early\_stop**’ directory. We can use the below code to visualize metrics in Tensor board, as shown below:

1. tensorboard --logdir=early\_stop/ --host localhost --port 8088



The model building stops at the **21th** epoch when the model doesn’t see any certain improvement in validation loss.

**Note:** While finding the best hyperparameter values we have not focused on tuning each algorithm's parameters. Currently, we followed their default values. Although it is advised to take care of each parameter involved in each algorithm when working on an actual project.

**Building a optimal network:** So far, we have arrived at the following optimum values:

| **Hyperparameter** | **Values** |
| --- | --- |
| Optimization algorithm | Adam |
| Activation function | ReLU |
| Learning rate | 1E-4 to 1E-3 |
| Weight | He |
| Bias | 0.01 |
| Number of nodes in a layer | Dropout regularization at 30% |
| Regularization | Ridge |
| Number of epochs | 21 |

Let us feed them to the model building process and get the results.

1. *# Specifying all the parameters we will be using in our network*
2. input\_num\_units = (32, 32, 3)
3. hidden\_num\_units = 500
4. output\_num\_units = 3
5. *# Optimum values*
6. optimizer = keras.optimizers.Adam()
7. activation = 'relu'
8. cb\_triangular\_lr = CyclicLR(base\_lr=0.0001, max\_lr=0.001, step\_size=2000., mode='triangular2')
9. weights = keras.initializers.he\_normal()
10. bias = keras.initializers.Constant(value=0.01)
11. dropout = 0.30
12. regularizer = keras.regularizers.l2()
13. epochs = 21
14. batch\_size = 128

Next, let us define the network:

1. *# Defining the network*
2. model = Sequential([
3. InputLayer(input\_shape=input\_num\_units),
4. Flatten(),
5. Dense(units=hidden\_num\_units, kernel\_initializer=weights,
6. bias\_initializer=bias, activation=activation,
7. kernel\_constraint=keras.constraints.maxnorm(3),
8. kernel\_regularizer=regularizer),
9. Dropout(dropout),
10. Dense(units=output\_num\_units, kernel\_initializer=weights, activation='softmax'),
11. ])

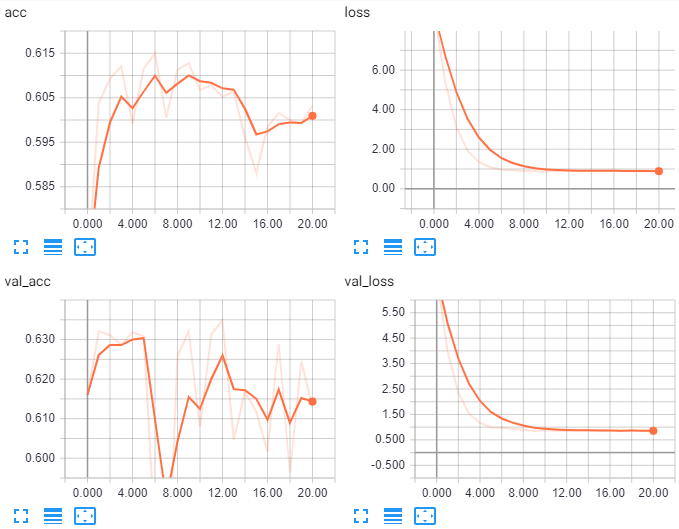
Let us compile and train the network:

1. *# Compiling and Training Network*
2. model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])
3. cb\_save = keras.callbacks.TensorBoard(log\_dir='optimum\_model', write\_graph=False)
4. model.fit(train\_x, train\_y, batch\_size=batch\_size, epochs=epochs, verbose=1,
5. validation\_split=0.2, callbacks = [cb\_triangular\_lr, cb\_save])

The model stops at the 21st epoch. The final validation accuracy of the model is **61.3%** and the training accuracy is 60.33% which is quite closer to the validation accuracy. Though the current validation accuracy is slightly less than the baseline model which was **64.51%**but the current model has less overfitting and thus more robust to the previous one.

The code below starts the visualization in Tensorboard:

1. tensorboard --logdir=optimum\_model/ --host localhost --port 8088

Since the data set itself needs a heavy image preprocessing before feeding into the network, therefore, we can’t expect a high accuracy just by selecting optimum hyperparameters. However, by tuning each algorithm on their parameters, an increase in the current result can be expected.

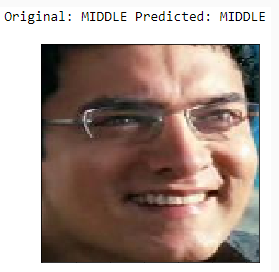
You can save the model to an H5 format file and load it as shown below:

1. *# Saving the model*
2. model.save('optimum\_model.h5')
3. *# Loading the model*
4. from keras.models import load\_model
5. model = load\_model('optimum\_model.h5')

# Final prediction

Let us take the previous example wherein we used the normal model to predict the age group of an image with index **2481**. Last time, it incorrectly predicted the age group as "Young" or in other words misclassified the movie character. Let us see what the prediction is with the improved model.

1. *# Visual Inspection of predictions*
2. idx = 2481
3. img\_name = test.ID[idx]
4. img = imageio.imread(os.path.join('age\_detection\_test/Test', img\_name))
5. plt.imshow(np.array(Image.fromarray(img).resize((128, 128))))
6. pred = model.predict\_classes(test\_x)
7. print('Original:', train.Class[idx], 'Predicted:', lb.inverse\_transform(pred)[idx])



We can observe that this time the prediction is the "Middle" age group is right on the money!

Predicting the house price - Try Out

Predicting the house price is one of the main and commonly known regression problems which can also be tackled by neural networks. In this try-out, you will learn two concepts:

* Handling categorical data to feed in neural network
* Building a regression neural network

The [dataset](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012776490282680320258/web-hosted/assets/train.csv) for house price prediction has been taken from Kaggle which consists of 80 features and 1400+ rows.

1. *# Importing necessary libraries*
2. import numpy as np
3. import pandas as pd
4. from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
5. from keras.models import Sequential
6. from keras.layers import Dense, Flatten, InputLayer
7. import keras
8. *# Reading the data*
9. train = pd.read\_csv('train.csv')
10. trainX, trainY = train.iloc[:, :train.shape[1]-1], train.iloc[:, train.shape[1]-1]
11. *# There are a total of 43 categorical columns*
12. categoricals = trainX.loc[:, trainX.dtypes == 'O'].columns
13. len(categoricals) *# 43*
14. *# Preprocessing step:*
15. *# One Hot Encoder cannot work with NaN, hence filling NaN with mode of categorical columns*
16. cat\_features = trainX.loc[:, categoricals]
17. cat\_features = cat\_features.fillna(cat\_features.mode().iloc[0, :])
18. *# One hot encoding these features*
19. ohe = OneHotEncoder(handle\_unknown='ignore')
20. res = ohe.fit\_transform(cat\_features).toarray()
21. cols = np.array([])
22. for i in range(cat\_features.shape[1]):
23. cols = np.concatenate((cols, categoricals[i] + '\_' + np.sort(cat\_features.iloc[:, i].unique())))
24. cat = pd.DataFrame(res, columns=cols)
25. *# Total 252 categorical features*
26. cat.shape *# (1460, 252)*
27. *# Dropping original categorical variables*
28. trainX = trainX.drop(categoricals, axis=1)
29. *# Concatenating the One Hot Encoded variables to the train dataset*
30. trainX = pd.concat([trainX, cat], axis=1)
31. *# New data shape*
32. trainX.shape *# (1460, 289)*
33. *# Filling the NaN with median*
34. trainX.fillna(trainX.median(), inplace=True)
35. *# Normalizing training features*
36. scalar = MinMaxScaler()
37. norm\_train = pd.DataFrame(scalar.fit\_transform(trainX), columns=trainX.columns)
38. *# Normalizing training target*
39. scalar\_target = MinMaxScaler()
40. trainY = scalar\_target.fit\_transform(trainY.values.reshape(-1, 1))
41. *# Defining the network*
42. model = Sequential([
43. Dense(norm\_train.shape[1], input\_dim=norm\_train.shape[1], activation='sigmoid'),
44. Dense(units=norm\_train.shape[1]*//2, activation='sigmoid'),*
45. Dense(units=1, activation='softmax'),
46. ])
47. *# Printing model summary*
48. model.summary()
49. *# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*
50. *# Layer (type) Output Shape Param #*
51. *# =================================================================*
52. *# dense\_34 (Dense) (None, 289) 83810*
53. *# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*
54. *# dense\_35 (Dense) (None, 144) 41760*
55. *# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*
56. *# dense\_36 (Dense) (None, 1) 145*
57. *# =================================================================*
58. *# Total params: 125,715*
59. *# Trainable params: 125,715*
60. *# Non-trainable params: 0*
61. *# Compiling and Training Network*
62. model.compile(optimizer='sgd', loss='mean\_squared\_error')
63. model.fit(trainX, trainY, batch\_size=512, epochs=20, verbose=1, validation\_split=0.2)

The baseline model results in a constant train and validation loss of 0.6483 and 0.6456. The results can further be improved by tuning the hyper-parameters and doing proper preprocessing.

# Your task

You are required to build a new model with five hidden layers along with following optimum hyperparameters:

* activation function, number of nodes in a layer, and epochs.