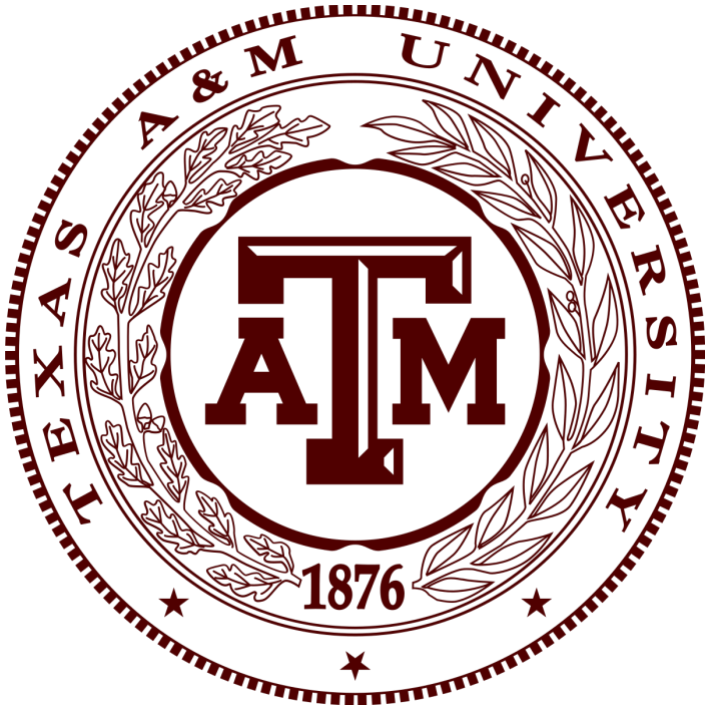


Qusai Amer

Neural Networks

CSCE 633

Homework #3

10/15/2020

COMPUTER SCIENCE & ENGINEERING
TEXAS A&M UNIVERSITY

Background

The following report goes over an application of the artificial neural network using Keras. Neural network is a cutting-edge learning method which can be used both for supervised and unsupervised learning. The building block of neural network is the neuron which uses an activation function to process the input to the output. This activation function, or transfer function, calculates the weighted sum of the input and using a threshold, it decides what is going to be the output. The three main components of neural network include, input layer, hidden layer, and output layer. The input layer, also called visible layer, is exposed to the input while the hidden layer is not. Having many hidden layers is known as deep learning. The output layer can be a vector or a value depending on the problem.

In this particular application, a neural network model is developed classify face images by their expression. The input data has 28709 samples, where each is a grayscale image with 48 × 48 pixels. The objective is to classify each face image based on the facial expression using feedforward neural network (FNN) and convolutional neural network (CNN). This model will be developed using Keras which is an interface for TensorFlow library which makes the use of (ANN) straightforward and in few lines of code.

Visualization

First the data is loaded using pandas into training, validation, and testing sets. All python packages needed are also imported such as matplotlib, numpy, and keras.

```
# Import required packages and read data into dataframe via pandas

import pandas as pd
import matplotlib.cm as cm
import warnings
import matplotlib.pyplot as plt
import numpy as np
warnings.filterwarnings("ignore")
from keras.datasets import mnist
from keras.utils import to_categorical

dataTrain = pd.read_csv('Q2_Train_Data.csv')
dataVal = pd.read_csv('Q2_Validation_Data.csv')
dataTest = pd.read_csv('Q2_Test_Data.csv')
```

Current view of the data:

```
dataTrain.head(3)
```

	emotion	pixels
0	0	70 80 82 72 58 58 60 63 54 58 60 48 89 115 121...
1	0	151 150 147 155 148 133 111 140 170 174 182 15...
2	2	231 212 156 164 174 138 161 173 182 200 106 38...

Then for each set the pixels column and the labels, emotion, columns are separated.

```
# Get the data frames of the images(pixels) and labels

#Train
imgTrain = dataTrain['pixels']
labelTrain = dataTrain['emotion']

#Validation
imgVal = dataVal['pixels']
labelVal = dataVal['emotion']

#Test
imgTest = dataTest['pixels']
labelTest = dataTest['emotion']
```

After that the pixels column which has 2304 space-separated pixel values is reshaped to 48 × 48 matrix using NumPy.

```
#Train
list1 = []
for i in range(0,len(imgTrain)):
    y = np.fromstring(imgTrain[i],dtype=int, sep=" ")
    z = np.reshape(y, (48, 48))
    list1 += [z]
imgs_train = np.array(list1)
labels_train = np.array(labelTrain)
```

Size of the training images matrix

```
imgs_train.shape
```

```
(28709, 48, 48)
```

Note: this is also done to the validation and testing sets.

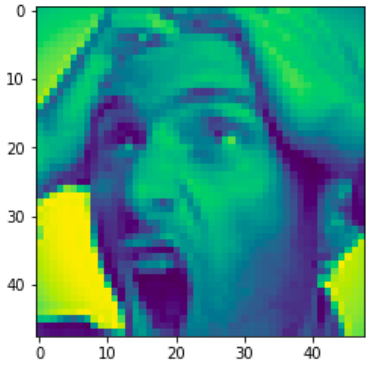
(a) To randomly select and visualize one image per emotion. I followed the following steps:

First index of each emotion is obtained using pandas and stored in a list:

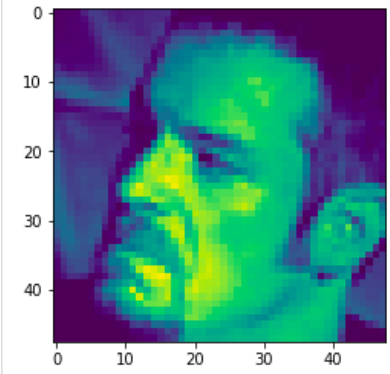
```
# Get index of each emotion

Angry = dataTrain.index[dataTrain['emotion'] == 0].tolist()
Disgust = dataTrain.index[dataTrain['emotion'] == 1].tolist()
Fear = dataTrain.index[dataTrain['emotion'] == 2].tolist()
Happy = dataTrain.index[dataTrain['emotion'] == 3].tolist()
Sad = dataTrain.index[dataTrain['emotion'] == 4].tolist()
Surprise = dataTrain.index[dataTrain['emotion'] == 5].tolist()
Neutral = dataTrain.index[dataTrain['emotion'] == 6].tolist()
```

Then a random index is created which it shuffles the list of each emotion. This random index is used to show the image of each emotion.

```
#Angry
randAI = np.random.permutation(Angry)
plt.imshow(imgs_train[randAI[0]])
plt.show()
```



```
#Angry
randAI = np.random.permutation(Angry)
plt.imshow(imgs_train[randAI[0]])
plt.show()
```



Visualization of all remaining emotions is found in the annex section below.

(b) Data exploration: Count the number of samples per emotion in the training

```
print(f"There are {len(Angry)} samples in the Angry expression")
print(f"There are {len(Disgust)} samples in the Disgust expression")
print(f"There are {len(Fear)} samples in the Fear expression")
print(f"There are {len(Happy)} samples in the Happy expression")
print(f"There are {len(Sad)} samples in the Sad expression")
print(f"There are {len(Surprise)} samples in the Surprise expression")
print(f"There are {len(Neutral)} samples in the Neutral expression")
```

```
There are 3995 samples in the Angry expression
There are 436 samples in the Disgust expression
There are 4097 samples in the Fear expression
There are 7215 samples in the Happy expression
There are 4830 samples in the Sad expression
There are 3171 samples in the Surprise expression
There are 4965 samples in the Neutral expression
```

Forward Neural Network (FNN)

The FNN are also known as multilayer perceptron is a simpler form of the neural network. Mainly because it only moves one direction from the input to the output with no loops, or cycles. As mentioned above the Keras tool is used to define the model as well as train, compile, and evaluate. The model used is Sequential which means that the layers are represented linearly, in linear stacks. Then the class Dense which mean fully connected layer is used to define each layer . Note that the first layer is the input layer and its number of neurons usually equal the number of input data which in this application would be 2307.

(ci) The remaining characteristics of the model are considered hyperparameters which must be tunned using the validation set. Then the best model is used in the test set. Hyperparameter tunning is searching the hyperparameter space for optimal values. For now, random search is used to find the optimal hyperparameters and latter model based method will be used. The different combinations used include the following:

Sample model for FNN:

```
import warnings
warnings.filterwarnings("ignore")  # Ignore some warning logs


from keras.models import Sequential
from keras.layers import Dense, Dropout


#  Define a Feed-Forward Model with 2 hidden layers with dimensions nodes1 and nodes2 Neurons
model = Sequential([
  Dense(2304//2, activation='relu', input_shape=(48*48,), name="first_hidden_layer"),
  Dense(7, activation='softmax'), #converts output to probabilities
])

#  Validate your Model Architecture
print(model.summary())


#Dense(2304//8, activation='relu', name="third_hidden_layer"), Dropout(0.3),
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
first_hidden_layer (Dense)   (None, 1152)              2655360
_________________________________________________________________
dense (Dense)                (None, 7)                 8071
=================================================================
Total params: 2,663,431
Trainable params: 2,663,431
Non-trainable params: 0
_________________________________________________________________
None
```

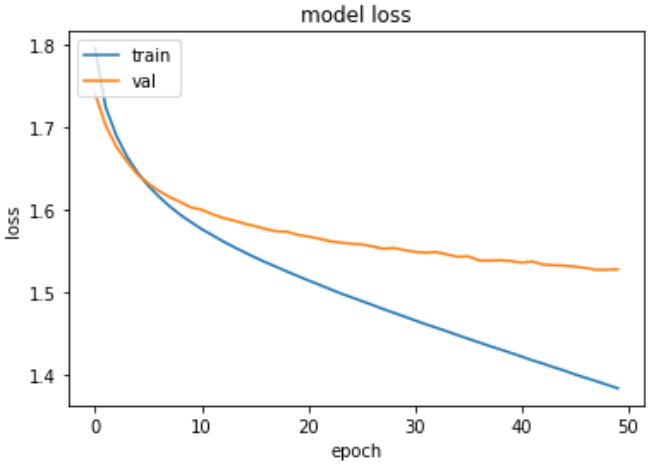
Combination One

Hyperparameter	**Value**
Number of hidden layer	One
Number of nodes	1st layer: 1152
Dropout	-
Epochs	50
Batch size	200
Activation function	Rule
Optimizer	SGD

Results

Average accuracy on training set	42%
Accuracy on validation set	41%
Running time	34 seconds

Cross-entropy loss vs iterations



Reasoning

For combination one, one hidden layer is used with a node number, units, same as half the input samples. The running time of the model is relatively long with 34 second; mainly because the number of epochs is slightly on the high end. It also can be seen in the graph that after a certain epoch the validation set loss don't decrease as much. For the next combination, another layer will be added with 25% dropout to increase the performance of the model. Also the number of epochs used will decrease to 15 to lower the running time and computational cost.

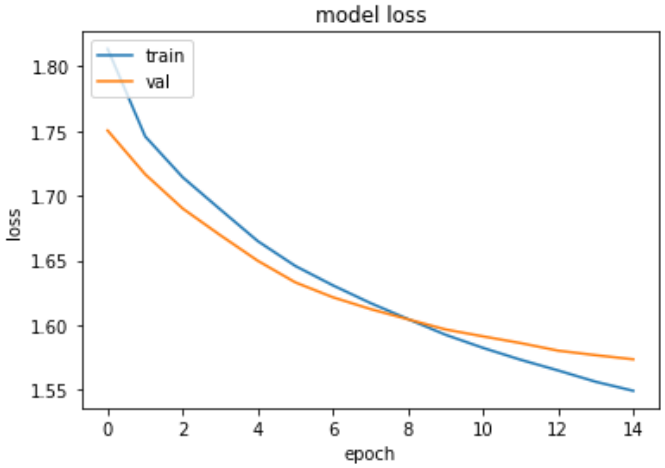
Combination Two

Hyperparameter	**Value**
Number of hidden layer	Two
Number of nodes	1st layer: 1152, 2nd layer: 384
Dropout	25%
Epochs	15
Batch size	200
Activation function	Rule
Optimizer	SGD

Results

Average accuracy on training set	~37%
Accuracy on validation set	39.1%
Running time	19.6 seconds

Cross-entropy vs iterations



Reasoning

Note as another layer is added, the capacity of the model increased. The running time decreased from 34 sections to 19.6 seconds which is more efficient than the previous combination. To better make the model representative the batch size will decrease to 100, this is also expected to decrease the loss in the validation set even more. Also, the optimization method Adam will be tested.

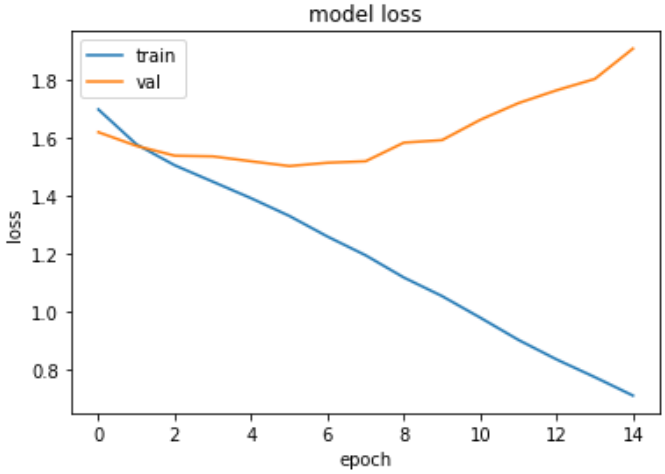
Combination Three

Hyperparameter	**Value**
Number of hidden layer	two
Number of nodes	1st layer: 1152, 2nd layer: 384
Dropout	25%
Epochs	15
Batch size	100
Activation function	Rule
Optimizer	Adam

Results

Average accuracy on training set	55%
Accuracy on validation set	46.8%
Running time	30 seconds

Cross-entropy vs iterations



Reasoning

It is clearly seen in this combination, that the model is overfitting to the training set. This may be due to the use of Adam optimizer which optimizes the learning of the weights resulting in a very accurate weight estimation for training set which results in overfitting. There are two ways this can be solved, first make Adam more stable by reducing the learning rate by increasing the batch size or the epochs; or to go back to using the normal SGD with the same batch size and epochs. The latter scenario will be applied in the next combination because SGD will do the job for FNN.

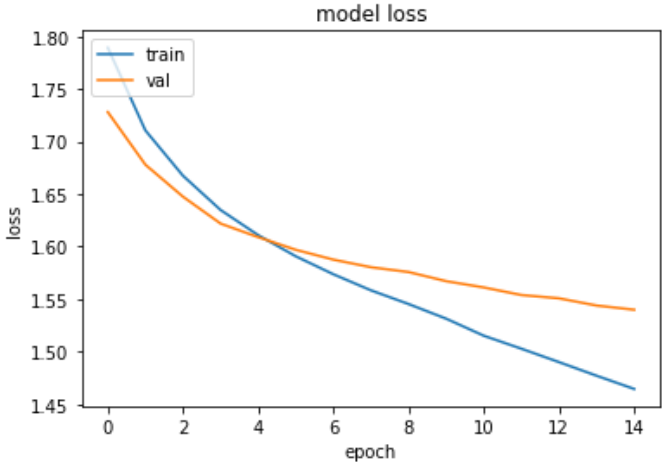
Combination Four

Hyperparameter	**Value**
Number of hidden layer	two
Number of nodes	1st layer: 1152, 2nd layer: 384
Dropout	30%
Epochs	15
Batch size	100
Activation function	Rule
Optimizer	SGD

Results

Average accuracy on training set	40%
Accuracy on validation set	41.4%
Running time	28.7 seconds

Cross-entropy vs iterations



Reasoning

Note as the SGD is used, the model doesn't overfit anymore. Also, note that the decreased batch size of 100, resulted in a more representative model that achieves a lower validation loss. This model compared with the previous cases is considered the best model. Thus, this hyperparameter combination will be used to get the performance of the model on the test dataset.

(cii) FNN Conclusion

The best model obtained through the random search, trial and error, hyperparameter tunning is combination four. This model is now tested on the test set; it gives the following results:

```
[36] # Evaluate your model's performance on the test dataset
     flatten_imgs_test = imgs_test_norm.reshape((-1, 48*48))

     performance = model.evaluate(flatten_imgs_test, to_categorical(labels_test))

     113/113 [==============================] - 0s 4ms/step - loss: 1.5327 - accuracy: 0.4174
```

Note that the accuracy on test dataset: 41.7%

Convolutional Neural Network (CNN)

The convolutional neural network also known as deep neural network are mostly used in image/video recognition, classification, and language processing. What is unique about CNN is that they learn the features of the data which eliminate the need for additional feature engineering. Similar to FNN, CNN consists of the same layers but the hidden layers here consists of a series of convolutional layers. Convolutional kernels which are filters are also defined to extract features from the data. To prevent the model from overfitting a max pooling layer is added between convolutional layers and dropout is also specified for each layer that randomly removes units.

(di) Similar to FNN, first the CNN model will undergo hyperparameter tuning to find the best model on the validation set. The different combinations used include the following:

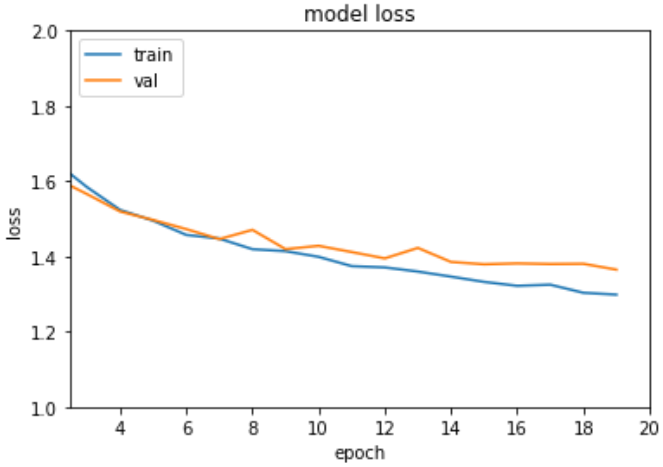
CNN Combination One

Hyperparameter	**Value**
Number of hidden layer	three
Number of filters	1st layer:32 2nd layer: 32 3rd layer:64
Number of kernels	1st layer:1 2nd layer: 1 3rd layer:1
Dropout	2nd: 25% 3rd: 25%
Max pooling	(2,2) after 1st and 3rd layers
Epochs	20
Batch size	500
Activation function	Rule
Optimizer	Adam

Results

Average accuracy on training set	42%
Accuracy on validation set	49%
Running time	1.2 mins

Cross-entropy vs iterations



Reasoning

For the first combination, three hidden convolutional layers are used. Dropout is used along max pooling to prevent overfitting. Also the Rule activation function and Adam optimizer are fixed for CNN since they both give relatively more accurate model representation. Note in the graph that the validation set increases at some points which shows that the model dose overfit over the training, in order to fix this the batch size will go down. Also to increase the model capacity, another hidden layer will be added.

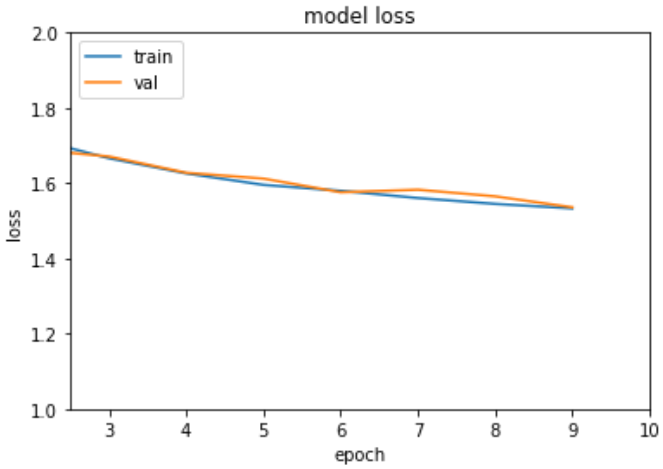
CNN Combination Two

Hyperparameter	Value			
Number of hidden layer	three			
Number of filters	1st layer:32	2nd layer: 32	3rd layer:64	4th :64
Number of kernels	1st layer:2	2nd layer: 2	3rd layer:2	4th : 2
Dropout	1st: 0%	2nd: 25%	3rd: 25%	4th : 25%
Max pooling	(2,2) after 1st 3rd and 4th layers			
Epochs	100			
Batch size	200			
Activation function	Rule			
Optimizer	Adam			

Results

Average accuracy on training set	38%
Accuracy on validation set	44%
Running time	45.9 mins

Cross-entropy vs iterations



Reasoning

As seen above, for the second combination different hyperparameters changed. First, another hidden layer is added to further increase the model capacity. Then, the number of epochs is decreased to 10 and the batch size is decreased to 200 in order to increase the learning rate. It can be seen that the validation set loss is more stable now. For the next combination, a the last convolutional layer will be replaced by a fully connected layer, Dense, which is a better link to the output layer.

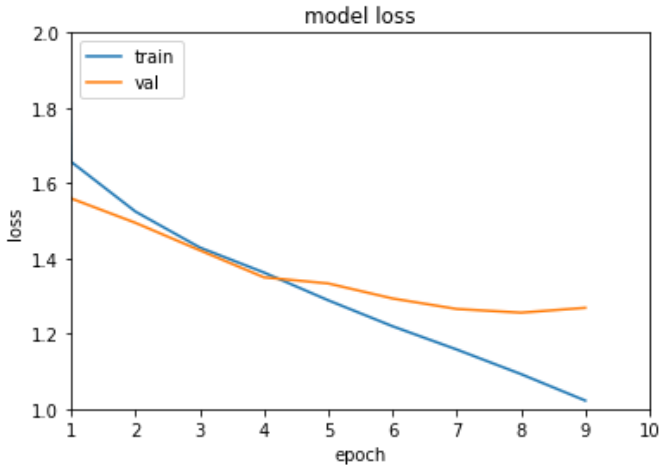
CNN Combination Three

Hyperparameter	**Value**			
Number of hidden layer	three			
Number of filters	1st layer:32	2nd layer: 32	3rd layer:64	4th :64
Number of kernels	1st layer:2	2nd layer: 2	3rd layer:2	4th : 2
Dropout	1st: 0%	2nd: 25%	3rd: 25%	4th : 25%
Max pooling	(2,2) after 1st 3rd and 4th layers			
Epochs	10			
Batch size	200			
Activation function	Rule			
Optimizer	Adam			

Results

Average accuracy on training set	45%
Accuracy on validation set	52%
Running time	80.6 seconds

Cross-entropy vs iterations



Reasoning

It can be seen that the training loss is further decreased after using the dense layer before the output layer. Now, for the next combination, the batch size will decrease to 100 while keeping everything the same increase the loss in the validation set.

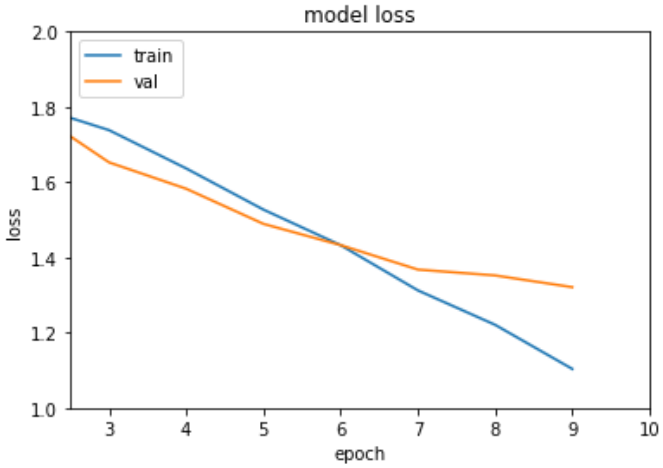
CNN Combination Four

Hyperparameter	**Value**			
Number of hidden layer	three			
Number of filters	1st layer:32	2nd layer: 32	3rd layer:64	4th :64
Number of kernels	1st layer:2	2nd layer: 2	3rd layer:2	4th : 2
Dropout	1st: 0%	2nd: 25%	3rd: 25%	4th : 25%
Max pooling	(2,2) after 1st 3rd and 4th layers			
Epochs	10			
Batch size	100			
Activation function	Rule			
Optimizer	Adam			

Results

Average accuracy on training set	42%
Accuracy on validation set	52%
Running time	90.9 seconds

Cross-entropy vs iterations



Reasoning

Compared to the previous combination, it can be seen that the loss on the validation set is further decreased. Also, note that the accuracy on the validation set is maintained a 52% which is on the higher end. Note that unlike FNN, CNN requires more hyperparameter tunning as it has more hyperparameters relating to the filters and different kinds of layers than FNN.

(dii) CNN Conclusion

The best model obtained through the random search, trial and error, hyperparameter tunning is combination four. This model is now tested on the test set; it gives the following results:

```
[ ] performance = cnn_model.evaluate(test_images_3d, to_categorical(labels_test))
```

```
113/113 [==============================] - 0s 4ms/step - loss: 2.5640 - accuracy: 0.5430
```

Note that the accuracy on test dataset: 54.3%

(e) Bayesian optimization

Bayesian optimization is a model-based method for finding the optimal hyperparameter values. This approach can achieve better performance on the test set while requiring fewer iterations than random search. Moreover, there are now a number of Python libraries that make implementing Bayesian hyperparameter tuning simple for any machine learning model such as hyperopt.

There hyperparameters that are tunned using the Bayesian optimization are the following:

Hyperparameter	Range
Kernel Numbers	[1,3,5]
Dropout	[10% to 35%]
Batch size	[100,250,400]
Epochs number	[5,10,15]

Sample of the evaluations:

```
------------------------------------------------------
Hyperparameters:
{'batch_size': 400, 'conv_kernel_size': 5, 'dropout_prob': 0.13609338247392272, 'epochs': 15}
Accuracy:
0.492337703704834
------------------------------------------------------
Hyperparameters:
{'batch_size': 100, 'conv_kernel_size': 5, 'dropout_prob': 0.3100995525427845, 'epochs': 10}
Accuracy:
0.5190861225128174
------------------------------------------------------
Hyperparameters:
{'batch_size': 100, 'conv_kernel_size': 1, 'dropout_prob': 0.29879732826168526, 'epochs': 5}
Accuracy:
0.2574533224105835
------------------------------------------------------
Hyperparameters:
{'batch_size': 100, 'conv_kernel_size': 3, 'dropout_prob': 0.23341791239623988, 'epochs': 5}
Accuracy:
0.4054054021835327
------------------------------------------------------
```

Results

25 different evolutions were made. The highest resulted accuracy in the validation set is 51.9% using a batch size of 100, 10 epochs, 30% dropout, and 5 kernels. This results in an accuracy on the test set of 51.7%. You can find the full code and full results in the annex below.

Fine-tuning

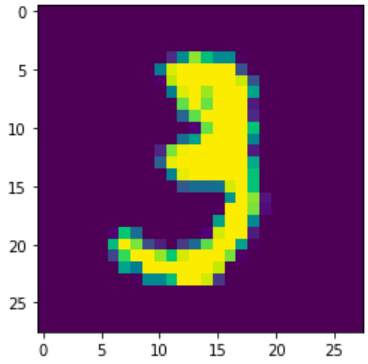
The process of fine-tuning is linked with the idea of transfer learning which mainly mean using a pretrained learning model to a new task. The fine-tuning comes in place to make sure that the model would yield appropriate results for the new task. The fine-tuning process comprises of importing the old model some layers, usually at the higher-level end, get removed and a set of new layers are added. The weights of the per-trained model are frozen and only the weights of the new layers will be changing.

The following steps are taken to apply fine-tuning to per-trained model:

1- The MNIST data is loaded

```
 #Load MNISt dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Check number of samples (60000 in training and 10000 in test)
# Each image has 28 x 28 pixels
print("Train Image Shape: ", train_images.shape, "Train Label Shape: ", train_labels.shape)
print(type(train_images))
print("Test Image Shape: ", test_images.shape, "Test Label Shape: ", test_labels.shape)

#  Visualizing a random image (11th) from training dataset
print("Visualizing a random image (11th) from training dataset")
_ = plt.imshow(train_images[10])
```

```
Train Image Shape:  (60000, 28, 28) Train Label Shape:  (60000,)
<class 'numpy.ndarray'>
Test Image Shape:  (10000, 28, 28) Test Label Shape:  (10000,)
Visualizing a random image (11th) from training dataset
```



2- The MNIST is trained

```
Model: "sequential_59"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_179 (Conv2D)          (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_180 (Conv2D)          (None, 24, 24, 32)        9248
_________________________________________________________________
max_pooling2d_118 (MaxPoolin (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_181 (Conv2D)          (None, 10, 10, 64)        18496
_________________________________________________________________
conv2d_182 (Conv2D)          (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_119 (MaxPoolin (None, 4, 4, 64)          0
_________________________________________________________________
flatten_56 (Flatten)         (None, 1024)              0
_________________________________________________________________
dense_110 (Dense)            (None, 512)               524800
_________________________________________________________________
dense_111 (Dense)            (None, 10)                5130
=================================================================
Total params: 594,922
Trainable params: 594,922
Non-trainable params: 0
_________________________________________________________________
None
```

3- The last layer, output layer is deleted from the MNIST model

```
Model: "sequential_59"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_179 (Conv2D)          (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_180 (Conv2D)          (None, 24, 24, 32)        9248
_________________________________________________________________
max_pooling2d_118 (MaxPoolin (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_181 (Conv2D)          (None, 10, 10, 64)        18496
_________________________________________________________________
conv2d_182 (Conv2D)          (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_119 (MaxPoolin (None, 4, 4, 64)          0
_________________________________________________________________
flatten_56 (Flatten)         (None, 1024)              0
=================================================================
Total params: 64,992
Trainable params: 64,992
Non-trainable params: 0
_________________________________________________________________
None
```

4- The output layer for FER is added to the end of the MNIST model

```
Model: "sequential_59"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_179 (Conv2D)          (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_180 (Conv2D)          (None, 24, 24, 32)        9248
_________________________________________________________________
max_pooling2d_118 (MaxPoolin (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_181 (Conv2D)          (None, 10, 10, 64)        18496
_________________________________________________________________
conv2d_182 (Conv2D)          (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_119 (MaxPoolin (None, 4, 4, 64)          0
_________________________________________________________________
flatten_56 (Flatten)         (None, 1024)              0
_________________________________________________________________
dense_112 (Dense)            (None, 7)                 7175
=================================================================
Total params: 72,167
Trainable params: 72,167
Non-trainable params: 0
_________________________________________________________________
None
```

5- The images are resized from (48,48) to (28,28) similar to MINST images

```
[136] # Resize Trian
      resized_images = []
      train_images_3d = imgs_train.reshape(28709,48,48,1)

      for img in train_images_3d:
              img = img.reshape((48,48)).astype('float32')  # <-- convert image to float32
              resized_img = cv2.resize(img, dsize=(28, 28))
              resized_images.append(resized_img)
      resized_images_train = np.array(resized_images)

      # Resize Validation
```

6- The model is then fit to the FER data set

Combination One

```
Epoch 20/20
29/29 [==============================] - 2s 63ms/step - loss: 0.1771 - accuracy: 0.9508
113/113 [==============================] - 0s 4ms/step - loss: 6.9938 - accuracy: 0.1819
```

Note that the current accuracy is 18% on the validation set. To improve it a hyperparameter fine-tuning will can applied.

Combination Two

Additional Dense layer is added before the output layer with units of 576. Also the epochs number and the batch size are modified to 5 and 50 respectively. Thus, the new accuracy is 25%.

```
model2.fit(train_images_3d3, to_categorical(labels_train), epochs=5, batch_size=20,)
performance = model2.evaluate(val_images_3d3, to_categorical(labels_val))
```

```
Epoch 1/5
1436/1436 [==============================] - 6s 4ms/step - loss: 1.8099 - accuracy: 0.2513
Epoch 2/5
1436/1436 [==============================] - 6s 4ms/step - loss: 1.8099 - accuracy: 0.2513
Epoch 3/5
1436/1436 [==============================] - 5s 4ms/step - loss: 1.8100 - accuracy: 0.2513
Epoch 4/5
1436/1436 [==============================] - 5s 4ms/step - loss: 1.8100 - accuracy: 0.2513
Epoch 5/5
1436/1436 [==============================] - 5s 4ms/step - loss: 1.8100 - accuracy: 0.2513
113/113 [==============================] - 0s 3ms/step - loss: 1.8120 - accuracy: 0.2494
```

Combination three

To further make the data more flexible the batch size is decreased to 20.

```
Epoch 1/5
1436/1436 [==============================] - 5s 3ms/step - loss: 1.8416 - accuracy: 0.2510
Epoch 2/5
1436/1436 [==============================] - 5s 3ms/step - loss: 1.8110 - accuracy: 0.2513
Epoch 3/5
1436/1436 [==============================] - 5s 3ms/step - loss: 1.8100 - accuracy: 0.2513
Epoch 4/5
1436/1436 [==============================] - 5s 4ms/step - loss: 1.8100 - accuracy: 0.2513
Epoch 5/5
1436/1436 [==============================] - 5s 3ms/step - loss: 1.8100 - accuracy: 0.2513
113/113 [==============================] - 0s 3ms/step - loss: 1.8110 - accuracy: 0.2494
```

It can be seen that the accuracy did not change from the previous combination. This it is used as the best model

Histogram of Oriented Gradients (HOG)

The HOG is a feature extraction method. The pixels are represented in a histogram by splitting them into blocks which each block has a certain about of cells. Then histogram cells can be used as an input to an FNN.

The HOG can be applied using the following steps:

1. computing the gradient image in x and y
2. computing gradient histograms
3. normalising across blocks
4. flattening into a feature vecto

Results