Constructing Convolutional Neural Networks on EMNIST

Table of Contents

Summary

The report focuses on investigating, analyzing, and explaining how convolutional neural networks (CNNs) are able to predict the accuracy of the EMNIST dataset. Furthermore, the report shall shine light in regards of the efficiency and effectiveness of CNNs in comparison to a multilayer perceptron (MLP).

It is evident through the investigation that CNN models are the most accurate at predicting the accuracy of the six classes of the EMNIST dataset. However, this does not conclude that they should be used over the MLP models or the VGG16 models, as the MLP model was faster at training when compared to the CNN in terms of the MNIST class.

Furthermore, the VGG16 provided the lowest prediction accuracy and took the longest to train in regards of the MNIST dataset. This suggests that the model was either constructed ineffectively or could utilize hyper tuning.

In terms of this investigation, the CNN models should be chosen over the MLP and VGG16 CNN in regards of the EMNIST dataset. However, there is room for improvement for all three models.

Introduction

Through investigating the capabilities of the CNNs it has become evident that the CNN models are able to accurately predict the EMNIST dataset. Moreover, the report shall shine light on how the CNN model was built to analyze the EMNIST dataset.

A convolutional neural network (CNN) model is a form of deep learning that is able to process data through a grid pattern. A CNN is constructed to be able to adaptively learn spatial hierarchies of features from low to high patterns.

While a multilayer perceptron (MLP) is an artificial neural network that is able to generate outputs from a set of inputs. MLPs are known for their input nodes that consist of several layers that are connected as a directed graph between the input and output layers.

The EMNIST dataset is a continuation of the MNIST dataset, where a set of handwritten character digits are derived from the NIST Special Database 19. The EMNIST dataset is in a 28x28 image pixel format that directly matches the MNIST dataset. The EMNIST dataset consists of six classes, of which are digits, byclass, bymerge, balanced, letters, and mnist. Below is a quick summary of the six classes (Cohen, Afshar, Tapson, & Van Schaik, 2019).

Class	***Number of Characters***	***Balanced or Unbalanced***
digits	280,000	10 balanced classes
byclass	814,255	47 unbalanced classes
bymerge	814,255	47 unbalanced classes
balanced	131,600	47 balanced classes
letters	145,600	26 balanced classes
mnist	70,000	10 balanced classes

A pre-trained model can be viewed as a model that has been previously built and trained on an extremely large dataset. A pre-trained model can be used as an extension to the model that the user has already built to increase the effectiveness of the model.

Within the realm of pre-trained networks, VGG16 is a CNN model with 16 layers that has been trained on over one million images from the ImageNet database. The VGG16 model is able to classify images into a thousand different object categories.

Within our investigation, the VGG16 pre-trained model shall be used as an extension to the MNIST class CNN model. Utilizing the VGG16 within the CNN model for the MNIST class will shine light on whether the model will become more accurate and proficient at predicating the accuracy.

Methods

Due to the fact that the EMNIST dataset is large, the procedure of creating the CNN code can be split into six major stages.

First stage

The first stage of creating the code for the CNN model was to initialize a workspace. Initializing a workspace is considered the fundamental feature of the code. Within the workspace, the libraries required were loaded in, the EMNIST dataset was connected to the IDE through a google drive directory, a function was created to load a specific class of the EMNIST dataset and create the train, test, and validation variables.

Second stage

The second stage of the code consisted of data pre-processing, where the train, test, and validation variables were modified to be able to utilize within the CNN model. The data pre-processing consisted of reshaping x_train, x_test, and x_val into the size (28*28*1), where then all the variables were converted into a 'float32' shape. Furthermore, the y_train, y_test, and y_val were converted into categorical variables. The final stage of data pre-processing consisted of creating the CNN model parameters, where the batch size, epochs and verbose have been selected and identified.

Third stage

The third stage of the code consisted of building a CNN model. Constructing the CNN model incorporated five components:

⇒ Convolutional Layers

- Convolution layers apply convolution filters to the image, to which the layer performs mathematical operations to produce a single output feature map. Furthermore, the convolution layer utilizes a 'ReLU' activation to introduce nonlinearities into the model.

⇒ Pooling Layers

- This layer extracts subregions of the feature map in 2*2 pixel tiles and ensures that their maximum values are kept while the other values are disregarded. In order to reduce the processing time and dimensionality of the feature map, the pooling layer down samples the data from the image that was extracted by the convolutional layer

⇒ Batch Normalization

- This layer is utilized to enable layers within the network to learn in a more independent method. This layer normalizes the output of the previous layers.

⇒ Flatten

- This layer is utilized when converting the 2D arrays from the pooled feature maps into a single linear vector. This input is then utilized as an input within the fully connected layer to classify images.

⇒ Dense layers

- The Dense layer is the final layer that connects every node within the layer to the previous layer. Furthermore, this layer performs a classification on the features extracted by the pooling and convolutional layers.

After the model was built, the model was then compiled with the 'adam' optimizer, 'categorical_crossentropy' loss, and an 'accuracy' metrics.

As the EMNIST dataset consists of six different sized classes, each class was built with a CNN model that corresponds to the size of the class.

Fourth stage

The fourth stage consisted of fitting the model to the training data, with the validation data acting as data points to evaluate the loss and metrics at the end of each epoch. The model was fitted to a batch size of 128, epochs of 10, and a verbose of 2. As the model was being trained, a time function was utilized to record the amount of time was taken.

Fifth stage

The fifth stage consisted of plotting the loss and accuracy of the model.

Sixth stage

The final stage of building the CNN was to fit the model created to the training, test, and validation data. Fitting the CNN model to these variables was to calculate the prediction accuracy and loss of the CNN model on the EMNIST dataset, while also recording the amount of time taken.

Findings

Throughout this investigation, it is prominent to highlight that the CNN models are efficient and effective at predicting the accuracy of the EMNIST dataset. CNN models outperform MLP models in regards of accuracy and time taken to build and train.

To ensure a deeper understanding of the performance of the CNN models, an investigation into to these models shall be applied. The investigation will revolve around analyzing the overall performance and accuracy of the CNN models in comparison to MLP's.

Results of CNN models

Through creating six different CNN models for each class of the EMNIST dataset, the results can be seen in the table below

Class	Train		Test		Validation	
	Time (seconds)	Accuracy	Time (seconds)	Accuracy	Time (seconds)	Accuracy
Digits	15.81	99.85%	5.26	99.52%	5.21	99.44%
Byclass	40.81	87.48%	7.91	86.57%	7.94	86.46%
Bymerge	83.02	93.05%	20.77	90.56%	11.21	90.66%
Balanced	9.34	93.77%	1.81	87.63%	1.83	87.43

Letters	10.63	97.12%	2.62	94.07%	1.87	94.3%
MNIST	4.47	99.5%	0.95	98.95%	0.96	98.97%

It is evident that from the table above, the CNN model was most accurate at predicting the accuracy of the digits class of the EMNIST dataset. The CNN model predicted the test accuracy at 99.52% which only took 5.26 seconds to run the model over the test variables. Furthermore, it is evident that the train accuracy is exceptionally high compared to the other classes (excluding MNIST). The train accuracy of the CNN model was 99.85% that was ran in 15.81 seconds, while the validation accuracy was 99.44% that was ran in 5.21 seconds. The CNN model was exceptionally in regards of the digits class of the EMNIST dataset. This could be a result of carefully choosing the components when constructing the model and ensuring that the hyper tuning of the model was carefully done.

It is prominent to state that the CNN model was also highly effective in regards of the MNIST class of the EMNIST dataset. The CNN model predicted the test accuracy at 98.95% which took 0.95 seconds to run, while the training accuracy was predicted at 99.5% which took 4.47 seconds to run, and finally the validation which was predicted at 98.97% which took 0.96 seconds to run. The CNN model was highly effective at predicting the MNIST class. This could be a result of choosing the correct components when constructing the CNN model, while also taking into consideration that the hyper tuning of the model was additionally carefully selected.

However, the CNN model did not perform as well as it did in regards of the byclass, bymerge and balanced classes of the EMNIST dataset. The highest accuracy predicted from these classes was the bymerge class with a predicted accuracy of 90.56% which took 20.77 seconds, while the rest of the classes test accuracy was under 90%. The poor performance of the CNN models in regards of these classes could have been caused by a number of factors, of which include, poorly selecting the components when constructing the model, and not hyper tuning the model effectively. These factors have impacted the overall predicted accuracy of those three classes and the time taken to run the model on them.

CNN in comparison to MLP

The difference between CNN and MLP models can be found within the type of inputs each model takes. As CNN models take tensor inputs, while MLP models take vectors as an input. This concludes that within image classification, CNN models are more proficient at comprehending spatial relation, hence CNN models should outperform MLP models when given complicated images. Therefore, the investigation into CNNs in comparison to MLPs will be focused on the MNIST class of the EMNIST dataset.

The results of the CNN and MLP models regarding the MNIST class of the EMNIST dataset can be seen in the table below.

Model	**Train**		**Test**		**Validation**	
	Time (seconds)	**Accuracy**	**Time (seconds)**	**Accuracy**	**Time (seconds)**	**Accuracy**
CNN	4.47	99.5%	0.95	98.95%	0.96	98.97%
MLP	2.76	99.38%	0.54	98.29%	0.54	98.24%

Ratio	**1.62**	**1.0012**	**1.76**	**1.0067**	**1.78**	**1.0074**

The table above indicates that the CNN model exceeded the MLP model at predicting the accuracy of the MNIST class of the EMNIST dataset, while the MLP exceeded the CNN in regards of time taken to run the model. It is evident that the ratio between the CNN and the MLP models in regards of the training variable accuracy is 1.0012, while the test was 1.0067, and the validation was 1.0074. While the ratio between the CNN and the MLP in regards of time for the training variable was 1.62, the test time was 1.76, and the validation was 1.78.

Although the CNNs outperformed the MLPs in regards of predicting the accuracy, it is essential to understand that CNNs consume more time running over the data, as the number of parameters within the CNN model are more than the parameters within the MLP model. Furthermore, CNNs are typically faster at converging than MLP models in terms of epochs.

Furthermore, CNN models are able to go deeper in regards of image classification, this is due to the fact that the layers within the CNN model are sparsely connected instead of fully connected as MLP models do. Moreover, CNN models utilize smaller and shared weights that are easier to train in comparison to MLP models.

Overall, in terms of prediction accuracy of the MNIST class of the EMNIST dataset, it is evident that the CNN model has provided better prediction accuracies at the cost of time taken to run the model.

VGG16 CNN in comparison to MLP

The results of the VGG16 CNN model compared to the MLP model can be seen in the table below

Model	**Train**		**Test**		**Validation**	
	Time (seconds)	**Accuracy**	**Time (seconds)**	**Accuracy**	**Time (seconds)**	**Accuracy**
CNN	4.47	99.5%	0.95	98.95%	0.96	98.97%
MLP	2.76	99.38%	0.54	98.29%	0.54	98.24%
VGG16 CNN	47.48	98.68%	9.26	97.7%	10.27	97.46

The table above indicates the VGG16 CNN model did not perform as expected, as the VGG16 model was the worst performer in comparison to the CNN and MLP models.

While constructing the VGG16 CNN model, 20 epochs have been utilized. This has been done due to the fact that the prediction accuracy of the model was extremely low. Through investigating into the images that the VGG16 has been trained on, it has been recognized that the model was primarily trained on everyday image objects. The images that the VGG16 has been trained on do not consist of any numbers.

As the VGG16 has not been trained on any digit images, it could be the reasoning behind the lower prediction accuracies than the CNN and the MLP models. However, another reasoning behind the low prediction accuracy could be a result of poorly constructing the layers.

However, the loss and accuracy plots of the VGG16 CNN model suggest that if the model was to be trained with a larger number of epochs, it could reach a higher accuracy of prediction. Furthermore, the loss and accuracy plot of the VGG16 is better than the MLP and CNN plots, in terms of the training and validation loss and accuracy.

The overall findings of this investigation conclude to the fact that CNN models are proficient at predicting the accuracy of classes within the EMNIST dataset. Where the CNN model was able to outperform the MLP model in regards of prediction accuracy as well as outperform the pre-trained VGG16 model in terms of accuracy and time taken.

This concludes to the fact that CNNs are easier to construct over MLP models and should be considered instead of MLPs when constructing image classification models.

Conclusions

Through investigating the construction of a convolutional neural network (CNN) it has become apparent that these models are simpler to construct, build, and train in comparison to multilayer perceptron's (MLP) and the pre-trained VGG16.

By constructing a CNN model for the six classes of the EMNIST dataset, it is evident that the model can accurately predict the digits, letters and MNIST classes. While the CNN model underperformed within the classes of byclass, bymerge and balanced, where the prediction accuracy was fairly low compared to the high prediction accuracy classes.

By comparing the CNN model to the MLP it was evident that CNN models had a higher prediction accuracy than the MLP, yet the MLP had a faster training time. Moreover, it is relevant to understand that CNNs can outperform MLP models in regards of processing more complex images in the field of image classification

Furthermore, the VGG16 compared to the CNN and MLP has shown that the VGG16 was not suitable for the EMNSIT dataset. This is due to the fact that the VGG16 CNN model required the longest time to train the mode and provided the lowest prediction accuracies compared to the CNN and MLP models.

Recommendations

In terms of constructing better CNN models, it is essential that the model can be constructed to provide high prediction accuracies and fast training time. Therefore, the CNN model for the following classes, byclass, bymerge and balanced should be reconstructed or hyper tuned to provide a higher prediction accuracy. Furthermore, the model can be improved through either running more epochs or hyper tuning the model to become better.

In terms of the comparison between the CNN and the MLP models, it would be recommended to construct and build a CNN and MLP model for each class of the EMNIST dataset. Creating a model for each class of the EMNIST dataset would provide more insight into whether or not MLPs are also as efficient at predicting the accuracy compared to the CNN models.

In terms of the comparison between the VGG16 CNN, CNN, and MLP comparison, it is essential to reconstruct and hyper tune the VGG16 model. Reconstructing and hyper tuning the pre-trained VGG16 model is essential as the prediction results and time taken to run the model compared to the CNN and MLP were low. If the VGG16 model could not provide prediction accuracies closer

to the CNN and the MLP then another pre-trained model should be considered as another pre-trained model could be more effective at predicting the accuracy.

Appendix

Code:

```
# -*- coding: utf-8 -*-
"""DTSC 301 - Assignment 2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1c-9pRnfprUXRf7mWXFr4SeQGqRTKuudV

# DTSC13-301 Assignment 2
### #13703981

### Initializing Workspace
"""

pip install idx2numpy &> /dev/null

pip install tensorflow &> /dev/null

# Importing libraries
import idx2numpy
import tensorflow as tf
from tensorflow import keras
from keras import layers, models
from tensorflow.keras import layers
from keras.models import Sequential
from keras.layers import MaxPool2D, Dropout, Flatten, Dense, Reshape, Conv2D,
BatchNormalization
import tensorflow as tf
from tensorflow.keras.utils import img_to_array, array_to_img, to_categorical
from tensorflow.keras.applications import VGG16
import matplotlib.pyplot as plt
import numpy as np
from datetime import timedelta
import time as tm
from google.colab import drive

# Connnecting to google drive
drive.mount('/content/drive')

# Data importing function
def importer(location, dataset):
    x_train = idx2numpy.convert_from_file(location + '/emnist-' + dataset + '-train-images-idx3-
ubyte')
```

```
    y_train = idx2numpy.convert_from_file(location + '/emnist-' + dataset + '-train-labels-idx1-ubyte')
    x_test = idx2numpy.convert_from_file(location + '/emnist-' + dataset + '-test-images-idx3-ubyte')
    y_test = idx2numpy.convert_from_file(location + '/emnist-' + dataset + '-test-labels-idx1-ubyte')
    x_train, x_test = x_train / 255.0, x_test / 255.0

    break_point = (x_train.shape[0] - x_test.shape[0])

    x_val = x_train[break_point:]
    x_train = x_train[:break_point]
    y_val = y_train[break_point:]
    y_train = y_train[:break_point]
    return x_train, y_train, x_val, y_val, x_test, y_test

loc = '/content/drive/MyDrive/EMNIST Data'

"""### Digits"""

# Importing digits dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'digits')

# Data Pre-Processing
## Reshaping data
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting data into float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_val = x_val.astype('float32')

y_train = y_train.astype('float32')
y_test = y_test.astype('float32')
y_val = y_val.astype('float32')

## Converting to categorical
y_train = to_categorical(y_train, num_classes = 10)
y_val = to_categorical(y_val, num_classes = 10)
y_test = to_categorical(y_test, num_classes = 10)

## Model Parameters
INPUT_SHAPE = (28,28,1)
BATCH_SIZE = 128
```

```
EPOCHS = 10
VERBOSE = 2

# Creating model
model = Sequential()

model.add(Conv2D(32, kernel_size=(3,3), activation='relu', input_shape=INPUT_SHAPE))
model.add(MaxPool2D((2,2)))

model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(10, activation='softmax'))

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

model.summary()

# Model Fitting
start = tm.time()
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    verbose=1, validation_data=(x_val, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)
```

```
# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train, y_train, verbose = False)
end = tm.time() - start
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

"""### Byclass"""

# Importing byclass dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'byclass')

# Data Pre-Processing
## Reshaping data
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting data into float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_val = x_val.astype('float32')

y_train = y_train.astype('float32')
y_test = y_test.astype('float32')
y_val = y_val.astype('float32')

## Converting to categorical
y_train = to_categorical(y_train)
```

```
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)

## Model Parameters
BATCH_SIZE = 128
EPOCHS = 10
VERBOSE = 2

# Creating Model
model = Sequential()

model.add(Conv2D(32, kernel_size=(3,3), activation='relu', input_shape=(28,28,1)))
model.add(MaxPool2D((2,2)))

model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(100, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(62, activation='softmax'))

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

model.summary()

# Model Fitting
start = tm.time()
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    verbose=1, validation_data=(x_val, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
```

```
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)

# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train, y_train, verbose = False)
end = tm.time() - start
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

"""### Bymerge"""

# Importing bymerge dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'bymerge')

# Data Pre-Processing
## Reshaping data
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting data into float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_val = x_val.astype('float32')
```

```
y_train = y_train.astype('float32')
y_test = y_test.astype('float32')
y_val = y_val.astype('float32')

## Converting to categorical
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)

## Model Parameters
BATCH_SIZE = 128
EPOCHS = 10
VERBOSE = 2

# Creating model
model=Sequential()

model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(512,activation="relu"))

model.add(Dense(47,activation="softmax"))

# Compiling Model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

# Model Fitting
start = tm.time()
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
```

```
                    verbose=1, validation_data=(x_val, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)

# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train, y_train, verbose = False)
end = tm.time() - start
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

"""### Balanced"""

# Importing balanced dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'balanced')

# Data Pre-Processing
## Reshaping data
```

```
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting data into float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_val = x_val.astype('float32')

y_train = y_train.astype('float32')
y_test = y_test.astype('float32')
y_val = y_val.astype('float32')

## Converting to categorical
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)

# Model Parameters
BATCH_SIZE = 128
EPOCHS = 10
VERBOSE = 2

# Creating model
model=Sequential()

model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(512,activation="relu"))

model.add(Dense(47,activation="softmax"))

# Compiling Model
```

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

model.summary()

# Model Fitting
start = tm.time()
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    verbose=1, validation_data=(x_val, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)

# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train, y_train, verbose = False)
end = tm.time() - start
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
```

```
print('Time taken: ', round(end, 2), 'seconds')

"""### Letters"""

# Importing letters dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'letters')

# Data Pre-Processing
## Reshaping data
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting data into float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_val = x_val.astype('float32')

y_train = y_train.astype('float32')
y_test = y_test.astype('float32')
y_val = y_val.astype('float32')

## Converting to categorical
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)

# Model Parameters
BATCH_SIZE = 128
EPOCHS = 10
VERBOSE = 2

# Creating model
model=Sequential()

model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
```

```
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(512,activation="relu"))

model.add(Dense(27,activation="softmax"))

# Compiling Model
model.compile(optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy'])

model.summary()

# Model Fitting
start = tm.time()
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    verbose=1, validation_data=(x_val, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)

# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train, y_train, verbose = False)
end = tm.time() - start
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
```

```
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

"""### MNIST"""

# Importing dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'mnist')

# Data Pre-Processing
## Reshaping data
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting data into float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_val = x_val.astype('float32')

y_train = y_train.astype('float32')
y_test = y_test.astype('float32')
y_val = y_val.astype('float32')

## Converting to categorical
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)

# Model Parameters
BATCH_SIZE = 128
EPOCHS = 10
VERBOSE = 2

# Creating model
model=Sequential()

model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
```

```
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(512,activation="relu"))

model.add(Dense(10,activation="softmax"))

# Compiling Model
model.compile(optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy'])

model.summary()

# Model Fitting
start = tm.time()
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    verbose=1, validation_data=(x_val, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)

# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train, y_train, verbose = False)
end = tm.time() - start
```

```
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

"""### MNIST utlizing VGG16"""

# Importing digits dataset
x_train, y_train, x_val, y_val, x_test, y_test = importer(loc, 'mnist')

# Data Pre-Processing
## Reshaping variable
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
x_val = x_val.reshape(x_val.shape[0], 28,28,1)

## Converting variable to categorical
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)

## ads
x_train1 = tf.image.grayscale_to_rgb(tf.constant(x_train))
x_train1 = tf.pad(x_train1, ((0, 0), (2, 2), (2, 2), (0, 0)))
x_train1 = keras.applications.resnet_v2.preprocess_input(tf.cast(x_train1, tf.float32))

x_test1 = tf.image.grayscale_to_rgb(tf.constant(x_test))
x_test1 = tf.pad(x_test1, ((0, 0), (2, 2), (2, 2), (0, 0)))
x_test1 = keras.applications.resnet_v2.preprocess_input(tf.cast(x_test1, tf.float32))

x_val1 = tf.image.grayscale_to_rgb(tf.constant(x_val))
```

```
x_val1 = tf.pad(x_val1, ((0, 0), (2, 2), (2, 2), (0, 0)))
x_val1 = keras.applications.resnet_v2.preprocess_input(tf.cast(x_val1, tf.float32))

## Model parameters
lambda_ = 1e-3

# Creating model
model = keras.Sequential()
core = keras.applications.ResNet152V2(
   include_top=False,
   weights='imagenet',
   input_shape=x_train1.shape[1:])

for layer in core.layers[:-2]:
   layer.trainable = False

model.add(core)
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation='softmax',
                        kernel_regularizer=keras.regularizers.l2(lambda_)))

model.compile(
   optimizer=keras.optimizers.Adam(),
   loss='categorical_crossentropy',
   metrics=['accuracy'])

model.summary()

# Model fitting
start = tm.time()
history = model.fit(x_train1, y_train, batch_size=128, epochs=20,
                    verbose=1, validation_data=(x_val1, y_val))
end = tm.time() - start

print('time taken{}'.format(timedelta(seconds=end)))

# Plotting model training and validation curves
fig, ax = plt.subplots(2,1, figsize=(18, 10))
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)
```

```
# Model loss and accuracy on train set
start = tm.time()
val_loss, val_acc = model.evaluate(x_train1, y_train, verbose = False)
end = tm.time() - start
print('Model accuracy on train data: ', round(val_acc*100,2))
print('Model loss on train data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on test set
start = tm.time()
val_loss, val_acc = model.evaluate(x_test1, y_test, verbose = False)
end = tm.time() - start
print('Model accuracy on test data: ', round(val_acc*100,2))
print('Model loss on test data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')

# Model loss and accuracy on validation set
start = tm.time()
val_loss, val_acc = model.evaluate(x_val1, y_val, verbose = False)
end = tm.time() - start
print('Model accuracy on validation data: ', round(val_acc*100,2))
print('Model loss on validation data: ', round(val_loss, 5))
print('Time taken: ', round(end, 2), 'seconds')
```