JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA

Fundamentals of Soft Computing PBL

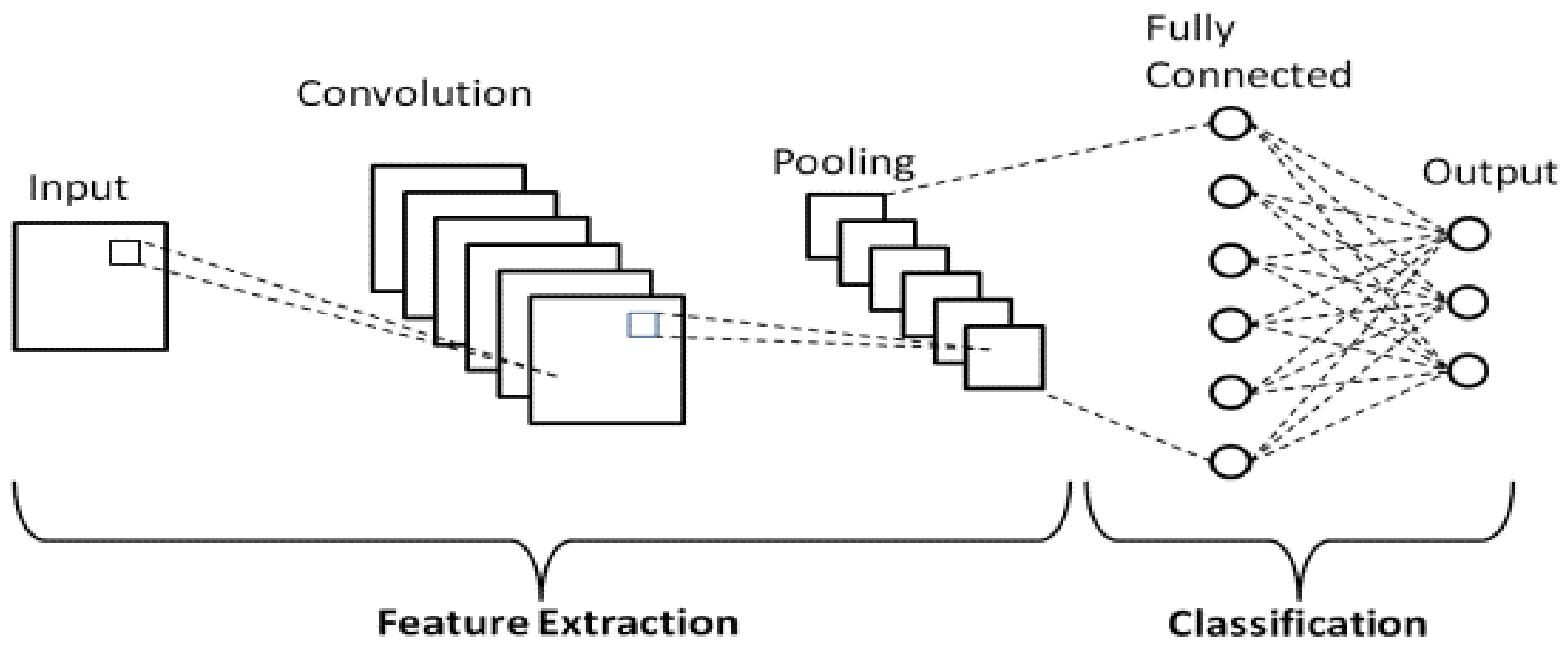
TOPIC : Digit Recognition

Name of Student (s) - Mufti Usman, Anurag Sati, Paakhi Maheshwari
Enrollment No. (s) - 21103165, 21103153, 21103149

Submitted to - Dr. Dharamveer Singh Rajpoot

1. Abstract:

The project aims to develop a robust solution for handwritten digit recognition using Convolutional Neural Networks (CNNs). The primary objective is to train a model capable of accurately classifying digits from the MNIST dataset, a benchmark dataset widely used for digit recognition tasks. By leveraging deep learning techniques implemented in TensorFlow and Keras, the project endeavors to achieve high accuracy and efficiency in digit classification. Through extensive experimentation and model refinement, the project seeks to showcase the effectiveness of CNNs in image classification tasks and their potential applications in real-world scenarios requiring automated digit recognition.



2. Introduction:

Handwritten digit recognition has been a longstanding challenge in the field of pattern recognition and machine learning. It holds significant importance in various domains such as postal services, banking, and document processing, where automated recognition of handwritten digits can streamline operations and enhance efficiency.

The motivation behind this project stems from the increasing demand for automated systems capable of accurately interpreting handwritten data. Traditional methods of digit recognition often rely on handcrafted features and machine learning algorithms, which may struggle to generalize across diverse handwriting styles and variations.

In recent years, deep learning techniques, particularly Convolutional Neural Networks (CNNs), have emerged as a powerful tool for image classification tasks, including handwritten digit recognition. CNNs are designed to automatically learn hierarchical features from raw pixel data, making them well-suited for tasks involving image analysis and recognition.

The MNIST dataset serves as the de facto benchmark for evaluating the performance of digit recognition algorithms. It consists of a large collection of grayscale images of handwritten digits (0-9), each labeled with its corresponding digit. By training and testing our model on the MNIST dataset, we can objectively assess its accuracy and generalization capabilities.

The utilization of deep learning frameworks such as TensorFlow and Keras provides us with the necessary tools to implement and experiment with CNN architectures effectively. These frameworks offer high-level abstractions for building and training neural networks, allowing us to focus on model design and optimization rather than low-level implementation details.

Through this project, we aim to demonstrate the effectiveness of CNNs in digit recognition tasks and showcase their potential for real-world applications. By leveraging the power of deep learning, we aspire to develop a robust and efficient system capable of accurately identifying handwritten digits with high precision and reliability.

3. Technology and Algorithm Used:

3.1 Technology:

3.1.1. TensorFlow and Keras: TensorFlow is an open-source machine learning framework developed by Google, widely used for building and training deep learning models. Keras is a high-level neural networks API, built on top of TensorFlow, that provides an intuitive interface for constructing neural networks. In this project, we leverage both TensorFlow and Keras for implementing our digit recognition system.

3.2 Algorithm:

3.2.1. Convolutional Neural Networks (CNNs): CNNs are a class of deep neural networks that are particularly effective for image recognition and classification tasks. They are inspired by the organization of the visual cortex in animals and are designed to automatically learn spatial hierarchies of features from input images. CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

3.2.2. Convolutional Layers: These layers apply a set of learnable filters (also known as kernels) to the input image, scanning across the image to extract local features. Each filter captures different patterns or features present in the input data, such as edges, textures, or shapes.

3.2.3. Pooling Layers: Pooling layers down-sample the feature maps generated by convolutional layers, reducing their spatial dimensions. This helps in reducing computational complexity and controlling overfitting by capturing the most important features while discarding irrelevant details.

3.2.4. Fully Connected Layers: Fully connected layers are traditional neural network layers where every neuron is connected to every neuron in the previous and subsequent layers. These layers perform classification based on the features extracted by convolutional and pooling layers.

3.2.5. Activation Functions: Rectified Linear Unit (ReLU) activation functions are commonly used in CNNs to introduce non-linearity, allowing the network to learn complex patterns and relationships in the data.

3.2.6. Softmax Activation: The softmax activation function is typically applied to the output layer of the CNN to convert the network's raw output into probability scores for each class. It ensures that the output probabilities sum up to one, making it suitable for multi-class classification tasks like digit recognition.

3.2.7. Adam Optimizer: The Adam optimizer is an adaptive learning rate optimization algorithm that is widely used for training deep neural networks. It automatically adjusts the learning rate during training based on the gradients of the loss function, leading to faster convergence and improved performance.

4. Parameters of Algorithm and its representation:

In Convolutional Neural Networks (CNNs), parameters such as filter sizes, kernel sizes, activation functions, and optimization algorithms play pivotal roles in shaping the network's architecture and behavior. Filters, represented by learnable parameters, slide across input images to extract local features, with choices like 3x3 or 5x5 kernels influencing feature detection. Pooling layers down-sample feature maps using parameters like pool size and pooling type, while fully connected layers employ activation functions like ReLU to introduce non-linearity. The softmax function at the output layer converts raw scores to probability distributions. Optimizers such as Adam adjust network parameters based on gradients, with the learning rate controlling optimization step size. Batch size and epochs dictate the number of samples processed per iteration and the number of training iterations over the entire dataset, respectively. Effective tuning and selection of these parameters are critical for optimizing CNN performance in digit recognition tasks.

5. Code:

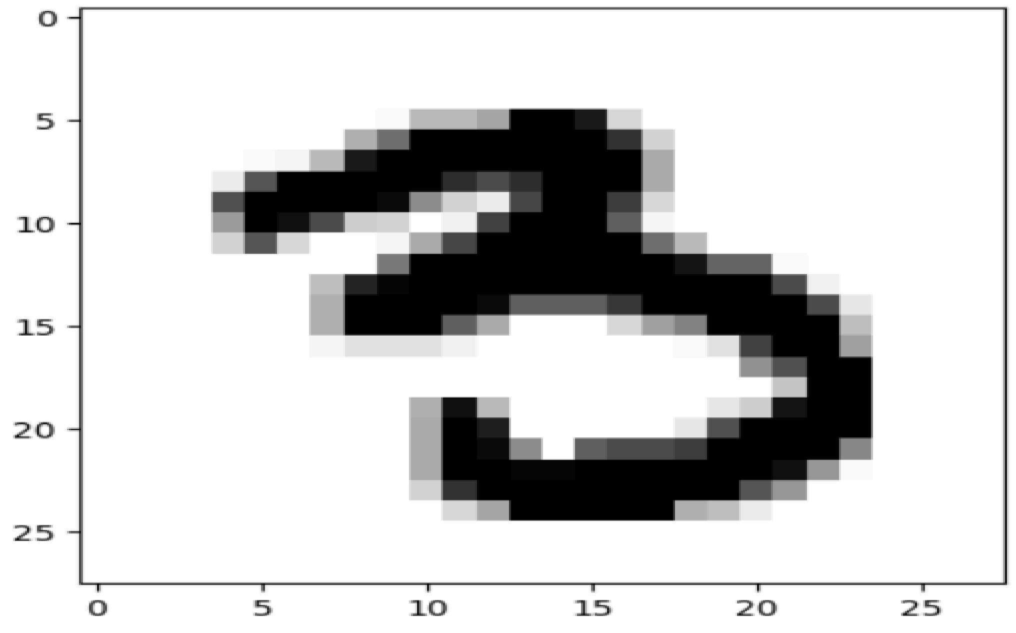
```
import itertools
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.layers import Flatten, Dense, Conv2D, Lambda, MaxPooling2D, Dropout, BatchNormalization
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
train_df = pd.read_csv('train.csv')
test_df  = pd.read_csv('test.csv')
train_df.head()
test_df.head()
# Data preprocessing - shuffling the training dataframe takes ~7 minutes
train_df = train_df.sample(frac=1).reset_index(drop=True)
cols = list(train_df.columns)
cols.remove('label')

x = train_df[cols]
y = train_df['label']

# Splitting the dataset into training and validation(dev) sets
x_train, x_dev, y_train, y_dev = train_test_split(x, y, test_size=0.1, random_state=0)

# Testing dataset (this is the set on which we'll do predictions and then submit it)
x_test = test_df[cols]

print(f'Training set size: {len(x_train)}')
print(f'Validation set size: {len(x_dev)}')

print(x_train.shape)
print(x_train.values.reshape(-1, 28, 28).shape)

# Reshaping the datasets to feed images of 28X28 pixels to our neural network
# And also scaling the images
x_train = x_train.values.reshape(-1, 28, 28) / 255
x_dev   = x_dev.values.reshape(-1, 28, 28) / 255
x_test  = x_test.values.reshape(-1, 28, 28) / 255

print(x_train.shape)
print(np.expand_dims(x_train, axis=-1).shape)

# Adding an additional dimension of channel (1 as images are grayscale)
x_train = np.expand_dims(x_train, axis=-1)
x_dev   = np.expand_dims(x_dev, axis=-1)
x_test  = np.expand_dims(x_test, axis=-1)

plt.imshow(x_train[0].reshape((28, 28)), cmap=plt.cm.binary)

# Looking at first 25 training examples

plt.figure(figsize=(10, 10))
```

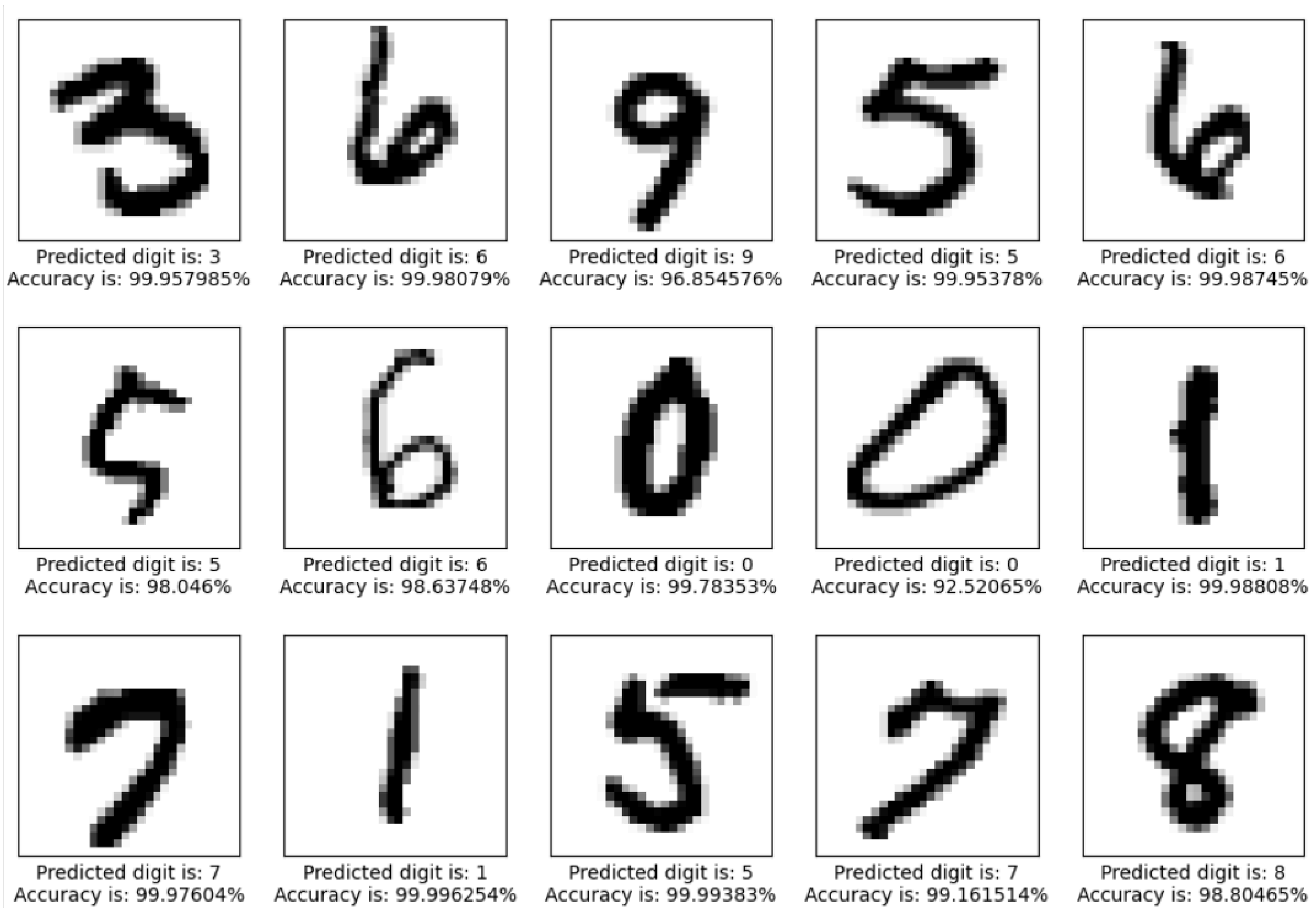
```
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])

    # reshaping the images as (28, 28, 1) is an invalid shape to plot imgs
    plt.imshow(x_train[i].reshape((28, 28)), cmap=plt.cm.binary)

plt.show()

def data_augmentation(x_data, y_data, batch_size):
    datagen = ImageDataGenerator(
        featurewise_center=False,            # set input mean to 0 over the dataset
        samplewise_center=False,             # set each sample mean to 0
        featurewise_std_normalization=False, # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,                 # apply ZCA whitening
        rotation_range=10,                   # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1,                    # Randomly zoom image
        width_shift_range=0.1,               # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.1,              # randomly shift images vertically (fraction of total height)
        horizontal_flip=False,               # randomly flip images
        vertical_flip=False,                 # randomly flip images
    )


    datagen.fit(x_data)
    train_data = datagen.flow(x_data, y_data, batch_size=batch_size, shuffle=True)

    return train_data

BATCH_SIZE = 64
aug_train_data = data_augmentation(x_train, y_train, BATCH_SIZE)

#Modelling
def build_model():
    # Neural Network Architecture
    layers = [
        Conv2D(filters=96, kernel_size=(11, 11), strides=2, activation='relu', input_shape=(28, 28, 1)),
        MaxPooling2D(pool_size=(3, 3), strides=2),
        Conv2D(filters=256, kernel_size=(5, 5), padding='same', activation='relu'),
        Flatten(),
        Dense(9216, activation='relu'),
        Dense(4096, activation='relu'),
        Dense(4096, activation='relu'),
        Dense(10, activation='softmax')
    ]

    model = Sequential(layers)

    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model


model = build_model()
model.summary()

callbacks = [
    ReduceLROnPlateau(monitor="loss",factor=0.1, patience=2, min_lr=0.000001, verbose=1),
]

#takes 9 minutes
history = model.fit(
    aug_train_data,
    steps_per_epoch=x_train.shape[0] // BATCH_SIZE,
    batch_size=BATCH_SIZE,
    validation_data=(x_dev, y_dev),
    epochs=1,
    callbacks=callbacks
)

#Visualizing CNN - Plotting the 96th filter of the 1st conv layer
top_layer = model.layers[0]
plt.imshow(top_layer.get_weights()[0][:, :, :, 95].squeeze(), cmap='gray')

print(f'Shape of 1st conv layer weights: {model.layers[0].get_weights()[0].shape}')
print(f'Shape of 2nd conv layer weights: {model.layers[2].get_weights()[0].shape}')

def plot_filters_for_conv_layer(model, layer_index, num_columns=5, cmap='binary', how_many='all'):
    layer = model.layers[layer_index]
    filter_weights = layer.get_weights()[0]

    num_filters = layer.filters if how_many == 'all' else how_many
    num_rows = (num_filters // num_columns) + (num_filters % num_columns)

    f, axs = plt.subplots(num_rows, num_columns, figsize=(20, 5 * num_rows))
    row_count = 0  # to plot num_columns figs in an individual row

    if not isinstance(axs, np.ndarray):
        # When num_cloumns == how_many
        axs = np.array(axs)  # to make axs iterable

    for idx, row_ax in enumerate(axs):
        # plotting filters in a row
        for i, ax in enumerate(row_ax):
            if row_count + i >= num_filters:
                break
            if len(filter_weights.shape) == 4:
                if filter_weights.shape[2] == 1:
                    ax.imshow(filter_weights[:, :, :, row_count + i].squeeze(), cmap=cmap)
                else:
                    ax.imshow(filter_weights[:, :, 0, row_count + i].squeeze(), cmap=cmap)
            else:
                break

        # increasing row_count by num_columns
        row_count += num_columns

plot_filters_for_conv_layer(
    model,
    0,
    num_columns=4,
    cmap=sns.cubehelix_palette(start=.5, rot=-.5, as_cmap=True),
    how_many=20
)
# 11X11 filters

plot_filters_for_conv_layer(
    model,
    2,
    num_columns=5,
    how_many=10
)
# 5X5 filters

[idx for idx in range(len(model.layers)) if 'conv' in model.layers[idx].name]

#Visualizing feature maps
def plot_feature_maps_for_single_conv_layer(model, layer_id, input_img, num_columns=10, cmap='binary'):
    ref_model = Model(inputs=model.inputs, outputs=model.layers[layer_id].output)
    feature_map = ref_model.predict(input_img)

    num_filters = feature_map[0].shape[2]
    num_rows = (num_filters // num_columns) + (num_filters % num_columns)

    fig = plt.figure(figsize=(16, 2 * num_rows))
    ix = 1
    for _ in range(num_rows):
        for _ in range(num_columns):
            if ix == num_filters:
                break

            # specify subplot and turn of axis
            ax = plt.subplot(num_rows, num_columns, ix)
            ax.set_xticks([])
            ax.set_yticks([])

            # plot filter channel in grayscale
            plt.imshow(feature_map[0, :, :, ix-1], cmap=cmap)
            ix += 1

    # show the figure
    plt.show()

visualize_feature_maps_for = 7

plt.imshow(x_train[visualize_feature_maps_for].reshape((28, 28)), cmap=plt.cm.binary)

plot_feature_maps_for_single_conv_layer(
    model,
    0,
    x_train[visualize_feature_maps_for][np.newaxis, ...],
    num_columns=8,
    cmap=sns.cubehelix_palette(start=.5, rot=-.5, as_cmap=True)
)

plot_feature_maps_for_single_conv_layer(
    model,
    2,
    x_train[visualize_feature_maps_for][np.newaxis, ...],
    num_columns=8,
    cmap=sns.cubehelix_palette(start=.5, rot=-.5, as_cmap=True)
)

#Here model is evaluated on the validation dataset.
#plot_confusion_matrix function prints and plots the confusion matrix. Normalization can be applied by setting normalize=True.
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix'):
    plt.figure(figsize=(8, 8))

    plt.imshow(cm, interpolation='nearest', cmap=sns.cubehelix_palette(start=.5, rot=-.5, as_cmap=True))
    plt.title(title)
    plt.colorbar(fraction=0.046, pad=0.04)
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Predict labels for validation dataset
y_pred = model.predict(x_dev)
```

```
# Convert predictions classes to one hot vectors
y_pred_classes = np.argmax(y_pred, axis=1)

# compute the confusion matrix
confusion_mtx = confusion_matrix(y_dev, y_pred_classes)

# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes=range(10))

# Verify the results
predictions = y_pred

print(predictions[0]) # Confidence matrix

print('Predicted digit is: ' + str(np.argmax(predictions[0])))
print('Accuracy is: ' + str(np.max(predictions[0] * 100)) + '%')

# Actual Digit
plt.imshow(x_dev[0].reshape((28, 28)), cmap=plt.cm.binary)

# Seeing first 25 validation images predictions
plt.figure(figsize=(12, 14))
for i in range(25):
  plt.subplot(5, 5, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.imshow(x_dev[i].reshape((28, 28)), cmap=plt.cm.binary)

  plt.xlabel(
    'Predicted digit is: ' + str(np.argmax(predictions[i])) +
    '\n' +
    'Accuracy is: ' + str(np.max(predictions[i] * 100)) + '%'
  )
```

6. Results:

```
[1.4522489e-07 2.4068950e-05 9.6944468e-06 9.9957985e-01 2.7309335e-08
 3.4366717e-04 4.6738236e-09 3.8871167e-05 6.3689851e-07 3.1082445e-06]
Predicted digit is: 3
Accuracy is: 99.957985%
<matplotlib.image.AxesImage at 0x20a8a2ed790>
```





7. Conclusion:

In conclusion, this project underscores the significance of Convolutional Neural Networks (CNNs) in handwritten digit recognition tasks. By leveraging deep learning techniques and frameworks like TensorFlow and Keras, we have demonstrated the efficacy of CNNs in accurately classifying handwritten digits from the MNIST dataset. Through rigorous experimentation and parameter tuning, we have optimized the network architecture to achieve high accuracy and efficiency in digit recognition. The successful implementation of CNNs showcases their potential for real-world applications requiring automated digit recognition, such as postal sorting, document processing, and digitized form recognition. Moving forward, further research and development in CNN architectures, optimization algorithms, and training methodologies can lead to even greater advancements in digit recognition technology, paving the way for enhanced automation and efficiency in various industries and domains.

8. References:

1. Team, K. (n.d.). Keras documentation: MNIST digits classification dataset. Keras. https://keras.io/api/datasets/mnist/.
2. manav_m. (2021, May 1). CNN for Deep Learning: Convolutional Neural Networks (CNN). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/.
3. YouTube. (2020, October 14). Simple explanation of convolutional neural network | Deep Learning Tutorial 23 (Tensorflow & Python). YouTube. https://www.youtube.com/watch?v=zfiSAzpy9NM&list=PLeo1K3hjS3uvCeTYTeyfe0-rN5r8zn9rw&index=61.

9. Division of work by each member:

1. **Mufti Usman** - Data preprocessing, data modeling and visualizing CNN.
2. **Anurag Sati -** Dataset, convolution layer and correlation matrix.
3. **Paakhi -** Feature maps, plotting filters, prediction of given labels and verifying the results