Technical Report for Assignment - 3

Name: Siva Rama Rohan Sunkarapalli **Email:** ssunkarapalli@crimson.ua.edu

Statement: For this assignment's preparation, I have utilized ChatGPT, a language model created by OpenAI. Within this assignment, the ChatGPT was used for purposes such as brainstorming and grammatical corrections.

1. Introduction

In this assignment, we are using 2-dimensional Convolution Neural Networks (CNNs) to classify images of handwritten digits from 0 to 9 from a well-known dataset called MNIST. The structure of the project is designed to make the CNN model learn to recognize handwritten digits by using filters and kernels settings that yield best performance. The performance metric used to evaluate the model effectiveness in the image classification project is accuracy.

The objective of the project is to achieve maximum accuracy and implement visualization methods to inspect learnable weights in the model at various neural network layers. These insights assisted in optimizing filter and kernel configuration. Filter configuration used in the project is designed to capture edges, curves, and simple textures of the images. Over the multiple layers, we understand that the selected filters assisted in capturing pixel patterns in the images from varying levels of details. They were pivotal in improving the model's performance for the selected architecture.

The structure of the report is designed to discuss data preprocessing, model configuration, cross-validation experiment to find the optimum model hyperparameters, and performance valuation in section 2. In section 3, we discuss results of the train and test datasets by presenting the corresponding accuracy and binary cross entropy loss. In the next section, we visually inspect filters and kernels used in the model architecture. Finally, we present insights from visualization and lessons learned.

2. CNN Implementation

In this section, we aim to discuss data loading and preprocessing methods followed by CNN model design and implementation, train 5-fold cross-validation to find optimum number of epochs to train the model, and performance evaluation in the end.

2.1. Methodology

2.1.1. Dataset Loading and Pre-processing

The MNIST dataset is popular in the data science community and widely used as a benchmark to evaluate model design experiments. Dr. LeCun is the author of the dataset, and it is publicly available. There exist many repositories of the exact dataset and they are also well integrated within the PyTorch and TensorFlow dataset packages in the image format. The challenge with the original data is that it is available in unsigned-byte type and had to use "idx2numpy" package to convert it into a NumPy array where each element represents grayscale pixel value. The MNIST dataset comprises 60,000 images in the train and 10,000 in the test datasets. The images are normalized between 0 and 1 by dividing the original values by 255. The scaling ensures the neural network converges faster. Various authors recommend scaling the pixel between [-1, 1] for faster convergence to the optimal solution. We did not adopt the suggested scaling for simplicity. Corresponding to each image, the authors provided labels from 0-9 digits. The size of the image is 28 x 28 pixels.

The following code snippet shows data loading and converting into NumPy arrays using the "idx2numpy" package.

```
[ ]  # load train and test images and convert into numpy
     # Train
     train_images = idx2numpy.convert_from_file(f'{data_dir}/train-images.idx3-ubyte')
     train_labels = idx2numpy.convert_from_file(f'{data_dir}/train-labels.idx1-ubyte')

     # Test
     test_images = idx2numpy.convert_from_file(f'{data_dir}/t10k-images.idx3-ubyte')
     test_labels = idx2numpy.convert_from_file(f'{data_dir}/t10k-labels.idx1-ubyte')
```

Figure: 1 – Data Loading and converting

We employed a stratified K-fold strategy to investigate optimum epochs to train the model, and preprocessing step - that includes scaling the pixel - was integrated in the pipeline.

The following code inspects sample images in the train and test dataset.

```
# inspect
print('Samples in the train dataset: ', train_images.shape[0])
print('Samples in the test dataset : ', test_images.shape[0])
```

```
Samples in the train dataset:  60000
Samples in the test dataset :  10000
```

Figure: 2 – Sample images in train and test dataset

2.1.2. CNN Model Design and Implementation

In this project, we used TensorFlow framework, and the sequential module exists to design the model architecture. The following code snippet presents the python definition that returns model design to reuse in both cross-validation and final training with entire train dataset.

```
# 3-layer CNN Model
def define_model():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))       # 10-classes
    return model
```

Figure: 3 – layer CNN Model

- **Model Architecture**

The model consists of three 2-dimensional CNN layers with 32, 64, and 64 channels (kernels) respectively, with 3x3 filter size, and ReLU activation. Model parameters from each layer are passed into a maximum pooling layer with a 2x2 filter for sample downsizing. Output from the third layer is flattened and passed into a linear layer with 64 units to produce 10 units. Each of the 10 units corresponds to logits for 10 target classes from 0-9 digits. Output from the final layer is passed to the SoftMax layer to estimate the index of high likelihood digits. The total trainable parameter in the model is 93322.

2.1.3. Training and Validation

We employed a 5-fold StratfiedKFold cross-validation on target labels to estimate optimum hyperparameters. For simplicity, the parameter tuning was limited to the number of epochs. The general principle is to train each fold to estimate epoch where the solution coverage measure with maximum accuracy of the model, then took the average of 5-folds to estimate the optimum number of epochs. We used it to train the model on the entire training dataset. This process enables us to use all the training dataset. The effectiveness of model architecture and tuned hyperparameters are evaluated on the provided test data.

- **Fold Cross-Validation**

The sequence of steps to create a 5-fold CV is shown below. We began with shuffling the indices of 60,000 train images and then used StratrifiedKFold library from Scikit learn package to split into 5-folds while ensuring the distribution of class labels is same across each fold. In the end, the folds are saved in the data repository to use in later, just-in-case.

```
## Split k-fold
# shuffle
df = df.sample(frac = 1, random_state = SEED).reset_index(drop = True)
df['kfold'] = -1

# split into multiple folds
kf = StratifiedKFold(n_splits=5)            # 5-folds

# populate the fold column
y = df['label']
for fold, (train_idx, valid_idx) in enumerate(kf.split(X=df, y=y), start = 1):
    df.loc[valid_idx, 'kfold'] = fold

df.sort_values(by = 'index', inplace = True, ascending = True)

# save the dataset
df.to_csv(f'{data_dir}/train_folds.csv')
df.head()
```

Figure: 4 - Fold Cross-Validation

- **Model Training**

The code snippet shown below is a function with an objective to train the model for a cross-validation fold. The following steps describes the high-level overview:

1. Extract train and validation indices of samples for the fold passed in the function argument.
2. Use the indices to save training and validation images and corresponding class labels.

3. Scaling the pixels of the images between 0 and 1 for the neural network units converges to solution faster.
4. One-hot encode the 10 class labels into nx10 array.
5. Instantiate the model definition presented earlier.
6. Passed Adam optimizer, categorical cross entropy for loss estimation, and accuracy for performance metric into the model compiler function.
7. Create an early stopping criterion to stop the training sequences if the validation loss does not improve for 3 consecutive epochs.
8. Train the model for 20 epochs with the 64-sample batch size.
9. Estimate loss and accuracy score for both train and validation samples at the end of the training and save the corresponding results in a list.

```
### Helper function to normalize and train a fold

def train(fold, results = []):

    ##### STEP 1: Extract fold data
    train_indices = df[df['kfold'] != fold]['index'].values.tolist()
    test_indices = df[df['kfold'] == fold]['index'].values.tolist()

    # Images
    trn_images = train_images[train_indices, :, :]
    tst_images = train_images[test_indices, :, :]

    # Labels
    trn_labels = df.loc[df['index'].isin(train_indices), 'label']
    tst_labels = df.loc[df['index'].isin(test_indices), 'label']

    ##### STEP 2: Normalize image pixels and apply ohe-hot-encoding to the labels
    # Image pixel normalization
    norm_trn_images = trn_images / 255.
    norm_tst_images = tst_images / 255.

    # One-hot-encode labels
    ohe_trn_labels = to_categorical(trn_labels)
    ohe_tst_labels = to_categorical(tst_labels)

    ##### STEP 3: Define model, complier, and earlystopping callback
    # instantiate model
    model = define_model()
    if fold == 1:
        # print model summary
        print(model.summary())

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

```
    # Earlystopping callback
    early_stopping = EarlyStopping(monitor='val_loss',
                                   patience=3,
                                   restore_best_weights=True)

    ##### STEP 4: Train model
    model.fit(norm_trn_images.reshape(-1, 28, 28, 1), ohe_trn_labels,
              validation_data = (norm_tst_images, ohe_tst_labels),
              epochs=20,
              batch_size=64,
              callbacks=[early_stopping])

    ##### STEP 5: Evaluate performance on both train and validation dataset
    train_loss, train_acc = model.evaluate(norm_trn_images, ohe_trn_labels)
    test_loss, test_acc = model.evaluate(norm_tst_images, ohe_tst_labels)
    results.append((fold, train_loss, train_acc, test_loss, test_acc))
    return results
```

Figure: 5 – Model Training

2.1.4. Performance Evaluation

The following table shows the performance of 5-fold cross-validation in terms of accuracy and loss between the model prediction and ground truth labels. Across the folds, the validation accuracy is tightly distributed with mean is 98.92% with standard deviation of 0.13%, while the train accuracy exceeds 99%.

	fold	train_loss	train_acc	valid_loss	valid_acc
0	1	0.015913	0.995229	0.034514	0.989167
1	2	0.021198	0.993229	0.035868	0.988417
2	3	0.013701	0.995938	0.042487	0.989333
3	4	0.024294	0.992708	0.040730	0.987750
4	5	0.004981	0.998667	0.031428	0.991333

Figure: 6 – Performance Evaluation

It was estimated that the best performance on the dataset can be achieved by training the model for 9 epochs. We can finetune other hyperparameters including batch size, optimizer, and number of 2D CNN layers in the model architecture along with filter and kernel size, but not considered for simplicity.

2.2 Results

2.2.1 Test Accuracy and Loss

With the optimum model architectures and hyperparameters established in the cross-validation stage, we trained the final model with all the 60,000 samples provided in the train dataset. Upon training, the sample model is used to make predictions of sample images provided in the test dataset. The accuracy and loss for the train and test datasets is shown below.

Train accuracy 99.72%
Train loss : 0.0081
Test accuracy : 99.18%
Test loss : 0.0264

Compared to the mean of the 5-fold cross-validation the accuracy of the test sample unseen by the model increased marginally. The performance improvement is attributed to the choice of selecting all samples in the train dataset for model training.

3. Visualization

3.1. Visualization of Filters and Kernels

The following plot_filters function takes weights from the convolution layer and size of the kernel.

```
## Helper function for visualization
def plot_filters(conv1_weights, kernel_size):
    plt.figure(figsize=(10, 6))
    for i in range(kernel_size):
        ax = plt.subplot(4, 8, i + 1)
        filter_weights = conv1_weights[:, :, :, i]
        plt.imshow(filter_weights[:, :, 0], cmap='viridis')
        plt.axis('off')
        ax.set_title(f'filter {i}')
    plt.tight_layout()
    plt.show()

## Load initial layers
conv1_weights = model.layers[0].get_weights()[0]
plot_filters(conv1_weights, kernel_size = 32)
```

Figure: 7.1 – Visualization of Filters and Kernels

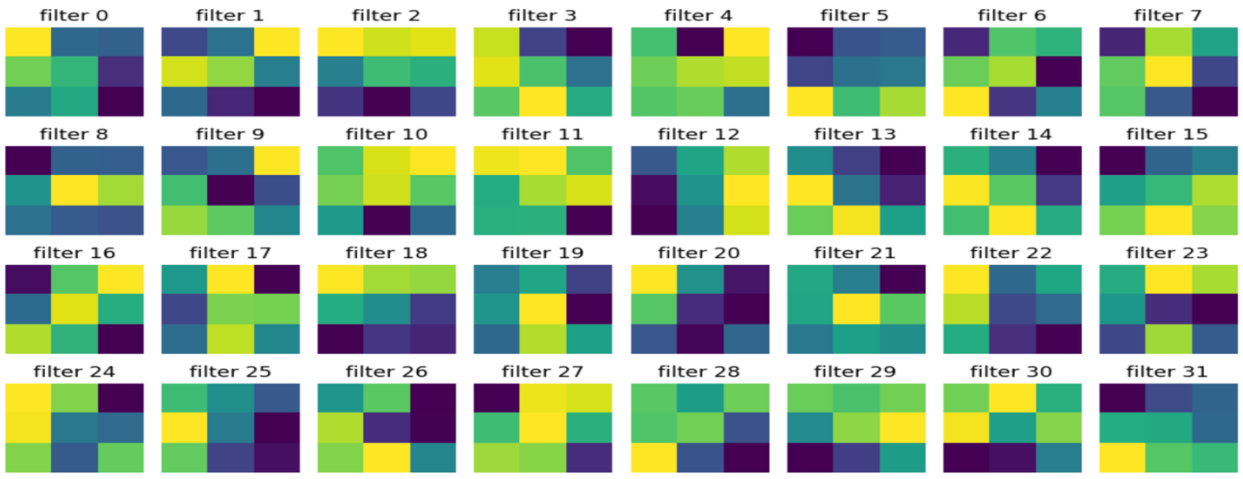

Figure: 7.2 – Visualization of Filters and Kernels

3.2. Feature Map Visualization

In the present model architecture, each of the 2D-CNN layers extract features with assistance for a filter map (sample shown above.) The following code is used to visualize the weights of each of the three layers.

```
from tensorflow.keras.models import Model

# Choose a sample input image for visualization (you can use any image from the test set)
sample_image = test_images[0]

# Create a model to extract feature maps from intermediate layers
layer_outputs = [layer.output for layer in model.layers[:5]]
activation_model = Model(inputs=model.input, outputs=layer_outputs)

# Get the feature maps for the sample image
activations = activation_model.predict(np.expand_dims(sample_image, axis=0))

# Visualization of feature maps
layer_names = [layer.name for layer in model.layers[:5]]

for layer_name, layer_activation in zip(layer_names, activations):
    print('')
    print(f'Layer name: {layer_name}')
    n_features = layer_activation.shape[-1]
    size = layer_activation.shape[1]

    n_cols = n_features // 4
    n_rows = (n_features + n_cols - 1) // n_cols

    plt.figure(figsize=(2 * n_cols, 2 * n_rows))
    for i in range(n_features):
        plt.subplot(n_rows, n_cols, i + 1)
        plt.imshow(layer_activation[0, :, :, i], cmap='viridis')
        plt.title(f'feature {i}')
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

Figure: 8.1 – Feature Map Visualization

Top CNN layer

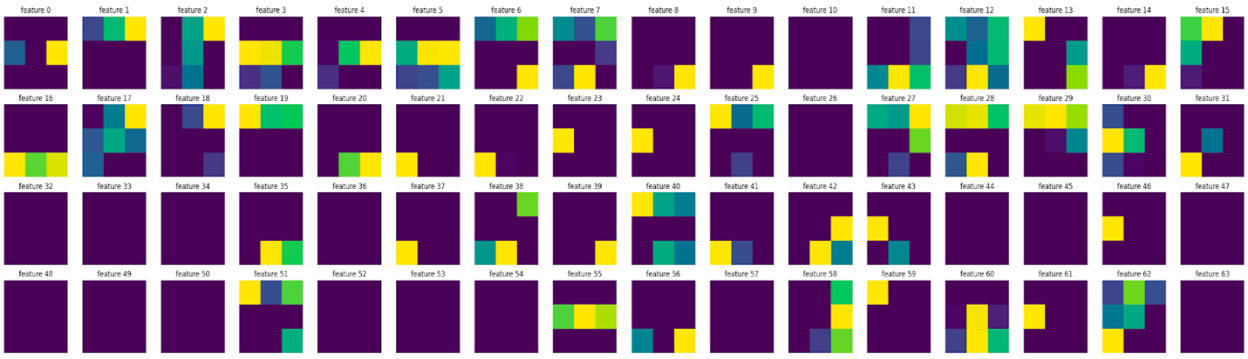

Figure: 8.2 – Feature Map Visualization

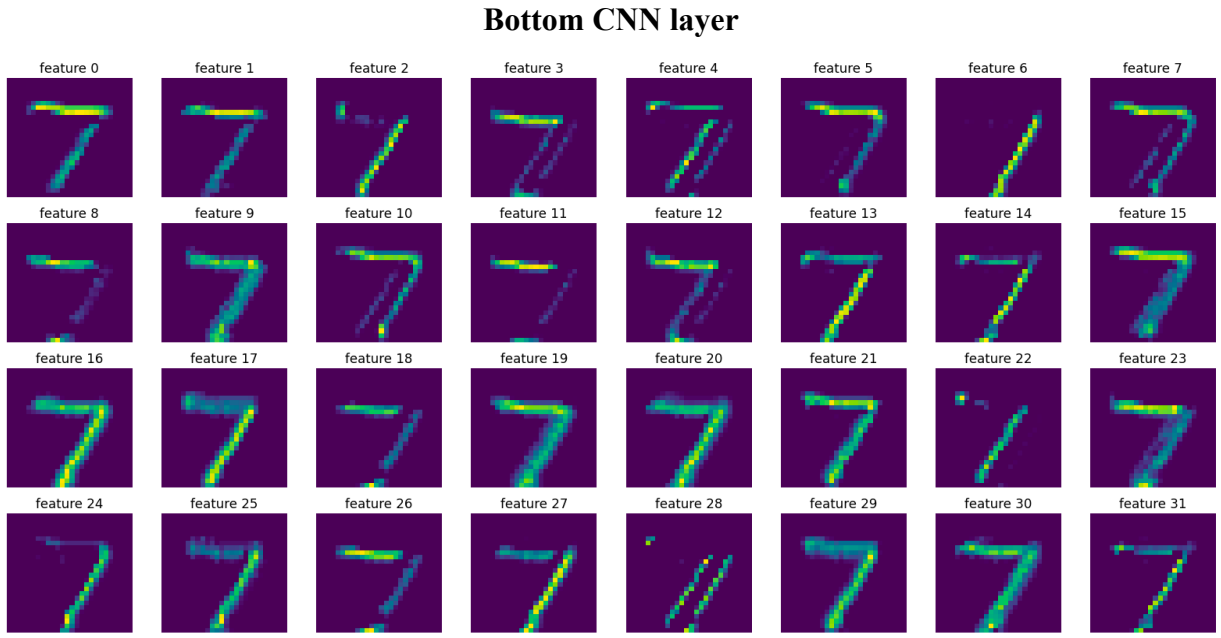

Figure: 8.3 – Feature Map Visualization

4. Discussion

4.1. Insights from Visualization

From the above filter images, we can infer that each filter is designed to capture edges, curves, and simple textures from the images.

- The filters highlighting the edges are trying to capture orientation of the edges such as horizontal, vertical, and diagonal (and the corresponding direction).
- The corner filter tries to identify regions where edges meet.
- These filters also try to find patterns in the input images such as dots, dashes, and any repetitions.
- These filters also try to find boundaries and shapes of the digits in the input images.

Feature maps of the 1st and 3rd CNN layers are shown above. We can observe that the top layer where input image pixels passed captured edges, curves, and simple textures. While the bottom CNN layer captures high level structure of the image, where the image of digit 7 can be identified with no effort.

4.2. Implementations and Future Work

In the project, we defined the model architecture and trained the model for several iterations because the weights are randomly initiated. We experimented with the training with GPU support on Google Collab, and yet the entire process took over 30 minutes including cross-validation and final model training. Although we did not try to train the pipeline on CPU, it is uncommon to see significant process time increase in comparisons.

In the future work, we can streamline the process utilizing pretrained model architectures such as ResNet which freezes all the convolution layers and trains only the top feedforward layer. The performance of this setup had results with over 95% accuracy in the literature.

In the future, we would also consider turning much larger hyperparameters including CNN layers in model architecture along with kernel and filter sizes.

5. Lessons Learned

In the project, we learned the importance of cross-validation and its application in hyperparameters. We also noticed that training the final model with all the images in the training dataset resulted in considerable improvement to the final performance. It is largely because the final model observed more samples and consequently was diverse in the patterns.

Visualizing the kernel and feature maps assisted in our understanding of how CNN layers process the images and pass information between each successive layer.

6. References

Dataset. Retrieved from: http://yann.lecun.com/exdb/mnist/