EIE4122 Deep Learning and Deep Neural Networks

Lab 1: CNNs for Handwritten Digit Classification

HAO Jiadong (20084595d)

1. CNN Understanding

Convolutional Neural Networks are a type of deep learning architecture commonly used for various computer vision tasks such as image recognition.

From the name, it is evident that the convolution layers are the most critical component of CNNs. Convolution is a mathematical operation that involves sliding a small window (filter) over the input image. The filter will generate feature maps by performing element-wise multiplication and then summation with the corresponding pixels in the input. From the lecture, I know that in practice, convolution is normally implemented by cross-correlation instead, and the number of feature maps is equivalent to the number of filters used.

To capture different levels of abstraction, CNNs typically have multiple convolutional layers stacked one after another. The network may learn some very simple features like corners and edges in the previous convolutional layers while much more complex features like the shape of the objects in the very last layers.

Pooling layers are often inserted after convolutional layers. Pooling is a downsampling process applied to reduce the dimensions of the feature maps while retaining the most essential information. There are many types of pooing operations, such as max pooling, average pooling, L2 pooling, etc.

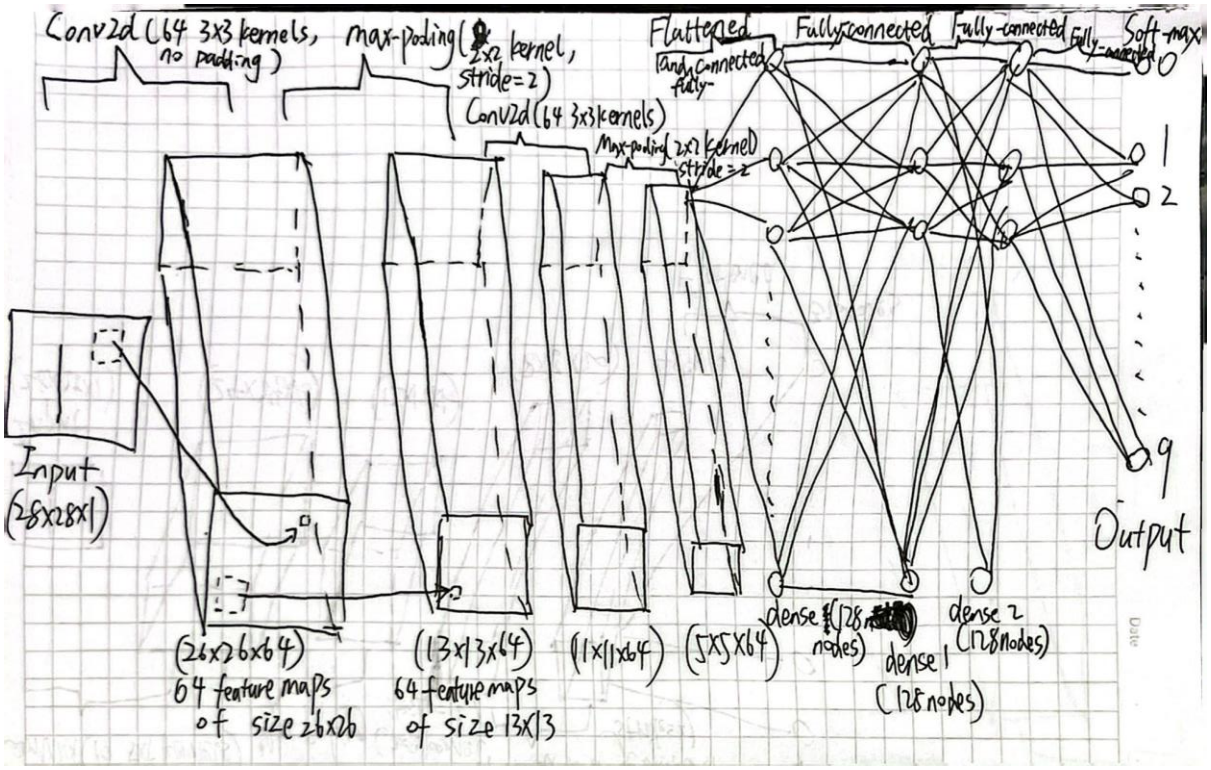
After several convolutional and pooling layers, the resulting feature maps are flattened into a one-dimensional vector (vectorization). This vector is then fed into fully connected layers (dense layers) to take in the features generated by the previous convolutional layers and make predictions based on the learned representations. If the task is classification, the output nodes of the dense layers will often be applied with a Softmax function to convert the raw output into a probability distribution of different classes.

Similar to traditional neural networks, CNN also adopts backpropagation to update its parameters in the direction to minimize the loss. The iterative forward propagation and backpropagation process continues until the model reaches convergence.

Overall, CNNs leverage the power of convolutions, pooling, fully connected layers, and backpropagation algorithms to automatically extract relevant features from images, enabling CNNs to achieve impressive performance in various computer vision tasks, including image classification, object detection, and image segmentation.

2. Explanation of the architecture of the two models built in Keras and PyTorch

For the CNN model in Keras:



Explanation of Model Architecture:

Initially, the input image is 28*28*1; after the first convolution layer (64 3*3 kernels), 64 feature maps are generated. Since no padding is by default, the size of each feature map is reduced by 2, becoming 26*26. Then, a max-pooling layer is applied with a filter size of 2*2. Since the stride is equal to the filter size by default (stride = 2), each feature map is sub-sampled horizontally and vertically by 2, hence, the size is reduced to 13*13.

The subsequent convolution layer and max pooling layer have the same architecture. Each of the 64 filters in the convolution layer goes through all 64 feature maps from the previous layer and sums the results at corresponding positions to generate one new feature map. So, the number of newly generated feature maps is 64. Then a max pooling layer is to further reduce the dimension of each feature map to 5*5.

Then, all the feature maps are flattened into a 1D vector and fully connected to 3 dense layers with 128 nodes each for the classification task.

Finally, the third dense layer is fully connected to the output layer with 10 nodes representing the number of 0-9. A Softmax function is applied to the output layer to gain the probability distribution of each number.

Explanation of the Number of Parameters for Each Layer:

First Conv2d：640 parameters

```
conv2d (Conv2D)              (None, 26, 26, 64)        640
```

Kernel size: 3*3, no. of kernels = 64, no. of bias = 64 (one for each kernel)
Total number of parameters = 3*3*64 + 64 = 640

Second Conv2d : 36928 parameters

```
conv2d_1 (Conv2D)            (None, 11, 11, 64)        36928
```

Kernel size: 3*3, no. of kernels = 64, no. of bias = 64 (one for each kernel)
For each kernel, since the previous layer has 64 feature maps, there are 64*3*3 = 576 parameters
There are 64 kernels in total, hence the total number of parameters = 576*64 + 64 (bias) = 36928

Number of inputs to the dense layer: 1600

```
max_pooling2d_1 (MaxPoolin   (None, 5, 5, 64)          0
g2D)
```

```
flatten (Flatten)            (None, 1600)              0
```

The number of inputs to the dense layer equals the values in the last max_pooling layer = 5*5*64 = 1600

First Dense Layer: 204928 parameters

```
dense (Dense)                (None, 128)               204928
```

Total number of parameters = 1600 * 128 (weights) + 128 (bias) = 204928

Second Dense Layer: 16512 parameters

```
dense_1 (Dense)              (None, 128)               16512
```

Total number of parameters = 128 * 128 (weights) + 128 (bias) = 16512

Third Dense Layer: 16512 parameters

```
dense_2 (Dense)              (None, 128)               16512
```

Total number of parameters = 128 * 128 (weights) + 128 (bias) = 16512

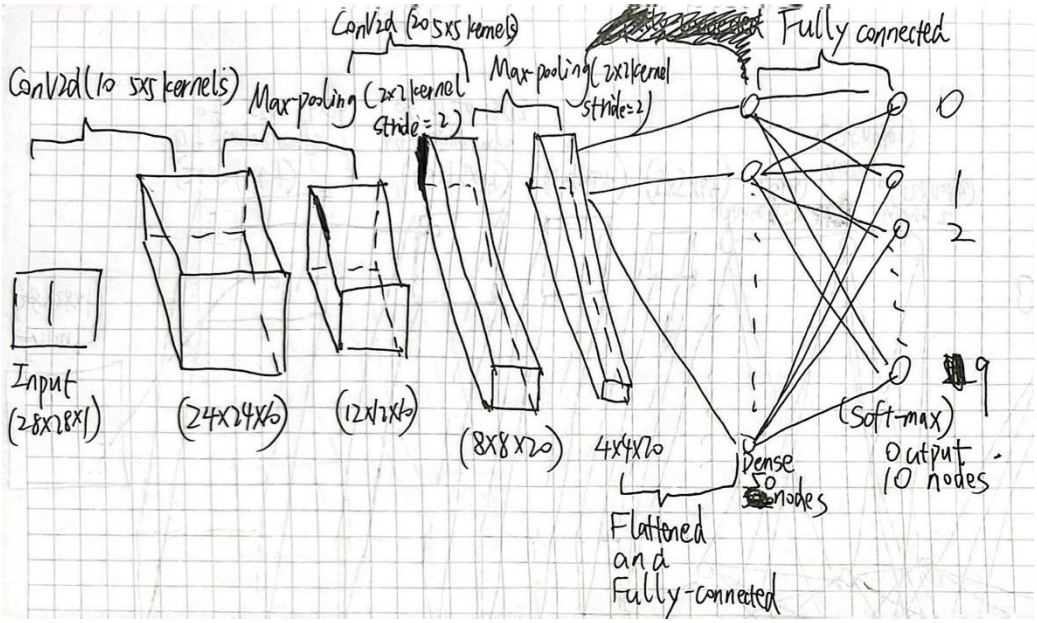
Fourth Dense Layer: 1290 parameters

```
dense_3 (Dense)              (None, 10)                1290
```

Total number of parameters = 128 * 10 (weights) + 10 (bias) = 1290

Insight: From the calculation, the number of parameters for the First Dense Layer (204928) is far more than the total number of parameters in the previous convolutional layer. Hence, in a CNN, most parameters are concentrated in the fully connected layers, which consumes most of the computational power. In other word, convolution layers are not fully connected, and the magic of CNN in dealing with massive image data is that the convolution filter help reduce the calculations significantly.

For the CNN model in Pytorch:



Explanation of Model Architecture:

Initially, the input image is 28*28*1, after the first convolution layer (10 5*5 kernels), 10 feature maps are generated. Since no padding is by default, the size of each feature map is reduced by 4, becoming 24*24. Then, a max-pooling layer is applied with a filter size of 2*2. Since the stride is equal to the filter size by default (stride = 2), each feature map is sub-sampled horizontally and vertically by 2, hence, the size is reduced to 12*12.

The subsequent convolution layer has 20 filters of size 5*5, hence generating 20 feature maps with size reduced to 8*8. Then, a max-pooling with a filter size of 2*2 further reduces the size of each feature map to 4*4.

After that, all the feature maps are flattened into a 1D vector of dimension 20*4*4 = 320 and fully connected to a dense layers with 50 nodes each to do the classification task.

Finally, the dense layer is fully connected to the output layer with 10 nodes representing the number of 0-9. A Softmax function is applied to the output layer to gain the probability distribution of each number.

3. Modification to the CNN model in Keras and Insights

i. Number of Convolution layers and Max-pooling layers

Modified Model: 1 convolution layer with a max-pooling layer

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```
Epoch 1/5
469/469 [==============================] - 89s 187ms/step - loss: 0.9490 - accuracy: 0.6999
Epoch 2/5
469/469 [==============================] - 87s 185ms/step - loss: 0.4211 - accuracy: 0.8757
Epoch 3/5
469/469 [==============================] - 85s 182ms/step - loss: 0.3052 - accuracy: 0.9125
Epoch 4/5
469/469 [==============================] - 86s 183ms/step - loss: 0.2433 - accuracy: 0.9308
Epoch 5/5
469/469 [==============================] - 83s 177ms/step - loss: 0.2086 - accuracy: 0.9417
Saving CNN to models/mnist_cnn_one_less_Convolutional_Layer.keras
```

```
Test loss: 0.07501036673784256
Test accuracy: 97.62%
```

Original Model: 2 convolution layers, each with a max-pooling layer

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
Epoch 1/5
469/469 [==============================] - 127s 266ms/step - loss: 1.3014 - accuracy: 0.5865
Epoch 2/5
469/469 [==============================] - 125s 266ms/step - loss: 0.5440 - accuracy: 0.8286
Epoch 3/5
469/469 [==============================] - 141s 301ms/step - loss: 0.3856 - accuracy: 0.8834
Epoch 4/5
469/469 [==============================] - 137s 293ms/step - loss: 0.3098 - accuracy: 0.9083
Epoch 5/5
469/469 [==============================] - 137s 291ms/step - loss: 0.2625 - accuracy: 0.9227
Saving CNN to models/mnist_cnn.keras
```

```
Test loss: 0.08607502281665802
Test accuracy: 97.39%
```

Modified Model: 3 convolution layers, each with a max-pooling layer

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Add one more convolutional layer
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```
Epoch 1/5
469/469 [==============================] - 131s 275ms/step - loss: 1.8171 - accuracy: 0.4208
Epoch 2/5
469/469 [==============================] - 127s 270ms/step - loss: 0.8732 - accuracy: 0.7084
Epoch 3/5
469/469 [==============================] - 129s 275ms/step - loss: 0.6174 - accuracy: 0.8077
Epoch 4/5
469/469 [==============================] - 130s 276ms/step - loss: 0.4899 - accuracy: 0.8530
Epoch 5/5
469/469 [==============================] - 127s 271ms/step - loss: 0.4071 - accuracy: 0.8804
Saving CNN to models/mnist_cnn_one_more_Convolutional_Layer.keras
```

```
Test loss: 0.1335838884115219
Test accuracy: 95.84%
```

Observations and insights:
The results show that as the number of the convolution layer and max-pooling layer increases, the performance drops (accuracy from 97.62% to 97.39%, further to 95.84%). It may indicate that the input data is so simple that one convolution layer and max-pooling layer are enough to deal with it. More convolution and max-pooling layers may lead to losing some important features because of a higher-level representation of the data.

ii. Activation functions

Original Model: Relu

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
Epoch 1/5
469/469 [==============================] - 127s 266ms/step - loss: 1.3014 - accuracy: 0.5865
Epoch 2/5
469/469 [==============================] - 125s 266ms/step - loss: 0.5440 - accuracy: 0.8286
Epoch 3/5
469/469 [==============================] - 141s 301ms/step - loss: 0.3856 - accuracy: 0.8834
Epoch 4/5
469/469 [==============================] - 137s 293ms/step - loss: 0.3098 - accuracy: 0.9083
Epoch 5/5
469/469 [==============================] - 137s 291ms/step - loss: 0.2625 - accuracy: 0.9227
Saving CNN to models/mnist_cnn.keras
```

```
Test loss: 0.08607502281665802
Test accuracy: 97.39%
```

Revised Model: Sigmoid

```
model.add(Conv2D(64, kernel_size=(3, 3), activation='sigmoid', input_shape=input_shape))
# model.add(Conv2D(64, kernel_size=(3, 3), activation='tanh', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='sigmoid'))
# model.add(Conv2D(64, (3, 3), activation='tanh'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# # Add one more convolutional layer
# model.add(Conv2D(64, (3, 3), activation='relu'))
# model.add(BatchNormalization())
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))

model.add(Flatten())

for i in range(0,3):
    model.add(Dense(128, activation='sigmoid'))
```

```
Epoch 1/5
469/469 [==============================] - 140s 295ms/step - loss: 1.1678 - accuracy: 0.6267
Epoch 2/5
469/469 [==============================] - 147s 314ms/step - loss: 0.5426 - accuracy: 0.8304
Epoch 3/5
469/469 [==============================] - 151s 321ms/step - loss: 0.3996 - accuracy: 0.8757
Epoch 4/5
469/469 [==============================] - 151s 321ms/step - loss: 0.3245 - accuracy: 0.9021
Epoch 5/5
469/469 [==============================] - 149s 318ms/step - loss: 0.2831 - accuracy: 0.9147
Saving CNN to models/mnist_cnn_Sigmoid.keras
```

```
Test loss: 0.1089678555727005
Test accuracy: 96.64%
```

Revised Model: Tanh

```
model.add(Conv2D(64, kernel_size=(3, 3), activation='tanh', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))


'''Kernel size of the convolution layer'''
# model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape))
# model.add(Conv2D(64, kernel_size=(2, 2), activation='relu', input_shape=input_shape))


'''activation function'''
# model.add(Conv2D(64, (3, 3), activation='relu'))
# model.add(Conv2D(64, (3, 3), activation='sigmoid'))
model.add(Conv2D(64, (3, 3), activation='tanh'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
for i in range(0,3):
    # model.add(Dense(128, activation='relu'))
    # model.add(Dense(128, activation='sigmoid'))
    model.add(Dense(128, activation='tanh'))
```

```
Epoch 1/5
469/469 [==============================] - 145s 306ms/step - loss: 1.3723 - accuracy: 0.5695
Epoch 2/5
469/469 [==============================] - 144s 306ms/step - loss: 0.6253 - accuracy: 0.8023
Epoch 3/5
469/469 [==============================] - 142s 302ms/step - loss: 0.4412 - accuracy: 0.8646
Epoch 4/5
469/469 [==============================] - 142s 304ms/step - loss: 0.3483 - accuracy: 0.8981
Epoch 5/5
469/469 [==============================] - 142s 303ms/step - loss: 0.2973 - accuracy: 0.9141
Saving CNN to models/mnist_cnn_Tanh.keras
```

```
Test loss: 0.12299702316522598
Test accuracy: 96.83%
```

Observations and insights:

From the results, the Relu activation (97.39%) outstands the other two activation functions, which are Sigmoid (96.64%) and Tanh (96.83%), verifying its advantage in improving the model performance. This may related to Relu's constant gradient, which can avoid the vanishing gradient problem.

iii. Size of the filter in the convolution layers

Revised Model: 2*2 kernel

```
'''Kernel size of the convolution layer'''
# model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, kernel_size=(2, 2), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

'''activation function'''
# model.add(Conv2D(64, (3, 3), activation='relu'))
# model.add(Conv2D(64, (3, 3), activation='sigmoid'))
# model.add(Conv2D(64, (3, 3), activation='tanh'))

'''Kernel size of the convolution layer'''
# model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, kernel_size=(2, 2), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```
Epoch 1/5
469/469 [==============================] - 130s 273ms/step - loss: 1.3698 - accuracy: 0.5672
Epoch 2/5
469/469 [==============================] - 120s 256ms/step - loss: 0.5780 - accuracy: 0.8191
Epoch 3/5
469/469 [==============================] - 120s 257ms/step - loss: 0.4130 - accuracy: 0.8770
Epoch 4/5
469/469 [==============================] - 121s 257ms/step - loss: 0.3343 - accuracy: 0.9020
Epoch 5/5
469/469 [==============================] - 121s 259ms/step - loss: 0.2892 - accuracy: 0.9156
Saving CNN to models/mnist_cnn_22kernel.keras
```

```
Test loss: 0.09933483600616455
Test accuracy: 96.90%
```

Original Model: 3*3 kernel

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
Epoch 1/5
469/469 [==============================] - 127s 266ms/step - loss: 1.3014 - accuracy: 0.5865
Epoch 2/5
469/469 [==============================] - 125s 266ms/step - loss: 0.5440 - accuracy: 0.8286
Epoch 3/5
469/469 [==============================] - 141s 301ms/step - loss: 0.3856 - accuracy: 0.8834
Epoch 4/5
469/469 [==============================] - 137s 293ms/step - loss: 0.3098 - accuracy: 0.9083
Epoch 5/5
469/469 [==============================] - 137s 291ms/step - loss: 0.2625 - accuracy: 0.9227
Saving CNN to models/mnist_cnn.keras
```

```
Test loss: 0.08607502281665802
Test accuracy: 97.39%
```

Revised Model: 4*4 kernel

```
'''Kernel size of the convolution layer'''
model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape))
# model.add(Conv2D(64, kernel_size=(2, 2), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

'''activation function'''
# model.add(Conv2D(64, (3, 3), activation='relu'))
# model.add(Conv2D(64, (3, 3), activation='sigmoid'))
# model.add(Conv2D(64, (3, 3), activation='tanh'))

'''Kernel size of the convolution layer'''
model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape))
# model.add(Conv2D(64, kernel_size=(2, 2), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```
Epoch 1/5
469/469 [==============================] - 154s 326ms/step - loss: 1.3817 - accuracy: 0.5624
Epoch 2/5
469/469 [==============================] - 139s 297ms/step - loss: 0.5389 - accuracy: 0.8328
Epoch 3/5
469/469 [==============================] - 137s 292ms/step - loss: 0.3724 - accuracy: 0.8885
Epoch 4/5
469/469 [==============================] - 136s 290ms/step - loss: 0.2952 - accuracy: 0.9132
Epoch 5/5
469/469 [==============================] - 137s 292ms/step - loss: 0.2511 - accuracy: 0.9276
Saving CNN to models/mnist_cnn_44kernel.keras
```

```
Test loss: 0.07901279628276825
Test accuracy: 97.55%
```

Observations and insights:

From the results, as the size of the kernels in the convolution layers increases from 2*2 to 4*4, the test accuracy also increases from 96.90% to 97.55%. That may be because a larger receptive field can capture more contextual information from the image, hence, some more complex patterns or features may be identified. However, in practice, the best size of the receptive field depends on the specific task and dataset. It is important to experiment with different sizes of the convolution filter to identify the one with the best performance.

4. Modification to the CNN model in Pytorch and Insights

i. Number of kernels in each convolution layer

Revised Model: 5 kernels in the first convolution layer, 10 in the second

```
#number of kernels shrink to 5 and 10 respectively
self.conv1 = nn.Conv2d(1, 5, kernel_size=5)
self.conv2 = nn.Conv2d(5, 10, kernel_size=5)
self.conv2_drop = nn.Dropout2d()
self.fc1 = nn.Linear(160, 50) # generate 10 feature maps, each size is 4*4
self.fc2 = nn.Linear(50, 10)
```

```
Test set: Average loss: 0.1738, Accuracy: 9461/10000
```

Original Model: 10 kernels in the first convolution layer, 20 in the second

```
# the model building blocks are defined below
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5

# randomly drop out 50% of the neurons
self.conv2_drop = nn.Dropout2d() # The default dropout rate is 50%

self.fc1 = nn.Linear(320, 50) # number of input neurons: 320, number of output neurons: 50
# 50 neurons fully connected layer.

self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
# 10 neurons to represent our 10 classes.
```

```
Test set: Average loss: 0.1062, Accuracy: 9652/10000
```

Revised Model: 20 kernels in the first convolution layer, 40 in the second

```
#number of kernels expand to 20 and 40 respectivel
self.conv1 = nn.Conv2d(1, 20, kernel_size=5)
self.conv2 = nn.Conv2d(20, 40, kernel_size=5)
self.conv2_drop = nn.Dropout2d()
self.fc1 = nn.Linear(640, 50) # generate 40 feature maps, each size is 4*4
self.fc2 = nn.Linear(50, 10)
```

```
Test set: Average loss: 0.0808, Accuracy: 9747/10000
```

Observations and insights:

From the results, as the number of kernels increases in the convolution layer, the model's performance also increases (the accuracy increases from 94.61% to 97.47%). Increasing the number of kernels means increasing the model capacity. The model with more convolution filters can capture more features that can be potentially helpful to the classification task. Hence, the performance of the models increases as the number of kernels increases.

ii. Number of hidden layers in the fully-connected network

Original Model: 1 hidden layer with 50 nodes

```
# the model building blocks are defined below
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5

# randomly drop out 50% of the neurons
self.conv2_drop = nn.Dropout2d() # The default dropout rate is 50%

self.fc1 = nn.Linear(320, 50) # number of input neurons: 320, number of output neurons: 50
# 50 neurons fully connected layer.

self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
# 10 neurons to represent our 10 classes.
```

```
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.335452

Test set: Average loss: 0.1062, Accuracy: 9652/10000
```

Modified Model: 2 hidden layers with 50 nodes each

```
# Number of hidden layers = 2, each layer have 50 nodes
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel:
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel:
# randomly drop out 50% of the neurons
self.conv2_drop = nn.Dropout2d() # The default dropout rate is
self.fc1 = nn.Linear(320, 50) # number of input neurons: 320,
# 50 neurons fully connected layer.
self.fc2 = nn.Linear(50, 50)
self.fc3 = nn.Linear(50, 10)
# 10 neurons to represent our 10 classes.
```

```
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.409205

Test set: Average loss: 0.1650, Accuracy: 9545/10000
```

Modified Model: 3 hidden layers with 50 nodes each

```
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) #
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) #
# randomly drop out 50% of the neurons
self.conv2_drop = nn.Dropout2d() # The default
self.fc1 = nn.Linear(320, 50) # number of inp
# 50 neurons fully connected layer.
self.fc2 = nn.Linear(50, 50)
self.fc3 = nn.Linear(50, 50)
self.fc4 = nn.Linear(50, 10)
# 10 neurons to represent our 10 classes.
```

```
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.920303

Test set: Average loss: 0.4058, Accuracy: 9214/10000
```

Observations and insights:
From the results, as the number of hidden layers increases, the training loss and the test loss both increase as the number of hidden layers increases. Firstly, the problem could not be overfitting because the training loss also increases as the model complexity increases. One possible explanation is that the training data is not sufficient for such complex models to take in. More dense layers mean a higher capacity to learn. If the dataset is not diverse and large enough, the complex dense layers may not be able to learn the underlying patterns of the data effectively.

iii. Number of nodes in the fully-connected network

Modified Model: 20 nodes in the hidden layer

```
# Number of nodes in the hidden layer is reduced to 20
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel:
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel:
self.conv2_drop = nn.Dropout2d() # The default dropout rate i
self.fc1 = nn.Linear(320, 20) # number of input neurons: 320
# 50 neurons fully connected layer.
self.fc2 = nn.Linear(20, 10) # Number of input neurons: 50,
# 10 neurons to represent our 10 classes.
```

```
Test set: Average loss: 0.1425, Accuracy: 9633/10000
```

Original Model: 50 nodes in the hidden layer

```
# the model building blocks are defined below
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5

# randomly drop out 50% of the neurons
self.conv2_drop = nn.Dropout2d() # The default dropout rate is 50%

self.fc1 = nn.Linear(320, 50) # number of input neurons: 320, number of output neurons: 50
# 50 neurons fully connected layer.

self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
# 10 neurons to represent our 10 classes.
```

```
Test set: Average loss: 0.1062, Accuracy: 9652/10000
```

Modified Model: 80 nodes in the hidden layer

```
# Number of nodes in the hidden layer is increased to 80
self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1,
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 1
self.conv2_drop = nn.Dropout2d() # The default dropout rate is
self.fc1 = nn.Linear(320, 80) # number of input neurons: 320,
# 50 neurons fully connected layer.
self.fc2 = nn.Linear(80, 10) # Number of input neurons: 50, Nu
# 10 neurons to represent our 10 classes.
```

```
Test set: Average loss: 0.0975, Accuracy: 9707/10000
```

Observations and insights:
From the results, as the number of nodes in the hidden layers increases from 20 to 80, the test accuracy of the model increases from 96.33% to 97.07%. That may be because one hidden layer with 20 nodes is too simple to deal with so many feature maps provided by the previous convolution layers. As the number of nodes in the hidden layer increases, the capacity of the fully-connected network gets stronger, fitting better for the complex features provided by the convolution layers.

5. Conclusion

During this lab, I gained valuable hands-on experience implementing and fine-tuning two CNN networks using PyTorch and Keras. This exercise allowed me to become familiar with two prominent deep-learning frameworks and learn how to leverage the power of Google Colab for efficient experimentation. Through writing the code and analyzing the network architectures, I developed a deeper understanding of CNNs and their underlying principles. Furthermore, fine-tuning the models and analyzing the tuning results provided valuable insights into optimizing CNN architectures for improved performance. Overall, this experience has been incredibly meaningful and has greatly enriched my understanding of CNNs and deep learning as a whole.