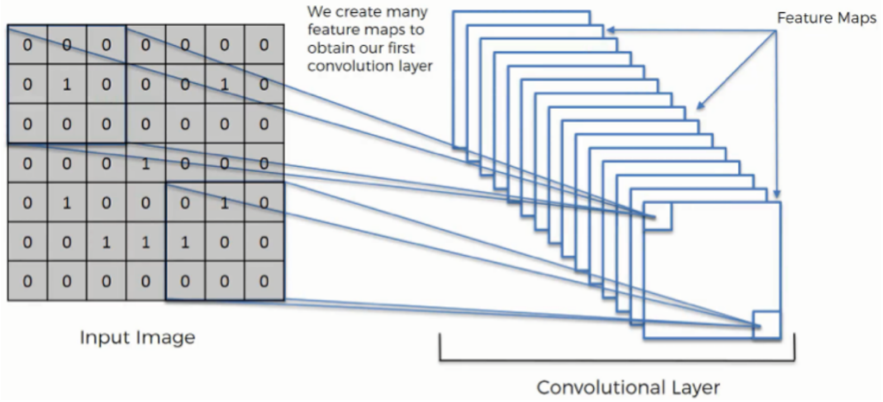
CNNs for Handwritten Digit Classification

Objectives and Outcomes

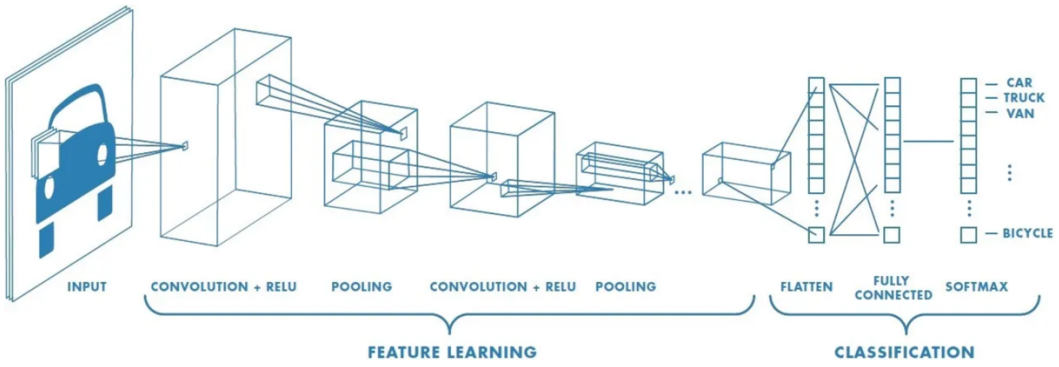
Use Keras and PyTorch to develop the convolutional neural networks for handwritten digit classification. This means the training and testing pattern classification systems based on Keras and PyTorch to modify the code and obtain meaningful results (training accuracy and loss). For example, change the number of layers, neurons, activation functions, kernel size, and filter sizes to track how it impacts the model's performance.

CNN Introduction

The Convolutional Neural Network (CNN) is a deep learning model architecture designed to process and analyze images. The CNNs can extract hierarchical features from input images using convolutional filters, which generate feature maps in the next layers. By doing this, CNNs can effectively classify the images.



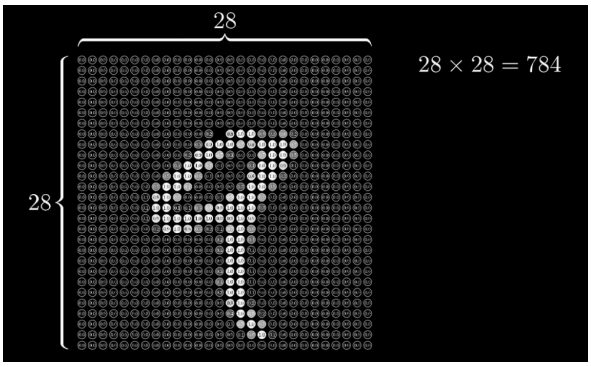
Structure of CNN



The CNN architecture structure consists of several layers, such as the input layer, convolutional layers (Including activation function), pooling layers, a fully connected layer, and an output layer.

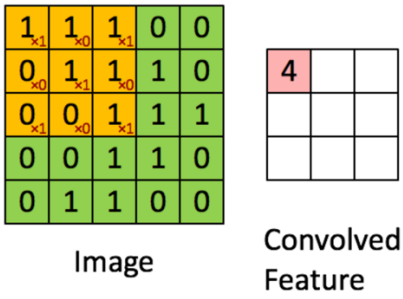
(1) Input Layer:

Receiving the raw input data (e.g., image), which includes dimensions and channels (e.g., 28 x 28 x 1). The figure 3 is an example.



(2) Convolutional Layers + Activation Function:

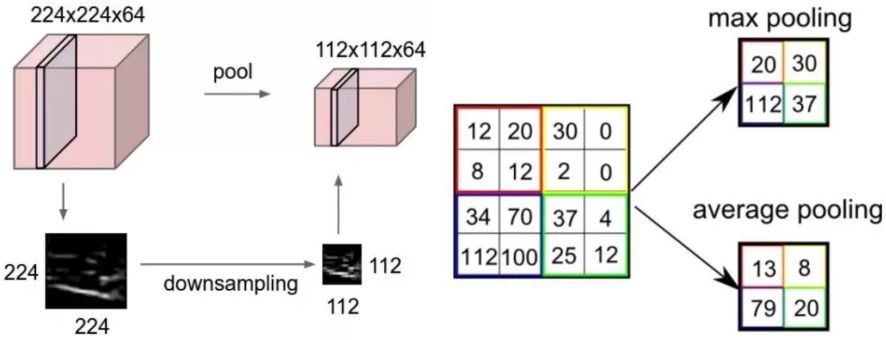
Filters (kernels) slide over the input data to extract features like edges, textures, or patterns. Each filter detects different features in the image. Each convolution operation will apply an activation function (e.g., ReLU, sigmoid, tanh) to introduce non-linearity, allowing it to learn more complex patterns and help mitigate the vanishing gradient problem.



Convoluting 5x5x1 image with 3x3x1 kernel to get 3x3x1 convolved feature (Feature map)

(3) Pooling Layers:

Reducing the spatial dimensions of feature maps by taking the max or average to decrease computational load and controlling overfitting.



(4) Fully Connected Layers:

Before passing data into the fully connected layer, the output of the previous layers (From the convolution and pooling layers 2D or 3D feature map) is flattened into a 1D vector. This flattening converts the multidimensional output into a vector so that the fully connected layer can use it. The fully connected layers connect every neuron in one layer to every neuron in the next layer. It takes the high-level features extracted by the convolutional and pooling layers and uses them to make predictions or classifications.

(5) Output Layer:

This final layer gives the prediction or classification result. The activation function (e.g., Softmax) is applied to the output of this layer to convert the raw output scores into probabilities.

Methodology

MNIST dataset:
Image size: 28 (Height) x 28 (Width)
Channel: 1
Dimension: 28 (Height) x 28 (Width) x 1 (Channel)
Pixel: 28 (Height) x 28 (Width) = 784 pixels

(a) Keras Part

mnist_cnn_keras.py initial code (Unchanged):

```
# CNN structure definition
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))
model.add(Flatten())
for i in range(0,3):
    model.add(Dense(128, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

1. Model Initialization

```
model = Sequential()
```

It creates a linear stack of layers, meaning each layer is added one after the other.

2. First Convolutional Layer

```
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
```

The convolutional layer is to extract the image features.

64: It is convolution kernels number.

kernel_size=(3,3): Convolution kernels size is 3 x 3 pixels (9 pixels).

activation= 'relu': The activation function uses ReLU, which can set negative outputs to 0 and leave positive values unchanged.

input_shape=input_shape: The Input_shape is (28 (Height) x 28 (Width) x 1 (Channel)).

Padding size: 0 (Valid).

Stride size: (1,1) (Default).

Calculate the output of first convolution:

$$n_{in}: \text{number of input features}$$
$$n_{out}: \text{number of output features}$$
$$k: \text{convolution kernel size}$$
$$p: \text{convolution padding size}$$
$$s: \text{convolution stride size}$$
$$n_{out} = \left[\frac{n_{in} + 2p - k}{s}\right] + 1$$
$$n_{out} = \left[\frac{28 + (2(0)) - 3}{1}\right] + 1$$
$$n_{out} = 26$$

The output of first convolution size is 26 x 26 pixels and convolution kernels number are 64. Therefore, the first convolution layer is 26 x 26 x 64.

3. Batch Normalization

```
model.add(BatchNormalization())
```

Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. By doing this, the layer normalizes the output of the previous layer to ensure a stable learning process by keeping the activations of each layer normalized.

4. Pooling Layer (Max)

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Max pooling can reduce the feature map size to decrease the computational load while retaining the most essential features. As a result, it can reduce the number of parameters and computational burden and helps prevent overfitting.

pool_size=(2, 2): pooling window is 2 x 2 pixels (4 pixels).

Stride size: Not mentioned, so the stride defaults to (2, 2) (Same with pool size).

Calculate the poling output:

First convolution layer is 26 x 26 x 64.

H_{in}: Height of the input

F: Size of the pooling filter

S: Stride size

P: Padding size

$$H_{out} = [\frac{H_{in} - F}{S} + 1]$$

$$H_{out} = [\frac{26 - 2}{2} + 1]$$

$$H_{out} = 13$$

The output of first pooling size is 13 x 13 pixels and convolution kernels number are 64. Therefore, the first pooling layer is 13 x 13 x 64.

5. Dropout Layer

```
model.add(Dropout(0.25))
```

Dropout(0.25): During training, 25% of the neurons in the current layer will be dropped to help the model generalize better and reduce the chance of overfitting. The network will randomly set 25% of the neurons to 0 to make them inactive for that iteration.

6. Second Convolution Layer

```
model.add(Conv2D(64, (3, 3), activation='relu'))
```

The second convolutional layer has 64 filters of size 3x3. It is similar to the first layer but doesn't need the input_shape because it inherits the shape from the previous layer.

Calculate the output of second convolution:

First convolution layer is 13 x 13 x 64.

n_{in}: number of input features

n_{out}: number of output features

k: convolution kernel size

p: convolution padding size

s: convolution stride size

$$n_{out} = \left[\frac{n_{in} + 2p - k}{s}\right] + 1$$

$$n_{out} = \left[\frac{13 + (2(0)) - 3}{1}\right] + 1$$

$$n_{out} = 11$$

The output of second convolution size is 11 x 11 pixels and convolution kernels number are 64. Therefore, the second convolution layer is 11 x 11 x 64.

7. Second Batch Normalization

```
model.add(BatchNormalization())
```

It is similar to the pervious batch normalization. The layer normalizes the output of the previous layer to ensure a stable learning process by keeping the activations of each layer normalized.

8. Second Pooling Layer

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

The second pooling layer is similar to first pooling layer.

pool_size=(2, 2): pooling window is 2 x 2 pixels.

Stride size: defaults to (2, 2).

Calculate the pooling output:

Second convolution layer is 11 x 11 x 64.

H_{in}: Height of the input

F: Size of the pooling filter

S: Stride size

P: Padding size

$$H_{out} = [\frac{H_{in} - F}{S} + 1]$$

$$H_{out} = [\frac{11 - 2}{2} + 1]$$

$$H_{out} \approx 5$$

The output of second pooling size is 5 x 5 pixels and convolution kernels number are 64. Therefore, the second pooling layer is 5 x 5 x 64.

9. Second Dropout Layer

```
model.add(Dropout(0.25))
```

It similar to first dropout layer. The 25% of the neurons in the current layer will be dropped to help the model generalize better and reduce the chance of overfitting.

10. Flatten Layer

```
model.add(Flatten())
```

This layer flattens the multi-dimensional feature maps into a one-dimensional vector to pass the data to a fully connected layer. The feature maps are still multi-dimensional after two rounds of convolution and pooling. Flatten compresses these features into a long vector, allowing the subsequent fully connected layer to process them.

11. Fully Connected Layer

```
for i in range(0,3):
    model.add(Dense(128, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
```

for i in range(0,3):: Adding 3 fully connected layers.

Dense(128, activation='relu'): Fully connected (dense) layer with 128 neurons.

BatchNormalization(): Stabilizing and speeding up training by ensuring that the output has a mean of 0 and standard deviation of 1.

Dropout(0.5): The 50% of the neurons in the current layer will be dropped to help the model generalize better and reduce the chance of overfitting.

12. Output Layer

```
model.add(Dense(num_classes, activation='softmax'))
```

Output the prediction or classification result.

Dense(num_classes): Number of classes prediction (Numbers 0 to 9 so the number of classes is 10).

activation='softmax': It converts the raw output scores from the neural network into probabilities. Each element in the output array will be a number between 0 and 1, and the sum of all the output values will be 1.

13. Initial Code Result (Unchanged)

```
!python mnist_cnn_keras.py --mode 'train' --batch-size 128 --epochs 5 --
model_path 'models/mnist_cnn.keras'
```

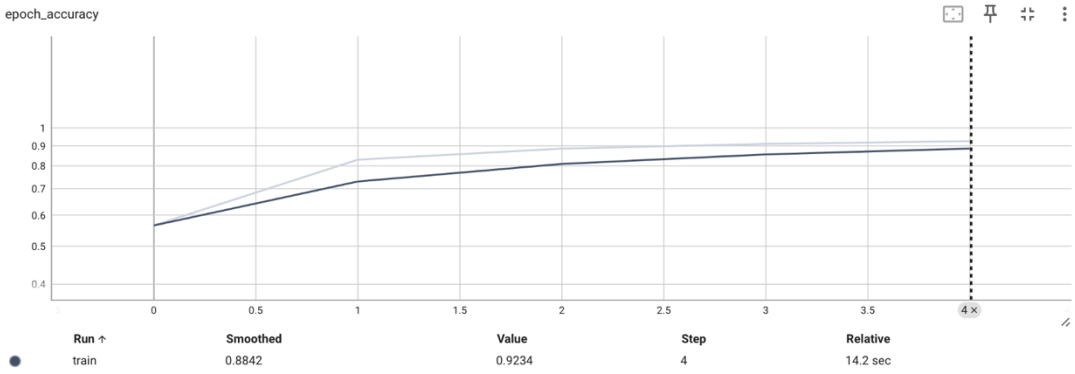
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 64)	640
batch_normalization (BatchNormalization)	(None, 26, 26, 64)	256
max_pooling2d (MaxPooling2D)	(None, 13, 13, 64)	0
dropout (Dropout)	(None, 13, 13, 64)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	36,928
batch_normalization_1 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_1 (Dropout)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
batch_normalization_2 (BatchNormalization)	(None, 128)	512
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 128)	16,512
batch_normalization_3 (BatchNormalization)	(None, 128)	512
dropout_3 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 128)	16,512
batch_normalization_4 (BatchNormalization)	(None, 128)	512
dropout_4 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

Changing the Activation Function in the Convolution Layer (e.g., ReLU, Sigmoid, Tanh)

ReLU

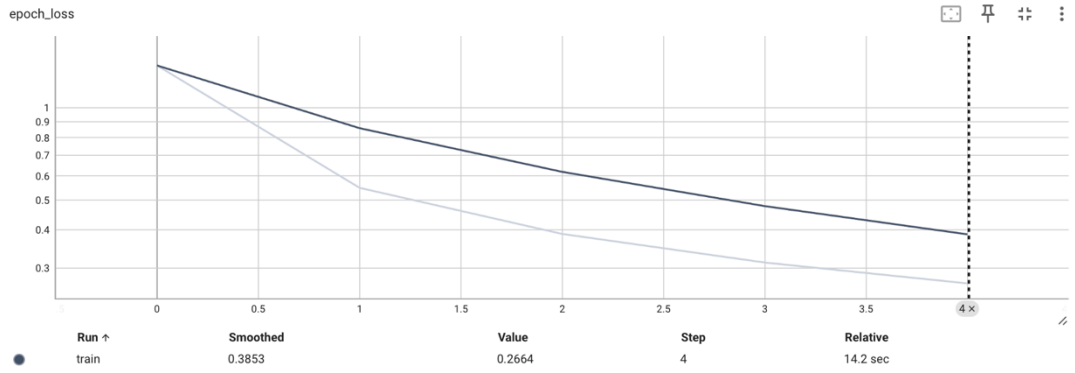
```
# CNN structure definition
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
for i in range(0,3):
    model.add(Dense(128, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Accuracy (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
0.563	0.8282	0.8838	0.9086	0.9234

Loss (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
1.373	0.5466	0.3864	0.3116	0.2664

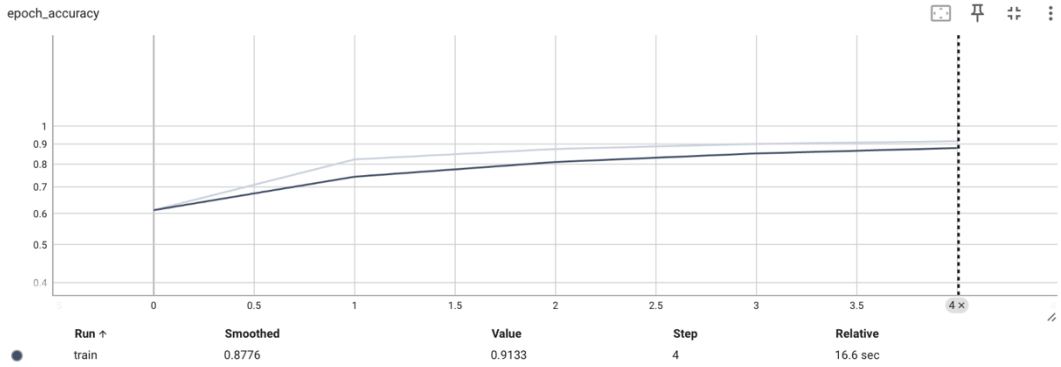
Test Result

Test Accuracy	Test Loss
97.33%	0.08525070548057556

Sigmoid

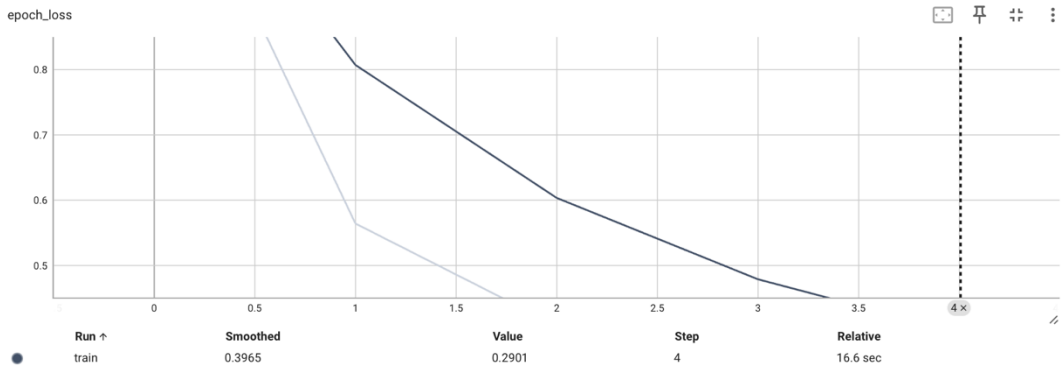
```
# CNN structure definition
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='sigmoid', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='sigmoid'))
model.add(BatchNormalization())
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'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Accuracy (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
0.6097	0.821	0.8728	0.8993	0.9133

Loss (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4

1.213	0.5635	0.407	0.331	0.2901

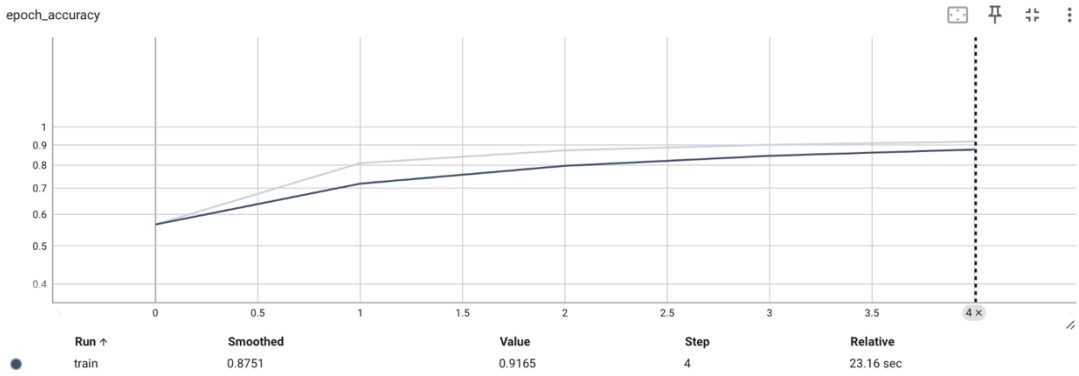
Test Result

Test Accuracy	Test Loss
96.74%	0.10960248112678528

Tanh

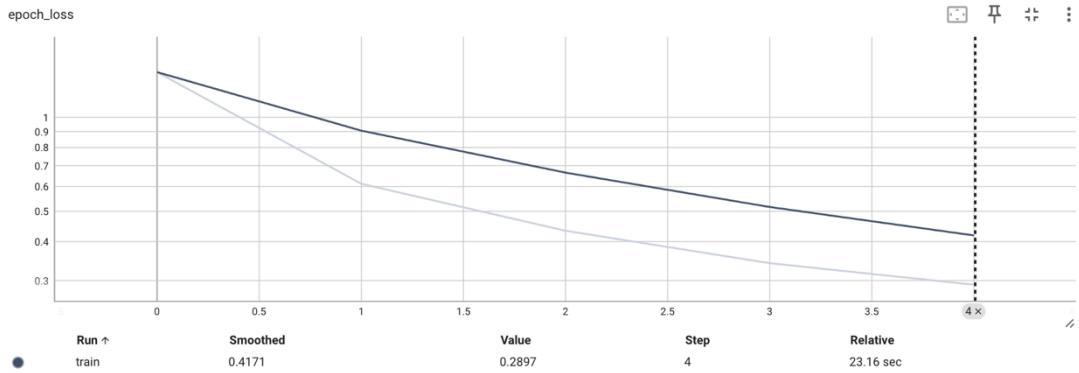
```
# CNN structure definition
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='tanh', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='tanh'))
model.add(BatchNormalization())
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='tanh'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Accuracy (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
0.5644	0.8081	0.871	0.8999	0.9165

Loss (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
1.394	0.6109	0.4317	0.3397	0.2897

Test Result

Test Accuracy	Test Loss
96.87%	0.12391960620880127

	ReLU	**Sigmoid**	**Tanh**
Accuracy Comparison (Epoch 5)			
Step 0:	0.563	0.6097	0.5644
Step 1:	0.8282	0.821	0.8081
Step 2:	0.8838	0.8728	0.871
Step 3:	0.9086	0.8993	0.8999
Step 4:	0.9234	0.9133	0.9165
Loss Comparison (Epoch 5)			
Step 0:	1.373	1.213	1.394
Step 1:	0.5466	0.5635	0.6109
Step 2:	0.3864	0.407	0.4317
Step 3:	0.3116	0.331	0.3397
Step 4:	0.2664	0.2901	0.2897
Test Result Comparison (Epoch 5)			
Test Accuracy:	97.33%	96.74%	96.87%
Test Loss:	0.08525070548057556	0.10960248112678528	0.12391960620880127

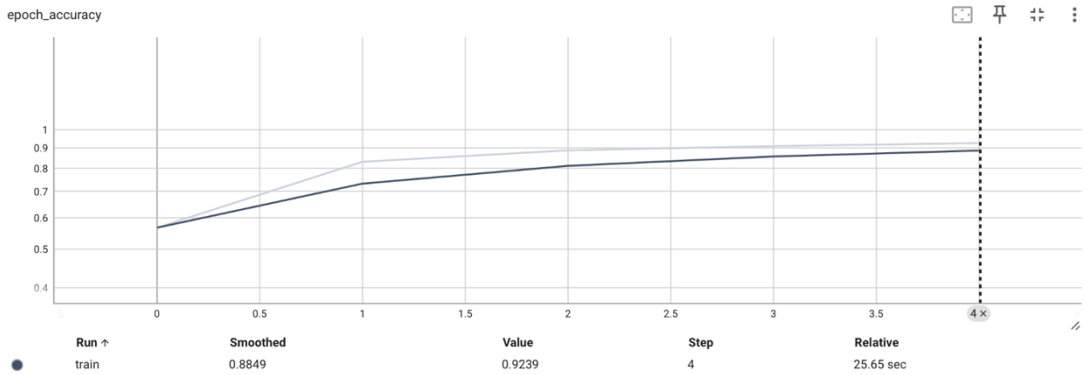
Changing the Number of Convolution Layer (From Two Layers to One Layer)

Two Convolution Layer

```
# CNN structure definition
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))
model.add(Flatten())
for i in range(0,3):
    model.add(Dense(128, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 64)	640
batch_normalization (BatchNormalization)	(None, 26, 26, 64)	256
max_pooling2d (MaxPooling2D)	(None, 13, 13, 64)	0
dropout (Dropout)	(None, 13, 13, 64)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	36,928
batch_normalization_1 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_1 (Dropout)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
batch_normalization_2 (BatchNormalization)	(None, 128)	512
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 128)	16,512
batch_normalization_3 (BatchNormalization)	(None, 128)	512
dropout_3 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 128)	16,512
batch_normalization_4 (BatchNormalization)	(None, 128)	512
dropout_4 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

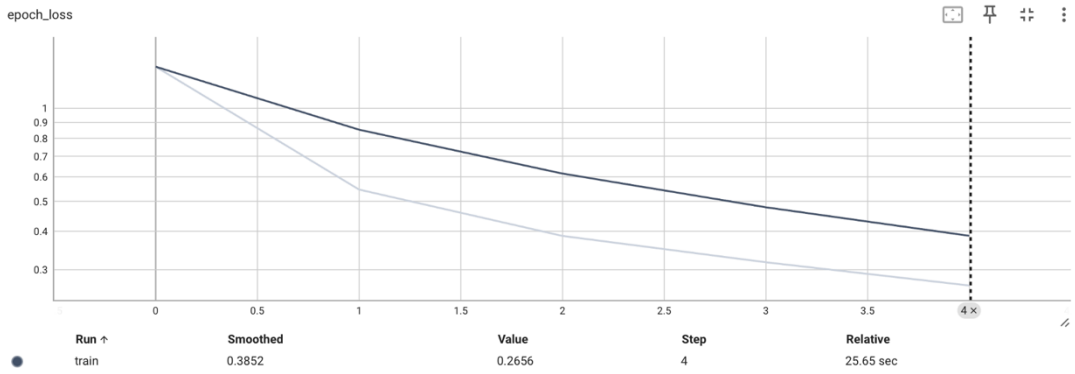
Accuracy (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4

0.5652	0.829	0.886	0.9083	0.9239

Loss (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
1.363	0.5442	0.3848	0.316	0.2656

Test Result

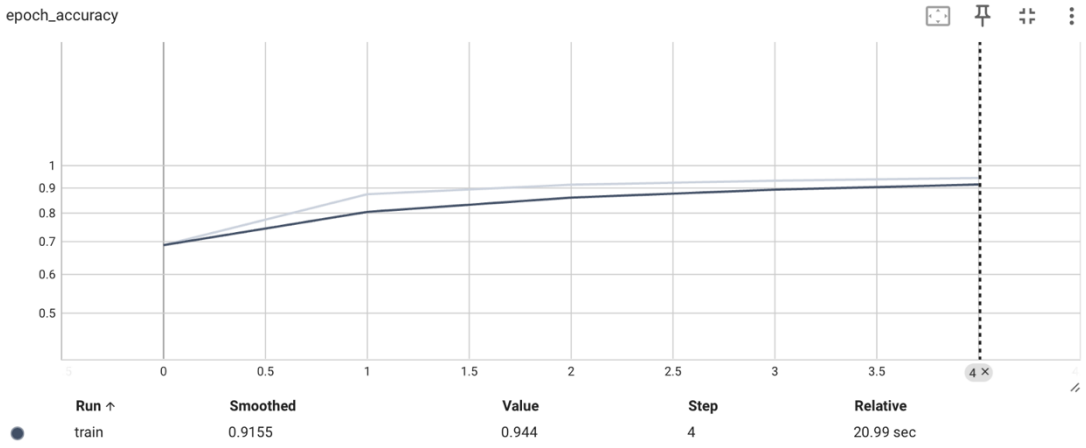
Test Accuracy	Test Loss
97.45%	0.08515661954879761

One Convolution Layer

```
# CNN structure definition
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))
model.add(Flatten())
for i in range(0,3):
    model.add(Dense(128, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

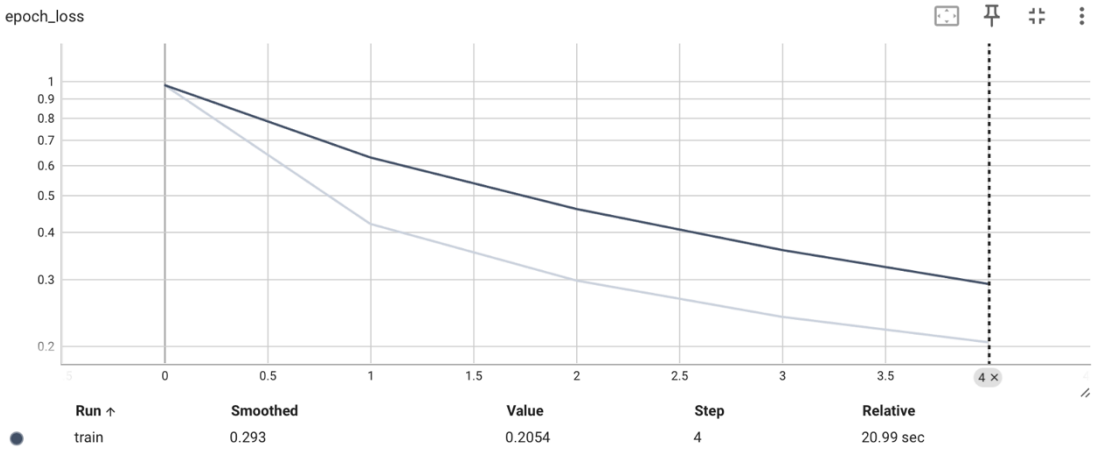
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 64)	640
batch_normalization (BatchNormalization)	(None, 26, 26, 64)	256
max_pooling2d (MaxPooling2D)	(None, 13, 13, 64)	0
dropout (Dropout)	(None, 13, 13, 64)	0
flatten (Flatten)	(None, 10816)	0
dense (Dense)	(None, 128)	1,384,576
batch_normalization_1 (BatchNormalization)	(None, 128)	512
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 128)	16,512
batch_normalization_2 (BatchNormalization)	(None, 128)	512
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 128)	16,512
batch_normalization_3 (BatchNormalization)	(None, 128)	512
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

Accuracy (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
0.6888	0.8748	0.9145	0.932	0.944

Loss (Epoch 5)



Step 0	Step 1	Step 2	Step 3	Step 4
0.9833	0.4222	0.2991	0.2397	0.2054

Test Result

Test Accuracy	Test Loss
97.73%	0.07190988957881927

	Two Layers	**One Layer**
Accuracy Comparison (Epoch 5)		
Step 0:	0.5652	0.6888
Step 1:	0.829	0.8748
Step 2:	0.886	0.9145
Step 3:	0.9083	0.932
Step 4:	0.9239	0.944
Loss Comparison (Epoch 5)		
Step 0:	1.363	0.9833
Step 1:	0.5442	0.4222
Step 2:	0.3848	0.2991
Step 3:	0.316	0.2397
Step 4:	0.2656	0.2054
Test Result Comparison (Epoch 5)		
Test Accuracy:	97.45%	97.73%
Test Loss:	0.08515661954879761	0.07190988957881927

(b) Pytorch Part

mnist_cnn_pytorch.py initial code (Unchanged):

```
# Convolutional Neural Network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        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
        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
        self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x,-1)
model = Net()
```

The PyTorch explicitly define the layers in __init__() and the forward propagation logic in forward().

1. First Convolutional Layer

```
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
```

```
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
```

The first convolutional layer applies the ReLU activation function and max pooling with a 2x2 window to reduce the dimensions of the feature map.

1: Channel.

10: Convolution kernels number.

Kernel_size=5: Convolution kernels size is 5 x 5 pixels.

The output of first convolution:

n_{in}: number of input features

n_{out}: number of output features

k: convolution kernel size

p: convolution padding size

s: convolution stride size

$$n_{out} = \left[\frac{n_{in} + 2p - k}{s}\right] + 1$$

$$n_{out} = \left[\frac{28 + (2(0)) - 5}{1}\right] + 1$$

$$n_{out} = 24$$

The output of first convolution is 24 x 24 x 10.

The output of first pooling:

H_{in}: Height of the input

F: Size of the pooling filter

S: Stride size

P: Padding size

$$H_{out} = [\frac{H_{in} - F}{S} + 1]$$

$$H_{out} = [\frac{24 - 2}{2} + 1]$$

$$H_{out} = 12$$

The output of first pooling size is 12 x 12 x 10.

2. Second Convolutional Layer

```
self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5
self.conv2_drop = nn.Dropout2d() # The default dropout rate is 50%

x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
```

The second convolutional layer applies the ReLU activation function, max pooling with a 2 x 2 window, and 50% of the neurons will be randomly dropped (set to zero).

10: Input channels (Number of feature maps from the previous convolutional layer).

20: Convolution kernels number.

Kernel_size=5: Convolution kernels size is 5 x 5 pixels.

The output of second convolution:

n_{in}: number of input features

n_{out}: number of output features

k: convolution kernel size

p: convolution padding size

s: convolution stride size

$$n_{out} = \left[\frac{n_{in} + 2p - k}{s}\right] + 1$$

$$n_{out} = \left[\frac{12 + (2(0)) - 5}{1}\right] + 1$$

$$n_{out} = 8$$

The output of second convolution is 8 x 8 x 20.

The output of second pooling:

H_{in}: Height of the input

F: Size of the pooling filter

S: Stride size

P: Padding size

$$H_{out} = [\frac{H_{in} - F}{S} + 1]$$

$$H_{out} = [\frac{8-2}{2} + 1]$$

$$H_{out} = 4$$

The output of second pooling size is 4 x 4 x 20.

3. Flatten Layer

```
        x = x.view(-1, 320)
```

This layer flattens the multi-dimensional feature maps into a one-dimensional vector to pass the data to a fully connected layer. The -1 argument automatically infers the batch size. The 320 (4 x 4 x 20) is the total number of features after the convolution and pooling layers.

4. First Fully Connected Layer

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

```
        x = F.relu(self.fc1(x))
```

The fully connected layer takes 320 input features, produces 50 output features, and applies the ReLU activation function.

5. Dropout Layer

```
        x = F.dropout(x, training=self.training)
```

Some neurons in the fully connected layer are randomly set to zero during training to prevent overfitting.

6. Second Fully Connected Layer

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

```
        x = self.fc2(x)
```

The fully connected layer takes 50 input features and produces 10 output features.

7. Output Layer

```
        return F.log_softmax(x,-1)
model = Net()
```

The final output from the last fully connected layer of the neural network and the softmax function convert the raw output scores of the network into a probability distribution. The dim=-1 refers to the last dimension of a tensor, no matter how many dimensions the tensor has.

8. Initial Code Result (Unchanged)

```
!python mnist_cnn_pytorch.py --mode 'train' --batch-size 128 --epochs 5 --
model_path 'models/mnist_cnn.pth'
```

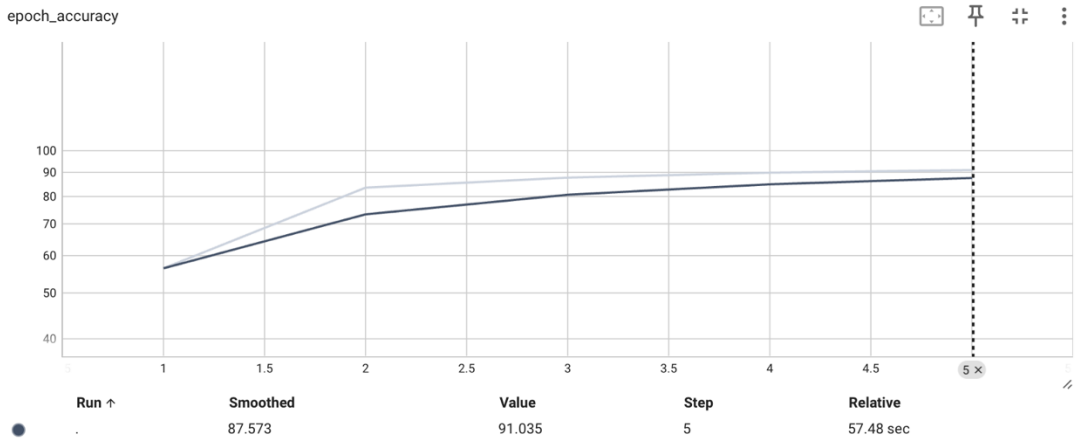
```
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.338448
Train Epoch: 1 [12800/60000 (21%)]      Loss: 1.858987
Train Epoch: 1 [25600/60000 (43%)]      Loss: 1.068790
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.750236
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.639476
Train Epoch: 2 [0/60000 (0%)]   Loss: 0.629386
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.407592
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.481759
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.492225
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.512075
Train Epoch: 3 [0/60000 (0%)]   Loss: 0.294531
Train Epoch: 3 [12800/60000 (21%)]      Loss: 0.375103
Train Epoch: 3 [25600/60000 (43%)]      Loss: 0.322567
Train Epoch: 3 [38400/60000 (64%)]      Loss: 0.469571
Train Epoch: 3 [51200/60000 (85%)]      Loss: 0.452315
Train Epoch: 4 [0/60000 (0%)]   Loss: 0.441263
Train Epoch: 4 [12800/60000 (21%)]      Loss: 0.378823
Train Epoch: 4 [25600/60000 (43%)]      Loss: 0.296375
Train Epoch: 4 [38400/60000 (64%)]      Loss: 0.210502
Train Epoch: 4 [51200/60000 (85%)]      Loss: 0.343971
Train Epoch: 5 [0/60000 (0%)]   Loss: 0.303392
Train Epoch: 5 [12800/60000 (21%)]      Loss: 0.282017
Train Epoch: 5 [25600/60000 (43%)]      Loss: 0.301219
Train Epoch: 5 [38400/60000 (64%)]      Loss: 0.509190
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.338769
Saving CNN to models/mnist_cnn.pth
```

Changing the Number of Output Neurons (From 50 to 25)

50 Neurons

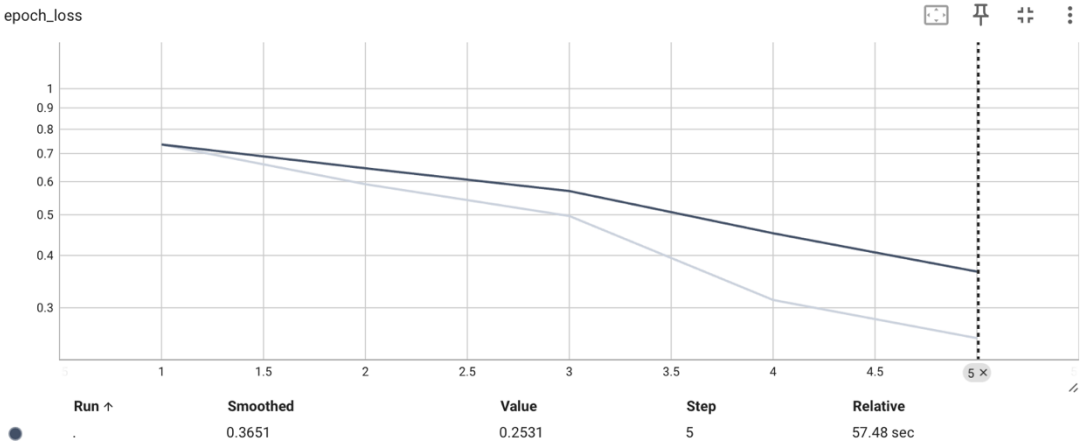
```
# Convolutional Neural Network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5
        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
        self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x,-1)
model = Net()
```

Accuracy (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
56.41	83.55	87.74	89.88	91.04

Loss (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
0.7351	0.5908	0.4959	0.3123	0.2531

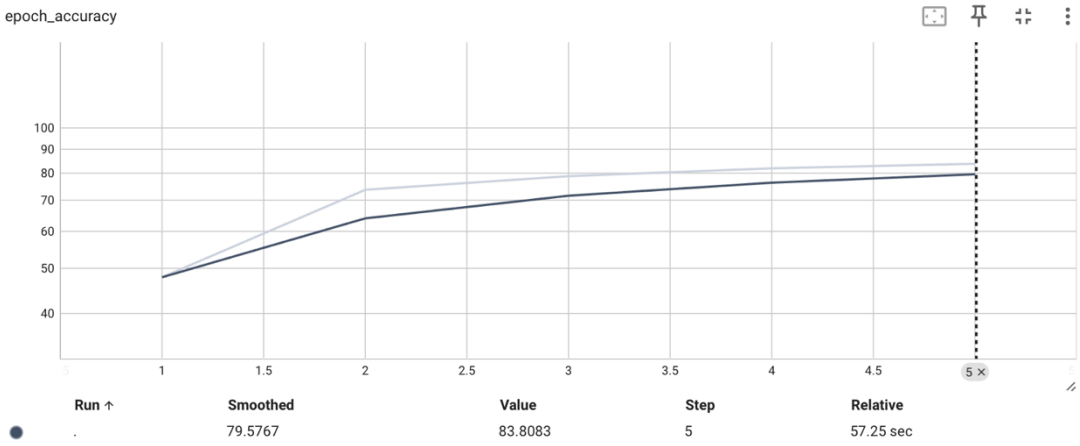
Test Result

Accuracy	Average Loss
9660/10000 (97%)	0.1065

25 Neurons

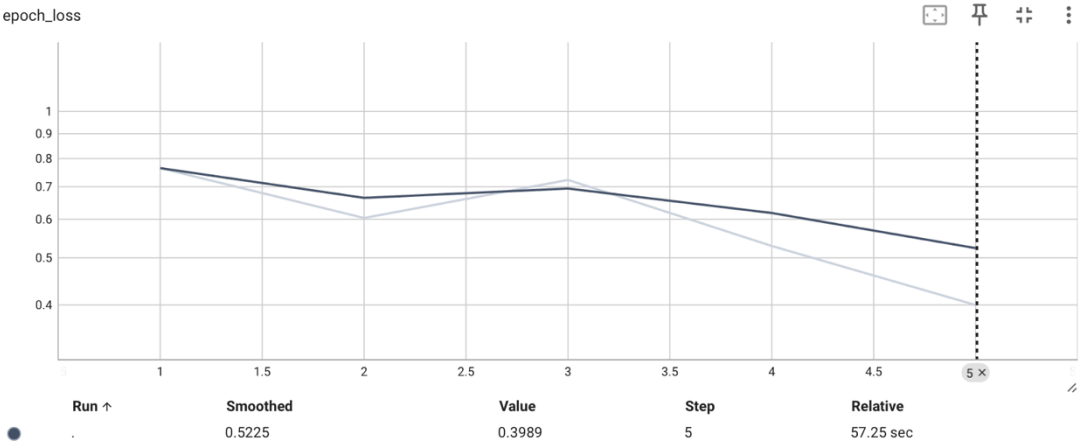
```
# Convolutional Neural Network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5
        self.conv2_drop = nn.Dropout2d() # The default dropout rate is 50%
        self.fc1 = nn.Linear(320, 25) # number of input neurons: 320, number of output neurons: 25
        self.fc2 = nn.Linear(25, 10) # Number of input neurons: 25, Number of output neurons: 10
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x,-1)
```

Accuracy (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
47.86	73.71	78.83	81.93	83.81

Loss (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
0.7637	0.6028	0.7222	0.5279	0.3989

Test Result

Accuracy	Average Loss
9624/10000 (96%)	0.1313

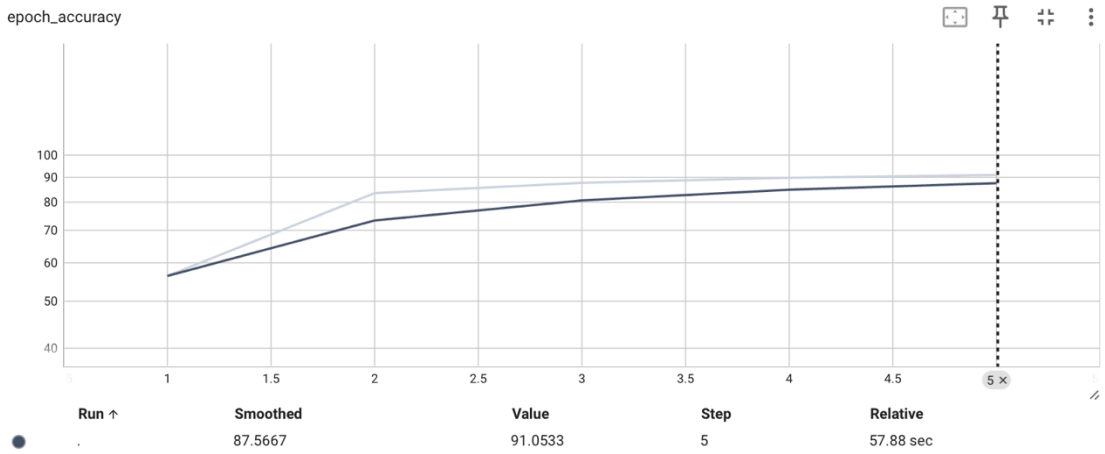
	50 Neurons	**25 Neurons**
Accuracy Comparison (Epoch 5)		
Step 1:	56.41	47.86
Step 2:	83.55	73.71
Step 3:	87.74	78.83
Step 4:	89.88	81.93
Step 5:	91.04	83.81
Loss Comparison (Epoch 5)		
Step 1:	0.7351	0.7637
Step 2:	0.5908	0.6028
Step 3:	0.4959	0.7222
Step 4:	0.3123	0.5279
Step 5:	0.2531	0.3989
Test Result Comparison (Epoch 5)		
Accuracy:	9660/10000 (97%)	9624/10000 (96%)
Accuracy Loss:	0.1065	0.1313

Changing the Type of Pooling (From Max to Average)

Max Pooling

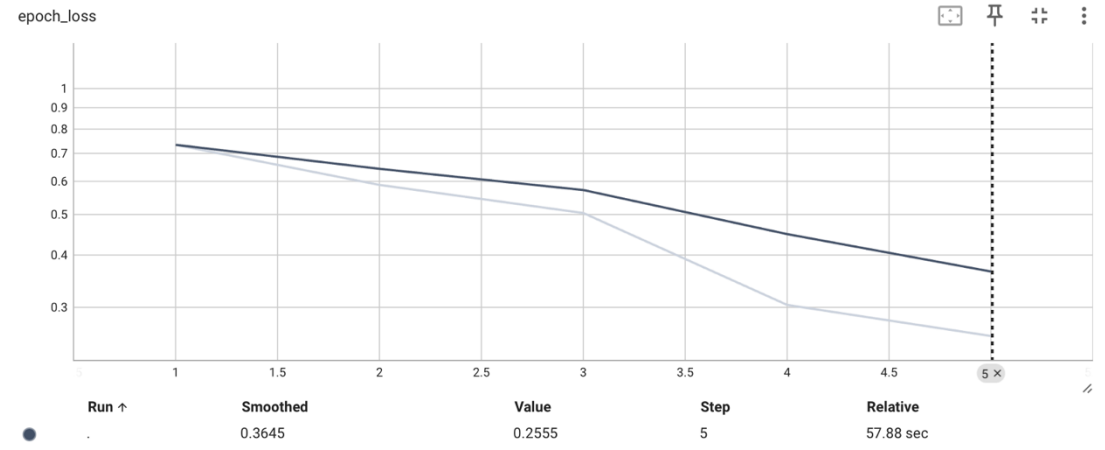
```
# Convolutional Neural Network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5
        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
        self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x,-1)
```

Accuracy (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
56.4	83.55	87.71	89.85	91.05

Loss (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
0.733	0.5875	0.5032	0.3032	0.2555

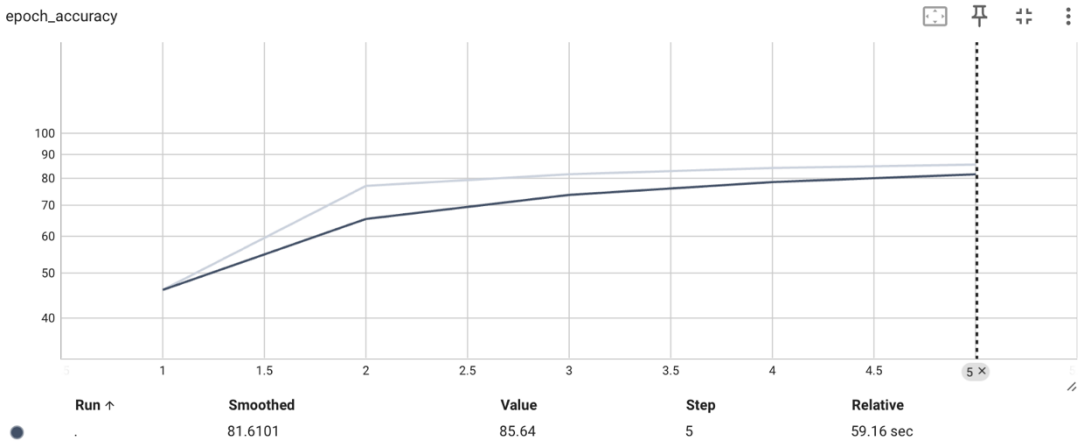
Test Result

Accuracy	Average Loss
9658/10000 (97%)	0.1073

Average Pooling

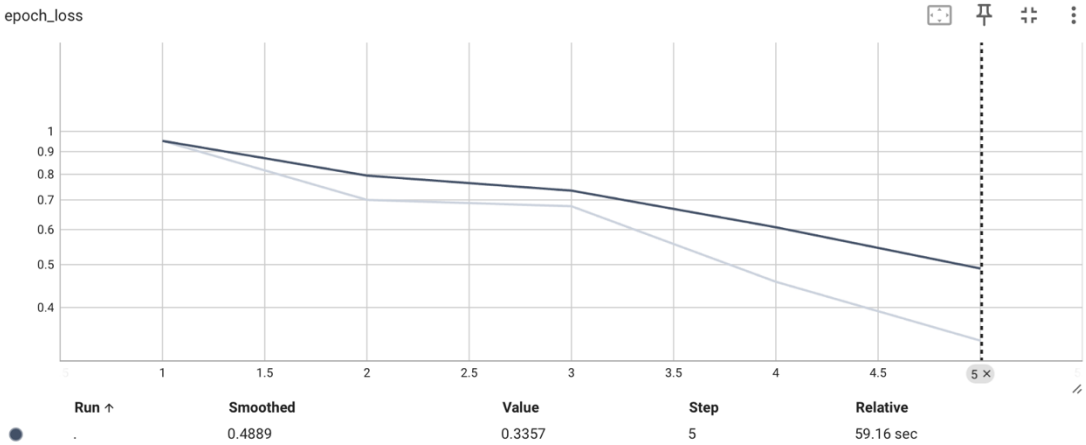
```
# Convolutional Neural Network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5) # Input channel: 1, Output channel: 10, Filter size: 5x5
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5) # Input channel: 10, Output channel: 20, Filter size: 5x5
        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
        self.fc2 = nn.Linear(50, 10) # Number of input neurons: 50, Number of output neurons: 10
    def forward(self, x):
        x = F.relu(F.avg_pool2d(self.conv1(x), 2))
        x = F.relu(F.avg_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x,-1)
```

Accuracy (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
46.01	77.06	81.64	84.2	85.64

Loss (Epoch 5)



Step 1	Step 2	Step 3	Step 4	Step 5
0.9508	0.6991	0.6764	0.4562	0.3357

Test Result

Accuracy	Average Loss
9359/10000 (94%)	0.2147

	Max Pooling	**Average Pooling**
Accuracy Comparison (Epoch 5)		
Step 1:	56.4	46.01
Step 2:	83.55	77.06
Step 3:	87.71	81.64
Step 4:	89.85	84.2
Step 5:	91.05	85.64
Loss Comparison (Epoch 5)		
Step 1:	0.733	0.9508
Step 2:	0.5875	0.6991
Step 3:	0.5032	0.6764
Step 4:	0.3032	0.4562
Step 5:	0.2555	0.3357
Test Result Comparison (Epoch 5)		
Accuracy:	9658/10000 (97%)	9359/10000 (94%)
Accuracy Loss:	0.1073	0.2147

Conclusion

In conclusion, the lab exercise explored the convolutional neural networks for handwritten digit classification using Keras and PyTorch frameworks to observe the effects on model performance by changing the number of layers, neurons, and activation functions to gain training accuracy and loss results. This lab can change the number of epochs, activation function, filter sizes, dense, batch size, pooling type (Max and Average), and convolution layer to observe the different results. Therefore, I randomly chose the four methods to track how they impact the model's performance.

Regarding the Keras part, I have changed the activation function in the convolution layer, such as ReLU, Sigmoid, and Tanh. As a result, with ReLU, the testing accuracy reached 97.33%, and the test loss was 0.085. With Sigmoid, the testing accuracy reached 96.74%, and the test loss was 0.1096. With Tanh, the testing accuracy reached 96.87%, and the test loss was 0.1239. In view of this, ReLU is the best-performing activation function in this case, both in terms of accuracy and loss.

Moreover, I have changed the number of convolution layers from two layers to one layer. The testing accuracy reached 97.45% in two convolution layers, and the test loss was 0.0852, while the testing

accuracy reached 97.73% in one convolution layer, and the test loss was 0.0719. Therefore, adding more convolution layers can sometimes lead to overfitting or higher complexity without improving performance.

Concerning the Pytorch part, I have changed the number of output neurons from 50 to 25. The 50 neurons' accuracy was 97% (9660/10000), and the accuracy loss was 0.1065, while the 25 neurons' accuracy was 96% (9624/10000), and the accuracy loss was 0.1313. Because of this, the more output neurons allow the model to have more capacity to learn from the data, while fewer neurons may limit the model's expressiveness, leading to a slight decline in performance.

Furthermore, I have changed the type of pooling from max to average. The max pooling was 97% (9658/10000), and the accuracy loss was 0.1073, while the average pooling accuracy was 94% (9359/10000), and the accuracy loss was 0.2147. Thus, max pooling is the better choice for the model, as it helps capture essential features and yields better overall accuracy and lower loss.