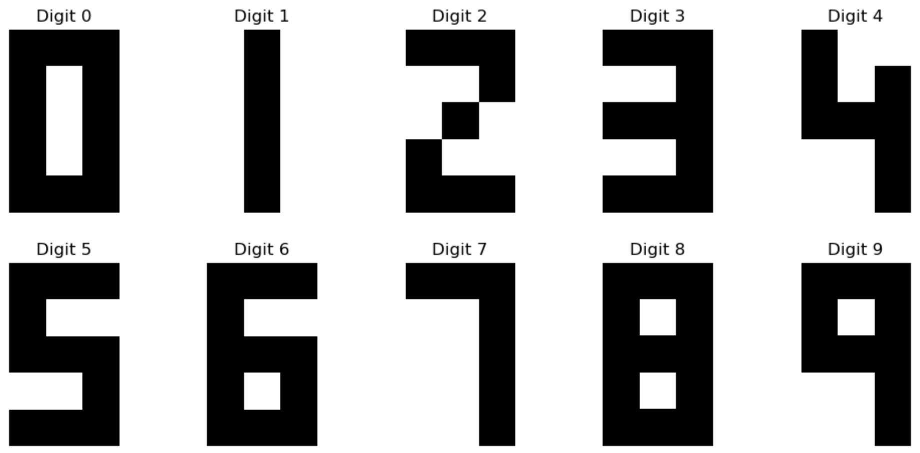
Assignment 03: CNN’s

For this task, you will be using the baseline code for CNN’s provided in the class jupyter notebook by the name of convolution-neural-network.ipynb. **Do not** use any ML libraries. The notebook has the following functions:

1. conv2D(image, kernel)
2. relu(x)
3. max_pooling(image, pool_size)
4. avg_pooling(image, pool_size)
5. dense_layer(input, weight, bias)
6. softmax(x)

Task 0: Understand the Working Example

To through the working example of digit recognition (0, 1, …, 9). Each digit is marked as a binary 5x5 array of 0’s and 1’s, which are placed in the **<u>dataset</u>** array, which is accessible as dataset[0], dataset[1], …, dataset[9].



Additionally provided are **<u>labels</u>** array in the form of “one hot vectors”. A one hot vector is typically used for n class problems. For n classes, there are n one-hot vectors, and each vector has a size of n. In the contents of each vector, there is only a single 1 in the vector, representing the class, whereas the other contents of the vector are 0. You can visualize it as a kind of large identity matrix of size n x n.

For this toy example, we are only considering a single random **kernel** of size 3 x 3. For the dense fully connected layer, we also have random **fc_weights**, and a bias term **fc_bias**.

Forward Pass: The convolution network forward pass then follows these stages:

Stage Name	Convolution 2D + Relu	Max Pooling	Dense Fully Connected Layer + Softmax
Variable Names	feature_map → activated	→ pooled	→ flattened → dense_output → output_layer

Backward Pass: Once the output_layer is received, the loss gradient by comparing against the labels array is computed. What follows is then a backward pass where learning is carried out for fc_weights through **fc_weights_gradient**, and fc_bias through **fc_bias_gradient**. There is no such learning for the max_pooling layer, only logic for the selection of the max_pooled value is present and appears as **feature_map_gradient**. Finally, learning for the kernel takes place and is present in **kernel_gradient**. When all is done, the weights are updated through obtained gradients via gradient descent algorithm, where **learning_rate** = 0.1.

The above forward and backward pass repeats for each image in the dataset. This is controlled through the following for loop:

```
for img, label in zip(dataset, labels)
```

The exercise is then repeated for a number of **epochs**. In our case, we are looking at epochs = 200. This is controlled through the following for loop:

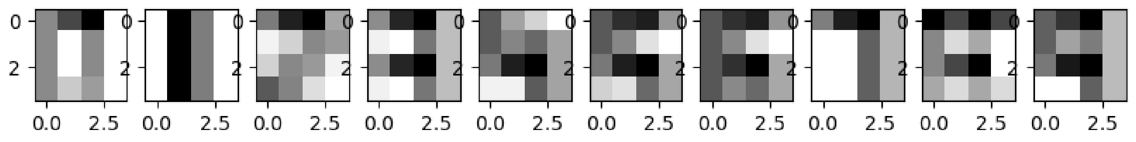
```
for epoch in range(epochs):
```

Now attempt the following questions:

	Question	Answer
1	What are the sizes of the arrays feature_map, activated, pooled, flattened, dense_output, and output_layer?	
2	Provide the expected values of arrays feature_map and output_layer in the 1st epoch.	
3	Provide the expected values of arrays feature_map and output_layer in the last epoch.	
4	Explain why I have added 1e-9 to the output_layer when calculating the loss function?	
5	Present a single plot showing the avg_loss against epochs for learning rate = 0.1, 0.01, 0.001, and 0.0001.	
6	What is the loss at 200th epoch for kernel sizes of 2x2, 3x3, and 5x5.	
7	Place the following code in your for loop and inspect at epochs 1, 50, 100, 150, and 200 what your feature_map looks like. Give the output image as your answer. If you can come up with an explanation, do provide it.	

```
fig, axes = plt.subplots(1, 10, figsize=(10, 5))
  for i, ax in enumerate(axes.flat):
    ax.imshow(conv2d(dataset[i],kernel), cmap='gray_r')
```

For reference, the sample output for 2x2 kernel appears on the 100th epoch looks as (Note: yours may be different):

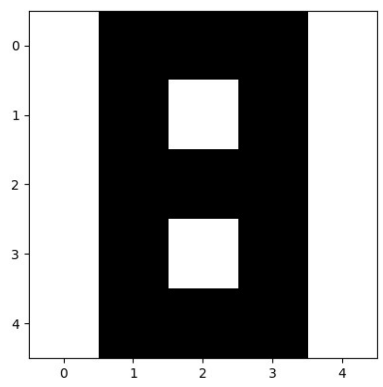


Task 1: Creating and Evaluating Test Cases

To run test cases and evaluate performance, you can start with the following baseline code:

```
usethis = 8

test_image = dataset[usethis]
test_label = labels[usethis]
plt.imshow(test_image, cmap='gray_r')

# Forward Pass for Testing
feature_map  = conv2d(test_image, kernel)
activated    = relu(feature_map)
pooled       = max_pooling(activated, pool_size=2)
flattened    = pooled.flatten()
dense_output = dense_layer(flattened, fc_weights, fc_bias)
output_layer = softmax(dense_output)

# Predicted class

predicted_class = np.argmax(output_layer)
actual_class    = np.argmax(test_label)

print("Predicted:", predicted_class, "Actual:", actual_class)
```

Running this code means that you should be able to see output like the following:



```
Predicted: 8 Actual: 8
```

Attempt the following questions:

	Question	Answer
1	Are your predicted (predicted_label) and actual (actual_class) the same? If so, how much does this hold against random noise added to the image?	
2	Are there some images which always gives correct predicted class? If so, which ones are they?	
3	Are there some images which always gives wrong predicted class? If so, which ones are they?	

4 Add a confusion matrix to the code. In the confusion matrix, the rows would represent the actual class, whereas the columns would represent the predicted class. For this, make the following changes in the code:

1. Place your test code inside a for loop and evaluate 1000 times. E.g.:

```
for i in range(10):
    # Test Code here
```

2. Create a random confusion matrix of size 10x10 (for all classes) outside your for loop.

```
confusion_matrix = np.zeros((10, 10), dtype=int)
```

3. Then, update your confusion matrix simply as:

```
confusion_matrix[actual_class][predicted_class] += 1
```

4. You can then view the confusion matrix directly, or visualize it as:

```
plt.imshow(confusion_matrix)
```

Show this confusion matrix as your answer.

5 Modify your code to determine True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN). In a 2 class system, this would be the regions:

	Positive	Negative
Positive	True Positive TP	False Negative FN: Type II Error
Negative	False Positive FP: Type I Error	True Negative TN

In a 10 class case, an example region based representation for class 3 would be the following identified areas of your confusion matrix:

	0	1	2	3	4	5	6	7	8	9
0	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
1	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN

2	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
3	FN	FN	FN	TP	FN	FN	FN	FN	FN	FN
4	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
5	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
6	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
7	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
8	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN
9	TN	TN	TN	FP	TN	TN	TN	TN	TN	TN

Similar regions can be identified for all other classes using the following code:

```
for cls in range(num_classes):
 tp_tmp = confusion_matrix[cls][cls]
 fp_tmp = confusion_matrix[:, cls].sum() - tp_tmp
 fn_tmp = confusion_matrix[cls, :].sum() - tp_tmp
 tn_tmp = confusion_matrix.sum() - (tp_tmp+fp_tmp+fn_tmp)
```

Note that the above needs to be done for each class. You can initialize these values centrally for all classes as:

```
tp = fp = fn = tn = 0
```

and then simply adding each one individually as:

```
tp += tp_tmp
```

When done, print your total TP, FP, FN, and TN values as the following table:

TP: ____	FN: ______
FP: ____	TN: ______

Modify your code to include Precision, Recall/Sensitivity, Specificity, and Negative Prediction. All these 4 terms correspond to the following regions:

	Positive	Negative	
Positive	True Positive TP	False Negative FN: Type II Error	Recall / Sensitivity
Negative	False Positive FP: Type I Error	True Negative TN	Specificity
	Precision	Negative Prediction	

Their formulas are:

6
```
precision = tp / (tp + fp)
recall    = tp / (tp + fn)
specifity = tn / (tn + fp)
neg_pred  = tn / (tn + fn)
```

Also include into your calculations Accuracy and F1 score, calculated as:

```
accuracy = (tp + tn) / (tp + tn + fp + fn)
f1_score = 2 * (precision * recall) / (precision + recall)
```

To avoid divide by zero, ensure you make the calculation condition, e.g.

```
precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0
```

When done, report all these values as your answer.

You will note that so far, you are testing on the basis of images which are already trained.
7 To make the test cases, different, you may consider adding some noise to the test images

either directly, or randomly. As an example, following will add a 1 to position 0, 0 of the image, and 0,4 position of the image:

```
test_image[0][0] = 1
test_image[0][4] = 1
```

Alternatively, you may add two pieces of noise randomly across the image as:

```
for f in range(2):
 test_image[random.randint(0,4)][random.randint(0,4)] = 1;
```

What is the impact on accuracy, precision, and recall in the cases of addition of noise. Provide your data as a table:

Noise Term Quantity	0	1	2	3	4
Accuracy					
Precision					
Recall					

Task 2: Adding an Additional Kernel

So far, you have been considering a single kernel in your CNN, given as:

```
kernel = np.random.randn(3, 3)
```

Interact with ChatGPT or other AI tool and try to include code and results for inclusion of one additional kernel to your code. Then, report the following table:

Noise Term Quantity	0	1	2	3	4
Accuracy					
Precision					
Recall					

Deliverable

Submit a report containing answers asked, along with your full jupyter-notebook working.