

Homework 1

Question 1 = 15 marks

1 = 7 marks

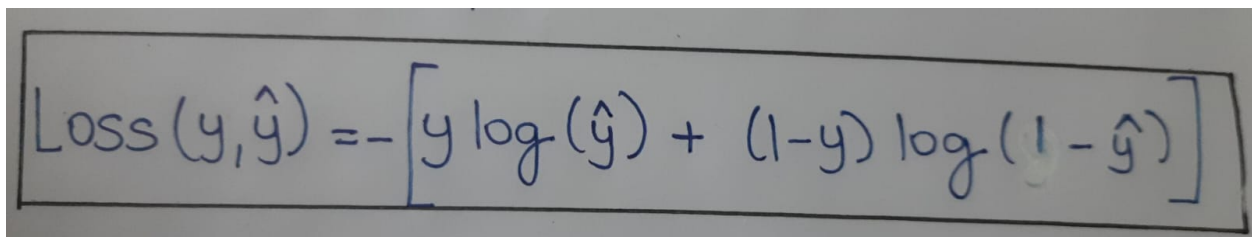
a = 1 marks

MSE might not be a good choice for the loss function because,

- It is more suitable for regression problems where the predictions are continuous and not classification task. In classification tasks, where the output is discrete (e.g., binary labels), the interpretation of squared differences may not be meaningful, and other loss functions like cross-entropy are typically more appropriate.
- In classification tasks, particularly when dealing with imbalanced datasets, MSE can be sensitive to class imbalances. It might not penalize misclassifications effectively, especially if one class (e.g., sweet papayas) is dominant in the dataset.

b = 1 marks

y = Target label



A handwritten formula for binary cross-entropy loss is shown inside a rectangular box. The formula is:
$$\text{Loss}(y, \hat{y}) = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$

c = 1 marks

$$\text{Loss} = - [0. \log(0.9) + 1.\log(0.1)]$$

$$\Rightarrow - [0 + - \log(10)]$$

$\Rightarrow \log(10)$ {log has a base of 2}

3.322

d = 2 marks

All logarithms are in base 2

1. For the first example:

$$\text{BCE}(1, 0.1) = -(1 \times \log(0.1) + (1-1) \times \log(1-0.1))$$

$$\Rightarrow -(\log(0.1))$$

$$\Rightarrow \log(10)$$

$$\Rightarrow 3.322$$

2. For the second example:

$$\text{BCE}(0, 0.2) = -(0 \times \log(0.2) + (1-0) \times \log(1-0.2))$$

$$\Rightarrow -(\log(0.8))$$

$$\Rightarrow -(\log(8) - \log(10))$$

$$\Rightarrow -(3 - \log(10))$$

$$\Rightarrow 0.322$$

3. For the third example:

$$\text{BCE}(0, 0.7) = -(0 \times \log(0.7) + (1-0) \times \log(1-0.7))$$

$$\Rightarrow -(\log(0.3))$$

$$\Rightarrow -(\log(3) - \log(10))$$

$$\Rightarrow -(1.585 - 3.322)$$

$$\Rightarrow 1.737$$

$$\text{Average Loss} = (3.322 + 0.322 + 1.737) / 3$$

1.793

e = 2 marks

In L2 regularization, we do,

Cost function = Loss function + Penalty Term

⇒ Loss Function = **LBCE**

⇒ Penalty Term(in L2 regularization) = $\lambda/n (\sum ||W||^2)$

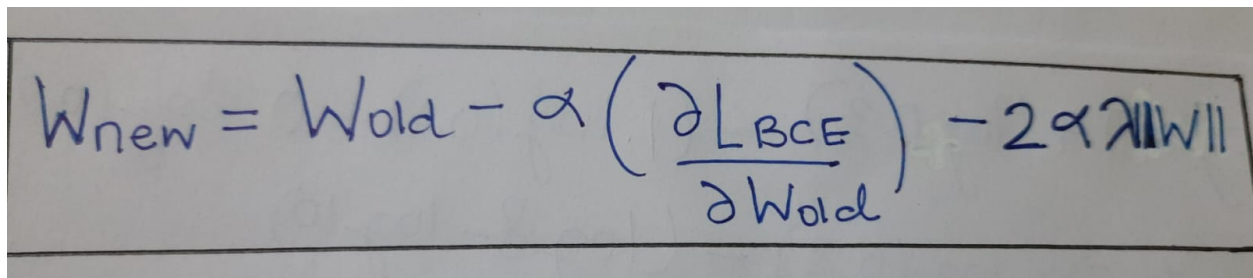
⇒ Penalty Term for one trainable weight = $\lambda (||W||^2)$

Cost function = **LBCE** + $\lambda (||W||^2)$

Weight Update Rule,

$W_{new} = W_{old} - \alpha (\partial \text{CostFunction} / \partial W_{old})$

$W_{new} = W_{old} - \alpha (\partial (\text{LBCE} + \lambda (||W||^2)) / \partial W_{old})$



A photograph of a handwritten equation on a piece of paper. The equation is:
$$W_{new} = W_{old} - \alpha \left(\frac{\partial L_{BCE}}{\partial W_{old}} \right) - 2\alpha \lambda ||W||$$

Comparing model A (with L2 regularization) to model B (without L2 regularization) —

We would expect the weights of model A to be smaller in magnitude compared to the weights of model B.

This is because L2 regularization penalizes large weights by adding a term proportional to the weights themselves in the loss function.

As a result, during training, model A will tend to learn simpler patterns with smaller weights, leading to a more robust model that is less prone to overfitting compared to model B.

2 = 8 marks

Softmax Function = Activation Function used in neural networks.

Input x of shape $D_x \times 1$ —

- This means that the input x is a column vector with D_x elements. For example, if $D_x=3$, then x could be represented as:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

One-hot encoded label y :

- The label y is represented as a one-hot encoded vector of length K .
- This means that if there are K classes, then y is a vector with K elements where only one element is 1 (indicating the class) and all other elements are 0.
- For example, if $K=3$ and y corresponds to the second class, y could be represented as:

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Hidden layer with D_a nodes:

- The hidden layer has D_a nodes, meaning that there are D_a neurons in the hidden layer. Each neuron in the hidden layer will produce an output, resulting in a vector $z^{[1]}$ of size $D_a \times 1$.
- For example, if $D_a=4$, then $z^{[1]}$ could be represented as:
- z_k is the k -th element of vector $z^{[1]}$.

$$z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

There is the input layer with D_x neurons.

There is the hidden layer with D_a neurons.

There is the output layer with K neurons.

a = 1 marks

Shape of $W[2] = K * D_a$

Since we are doing a K class classification and there are D_a nodes in the hidden layer.

The input in the second layer will depend on D_a .

The output of the second layer will depend on K .

Shape of $b[2] = K * 1$

Since there is only one bias term for each node in the output layer (i.e. K nodes)

Shape of the output for m example samples —

D_x = Number of features

m = Number of samples

X dimension = $D_x * m$

W_1 dimension = $D_a * D_x$

b_1 dimension = $D_a * 1$

$\Rightarrow W_1 * x + b_1$

$W_1 * x$ is of dimension = $D_a * m$

b_1 is of dimension = $D_a * 1$

We can add them using broadcasting. It allows for the element-wise addition of arrays with different shapes by automatically expanding the dimensions of the smaller array to match the dimensions of the larger one.

The bias vector is effectively expanded to have dimensions $D_a * m$, allowing for the element-wise addition with $W1*x$

b = 1 marks

$$\underline{1.2.b} \Rightarrow \hat{y} = \text{softmax}(z^{[2]})$$

$$\Rightarrow \hat{y}_k = \frac{\exp(z_k^{[2]})}{\sum_{j=1}^K \exp(z_j^{[2]})}$$

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \dots \\ \hat{y}_K \end{bmatrix}$$

\hat{y}_k is the probability of class k given the input $z^{[2]}$ & ensures that $\sum_{k=1}^K \hat{y}_k = 1$

$$\frac{\partial \hat{y}_k}{\partial z_k^{[2]}} = \frac{\partial}{\partial z_k^{[2]}} \left(\frac{\exp(z_k^{[2]})}{\sum_{j=1}^K \exp(z_j^{[2]})} \right) \quad \text{Using quotient Rule}$$

$$\Rightarrow \left[\frac{\partial}{\partial z_k^{[2]}} (\exp(z_k^{[2]})) \cdot \left(\sum_{j=1}^K \exp(z_j^{[2]}) \right) \right.$$

$$\left. - (\exp(z_k^{[2]})) \cdot \frac{\partial}{\partial z_k^{[2]}} \left(\sum_{j=1}^K \exp(z_j^{[2]}) \right) \right]$$

$$\left(\sum_{j=1}^K \exp(z_j^{[2]}) \right)^2$$

$$\Rightarrow \frac{\exp(z_k^{[2]}) \left[\sum_{j=1}^K \exp(z_j^{[2]}) \right] - \left[\exp(z_k^{[2]}) \right]^2}{\left(\sum_{j=1}^K \exp(z_j^{[2]}) \right)^2}$$

$$\Rightarrow \frac{\exp(z_k^{[2]})}{\sum_{j=1}^k \exp(z_j^{[2]})} \times \frac{\left(\sum_{j=1}^k \exp(z_j^{[2]}) - \exp(z_k^{[2]}) \right)}{\sum_{j=1}^k \exp(z_j^{[2]})}$$

$$\Rightarrow \boxed{\hat{y}_k (1 - \hat{y}_k)} = \frac{\partial \hat{y}_k}{\partial z_k^{[2]}}$$

c = 1 marks

1.2.c \Rightarrow To find, $\frac{\partial \hat{y}_k}{\partial z_i^{[2]}}$ for some $i \neq k$

$$\Rightarrow \frac{\partial \hat{y}_k}{\partial z_i^{[2]}} = \frac{\partial}{\partial z_i^{[2]}} \left[\frac{\exp(z_k^{[2]})}{\sum_{j=1}^K \exp(z_j^{[2]})} \right]$$

Using quotient Rule

$$u = \exp(z_k^{[2]}) \quad v = \sum_{j=1}^K \exp(z_j^{[2]})$$

$$\Rightarrow \frac{u'v - v'u}{v^2} \Rightarrow u' = 0 = \frac{\partial (\exp(z_k^{[2]}))}{\partial z_i^{[2]}}$$

$$\Rightarrow -\exp(z_k^{[2]}) \cdot \exp(z_i^{[2]})$$

$$\left[\sum_{j=1}^K (\exp(z_j^{[2]})) \right]^2$$

$$\Rightarrow \boxed{-\hat{y}_k \cdot \hat{y}_i}$$

d = 3 marks

(4)

1.2.d $\Rightarrow L = - \sum_{i=1}^K y_i \cdot \log(\hat{y}_i)$

$$\Rightarrow \frac{\partial L}{\partial z_i^{[2]}} = \frac{\partial L}{\partial \hat{y}_k} \times \frac{\partial \hat{y}_k}{\partial z_i^{[2]}}$$

Case 1 = when $i=k$

$$\begin{aligned} \frac{\partial L}{\partial \hat{y}_k} &= \frac{\partial}{\partial \hat{y}_k} \left(- \left(\sum_{k=1}^K y_k \log(\hat{y}_k) \right) \right) \\ &= - y_k / \hat{y}_k \end{aligned}$$

$$\frac{\partial \hat{y}_k}{\partial z_k^{[2]}} = \hat{y}_k (1 - \hat{y}_k)$$

$$\Rightarrow \boxed{\frac{\partial L}{\partial z_k^{[2]}} = - y_k (1 - \hat{y}_k)}$$

Case 2 = when $i \neq k$

$$\frac{\partial L}{\partial \hat{y}_k} = 0 \quad \text{when } i \neq k \quad y_k = 0$$

$$\frac{\partial \hat{y}_k}{\partial z_i^{[2]}} = - \hat{y}_k \cdot \hat{y}_i$$

so, $\boxed{\frac{\partial L}{\partial z_i^{[2]}} = 0}$

$$\frac{\partial L}{\partial z_i^{[2]}} = \begin{cases} -y_i (1 - \hat{y}_i) & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases}$$

e = 2 marks

Softmax function has 2 steps—

1. Finding the exponents of input values (logits)
2. Normalizing the exponent terms by finding their sum

The problems—

Overflow = Exponentiating large input values can lead to extremely large values, which may exceed the range of representable numbers in a computer's floating-point format, causing overflow errors.

Underflow = Exponentiating small input values can result in extremely small values, potentially leading to numerical precision issues due to the limited precision of floating-point arithmetic, known as underflow.

To address these numerical stability issues, a common approach is to use a modified version of the softmax function that incorporates techniques to prevent overflow and underflow.

One such technique is the log-sum-exp trick, which involves subtracting the maximum input value from each input before exponentiating them.

This ensures that the largest exponentiated value is 1, preventing overflow, while maintaining numerical stability.

for each input term z_i
do $z_i = z_i - z_{\max}$
find exponent,
 $\exp(z_i - z_{\max})$
Normalise the exponent,
 $\Rightarrow \frac{\exp(z_i - z_{\max})}{\sum_{j=1}^K \exp(z_j - z_{\max})}$

For example if we have $z = [1000, 1001, 999]$

Then we subtract 1001 from each term and z becomes, $z = [-1, 0, -2]$

Then we find the exponents, e^{-1} , e^0 and e^{-2}

And then we normal the terms.

Question 2 = 40 marks

1 = 5 marks

- I downloaded the data set
- I uploaded it to the google drive
- I made the appropriate class mapping

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Define the path to your dataset in Google Drive
data_dir = "/content/drive/MyDrive/Russian_WildLife_Dataset"

# Define class mapping
class_mapping = {
    'amur_leopard': 0,
    'amur_tiger': 1,
    'birds': 2,
    'black_bear': 3,
    'brown_bear': 4,
    'dog': 5,
    'roe_deer': 6,
    'sika_deer': 7,
    'wild_boar': 8,
    'people': 9
}
```

- Then i made the data set for the data.
- Created an instance of the class 'animal_data'

```
class AnimalData(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform
        self.dataset = ImageFolder(root=self.data_dir, transform=self.transform)

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        # Get the item from the dataset
        image, label = self.dataset[idx]

        # Move the image tensor to GPU
        image = image.to(torch.device('cuda'))

        return image, label
```

```
# Create dataset using AnimalData
animal_data = AnimalData(data_dir=data_dir, transform=transform)
```

- Then did a stratified random split of the data

```
# Define the ratio for train, validation, and test sets
train_ratio = 0.7
val_ratio = 0.1
test_ratio = 0.2

# Calculate the sizes of each set
num_data = len(animal_data)
num_train = int(train_ratio * num_data)
num_val = int(val_ratio * num_data)
num_test = num_data - num_train - num_val

# Use random_split to split the dataset
train_data, val_data, test_data = random_split(animal_data, [num_train, num_val, num_test])

batch_size = 32

# Move datasets to GPU
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)

# Function to move data loader to GPU
def move_to_gpu(data_loader):
    for batch in data_loader:
        yield [item.to(torch.device('cuda')) for item in batch]

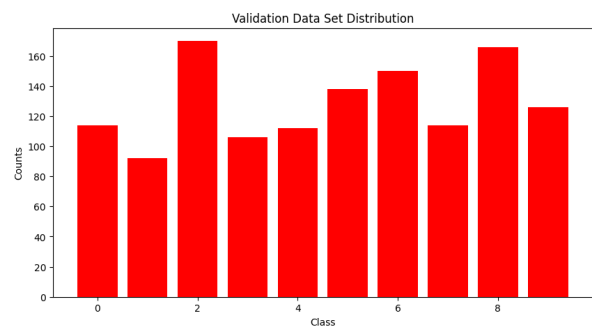
# Move data loaders to GPU
train_loader = move_to_gpu(train_loader)
val_loader = move_to_gpu(val_loader)
test_loader = move_to_gpu(test_loader)

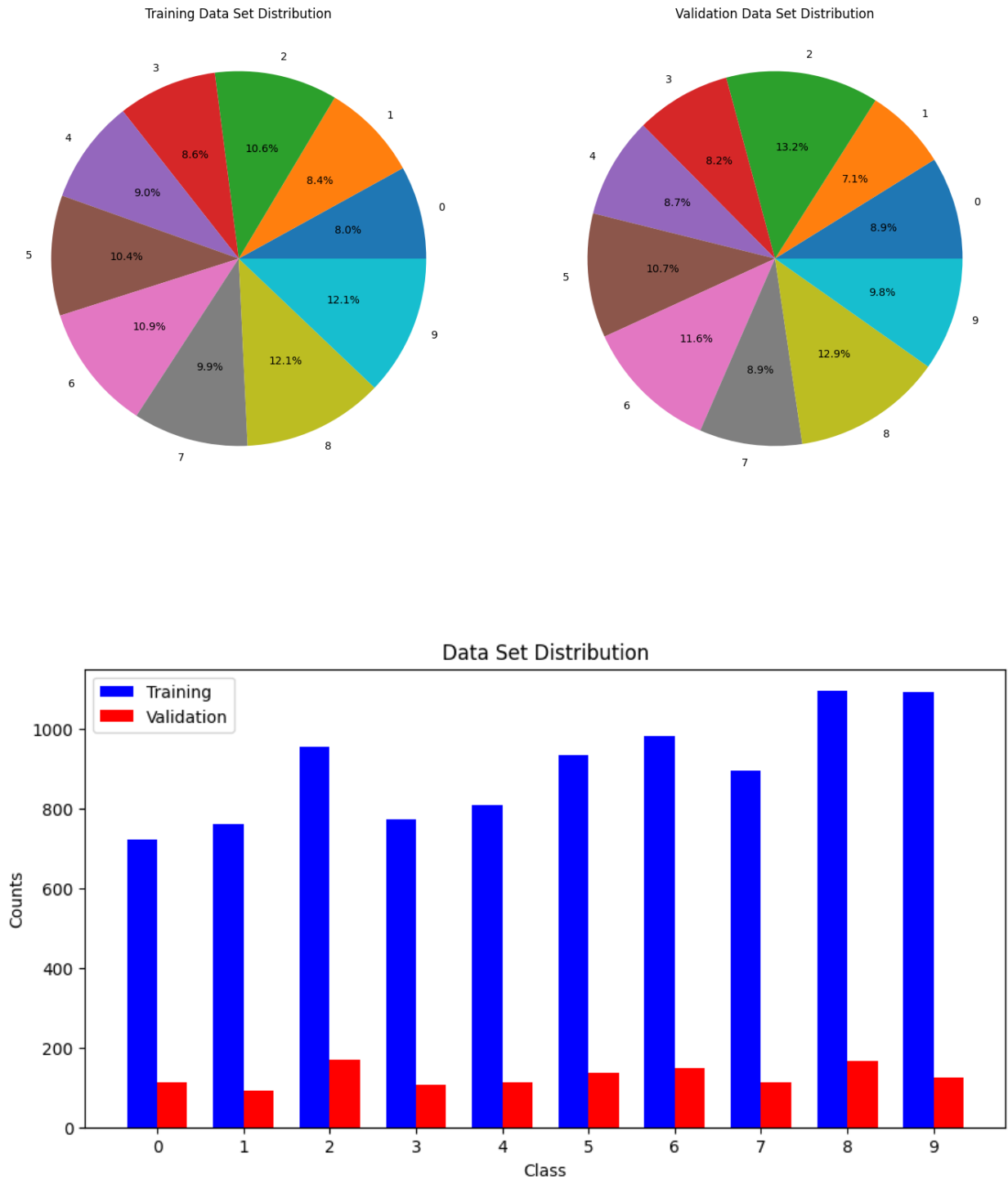
# Print the sizes of each set
print("Train set size:", len(train_data))
print("Validation set size:", len(val_data))
print("Test set size:", len(test_data))
```

- Created data loaders for all the splits (train, val, test) using PyTorch

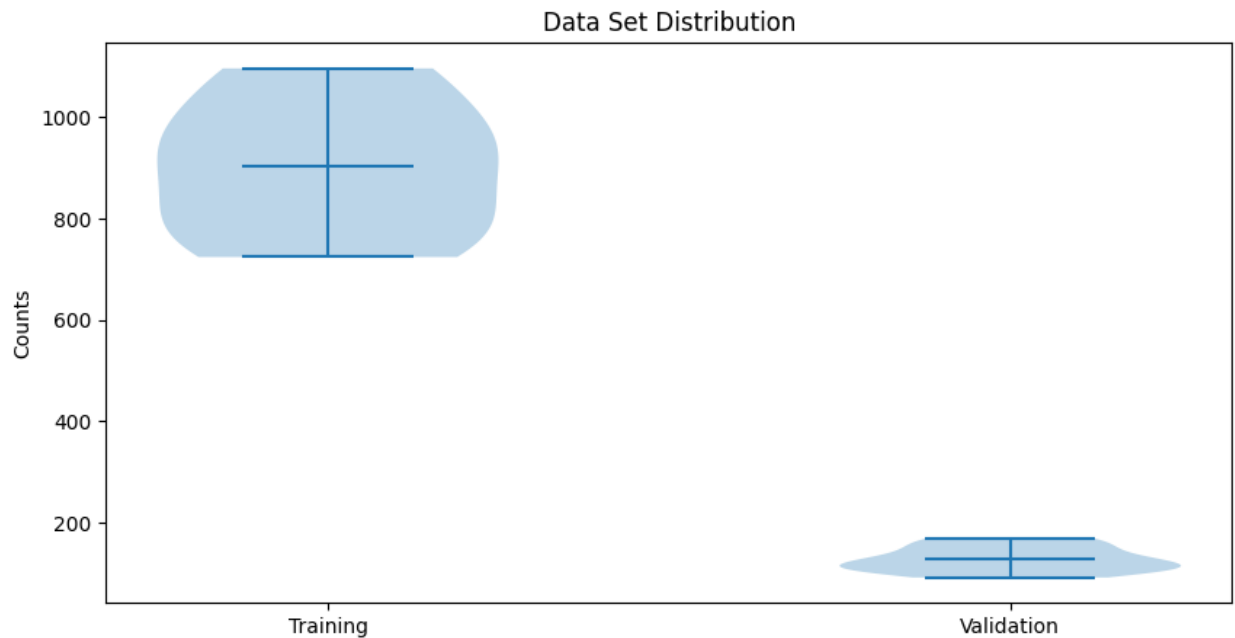
Visualisation

```
✓ 0s global_training_class_counts, global_validation_class_counts  
[724, 760, 956, 772, 808, 936, 984, 896, 1092, 1092],  
[114, 92, 170, 106, 112, 138, 150, 114, 166, 126]
```





Violin Plot = Shows that the training data is spread more uniformly as compared to the validation data, which is why the shape of the training data is more close to circular than validation data.



2 = 10 marks = CNN from scratch

Convolutional Layers:

- Three convolutional layers are defined (`self.conv1` , `self.conv2` , `self.conv3`)
- Each convolutional layer has a kernel size of 3×3 , padding of 1, and stride of 1
- The number of feature maps for the first layer is 32, for the second layer is 64, and for the third layer is 128

Max Pooling Layers:

- Three max pooling layers are defined (`self.maxpool1` , `self.maxpool2` , `self.maxpool3`)
- The first max pooling layer has a kernel size of 4×4 and a stride of 4 after the first convolutional layer

- The second and third max pooling layers have a kernel size of 2×2 and a stride of 2 after the second and third convolutional layers, respectively

Flattening and Classification Head:

- The output of the final max pooling layer is flattened using `x.view(-1, 128 * 16 * 16)` (assuming the input image size is 32×32)
- A fully connected layer (`self.fc`) with 10 output units (assuming there are 10 classes for classification) is added as the classification head on top of the flattened output.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # First Convolutional Layer: 3x3 kernel, 32 input channels, 32 output channels
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1)
        # Max pooling layer: 4x4 kernel, stride 4
        self.maxpool1 = nn.MaxPool2d(kernel_size=4, stride=4)

        # Second Convolutional Layer: 3x3 kernel, 32 input channels, 64 output channels
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        # Max pooling layer: 2x2 kernel, stride 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Third Convolutional Layer: 3x3 kernel, 64 input channels, 128 output channels
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        # Max pooling layer: 2x2 kernel, stride 2
        self.maxpool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layer
        self.fc = nn.Linear(128 * 16 * 16, 10) # Corrected input size

    def forward(self, x):
        # First Convolutional Layer
        x = F.relu(self.conv1(x))
        x = self.maxpool1(x)

        # Second Convolutional Layer
        x = F.relu(self.conv2(x))
        x = self.maxpool2(x)

        # Third Convolutional Layer
        x = F.relu(self.conv3(x))
        x = self.maxpool3(x)

        # Flatten the output for the fully connected layer
        x = x.view(-1, 128 * 16 * 16) # Corrected input size

        # Fully connected layer
        x = self.fc(x)
        return x
```

```
# Initialize WandB
wandb.init(project="Q2_2", entity="mjzeus1729", name="computer_vision_hw_1", group="mjzeus1729")

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# Define the number of epochs
num_epochs = 10 # Training for 10 epochs
```

```
Epoch 1/10, Train Loss: 0.3180, Train Acc: 0.8980, Val Loss: 3.5138, Val Acc: 0.4386
Epoch 2/10, Train Loss: 0.2140, Train Acc: 0.9304, Val Loss: 3.7392, Val Acc: 0.4576
Epoch 3/10, Train Loss: 0.1709, Train Acc: 0.9467, Val Loss: 4.3590, Val Acc: 0.4428
Epoch 4/10, Train Loss: 0.1748, Train Acc: 0.9404, Val Loss: 4.5239, Val Acc: 0.4619
Epoch 5/10, Train Loss: 0.2596, Train Acc: 0.9171, Val Loss: 3.8876, Val Acc: 0.4555
Epoch 6/10, Train Loss: 0.2014, Train Acc: 0.9358, Val Loss: 4.0736, Val Acc: 0.4237
Epoch 7/10, Train Loss: 0.1486, Train Acc: 0.9534, Val Loss: 4.7092, Val Acc: 0.4258
Epoch 8/10, Train Loss: 0.1265, Train Acc: 0.9604, Val Loss: 5.1831, Val Acc: 0.4449
Epoch 9/10, Train Loss: 0.0990, Train Acc: 0.9709, Val Loss: 5.5324, Val Acc: 0.4428
Epoch 10/10, Train Loss: 0.1419, Train Acc: 0.9594, Val Loss: 5.4187, Val Acc: 0.4216
```

Run history:



Run summary:

Epoch	10
Train Accuracy	0.95944
Train Loss	0.14187
Validation Accuracy	0.42161
Validation Loss	5.4187

View run **computer_vision_hw_1** at: https://wandb.ai/mjzeus1729/Q2_2/runs/45qfz5ks

Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `./wandb/run-20240217_143701-45qfz5ks/logs`

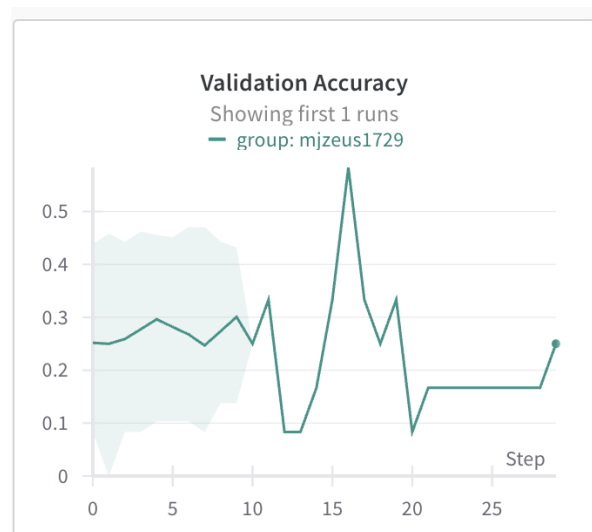
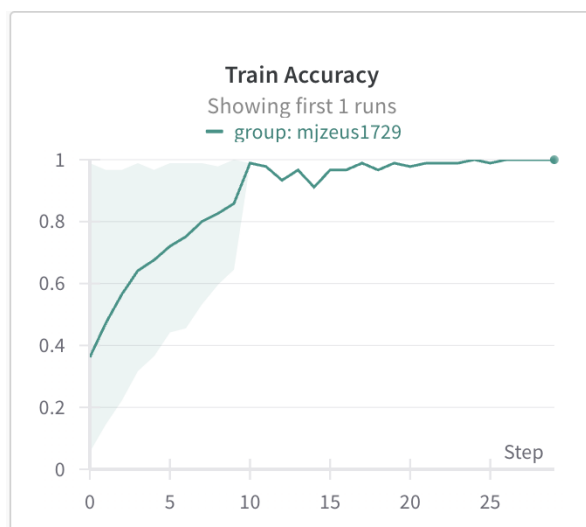
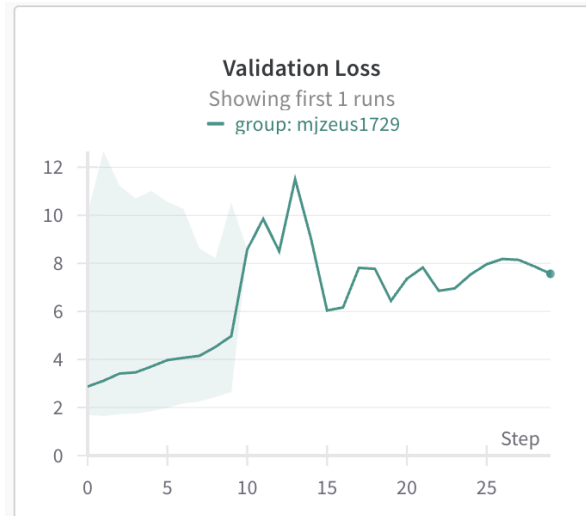
Training completed!

Final Train Loss: 0.1419, Final Train Acc: 0.9594, Final Val Loss: 5.4187, Final Val Acc: 0.4216

Plots

Training Data

Validation Data



- The training accuracy consistently increases with each epoch, reaching a high value of around 0.9594 by the final epoch.
- The validation accuracy, however, fluctuates and does not exhibit a consistent increasing trend. It oscillates around a certain range without showing significant improvement over epochs.
- The validation loss also shows an increasing trend over epochs, which suggests that the model's performance on unseen data is deteriorating as training progresses.

Given these observations, it appears that there is a case of overfitting occurring in the model. The model is learning to perform well on the training data but struggles to generalize to unseen validation data, as evidenced by the decreasing validation accuracy and increasing validation loss.

Testing = Accuracy = 0.4417

Testing = F1-Score = 0.4380



3 = 10 marks = resplendent-noodles-11

I trained another classifier on resnet 18 using the same strategy as above.

Used Cross Entropy Loss and Adam Optimiser

Initialised wandb.

```
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', pretrained=True)
model.eval()
```

```
# Define the loss function
criterion = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

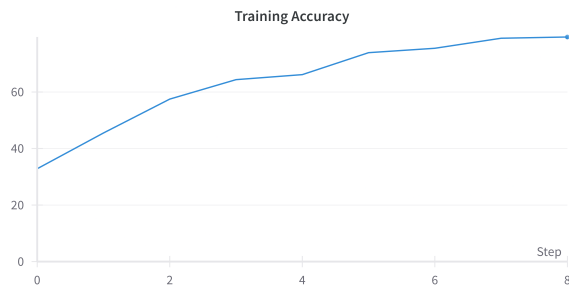
# Number of epochs
num_epochs = 10
```

```
# Initialize WandB with your project name and optionally specify the entity
wandb.init(project="question_2_computer_vision", entity="mjzeus1729")
```

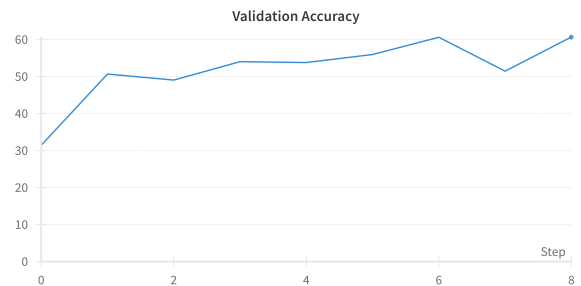
Epoch	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
Epoch 1/10	3.5657	32.85%	6.7021	31.44%
Epoch 2/10	1.7732	45.50%	1.4538	50.70%
Epoch 3/10	1.2434	57.49%	1.6321	49.07%
Epoch 4/10	1.0979	64.37%	1.4864	54.04%
Epoch 5/10	1.0332	66.15%	1.6345	53.80%
Epoch 6/10	0.8249	73.92%	1.4630	55.98%
Epoch 7/10	0.6791	75.47%	1.4811	60.64%
Epoch 8/10	0.6494	79.02%	1.8660	51.48%
Epoch 9/10	0.5825	79.47%	1.3316	60.71%
Epoch 10/10	0.3734	88.12%	1.6245	58.54%

Plots

Training Data



Validation Data



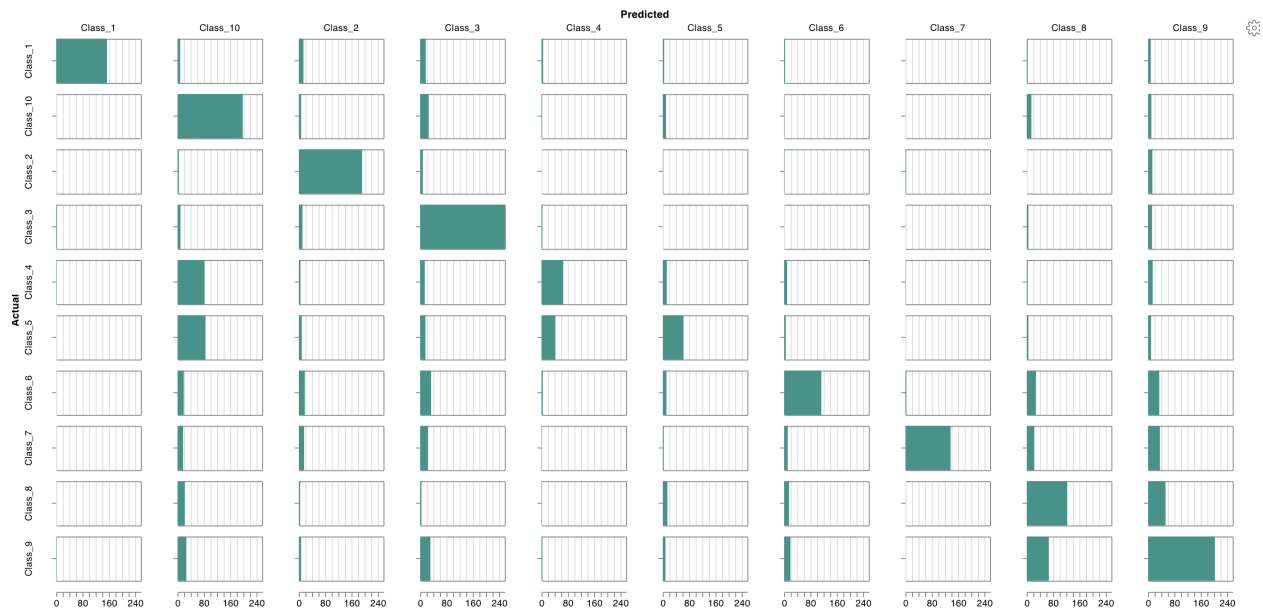
The model does not seem to be overfitting.

- Training loss and validation loss are decreasing. In overfitting scenarios, we typically see the training loss decreasing significantly while the validation loss starts increasing, indicating that the model is fitting too closely to the training data and failing to generalize to unseen data. However, in this case, both the training and validation losses are decreasing, suggesting that the model is learning effectively without showing signs of overfitting.

- The accuracies are going up, which also indicates that the model might not be overfitting.

Testing Data and Confusion Matrix

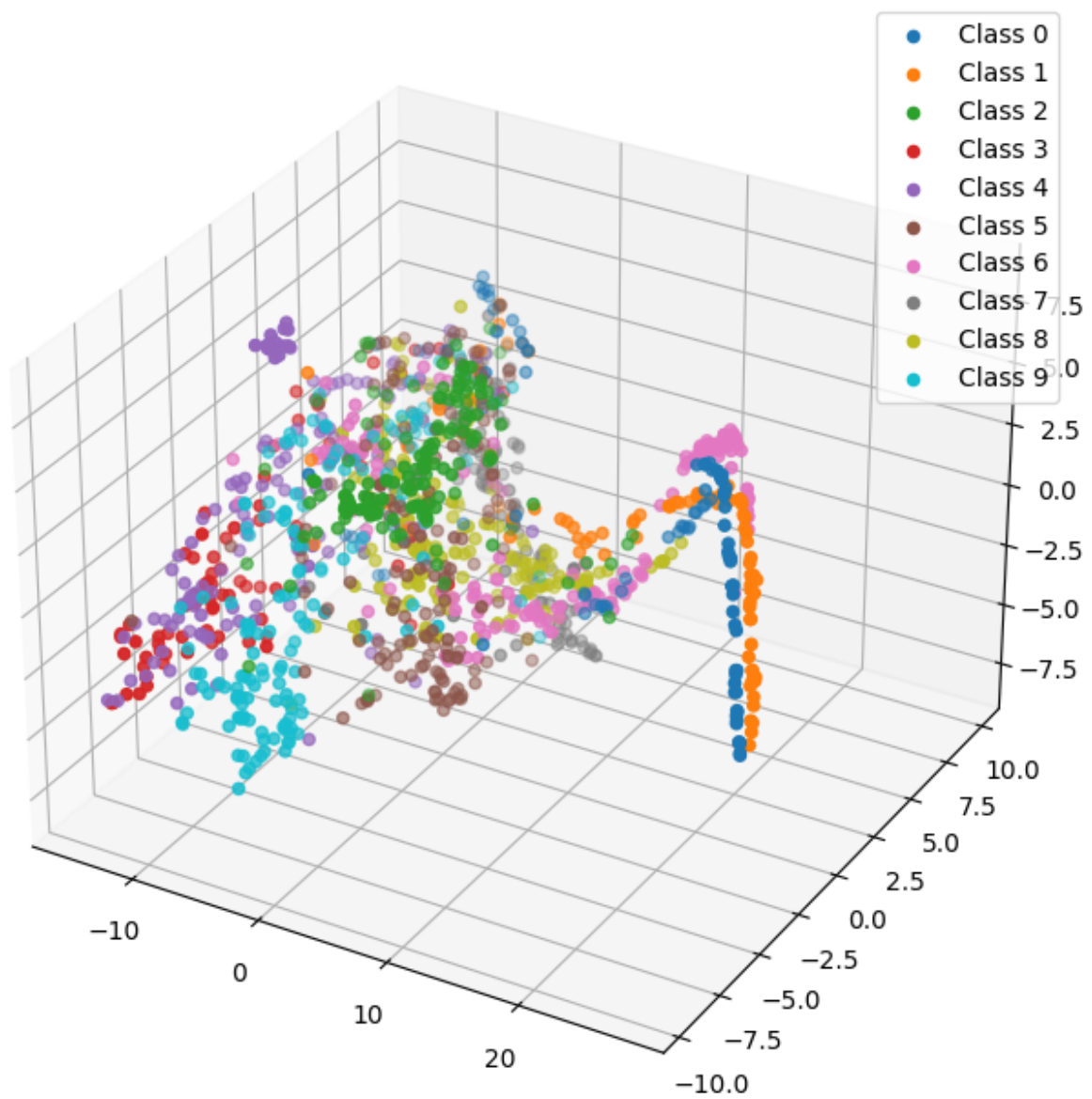
Test Accuracy: 58.42%
Test F1-Score: 0.5773



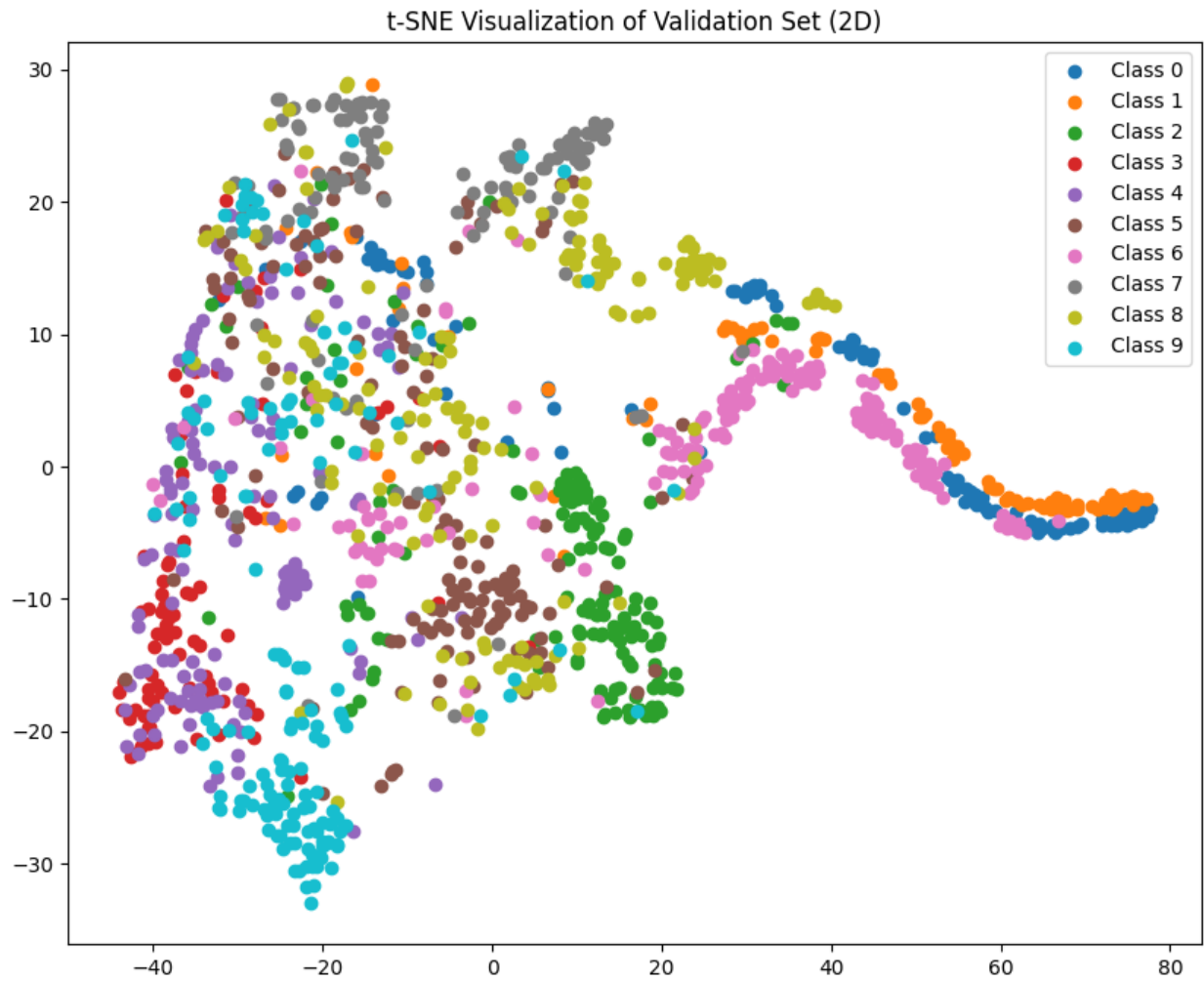
Feature Extraction

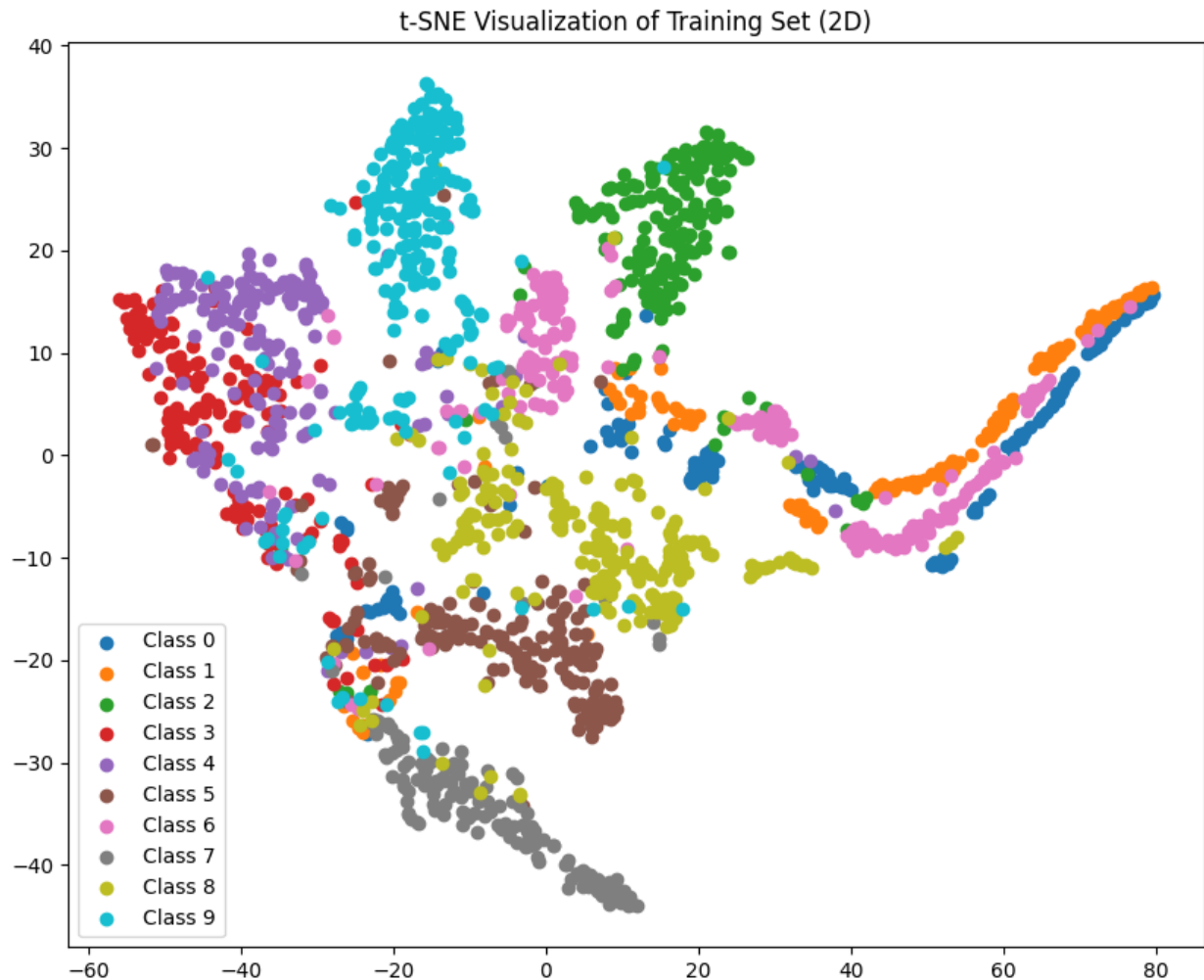
3D

t-SNE Visualization of Validation Set (3D)



2D





4 = 10 marks = golden-snake-12

Data Augmentation Techniques

I have used three augmentation techniques

- Random Crop
- Random Perspective
- Random Saturation

```
transform = transforms.Compose([
    transforms.RandomPerspective(distortion_scale=0.5), # Random perspective
    transforms.ColorJitter(brightness=0.5, contrast=0.5), # Random contrast
    transforms.ColorJitter(saturation=0.5), # Random saturation
    transforms.Resize(image_size), # Resize images to 128x128 pixels
    transforms.RandomCrop(size=(image_size, image_size)), # Random crop
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # Mean and std for a single channel
])
```

I trained another classifier on resnet 18 using the same strategy as above.

Used Cross Entropy Loss and Adam Optimiser

Initialised wandb.

```
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', pretrained=True)
model.eval()
```

```
# Define the loss function
criterion = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

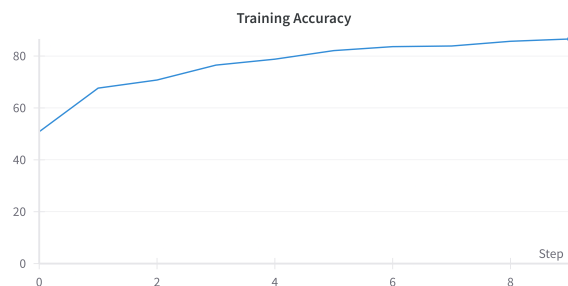
# Number of epochs
num_epochs = 10
```

```
# Initialize WandB with your project name and optionally specify the entity
wandb.init(project="question_2_computer_vision", entity="mjzeus1729")
```

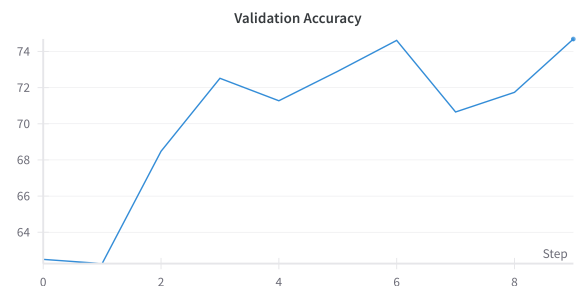
```
Epoch 1/10, Training Loss: 1.8625, Training Accuracy: 50.83%, Validation Loss: 1.0333, Validation Accuracy: 62.50%
Epoch 2/10, Training Loss: 0.9550, Training Accuracy: 67.64%, Validation Loss: 1.1549, Validation Accuracy: 62.27%
Epoch 3/10, Training Loss: 0.8642, Training Accuracy: 70.77%, Validation Loss: 0.9600, Validation Accuracy: 68.48%
Epoch 4/10, Training Loss: 0.6932, Training Accuracy: 76.51%, Validation Loss: 0.7935, Validation Accuracy: 72.52%
Epoch 5/10, Training Loss: 0.6181, Training Accuracy: 78.80%, Validation Loss: 0.8791, Validation Accuracy: 71.27%
Epoch 6/10, Training Loss: 0.5184, Training Accuracy: 82.10%, Validation Loss: 0.8230, Validation Accuracy: 72.90%
Epoch 7/10, Training Loss: 0.4901, Training Accuracy: 83.63%, Validation Loss: 0.8384, Validation Accuracy: 74.61%
Epoch 8/10, Training Loss: 0.4764, Training Accuracy: 83.88%, Validation Loss: 0.9536, Validation Accuracy: 70.65%
Epoch 9/10, Training Loss: 0.4106, Training Accuracy: 85.70%, Validation Loss: 0.9481, Validation Accuracy: 71.74%
Epoch 10/10, Training Loss: 0.4042, Training Accuracy: 86.58%, Validation Loss: 0.7884, Validation Accuracy: 74.69%
```

Plots

Training Data



Validation Data

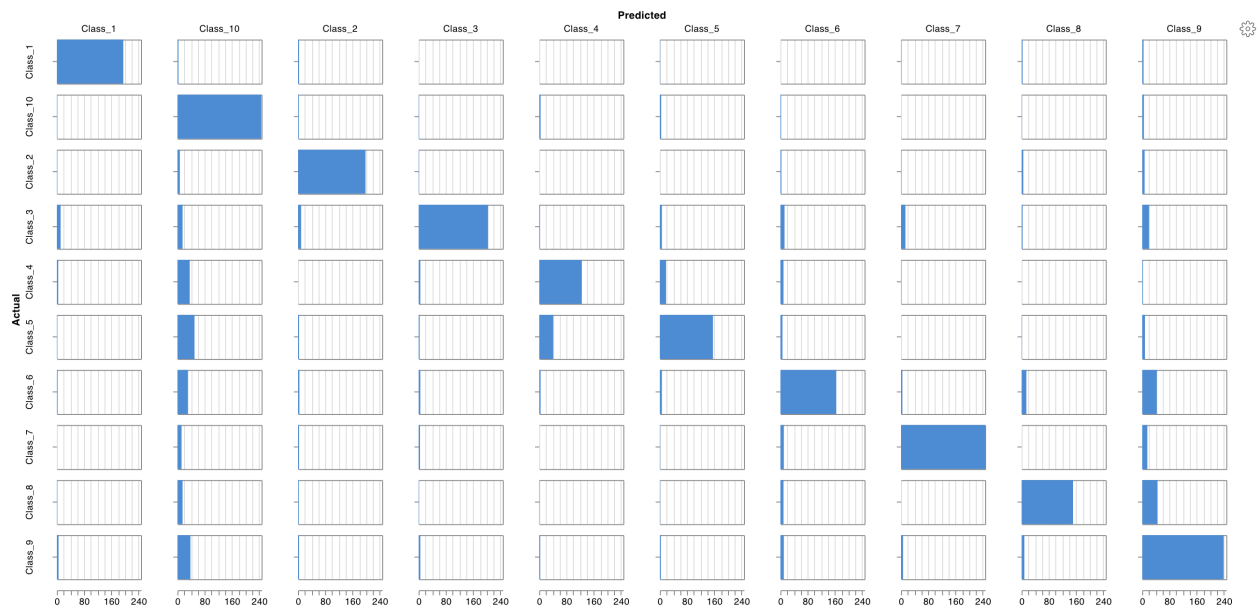


There is no overfitting.

- The training loss and validation losses are decreasing.
- The accuracies are increasing.

Testing Data and Confusion Matrix

Test Accuracy: 74.67%
Test F1-Score: 0.7513



6 = 5 marks

1. CNN Architecture:

- Achieved a high training accuracy of around 95.94%, indicating good learning capability on the training data.
- However, exhibited overfitting as indicated by fluctuating validation accuracy and increasing validation loss, suggesting poor generalization to

unseen data.

- Testing accuracy was relatively low at 44.17%, indicating poor performance on unseen test data.

2. Fine-tuned ResNet Model:

- Testing accuracy improved significantly to 58.42% compared to the CNN model, indicating better generalization.
- F1-score also improved to 0.57, suggesting improved performance in correctly classifying the test data.
- No signs of overfitting observed, as both training and validation losses decreased, and accuracies improved consistently.

3. ResNet Model with Augmentation:

- Utilized additional data augmentation techniques such as random crop, random perspective, and random saturation.
- Testing accuracy further improved to 75%, indicating the effectiveness of data augmentation in enhancing model performance.
- Achieved an F1-score of 0.75, indicating a balanced performance in precision and recall.
- No signs of overfitting observed, with both training and validation losses decreasing and accuracies consistently increasing.

4. General Comparison:

- The CNN model exhibited significant overfitting, with poor performance on unseen data.
- Fine-tuning a pre-trained ResNet model improved generalization and performance on the test data compared to the CNN model.
- Incorporating data augmentation further improved model performance, resulting in higher testing accuracy and F1-score.
- The ResNet model with augmentation demonstrated the best performance among the three models, suggesting the effectiveness of leveraging pre-trained architectures and data augmentation techniques for image classification tasks.

- The consistent decrease in training and validation losses, along with increasing accuracies, indicates successful training and generalization of the models.
- Overall, the ResNet model with augmentation outperformed both the CNN architecture and the fine-tuned ResNet model, highlighting the importance of leveraging advanced architectures and data augmentation for improving model performance and generalization.

5. **Model Complexity:**

- The CNN architecture was relatively simple compared to the ResNet models, with only three convolutional layers and max-pooling layers.
- Fine-tuning a pre-trained ResNet model increased model complexity by leveraging a deeper architecture pre-trained on a large dataset like ImageNet.
- The ResNet model with augmentation maintained the complexity of the fine-tuned ResNet model but further enhanced its performance through data augmentation, effectively increasing the effective size of the training dataset.

6. **Computational Efficiency:**

- The CNN architecture trained relatively quickly due to its simplicity, but suffered from overfitting and poor generalization.
- Fine-tuning a pre-trained ResNet model required more computational resources for training due to its deeper architecture and larger parameter space.
- The ResNet model with augmentation also required significant computational resources due to both the architecture complexity and the additional overhead of data augmentation during training.

7. **Impact of Data Augmentation:**

- Data augmentation played a crucial role in improving the performance of the ResNet model, enabling it to achieve a higher testing accuracy and F1-score compared to the fine-tuned ResNet model without augmentation.
- Augmentation helped mitigate overfitting by providing more diverse training samples and enhancing the model's ability to generalize to unseen data.