Task 1 - Grid Game

January 1, 2023

1 Task 1 - Grid Game

[1]:
```
# Import libraries
import matplotlib.pyplot as plt
import numpy as np

# Set the seed for the random integer generators
np.random.seed(46853)
```

[2]:
```
# Prompt the player to enter the grid size
grid_width = int(input("Enter the grid width: "))

# Prompt the player to enter the grid size
grid_height = int(input("Enter the grid height: "))

# Prompt the player to enter the distribution parameters
distribution_mean = float(input("Enter the distribution mean: "))
distribution_stddev = float(input("Enter the distribution standard deviation:␣
↪"))
```

```
Enter the grid width: 15
Enter the grid height: 11
Enter the distribution mean: 4
Enter the distribution standard deviation: 5
```

[3]:
```
# Generate a grid with random cell values using a normal distribution based on␣
↪the user input
# and generate the integers to be between 0 and 9
grid = [[np.around(np.clip(np.random.normal(distribution_mean,␣
↪distribution_stddev), 0, 9)).astype(int)
        for _ in range(grid_height)] for _ in range(grid_width)]

# Define the start and goal positions
start = (0, 0)
goal = (grid_width - 1, grid_height - 1)
```

[4]:
```
# Simple Heuristic Algorithm - Not used but developed into the Dijkstra␣
↪algorithm below
```

```python
def heuristic_algorithm(beginning, end):
    # Create a set to store the visited nodes
    visited_nodes = set()

    # Create a queue for the nodes to visit and add the start node to the queue
    queue = [beginning]

    # Create a dictionary to store the came_from values of the nodes
    came_from_dict = {beginning: None}

    # While the queue is not empty
    while queue:
        # Get the next node in the queue
        current_node = queue.pop(0)

        # If the current node is the goal, return the came_from dictionary
        if current_node == end:
            return came_from_dict

        # Add the current node to the visited set
        visited_nodes.add(current_node)

        # For each neighbour of the current node
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            # Calculate the position of the neighbour
            x_pos, y_pos = current_node
            neighbour = (x_pos + dx, y_pos + dy)

            # If the neighbour is out of bounds or has been visited, skip it
            if not (0 <= x_pos + dx < grid_width) or not (0 <= y_pos + dy <␣
 ↪grid_height) or neighbour in visited_nodes:
                continue

            # Set the came_from value of the neighbour to the current node
            came_from_dict[neighbour] = current_node

            # Add the neighbour to the queue
            queue.append(neighbour)
```

[7]:
```python
# Dijkstra Algorithm developed from the Heuristic Algorithm
def dijkstra(beginning, end):
    # Create a set to store the visited nodes
    visited_nodes = set()

    # Create a min-priority queue for the nodes to visit and add the start node␣
 ↪to the queue with a priority of 0
    queue = [(0, beginning)]
```

```python
    # Create a dictionary to store the costs of the nodes
    cost_dict = {beginning: 0}

    # Create a dictionary to store the came_from values of the nodes
    came_from_dict = {beginning: None}

    # While the queue is not empty
    while queue:
        # Sort the queue in ascending order by cost
        queue.sort(key=lambda x_sort: x_sort[0])

        # Get the node with the lowest priority
        cost, current_node = queue.pop(0)

        # If the current node is the goal, return the came_from dictionary
        if current_node == end:
            return came_from_dict

        # Add the current node to the visited set
        visited_nodes.add(current_node)

        # For each neighbour of the current node
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            # Calculate the position of the neighbour
            x_pos, y_pos = current_node
            neighbour = (x_pos + dx, y_pos + dy)

            # If the neighbour is out of bounds or has been visited, skip it
            if not (0 <= x_pos + dx < grid_width) or not (0 <= y_pos + dy <
↪grid_height) or neighbour in visited_nodes:
                continue

            # Calculate the cost to reach the neighbour
            cost = cost_dict[current_node] + grid[x_pos][y_pos]

            # If the cost is lower than the current cost of the neighbour
            if cost < cost_dict.get(neighbour, float('inf')):
                # Update the cost of the neighbour
                cost_dict[neighbour] = cost

                # Add the neighbour to the queue with the calculated cost
                queue.append((cost, neighbour))

                # Set the came_from value of the neighbour to the current node
                came_from_dict[neighbour] = current_node
```

[11]:

```python
# Find the shortest path using Dijkstra's algorithm
came_from = dijkstra(start, goal)

# Create a figure with a specified size
fig = plt.figure(figsize=(grid_width, grid_height))

# Rotate the grid to correct the position of the grid
grid = np.rot90(grid, k=-1)

# Display the grid as an image with no shading
plt.imshow(grid, cmap=None, alpha=0)

# Rotate the grid back to correct the position of the path
grid = np.rot90(grid)

# Display the numbers in the center of each grid square
for i in range(grid_width):
    for j in range(grid_height):
        plt.text(i, j, grid[i][j], ha="center", va="center", color="black",␣
 ↪fontsize=20)

# Reconstruct the path from the came_from dictionary
path = []
current = goal
while current != start:
    path.append(current)
    current = came_from[current]
path.append(start)

# Get the coordinates of the points on the path
x, y = zip(*path)

# Draw the path on the grid
plt.plot(x, y, c="green", linewidth=3, alpha=0.8)

# Turn off the x-axis tick marks and labels
plt.tick_params(axis='x', which='both', bottom=False, top=False,␣
 ↪labelbottom=False)

# Turn off the y-axis tick marks and labels
plt.tick_params(axis='y', which='both', left=False, right=False,␣
 ↪labelleft=False)

# Show the plot
plt.show()
```

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

Task 2 - MNIST Dataset Neural Network

January 1, 2023

1 Task 2 - MNIST Dataset Neural Network

[1]:
```
# Import libraries
import numpy as np
from keras.datasets import mnist
from matplotlib import pyplot as plt

# Set random seed for reproducibility
np.random.seed(46853)
```

[2]:
```
# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Preprocess the data
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# Convert the labels to categorical variables
y_train = np.eye(10)[y_train]
y_test = np.eye(10)[y_test]

# Initialize the weights and biases
weights = {}
biases = {}

# Initialize the input layer weights and biases
weights['w1'] = np.random.randn(784, 512)
biases['b1'] = np.zeros((1, 512))

# Initialize the hidden layer weights and biases
weights['w2'] = np.random.randn(512, 512)
biases['b2'] = np.zeros((1, 512))
```

```
# Initialize the output layer weights and biases
weights['w3'] = np.random.randn(512, 10)
biases['b3'] = np.zeros((1, 10))

# Set the learning rate
learning_rate = 0.01

# Set the dropout rate
dropout_rate = 0.5

# Set the number of epochs
num_epochs = 100

# Set the batch size
batch_size = 8

# Set the number of batches
num_batches = X_train.shape[0] // batch_size

# Create arrays for the accuracy and loss measurements
mean_accuracies = []
mean_losses = []
```

[3]:
```
# Define sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Define the SGD optimiser
def sgd_optimiser(grad_w1, grad_b1, grad_w2, grad_b2, grad_w3, grad_b3,␣
 ↪learning_rate, weights, biases):
    weights['w1'] -= learning_rate * grad_w1
    biases['b1'] -= learning_rate * grad_b1
    weights['w2'] -= learning_rate * grad_w2
    biases['b2'] -= learning_rate * grad_b2
    weights['w3'] -= learning_rate * grad_w3
    biases['b3'] -= learning_rate * grad_b3

# Calculate the cross-entropy loss
def cross_entropy_loss(y, y_pred, smoothing=1e-8):
    # Add the smoothing factor to the prediction to avoid division by zero
    y_pred = y_pred + smoothing
    return -np.sum(y * np.log(y_pred))
```

[4]:
```
# Define the test for the sigmoid function
def test_sigmoid_layer():
    # Test 1: Check that the sigmoid function returns the expected output
    z = np.array([[-2, -1, 0, 1, 2]])
```

```
    expected_output = np.array([[0.119203, 0.268941, 0.5, 0.731059, 0.880797]])
    output = sigmoid(z)
    assert np.allclose(output, expected_output), f"Expected {expected_output},␣
 ↪but got {output}"

    # Test 2: Check that the sigmoid function is stable when the input is very␣
 ↪large
    z = np.array([[100, 1000, 10000]])
    expected_output = np.array([[1, 1, 1]])
    output = sigmoid(z)
    assert np.allclose(output, expected_output), f"Expected {expected_output},␣
 ↪but got {output}"

# Run the test suite
test_sigmoid_layer()
```

[5]:
```
# Train the model
for epoch in range(num_epochs):
    # Shuffle the training data
    permutation = np.random.permutation(X_train.shape[0])
    X_train = X_train[permutation]
    y_train = y_train[permutation]

    # Loop over the batches
    for batch in range(num_batches):
        # Get the batch data
        start = batch * batch_size
        end = start + batch_size
        X_batch = X_train[start:end]
        y_batch = y_train[start:end]

        # Forward propagation
        z1 = X_batch.dot(weights['w1']) + biases['b1']
        a1 = sigmoid(z1)

        # Implement dropout
        mask = np.random.binomial(1, 1 - dropout_rate, size=a1.shape)
        a1 *= mask

        z2 = a1.dot(weights['w2']) + biases['b2']
        a2 = sigmoid(z2)

        # Implement dropout
        mask = np.random.binomial(1, 1 - dropout_rate, size=a2.shape)
        a2 *= mask

        z3 = a2.dot(weights['w3']) + biases['b3']
```

```
        a3 = sigmoid(z3)

        # Calculate the loss
        loss = cross_entropy_loss(y_batch, a3)

        # Backward propagation
        grad_z3 = a3 - y_batch
        grad_w3 = a2.T.dot(grad_z3) / batch_size
        grad_b3 = np.sum(grad_z3, axis=0, keepdims=True) / batch_size

        grad_a2 = grad_z3.dot(weights['w3'].T)

        # Scale the gradients by the inverse of the dropout rate
        grad_a2 *= 1.0 / (1.0 - dropout_rate)

        grad_z2 = grad_a2 * sigmoid(z2) * (1 - sigmoid(z2))
        grad_w2 = a1.T.dot(grad_z2) / batch_size
        grad_b2 = np.sum(grad_z2, axis=0, keepdims=True) / batch_size

        grad_a1 = grad_z2.dot(weights['w2'].T)

        # Scale the gradients by the inverse of the dropout rate
        grad_a1 *= 1.0 / (1.0 - dropout_rate)

        grad_z1 = grad_a1 * sigmoid(z1) * (1 - sigmoid(z1))
        grad_w1 = X_batch.T.dot(grad_z1) / batch_size
        grad_b1 = np.sum(grad_z1, axis=0, keepdims=True) / batch_size

        # Update the weights and biases
        sgd_optimiser(grad_w1, grad_b1, grad_w2, grad_b2, grad_w3, grad_b3,␣
↪learning_rate, weights, biases)

        # print statistics
        if batch % 1000 == 999:  # print every 2000 mini-batches
            print("Batch: " + str(batch))
            running_loss = 0.0

    print("Epochs:", epoch + 1)

    # Calculate the accuracy on the test set
    z1 = X_test.dot(weights['w1']) + biases['b1']
    a1 = sigmoid(z1)
    z2 = a1.dot(weights['w2']) + biases['b2']
    a2 = np.maximum(z2, 0)  # ReLU activation function
    z3 = a2.dot(weights['w3']) + biases['b3']

    # Shift the input values down to prevent overflow
```

```
    z3_shift = z3 - np.max(z3, axis=1, keepdims=True)

    # Calculate the softmax activations
    a3 = np.exp(z3_shift) / np.sum(np.exp(z3_shift), axis=1, keepdims=True)

    loss = cross_entropy_loss(y_test, a3)
    mean_loss = loss / len(y_test)
    mean_losses.append(mean_loss)

    # Calculate the predictions
    predictions = a3.argmax(axis=1)

    # Calculate the number of correct predictions
    num_correct = (predictions == y_test.argmax(axis=1)).sum()

    # Calculate the mean accuracy
    accuracy = num_correct / len(predictions)

    # Store the accuracy in a list
    mean_accuracies.append(accuracy)

    # Print the accuracy
    print(f'Accuracy: {accuracy:.4f}')
```

```
Batch: 999
Batch: 1999
Batch: 2999
Batch: 3999
Batch: 4999
Batch: 5999
Batch: 6999
Epochs: 1
Accuracy: 0.8565
Batch: 999
Batch: 1999
Batch: 2999
Batch: 3999
Batch: 4999
Batch: 5999
Batch: 6999
Epochs: 2
Accuracy: 0.8977
Batch: 999
Batch: 1999
Batch: 2999
Batch: 3999
Batch: 4999
Batch: 5999
```

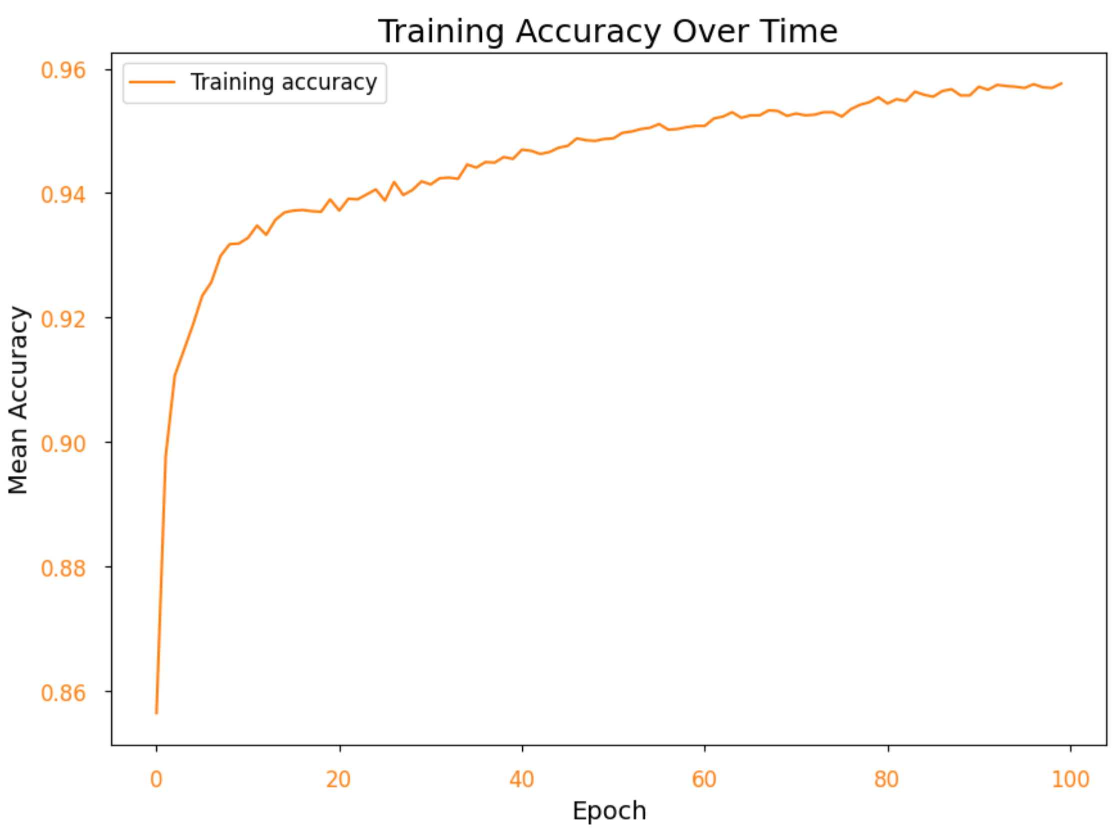
```
Batch: 6999
Epochs: 99
Accuracy: 0.9569
Batch: 999
Batch: 1999
Batch: 2999
Batch: 3999
Batch: 4999
Batch: 5999
Batch: 6999
Epochs: 100
Accuracy: 0.9576
```

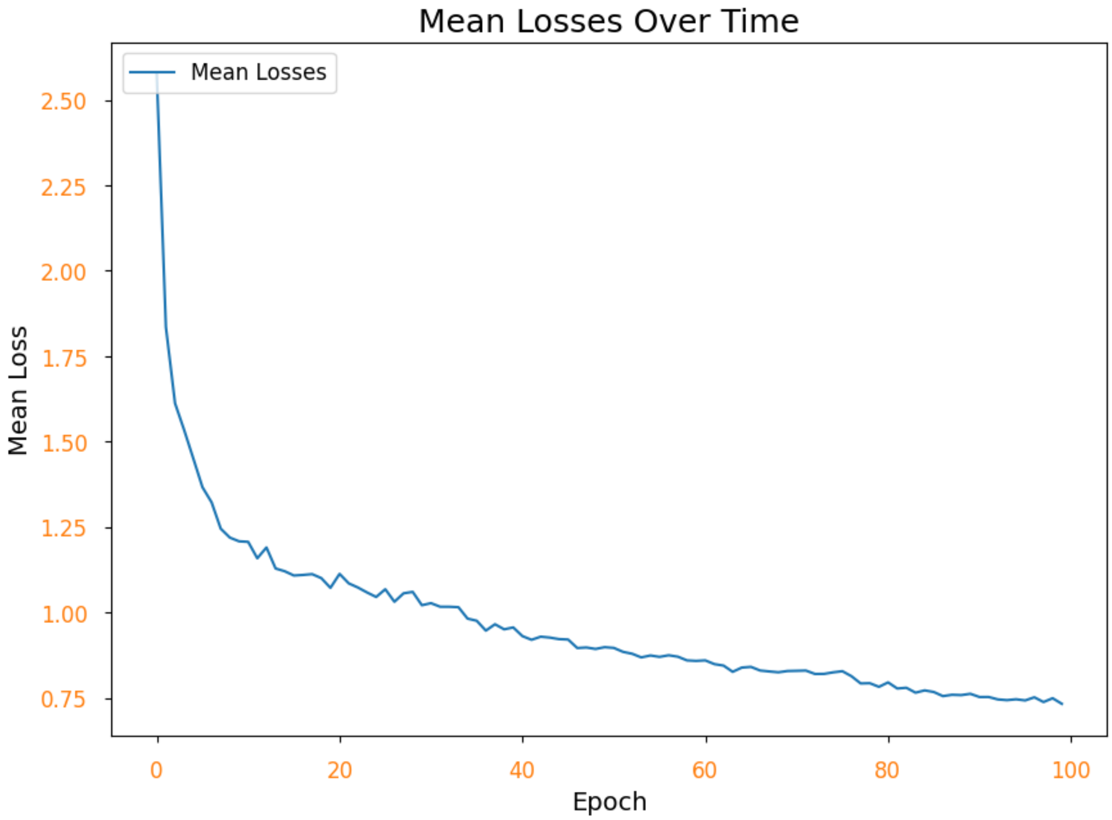
[16]:
```python
# Calculate the overall accuracy
overall_accuracy = sum(mean_accuracies) / len(mean_accuracies)

# Print the overall accuracy
print((f'Overall accuracy: {overall_accuracy * 100:.1f}') + "%")

# Calculate the overall loss
overall_loss = sum(mean_losses) / len(mean_losses)

# Print the overall mean loss
print((f'Overall mean loss: {overall_loss:.2f}') + "%")
```

```
Overall accuracy: 94.5%
Overall mean loss: 0.97%
```

[7]:
```python
# Set the figure size
plt.figure(figsize=(10, 7))

# Plot the training accuracy
plt.plot(mean_accuracies, color='tab:orange', label='Training accuracy')

# Add a title and axis labels
plt.title('Training Accuracy Over Time', fontsize=18)
plt.xlabel('Epoch', fontsize=14)
plt.ylabel('Mean Accuracy', fontsize=14)

# Set the font size and style of the tick labels
plt.tick_params(axis='both', which='major', labelsize=12, labelcolor='tab:
↪orange', pad=10)

# Add a legend
plt.legend(loc='upper left', fontsize=12)

# Show the plot
plt.show()
```

```
# Set the figure size
plt.figure(figsize=(10, 7))

# Plot the training accuracy
plt.plot(mean_losses, color='tab:blue', label='Mean Losses')

# Add a title and axis labels
plt.title('Mean Losses Over Time', fontsize=18)
plt.xlabel('Epoch', fontsize=14)
plt.ylabel('Mean Loss', fontsize=14)

# Set the font size and style of the tick labels
plt.tick_params(axis='both', which='major', labelsize=12, labelcolor='tab:
 ↪orange', pad=10)

# Add a legend
plt.legend(loc='upper left', fontsize=12)

# Show the plot
plt.show()
```




Mean Losses Over Time
Mean Losses
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
Mean Loss
0
20
40
60
80
100
Epoch

Task 3 - PyTorch

January 1, 2023

1 Task 3 - PyTorch

[1]:
```python
# Import libraries
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Set random seeds for reproducibility
torch.manual_seed(46853)
np.random.seed(46853)
```

[2]:
```python
# Download and load the CIFAR-10 dataset
# dataset = torchvision.datasets.CIFAR10(root='data/', download=True,␣
 ↪transform=transforms.ToTensor())

# Download and load the FASHION-MNIST dataset
# dataset = torchvision.datasets.FashionMNIST(root='data/', download=True,␣
 ↪transform=transforms.ToTensor())

# Download and load the KMNIST dataset
dataset = torchvision.datasets.KMNIST(root='data/', download=True,␣
 ↪transform=transforms.ToTensor())

# Split the dataset into a training set and a test set
num_train = int(0.8 * len(dataset))
num_test = len(dataset) - num_train
train_dataset, test_dataset = torch.utils.data.random_split(dataset,␣
 ↪[num_train, num_test])

# Create data loaders for the training and test sets
batch_size = 8
train_loader = torch.utils.data.DataLoader(train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,␣
↪shuffle=False)
```

```
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-images-
idx3-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-images-
idx3-ubyte.gz to data/KMNIST\raw\train-images-idx3-ubyte.gz

100.0%

Extracting data/KMNIST\raw\train-images-idx3-ubyte.gz to data/KMNIST\raw

Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-labels-
idx1-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-labels-
idx1-ubyte.gz to data/KMNIST\raw\train-labels-idx1-ubyte.gz

100.0%

Extracting data/KMNIST\raw\train-labels-idx1-ubyte.gz to data/KMNIST\raw

Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-images-
idx3-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-images-
idx3-ubyte.gz to data/KMNIST\raw\t10k-images-idx3-ubyte.gz

100.0%

Extracting data/KMNIST\raw\t10k-images-idx3-ubyte.gz to data/KMNIST\raw

Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-labels-
idx1-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-labels-
idx1-ubyte.gz to data/KMNIST\raw\t10k-labels-idx1-ubyte.gz

100.0%

Extracting data/KMNIST\raw\t10k-labels-idx1-ubyte.gz to data/KMNIST\raw
```

[3]:
```
# # Network for CIFAR-10
# class CNN(nn.Module):
#     def __init__(self):
#         super(CNN, self).__init__()
#         self.conv1 = nn.Conv2d(3, 6, 5)
#         self.batch_norm1 = nn.BatchNorm2d(6)  # Batch normalization layer
#         self.pool = nn.MaxPool2d(2, 2)
#         self.conv2 = nn.Conv2d(6, 16, 5)
#         self.batch_norm2 = nn.BatchNorm2d(16)  # Batch normalization layer
#         self.fc1 = nn.Linear(16 * 5 * 5, 120)
```

```
#         self.dropout = nn.Dropout(p=0.5)
#         self.fc2 = nn.Linear(120, 128)
#         self.batch_norm3 = nn.BatchNorm1d(128)  # Batch normalization layer
#         self.fc3 = nn.Linear(128, 84)
#         self.batch_norm4 = nn.BatchNorm1d(84)  # Batch normalization layer
#         self.fc4 = nn.Linear(84, 10)

#     def forward(self, x):
#         x = self.pool(F.relu(self.batch_norm1(self.conv1(x))))  # Batch␣
↪normalization on the output of the first convolutional layer
#         x = self.pool(F.relu(self.batch_norm2(self.conv2(x))))  # Batch␣
↪normalization on the output of the second convolutional layer
#         x = x.view(-1, 16 * 5 * 5)
#         x = F.relu(self.fc1(x))
#         x = self.dropout(x)
#         x = F.relu(self.batch_norm3(self.fc2(x)))  # Batch normalization on␣
↪the output of the second fully connected layer
#         x = F.relu(self.batch_norm4(self.fc3(x)))  # Batch normalization on␣
↪the output of the third fully connected layer
#         x = self.fc4(x)
#         return x
```

[4]:
```
# # Network for FASHION-MNIST
# class CNN(nn.Module):
#     def __init__(self):
#         super(CNN, self).__init__()
#         self.conv1 = nn.Conv2d(1, 6, 3, stride=1, padding=1)  # Change the␣
↪input channels and kernel size
#         self.batch_norm1 = nn.BatchNorm2d(6)
#         self.pool = nn.MaxPool2d(2, 2)
#         self.conv2 = nn.Conv2d(6, 16, 3, stride=1, padding=1)
#         self.batch_norm2 = nn.BatchNorm2d(16)
#         self.fc1 = nn.Linear(16 * 7 * 7, 120)  # Change the input size to␣
↪match the new feature map size
#         self.dropout = nn.Dropout(p=0.5)
#         self.fc2 = nn.Linear(120, 128)
#         self.batch_norm3 = nn.BatchNorm1d(128)
#         self.fc3 = nn.Linear(128, 84)
#         self.batch_norm4 = nn.BatchNorm1d(84)
#         self.fc4 = nn.Linear(84, 10)

#     def forward(self, x):
#         x = self.pool(F.relu(self.batch_norm1(self.conv1(x))))
#         x = self.pool(F.relu(self.batch_norm2(self.conv2(x))))
#         x = x.view(-1, 16 * 7 * 7)  # Change the shape of the feature maps
#         x = F.relu(self.fc1(x))
#         x = self.dropout(x)
```

```
#         x = F.relu(self.batch_norm3(self.fc2(x)))
#         x = F.relu(self.batch_norm4(self.fc3(x)))
#         x = self.fc4(x)
#         return x
```

[5]:
```
# Network for KMNIST
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, stride=1, padding=1)
        self.batch_norm1 = nn.BatchNorm2d(6)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 3, stride=1, padding=1)
        self.batch_norm2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16 * 7 * 7, 120)
        self.dropout = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(120, 128)
        self.batch_norm3 = nn.BatchNorm1d(128)
        self.fc3 = nn.Linear(128, 84)
        self.batch_norm4 = nn.BatchNorm1d(84)
        self.fc4 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.batch_norm1(self.conv1(x))))
        x = self.pool(F.relu(self.batch_norm2(self.conv2(x))))
        x = x.view(-1, 16 * 7 * 7)  # Change the shape of the feature maps
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.batch_norm3(self.fc2(x)))
        x = F.relu(self.batch_norm4(self.fc3(x)))
        x = self.fc4(x)
        return x
```

[6]:
```
# Define the model
model = CNN()
criterion = nn.CrossEntropyLoss()
optimiser = torch.optim.SGD(model.parameters(), lr=0.0066)

# Define lists to store the training loss and accuracy
train_losses = []
mean_accuracies = []
```

[7]:
```
# Train the CNN
for epoch in range(50):  # loop over the dataset multiple times

    running_loss = 0.0
    running_accuracy = 0.0
```

```python
    num_batches = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimiser.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimiser.step()

        # Calculate the accuracy for this mini-batch
        accuracy = (outputs.argmax(dim=1) == labels).float().mean().item()

        # Adds this to the running accuracy
        running_accuracy += accuracy
        num_batches += 1

        # print statistics
        running_loss += loss.item()
        if i % 500 == 499:  # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            train_losses.append(running_loss / 2000)
            print("Accuracy: " + str(accuracy))
            running_loss = 0.0

    # Calculate the mean accuracy for the epoch
    mean_accuracy = running_accuracy / num_batches
    mean_accuracies.append(mean_accuracy)
```

```
[1,   500] loss: 0.453
Accuracy: 0.125
[1,  1000] loss: 0.320
Accuracy: 0.625
[1,  1500] loss: 0.252
Accuracy: 0.75
[1,  2000] loss: 0.223
Accuracy: 0.875
[1,  2500] loss: 0.208
Accuracy: 0.75
[1,  3000] loss: 0.193
Accuracy: 0.875
[1,  3500] loss: 0.175
Accuracy: 0.875
```

```
[49,  4000] loss: 0.019
Accuracy: 0.875
[49,  4500] loss: 0.017
Accuracy: 0.875
[49,  5000] loss: 0.015
Accuracy: 1.0
[49,  5500] loss: 0.015
Accuracy: 1.0
[49,  6000] loss: 0.019
Accuracy: 1.0
[50,   500] loss: 0.017
Accuracy: 1.0
[50,  1000] loss: 0.016
Accuracy: 0.875
[50,  1500] loss: 0.016
Accuracy: 1.0
[50,  2000] loss: 0.015
Accuracy: 0.75
[50,  2500] loss: 0.014
Accuracy: 0.875
[50,  3000] loss: 0.019
Accuracy: 1.0
[50,  3500] loss: 0.016
Accuracy: 1.0
[50,  4000] loss: 0.014
Accuracy: 1.0
[50,  4500] loss: 0.016
Accuracy: 1.0
[50,  5000] loss: 0.015
Accuracy: 1.0
[50,  5500] loss: 0.015
Accuracy: 1.0
[50,  6000] loss: 0.022
Accuracy: 1.0
```

[8]:

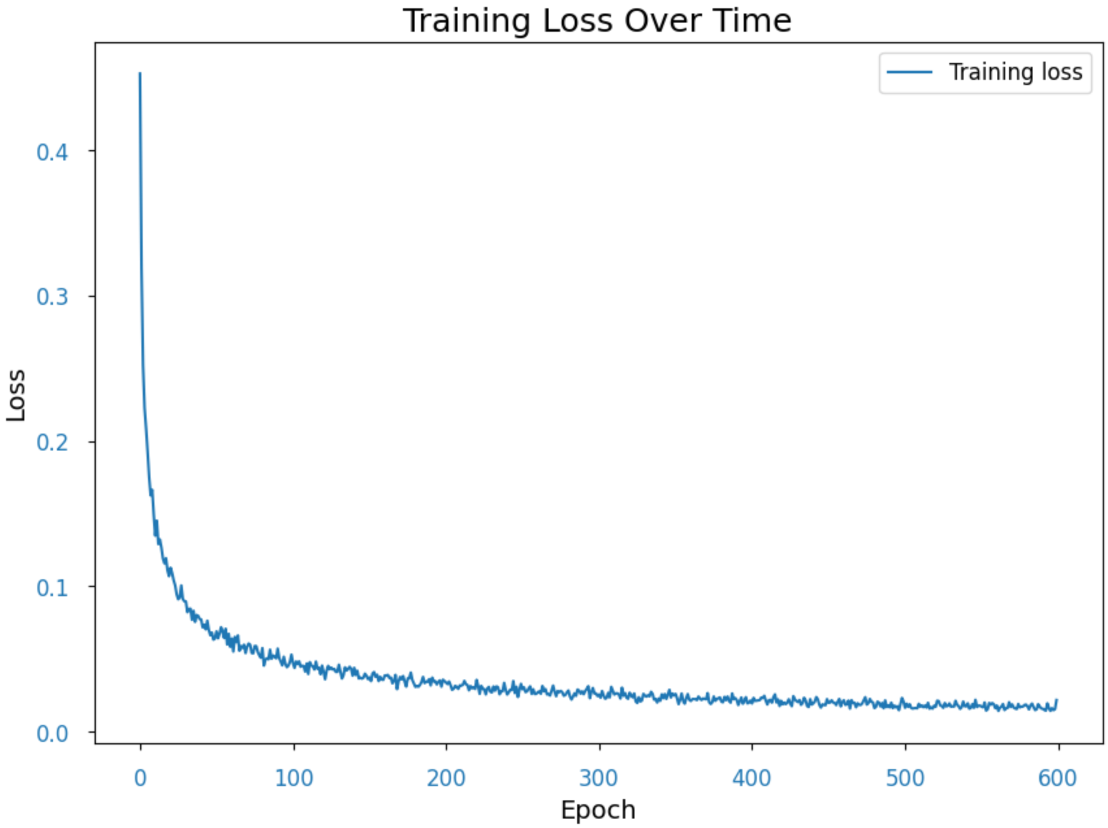
```python
# Evaluate the CNN on the test set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the test images: %d %%' % (
```

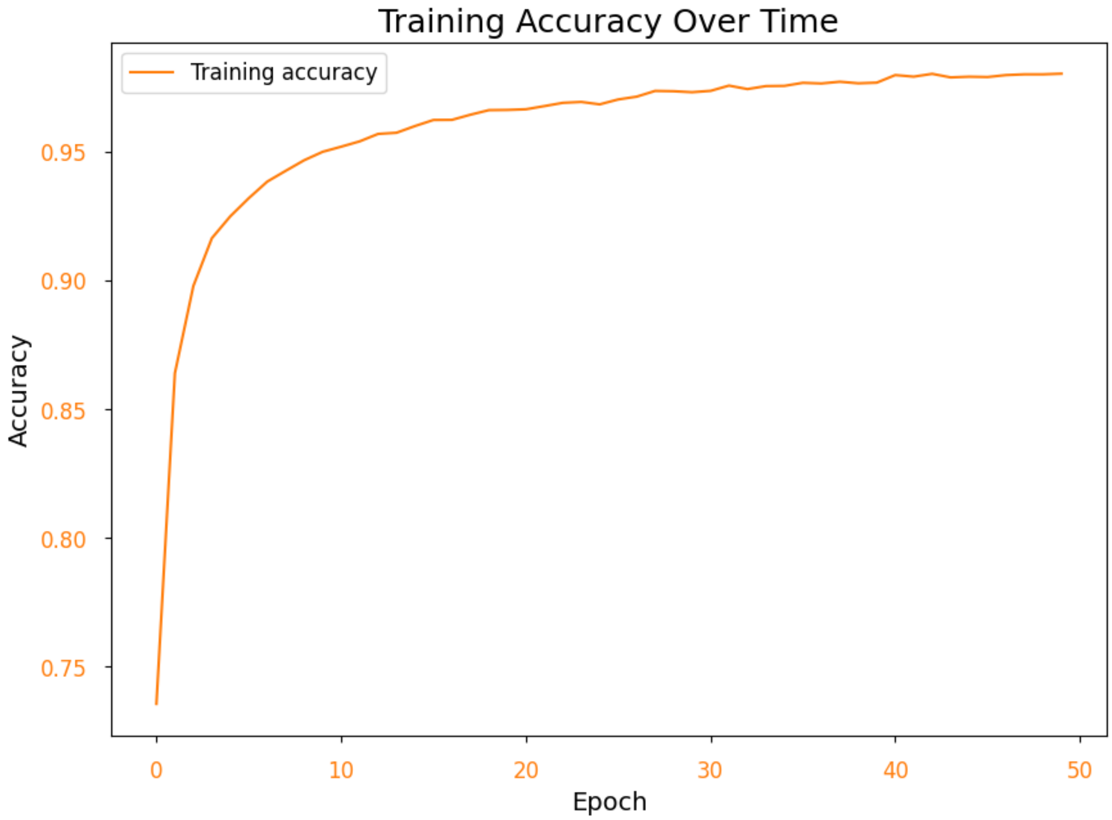
```
        100 * correct / total))
```

```
Accuracy of the network on the test images: 96 %
```

[9]:
```
# Set the figure size
plt.figure(figsize=(10, 7))

# Set the colors for the lines and markers
train_color = 'tab:blue'
test_color = 'tab:orange'
lr_color = 'tab:green'

# Plot the training loss
plt.plot(train_losses, color=train_color, label='Training loss')

# Add a title and axis labels
plt.title('Training Loss Over Time', fontsize=18)
plt.xlabel('Epoch', fontsize=14)
plt.ylabel('Loss', fontsize=14)

# Set the font size and style of the tick labels
plt.tick_params(axis='both', which='major', labelsize=12,␣
 ↪labelcolor=train_color, pad=10)

# Add a legend
plt.legend(loc='upper right', fontsize=12)

# Show the plot
plt.show()

# Set the figure size
plt.figure(figsize=(10, 7))

# Plot the training accuracy
plt.plot(mean_accuracies, color=test_color, label='Training accuracy')

# Add a title and axis labels
plt.title('Training Accuracy Over Time', fontsize=18)
plt.xlabel('Epoch', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)

# Set the font size and style of the tick labels
plt.tick_params(axis='both', which='major', labelsize=12,␣
 ↪labelcolor=test_color, pad=10)

# Add a legend
plt.legend(loc='upper left', fontsize=12)
```

```
# Show the plot
plt.show()
```




Training Accuracy Over Time
Training accuracy
Accuracy
0.95
0.90
0.85
0.80
0.75
0
10
20
30
40
50
Epoch