Question(1)

Q(a)

Download and load dataset.

Normalized images data into [0,1] and split **10000** of 60000 of the whole train datasets as validation dataset.

The final size of train and validation is **50000, 10000.**

Since kmnist dataset size is **28 * 28** and only 1 channel(greyscale), the input shape is **28*28*1**.

```
from extra_keras_datasets import kmnist
# load training data, labels; and testing data and their true labels
(train_images, train_labels), (test_images, test_labels) = kmnist.load_data(type='kmnist')
# input image dimensions
img_x, img_y = 28, 28

# reshape the data into a 4D tensor - (sample_number, x_img_size, y_img_size, num_channels)
train_images = train_images.reshape(train_images.shape[0], img_x, img_y, 1)
test_images  = test_images.reshape(test_images.shape[0], img_x, img_y, 1)
input_shape = (img_x, img_y, 1)

# normalize input between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# split 10000 from training data as validation data
validation_size = 10000
validation_images = train_images[:validation_size]
validation_labels = train_labels[:validation_size]
train_images = train_images[validation_size:]
train_labels = train_labels[validation_size:]
```

Q(b)

Build an ANN with input layer and 2 hidden layers.

Flatten input image in input layer.

Add two hidden layers with **256** and **128** neurons, each hidden layer utilizes **relu** as activation function, and each layer add **BatchNormalization** and **Dropout** to prevent overfit.

```
model = tf.keras.Sequential()

# input layer
model = Sequential()
model.add(Flatten(input_shape=input_shape))  # Flatten the input images

# hidden layers 1
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())  # Batch normalization
model.add(Dropout(0.5))          # Dropout layer

# hidden layers 2
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())  # Batch normalization
model.add(Dropout(0.5))          # Dropout layer
```

Q(c)

Add output layer with **10** neurons with **softmax** as activation function.

```
# output layer
model.add(Dense(10, activation='softmax'))
```

Each neuron represents one class of classifier.

The **softmax** activation function is ideal for multi-class classification problems, since the sum of output after transformed by softmax is 1, and each output is between [0,1], it can represent the probability of predicting each class.

Q(d)

```python
# compile and train the model with Adam optimizer

model_adam = build_model()
model_adam.compile(optimizer=Adam(),
                   loss=SparseCategoricalCrossentropy(),
                   metrics=['accuracy'])

history_adam = model_adam.fit(train_images, train_labels,
                              epochs=20,
                              batch_size=128,
                              validation_data=(validation_images, validation_labels),
                              verbose=1)

# compile and train the model with RMSprop optimizer
model_rmsprop = build_model()
model_rmsprop.compile(optimizer=RMSprop(),
                      loss=SparseCategoricalCrossentropy(),
                      metrics=['accuracy'])

history_rmsprop = model_rmsprop.fit(train_images, train_labels,
                                    epochs=20,
                                    batch_size=128,
                                    validation_data=(validation_images, validation_labels),
                                    verbose=1)
# evaluate both models
test_loss_adam, test_accuracy_adam = model_adam.evaluate(test_images, test_labels)
test_loss_rmsprop, test_accuracy_rmsprop = model_rmsprop.evaluate(test_images, test_labels)

print(f'Adam Optimizer - Test accuracy: {test_accuracy_adam:.4f}')
print(f'RMSprop Optimizer - Test accuracy: {test_accuracy_rmsprop:.4f}')
```

Utilize **SparseCategoricalCrossentropy** as loss function. Because the labels are integers between [0,9], not one-hot format.

As for optimizer, used **Adam** and **RMSprop**.

Adam combines the benefits of **AdaGrad** and **RMSprop** by using running averages of both the gradients (first moment) and the squared gradients (second moment). This results in individual adaptive learning rates for different parameters.

RMSprop maintains a moving average of the squared gradients to adjust the learning rate for each parameter.

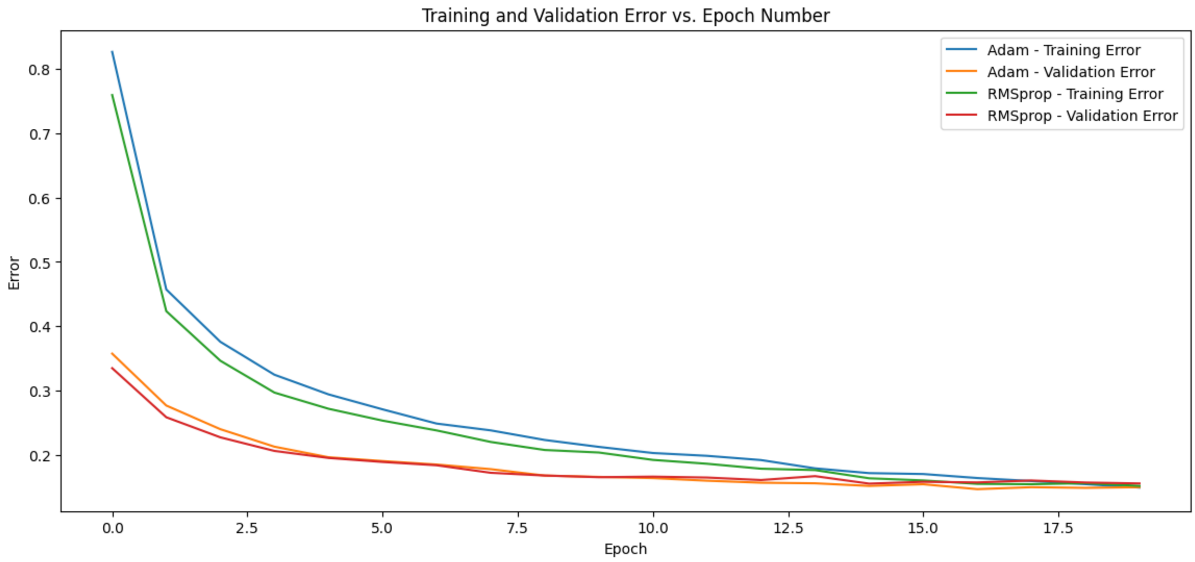
From the result on test dataset, we can see that **RMSprop** has outperformed **Adam** slightly, achieving a test accuracy of 89.40% compared to Adam’s 88.72%. However, the differences are not substantial, indicating both optimizers are effective for this problem.

Q(e)

Code for plotting.

```
# Plotting the performance
plt.figure(figsize=(14, 6))

# Training and validation error for Adam optimizer
plt.plot(history_adam.history['loss'], label='Adam - Training Error')
plt.plot(history_adam.history['val_loss'], label='Adam - Validation Error')

# Training and validation error for RMSprop optimizer
plt.plot(history_rmsprop.history['loss'], label='RMSprop - Training Error')
plt.plot(history_rmsprop.history['val_loss'], label='RMSprop - Validation Error')

plt.title('Training and Validation Error vs. Epoch Number')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.legend()
plt.show()
```

Result:



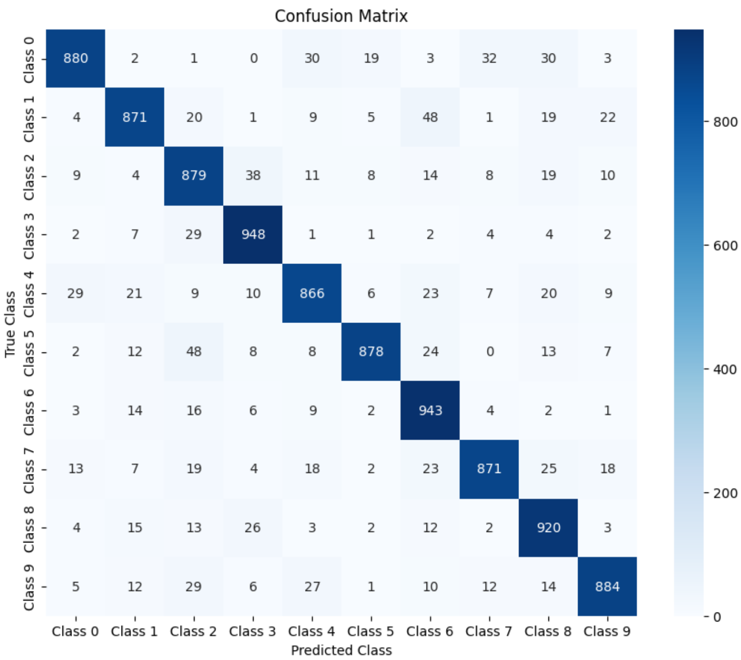
From plot we can see that losses of both validation and training are decreasing with number of epochs, which means no overfitting.

Q(f)

Code snippets:

```python
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# Predict the classes
predictions = model_rmsprop.predict(test_images)
predicted_classes = np.argmax(predictions, axis=1)

# Classification report
class_report = classification_report(test_labels, predicted_classes, target_names=[f'Class {i}' for i in range(10)])
print(class_report)

# Confusion matrix
conf_matrix = confusion_matrix(test_labels, predicted_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=[f'Class {i}' for i in range(10)], yticklabels=[f'Class {i}' for i in range(10)])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Class')
plt.ylabel('True Class')
plt.show()

# Calculate accuracy for each class
class_accuracies = {}
for i in range(10):
    true_positives = conf_matrix[i, i]
    total_samples = np.sum(conf_matrix[i, :])
    class_accuracies[f'Class {i}'] = true_positives / total_samples

print("Class-wise Accuracies:")
for class_name, accuracy in class_accuracies.items():
    print(f'{class_name}: {accuracy:.2f}')
```

Result:




```
Class-wise Accuracies:
Class 0: 0.88
Class 1: 0.87
Class 2: 0.88
Class 3: 0.95
Class 4: 0.87
Class 5: 0.88
Class 6: 0.94
Class 7: 0.87
Class 8: 0.92
Class 9: 0.88
```

We can see that model's prediction on all classes are above 87% which means the model is well generalization.

Classes 3 ,6 and 8 have particularly high accuracy (95% ,94% and 92% respectively), indicating the model performs very well on these three classes.

Class 1 and Class 9 have a moderate number of misclassifications with other classes comparing to other classes.

Question(2)

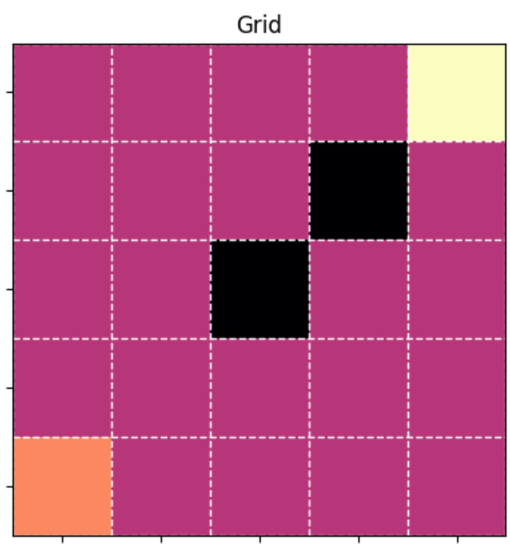
Q(A)

Initialization of environment

```
class Gridworld:
    def __init__(self, size=5):
        self.size = size
        self.start_state = (0, 0)
        self.goal_state = (4, 4)
        self.obstacles = [(2, 2), (3, 3)]
        self.actions = {
            'U': np.array([0, 1]),
            'D': np.array([0, -1]),
            'L': np.array([-1, 0]),
            'R': np.array([1, 0])
        }
        self.reset()

    def reset(self):
        self.agent_position = self.start_state
        return self.agent_position

    def step(self, action):
        action = self.actions[action]
        next_state = tuple(np.array(self.agent_position) + action)
        x, y = next_state

        if x < 0 or x >= self.size or y < 0 or y >= self.size or next_state in self.obstacles:
            return self.agent_position, 0, False
        elif next_state == self.goal_state:
            self.agent_position = next_state
            return self.agent_position, 1, True
        else:
            self.agent_position = next_state
            return self.agent_position, -0.1, False
```

Class **GridWorld** initialize with grid size(default=5),start state(0,0),goal state(4,4) and obstacles(2,2),(3,3). And actions U,D,L,R represent move of agent.

Step function calculate the next state, reward/penalty and whether agent finish episode.



Left is the visualization of Grid.

Q(B)

1.The size of Q-table should be (number of state) * (number of actions), in this case, It's 5*5*4.All values are initialized as 0.

```
self.q_table = np.zeros((env.size, env.size, len(self.actions)))
```

2.a Exploration strategy

```
# epsilon-greedy policy
def choose_action(self, state):
    if np.random.uniform(0, 1) < self.epsilon:
        return np.random.choice(self.actions)  # Exploration
    else:
        state_action = self.q_table[state[0], state[1], :]
        action_index = np.argmax(state_action)
        return self.actions[action_index]  # Exploitation
```

2.b Learning rate and discount factor

```
class QLearningAgent:
    def __init__(self, env, alpha=0.1, gamma=0.95, epsilon=0.05, episodes=6000, info_frequency=100, log=True):
        self.env = env
        self.alpha = alpha  # learning rate
        self.gamma = gamma  # discount factor
        self.epsilon = epsilon  # epsilon-greedy exploration
        self.episodes = episodes
```

2.c Update rule for Q values

```
# update rule
def update_q_table(self, state, action, reward, next_state):
    action_index = self.actions.index(action)
    next_state_value = np.max(self.q_table[next_state[0], next_state[1], :])
    td_target = reward + self.gamma * next_state_value
    td_error = td_target - self.q_table[state[0], state[1], action_index]
    self.q_table[state[0], state[1], action_index] += self.alpha * td_error
```

To be more specific:

New_Q = Old_Q + alpha [reward + gamma(max(value of next state)) – Old_Q]

3.Code for training:

```python
def learn(self):
    for episode in range(self.episodes):
        state = self.env.reset()  # Initialize state
        done = False
        total_reward = 0
        steps = 0
        while not done:  # Loop until episode ends
            action = self.choose_action(state)  # Choose action based on epsilon-greedy policy
            next_state, reward, done = self.env.step(action)  # Take action and observe reward
            self.update_q_table(state, action, reward, next_state)
            state = next_state
            total_reward += reward
            steps += 1
        self.cumulative_rewards.append(total_reward)
        if self.log:
          print(f'Episode {episode + 1}/{self.episodes}, Total steps: {steps}, Total reward: {np.round(total_reward, 2)}')
```

Training log:

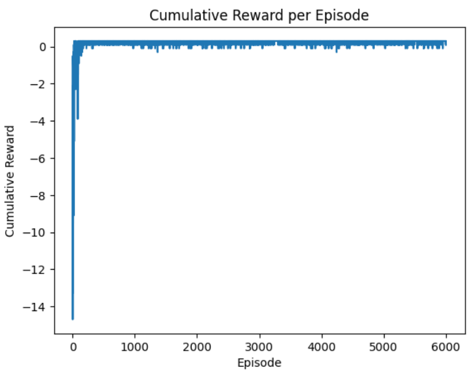
```
Episode 5959/6000, Total steps: 8, Total reward: 0.3
Episode 5960/6000, Total steps: 8, Total reward: 0.3
Episode 5961/6000, Total steps: 9, Total reward: 0.3
Episode 5962/6000, Total steps: 8, Total reward: 0.3
Episode 5963/6000, Total steps: 8, Total reward: 0.3
Episode 5964/6000, Total steps: 8, Total reward: 0.3
Episode 5965/6000, Total steps: 8, Total reward: 0.3
Episode 5966/6000, Total steps: 8, Total reward: 0.3
Episode 5967/6000, Total steps: 8, Total reward: 0.3
```

Both episode and total steps are recorded.

Code for plot:

```python
def plot_rewards(self):
    plt.plot(range(self.episodes), self.cumulative_rewards)
    plt.xlabel('Episode')
    plt.ylabel('Cumulative Reward')
    plt.title('Cumulative Reward per Episode')
    plt.show()
```

Plot of cumulative reward:



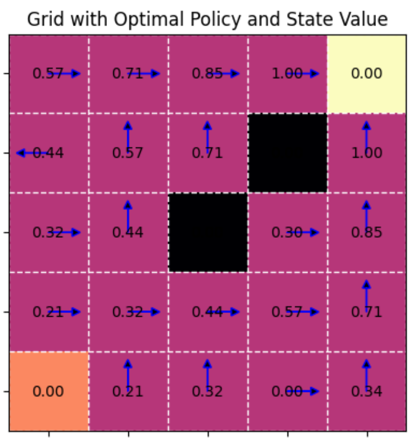
Q(C)

Code for experiment and plot:

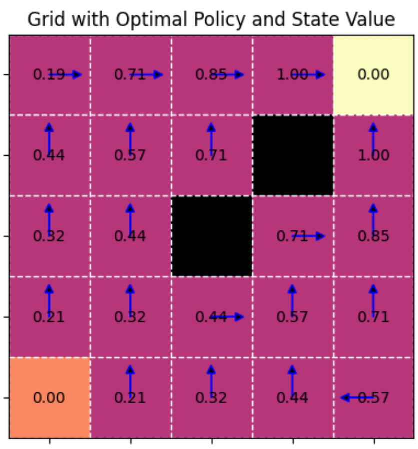
```python
def get_optimal_policy(self):
    for i in range(self.env.size):
        for j in range(self.env.size):
            if (i, j) in self.env.obstacles or (i, j) == self.env.goal_state:
                continue
            optimal_action_index = np.argmax(self.q_table[i, j, :])
            self.optimal_policy[i, j] = self.actions[optimal_action_index]

def get_state_value(self):
    for i in range(self.env.size):
        for j in range(self.env.size):
            if (i, j) in self.env.obstacles or (i, j) == self.env.goal_state or (i, j) == self.env.start_state:
                continue
            optimal_action_index = np.argmax(self.q_table[i, j, :])
            self.state_values[i, j] = np.round(self.q_table[i, j, optimal_action_index], 2)

def plot_optimal_policy(self):
    self.env.render(policy=self.optimal_policy)

def plot_optimal_values(self):
    self.env.render(state_values=self.state_values)

def plot_optimal_policy_and_values(self):
    self.env.render(policy=self.optimal_policy, state_values=self.state_values)
```

```python
def train_and_plot(alpha, gamma, epsilon, episodes=6000):
    agent = QLearningAgent(env, alpha=alpha, gamma=gamma, epsilon=epsilon, episodes=episodes, log=False)
    agent.learn()
    print(f"Parameters: alpha={alpha}, gamma={gamma}, epsilon={epsilon}")
    agent.plot_optimal_policy_and_values()

# Define environment
env = Gridworld(size=5)

# Train and plot for different parameter settings
# Baseline
print("Training Agent 1: alpha=0.1, gamma=0.95, epsilon=0.1")
train_and_plot(alpha=0.1, gamma=0.95, epsilon=0.05)

# higher learning rate
print("Training Agent 2: alpha=0.9, gamma=0.95, epsilon=0.1")
train_and_plot(alpha=0.9, gamma=0.95, epsilon=0.05)

# lower discount factors
print("Training Agent 3: alpha=0.1, gamma=0.5, epsilon=0.1")
train_and_plot(alpha=0.1, gamma=0.5, epsilon=0.05)

# higher exploration rate
print("Training Agent 4: alpha=0.1, gamma=0.95, epsilon=0.5")
train_and_plot(alpha=0.1, gamma=0.95, epsilon=0.5)
```

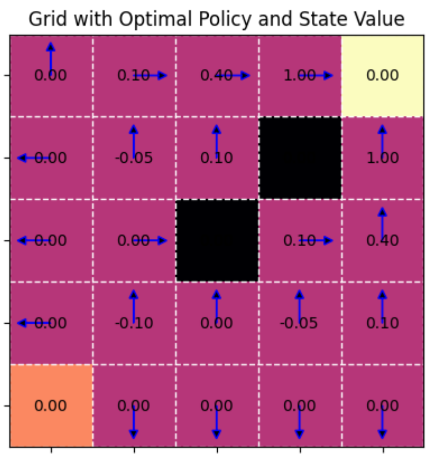
Following are the policies in different parameters:



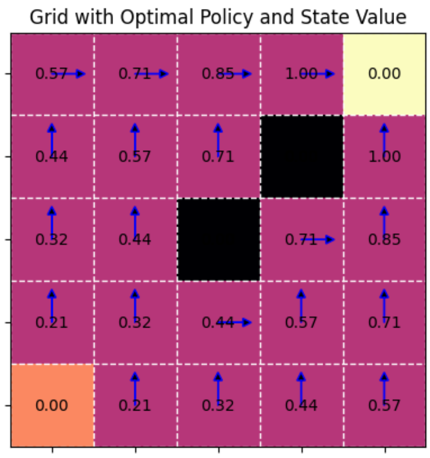
Baseline model



Higher alpha



Lower discount factor



Higher exploration rate

So for different parameters:

1. Higher alpha: The agent learns faster and may exhibit more aggressive policy updates.
2. Lower discount factor: The agent focuses more on immediate rewards.
3. Higher exploration rate: The policy is more varied and can capture a wider range of possible strategies.

To conclude:

Learning Rate (alpha): Determines the size of updates to Q-values; higher α results in faster but potentially less stable learning, while lower α leads to slower but more stable learning.

Discount Factor (gamma): Controls the importance of future rewards; higher γ values make the agent prioritize long-term gains, while lower γ values make it focus more on immediate rewards.

Exploration Rate (epsilon): Balances exploration and exploitation; higher ε values encourage more exploration of the environment, while lower ε values favor exploiting known information.

Q(D)

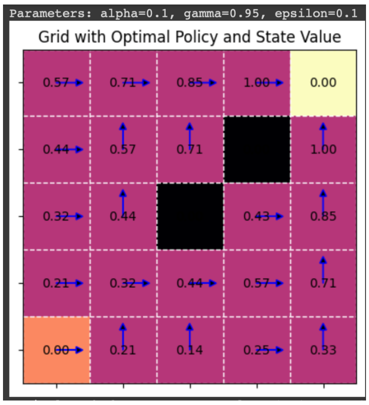
Code:

```python
# Helper function to train an agent and plot the optimal policy with the path from start to goal
def train_and_evaluate(alpha, gamma, epsilon, episodes=6000):
    agent = QLearningAgent(env, alpha=alpha, gamma=gamma, epsilon=epsilon, episodes=episodes, log=False)
    agent.learn()
    print(f"Parameters: alpha={alpha}, gamma={gamma}, epsilon={epsilon}")
    agent.plot_optimal_policy_and_values()
    return agent.optimal_policy

# Define environment
env = Gridworld(size=5)

# Train and evaluate the agent with specific parameters
optimal_policy = train_and_evaluate(alpha=0.1, gamma=0.95, epsilon=0.1)

# Function to trace the optimal path from start to goal state
def trace_optimal_path(policy, start_state, goal_state):
    current_state = start_state
    path = [current_state]
    while current_state != goal_state:
        action = policy[current_state[0], current_state[1]]
        next_state = tuple(np.array(current_state) + env.actions[action])
        path.append(next_state)
        current_state = next_state
    return path

# Trace and print the optimal path
optimal_path = trace_optimal_path(optimal_policy, env.start_state, env.goal_state)
print("Optimal Path from Start to Goal:", optimal_path)
```

```python
# Visualize the optimal path
def plot_optimal_path(optimal_path):
    fig, ax = plt.subplots()
    grid = np.zeros((env.size, env.size))
    for obstacle in env.obstacles:
        grid[obstacle] = -1
    grid[env.goal_state] = 1
    grid[env.start_state] = 0.5
    ax.imshow(grid, cmap='magma', interpolation='none')

    ax.set_xticks(np.arange(env.size + 1) - 0.5, minor=True)
    ax.set_yticks(np.arange(env.size + 1) - 0.5, minor=True)
    ax.grid(which='minor', color='white', linestyle='--', linewidth=1)
    ax.tick_params(which='minor', size=0)
    ax.set_xticklabels([])
    ax.set_yticklabels([])

    for (i, j) in optimal_path:
        ax.text(j, i, 'o', ha='center', va='center', color='blue')

    plt.title("Optimal Path from Start to Goal")
    plt.gca().invert_yaxis()
    plt.show()

# Plot the optimal path
plot_optimal_path(optimal_path)
```

Result:

```
Parameters: alpha=0.1, gamma=0.95, epsilon=0.1
```




```
Optimal Path from Start to Goal: [(0, 0), (1, 0), (1, 1), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (4, 4)]
```