MNIST Fashion Classification

Nizar ZEROUALE

Application 1 :

CODE :

```
import numpy as np
from sklearn.model_selection import KFold
import keras
import np_utils
from keras.optimizers.legacy import SGD
from keras.datasets import fashion_mnist
from keras import layers
from matplotlib import pyplot
import cv2
from keras.src.utils.np_utils import to_categorical

def prepareData(trainX, trainY, testX, testY):

    #TODO - Application 1 - Step 4a - reshape the data to be of size
[samples][width][height][channels]
    trainX = trainX.reshape((trainX.shape[0],28,28,1))
    testX = testX.reshape((testX.shape[0],28,28,1))

    #TODO - Application 1 - Step 4b - normalize the input values
    trainX = trainX / 255
    testX = testX / 255

    #TODO - Application 1 - Step 4c - Transform the classes labels
into a binary matrix
    trainY = to_categorical(trainY, 10)
    testY = to_categorical(testY, 10)

    return trainX, trainY, testX, testY

def defineModel(input_shape, num_classes):

    #TODO - Application 1 - Step 6a - Initialize the sequential
model
    model = keras.models.Sequential()

    #TODO - Application 1 - Step 6b - Create the first hidden layer
as a convolutional layer
    model.add(layers.Conv2D(32, kernel_size=(3, 3),
activation='relu', input_shape=input_shape,
kernel_initializer='he_uniform'))

    #TODO - Application 1 - Step 6c - Define the pooling layer
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    #TODO - Application 1 - Step 6d - Define the flatten layer
    model.add(layers.Flatten())
    #TODO - Application 1 - Step 6e - Define a dense layer of size
16
    model.add(layers.Dense(16, activation='relu',
kernel_initializer='he_uniform'))

    #TODO - Application 1 - Step 6f - Define the output layer
    model.add(layers.Dense(num_classes, activation='softmax'))

    #TODO - Application 1 - Step 6g - Compile the model
    model.compile(loss='categorical_crossentropy',
optimizer=SGD(learning_rate = 0.01, momentum = 0.9),
metrics=['accuracy'])

    return model


def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY):

    #TODO - Application 1 - Step 6 - Call the defineModel function
    input_shape = (28,28,1)
    num_classes = 10
    model = defineModel(input_shape, num_classes)

    #TODO - Application 1 - Step 7 - Train the model
    model.fit(trainX, trainY, validation_data=(testX, testY),
epochs=5, batch_size=32, verbose=2)

    #TODO - Application 1 - Step 8 - Evaluate the model
    #Evaluate the model on the test data
    scores = model.evaluate(testX, testY, verbose=0)

    # Print the classification error rate
    print(f"Classification Error Rate: {(1-scores[1]) * 100:.2f}%")

    return

def main():

    #TODO - Application 1 - Step 2 - Load the Fashion MNIST dataset
in Keras
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()

    #TODO - Application 1 - Step 3 - Print the size of the
train/test dataset
    #print(trainX.shape[0])
    # result was 60000

    #TODO - Application 1 - Step 4 - Call the prepareData method
    trainX, trainY, testX, testY = prepareData(trainX, trainY,
testX, testY)

    #TODO - Application 1 - Step 5 - Define, train and evaluate the
model in the classical way
    defineTrainAndEvaluateClassic(trainX, trainY, testX, testY)


    return
```

OUTPUT :

```
#Locally
Epoch 1/5
1875/1875 - 8s - loss: 0.4450 - accuracy: 0.8394 - val_loss: 0.3511 - val_accuracy: 0.8784 - 8s/epoch
- 4ms/step
.
.
.
Epoch 5/5
1875/1875 - 8s - loss: 0.2239 - accuracy: 0.9176 - val_loss: 0.2827 - val_accuracy: 0.9013 - 8s/epoch
- 4ms/step
Classification Error Rate: 9.87%

#Using Collab :
Epoch 1/5
1875/1875 - 8s - loss: 0.4677 - accuracy: 0.8336 - val_loss: 0.4084 - val_accuracy: 0.8563 - 8s/epoch
- 4ms/step
.
.
.
Epoch 5/5
1875/1875 - 8s - loss: 0.2399 - accuracy: 0.9129 - val_loss: 0.3001 - val_accuracy: 0.8920 - 8s/epoch
- 4ms/step
Classification Error Rate: 10.80%
```

Exercise 1:

We add the displayFirst9ImagesWithCV() function, and call it in the main

CODE:

```python
# Display the first 9 images
def displayFirst9ImagesWithCV(trainX):
    for i in range(9):
        img = trainX[i]
        window_name = 'Image ' + str(i+1)
        cv2.imshow(window_name, img)

        # Wait for a key press to move to the next image
        cv2.waitKey(0)

    # Close all the windows
    cv2.destroyAllWindows()
```

Exercise 2:

No of filters	8	16	32	64	128
Classification Err %	10.91	10.23	10.22	10.58	11.11

As the number of filters increases from 8 to 32, the classification error percentage decreases, suggesting an improvement in system accuracy. This indicates that more filters allow the model to capture a richer set of features from the input images, which can help in better distinguishing between different clothing categories.
Beyond 32 filters, the trend reverses slightly or plateaus. The classification error increases when moving from 32 to 64 filters and then slightly decreases again at 128 filters, but not below the lowest error observed at 32 filters.
The classification error percentage suggests that the model with 32 filters might be more efficient or has reached a more optimal state compared to other configurations.
However, the convergence time increases with the number of filters. This is because more filters mean more parameters to train, leading to increased computational complexity and longer training times.

Exercise 3:

No of neurons	16	64	128	256	512
Classification Err %	10.80	10.13	9.91	9.84	8.93

As the number of neurons increases from 16 to 512, there's a consistent decrease in the classification error percentage, from 10.80% down to 8.93%. It means that adding more neurons to the network improves its ability to learn and model the complexities of the input data.

Exercise 4:

No of epochs	1	2	5	10	20
Classification Err %	12.75	11.80	11.06	10.22	10.04

There's a consistent decrease in the classification error percentage as the number of epochs increases from 1 to 20. This trend highlights that allowing the model more time to learn from the data helps in refining its weights and biases to better distinguish between different clothing categories.
However, increasing the number of epochs directly impacts the convergence time, as the model takes longer to train with more epochs.

Exercise 5:

Learning Rate	0.1	0.01	0.001	0.0001	0.00001
Classification Err %	15.33	12.58	12.31	16.01	29.44

We see that a higher learning rate may lead to faster convergence but at the risk of overshooting the minimum, while too low a learning rate results in very slow convergence. The optimal learning rate range (0.01 to 0.001) likely provides a balance where the model converges to a satisfactory solution efficiently without taking too long or risking significant overshooting.

Exercise 6:

Dropout %	0.1	0.2	0.3	0.4	0.5
Classification Err %	10.88	10.48	10.79	11.65	11.88

The system accuracy is maximally enhanced at a dropout rate of 0.2%, where it helps to mitigate overfitting while allowing the model to maintain sufficient capacity to learn from the training data. Both too low and too high dropout rates negatively impact the model's accuracy, with lower rates risking overfitting and higher rates preventing the model from learning effectively due to excessive information loss.

Exercise 7:

Specify the optimal values for the following parameters (the number of filters in the convolutional layer, the size of the convolutional kernel, the number of neurons in the dense hidden layer, the number of epochs used for training, the learning rate) that maximize the system accuracy.

Based on the experiments and the observations from each parameter's influence on system accuracy and convergence time, the optimal values for maximizing system accuracy can be specified as follows:

● Number of Filters in Convolutional Layer:	32
● Size of Convolutional Kernel:	3x3
● Number of Neurons in the Dense Hidden Layer:	512
● Number of Epochs for Training:	20
● Learning Rate:	0.001

OUTPUT: (Tested for these parameters)

```
Epoch 1/20
1875/1875 - 25s - loss: 0.5503 - accuracy: 0.8032 - val_loss: 0.4295 - val_accuracy: 0.8478 -
25s/epoch - 13ms/step
.
.
.
Epoch 20/20
1875/1875 - 29s - loss: 0.1827 - accuracy: 0.9333 - val_loss: 0.2447 - val_accuracy: 0.9111 -
29s/epoch - 16ms/step
Classification Error Rate: 8.89%
```

Exercise 8:

Using the pre-trained model (saved above) , write a Python script able to make an automatic prediction regarding the category of the image presented in Fig.1. For this example, we expect class “2” that corresponds to a “Pullover”.

CODE :

```
import numpy as np
from keras.preprocessing.image import load_img, img_to_array
from keras.models import load_model

# Function to load and prepare the image in the right format
def load_image(filename):
    # Load the image
    img = load_img(filename, color_mode='grayscale',
target_size=(28, 28))
    # Convert the image to array
    img = img_to_array(img)
    # Reshape into a single sample with 1 channel
    img = img.reshape(1, 28, 28, 1)
    # Prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    return img

# Function to load the model and make a prediction
def predict_image_class(model, image):
    # Predict the class
    result = model.predict(image)
    # Get the index of the highest probability
    prediction = np.argmax(result, axis=1)
    return prediction[0]

# Load the image
filename = 'sample_image.png'
img = load_image(filename)

# Load the model
model = load_model('Fashion_MNIST_model.h5')
```

```
# Make a prediction
prediction = predict_image_class(model, img)

# Map the prediction to the actual class names (looked for in the
net)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
predicted_class = class_names[prediction]

print(f'Predicted class: {predicted_class}')
```

OUTPUT :

```
python3 Prediction_test.py
1/1 [==============================] - 0s 57ms/step
Predicted class: Pullover
```

Application 2 :

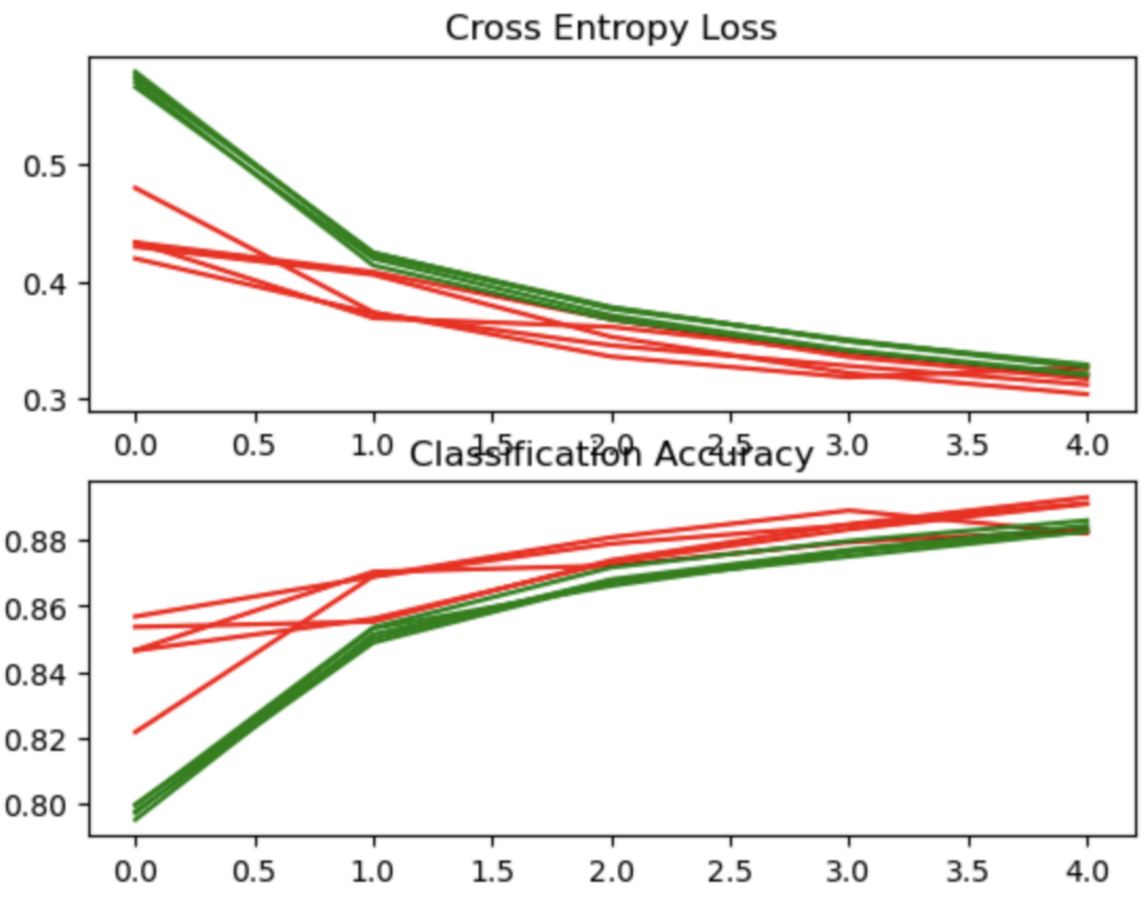
We define the functions `summarizeLearningCurvesPerformances()` and `defineTrainAndEvaluateKFolds(),` that we will call in the `main():`

CODE :

```
def summarizeLearningCurvesPerformances(histories, accuracyScores):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='green',
label='train')
        pyplot.plot(histories[i].history['val_loss'], color='red',
label='test')

        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='green',
label='train')
        pyplot.plot(histories[i].history['val_accuracy'],
color='red', label='test')

        #print accuracy for each split
        print("Accuracy for set {} = {}".format(i,
accuracyScores[i]))
    pyplot.show()
    print('Accuracy: mean = {:.3f} std = {:.3f}, n =
{}'.format(np.mean(accuracyScores) * 100, np.std(accuracyScores) *
100, len(accuracyScores)))


def defineTrainAndEvaluateKFolds(trainX, trainY, testX, testY):
    k_folds = 5
    accuracyScores = []
    histories = []

    #Application 2 - Step 2 - Prepare the cross validation datasets
    kfold = KFold(k_folds, shuffle=True, random_state=1)
    for train_idx, val_idx in kfold.split(trainX):

        #TODO - Application 2 - Step 3 - Select data for train and
validation
        trainX_fold, trainY_fold = trainX[train_idx],
trainY[train_idx]
        valX_fold, valY_fold = trainX[val_idx], trainY[val_idx]

        #TODO - Application 2 - Step 4 - Build the model - Call the
defineModel function
        num_classes = 10
        model = defineModel((28, 28, 1), num_classes)

        #TODO - Application 2 - Step 5 - Fit the model
        history = model.fit(trainX_fold, trainY_fold, epochs=5,
batch_size=32, validation_data=(valX_fold, valY_fold), verbose=2)

        #TODO - Application 2 - Step 6 - Save the training related
information in the histories list
        histories.append(history)

        #TODO - Application 2 - Step 7 - Evaluate the model on the
test dataset
        _, accuracy = model.evaluate(testX, testY, verbose=0)

        #TODO - Application 2 - Step 8 - Save the accuracy in the
accuracyScores list
        accuracyScores.append(accuracy)

    return histories, accuracyScores


def main():

    #TODO - Application 1 - Step 2 - Load the Fashion MNIST dataset
in Keras
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()

    # Display the first 9 images
    #displayFirst9ImagesWithCV(trainX)

    #TODO - Application 1 - Step 3 - Print the size of the
train/test dataset
    #print(trainX.shape[0])

    #TODO - Application 1 - Step 4 - Call the prepareData method
    trainX, trainY, testX, testY = prepareData(trainX, trainY,
testX, testY)

    #TODO - Application 1 - Step 5 - Define, train and evaluate the
model in the classical way
    #defineTrainAndEvaluateClassic(trainX, trainY, testX, testY)
    #TODO - Application 2 - Step 1 - Define, train and evaluate the
model using K-Folds strategy
    histories, accuracyScores = defineTrainAndEvaluateKFolds(trainX,
trainY, testX, testY)
    #TODO - Application 2 - Step9 - System performance presentation
    summarizeLearningCurvesPerformances(histories, accuracyScores)

    return
```

OUTPUT :

```
Epoch 1/5
1500/1500 - 11s - loss: 0.5762 - accuracy: 0.7972 - val_loss: 0.4333 - val_accuracy: 0.8466 -
11s/epoch - 7ms/step
…
Epoch 5/5
1500/1500 - 11s - loss: 0.3259 - accuracy: 0.8837 - val_loss: 0.3200 - val_accuracy: 0.8830 -
11s/epoch - 7ms/step
Epoch 1/5
1500/1500 - 11s - loss: 0.5793 - accuracy: 0.7952 - val_loss: 0.4798 - val_accuracy: 0.8217 -
11s/epoch - 7ms/step
…
Epoch 5/5
1500/1500 - 10s - loss: 0.3286 - accuracy: 0.8826 - val_loss: 0.3258 - val_accuracy: 0.8821 -
10s/epoch - 7ms/step
Epoch 1/5
1500/1500 - 11s - loss: 0.5663 - accuracy: 0.7998 - val_loss: 0.4331 - val_accuracy: 0.8537 -
11s/epoch - 7ms/step
…
Epoch 5/5
1500/1500 - 10s - loss: 0.3207 - accuracy: 0.8845 - val_loss: 0.3163 - val_accuracy: 0.8930 -
10s/epoch - 7ms/step
Epoch 1/5
1500/1500 - 10s - loss: 0.5705 - accuracy: 0.7995 - val_loss: 0.4296 - val_accuracy: 0.8465 -
10s/epoch - 7ms/step
…
Epoch 5/5
1500/1500 - 11s - loss: 0.3194 - accuracy: 0.8860 - val_loss: 0.3033 - val_accuracy: 0.8910 -
11s/epoch - 7ms/step
Epoch 1/5
1500/1500 - 11s - loss: 0.5741 - accuracy: 0.7977 - val_loss: 0.4194 - val_accuracy: 0.8568 -
11s/epoch - 8ms/step
…
Epoch 5/5
1500/1500 - 11s - loss: 0.3265 - accuracy: 0.8826 - val_loss: 0.3114 - val_accuracy: 0.8913 -
11s/epoch - 8ms/step

Accuracy for set 0 = 0.8788999915122986
Accuracy for set 1 = 0.8738999962806702
Accuracy for set 2 = 0.882099986076355
Accuracy for set 3 = 0.8863999843597412
Accuracy for set 4 = 0.8806999921798706
```



Accuracy: mean = 88.040 std = 0.409, n = 5

Exercise 9:

Adding padding to the convolutional operation can often result in better model performance, as more of the input image of feature maps are given an opportunity to participate or contribute to the output. By default, the convolutional operation uses 'valid' padding, which means that convolutions are only applied where possible. This can be changed to padding='same' so that zero values are added around the input such that the output has the same size as the input. For Application 1, how is the system accuracy influenced by the padding operation?

Adding padding to the convolutional operation, specifically using padding='same', allows the convolution to process the edge pixels of the input image.
To do so, a change in the model definition must be done

CODE :

```
def defineModel(input_shape, num_classes):

    #TODO - Application 1 - Step 6a - Initialize the sequential
model
    model = keras.models.Sequential()
    #TODO - Application 1 - Step 6b - Create the first hidden layer
as a convolutional layer
    model.add(layers.Conv2D(32, kernel_size=(3, 3),
activation='relu', input_shape=input_shape,
kernel_initializer='he_uniform', padding='same'))

    #TODO - Application 1 - Step 6c - Define the pooling layer
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    # Add Dropout
    model.add(layers.Dropout(0.2))

    #TODO - Application 1 - Step 6d - Define the flatten layer
    model.add(layers.Flatten())

    #TODO - Application 1 - Step 6e - Define a dense layer of size
16
    model.add(layers.Dense(512, activation='relu',
kernel_initializer='he_uniform'))

    #TODO - Application 1 - Step 6f - Define the output layer
    model.add(layers.Dense(num_classes, activation='softmax'))

    #TODO - Application 1 - Step 6g - Compile the model
    model.compile(loss='categorical_crossentropy',
optimizer=SGD(learning_rate = 0.001, momentum = 0.9),
metrics=['accuracy'])

    #Save the model
    #model.save('Fashion_MNIST_model.h5')

    return model
```

OUTPUT :

```
Epoch 1/20
1875/1875 - 13s - loss: 0.5406 - accuracy: 0.8087 - val_loss: 0.4193 - val_accuracy: 0.8523 -
13s/epoch - 7ms/step
.
.
.
Epoch 20/20
1875/1875 - 14s - loss: 0.1808 - accuracy: 0.9355 - val_loss: 0.2431 - val_accuracy: 0.9135 -
14s/epoch - 7ms/step
Classification Error Rate: 8.65%
```

We can observe that there is a slight improvement in accuracy, as we obtained a Classification Error Rate of 8.89% earlier, with a 'valid' padding, and the convergence time is significantly lower.
Without padding, the convolution operation focuses more on the center of the image than on the edges, potentially leading to boundary artifacts where edge information is underutilized. By applying padding='same', the network treats edge pixels equally, leading to a more balanced and effective learning process. This can contribute to both improved accuracy and

faster convergence since the model does not need to compensate for the underrepresented edge information.
The combination of comprehensive feature extraction, efficient gradient propagation, and balanced learning across the entire image likely contributes to optimized learning dynamics. This results in a model that not only performs slightly better in terms of accuracy but also learns more efficiently, leading to faster convergence.

Exercise 10:

An increase in the number of filters used in the convolutional layer can often improve performance, as it can provide more opportunity for extracting simple features from the input images. This is especially relevant when very small filters are used, such as 3×3 pixels. By applying the padding operation (padding='same') within the convolutional process, increase the number of filters (in the convolutional layer) from 32 to 64. For Application 1, how is the system accuracy influenced by this parameter?

CODE :
```
def defineModel(input_shape, num_classes):

    #TODO - Application 1 - Step 6a - Initialize the sequential
model
    model = keras.models.Sequential()

    #TODO - Application 1 - Step 6b - Create the first hidden layer
as a convolutional layer
    model.add(layers.Conv2D(64, kernel_size=(3, 3),
activation='relu', input_shape=input_shape,
kernel_initializer='he_uniform', padding='same'))

    #TODO - Application 1 - Step 6c - Define the pooling layer
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    # Add Dropout
    model.add(layers.Dropout(0.2))

    #TODO - Application 1 - Step 6d - Define the flatten layer
    model.add(layers.Flatten())

    #TODO - Application 1 - Step 6e - Define a dense layer of size
16
    model.add(layers.Dense(512, activation='relu',
kernel_initializer='he_uniform'))

    #TODO - Application 1 - Step 6f - Define the output layer
    model.add(layers.Dense(num_classes, activation='softmax'))

    #TODO - Application 1 - Step 6g - Compile the model
    model.compile(loss='categorical_crossentropy',
optimizer=SGD(learning_rate = 0.001, momentum = 0.9),
metrics=['accuracy'])
```

```
    #Save the model
    #model.save('Fashion_MNIST_model.h5')

    return model
```

OUTPUT :

```
Epoch 1/20
1875/1875 - 25s - loss: 0.5240 - accuracy: 0.8131 - val_loss: 0.4410 - val_accuracy: 0.8367 -
.
.
.
Epoch 20/20
1875/1875 - 29s - loss: 0.1623 - accuracy: 0.9417 - val_loss: 0.2429 - val_accuracy: 0.9132 -
29s/epoch - 16ms/step
Classification Error Rate: 8.68%
```

Conclusion :

In this lab, we delved into the realm of deep learning through the lens of clothing classification, employing Convolutional Neural Networks (CNNs) and advanced training methodologies such as k-fold cross-validation. The lab was structured around two primary applications, each focusing on leveraging CNNs to classify items within the Fashion-MNIST dataset.

The initial phase of the lab, Application 1, highlighted the fundamental role of CNN architecture in determining model accuracy. By varying the number of filters, kernel sizes, and neurons in dense layers, we gained insights into how these parameters impact the model's ability to learn from the data. The experiments demonstrated that models with optimized configurations could achieve lower classification error rates, showcasing the delicate balance between model complexity and overfitting.

Furthermore, the introduction of 'same' padding as opposed to 'valid' padding underscored the importance of input feature preservation for enhancing model accuracy. This adjustment not only improved classification performance but also significantly accelerated convergence, illustrating the efficiency gains achievable through thoughtful architectural choices.

Application 2 introduced the concept of k-fold cross-validation, a robust training strategy designed to enhance model generalizability. By dividing the training dataset into several folds and systematically rotating the validation set, we observed improvements in model robustness and gained a more reliable estimate of its performance on unseen data. This methodological shift underscored the importance of validation strategies in achieving models that generalize well across diverse datasets.

In conclusion, this lab provided a comprehensive exploration of CNNs in the context of clothing classification, from the basics of model architecture to advanced training and validation techniques. The exercises not only highlighted the critical factors that influence model performance but also offered practical insights into optimizing deep learning models for image classification tasks.