Assignment 1 - Building a Vision Model with Keras

In this assignment, you will build a simple vision model using Keras. The goal is to classify images from the Fashion MNIST dataset, which contains images of clothing items.

You will:

1. Load and inspect the Fashion MNIST dataset.
2. Run a simple baseline model to establish a performance benchmark.
3. Build and evaluate a simple CNN model, choosing appropriate loss and metrics.
4. Design and run controlled experiments on one hyperparameter (e.g., number of filters, kernel size, etc.) and one regularization technique (e.g., dropout, L2 regularization).
5. Analyze the results and visualize the model's performance.

1. Loading and Inspecting the Dataset

Fashion MNIST is a dataset of grayscale images of clothing items, with 10 classes. Each image is 28x28 pixels, like the MNIST dataset of handwritten digits. Keras provides a convenient way to load this dataset.

In this section, you should:

- ☑ Inspect the shapes of the training and test sets to confirm their size and structure.
- ☑ Convert the labels to one-hot encoded format if necessary. (There is a utility function in Keras for this.)
- ☑ Visualize a few images from the dataset to understand what the data looks like.

```
import numpy as np
import tensorflow as tf
print("Num GPUs Available:", len(tf.config.list_physical_devices('GPU')))
from keras.models import Sequential
from keras.layers import Dense, Flatten
import keras
from keras import layers, models
import matplotlib.pyplot as plt
```

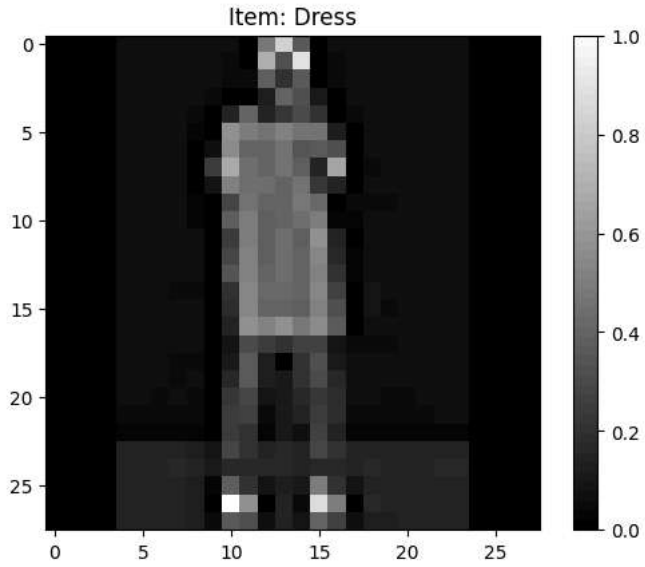
```
Num GPUs Available: 1
```

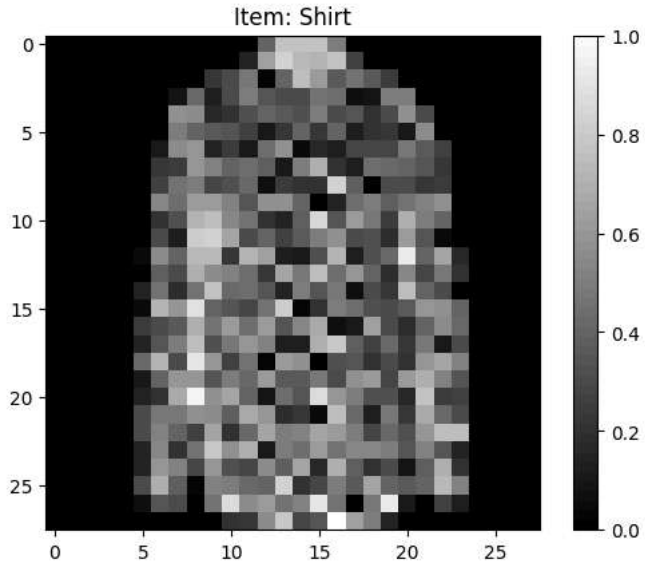
```
from tensorflow.keras.datasets import fashion_mnist
(X_train, _y_train), (X_test, _y_test) = fashion_mnist.load_data()

# Normalize the pixel values to be between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Classes in the Fashion MNIST dataset
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]


# Inspect the shapes of the datasets
print('Shapes of X_train:{}, X_test:{}'.format(X_train.shape, X_test.shape))

# Convert labels to one-hot encoding
from tensorflow.keras.utils import to_categorical

# # transform y
classes = len(class_names)
y_train = to_categorical(_y_train.flatten(), num_classes=classes)
y_test = to_categorical(_y_test.flatten(), num_classes=classes)
```

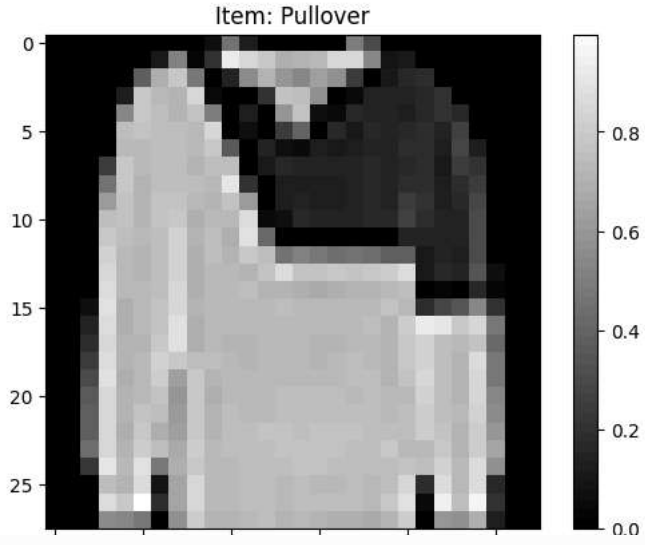
```
Shapes of X_train:(60000, 28, 28), X_test:(10000, 28, 28)
```

```
def plot_image(
    x: np.array,
    y: np.array,
    labels: np.array,
    idx: int = 0
```

```
    ):
    """Plot image
    """
    image_idx = idx
    plt.imshow(data[idx], cmap='gray')
    plt.colorbar()
    plt.title("Item: {}".format(labels[y[idx]]))
    plt.show()

data = X_train
labels = np.array(class_names)
for i in np.random.randint(0, data.shape[0], 3):
    plot_image(data, _y_train, labels, i)
```

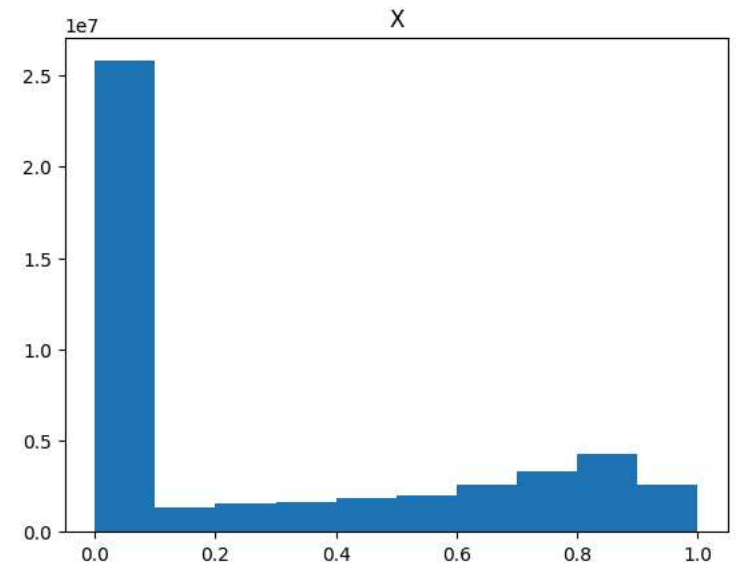


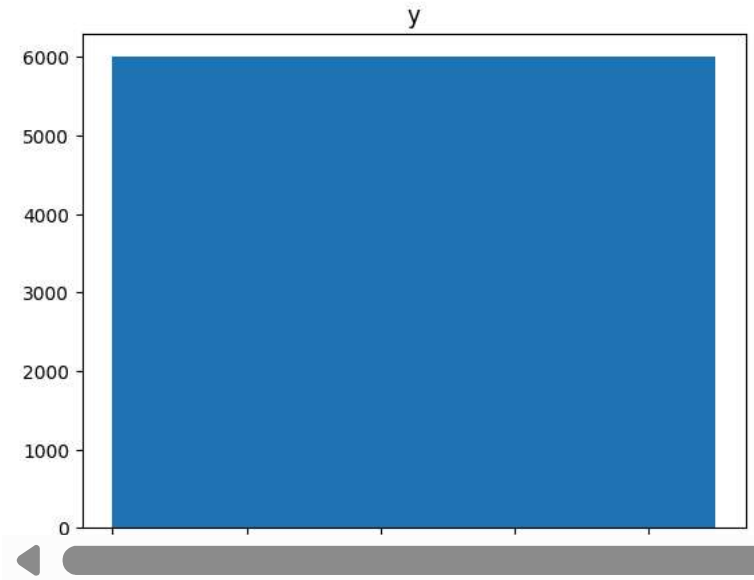





```
def plot_hist(
    array: np.array,
    title: str
    ):
    """Plot histogram for a given array
    """
```

```
    fig, ax = plt.subplots()
    ax.hist(array)
    plt.title(title)
    plt.show()

plot_hist(X_train.flatten(), 'X')
plot_hist(_y_train.flatten(), 'y')
```





Reflection

1. Does the data look as expected? How is the quality of the images?

- Yes. The images are in black and white (or just one channel) as X is given in dimensions `(n, n, 1)`.

2. Are there any issues with the dataset that you notice?

- The resolution is low and the absence of color (a second and third channel)could translate into a lack of information such that the model might not be able to differentiate between similar objects such as `"T-shirt/top"` and `"Shirt"`

2. Baseline Model

In this section, you will create a linear regression model as a baseline. This model will not use any convolutional layers, but it will help you understand the performance of a simple model on this dataset. You should:

- ☑ Create a simple linear regression model using Keras.
- ☑ Compile the model with an appropriate loss function and optimizer.
- ☑ Train the model on the training set and evaluate it on the test set.

A linear regression model can be created using the `Sequential` API in Keras. Using a single `Dense` layer with no activation function is equivalent to a simple linear regression model. Make sure that the number of units in the output layer matches the number of classes in the dataset.

Note that for this step, we will need to use `Flatten` to convert the 2D images into 1D vectors before passing them to the model. Put a `Flatten()` layer as the first layer in your model so that the 2D image data can be flattened into 1D vectors.

```
# from keras.models import Sequential
# from keras.layers import Dense, Flatten
# import keras
# from keras import layers, models

# Create a simple linear regression model
# model = Sequential()
# You can use `model.add(<layer>)` to add layers to the model
# Compile the model using `model.compile()`
# Train the model with `model.fit()`
# Evaluate the model with `model.evaluate()`
lr = 1e-2
model = models.Sequential()
model.add(
    layers.Flatten(input_shape=(28, 28)))
model.add(
    layers.Dense(56, activation='relu'))
model.add(
    layers.Dense(112, activation='relu'))
model.add(
    layers.Dense(len(class_names), activation='softmax'))

model.summary()

opt = keras.optimizers.Adam(learning_rate=lr)

model.compile(
    optimizer=opt,
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
    metrics=['accuracy', keras.metrics.F1Score, keras.metrics.AUC]
    )

history = model.fit(
    X_train,
    y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.2
    )
```

Model: "sequential_31"

Layer (type)	Output Shape	Param #
flatten_16 (Flatten)	(None, 784)	0
dense_73 (Dense)	(None, 56)	43,960
dense_74 (Dense)	(None, 112)	6,384
dense_75 (Dense)	(None, 10)	1,130

Total params: 51,474 (201.07 KB)
Trainable params: 51,474 (201.07 KB)
Non-trainable params: 0 (0.00 B)

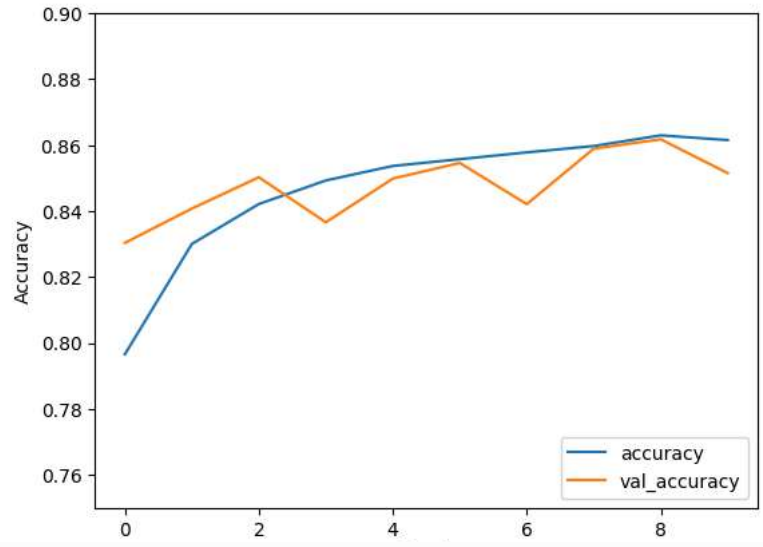
```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.7477 - auc_15: 0.9680 - f1_score: 0.7439 - loss: 0.6959 - val_accuracy: 0.8303
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8244 - auc_15: 0.9849 - f1_score: 0.8229 - loss: 0.4865 - val_accuracy: 0.8407
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.8414 - auc_15: 0.9875 - f1_score: 0.8388 - loss: 0.4400 - val_accuracy: 0.8503
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.8508 - auc_15: 0.9880 - f1_score: 0.8486 - loss: 0.4235 - val_accuracy: 0.8366
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.8536 - auc_15: 0.9889 - f1_score: 0.8511 - loss: 0.4068 - val_accuracy: 0.8498
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.8581 - auc_15: 0.9894 - f1_score: 0.8570 - loss: 0.3939 - val_accuracy: 0.8546
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.8621 - auc_15: 0.9901 - f1_score: 0.8617 - loss: 0.3846 - val_accuracy: 0.8421
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.8608 - auc_15: 0.9897 - f1_score: 0.8593 - loss: 0.3900 - val_accuracy: 0.8589
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.8650 - auc_15: 0.9901 - f1_score: 0.8635 - loss: 0.3776 - val_accuracy: 0.8618
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8640 - auc_15: 0.9903 - f1_score: 0.8629 - loss: 0.3786 - val_accuracy: 0.8515
```

```
test_loss, test_acc, test_f1, test_auc = model.evaluate(X_test,  y_test)
print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
    test_loss, test_acc, np.mean(np.array(test_f1)), test_auc))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8450 - auc_15: 0.9847 - f1_score: 0.8454 - loss: 0.4777
test_loss: 0.496, test_acc: 0.841, test_f1: 0.840, test_auc: 0.984
```

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.75, 0.9])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7fc78496d810>
```



Reflection: What is the performance of the baseline model? How does it compare to what you expected? Why do you think the performance is at this level?

Answer:

> - On test set: accuracy: `test_loss: 0.496, test_acc: 0.841, test_f1: 0.840, test_auc: 0.984`
> - It performs better than I expected considering that no hyperparameter tunning was performed. Also, it's a reasonable good perfomance (high accuracy, f1 score, and AUC) given that only Dense and not Conv2D layers were used
> - However, from the learning curve it is possible to say that the learning is not happening smoothly and that the model is slightly overfitting.

3. Building and Evaluating a Simple CNN Model

In this section, you will build a simple Convolutional Neural Network (CNN) model using Keras. A convolutional neural network is a type of deep learning model that is particularly effective for image classification tasks. Unlike the basic neural networks we have built in the labs, CNNs can accept images as input without needing to flatten them into vectors.

You should:

- ☑ Build a simple CNN model with at least one convolutional layer (to learn spatial hierarchies in images) and one fully connected layer (to make predictions).
- ☑ Compile the model with an appropriate loss function and metrics for a multi-class classification problem.
- ☑ Train the model on the training set and evaluate it on the test set.

Convolutional layers are designed to accept inputs with three dimensions: height, width and channels (e.g., RGB for color images). For grayscale images like those in Fashion MNIST, the input shape will be (28, 28, 1).

When you progress from the convolutional layers to the fully connected layers, you will need to flatten the output of the convolutional layers. This can be done using the `Flatten` layer in Keras, which doesn't require any parameters.

```
from keras.layers import Conv2D


# Reshape the data to include the channel dimension
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Create a simple CNN model
model = Sequential()

# Define params for baseline
# Pooling
pooling_params = {
    'pool_size': (2, 2),
    'strides': (1, 1)
    }

# Conv2D nodes:
conv2_filters = {
    1: 56,
    2: 112,
    3: 2224
    }
dense_nodes ={
    1: 56,
    2: 112,
    3: 224
}
# learning rate
lr = 1e-3

model = models.Sequential()
# 3 CNN w/default hyperparams
# 1
```

```
model.add(
    layers.Conv2D(
        conv2_filters[1],
        (2, 2),
        activation='relu',
        input_shape=(28, 28, 1)
        )
    )
model.add(
    layers.MaxPooling2D(**pooling_params)
    )
# 2
model.add(
    layers.Conv2D(
        conv2_filters[2],
        (2, 2),
        activation='relu'
        ))
model.add(
    layers.MaxPooling2D(**pooling_params)
    )
# 3
model.add(
    layers.Conv2D(
        conv2_filters[3],
        (2, 2),
        activation='relu',
        ))

# flatten layer
model.add(
    layers.Flatten())

# 3 Dense layers w/default hyperparams
# 1
model.add(
    layers.Dense(
        dense_nodes[1],
        activation='relu'
        ))
# 2
model.add(
    layers.Dense(
        dense_nodes[2],
        activation='relu'
        ))

# 3
model.add(
    layers.Dense(
        dense_nodes[3],
        activation='relu'
        ))

# output
model.add(
    layers.Dense(len(class_names), activation='softmax')
    )

model.summary()

opt = keras.optimizers.Adam(learning_rate=lr)

model.compile(
    optimizer=opt,
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
    metrics=['accuracy', keras.metrics.F1Score, keras.metrics.AUC]
    )

history = model.fit(
```

```
    X_train,
    y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.2
    )
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 27, 27, 56)	280
max_pooling2d (MaxPooling2D)	(None, 26, 26, 56)	0
conv2d_1 (Conv2D)	(None, 25, 25, 112)	25,200
max_pooling2d_1 (MaxPooling2D)	(None, 24, 24, 112)	0
conv2d_2 (Conv2D)	(None, 23, 23, 2224)	998,576
flatten (Flatten)	(None, 1176496)	0
dense (Dense)	(None, 56)	65,883,832
dense_1 (Dense)	(None, 112)	6,384
dense_2 (Dense)	(None, 224)	25,312
dense_3 (Dense)	(None, 10)	2,250

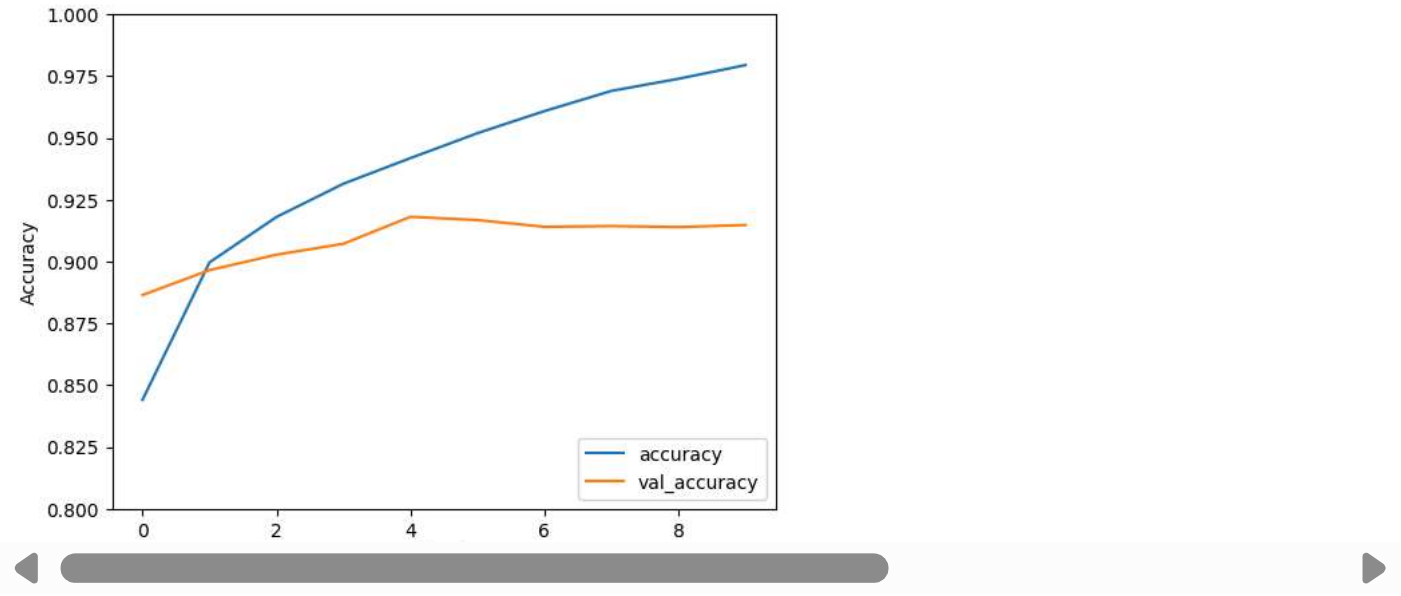
```
 Total params: 66,941,834 (255.36 MB)
 Trainable params: 66,941,834 (255.36 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 99s 60ms/step - accuracy: 0.7809 - auc: 0.9711 - f1_score: 0.7781 - loss: 0.6236 - val_accuracy: 0.8864 -
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 134s 59ms/step - accuracy: 0.9001 - auc: 0.9946 - f1_score: 0.8996 - loss: 0.2743 - val_accuracy: 0.8965
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 58ms/step - accuracy: 0.9151 - auc: 0.9961 - f1_score: 0.9152 - loss: 0.2253 - val_accuracy: 0.9028
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 87s 58ms/step - accuracy: 0.9351 - auc: 0.9973 - f1_score: 0.9344 - loss: 0.1805 - val_accuracy: 0.9072 -
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 58ms/step - accuracy: 0.9431 - auc: 0.9980 - f1_score: 0.9432 - loss: 0.1536 - val_accuracy: 0.9181
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 144s 60ms/step - accuracy: 0.9558 - auc: 0.9984 - f1_score: 0.9555 - loss: 0.1207 - val_accuracy: 0.9168
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 90s 60ms/step - accuracy: 0.9643 - auc: 0.9990 - f1_score: 0.9638 - loss: 0.0985 - val_accuracy: 0.9140 -
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 60ms/step - accuracy: 0.9708 - auc: 0.9991 - f1_score: 0.9707 - loss: 0.0796 - val_accuracy: 0.9143
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 147s 64ms/step - accuracy: 0.9757 - auc: 0.9995 - f1_score: 0.9755 - loss: 0.0644 - val_accuracy: 0.9139
Epoch 10/10
```

```
test_loss, test_acc, test_f1, test_auc = model.evaluate(X_test,  y_test)
print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
    test_loss, test_acc, np.mean(np.array(test_f1)), test_auc))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 26ms/step - accuracy: 0.9081 - auc: 0.9825 - f1_score: 0.9084 - loss: 0.4733
test_loss: 0.478, test_acc: 0.909, test_f1: 0.909, test_auc: 0.982
```

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7ab0182f9e90>
```



Reflection: Did the CNN model perform better than the baseline model? If so, by how much? What do you think contributed to this improvement?

Answer

- Previous:
  - test_loss: 0.496, test_acc: 0.841, test_f1: 0.840, test_auc: 0.984
- CNN:
  - test_loss: 0.478, test_acc: 0.909, test_f1: 0.909, test_auc: 0.982
- Overall, the resulting accuracy, f1 score and AUC are higher than the Dense layer-based model, however, as the learning curve shows, the CNN model is **overfitting** noticeably.
- The the addition of Conv2D layers, which help to learn more information from the pictures as 2D objects (grids), might be reason behind such overfitting.

3. Designing and Running Controlled Experiments

In this section, you will design and run controlled experiments to improve the model's performance. You will focus on one hyperparameter and one regularization technique. You should:

- ☑ Choose one hyperparameter to experiment with (e.g., number of filters, kernel size, number of layers, etc.) and one regularization technique (e.g., dropout, L2 regularization). For your hyperparameter, you should choose at least three different values to test (but there is no upper limit). For your regularization technique, simply test the presence or absence of the technique.
- ☑ Run experiments by modifying the model architecture or hyperparameters, and evaluate the performance of each model on the test set.
- ☑ Record the results of your experiments, including the test accuracy and any other relevant metrics.
- ☑ Visualize the results of your experiments using plots or tables to compare the performance of different models.

The best way to run your experiments is to create a `for` loop that iterates over a range of values for the hyperparameter you are testing. For example, if you are testing different numbers of filters, you can create a loop that runs the model with 32, 64, and 128 filters. Within the loop, you can compile and train the model, then evaluate it on the test set. After each iteration, you can store the results in a list or a dictionary for later analysis.

Note: It's critical that you re-initialize the model (by creating a new instance of the model) before each experiment. If you don't, the model will retain the weights from the previous experiment, which can lead to misleading results.

```
# A. Test Hyperparameters
```

```
# Reshape the data to include the channel dimension
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

def create_model(
      lr: float = 1e-2,
      kernels = (2, 2),
      conv2_filter_1= 56,
      conv2_filter_2= 112,
      conv2_filter_3= 2224,
      dense_nodes_1= 56,
      dense_nodes_2= 112,
      dense_nodes_3= 224,
      initializer_scheme=keras.initializers.GlorotUniform()
      ):

    model = models.Sequential()
    # 3 CNN
    # 1
    model.add(
        layers.Conv2D(
            conv2_filter_1,
            kernels,
            activation='relu',
            input_shape=(28, 28, 1),
            kernel_initializer=initializer_scheme
            )
        )
    model.add(
        layers.MaxPooling2D(
            pool_size= (2, 2),
            strides= (1, 1)
            )
        )
    # 2
    model.add(
        layers.Conv2D(
            conv2_filter_2,
            kernels,
            activation='relu',
            kernel_initializer=initializer_scheme
            ))
    model.add(
        layers.MaxPooling2D(
            pool_size= (2, 2),
            strides= (1, 1)
        )
        )
    # 3
    model.add(
        layers.Conv2D(
            conv2_filter_3,
            kernels,
            activation='relu',
            kernel_initializer=initializer_scheme
            ))

    # flattening layer
    model.add(
        layers.Flatten())

    # 3 Dense layers
    # 1
    model.add(
        layers.Dense(
            dense_nodes_1,
            activation='relu',
            kernel_initializer=initializer_scheme
            ))
    # 2
```

```
    model.add(
        layers.Dense(
            dense_nodes_2,
            activation='relu',
            kernel_initializer=initializer_scheme
            ))

    # 3
    model.add(
        layers.Dense(
            dense_nodes_3,
            activation='relu',
            kernel_initializer=initializer_scheme
            ))

    # output
    model.add(
        layers.Dense(len(class_names), activation='softmax')
        )

    # model.summary()

    opt = keras.optimizers.Adam(learning_rate=lr)

    model.compile(
        optimizer=opt,
        loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
        metrics=['accuracy', keras.metrics.F1Score, keras.metrics.AUC]
        )
    return model

def train_model(
    model,
    X_train,
    y_train
    ):
    # model = create_model()
    return model.fit(
        X_train,
        y_train,
        epochs=10,
        batch_size=32,
        validation_split=0.2
        )

def validate_model(model, X_test, y_test):
        test_loss, test_acc, test_f1, test_auc = model.evaluate(X_test,  y_test)
        return test_loss, test_acc, test_f1, test_auc


# Tunning
# learning rate
lrs = [0.1, 1e-2, 1e-3]

for lr_i in lrs:
    model = create_model(lr=lr_i)
    trained_model = train_model(model, X_train, y_train)
    test_loss, test_acc, test_f1, test_auc = validate_model(trained_model, X_test, y_test)
    print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
        test_loss, test_acc, np.mean(np.array(test_f1)), test_auc
        )
    )
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`in
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 134s 59ms/step - accuracy: 0.1021 - auc_1: 0.5014 - f1_score: 0.0932 - loss: 29340.1211 - val_accuracy:
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 136s 59ms/step - accuracy: 0.1009 - auc_1: 0.4993 - f1_score: 0.0939 - loss: 2.3167 - val_accuracy: 0.1
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 88s 59ms/step - accuracy: 0.0987 - auc_1: 0.4983 - f1_score: 0.0943 - loss: 2.3153 - val_accuracy: 0.10
```

```
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 140s 58ms/step - accuracy: 0.0960 - auc_1: 0.4978 - f1_score: 0.0915 - loss: 2.3150 - val_accuracy: 0.1
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 58ms/step - accuracy: 0.0996 - auc_1: 0.4969 - f1_score: 0.0934 - loss: 2.3165 - val_accuracy: 0.1
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 88s 59ms/step - accuracy: 0.0999 - auc_1: 0.4983 - f1_score: 0.0953 - loss: 2.3163 - val_accuracy: 0.09
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 58ms/step - accuracy: 0.1004 - auc_1: 0.4996 - f1_score: 0.0939 - loss: 2.3145 - val_accuracy: 0.1
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 87s 58ms/step - accuracy: 0.1011 - auc_1: 0.4989 - f1_score: 0.0953 - loss: 2.3160 - val_accuracy: 0.09
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 58ms/step - accuracy: 0.0984 - auc_1: 0.4993 - f1_score: 0.0905 - loss: 2.3136 - val_accuracy: 0.0
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 143s 59ms/step - accuracy: 0.0972 - auc_1: 0.4972 - f1_score: 0.0910 - loss: 2.3166 - val_accuracy: 0.0
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 21ms/step - accuracy: 0.0998 - auc_1: 0.5029 - f1_score: 0.0181 - loss: 2.3104
test_loss: 2.312, test_acc: 0.100, test_f1: 0.018, test_auc: 0.500
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 94s 60ms/step - accuracy: 0.6609 - auc_2: 0.9383 - f1_score: 0.6567 - loss: 2.0234 - val_accuracy: 0.82
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 138s 59ms/step - accuracy: 0.8413 - auc_2: 0.9866 - f1_score: 0.8404 - loss: 0.4486 - val_accuracy: 0.8
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 90s 60ms/step - accuracy: 0.8482 - auc_2: 0.9874 - f1_score: 0.8474 - loss: 0.4324 - val_accuracy: 0.82
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 140s 59ms/step - accuracy: 0.8555 - auc_2: 0.9885 - f1_score: 0.8537 - loss: 0.4143 - val_accuracy: 0.8
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 90s 60ms/step - accuracy: 0.8622 - auc_2: 0.9895 - f1_score: 0.8608 - loss: 0.3894 - val_accuracy: 0.86
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 60ms/step - accuracy: 0.8698 - auc_2: 0.9901 - f1_score: 0.8689 - loss: 0.3783 - val_accuracy: 0.8
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 91s 60ms/step - accuracy: 0.8717 - auc_2: 0.9908 - f1_score: 0.8705 - loss: 0.3639 - val_accuracy: 0.86
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 139s 58ms/step - accuracy: 0.8738 - auc_2: 0.9910 - f1_score: 0.8737 - loss: 0.3619 - val_accuracy: 0.8
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.8688 - auc_2: 0.9898 - f1_score: 0.8687 - loss: 0.3833 - val_accuracy: 0.8
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.8773 - auc_2: 0.9911 - f1_score: 0.8772 - loss: 0.3533 - val_accuracy: 0.8
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 15ms/step - accuracy: 0.8717 - auc_2: 0.9903 - f1_score: 0.8718 - loss: 0.3714
test_loss: 0.377, test_acc: 0.869, test_f1: 0.868, test_auc: 0.990
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 96s 62ms/step - accuracy: 0.7894 - auc_3: 0.9748 - f1_score: 0.7863 - loss: 0.5835 - val_accuracy: 0.87
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 90s 60ms/step - accuracy: 0.9022 - auc_3: 0.9945 - f1_score: 0.9024 - loss: 0.2707 - val_accuracy: 0.90
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 91s 61ms/step - accuracy: 0.9221 - auc_3: 0.9963 - f1_score: 0.9214 - loss: 0.2125 - val_accuracy: 0.88
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 60ms/step - accuracy: 0.9348 - auc_3: 0.9972 - f1_score: 0.9340 - loss: 0.1769 - val_accuracy: 0.9
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 60ms/step - accuracy: 0.9493 - auc_3: 0.9982 - f1_score: 0.9489 - loss: 0.1368 - val_accuracy: 0.9
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 60ms/step - accuracy: 0.9618 - auc_3: 0.9986 - f1_score: 0.9616 - loss: 0.1090 - val_accuracy: 0.9
```

```
# checking initializers

lr_i = 1e-3  # From previous training
initializers = [
    keras.initializers.HeNormal(seed=None),
    keras.initializers.TruncatedNormal(mean=0., stddev=1.)
    ]

for initializer_i in initializers:
    model = create_model(initializer_scheme=initializer_i, lr=lr_i)
    trained_model = train_model(model, X_train, y_train)
    test_loss, test_acc, test_f1, test_auc = validate_model(trained_model, X_test, y_test)
    print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
        test_loss, test_acc, np.mean(np.array(test_f1)), test_auc
        )
    )
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 98s 63ms/step - accuracy: 0.7798 - auc_4: 0.9545 - f1_score: 0.7772 - loss: 5.8737 - val_accuracy: 0.8858
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 135s 59ms/step - accuracy: 0.8958 - auc_4: 0.9935 - f1_score: 0.8952 - loss: 0.2951 - val_accuracy: 0.889
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 59ms/step - accuracy: 0.9102 - auc_4: 0.9952 - f1_score: 0.9106 - loss: 0.2471 - val_accuracy: 0.894
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 90s 60ms/step - accuracy: 0.9236 - auc_4: 0.9965 - f1_score: 0.9238 - loss: 0.2074 - val_accuracy: 0.9048
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 59ms/step - accuracy: 0.9315 - auc_4: 0.9972 - f1_score: 0.9316 - loss: 0.1815 - val_accuracy: 0.901
```

```
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.9423 - auc_4: 0.9975 - f1_score: 0.9420 - loss: 0.1582 - val_accuracy: 0.903
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.9538 - auc_4: 0.9981 - f1_score: 0.9536 - loss: 0.1265 - val_accuracy: 0.909
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 59ms/step - accuracy: 0.9669 - auc_4: 0.9988 - f1_score: 0.9667 - loss: 0.0952 - val_accuracy: 0.903
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.9702 - auc_4: 0.9990 - f1_score: 0.9701 - loss: 0.0863 - val_accuracy: 0.902
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.9760 - auc_4: 0.9993 - f1_score: 0.9761 - loss: 0.0669 - val_accuracy: 0.903
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 15ms/step - accuracy: 0.9022 - auc_4: 0.9821 - f1_score: 0.9046 - loss: 0.4453
test_loss: 0.451, test_acc: 0.903, test_f1: 0.905, test_auc: 0.982
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 99s 63ms/step - accuracy: 0.6621 - auc_5: 0.8123 - f1_score: 0.6621 - loss: 123175.2812 - val_accuracy: 0
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 134s 59ms/step - accuracy: 0.5970 - auc_5: 0.8291 - f1_score: 0.6061 - loss: 5650.1899 - val_accuracy: 0.
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.1749 - auc_5: 0.5988 - f1_score: 0.1736 - loss: 13.1097 - val_accuracy: 0.16
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 59ms/step - accuracy: 0.1675 - auc_5: 0.5925 - f1_score: 0.1536 - loss: 2.5935 - val_accuracy: 0.153
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.1825 - auc_5: 0.6125 - f1_score: 0.1736 - loss: 3.2386 - val_accuracy: 0.199
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 89s 59ms/step - accuracy: 0.1833 - auc_5: 0.6166 - f1_score: 0.1922 - loss: 2.5591 - val_accuracy: 0.1058
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 59ms/step - accuracy: 0.1114 - auc_5: 0.5110 - f1_score: 0.0887 - loss: 2.2846 - val_accuracy: 0.105
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.1082 - auc_5: 0.5127 - f1_score: 0.0933 - loss: 2.2849 - val_accuracy: 0.105
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 59ms/step - accuracy: 0.1073 - auc_5: 0.5106 - f1_score: 0.0878 - loss: 2.2847 - val_accuracy: 0.108
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 143s 60ms/step - accuracy: 0.1048 - auc_5: 0.5101 - f1_score: 0.0890 - loss: 2.2846 - val_accuracy: 0.107
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 16ms/step - accuracy: 0.1047 - auc_5: 0.5057 - f1_score: 0.0303 - loss: 2.3995
test_loss: 2.770, test_acc: 0.107, test_f1: 0.031, test_auc: 0.507
```

```
# B. Test presence or absence of regularization

# Reshape the data to include the channel dimension
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

def create_model(
      lr: float = 1e-2,
      kernels = (2, 2),
      conv2_filter_1= 56,
      conv2_filter_2= 112,
      conv2_filter_3= 2224,
      dense_nodes_1= 56,
      dense_nodes_2= 112,
      dense_nodes_3= 224,
      initializer_scheme=keras.initializers.GlorotUniform(),
      dropout_rate = 0.
      ):

    model = models.Sequential()
    # 3 CNN
    # 1
    model.add(
        layers.Conv2D(
            conv2_filter_1,
            kernels,
            activation='relu',
            input_shape=(28, 28, 1),
            kernel_initializer=initializer_scheme
            )
        )
    model.add(
        layers.MaxPooling2D(
            pool_size= (2, 2),
            strides= (1, 1)
            )
        )
    model.add(layers.SpatialDropout2D(dropout_rate))
    # 2
```

```
model.add(
    layers.Conv2D(
        conv2_filter_2,
        kernels,
        activation='relu',
        kernel_initializer=initializer_scheme
        ))
model.add(
    layers.MaxPooling2D(
        pool_size= (2, 2),
        strides= (1, 1)
    )
    )
model.add(layers.SpatialDropout2D(dropout_rate))
# 3
model.add(
    layers.Conv2D(
        conv2_filter_3,
        kernels,
        activation='relu',
        kernel_initializer=initializer_scheme
        ))

# flattening layer
model.add(
    layers.Flatten())

# 3 Dense layers
# 1
model.add(
layers.BatchNormalization()
)
model.add(
    layers.Dense(
        dense_nodes_1,
        activation='relu',
        kernel_initializer=initializer_scheme
        ))
model.add(layers.Dropout(rate=dropout_rate))
# 2
model.add(
layers.BatchNormalization()
)
model.add(
    layers.Dense(
        dense_nodes_2,
        activation='relu',
        kernel_initializer=initializer_scheme
        ))
model.add(layers.Dropout(rate=dropout_rate))
# 3
model.add(
layers.BatchNormalization()
)
model.add(
    layers.Dense(
        dense_nodes_3,
        activation='relu',
        kernel_initializer=initializer_scheme
        ))
model.add(layers.Dropout(rate=dropout_rate))
# output
model.add(
    layers.Dense(len(class_names), activation='softmax')
    )

# model.summary()

opt = keras.optimizers.Adam(learning_rate=lr)
```

```
    model.compile(
        optimizer=opt,
        loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
        metrics=['accuracy', keras.metrics.F1Score, keras.metrics.AUC]
        )
    return model

def train_model(
    model,
    X_train,
    y_train
    ):
    # model = create_model()
    return model.fit(
        X_train,
        y_train,
        epochs=10,
        batch_size=32,
        validation_split=0.2
        )

def validate_model(model, X_test, y_test):
        test_loss, test_acc, test_f1, test_auc = model.evaluate(X_test,  y_test)
        return test_loss, test_acc, test_f1, test_auc
```

> absence of regularization has been tested in A

```
# test batchnormalization
lr_i = 1e-3
initializer_i = keras.initializers.GlorotUniform()
dropout_i = 0.0

model = create_model(
        lr=lr_i, initializer_scheme=initializer_i, dropout_rate=dropout_i
        )
trained_model = train_model(model, X_train, y_train)
test_loss, test_acc, test_f1, test_auc = validate_model(trained_model.model, X_test, y_test)
print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
    test_loss, test_acc, np.mean(np.array(test_f1)), test_auc
    )
)
```

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 118s 75ms/step - accuracy: 0.8018 - auc_6: 0.9794 - f1_score: 0.8005 - loss: 0.5448 - val_accuracy: 0.889
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 117s 78ms/step - accuracy: 0.8893 - auc_6: 0.9937 - f1_score: 0.8892 - loss: 0.3009 - val_accuracy: 0.904
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 143s 79ms/step - accuracy: 0.9097 - auc_6: 0.9954 - f1_score: 0.9091 - loss: 0.2456 - val_accuracy: 0.905
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 136s 75ms/step - accuracy: 0.9220 - auc_6: 0.9966 - f1_score: 0.9221 - loss: 0.2111 - val_accuracy: 0.908
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 112s 75ms/step - accuracy: 0.9301 - auc_6: 0.9970 - f1_score: 0.9300 - loss: 0.1898 - val_accuracy: 0.918
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 74ms/step - accuracy: 0.9400 - auc_6: 0.9977 - f1_score: 0.9400 - loss: 0.1671 - val_accuracy: 0.924
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 146s 77ms/step - accuracy: 0.9468 - auc_6: 0.9982 - f1_score: 0.9468 - loss: 0.1455 - val_accuracy: 0.909
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 77ms/step - accuracy: 0.9548 - auc_6: 0.9985 - f1_score: 0.9546 - loss: 0.1226 - val_accuracy: 0.923
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 77ms/step - accuracy: 0.9621 - auc_6: 0.9989 - f1_score: 0.9620 - loss: 0.1028 - val_accuracy: 0.922
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 77ms/step - accuracy: 0.9663 - auc_6: 0.9991 - f1_score: 0.9663 - loss: 0.0914 - val_accuracy: 0.909
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 20ms/step - accuracy: 0.8966 - auc_6: 0.9877 - f1_score: 0.8966 - loss: 0.3625
test_loss: 0.352, test_acc: 0.900, test_f1: 0.899, test_auc: 0.989
```

```
# test dropout
lr_i = 1e-3
initializer_i = keras.initializers.GlorotUniform()
dropouts = [0.1, 0.2, 0.5]
```

```
for dropout_i in dropouts:
    model = create_model(
        lr=lr_i, initializer_scheme=initializer_i, dropout_rate=dropout_i
        )
    trained_model = train_model(model, X_train, y_train)
    test_loss, test_acc, test_f1, test_auc = validate_model(trained_model.model, X_test, y_test)
    print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
        test_loss, test_acc, np.mean(np.array(test_f1)), test_auc
        )
    )
```

```
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 146s 79ms/step - accuracy: 0.9086 - auc_10: 0.9952 - f1_score: 0.9078 - loss: 0.2525 - val_accuracy: 0.
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 79ms/step - accuracy: 0.9141 - auc_10: 0.9956 - f1_score: 0.9143 - loss: 0.2341 - val_accuracy: 0.
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 79ms/step - accuracy: 0.9229 - auc_10: 0.9964 - f1_score: 0.9226 - loss: 0.2118 - val_accuracy: 0.
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 138s 76ms/step - accuracy: 0.9302 - auc_10: 0.9965 - f1_score: 0.9300 - loss: 0.1930 - val_accuracy: 0.
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 146s 79ms/step - accuracy: 0.9354 - auc_10: 0.9975 - f1_score: 0.9356 - loss: 0.1751 - val_accuracy: 0.
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 79ms/step - accuracy: 0.9405 - auc_10: 0.9978 - f1_score: 0.9401 - loss: 0.1609 - val_accuracy: 0.
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 21ms/step - accuracy: 0.9113 - auc_10: 0.9946 - f1_score: 0.9119 - loss: 0.2521
test_loss: 0.245, test_acc: 0.913, test_f1: 0.913, test_auc: 0.995
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 137s 80ms/step - accuracy: 0.6802 - auc_11: 0.9508 - f1_score: 0.6757 - loss: 0.8641 - val_accuracy: 0.
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 134s 78ms/step - accuracy: 0.8458 - auc_11: 0.9880 - f1_score: 0.8447 - loss: 0.4293 - val_accuracy: 0.
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 145s 80ms/step - accuracy: 0.8681 - auc_11: 0.9905 - f1_score: 0.8682 - loss: 0.3722 - val_accuracy: 0.
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 137s 77ms/step - accuracy: 0.8766 - auc_11: 0.9920 - f1_score: 0.8767 - loss: 0.3422 - val_accuracy: 0.
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 119s 80ms/step - accuracy: 0.8910 - auc_11: 0.9932 - f1_score: 0.8900 - loss: 0.3059 - val_accuracy: 0.
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 80ms/step - accuracy: 0.8967 - auc_11: 0.9936 - f1_score: 0.8949 - loss: 0.2918 - val_accuracy: 0.
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 119s 79ms/step - accuracy: 0.9036 - auc_11: 0.9943 - f1_score: 0.9034 - loss: 0.2748 - val_accuracy: 0.
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 139s 78ms/step - accuracy: 0.9088 - auc_11: 0.9953 - f1_score: 0.9085 - loss: 0.2508 - val_accuracy: 0.
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 77ms/step - accuracy: 0.9196 - auc_11: 0.9961 - f1_score: 0.9193 - loss: 0.2256 - val_accuracy: 0.
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 115s 77ms/step - accuracy: 0.9195 - auc_11: 0.9959 - f1_score: 0.9185 - loss: 0.2198 - val_accuracy: 0.
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 21ms/step - accuracy: 0.9168 - auc_11: 0.9946 - f1_score: 0.9165 - loss: 0.2439
test_loss: 0.237, test_acc: 0.920, test_f1: 0.919, test_auc: 0.995
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 127s 79ms/step - accuracy: 0.3931 - auc_12: 0.8320 - f1_score: 0.3874 - loss: 1.6344 - val_accuracy: 0.
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 136s 78ms/step - accuracy: 0.6955 - auc_12: 0.9604 - f1_score: 0.6799 - loss: 0.8271 - val_accuracy: 0.
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 145s 81ms/step - accuracy: 0.7350 - auc_12: 0.9690 - f1_score: 0.7207 - loss: 0.7231 - val_accuracy: 0.
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 141s 80ms/step - accuracy: 0.7663 - auc_12: 0.9750 - f1_score: 0.7522 - loss: 0.6457 - val_accuracy: 0.
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 80ms/step - accuracy: 0.7768 - auc_12: 0.9766 - f1_score: 0.7707 - loss: 0.6215 - val_accuracy: 0.
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 80ms/step - accuracy: 0.7926 - auc_12: 0.9788 - f1_score: 0.7861 - loss: 0.5893 - val_accuracy: 0.
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 80ms/step - accuracy: 0.8069 - auc_12: 0.9803 - f1_score: 0.8031 - loss: 0.5620 - val_accuracy: 0.
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 138s 77ms/step - accuracy: 0.8170 - auc_12: 0.9814 - f1_score: 0.8134 - loss: 0.5351 - val_accuracy: 0.
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 77ms/step - accuracy: 0.8288 - auc_12: 0.9840 - f1_score: 0.8254 - loss: 0.4995 - val_accuracy: 0.
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 116s 77ms/step - accuracy: 0.8315 - auc_12: 0.9842 - f1_score: 0.8292 - loss: 0.4947 - val_accuracy: 0.
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 21ms/step - accuracy: 0.8793 - auc_12: 0.9917 - f1_score: 0.8799 - loss: 0.3727
test_loss: 0.370, test_acc: 0.875, test_f1: 0.875, test_auc: 0.992
```

Reflection: Report on the performance of the models you tested. Did any of the changes you made improve the model's performance? If so, which ones? What do you think contributed to these improvements? Finally, what combination of hyperparameters and regularization techniques yielded the best performance?

Answer

- Yes, the learning rate, batchnorm and dropout.

> - Altogether, a good performance was achieved at the time overfitting was reduced in comparison with the baseline It has to be noted that the approach followed here for the selection of hyperparameters was in a sense naive, i.e., the parameters were optimized separately which neglects any "collaborative" effects between differenet hyperparameters

5. Training Final Model and Evaluation

In this section, you will train the final model using the best hyperparameters and regularization techniques you found in the previous section. You should:

- ☑ Compile the final model with the best hyperparameters and regularization techniques.
- ☑ Train the final model on the training set and evaluate it on the test set.
- ☑ Report the final model's performance on the test set, including accuracy and any other relevant metrics.

```
lr_i = 1e-3
initializer_i = keras.initializers.GlorotUniform()
dropout_i = 0.2

model = create_model(
    lr=lr_i, initializer_scheme=initializer_i, dropout_rate=dropout_i
    )
trained_model = train_model(model, X_train, y_train)
test_loss, test_acc, test_f1, test_auc = validate_model(trained_model.model, X_test, y_test)
print('test_loss: {:.3f}, test_acc: {:.3f}, test_f1: {:.3f}, test_auc: {:.3f}'.format(
    test_loss, test_acc, np.mean(np.array(test_f1)), test_auc
    )
)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 130s 80ms/step - accuracy: 0.6890 - auc_13: 0.9536 - f1_score: 0.6835 - loss: 0.8370 - val_accuracy: 0.85
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 120s 80ms/step - accuracy: 0.8470 - auc_13: 0.9876 - f1_score: 0.8456 - loss: 0.4348 - val_accuracy: 0.88
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 137s 77ms/step - accuracy: 0.8678 - auc_13: 0.9904 - f1_score: 0.8675 - loss: 0.3724 - val_accuracy: 0.88
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 145s 79ms/step - accuracy: 0.8816 - auc_13: 0.9923 - f1_score: 0.8800 - loss: 0.3299 - val_accuracy: 0.91
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 79ms/step - accuracy: 0.8914 - auc_13: 0.9935 - f1_score: 0.8909 - loss: 0.3041 - val_accuracy: 0.91
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 119s 79ms/step - accuracy: 0.8976 - auc_13: 0.9941 - f1_score: 0.8969 - loss: 0.2851 - val_accuracy: 0.91
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 138s 76ms/step - accuracy: 0.9056 - auc_13: 0.9944 - f1_score: 0.9055 - loss: 0.2680 - val_accuracy: 0.91
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 146s 79ms/step - accuracy: 0.9125 - auc_13: 0.9951 - f1_score: 0.9123 - loss: 0.2482 - val_accuracy: 0.92
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 79ms/step - accuracy: 0.9220 - auc_13: 0.9958 - f1_score: 0.9214 - loss: 0.2236 - val_accuracy: 0.92
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 142s 79ms/step - accuracy: 0.9226 - auc_13: 0.9961 - f1_score: 0.9221 - loss: 0.2180 - val_accuracy: 0.92
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 20ms/step - accuracy: 0.9186 - auc_13: 0.9949 - f1_score: 0.9181 - loss: 0.2314
test_loss: 0.228, test_acc: 0.919, test_f1: 0.918, test_auc: 0.995
```

```
plt.plot(trained_model.history['accuracy'], label='accuracy')
plt.plot(trained_model.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.75, 1.0])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7a2ea4efd550>
```

Reflection: How does the final model's performance compare to the baseline and the CNN model? What do you think contributed to the final model's performance? If you had time, what other experiments would you run to further improve the model's performance?

Answer

- The accuracy, f1 score, and AUC don't improve as much (>0.5 units), however, the model tends to overfit less.
- The above is most likely the result of including `Dropout` layers for both the Conv2D and Dense layers sections
- Using `keras tuner` [illegible] d effects of hyperparameters, i.e., not optimizing the learning rate separately from the optimization of the dropout rate.
- I would also optimize the number of filters and nodes in the Conv2D and Dense layers repsectively.
- In addition, I would like to try other tools to evaluate the performance of the model, for example, confusion matrices:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import pandas as pd

y_pred_probs = trained_model.model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1)
cm = confusion_matrix(_y_test, y_pred)

df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)

plt.figure(figsize=(10, 8))
sns.heatmap(df_cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.show()
```