ca1_dat300_2025

September 30, 2025

1 Compulsory Assignment 1: Dense neural networks - Implementing an ANN with Keras

Please fill out the the group name, number, members and optionally the group name below.

Group number 20
Group member 1: Eirik Årdal Hjelm
Group member 2: Gabriel Fayomi
Group member 3: Mads Teien

2 Assignment submission

To complete this assignment, answer the all the questions in this notebook and write the code required to implement different models. **Submit the assignment by handing in this notebook as both an .ipynb file and a .pdf file**.

Here are some do's and don'ts for the submission:

- Read questions thoroughly before answering.
- Make sure to answer all questions.
- Ensure all code cells are run.
- Label all axes in plots.
- Ensure all figures are visible in the PDF.
- Provide a brief explanation of how your code works

3 Introduction

In this assignment we will work with the task of classifying hand gestures from the Sign Language MNIST dataset. This time you will implement the network using the Keras API of the TensorFlow library. TensorFlow and PyTorch are both free open-source software libraries intended to simplify multiplication of tensors, but are mostly used for the design and implementation of deep neural networks. Both libraries simplify the implementation of neural networks, and allow for faster training of networks by utlizing hardware acceleration with Graphical Processing Units (GPUs) or Tensor Processing Units (TPUs)

TensorFlow was developed by Google Brain for internal use in Google and was initially released under Apache 2.0 License in 2015 [1]. Keras was initially released as separate software library, developed by François Chollet, to simplify the Python interface for design of artificial neural networks. Up until version 2.3 Keras supported multiple backend libraries including TensorFlow, Microsoft

Cognitive Toolkit, Theano, and PlaidML [2]. When TensorFlow 2.0 was released in 2019, keras was included as a TensorFlow specific API that is accessible by:

```
import tensorflow.keras as ks
```

PyTorch was originally developed by Meta AI (formerly known as Facebook) in 2016, but is now under umbrella of the Linux foundation, and is open-source under the BSD license [3]. While TensorFlow was the most popular framework for a long time, PyTorch has been gaining more and more users in the last five years and is now more used in industry and is becoming more popular in research as well.

The lectures of DAT300 will be taught using the Keras API in TensorFlow, and we recommend you to stick with Keras and TensorFlow for this course as it is easier for beginners to get started with.

4 Dataset descirption

The Sign Language MNIST dataset is a collection of grayscale images of size 28×28 pixels, representing hand gestures for the letters A–Y in American Sign Language (ASL). The letters J and Z are excluded since they involve motion.

- Training set: 27,455 images
- Test set: 7,172 images
- Number of classes: 24 (letters A–Y, excluding J and Z)
- Format: Each image is stored as a flattened vector of 784 pixels, with an accompanying label indicating the class (0–23).

This dataset is commonly used as a benchmark for image classification tasks, similar to the original MNIST handwritten digits dataset, but adapted for sign language recognition.

5 Assignment structure

1. Part 1: Import, preprocess, and visualize the data.
2. Part 2: Design your own Dense Neural Network (NN) architecture for classifying MNIST in Keras.
3. Part 3: Train one of the Machine Learning classifiers that you learned about in DAT200.
4. Part 4: Compare and discuss the results.

5.1 Supporting code

You may find the code below useful for plotting the metrics and calculating the F1-score for your model in Part 2.

[18]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score
```

```
import tensorflow as tf
import distutils as _distutils
import importlib
import inspect as _inspect

"""
A function that plots the training and validation metrics over epochs.
"""
def plot_training_history(training_history_object, list_of_metrics=None):

    """
    training_history_object: Object returned by model.fit() function in keras
    list_of_metrics: A list of MAX two metrics to be plotted
    """
    history_dict = training_history_object.history
    if list_of_metrics is None:
        list_of_metrics = [key for key in list(history_dict.keys()) if 'val_'␣
 ↪not in key]

    trainHistDF = pd.DataFrame(history_dict)
    train_keys = list_of_metrics
    valid_keys = ['val_' + key for key in train_keys]
    nr_plots = len(train_keys)
    fig, ax = plt.subplots(1,nr_plots,figsize=(5*nr_plots,4))
    for i in range(len(train_keys)):
        ax[i].plot(np.array(trainHistDF[train_keys[i]]), label='Training')
        ax[i].plot(np.array(trainHistDF[valid_keys[i]]), label='Validation')
        ax[i].set_xlabel('Epoch')
        ax[i].set_title(train_keys[i])
        ax[i].grid('on')
        ax[i].legend()
    fig.tight_layout
    plt.show()

"""
Custom Keras callback for computing the F1-score after each epoch.
This callback calculates both training and validation F1-scores
"""
class F1ScoreCallback(tf.keras.callbacks.Callback):
    def __init__(self, X_train, y_train, X_val, y_val):
        super().__init__()
        self.X_train, self.y_train = X_train, y_train
        self.X_val, self.y_val = X_val, y_val

    def on_epoch_end(self, epoch, logs=None):
        # Training F1
        y_train_pred = self.model.predict(self.X_train, verbose=0).
↪argmax(axis=1)
        f1_train = f1_score(self.y_train, y_train_pred, average="macro")

        # Validation F1
        y_val_pred = self.model.predict(self.X_val, verbose=0).argmax(axis=1)
        f1_val = f1_score(self.y_val, y_val_pred, average="macro")

        logs["f1"] = f1_train
        logs["val_f1"] = f1_val
        print(f" - f1: {f1_train:.4f} - val_f1: {f1_val:.4f}")
```

5.2 Library imports

[36]:
```
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Dropout, BatchNormalization
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
import time
```

6 Task 1: Importing, preprocess and visualizing the data

In this assignment you yourselves will be responsible for the data-preprocessing. Use the cells below for preprocessing and visualization, and optionally some exploration of the dataset if you feel inclined.

6.1 Importing data

You will need to upload both CSV files from the zip folder to Google Colab and run the code sell below to load the data.

[37]:
```
# Sign language MNIST
train = pd.read_csv("sign_mnist_train.csv")
test  = pd.read_csv("sign_mnist_test.csv")

X_train = train.drop(columns=['label']).to_numpy(dtype=np.float32)
y_train = train['label'].to_numpy(dtype=np.int32)
X_test  = test.drop(columns=['label']).to_numpy(dtype=np.float32)
```

```
y_test  = test['label'].to_numpy(dtype=np.int32)
```

[38]:
```
# Split train set into train + validation to monitor overfitting during training
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42, stratify=y_train
)
```

6.2 Task 1.1 Preprocessing

Preprocess the data in whatever way you find sensible. Remember to comment on what you do.

[39]:
```
# Normalize pixel values to [0, 1]
X_train = X_train / 255.0
X_test = X_test / 255.0
X_val = X_val / 255.0

# One hot encode the labels
y_train = y_train.reshape(-1, 1)  # make it 2D
y_test  = y_test.reshape(-1, 1)
y_val   = y_val.reshape(-1, 1)

encoder = OneHotEncoder(sparse_output=False)
y_train = encoder.fit_transform(y_train)
y_test  = encoder.transform(y_test)
y_val   = encoder.transform(y_val)

# Check that the dimensions are correct
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
print("y_val shape:", y_val.shape)
```

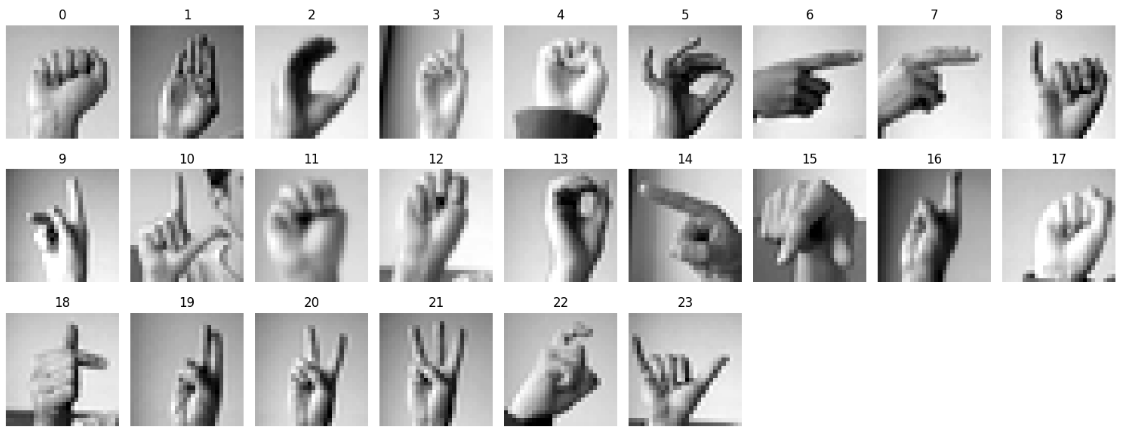
```
y_train shape: (21964, 24)
y_test shape: (7172, 24)
y_val shape: (5491, 24)
```

6.3 Task 1.2 Visualization

Visualize the data in whatever manner you find helpful/sensible and briefly comment on the plots.

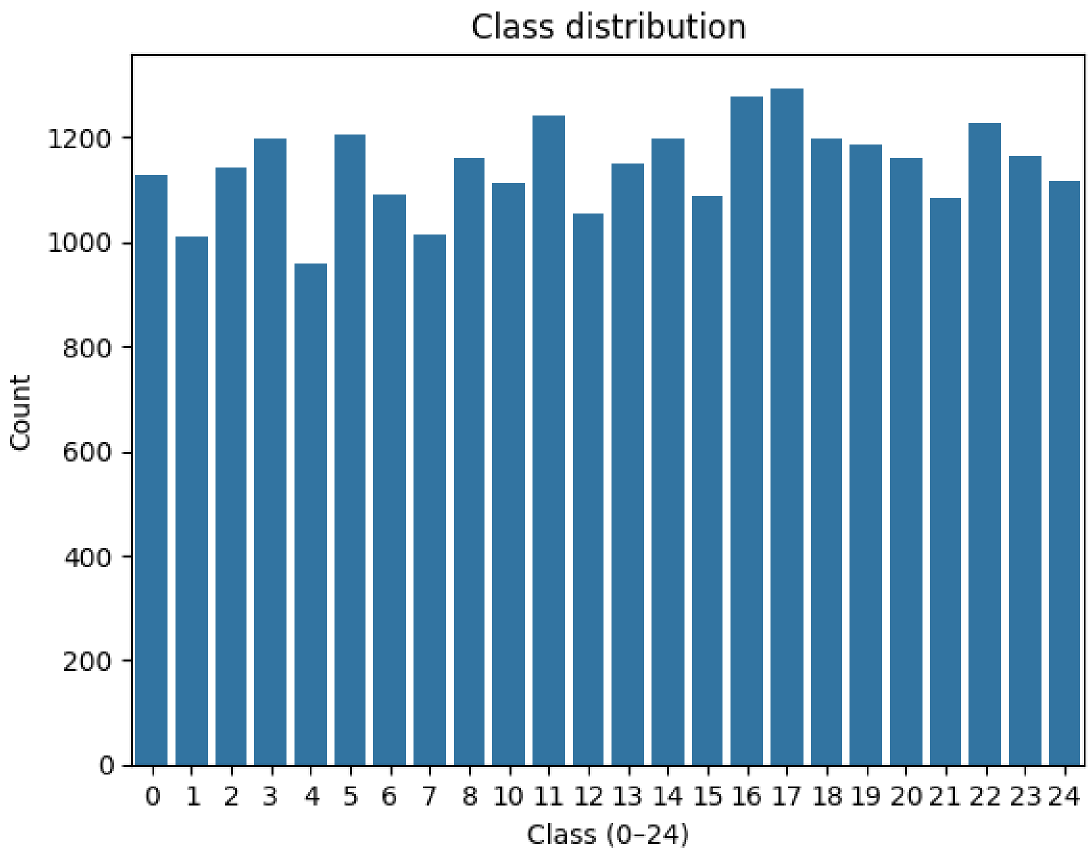
6.3.1 Visualizing the classes

[40]:
```
y_labels = np.argmax(y_train, axis=1)   # back to class labels
classes = np.unique(y_labels)

plt.figure(figsize=(15, 6))
for i, c in enumerate(classes):
    idx = np.where(y_labels == c)[0][0]
    plt.subplot(3, 9, i+1)
    plt.imshow(X_train[idx].reshape(28, 28), cmap="gray")
    plt.title(f"{c}")
    plt.axis("off")

plt.tight_layout()
plt.show()
```



6.3.2 Class distribution plot

[41]:
```
# Plot class distribution to see if the dataset is balanced
sns.countplot(x=train["label"])
plt.xlabel("Class (0-24)")
plt.ylabel("Count")
plt.title("Class distribution")
plt.show()
```



7 Task 2: Design your own ANN architecture

In this task you are free to design the network architecture for the MNIST Gesture recognition challenge with a couple of stipulations: * use **only Dense or fully connected layers**, * use both **accuracy and the F1-score** as performance metrics.

Otherwise, you are free to use whatever loss-function, optimizer and activation functions you want and train it for as many epochs you want.

7.1 Task 2.1: Implement your own network architecture

Design your network below:

(Feel free to add as many code and markdown cells as you want)

```
[111]: n_features = X_train.shape[1]    # number of features in the input data
       n_classes = y_train.shape[1]     # number of output classes (one-hot encoded)

       # Define the model architecture
       model = Sequential([
           Input(shape=(n_features,)),        # Input layer
```

```
    BatchNormalization(),
    Dense(256, activation='relu', kernel_regularizer=regularizers.l2(1e-4)),  ␣
 ↪ # Hidden layer 1
    BatchNormalization(),
    Dropout(0.1),
    Dense(128, activation='relu', kernel_regularizer=regularizers.l2(1e-4)),  ␣
 ↪ # Hidden layer 2
    Dense(n_classes, activation='softmax')  # Output layer
])

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=1e-3),
    loss='categorical_crossentropy',
    metrics=['accuracy']  # F1-score is added via callback
)

model.summary()
```

```
Model: "sequential_17"


 Layer (type)                      Output Shape                    Param #

 batch_normalization_23            (None, 784)                       3,136
 (BatchNormalization)

 dense_56 (Dense)                  (None, 256)                     200,960

 batch_normalization_24            (None, 256)                       1,024
 (BatchNormalization)

 dropout_9 (Dropout)               (None, 256)                           0

 dense_57 (Dense)                  (None, 128)                      32,896

 dense_58 (Dense)                  (None, 24)                        3,096


 Total params: 241,112 (941.84 KB)

 Trainable params: 239,032 (933.72 KB)

 Non-trainable params: 2,080 (8.12 KB)
```

7.2 Task 2.2: Train your network and visualize the training history

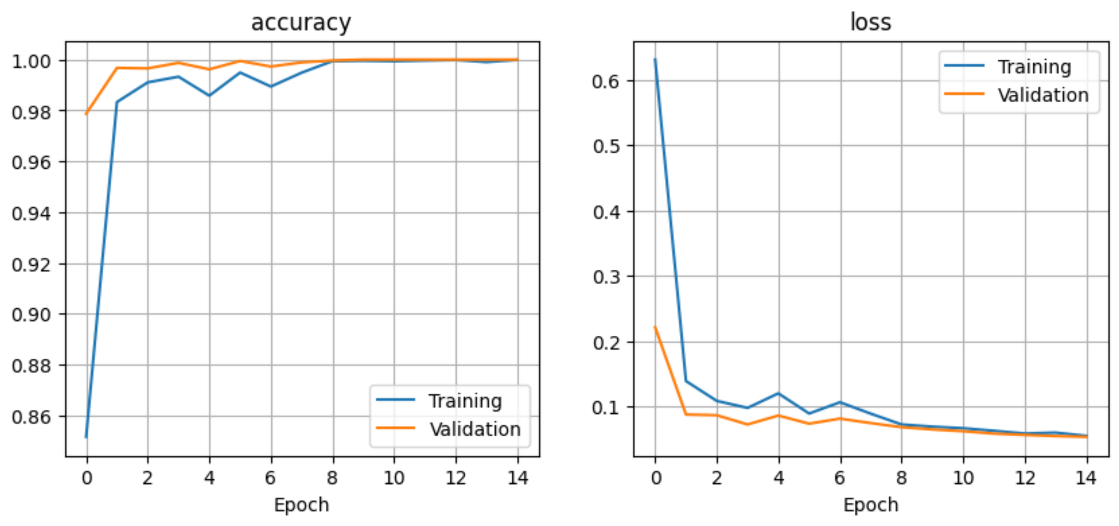
Train the model and plot the training history in the code cell(s) below. Feel free to use the function `plot_training_history()` and the custom Keras callback `F1ScoreCallback()` from Supporting code section.

[112]:
```
# Prepare integer labels for F1ScoreCallback
y_train_int = np.argmax(y_train, axis=1)
y_val_int = np.argmax(y_val, axis=1)


# Create the callback
f1_callback = F1ScoreCallback(X_train, y_train_int, X_val, y_val_int)
callbacks = [
    f1_callback,
    EarlyStopping(monitor="val_f1", mode="max", patience=5,␣
  ↪restore_best_weights=True),
    ReduceLROnPlateau(monitor="val_f1", mode="max", factor=0.5, patience=2,␣
  ↪min_lr=1e-5, verbose=1),
]


# Record start time
start_time = time.time()


# Train the model
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
    batch_size=64,
    callbacks=callbacks,
    verbose=1
)

# Record end time
end_time = time.time()
training_time = end_time - start_time

# Visualize training history
plot_training_history(history, list_of_metrics=['accuracy', 'loss'])

print(f"Training completed in {training_time:.2f} seconds")
```

```
Epoch 1/50
341/344             0s 3ms/step -
accuracy: 0.6629 - loss: 1.2877 - f1: 0.9824 - val_f1: 0.9788
```

```
344/344             3s 6ms/step -
accuracy: 0.8514 - loss: 0.6311 - val_accuracy: 0.9787 - val_loss: 0.2210 - f1:
0.9824 - val_f1: 0.9788 - learning_rate: 0.0010
Epoch 2/50
328/344             0s 2ms/step -
accuracy: 0.9775 - loss: 0.1685 - f1: 0.9955 - val_f1: 0.9968
344/344             2s 5ms/step -
accuracy: 0.9833 - loss: 0.1386 - val_accuracy: 0.9967 - val_loss: 0.0874 - f1:
0.9955 - val_f1: 0.9968 - learning_rate: 0.0010
Epoch 3/50
340/344             0s 2ms/step -
accuracy: 0.9900 - loss: 0.1128 - f1: 0.9982 - val_f1: 0.9965
344/344             2s 5ms/step -
accuracy: 0.9910 - loss: 0.1083 - val_accuracy: 0.9965 - val_loss: 0.0863 - f1:
0.9982 - val_f1: 0.9965 - learning_rate: 0.0010
Epoch 4/50
337/344             0s 2ms/step -
accuracy: 0.9874 - loss: 0.1189 - f1: 0.9989 - val_f1: 0.9987
344/344             2s 5ms/step -
accuracy: 0.9933 - loss: 0.0975 - val_accuracy: 0.9987 - val_loss: 0.0721 - f1:
0.9989 - val_f1: 0.9987 - learning_rate: 0.0010
Epoch 5/50
336/344             0s 2ms/step -
accuracy: 0.9855 - loss: 0.1214 - f1: 0.9961 - val_f1: 0.9962
344/344             2s 5ms/step -
accuracy: 0.9858 - loss: 0.1198 - val_accuracy: 0.9962 - val_loss: 0.0858 - f1:
0.9961 - val_f1: 0.9962 - learning_rate: 0.0010
Epoch 6/50
321/344             0s 2ms/step -
accuracy: 0.9929 - loss: 0.0943 - f1: 0.9994 - val_f1: 0.9994
344/344             2s 5ms/step -
accuracy: 0.9949 - loss: 0.0889 - val_accuracy: 0.9995 - val_loss: 0.0733 - f1:
0.9994 - val_f1: 0.9994 - learning_rate: 0.0010
Epoch 7/50
322/344             0s 2ms/step -
accuracy: 0.9908 - loss: 0.1013 - f1: 0.9980 - val_f1: 0.9973
344/344             2s 5ms/step -
accuracy: 0.9894 - loss: 0.1062 - val_accuracy: 0.9973 - val_loss: 0.0811 - f1:
0.9980 - val_f1: 0.9973 - learning_rate: 0.0010
Epoch 8/50
337/344             0s 2ms/step -
accuracy: 0.9932 - loss: 0.0923 - f1: 0.9996 - val_f1: 0.9989

Epoch 8: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
344/344             2s 5ms/step -
accuracy: 0.9949 - loss: 0.0887 - val_accuracy: 0.9989 - val_loss: 0.0742 - f1:
0.9996 - val_f1: 0.9989 - learning_rate: 0.0010
Epoch 9/50
```

```
342/344            0s 2ms/step -
accuracy: 0.9989 - loss: 0.0743 - f1: 1.0000 - val_f1: 0.9997
344/344            2s 5ms/step -
accuracy: 0.9994 - loss: 0.0722 - val_accuracy: 0.9996 - val_loss: 0.0680 - f1:
1.0000 - val_f1: 0.9997 - learning_rate: 5.0000e-04
Epoch 10/50
322/344            0s 2ms/step -
accuracy: 0.9990 - loss: 0.0718 - f1: 1.0000 - val_f1: 1.0000
344/344            2s 5ms/step -
accuracy: 0.9995 - loss: 0.0688 - val_accuracy: 1.0000 - val_loss: 0.0645 - f1:
1.0000 - val_f1: 1.0000 - learning_rate: 5.0000e-04
Epoch 11/50
344/344            0s 2ms/step -
accuracy: 0.9995 - loss: 0.0667 - f1: 1.0000 - val_f1: 1.0000
344/344            2s 5ms/step -
accuracy: 0.9994 - loss: 0.0664 - val_accuracy: 1.0000 - val_loss: 0.0618 - f1:
1.0000 - val_f1: 1.0000 - learning_rate: 5.0000e-04
Epoch 12/50
340/344            0s 2ms/step -
accuracy: 0.9995 - loss: 0.0644 - f1: 1.0000 - val_f1: 1.0000

Epoch 12: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
344/344            2s 5ms/step -
accuracy: 0.9996 - loss: 0.0625 - val_accuracy: 1.0000 - val_loss: 0.0582 - f1:
1.0000 - val_f1: 1.0000 - learning_rate: 5.0000e-04
Epoch 13/50
341/344            0s 2ms/step -
accuracy: 1.0000 - loss: 0.0587 - f1: 1.0000 - val_f1: 1.0000
344/344            2s 5ms/step -
accuracy: 0.9999 - loss: 0.0583 - val_accuracy: 1.0000 - val_loss: 0.0561 - f1:
1.0000 - val_f1: 1.0000 - learning_rate: 2.5000e-04
Epoch 14/50
325/344            0s 3ms/step -
accuracy: 0.9976 - loss: 0.0652 - f1: 1.0000 - val_f1: 1.0000

Epoch 14: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
344/344            2s 5ms/step -
accuracy: 0.9990 - loss: 0.0596 - val_accuracy: 1.0000 - val_loss: 0.0544 - f1:
1.0000 - val_f1: 1.0000 - learning_rate: 2.5000e-04
Epoch 15/50
329/344            0s 2ms/step -
accuracy: 1.0000 - loss: 0.0549 - f1: 1.0000 - val_f1: 1.0000
344/344            2s 5ms/step -
accuracy: 0.9999 - loss: 0.0547 - val_accuracy: 1.0000 - val_loss: 0.0532 - f1:
1.0000 - val_f1: 1.0000 - learning_rate: 1.2500e-04
```



```
Training completed in 28.34 seconds
```

[113]:
```
# Predict on the test set
# Loss og accuracy
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test loss: {test_loss:.4f}")
print(f"Test accuracy: {test_acc:.4f}")

# F1-score
y_pred_probs = model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test, axis=1)

f1 = f1_score(y_true, y_pred, average='macro')
print(f"Test F1-score: {f1:.4f}")
```

```
Test loss: 0.8845
Test accuracy: 0.8300
225/225              0s 741us/step
Test F1-score: 0.8108
```

7.3 Task 2.3: Discuss the results

Question 2.3.1: What could happen if the model is too shallow or too deep?

Answer: When the model is too shallow, it might fail to capture the complexity of the data. This can lead to underfitting, where the model will perfrom poorly on both the training and test data. If the model is too deep, gradients can become very small during backpropagation, making it especially difficult for early layers to learn. It is also prone to overfitting.

Question 2.3.2: How does the choice of train/validation/test split ratio affect the training process and the final performance of a deep learning model?

Answer: You need to have a good balance in the train/validation/test split ratio. If the training set is too small, it might fail to learn the underlying patterns, and therefore underfit. A large training set is generally better. The validation set is used to tune hyperparameters and monitor overfitting. If it's too small, the performance estimate may be unreliable. If it's too large, you're maybe "wasting" data that could've been used for training. You also need a big enough test set to get a trustworthy estimate of the final performance of the model. Common splits might be 80/10/10 or 70/15/15, but it depends on the size of the dataset. For small datasets, you should consider cross-validation.

Question 2.3.3: How do accuracy and F1-Score values compare (are they similar or very different from each other)? What does it tell you about the MNIST dataset and which one of these metrics is more reliable in this case?

Answer: Accuracy is simply the number of correctly classified samples divided by the total number of samples. The F1-score is the harmonic mean of precision and recall. The dataset is relatively balanced across classes, so we can expect similar results from the two evalutaion metrics. If there was class imbalances, the accuracy could be misleading because it could predict majority classes correctly while completely ignoring minority classes. If that was the case, F1-Score would be more reliable.

Question 2.3.4: Explain **very briefly** how each of the following model hyperparameters can impact the model's performance: - Number of layers - More layers = can learn complex patterns - Too many = overfitting or vanishing gradients - Too few = underfitting - Number of neurons in a layer - More neurons = captures complex patterns - Too many = memorizes training data, results in overfitting - Too few = underfitting - Batch size - Large batch = faster training, but might overfit - Small batch = slower, but often generalizes better - Optimizers - It determines how weights are updated. The choice affects convergence speed and final perfomance. - Regularization techniques (such as L2 regularization). - Penalize large weights to reduce overfitting. It imporves generalization to unseen data.

8 Task 3: Design and train a classical machine learning classifier

Pick your **favourite** machine learning classifer that you learned about in DAT200 and train it for the MNIST gesture recognition problem. (Hint: use the scikit-learn library). Remember to use **accuracy and the F1-score** as performance metrics.

[114]:
```python
# Initialize classifier
rf_clf = RandomForestClassifier(
    n_estimators=200,
    max_depth=15,
    random_state=42,
    n_jobs=-1
)

# Convert one-hot encoded labels to integers
y_train_int = y_train.argmax(axis=1)
y_val_int   = y_val.argmax(axis=1)

# Record start time
```

```
start_time = time.time()

# Train on training set
rf_clf.fit(X_train, y_train_int)

# Record end time
end_time = time.time()
training_time = end_time - start_time

# Evaluate on training set
y_train_pred = rf_clf.predict(X_train)
train_acc = accuracy_score(y_train_int, y_train_pred)
train_f1  = f1_score(y_train_int, y_train_pred, average='macro')

# Evaluate on validation set
y_val_pred = rf_clf.predict(X_val)
val_acc = accuracy_score(y_val_int, y_val_pred)
val_f1  = f1_score(y_val_int, y_val_pred, average='macro')

print(f"Training completed in {training_time:.2f} seconds")
print(f"Training Accuracy: {train_acc:.4f}, Training F1-Score: {train_f1:.4f}")
print(f"Validation Accuracy: {val_acc:.4f}, Validation F1-Score: {val_f1:.4f}")
```

```
Training completed in 6.65 seconds
Training Accuracy: 1.0000, Training F1-Score: 1.0000
Validation Accuracy: 0.9967, Validation F1-Score: 0.9968
```

[115]:
```
# Convert one-hot encoded test labels to integers
y_test_int = y_test.argmax(axis=1)

# Predict on test set
y_test_pred = rf_clf.predict(X_test)

# Compute metrics
test_acc = accuracy_score(y_test_int, y_test_pred)
test_f1  = f1_score(y_test_int, y_test_pred, average='macro')

print(f"Random Forest Test Accuracy: {test_acc:.4f}")
print(f"Random Forest Test F1-Score: {test_f1:.4f}")
```

```
Random Forest Test Accuracy: 0.7974
Random Forest Test F1-Score: 0.7823
```

9 Task 4: Compare and discuss

Evaluate the ANN model you implemented in Task 2 against the classical machine learning model from Task 3, using the test dataset. Compare the two models based on:

- Accuracies and F1-scores they attain
- Time it takes to train them

Did you experience any trouble when training models in tasks 2 and 3?

Task 4 discussion Here:

The evaluation of both the ANN and the Random forest Classifier was done after training and validation. Both the accuracy and the F1-scores are very similiar, indicating similar performance, but with the ANN slightly edging out.

But the time it took to train the ANN was way more than for the Random Forest Classifier (about 30 seconds vs 6-7 seconds). This will of course depend on the batch size, number of epochs etc.