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 1: Milad Tootoonchi
Group member 2: Sania Minaipour
Group member 3: Makka Dulgaeva
Group: Group 26

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

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.

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.

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.

Supporting code

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

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score
import tensorflow as tf

"""
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 i

    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}")
```

Library imports

In [2]:
```
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import models, layers

from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, StratifiedKFold
```

```
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, f1_score, classification_report
```

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.

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.

In [3]:
```
# Sign language MNIST
train = pd.read_csv("sign_mnist_train.csv")
test  = pd.read_csv("sign_mnist_test.csv")

"""
This code splits the training and test datasets into input features (X) and labels
The 'label' column is extracted as the target variable (y), while the remaining col
are used as features (X). The data is converted into NumPy arrays, with float32 for
and int32 for y, to be efficiently used in machine learning models.
"""
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)
```

Task 1.1 Preprocessing

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

In [4]:
```
# Check missing values in features
print("Missing values for training set:", np.isnan(X_train).sum())
print("Missing values for testing set:", np.isnan(X_test).sum())

# Checking the shape of X_train
print("Shape:", X_train.shape)

# Checking number of classes
num_classes = len(np.unique(y_train))
print("Number of classes", num_classes)
```

```
Missing values for training set: 0
Missing values for testing set: 0
Shape: (27455, 784)
Number of classes 24
```

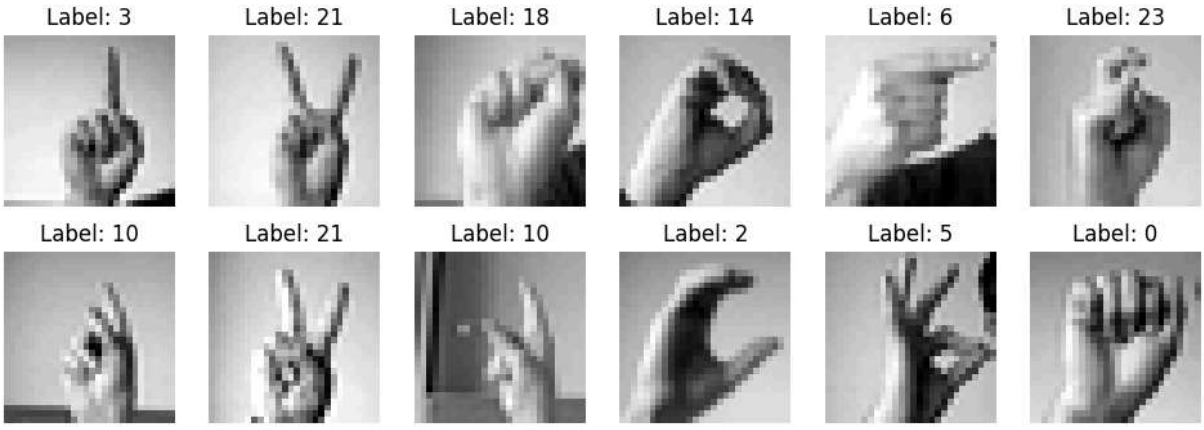
In [5]:
```
# Normalizing the data, pixel values are in range 0 - 255, therefore we will devide
# We will also flatten the image with reshape function.
X_train = X_train.reshape(-1, 28 * 28).astype('float32') / 255.0
X_test = X_test.reshape(-1, 28 * 28).astype('float32') / 255.0
```

In [6]:
```
# One-hot encoding with to_categorical function.
y_train_oh = to_categorical(y_train)
y_test_oh = to_categorical(y_test)
```

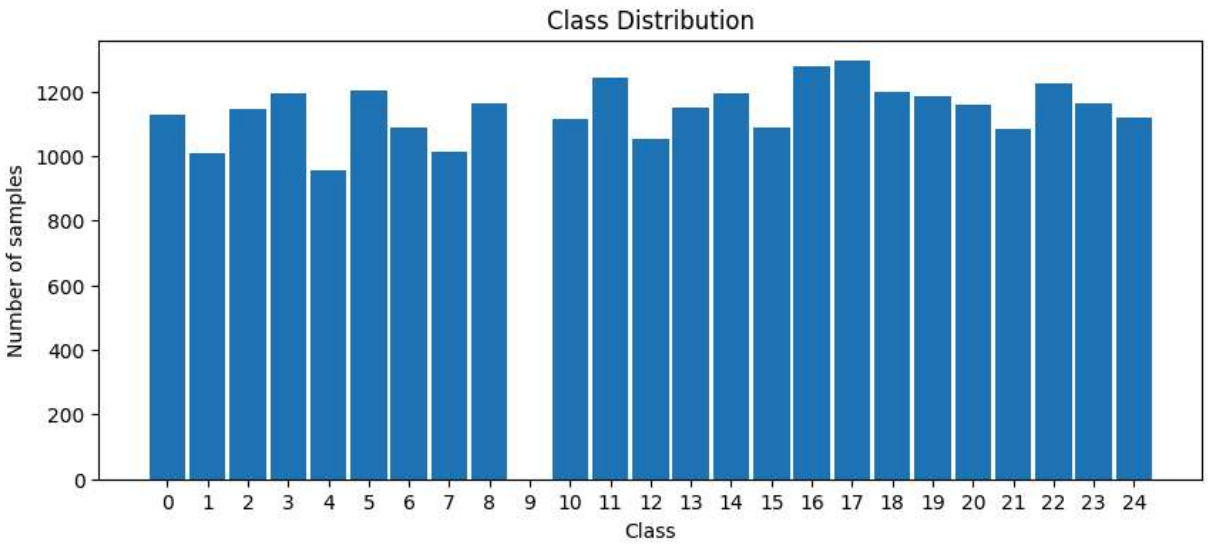
Task 1.2 Visualization

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

In [7]:
```
"""
This code randomly selects 12 images from the training set and displays them in a 2
reshaping flattened images and handling one-hot encoded labels if needed.
"""

# Let us look at 12 random images after preprocessing to check how it looks like wh
indices = np.random.choice(len(X_train), 12, replace = False)
plt.figure(figsize = (12, 4))

for i, idx in enumerate(indices):
    img = X_train[idx].reshape(28, 28)  # reshape if flattened
    label = y_train[idx].argmax() if y_train.ndim > 1 else y_train[idx]

    plt.subplot(2, 6, i + 1)
    plt.imshow(img, cmap = 'gray')
    plt.title(f"Label: {label}")
    plt.axis('off')

plt.show()
```



In [8]:
```
"""
This code checks if y_train is one-hot encoded (ndim > 1).
If so, it converts it to class indices using argmax; otherwise,
it uses the labels as they are.
Then it plots a histogram showing the distribution of samples per class in the trai
"""

# Checking if the dataset is balanced or not
if y_train.ndim > 1:
    labels = y_train.argmax()
else:
    labels = y_train

plt.figure(figsize = (10, 4))
plt.hist(labels, bins = np.arange(26) - 0.5, rwidth = 0.9)
plt.xticks(range(25))
plt.xlabel("Class")
plt.ylabel("Number of samples")
plt.title("Class Distribution")
plt.show()
```



In [9]:
```
"""
This code visualizes one example image for each class in the dataset.
For each class from 0 to 24, it finds the first occurrence in labels,
reshapes the corresponding X_train image to 28*28,
and plots it in a 5*5 grid with the class number as the title.
"""

# Visualizing every letter
plt.figure(figsize = (10, 10))

classes = 25  # 0 to 24
for i in range(classes):
    indices = np.where(labels == i)[0]
    if len(indices) == 0:
        continue  # skip missing class

    idx = indices[0]  # take the first example
```

```
    img = X_train[idx].reshape(28, 28)
    plt.subplot(5, 5, i + 1)
    plt.imshow(img, cmap = 'gray')
    plt.title(f"{i}")
    plt.axis('off')

plt.show()
```

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.

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)

In [10]:
```
"""
Defines a deep neural network with multiple dense layers using ReLU activations
and a final layer with tanh;
compiled with Adam optimizer, MSE loss, and accuracy metric.
"""

model = models.Sequential([
    layers.Dense(1000, activation = 'relu', input_dim=X_train.shape[1]),
    layers.Dense(800, activation = 'relu'),
    layers.Dense(600, activation = 'relu'),
    layers.Dense(num_classes + 1, activation = 'tanh')   # Including Z for num_clas
])

# Compiling model
model.compile(
    optimizer = 'adam',
    loss = 'mean_squared_error',
    metrics = ['accuracy']                    # We'll use F1 as callback
)
```

```
/home/milad/repositories/Machine-Learning-Deep-Learning/.venv/lib/python3.12/site-pa
ckages/keras/src/layers/core/dense.py:92: UserWarning: Do not pass an `input_shape`/
`input_dim` argument to a layer. When using Sequential models, prefer using an `Inpu
t(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2025-10-01 08:41:42.865327: E external/local_xla/xla/stream_executor/cuda/cuda_platf
orm.cc:51] failed call to cuInit: INTERNAL: CUDA error: Failed call to cuInit: UNKNO
WN ERROR (303)
```

Task 2.2: Train your network and visualize the training history

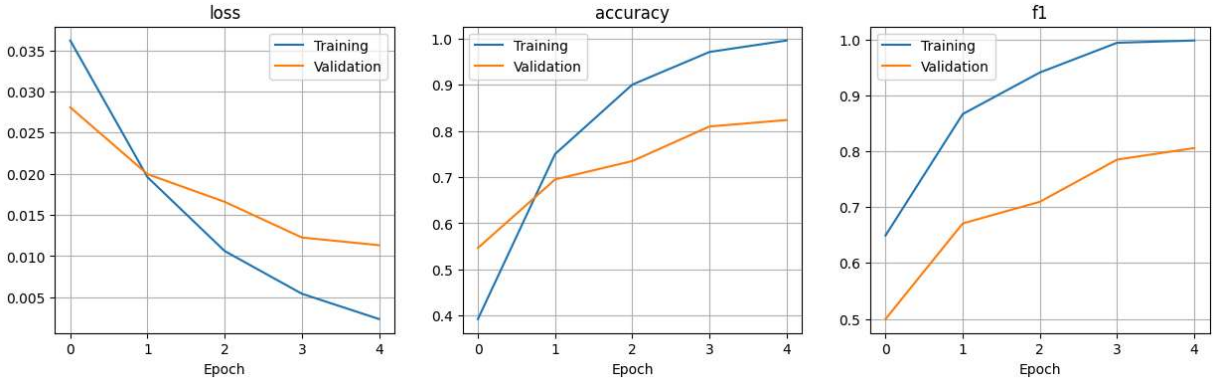
Train the model and plot the training history in the code cell(s) below. Feel free to use the function `plot_training_history()` and the custom Keras callback `F1ScoreCallback()` from Supporting code section.

In [11]:
```
"""
Trains the model on the one-hot encoded data for 25 epochs with batch size 128,
validating on test data and computing F1 scores after each epoch.
"""
```

```
history = model.fit(
    X_train,
    y_train_oh,
    validation_data = (X_test, y_test_oh),
    epochs = 5,
    batch_size = 100,
    callbacks = [F1ScoreCallback(X_train, y_train, X_test, y_test)]
)
```

```
Epoch 1/5
275/275 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.2317 - loss: 0.0525 – f1: 0.
6492 – val_f1: 0.4997
275/275 ━━━━━━━━━━━━━━━━━━━━ 10s 34ms/step - accuracy: 0.3926 - loss: 0.0362 - val_a
ccuracy: 0.5466 - val_loss: 0.0281 - f1: 0.6492 - val_f1: 0.4997
Epoch 2/5
274/275 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.6901 - loss: 0.0227 – f1: 0.
8663 – val_f1: 0.6707
275/275 ━━━━━━━━━━━━━━━━━━━━ 11s 40ms/step - accuracy: 0.7499 - loss: 0.0196 - val_a
ccuracy: 0.6949 - val_loss: 0.0200 - f1: 0.8663 - val_f1: 0.6707
Epoch 3/5
274/275 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.8689 - loss: 0.0126 – f1: 0.
9407 – val_f1: 0.7095
275/275 ━━━━━━━━━━━━━━━━━━━━ 9s 31ms/step - accuracy: 0.9002 - loss: 0.0106 - val_ac
curacy: 0.7347 - val_loss: 0.0166 - f1: 0.9407 - val_f1: 0.7095
Epoch 4/5
275/275 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.9604 - loss: 0.0063 – f1: 0.
9938 – val_f1: 0.7849
275/275 ━━━━━━━━━━━━━━━━━━━━ 9s 32ms/step - accuracy: 0.9708 - loss: 0.0054 - val_ac
curacy: 0.8095 - val_loss: 0.0123 - f1: 0.9938 - val_f1: 0.7849
Epoch 5/5
273/275 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9947 - loss: 0.0027 – f1: 0.
9978 – val_f1: 0.8055
275/275 ━━━━━━━━━━━━━━━━━━━━ 9s 33ms/step - accuracy: 0.9957 - loss: 0.0024 - val_ac
curacy: 0.8235 - val_loss: 0.0113 - f1: 0.9978 - val_f1: 0.8055
```

In [12]:
```
# Plotting history for loss, accuracy and f1.
plot_training_history(history, list_of_metrics = ['loss', 'accuracy', 'f1'])
```



Task 2.3: Discuss the results

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

If the model was too shallow it can not capture complex patterns and the model will underfit.
If the model was too deep it would perform well on training data, however would it would peform poorly on unseen data and overfit.

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?

More training data usualy improved learning and reduces underfitting, and too little data can make the model underperform.
It can also make the model training take longer time.
Too little validation data size makes the feedback unrelieable.
Too little test data makes the more variance in performence estimate.

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?

The accuracy value curve are more smoother on the training set than the f1, however f1 and accuracy are more similar on the validation set. The f1 training curve also has similar shape with its validation curve. The difference between f1 and accuracy tells us that the classes in the MNIST dataset is unbalanced and therefore the f1 is more reliale.

Question 2.3.4: Explain **very briefly** how each of the following model hyperparameters can impact the model's performance:

- Number of layers - Many layers can capture complex patterns, but too many can cause overfitting and slow training.
- Number of neurons in a layer - More neurons increase capacity to learn, but too many can overfit.
- Batch size - Large batches makes stable gradients; smaller batches makes noise but may help generalization.
- Optimizers - Different optimizers can affect convergence speed and stability.
- Regularization techniques (such as L2 regularization) - Prevents overfitting by penelize large weights.

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.

Modelling

In [13]:
```python
clf = SVC(random_state = 42)

clf.fit(X_train, y_train)

# Predictions
y_pred = clf.predict(X_test)

# Evaluation
acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average = "weighted")

print(f"Accuracy: {acc:.4f}")
print(f"F1 Score: {f1:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.8417
F1 Score: 0.8442

Classification Report:
              precision    recall  f1-score   support

           0       0.94      1.00      0.97       331
           1       1.00      0.99      0.99       432
           2       0.88      0.99      0.93       310
           3       0.91      1.00      0.95       245
           4       0.94      0.99      0.97       498
           5       0.78      0.83      0.80       247
           6       0.93      0.93      0.93       348
           7       0.98      0.94      0.96       436
           8       0.80      0.90      0.85       288
          10       0.79      0.59      0.67       331
          11       0.87      1.00      0.93       209
          12       0.85      0.74      0.79       394
          13       0.90      0.68      0.78       291
          14       0.99      0.83      0.90       246
          15       1.00      1.00      1.00       347
          16       1.00      0.99      0.99       164
          17       0.28      0.54      0.37       144
          18       0.71      0.78      0.74       246
          19       0.80      0.70      0.75       248
          20       0.57      0.63      0.60       266
          21       0.81      0.59      0.68       346
          22       0.57      0.76      0.65       206
          23       0.82      0.76      0.79       267
          24       0.85      0.76      0.80       332

    accuracy                           0.84      7172
   macro avg       0.83      0.83      0.83      7172
weighted avg       0.86      0.84      0.84      7172
```

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?

In [14]:
```
print(f"Accuracy ANN: {history.history["val_accuracy"][-1]}")
print(f"F1 Score ANN: {history.history["val_f1"][-1]}")
print(f"Accuracy CSV: {acc:.4f}")
print(f"F1 Score CSV: {f1:.4f}")
```

```
Accuracy ANN: 0.8234801888465881
F1 Score ANN: 0.805482539144292
Accuracy CSV: 0.8417
F1 Score CSV: 0.8442
```

Task 4 discussion Here:

The ANN got worse accuracy and f1 score than the SVC. in addition, The ANN is overfitting alot more than the SVC. The ANN used 48 seconds on training, while the SVC used 1 minute and 26 seconds. The ANN trained in around half the time of the SVC, however SVC had better accuracy making it the more efficient model. When we trained the ANN we were unsure how to tune the hyperparameters, but it got easier to descide when we tested different values. Another problem we had was to know if we should use one-hot-encoded y_labels or not. We found out that only the training ANN training and validation needed the one-hot-encoding.