```
# Imports from the entire program
import pandas as pd
import statistics
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import tensorflow as tf

from sklearn.preprocessing import OneHotEncoder # One-hot encoding
from sklearn.model_selection import train_test_split
from tensorflow import keras
from keras.models import Sequential
from keras.layers.core import Dense
from sklearn.model_selection import KFold
from sklearn import metrics
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.metrics import plot_confusion_matrix
```

Read data from CSV

This section aims to read all the necessary data from the CSV file. As we can see it contains two columns for each of the points generated by the posenet model. For instance, x0 and y0 correspond to the first point. Additionally, the last column corresponds to the label of the point.

```
data = pd.read_csv('Data.csv')
data
```

	x0	y0	x1	y1	x2	y2	x3	y3	x4	y5	
0	360.013705	159.627099	354.746301	153.009386	367.157390	152.534758	347.461785	156.530904	376.158980	156.660882	337.79
1	361.581720	161.648523	356.569929	154.260259	368.360679	153.739996	349.923240	155.402228	376.936939	157.994073	339.56
2	363.174721	163.991466	356.680321	156.701219	369.022974	156.285017	349.894964	159.454501	377.938579	159.107582	342.25
3	362.761304	165.765602	356.488025	158.970721	369.087571	158.905336	349.773206	162.725960	377.430370	161.279480	341.12
4	365.350581	166.813330	358.592403	160.345269	371.793395	159.945634	353.128193	164.370132	379.658182	162.992144	346.70
...	...	...	...	...	...	...	...	...	...	...	
2407	334.327034	125.463753	325.140217	118.473388	341.558533	116.625971	316.243351	124.584382	351.980481	124.822275	311.90
2408	331.771731	110.809423	321.946385	101.143517	339.960870	100.040202	314.941979	107.790238	349.942985	109.579380	308.97
2409	332.634275	102.314806	324.506045	92.641135	341.494119	93.239140	316.059739	97.963268	351.644976	101.509621	311.29

Data preparation

This section aims to prepare data to be fed into the neural network

One Hot Encoding

It represents each categorical variable with a binary vector that has one element for each unique label and marking the class label with a 1 and all other elements 0.

```
def one_hot_encoding_targets(y_train, y_test):
  # Creating one hot encoder object
  onehotencoder = OneHotEncoder()
  # Reshape the 1-D country array to 2-D as fit_transform expects 2-D and finally fit the object
  y_train_enc = onehotencoder.fit_transform(y_train.values.reshape(-1,1)).toarray()
  y_test_enc = onehotencoder.fit_transform(y_test.values.reshape(-1,1)).toarray()

  return y_train_enc, y_test_enc
```

```
    return y_train_ohe, y_test_ohe
```

Normalization and shuffling

Normalization is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to use a common scale, without distorting differences in the ranges of values or losing information

Shuffling data serves the purpose of reducing variance and making sure that models remain general and overfit less by removing bias.

```
# Max and min values are calculated to be used at javascript's normalization
maximun = data.drop(["label"], axis = 1).to_numpy().max()
minimun = data.drop(["label"], axis = 1).to_numpy().min()
print("Max: " + str(maximun))
print("Min: " + str(minimun))
```

```
    Max: 637.5225709626529
    Min: 1.3238226404056377
```

```
# Split into x and y
x_not_normalized = data.drop(["label"], axis = 1)
y_not_discretized = data["label"]

# Normalize x
aux = 'xs.'
for column in x_not_normalized:
  for j in range(0, len(x_not_normalized[column])):
    x_not_normalized[column][j] = (x_not_normalized[column][j] - minimun)/(maximun - minimun)
x = x_not_normalized

# Split into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y_not_discretized, test_size=0.1, random_state=42)

antes = y_train
# Prepare output data
y_train, y_test = one_hot_encoding_targets(y_train, y_test)
```

```
# At this moment x_train, y_train, x_test, y_test are available for the NN
#x_train.shape


pd.set_option('max_rows', 99999)
print(antes)
```

```
    1320        left_hip
    1596     left_dorsal
    2377           lotus
    1359        left_hip
    792             tree
    1211       right_hip
    1604     left_dorsal
    361                y
    1970            tree
    952         triangle
    149         left_hip
    208         left_hip
    1091           lotus
    2057             sun
    642         mountain
    1378        left_hip
    332                y
    1488               y
    2020            tree
    937         triangle
    1721    right_dorsal
    163         left_hip
    438     right_dorsal
    1264       right_hip
    1815        mountain
    1658    right_dorsal
    544         mountain
    1725    right_dorsal
    869              sun
    49         right_hip
    67         right_hip
    1764        mountain
    48         right_hip
    598         mountain
```

```
1223        right_hip
1753     right_dorsal
808               sun
124         right_hip
1925             tree
564          mountain
829               sun
1368         left_hip
414      right_dorsal
2226         triangle
2247         triangle
321                 y
1980             tree
1404         left_hip
891               sun
1430                y
1061            lotus
694              tree
530          mountain
2109              sun
240          left_hip
70          right_hip
1125            lotus
1776         mountain
73          right hip
```

```
with np.printoptions(threshold=np.inf):
    print(y_train)
```

```
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```

```
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Neural network Model

This section aims to design and test different experiments feeding the data into several neural network models. So it can be determined **which architecture delivers the best results** for the data.

The **K-Fold Cross-Validation** method is an iterative process. It consists of randomly dividing the data into k groups of approximately equal size, k-1 groups are used to train the model and one of the groups is used as validation. This process is repeated k times using a different group as validation in each iteration. The process generates k error estimates, the average of which is used as the final estimate.

The experiments are going to be run using **10 folds** (i.e. k = 10).

```
# Store the results of the different experiments
experiments = []
```

```
'''This function creates the data structures necessaries for creating
the accuracy and loss graphs at the k-fold.'''
def create_graph_structures(history):
  aux_accuracy = []
  aux_val_accuracy = []
  aux_loss = []
  aux_val_loss = []

  for element in history.history['accuracy']:
    aux_accuracy.append(element)

  for element in history.history['val_accuracy']:
    aux_val_accuracy.append(element)

  for element in history.history['loss']:
    aux_loss.append(element)
```

```
    for element in history.history['val_loss']:
      aux_val_loss.append(element)

    return aux_accuracy, aux_val_accuracy, aux_loss, aux_val_loss

  '''This function calculates the mean for every epoch taking into account all the folds involved.  '''
  def calculate_means (graph_data_accuracy, graph_data_val_accuracy, graph_data_loss, graph_data_val_loss, number_epochs):
    mean_accuracy = []
    mean_val_accuracy = []
    mean_loss = []
    mean_val_loss = []

    # Name of the columns. Epochs
    name_columns = []
    for i in range(0, number_epochs):
      name_columns.append(str(i))

    df_accuracy = pd.DataFrame(np.array(graph_data_accuracy), columns=name_columns)
    df_val_accuracy = pd.DataFrame(np.array(graph_data_val_accuracy), columns=name_columns)
    df_loss = pd.DataFrame(np.array(graph_data_loss), columns=name_columns)
    df_val_loss = pd.DataFrame(np.array(graph_data_val_loss), columns=name_columns)

    for column in df_accuracy:
      mean_accuracy.append(df_accuracy[column].mean())

    for column in df_val_accuracy:
      mean_val_accuracy.append(df_val_accuracy[column].mean())

    for column in df_loss:
      mean_loss.append(df_loss[column].mean())

    for column in df_val_loss:
      mean_val_loss.append(df_val_loss[column].mean())

    return mean_accuracy, mean_val_accuracy, mean_loss, mean_val_loss
```

Experiment 1

Name: 001-Poses-34-10

Learning Rate: 0.3

Epochs: 30

Architecture: 34D-10D

```
LEARNING_RATE = 0.3
EPOCHS = 30
NUM_FOLDS = 10
VERBOSITY = False
NAME = '001-Poses-34-10'

# Define the K-fold Cross Validator
kfold = KFold(n_splits=NUM_FOLDS, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
result = []

# Define structures for storing results in the graphs
graph_data_accuracy = []
graph_data_val_accuracy = []
graph_data_loss = []
graph_data_val_loss = []

for cv_train, cv_validation in kfold.split(x_train, y_train):

  '''
  Neural network structure: 34 inputs and x neurons at the output layer for x classes
  The hidden layer uses a rectifier activation function which is a good practice
  Softmax function at the output layer is used for the multiclass
  '''
```

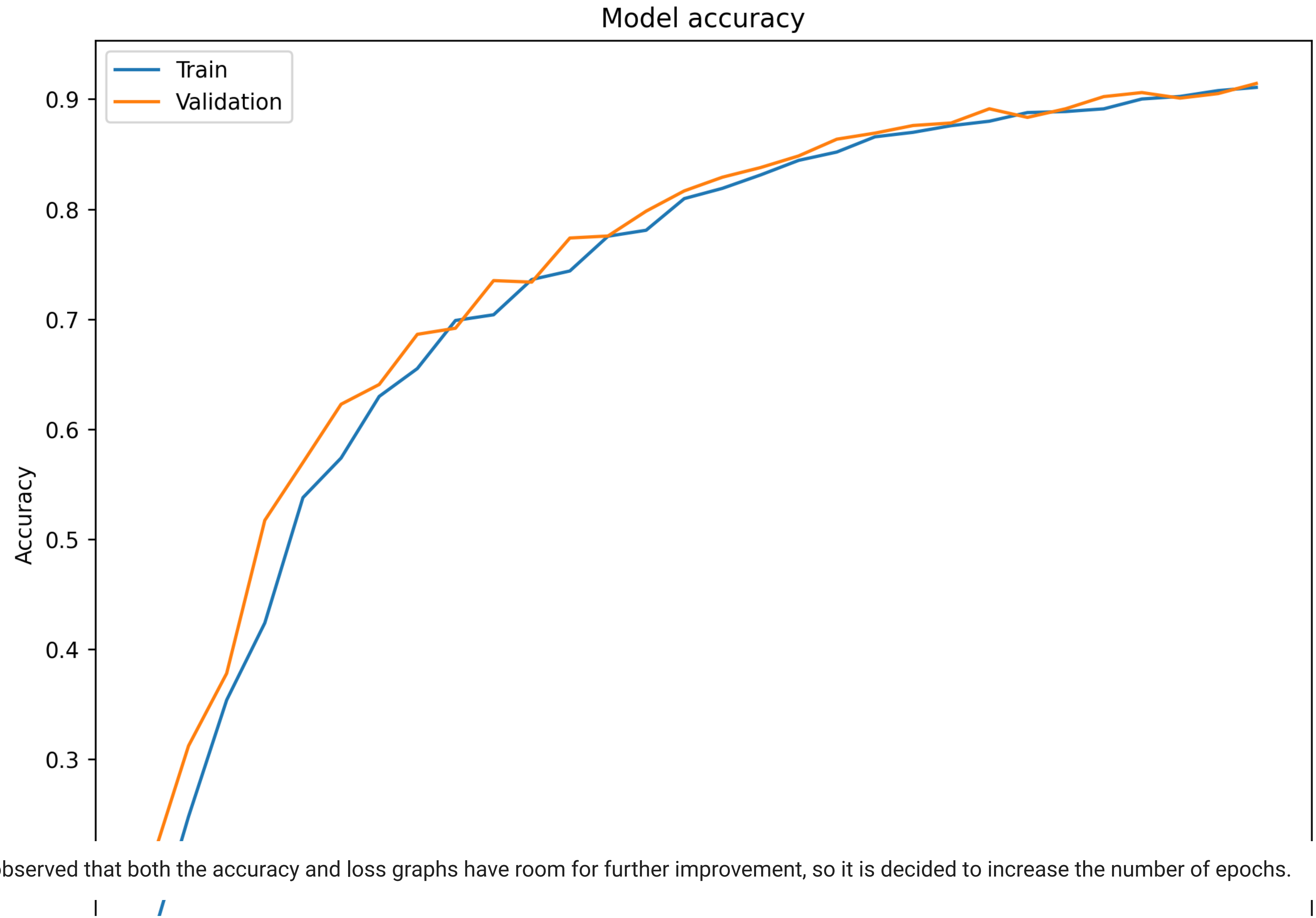
```
    # Neural network structure
    model = Sequential()
    model.add(Dense(34, input_dim=34, activation='relu'))
    model.add(Dense(10, activation='softmax'))

    '''
    Efficient Adam gradient descent optimization algorithm
    Logarithmic loss function, categorical_crossentropy
    '''
    # Model creation
    adam_optimizer = keras.optimizers.Adam(lr= LEARNING_RATE)
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Training
    history = model.fit(x_train.iloc[cv_train], y_train[cv_train], epochs=EPOCHS, validation_data=(x_train.iloc[cv_validat

    aux_accuracy, aux_val_accuracy, aux_loss, aux_val_loss = create_graph_structures(history)
    graph_data_accuracy.append(aux_accuracy)
    graph_data_val_accuracy.append(aux_val_accuracy)
    graph_data_loss.append(aux_loss)
    graph_data_val_loss.append(aux_val_loss)

    # Generate generalization metrics
    scores = model.evaluate(x_train.iloc[cv_validation], y_train[cv_validation], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}
    result.append(scores[1])

    # Fold number
    fold_no = fold_no + 1

  # Metrics for graphs
  mean_accuracy, mean_val_accuracy, mean_loss, mean_val_loss = calculate_means (graph_data_accuracy, graph_data_val_accura
```

```
# Print mean accuracy for the experiment
experiments.append(statistics.mean(result))
print(f'Accuracy K-Fold: {statistics.mean(result)*100}%')
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.6035377979278564; accuracy of 91.24423861503601%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.5681412220001221; accuracy of 84.3317985534668%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.46042510867118835; accuracy of 96.77419066429138%
------------------------------------------------------------------------
Training for fold 4 ...
Score for fold 4: loss of 0.5588498115539551; accuracy of 90.78341126441956%
------------------------------------------------------------------------
Training for fold 5 ...
Score for fold 5: loss of 0.6620463132858276; accuracy of 89.86175060272217%
------------------------------------------------------------------------
Training for fold 6 ...
Score for fold 6: loss of 0.49260202050209045; accuracy of 93.54838728904724%
------------------------------------------------------------------------
Training for fold 7 ...
Score for fold 7: loss of 0.5098462700843811; accuracy of 95.39170265197754%
------------------------------------------------------------------------
Training for fold 8 ...
Score for fold 8: loss of 0.5565317869186401; accuracy of 94.47004795074463%
------------------------------------------------------------------------
Training for fold 9 ...
Score for fold 9: loss of 0.5560810565948486; accuracy of 88.94008994102478%
------------------------------------------------------------------------
Training for fold 10 ...
Score for fold 10: loss of 0.5659332871437073; accuracy of 88.94008994102478%
Accuracy K-Fold: 91.42857074737549%
```

```
# Summarize history for accuracy
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_accuracy)
plt.plot(mean_val_accuracy)
```

```
plt.plot(mean_val_accuracy)
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.savefig(NAME + '_Accuracy.png')
plt.show()

# Summarize history for loss
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_loss)
plt.plot(mean_val_loss)
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.savefig(NAME + '_Loss.png')
plt.show()
```



It is observed that both the accuracy and loss graphs have room for further improvement, so it is decided to increase the number of epochs.

Experiment 2

Name: 002-Poses-34-10

Learning Rate: 0.3

Epochs: 100

Architecture: 34D-10D

Validation

```
LEARNING_RATE = 0.3
EPOCHS = 100
NUM_FOLDS = 10
VERBOSITY = False
NAME = '002-Poses-34-10'

# Define the K-fold Cross Validator
kfold = KFold(n_splits=NUM_FOLDS, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
result = []

# Define structures for storing results in the graphs
graph_data_accuracy = []
graph_data_val_accuracy = []
graph_data_loss = []
graph_data_val_loss = []

for cv_train, cv_validation in kfold.split(x_train, y_train):

  '''
  Neural network structure: 34 inputs and x neurons at the output layer for x classes
  The hidden layer uses a rectifier activation function which is a good practice
  Softmax function at the output layer is used for the multiclass
  '''
  # Neural network structure
  model = Sequential()
  model.add(Dense(34, input_dim=34, activation='relu'))
```

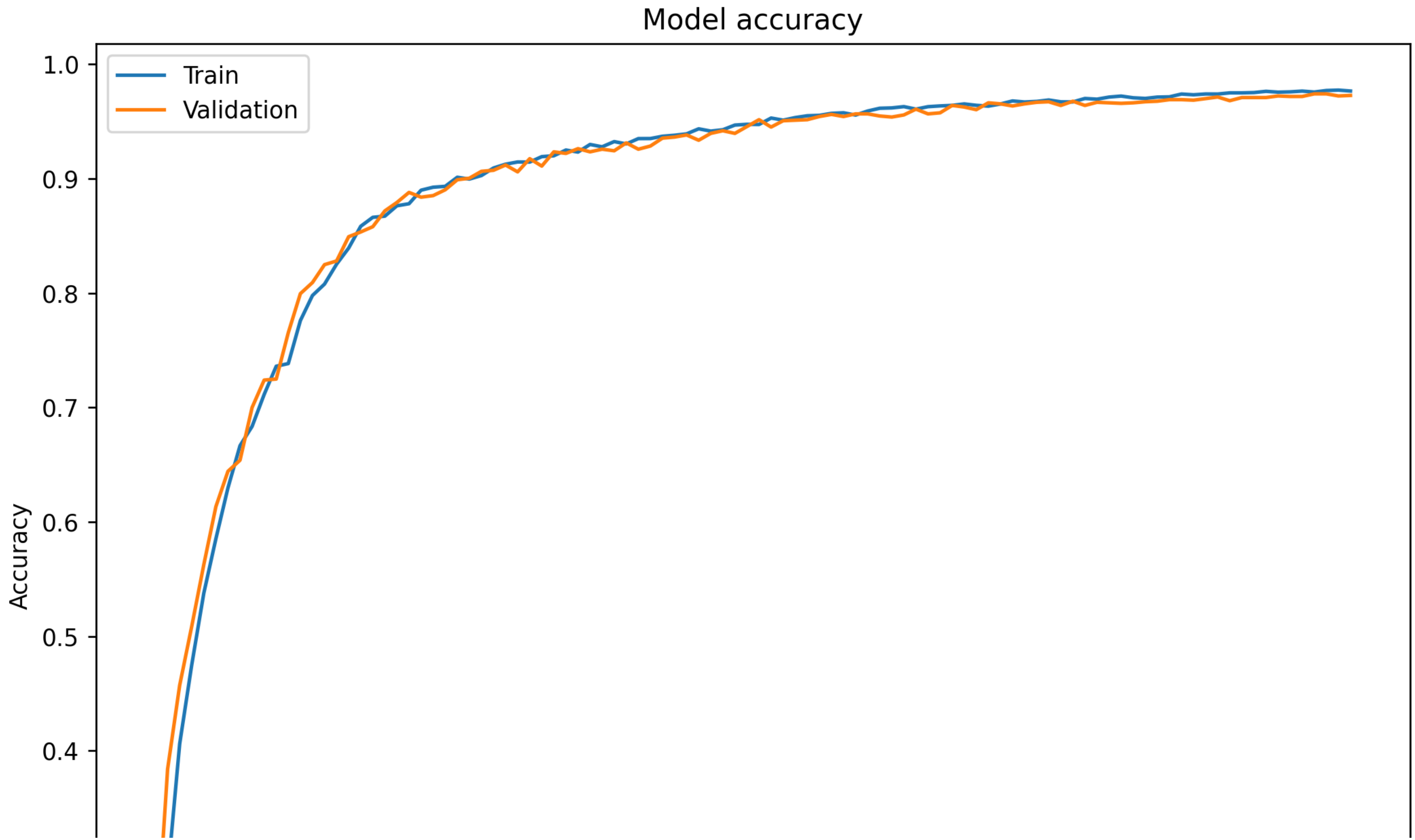
```
    model.add(Dense(10, activation='softmax'))

    '''
    Efficient Adam gradient descent optimization algorithm
    Logarithmic loss function, categorical_crossentropy
    '''
    # Model creation
    adam_optimizer = keras.optimizers.Adam(lr= LEARNING_RATE)
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Training
    history = model.fit(x_train.iloc[cv_train], y_train[cv_train], epochs=EPOCHS, validation_data=(x_train.iloc[cv_validat

    aux_accuracy, aux_val_accuracy, aux_loss, aux_val_loss = create_graph_structures(history)
    graph_data_accuracy.append(aux_accuracy)
    graph_data_val_accuracy.append(aux_val_accuracy)
    graph_data_loss.append(aux_loss)
    graph_data_val_loss.append(aux_val_loss)

    # Generate generalization metrics
    scores = model.evaluate(x_train.iloc[cv_validation], y_train[cv_validation], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}
    result.append(scores[1])

    # Fold number
    fold_no = fold_no + 1

  # Metrics for graphs
  mean_accuracy, mean_val_accuracy, mean_loss, mean_val_loss = calculate_means (graph_data_accuracy, graph_data_val_accura

  # Print mean accuracy for the experiment
  experiments.append(statistics.mean(result))
```

```
experiments.append(statistics.mean(result))
print(f'Accuracy K-Fold: {statistics.mean(result)*100}%')
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.16666236519813538; accuracy of 97.23502397537231%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.15920200943946838; accuracy of 97.69585132598877%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.12196152657270432; accuracy of 97.69585132598877%
------------------------------------------------------------------------
Training for fold 4 ...
Score for fold 4: loss of 0.14742060005664825; accuracy of 98.1566846370697%
------------------------------------------------------------------------
Training for fold 5 ...
Score for fold 5: loss of 0.15462125837802887; accuracy of 97.23502397537231%
------------------------------------------------------------------------
Training for fold 6 ...
Score for fold 6: loss of 0.2937469184398651; accuracy of 93.08755993843079%
------------------------------------------------------------------------
Training for fold 7 ...
Score for fold 7: loss of 0.19019442796707153; accuracy of 99.07833933830261%
------------------------------------------------------------------------
Training for fold 8 ...
Score for fold 8: loss of 0.16794750094413757; accuracy of 97.69585132598877%
------------------------------------------------------------------------
Training for fold 9 ...
Score for fold 9: loss of 0.1330963373184204; accuracy of 98.1566846370697%
------------------------------------------------------------------------
Training for fold 10 ...
Score for fold 10: loss of 0.1346488893032074; accuracy of 96.77419066429138%
Accuracy K-Fold: 97.28110611438751%
```

```
# Summarize history for accuracy
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_accuracy)
plt.plot(mean_val_accuracy)
plt.title('Model accuracy')
plt.ylabel('Accuracy')
```

```
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')
    plt.savefig(NAME + '_Accuracy.png')
    plt.show()

    # Summarize history for loss
    plt.figure(figsize=(10,8), dpi=250)
    plt.plot(mean_loss)
    plt.plot(mean_val_loss)
    plt.title('Model loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper right')
    plt.savefig(NAME + '_Loss.png')
    plt.show()
```



It can be seen that there is still room for improvement, so in the next test the number of layers in the network is increased to see if further improvement is possible.

Experiment 3

Name: 003-Poses-34-20-10

Learning Rate: 0.3

Epochs: 100

Architecture: 34D-20D-10D

```
LEARNING_RATE = 0.3
EPOCHS = 100
NUM_FOLDS = 10
VERBOSITY = False
NAME = '003-Poses-34-20-10'

# Define the K-fold Cross Validator
kfold = KFold(n_splits=NUM_FOLDS, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
result = []

# Define structures for storing results in the graphs
graph_data_accuracy = []
graph_data_val_accuracy = []
graph_data_loss = []
graph_data_val_loss = []

for cv_train, cv_validation in kfold.split(x_train, y_train):

  '''
  Neural network structure: 34 inputs and x neurons at the output layer for x classes
  The hidden layer uses a rectifier activation function which is a good practice
  Softmax function at the output layer is used for the multiclass
  '''
  # Neural network structure
  model = Sequential()
  model.add(Dense(34, input_dim=34, activation='relu'))
```

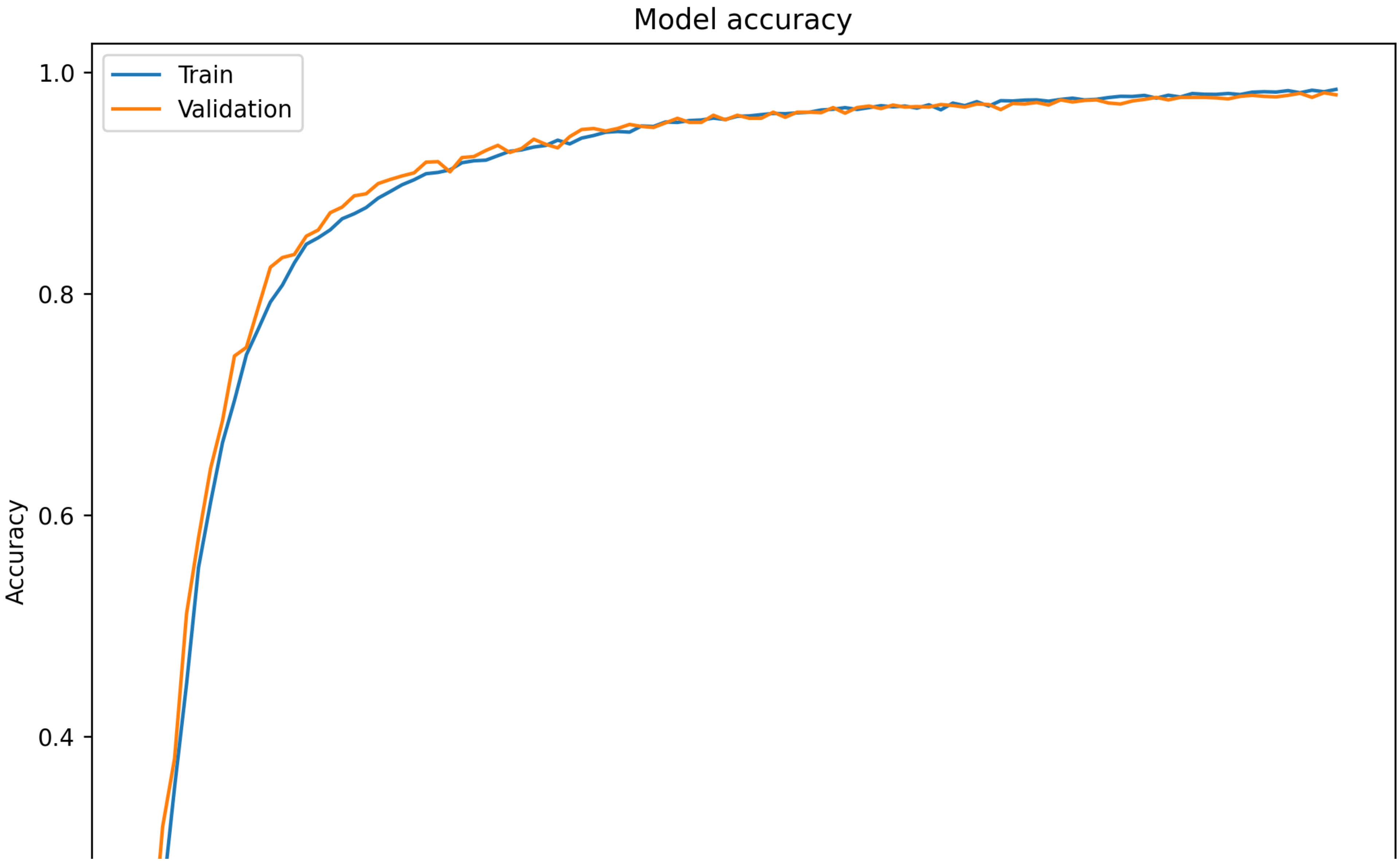
```
    model.add(Dense(20, activation='relu'))
    model.add(Dense(10, activation='softmax'))

    '''
    Efficient Adam gradient descent optimization algorithm
    Logarithmic loss function, categorical_crossentropy
    '''
    # Model creation
    adam_optimizer = keras.optimizers.Adam(lr= LEARNING_RATE)
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Training
    history = model.fit(x_train.iloc[cv_train], y_train[cv_train], epochs=EPOCHS, validation_data=(x_train.iloc[cv_validat

    aux_accuracy, aux_val_accuracy, aux_loss, aux_val_loss = create_graph_structures(history)
    graph_data_accuracy.append(aux_accuracy)
    graph_data_val_accuracy.append(aux_val_accuracy)
    graph_data_loss.append(aux_loss)
    graph_data_val_loss.append(aux_val_loss)

    # Generate generalization metrics
    scores = model.evaluate(x_train.iloc[cv_validation], y_train[cv_validation], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}
    result.append(scores[1])

    # Fold number
    fold_no = fold_no + 1

  # Metrics for graphs
  mean_accuracy, mean_val_accuracy, mean_loss, mean_val_loss = calculate_means (graph_data_accuracy, graph_data_val_accura

  # Print mean accuracy for the experiment
```

```
# Print mean accuracy for the experiment
experiments.append(statistics.mean(result))
print(f'Accuracy K-Fold: {statistics.mean(result)*100}%')
```

```
----------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.1991598904132843; accuracy of 96.77419066429138%
----------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.10918669402599335; accuracy of 97.23502397537231%
----------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.030995884910225868; accuracy of 99.53917264938354%
----------------------------------------------------------------------
Training for fold 4 ...
Score for fold 4: loss of 0.10683253407478333; accuracy of 98.61751198768616%
----------------------------------------------------------------------
Training for fold 5 ...
Score for fold 5: loss of 0.17747177183628082; accuracy of 99.07833933830261%
----------------------------------------------------------------------
Training for fold 6 ...
Score for fold 6: loss of 0.0678563341498375; accuracy of 98.61751198768616%
----------------------------------------------------------------------
Training for fold 7 ...
Score for fold 7: loss of 0.13027770817279816; accuracy of 96.77419066429138%
----------------------------------------------------------------------
Training for fold 8 ...
Score for fold 8: loss of 0.16956791281700134; accuracy of 96.77419066429138%
----------------------------------------------------------------------
Training for fold 9 ...
Score for fold 9: loss of 0.06684155762195587; accuracy of 98.61751198768616%
----------------------------------------------------------------------
Training for fold 10 ...
Score for fold 10: loss of 0.17308975756168365; accuracy of 97.69585132598877%
Accuracy K-Fold: 97.97234952449799%
```

```
# Summarize history for accuracy
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_accuracy)
plt.plot(mean_val_accuracy)
plt.title('Model accuracy')
```

```
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.savefig(NAME + '_Accuracy.png')
plt.show()

# Summarize history for loss
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_loss)
plt.plot(mean_val_loss)
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.savefig(NAME + '_Loss.png')
plt.show()
```



As there is still limited room for improvement of the parameters, the number of epochs is increased to 200 in order to check on the graphs which value is appropriate

Experiment 4

Name: 004-Poses-34-20-10

Learning Rate: 0.3

Epochs: 200

Architecture: 34D-20D-10D

```
LEARNING_RATE = 0.3
EPOCHS = 200
NUM_FOLDS = 10
VERBOSITY = False
NAME = '004-Poses-34-20-10'

# Define the K-fold Cross Validator
kfold = KFold(n_splits=NUM_FOLDS, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
result = []

# Define structures for storing results in the graphs
graph_data_accuracy = []
graph_data_val_accuracy = []
graph_data_loss = []
graph_data_val_loss = []

for cv_train, cv_validation in kfold.split(x_train, y_train):

  '''
  Neural network structure: 34 inputs and x neurons at the output layer for x classes
  The hidden layer uses a rectifier activation function which is a good practice
  Softmax function at the output layer is used for the multiclass
  '''
  # Neural network structure
  model = Sequential()
  model.add(Dense(34, input_dim=34, activation='relu'))
```

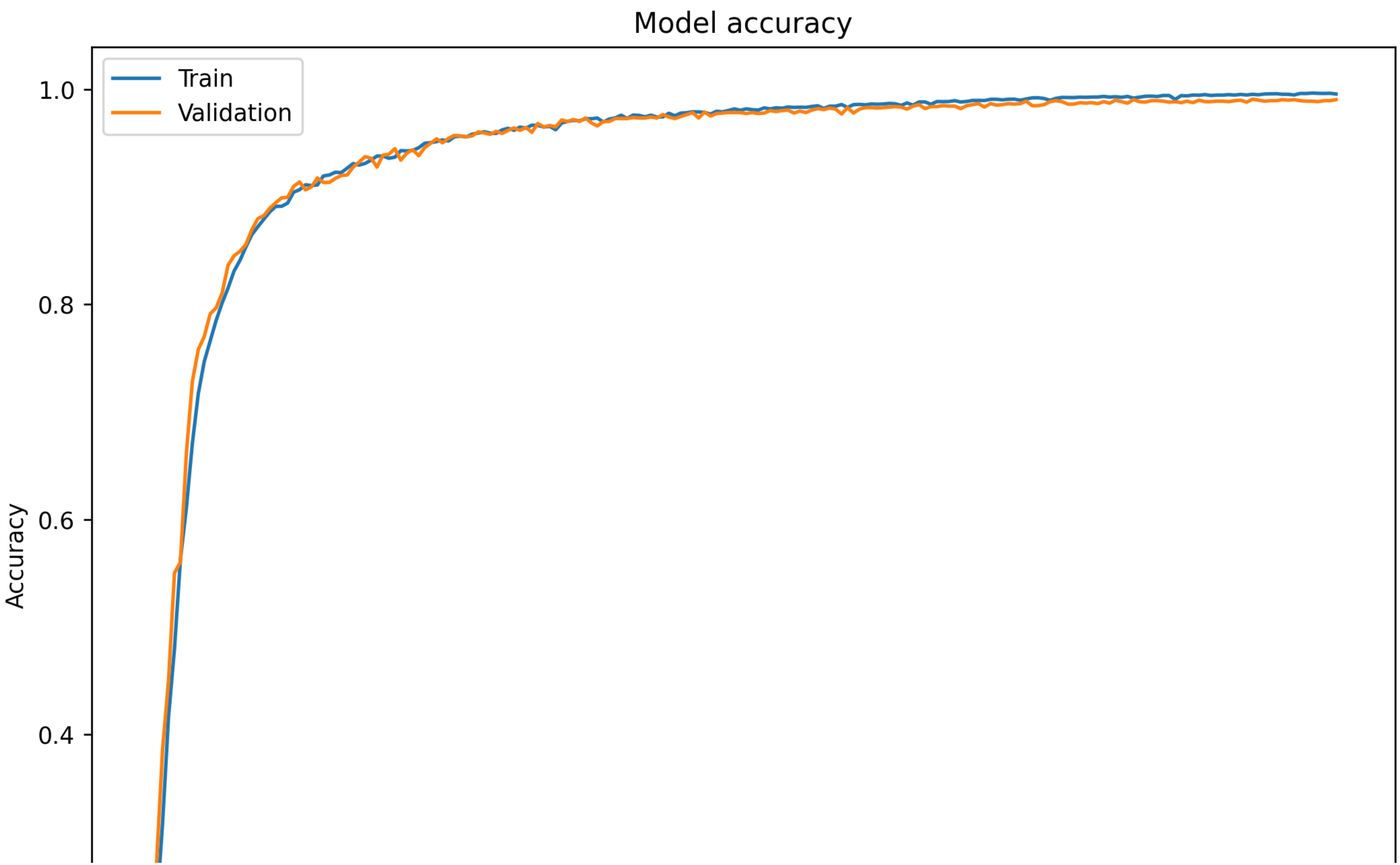
```
        model.add(Dense(20, activation='relu'))
        model.add(Dense(10, activation='softmax'))

        '''
        Efficient Adam gradient descent optimization algorithm
        Logarithmic loss function, categorical_crossentropy
        '''
        # Model creation
        adam_optimizer = keras.optimizers.Adam(lr= LEARNING_RATE)
        model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
                      metrics=['accuracy'])

        # Generate a print
        print('------------------------------------------------------------------------')
        print(f'Training for fold {fold_no} ...')

        # Training
        history = model.fit(x_train.iloc[cv_train], y_train[cv_train], epochs=EPOCHS, validation_data=(x_train.iloc[cv_validat

        aux_accuracy, aux_val_accuracy, aux_loss, aux_val_loss = create_graph_structures(history)
        graph_data_accuracy.append(aux_accuracy)
        graph_data_val_accuracy.append(aux_val_accuracy)
        graph_data_loss.append(aux_loss)
        graph_data_val_loss.append(aux_val_loss)

        # Generate generalization metrics
        scores = model.evaluate(x_train.iloc[cv_validation], y_train[cv_validation], verbose=0)
        print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}
        result.append(scores[1])

        # Fold number
        fold_no = fold_no + 1

    # Metrics for graphs
    mean_accuracy, mean_val_accuracy, mean_loss, mean_val_loss = calculate_means (graph_data_accuracy, graph_data_val_accura

    # Print mean accuracy for the experiment
```

```
# Print mean accuracy for the experiment
experiments.append(statistics.mean(result))
print(f'Accuracy K-Fold: {statistics.mean(result)*100}%')
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Score for fold 1: loss of 0.03626621142029762; accuracy of 98.1566846370697%
------------------------------------------------------------------------
Training for fold 2 ...
Score for fold 2: loss of 0.008383721113204956; accuracy of 100.0%
------------------------------------------------------------------------
Training for fold 3 ...
Score for fold 3: loss of 0.1472853124141693; accuracy of 98.61751198768616%
------------------------------------------------------------------------
Training for fold 4 ...
Score for fold 4: loss of 0.04190723970532417; accuracy of 99.53917264938354%
------------------------------------------------------------------------
Training for fold 5 ...
Score for fold 5: loss of 0.03778855875134468; accuracy of 98.61751198768616%
------------------------------------------------------------------------
Training for fold 6 ...
Score for fold 6: loss of 0.02293597534298968; accuracy of 99.53917264938354%
------------------------------------------------------------------------
Training for fold 7 ...
Score for fold 7: loss of 0.1405186504125595; accuracy of 99.07833933830261%
------------------------------------------------------------------------
Training for fold 8 ...
Score for fold 8: loss of 0.03196578845381737; accuracy of 99.07833933830261%
------------------------------------------------------------------------
Training for fold 9 ...
Score for fold 9: loss of 0.06489590555429459; accuracy of 99.07833933830261%
------------------------------------------------------------------------
Training for fold 10 ...
Score for fold 10: loss of 0.07511074095964432; accuracy of 98.61751198768616%
Accuracy K-Fold: 99.03225839138031%
```

```
# Summarize history for accuracy
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_accuracy)
plt.plot(mean_val_accuracy)
plt.title('Model accuracy')
```

```
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.savefig(NAME + '_Accuracy.png')
plt.show()

# Summarize history for loss
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_loss)
plt.plot(mean_val_loss)
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.savefig(NAME + '_Loss.png')
plt.show()
```



It is determined that from epoch 125 onwards there is no palpable improvement of the model and therefore it is set as the final number of epochs. The learning rate is not modified as it presents a compromise value between speed and results obtained.

Final model

Name: 005-Poses-34-20-10

Learning Rate: 0.3

Epochs: 125

Architecture: 34D-20D-10D

Validation

```
LEARNING_RATE = 0.3
EPOCHS = 125
NUM_FOLDS = 10
VERBOSITY = True
NAME = '005-Poses-34-20-10'

# Define the K-fold Cross Validator
kfold = KFold(n_splits=NUM_FOLDS, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
result = []

# Define structures for storing results in the graphs
graph_data_accuracy = []
graph_data_val_accuracy = []
graph_data_loss = []
graph_data_val_loss = []

for cv_train, cv_validation in kfold.split(x_train, y_train):

  '''
  Neural network structure: 34 inputs and x neurons at the output layer for x classes
  The hidden layer uses a rectifier activation function which is a good practice
  Softmax function at the output layer is used for the multiclass
  '''
  # Neural network structure
  model_final = Sequential()
  model_final.add(Dense(34, input_dim=34, activation='relu'))
```

```
    model_final.add(Dense(20, activation='relu'))
    model_final.add(Dense(10, activation='softmax'))

    '''
    Efficient Adam gradient descent optimization algorithm
    Logarithmic loss function, categorical_crossentropy
    '''
    # Model creation
    adam_optimizer = keras.optimizers.Adam(lr= LEARNING_RATE)
    model_final.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Training
    history = model_final.fit(x_train.iloc[cv_train], y_train[cv_train], epochs=EPOCHS, validation_data=(x_train.iloc[cv_v

    aux_accuracy, aux_val_accuracy, aux_loss, aux_val_loss = create_graph_structures(history)
    graph_data_accuracy.append(aux_accuracy)
    graph_data_val_accuracy.append(aux_val_accuracy)
    graph_data_loss.append(aux_loss)
    graph_data_val_loss.append(aux_val_loss)

    # Generate generalization metrics
    scores = model_final.evaluate(x_train.iloc[cv_validation], y_train[cv_validation], verbose=0)
    print(f'Score for fold {fold_no}: {model_final.metrics_names[0]} of {scores[0]}; {model_final.metrics_names[1]} of {sc
    result.append(scores[1])

    # Fold number
    fold_no = fold_no + 1

  # Metrics for graphs
  mean_accuracy, mean_val_accuracy, mean_loss, mean_val_loss = calculate_means (graph_data_accuracy, graph_data_val_accura

  # Print mean accuracy for the experiment
```

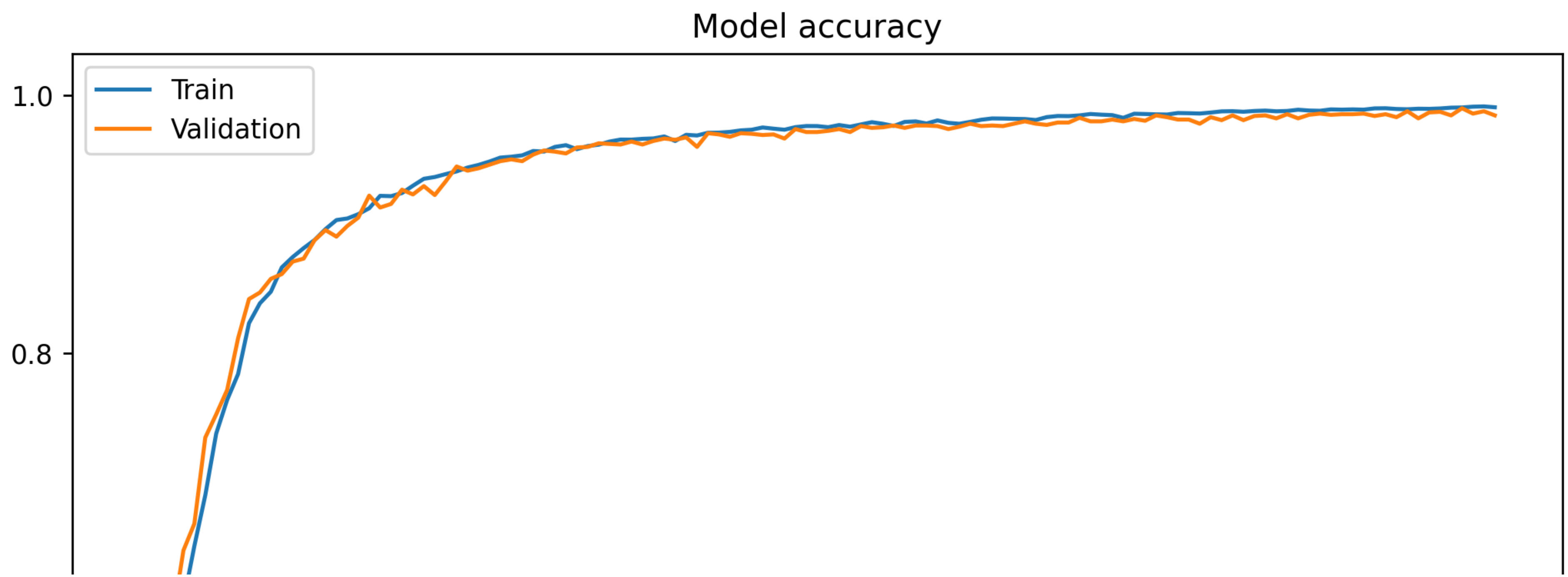
```
    # Print mean accuracy for the experiment
    experiments.append(statistics.mean(result))
    print(f'Accuracy K-Fold: {statistics.mean(result)*100}%')
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Epoch 1/125
62/62 [==============================] - 1s 6ms/step - loss: 2.2673 - accuracy: 0.1398 - val_loss: 2.1586 - val_acc
Epoch 2/125
62/62 [==============================] - 0s 2ms/step - loss: 2.1034 - accuracy: 0.4280 - val_loss: 2.0058 - val_acc
Epoch 3/125
62/62 [==============================] - 0s 2ms/step - loss: 1.9237 - accuracy: 0.5005 - val_loss: 1.8023 - val_acc
Epoch 4/125
62/62 [==============================] - 0s 2ms/step - loss: 1.7246 - accuracy: 0.5860 - val_loss: 1.5828 - val_acc
Epoch 5/125
62/62 [==============================] - 0s 2ms/step - loss: 1.5136 - accuracy: 0.6235 - val_loss: 1.3982 - val_acc
Epoch 6/125
62/62 [==============================] - 0s 2ms/step - loss: 1.3284 - accuracy: 0.7345 - val_loss: 1.2398 - val_acc
Epoch 7/125
62/62 [==============================] - 0s 2ms/step - loss: 1.1585 - accuracy: 0.7428 - val_loss: 1.1033 - val_acc
Epoch 8/125
62/62 [==============================] - 0s 2ms/step - loss: 1.0414 - accuracy: 0.7267 - val_loss: 1.0011 - val_acc
Epoch 9/125
62/62 [==============================] - 0s 2ms/step - loss: 0.9184 - accuracy: 0.7769 - val_loss: 0.9170 - val_acc
Epoch 10/125
62/62 [==============================] - 0s 2ms/step - loss: 0.8722 - accuracy: 0.7709 - val_loss: 0.8484 - val_acc
Epoch 11/125
62/62 [==============================] - 0s 2ms/step - loss: 0.7828 - accuracy: 0.8306 - val_loss: 0.7650 - val_acc
Epoch 12/125
62/62 [==============================] - 0s 2ms/step - loss: 0.7263 - accuracy: 0.8364 - val_loss: 0.7121 - val_acc
Epoch 13/125
62/62 [==============================] - 0s 2ms/step - loss: 0.6789 - accuracy: 0.8322 - val_loss: 0.6693 - val_acc
Epoch 14/125
62/62 [==============================] - 0s 2ms/step - loss: 0.6369 - accuracy: 0.8200 - val_loss: 0.6375 - val_acc
Epoch 15/125
62/62 [==============================] - 0s 2ms/step - loss: 0.5753 - accuracy: 0.8581 - val_loss: 0.6044 - val_acc
Epoch 16/125
62/62 [==============================] - 0s 2ms/step - loss: 0.5755 - accuracy: 0.8492 - val_loss: 0.5703 - val_acc
Epoch 17/125
62/62 [==============================] - 0s 2ms/step - loss: 0.5152 - accuracy: 0.8866 - val_loss: 0.5526 - val_acc
Epoch 18/125
62/62 [==============================] - 0s 2ms/step - loss: 0.5072 - accuracy: 0.8679 - val_loss: 0.5275 - val_acc
```

```
Epoch 19/125
62/62 [==============================] - 0s 2ms/step - loss: 0.4834 - accuracy: 0.8767 - val_loss: 0.4989 - val_acc
Epoch 20/125
62/62 [==============================] - 0s 2ms/step - loss: 0.4457 - accuracy: 0.8857 - val_loss: 0.4882 - val_acc
Epoch 21/125
62/62 [==============================] - 0s 2ms/step - loss: 0.4277 - accuracy: 0.8819 - val_loss: 0.4620 - val_acc
Epoch 22/125
62/62 [==============================] - 0s 2ms/step - loss: 0.4197 - accuracy: 0.8816 - val_loss: 0.4568 - val_acc
Epoch 23/125
62/62 [==============================] - 0s 2ms/step - loss: 0.4139 - accuracy: 0.8901 - val_loss: 0.4498 - val_acc
Epoch 24/125
62/62 [==============================] - 0s 2ms/step - loss: 0.3618 - accuracy: 0.9096 - val_loss: 0.4177 - val_acc
Epoch 25/125
62/62 [==============================] - 0s 2ms/step - loss: 0.3752 - accuracy: 0.8906 - val_loss: 0.4076 - val_acc
Epoch 26/125
62/62 [==============================] - 0s 2ms/step - loss: 0.4027 - accuracy: 0.8679 - val_loss: 0.4019 - val_acc
Epoch 27/125
62/62 [==============================] - 0s 2ms/step - loss: 0.3386 - accuracy: 0.9128 - val_loss: 0.3891 - val_acc
Epoch 28/125
62/62 [==============================] - 0s 2ms/step - loss: 0.3442 - accuracy: 0.9095 - val_loss: 0.3926 - val_acc
```

```
# Summarize history for accuracy
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_accuracy)
plt.plot(mean_val_accuracy)
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.savefig(NAME + '_Accuracy.png')
plt.show()

# Summarize history for loss
plt.figure(figsize=(10,8), dpi=250)
plt.plot(mean_loss)
plt.plot(mean_val_loss)
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
```

```
    plt.legend(['Train', 'Validation'], loc='upper right')
    plt.savefig(NAME + '_Loss.png')
    plt.show()
```

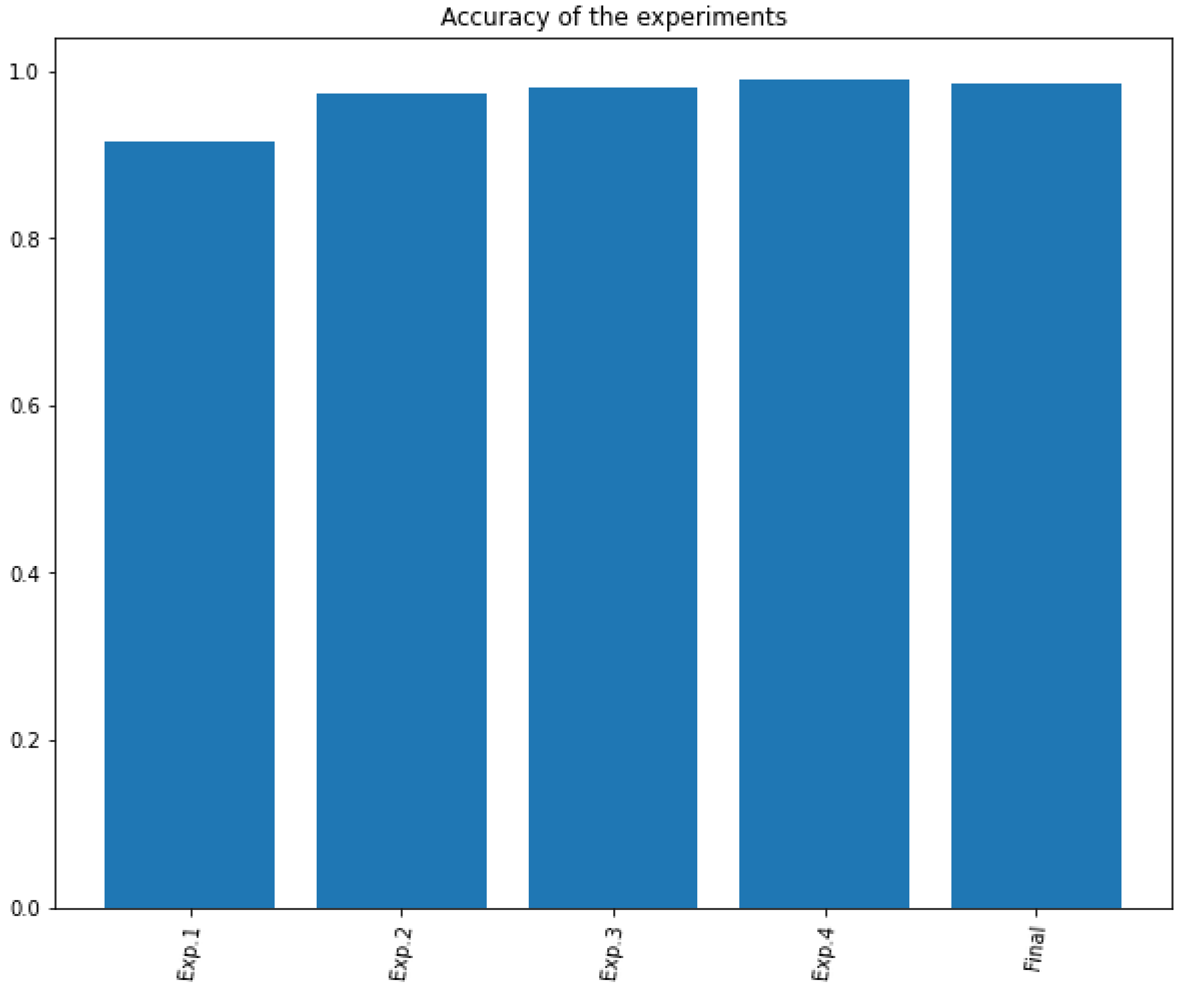


Evaluation of the experiments

Here, the results of the different experiments are going to be compared so that the best one can be selected.

Firstly, the main metric involved is the **accuracy** of the models, so let's analyse the results.

```
# A plot with all the accuracies is generated
langs = ['Exp.1', 'Exp.2', 'Exp.3', 'Exp.4', 'Final']

plt.figure(figsize=(10,8))
plt.bar(langs, experiments)
plt.title('Accuracy of the experiments')
plt.xticks(rotation='82.5')

plt.savefig('Accuracy_Bar_Plot.png',dpi=400)
plt.show()
```




```
# Returns the class corresponding to the predictions
def decide_class (result):
    maximum = np.max(result)
    index = np.where(result == maximum)
    for i in range (0, len(result)):
      if i == index[0][0]:
        result[i] = 1
      else:
        result[i] = 0
```

```
# Returns meaning of the result obatined
def decodeResult(result):
  maximum = np.max(result)
  index = np.where(result == maximum)
  if index[0][0] == 0:
    return "left_dorsal"
  elif index[0][0] == 1:
    return "left_hip"
  elif index[0][0] == 2:
    return "lotus"
  elif index[0][0] == 3:
    return "mountain"
  elif index[0][0] == 4:
    return "right_dorsal"
  elif index[0][0] == 5:
    return "right_hip"
  elif index[0][0] == 6:
    return "sun"
  elif index[0][0] == 7:
    return "tree"
  elif index[0][0] == 8:
    return "triangle"
  else:
    return "y"
```

```
model_final.summary()
```

```
Model: "sequential_49"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_127 (Dense)            (None, 34)                1190
_________________________________________________________________
dense_128 (Dense)            (None, 20)                700
_________________________________________________________________
dense_129 (Dense)            (None, 10)                210
=================================================================
```

```
Total params: 2,100
Trainable params: 2,100
Non-trainable params: 0
_________________________________________________________________
```

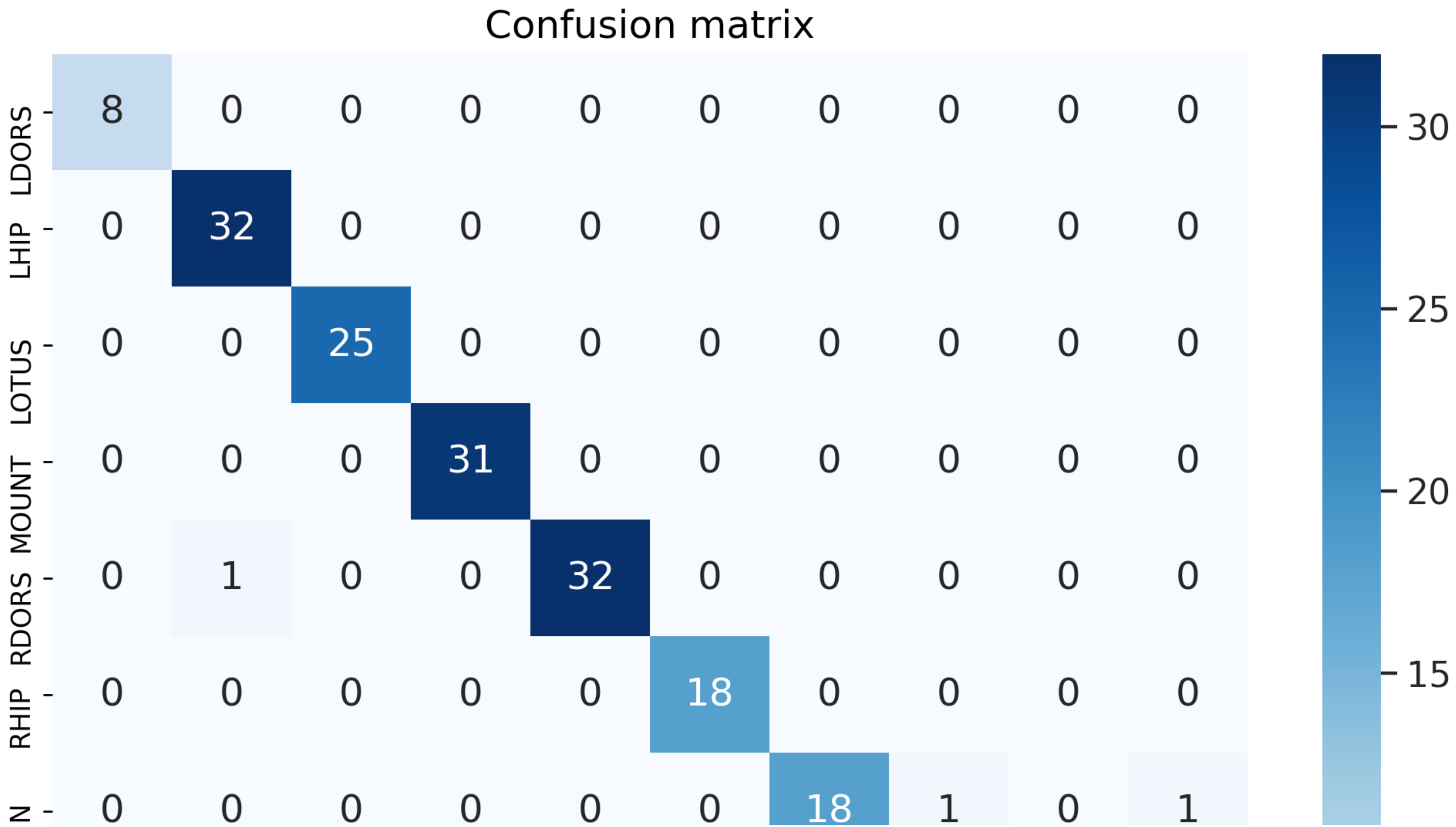
```
loss, accuracy = model_final.evaluate(x_test, y_test)
print('Accuracy: %.2f' % (accuracy*100))
print('Loss: %.2f' % (loss*100))
```

```
8/8 [==============================] - 0s 2ms/step - loss: 0.0679 - accuracy: 0.9835
Accuracy: 98.35
Loss: 6.79
```

```
# Predictions are generated
y_pred = model_final.predict(x_test)

# Transforms predictions into a single class
for element in y_pred:
  element = decide_class(element)

new_y_pred = []
# Decode predictions using neural network
for i in range(0, len(y_pred)):
  new_y_pred.append(decodeResult(y_pred[i]))

new_y_test = []
aux = np.array(y_test)
# Decode predictions of the test
for i in range(0, len(aux)):
  new_y_test.append(decodeResult(aux[i]))


# More significative metrics are calculated
confmat = metrics.confusion_matrix(new_y_test, new_y_pred)
accuracy = metrics.accuracy_score(new_y_test, new_y_pred)
precision = metrics.precision_score(new_y_test, new_y_pred, average='micro')
recall = metrics.recall_score(new_y_test, new_y_pred, average='micro')
f1 = metrics.f1_score(new_y_test, new_y_pred, average='micro')
```

```
f1 = metrics.f1_score(new_y_test, new_y_pred, average="micro")

print("Evaluation metrics for neural network model:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"f1: {f1:.4f}")

# Confusion matrix is displayed
plt.figure(figsize=(10,8), dpi=250)
ax = plt.subplot()
sns.set(font_scale=1.2)
sns.heatmap(confmat, annot=True, ax=ax, cmap="Blues", fmt="g");
ax.tick_params(axis='both', which='major', labelsize=10)
ax.xaxis.set_ticklabels(['LDORS', 'LHIP', 'LOTUS', 'MOUNT', 'RDORS', 'RHIP', 'SUN', 'TREE', 'TRIANGLE', 'Y']);
ax.yaxis.set_ticklabels(['LDORS', 'LHIP', 'LOTUS', 'MOUNT', 'RDORS', 'RHIP', 'SUN', 'TREE', 'TRIANGLE', 'Y']);
plt.title('Confusion matrix')
plt.savefig('Confusion_Matrix.png')
plt.show()
```

```
Evaluation metrics for neural network model:
Accuracy: 0.9835
Precision: 0.9835
Recall: 0.9835
f1: 0.9835
```



Download model

Model is saved into a compatible form with tensorflow.js and a zip is created to make model easy to download.

```
# Save model
model_final.save('model.h5')
```

```
pip install tensorflowjs
```

```
Collecting tensorflowjs
  Downloading https://files.pythonhosted.org/packages/d4/1f/632d04bec71d4736a4e0e512cf41236b3416ac00d0a532f6511a829d
     |████████████████████████████████| 71kB 3.7MB/s
Collecting tensorflow-hub<0.10,>=0.7.0
  Downloading https://files.pythonhosted.org/packages/ac/83/a7df82744a794107641dad1decaad017d82e25f0e1f761ac9204829e
     |████████████████████████████████| 112kB 13.7MB/s
Requirement already satisfied: six<2,>=1.12.0 in /usr/local/lib/python3.7/dist-packages (from tensorflowjs) (1.15.0)
Requirement already satisfied: tensorflow<3,>=2.1.0 in /usr/local/lib/python3.7/dist-packages (from tensorflowjs) (2
Requirement already satisfied: h5py<3,>=2.8.0 in /usr/local/lib/python3.7/dist-packages (from tensorflowjs) (2.10.0)
Requirement already satisfied: protobuf>=3.8.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow-hub<0.10,>
Requirement already satisfied: numpy>=1.12.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow-hub<0.10,>=0
Requirement already satisfied: keras-preprocessing~=1.1.2 in /usr/local/lib/python3.7/dist-packages (from tensorflow
Requirement already satisfied: typing-extensions~=3.7.4 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3
Requirement already satisfied: termcolor~=1.1.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0
Requirement already satisfied: wrapt~=1.12.1 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0->t
Requirement already satisfied: opt-einsum~=3.3.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.
Requirement already satisfied: wheel~=0.35 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0->ten
Requirement already satisfied: google-pasta~=0.2 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.
Requirement already satisfied: tensorboard~=2.4 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0
Requirement already satisfied: absl-py~=0.10 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0->t
Requirement already satisfied: tensorflow-estimator<2.5.0,>=2.4.0 in /usr/local/lib/python3.7/dist-packages (from te
Requirement already satisfied: flatbuffers~=1.12.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.
Requirement already satisfied: astunparse~=1.6.3 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.
Requirement already satisfied: gast==0.3.3 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0->ten
Requirement already satisfied: grpcio~=1.32.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>=2.1.0->
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from protobuf>=3.8.0->tensorflo
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->ten
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4-
Requirement already satisfied: google-auth<2,>=1.6.3 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->t
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python3.7/dist-packages (from tens
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.7/dist-packages (from tensorb
Requirement already satisfied: importlib-metadata; python_version < "3.8" in /usr/local/lib/python3.7/dist-packages
```

```
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0-
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->ten
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7/dist-packages (from google-auth<2,>
Requirement already satisfied: rsa<5,>=3.1.4; python_version >= "3.6" in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from google-auth<2,
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.7/dist-packages (from google-auth-
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata; python_
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.
Installing collected packages: tensorflow-hub, tensorflowjs
  Found existing installation: tensorflow-hub 0.11.0
    Uninstalling tensorflow-hub-0.11.0:
      Successfully uninstalled tensorflow-hub-0.11.0
Successfully installed tensorflow-hub-0.9.0 tensorflowjs-3.3.0
```

```python
# Save model into a compatible form with tensorflow.js
import tensorflowjs as tfjs
import os

currdir = os.getcwd()
path = currdir + "/tfjs_files"

try:
    os.mkdir(path)
except OSError:
    print ("Creation of the directory %s failed" % path)
else:
    print ("Successfully created the directory %s " % path)

tfjs.converters.save_keras_model(model_final, path)
```

```
Successfully created the directory /content/tfjs_files
/usr/local/lib/python3.7/dist-packages/tensorflowjs/converters/keras_h5_conversion.py:123: H5pyDeprecationWarning: T
  return h5py.File(h5file)
```

```
import shutil

# Create zip to make model easy to download
shutil.make_archive('tfjs_files', 'zip', path)
```

```
'/content/tfjs_files.zip'
```