Classification model on MNIST databases

from the teachings of C. Labruère Chazal, and X. Dupuis

Classification of numbers

The MNIST (Mixed national Institute of Standards and Technology) database contains the normalized black and white images centered 28 of handwritten $0, 1, 2, \ldots, 9$ numbers. The training base contains 60000 images and the test base 10000 images.

Librairies

In [ ]:
```
# General libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from itertools import cycle

# Machine Learning
from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import RocCurveDisplay
from sklearn.model_selection import cross_val_score

# Deep Learning / Datasets
from tensorflow import keras
from keras import layers
```

Importation of the MNIST database

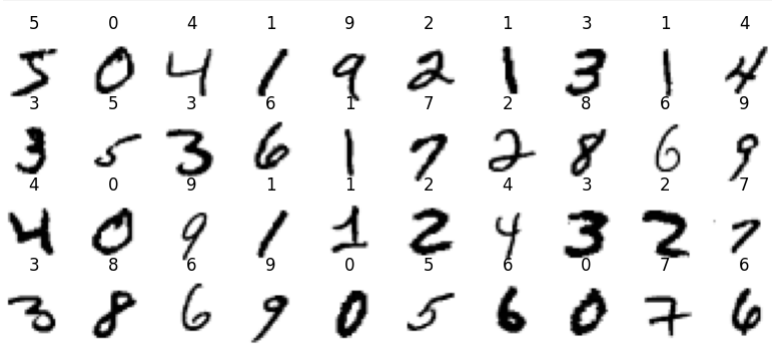
The dataset is already divided into a training set (60,000 images) and a test set (10,000 images).

In [ ]:
```
# MNIST database with Keras
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()

# Dimension of the dataset
print(f'dimensions de X_train : {X_train.shape}')
print(f'dimensions de Y_train : {y_train.shape}')
print(f'dimensions de X_test : {X_test.shape}')
print(f'dimensions de Y_test : {y_test.shape}')

# Data normalization : Pixels between 0 and 255 are reduced between 0 and 1.
x_max = X_train.max()
X_train, X_test = X_train / x_max, X_test / x_max
```

```
dimensions de X_train : (60000, 28, 28)
dimensions de Y_train : (60000,)
dimensions de X_test : (10000, 28, 28)
dimensions de Y_test : (10000,)
```

Overview of the dataset

In [ ]:
```
# We display the first 40 observations of the data set
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols, n_rows))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        # Data in matrix (image) form
        plt.imshow(X_train[index], cmap="binary")
        plt.axis('off')
        # Labels corresponding to each observation
        plt.title(y_train[index])
plt.show()
```



Machine Learning : Multi-class classification

We start by reshaping the train and test samples into vectors of dimension $(60000, 784)$ and $(10000, 784)$.

In [ ]:
```
X_train = X_train.reshape(60000,784)
X_test = X_test.reshape(10000,784)
```

We build a `SGDClassifier` and train it on `X_train`, `y_train`. We will then look at the precision on the test sample.

In [ ]:
```
# Design and training
sgd_clf=SGDClassifier(random_state=123)
sgd_clf.fit(X_train,y_train)

# We calculate the accuracy on the test sample
print(f'Accuracy : {round(accuracy_score(y_test,sgd_clf.predict(X_test))*100,2)}%')
```

```
Accuracy : 91.37%
```

We'll use `cross_val_score` to choose either the SGDClassifier or the RandomForestClassifier. It returns a table of the estimator scores for each run of the cross-validation. We do this because computing the precision of a RandomForest is pointless.

In [ ]:
```
SGD_CV = cross_val_score(sgd_clf,X_test,y_test,cv=3, scoring="accuracy")
print(f'Cross validation output : {SGD_CV}')
```

```
Cross validation output : [0.83293341 0.87938794 0.88388839]
```

In [ ]:
```
# Design and training
forest_clf=RandomForestClassifier(random_state=123)
forest_clf.fit(X_train,y_train)

# Evaluation by cross-validation
RF_CV = cross_val_score(forest_clf,X_test,y_test,cv=3, scoring="accuracy")
print(f'Cross validation output : {RF_CV}')
```

```
Cross validation output : [0.92291542 0.93729373 0.95829583]
```

The purpose of this section was to provide an overview of the performance obtained via Machine Learning. The common Machine Learning models offer good results on the MNIST database (Numbers), but Deep Learning models are outperforming Machine Learning models. This is what we will study in the next section.

Deep Learning

We start by creating a neural network using `Keras.Sequential`. We then train it by specifying a loss function, an optimizer and possibly some metrics. Here we will use the function `categorical_crossentropy`, the optimizer `sgd` and the metric `accuracy`.

In [ ]:
```
# Neural Network Creation by adding Layers
model = keras.Sequential()
model.add(layers.Dense(128, name = 'hidden1', activation = 'relu', input_shape = (X_train.shape[1],)))
model.add(layers.Dense(128, name = 'hidden2', activation = 'relu'))
model.add(layers.Dense(10, name = 'output', activation = 'softmax'))
model.summary()
```

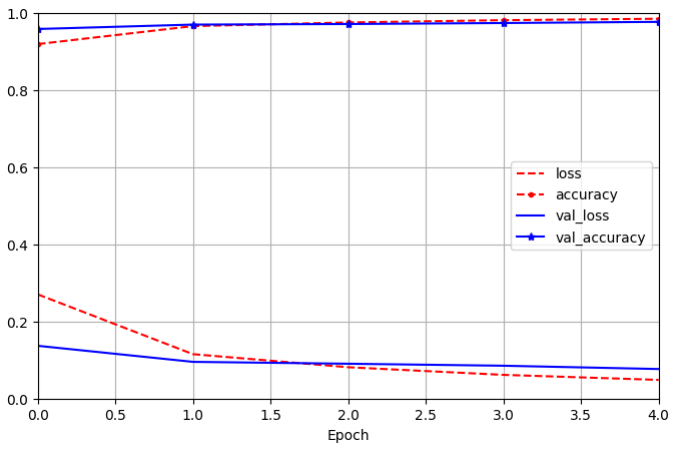
```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden1 (Dense)             (None, 128)               100480

 hidden2 (Dense)             (None, 128)               16512

 output (Dense)              (None, 10)                1290

=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden1 (Dense)             (None, 128)               100480

 hidden2 (Dense)             (None, 128)               16512

 output (Dense)              (None, 10)                1290

=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
_________________________________________________________________
Epoch 1/5
1875/1875 [==============================] - 2s 946us/step - loss: 0.2701 - accuracy: 0.9188 - val_loss: 0.1369 - val_accuracy: 0.9579
Epoch 2/5
1875/1875 [==============================] - 2s 919us/step - loss: 0.1152 - accuracy: 0.9654 - val_loss: 0.0953 - val_accuracy: 0.9693
Epoch 3/5
1875/1875 [==============================] - 2s 906us/step - loss: 0.0813 - accuracy: 0.9748 - val_loss: 0.0904 - val_accuracy: 0.9708
Epoch 4/5
1875/1875 [==============================] - 2s 879us/step - loss: 0.0615 - accuracy: 0.9808 - val_loss: 0.0852 - val_accuracy: 0.9734
Epoch 5/5
1875/1875 [==============================] - 2s 867us/step - loss: 0.0485 - accuracy: 0.9845 - val_loss: 0.0767 - val_accuracy: 0.9764
```

The summary gives us the number of parameters of each layer.

In [ ]:
```
# Network Training
model.compile(loss = 'sparse_categorical_crossentropy',  # 'sparse_categorical_crossentropy' or keras.losses.SparseCategoricalCrossentropy()
              optimizer = keras.optimizers.SGD(learning_rate = 0.1), # 'sgd' or keras.optimizers.SGD()
              metrics = ['accuracy'])  # 'accuracy' or keras.metrics.SparseCategoricalAccuracy()

fit = model.fit(X_train, y_train, batch_size = 32, epochs = 5, validation_data = (X_test, y_test))
```

We now graphically represent the accuracy and the loss on the test set train.

In [ ]:
```
pd.DataFrame(fit.history).plot(
    figsize=(8, 5), xlim=[0, 4], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```



We can then evaluate the model as follows.

In [ ]:
```
print('test loss and accuracy:')
print(model.evaluate(X_test, y_test, batch_size = X_test.shape[0], verbose = 0))
```

```
test loss and accuracy:
[0.07667961716651917, 0.9764000177383423]
```

Clothing classification

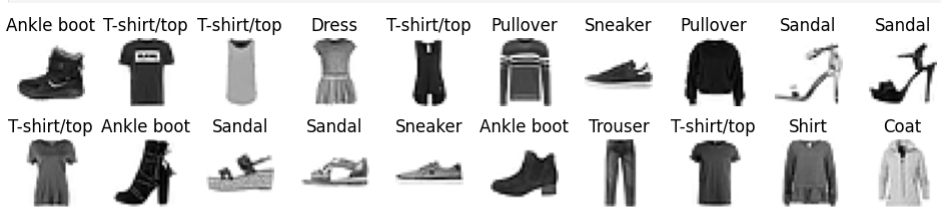
Importation of MNIST fashion database

Similarly to the MNIST database (numbers) the dataset is already divided into a training set (60,000 images) and a test set (10,000 images).

In [ ]:
```
# Train set & test set
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
# Tags
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]

print(f'dimensions de X_train : {X_train.shape}')
print(f'dimensions de y_train : {y_train.shape}')
print(f'dimensions de X_test : {X_test.shape}')
print(f'dimensions de y_test : {y_test.shape}')

# Data normalization
x_max = X_train.max()
X_train, X_test = X_train / x_max, X_test / x_max
```

```
dimensions de X_train : (55000, 28, 28)
dimensions de y_train : (55000,)
dimensions de X_test : (10000, 28, 28)
dimensions de y_test : (10000,)
```

In [ ]:
```
# We display the first 40 observations of the data set
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary")
        plt.axis('off')
        plt.title(class_names[y_train[index]])
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```



Neural Network Creation

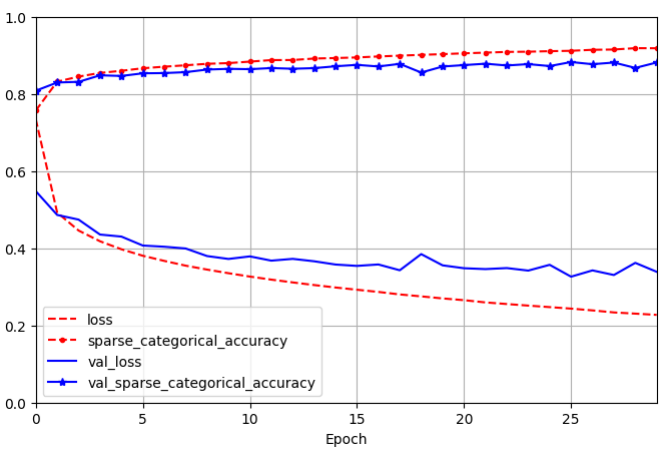
In [ ]:
```
# Neural Network Creation (all layers at once)
model = keras.Sequential(
    [
        layers.Flatten(name = 'input', input_shape = X_train.shape[1:]), # Flatten is for a set of matrix shape
        layers.Dense(300, name = 'hidden1', activation = 'relu'),
        layers.Dense(100, name = 'hidden2', activation = 'relu'),
        layers.Dense(10, name = 'output', activation = 'softmax')
    ]
)
model.summary()
```

```
Model: "sequential_49"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (Flatten)             (None, 784)               0

 hidden1 (Dense)             (None, 300)               235500

 hidden2 (Dense)             (None, 100)               30100

 output (Dense)              (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```


```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (Flatten)             (None, 784)               0

 hidden1 (Dense)             (None, 300)               235500

 hidden2 (Dense)             (None, 100)               30100

 output (Dense)              (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```


In [ ]:
```
# Compilation
model.compile(loss = 'sparse_categorical_crossentropy' ,
              optimizer = 'sgd',
              metrics = ['sparse_categorical_accuracy'])

fit = model.fit(X_train, y_train, epochs=30,
                validation_data=(X_test, y_test))
```

In [ ]:
```
pd.DataFrame(fit.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```



There is a slight difference between the two curves for both loss and accuracy (between train and test) which means that the model fits better on the training sample. The curves do not seem to meet after a few epochs, we can suppose that there is overfitting.

In [ ]:
```
print('test loss and accuracy:')
print(model.evaluate(X_test, y_test, batch_size = X_test.shape[0], verbose = 0))
```

```
test loss and accuracy:
[0.33899614214897156, 0.8813999891281128]
```

The accuracy is acceptable, but we would like to improve it. We can start by doing more epochs and use an early stop in case of stagnation while keeping the best model obtained.

In [ ]:
```
# Implementation of an early stop and a callback
save_best = keras.callbacks.ModelCheckpoint(filepath='/home/jsauce/model-best.h5', verbose=1, save_best_only = True, monitor = 'val_sparse_categorical_accuracy', mode = 'max')
early = keras.callbacks.EarlyStopping(monitor='val_sparse_categorical_accuracy', patience=4, verbose=1, mode='max', restore_best_weights=True)

fit = model.fit(X_train, y_train, batch_size = 32, epochs = 30, validation_data = (X_test, y_test), callbacks = [save_best, early])
```

```
Epoch 1/30
1699/1719 [==========================>.] - ETA: 0s - loss: 0.2232 - sparse_categorical_accuracy: 0.9199
Epoch 1: val_sparse_categorical_accuracy improved from -inf to 0.88060, saving model to /home/jsauce\model-best.h5
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2233 - sparse_categorical_accuracy: 0.9197 - val_loss: 0.3384 - val_sparse_categorical_accuracy: 0.8806
Epoch 2/30
1705/1719 [==========================>.] - ETA: 0s - loss: 0.2201 - sparse_categorical_accuracy: 0.9211
Epoch 2: val_sparse_categorical_accuracy improved from 0.88060 to 0.88520, saving model to /home/jsauce\model-best.h5
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2200 - sparse_categorical_accuracy: 0.9211 - val_loss: 0.3267 - val_sparse_categorical_accuracy: 0.8852
Epoch 3/30
1687/1719 [==========================>.] - ETA: 0s - loss: 0.2161 - sparse_categorical_accuracy: 0.9223
Epoch 3: val_sparse_categorical_accuracy did not improve from 0.88520
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2162 - sparse_categorical_accuracy: 0.9225 - val_loss: 0.3313 - val_sparse_categorical_accuracy: 0.8820
Epoch 4/30
1704/1719 [==========================>.] - ETA: 0s - loss: 0.2133 - sparse_categorical_accuracy: 0.9234
Epoch 4: val_sparse_categorical_accuracy did not improve from 0.88520
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2136 - sparse_categorical_accuracy: 0.9233 - val_loss: 0.3300 - val_sparse_categorical_accuracy: 0.8836
Epoch 5/30
1703/1719 [==========================>.] - ETA: 0s - loss: 0.2097 - sparse_categorical_accuracy: 0.9250
Epoch 5: val_sparse_categorical_accuracy did not improve from 0.88520
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2093 - sparse_categorical_accuracy: 0.9251 - val_loss: 0.3362 - val_sparse_categorical_accuracy: 0.8810
Epoch 6/30
1708/1719 [==========================>.] - ETA: 0s - loss: 0.2065 - sparse_categorical_accuracy: 0.9259
Epoch 6: val_sparse_categorical_accuracy did not improve from 0.88520
Restoring model weights from the end of the best epoch: 2.
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2062 - sparse_categorical_accuracy: 0.9260 - val_loss: 0.3290 - val_sparse_categorical_accuracy: 0.8849
Epoch 6: early stopping
```

In [ ]:
```
print('test loss and accuracy:')
print(model.evaluate(X_test, y_test, batch_size = X_test.shape[0], verbose = 0))
```

```
test loss and accuracy:
[0.3266647756099701, 0.885200023651123]
```

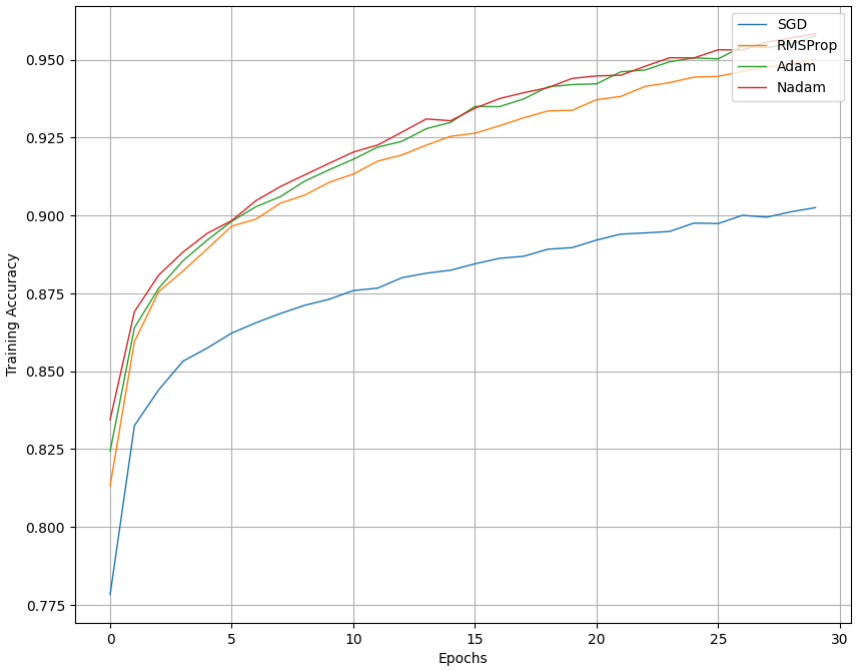
There is effectively a stagnation. We will therefore proceed differently by first focusing on a better choice of optimizer.

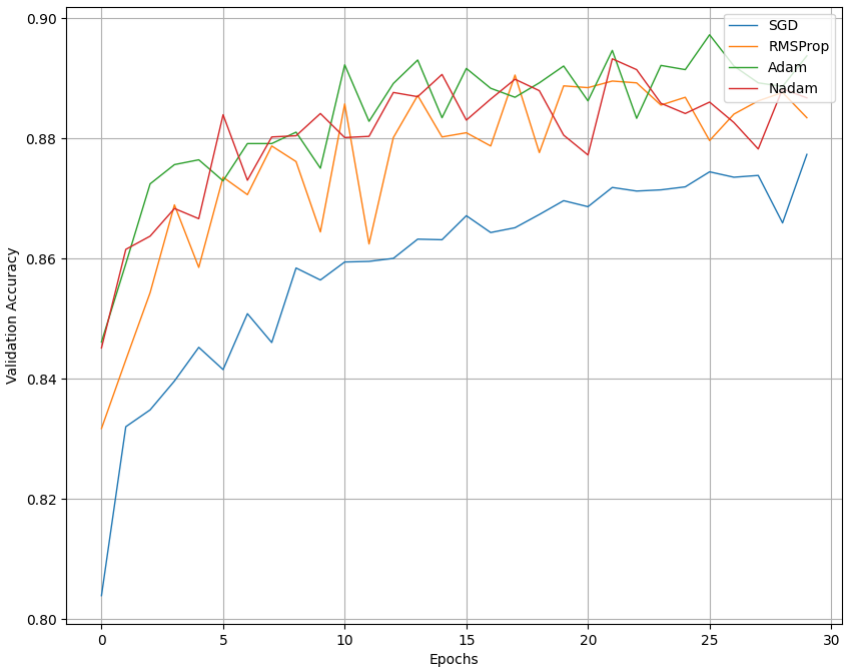
Searching for the best optimizer

We consider the same model structure. We compile it via different optimizers then we train it on 30 epochs. We will then represent graphically accuracy versus epochs of each network.

In [ ]:
```
def my_network(Optimizer):

    model = keras.Sequential(
    [
        layers.Flatten(name = 'input', input_shape = X_train.shape[1:]),
        layers.Dense(300, name = 'hidden1', activation = 'elu'),
        layers.Dense(100, name = 'hidden2', activation = 'elu'),
        layers.Dense(10, name = 'output', activation = 'softmax')
    ]
    )

    model.compile(loss=keras.losses.sparse_categorical_crossentropy,
                  optimizer=Optimizer,
                  metrics=[keras.metrics.sparse_categorical_accuracy])

    return model

SGD_fit = my_network('sgd').fit(X_train, y_train, batch_size = 32, epochs = 30, validation_data = (X_test, y_test), callbacks = [save_best])
Adam_fit = my_network('Adam').fit(X_train, y_train, batch_size = 32, epochs = 30, validation_data = (X_test, y_test), callbacks = [save_best])
Nadam_fit = my_network('Nadam').fit(X_train, y_train, batch_size = 32, epochs = 30, validation_data = (X_test, y_test), callbacks = [save_best])
RMSprop_fit = my_network('RMSprop').fit(X_train, y_train, batch_size = 32, epochs = 30, validation_data = (X_test, y_test), callbacks = [save_best])
```

In [ ]:
```
# Graphic illustration
for accur in ('sparse_categorical_accuracy', 'val_sparse_categorical_accuracy'):
    plt.figure(figsize=(10, 8))
    opt_names = "SGD RMSProp Adam Nadam"
    for fit, opt_name in zip((SGD_fit, RMSprop_fit, Adam_fit, Nadam_fit),
                             opt_names.split()):
        plt.plot(fit.history[accur], label=f"{opt_name}", linewidth=1)

    plt.grid()
    plt.xlabel("Epochs")
    plt.ylabel({'sparse_categorical_accuracy': "Training Accuracy", 'val_sparse_categorical_accuracy': "Validation Accuracy"}[accur])
    plt.legend(loc="upper right")
    plt.show()
```



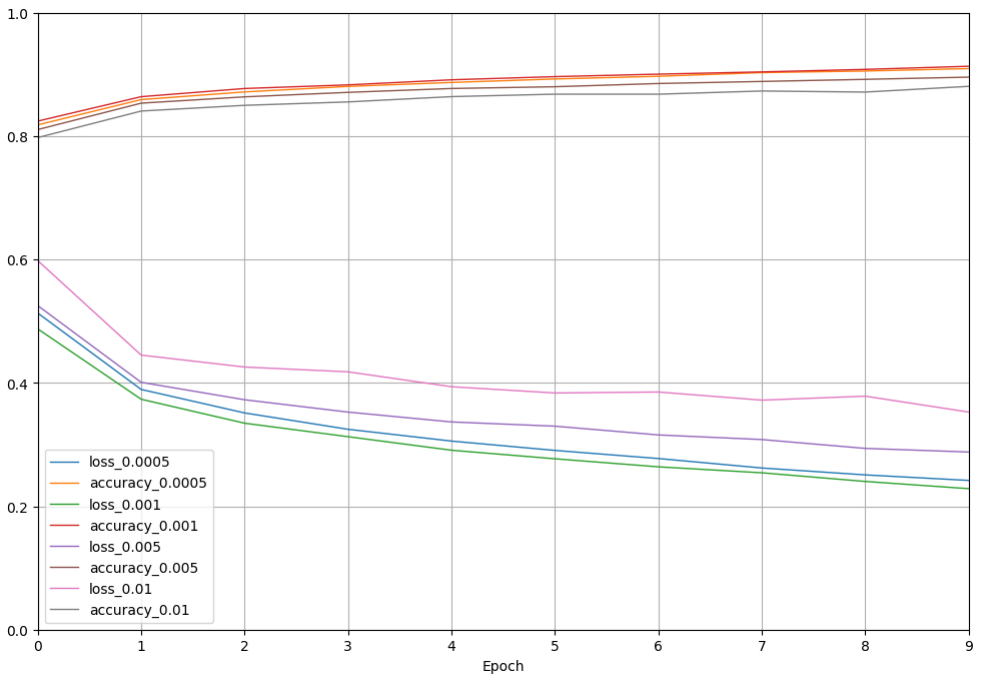


We choose the optimizer Adam which seems to be the most relevant. We are now going to rebuild a network with it. Moreover, we will deepen the research by considering an alpha learning rate. The goal of the approach is to find the best learning rate. We proceed as before, this time we build several networks with different learning rate.

In [ ]:
```
def my_network_LR(alpha):

    model = keras.Sequential(
    [
        layers.Flatten(name = 'input', input_shape = X_train.shape[1:]),
        layers.Dense(300, name = 'hidden1', activation = 'elu'),
        layers.Dense(100, name = 'hidden2', activation = 'elu'),
        layers.Dense(10, name = 'output', activation = 'softmax')
    ]
    )

    model.compile(loss=keras.losses.sparse_categorical_crossentropy,
              optimizer=keras.optimizers.Adam(learning_rate=alpha),
              metrics=[keras.metrics.sparse_categorical_accuracy])

    return model
```

In [ ]:
```
alpha_val=[0.0005,0.001,0.005,0.01]
fits = []
for alpha in alpha_val :
    fits.append(my_network_LR(alpha).fit(X_train, y_train, batch_size = 64,  epochs=10))
```

In [ ]:
```
fig, ax = plt.subplots(figsize=(12,8))
for i in range(4):
      df = pd.DataFrame(fits[i].history)
      df = df.rename(columns={"loss": f"loss_{alpha_val[i]}", "sparse_categorical_accuracy": f"accuracy_{alpha_val[i]}"})
      df.plot(ax=ax, xlim=[0, 9], ylim=[0, 1], grid=True, xlabel="Epoch", linewidth=1)
```



We choose a learning rate of 0.001.

In [ ]:
```
my_model = my_network_LR(0.001)

# Callbacks Implementation
checkpoint_filepath = '/home/jsauce/model-RMS_Prop-best.h5'
save_best = keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath, verbose=1, save_best_only = True, monitor = 'val_sparse_categorical_accuracy', mode = 'max')
early = keras.callbacks.EarlyStopping(monitor='val_sparse_categorical_accuracy', patience=4, verbose=1, mode='max', restore_best_weights=True)


my_model.fit(X_train, y_train, batch_size = 64,  epochs=30,
                    validation_data=(X_test, y_test), callbacks = [save_best, early])
```

```
Epoch 1/30
860/860 [==============================] - ETA: 0s - loss: 0.4875 - sparse_categorical_accuracy: 0.8229
Epoch 1: val_sparse_categorical_accuracy improved from -inf to 0.84830, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 2s 2ms/step - loss: 0.4875 - sparse_categorical_accuracy: 0.8229 - val_loss: 0.4270 - val_sparse_categorical_accuracy: 0.8483
Epoch 2/30
835/860 [============================>.] - ETA: 0s - loss: 0.3729 - sparse_categorical_accuracy: 0.8632
Epoch 2: val_sparse_categorical_accuracy did not improve from 0.84830
860/860 [==============================] - 1s 2ms/step - loss: 0.3736 - sparse_categorical_accuracy: 0.8631 - val_loss: 0.4221 - val_sparse_categorical_accuracy: 0.8444
Epoch 3/30
830/860 [===========================>..] - ETA: 0s - loss: 0.3366 - sparse_categorical_accuracy: 0.8753
Epoch 3: val_sparse_categorical_accuracy improved from 0.84830 to 0.86190, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.3362 - sparse_categorical_accuracy: 0.8753 - val_loss: 0.3880 - val_sparse_categorical_accuracy: 0.8619
Epoch 4/30
827/860 [===========================>..] - ETA: 0s - loss: 0.3106 - sparse_categorical_accuracy: 0.8842
Epoch 4: val_sparse_categorical_accuracy improved from 0.86190 to 0.86560, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.3118 - sparse_categorical_accuracy: 0.8839 - val_loss: 0.3720 - val_sparse_categorical_accuracy: 0.8656
Epoch 5/30
824/860 [===========================>..] - ETA: 0s - loss: 0.2910 - sparse_categorical_accuracy: 0.8922
Epoch 5: val_sparse_categorical_accuracy improved from 0.86560 to 0.87100, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.2924 - sparse_categorical_accuracy: 0.8917 - val_loss: 0.3589 - val_sparse_categorical_accuracy: 0.8710
Epoch 6/30
838/860 [============================>.] - ETA: 0s - loss: 0.2747 - sparse_categorical_accuracy: 0.8968
Epoch 6: val_sparse_categorical_accuracy improved from 0.87100 to 0.88060, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.2753 - sparse_categorical_accuracy: 0.8965 - val_loss: 0.3365 - val_sparse_categorical_accuracy: 0.8806
Epoch 7/30
836/860 [============================>.] - ETA: 0s - loss: 0.2632 - sparse_categorical_accuracy: 0.9010
Epoch 7: val_sparse_categorical_accuracy did not improve from 0.88060
860/860 [==============================] - 1s 1ms/step - loss: 0.2634 - sparse_categorical_accuracy: 0.9011 - val_loss: 0.3622 - val_sparse_categorical_accuracy: 0.8717
Epoch 8/30
859/860 [============================>.] - ETA: 0s - loss: 0.2518 - sparse_categorical_accuracy: 0.9050
Epoch 8: val_sparse_categorical_accuracy improved from 0.88060 to 0.88080, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.2518 - sparse_categorical_accuracy: 0.9050 - val_loss: 0.3452 - val_sparse_categorical_accuracy: 0.8808
Epoch 9/30
827/860 [===========================>..] - ETA: 0s - loss: 0.2375 - sparse_categorical_accuracy: 0.9101
Epoch 9: val_sparse_categorical_accuracy improved from 0.88080 to 0.88360, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.2377 - sparse_categorical_accuracy: 0.9099 - val_loss: 0.3371 - val_sparse_categorical_accuracy: 0.8836
Epoch 10/30
855/860 [============================>.] - ETA: 0s - loss: 0.2257 - sparse_categorical_accuracy: 0.9144
Epoch 10: val_sparse_categorical_accuracy improved from 0.88360 to 0.88640, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.2256 - sparse_categorical_accuracy: 0.9145 - val_loss: 0.3476 - val_sparse_categorical_accuracy: 0.8864
Epoch 11/30
823/860 [===========================>..] - ETA: 0s - loss: 0.2189 - sparse_categorical_accuracy: 0.9182
Epoch 11: val_sparse_categorical_accuracy did not improve from 0.88640
860/860 [==============================] - 1s 1ms/step - loss: 0.2189 - sparse_categorical_accuracy: 0.9182 - val_loss: 0.3309 - val_sparse_categorical_accuracy: 0.8862
Epoch 12/30
848/860 [============================>.] - ETA: 0s - loss: 0.2087 - sparse_categorical_accuracy: 0.9210
Epoch 12: val_sparse_categorical_accuracy did not improve from 0.88640
860/860 [==============================] - 1s 2ms/step - loss: 0.2093 - sparse_categorical_accuracy: 0.9207 - val_loss: 0.3623 - val_sparse_categorical_accuracy: 0.8788
Epoch 13/30
853/860 [============================>.] - ETA: 0s - loss: 0.2022 - sparse_categorical_accuracy: 0.9234
Epoch 13: val_sparse_categorical_accuracy improved from 0.88640 to 0.88890, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.2022 - sparse_categorical_accuracy: 0.9234 - val_loss: 0.3417 - val_sparse_categorical_accuracy: 0.8889
Epoch 14/30
827/860 [===========================>..] - ETA: 0s - loss: 0.1920 - sparse_categorical_accuracy: 0.9271
Epoch 14: val_sparse_categorical_accuracy improved from 0.88890 to 0.89330, saving model to /home/jsauce\model-RMS_Prop-best.h5
860/860 [==============================] - 1s 2ms/step - loss: 0.1927 - sparse_categorical_accuracy: 0.9271 - val_loss: 0.3302 - val_sparse_categorical_accuracy: 0.8933
Epoch 15/30
850/860 [============================>.] - ETA: 0s - loss: 0.1815 - sparse_categorical_accuracy: 0.9317
Epoch 15: val_sparse_categorical_accuracy did not improve from 0.89330
860/860 [==============================] - 1s 2ms/step - loss: 0.1817 - sparse_categorical_accuracy: 0.9317 - val_loss: 0.3668 - val_sparse_categorical_accuracy: 0.8874
Epoch 16/30
858/860 [============================>.] - ETA: 0s - loss: 0.1800 - sparse_categorical_accuracy: 0.9307
Epoch 16: val_sparse_categorical_accuracy did not improve from 0.89330
860/860 [==============================] - 1s 1ms/step - loss: 0.1801 - sparse_categorical_accuracy: 0.9306 - val_loss: 0.3496 - val_sparse_categorical_accuracy: 0.8907
Epoch 17/30
831/860 [===========================>..] - ETA: 0s - loss: 0.1691 - sparse_categorical_accuracy: 0.9346
Epoch 17: val_sparse_categorical_accuracy did not improve from 0.89330
860/860 [==============================] - 1s 1ms/step - loss: 0.1688 - sparse_categorical_accuracy: 0.9348 - val_loss: 0.3486 - val_sparse_categorical_accuracy: 0.8919
Epoch 18/30
855/860 [============================>.] - ETA: 0s - loss: 0.1642 - sparse_categorical_accuracy: 0.9372
Epoch 18: val_sparse_categorical_accuracy did not improve from 0.89330
Restoring model weights from the end of the best epoch: 14.
860/860 [==============================] - 1s 1ms/step - loss: 0.1643 - sparse_categorical_accuracy: 0.9372 - val_loss: 0.3712 - val_sparse_categorical_accuracy: 0.8857
Epoch 18: early stopping
```

Out[ ]:
```
<keras.callbacks.History at 0x22973d67550>
```

In [ ]:
```
print('test loss and accuracy:')
print(my_model.evaluate(X_test, y_test))
```

```
test loss and accuracy:
313/313 [==============================] - 0s 805us/step - loss: 0.3302 - sparse_categorical_accuracy: 0.8933
[0.33017459511756897, 0.8932999968528748]
```

We managed to gain some accuracy despite the overfitting on the data.

Making predictions

In [ ]:
```
X_new = X_test[:10]
y_proba = my_model.predict(X_new)
y_proba.round(2)
y_pred = y_proba.argmax(axis=-1)

# We try to predict the first 10 observations
n_rows = 1
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_test[index], cmap="binary")
        plt.axis('off')
        plt.title(class_names[y_pred[index]])
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

```
1/1 [==============================] - 0s 32ms/step
```

We are able to correctly predict the first 10 observations.

Brief conclusion: Neural networks are very powerful for classification and are very interesting methods that can be adapted according to many parameters.