Table of Contents

Introduction to the Multi Layer Perceptron

Introduction : a visual intuition of activation functions

Using a regression task on the sinus function, we'll try to get an intuition of the effect of activation functions.

In [ ]:

```
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time
```

In [ ]:

```
from sklearn.model_selection import train_test_split
```

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.utils import plot_model
from tensorflow.keras.callbacks import EarlyStopping
```

In [ ]:

```
X = np.arange(-3*np.pi, 3*np.pi, 0.01)
y = np.sin(X)

plt.figure(figsize=(13, 8))
plt.plot(X, y, label='sinus', color='red')
plt.legend()
plt.show()
```

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)

plt.figure(figsize=(13, 8))
plt.scatter(X_train, y_train, alpha=0.03, label='training data')
plt.scatter(X_test, y_test, alpha=0.5, label='testing data')

plt.legend()
plt.show()
```

$$[TODO - Students]$$

#

- Build a model with 1 hidden layer in 1 dimension and train it on x_train, y_train.
- What activation should we use for ouput layer ? A linear activation function since we are in a regression problem
- What loss should we use ? MSE since we are in a regression problem #
- Try different activations for the hidden layer and plot the predictions obtained on x_test
- Plot also a learning curve

In [ ]:
```
# Input layer, you have to fix the number of features
inputs = Input(shape=(1,), name="inputs")
# One hidden layer with 1 neuron
x_sig = Dense(1, activation="sigmoid", name="layer_1")(inputs)
out_sig = Dense(1, activation="linear", name="output")(x_sig)
model = Model(inputs, out_sig)
model.compile(loss="mse", optimizer="adam")
model.summary()

history = model.fit(X_train, y_train, epochs=100, validation_split=0.33)
```

In [ ]:
```
# Plot leaning cuve
plt.figure(figsize=(13, 8))
plt.plot(history.history['loss'])
plt.show()
```

In [ ]:
```
def plot_prediction(title, model):
```

```
    y_hat_test = model.predict(X_test)

    plt.figure()
    plt.scatter(X_test, y_test, label='ground_truth', alpha=0.1)
    plt.scatter(X_test, y_hat_test, label='predicted', alpha=0.5)
    plt.legend()

    plt.title(title)
    plt.show()
```

In [ ]:

```
plot_prediction("Sigmoid", model)
```

$$[TODO - Students]$$

Try adding layers and increasing the layers dimension to better fit the test data. You can use the following function to quickly build your models.

- Try different n_layers (for example 1, 10, 100)
- Try different hidden_dim (for example 32, 128, 256, 512)
- Try different bach size
- Try to understand the `patience` parameters of early stopping -> It represents the number of epochs to wait before early stop if no progress is made on the validation set

In [ ]:

```
def build_sin_regression(activation, n_layers, hidden_dim):
    input = Input(shape=(1,), name='input')

    for i in range(n_layers):
        if i == 0:
            x = Dense(input_shape=(1,), units=hidden_dim,
                      activation=activation, name='layer_'+str(i))(input)
        else:
            x = Dense(units=hidden_dim, activation=activation,
                      name='layer_'+str(i))(x)

    output = Dense(1, activation='linear', name='output')(x)
    model = Model(input, output, name='sinus_regression')
    return model
```

In [ ]:
```
n_layers_val = [1, 10, 100]
hidden_dim_val = [32, 128, 256, 512]
batch_size_val = [16, 24]

# try all combinations of provided values
for n_layers in n_layers_val:
    for hidden_dim in hidden_dim_val:
        for batch_size in batch_size_val:
            model = build_sin_regression(
                activation="sigmoid", n_layers=n_layers, hidden_dim=hidden_dim)
            model.compile(loss='mse', optimizer='adam')
            model.summary()

            plot_model(model)

            callbacks_list = [EarlyStopping(monitor='val_loss', min_delta=0.005, patience=8, verbose=2, mode='min', r
                              ]

            history = model.fit(X_train, y_train, validation_split=0.1,
                                callbacks=callbacks_list, batch_size=batch_size, epochs=20)

            plt.figure(figsize=(13, 8))
            plt.plot(history.history['loss'], label="loss")
            plt.plot(history.history['val_loss'], label="val_loss")
            plt.legend()
            plt.show()

            y_hat_test = model.predict(X_test)

            plt.scatter(X_test, y_test, label='ground_truth', alpha=0.1)
            plt.scatter(X_test, y_hat_test, label='predicted', alpha=0.5)

            plt.legend()
            plt.show()
```

Build an MLP to classify MNIST images

Every MNIST data point has two parts: an image of a handwritten digit and a corresponding label. We'll call the images “x” and the labels “y”. Both the training set and test set contain images and their corresponding labels; for example, the training images are mnist.train.images and the training labels are mnist.train.labels.

In [ ]:
```
# Load dataset

# the data, shuffled and split between a train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train.shape, y_train.shape
```

In [ ]:
```
# Reshape the image from 3d to 2d (nb_items, other dime)
X_train = X_train.reshape(60000, 784)
```

In [ ]:
```
# Normalize the data (input between 0 and 1)
```

In [ ]:
```
# One hot encode the label
```

In [ ]:
```
# Buid MLP model
# You can use the following function
def build_MLP(input_shape, activation, layers, nb_class):
    input = Input(shape=(input_shape,), name='input')

    for i, hidden_size in enumerate(layers):
        if i == 0:
            x = Dense(input_shape=(input_shape,), units=hidden_size,
                      activation=activation, name='layer_'+str(i))(input)
        else:
            x = Dense(units=hidden_size, activation=activation,
                      name='layer_'+str(i))(x)

    output = Dense(nb_class, activation='softmax', name='output')(x)
    model = Model(input, output, name='mnist_classifier')
    model.summary()
    return model


model = build_MLP()  # TO BE COMPLETED
```

In [ ]:
```
# Compile and fit the model
callbacks_list = [EarlyStopping(monitor='val_accuracy', min_delta=0.005, patience=20,
                                verbose=2, mode='min', restore_best_weights=True)
```

```
                  ]

model.compile(loss='categorical_crossentropy',
              metrics=["accuracy"], optimizer='adam')
history = model.fit(''' X''', ''' y ''', validation_split=0.1, callbacks=callbacks_list,
                    batch_size=32, epochs=20)
```

In [ ]:

```
# Print history keys
history.history.keys()
```

In [ ]:

```
# Babysit your model
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(26, 8))

ax1.plot(history.history['loss'], label="loss")
ax1.plot(history.history['val_loss'], label="val_loss")
ax1.legend()
ax2.plot(history.history['accuracy'], label="accuracy")
ax2.plot(history.history['val_accuracy'], label="val_accuracy")
ax2.legend()
plt.show()
```

In [ ]:

```
# Evaluate the model
score = model.evaluate(X_test, y_test_enc)
print('Test loss:', score[0])
print('Test accuracy', score[1])
```

In [ ]:

```
# Modify the network in order to obtain better accuracy (better than 0.96)
```