LAB 2

Topics 1/2: save/load models, evaluate models, models for prediction

In [ ]:
```
import tensorflow as tf
import numpy as np
```

In [ ]:
```
tf.random.set_seed(1234)
np.random.seed(1234)
```

Useful if we want to modify something in the model (with the same seed we can compare)

The model depend on the initialization, maybe we change the model and the initialization and we obtain better results because of the init.

In [ ]:
```
# Only if you want to use Google Drive in Colab to save your models
from google.colab import drive
```

In [ ]:
```
# Mount Google Drive.
# After running the cell, click to the link and follow the instructions to obtain your authorization code.
# Once the operation is completed you can access your Google Drive home in '/content/drive/My Drive/'
drive.mount('/content/drive')
```

In [ ]:
```
# We can now save files in Drive
with open('/content/drive/My Drive/Keras2/test.txt', 'w') as f:
  f.write('Hello Google Drive!')
```

In [ ]:
```
!cat /content/drive/My\ Drive/Keras2/test.txt
```

Let's try to save a model. In this example we will create a model, and we will save the model in memory (Google Drive, if you are using Colab, or your local memory)

This time we create the model in a SEQUENTIAL way (equivalent)

In [ ]:
```
# Create base model (e.g., Input -> Hidden -> Out)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
model.add(tf.keras.layers.Dense(units=64, activation=tf.keras.activations.sigmoid))
model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
```

← classification layer

In [ ]:
```
# Visualize created model as a table
model.summary()

# Visualize initialized weights
tf.reduce_sum(model.layers[1].weights[0])
```

layer	Output Shape	param
Flatten	(None, 784)	0
dense	(None, 64)	50240
dense_1	(None, 10)	650

biases: (784 * 64 + 64), (64 * 10 + 10)

In [ ]:
```
# Create a random input
random_input = tf.random.uniform([1, 28, 28])
```

we're simulating a single image ⇒ batchsize = 1

In [ ]:
```
# Compute the output given the random input (this is used later to compare saved and loaded models)
output = model(random_input)
```

In [ ]:
```
# We have two ways to save the model (see slides):
# 1. Save the entire model.
#     When you load the model, it will be already a Model object which can be used for training/evaluation.
#     Very useful for example for web/android applications.
#      - save
#                     model.save('/PATH/TO/YOUR/MODEL')
#      - load
#                     loaded_model = tf.keras.models.load_model('/PATH/TO/YOUR/MODEL')
#
# 2. Save only the weights
#     In this case we are only saving the weights of the model.
#     To load the model (the weights) we need to create the model again and redefine the same model structure of the saved model.
#      - save
#                     model.save_weights('/PATH/TO/YOUR/MODEL/WEIGHTS/')
#      - load
#                     loaded_model = tf.keras.Sequential()
#                     ...
#                     loaded_model.load_weights('/PATH/TO/YOUR/MODEL/WEIGHTS/')
```

1. Save the entire model

In [ ]:
```
# Let's try the first method
# Save the model
model.save('/content/drive/My Drive/Keras2/saved_model')  # use your local directory if you are not using Drive
```

In [ ]:
```
del model
```

In [ ]:
```
# Load the model
loaded_model = tf.keras.models.load_model('/content/drive/My Drive/Keras2/saved_model')  # use your local directory if you are not using Drive
```

(with this way, the loaded-model is ready to be used)

In [ ]:
```
# Check the output of the loaded model is the same of the output computed before saving
np.testing.assert_allclose(loaded_model(random_input), output)
```

2. Save only the weights

In [ ]:
```
# Let's try the second method (only weights)

loaded_model.save_weights('/content/drive/My Drive/Keras2/saved_model_weights') # use your local directory if you are not using Drive
```

In [ ]:
```
del loaded_model
```

In [ ]:
```
# We have to create the model again

new_model = tf.keras.Sequential()
new_model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
new_model.add(tf.keras.layers.Dense(units=64, activation=tf.keras.activations.sigmoid))
new_model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
```

this MUST be the same model structure

In [ ]:
```
# Check that the weights are not the same of the ones of the original model
tf.reduce_sum(new_model.layers[1].weights[0])
```

In [ ]:
```
# Load saved weights
new_model.load_weights('/content/drive/My Drive/Keras2/saved_model_weights')
```

In [ ]:
```
# Check again the weights. Now should be the same.
tf.reduce_sum(new_model.layers[1].weights[0])

# Check the output of the loaded model is the same of the output computed before saving
np.testing.assert_allclose(loaded_model(random_input), output)
```

In [ ]:
```
# Once the model is loaded we can use it for example to evaluate the performance on a new test set or to predict an output given specific input.
# Let's try to use a loaded model for evaluation.
# Please notice that in this example I will use a model I trained before. You have to replace it with your trained model.
```

In [ ]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

In [ ]:
```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
```

In [ ]:
```
model.load_weights('/content/drive/My Drive/Keras2/classification_experiments_/FC_Oct13_22-51-04/ckpts/cp_10.ckpt')
```

In [ ]:
```
# You have to compile also before using it for evaluation

# Optimization params
# -------------------

# Loss
loss = tf.keras.losses.CategoricalCrossentropy()

# learning rate
lr = 1e-5
optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
# -------------------

# Validation metrics
# -------------------

metrics = ['accuracy']
# -------------------

# Compile Model
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
```

In [ ]:
```
# evaluate method is similar to the fit one, except that no training is performed.
# The output is computed for each sample (batch) of your test set and the losses/metrics are evaluated.
# Finally, the overall losses and metrics are returned.

eval_out = model.evaluate(x=x_test / 255.,
                          y=tf.keras.utils.to_categorical(y_test),
                          batch_size=8,
                          verbose=1)
eval_out
```

name of the loaded/trained model

it prints 2 values: [•, •] = [loss on the test set, accuracy on the test set]

In [ ]:
```
# In the following an example of how to use the model for prediction. (≠ evaluation)
# In this example a drawn (literally with Paint :)) image is used. Just for fun!
# You will find it in the class folder.

# Compute output given x
from PIL import Image

shoe_img = Image.open('/content/drive/My Drive/Keras2/shoe.png').convert('L')
```

In [ ]:
```
shoe_img.resize((256, 256))
```

In [ ]:
```
shoe_arr = np.expand_dims(np.array(shoe_img), 0)  # to obtain the batch dimension

out_softmax = model.predict(x=shoe_arr / 255.)

out_softmax  # output is the probability distribution (softmax)
```

0 T-shirt/top \ 1 Trouser \ 2 Pullover \ 3 Dress \ 4 Coat \ 5 Sandal \ 6 Shirt \ 7 Sneaker \ 8 Bag \ 9 Ankle boot \

In [ ]:
```
# Get predicted class as the index corresponding to the maximum value in the vector probability
predicted_class = tf.argmax(out_softmax, 1)
predicted_class
```

Topics 2/2: Overfitting, model checkpoints, TensorBoard

In [ ]:
```
import tensorflow as tf
import numpy as np
```

In [ ]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

In [ ]:
```
tf.random.set_seed(1234)
np.random.seed(1234)
```

In [ ]:
```
from google.colab import drive
```

In [ ]:
```
drive.mount('/content/drive')
```

In [ ]:
```
# Load built-in dataset
# ------------------------------------------------------------------------------
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

In [ ]:
```
# Split in training and validation sets
# e.g., 50000 samples for training and 10000 samples for validation
x_valid = x_train[50000:]
y_valid = y_train[50000:]
x_train = x_train[:50000]
y_train = y_train[:50000]
```

In [ ]:
```
# Create Training Dataset object
# ------------------------------------------------------------------------------
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))

# Shuffle
train_dataset = train_dataset.shuffle(buffer_size=x_train.shape[0])

# Normalize images
def normalize_img(x_, y_):
    return tf.cast(x_, tf.float32) / 255., y_

train_dataset = train_dataset.map(normalize_img)

# 1-hot encoding <- for categorical cross entropy
def to_categorical(x_, y_):
    return x_, tf.one_hot(y_, depth=10)

train_dataset = train_dataset.map(to_categorical)

# Divide in batches
bs = 32
train_dataset = train_dataset.batch(bs)

# Repeat
# Without calling the repeat function the dataset  will be empty after consuming all the images
train_dataset = train_dataset.repeat()
```

In [ ]:
```
# Create Validation Dataset
# ------------------------------------------------------------------------------
valid_dataset = tf.data.Dataset.from_tensor_slices((x_valid, y_valid))
valid_dataset = valid_dataset.map(normalize_img)   # Normalize images
valid_dataset = valid_dataset.map(to_categorical)  # 1-hot encoding
valid_dataset = valid_dataset.batch(bs)            # Divide in batches
valid_dataset = valid_dataset.repeat()             # Repeat
```

In [ ]:
```
# Create Test Dataset
# ------------------------------------------------------------------------------
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.map(normalize_img)     # Normalize images
test_dataset = test_dataset.map(to_categorical)    # 1-hot encoding
test_dataset = test_dataset.batch(1)               # Divide in batches
```

In [ ]:
```
# In the following we create a different model based on the 'which_model' param:
# - base model (FC)
# - base model + dropout (FC_Dropout)
# - base model + weight decay (FC_WD)
# - base model + dropout + weight decay (FC_Dropout_WD)

# Please notice that in this example a deeper model is used w.r.t. the one of the previous class
# (with an higher number of parameters).
# This is to highlight the problem of overfitting.

which_model = 'FC'

if which_model == 'FC':
  model = tf.keras.Sequential()
  model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
  model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                  kernel_regularizer=tf.keras.regularizers.l2(0.001)))
  model.add(tf.keras.layers.Dropout(0.5))
  model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                  kernel_regularizer=tf.keras.regularizers.l2(0.001)))
  model.add(tf.keras.layers.Dropout(0.5))
  model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                  kernel_regularizer=tf.keras.regularizers.l2(0.001)))
  model.add(tf.keras.layers.Dropout(0.5))
  model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
elif which_model == 'FC_Dropout':
  model = tf.keras.Sequential()
  model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
  model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
  model.add(tf.keras.layers.Dropout(0.5))
```

10 classes

```
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
elif which_model == 'FC_WD':
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                    kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                    kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                    kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
elif which_model == 'FC_Dropout_WD':
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                    kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                    kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(units=512, activation=tf.keras.activations.relu,
                                    kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
```

In [ ]:

```
# Visualize created model as a table
model.summary()

# Visualize initialized weights
# model.weights
```

In [ ]:

```
# Optimization params
# -------------------------------------------------------------------------------
loss      = tf.keras.losses.CategoricalCrossentropy()   # Loss
lr        = 1e-3                                        # Learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=lr)  # Optimizer

# Validation metrics
# -------------------------------------------------------------------------------
metrics = ['accuracy']

# Compile Model
# -------------------------------------------------------------------------------
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
```

We do this here so we can see live what's happening

In [ ]:

```
# This cell is useful if you want to visualize Tensorboard in your Colab notebook.
# Please run the cell before training, otherwise you have to wait the training cell is completed before running this one

# If you are using jupyter notebook, you can skip this cell and open Tensorboard with the following:
# From terminal: tensorboard --logdir /PATH/TO/YOUR/EXPERIMENTS/ --port PORT
# Go to 127.0.0.1:PORT in your web browser

%load_ext tensorboard
%tensorboard --logdir /content/drive/My\ Drive/ANN&DL/classification_experiments_/ --port 6009
```

We train the model s.t. it'll save the model status at each training epoch. (So we're also able to visualize what's going on)

In [ ]:

```
import os
from datetime import datetime

cwd = '/content/drive/My Drive/ANN&DL'  # use your local directory if you are not using Drive

exps_dir = os.path.join(cwd, 'classification_experiments_')
if not os.path.exists(exps_dir):
    os.makedirs(exps_dir)

now = datetime.now().strftime('%b%d_%H-%M-%S')

exp_name = which_model

exp_dir = os.path.join(exps_dir, exp_name + '_' + str(now))
if not os.path.exists(exp_dir):
    os.makedirs(exp_dir)

callbacks = []

# Model checkpoint
# -------------------------------------------------------------------------------
ckpt_dir = os.path.join(exp_dir, 'ckpts')
if not os.path.exists(ckpt_dir):
    os.makedirs(ckpt_dir)

ckpt_callback = tf.keras.callbacks.ModelCheckpoint(filepath=os.path.join(ckpt_dir, 'cp_{epoch:02d}.ckpt'),
                                                   save_weights_only=True)  # False saves the entire model
callbacks.append(ckpt_callback)

# Visualize Learning on Tensorboard
# -------------------------------------------------------------------------------
tb_dir = os.path.join(exp_dir, 'tb_logs')
if not os.path.exists(tb_dir):
    os.makedirs(tb_dir)

# By default shows losses and metrics for both training and validation
tb_callback = tf.keras.callbacks.TensorBoard(log_dir=tb_dir,
                                             profile_batch=0,
                                             histogram_freq=1)  # if 1 shows weights histograms
callbacks.append(tb_callback)
```

This is how we save the model in an automatic way during trainings

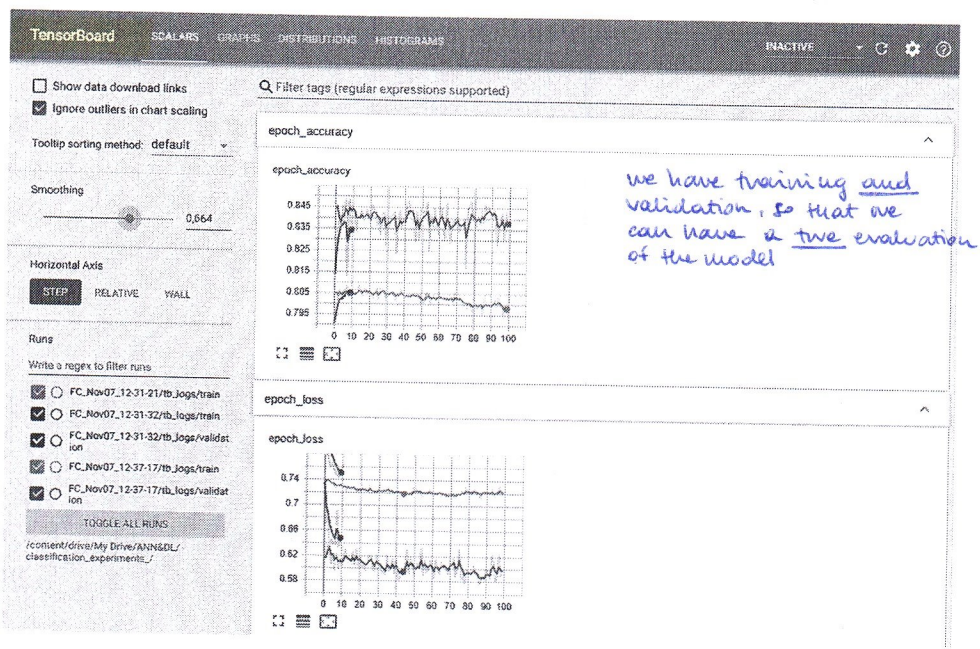
tolerance (= how many epochs we want to wait until we stop the training)

```
# Early Stopping
# ---------------------------------------------------------------------
early_stop = False
if early_stop:
    es_callback = tf.keras.callback.EarlyStopping(monitor='val_loss', patience=10)
    callbacks.append(es_callback)

# ---------------------------------------------------------------------
model.fit(x=train_dataset,
          epochs=100,  #### set repeat in training dataset
          steps_per_epoch=int(np.ceil(x_train.shape[0] / bs)),
          validation_data=valid_dataset,
          validation_steps=int(np.ceil(x_valid.shape[0] / bs)),
          callbacks=callbacks)

# How to visualize Tensorboard

# 1. tensorboard --logdir EXPERIMENTS_DIR --port PORT     <- from terminal
# 2. localhost:PORT   <- in your browser
```

if we want it ⇒ **True**

To prevent overfitting

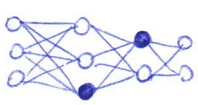
here we can specify which is the quantity we rely on for the early stopping (we can choose **ACCURACY** or **LOSS**)



we have training and validation, so that we can have a twe evaluation of the model

Reduce the overfitting:

- reduce the number of parameters
- **early stopping**
- **dropout** (based on the idea that an ensamble of methods helps in preventing overfitting ⇒ if we take the average prediction of different models we are able to obtain better results than a single model) (it's implemented by using some noise in the network. This corresponds to randomly turn off some neurons → This is implemented through: `tf.keras.layers.Dropout(p)` (that comes after the activation function), where p is the probability for each neuron to be turned off.)



- **weight decay** (we can prevent some weights to acquire higher levels. In this way we limit the model complexity. For example we can say that we want $\min \|\underline{w}\|_2^2$: how can we let it know? We add something to the loss function. We introduce also a regularization γ (useful in the case in which our initial weights are already too high) ⇒ $\text{loss} = \sum_{n=1}^{N} (t_n - g(x_n|w))^2 + \gamma \|\underline{w}\|_2$

  $\|\underline{w}\|_2 = \sum_{q=1}^{Q} (w_q)^2$

- **data augmentation**