# LAB 01 - CNNs for Image Classification

We will start by building a simple tf.keras model for image classification. Later, we should customize our model and check some of the options available to us during the training procedure.
Please make sure you have Tensorflow 2.0.0-rc2 installed, accompanied by Tensorflow Datasets.

```
pip install tensorflow==2.0.0-rc2
pip install tensorflow-datasets
```

Training time can be significantly improved by installing tensorflow-gpu. You can find the official instructions here: `https://www.tensorflow.org/install/gpu`. Please note that this procedure will take a considerable amount of time.

## Tensor operations

In the context of this framework all operations are performed on tensors. For simplicity, you can think of a tensor as a mutidimensional matrix.

Roughly, there are Tensorflow implementations for all NumPy operations and some additional ones. Operations will be presented when needed. You can find a complete list here: `https://www.tensorflow.org/api_docs/python/tf/#functions`.

A first example:

```
import tensorflow as tf
input = tf.constant([1, 4, 2, 43, 32, 3, 5, 6])
x = tf.reshape(input, [-1, 2])
```

```
a, b = tf.unstack(x, num = 2, axis = 1)
print( tf.reduce_sum(a - b) )
```

We've stated by defining a constant and performing a sequence of simple operations on it. Your first task is to print the output value of each succesive operation and give an interpretation for the final result.

In addition, starting from version 2.0, you have the option of writing normal python functions and decorate them using @tf.function. Those procedures will be automatically compiled to tensor based operations. For example, the following code achieves the same result as the one above.

```
@tf.function
def f(input):
    sum = 0
    for i in range(input.shape[0]):
        if i % 2 == 0:
            sum -= input[i]
        else:
            sum += input[i]
        # OR:
        # sum += (float(i % 2 == 0) - .5) * 2. * input[i]
    return sum
input = tf.constant([1, 4, 2, 43, 32, 3, 5, 6])
print(f(input))
```

This being said, optimal code will always be written using well thought tensor operations that make full use of the GPU.

Finally, we can wrap sets of operations using tf functions [1] or models.

[1] https://www.tensorflow.org/api_docs/python/tf/keras/backend/function

```
input = tf.constant([1, 4, 2, 43, 32, 3, 5, 6])
class CustomModel(tf.keras.models.Model):
    def call(self, input):
        x = tf.reshape(input, [-1, 2])
        a, b = tf.unstack(x, num = 2, axis = 1)
        return tf.reduce_sum(a - b)
model = CustomModel()
print(model(input))
```

We focus on models that implement neural networks but we encourage you to run some experiments with custom models or functions as individual work at home. That being said we will continue by building a simple CNN architecture.

## Classification on the CIFAR Dataset

First and foremost, load the CIFAR10 dataset using the Tensorflow Datasets library [2].

```
import math
import tensorflow_datasets as tfds
import tensorflow as tf
import tensorflow.keras.layers as layers

ds, meta = tfds.load('cifar10', as_supervised = True, with_info = True)
train_ds = ds['train']
```

This library is designed to give easy access to a sum of very popular datasets. In this scenario *ds* is a dictionary with "train" and "test" keys pointing to the corespoding sets. The *meta* variable conveys descriptive information such as: number of samples per split, input size, name of the article accompanying dataset, etc. You can iterate through samples by casting the dataset to numpy.

```
import matplotlib.pyplot as plt
for np_image, label in tfds.as_numpy(train_ds):
```

[2] https://www.tensorflow.org/datasets/overview

```
    print(label)
    plt.imshow(np_image)
    plt.show()
```

Subsequently, we should start building our model. tf.keras makes this process painless by provinding implementations for all commonly used neural network components. Implementing regular tasks usually consists only in combining those predefined components. For example, building a CNN with 2 convolutional layers, max pooling and two fully connected layers can be implemented as follows:

```
input = tf.keras.Input(shape = (32, 32, 3))
layer = input
for _ in range(2):
    layer = layers.Conv2D(64, (5, 5), activation = 'relu')(layer)
    layer = layers.MaxPool2D((2, 2))(layer)
layer = layers.Flatten()(layer)
for u, a in zip([64, 10], ['relu', 'softmax']):
    layer = layers.Dense(u, activation = a)(layer)
model = tf.keras.Model(inputs = input, outputs = layer)
```

A full list of tf.keras layers can be found here: `https://www.tensorflow.org/api_docs/python/tf/keras/layers`.

We can now specify the optimizer, loss and metrics to complete the definition of the model.

```
model.compile(
    optimizer = 'adam',
    loss = 'sparse_categorical_crossentropy',
    metrics = ['accuracy']
)
```

A lot of tf.keras components like losses, optimizers or activation functions can be speficied via a string containing their name, but you can always give the actual function as a parameter or feed a function of your own.

Model training is handled by the *fit* [3] procedure. Feeding the dataset, the number of iterations per epoch and lastly, the number of epochs, is enough to get training started. Also, now would be a proper time to mention that the dataset previously loaded is a tf.Dataset. Essentialy, it is a structure that is well optimized for data feeding on which several specific operations can be applied. All needed information can be found here: `https://www.tensorflow.org/api_docs/python/tf/data/Dataset`. The following code illustrates partitioning the train set into batches of 64 samples and running optimization for 5 epochs.

```
batch_size = 64
n_samples = meta.splits['train'].num_examples
model.fit(
    train_ds.batch(batch_size).repeat(),
    epochs = 5,
    steps_per_epoch = math.ceil(n_samples / batch_size)
)
```

**Exercise:** Create and fit a CNN on the smallnorb dataset.

**Exercise:** Add a Batch Normalization layer before each convolution activation.

## Model Evaluation

When fitting, validation can be performed at the end of each epoch. In order to do so, we should create a validation set by split the training set when loading the data and then feed the validation parameters to the training procedure.

```
(train_ds, val_ds), meta = tfds.load(
    'cifar10',
    as_supervised = True,
    with_info = True,
```

[3] `https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit`

```
    split = list(tfds.Split.TRAIN.subsplit(weighted = (8, 2)))
) # loads the training set only and then splits it
n_samples = meta.splits['train'].num_examples
model.fit(
    train_ds.batch(batch_size).repeat(),
    steps_per_epoch = math.ceil(n_samples * .8 / batch_size),
    epochs = 5,
    validation_data = val_ds.batch(batch_size),
    validation_steps = math.ceil(n_samples * .2 / batch_size),
)
```

Model saving is done via a Callback. Callbacks are tfs way of allowing the programmer to intervene in the fitting procedure and call routines at specific moments [4][5]. The following code illustrates saving the model after every epoch.

```
if not os.path.exists('./checkpoints'):
    os.makedirs('./checkpoints')
checkpoint_callback =  tf.keras.callbacks.ModelCheckpoint(
    filepath = './checkpoints/model.hdf5'
)
model.fit(
    ...,
    callbacks = [
        checkpoint_callback
    ]
)
```

We can also specify that we want to keep the version of the model that performs best on the validation data.

```
checkpoint_callback =  tf.keras.callbacks.ModelCheckpoint(
    filepath = './checkpoints/model.hdf5',
    save_best_only = True
)
```

[4] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks

[5] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback

Loading models can be done specifying the path from which we want to load the models. Subsequently we can evaluate them in any way we see fit. Cassifying a single instance is done as follows:

```
import tensorflow as tf
import cv2
import numpy as np

model = tf.keras.models.load_model('./checkpoints/model.hdf5')
im = np.float32(
    np.reshape(
        cv2.imread('./cifar_test.png'),
        [1, 32, 32, 3],
    )
) # adjusting the (32, 32) image to represent
# a batch containing a single image with a single channel
print(np.argmax(model.predict(im), axis = -1))
# the last layer (the output of the network) has shape (batch_size, 10)
# and uses softmax activation
# thus, using argmax on the last dimension gives us the predicted class
```

Finally, the following code evaluates the loaded model on the test set.

```
ds, meta = tfds.load('cifar10', as_supervised = True, with_info = True)
test_ds = ds['test']

model.evaluate(
    test_ds.batch(64)
)
```

## Loading Data

Loading locally saved data can be done by extending tf.keras.utils.Sequence; a class that does this is usually called a *generator* [6]. You can feed the generator as a replacement for

[6] https://www.tensorflow.org/api_docs/python/tf/keras/utils/Sequence

the tf.Dataset when fitting. Its implementation should contain a function that returns the number of iterations per epoch and one that feeds a batch. Optionally, you can specify additional operations to be performed at the end of each epoch.

Start by downloading the mnist dataset from our website and run the following code:

```
batch_size = 10

class Generator(tf.keras.utils.Sequence):
    def __init__(self, image_dir, batch_size):
        self.image_paths = glob.glob(image_dir + '/*')
        np.random.shuffle(self.image_paths)

        self.batch_size = batch_size

    def __len__(self):
        return math.ceil(len(self.image_paths) / self.batch_size)

    def __getitem__(self, iteration_n):
        filepaths = self.image_paths[
            batch_size * iteration_n: batch_size * (iteration_n + 1)
        ]
        return np.array([
            np.reshape(
                cv2.imread(filepath, cv2.IMREAD_GRAYSCALE),
                (28, 28, 1)
            )
            for filepath in filepaths
        ]), np.array([
            int(os.path.basename(filepath).split('_')[0])
            for filepath in filepaths
        ])


    def on_epoch_end(self):
        np.random.shuffle(self.image_paths)
```

```
model.fit(
    Generator('./mnist/train', batch_size = batch_size),
    epochs = 5,
)
```

Note we could have designed the class to load data by default from the specified folder and not feed it as a parameter. We chose to implement it as such to be able to instantiate another ganerator to feed the validation data. Also note that it is no longer neccessary to specify the steps_per_epoch parameter.

**Exercise:** Run the code without shuffling the data. Assess what is happening and find a plausible explanation for it.

Finally, image preprocessing can be performed inside you generator __getitem__ function. In the case of tf.Dataset this can be achieved by mapping a preprocessing function over the dataset. Sample usage:

```
def format_example(image, label):
    image = tf.cast(image, tf.float32)
    image = image / 255.
    image = tf.image.resize(image, (128, 128))
    return image, label
train_ds = train_ds.map(format_example)
```

## Tensorboard

Tensorboard is a tool that helps you visualize what is happening inside your model while training. The callback that adds this functionality is presented below. All data will be stored in the ./logs folder.

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir = './logs',
    histogram_freq = 0,
    batch_size = batch_size,
    write_graph = True,
    write_grads = False,
    write_images = False,
    embeddings_freq = 0,
    embeddings_layer_names =None,
    embeddings_metadata = None,
    embeddings_data = None,
    update_freq = 'batch'
)
if os.path.exists('./logs'):
    shutil.rmtree('./logs')
model.fit(
    ...,
    callbacks = [
        ...,
        tensorboard_callback
    ]
)
```

To run the example you should:

1. Start the training
2. Run the following command in a separate shell: tensorboard --logdir ./logs
3. Open localhost:6006 inside your browser

Notice that we delete the log folder before the start of each training. This prevents amalgamation between data from different runs. The tensorboard shell process should also be stopped and restarted with each training rerun.

## Defining Custom Metrics

When classifying images it is often the case that the programmer should implement a custom loss function or a custom metric. In order to feed your own functions to those parameters they should take as input y_true and y_pred and return the desired metric or loss value. The inputs are tensors and will always have the batch_size as their first dimension; the rest of them being given by the shape of the final layer.

A dummy evaluation metric that measures the mean of all maximal activations from the last layers can be implemented as such:

```
def custom_metric(y_true, y_pred):
    return tf.reduce_mean(tf.reduce_max(y_pred, axis = -1))

model.compile(
    ...,
    metrics = [custom_metric]
)
```

Defining losses can be done in the same manner, the only difference being that the function should be fed to the coresponding parameter. As a final task you have to implement a few custom components with the help of your superviser in order to become more familiar with tensor operations. Return the mean per batch for all tasks.

**Exercise:** Define a metric that measures the absolute difference between the activation coresponding to the correct class and the maximum activation from the rest of the classes.

**Exercise:** Define a metric that measures the mean absolute difference between the activation coresponding to the correct class and rest of the activations.

**Exercise:** Define loss that penalises the mean absolute difference between the activation coresponding to the correct class and rest of the activations as long as it is below 0.9.