# Deep Learning

## Practical Exercises - Week 8

Introduction to TensorFlow

ICAI

FS 2024

# 1 Introduction

In the lab of last week we learned the fundamental principles, commands and functions of TensorFlow 2 and implemented our first logistic regression classifier. This week we will implement a more interesting and advanced classifier. Specifically, we will implement a deep feed-forward network which classifies handwritten digits into ten different classes.

# 2 MNIST For ML Beginners

When learning to program, it's considered a tradition that the first program prints `"Hello World"`. Just like programming has Hello World, machine learning has MNIST. MNIST is a simple computer vision dataset. It consists of images of handwritten digits as shown in Figure 1. It also includes labels for each image, telling us which digit it is.

Figure 1: Some handwritten digits of the MNIST dataset. The labels for these images are 5, 0, 4, and 1.

In this tutorial, we're going to train a model which takes an image as input and predicts the corresponding digit class. Our goal isn't to train a really elaborate model that achieves state-of-the-art performance (we will look at this later), but rather to dip a toe into using TensorFlow 2. We will start with a very simple model, called a Softmax Regression[1].

[1] Softmax Regression is a generalization of logistic regression that can be used for multi-class classification, under the assumption that the classes are mutually exclusive.

This tutorial comprises only a few lines of code. However, it is very important to understand the ideas behind it, both how TensorFlow works and the core machine learning concepts. Therefore, we will work through the code very carefully. This lab follows closely this TensorFlow tutorial.

## 2.1 The MNIST Dataset

The MNIST data can be downloaded with the following code:

```
import tensorflow as tf

# Load
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The MNIST data is split into two parts: 60'000 data samples of training data and 10'000 samples of test data. Because we also need validation data we further split the train dataset into 55'000 training data samples and 5'000 validation data samples. Splitting the dataset into training set and validation set is crucial in machine learning. We have to make sure that the trained model is evaluated with data that were not used during training. This is the only way to obtain an unbiased estimate of the generalization error.

As mentioned earlier, every MNIST data sample consists of 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.

Each image is 28 pixels by 28 pixels. We can interpret this as a matrix of numbers as shown in Figure 2. We can flatten this array into a vector of $28 \cdot 28 = 784$ numbers. It doesn't matter how we flatten the array, as long as we're consistent. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space, with a very rich structure.

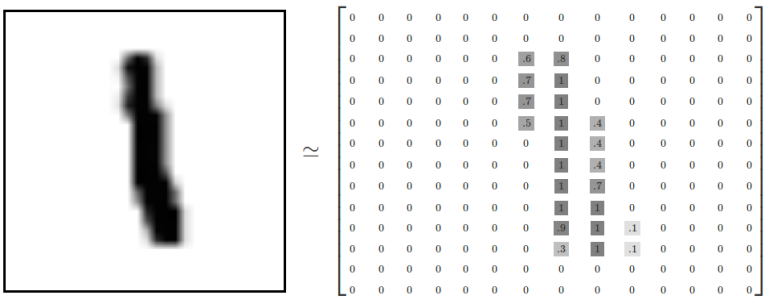

Figure 2: An image of 28 by 28 pixels, which can be interpreted as a big array of numbers.

Flattening the data throws away information about the 2D structure of the image. Isn't that bad? Well, the best computer vision methods do exploit this structure, and we will do the same in later labs. But the simple method we will be using here (a softmax regression) won't.

The result is that `x_train` is a tensor (an n-dimensional array) with a shape of `[55000, 784]` (see Figure 3). The first dimension is an index into the list of images and the second dimension is the index for each pixel in each image. Each entry in the tensor is a pixel intensity between 0 and 1, for a particular pixel in a particular image.

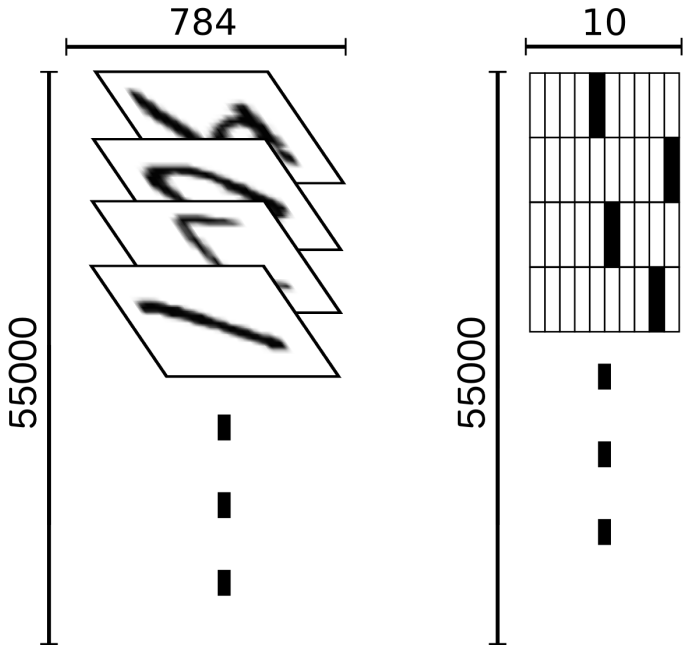


Figure 3: Graphical representation of the two tensors `x_train` (left) and `y_train` (right).

Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit drawn in the image. For the purposes of this tutorial, we want our labels to be "one-hot vectors". A one-hot vector is a vector which is 0 in most dimensions, and 1 in a single dimension. In this case, the nth digit will be represented as a vector with an 1 in the nth dimension. For example, 3 would be `[0,0,0,1,0,0,0,0,0,0]`. Consequently, `y_train` is a `[55000, 10]` matrix of floats as shown on the right of Figure 3.

We're now ready to define our model!

## 2.2 Softmax Regressions

We know that every image in MNIST is a handwritten digit between zero and nine. So the image belongs to one of ten possible classes. Our aim is to train a classifier which takes a MNIST image as an input and outputs the probability of this image belonging to one of the ten classes. For example, our model might take the image of the digit nine as an input and output this array

```
[0.004, 0.001, 0.010, 0.004, 0.120, 0.004, 0.003 , 0.078 , 0.032, 0.743]
```

Softmax regression is a suitable model for this task. If you want to assign probabilities to an object belonging to one of several different classes, softmax is the appropriate function

to apply, The softmax function outputs a list of values between 0 and 1 which add up to 1. In a later lab, we are going to take a closer look at deep feed-forward networks and convolutional neural networks. If these advanced neural networks are used as multi-class classifier they are designed with a softmax function as output layer.

A softmax regression consists of two operations: First we add up the evidence of our input being in certain classes, and then we convert that evidence into probabilities.

To tally up the evidence that a given image is in a particular class, we perform a weighted sum of the pixel intensities. The weight is negative if that pixel having a high intensity is evidence against the image being in that class, and positive if it is evidence in favor.

The diagram in Figure 4 shows these weights of a trained softmax regression model. Red represents negative weights, while blue represents positive weights.

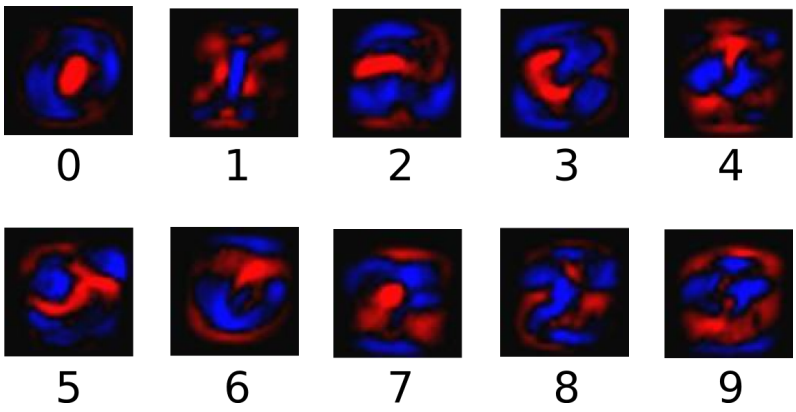


Figure 4: Weights that a model has learned to distinguish between the nine classes of handwritten digits. Red indicates negative weights, while blue indicates positive weights.

We also add some extra evidence called a bias. Thus, the evidence (pre-activation) for a class $i$ given an input $\boldsymbol{x}$ is given by:

$$a_i = \sum_j W_{i,j}\, x_j + b_i,$$

where $W_{i,j}$ and $b_i$ are the weights and bias for class $i$, and $j$ is the index of the input feature vector $\boldsymbol{x_j}$ (in our case a flattened image). In a next step, the evidence $a_i$ is converted into a probability mass function over all possible classes $\boldsymbol{y}$ using the softmax-function:

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{a}).$$

You can think of the softmax operation as a function which converts the evidence of a particular class into a probability of a particular class. It's defined as:

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{a}) = \text{normalize}(\exp(\boldsymbol{a})).$$

If you expand that equation out, you get:

$$y_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$$

Thus, to calculate the probabilities we first apply the exponential function to every $a_i$ value (which makes it strictly positive) and then divide by the sum over all $\exp(a_i)$ values (to normalize the output).

Figure 5 shows an illustration of the softmax regression model. For each output $y_i$, we compute a weighted sum of the inputs $x_j$, add a bias $b_i$, and then apply the softmax-function. If we write that out as equations, we get:

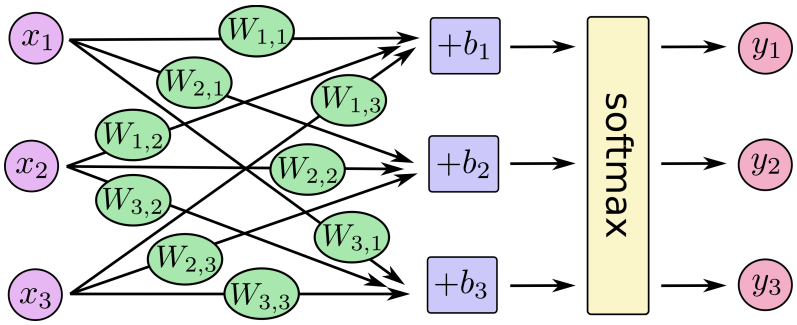


Figure 5: Graphical representation of the softmax regression model.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left( \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \right)$$

We can "vectorize" this calculation and represent it as a matrix multiplication and vector addition which makes it computationally more efficient.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left( \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

More compactly, this can can be written as:

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}).$$

Now let's implement these equations with TensorFlow.

## 2.3 Implementing the Softmax Regression

In this section, we apply softmax regression to the MNIST data. The full code is in the script `mnist_softmax.py`.

In a first step let's import all needed packages, load and prepare the MNIST dataset.

```
import numpy as np
import tensorflow as tf

# Load MNIST data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The images come in matrix form of 28x28 `uint8` values. Moreover, there exists no validation data split. Therefore, we first convert the images into arrays of shape `(784,)` and datatype `float32` whose values lie between 0.0 and 1.0. Additionally, we split off a validation dataset from the training dataset.

```
# Scale images
x_train = x_train / 255.0
x_test  = x_test / 255.0

# Flatten images
x_train = x_train.reshape([len(x_train), -1]).astype("float32")
x_test  = x_test.reshape([len(x_test), -1]).astype("float32")

# Split off validation dataset from training dataset
indices = np.random.choice(len(y_train), 5000, replace=False)
x_valid = x_train[indices, :]
y_valid = y_train[indices]
x_train = np.delete(x_train, indices, axis=0)
y_train = np.delete(y_train, indices, axis=0)
```

Next we have to convert the image labels (`y_train`, `y_valid` and `y_test`) into one-hot vectors as described in subsection 2.1. All three splits are then converted to a `tf.data.Dataset` which allows us to apply some preprocessing operations (such as batching and shuffling) and lets us efficiently loop over the dataset split. For more information about TensorFlow input pipelines see here.

```
# Convert labels to one-hot tensor
y_train = tf.one_hot(y_train, 10)
y_test  = tf.one_hot(y_test, 10)
y_valid  = tf.one_hot(y_valid, 10)

# Create datasets
BATCH_SIZE = 32
train_ds = (
    tf.data.Dataset.from_tensor_slices((x_train, y_train))
    .shuffle(len(y_train), reshuffle_each_iteration=True)
    .batch(BATCH_SIZE)
)
valid_ds = tf.data.Dataset.from_tensor_slices((x_valid, y_valid)).batch(BATCH_SIZE)
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(BATCH_SIZE)
```

As you can see in the above code section, here we use a mini-batch size of 32.

Now we finally get to build our softmax regression model. In contrast to TensorFlow 1, in TensorFlow 2 it is very simple to define a model. In fact, there are many different ways to define a model in TensorFlow 2 (see here for more information). Here we will use the subclassing method, whereby we inherit from the class `tf.keras.Model`.

```
# Create model
class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.d1 = tf.keras.layers.Dense(10, use_bias=True)
        self.s1 = tf.keras.layers.Softmax()

    def call(self, x):
        x = self.d1(x)
        return self.s1(x)

# Create an instance of the model
model = MyModel()
```

Thanks to this inheritance concept we can use the methods of the parent class `tf.keras.Model` (you might know this concept from Python programming). In the class constructor `__init__()` we define all the layers which we need in order to build the model. In case of softmax regression we need a `Dense` layer (for the matix multiplication) and a `Softmax` layer (for probability conversion). These methods are then invoked in the correct order in the `call` method of the same class. Next we instantiate an object of this child class. Note that `call( )` is an overridden parent class method which is called when the instance `model` is called.

Next we need to define a loss function (cross-entropy in our case) and a way to minimize this loss function (stochastic gradient descent with a learning-rate of 0.01 in our case). Furthermore, we will instantiate two metric objects for each dataset split, which lets us easily accumulate the results over all iterations.

```
# Choose an optimizer and loss function for training:
loss_object = tf.keras.losses.CategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

# Select metrics to measure the loss and the accuracy of the model.
# These metrics accumulate the values over epochs
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.CategoricalAccuracy(name='train_accuracy')

valid_loss = tf.keras.metrics.Mean(name='valid_loss')
valid_accuracy = tf.keras.metrics.CategoricalAccuracy(name='valid_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.CategoricalAccuracy(name='test_accuracy')
```

The training, validation and testing process is performed with mini-batches of 32 data samples. In a next step we have to specify the operations which need to be carried out on such a mini-batch.

A training step looks similar to what we've seen in last week's lab. We have to calculate the gradients which are subtracted from all the trainable variables of our model. Additionally, we accumulate the loss and accuracy value of each training step by calling the `train_loss` and `train_accuracy` object respectively.

In a validation and testing step we simply calculate the loss and accuracy and accumulate the values.

```
# Use tf.GradientTape to train the model
@tf.function
def train_step(images, labels):
    with tf.GradientTape() as tape:
        # training=True is only needed if there are layers with different
        # behavior during training versus inference (e.g. Dropout).
        predictions = model(images, training=True)
        loss = loss_object(labels, predictions)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)
    train_accuracy(labels, predictions)


# Validate the model
@tf.function
def valid_step(images, labels):
    # training=False is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=False)
    loss = loss_object(labels, predictions)

    valid_loss(loss)
    valid_accuracy(labels, predictions)


# Test the model
@tf.function
def test_step(images, labels):
    # training=False is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=False)
    loss = loss_object(labels, predictions)

    test_loss(loss)
    test_accuracy(labels, predictions)
```

Now we finally get to train, validate and test our model. Specifically, we train and validate our model for 25 epochs (in each epoch we run over the whole training dataset and validation dataset). At the end we test whether our model also generalizes to previously unseen data points by evaluating the loss and accuracy of the test dataset.

```
EPOCHS = 25
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_state()
    train_accuracy.reset_state()
    valid_loss.reset_state()
    valid_accuracy.reset_state()

    for images, labels in train_ds:
        train_step(images, labels)

    for valid_images, valid_labels in valid_ds:
        valid_step(valid_images, valid_labels)
```

```
    print(
        "Epoch {:2d}: ".format(epoch + 1),
        "Train Loss: {:3.3f}, ".format(train_loss.result()),
        "Train Accuracy: {:3.3f}%, ".format(train_accuracy.result() * 100),
        "Validation Loss: {:3.3f}, ".format(valid_loss.result()),
        "Validation Accuracy: {:3.3f}%".format(valid_accuracy.result() * 100),
    )

# Test resulting classifier
test_loss.reset_state()
test_accuracy.reset_state()
for test_images, test_labels in test_ds:
    test_step(test_images, test_labels)

print(
    "\nTesting result: ",
    "Test Loss: {:3.3f}, ".format(test_loss.result()),
    "Test Accuracy: {:3.3f}%".format(test_accuracy.result() * 100),
)
```

The test accuracy should be about 92%. Is that good? Well, not really. In fact, it's pretty bad. This is because we're using a very simple model. The best models can get to over 99.7% accuracy! However, what matters is what we learned from the implementation of this model.

**Disclaimer:** Please note that there also exists a simpler way to train a model than explained above (see here for more details). However, unlike the method above, you cannot customize the training step.

## 2.4 Exercise

This whole training, validation and testing procedure is implemented in the module `mnist_softmax.py`. Modify the softmax regression model and add a few more layers. Specifically, add another Dense layer and add a ReLU layer after the first Dense layer. This increases the capacity of the model and the insertion of the nonlinear ReLU layer lets the model also learn nonlinear decision boundaries.

By how much does the test accuracy increase? Can you further improve this result by inserting even more layers?