Deep Learning

Practical Exercises - Week 7

Introduction to TensorFlow

ICAI

FS 2024

1 Choosing a Deep Learning Framework

Deep Learning is a field of computer science that is experiencing a lot of progress right now. Therefore, there exist many different frameworks for different programming languages. The most popular frameworks include TensorFlow, Keras, Pytorch and Caffe.

Some frameworks are quite easy to use, while others require a deeper understanding of the underlying computational graphs. In this course, we will use the TensorFlow 2 framework. TensorFlow is an open-source platform that allows efficient training and execution of machine learning models. It comes with multiple levels of abstraction. This means you can use high-level APIs like Keras to make things a little easier.

2 Topic

In the next three lab sessions we will introduce the concepts of TensorFlow and TensorBoard. For this introduction we work with slightly modified versions of the TensorFlow tutorials from here. Also test the commands provided, as this is a good way to get familiar with the library.

In lab 7, we start with the basic concepts of TensorFlow. Lab 8 will delve into more advanced topics, and in lab 9, we will introduce TensorBoard (a tool to visualize the graph and the learning process).

3 Getting Started With TensorFlow

If you have not installed TensorFlow yet, please follow the instructions under this link.

This guide gets you started programming in TensorFlow. It begins with a tutorial on the lowest level API (TensorFlow Core), which contains the most important building blocks.

3.1 Importing TensorFlow

The canonical import statement for TensorFlow programs is as follows:

```
import tensorflow as tf
```

If `import tensorflow as tf` does not work, Tensorflow is not installed correctly. To resolve this issue, see build and install error messages.
Check your TensorFlow version with the following command

```
tf.__version__
```

```
'2.3.0'
```

Moreover, make sure that Eager Execution is active

```
tf.executing_eagerly()
```

```
True
```

3.2 Tensors

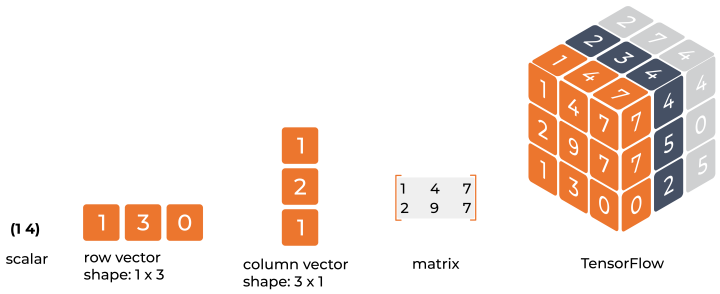


Figure 1: Visualization of tensors with different dimensionality

The **tensor** is a central data unit which is optimized for TensorFlow's dataflow graph. This allows very fast tensor multiplications and calculation of gradients. A tensor consists of a set of primitive values of a specific type, shaped into an array of any number of dimensions. Here you can see the complete list of TensorFlow data types. A tensor's rank is its number of dimensions. Figure 1 visualizes some examples.

3.2.1 tf.constant

Constants are tensors which are initialized directly and are immutable once created. While in TensorFlow 1 they had to be run in a `Session` in order to evaluate them, in TensorFlow 2 they can be displayed using eager execution by default.
If the argument dtype is not specified, then the type is inferred from the type of value.

```
# int32 tensor by default
rank_0_tensor = tf.constant(4)

# tensors can also have names (in the computational graph)
# float32 tensor by default
named_tensor = tf.constant(7.2, name='my_named_tuple')

# float32 tensor of rank 1
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])

# tensors can be n-arrays, and we can specify the data type
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=tf.float16)
```

The commands

```
print(rank_0_tensor.numpy())
print(named_tensor)
print(rank_1_tensor)
print(rank_2_tensor)
```

result in

```
4
tf.Tensor(7.2, shape=(), dtype=float32)
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
tf.Tensor(
[[1. 2.]
[3. 4.]
[5. 6.]], shape=(3, 2), dtype=float16)
```

A constant is an immutable object and therefore can not be changed after it is created.

3.2.2 tf.Variable

A variable is the recommended way to represent data manipulated by your program. Variables are usually weights and biases of a model that are optimized during training. The following covers how to create, update and manage instances of `tf.Variable`.
Remark: The lowercase c in tf.constant() indicates that tf.constant is an operation, while the capital V in tf.Variable indicates that it is a class with many operations.

```
# Create variable tensor
var = tf.Variable(3.)
print(var)
```

```
# Reassign the value of a Variable
var.assign(4)
print(var.numpy())
```

This code fragment produces the following output

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=4.0>
4.0
```

The following commands display some tensor properties

```
# Pass tf.constant to initialize tf.Variable
my_tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
my_var = tf.Variable(my_tensor)
%
# view properties of tensor
print('name  : ', my_var.name)
print('type  : ', my_var.dtype)
print('shape : ', my_var.shape)
print('device: ', my_var.device)
```

```
name  :  Variable:0
type  :  <dtype: 'float32'>
shape :  (2, 2)
device:  /job:localhost/replica:0/task:0/device:GPU:0
```

3.2.3 Variable Assignment

Calling `assign( )` only works if the shape of the newly assigned tensor is compatible with the shape of assigned variable.

```
d = tf.Variable([4.0, 5.5])
# This will keep the same dtype, float32
d.assign([10, 20])

# Fails, as it resizes the variable
try:
   d.assign([1.0, 2.0, 3.0])
except Exception as exc:
   print(f"{type(exc).__name__}: {exc}")
```

```
ValueError: Shapes (2,) and (3,) are incompatible
```

Incrementing or decrementing variable values works as follows

```
# Inplace increase/decrease variable values
f = tf.Variable(3.)
print('original value: ', f.numpy())
print('add 1:',           f.assign_add(1.).numpy())
print('subtract 5:',      f.assign_sub(5.).numpy())

# Same but for arrays
g = tf.Variable([3.0, 5.2])
print("\nIncrease/Decrease values in arrays")
print('original value: ', g.numpy())
```

```
print('add [1,4]: ',      g.assign_add([1,4]).numpy())
print('subtract [3,5]',   g.assign_sub([3,5]).numpy())
```

This produces the following output

```
original value:  3.0
add 1: 4.0
subtract 5: -1.0

Increase/Decrease values in arrays
original value:  [3.  5.2]
add [1,4]:  [4.  9.2]
subtract [3,5] [1.  4.2]
```

3.2.4 Placing Variables and Constants

For better performance, TensorFlow will attempt to place tensors and variables on the fastest device compatible with its dtype. This means most variables are placed on a GPU if one is available. However, this too is a process that can be overriden if desired. The following shows an example of such an instance. In particular, a float variable is placed on a CPU even if a GPU is available.

```
with tf.device('CPU:0'):
   # create tensor
   my_var = tf.Variable(my_tensor)

print(my_var.device)
```

```
/job:localhost/replica:0/task:0/device:CPU:0
```

Placing variables is an important aspect of Tensorflow. This is because it is possible to set the location of a variable or tensor on one device, yet do the computation on another device. This may introduce delay when devices communicate with one-another, but should you want to have multiple GPU workers yet only one copy of the variables this approach is worth considering. See automatic distribution for details.

```
tf.debugging.set_log_device_placement(True)
with tf.device('CPU:0'):
   l = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
   m = tf.Variable([[1.0, 2.0, 3.0]])
with tf.device('GPU:0'):
   # Element-wise multiply
   n = l * m
print(n)
```

```
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mul in device /job:localhost/replica:0/task:0/device:GPU:0
tf.Tensor(
[[ 1.  4.  9.]
[ 4. 10. 18.]], shape=(2, 3), dtype=float32)
```

3.2.5 Higher-dimensional Tensors

In the area of deep learning we often work with higher-dimensional tensors of rank three, four or even higher. For example, if a neural network is trained with color images, the training is performed with batches of three-dimensional images which results in a tensor of rank four. If the temporal axis is also taken into account (video stream) this results in a tensor of rank five. When debugging, it is important to have an understanding of the tensor axes and their meaning.

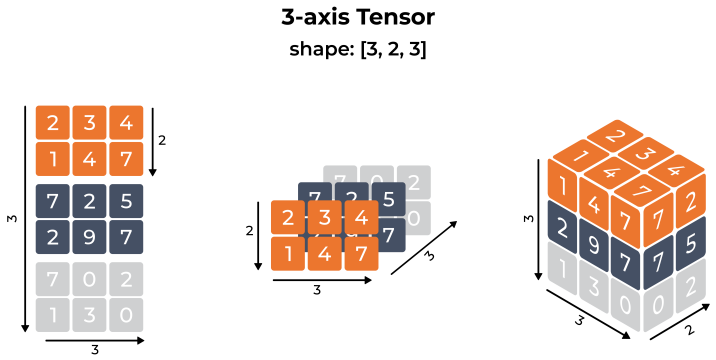


Figure 2: 3D (Rank-3) Tensor

The definition of multi-dimensional tensors is identical to the definition of a multi-dimensional NumPy array.

```
rank_3_tensor = tf.constant([[[2,3,4],
                              [1,4,7]],
                             [[7,2,5],
                              [2,9,7]],
                             [[7,0,2],
                              [1,3,0]]])
print(rank_3_tensor)
```

```
tf.Tensor(
[[[2 3 4]
[1 4 7]]

[[7 2 5]
[2 9 7]]

[[7 0 2]
[1 3 0]]], shape=(3, 2, 3), dtype=int32)
```

Exercise: To test your understanding of tensors, create the tensor below. For simplicity, set all unknown entries to zero.

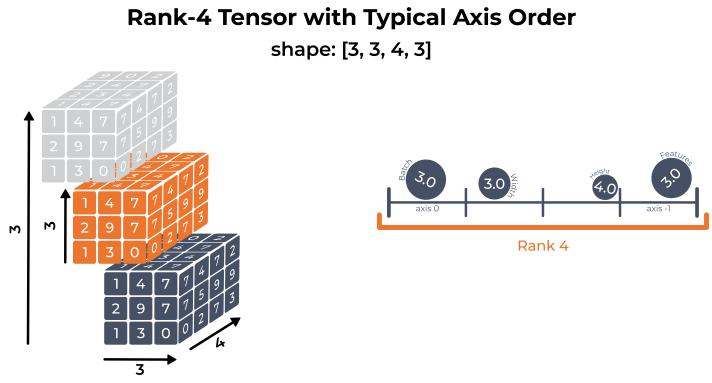


Figure 3: 4D (Rank-4) Tensor

3.3 TensorFlow Operations

NumPY	TensorFlow
`a = np.zeros((2,2));` `b = np.ones((2,2))`	`a = tf.zeros((2,2));` `b = tf.ones((2,2))`
`np.sum(b, axis=1)`	`tf.reduce_sum(a, reduction_indices=[1])`
`a.shape`	`a.get_shape()`
`np.reshape(a,(1,4))`	`tf.reshape(a,(1,4))`
`b * 5 + 1`	`b * 5 + 1`
`np.dot(a,b)`	`tf.matmul(a,b)`
`a[0,0], a[:,0], a[0,:]`	`a[0,0], a[:,0], a[0,:]`

Figure 4: Comparison Numpy and TensorFlow operations

Basic Arithmetic on Tensors

For almost any NumPy operation / function there also exists a TensorFlow counterpart for tensors (Figure 4 shows a selection). The following presents the most important arithmetic operators as well as their overloaded versions. For a comprehensive list of possible mathematical operations see here.

```
# create tensors
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[10, 20],
                 [30, 40]])
```

```
# Addition
c = tf.add(a, b)

# Alternatively, element-wise addition using operator overloading
c = a + b
```

```
# Element-wise multiplication
c = tf.multiply(a, b)
# Alternatively, element-wise multiplication using operator overloading
c = a * b

# Matrix multiplication
c = tf.matmul(a, b)
# Alternatively, Matrix multiplication
c = a @ b
```

3.3.1 Indexing and Slicing a Tensor

TensorFlow follows standard Python indexing rules. For details see: Indexing Tensors.
Note: Integer indexing removes the dimension, while range indexing (i.e. slice) keeps the dimension.

```
my_tensor = tf.constant([0, 1, 2, 3, 5, 6, 7, 8, 10, 20, 30, 40])
# indexing with a sclar removes the dimension
print("First:", my_tensor[0].numpy())
print("Second:", my_tensor[1].numpy())
print("Last:", my_tensor[-1].numpy())
```

```
First: 0
Second: 1
Last: 40
```

```
print("Everything:", my_tensor[:].numpy())
print("Before 4:", my_tensor[:4].numpy())
print("From 4 to the end:", my_tensor[4:].numpy())
print("From 2, before 8:", my_tensor[2:8].numpy())
print("Every other item:", my_tensor[::2].numpy())
print("Reversed:", my_tensor[::-1].numpy())
```

```
Everything: [ 0  1  2  3  5  6  7  8 10 20 30 40]
Before 4: [0 1 2 3]
From 4 to the end: [ 5  6  7  8 10 20 30 40]
From 2, before 8: [2 3 5 6 7 8]
Every other item: [ 0  2  5  7 10 30]
Reversed: [40 30 20 10  8  7  6  5  3  2  1  0]
```

Also multi-axis indexing and slicing follows the standard Python rules and therefore works exactly the same as with a NumPy array.

```
rank_2_tensor = tf.constant([[1., 2.],
                             [3., 4.],
                             [5., 6.]])

# Get row and column tensors using combination of indexing and slices
print("Second row:", rank_2_tensor[1, :].numpy())
print("Second column:", rank_2_tensor[:, 1].numpy())
print("Last row:", rank_2_tensor[-1, :].numpy())
print("First item in last column:", rank_2_tensor[0, -1].numpy())
print("Skip the first row:")
print(rank_2_tensor[1:, :].numpy(), "\n")
```

```
Second row: [3. 4.]
Second column: [2. 4. 6.]
Last row: [5. 6.]
First item in last column: 2.0
Skip the first row:
[[3. 4.]
[5. 6.]]
```

3.3.2 Manipulating Tensor Shapes

TensorFlow also provides similar functions as NumPy in terms of shape manipulation. A selection of the most important functions is provided below.

```
# shape returns a `TensorShape` object that shows the size on each dimension
var_shape = tf.Variable(tf.zeros((4,3,2,1)))
print('Original shape: ', var_shape.shape)

# exchange two axes
var_shape = tf.transpose(var_shape, perm=[0, 2, 1, 3])
print('Exchanged dims: ', var_shape.shape)

# expand dimension
var_shape = tf.expand_dims(var_shape, axis=0)
print('Extended dims:  ', var_shape.shape)

# minimize number of dimensions
var_shape = tf.squeeze(var_shape)
print('Squeezed dims:  ', var_shape.shape)
```

```
Original shape:  (4, 3, 2, 1)
Exchanged dims:  (4, 2, 3, 1)
Extended dims:   (1, 4, 2, 3, 1)
Squeezed dims:   (4, 2, 3)
```

3.3.3 Broadcasting Tensors

Broadcasting in TensorFlow is borrowed directly from the notion of NumPy array broadcasting. The general idea is that, under certain conditions, a smaller tensor can be 'stretched' to fit larger tensors when running combined operations on both. Following are some examples of this concept.

```
x = tf.constant([[1, 3, 5],
                 [7, 9, 11]])
y = tf.constant(3)
z = tf.constant([3, 3, 3])

# Same results
print(tf.multiply(x, 3))
print(x * y)
print(x * z)
```

```
tf.Tensor(
[[ 3  9 15]
[21 27 33]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 3  9 15]
[21 27 33]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 3  9 15]
[21 27 33]], shape=(2, 3), dtype=int32)
```

If you really understand how broadcasting works you are able to predict the output of the following commands

```
# To really grasp the broadcasting idea
x = tf.reshape(x,[-1,1])
y = tf.range(1, 5)
print(x, "\n")
print(y, "\n")
print(tf.multiply(x, y))
```

3.3.4 Ragged, String, and Sparse Tensors (Optional)

TensorFlow has several ways of dealing with 'abnormal' tensors. In particular:

- Ragged Tensors - variable number of element along some axis.
- String Tensors - atomic dtype that cannot be indexed similar to Python strings.
- Sparse Tensors - for handling sparse data, like a very wide embedding space.

3.4 Tensorflow Graphs and Executions

TensorFlow uses a dataflow graphs (see Figure 5) to represent your computation in terms of the dependencies between individual operations. In a dataflow graph, the nodes represent units of computation, and the edges represent the data consumed or produced by a computation. For example, in a TensorFlow graph, the `tf.multiply` operation would correspond to a single node with two incoming edges (the matrices to be multiplied) and one outgoing edge (the result of the multiplication).

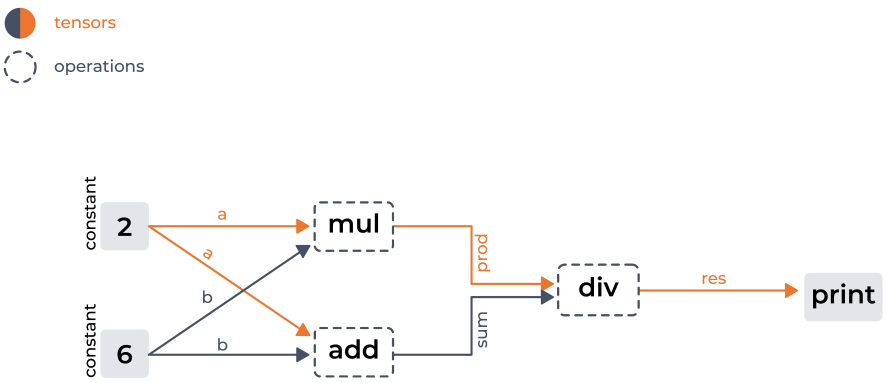


Figure 5: Representation of a dataflow graph

In TensorFlow 1.x a dataflow graph was created by explicitly defining the mathematical operations and then invoking the `run` method of a `Session` object.

In TensorFlow 2.x, you can decorate Python functions using `tf.function` to mark them for just in time (JIT) compilation. This change is done to make TensorFlow more Pythonic which enables eager execution by default, encourages the encapsulation of graph computations as Python functions, and aligns the state in the TensorFlow runtime with the state in the Python program. All that to say, functions, not sessions in TF 2.x:

```
# TensorFlow 1.x
outputs = session.run(f(placeholder), feed_dict={placeholder: input})
# TensorFlow 2.x
outputs = f(input)
```

Using graphs directly is deprecated in TensorFlow 2.x.

3.4.1 TensorFlow Function – @tf.function

The `@tf.function` decorator indicates that the function is to be transformed into a Python-independent dataflow graph. Use the tf.function to get performant and portable models, but note that tf.function is not a one-size-fits-all solution for faster computation. For more on common issues you may enconter when using tf.function and how to deal with them, see here. Now let's get to some TF functions.

```
# example function
@tf.function
def f(x,y):
    return x ** 2 + y

x = tf.constant([2, 4])
y = tf.constant([4, -2])

f(x,y)
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 8, 14], dtype=int32)>
```

A function you define is just like a core TensorFlow operation: You can execute it eagerly; you can compute gradients; and so on.

Keep the following recommendations in mind when working with `tf.function`:

- Debug in eager mode, then decorate with `@tf.function`. Alternatively, you can globally disable `@tf.function` by first calling `tf.config.run_functions_eagerly(True)`.
- Do not rely on Python side effects like list appends.
- `tf.function` works best with TensorFlow ops; NumPy and Python calls are converted to constants.

TRACING

To create a dataflow graph from a `tf.function` decorated Python function, static dtypes and shape dimensions are required. Based on the given inputs, the appropriate graph is selected (if function has already been called with the corresponding parameters before) or retraced (if function is called for the first time with the corresponding parameters). Once you understand why and when tracing happens, it's much easier to use `tf.function` effectively!

```
@tf.function
def double(a):
    print("Tracing with", a)
    return a + a

print(double(tf.constant(1)))
print()
print(double(tf.constant(1.1)))
print()
print(double(tf.constant("a")))
print()
# This doesn't print 'Tracing with ...'
print(double(tf.constant("b")))
```

```
Tracing with Tensor("a:0", shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)

Tracing with Tensor("a:0", shape=(), dtype=float32)
tf.Tensor(2.2, shape=(), dtype=float32)

Tracing with Tensor("a:0", shape=(), dtype=string)
tf.Tensor(b'aa', shape=(), dtype=string)

tf.Tensor(b'bb', shape=(), dtype=string)
```

Note that if you repeatedly call a Function with the same argument type and shape, TensorFlow will reuse a previously traced graph, as the generated graph would be identical (last instance of `double( )` function in the above example).

Retracing ensures that TensorFlow generates correct graphs for each set of inputs. However, tracing is an expensive operation! If your function retraces a new graph for every call, you'll find that your code executes more slowly than if you didn't use `tf.function`.

When tracking down issues that only appear within tf.function, here are some tips:

- Plain old Python print calls only execute during tracing, helping you track down when your function gets (re)traced.
- tf.print calls will execute every time, and can help you track down intermediate values during execution.

To control the tracing behavior, you can specify the input signature in `tf.function`.

INPUT SIGNATURE

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None], dtype=tf.int32),))
def g(x):
    print('Tracing with', x)
    return x

# No retrace!
print(g(tf.constant([1, 2, 3])))
print(g(tf.constant([1, 2, 3, 4, 5])))

# Error!
print(g(tf.constant([1.0, 2.0, 3.0])))
```

```
Tracing with Tensor("x:0", shape=(None,), dtype=int32)
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
tf.Tensor([1 2 3 4 5], shape=(5,), dtype=int32)
ValueError: Python inputs incompatible with input_signature.
```

This input signature ensures that the function `g( )` is only invoked with a 1-dimensional `int32` tensor. However, the `[None]` dimension allows the traced graph to be reused for variably sized 1-dimensional tensors. This is useful to avoid creating multiple graphs when tensors have dynamic shapes, or to restrict the shape and datatype of tensors that can be used.

Python arguments are given a special treatment in a TF function. Starting with TensorFlow 2.3, Python arguments are constrained to take the value set during tracing. In the following example you can see that the function `pow( )` is retraced when called with a different value.

PYTHON ARGUMENTS

```
# example function
@tf.function
def pow(a, b):
    print('Tracing with ', b)
    return a ** b

# pow is traced for the first time
print(pow(tf.constant(2), 2))

# pow is retraced
print(pow(tf.constant(2), 3))
```

```
Tracing with  2
tf.Tensor(4, shape=(), dtype=int32)
Tracing with  3
tf.Tensor(8, shape=(), dtype=int32)
```

Thus, it is best if you only work with TensorFlow objects and operations unless you know exactly what you are doing.

PYTHON SIDE EFFECTS

Python side effects like printing, appending to lists, and mutating globals only happen the first time you call a function with a set of inputs. Afterwards, the traced `tf.Graph` is reexecuted, without executing the Python code.
The general rule of thumb is to only use Python side effects to debug your traces. Otherwise, TensorFlow ops like tf.Variable.assign, tf.print, and tf.summary are the best way to ensure your code will be traced and executed by the TensorFlow runtime with each call.

OPERATION TRANSFORMATIONS

A subset of Python code is automatically transformed into graph-compatible TensorFlow ops. This includes control flow like `if`, `for` and `while`.
TensorFlow will convert some `if` <condition> statements into the equivalent `tf.cond` calls. This substitution is made if <condition> is a Tensor. Otherwise, the if statement is executed as a Python conditional, which means that only one branch is added to the graph.
TensorFlow will convert some `for` and `while` statements into the equivalent TensorFlow looping ops, like `tf.while_loop`. If not converted, the `for` or `while` loop is executed as a Python loop, which means that additional ops are added for each iteration of the loop.

FINAL COMMENTS

When using TensorFlow 2.x, it is recommended that users refactor their code into smaller functions that are called as needed. Moreover, it's not necessary to decorate each of these smaller functions with tf.function; only use tf.function to decorate high-level computations, for example, one step of training or the forward pass of your model.

3.5 Linear Regression Example

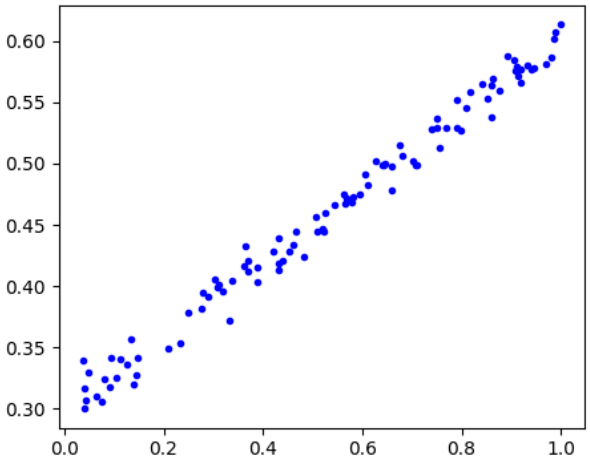
In the following example we are trying to make use of everything we have learned so far. A simple linear regression model is trained by applying a gradient descent method. We will use `tf.GradientTape( )` to compute the gradient of the loss function with respect to the trainable variables. GradientTape is considered an advanced TensorFlow feature, and is an extremely powerful way of custom differentiation.

In a first step we define a function which creates noisy dataset samples.

```
# create a noisy dataset that resembles y=mx+b+some_noise
def noisy_data(m=0.3, b=.3, n=100):
    x = tf.random.uniform(shape=(n,))
    some_noise = tf.random.normal(shape=(len(x),), stddev=0.01)
    y = m * x + b + some_noise
    return x, y

# create training dataset
x_train, y_train = noisy_data()

# visualize the data as scatter plot
import matplotlib.pyplot as plt
plt.plot(x_train, y_train, 'b.')
plt.show()
```



In a next step we define the trainable variables `m` and `b`, the linear regression model `predict_response` and the loss function `squared_error`.

```
# declare trainable variables
m = tf.Variable(0.)
b = tf.Variable(0.)

# prediction model
@tf.function
def predict_response(x):
    y = m * x + b
    return y

# loss function
@tf.function
def squared_error(y_pred, y_true):
    return tf.reduce_mean(tf.square(y_pred - y_true))

# compute loss prior to training
y_pred = predict_response(x_train)
loss = squared_error(y_pred y_train)
print(f"Loss prior to training: {loss.numpy():.6f}")
```

```
Loss prior to training: 0.2136491
```

Now we want to adjust the trainable variables `m` and `b` to make the linear regression model fit the data as good as possible. Therefore, we compute the gradient of the loss function with respect to the two parameters using tf.GradientTape() and then subtract a fraction of the gradient from the parameters in each training step. As a result, the loss is minimized.

```
# gradient descent from scratch
learning_rate = 0.05
steps = 201

for i in range(steps):
    with tf.GradientTape() as tape:
        y_pred = predict_response(x_train)
        loss = squared_error(y_pred, y_train)
```

```
    # compute gradient of loss with respect to m and b
    gradient = tape.gradient(loss, [m, b])

    # move a small step in the direction of the negative gradient
    m.assign_sub(gradient[0] * learning_rate)
    b.assign_sub(gradient[1] * learning_rate)

    # track steps
    if i % 50 == 0:
        print(f"Step number {i}: loss = {loss.numpy():.6f}")

print(f"\nGradient after {steps} steps:", gradient)
```

```
Step number 0: loss = 0.213649
Step number 50: loss = 0.000517
Step number 100: loss = 0.000332
Step number 150: loss = 0.000227
Step number 200: loss = 0.000176

Gradient after 201 steps: [<tf.Tensor: shape=(), dtype=float32,
numpy=-0.0037134492>, <tf.Tensor: shape=(), dtype=float32, numpy=0.002074287>]
```

3.6 Exercise

Modify the above example to fit a simple logistic regression model to the following dataset:

```
# create a noisy classification dataset
def noisy_data(n=100):
    x = tf.random.uniform(shape=(n,))
    y = tf.convert_to_tensor(np.random.normal(x, 0.1) > 0.5, dtype=tf.float32)
    return x, y

# create training dataset
x_train, y_train = noisy_data()
```

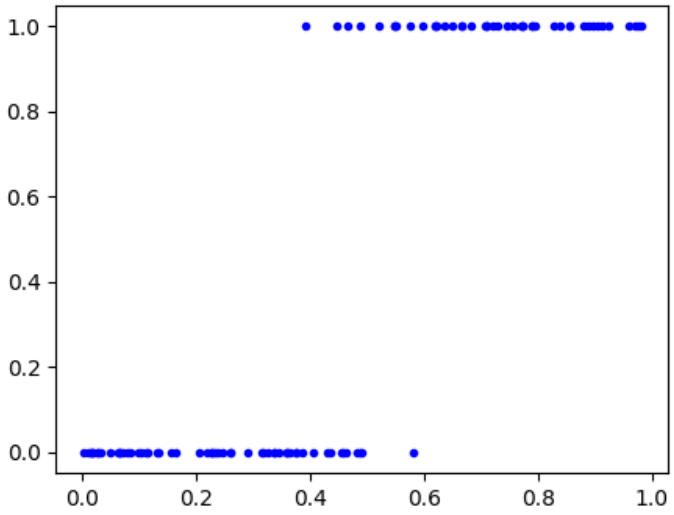


Figure 6: Classification dataset

Use the script log_reg.py which contains the data generating function.

Hint: Use the tf.keras.losses.BinaryCrossentropy(from_logits=True) class as loss for the training process.
A logistic regression model looks as follows

$$P(Y = 1|X = x) = \frac{1}{1 + e^{-(m \cdot x + b)}}$$

where m and b are the trainable parameters.

3.7 Additional Content and Sources

- TensorFlow installation guide
- TensorFlow tutorials
- TensorFlow input data pipeline The data input pipeline is an important part of a TensorFlow script and can slow down the training process if implemented incorrectly.