TensorFlow

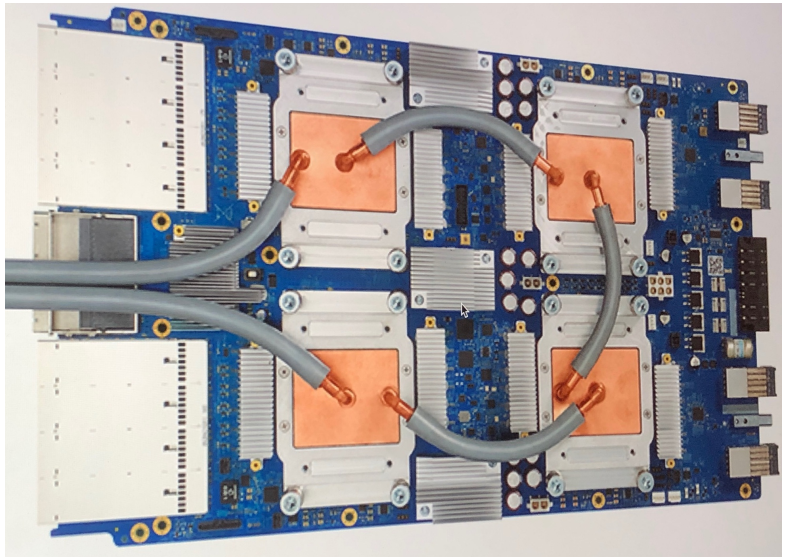
Introduction

First, we need to understand the dimensionality of a tensor

Scalar:	1 x 1	Tensor rank: 0
Vector:	N x 1	Tensor rank: 1
Matrix:	N x N	Tensor rank: 2
Tensor:	It’s the next step	Generalization of the concept

Introduction

- TensorFlow utilizes both CPU and GPU automatically
  This is fundamental!
- Recently, Google introduced **TPU** (**Tensor Processing Unit**)
  This enhances performance even further
- TensorFlow is the best choice for ANN



- Google has used TPUs for **Google Street View** text processing, finding all the text in the Street View database in less than five days
- In **Google Photos**, an individual TPU can process over 100 million photos a day
- It is also used in **RankBrain** which Google uses to provide search results

Compared to a graphics processing unit (GPU), it is designed for a high volume of low precision computation (as little as 8-bit precision) with more input/output operations per joule, and lacks hardware for texture mapping

Introduction

To simplify the horrible mess that TensorFlow 1 was, TensorFlow 2 now integrates **Keras**

- **Keras** is an open-source high-level package used as interface for TensorFlow instead of a separate library

- **Keras** was already widely adopted

  Google followed the path of "*TF2 is basically Keras! Love us too as you love Keras*"

  - TF2 is like an updated, more versatile version of Keras

Let’s “machine learning” with TensorFlow

TensorFlow coding

The example showed here is similar to the previous one

➢ However, using Numpy we would need around 100 LOC, while with TF2 we just need around 20!

- Let's start importing the libraries

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

Generating random data

Very similar to the previous example, but one line

```
observations_nr = 1000
x_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))
z_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))

inputs = np.column_stack((x_values, z_values))

noise = np.random.uniform(-1,1,(observations_nr,1))
targets = 4*x_values - 3*z_values + 2 + noise

np.savez('TF_intro', inputs=inputs, targets=targets)
```

```python
observations_nr = 1000
x_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))
z_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))

inputs = np.column_stack((x_values, z_values))

noise = np.random.uniform(-1,1,(observations_nr,1))
targets = 4*x_values - 3*z_values + 2 + noise

np.savez('TF_intro', inputs=inputs, targets=targets)
```

`np.savez('TF_intro', inputs=inputs, targets=targets)`

- TensorFlow doesn't work well with data stored in classic .csv or .xlxs files

  It likes to work with tensors (and it makes certainly sense)

- The extension that TF2 likes is **.npz**

  This is a NumPy's file type used to store n-dimensional arrays

➢ For this reason, we need to preprocess data so to become an **.npz** file type

  `np.savez` does just that

```python
observations_nr = 1000
x_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))
z_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))

inputs = np.column_stack((x_values, z_values))

noise = np.random.uniform(-1,1,(observations_nr,1))
targets = 4*x_values - 3*z_values + 2 + noise

np.savez('TF_intro', inputs=inputs, targets=targets)
```

```python
np.savez('TF_intro', inputs=inputs,
         targets=targets)
```

Input file name that will be created from the specified inputs and targets

Label to use for the inputs and where to get them from

Label to use for the targets and where to get them from

```python
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

Now, let's train the model

- In the first line, we load the file created before
  - This is not strictly required (we already have the data), but we want to get used to it

  This is what you will likely do very often, after all

`training_data` becomes an associative array with keys "*inputs*" and "*targets*"

```python
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

- The number of inputs is 2

  In fact, we have two input variables, `x` and `z`

- The number of output is 1

  That is, `y`

```
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

- With TensorFlow, we need to build the model

  We are using Keras, which TF is based on

- **Sequential** is a Keras function that indicates how the model will be generated

  We basically define the **output layer**

```python
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

- We know that

```
output = np.dot(inputs, weights) + bias
```

We need to use the Keras function `Dense` on top of the layers

- It takes the inputs to the model and calculates the dot product of the inputs and weights, adding the bias

  It literally does what we achieved with `np.dot`

```
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

After taking care of the Data and Model, we need **Optimization algorithm** and **Objective function**

- The **`compile`** method allows to specify both

➢ **Stochastic Gradient Descent** (sgd) is a generalization of the Gradient Descent seen before (that was too simple…)

There are several optimizer that you can use: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

```python
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

We want to use **L2-Norm Loss** here too

- However, it is called "***mean_squared_error***"

- We just add the string as parameter of the `compile` method

```
training_data = np.load('TF_intro.npz')
input_size = 2
output_size = 1

model = tf.keras.Sequential([
    tf.keras.layers.Dense(output_size)
])
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=0)
```

The last line is the **`fit`** function

- It indicates to the model which data to fit

  `training_data` contains the inputs and targets tensors, so we use it to specify them and feed `fit`

- Finally, we set the number of iterations to 100

  In TensorFlow we call them `epochs`

Let’s get an output out of this

If you run the code, you should get something that looks similar to this:

```
<tensorflow.python.keras.callbacks.History at 0x241ffe88ec8>
```

That’s not very informative

- It just states that a model is stored in an object in memory

- If you try setting `verbose=1`, you will see much more
  - It is technically a progress bar, but in text form…

Let's get an output out of this

We can check the **weights** and the **biases** in this way:

```
In [10]: model.layers[0].get_weights()

Out[10]: [array([[ 4.0049505],
                 [-3.021545 ]], dtype=float32), array([1.9594544], dtype=float32)]
```

Note that we have only 1 layer, so the index [0] in `model.layers[0]`

- The function `get_weights` returns both **weights** and **biases**

Let's get an output out of this

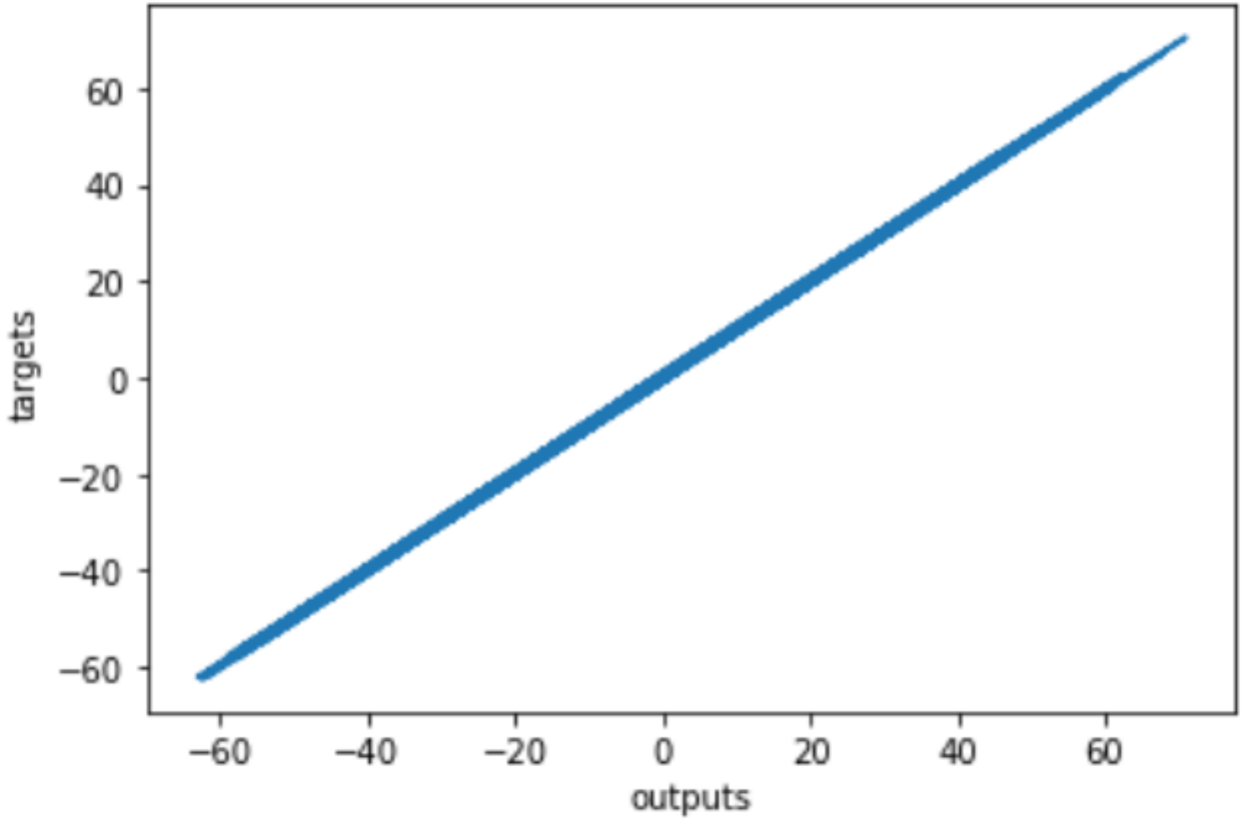
We can check the weights and the biases in this way:

```
In [10]: model.layers[0].get_weights()

Out[10]: [array([[ 4.0049505],
                 [-3.021545 ]], dtype=float32), array([1.9594544], dtype=float32)]
```

$$f(x, z) = 4x - 3z + 2 + \text{noise}$$

We can plot the data now:

```
plt.plot(np.squeeze(model.predict_on_batch(training_data['inputs'])),
         np.squeeze(training_data['targets']))
plt.xlabel('outputs')
plt.ylabel('targets')
plt.show()
```



- The line should be as closer as possible to 45 degrees
- That’s because the model’s output is very close to the target

Extract the outputs

We want to extract the outputs to make predictions:

```
In [9]: model.predict_on_batch(training_data['inputs'])

Out[9]: <tf.Tensor: shape=(1000, 1), dtype=float32, numpy=
        array([[-6.10946655e+01],
               [ 1.58667259e+01],
               [ 3.40522499e+01],
               [ 4.07857561e+00],
               [-4.56730881e+01],
               [ 5.59957047e+01],
               [-2.05926180e+00],
               [-5.58760214e+00],
               [-1.77973330e+00],
               [ 1.79382668e+01],
               [ 1.44714890e+01],
               [-3.07449570e+01],
               [ 3.78166733e+01],
               [-1.70888865e+00],
               [ 4.76942778e+00],
               [ 5.32574892e+00],
               [-2.02226143e+01],
               [-3.82981186e+01],
```

- We obtain an array where each value corresponds to an input
- These are the values compared to the targets and evaluated via the **Loss function**

  In out example, the outputs are generated after 100 epochs of training

Extract the outputs

We want to extract the outputs to make predictions:

Now we try to compare them manually

- Thus, we display the training targets and **round** all values to one digit after the dot

  So they are easily readable

- What we see is that the **outputs** and the **targets** are very close to each other but not exactly the same

```
In [10]: training_data['targets'].round(1)

Out[10]: array([[-60.3],
                [ 16.6],
                [ 33.2],
                [  5.1],
                [-45.7],
                [ 56.8],
                [ -2.6],
                [ -5. ],
                [ -1.6],
                [ 17.3],
                [ 14.9],
                [-29.9],
                [ 37.3],
                [ -1.5],
                [  4.4],
                [  5.6],
                [-20.8],
                [-38.6],
                [-34.3]
```

Extract the outputs

We want to extract the outputs to make predictions:

```
In [10]: training_data['targets'].round(1)

Out[10]: array([[-60.3],
                [ 16.6],
                [ 33.2],
                [  5.1],
                [-45.7],
                [ 56.8],
                [ -2.6],
                [ -5. ],
                [ -1.6],
                [ 17.3],
                [ 14.9],
                [-29.9],
                [ 37.3],
                [ -1.5],
                [  4.4],
                [  5.6],
                [-20.8],
                [-38.6],
                [-34.3]
```

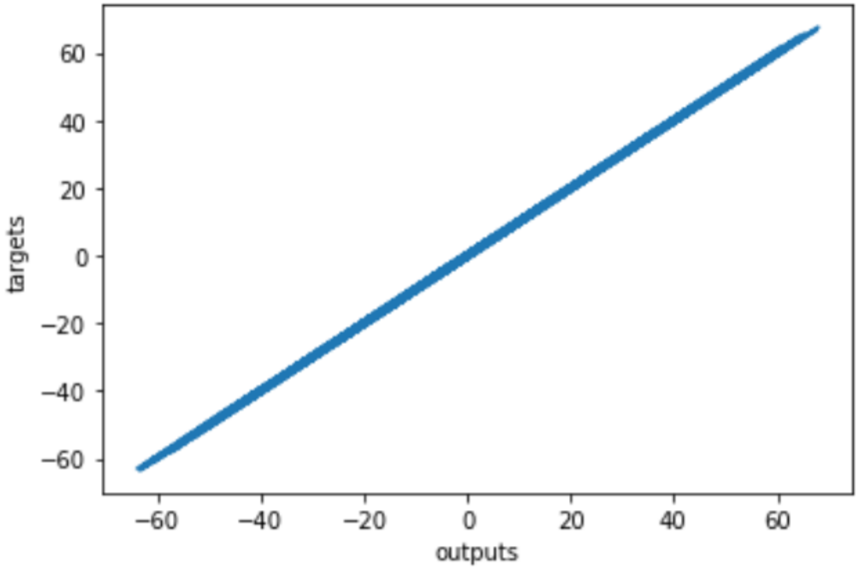
```
In [9]: model.predict_on_batch(training_data['inputs'])

Out[9]: <tf.Tensor: shape=(1000, 1), dtype=float32, numpy=
        array([[-6.10946655e+01],
               [ 1.58667259e+01],
               [ 3.40522499e+01],
               [ 4.07857561e+00],
               [-4.56730881e+01],
               [ 5.59957047e+01],
               [-2.05926180e+00],
               [-5.58760214e+00],
               [-1.77973330e+00],
               [ 1.79382668e+01],
               [ 1.44714890e+01],
               [-3.07449570e+01],
               [ 3.78166733e+01],
               [-1.70888865e+00],
               [ 4.76942778e+00],
               [ 5.32574892e+00],
               [-2.02226143e+01],
               [-3.82981186e+01],
```

Plotting the targets against the outputs

Finally we can plot the outputs against the targets

- We expect them to be very close to each other, thus, the line should be as close to 45 degrees as possible

```
In [12]: plt.plot(np.squeeze(model.predict_on_batch(training_data['inputs'])),
                  np.squeeze(training_data['targets']))
         plt.xlabel('outputs')
         plt.ylabel('targets')
         plt.show()
```



Plotting the targets against the outputs

- So, we have successfully built our first machine learning algorithm with Tensor Flow

Customizing your model – Data generation

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

```
observations = 1000

xs = np.random.uniform(low=-10, high=10, size=(observations,1))
zs = np.random.uniform(-10, 10, (observations,1))

generated_inputs = np.column_stack((xs,zs))

noise = np.random.uniform(-1, 1, (observations,1))

generated_targets = 4*xs - 3*zs + 2 + noise

np.savez('TF_intro', inputs=generated_inputs, targets=generated_targets)
```

Customizing your model

```
training_data = np.load('TF_intro.npz')
```

```
input_size = 2
output_size = 1

model = tf.keras.Sequential([
                            tf.keras.layers.Dense(output_size,
                                                  kernel_initializer=tf.random_uniform_initializer(
                                                      minval=-0.1, maxval=0.1),
                                                  bias_initializer=tf.random_uniform_initializer(
                                                      minval=-0.1, maxval=0.1)
                                                  )
                            ])

custom_optimizer = tf.keras.optimizers.SGD(learning_rate=0.02)

model.compile(optimizer=custom_optimizer, loss='mean_squared_error')

model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=2)
```

Customizing your model

Basically, we can set a **random uniform initializer**, instead of inputting the starting values ourselves

```
training_data = np.load('TF_intro.npz')
```

```
input_size = 2
output_size = 1

model = tf.keras.Sequential([
                            tf.keras.layers.Dense(output_size,
                                                  kernel_initializer=tf.random_uniform_initializer(
                                                      minval=-0.1, maxval=0.1),
                                                  bias_initializer=tf.random_uniform_initializer(
                                                      minval=-0.1, maxval=0.1)
                                                  )
                            ])

custom_optimizer = tf.keras.optimizers.SGD(learning_rate=0.02)

model.compile(optimizer=custom_optimizer, loss='mean_squared_error')

model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=2)
```

We can add a **kernel initializer** and a **bias initializer**

Kernel here is the broader term for a **weight**

Customizing your model

We use Stochastic Gradient Descent, setting the **learning rate** to 0.02

```
training_data = np.load('TF_intro.npz')
```

```
input_size = 2
output_size = 1

model = tf.keras.Sequential([
                            tf.keras.layers.Dense(output_size,
                                                  kernel_initializer=tf.random_uniform_initializer(
                                                      minval=-0.1, maxval=0.1),
                                                  bias_initializer=tf.random_uniform_initializer(
                                                      minval=-0.1, maxval=0.1)
                                                  )
                            ])

custom_optimizer = tf.keras.optimizers.SGD(learning_rate=0.02)

model.compile(optimizer=custom_optimizer, loss='mean_squared_error')

model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=2)
```

NOTE: Here, instead of having the string "*SGD*", you specify the variable '`custom_optimizer`'

HW Assignment

Using the same code seen in class, please solve the following exercises:

1. Change the number of observations to 100,000 and see what happens
2. Change the number of observations to 1,000,000 and see what happens
3. Play around with the learning rate. Interesting values are like:
   a) 0.0001
   b) 0.001
   c) 0.1
   d) 1
4. Change the loss function. L2-norm loss (without dividing by 2) is a good way to start.
5. Try with the L1-norm loss, given by the sum of the ABSOLUTE value $|y_j - t_j|$
6. Create a function $\mathrm{f(x, z)} = 13 * \mathrm{x\_values} + 7 * \mathrm{z\_values} - 12$. Does the algorithm work in the same way?