# Data Science: Principles and Practice

## Lecture 5: Deep Learning, Part II

Marek Rei

UNIVERSITY OF CAMBRIDGE

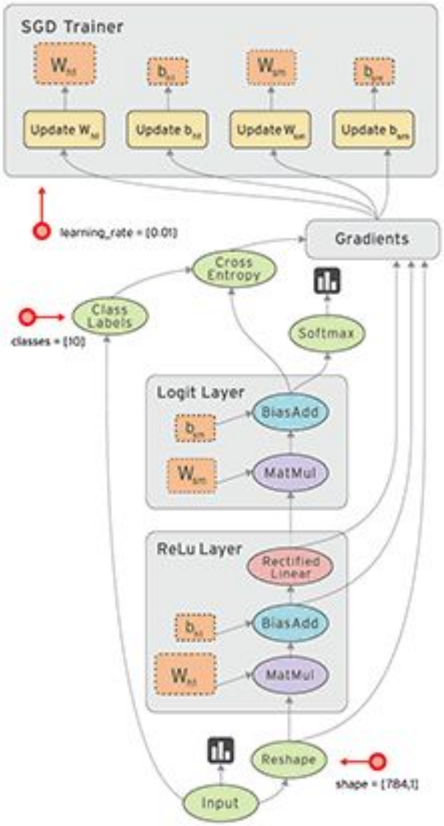
# Today:

Focusing on **Tensorflow**

Giving you all the **basics** you need in order to use Tensorflow for building neural networks.

Can't cover everything (not even close).
There is a lot of **material online** if you're looking for how to do something specific in Tensorflow.

Looking at some **practical tips** for training neural networks.



# Tensorflow

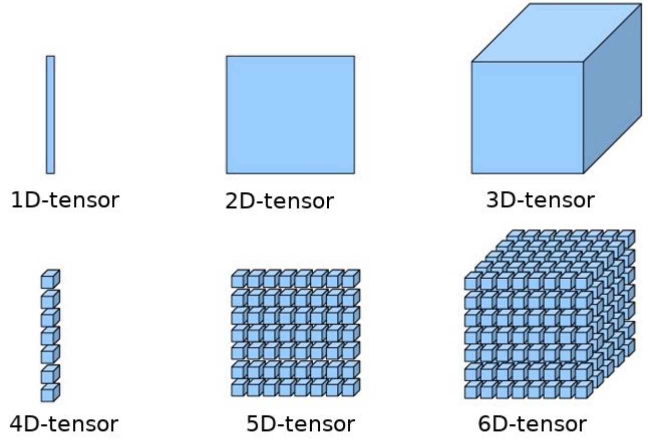
Open source library for implementing **neural networks**.

Developed by **Google**, for both production code and research.

Performs **automatic differentiation**.

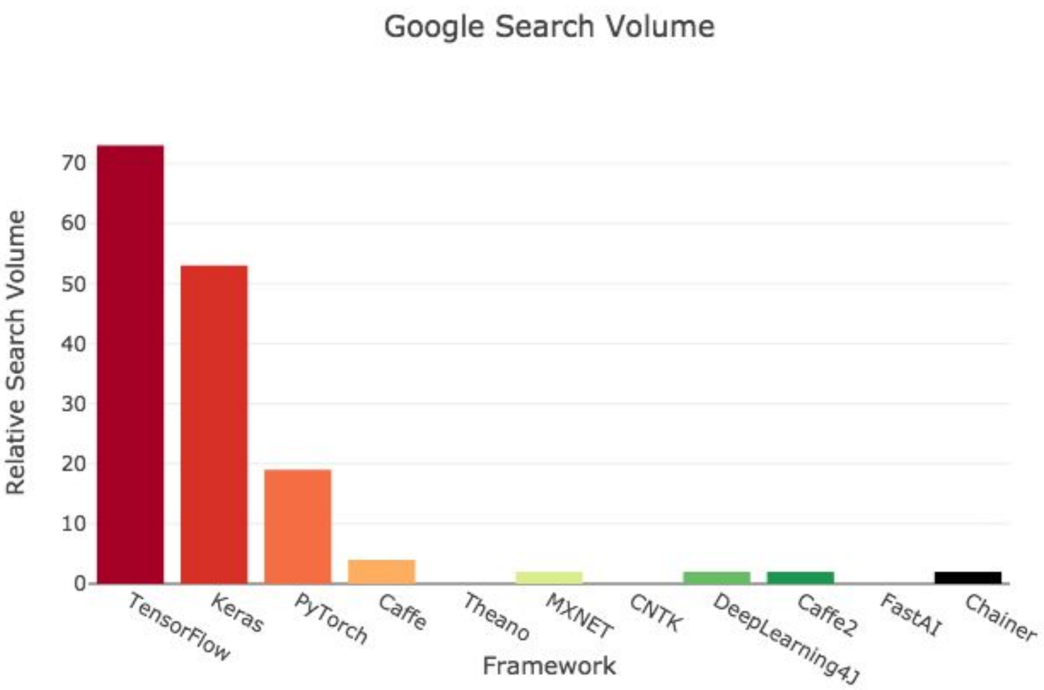
Comes with many neural network **modules** implemented.

**Tensor** - an n-dimensional vector.



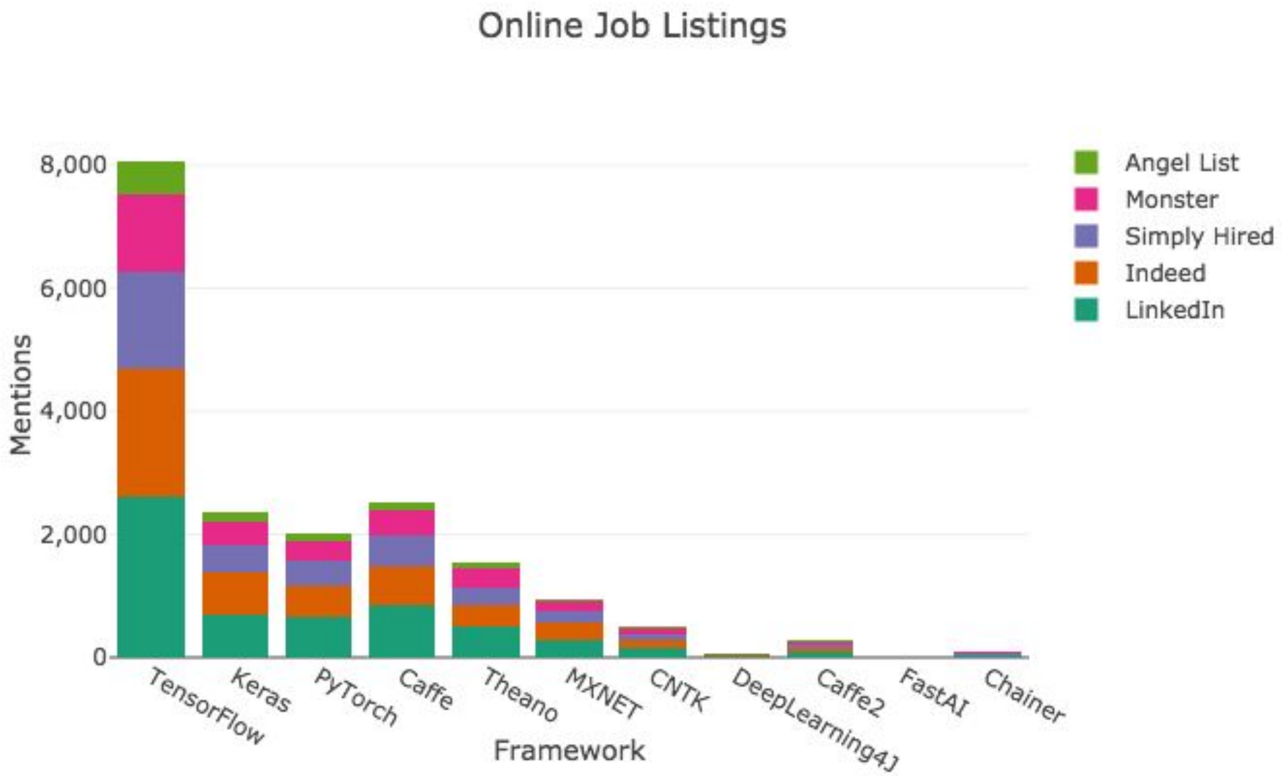

https://www.cc.gatech.edu/~san37/post/dlhc-start/

# Why Tensorflow?




https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

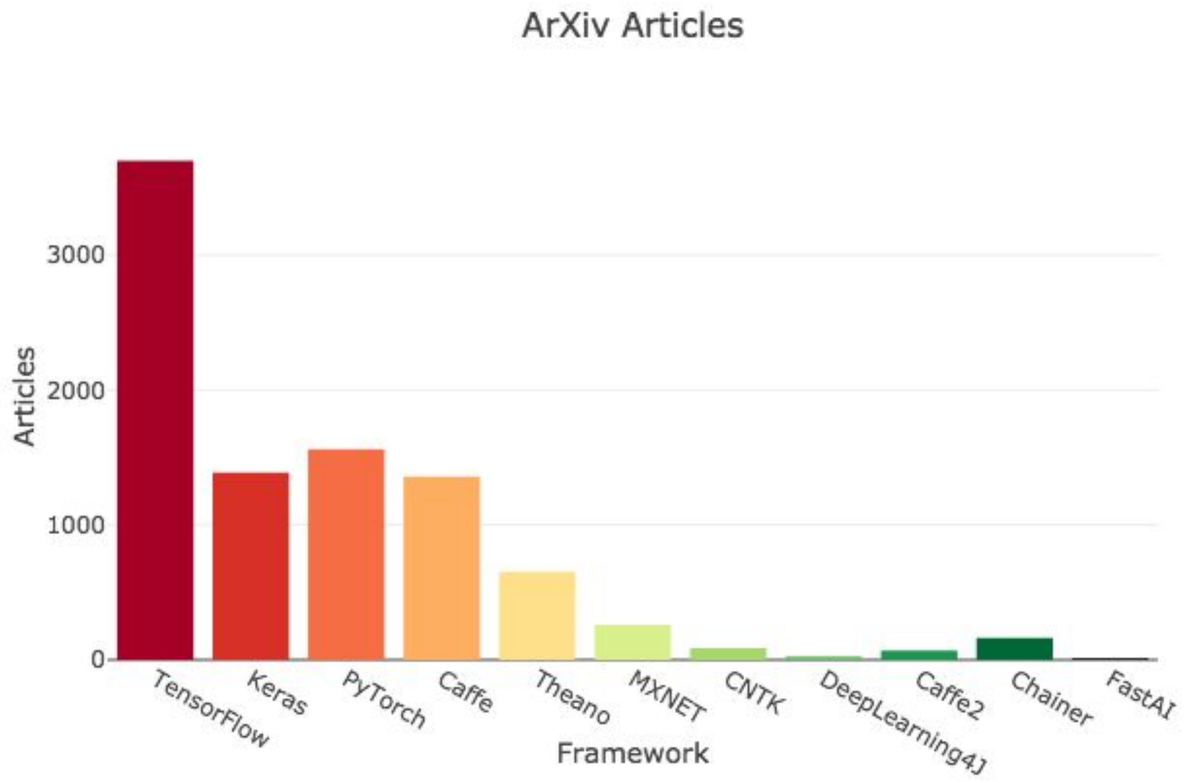
# Why Tensorflow?




https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

# Companies Using Tensorflow

# Why Tensorflow?




https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

# Tensorflow: The First Steps

# Very Minimal Tensorflow

```
import tensorflow as tf

a = tf.placeholder(tf.float32, name="a")
b = tf.placeholder(tf.float32, name="b")
y = a + b

with tf.Session() as sess:
    result = sess.run(y,
                      feed_dict={a:4, b:5})
    print("Result: ", result)
```

```
Result:  9.0
```

One of the **smallest examples** of running Tensorflow, while actually looking like a normal Tensorflow code.

Creates a **computation graph** that takes two inputs and sums them together.

We then **execute this graph** with values 4 and 5, and print the result.

Let's go though this in more detail!

# Very Minimal Tensorflow

```python
import tensorflow as tf

a = tf.placeholder(tf.float32, name="a")
b = tf.placeholder(tf.float32, name="b")
y = a + b

with tf.Session() as sess:
    result = sess.run(y,
                      feed_dict={a:4, b:5})
    print("Result: ", result)
```

```
Result:  9.0
```

import tensorflow as tf

Install tensorflow for **CPU**:

```
pip install tensorflow
```

Install tensorflow for **GPU**:

```
pip install tensorflow-gpu
```

Azure notebooks already have tensorflow installed!

# Very Minimal Tensorflow

```python
import tensorflow as tf

a = tf.placeholder(tf.float32, name="a")
b = tf.placeholder(tf.float32, name="b")
y = a + b

with tf.Session() as sess:
    result = sess.run(y,
                      feed_dict={a:4, b:5})
    print("Result: ", result)
```

```
Result:  9.0
```

tf.placeholder()

Define an **input argument** for our network.

Can have different **types**
(float32, float64, int32, …)

and **shapes**
(scalar, vector, matrix, …)

Right now, we defined two single **scalar placeholders**: a and b.

# Very Minimal Tensorflow

```
import tensorflow as tf

a = tf.placeholder(tf.float32, name="a")
b = tf.placeholder(tf.float32, name="b")
y = a + b

with tf.Session() as sess:
    result = sess.run(y,
                      feed_dict={a:4, b:5})
    print("Result: ", result)
```

```
Result:  9.0
```

$y = a + b$

Probably the most **important** thing to understand about Tensorflow!

# Symbolic Graphs

We first construct a **symbolic graph** and then apply it later with suitable data.

For example, what happens when this Tensorflow line is **executed** in our code?

```
y = a + b
```

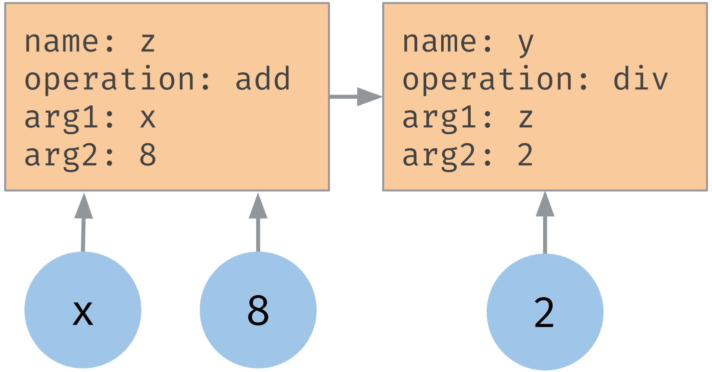
The system takes **a** and **b**, adds them together and stores the value in y. Right?

Not really!

Instead, we create a Tensorflow-specific object **y** that knows its value can be calculated by summing together **a** and **b**. But the addition itself is not performed here!

# Symbolic Graphs

Can construct a whole network structure by intuitively **combining operations**.




```
z = x + 8

y = z / 2
```

We can only use **Tensorflow-specific*** operations to construct a Tensorflow graph - they return Tensorflow objects, as opposed to trying to execute the operation.

* Most of numpy and standard operations are compatible with Tensorflow

# Very Minimal Tensorflow

```python
import tensorflow as tf

a = tf.placeholder(tf.float32, name="a")
b = tf.placeholder(tf.float32, name="b")
y = a + b

with tf.Session() as sess:
    result = sess.run(y,
                      feed_dict={a:4, b:5})
    print("Result: ", result)
```

```
Result:  9.0
```

tf.Session()

Constructs the **environment** in which the operations are performed and evaluated.

Allocates the **memory** to store current value of valuables.

When starting a new session, all the values will be **reset**.

# Very Minimal Tensorflow

```python
import tensorflow as tf

a = tf.placeholder(tf.float32, name="a")
b = tf.placeholder(tf.float32, name="b")
y = a + b

with tf.Session() as sess:
    result = sess.run(y,
                      feed_dict={a:4, b:5})
    print("Result: ", result)
```

```
Result:  9.0
```

sess.run()

**Execute** the network - actually perform the calculations in the symbolic graph.

Specify which values you want calculated and **returned** from the graph.

**feed_dict** specifies the values that you give to placeholders for this execution.

**result** contains the executed value of y.

The keys in feed_dict are the tensors!

# Training a Network

# Training Tensorflow

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

An example of defining a network with trainable parameters and actually **optimizing** them.

Technically **linear regression**…

but we can add non-linearities and more neurons to make it into a proper **neural network**.

$$\theta_1 \cdot 1.0 + \theta_2 \cdot 1.0 = 20.0$$

Some new parts. Let's take a look!

# Training Tensorflow

```python
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

tf.placeholder()

This time creating 3 placeholders:

**x** is a vector of length 2

**target** and **learning_rate** are scalars

# Training Tensorflow

```python
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

tf.get_variable()

These Variable objects contain model **parameters** that are updated during model training.

At the moment, we are manually **initializing** it with values.

Normally, we would just specify the shape and initialize **randomly**.

# Training Tensorflow

```python
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

tf.train.GradientOptimizer()

This is where we define the **strategy** for our model training.

**Other strategies** are available:

```
tf.train.AdadeltaOptimizer()
tf.train.AdagradOptimizer()
tf.train.AdamOptimizer()
tf.train.RMSPropOptimizer()
tf.train.MomentumOptimizer()
```

Part of the computation graph!

# Training Tensorflow

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

optimizer.minimize()

**Updates** all the variables in the graph

- **minimizing** the loss function
- following the **optimizer** strategy

optimizer.**variables()** can give us a list of all the variables that it updates.

optimizer.**compute_gradients()** calculates gradients without updating the variables.

Also part of the computation graph!

# Training Tensorflow

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

tf.global_variables_initializer()

This is where all the variables get **initialized**.

Just something you need to call after constructing the network to actually get the **values** into the variables.

# Training Tensorflow

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

sess.run()

Both **y** and **train_op** are returned by the sess.run() function.

The parameters are **updated** whenever we ask the model to return train_op.

**feed_dict** now contains a vector and two scalars.

# Training Tensorflow

```python
import tensorflow as tf

x = tf.placeholder(tf.float32, [2], name="x")
target = tf.placeholder(tf.float32, name="target")
learning_rate = tf.placeholder(
    tf.float32,
    name="learning_rate")

W = tf.get_variable("W", initializer=[0.2, 0.7])
y = tf.reduce_sum(x * W)

loss = tf.pow(target - y, 2.0)
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)
train_op = optimizer.minimize(loss)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(10):
        result, _ = sess.run(
            [y, train_op],
            feed_dict={x: [1.0, 1.0],
                       target: 20.0,
                       learning_rate: 0.1})
        print("Result: ", result)
```

Result:

```
0.9
8.54
13.124001
15.874401
17.52464
18.514786
19.108871
19.46532
19.679192
19.807514
```

# Useful Things to Know about Deep Learning

# PyTorch

Pytorch is designed for **eager execution** - no symbolic graphs, operations are performed where they appear in the code.

## Advantages of Symbolic Graphs

- Can be internally optimized
- Faster (in theory)
- Easily deployable, even across languages

## Advantages of Eager Execution

- Easier to understand
- Easier to debug
- Supports dynamic graphs

The newest Tensorflow now also has **eager execution support**

... but it's still very much in active development.

# Randomness in the Network

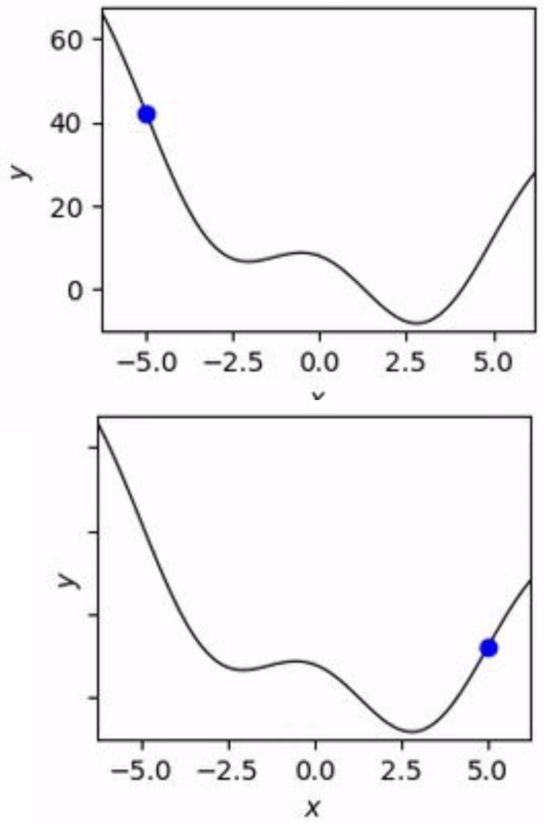
Different **random initializations** lead to different results.

**Solution:** Explicitly set the random seed.
All the random seeds!

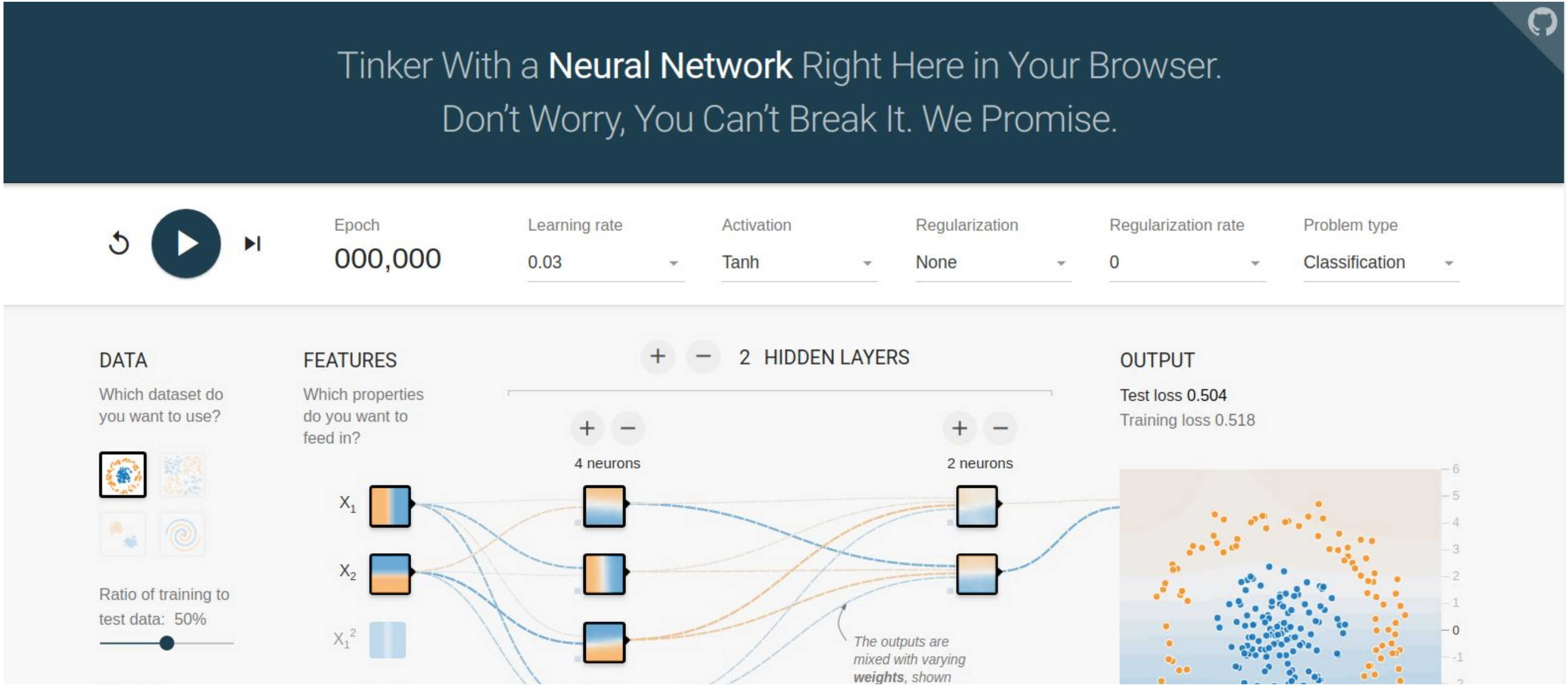
BUT!

**GPU threads** finish in a random order, also leading to randomness!
Small rounding errors really add up!
Doesn't affect all operations.

**Solution:** Embrace randomness, run with different random seeds and report the average.



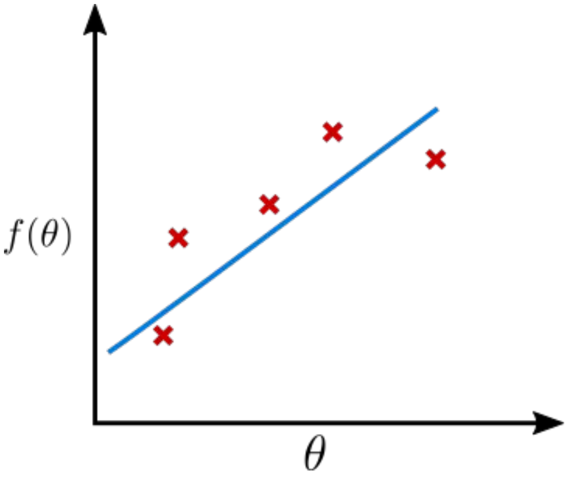
# Tensorflow Playground



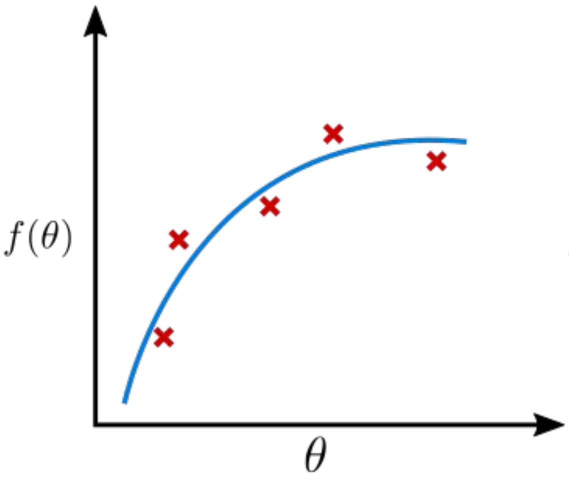

playground.tensorflow.org

# Fitting to the Data

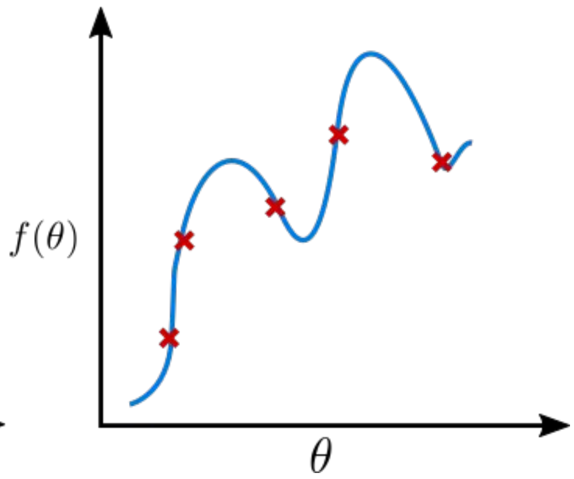
**Underfitting**
The model does not have the capacity to properly model the data.




**Ideal fit**




**Overfitting**
Too complex, the model memorizes the data, does not generalize.



# Splitting the Dataset

In order to get realistic results for our experiments, we need to evaluate on a **held-out test set**.

Using a separate development set for choosing hyperparameters is even better.

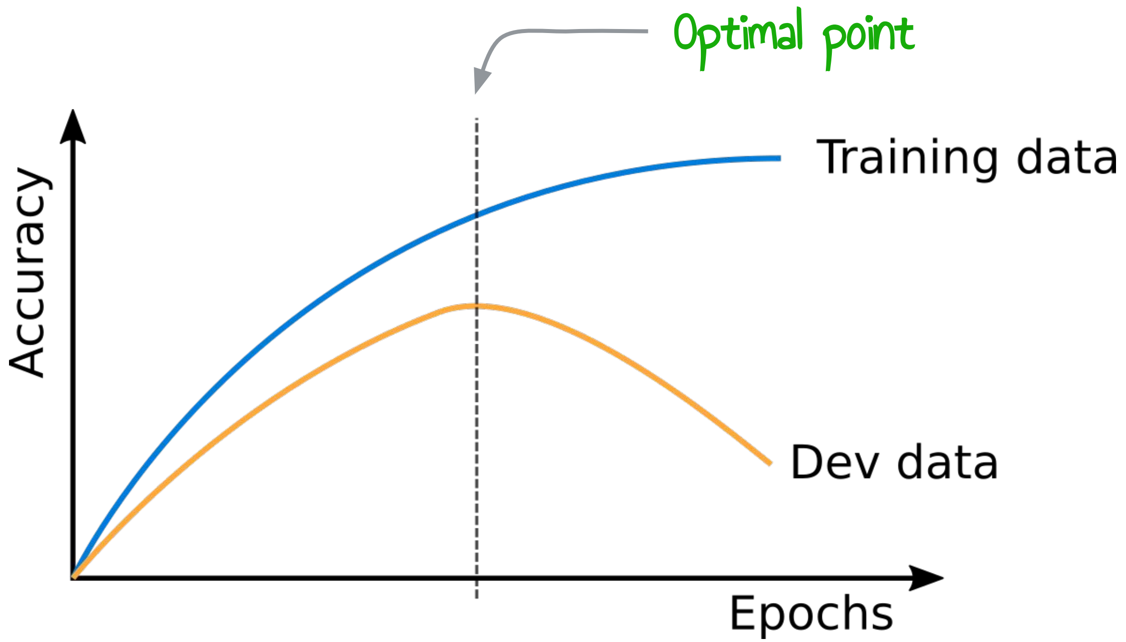

For training your models, fitting the parameters

For continuous evaluation and hyperparameter selection

For realistic evaluation once the training and tuning is done

# Early Stopping

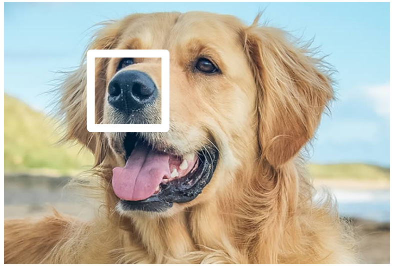
A sufficiently powerful model will keep improving on the training data until it **overfits**.
We can use the **development** data to choose when to stop.

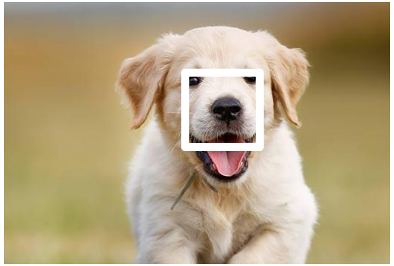


# Convolutional Neural Networks

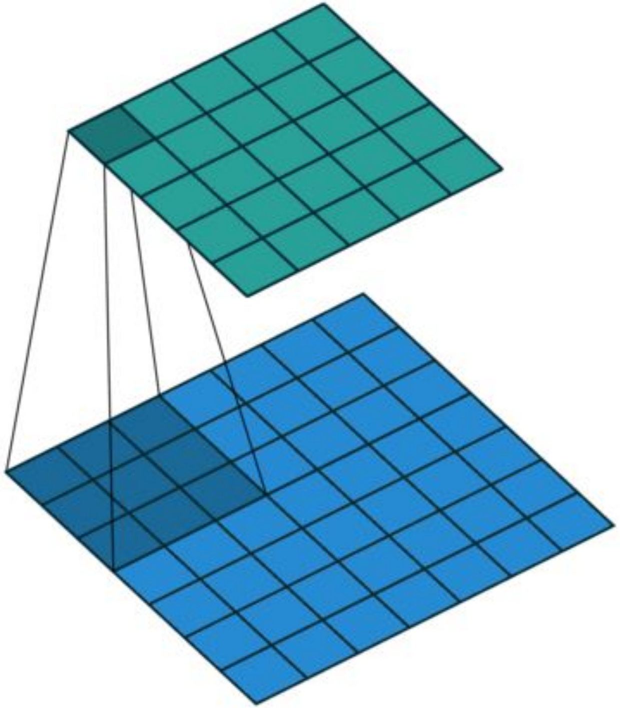
Neural modules operating **repeatedly** over different subsections of the input space.

Great when **searching** for feature patterns, without knowing where they might be located in the input.





The main driver in **image recognition**.
Can also be used for text.



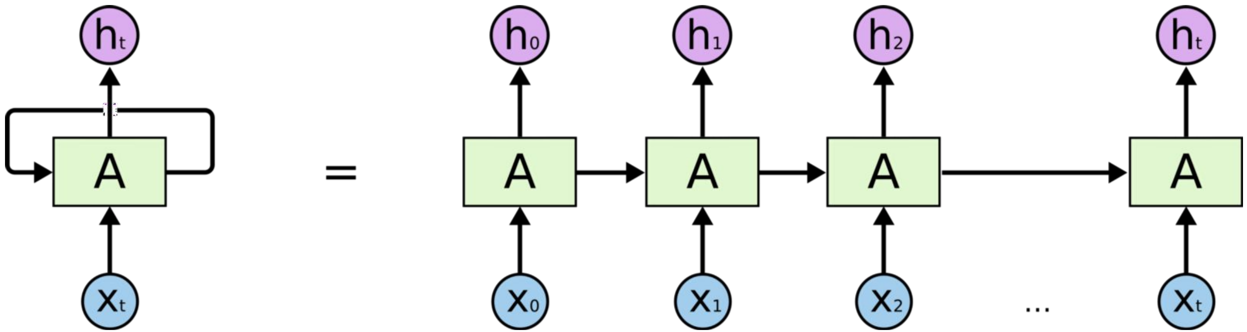
https://github.com/vdumoulin/conv_arithmetic

# Recurrent Neural Networks

Designed to process **input sequences** of arbitrary length.

Each hidden state A is calculated based on the **current input** and the **previous hidden state**.

Main neural architecture for **processing text**, with each input being a word representation.



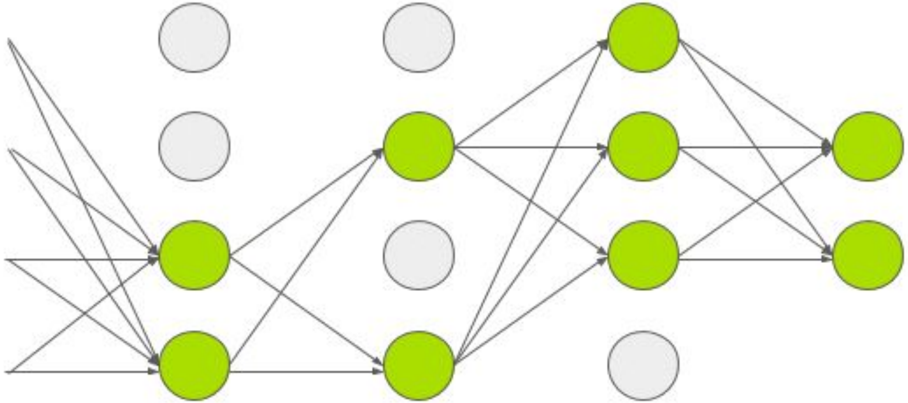

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

# Dropout

During training, randomly set some activations to **zero.**

Typically **drop 50%** of activations in a layer

Form of regularization - prevents the network from **relying** on any one node.



https://www.learnopencv.com/understanding-alexnet/

# GPU Acceleration

**Parallelize** large matrix operations to the GPU.
Really makes a difference!
Doesn’t help for small networks

Need to install **CUDA**:
https://developer.nvidia.com/cuda-downloads

**CuDNN** also recommended:
https://developer.nvidia.com/cudnn

Can control which GPUs Tensorflow sees

```
CUDA_VISIBLE_DEVICES=0  python experiment.py
CUDA_VISIBLE_DEVICES=’’ python experiment.py
```

No GPUs on Azure Notebooks unfortunately

# TensorBoard

A tool for **visualizing** your own Tensorflow networks.