Introduction to TensorFlow and Deep Learning

Lecture 1: TensorFlow basic concepts

IADS Summer School, 1st August 2022

Dr Michael Fairbank

University of Essex, UK

Email: m.fairbank@essex.ac.uk

Today’s Outline

Lecture 1 (9:00am-10.30am):

- TensorFlow Basics
- Gradient Descent

Lecture 2 (11:00am-12.30pm):

- Neural Networks + Deep Learning

Lecture 3 (1:30pm-3.00pm):

- Managing Data – loading, training, regularization, saving.

Lecture 4 (3:30pm-5.00pm):

- Convolutional Neural Networks and Vision
- Introduction to Recurrent Neural Networks (* If time permits)

Course Motivations

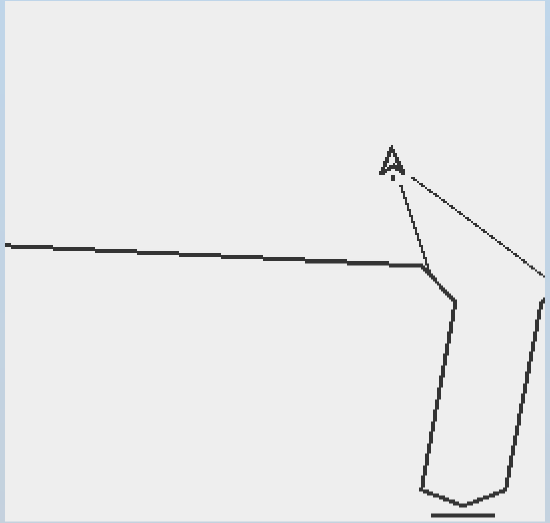
- An introductory course
- Provide a good understanding of fundamentals of
  - TensorFlow
  - Backpropagation / AutoDiff
  - Training Neural Networks
- Not aiming to be a Keras guide to the fastest way to build neural networks
  - Hopefully give deeper understanding than that
  - However Keras will be introduced along the way

Motivations for using TensorFlow

- Ideal for neural networks and deep learning:
  - Fast tensor + matrix multiplication (using GPU if present)
  - “Automatic differentiation” is included
    - no need to write your own backprop code
  - Lots of neural network structures included
    - MLPs
    - CNNs
    - RNNs
    - LSTMs
  - Lots of optimisation algorithms included
    - Stochastic Gradient Descent
    - RMSprop
    - Adam
- It’s all supported and developed by someone else (Google)

About myself

Been developing Neural Networks since 1995



Learning algorithms for NNs and Reinforcement Learning

Recently: Applications of NNs to Electrical Power systems, Motors, Question and Answering, Target Space

Acknowledgements:

- Many pictures were created by others
- Some python code scripts were based on code from e.g. tensorflow.org

TensorFlow – Installation

- Requires Python 3.
  - Java/C/Go flavours also available, but not as stable

- Usually it's simple "pip install tensorflow" for Python
  - See https://www.tensorflow.org/install/

- Should have Python3 + TensorFlow v 2.9.0 installed for this course.
  - CPU version
  - Also check you have matplotlib, pandas, numpy, jupyter-notebooks:
    - pip install matplotlib pandas numpy notebook tensorflow
  - Can use google colab if you prefer

TensorFlow – Check installation

```
import tensorflow as tf
print(tf.__version__)
2.9.0
```

Try this.
(Use the accompanying jupyter notebook:

- lecture1-notebook.ipynb
- Run the appropriate notebook cell with “ctrl+enter”)

What's a tensor?

- In TensorFlow, a "rank n" tensor is equivalent to an "n dimensional array".

- [1 2 3 4] A rank-1 tensor (a vector)
- [[1 2]
  [3 4]] A rank-2 tensor (a matrix)
- 2 A rank-0 tensor (a scalar)

- Aim to build our algorithms out of tensors and matrix multiplications
  - so we can pass their manipulation off to the graphics card / low-level compiled code.

Basic concepts- Tensor scalars, and numpy

```
a=tf.constant(2,tf.float32)
b=tf.constant(3,tf.float32)
c=tf.add(a,b)
print(c)
```

A rank-0 tensor (i.e. a scalar) of type float32

```
tf.Tensor(5.0, shape=(), dtype=float32)
```

This just says c is a Tensor of shape() (rank zero), type float32, with value 5.0

```
print(c.numpy())
5.0
```

Converts from tensorflow datatype to a numpy datatype

You can try any of these in the lecture1 notebook.
Don't rush to keep up though: you can go back after the lecture and fill in any gaps

Basic concepts – tensor addition

```
a=tf.constant([[1,2],[3,4]])
b=tf.constant([[5,6],[8,9]])
c=tf.add(a,b)
print(c.numpy())
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix} = \begin{pmatrix} ? & ? \\ ? & ? \end{pmatrix}$$

```
[[ 6  8]
 [11 13]]
```

Numpy Array

Basic concepts – tensor multiplication

```
a=tf.constant([[1,2],[3,4]])
b=tf.constant([[5,6],[8,9]])
c=tf.multiply(a,b)
print(c.numpy())
```

Elementwise multiplication
("Hadamard product")

$$c_{ij} = b_{ij} b_{ij}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix} = \begin{pmatrix} ? & ? \\ ? & ? \end{pmatrix}$$

```
[[ 5  12]
 [24 36]]
```

https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

Basic concepts – matrix multiplication

```
a=tf.constant([[1,2],[3,4]],tf.float32)
b=tf.constant([[1],[1]], tf.float32)
c=tf.matmul(a,b)
print(c.numpy())
[[3]
 [7]]
>>>
```

A rank-2 tensor (i.e. a 2*2 matrix)

A rank-2 tensor (a 2*1 matrix)

Matrix multiplication
(c.f. previous slide which was tf.multiply = elementwise multiplication)

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \end{pmatrix}$$

Basic concepts – datatypes

```
>>>a=tf.constant(3.2, tf.float32)
>>>b=tf.constant(3, tf.int32)
>>>c=tf.constant([1,2,3], tf.float32)
>>>d=tf.constant(5)
>>>e=tf.constant(5.0)
```

When you create a tensor, you should specify its datatype

This defaults to int32

This defaults to float32

- Also have tf.float64, tf.int64, tf.bool
  - See https://www.tensorflow.org/api_docs/python/tf/dtypes/DType

Basic concepts – casting datatypes (1)

- You can convert one datatype to another as follows:

```
a=tf.constant([[1,2],[3,-4]],tf.float32)
print(tf.cast(a,tf.int32).numpy())
[[ 1  2]
 [ 3 -4]]
```

This is now an integer tensor

```
b=tf.constant([True, False, True], tf.bool)
print(tf.cast(b,tf.int32).numpy())
[1 0 1]
```

Bools cast using True=1, False=0

Basic concepts – casting datatypes (2)

- You can't add datatypes that don't match

```
>>>a=tf.constant(3.0, tf.float32)
>>>b=tf.constant(3, tf.int32)
>>>c=tf.add(a,b)
…
tensorflow.python.framework.errors_impl.InvalidArgumentError:
cannot compute Add as input #1(zero-based) was expected to be a
float tensor but is a int32 tensor [Op:Add]
```

- We have to cast like this:

```
c=tf.add(a,tf.cast(b,tf.float32))
print(c.numpy())
6.0
```

Basic concepts – tensor shape (1)

- Tensor shapes must match for most operations
  - E.g. $\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} + \begin{pmatrix}1 & 2 & 3\end{pmatrix} = ???$

```
a=tf.constant([1,2])
b=tf.constant([2,3,1])
f=tf.add(a,b)
…
tensorflow.python.framework.errors_impl.InvalidArgumentError: Incompatible shapes: [2] vs. [3] [Op:Add]
…
```

Basic concepts – tensor shape (2)

- However there is a shorthand that violates these size-matching rules
- When the rank of one matrix is less than the other it tries to add them in the most sensible way (if possible)

  - $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 1 = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$

```
a=tf.constant([1,2])
b=tf.constant(1)
print(tf.add(a,b).numpy())
[2 3]
```

Basic concepts – tensor shape (3)

```
a=tf.constant([[1,2],[3,4]])
b=tf.constant([10,20])
print(tf.add(a,b).numpy())
…
```

- $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 10 & 20 \end{pmatrix} = \begin{pmatrix} ? & ? \\ ? & ? \end{pmatrix}$

- This behaviour is called “broadcasting”
  - For the precise rules, see https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Elementwise Tensor operations

- These elementwise operations produce a tensor of equal size to the input

```
a=tf.constant([[1,2],[3,-4]],tf.float32)
print(tf.square(a).numpy())
```

```
[[  1.    4.]
 [  9.   16.]]
```

```
print(tf.abs(a).numpy())
```

```
[[ 1.  2.]
 [ 3.  4.]]
```

```
print(tf.tanh(a).numpy())
```

```
[[ 0.76159418  0.96402758]
 [ 0.99505472 -0.99932921]]
```

Comparison Tensor operations

- See https://www.tensorflow.org/api_docs/python/tf/math/greater

```
a=tf.constant([1,2,3])
b=tf.constant([5,1,7])
print(tf.greater(a,b).numpy())
```

- Which elements of matrix *a* are bigger than those corresponding elements of *b*?

```
[False  True False]
```

```
print(tf.greater(a,1).numpy())
```

```
[False  True  True]
```

This is “broadcasting” the mismatching tensor sizes, i.e. changing 1 to [1,1,1] before comparison

Basic concepts – operator shorthand

- Some math functions have shorthand operators.
- The most common ones are:

a+b	tf.add(a,b)
a-b	tf.subtract(a,b)
a*b	tf.multiply(a,b)
a>b	tf.greater(a,b)

Q: Elementwise multiplication, or matrix product?

```
a=tf.constant([[1,2],[3,4]])
b=tf.constant([[5,6],[8,9]])
print((a+b).numpy())
```

```
[[ 6  8]
 [11 13]]
```

More info: https://stackoverflow.com/questions/37900780/in-tensorflow-what-is-the-difference-between-tf-add-and-operator

Basic concepts – variables vs. constants (1)

Unlike “constants”, all “Variables” can be updated.

```
W = tf.Variable([0.3], tf.float32)
W.assign([-1.0])
print(W.numpy())
[-1.]
```

Constants however, once created, cannot be “reassigned”

But (confusingly) you can do:

```
x = tf.constant([0.3], tf.float32)
x = tf.constant([-1], tf.float32)
print(x.numpy())
[-1.]
```

Q: What’s the difference between this reassignment and W.assign(…)?

Basic concepts – variables vs. constants (2)

Variables also have **assign_add** and **assign_sub**

```
W = tf.Variable([0.3], tf.float32)
W.assign_add([1.0])
print(W.numpy())
[1.3]
W.assign_sub([1.0])
print(W.numpy())
???
```

Analogous to W+=1

Aggregation functions (1)

```
a=tf.constant([[1,2],[3,4]],tf.float32)
print(tf.reduce_sum(a).numpy())
10.0
print(tf.reduce_mean(a).numpy())
2.5
print(tf.reduce_max(a).numpy())
4.0
```

Aggregation functions (2)

Challenge question:

```
a=tf.constant([[1,2],[3,4]],tf.float32)
print(tf.reduce_sum(tf.cast(a>1,tf.float32)).numpy())
```

Aggregation functions (3)

- Argmax counts the index at which the max element appears
- Reduce_max vs argmax:

```
a=tf.constant([4,0,5,-4],tf.float32)
print(tf.reduce_max(a).numpy())
5.0
print(tf.argmax(a).numpy())
2
```

This indicates the 3rd element
(since indices start counting at zero)

Aggregation functions across an axis (1)

- Sum the elements of a matrix across one given axis:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Axis 0

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Axis 1

```
a=tf.constant([[1,2],[3,4]])
print(tf.reduce_sum(a, axis=0).numpy())
[4 6]
print(tf.reduce_sum(a, axis=1).numpy())
[3 7]
```

Aggregation functions across an axis (2)

- Aggregate elements of a matrix across one given axis:

$$\begin{pmatrix} 5 & 10 & 0 \\ 3 & 4 & 12 \end{pmatrix} \qquad \begin{pmatrix} 5 & 10 & 0 \\ 3 & 4 & 12 \end{pmatrix}$$

Axis 0 Axis 1

```
a=tf.constant([[5,10,0],[3,4,12]])
print(tf.reduce_max(a, axis=0).numpy())
[ 5 10 12]
print(tf.argmax(a, axis=1).numpy())
[1 2]
```

Summary

- Tensors
  - Rank, shape,datatypes
- .numpy()
- Constants vs Variables
- Operators on tensors
  - Math functions
    - Add, multiply, matmult, tanh, greater
  - Aggregate functions
    - Max, argmax, reduce_sum

Automatic differentiation

Automatic differentiation (Autodiff) (1)

TensorFlow knows that:

$$y = x^2$$

$$\Rightarrow \frac{dy}{dx} = 2x$$

```
x = tf.Variable(3.0, tf.float32)
with tf.GradientTape() as tape:
      y=tf.pow(x,2.0)
dydx=tape.gradient(y, x)
print(dydx.numpy())
6.0
```

Automatic differentiation (Autodiff) (2)

- **Autodiff** is fast and exact differentiation
  - Not numerical differentiation (which is neither exact nor fast)
  - Not symbolic differentiation either
  - it’s something in between
    - If you give it code to compute a function f(x), it will write corresponding program code for you that calculated df/dx

- Further reading: https://justindomke.wordpress.com/2009/02/17/automatic-differentiation-the-most-criminally-underused-tool-in-the-potential-machine-learning-toolbox/

Automatic differentiation (Autodiff) (3)

Autodiff also works when there is more than one input variable:

$$f(x,y) = 3x^2 + y$$
$$\Rightarrow \frac{\partial f}{\partial x} = 6x,\ \frac{\partial f}{\partial y} = 1$$

“Partial derivatives”

```
x=tf.Variable(4.0,tf.float32)
y=tf.Variable(2.0,tf.float32)
with tf.GradientTape() as tape:
	f=tf.pow(x,2.0)*3.0+y
[dfdx, dfdy]=tape.gradient(f, [x,y])
print(dfdx.numpy(), dfdy.numpy())
```

Fetching two derivatives at once

```
24.0 1.0
```

Automatic differentiation (Autodiff) (4)

If you want a derivative w.r.t. a constant, then you need tape.watch(…)

```
x=tf.constant(4.0,tf.float32)
with tf.GradientTape() as tape:
        tape.watch(x)
        f=tf.pow(x,3.0)
dfdx=tape.gradient(f, x)
print(dfdx.numpy(), dfdy.numpy())
```

A “constant” requires watching

```
24.0 1.0
```

Automatic differentiation (Autodiff) (5)

Autodiff also works when the input variables are higher rank tensors

$$f(W, x, y) = ||\tanh(Wx) - y||^2$$

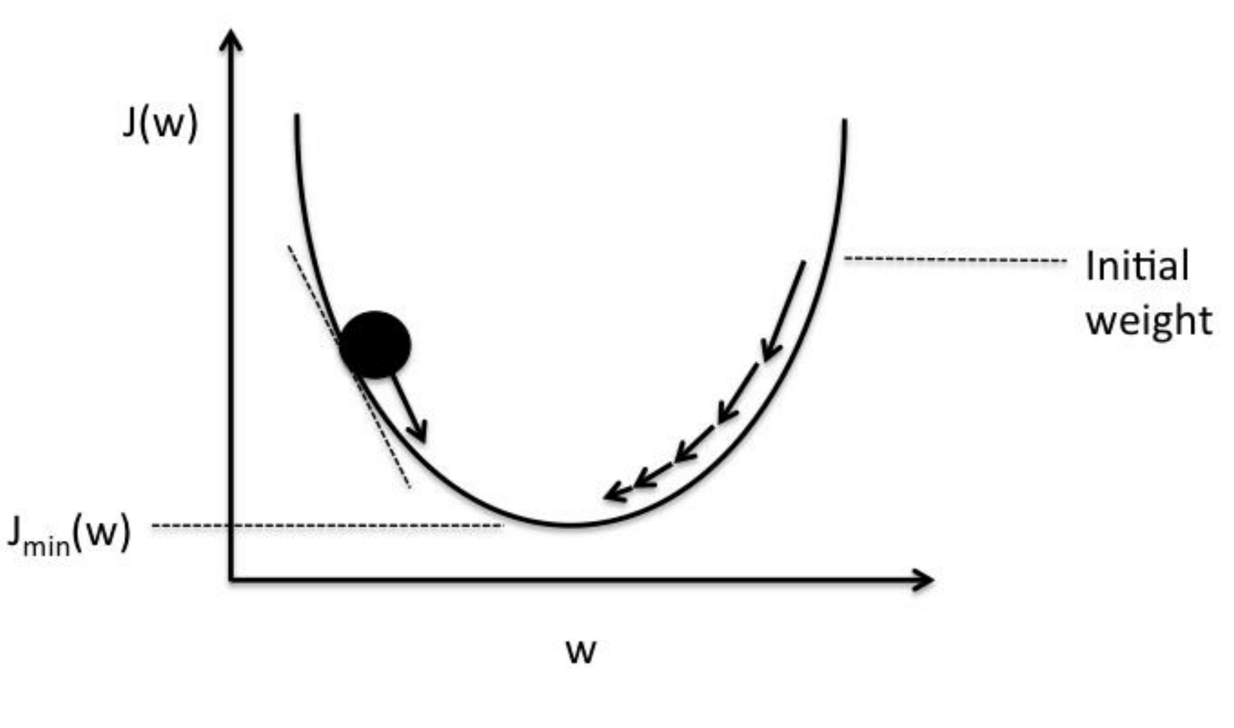
E.g. a 2*2 matrix

Automatic differentiation (Autodiff) (6)

- AUTODIFF makes neural-network training much easier to program
  - Autodiff replaces “backpropagation” programming
  - Backpropagation is <u>replaced</u> by autodiff

Introduction to Gradient Descent

Introduction to Gradient Descent



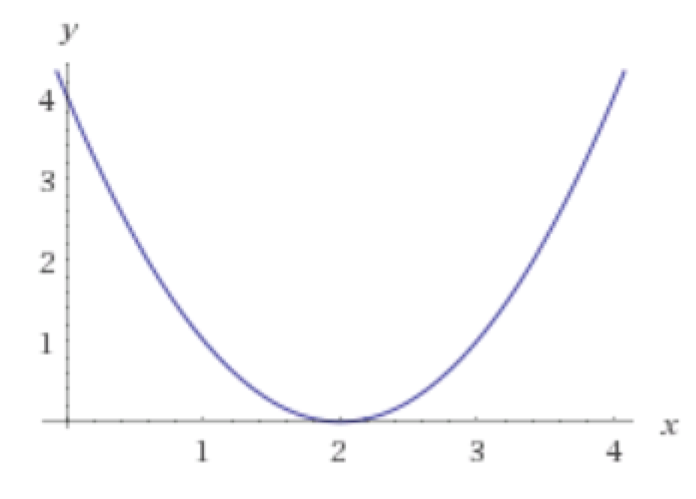
Schematic of gradient descent.

Image credit: https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

$$w_{t+1} = w_t - \eta \frac{\partial J}{\partial w}$$

Example 1D Gradient Descent problem

- Use gradient descent to find the minimum of $y = x^2 - 4x + 4$



- Start with any x_0
- Iterate with $x_{t+1} = x_t - \eta \frac{dy}{dx}$
- η is "learning rate"

Q: What will happen if η too large, too small or just right??

Example 1D Gradient Descent problem

Exercise 1: Use gradient descent to find the minimum of $y = x^2 - 4x + 4$. Complete this code in the accompanying jupyter notebook under "Exercise 1"

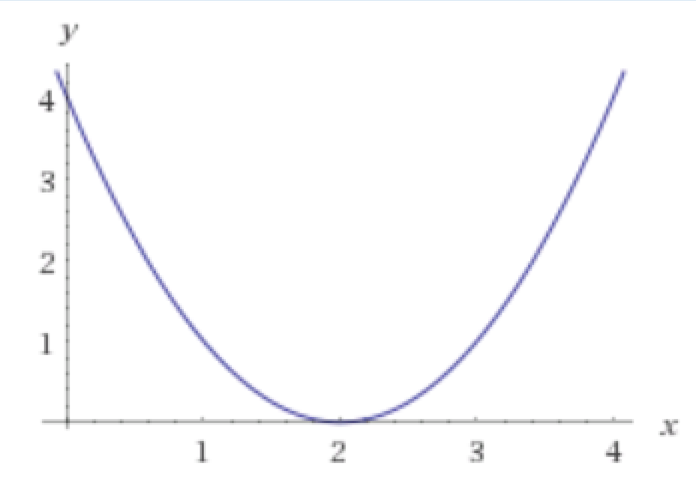
```
import tensorflow as tf

eta = 0.1 # learning rate
x = tf.Variable(10.0, tf.float32) # arbitrary initial value

for i in range(50):
        with tf.GradientTape() as tape:
                y=#TODO put in formula for y in terms of x here
        dydx=tape.gradient(# TODO finish this line
        x.assign(#TODO finish x_(t+1)=x_t-eta*dydx
        print("iteration:",i, "x:", x.numpy(), "y:", y.numpy())
```

Example 1D Gradient Descent problem

Observations:

- We didn't need to give the iterative variable's steps different variable names $x_1, x_2, \ldots$ We just called them all "x"
  - So $x_{t+1} = x_t - \eta \frac{dy}{dx}$ became $x = x - \eta \frac{dy}{dx}$
- It looks pretty inefficient:
  - Recalculates out the automatic differentiation formula every step of loop!

Example 1D Gradient Descent problem



Observations:

We should have correctly found the minimum at x=2, y=0:

iteration: 41 x: 2.00068 y: 4.76837e-07

iteration: 42 x: 2.00054 y: 4.76837e-07

iteration: 43 x: 2.00044 y: 0.0

iteration: 44 x: 2.00035 y: 0.0

...

Example 1D Gradient Descent problem

Observations:

- If eta too high, it fails
- If eta too low, solution is very slow.

Recommendation:

- Plot the value of the function you are trying to minimise vs. iteration number
- Make sure it is decreasing
- Reduce eta if necessary

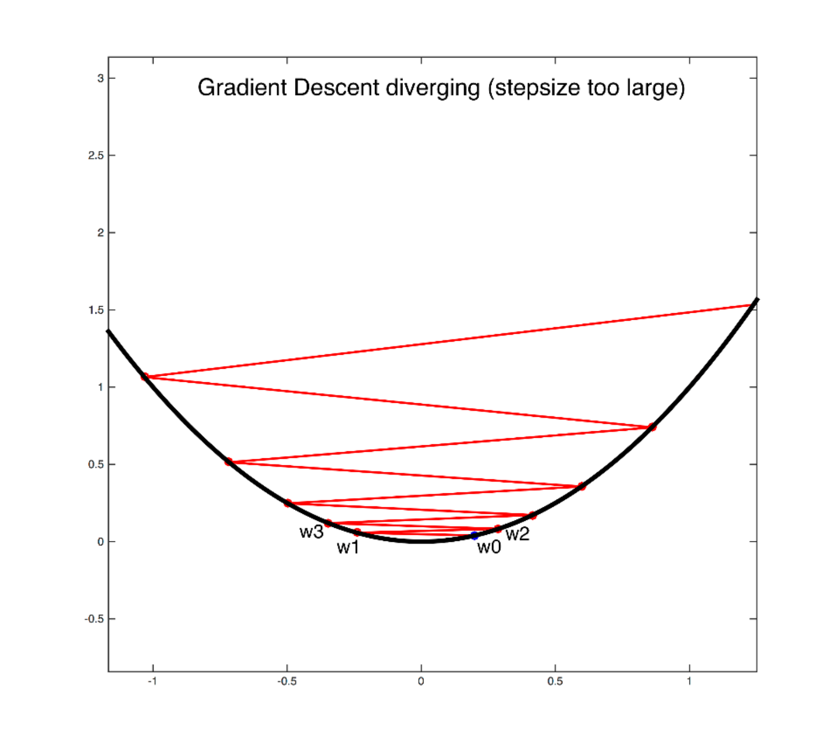


Image source: http://www.cs.cornell.edu/courses/cs4780/2017sp/lectures/images/gradient_de

Example 1D Gradient Descent problem

Inefficiency:

- Recalculates the automatic differentiation formula every step of loop!

Fix this by:

1. pulling out guts of main loop into a separate python function
   - def do_update():
2. Annotate this function by “@tf.function”

Optimised version:

```
import tensorflow as tf
eta = 0.1 # learning rate
x = tf.Variable(10.0, tf.float32) # arbitrary initial value

@tf.function
def do_update(x):
      with tf.GradientTape() as tape:
            y=tf.pow(x,2.0)-4.0*x+4.0
      dydx=tape.gradient(y, x)
      x.assign(x-dydx*eta)
      return y

for i in range(50):
      y=do_update(x)
      print("iteration:",i, "x:", x.numpy(), "y:", y.numpy())
```

Put these changes into your notebook as a new script, under “Exercise1, Version 2”

Optimised version: @tf.function

- Use function annotation @tf.function to speed up execution, take advantage of the GPU, and for saving models
  - It allows tensorflow to cache the graph of computations so that it doesn't have to recalculate them (or the derivatives) every iteration.
- Adding the @tf.function in this task sped things up by around 4 times
- Put @tf.function around the main functionality of your training loop
  - After you've debugged things
  - Warning – the function you are optimising must not refer to any global variables (unless they are strictly constants)
- See [Better performance with tf.function | TensorFlow Core]

Using a built-in optimizer

- This will behave identically to our initial solution

```
import tensorflow as tf

eta = 0.1 # learning rate
x = tf.Variable(10.0, tf.float32) # arbitrary initial value

optimizer = tf.keras.optimizers.SGD(eta)
def calc_y():
	y=tf.pow(x,2.0)-4.0*x+4.0
	return y

for i in range(50):
	optimizer.minimize(calc_y, [x])
	print("iteration:",i, "x:", x.numpy(), "y:", calc_y().numpy())
```

Put these changes into your notebook as a new script, under "Exercise1 version 3"

Using a built-in optimizer

- Built-in optimizer

```
optimizer = tf.keras.optimizers.SGD(eta)
```

  - SGD=Stochastic Gradient Descent
  - Other optimizers are available
- How does it know what variables to optimise?

```
optimizer.minimize(calc_y, [x])
```

Exercise1, Version 3:
1. Modify your code to use **`tf.keras.optimizers.SGD`**
2. Check it behaves identically as before.

Using a built-in optimizer

```
optimizer = tf.keras.optimizers.SGD(eta)
```

- Other built-in optimizers work better with neural networks:

```
optimizer = tf.keras.optimizers.Adam()
```

```
optimizer = tf.keras.optimizers.RMSProp()
```

More on Gradient Descent

Gradient descent is heavily used in neural--network optimisation.

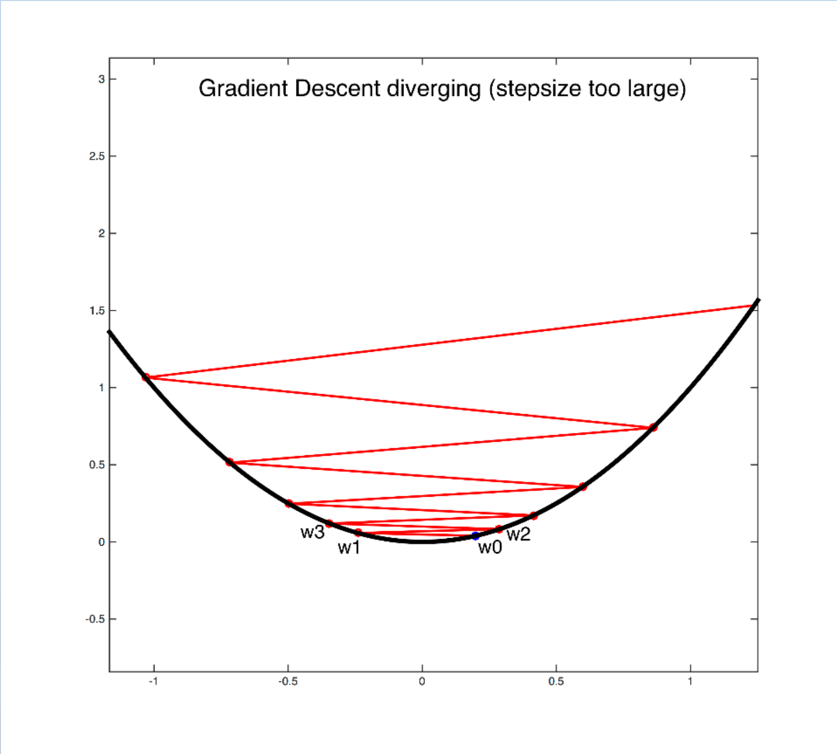


Image source: http://www.cs.cornell.edu/courses/cs4780/2017sp/lectures/images/gradient_de

Gradient Descent Local Minima

- Gradient descent will only head to the nearest local minimum

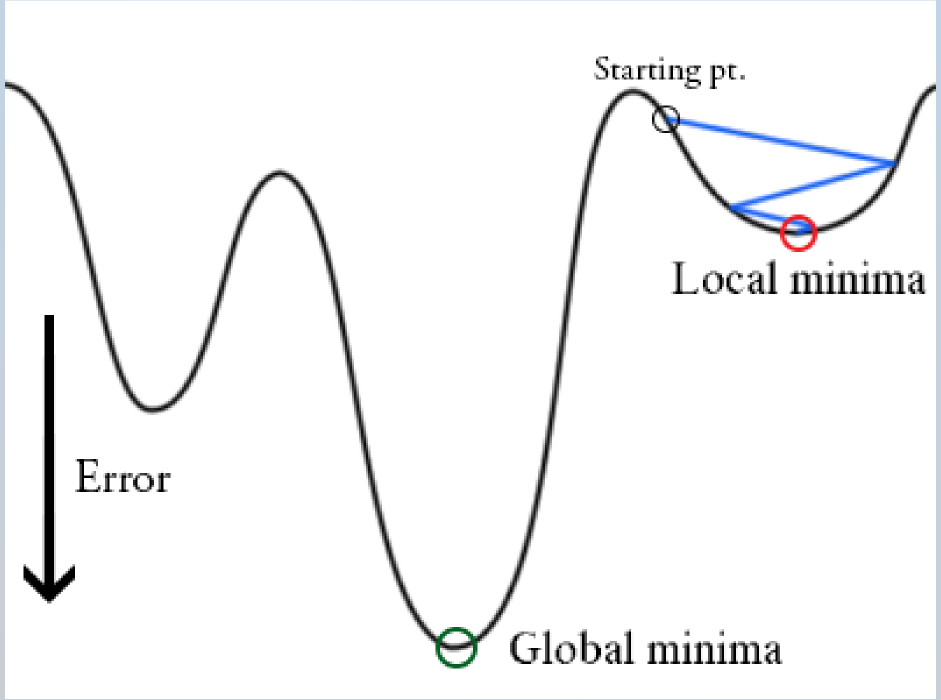


Image source: https://static.thinkingandcomputing.com/2014/03/bprop.png

- Might need several attempts from different random starting positions

Multidimensional Gradient Descent

- Gradient descent works too in multidimensional search spaces

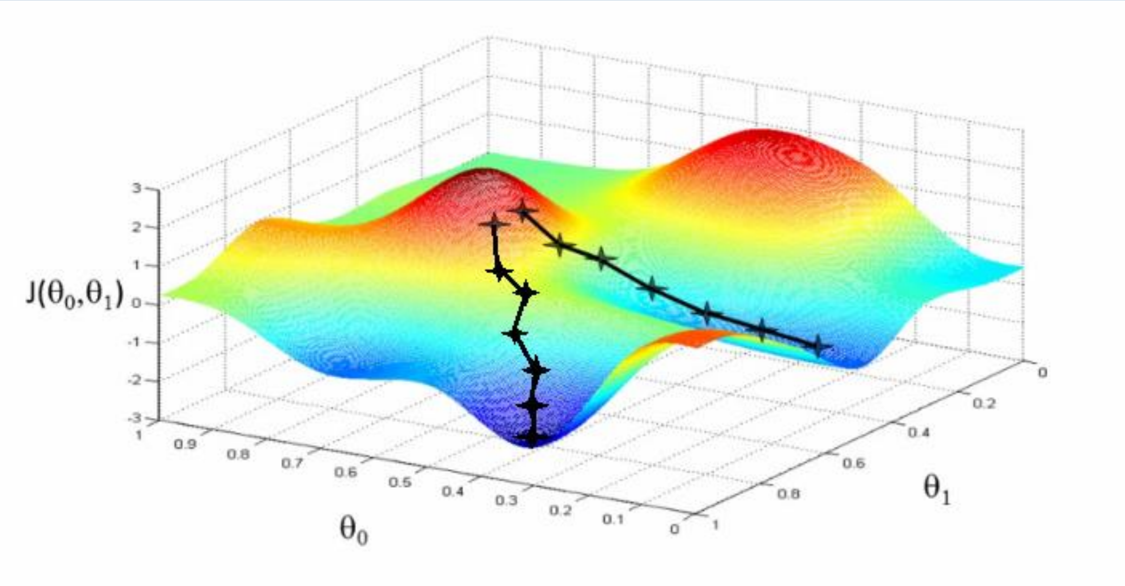


Image source: http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

$$w_{t+1} = w_t - \eta \frac{\partial J}{\partial w}$$