BCF mini course: Deep Learning and Macro-Finance Models

Goutham Gopalakrishna

École Polytechnique Fédérale de Lausanne (EPFL)

Swiss Finance Institute (SFI)

VSRC, Princeton University

February, 2023

Princeton University

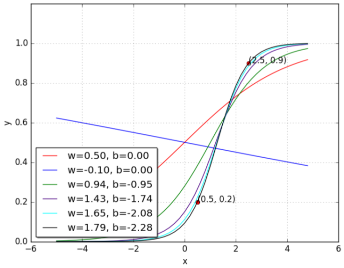
Roadmap

- Part-1: Introduction to numerical methods, challenges faced by traditional methods
  - Why neural networks and deep learning
  - Function approximators
  - Comparison with existing methods
- Part-2: Deep learning principles, high-dimensional optimization techniques in machine learning
  - Gradient descent and variants
  - Under the hood: Activation functions, Parameter initialization
  - Object oriented programming principles
- Part-3: Application to solve macro-finance models with aggregate shocks

References

- Textbooks:
  1. Raul Rojas. Neural Networks: A Systematic Introduction. 1996
  2. Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. An MIT Press book. 2016
- Other sources
  1. Dive into deep learning (interactive learning material)
  2. Machine learning for macroeconomics (teaching slides) by Jesús Fernández-Villaverde
  3. Neural networks (teaching slides) by Hugo Larochelle
  4. Deep learning CS6910 (teaching slides) by Mitesh Khapra

Learning by trial and error



Let us try some other values of w, b

w	b	$\mathscr{L}(w,b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

More principled way of doing this guesswork is what learning is all about!

Gradient descent

- Gradient descent rule: Move in the direction of gradient
- Parameter update equations

$$w_{t+1} = w_t - \eta \nabla w_t \tag{1}$$

$$b_{t+1} = b_t - \eta \nabla b_t \tag{2}$$

where

$$\nabla w_t = \frac{\partial \mathcal{L}(w, b)}{\partial w} \tag{3}$$

$$\nabla b_t = \frac{\partial \mathcal{L}(w, b)}{\partial b} \tag{4}$$

evaluated at $w = w_t, b = b_t$, and t is the iteration number.

- The update equation is sometimes written simply as

$$w = w - \eta \nabla w$$

and similarly for b

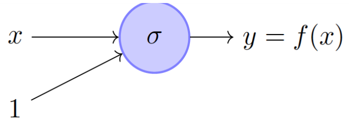
Gradient descent algorithm

```
$t \leftarrow 0$
$max\_iter \leftarrow 1000$
**while** $t < max\_iter$ **do**
    $w_{t+1} \leftarrow w_t - \eta \nabla w_t$
    $b_{t+1} \leftarrow b_t - \eta \nabla b_t$
    $t \leftarrow t + 1$
**end**
```

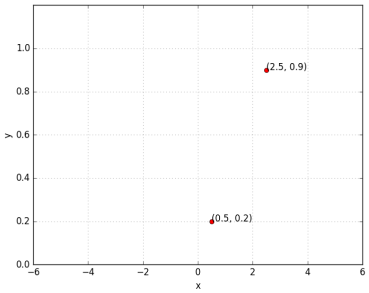
Algorithm 1: Gradient descent algorithm.

How to obtain ∇w_t and ∇b_t?

Gradient descent



$$f(x) = \frac{1}{1+e^{-(w\cdot x+b)}}$$



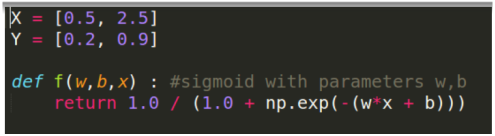
- Let's assume that there is only one point to fit (x, y)

$$\mathcal{L}(w,b) = 0.5 * (f^{ANN}(x) - y)^2$$
$$\nabla w = \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial}{\partial w}(0.5 * (f^{ANN}(x) - y)^2)$$
$$\dots$$
$$\nabla w = (f^{ANN}(x) - y) * f^{ANN}(x) * (1 - f^{ANN}(x)) * x$$

- For two points,

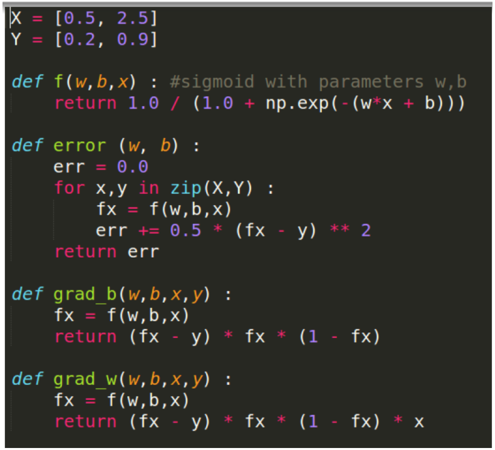
$$\nabla w = \sum_{i=1}^{2}(f^{ANN}(x_i) - y_i) * f^{ANN}(x_i) * (1 - f^{ANN}(x_i)) * x_i$$
$$\nabla b = \sum_{i=1}^{2}(f^{ANN}(x_i) - y_i) * f^{ANN}(x_i) * (1 - f^{ANN}(x_i))$$

Gradient descent



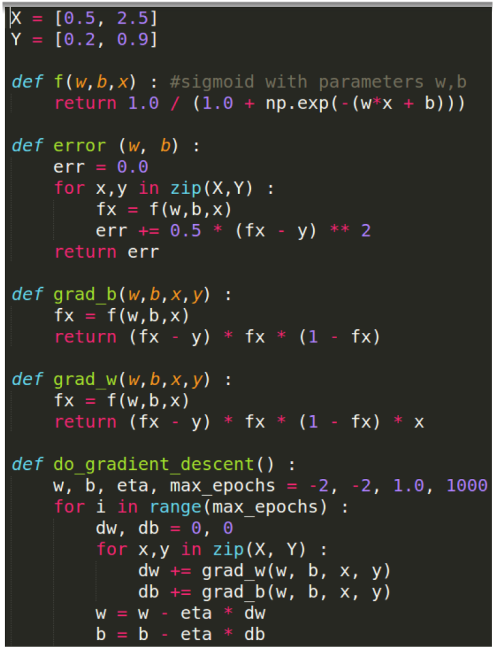
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))
```

Gradient descent



```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x
```

Gradient descent



```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

Momemtum gradient descent

- Navigating plateaus take a lot of time since gradients are small
- Momentum based gradient descent fixes the problem
- If you are being repeatedly asked to move in the same direction, then it is a good idea to take bigger steps in that direction

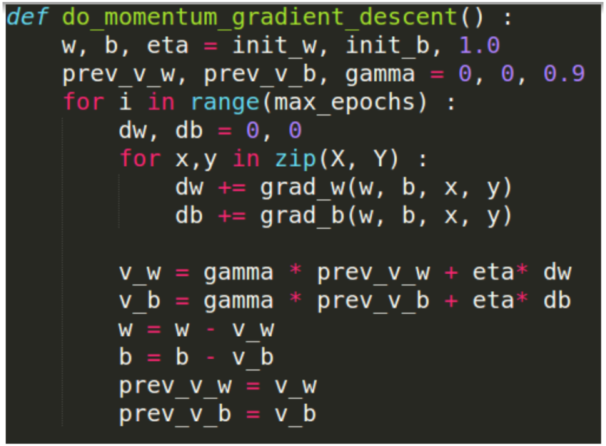
$$\begin{aligned} u_t &= \beta u_{t-1} + \nabla w_t \\ w_{t+1} &= w_t - \eta u_t \end{aligned}$$

After some algebra, we have

$$u_t = \sum_{\tau=0}^{t} \beta^{t-\tau} \nabla w_\tau$$

That is, u_t is the exponentially weighted average of current and all past gradients

Momemtum gradient descent



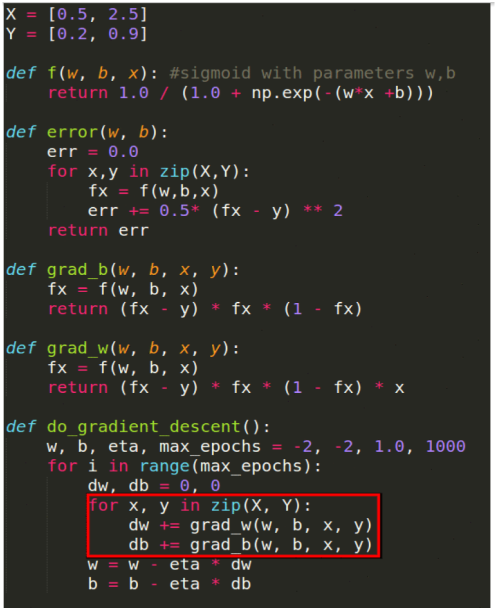
```
def do_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        v_w = gamma * prev_v_w + eta* dw
        v_b = gamma * prev_v_b + eta* db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```

Nesterov accelerated descent

- Look ahead before you descend
- The update rule is as follows

$$\begin{aligned} w_{look\_ahead} &= w_t - \gamma update_{t-1} \\ update_t &= \gamma * update_{t-1} + \eta \nabla w_{look\_ahead} \\ w_{t+1} &= w_t - update_t \end{aligned}$$

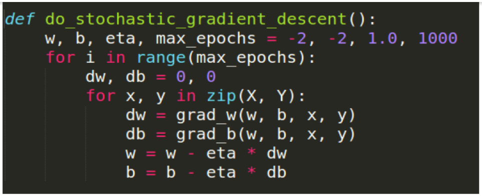
Stochastic gradient descent



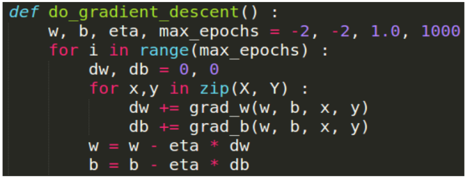
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- In gradient descent, the gradients are computed as the summation of gradients at all points
- Updating the parameters this way is costly especially in large datasets
- An alternative is to update for each data point $\Longrightarrow$ Stochastic gradient descent

Stochastic gradient descent



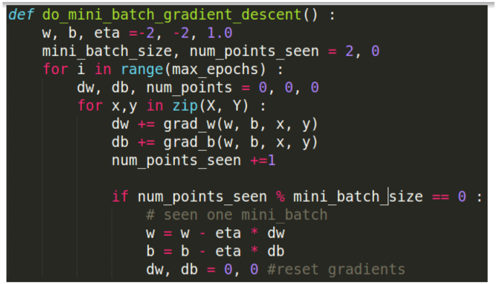
```
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```



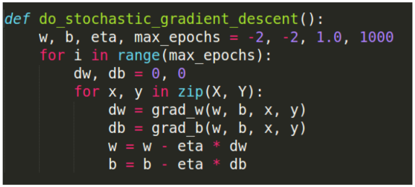
```
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that in the stochastic gradient descent, the parameters are updated for each data point
- The computed gradients are therefore approximations
- This makes the descent stochastic. This is because at each point, the parameters are updated in the direction most favourable to it, without being concerned about other points
- There is no guarantee that at each step the loss is reduced
- Sometimes, the oscillations can be wild. How can we reduce these oscillations? We can use mini-batch gradient descent

Mini-batch gradient descent



```
def do_mini_batch_gradient_descent() :
    w, b, eta =-2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini_batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```



```
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- In gradient descent, the the parameters are updated after seeing all data points
- In stochastic gradient descent, the parameters are updated for each data point
- In mini-batch gradient descent, the parameters are updated after seeing mini-batch number of data points

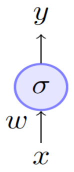
More variants

- Adagrad , RMSProp, Adam: Adjust the learning rate to make sure that parameters pertaining to sparse features get updated properly

Update rule for Adam

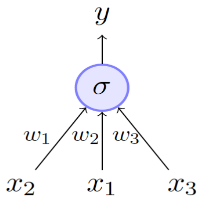
$$
\begin{aligned}
m_t &= \beta_t * m_{t-1} + (1-\beta_t) * \nabla w_t \\
v_t &= \beta_2 * v_{t-1} + (1-\beta_2) * (\nabla w_t)^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta_t}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t
\end{aligned}
$$

Backpropagation



- We saw how to train a network with no hidden layers and only one neuron

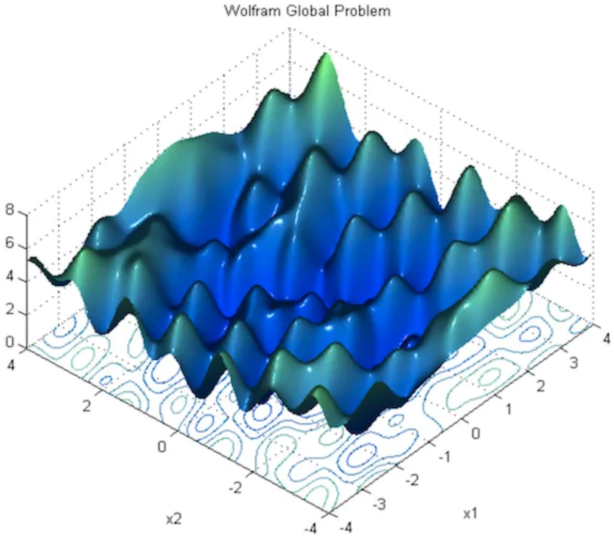
$$
\begin{aligned}
w &= w - \eta \nabla w \\
\nabla w &= \frac{\partial \mathcal{L}(\boldsymbol{w}}{\partial w} \\
&= (f(\boldsymbol{x}) - y) * f(\boldsymbol{x}) * (1 - f(\boldsymbol{x}) * x
\end{aligned}
$$



- Extension to a network with multiple input is straightforward

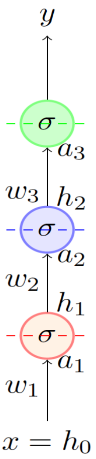
$$
\begin{aligned}
w_1 &= w_1 - \eta \nabla w_1 \\
w_2 &= w_2 - \eta \nabla w_2 \\
w_3 &= w_3 - \eta \nabla w_3 \\
\nabla w_i &= (f(\boldsymbol{x}) - y) * f(\boldsymbol{x}) * (1 - f(\boldsymbol{x})) * x_i
\end{aligned}
$$

Loss function



Source: Yoshua Bengio

Backpropagation



$$a_i = w_i h_{i-1}; h_i = \sigma(a_i)$$

$$a_1 = w_1 * x = w_1 * h_0$$

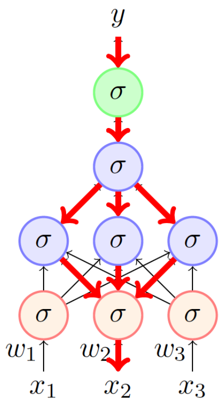
- With a deeper network, gradients are computed by backpropagation

$$\nabla w_1 = \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial y} \cdot \frac{\partial y}{\partial a_3} \cdot \frac{\partial a_3}{\partial h_2} \cdot \frac{\partial y}{\partial a_3} \cdot \frac{\partial h_2}{\partial a_2} \cdots \frac{\partial a_1}{\partial w_1}$$

- Extension to a network with multiple input is straightforward

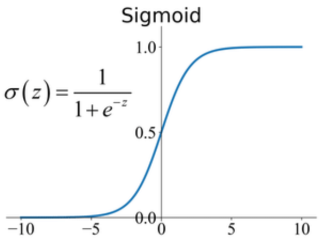
$$\begin{aligned} w_1 &= w_1 - \eta \nabla w_1 \\ w_2 &= w_2 - \eta \nabla w_2 \\ w_3 &= w_3 - \eta \nabla w_3 \\ \nabla w_i &= (f(\boldsymbol{x}) - y) * f(\boldsymbol{x}) * (1 - f(\boldsymbol{x})) * x_i \end{aligned}$$

Backpropagation

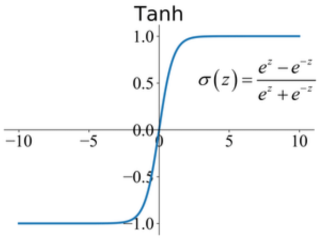


- With a deeper and wider network, gradients are computed by backpropagation across multiple paths
- other than this, the principles remain the same

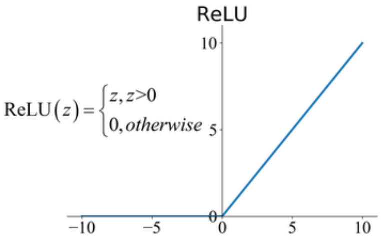
Activation functions



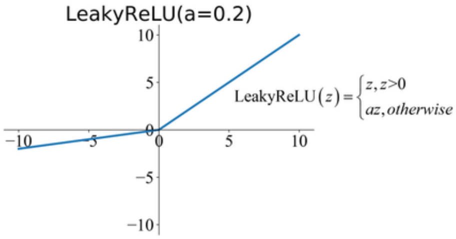
(a)



(b)



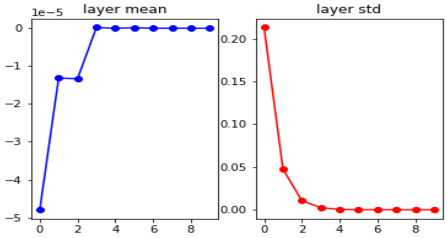
(c)



(d)

Weight initialization

An example with 10 layers, 500 neurons in each layer, and tahn activation function



- Parameters distributed normally in the initial layer (W_1=randn(inputDim,outputDim))
- Compute output in each later as $h_i = \sigma(W_i h_{i-1})$, where i is the layer number and h_{i-1} is the input
- Distribution of output values collapses in the interior layers. Learning is shut down as a result

$$W = W - \eta * \nabla W$$

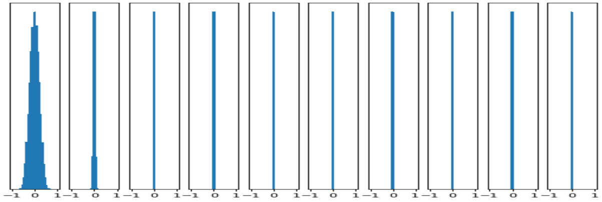
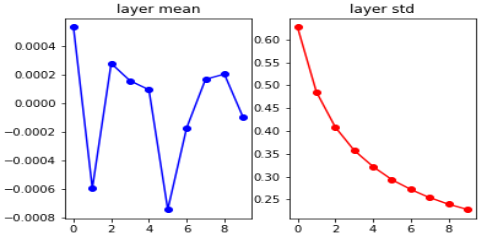


Figure: Distribution of output h_i in each layer

Weight initialization

An example with 10 layers, 500 neurons in each layer, and tahn activation function



- Parameters are Xavier-initialized in initial layer (W_1=randn(inputDim,outputDim)/sqrt(inputDim))
- Intuitively, more number of input dimensions require smaller weights ($h_i = \sigma(W_i h_{i-1}) = \sigma\big(\sum_i w_i^j h_{i-1,j}\big)$)
- Distribution of output values converge to normal distribution in interior layers
- Learning is restored!

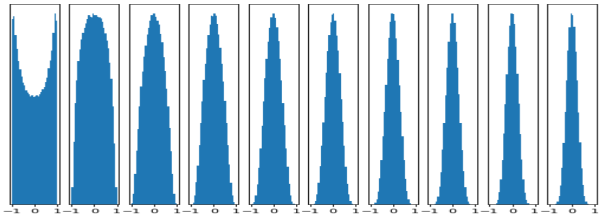


Figure: Distribution of output h_i in each layer

Object oriented programming

- Objected oriented programming is used to create neat and reusable code
- Core principles are
  1. Classes and objects
  2. Inheritance
  3. Polymorphism
  4. Encapsulation
  5. Abstraction
- We will focus on classes and objects, and encapsulation
- A class is a collection of objects

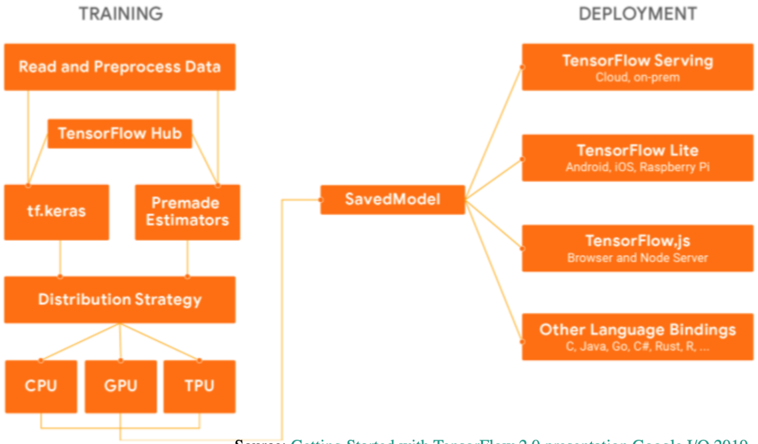
Object oriented programming

```python
class Model:
    def __init__(self,params):
        self.params = params
        #initialize some stuff
    def ComputeEquilibrium(self,optimalParameters=None):
        #compute prices and policies

params = {'gamma':2,'ah':0.05}
model1 = Model(params)
model1.ComputeEquilibrium()

params['ah'] = 0.1
model2 = Model(params)
model2.ComputeEquilibrium()
```

Decorators

```
    def calculate_time(func):
        #the inner1 function takes arguments through *args and **kwargs

        def inner1(*args, **kwargs):

            # storing time before function execution
            begin = time.time()

            func(*args, **kwargs)

            # storing time after function execution
            end = time.time()
            print("Total time taken in : ", func.__name__, end - begin)

        return inner1




    # Let's write a function to compute factorial and wrap it with decorator
    @calculate_time
    def factorial(num):
        print(math.factorial(num))

    # calling the function.
    factorial(10)
```

Tensorflow



Source: Getting Started with TensorFlow 2.0 presentation Google I/O 2019

Tensorflow

Graph construction

```
A = tf.constant(1, dtype = tf.float32)

B = tf.constant(2, dtype = tf.float32)

y = tf.add(A,B)

print(y)






#No evaluation has yet taken place
>> Tensor("Add:0", shape=(), dtype=float32)
```

```
with tf.Session() as sess:
    print(sess.run(y))
```

Tensorflow

```python
def inner_function(x, y, b):
  x = tf.matmul(x, y)
  x = x + b
  return x

# Use the decorator to make `outer_function` a `Function`.
@tf.function
def outer_function(x):
  y = tf.constant([[2.0], [3.0]])
  b = tf.constant(4.0)

  return inner_function(x, y, b)

# Note that the callable will create a graph that
# includes `inner_function` as well as `outer_function`.
outer_function(tf.constant([[1.0, 2.0]])).numpy()
```