IMPROVING DEEP NEURAL NETWORKS: HYPERPARAMETER TUNING, REGULARIZATION AND OPTIMIZATION

By:
Satish Deshbhratar

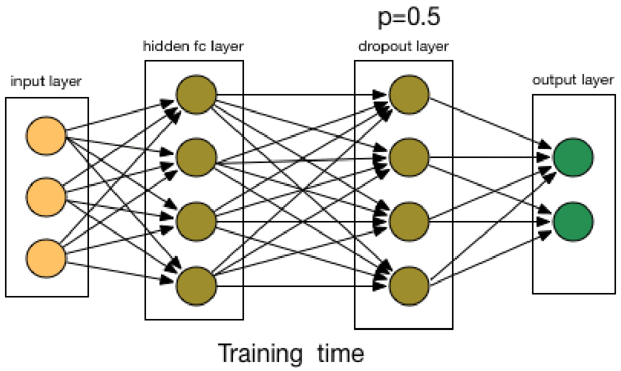
BIAS / VARIANCE

- Applied ML: Iterative process to detect the hyperparameters
- Hyperparameters: learning rate, #iterations, #hidden layers/units…
- 100 → 1,000,000 : 60% training set – 20% development (cross validation) set – 20% test set
- 1,000,000+: 98% training set – 1% dev set – 1% test set
- PS: Mismatched dev / test distribution is bad. They have to come from the same distribution.
- PS: dev set is essential, but test set is not.
- High bias → Underfitting → High error in train → Train: 15% / Test: 16% (linear in a region)
- High variance → Overfitting → High error in difference → Train: 1% / Test: 11% (flexibility in a region)
- Bias variance tradeoff: They were used to be related inversely.
- High bias solutions: Bigger network / More layers / NN architecture / Different model / Longer train
- High variance solutions: More data / Regularization / NN architecture / Different model

REGULARIZATION

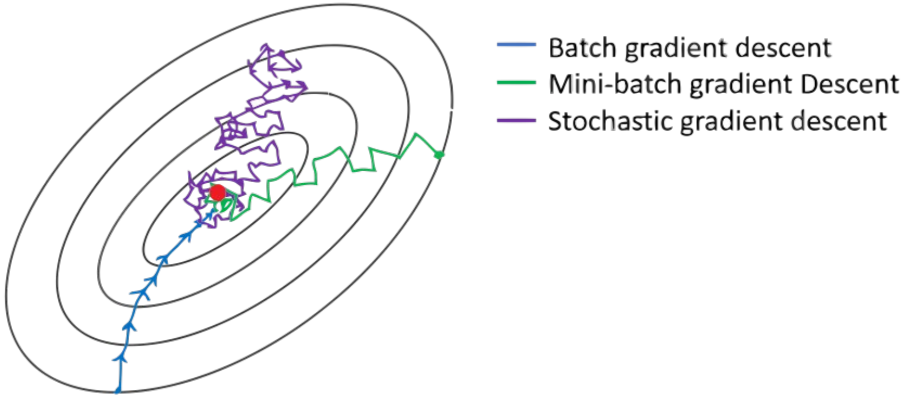
- $L2\ Regularization: \frac{\lambda}{2m}||w||_2^2 = \frac{\lambda}{2m}W^T.W$
- $L1\ Regularization: \frac{\lambda}{2m}||w||_1^1$
- $\boldsymbol{Frobinus\ Regularization}: \frac{\lambda}{2m}||W^{[l]}||_F^2 = \frac{\lambda}{2m}\sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}} w_{ij}^{[l]2}$
- → New backpropagation: $dW^{[l]} = Old\ term + \frac{\lambda}{m}w^{[l]}: Weight\ decay$
- Regularization vs overfitting: $\lambda \rightarrow +\infty, w^{[l]} \rightarrow 0, \Rightarrow$ $Logistic\ regression\ (Single\ NN)$
- Large λ → smoother decision boundry / smaller weights / remove over fitting

DROPOUT



- Concept: Remove some hidden units each iteration based on probability (inverted dropout)
- d3=np.random.rand(a3.shape[0],a3.shape[1]) < keep_prob #Generate 0's
- a3=a3*d3=np.multiply(a3,d3) #Element-wise multiplication
- a3=a3/keep_prob #Keep the values on the same scale ➔ Shrink weights
- PS: Each layer can have its own keep_prob; keep_prob=1 in first and last layer
- PS: Smaller keep_prob → More weights → Tend to overfit more
- PS: Dropout is only used in training

MINI-BATCH GRADIENT DESCENT

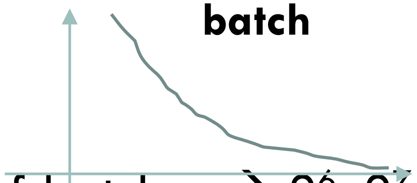


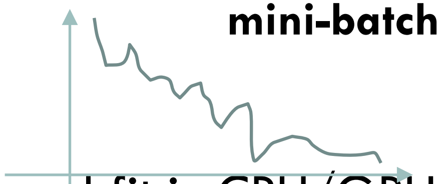
- Iterative model = Empiric model = Computational model
- <u>Batch gradient descent</u>: Mini-batch gradient descent with m #<u>Too long per iteration</u>
- <u>Stochastic gradient descent</u>: Mini-batch gradient descent with 1 #<u>Lose vectorization speed</u>, never converge
- $X \in (n_x, m)$; $Y \in (1,m)$; $X=[\underbrace{X^{(1)},\ldots,X^{(1000)}}_{X^{\{1\}}\ (n_x,1000)\text{: First mini-batch}},X^{(1001)},\ldots,X^{(2000)}\ldots]$
- for t=1,…,number of mini-batches: (each iteration, as if m=1000)
  - forward prop on $X^{[t\}}$
  - compute cost $J^{\{t\}} = \frac{1}{1000}\ldots$
  - backward prop

1) Shuffle randomly
2) Partition
PS: If it is not divisible by the mini-batch size, the last one will have smaller values

```
permutation = list(np.random.permutation(m))
shuffled_X = X[:, permutation]
shuffled_Y = Y[:, permutation].reshape((1,m))
```

- <u>Epoch</u> = Pass through training set = Iteration
- PS: X and Y are similarly split
- PS: Bigger learning rate for stochastic → Less noise

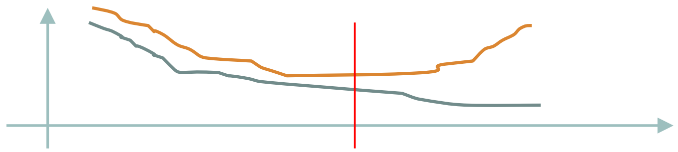




- m<2000 → Batch gradient descent ; Typical size for number of batches → 2^6, 2^7, 2^8… and fit in CPU/GPU
- Number of batches increase → Noise decrease

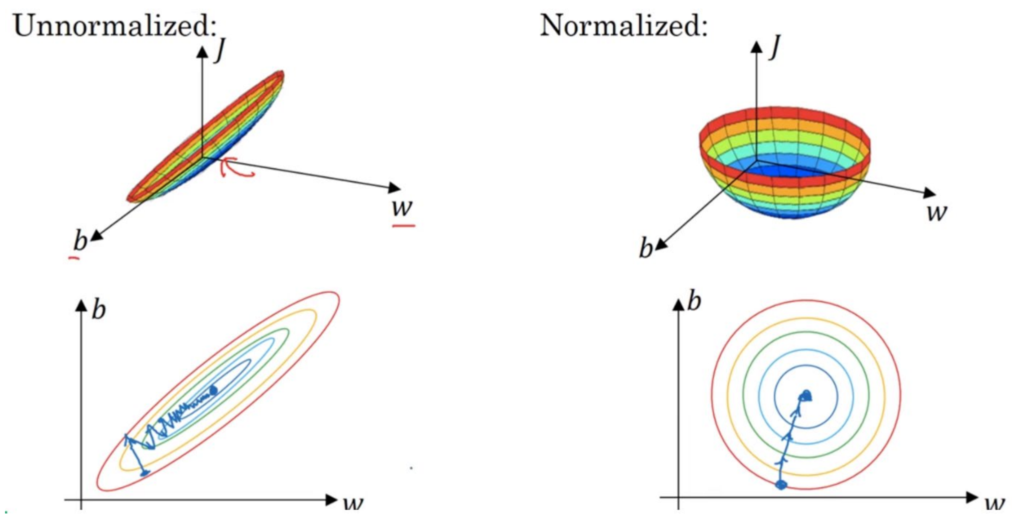
OTHER REGULARIZATION TECHNIQUES

- Data augmentation: Flip images horizontally, do random modifications, make distortions
- Early stopping:



- PS: A=(A<0,5) #Generate 0's and 1's

NORMALIZATION



- $\mu = \frac{1}{m}\sum_{i=1}^{m} X^{(i)}\,; X = X - \mu$ #Substract mean
- $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} {X^{(i)}}^2\,; X = \frac{X}{\sigma^2}$ $\#Normalize\ variance$
- PS: It must be the same for the train / test ➔ Faster optimization (symmetric distribution)

INITIALIZATION

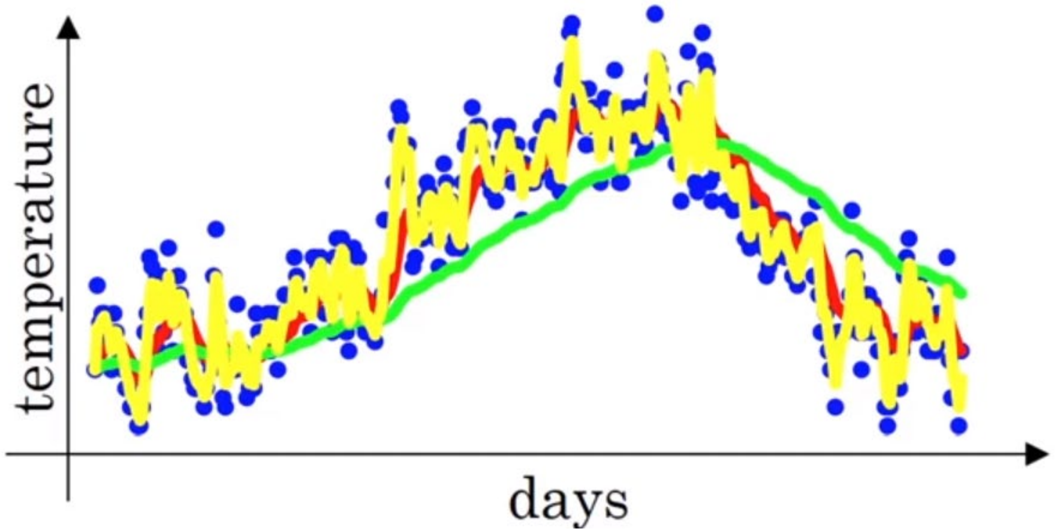
- Problem of Vanishing / Exploding gradients: Occurs when your derivatives are small/big
- Random initialization for W's to break symmetry; zeros for b's
- He initialization: $W^{[l]}$ = np.random.randn(shape)*np.sqrt($\frac{2}{n^{[l-1]}}$) #for RELU
- Another initialization: $W^{[l]}$ = np.random.randn(shape)*np.sqrt($\frac{1}{n^{[l-1]}}$) #for Tanh
- Xavier initialization: $W^{[l]}$ = np.random.randn(shape)*np.sqrt($\frac{1}{n^{[l-1]}+n^{[l]}}$)

GRADIENT CHECKING

- Purpose: Find error in backpropagation implementation (gradient calculation) (slower)
- $w^{[1]}, b^{[1]}$.. ➔ Reshape in a big θ vector
- $dw^{[1]}, db^{[1]}$.. ➔ Reshape in a big dθ vector
- $J(w^{[1]}, b^{[1]}, dw^{[1]}, db^{[1]}\ldots)$ ➔ J(θ)
- for i: $d\theta_{approx}(i) = \frac{J(\theta_1,..,\theta_i+\epsilon,..)-J(\theta_1,..,\theta_i-\epsilon,..)}{2\,\epsilon} = d\theta(i) = \frac{\partial J}{\partial \theta_i}$
- Check: $\frac{\left|\left|d\theta_{approx}-d\theta\right|\right|_2}{\left|\left|d\theta_{approx}\right|\right|_2+\left|\left|d\theta\right|\right|_2} \leq \epsilon = 10^{-7}$
- PS: Grad check only in debug (not in train); Run grad check without dropout
- If grad check fails, look at components $db^{[l]}$ and $dw^{[l]}$
- $\sum_k \sum_j {w_{j,k}^{[l]}}^2 \Leftrightarrow$ np.sum(np.square(w)) ; $\left|\left|x\right|\right|_2 \Leftrightarrow$ np.linalg.norm(x)

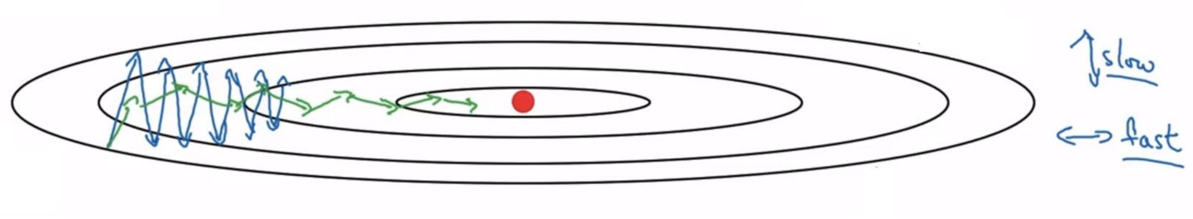
EXPONENTIALLY WEIGHTED AVERAGE

- $V_0 = 0; V_t = \beta V_{t-1} + (1-\beta)\theta_t$: $\boldsymbol{Averaging}$ $over\ last\ \frac{1}{1-\beta}\ days'\ temperature$
- β $\boxed{\nearrow}$ → Average graph becomes smoother and shift slightly to the right
- <u>Bias correction:</u> $\frac{V_t}{1-\beta^t}$ because V_t will be far from θ_1 at first



GRADIENT DESCENT WITH MOMENTUM

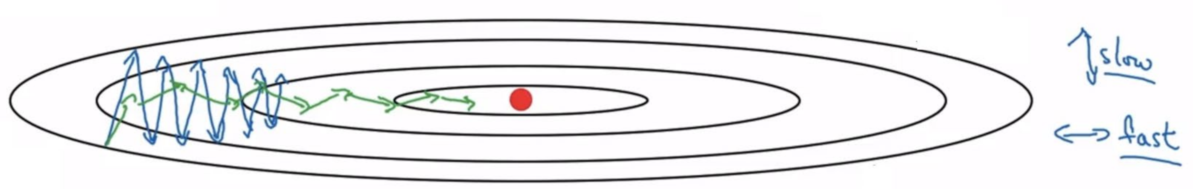
- For t:
  - Compute dw, db on current mini-batch
  - $v_{dw} = \beta v_{dw} + (1-\beta) v_{dw}$
  - $v_{db} = \beta v_{db} + (1-\beta) v_{db}$
  - $w = w - \alpha v_{dw}; b = b - \alpha v_{db}$



- We will be averaging on the <u>vertical</u> (<u>around 0</u>) and on the horizontal (<u>straight forward</u>)
- PS: Bias correction isn't needed because after few iterations we will be okay
- PS: β=0,9 in practice
- Another formulation:
  - $v_{dw} = \beta v_{dw} + v_{dw}$
  - $w = w - \frac{\alpha}{1-\beta} v_{dw}$
- PS: Momentum can be applied with any GD method

RMSPROP

- <u>RMSprop </u>= Root Mean Squared Propagation (Same objective as momentum)
- For t:
  - Compute dw, db on current mini-batch
  - $s_{dw} = \beta s_{dw} + (1-\beta)dw^2$ #Element-wise power
  - $s_{db} = \beta s_{db} + (1-\beta)db^2$
  - $w = w - \alpha \frac{dw}{\sqrt{s_{dw}}+\epsilon}$; $b = b - \alpha \frac{db}{\sqrt{s_{db}}+\epsilon}$ #We can get a bigger α now; ϵ to avoid divergence



ADAM

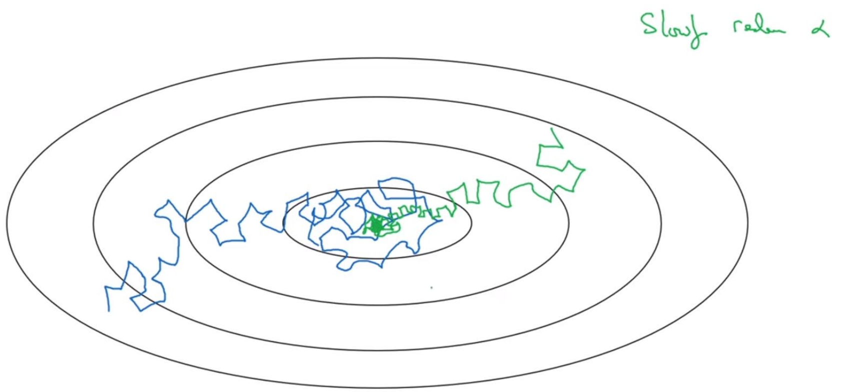
- ADAM = Momentum + RMSprop

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
  # can be mini-batch or batch gradient descent
  compute dw, db on current mini-batch

  vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
  vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

  sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
  sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

  vdW = vdW / (1 - beta1^t)      # fixing bias
  vdb = vdb / (1 - beta1^t)      # fixing bias

  sdW = sdW / (1 - beta2^t)      # fixing bias
  sdb = sdb / (1 - beta2^t)      # fixing bias
```

- α needs to be tuned
- $\beta_1 = 0{,}9$ (Momentum)
- $\beta_2 = 0{,}999$ (RMSprop)
- $\epsilon = 10^{-8}$ (Doesn't matter much)

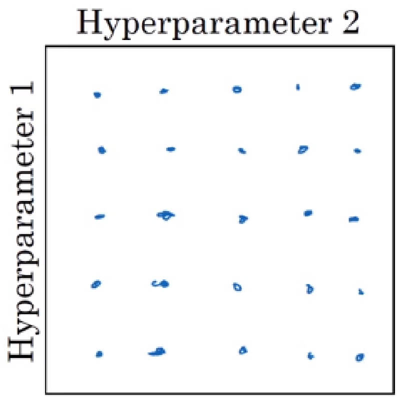
LEARNING RATE DECAY

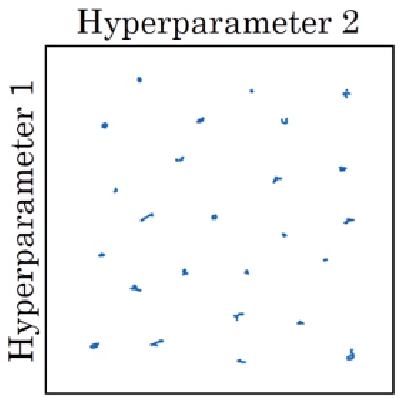
- <u>Concept</u>: At first, you can have a bigger value of α and in the end you can make it smaller
- $\alpha = \frac{1}{1+decay\ rate * epoch\ number}\alpha_0$
- $\alpha = 0{,}95^{epoch\ number}\ \alpha_0$
- $\alpha = \frac{k}{\sqrt{epoch\ number}}\alpha_0$



- We can use <u>manual decay</u> too
- It is not evident to find a <u>local optimum</u> (derivate = 0) in a high dimensional space
- Saddle points are not really a problem
- <u>Plateau</u>: Region where the derivate is close to 0 for a long time (slows the algorithm)

HYPERPARAMETERS





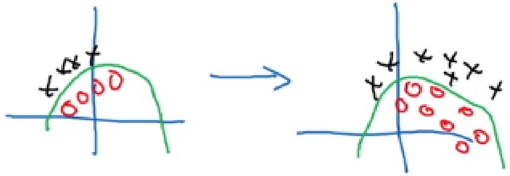
- PS: We try 25 couples of hyperparameters
- 1st priority: α
- 2nd priority: β momentum, #hidden units, mini-batch size
- 3rd priority: #layers, learning rate decay
- Course to fine search process: If a couple of hyperparameters is good, we zoom in their region and pick more random values
- Appropriate scale for hyperparameters: α for example goes from 0,0001 to 1, we choose a logarithmic scale for a better distribution: r=(log a - log b)*np.random.rand() + log b → $\alpha=10^r$
- β=0,9000 → 0,9005 ~ Generate 10 values
- β=0,9990 → 0,9995 ~ Generate 1000 values
- PS: Another method is: r=np.random.rand(interval)

➔ Linear scale is bad

- Hyperparameters approach: Panda (Train one model at a time) vs Caviar (Train many models in parallel)
- PS: We use Caviar approach if we have high computational resources

BATCH NORMALIZATION

- For $z^{(i)}$:
  - $\mu = \frac{1}{m}\sum_{i=1}^{m} Z^{(i)}$; $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} {Z^{(i)}}^2$;
  - $Z_{norm}^{(i)} = \frac{Z^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}}$ #ϵ to avoid division by zero
  - $\tilde{Z}^{(i)} = \gamma Z_{norm}^{(i)} + \beta$ #To manipulate the mean and the variance #γ and β are <u>learnable parameters</u>
  - $Use\ \tilde{Z}^{(i)}\ instead\ of\ Z^{(i)}\ to\ make\ inputs\ belong\ to\ other\ distrubtion$
- We use batch normalization in mini-batches
- The parameter b becomes 0, since mean(Z)=0 → <u>No need for b parameter</u>
- Shape of β and γ is ($n^{[l]}$,1)
- We can use Momentum, RMSprop and ADAM with Batch Normalization



- <u>Covariate shift problem</u>: Training your algorithm on black cats → It will be bad with colored cats
- → BN reduces this problem by guaranteeing that they will have mean 0 and variance 1: Stability
- BN has a slight regularization effects (not recommended). BN adds a slight noise.
- In test, we use BN of all the layers in the training using exponentially weighted average

MULTICLASS CLASSIFICATION – SOFTMAX REGRESSION

- C=#Classes {0,1,3,4} ➔ Output layer will have C units ($n^{[L]}$=C)
- → Last layer=Output layer=<u>Softmax layer</u>: $\begin{bmatrix} p(C=0/X) \\ p(C=1/X) \\ p(C=2/X) \\ p(C=3/X) \end{bmatrix}; \sum_{i=1}^{C} p(C=i/X) = 1$
- <u>Activation function</u>=<u>Softmax activation function</u>: $a_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{j=1}^{C} (e^{Z^{[L]}})j}$
- PS: Usually activation functions take floats and return floats, here operation is on vectors
- Softmax $\begin{matrix} 0,8 \\ 0,1 \\ 0,05 \\ 0,05 \end{matrix}$ vs Hardmax $\begin{matrix} 1 \\ 0 \\ 0 \\ 0 \end{matrix}$ ➔ Softmax is <u>generalization</u> of sigmoid with C ≠ 2
- <u>New loss function</u>: <u>Likelyhood function</u>: $L(\hat{y}, y) = -\sum_{j=1}^{C} y_j log \hat{y}_j$
- <u>New backpropagation (gradient descent)</u>: $dZ^{[L]} = \hat{y} - y$

TENSORFLOW (1)

- **Framework** = Library that contain DL functions (Caffe, Kera, Tensorflow)
- **Choice**: ease of programming (dev and deployment), running speed, truly open (open source + good governance{will stay open source})
- import tensorflow as tf
- w=tf.Variable(value,dtype=tf.float32) #Define a parameter to optimize
- cost_function = #Function of w here
- train=tf.train.GradientDescentOptimizer(learning_rate).minimize(cost_function)#Define learning algo
  - init=tf.global_variables_initializer()
  - session=tf.Session() #Start a tf session
  - session.run(init) #Initialize variables
  - session.run(w) #Run session to calculate w

Idiomatic lines (Always in your code)

- for i in range(numberOfIteration): session.run(train) #Calculate grad desc

TENSORFLOW (2)

- PS: We prepare forward for Tensorflow, and it calculates backward for us
- constant=tf.constant(value)
- x=tf.placeholder(tf.float32,size) #A variable to assign value to, later
- → session.run(train,feed_dict={x:valuesToAssign,y:v}) #Get data in cost_function i.e. in train
- PS: float 5. ≠ int 5 (in Python)
- with tf.Session as session:
  - session.run(init)
  - Print(session.run(w))

Another way to write the idiomatic lines

- Computation graph: What happens in tensorflow: It can compute backprop thanks to graph
- PS: In tf, we can't print(a+b) just like that, we have to run a session to be able to print it
- tf.one_hot(Y,numberOfClasses,axis=0) #Turn Y to a classification matrix

TENSORFLOW (3)

- w=tf.get_variable(shape,initializer=tf.typeOfInitialization())
- PS: [n_x,None] dimensions ⇔ number of lines = n_x, number of colons = #idc
- tf.matmul(A,B) #A*B
- tf.nn.activationFunction(A) #Apply activationFunction on A
- tf.nn.softmax_cross_entropy_with_logits(logits=y,labels=z) #Compute cost
- _, a=#Something that return two outputs here #First return = #idc
- PS: Main classes in tensorflow: Tensors (variables) and operators (functions/methods)