Deep Learning

Big Data & Machine Learning Bootcamp - Keep Coding

Outline

1. Mini batch gradient descent
2. Gradient descent with momentum
3. Adam optimization algorithm
4. Hyperparameters
5. Softmax Regression
6. Deep Learning Frameworks

1. Mini batch gradient descent

With the implementation of gradient descent on your whole training set, what you have to do is, you have to process your **entire training set before you take one little step of gradient descent**.

However, this is not the best way to train your network. There is a much faster way to train the system if you let the **gradient descent to make some progress before you finish processing the whole dataset**:

Mini batch gradient descent!

Sources:
- Coursera

1. Mini batch gradient descent

Mini batch gradient descent:

The idea is to use baby training sets to perform gradient descent.

Mini-batch gradient descent

Say $b = 10, m = 1000.$

Repeat {

  for $i = 1, 11, 21, 31, \ldots, 991$ {

    $$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

    (for every $j = 0, \ldots, n$)

  }

}

Number of iterations

Parameters update using a mini batch

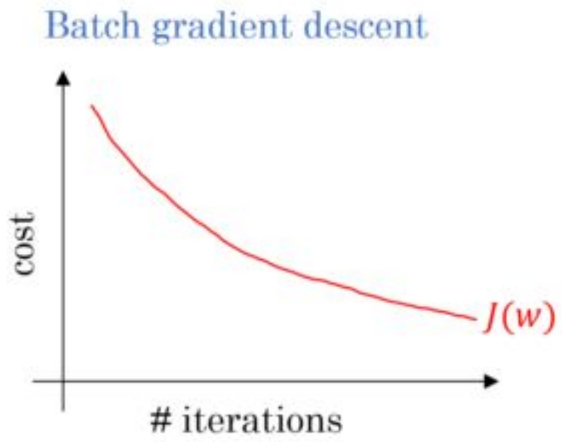
Sources:
- Coursera
- https://i.stack.imgur.com/h407Z.jpg

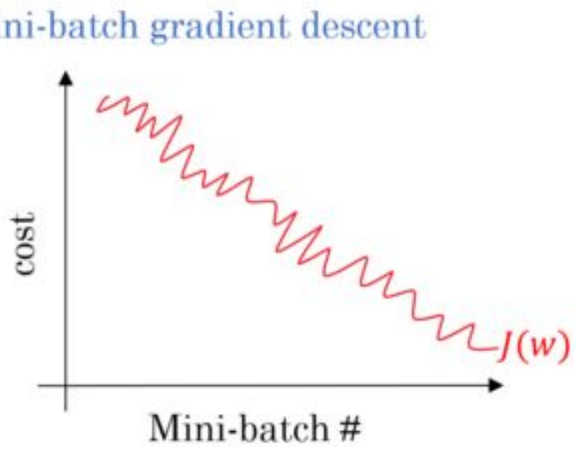
1. Mini batch gradient descent

One epoch is a single pass through the training set!

Using batch gradient descent, you only take one step per epoch.

Using mini batch gradient descent, you take #batches steps per epoch





Sources:
- Coursera
- https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

1. Mini batch gradient descent

One of the things to choose is the size of the mini batch.

If the mini batch is size = m (whole training set), then you are using **batch gradient descent**. The method we've been talking about up to now.

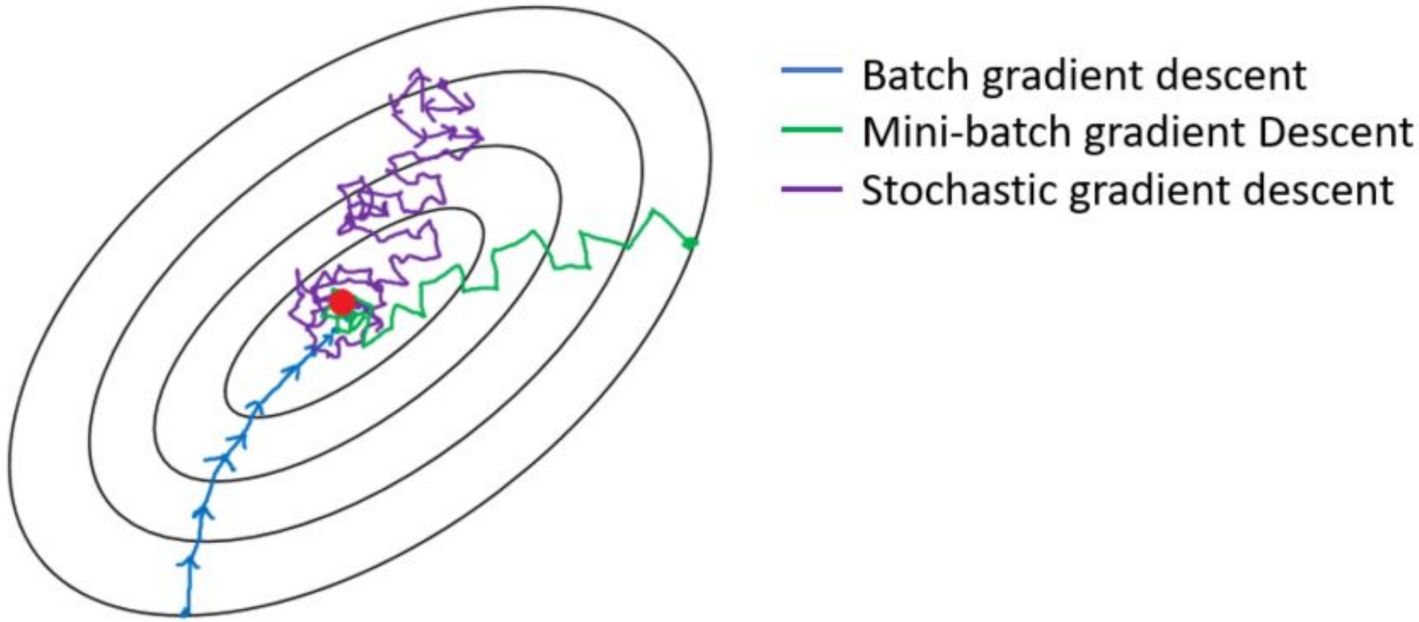
If the mini batch is size = 1, then you are using **Stochastic Gradient Descent.** Every samples in the training set is one batch

In practice you choose the batch size somewhere in between 1 and m (whole dataset)

Sources:
- Coursera

1. Mini batch gradient descent

Fastest learning occurs when using stochastic gradient descent!



Sources:
- Coursera
- https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

1. Mini batch gradient descent

Guidelines to choose your batch size:

If small training set (less than 1000 samples): Use batch gradient descent

If not, typical mini-batch sizes: 16, 32, 64, 128 …

Make sure the mini batch fits in the CPU/GPU you're using

Sources:
- Coursera

2. Gradient descent with momentum

They are a few of optimization algorithms that are faster than the gradient descent!

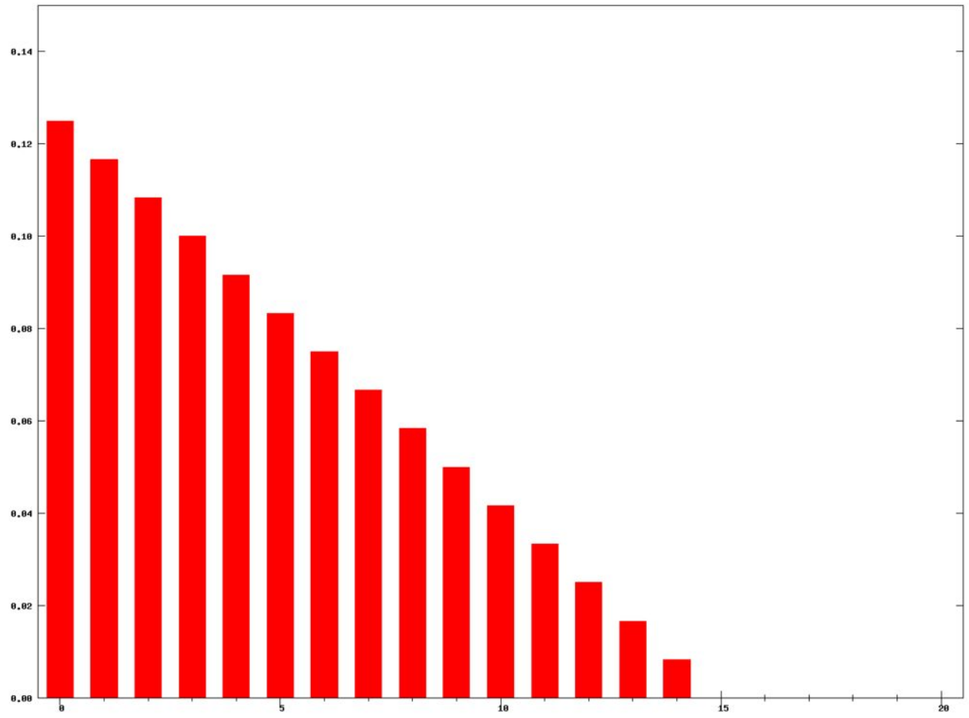
To understand those algorithms, we need to first understand what is **exponentially weighted averages** (also called exponentially moving averages)

They are a key component of faster optimization algorithms. Let's dive into that!

Sources:
- Coursera

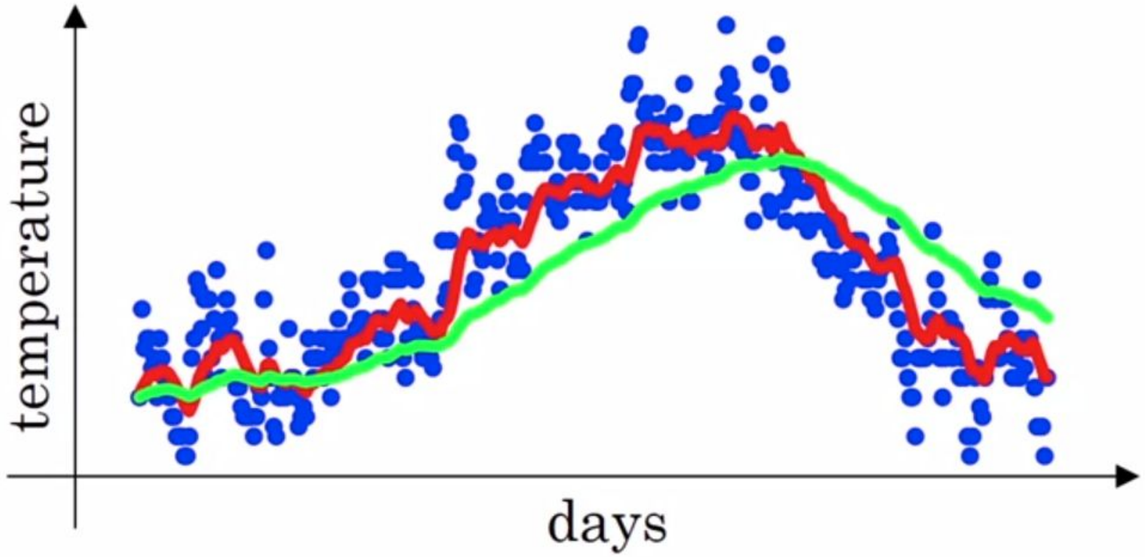
2. Gradient descent with momentum

Exponentially weighted averages is essentially a moving average method in which previous values are weighted/multiplied by an exponential decay value



Sources:
- Coursera
- https://en.wikipedia.org/wiki/Moving_average#/media/File:Weighted_moving_average_weights_N=15.png

2. Gradient descent with momentum



$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

Blue dots: Data

*Red line: $\beta = 0.9$ (**Intuition**: You focused on the last 10 values)*

*Blue line: $\beta = 0.98$ (**Intuition:** You focused on the last 50 values)*

Sources:
- Coursera

2. Gradient descent with momentum

The idea of exponentially weighted averages is used in the gradient descent with momentum, which almost always work faster than the standard gradient descent!

In one sentence, the basic idea is to compute an exponentially weighted average of the gradients and then use that gradient to update the weights!

For each iteration during training, you update the weights using the exponentially weighted average equation!

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1 - \beta) dW$$

$$v_{db} = \beta v_{db} + (1 - \beta) db$$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β $\qquad$ $\beta = 0.9$

Sources:
- Coursera

2. Gradient descent with momentum

In addition to the gradient descent with momentum algorithm, there is also another optimization algorithm called **RMSProp (root mean square prop)**

It is basically another algorithm that uses exponentially weighted averages to update the weights! It is called root mean square because you are squaring and computing the root of the derivatives to update the weights:

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

Sources:
- Coursera
- https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b

3. Adam optimization algorithm

Adam or Adaptive Moment Estimation is probably the most used optimization algorithm!

It is basically a **combination of RMSProp and gradient descent with momentum** optimization algorithms!

$$For\ each\ Parameter\ w^j$$

$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$
$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$
$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$
$\beta_1, \beta_2 : Hyperparameters$

Sorry for the change in the nomenclature here :)

The essence is that there are three hyperparameters when using Adam: learning rate, β_1 and β_2.

β_1 is commonly set to 0.9 and β_2 is commonly set in 0.99

Sources:
- Coursera
- https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

4. Hyperparameters

One of the things that might help speed up your learning algorithm, is to slowly reduce your learning rate over time. This is called **learning rate decay.**

$$newlr = 0.95^{epochNumber} * currentlr$$

$$newlr = \frac{currentlr}{\sqrt{epochNumber}}$$

$$newlr = \frac{currentlr}{1 + decayRate * epochNumber}$$

Learning rate is reduced by each epoch (pass through the whole training set)

Sources:
- Coursera

4. Hyperparameters

So far we have talked about many hyperparameters:

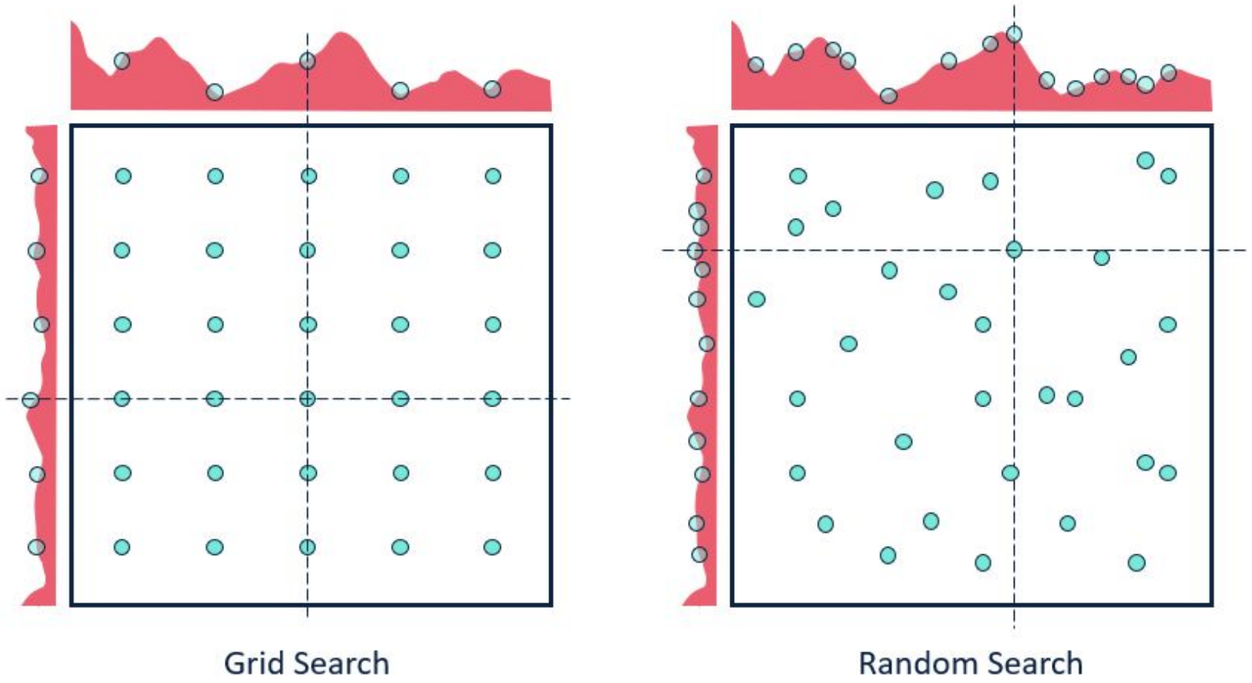
- Learning rate
- Momentum (~0.9)
- β_1, β_2 (Adam optimizer)
- Mini batch size
- Number of layers
- Number of hidden units
- Learning rate decay

Color means importance of the hyperparamters (Red: very important, Orange: mid importance and Green: low importance)

Sources:
- Coursera

4. Hyperparameters

In previous times, tuning process was easier as there were fewer hyperparameters to tune. Researchers commonly used **grid search** to find the optimal values for the algorithm.



However, as there many combinations to try, ***random search*** *and coarse to fine exploration are more recommended.*

Coarse to fine: *Start from big range and then focus areas that give you better results*

Sources:
- Coursera
- https://community.alteryx.com/t5/Data-Science/Hyperparameter-Tuning-Black-Magic/ba-p/449289

4. Hyperparameters

However, choosing the right scale for the random search is important!

For instance, the **appropriate scale for the learning rate is logarithmic.** This means, picking random values from

Another important **scale is the momentum**. Usually the range is 0.9 and 0.999

Remember that these are recommendations. You can still get good results with similar values!

Sources:
- Coursera

4. Hyperparameters

In practice, tuning the hyperparameters is a lot of trial and error. There are two approaches for this:

The Pandas Approach: You carefully select one model and a set of hyperparameters. Train it and evaluate once at a time periodically checking the performance.

The Caviar Approach: You create a lot models with different configurations and set of hyperparameters and evaluate which one works well. The disadvantage of this is that you need more computational resources.

For some applications the Pandas approach works better than the Caviar. And in general it is more used. BUT it is application dependent!

Sources:
- Coursera

4. Hyperparameters - Batch Normalization

Thanks to Sergey Ioffe and Christian Szegedy that invented the **Batch Normalization**. It makes your hyperparameter search problem much easier and **your neural network much more robust**. The choice of hyperparameters is a much bigger range of hyperparameters that work well, and will also enable you to much more easily train even very deep networks.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Essentially, the intuition is to apply the same normalization process to the activations between layers as we did for the input!

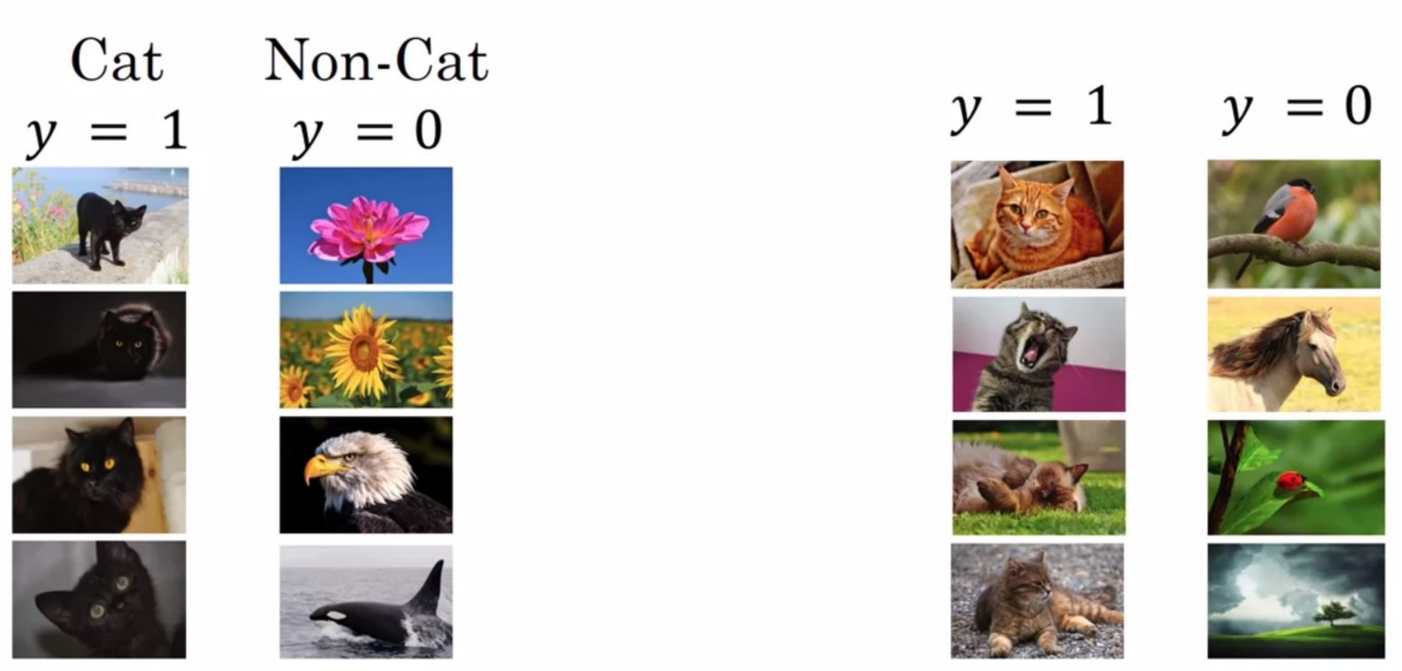
Extracting the mean and dividing by the variance

Sources:
- Coursera
- Ioffe, S. and Szegedy, C., 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift.

4. Hyperparameters - Batch Normalization

But why **Batch Normalization (BN)** helps? Basically it makes the neural network robust to **“covariate shift”**

Say you train a neural network on black cats only. If we don’t use BN, the performance of the system will be bad when testing on cats of different colors!



Sources:
- Coursera
- Ioffe, S. and Szegedy, C., 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift.

4. Hyperparameters - Batch Normalization

Batch Normalization (BN) during test time!

Batch normalization is performed on the mini batch! But **what happen during testing when we may want to test one sample at a time?**

Mean and variance are **estimated using exponentially weighted average on the means and variance computed during training time**. Then use them during testing!

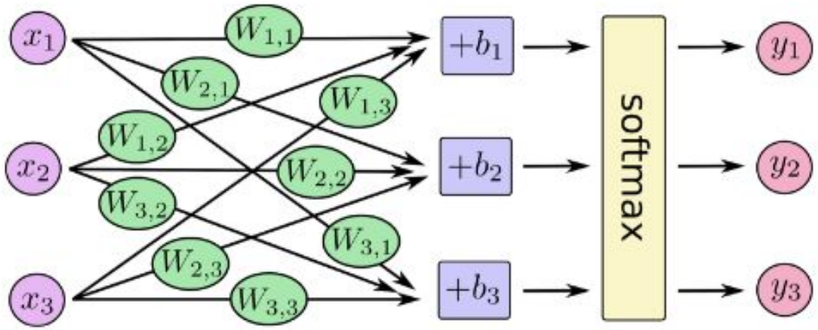
Sources:
- Coursera
- Ioffe, S. and Szegedy, C., 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift.

5. Softmax Regression

So far, the classification examples we've talked about have used **binary classification**, where we had two possible labels, 0 or 1. BUT What if we have multiple possible classes?

There's a generalization of logistic regression called **Softmax regression**, where we are trying to recognize one of C or one of multiple classes, rather than just recognize two classes.

*To do that, we used what is called the softmax layer with a **number of neurons equal to the number of classes***

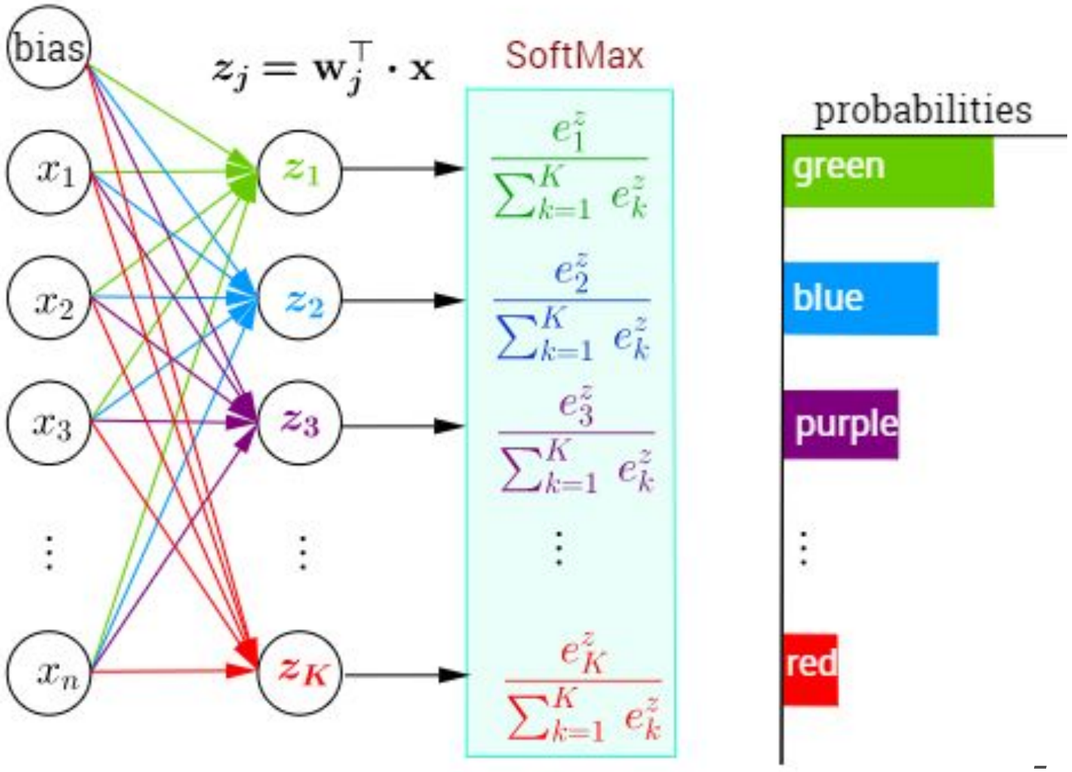


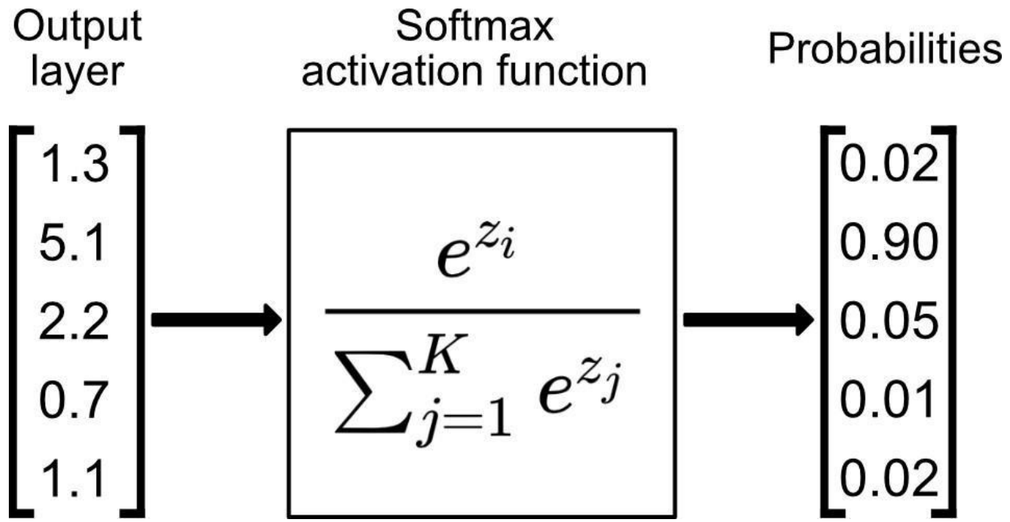
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_1 + W_{1,3}x_1 + b_1 \\ W_{2,1}x_2 + W_{2,2}x_2 + W_{2,3}x_2 + b_2 \\ W_{3,1}x_3 + W_{3,2}x_3 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

Sources:
- Coursera
- https://datascienceplus.com/mnist-for-machine-learning-beginners-with-softmax-regression/

5. Softmax Regression

The Softmax layer outputs probabilities:



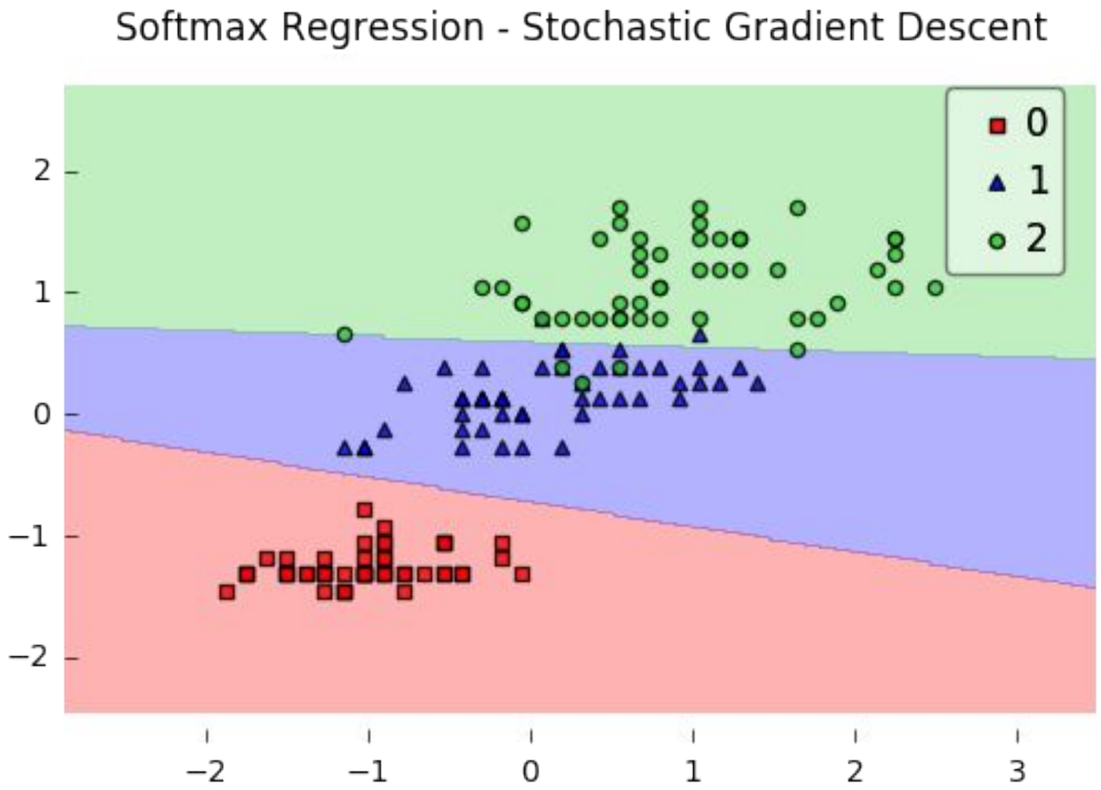


- *K is the number of classes!*
- *The output probabilities sum up to 1 as a probability distro does*

Sources:
- https://www.pinterest.co.uk/pin/849702654687897076/
- https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60

5. Softmax Regression

Example:



Sources:
- Coursera
- http://rasbt.github.io/mlxtend/user_guide/classifier/SoftmaxRegression/

5. Softmax Regression

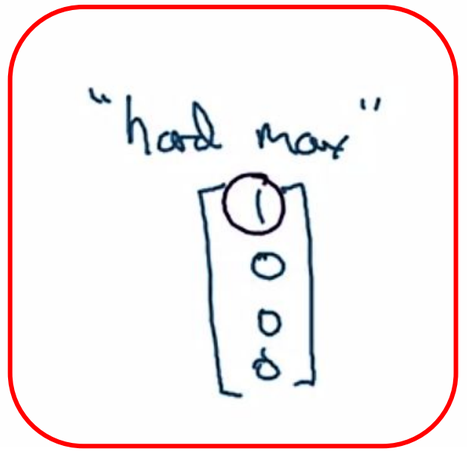
Intuitions to take away:

- Softmax regression is the generalization of logistic regression to C classes
- Why is it called softmax?

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

"soft max"

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$



Sources:
- Coursera
-

5. Softmax Regression

How do you encode the labels/classes when using softmax classification?

One-hot encoding

id	color
1	red
2	blue
3	green
4	blue

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

Sources:
- Coursera
- https://towardsdatascience.com/building-a-one-hot-encoding-layer-with-tensorflow-f907d686bf39

6. Deep Learning Frameworks

So far we've learned how to implement neural networks from scratch using Pyhton and using numpy! That's great because we have an idea of what these algorithms are doing.

BUT we'll see that **implementing more complex or bigger models that is not practical.**

For that reason, it is recommended to use deep learning frameworks!

To have an idea of what we're talking about is thinking on the difference between implementing matrix multiplication using for loops or using a numerical linear algebra library like numpy.

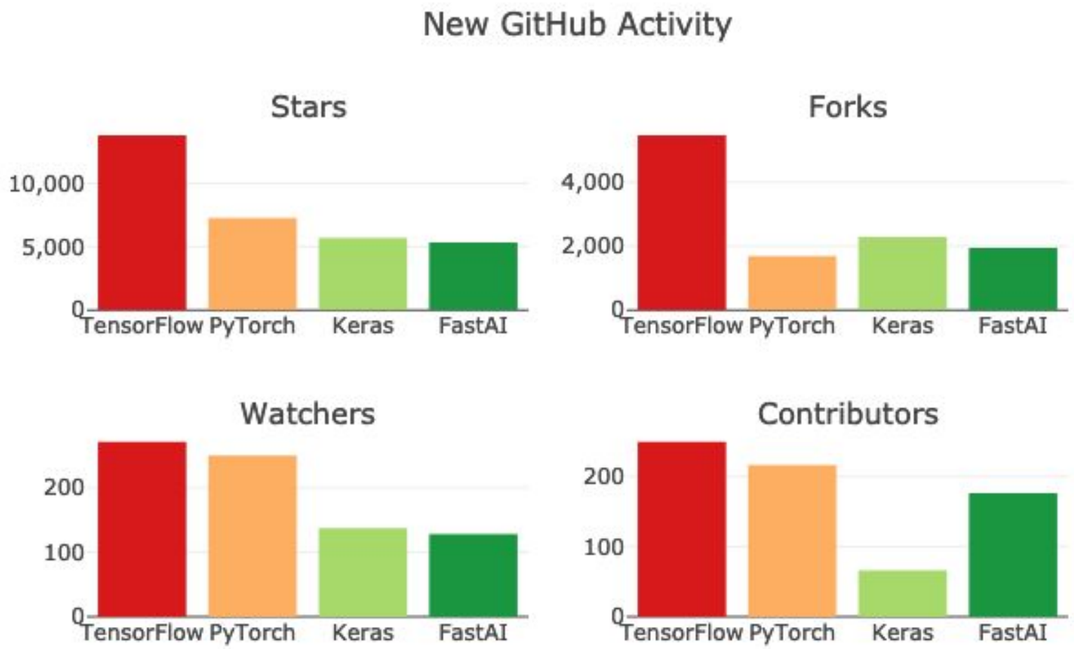
Sources:
- Coursera

6. Deep Learning Frameworks

The criteria to choose a deep learning framework is:

- Ease of programming (development and deployment)
- Running speed
- Open source

TensorFlow: https://www.tensorflow.org/
PyTorch: https://pytorch.org/
Keras: https://keras.io/
FastAI: https://www.fast.ai/



Sources:
- Coursera
- https://www.kdnuggets.com/2019/05/which-deep-learning-framework-growing-fastest.html

6. Deep Learning Frameworks

TensorFlow:

Let's see how to implement a simple loss function using TensorFlow! Tutorial:
https://github.com/pkmital/tensorflow_tutorials

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

What is a computation graph?
What is a session?
What is a Placeholder?
How can we transform from numpy array into a Tensor?

TensorFlow could easily be a complete module

Don't worry if you don't know many functions

Sources:
- Coursera

6. Deep Learning Frameworks

PyTorch:

Let’s see how to implement a simple loss function using PyTorch! Tutorial:
https://github.com/yunjey/pytorch-tutorial

A simple Linear Regression:
https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/linear_regression/main.py

Sources:
- Coursera

6. Deep Learning Frameworks

To take away:

- Deep Learning Frameworks already have many of the classes/functions we need to build a complex neural network
- A bit understanding of object orienting programming (OOP) is key to take full advantage of these frameworks
- You only need to implement the forward pass. The **Deep Learning Framework will deal with the backward pass!** Nice isn't it?

Sources:
- Coursera

Let’s move to Google Colab!

Notebooks:

- *5_TensorFlow_Tutorial.ipynb*

Sources:
- Coursera