Welcome to Deep Learning Online Bootcamp

Day 11 - Optimizing a Neural Network - Part 2

dφ

Democratizing Data Science Learning

Learning Objectives

Local and Global Minima

Gradient Descent Techniques

Vanishing and Exploding Gradients
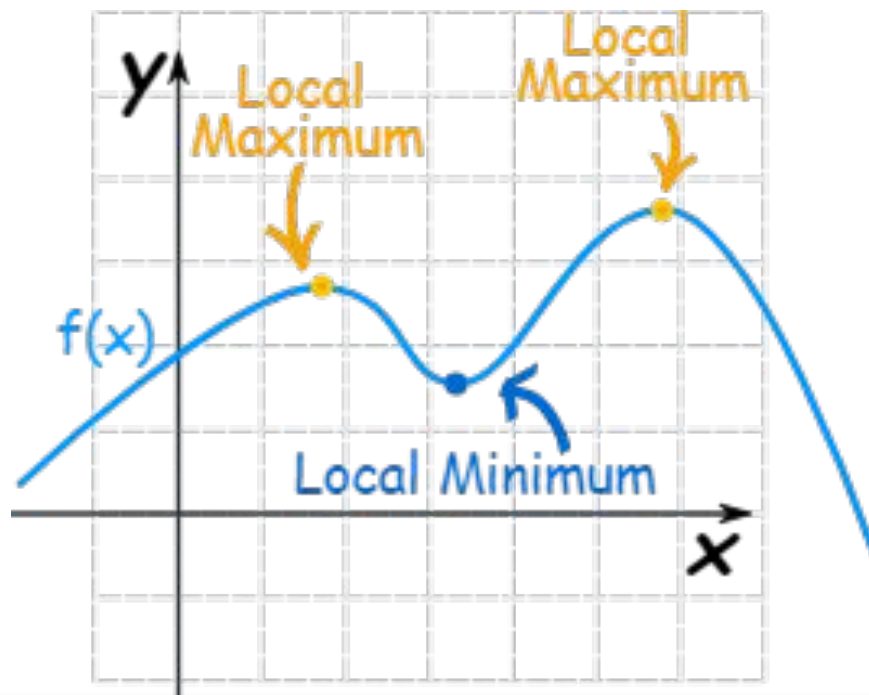
Random Initialization

Local and Global Minima

Local Minima and Maxima

Functions can have "hills and valleys": places where they reach a minimum or maximum value.

It may not be the minimum or maximum for the whole function, but locally it is.
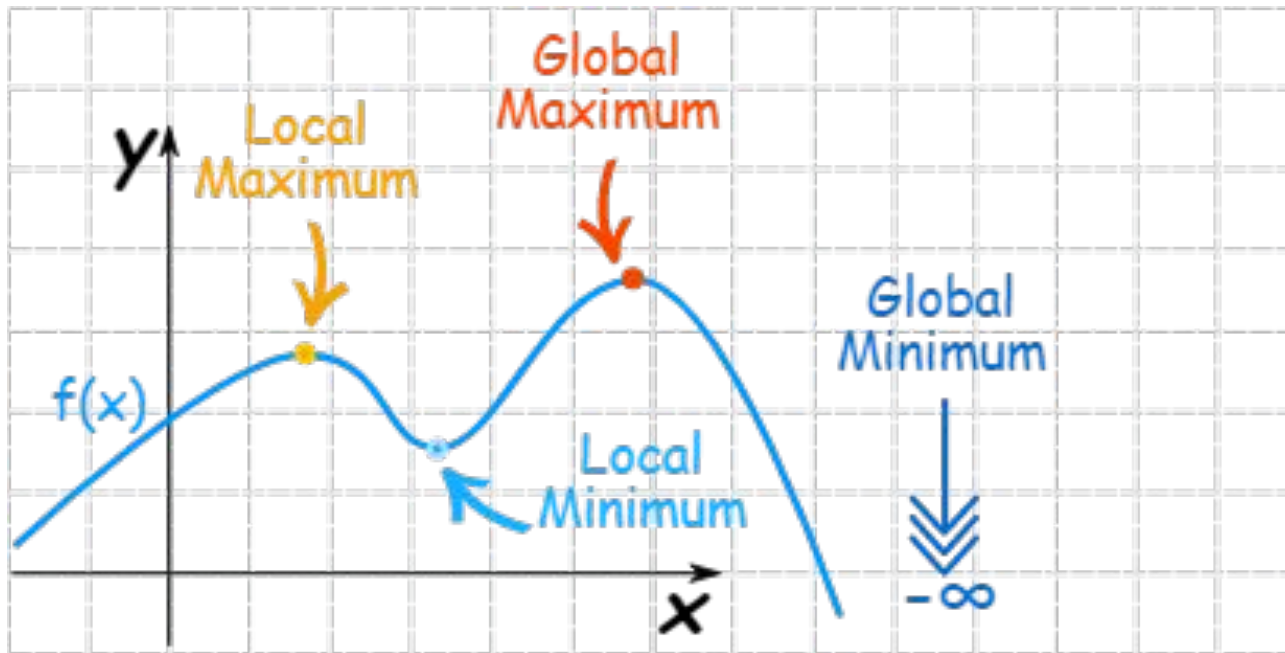
Global Minima and Maxima

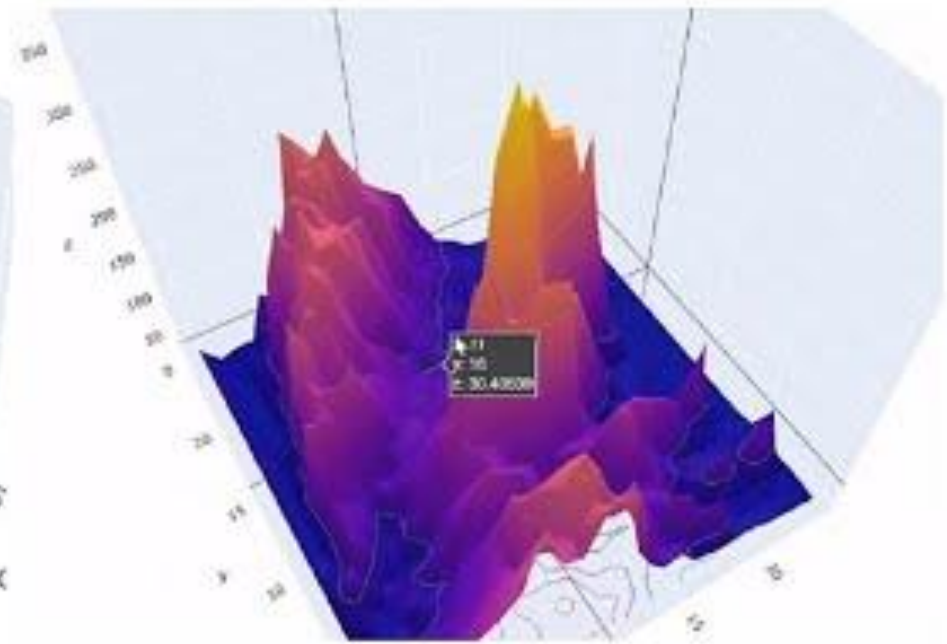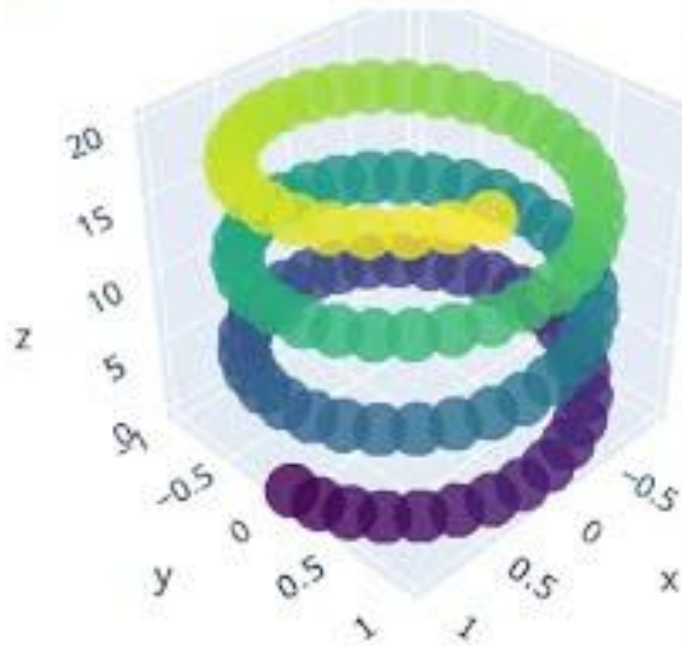The maximum or minimum over the entire function is called an "Absolute" or "Global" maximum or minimum.

There is **only one global maximum (and one global minimum)** but there **can be more than one local maximum or minimum.**

Gradient Descent with Visualization - Python

Convergence of Cost Function

As you might remember, cost function is minimized by gradient descent. Gradient descent is an efficient optimization algorithm that attempts to find a local or global minima of a function.

You might've heard 'repeat the process of Gradient Descent (GD) until convergence'. But what is exactly convergence?

Reaching a point in which GD makes very small changes in your cost function is called convergence, which doesn't necessarily mean it has reached the optimal result (but it is really quite near, if not on it).

At the minima point, the model has optimized the weights such that they minimize the cost function.
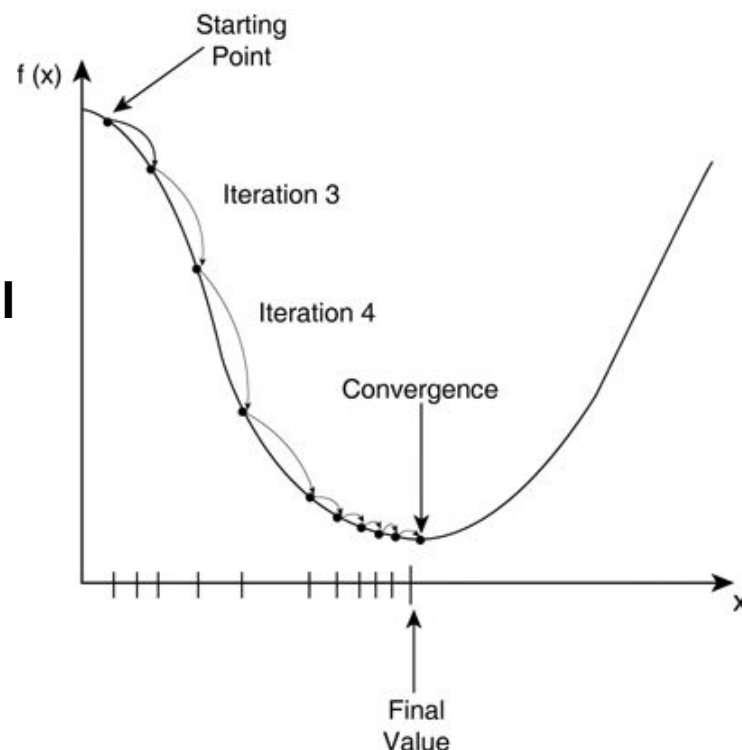
The `local minima' problem

We try to reduce to the loss function for any given problem.

The best model is obtained at global minimum i.e. where the loss is minimum. However, It is quite difficult to reach a global minima because in trainings we sometimes get stuck in local minimas.

The real challenge lies in skipping the local minima to achieve global minima and that is when you hit gold.

You can go through the below article to understand local minimas in neural network training:
https://www.allaboutcircuits.com/technical-articles/understanding-local-minima-in-neural-network-training/
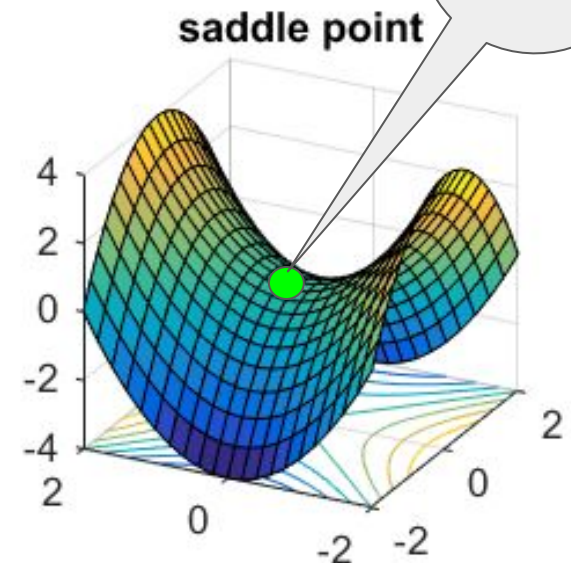
Saddle Point

It is nowadays being discussed that saddle points are in fact a more serious concern than local minima.

Now, What are saddle points? Look at the image below, Saddle points are the points lying at the center of the image on the right.

The name is inspired by the shape which is similar to a Horse saddle.
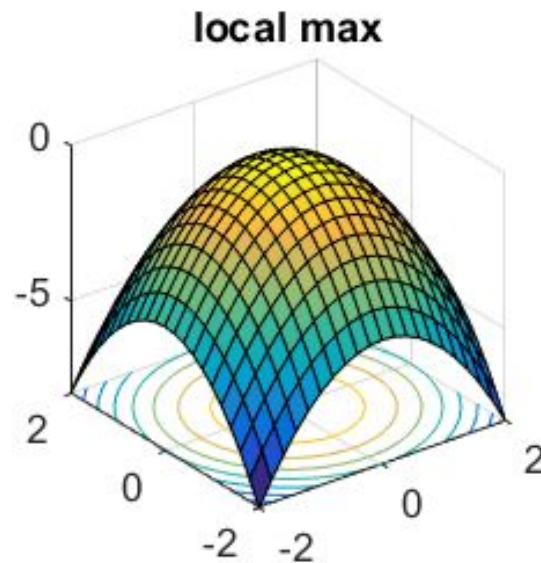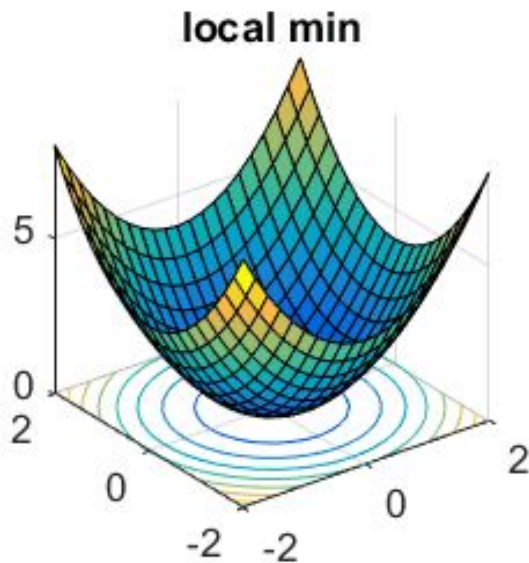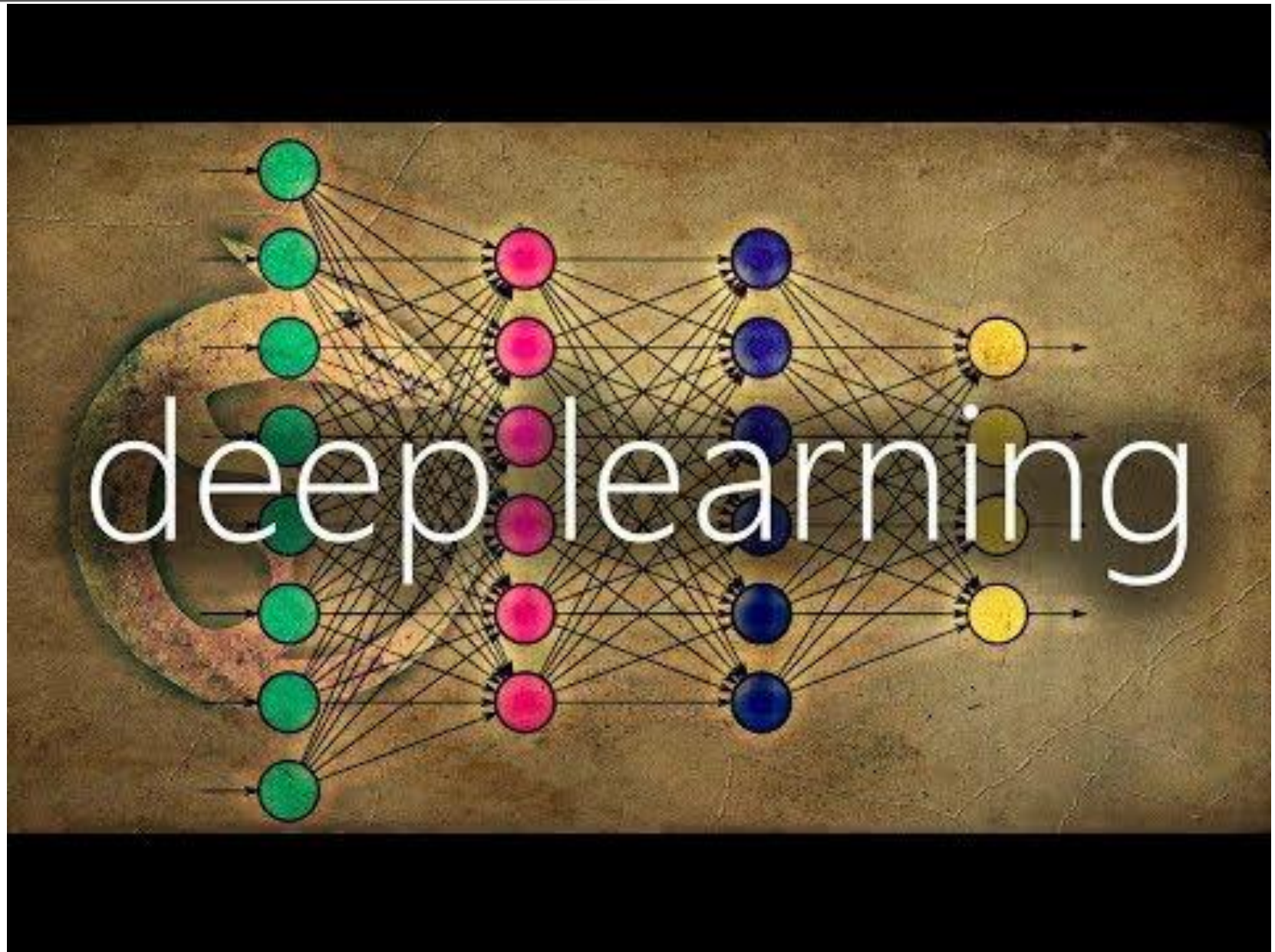
Techniques to reach Global Minima

A few concepts such as Stochastic Gradient Descent and different Weight Initialization techniques help in reaching the global minima. We'll now study these in a broader perspective.

Gradient Descent Techniques

Batch Gradient Descent

Remember the example of the mountain climber trying to climb down the valley?

There are different ways in which that man (weights) can go down the slope. Let's look into them one by one.

Batch Gradient Descent

You can only watch till 7:48 of the below video if you don't wish to delve into the implementation and only understand the concept.
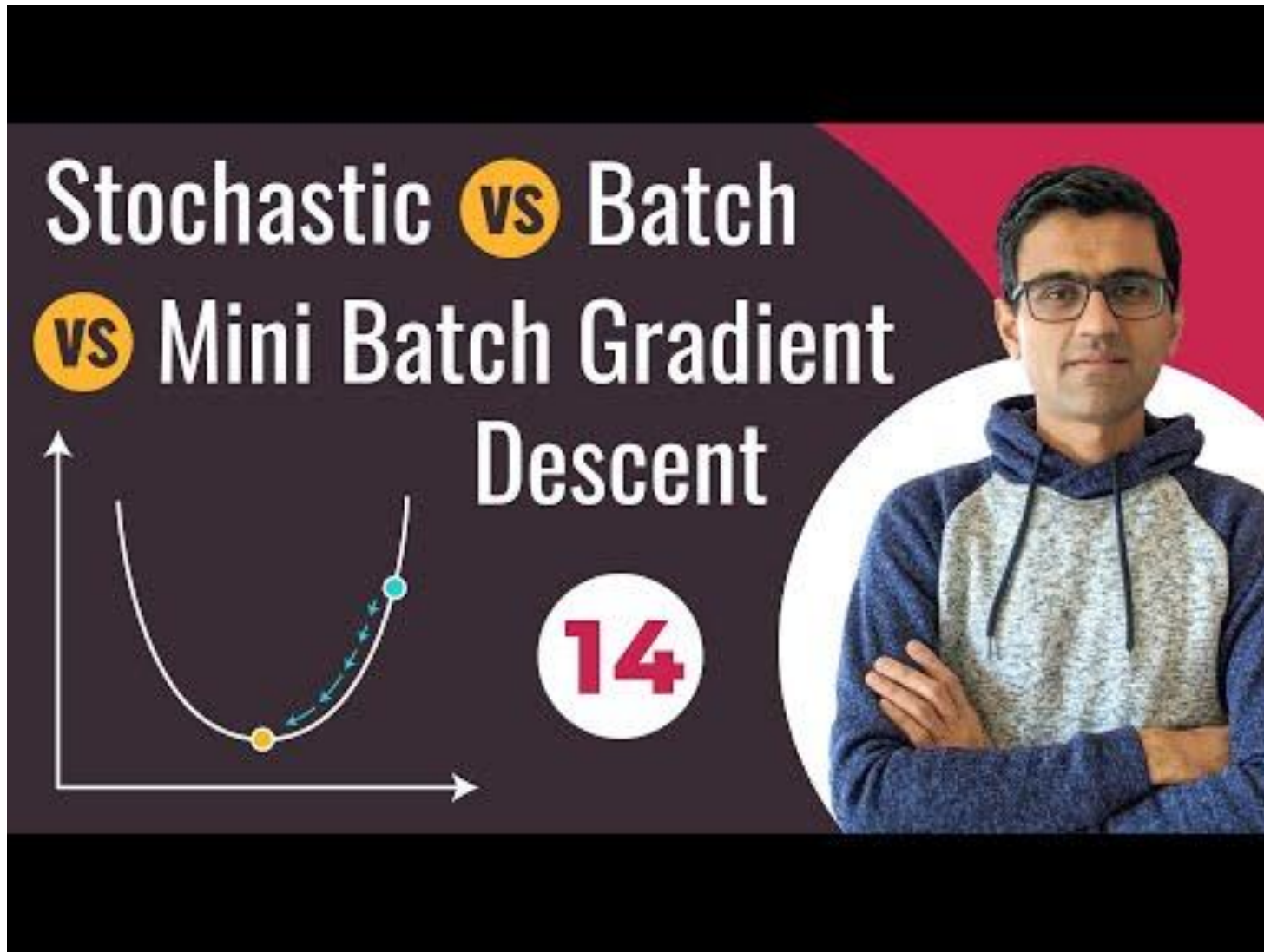
Stochastic and Batch Gradient Descent in practice

Link to the notebook used by the instructor in the previous video:

https://github.com/codebasics/py/blob/master/DeepLearningML/8_sgd_vs_gd/gd_and_sgd.ipynb

Batch Gradient Descent

In Batch Gradient Descent, **all the training data is taken into consideration to take a single step**. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

Batch Gradient Descent is great for relatively smooth error function curves. In this case, we move somewhat directly towards an optimum solution.

Batch Gradient Descent
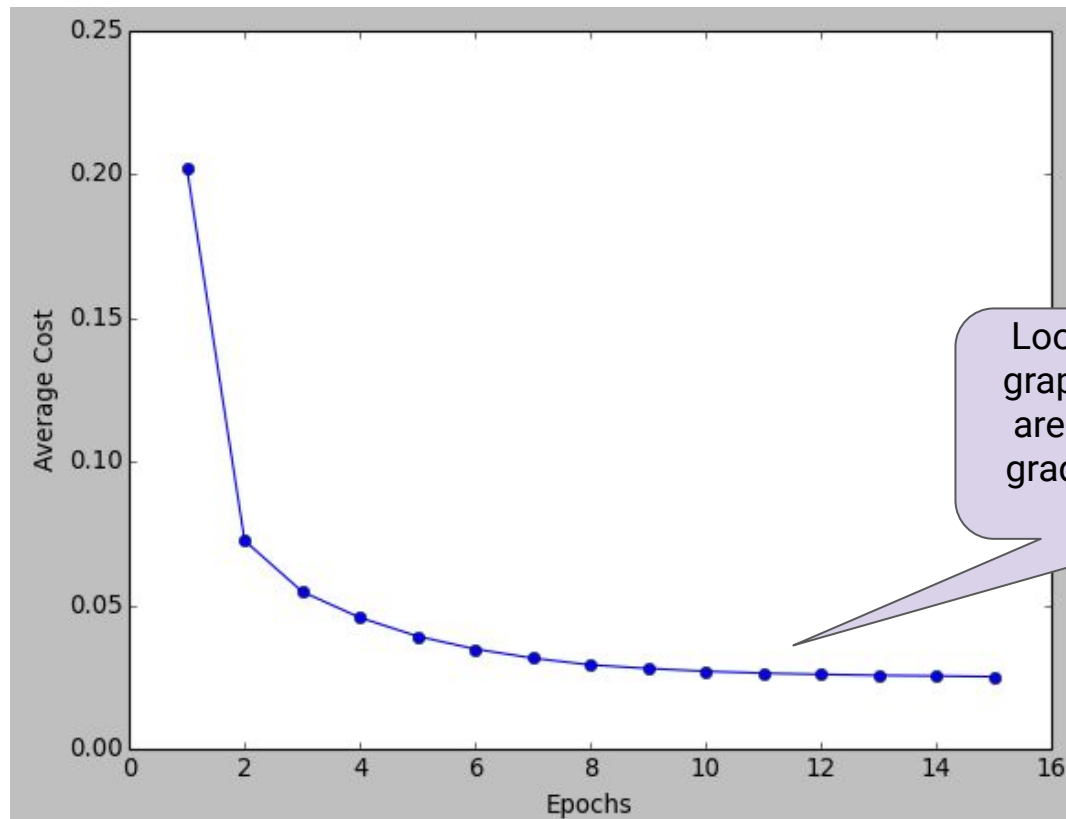
The **graph** of cost vs epochs is also **quite smooth** because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

Problem with Batch Gradient Descent

In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent.

But what if our dataset is very huge? Deep learning models crave for data. The more the data the more chances of a model to be good.

Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples! This does not seem an efficient way.

In short, since we need to calculate the gradient on the whole dataset to perform just one update, **batch gradient descent can be very slow.**

Stochastic Gradient Descent (SGD)

Batch Gradient Descent turns out to be a slower algorithm. So, for faster computation, we prefer to use stochastic gradient descent.

The first step of algorithm is to randomize (shuffle) the whole training set. Then, for updation of every parameter **we use only one training example in every iteration** to compute the gradient of cost function.

As it uses one training example in every iteration **this algo is faster for larger data set**. Still confused why? It is because the network no longer needs to process a number of examples(a batch) together at a time before making a weight update. It can quickly process 1 example, make an update and then move to the next.

In SGD, one might not achieve the best accuracy, but the computation of results are faster.

Stochastic Gradient Descent

We do the following steps in one epoch for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it's gradient
4. Use the gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for all the examples in training dataset

Stochastic Gradient Descent

Since we are considering just one example at a time the cost will fluctuate over the training examples and it will not necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.
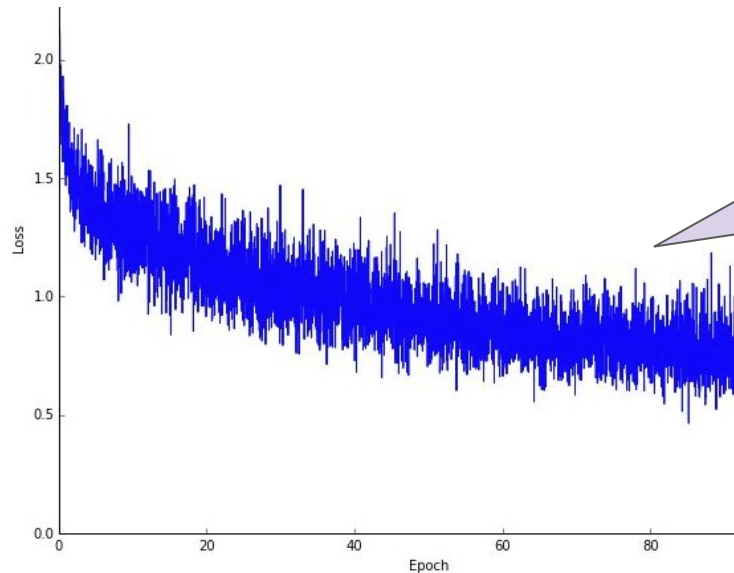
Observe how the graph is not so smooth here. It is because much more updates are happening here than in Batch GD.

Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.

SGD can be used for larger datasets. It converges faster when the **dataset is large** as it causes updates to the parameters more frequently.

Advantages of Stochastic Gradient Descent

1. It is easier to fit into memory due to a single training sample being processed by the network at a time.

2. It is computationally fast as only one sample is processed at a time.

3. For larger datasets it can converge faster as it causes updates to the parameters more frequently.

4. Due to frequent updates the steps taken towards the minima of the loss function have oscillations which can help getting out of local minimums of the loss function (in case the computed position turns out to be the local minimum).

Best of both worlds

We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent.

Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large.

Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets.

But, since in SGD we use only one example at a time, it can slow down the computations.

To tackle this problem, a mixture of Batch Gradient Descent and SGD is used. It is known as **Mini-batch Gradient Descent**.

Mini-batch Gradient Descent

Neither we use all the dataset all at once nor we use the single example at a time. We **use a batch of a fixed number of training examples which is less than the actual dataset** and call it a **mini-batch**. Doing this helps us achieve the advantages of both the former variants we saw.

So, after creating the mini-batches of fixed size, we do the following steps in one epoch:

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

Mini-batch Gradient Descent

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.

Mini-batch gradient descent
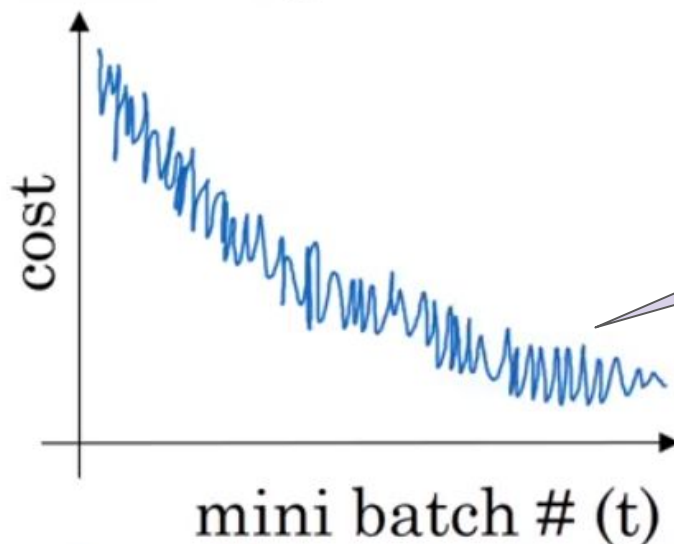
This graph is somewhere between BGD and SGD. It's neither too smooth, nor too fluctuating

cost

mini batch # (t)

So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well for faster computations.

Summary

Summarising the 3 Gradient Descent Techniques:

PARAMETERS	BATCH GD ALGORITHM	MINI BATCH ALGORITHM	STOCHASTIC GD ALGORITHM
ACCURACY	HIGH	MODERATE	LOW
TIME CONSUMING	MORE	MODERATE	LESS

Vanishing and Exploding Gradients
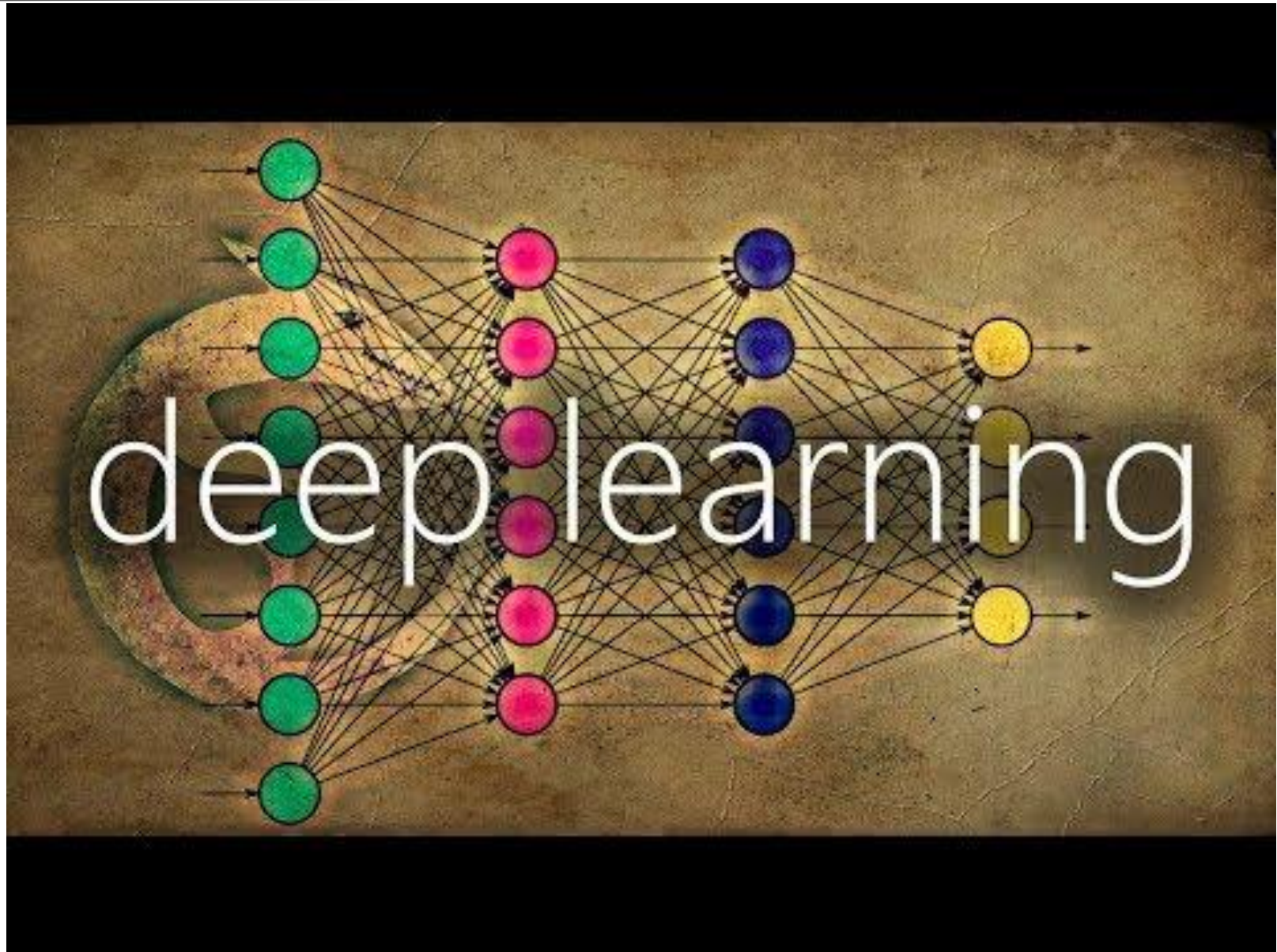
Vanishing and Exploding Gradients

Remember how we didn't use Sigmoid or Softmax in the hidden layers and used ReLU instead? Now you'll understand why.

In case of sigmoid and tanh activation functions, if your weights are large, then the gradient will be very (vanishingly) small, effectively preventing the weights from changing their value. This is because the derivative of weights will increase very slightly or possibly get smaller and smaller every iteration.

Using RELU/ leaky RELU as the activation function is a better choice, as it is relatively robust to the vanishing/exploding gradient issue (especially for networks that are not too deep).

Leaky ReLUs (a variation of ReLU Activation Function) never have 0 gradient. Thus they never die and training continues.

Random Initialization

Weights and Bias

We recommend going through the below article for a better understanding of weights and bias:

https://medium.com/fintechexplained/neural-networks-bias-and-weights-10b53e6285da
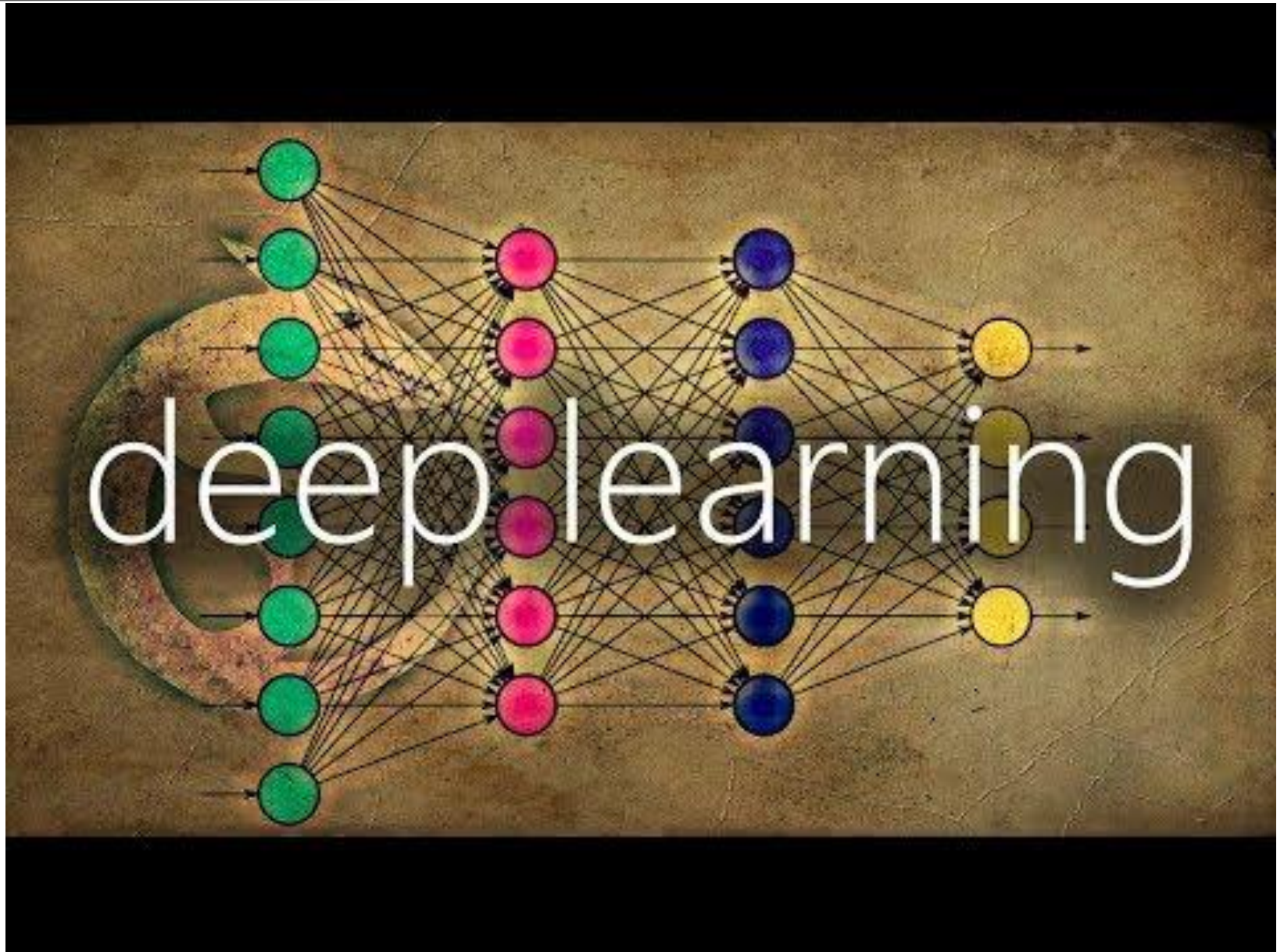
Weight Initialization

While starting the training of neural nets these parameters (typically the weights) are initialized in a number of different ways -

- sometimes, using constant values like 0's and 1's,

- sometimes with values sampled from some distribution (typically a uniform distribution or normal distribution),

- sometimes with other sophisticated schemes like Xavier Initialization.

You'll understand all of these in some time.

Importance of Weight Initialization

The weight initialization technique you choose for your neural network can determine **how quickly the network converges or whether it converges at all.** Although the initial values of these weights are just one parameter among many to tune, they are incredibly important. Their distribution affects the gradients and, therefore, the effectiveness of training.

This phenomena is largely due to the characteristics of activation functions.

The aim of weight initialization is to **prevent layer activation outputs from exploding or vanishing** during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Importance of Weight Initialization

Careful initialization of weights not only helps us to develop more reproducible neural nets but also it helps us in training them better.

Symmetry breaking is a common term you'll encounter here. It simply means that all the neurons shouldn't learn the exact same thing. Only if they are able to learn different features (are not completely symmetrical) that the model will be able to perform better.

Let us talk about 2 basic weight initialization techniques and how they can cause issues while training the model:

1. Zero initialization

Zero initialization serves no purpose. The neural net does not perform symmetry-breaking.

If we set all the weights to be zero, then all the the neurons of all the layers performs the same calculation, giving the same output and there by making the whole deep net useless.

If the weights are zero, complexity of the whole deep net would be the same as that of a single neuron and the predictions would be nothing better than random i.e it won't perform good at all.

Weights can be initialised to 1 too. That does perform better than initializing them to 0 but is still not able to break symmetry and perform well.

2. Random initialization

This serves the process of symmetry-breaking and gives much better accuracy. In this method, the weights are initialized very close to zero, but randomly. This helps in breaking symmetry and every neuron is no longer performing the same computation.

However, initializing weights randomly while working with a deep network can potentially lead to 2 issues — vanishing gradients or exploding gradients.

Key points to remember for weight initialization

1. **Weights should be small (not too small, medium small)**
   Large weights cause exploding gradients, specially while using sigmoid activation function.

2. **Weights should not be same**
   Same weights will prevent neural network from learning new features.

3. **Weights should have good variance**
   This will help each of the neuron to learn new features.

Other Weight Initializations

Here comes the other weight initialization techniques. Based on the previous key points, they try to tackle the problems with the basic initialization techniques and perform better, particularly in the case of deep neural networks.

Note: These are advanced concepts that might seem overwhelming at first. We are introducing these to you so that you are aware about the various commonly-used techniques available. There's no need to go into their depths or get worried about not understanding them properly at this moment. Just take these concepts slowly and you'll be a pro in no time! ;)

He init initialization

In this method, the weights are initialized keeping in mind the size of the previous layer which helps in attaining a global minimum of the cost function faster and more efficiently.

The weights are still random but differ in range depending on the size of the previous layer of neurons. This provides a controlled initialisation hence the faster and more efficient gradient descent.

Xavier/Glorot initialization

In short, Xavier Initialization helps signals reach deep into the network.

- If the weights in a network start too small, then the signal shrinks as it passes through each layer until it's too tiny to be useful.
- If the weights in a network start too large, then the signal grows as it passes through each layer until it's too massive to be useful.

Xavier initialization makes sure the weights are 'just right', keeping the signal in a reasonable range of values through many layers.

More initializers

Some more commonly used initializers include:

- Uniform initializer

- Kaiming initializer

Initializers in practice

- The below article shows how initializers can be implemented in Tensorflow Keras:
  https://keras.io/api/layers/initializers/

- A list of all initializers provided by Tensorflow Keras:
  https://www.tensorflow.org/api_docs/python/tf/keras/initializers

NOTE: One major thing to keep in mind is that the real benefit of using different initializers is only in the case of Deep Neural Networks where the problem of Vanishing/Exploding gradients can occur. Otherwise, you might not be able to see much performance improvement.

INTERESTING READ

A great article on Weight Initialization that studies the effects of different weight initialization schemes on a toy dataset. You can play with different initializers here and see how they work:

https://www.deeplearning.ai/ai-notes/initialization/

PS. There's no need to focus on the maths part

Slide Download Link

- You can download the slides here:
  https://docs.google.com/presentation/d/1XrIo4jdvqQZYh7zOaOAM7pz1r54KLDGTuIZkcpkJN0U/edit?usp=sharing

References

- https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a

- https://www.wandb.com/articles/the-effects-of-weight-initialization-on-neural-nets

dφ
Democratizing Data Science Learning

That's it for the day. Thank you!

Feel free to post any queries on the Discuss forum or #help channel on Slack

dφ

Democratizing Data Science Learning