HYBERPARAMTER

Different part of hyperparameters and identifying possible staring values and intuitions for a number of hyper parameters that you may have already come across, and that you will need to know in your network with deep learning.

A Hyperparameter need to set before applying a learning algorithm into a dataset.

The challenge with hyper parameters: is that there are no magic numbers that work everywhere.

The best number depend on each task and each dataset.

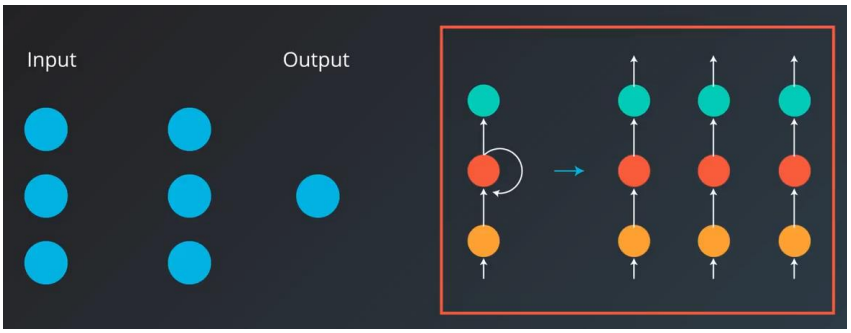
1- **Learning rate = 0.01**
2- **Minibatches size = 32**
3- **Epochs = 12**

So in addition to talking about starting values, we will try to touch on the intuition of why we would want to nudge a hyper parameter one way or another.

Generally speaking, we can break hyper parameters down in two categories.

1- **The first category is optimizer hyper parameter. (these are the variables related more to the optimization and training process than to model itself)**
   a. **Learning rate**
   b. **Minibatch size**
   c. **The number of training iterations or epochs**

2- **The second category is model hyper parameters (These are the variables that are more involved in the structure of the model.)**
   a. **These include the number of layers and hidden units**
   b. **And model specific hyper parameters for architectures**



HYBERPARAMTER

The most important hyper parameter of all, the learning rate

The single most important hyper parameter and one should always make sure that is has been tuned “

Even if apply the models that other people built to your own dataset, you will find that you will probably of different values for the learning rate to get the model to train properly.

If you took care to normalize the inputs to your model, then good staring point is usually

1- **0.01**

And these are the usual suspects of learning rates.

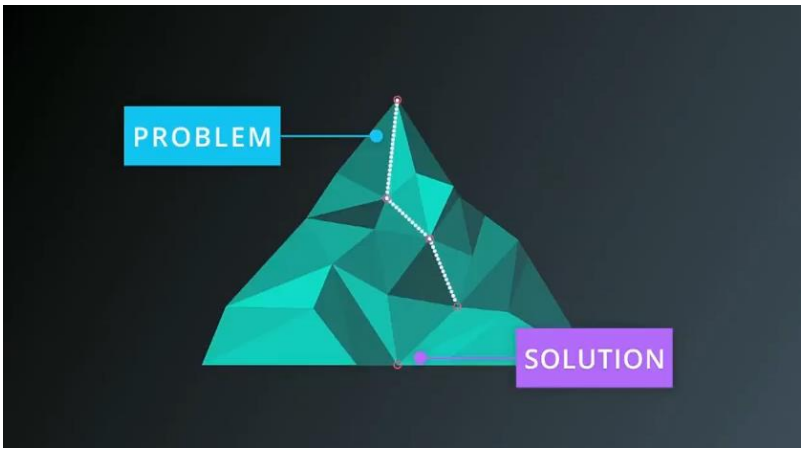
1- **0.1**
2- **0.01**
3- **0.001**
4- **0.001**
5- **0.00001**
6- **0.000001**

If you try one and your model doesn`t train. You can try others from this list.

Which of the others should you try ?

That depends on the behavior of the training error. TO better understand with we need to look at the intuition of the learning rate.

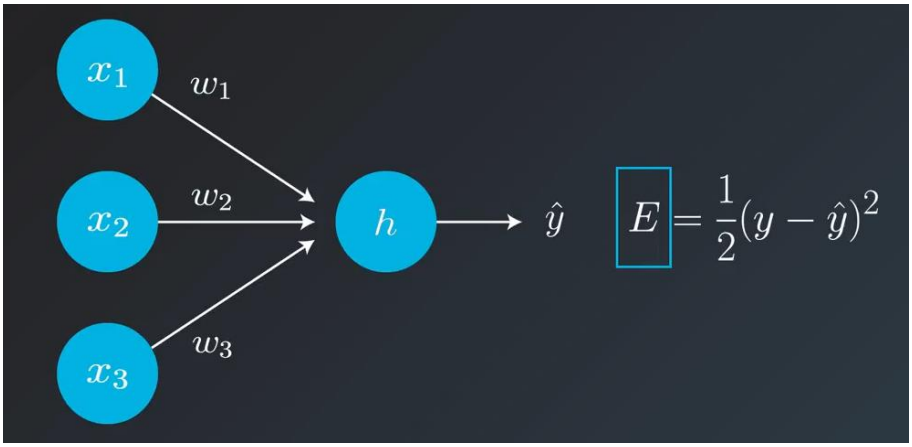
Earlier in the course we saw that when we use gradient descent to train a neural network model, the training task boils down to decreasing as much as we can.



During the learning step, we do a forward pass through the model, calculate the loss then find the gradient.

Let`s assume a simple case, in which our model has only one weight.

The gradient will tell us which way to nudge the current weight so that our predictions become more accurate.



To visualize the dynamics of the learning rate, let us plot the value of the weight versus the error value a prediction for a random training data point.

1- **In perfect case, and if we have zoom in to the correct part of the value of the curve the relationships of the weights and error values look like this idealized u-shape.**

Choosing a random weight and calculating the error value would give us a point like this one on the curve.

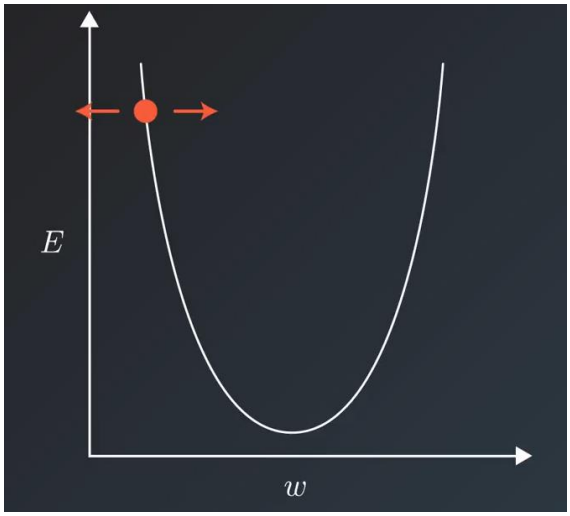
Note we don`t know what the curve looks like when we start.

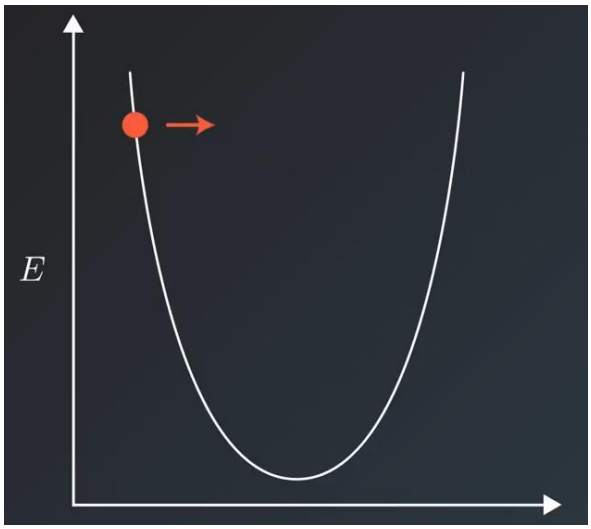
The only way to calculate the error at every weight point.

We are only drawing it here to clarify these dynamics.

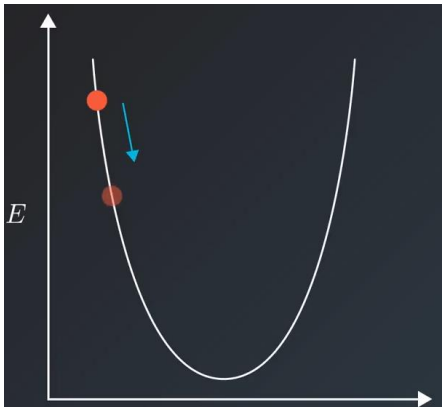
Calculating the gradient would tell us which direction to go to decrease the error

1- **If we do calculation correctly, the gradient will point out direction to go meaning whether we should increase or decrease the current value of weight.**

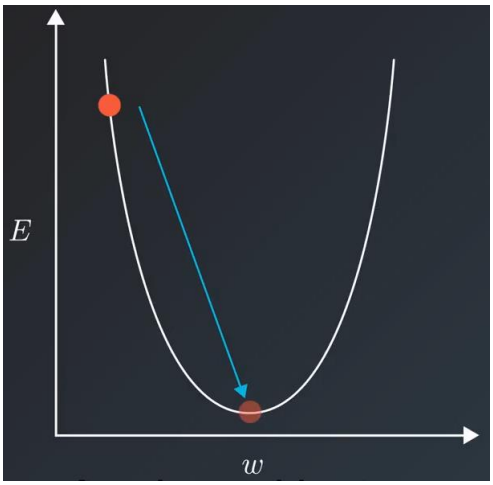




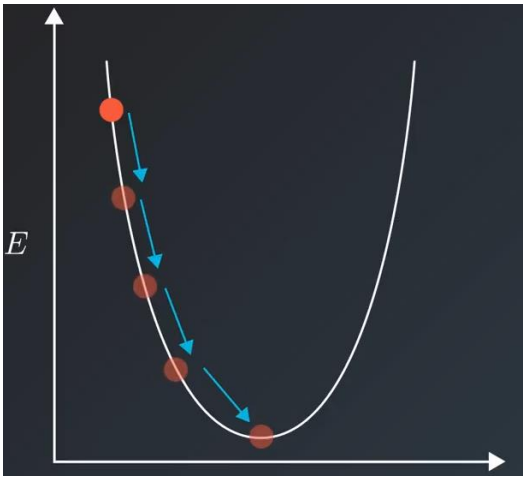
The learning rate is the multiplier we use to push the weight towards the right direction.



Now, if we had made a miraculously correct choice for our learning rate, then we would land on the best weight after only one training step.



If the learning rate we choose was smaller than the ideal rate then that`s ok.



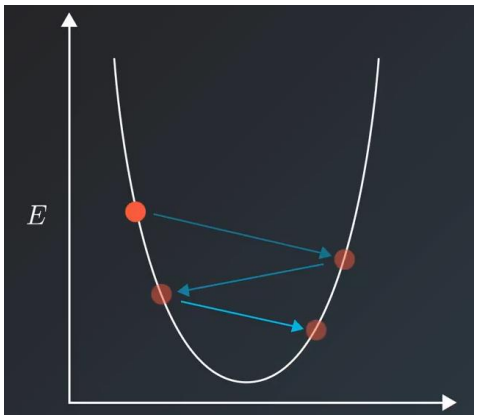
Our model can continue to learn until it finds a good value for the weight.

So each training step it will take a step closer until it land on that best weight at the end.

If, however the learning rates had been little then our training error would be decreasing but very slowly.

And we might go through hundreds or thousands of training steps without reaching the best value for our model. And it`s clear that in cases like this we need to do is to increase the learning rate.

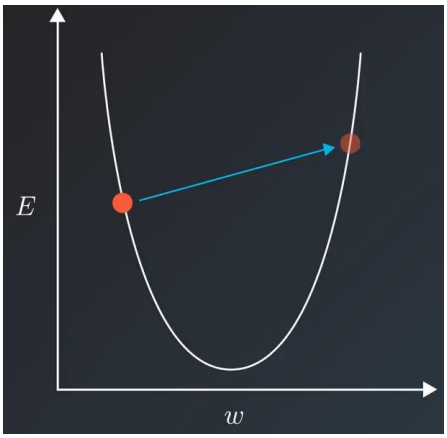
One other case is that if we choose a learning rate that is a larger that the ideal learning rate our updated value would overshoot the ideal weight value. And the next update it is overshoot it the other way.

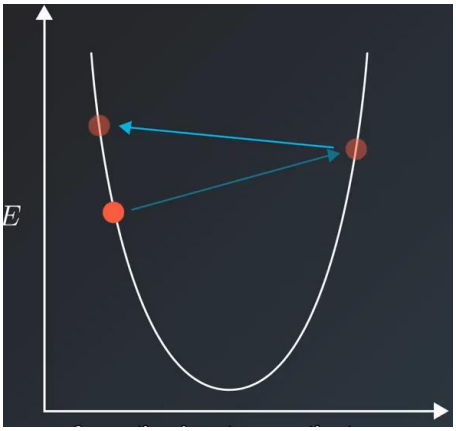


But it will, keep getting closer, and it would probably converge to a reasonable value.

Where this becomes problematic through is when we choose a learning rate that is much larger that ideal rate more than twice as much.

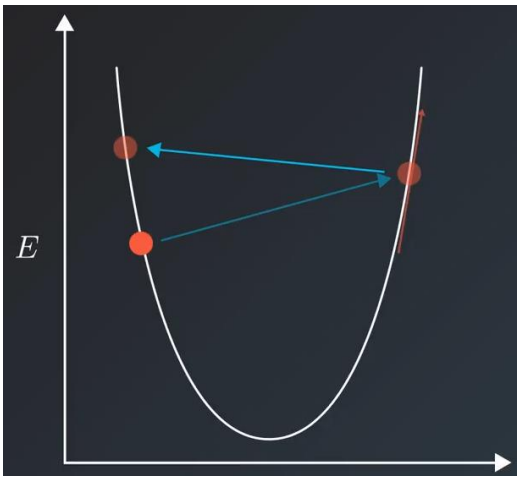
So in this case we will see the weights the talking a large step that not only overshoots the ideal weight but actually gets farther and farther from the best error that we can get at every step.





A contribute to this divergence s the gradient.

The gradient does not only contribute a direction but also a value that corresponds to the slope of the line tangent to the curve at point.



The higher the point is on the curve, the more steep the slope is and the larger the value of the gradient is.

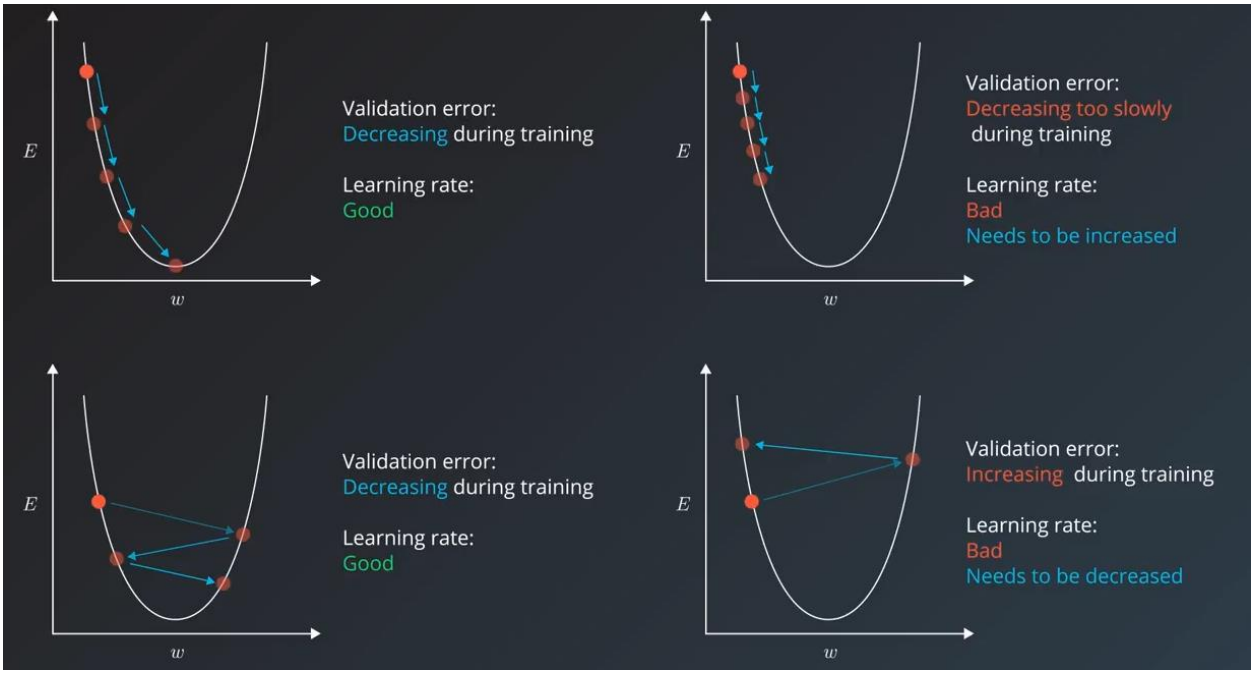
So this makes the problem of the large learning rate even worse.

So if our training error is actually increasing, we might want to try to decrease the learning rate and see what happens.

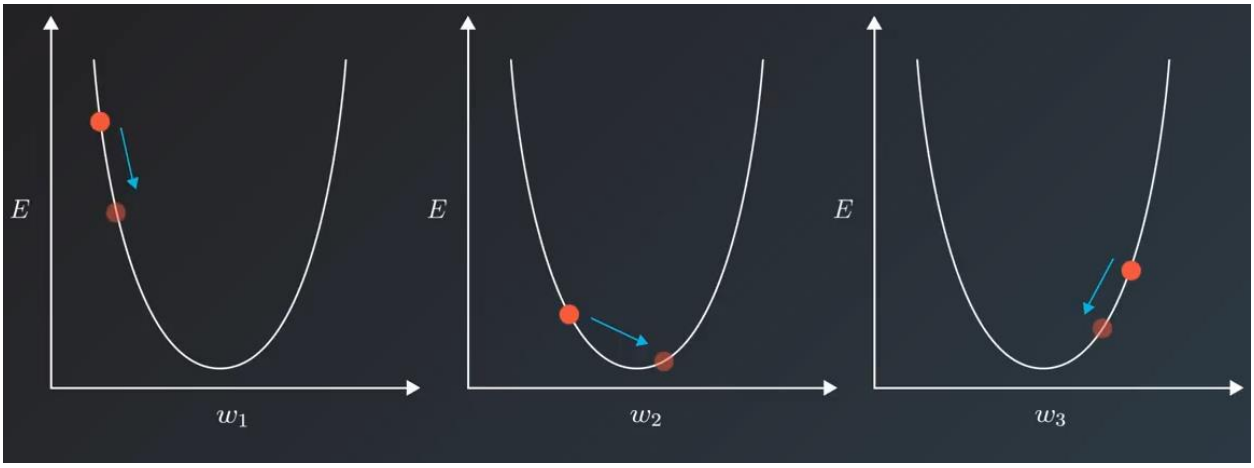
HYBERPARAMTER

This is general case come across when tuning your learning rate.

But note that this here is a simple example with only one parameter and ideal convex error curve.

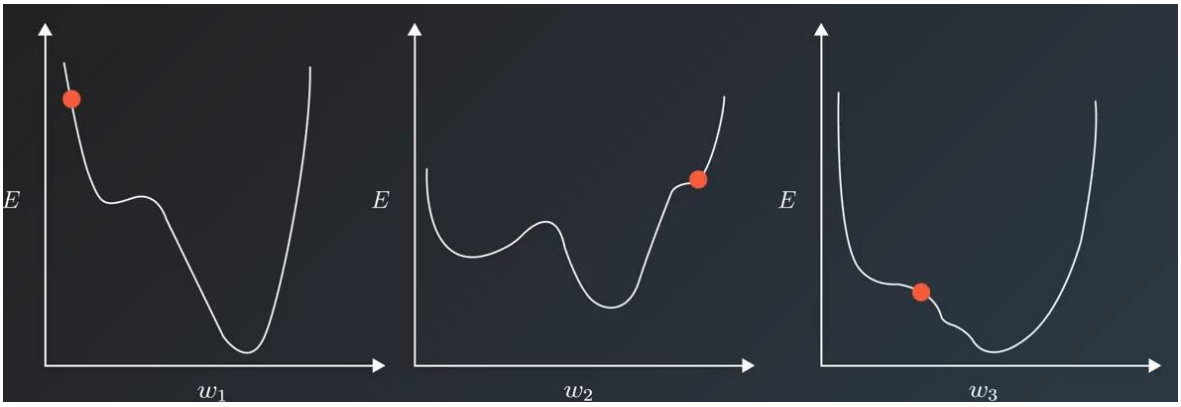


Things are more complicated in the real world, I'm sure I have seen You models are likely to have hundreds or thousands of parameter each with it`s own error curve that changes as the values of the weight change and the learning rate has shepherd all of them to the best values that produce the least error.



To make matters even more difficult for us, we don`t have actually have any guarantees that the error curves would be clean n-shapes.

They might in fact be more complex shapes with local minimum that the learning algorithm can mistake for the best values and convergence on.



This figure is obviously an oversimplification. Because the curve of three parameters versus the error value is actually a plane in four dimensionally space and hard to visualize.

So think of this complexity that these incredible Algor them can help us overcome.

It easy to be intimidated by a thousands or a million weights. Each with their own error curve that depend on the other values plus each weight having a random starting value and our diligent learning rate pushing them left and right in order to fit our training data and find the best model.

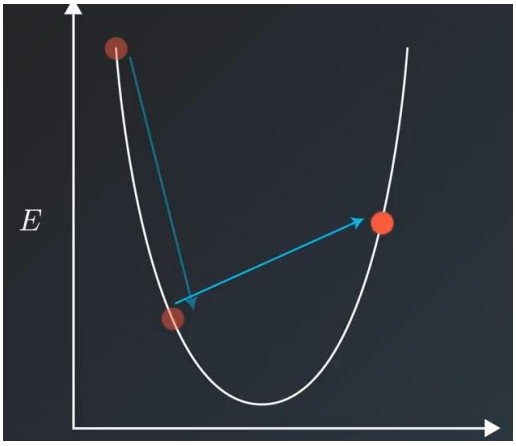
But this is the monster you have slain over and over again in the exercise and projects of this Nanodegree.

I just want to take a second and reflect on the incredible power we wield armed with these brilliant algorithms.

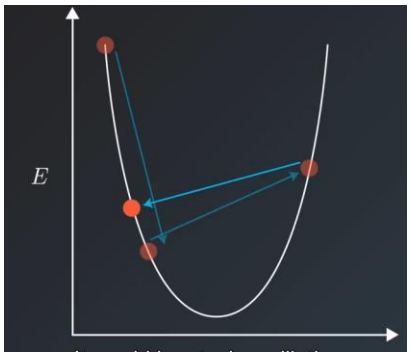
Now that we look at the intuition of the learning rates, and the indications that the training error give us that can help us tune the learning rate,

Let`s look at one specific case we can often face when tuning the learning rate.

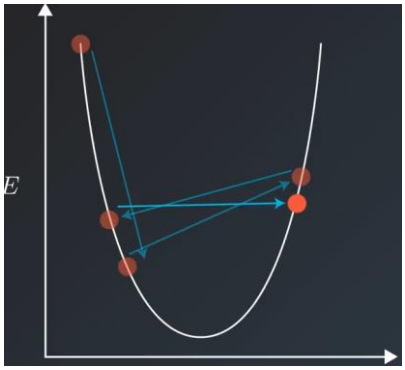
Think of the case where we choose a reasonable learning rate. It manages to decrease the error, but up to point,



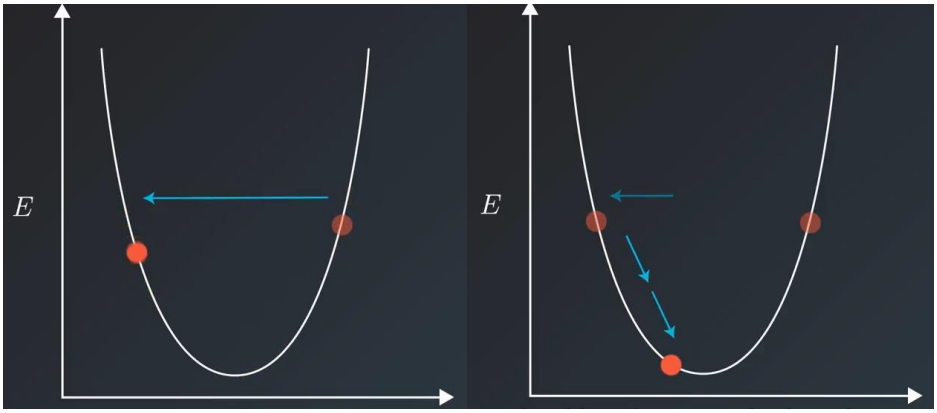
after which it`s unable to descend even through it didn`t reach the bottom yet.



It would be stuck oscillating between values that still have a better error value that when we started training, but are not he best values possible for the model.

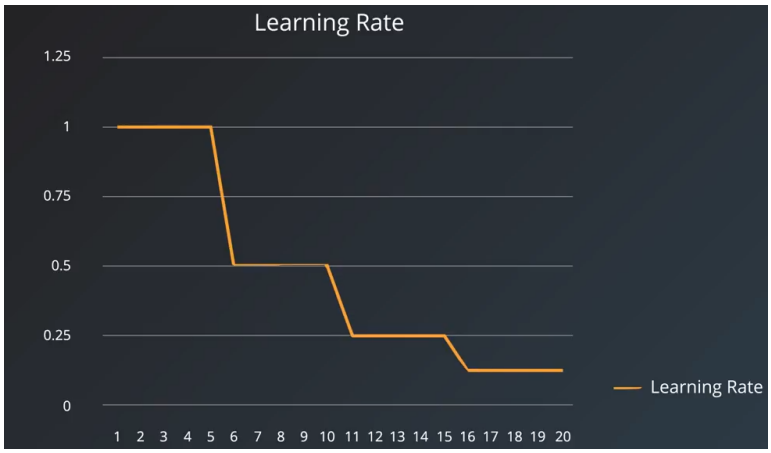


This scenario is where it`s useful to have our training algorithm decrease the learning rate throughput the training process.



This is a technique called learning rate decay. Intuitive ways to do this can be by decreasing the learning rate linearly.

So say, decrease it by half every 5 epochs, like this example.



You can also decrease the learning rate exponentially. So, for example we would multiplied the learning rate by 0.1 every 8 epochs for example.

In addition to simply decreasing the learning rate there more clever learning algorithms that have an adaptive rate.

These algorithms are just the learning rate based on what the learning algorithms knows about the problem and the data that it`s seen so far.

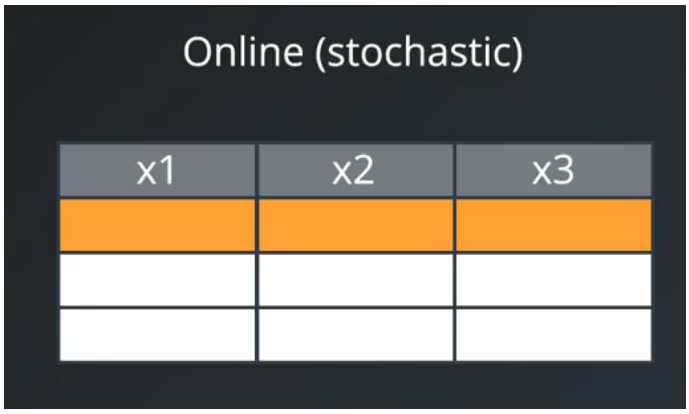
Adaptive learning: This means not only decreasing the learning rate when needed, but also increasing it when it appears to be too low. Bellow this video for using an adaptive learning algorithm in TensorFlow.

Minibatch Size

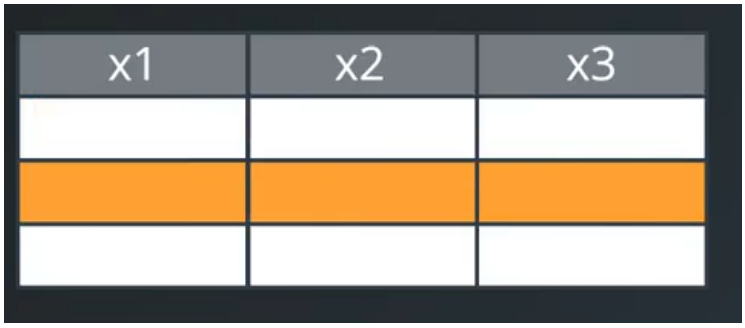
Minibatch size is another hyper parameter that no doubt you have run into a number of times already.

It has an effect on the resource requirement of the training process but also impact training speed and number iteration in way a that might not be trivial as you may think.

It is important to review a little bit of terminology here first. Historically there had been debate on whether it`s better to do online also called stochastic training where a fit a single example of the dataset to the model during a training step.



And using only one example do a forward pass, calculate the error and then propagate and set adjusted values for all your parameters.



And then do this again for each example in the dataset.

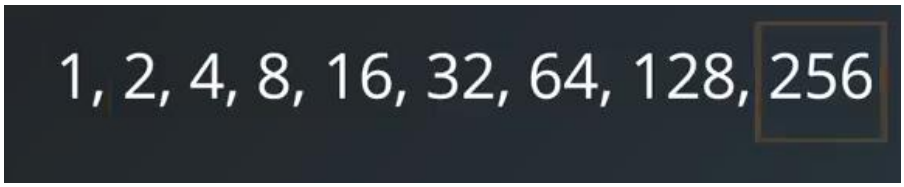
Or if it was better to feed the entire dataset to the training step and calculate that gradient using the error generated by looking at all the examples in the dataset. This is called batch training

The abstraction commonly used today is to set a minibatch size.

So online training is when the minibatch size is one, batch training is when the minibatch size is the same number of examples in the training set.

And we can set the minibatch size to any value between these two values.

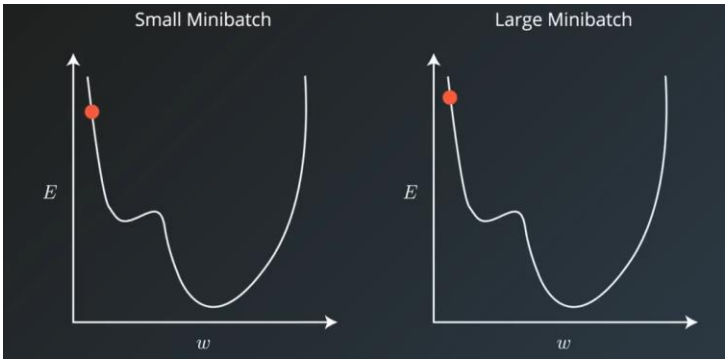
The recommended starting values for your experimentation are between one and a few hundred with 32 often being a good candidate.



A large minibatch size allow computational boosts that utilize matrix multiplication in the training calculations but that comes at the expense of needing more memory for the training process and generally more computational resources.

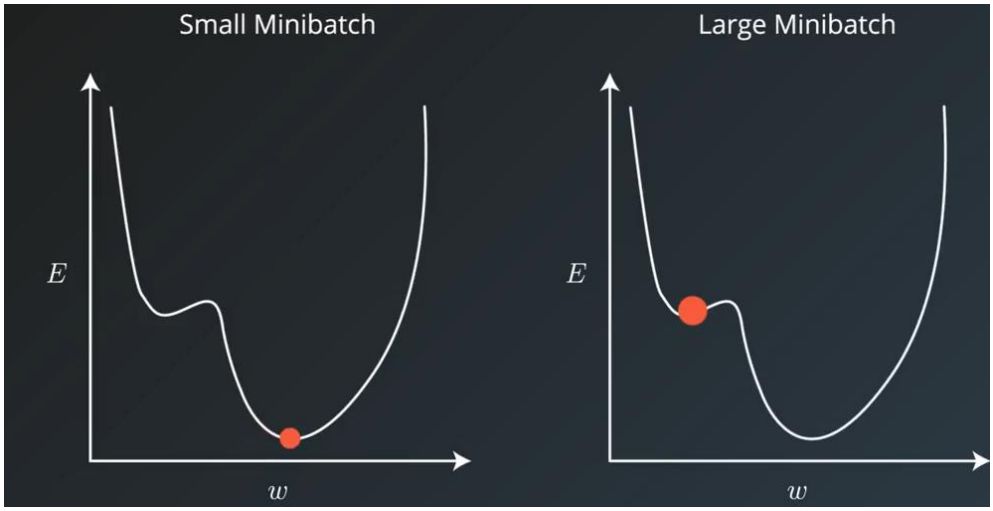
- **Some out of memory errors and TensorFlow can be eliminated by decreasing the miniating size.**

It`s important however to note that this computational boost comes at a cost.



HYBERPARAMTER

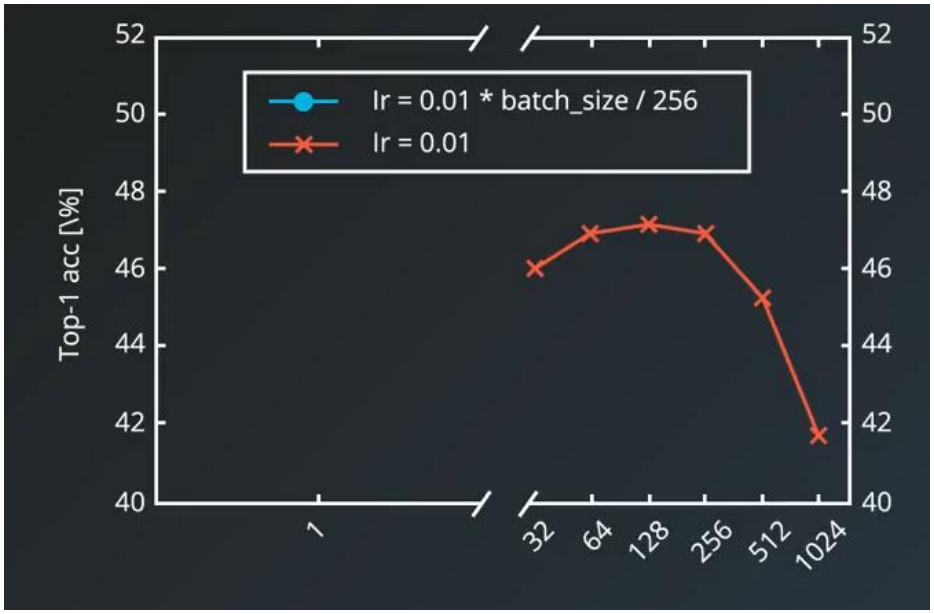
In practice, small minibatch size have more noise in their error calculations and this noise is often helpful in preventing the training process from stopping at local minima in the error curve rather than the global minima that creates the best model.



So, while the computational boost incentivizes us to increase the minibatch size this practical algorithmic benefit incentivizes us to actually make it smaller.

So, in addition to 32, you might also want to try experiment with 64 and 128 and 256. And depending on your data and task, you might have to experiment with other values as well.

The is an experimental result for the effective batch size on the convolutional neural nets.

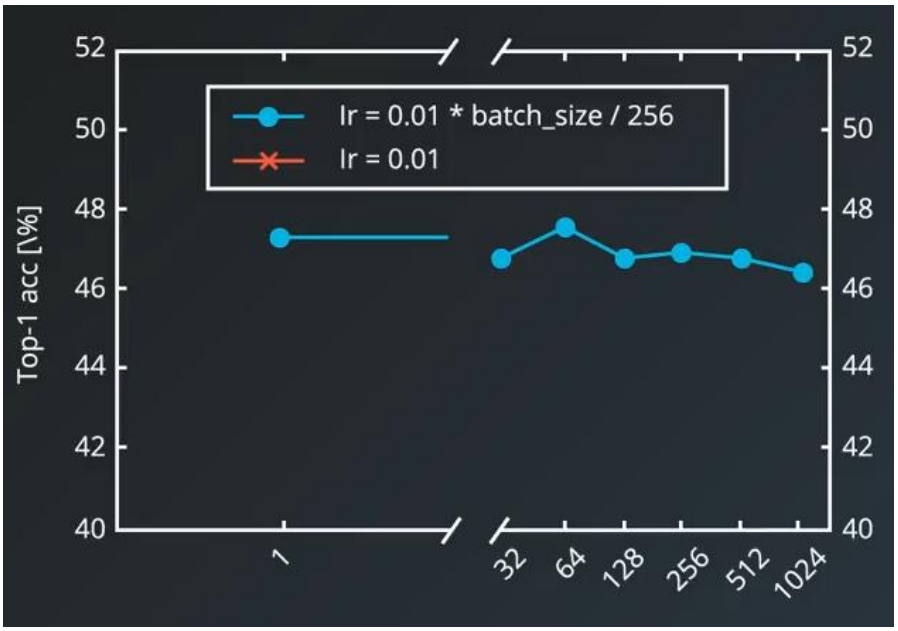


Source: Mishkin, Dmytro, Nikolay Sergievskiy, and Jiri Matas. "Systematic evaluation of CNN advances on the ImageNet." arXiv preprint arXiv:1606.02228 (2016).

HYBERPARAMTER

It`s from the paper titled systematic evolution of CNN advances on the image net. It shows that using the same learning rate. The accuracy of the model decreases the larger the minibatch size becomes.

Now this is not only the effect of the minibike size, but due to the fact that we need to change the learning rate if we change the batch size. We can see that the accuracy does decrease. But only slightly, the more increase the batch size.



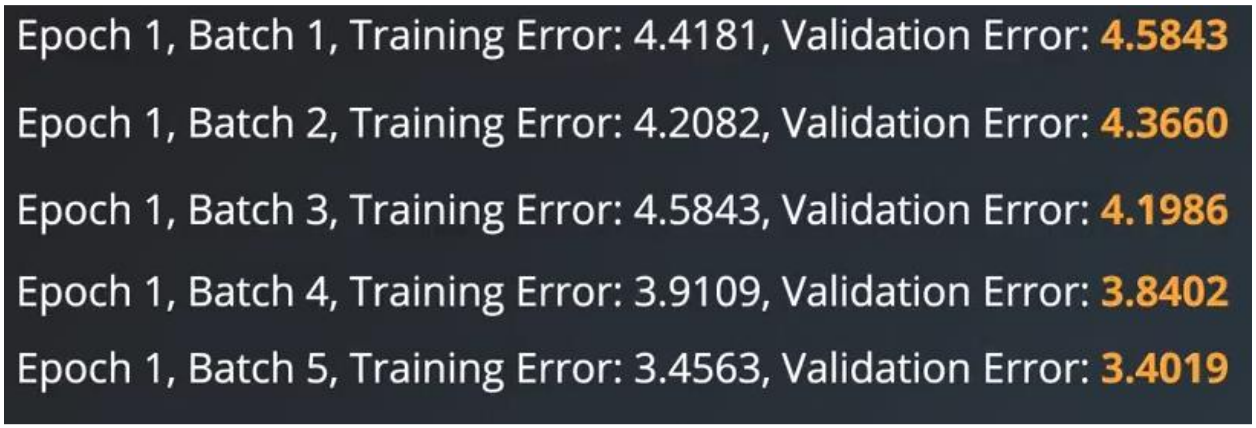
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048

So to sum up for the minibatch size, too small could be too slow, too large could be computationally taxing and could result in worse and 32 to 256 are potentially good staring values for you to experiment with.

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048

Number of iterations

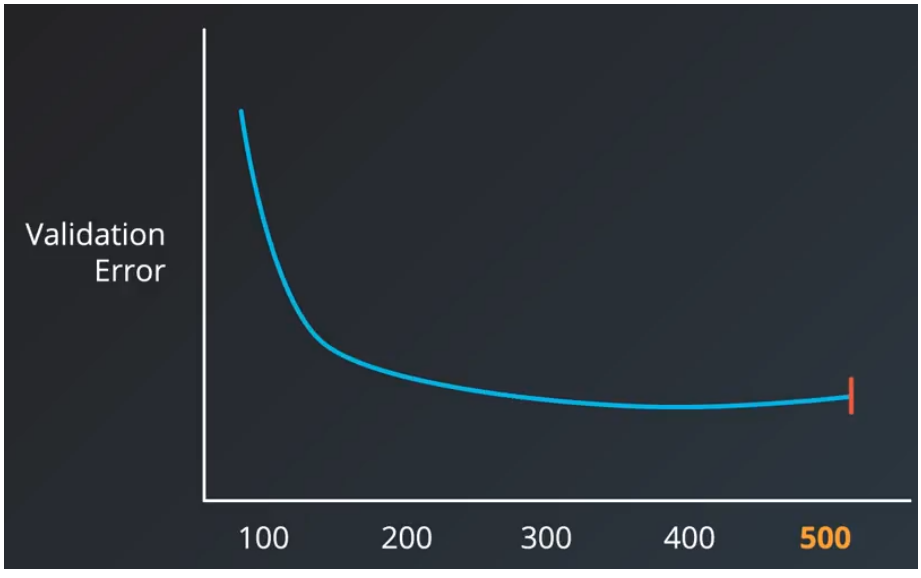
To choose the right number of iterations or number of epochs for our training step, the metric we should have our eyes on is the validation error.



The intuitive manual way is to have the model train for as many epochs or iterations that it takes, as long as the validation error keeps decreasing.

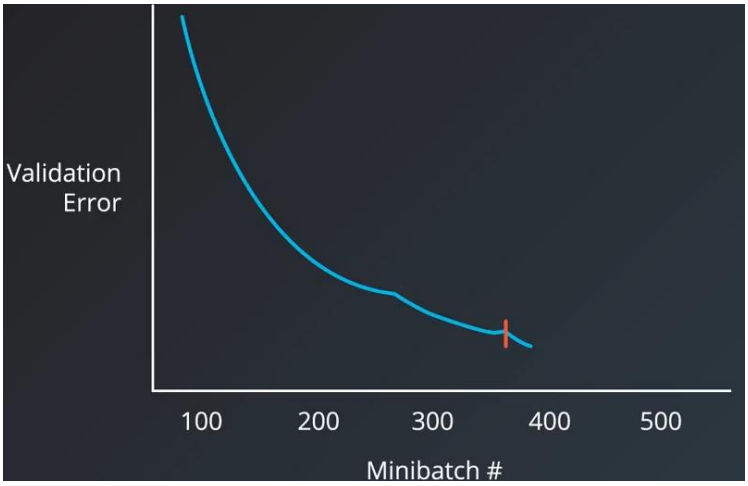
Luckily, however we can use a technique called early stopping to determine when to stop training a model.

Early stopping roughly works by monitoring the validation error, and stopping the training when it steps decreasing.



we must be a little flexible through in defining the stopping trigger .

Validation error will often move back and forth even if it`s on a downward trend.



So instead of stopping the training the first time we see the validation error increase, we can instead stop training

- **if the validation error has not improved in the last 10 or 20 steps.**

TensorFlow has support for early stopping. Check the text below the video for how to implement it in TensorFlow.

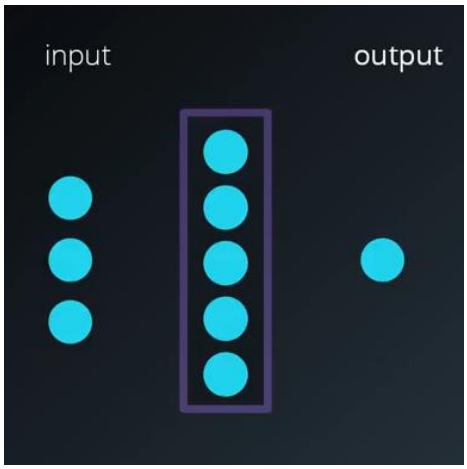
Number of Hidden Units - Layers

Let`s took about the hyper parameters that relate to the model itself rather than the training or optimization process.

The number of hidden units, in particular, is the parameter I felt was the most mysterious when I stared learning about machine learning.

The main requirement here is to set the number of hidden units that is quote unquote large enough.

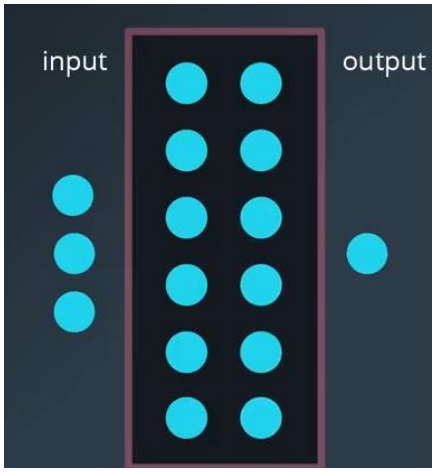
For a neural network to learn to approximation a function or prediction task, it needs to have enough quote unquote capacity to learn the function.



HYBERPARAMTER

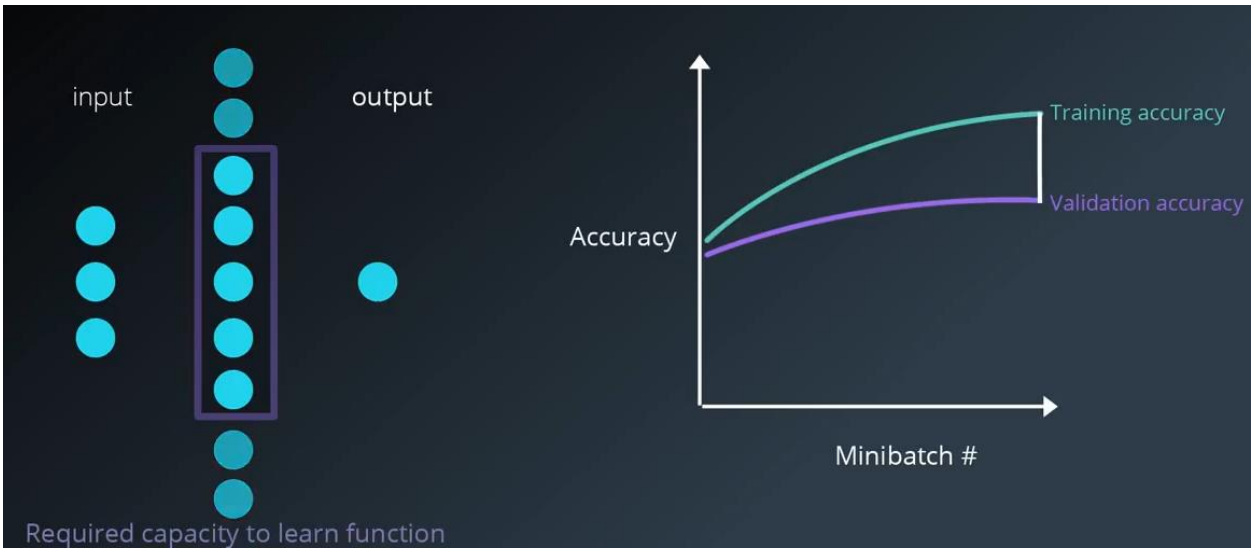
The more complex the function, the more learning capacity the model will need.

The number and architecture of the hidden units is the main measure for a model`s learning capacity.



If we provide the model with too much capacity, however, it might tend to overfit and just try to memorize the training set.

- **If you find your model overfitting your data meaning that the training accuracy is much better that the validation accuracy,**



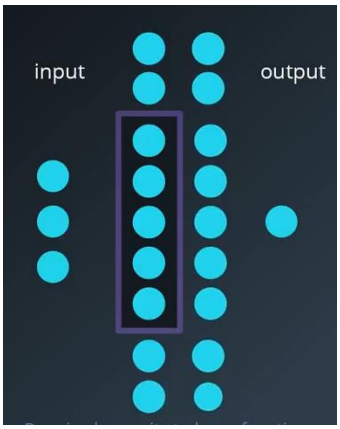
- **You might want to try to decrease the number of hidden units.**

You could also utilize regularization techniques like

1- **L2 regularization**
2- **Dropout**

So as far as number of units is concerned, the more the better.

A liitle larger than the ideal number is not a problem, but a much larger value can often lead to the overfitting.

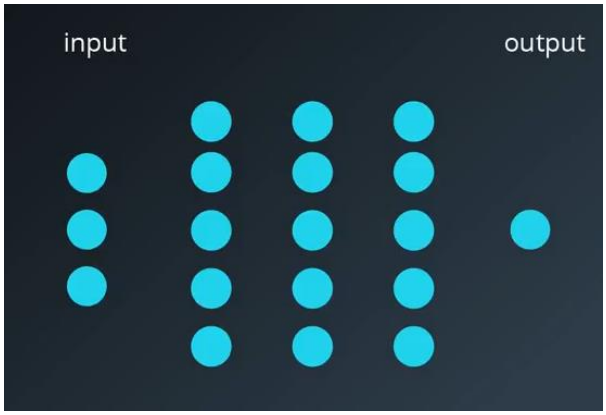


So if your model is not training, add more hidden units and track validation error. Keep adding hiddeing units unitl the validaion start getting worse.

Another heurstic involoving the first hidden layeris that setting it to a number larger than the number of the inputs has been observed to be beneficial in a number of task.

What aboout the number of layers ?

Andrej karpathy tells us that in practice it` often case that a three layer nerual net will outperform a two layer net,

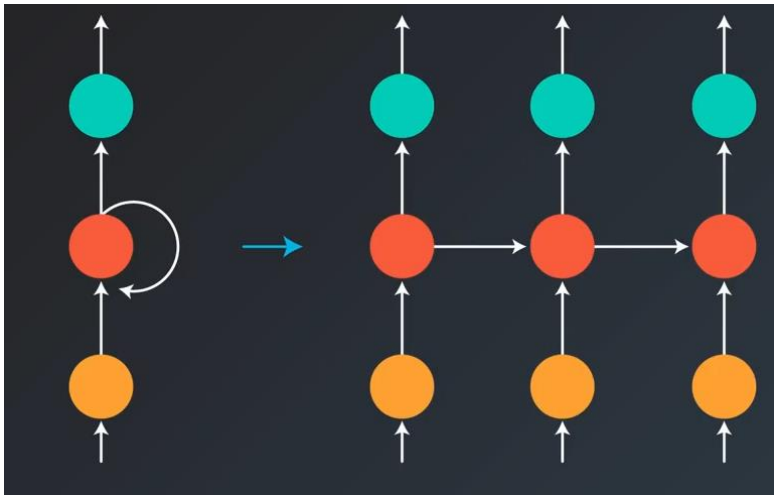


But going even deeper rarely helps much more.

The exception to this is convolutional nerual network where the deeper they are the better they perform.

RNN Hyper Paramter

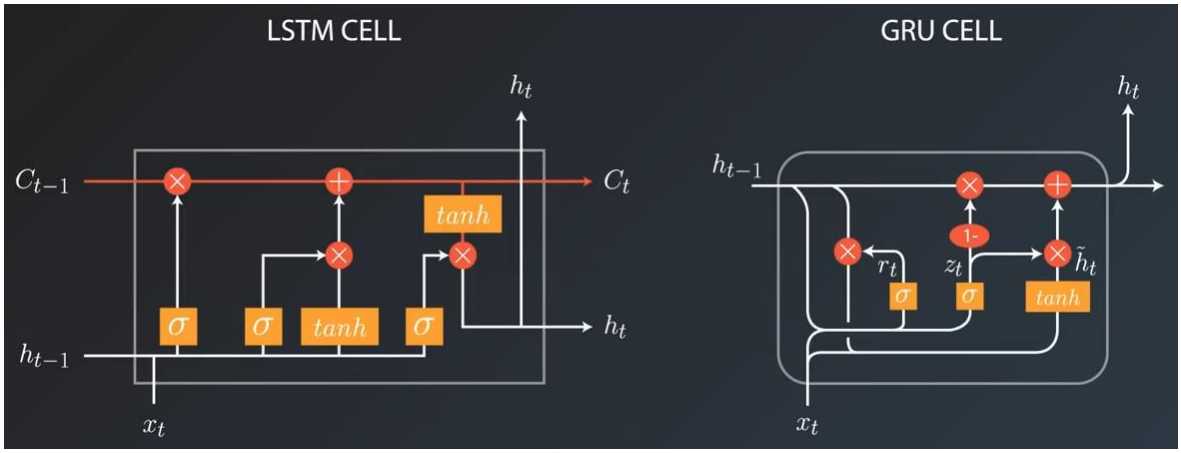
Two main choices we need to make when we need to build a recurrent neural network.



1- **Choose a cell type so a long short-term memory cell or vanilla RNN cell or gated recurrent unit cell**
2- **How deep the model is.**

How many layers will we stack? And since we will need word embeddings if our inputs are words, we will also look at embedding dimensionality.

In practice, LSTMs and GRUs perform better than vanilla RNN.



That much is clear.

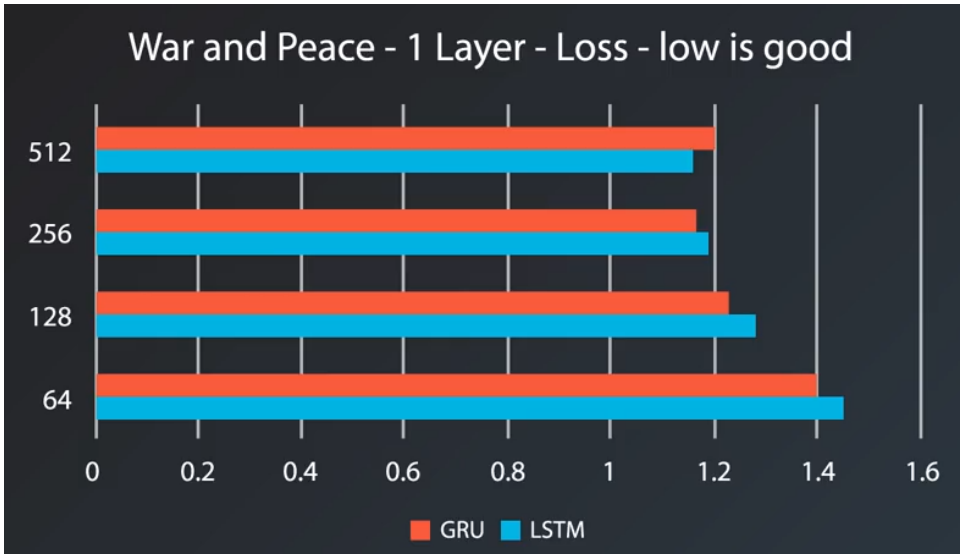
So, which of the two should you use? While LSTMs seemed to be more commonly used, it really depends on the task and the dataset.

Multiple researcher papers comparing the two did not announce a clear winner

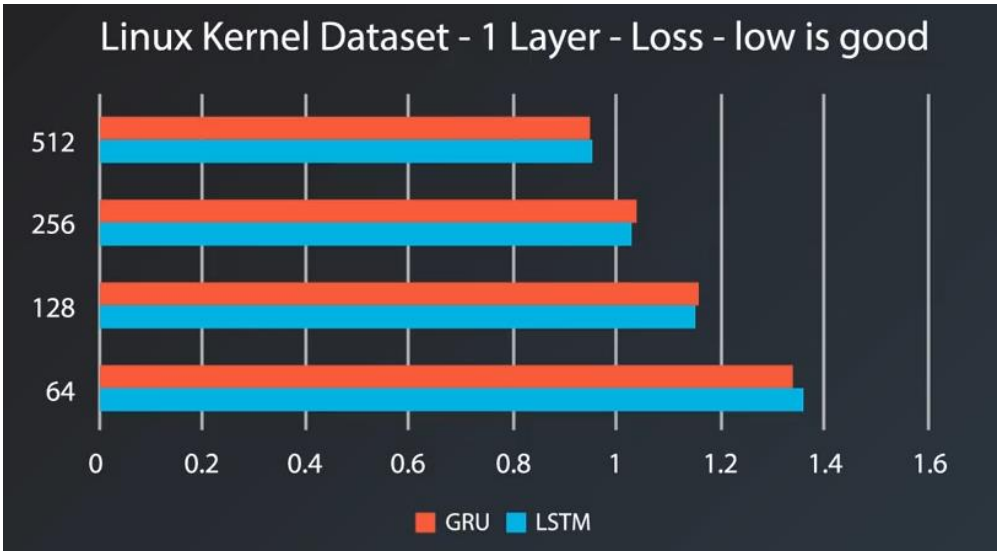
Let`s take an example task, character level language modeling.

A great paper titled visualizing and understanding your recurrent networks compare the two on two datasets the result on the first dataset saw GRUs doing better that LSTMs.

Better here being lower across entropy loss when comparting them at different sizes.



On a different dataset the two are tied each scoring better at different sizes.



In the text video below the video we elaborate more on this comparison.

But the recommendation here is to try both you're on dataset and task and compare.

Note that you don`t have to test this on your entire dataset.

You can try it on a random subset or your data.

Regarding the number of layers, result for character level language modeling show that a depth of at least two is shown to be beneficial.

But increasing it to three actually gives mixed results.

Another task like advanced speech recognition can shown improvements with five and even seven layers often without LSTMs cells

HYBERPARAMTER

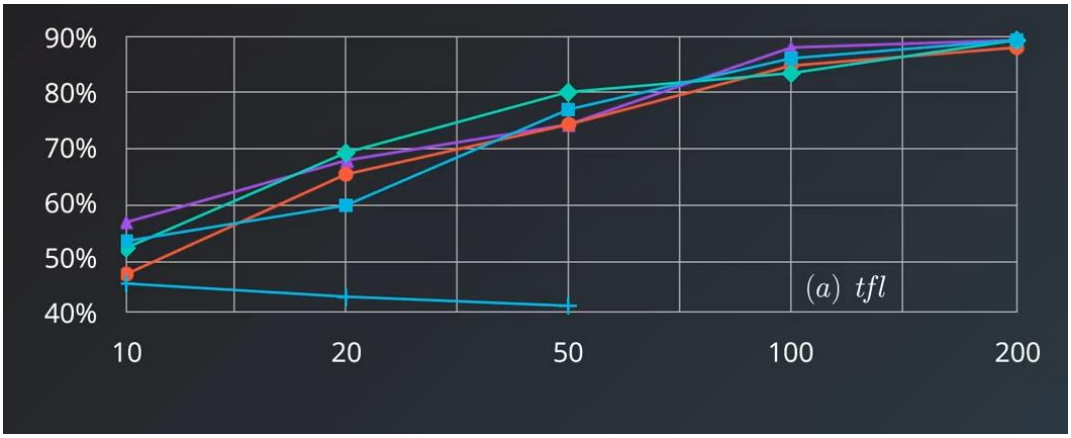
Model	Layers	Size	...	Word Error %
CTC CD Phone	5	600		15.5
CTC CD Phone	7	1000		14.2
CTC CD Phone	7	1000		14.7
CTC Spoken Words	5	600		14.5
CTC Spoken Words	7	1000		13.5
CTC Written Words	7	1000		13.4

In the text below the video we have provided number examples architectures for different tasks along with sizes and number of layers of each example.

If your RNN will be using words as inputs, then you will need to choose a size for the embedding vector.

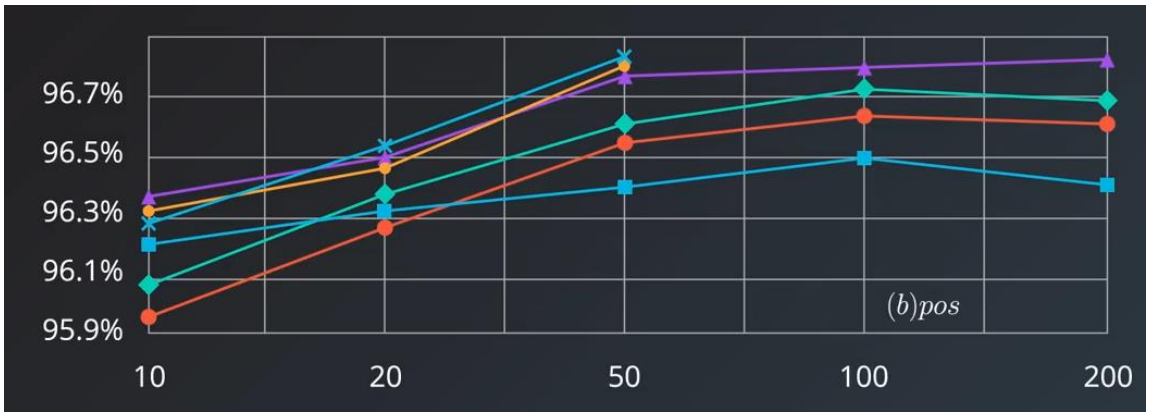
How do you go about choosing this number ?

Experimental results reporting a paper titled how to generate a good word embedding show that the performance of some tasks improve the larger we make the embedding, at least until a size of 200.



In another tests, however, only marginal improvements are released beyond the size of 50.

HYBERPARAMTER



Paper

Source: Lai, Siwei, et al. "How to generate a good word embedding." IEEE Intelligent Systems 31.6 (2016): 5-14.