First of all, tf.train.GradientDescentOptimizer is designed to use a constant learning rate for all variables in all steps. TensorFlow also provides out-of-the-box adaptive optimizers including the [tf.train.AdagradOptimizer](http://www.tensorflow.org/api_docs/python/train.html#AdagradOptimizer) and the [tf.train.AdamOptimizer](http://www.tensorflow.org/api_docs/python/train.html#AdamOptimizer), and these can be used as drop-in replacements.

However, if you want to control the learning rate with otherwise-vanilla gradient descent, you can take advantage of the fact that the learning\_rate argument to the [tf.train.GradientDescentOptimizer constructor](http://www.tensorflow.org/api_docs/python/train.html#GradientDescentOptimizer.__init__) can be a Tensor object. This allows you to compute a different value for the learning rate in each step

**Adagrad is an adaptive learning rate method**. In Adagrad we adopt the learning rate to the parameters. **We perform larger updates for infrequent parameters and smaller updates for frequent parameters.**

It is **well suited when we have sparse data as in large scale neural networks. GloVe word embedding uses adagrad where infrequent words required a greater update and frequent words require smaller updates**.

The choice of optimization algorithm for your deep learning model can mean the difference between good results in minutes, hours, and days.

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

<https://towardsdatascience.com/how-to-train-neural-network-faster-with-optimizers-d297730b3713>

<https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3>

Finally, to top it all off, the algorithm is ineffective — it requires the use of the entire training set in each iteration. This means that in every epoch we have to look at all the examples in order to perform next optimisation step. This may not be a problem when the training set consists of several thousand examples, but as I mentioned in one of my [articles](https://towardsdatascience.com/preventing-deep-neural-network-from-overfitting-953458db800a)— neural networks work best when they have millions of records at their disposal. In that case, it is hard to imagine that in every iteration we use the whole set. It would be a waste of both time and computer memory. All these factors cause that gradient descent, in its purest form, does not work in most cases.. Solution: mini-batch gradient descent

The use of optimizers is a good idea in our case, because it makes the time cost to increase but we don’t care, we just want to be super precise. Once we have trained the nn, we save the weights and bias and it takes same time wether it was a hard trainng or not