**Improving Deep Networks**

**Weight Initialization 131**

Weight initialization is an important consideration in the design of a neural network model.

* The nodes in neural networks are composed of parameters referred to as weights used to calculate a weighted sum of the inputs.
* Neural network models are fit using an optimization algorithm called stochastic gradient descent that incrementally changes the network weights to minimize a loss function, hopefully resulting in a set of weights for the mode that is capable of making useful predictions.
* This optimization algorithm requires a starting point in the space of possible weight values from which to begin the optimization process. Weight initialization is a procedure to set the weights of a neural network to small random values that define the starting point for the optimization (learning or training) of the neural network model.
* Each time, a neural network is initialized with a different set of weights, resulting in a different starting point for the optimization process, and potentially resulting in a different final set of weights with different performance characteristics.

Historically, weight initialization follows simple heuristics, such as:

* Small random values in the range [-0.3, 0.3]
* Small random values in the range [0, 1]
* Small random values in the range [-1, 1]

These heuristics continue to work well in general.

*“We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter very much, but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize”*

Nevertheless, more tailored approaches have been developed over the last decade that have become the defacto standard given they may result in a slightly more effective optimization (model training) process.

These modern weight initialization techniques are divided based on the type of activation function used in the nodes that are being initialized, such as “Sigmoid and Tanh” and “ReLU.”

## Weight Initialization for Sigmoid and Tanh

The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called “glorot” or “xavier” initialization.

It is named for Xavier Glorot, currently a research scientist at Google DeepMind

There are two versions of this weight initialization method, which we will refer to as “xavier” and “normalized xavier.”

*Glorot and Bengio proposed to adopt a properly scaled uniform distribution for initialization. This is called “Xavier” initialization […] Its derivation is based on the assumption that the activations are linear. This assumption is invalid for ReLU and PReLU.*

Both approaches were derived assuming that the activation function is linear, nevertheless, they have become the standard for nonlinear activation functions like Sigmoid and Tanh, but not ReLU.

Let’s take a closer look at each in turn.

### Xavier Weight Initialization

The xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range -(1/sqrt(n)) and 1/sqrt(n), where n is the number of inputs to the node.

weight = U [-(1/sqrt(n)), 1/sqrt(n)]

### Normalized Xavier Weight Initialization

The normalized xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range -(sqrt(6)/sqrt(n + m)) and sqrt(6)/sqrt(n + m), where n us the number of inputs to the node (e.g. number of nodes in the previous layer) and m is the number of outputs from the layer (e.g. number of nodes in the current layer).

weight = U [-(sqrt(6)/sqrt(n + m)), sqrt(6)/sqrt(n + m)]

## Weight Initialization for ReLU

The “xavier” weight initialization was found to have problems when used to initialize networks that use the rectified linear (ReLU) activation function.

As such, a modified version of the approach was developed specifically for nodes and layers that use ReLU activation, popular in the hidden layers of most multilayer Perceptron and convolutional neural network models.

The current standard approach for initialization of the weights of neural network layers and nodes that use the rectified linear (ReLU) activation function is called **“he” initialization**.

It is named for Kaiming He, currently a research scientist at Facebook,

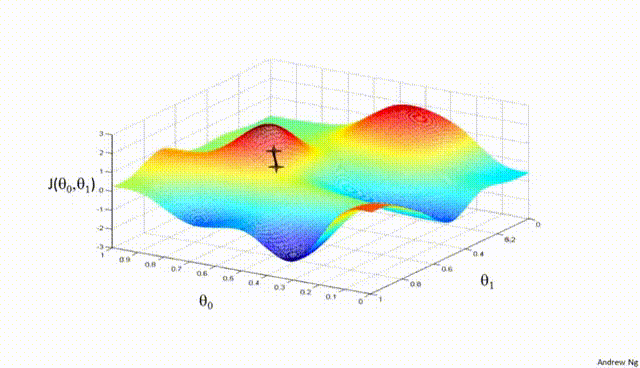
### He Weight Initialization

The he initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of sqrt(2/n), where n is the number of inputs to the node.

weight = G (0.0, sqrt(2/n))

**Unstable Gradients 137**

* After propagating the input features forward to the output layer through the various hidden layers consisting of different/same activation functions, we come up with a predicted probability of a sample belonging to the positive class ( generally, for classification tasks).
* Now, the backpropagation algorithm propagates backward from the output layer to the input layer calculating the error gradients on the way.
* Once the computation for gradients of the cost function w.r.t each parameter (weights and biases) in the neural network is done, the algorithm takes a gradient descent step towards the minimum to update the value of each parameter in the network using these gradients.



**Understanding the Problems**

#### Vanishing –

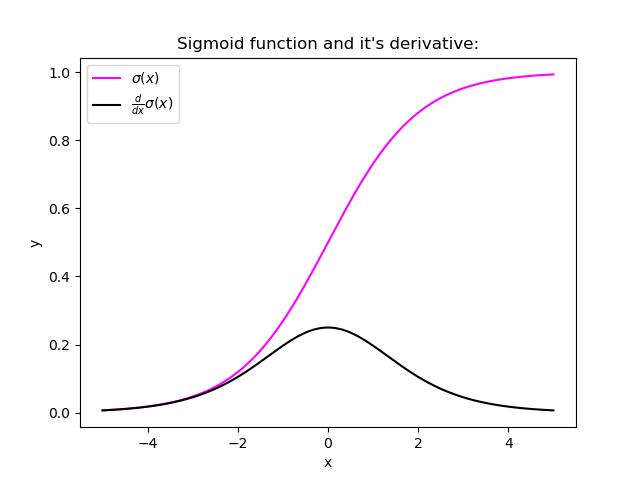
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the **vanishing gradients** problem.

#### Exploding –

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the **exploding gradients** problem.

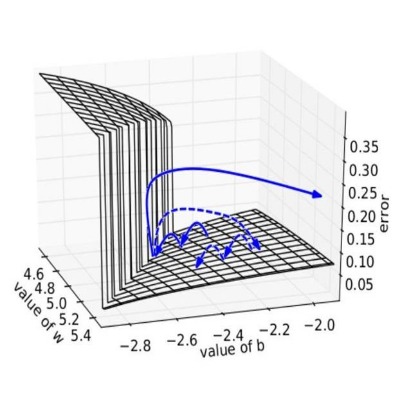
## *Why do the gradients even vanish/explode?*

Certain activation functions, like the logistic function (sigmoid), have a very huge difference between the variance of their inputs and the outputs. In simpler words, they shrink and transform a larger input space into a smaller output space that lies between the range of [0,1].



Observing the above graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.



## *How to know if our model is suffering from the Exploding/Vanishing gradient problem?*

Following are some signs that can indicate that our gradients are exploding/vanishing :

|  |  |
| --- | --- |
| **Exploding** | **Vanishing** |
| There is an exponential growth in the model parameters. | The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all). |
| The model weights may become NaN during training. | The model weights may become 0 during training. |
| The model experiences  avalanche learning. | The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations. |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Certainly, neither do we want our signal to explode or saturate nor do we want it to die out. The signal needs to flow properly both in the forward direction when making predictions as well as in the backward direction while calculating gradients.

## Solutions

Now that we are well aware of the vanishing/exploding gradients problems, it’s time to learn some techniques that can be used to fix the respective problems.

### 1. Proper Weight Initialization

In their paper, researchers Xavier Glorot, Antoine Bordes, and Yoshua Bengio proposed a way to remarkably alleviate this problem.

For the proper flow of the signal, the authors argue that:

1. The variance of outputs of each layer should be equal to the variance of its inputs.
2. The gradients should have equal variance before and after flowing through a layer in the reverse direction.

Although it is impossible for both conditions to hold for any layer in the network until and unless the number of inputs to the layer ( fanin ) is equal to the number of neurons in the layer ( fanout ), but they proposed a well-proven compromise that works incredibly well in practice: randomly initialize the connection weights for each layer in the network as described in the following equation which is popularly known as **Xavier initialization** (after the author’s first name) or **Glorot initialization** (after his last name).

*where  fanavg= ( fanin+ fanout) / 2*

* Normal distribution with mean *0* and variance *σ2 = 1/ fanavg*
* Or a uniform distribution between *-r*  and *+r* , with *r = sqrt( 3 / fanavg )*

Following are some more very popular weight initialization strategies for different activation functions, they only differ by the scale of variance and by the usage of either fanavg or fanin

*for uniform distribution, calculate r as:****r = sqrt( 3\*σ2****)*



*Using the above initialization strategies can significantly speed up the training and increase the odds of gradient descent converging at a lower generalization error.*

### *Wait, but how do we put these strategies into code ??*

Relax! we will not need to hardcode anything, Keras does it for us.

* Keras uses Xavier’s initialization strategy with uniform distribution.
* If we wish to use a different strategy than the default one, this can be done using the **kernel\_initializer** parameter while creating the layer. For example :

keras.layer.Dense(25, activation = "relu", kernel\_initializer="he\_normal")

or

keras.layer.Dense(25, activation = "relu", kernel\_initializer="he\_uniform")

* If we wish to use  use the initialization based on *fanavg*rather than *fanin*, we can use the VarianceScaling initializer like this :

he\_avg\_init = keras.initializers.VarianceScaling(scale=2., mode='fan\_avg', distribution='uniform')

keras.layers.Dense(20, activation="sigmoid", kernel\_initializer=he\_avg\_init)

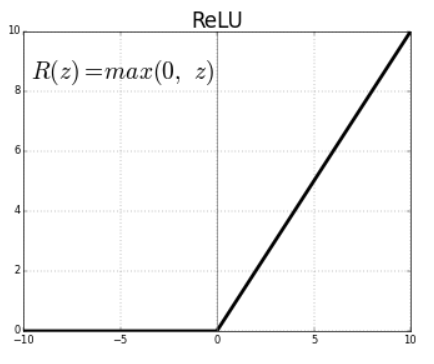
### 

### 2. Using Non-saturating Activation Functions

In an earlier section, while studying the nature of sigmoid activation function, we observed that its nature of saturating for larger inputs (negative or positive) came out to be a major reason behind the vanishing of gradients thus making it non-recommendable to use in the hidden layers of the network.

So to tackle the issue regarding the saturation of activation functions like sigmoid and tanh, we must use some other non-saturating functions like ReLu and its alternatives.

**ReLU ( Rectified Linear Unit )**

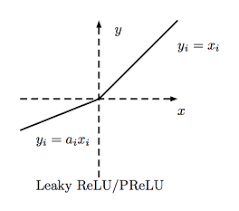


* *Relu(z) = max(0,z)*
* Outputs 0 for any negative input.
* Range: [0, infinity]

Unfortunately, the ReLu function is also not a perfect pick for the intermediate layers of the network “in some cases”. It suffers from a problem known as dying ReLus wherein some neurons just die out, meaning they keep on throwing 0 as outputs with the advancement in training.

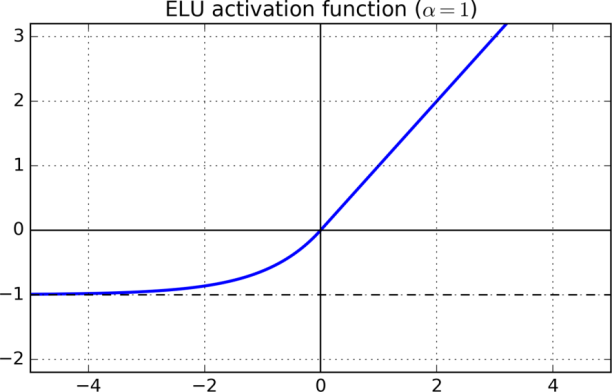
Some popular alternative functions of the ReLU that mitigates the problem of vanishing gradients when used as activation for the intermediate layers of the network  are LReLU, PReLU, ELU, SELU :

## ****LReLU (Leaky ReLU)****



* LeakyReLUα(z) = max(αz, z)
* The amount of “leak” is controlled by the hyperparameter **α**, it is the slope of the function for z < 0.
* The smaller slope for the leak ensures that the neurons powered by leaky Relu never die; although they might venture into a state of coma for a long training phase they always have a chance to eventually wake up.
* **α** can also be trained, that is, the model learns the value of α during training. This variant wherein α is now considered a parameter rather than a hyperparameter is called parametric leaky ReLu (**PReLU**).

**ELU (Exponential Linear Unit)**



* For z < 0, it takes on negative values which allow the unit to have an average output closer to 0 thus alleviating the vanishing gradient problem
* For z < 0, the gradients are non zero. This avoids the dead neurons problem.
* For α = 1, the function is smooth everywhere, this speeds up the gradient descent since it does not bounce right and left around z=0.
* A scaled version of this function ( **SELU:**Scaled ELU ) is also used very often in Deep Learning.

### 3. Batch Normalization

Using He initialization along with any variant of the ReLU activation function can significantly reduce the chances of vanishing/exploding problems at the beginning. However, it does not guarantee that the problem won’t reappear during training.

In 2015, Sergey Ioffe and Christian Szegedy proposed a paper in which they introduced a technique known as Batch Normalization to address the problem of vanishing/exploding gradients.

The Following key points explain the intuition behind BN and how it works:

* It consists of adding an operation in the model just before or after the activation function of each hidden layer.
* This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
* In other words, the operation lets the model learn the optimal scale and mean of each of the layer’s inputs.
* To zero-center and normalize the inputs, the algorithm needs to estimate each input’s mean and standard deviation.
* It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name “Batch Normalization”).

model = keras.models.Sequential([keras.layers.Flatten(input\_shape=[28, 28]),

keras.layers.BatchNormalization(),

keras.layers.Dense(300, activation="relu"),

keras.layers.BatchNormalization(),

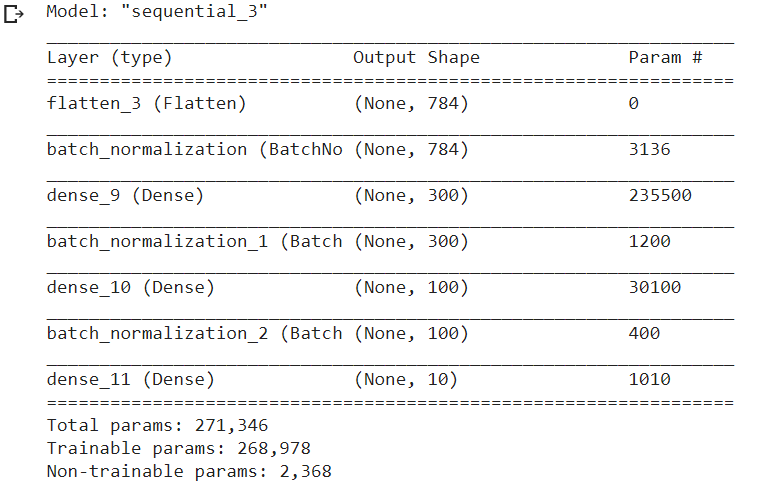
keras.layers.Dense(100, activation="relu"),

keras.layers.BatchNormalization(),

keras.layers.Dense(10, activation="softmax")])

we just added batch normalization after each layer ( dataset : FMNIST)

model.summary()



### 4. Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called Gradient Clipping.

* This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0.
* Meaning, all the partial derivatives of the loss w.r.t each  trainable parameter will be clipped between –1.0 and 1.0

optimizer = keras.optimizers.SGD(clipvalue = 1.0)

* The threshold is a hyperparameter we can tune.
* The orientation of the gradient vector may change due to this: for eg, let the original gradient vector be [0.9, 100.0] pointing mostly in the direction of the second axis, but once we clip it by some value, we get [0.9, 1.0] which now points somewhere around the diagonal between the two axes.
* To ensure that the orientation remains intact even after clipping, we should clip by norm rather than by value.

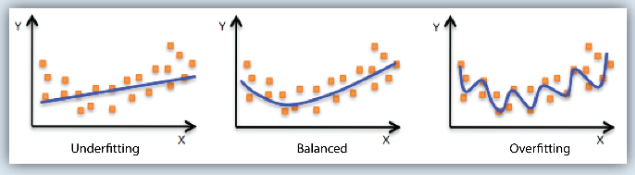
optimizer = keras.optimizers.SGD(clipnorm = 1.0)

* Now the whole gradient will be clipped if the threshold we picked is less than its ℓ2 norm. For eg: if clipnorm=1 then the vector [0.9, 100.0] will be clipped to [0.00899, 0.999995] , thus preserving its orientation.

**Model Generalization (Avoiding Overfitting) 140**

It is only with supervised learning that overfitting is a potential problem. Supervised learning in machine learning is one method for the model to learn and understand data.

With supervised learning, a model is given a set of labeled training data. The model learns to make predictions based on this training data, so the more training data the model has access to, the better it gets at making predictions. With training data, the outcome is already known. The predictions from the model and known outcomes are compared, and the model’s parameters are changed until the two align. The point of training is to develop the model’s ability to successfully generalize.



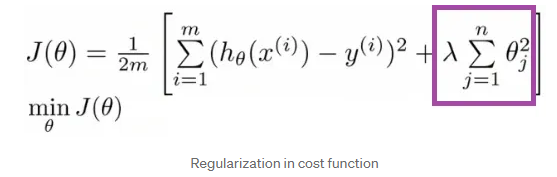
Generalization is a term used to describe a model’s ability to react to new data. That is, after being trained on a training set, a model can digest new data and make accurate predictions. A model’s ability to generalize is central to the success of a model. If a model has been trained too well on training data, it will be unable to generalize. It will make inaccurate predictions when given new data, making the model useless even though it is able to make accurate predictions for the training data. This is called overfitting. The inverse is also true. Underfitting happens when a model has not been trained enough on the data. In the case of underfitting, it makes the model just as useless and it is not capable of making accurate predictions, even with the training data.

The figure demonstrates the three concepts discussed above. On the left, the blue line represents a model that is underfitting. The model notes that there is some trend in the data, but it is not specific enough to capture relevant information. It is unable to make accurate predictions for training or new data. In the middle, the blue line represents a model that is balanced. This model notes there is a trend in the data, and accurately models it. This middle model will be able to generalize successfully. On the right, the blue line represents a model that is overfitting. The model notes a trend in the data, and accurately models the training data, but it is too specific. It will fail to make accurate predictions with new data because it learned the training data too well.

**L1 and L2 Regularization 141**

In supervised machine learning, models are trained on a subset of data aka training data. The goal is to compute the target of each training example from the training data. Now, overfitting happens when model learns signal as well as noise in the training data and wouldn’t perform well on new data on which model wasn’t trained on. Now, there are few ways you can avoid overfitting your model on training data like cross-validation sampling, reducing number of features, pruning, regularization etc.

Regularization basically adds the penalty as model complexity increases. Regularization parameter (lambda) penalizes all the parameters except intercept so that model generalizes the data and won’t overfit.



Regularization will add the penalty for higher terms. This will decrease the importance given to higher terms and will bring the model towards less complex equation.

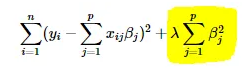
In order to create less complex (parsimonious) model when you have a large number of features in your dataset, some of the Regularization techniques used to address over-fitting and feature selection are:

* + **L1 Regularization**
  + **L2 Regularization**

A regression model that uses L1 regularization technique is called **Lasso Regression** and model which uses L2 is called **Ridge Regression**.

The key difference between these two is the penalty term.

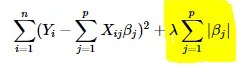
**Ridge regression** adds “squared magnitude” of coefficient as penalty term to the loss function. Here the highlighted part represents L2 regularization element.



Cost function

Here, if lambda is zero then you can imagine we get back OLS. However, if lambda is very large then it will add too much weight and it will lead to under-fitting. Having said that it’s important how lambda is chosen. This technique works very well to avoid over-fitting issue.

**Lasso Regression** (Least Absolute Shrinkage and Selection Operator) adds “absolute value of magnitude” of coefficient as penalty term to the loss function.



Cost function

Again, if lambda is zero then we will get back OLS whereas very large value will make coefficients zero hence it will under-fit.

The **key difference** between these techniques is that Lasso shrinks the less important feature’s coefficient to zero thus, removing some feature altogether. So, this works well for **feature selection** in case we have a huge number of features.

Traditional methods like cross-validation, stepwise regression to handle overfitting and perform feature selection work well with a small set of features but these techniques are a great alternative when we are dealing with a large set of features.

**Dropout 142**

*The term “dropout” refers to dropping out the nodes (input and hidden layer) in a neural network (as seen in Figure 1). All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p.*

Let’s try to understand with a given input x: {1, 2, 3, 4, 5} to the fully connected layer. We have a dropout layer with probability p = 0.2 (or keep probability = 0.8). During the forward propagation (training) from the input x, 20% of the nodes would be dropped, i.e. the x could become {1, 0, 3, 4, 5} or {1, 2, 0, 4, 5} and so on. Similarly, it applied to the hidden layers.

For instance, if the hidden layers have 1000 neurons (nodes) and a dropout is applied with drop probability = 0.5, then 500 neurons would be randomly dropped in every iteration (batch).

Generally, for the input layers, the keep probability, i.e. 1- drop probability, is closer to 1, 0.8 being the best as suggested by the authors. For the hidden layers, the greater the drop probability more sparse the model, where 0.5 is the most optimised keep probability, that states dropping 50% of the nodes.

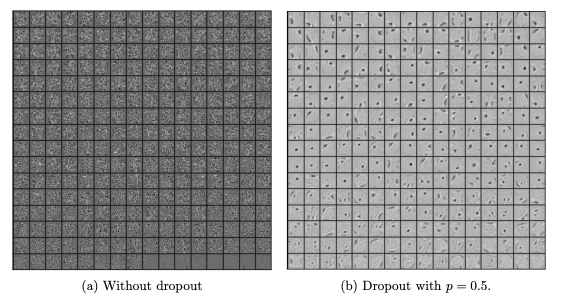
So how does dropout solves the problem of overfitting?

# How does it solve the Overfitting problem?

In the overfitting problem, the model learns the statistical noise. To be precise, the main motive of training is to decrease the loss function, given all the units (neurons). So in overfitting, a unit may change in a way that fixes up the mistakes of the other units. This leads to complex co-adaptations, which in turn leads to the overfitting problem because this complex co-adaptation fails to generalise on the unseen dataset.

Now, if we use dropout, it prevents these units to fix up the mistake of other units, thus preventing co-adaptation, as in every iteration the presence of a unit is highly unreliable. So by randomly dropping a few units (nodes), it forces the layers to take more or less responsibility for the input by taking a probabilistic approach.

This ensures that the model is getting generalised and hence reducing the overfitting problem.



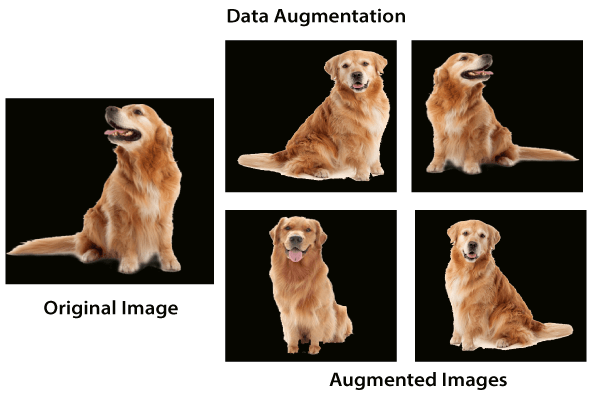
**Figure 2:** (a) Hidden layer features without dropout; (b) Hidden layer features with dropout (Image by [Nitish](https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf" \t "_blank))

From figure 2, we can easily make out that the hidden layer with dropout is learning more of the generalised features than the co-adaptations in the layer without dropout. It is quite apparent, that dropout breaks such inter-unit relations and focuses more on generalisation.

**Data Augmentation 145**

It is the process by which we create new data for our model to use during the training process.

This is done by taking our existing dataset and transforming or altering the image in useful ways to create new images.

  
After applying the transformation, the newly created images are known as augmented images because they essentially allow us to augment our dataset by adding new data to it. The data augmentation technique is useful because it allows our model to look at each image in our dataset from a variety of different perspective. This allows our model to extract relevant features more accurately and to obtain more feature-related data from each training image.

Now our biggest question is how we will use that augmentation to reduce overfitting. The overfitting occurs when our model is too closely fit the training set.

There is no need to start collecting new images and adding them to our datasets. We can use data augmentation which introduces minor alteration to our existing datasets such darker shading, flips, zooming, rotations or translation. Our model will interpret them as separate distinct images. It will not only reduce over fitting but it also prevents our network from learning irrelevant patterns and boosts overall performance. We have the following steps to perform data augmentation:

**Step 1:**

To perform data augmentation on training dataset, we have to make to make a separate transform statement. For validation dataset the transform will remain same. So we first copy our transform1 statement and treat it as transform\_train as:

transform\_train=transforms.Compose([transforms.Resize((32,32)),transforms.ToTensor(),transforms.Normalize((0.5,),(0.5,))])

**Step 2:**

Now, we will add alternation in our transform\_train statement. The alternations will be a RandomHorizontalFlip, RandomRotation which is used for rotation of an image by a certain angle and that angle will be passes as an argument.

transform\_train=transforms.Compose([transforms.Resize((32,32)),

        transform.RandomHorizontalFlip(),

        transform.RandomRotation(),

        transforms.ToTensor(),

        transforms.Normalize((0.5,),(0.5,))])

To add even more variety to our dataset, we will use a fine type transformation. Fine transformation represent simple transformation which preserve straight lines and planes with the object. Scaling, translation, shear and zooming is a transformation which fits this category.

transform\_train=transforms.Compose([transforms.Resize((32,32)),

        transform.RandomHorizontalFlip(),

        transform.RandomRotation(),

        transform.RandomAffine(0,shear=10,scale=(0.8,1.2)),

        transforms.ToTensor(),

        transforms.Normalize((0.5,),(0.5,))])

In RandomAffine(), the first argument is decrease which we set zero to deactivate rotation, second argument is the shear transformation and the last one is the scaling transformation and use a topple to define the range of zoom which we have required. We defined a lower and upper limit of 0.8 and 1.2 to scale images to 80 or 120 percent of their size.

**Step 3:**

Now, we move onto our next augmentation to create new augmented images with a randomized variety of brightness, contrast and saturation. We will add another transformation i.e. ColorJitter as:

transform\_train=transforms.Compose([transforms.Resize((32,32)),

        transform.RandomHorizontalFlip(),

        transform.RandomRotation(10),

        transform.RandomAffine(0,shear=10,scale=(0.8,1.2)),

        transform.ColorJitter(brightness=0.2,contrast=0.2,saturation=0.2)

        transforms.ToTensor(),

        transforms.Normalize((0.5,),(0.5,))])

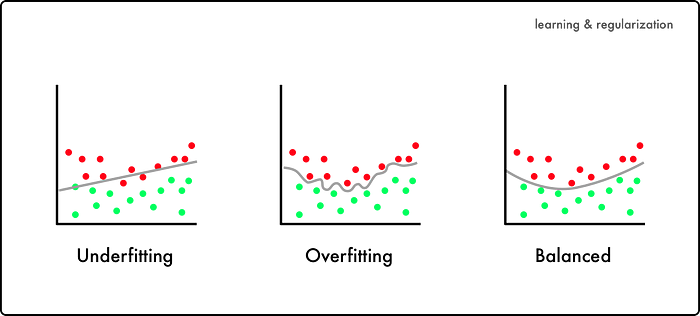
**Step 4:**

Before executing our code, we have to change the training\_dataset statement because now we have another transform for the training dataset. So

training\_dataset=datasets.CIFAR10(root='./data',train=True,download=True,transform=transform\_train

# Summary on different Techniques to Prevent Overfitting

Overfitting occurs when the model performs well on training data but generalizes poorly to unseen data. Overfitting is a very common problem in Machine Learning and there has been an extensive range of literature dedicated to studying methods for preventing overfitting. In the following, I’ll describe eight simple approaches to alleviate overfitting by introducing only one change to the data, model, or learning algorithm in each approach.



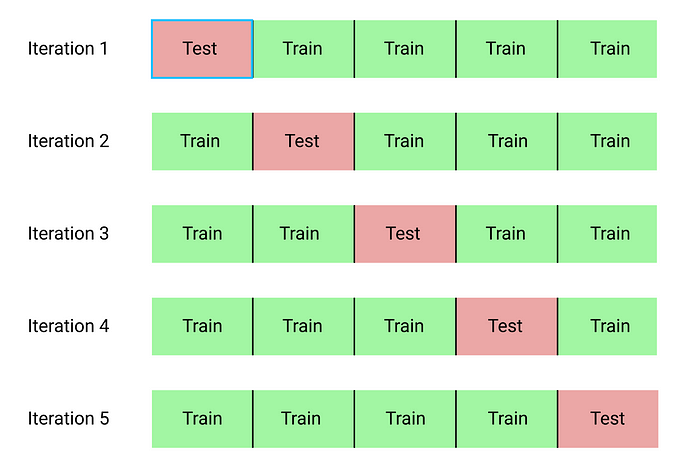
1. [Hold-out](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#c287)
2. [Cross-validation](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#d2d6)
3. [Data augmentation](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#f80d)
4. [Feature selection](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#253a)
5. [L1 / L2 regularization](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#d178)
6. [Remove layers / number of units per layer](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#87f3)
7. [Dropout](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#6f6a)
8. [Early stopping](https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d#98ac)

# 1. Hold-out (data)

Rather than using all of our data for training, we can simply split our dataset into two sets: training and testing. A common split ratio is 80% for training and 20% for testing. We train our model until it performs well not only on the training set but also for the testing set. This indicates good generalization capability since the testing set represents unseen data that were not used for training. However, this approach would require a sufficiently large dataset to train on even after splitting.

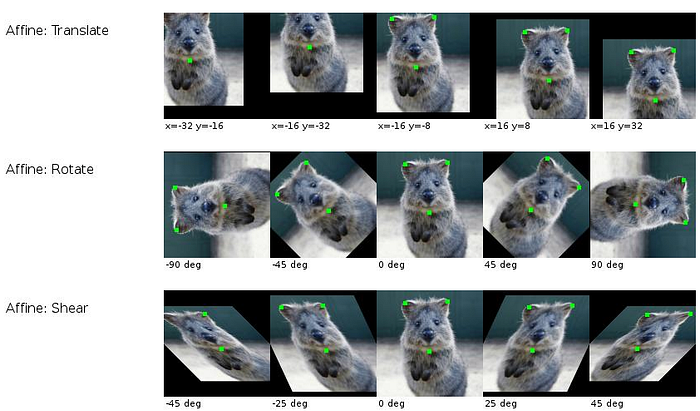
# 2. Cross-validation (data)

We can split our dataset into k groups (k-fold cross-validation). We let one of the groups to be the testing set (please see hold-out explanation) and the others as the training set, and repeat this process until each individual group has been used as the testing set (e.g., k repeats). Unlike hold-out, cross-validation allows all data to be eventually used for training but is also more computationally expensive than hold-out.



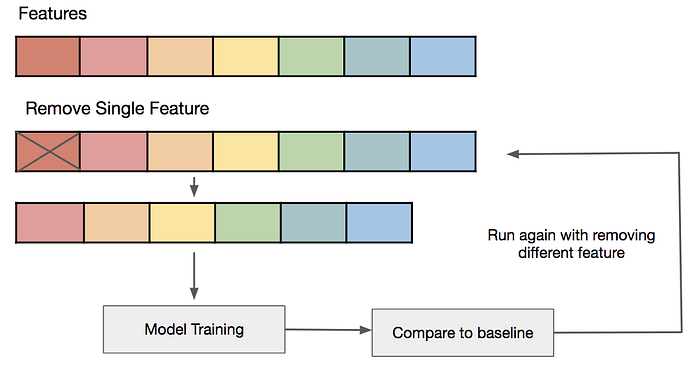
# 3. Data augmentation (data)

A larger dataset would reduce overfitting. If we cannot gather more data and are constrained to the data we have in our current dataset, we can apply data augmentation to artificially increase the size of our dataset. For example, if we are training for an image classification task, we can perform various image transformations to our image dataset (e.g., flipping, rotating, rescaling, shifting).



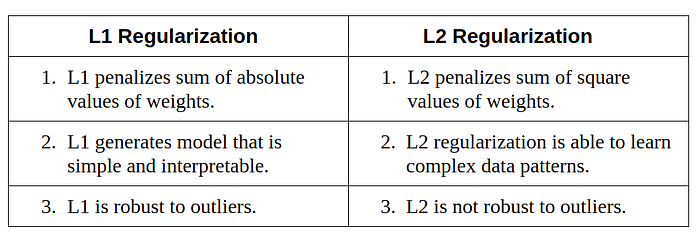
# 4. Feature selection (data)

If we have only a limited amount of training samples, each with a large number of features, we should only select the most important features for training so that our model doesn’t need to learn for so many features and eventually overfit. We can simply test out different features, train individual models for these features, and evaluate generalization capabilities, or use one of the various widely used feature selection methods.



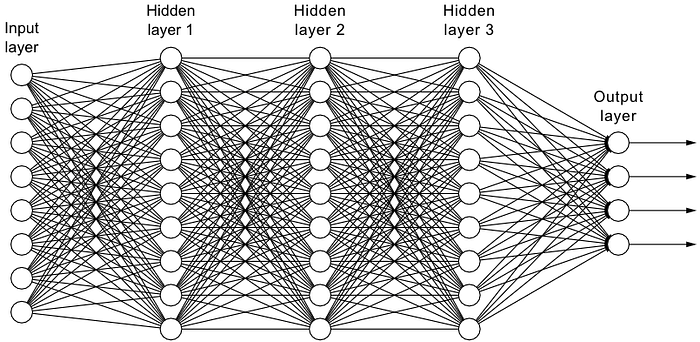
# 5. L1 / L2 regularization (learning algorithm)

Regularization is a technique to constrain our network from learning a model that is too complex, which may therefore overfit. In L1 or L2 regularization, we can add a penalty term on the cost function to push the estimated coefficients towards zero (and not take more extreme values). L2 regularization allows weights to decay towards zero but not to zero, while L1 regularization allows weights to decay to zero.



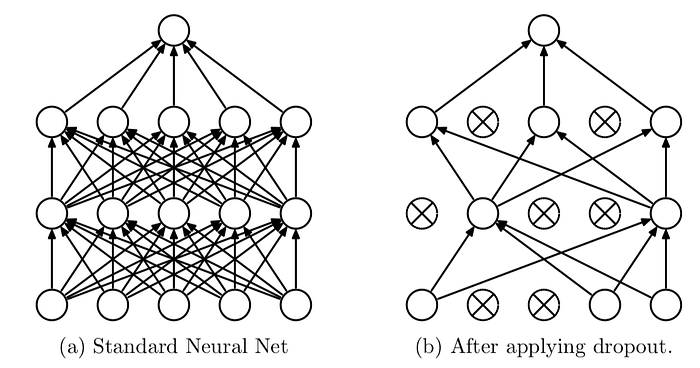
# 6. Remove layers / number of units per layer (model)

As mentioned in L1 or L2 regularization, an over-complex model may more likely overfit. Therefore, we can directly reduce the model’s complexity by removing layers and reduce the size of our model. We may further reduce complexity by decreasing the number of neurons in the fully-connected layers. We should have a model with a complexity that sufficiently balances between underfitting and overfitting for our task.



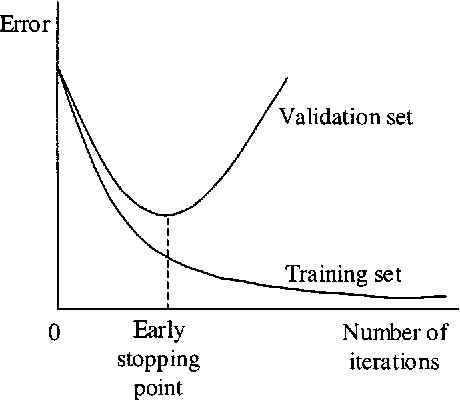
# 7. Dropout (model)

By applying dropout, which is a form of regularization, to our layers, we ignore a subset of units of our network with a set probability. Using dropout, we can reduce interdependent learning among units, which may have led to overfitting. However, with dropout, we would need more epochs for our model to converge.



# 8. Early stopping (model)

We can first train our model for an arbitrarily large number of epochs and plot the validation loss graph (e.g., using hold-out). Once the validation loss begins to degrade (e.g., stops decreasing but rather begins increasing), we stop the training and save the current model. We can implement this either by monitoring the loss graph or set an early stopping trigger. The saved model would be the optimal model for generalization among different training epoch values.



**Fancy Optimizers 145**

Choosing an appropriate optimizer for a deep learning model is important as it can greatly impact its performance. Optimization algorithms have different strengths and weaknesses and are better suited for certain problems and architectures.

**For example,** stochastic gradient descent is a simple and efficient optimizer that is widely used, but it may need help to converge on problems with complex, non-convex loss functions. On the other hand, Adam is a more sophisticated optimizer that combines the ideas of momentum and adaptive learning rates and is often considered one of the most effective optimizers in deep learning.

Here are a few pointers to keep in mind when choosing an optimizer:

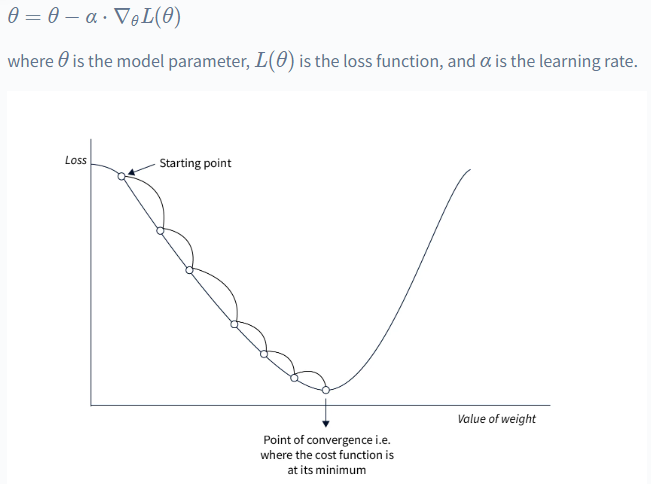
* Understand the problem and model architecture, as this will help you determine which optimizer is most suitable
* Experiment with different optimizers in deep learning to see which one works best for your problem
* Adjust the hyperparameters of the optimizer, such as the learning rate, to see if it improves performance
* Remember that the optimizer's choice is not the only factor affecting model performance.
* Other important factors include the choice of architecture, the quality of the data, and the amount of data available.

## Types of Optimizers

Many types of optimizers are available for training machine learning models, each with its **own strengths and weaknesses**. Some optimizers are better suited for certain types of models or data, while others are more general-purpose.

### Gradient Descent

Gradient descent is a simple optimization algorithm that updates the model's parameters to minimize the loss function. We can write the basic form of the algorithm as follows:



**Pros:**

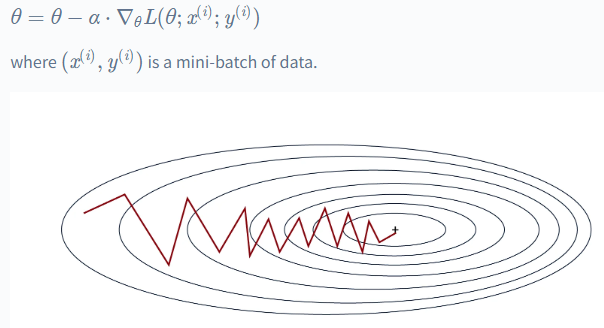
* Simple to implement.
* Can work well with a well-tuned learning rate.

**Cons:**

* It can converge slowly, especially for complex models or large datasets.
* Sensitive to the choice of learning rate.

### Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a variant of gradient descent that involves updating the parameters based on a small, randomly-selected subset of the data (i.e., a "mini-batch") rather than the full dataset. We can write the basic form of the algorithm as follows:



**Pros:**

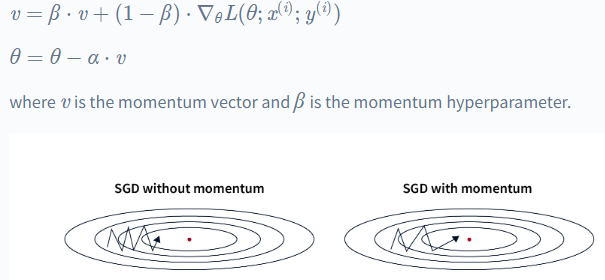
* It can be faster than standard gradient descent, especially for large datasets.
* Can escape local minima more easily.

**Cons:**

* It can be noisy, leading to less stability.
* It may require more hyperparameter tuning to get good performance.

### Stochastic Gradient Descent with Momentum

SGD with momentum is a variant of SGD that adds a "momentum" term to the update rule, which helps the optimizer to continue moving in the same direction even if the local gradient is small. The momentum term is typically set to a value between 0 and 1. We can write the update rule as follows:



**Pros:**

* It can help the optimizer to move more efficiently through "flat" regions of the loss function.
* It can help to reduce oscillations and improve convergence.

**Cons:**

* Can overshoot good solutions and settle for suboptimal ones if the momentum is too high.
* Requires tuning of the momentum hyperparameter.

**Nesterov Accelerated Gradient**

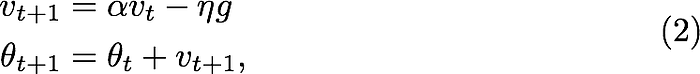
We recall the update rule of Stochastic Gradient Descent at each iteration *t*,

https://miro.medium.com/v2/resize:fit:700/1*fPRzjPkHgvEyACwH8weKXQ.png

where *θ* is the parameter that the algorithm will tweak to achieve acceptable loss, *g* is the gradients that show the opposite direction and how far should we tweak the parameters to minimize the loss, and *η* is the learning rate, a hyper-parameter that denotes how much we tweak our parameters with respect to the gradients, or how much we trust the gradients.

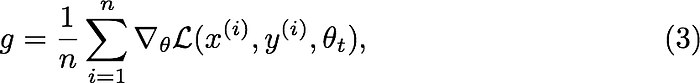
Unfortunately, there is some condition that even Stochastic Gradient Descent can become very slow, for example when the gradient is consistently small. This is due to the update rule of the algorithm that only depends on the gradients at each iteration only, as shown in equation (1). Noisy gradients can be a problem too since Stochastic Gradient Descent will frequently follow the wrong gradient.

In neural networks training, Momentum method can be used to mitigate these problems and accelerate learning compared to plain Stochastic Gradient Descent. Introduced in 1964 by Polyak, Momentum method is a technique that can accelerate gradient descent by taking accounts of previous gradients in the update rule at each iteration. This can be clearly observed in the update rule equation in every iteration, that is

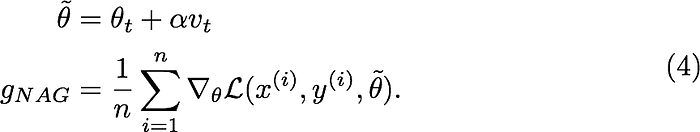


where *v* is the velocity term, the direction and speed at which the parameter should be tweaked and *α* is the decaying hyper-parameter, which determines how quickly accumulated previous gradients will decay, or how much we trust the accumulated previous gradients. If *α* is much bigger than *η*, the accumulated previous gradients will be dominant in the update rule so the gradient at the iteration will not change the current direction quickly. This is good in the condition when the gradient is noisy because the gradient will stay in the true direction for good. In the other hand, if *α* is much smaller than *η*, the accumulated previous gradients can act as a smoothing factor for the gradient.

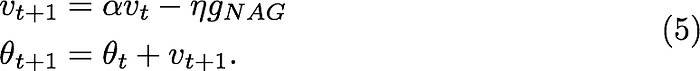
Another method that is closely related to Momentum method is Nesterov Accelerated Gradient. The difference between Momentum method and Nesterov Accelerated Gradient is in gradient computation phase. In Momentum method, the gradient was computed using current parameters *θ*𝑡



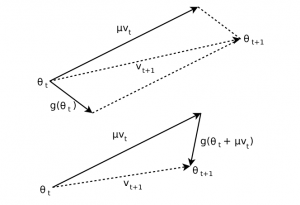
whereas in Nesterov Accelerated Gradient, we apply the velocity *vt* to the parameters *θ* to compute interim parameters *θ̃* . We then compute the gradient using the interim parameters



After we get the gradients, we update the parameters using similar update rule with the Momentum method (Eq. 2), with the only difference is the gradients



We can view Nesterov Accelerated Gradients as the correction factor for Momentum method. Consider the case when the velocity added to the parameters gives you immediate unwanted high loss, e.g., exploding gradient case. In this case, the Momentum method can be very slow since the optimization path taken exhibits large oscillations. In Nesterov Accelerated Gradient case, you can view it like *peeking* through the interim parameters where the added velocity will lead the parameters. If the velocity update leads to bad loss, then the gradients will direct the update back towards *θ*𝑡. This help Nesterov Accelerated Gradient to avoid the oscillations. The geometric visualization can be seen in Fig 1.

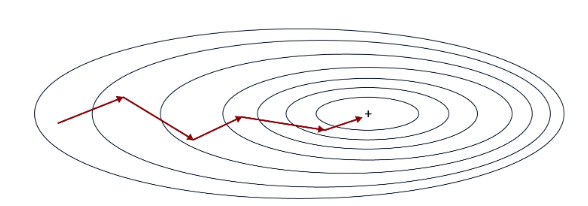


**Fig. 1** (Top) Momentum method, (Bottom) Nesterov Accelerated Gradient. *𝜇* is the decaying parameter, same as *α* in our case (Sutskever et al., 2013, Figure 1)

The distinction between Momentum method and Nesterov Accelerated Gradient updates was shown by Sutskever et al. in Theorem 2.1, i.e., both methods are distinct only when the learning rate *η* is reasonably large. When the learning rate *η* is relatively large, Nesterov Accelerated Gradients allows larger decay rate *α* than Momentum method, while preventing oscillations. The theorem also shows that both Momentum method and Nesterov Accelerated Gradient become equivalent when *η* is small.

### Mini-Batch Gradient Descent

Mini-batch gradient descent is similar to SGD, but instead of using a single sample to compute the gradient, it uses a **small, fixed-size "mini-batch" of samples**. The update rule is the same as for SGD, except that the gradient is averaged over the mini-batch. This can reduce noise in the updates and improve convergence.



**Pros:**

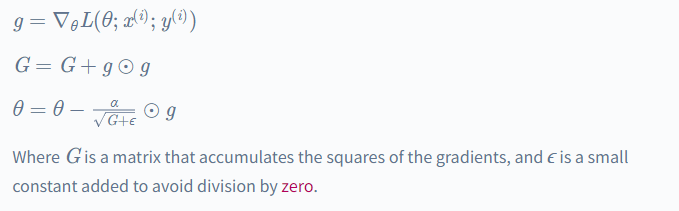
* It can be faster than standard gradient descent, especially for large datasets.
* Can escape local minima more easily.
* Can reduce noise in updates, leading to more stable convergence.

**Cons:**

* Can be sensitive to the choice of mini-batch size.

### Adagrad

Adagrad is an optimization algorithm that **uses an adaptive learning rate per parameter**. The learning rate is updated based on the historical gradient information so that parameters that receive many updates have a lower learning rate, and parameters that receive fewer updates have a larger learning rate. The update rule can be written as follows:



**Pros:**

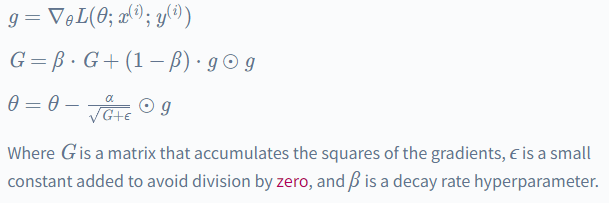
* It can work well with sparse data.
* Automatically adjusts learning rates based on parameter updates.

**Cons:**

* Can converge too slowly for some problems.
* Can stop learning altogether if the learning rates become too small.

### RMSProp

RMSProp is an optimization algorithm similar to Adagrad, but it **uses an exponentially decaying average** of the squares of the gradients rather than the sum. This helps to reduce the monotonic learning rate decay of Adagrad and improve convergence. We can write the update rule as follows:



**Pros:**

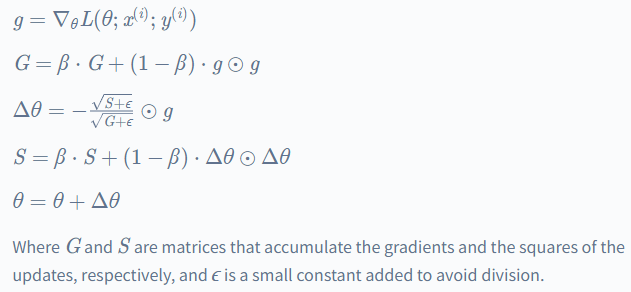
* It can work well with sparse data.
* Automatically adjusts learning rates based on parameter updates.
* Can converge faster than Adagrad.

**Cons:**

* It can still converge too slowly for some problems.
* Requires tuning of the decay rate hyperparameter.

### AdaDelta

AdaDelta is an optimization algorithm similar to RMSProp but does not require a hyperparameter learning rate. Instead, it uses an **exponentially decaying average** of the gradients and the squares of the gradients to determine the updated scale. We can write the update rule as follows:



**Pros:**

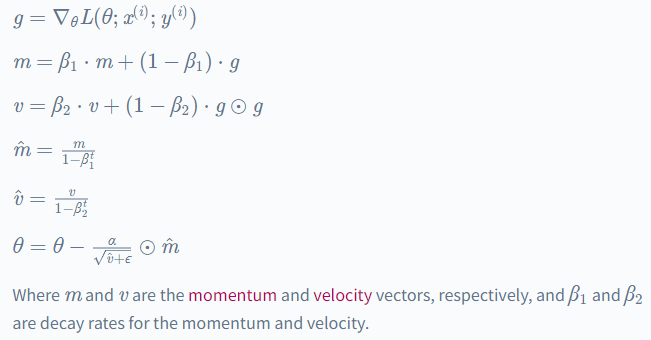
* Can work well with sparse data.
* Automatically adjusts learning rates based on parameter updates.

**Cons:**

* Can converge too slowly for some problems.
* Can stop learning altogether if the learning rates become too small.

### Adam

Adam (short for "adaptive moment estimation") is an optimization algorithm that combines the ideas of SGD with momentum and RMSProp. It **uses an exponentially decaying average of the gradients** and the squares of the gradients to determine the updated scale, similar to RMSProp. It also uses a momentum term to help the optimizer move more efficiently through the loss function. The update rule can be written as follows:



**Pros:**

* Can converge faster than other optimization algorithms.
* Can work well with noisy data.

**Cons:**

* It may require more tuning of hyperparameters than other algorithms.
* May perform better on some types of problems.

## How Do Optimizers Work in Deep Learning?

Optimizers in deep learning adjust the model's parameters to minimize the loss function. The loss function measures how well the model can make predictions on a given dataset, and the goal of training a model is to find the set of model parameters that yields the lowest possible loss.

The optimizer uses an optimization algorithm to search for the parameters that minimize the loss function. The optimization algorithm uses the gradients of the loss function to the model parameters to determine the direction in which we should adjust the parameters.

The **gradients** are computed using backpropagation, which involves applying the chain rule to compute the gradients of the loss function to each of the model parameters.

The **optimization algorithm** then adjusts the model parameters to minimize the loss function. This process is repeated until the loss function reaches a minimum or the optimizer reaches the maximum number of allowed iterations.

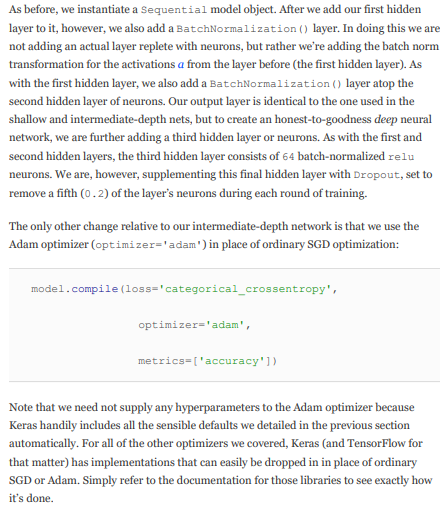
## Conclusion

* Optimizers in deep learning are essential, as they adjust the model's parameters to minimize the loss function.
* In general, the choice of which optimization algorithm to use will depend on the specific characteristics of the problem, such as the dataset's size and the model's complexity.
* It is important to consider each algorithm's pros and cons carefully and tune any relevant hyperparameters to achieve the best possible performance.
* Overall, understanding the role of optimizers in deep learning and the various available algorithms is essential for anyone looking to build and train effective machine learning models.

**A Deep Neural Network in Keras 147**

****

As before, we instantiate a Sequential model object. After we add our first hidden layer to it, however, we also add a BatchNormalization() layer. In doing this we are not adding an actual layer replete with neurons, but rather we’re adding the batch norm transformation for the activations a from the layer before (the first hidden layer). As with the first hidden layer, we also add a BatchNormalization() layer atop the second hidden layer of neurons. Our output layer is identical to the one used in the shallow and intermediate­depth nets, but to create an honest­to­goodness deep neural network, we are further adding a third hidden layer or neurons. As with the first and second hidden layers, the third hidden layer consists of 64 batch­normalized relu neurons. We are, however, supplementing this final hidden layer with Dropout, set to remove a fifth (0.2) of the layer’s neurons during each round of training. The only other change relative to our intermediate­depth network is that we use the Adam optimizer (optimizer='adam') in place of ordinary SGD optimization:

****

Note that we need not supply any hyperparameters to the Adam optimizer because Keras handily includes all the sensible defaults we detailed in the previous section automatically. For all of the other optimizers we covered, Keras (and TensorFlow for that matter) has implementations that can easily be dropped in in place of ordinary SGD or Adam. Simply refer to the documentation for those libraries to see exactly how it’s done. When we call the fit() method on our model, we discover that our digestion of all the additional theory in this chapter paid off: With our intermediate­depth network, our validation accuracy plateaued around 97.6%, but our deep net attained 97.87% validation accuracy after the 15th epoch of training, shaving 11% of our already­small error rate away.

**TensorBoard 152**

When evaluating the performance of your model epoch over epoch, it can be tedious and time­consuming to read individual results numerically, as in Figure 9.7, particularly if the model has been training for many epochs. Instead, TensorBoard (Figure 9.8) is a convenient, graphical tool for: œ visually tracking model performance in real time, œ reviewing historical model performances, and œ comparing model performances.



**Figure 9.7** Our deep neural network architecture peaked at a 97.87% validation accuracy at the 15th epoch, besting the accuracy of our shallow and intermediatedepth architectures. Due to the randomness of network initialization and training, you may obtain a slightly lower or a slightly higher accuracy with the identical architecture.

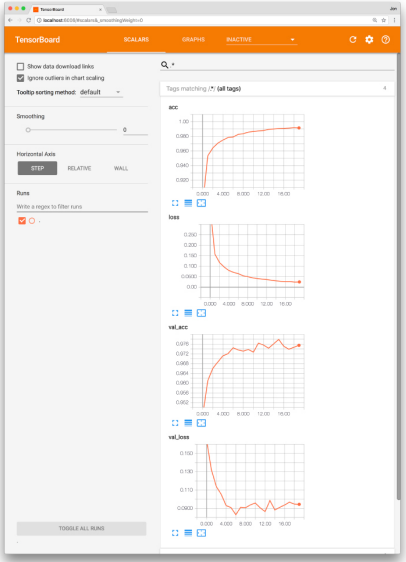


Figure 9.8 The TensorBoard dashboard enables you to, epoch over epoch, visually track your model’s cost (loss) and accuracy (acc) across both your training data and your validation val data.

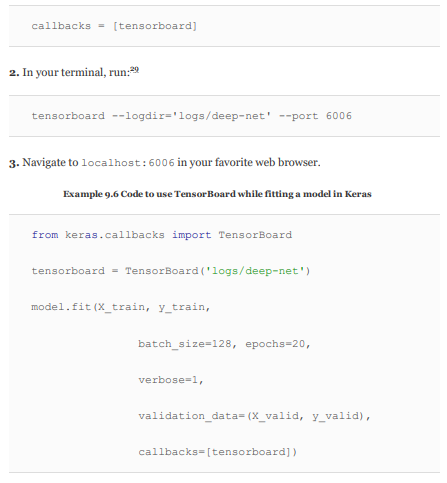
TensorBoard comes automatically with the TensorFlow library and instructions for getting it up and running are available via the TensorFlow site. It’s generally straightforward to set up though. Here, for example, is a procedure that adapts our Deep Net in Keras notebook for TensorBoard use on a Unix­based operating system, including Mac OS:

1. As shown in Example 9.6, change your Python code as follows:

a. Import the TensorBoard dependency from keras.callbacks

b. Instantiate a TensorBoard object (we’ll call it tensorboard) and specify a new, unique directory name (e.g., deep­net) that you’d like to create and have TensorBoard log data written into for this particular run of model­fitting: tensorboard = TensorBoard(log\_dir='logs/deep­net')

c. Pass the TensorBoard object as a callback parameter to the fit() method



By following the above steps or an analogous procedure for the circumstances of your particular operating system, you should see something like Figure 9.8 in your browser window. From there, you can visually track any given model’s cost and accuracy across both your training and validation data sets in real time as these metrics change epoch by epoch. This kind of performance tracking is one of the primary uses of TensorBoard, though the dashboard interface also provides heaps of other functionality, like visual breakdowns of your neural­network graph and the distribution of your model weights. You can learn about these additional features by reading the TensorBoard docs and exploring the interface on your own.