Chapter – 6

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

with **Nitish Srivastava** & **Kevin Swersky**

**Gradient Descent**

Lectures: Geoffrey Hinton

**Mini batch gradient descent** with some tricks

Momentum method

**Adaptive learning rates** for each connection

**Rmsprop**

**6.1 Overview of "Mini-Batch Gradient Descent"**

In this section, we're going to look at Stochastic Gradient Descent Learning (SGD) for a neural network, particularly the mini batch version, which is probably the most widely used learning algorithm for large neural networks.

* **Review: The error surface for a linear neuron**
* Let's review about the ***error surface*** of a ***linear neuron****:* The error surface means a surface that lies in a space where the horizontal axes correspond to the weights of the neural net and the vertical axis corresponds to the error it makes.
* For a ***linear neuron*** with a ***squared error***, that surface always forms a ***quadratic bowl***.
* The *vertical cross sections* are parabolas, and
* The *horizontal cross sections* are ellipses.

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

* For multi-layer non linear nets, the *error surface* is much more *complicated*, but as long as the ***weights*** aren't ***too big***, it's a ***smooth error surface***, and ***locally*** it's well approximated by a ***fraction*** of a ***quadratic bowl***.
* It might not be the ***bottom*** of the ***bowl*** but there's a *piece* of ***quadratic bowl*** that will ***fit*** the *local error surface* very well.
* **Convergence speed of full batch learning when the error surface is a quadratic bowl**

If we look at the conversion speed when we do *full-batch learning*, when the error surface is a ***quadratic bowl***, The obvious thing to do is ***go downhill*** to reduce the error.

|  |  |
| --- | --- |
| * Going downhill will reduces the error, but the direction of steepest descent does not point at the minimum unless the ellipse is a circle. * In the following figure, can see in the ***ellipse***, the ***direction*** of steepest descent is almost ***at right-angle*** to ***the direction we want to go*** in. * we've got a gradient that's very ***big across the ellipse***, which is the direction which we only want to travel a ***small distance***, and * the gradient's very ***small along the ellipse***, and that's the direction which we want to travel a ***large distance***.   It's precisely the ***wrong way around***. |  |

* You might think that studying *linear systems like this*, is not a good idea if you want to optimize ***big non-linear nets***. But even for these non-linear multi-line nets, this kind of a problem also arises. It's a very similar problem that arises even though the error surfaces aren't globally quadratic bowls.
* Locally they have all these same kind of properties, i.e. they tend to be very curved in some directions, and *very* ***uncurved*** *in other directions*.
* Even for ***non-linear multi-layer nets***, the error surface is ***locally quadratic***, so the same speed issues apply.
* **How the learning goes wrong**

|  |  |
| --- | --- |
| * The way the ***learning*** goes ***wrong*** if you use a ***big learning rate***, is that the weights ***slosh to and fro*** across the ravine (in the directions in which the area surface is very curved). * If the learning ***rate*** is ***too big***, this oscillation diverges. * What we want to achieve, is that * We ***move quickly*** along the ravine in directions that have ***small***, but *very* ***consistent*** gradients. * And we ***move slowly*** in directions with, ***big*** but very ***inconsistent*** *gradients*. That is if you go in that direction for a short distance, the gradient will ***reverse sign***. |  |

* **Stochastic gradient descent**

Before we go *into* how we *achieve* above *kind of movement*, let's talk a little bit about **Stochastic Gradient Descent**, and the motivation for using it.

* If the dataset is highly redundant, the **gradient** on the **first half** is almost **identical** to the gradient on the **second half**.

For a highly redundant dataset, if you compute the gradient for a weight on the ***first half*** of the data-set, you'll *get almost exactly* the *same answer* if you compute the *gradient* on the ***second half***.

* So instead of computing the full gradient, ***update*** the ***weights*** using the *gradient on the first half* and then *get* a *gradient* for the *new weights* on the *second half*.
* You'd be much better off computing the *gradient* on a ***subset*** *of the data*, then ***updating*** *the* ***weights*** and on the *remaining data*, computing the ***gradient*** for the ***updated weights***.
* The *extreme version* of this *approach* updates ***weights*** after ***each case***. It's called “online”.
* In that case, we're going to compute the ***gradient*** on a ***single*** *training case*, we're going to *update the weights* and then we're going to ***compute*** *the gradient* on the ***next*** *training case* using those *updated new weights*. That's called online learning.
* Mini-batches are usually better than online.

In general, we don't want to go for online-learning. It's usually better to use small mini batches, typically **10** or a **100** or even **1000** **examples**. Some of the benefits of using small-mini-batches are:

* ***Less computation is used updating the weights:*** One advantage of a small mini batch, is that less computation is used for actually updating the weights, because you do that less often, compared with online.
* Another advantage is that when you compute the gradient, you can compute the gradient for a whole ***bunch of cases in parallel***. Most computers are very good at doing **matrix-matrix multiplies**, and that will allow you to consider a ***whole bunch of training cases*** and apply the ***weights*** to a ***whole bunch of training cases*** at the ***same time*** to figure out the activities going into the next layer for all of those training cases.
* Thais **matrix-matrix multiplies**is very efficient, especially on a *graphics processor unit (GPU)*.
* Mini-batches need to be balanced for classes:
* One point about using mini batches is you wouldn't want to have *a mini batch* in which the *answer is always the same* and then on the *next mini batch* have a *different answer that's always the same*. That means we don't create mini-batches that contains elements from one class.
* That would cause the ***weights*** to ***slosh unnecessarily***. The ideal, if you have say ***10 classes***, would be to have a mini batch with say **10 examples** (each element from different class) or **100 examples** (multiples of 10), that has exactly the same number from each class in the mini batch.
* One way to *approximate that*, is simply to take ***all your data*** and just put it in ***random order*** and grab ***random mini-batches***. But you must avoid having *mini batches* that are *very uncharacteristic* of the *whole set of data* because the *mini-batches* are all of *one class*.
* **Two types of learning algorithm**

Basically there're two types of learning algorithms for neural nets.

* There are ***full gradient algorithms***, where you compute the gradient from *all of the training cases*. And once you've done that, there's a lot of clever ways to *speed up learning*.
* For example: there's thing like nonlinear versions of a method called non-linear conjugate gradient.
* The ***Optimization community*** has been studying the general problem of how you ***optimize smooth non-linear functions*** for many years.
* Now *Multi-Layer Neural Networks* are pretty ***untypical*** of the kinds of ***problems*** they study. So applying the ***methods they developed*** may need a lot of ***modification*** to make them work for these *multi-layer neural networks*.
* But when you have highly redundant and large training sets, for a Large Neural Network it's nearly always better to use Mini Batch Learning.
* The ***mini batches*** may need to be quite ***big*** when adapting ***fancy methods***.
* But that's ***not so bad*** because big mini batches are more ***computationally efficient***.
* **A basic mini-batch gradient descent algorithm**

Let's discuss a basic ***mini-batch grading descent learning algorithm***. This is what most people would use when they started training a big neural net on a ***big redundant data set***.

* *Guess an initial learning rate:* You start by guessing an ***initial learning rate***, and you look for if the network learned *satisfactorily* or if the *error* keeps *getting worse*, i.e. oscillates wildly. If it oscillates wildly, you ***reduce*** the ***learning rate***.
* *If the error keeps getting worse or oscillates wildly, reduce the learning rate:* You also look for if the ***error*** is ***falling too slowly***. You may expect that the *error* might *fluctuate a bit* if you measure it on a *validation set*, *because* the *gradient* on each mini-batch is just a ***rough estimate*** of the ***overall gradient***.
* So you *don't* want to *reduce* the *learning rate* every time the error rises.
* *If the error is falling fairly consistently but slowly, increase the learning rate:* What you're hope for is, that the ***error*** will ***fall*** ***fairly*** ***consistently***. And if it is *falling* fairly consistently and very slowly, you can probably increase the learning rate.
* Once you've got that working, you can then write a simple program to automate that way of adjusting the learning rate.
* *Towards the end of mini-batch learning, remove fluctuations in the final weights caused by the variations between mini-batches:* One thing that nearly always helps is, towards the end of learning with mini-batches, it helps to turn down the learning rate.
* That's because you're going to get fluctuations in the weights *caused by the* ***fluctuations*** *in the* ***gradients*** that come from the mini batches.
* And you'd like a final set of weights as a good compromise. So, when you turn down the learning rate, you're *smoothing away* those *fluctuations*, and getting a final set of weights that's good for many mini-batches.
* *Turn down the learning rate when the error stops decreasing:* A good time to turn down the learning rate is when the *error stops decreasing consistently*. And
* *Use the error on a separate validation set:* A good criterion for saying the error stopped decreasing is to use the ***error*** *on a* ***separate validation set***.
* i.e., it's a ***bunch of examples*** that you are ***not using*** for ***training*** and also they're not going to be ***used*** for your ***final test***.

**6.2 A bag of "Tricks" for mini batch gradient descent**

In this section, we are going to look at a number of issues that arise when using ***Stochastic Gradient Descent*** with ***mini-batches***. There are a large number of tricks that make things work much better. These are the kind of Black-Out of Neural Networks. We'll discuss some of the main tricks in this section.

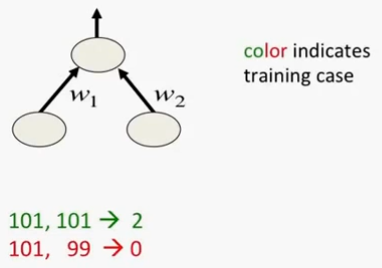
* **Initializing the weights**

The first issue is initializing the weights in your Neural Network. If ***two hidden units*** have exactly the *same* biases, and exactly the *same* **incoming** and **outgoing weights**, then they can never become different from one another. Because they always get exactly the same gradient.

* *So they can never learn to be different features:* To allow them to learn ***different feature detectors***, you need to start them off different from one another.
* We do this by using small random weights to initialize the weights. That breaks the symmetry.
* Those small ***random weights shouldn't*** all necessarily be the ***same size*** as each other. So if you've got a hidden unit that has a **very big** fan- in, if you use quite ***big weights*** it'll tend to ***saturated***, so you can afford to use *much smaller weights* for a hidden unit that has a *big fan-in*.
* If a ***hidden unit*** has a ***big fan-in***, **small changes** on many of its **incoming weights** can cause the learning to overshoot.
* If you have a hidden unit with a very small fan-in, then you want to use bigger weights. And since the weights are random, it scales with the **square root of the number of the weights**, **sqrt(fan-in)**.
* **Smaller incoming weights** when the **fan-in is big**, so initialize the weights to be proportional to **sqrt(fan-in)**.
* So a *good principle* is to make the ***size*** of the ***initial weights*** be proportional to the **square root of the fan-in**.
* We can also scale the learning rates for the weights the same way.
* What is the fan-in of a node? Fan-in is the *number of arrows coming into* a node. ***Fan-out*** is the number of ***arrows going out*** of a node.
* **Shifting the inputs**

One thing that has a surprisingly *big affect* on the *learning speed* of a neural network is shifting the inputs. That is ***adding*** a ***constant*** to each of the ***components*** of the ***inputs***.

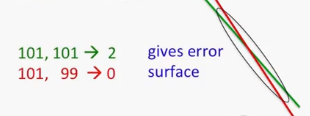
* But when you're using steepest decent, shifting an input value by adding a constant can make a *very big difference*. It usually helps to *shift each component* of the input, so that averaged over all of the training data, it has a value of zero. That is, make sure it's mean is zero.
* It usually helps to ***transform*** each ***component*** of the ***input vector*** so that it has ***zero mean*** over the ***whole training set***.
* So suppose we have a little neuron-like below, it's just a ***linear neuron*** with ***two weights***. Also suppose we have some training cases.



* The *first training case* is where the ***inputs*** are **101** and a **101**, you should give an **output** of **2**.
* The *second* one says when there are a **101** and **99** you should **output** a **0**.

We're using ***color*** to indicate specific training cases.

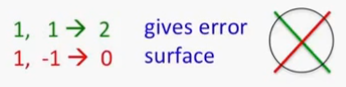
* If you look at the ***error surface*** for those two training cases, it looks like below.



* The green line is the line along which the *weights* will *satisfy the first training case*, and the red line is the line along which the weights will *satisfy the second training case*.
* Notice that, they're almost parallel. Therefore, when you combine them, you get a very elongated ellipse.
* One way to think about what's going on here is that, because we're using a squared error measure, we get a **Parabolic Trough** along the red line.
* The red line is the bottom of this **Parabolic Trough** that tells us the *squared error* we'll be getting on the *red case*.
* And there's another ***Parabolic Trough*** with the green line along its bottom.
* This may surprise your spatial intuition that: if you add together two parabolic troughs, you get a quadratic bowl. An elongated quadratic bowl, in this case.

So that's where the **error surface** came from.

* Now, if we subtract 100 from each of those two ***input components***, we'll get completely different **error-surface**. In this case, it's a circle, it's ideal.



* The green line is the line along which the **weights** add to **2**. (We're going to take the **first weight**, and multiply it by **1**. Also we take the **second weight** and multiply it by **1**, then we add them.)
* The red line is the line along which the two **weights** are equal. Because we're going to take the **first weight**, and multiply it by **one**. And we take the **second weight**, and multiply it by **-1**.
* So if the weights are equal, we'll be able to get that **0** that we need.

So the error surface in this case is a nice circle where gradient descent is really easy, and all we did was subtract **100** from every input.

|  |  |
| --- | --- |
| * If you're thinking about what happens ***not with*** the inputs but with the hidden units. It makes sense to have hidden units that are hyperbolic tangents that go between **-1** and **1**. The hyperbolic tangent is simply (2\*logistic -1). * The hypberbolic tangent (which is 2\*logistic -1) produces hidden activations that are roughly zero mean. * And the reason that makes sense is because then the ***activities*** of the ***hidden units*** are roughly ***mean zero*** and that should make the ***learning faster*** in the next layer. Of course, that's ***only true*** if the inputs to the *hyperbolic tangents* are distributed sensibly around zero. * But in that respect, a ***hyperbolic tangent*** is better than a ***logistic***. However there is ***other respects*** in which a ***logistic is better***. For example, *logistic* gives you a *rug to sweep things under*. It gives an ***output*** of ***zero***, and if you make the ***input*** even ***smaller*** *(negative)* than it was, the ***output*** is ***still zero***. So ***fluctuations*** in ***big negative*** inputs are ignored by the logistic. * For the *Hyperbolic Tangent* you have to go out to the ***end*** of its ***plateaus*** before it can ***ignore anything***. |  |

* **Scaling the inputs**

Another thing that makes a big difference on learning speed is scaling the inputs. When we use the steepest descent, *scaling the input* values is a very simple thing to do.

* It usually helps to transform each component of the input vector so that it has ***unit variance*** over the ***whole training set***.
* So it has a typical value of **1** or **-1**.

|  |  |
| --- | --- |
| * If we take a simple net with two weights and we look at the *error surface* when the *first component* is ***very small*** and the *second component* is ***much bigger****:* * We get an ***error surface*** in which we get an ***ellipse*** that's got * A very ***high curvature***, when the ***input******components big*** because *small changes* in the *weight* make a *big difference* in the ***output***. * And very ***low curvature*** in the direction in which the ***input component is small*** because *small changes* to the *weight* hardly make any difference to the ***error***. * The color here is indicating which axis we're using, not *which training example we're using*, as it did in the previous section. |  |
| * If we simply change the variance of the inputs, just **re-scale** them. Make the first component ***10 times as big*** and the second component ***10 times as small***, we now get a nice circular error surface. |  |

* **A more thorough method: Decorrelate the input components**

Shifting and scaling the inputs is a very simple thing to do, but ***Decorrelating the Input Components*** is bit more complicated. It's actually works even better because it's guaranteed to give you a ***circular error surface***.

* For a linear neuron, we try to *Decorrelate* the *Components* of the *Input Vectors*: if you take two components and look at how they're correlated with one another over the whole training set. Like, if you remember the early example how the number of ***portions of chips***, ***portions of ketchup*** might be highly ***correlated***.
* We want to get rid of those *correlations*. That will make learning much easier.
* ***Ways to De-Correlate:*** There are several different ways to *decorrelate inputs*. A reasonable method is to use ***Principal Components Analysis***.
* **Remove** the *(principal) components* that have the ***smallest eigenvalues***.
* This will already achieve some *dimensionality reduction*.
* And then scale the remaining components by ***dividing*** them by the ***square roots*** of their ***eigenvalues***.
* For a linear system, this will convert an *axis aligned elliptical error surface* into a *circular error surface*.
* Once you got a *circular error surface*, the *gradient* *points* straight towards the *minimum*, making learning really easy.
* **Common problems that occur in multilayer networks**

There are some ***common problems*** in *Multilayer Networks* that people encounter.

* A first problem is: If we start with a *too* biglearning rate, you drive the hidden units either to be firmly on, or firmly off.
* That is the incoming weights of each hidden unit will all become very ***big*** & ***positive*** or very ***big*** & ***negative***. And their state *no longer depends on the input* and of course that means that:
* The ***error derivative*** *is coming from* ***output*** won't affect them, because they are on the ***plateaus*** where the derivative is basically ***zero***. And so ***learning*** *will* ***stop***.
* The error derivatives for the hidden units will all become tiny and the error will not decrease.
* Because people are *expecting to see* local minimum, when learning stops they say, ***Oh, I'm at a local minimum and the error's terrible. i.e. these really bad local minimum***.
* Actually, this is usually a plateau, you got stuck out on the *end of a plateau*. But people often mistake it for a local minimum.
* A second problem that occurs, is, if you are classifying things (in classification networks) and you're using either a squared error or a cross entropy error. The *best guessing strategy* is normally to make each ***output unit*** always produce an output equal to the ***proportion of the time*** that it should be 1.
* The network *finds* this *strategy* ***quickly*** and so the error will fall quickly, but particularly if the ***network*** has ***many layers*** it may take a long time before it improves much on that by making use of the input.
* Because to improve over the ***guessing strategy*** it has to get sensible information from the input through all the hidden layers to the output and that *could take a* ***long time*** to learn if you start with *small weights*.
* So again, you ***learn quickly*** and then the ***error stops decreasing***, and it looks like another local minimum but actually it's another plateau.
* **Be careful about turning down the learning rate**

|  |  |
| --- | --- |
| We mentioned earlier that ***towards*** the ***end of learning***, you should *turn* ***down*** the learning rate. But be careful about turning down the learning rate too soon.   * When you *turn down the learning rate* you ***reduce*** the ***random fluctuations*** in the error due to the ***different gradients*** on *different mini*-*batches*. But of course you also reduce the ***rate of learning***. * Slower learning: So if you look at the ***red curve*** you see that when we ***turn*** the ***learning rate*** down we got a ***quick win***. The error fell, but after that we get slower learning. And if we do that ***too soon*** we're gonna ***loose*** relative to the ***green curve***. * So don't turn down the learning rate ***too soon***, or not ***too much***. |  |

* **Four ways to speed up mini-batch learning**

Now we discuss about four main ways to ***speed up mini-batch learning a lot***. The *previous* things we *discussed* about were kind of a ***bag of tricks*** for making *things work better*.

* Following are four methods all explicitly designed to make the ***learning*** go ***much faster***.

1. Use "momentum": ***Instead*** of using the gradient to ***change the position*** of the weight “particle”, use it to ***change the velocity***.

* In this method we don't use the ***gradient*** to change the ***position*** of the ***weights***. That is, if you think of the weights as a ***ball*** on the ***error*** ***surface***, standard gradient descent uses the ***gradient*** to change the ***position of that ball***.
* You simply ***multiply*** the gradient by a learning rate and change the *position* of the ball *by that* ***vector***.
* In the momentum method, we use the ***gradient*** to ***accelerate*** this ball. That is the gradient ***changes*** it's ***velocity***. And then the ***velocity*** is what changes the ***position*** of the ball.
* The reason that's different is because the ball can have ***momentum***. That is, it ***remembers previous gradients*** in its ***velocity***.

1. Use ***separate adaptive learning rates*** for ***each parameter***: Slowly adjust the rate using the consistency of the gradient for that parameter.

* A second method for speeding up mini-batch learning is to use a ***separate adaptive learning rate*** for ***each parameter***. And then to ***slowly adjust*** that ***learning rate*** based on ***empirical measurements***.
* The obvious *empirical measurement* is: Are we keeping making progress by changing the weights in the same direction? Or, Does the gradient keep oscillating around so that the sign of the grading keeps changing.
* If the sign of the grading keeps changing, what we're going to do is: reduce the learning rate and if it keeps staying the same, we're going to increase the learning rate.

1. rmsprop: Divide the *learning rate* for a *weight* by a ***running average of the magnitudes of recent gradients*** for that weight.

* If the ***gradients*** are ***big*** you divided by a ***large number*** and if the ***gradients*** is ***small*** and you divide then divide by ***small number***.
* That will deal very nicely with a wide range of different gradients. It's actually a ***mini batch version*** of just using the ***sign*** of the ***gradient*** which is a method called ***R-prop***, that was designed for *full batch learning*.

1. The final way of speeding up learning, is to use ***full batch learning***. And to use a fancy method ***from the optimization*** literature that takes ***curvature information*** into account.

* Adapt that method to work for ***neural nets***.
* Also try to adapt it so that it works with ***mini batches***. We'll discuss it later.

**6.3 The Momentum Method**

In this section we're going to look at the momentum method for ***improving the learning speed*** when doing ***Gradient Descent*** to ***Neural*** ***Network***.

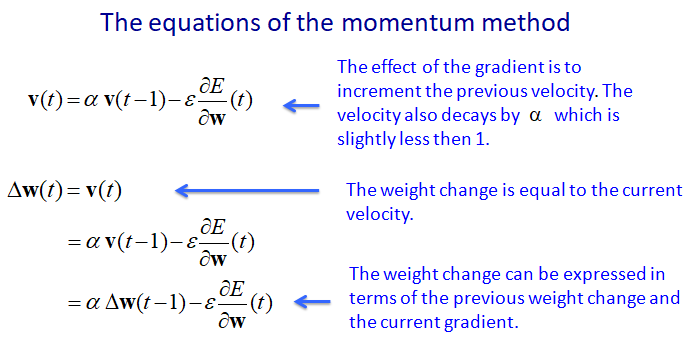
* The Momentum Method can be applied to Full Batch learning, but it also works for Mini Batch Learning.
* It's very widely used. And probably the commonest recipe for learning ***Big Neural Nets*** is to use: Stochastic Gradient Descent with mini batches combined with momentum.
* **The intuition behind the momentum method**

Let's first discuss the intuition behind the momentum method.

* Imagine a ball on the error surface, where the ***location*** *of the* ***ball*** *in the* ***horizontal plane*** represents the ***current weight vector***.
* The ball *starts off stationary* and so initially it will ***follow*** the direction of ***steepest descent***. It will *follow* the gradient.
* But as soon as it's *got some* ***velocity*** it'll no longer go in the ***same direction*** as the ***gradient***, it *no longer does steepest descent*.
* Its ***momentum*** will make it *keep going* in the ***previous*** *direction*.
* Obviously we wanted to get to a low point on the surface, so we *wanted* to lose energy. So we need to introduce a bit of viscosity. That is, we make its velocity die-off gently on *each update*.
* Momentum method damps oscillations in ***directions*** of ***high curvature*** by combining gradients with opposite signs
* So if you look at the red starting point, and then look at the green point we get to ***after two steps***, they have gradients that are pretty much ***equal*** and ***opposite***.

|  |  |
| --- | --- |
| * As a result, the ***gradient across the ravine*** has *cancelled* out. But the ***gradient along the ravine*** has *not cancelled* out. * *It builds up speed in directions with a gentle but consistent gradient:* ***Along the ravine***, we're going to keep ***building up speed***, and so, after the *momentum method* has *settled down*, it'll tend to go ***along*** the ***bottom of the ravine***, *accumulating velocity* as it goes, and if you're lucky, that'll make you ***go*** a whole *lot faster than* if you just do **steepest descent**. |  |

* **The equations of the momentum method**
* The effect of the gradient is to increment the previous velocity. The ***velocity*** also decays by which is slightly less then **1** (eg. 0.9).
* ***Time*** here is the ***updates*** of the ***weights***. So is the ***velocity vector*** that we got after ***mini batch*** **t-1**, attenuated a bit. So we multiply by some number like **0.9** i.e. .
* is actually kind of ***viscosity***, or it's related to viscosity. But unfortunately, we called it ***momentum***. So we now call ***alpha momentum***.
* Then we add in the ***effect*** of the ***current gradient***, which is to make us ***go downhill*** by some ***learning rate*** times the ***gradient*** that we have at ***time t***, and that'll be our ***new velocity*** at ***time t***.
* The weight change is equal to the current velocity. i.e. we make our ***weight change*** at time **t** equal to ***velocity***.
* The ***weight change*** can be expressed in terms of the ***previous weight change*** and the ***current gradient***.



* **The behavior of the momentum method**

The ***behavior*** of the momentum method is very intuitive.

* *If the error surface is a* ***tilted plane****, the ball reaches a* ***terminal velocity****:* On an error surface that's just a plane, the ball will reach some ***terminal velocity*** of which the gaining velocity that *comes from the* ***gradient*** is balanced by the *multiplicative attenuation of velocity* due to the ***momentum term***, which is really viscosity.

|  |  |
| --- | --- |
| * *If the momentum is* ***close to 1****, this is much* ***faster*** *than* ***simple gradient descent****:* If that momentum term is close to 1, then it'll be going down much faster than a *simple* ***gradient descent*** *method* *would*. So the terminal velocity, the velocity you get at time infinity is: ***gradient*** times the ***learning rate*** , multiplied by . So if is 0.99, you'll go 100 times as fast as you would with the *learning rate* alone. |  |

* Be careful in setting MOMENTUM: At the very beginning of learning, if you make the initial random weights quite big, there may be very large gradients. You'll have a *bunch of weights that's completely no good* for the task you're doing.
* You *don't want a big momentum*. Because you're going to quickly change them to make things better. And then you're going to start on the hard problem of: *finding out how to get just the* ***right relative values*** *of* ***different weights***. So you have *sensible feature detectors*.
* Since at the beginning of learning there may be very large gradients. So it *pays* at the *beginning of learning* to have a small momentum. It is probably better to have **0.5** than **zero**, because **0.5** will ***average out*** some ***sloshes*** and obvious ***ravines***.
* Once the ***large gradients have disappeared***, and you've reached the sort of *normal phase of learning*, where the weights are stuck in a ravine. And you need to *go along the bottom* of this *ravine* without ***sloshing to and fro*** sideways.
* You can ***smoothly raise*** the momentum to its ***final value*** (e.g. **0.9** or even **0.99**). Or you could raise it in one step, but ***that might start*** an ***oscillation***.
* You might think that, why didn't we just use a bigger learning rate? But what you'll discover is that, *using a* ***small learning rate*** *and a* ***big******momentum*** allows you to *get away with an* ***overall learning rate*** that's much bigger than you could have had if you used ***learning rate*** ***alone*** with ***no momentum***.
* If you use a ***big learning rate*** *by* ***itself***, you'll get big divergent oscillations (without the momentum) across the ravine.
* **A better type of momentum (Nesterov 1983)**

Very recently Ilya Sutskever has discovered that there's a better type of momentum.

* The standard momentum method first computes the ***gradient at the current location*** and then takes a big ***jump*** in the ***direction*** of the ***updated accumulated gradient***.
* The ***standard momentum method*** works by first computing the gradient at the current location. Then it combines that with its stored memory of ***previous gradients***, which is in the ***velocity*** of the ***ball***. And then it takes a big jump in the ***direction*** of the ***current gradient*** *combined with* ***previous gradients***. So that's its *accumulated gradient direction*.
* Ilya Sutskever (2012 unpublished) suggested a ***new form of momentum*** that often works better. It was inspired by the Nesterov method for *optimizing* ***convex functions***.
* First make a big jump in the direction of the previous accumulated gradient.
* Then we measure the gradient where we end up and make a correction.
* It's very similar, and you need a picture to really understand the difference. One way of thinking about what's going on is:
* In the ***Standard momentum method***, you add in the current gradient and then you gamble on this big jump.
* In the ***Nesterov method***, you use your previously accumulated gradient, you make the big jump and then you correct yourself at the place you've got to.

It's better to ***correct*** a ***mistake*** after you have ***made it***!

* **A picture of the Nesterov method**

|  |  |
| --- | --- |
| * First make a *big jump* in the *direction* of the previous accumulated gradient. * Then *measure the gradient* where you end up and make a correction. | Brown vector = jump,  Red vector = correction,  Green vector = accumulated gradient  Blue vectors = standard momentum |

* So here's the picture, when we first make the jump and then make a correction.
* ***Jump:*** Following is a step in the *direction* of the ***accumulated gradient***. So this depends on the *gradient* that we've *accumulated* on, in our *previous iteration*.



* ***Correction:*** We then measure the gradient, and go downhill in the direction of the gradient. Like below.



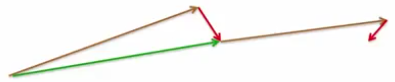
* ***Accumulated Gradient:*** We then combine that little correction step with the big jump we made to get our new accumulated gradient.



* Attenuate & next Jump: We then take that new accumulated gradient, we attenuate it by some number, like **0.9** or **0.99** and multiply it by that number, and we now take our ***next big jump*** in the ***direction*** of that *accumulated gradient*, like below:



* ***Correction:*** Then again, at the place where we end up, we measure the gradient and we go downhill. That correct any errors you made.



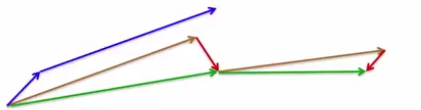
* ***Accumulated Gradient:*** And we our new accumulated gradient by combining correction & big jump.



* Standard Momentum Method: Now if you compare that with the standard momentum method, the *standard momentum method* starts with a accumulated gradient, like the *initial brown vector (big jump)*, but then it measures the gradient where it is,



* So it measures the gradient at its current location, and it *adds* that to the *brown vector*, so that it makes a jump like following big blue vector:



That is just the brown vector plus the current gradient. It turns out, if you're going to gamble, ***it's much better to 'gamble' and then make a 'correction'***, than to ***make a 'correction' and then 'gamble'***.

**6.4 Adaptive learning rates for each connection**

In this section, we're going to look at a method that was developed in the late 1980's by Robbie Jacobs and then improved by a number of other people.

* ***The idea is that:*** each connection in the neural net should have its own adaptive learning rate, which we set empirically by observing what happens to the weight on that connection when we update it.
* So that if the weight keeps ***reversing its gradient***, we turn down the learning rate. And if the gradient stays ***consistent***, we turn up the learning rate.
* **The intuition behind separate adaptive learning rates**

Let's first discuss why having separate adaptive learning rates on each connection is a good idea.

* ***The problem is that:*** In a deep multilayer net, the *appropriate* ***learning rates*** can ***vary*** widely between different ***weights***, especially between ***weights*** *in different* ***layers***.
* The magnitudes of the gradients are often *very different for different layers*, especially if the initial weights are small.
* Gradients can get very ***small*** in the ***early layers*** of very deep nets: For example, if we start with small weights, the gradients are start from much smaller in the initial layers than in the next layers.

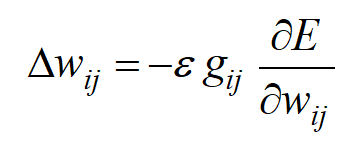
|  |  |
| --- | --- |
|  | * **Gradients** can get very **small** in the **early layers** of very deep nets. * The **fan-in** often **varies** widely between layers. |

* The fan-in often varies widely ***between layers***: Another factor that causes one different learning rate for different weights is the fan-in of the unit.
* The fan-in of a unit determines the size of the ***"overshoot" effects*** that you get when you ***simultaneously change*** *many* of the different ***incoming weights*** of a unit to *fix-up (correct)* the ***same error***.
* It maybe, that the unit *didn't get enough input*, when you ***change*** all these ***weights*** at the ***same time*** to *fix up the error*, it now gets too much input.
* Obviously, that effect is going to be bigger if there's a bigger fan-in.

So, the net in the diagram above has more or less the same fain-in for both layers, but that's very different in some nets.

* ***So use a global learning rate*** (which we set by hand) and then we're going to multiply it by an appropriate local gain that is determined ***empirically*** for ***each weight***.
* "**Empirically**" refers to ***knowledge*** or ***information*** gained through ***observation***, ***experimentation***, or ***experience*** rather than through ***theoretical reasoning*** ***alone***.
* **One way to determine the individual learning rates**
* *A simple way to determine what those local gains should be:* is to start with a local gain of **1** for every weight. So that, initially we're going to change the as below:

|  |  |
| --- | --- |
|  | = ***initial change*** of the weight,  = ***learning rate***  = the ***gain*** of 1  = ***error derivative*** for that weight . |



* Next, we're going to adapt .
* ***Increase*** *the* ***local gain*** *if the gradient for that weight* ***does not change sign****:* We're going to increase if the gradient for the weight does not change sign.
* We're going to use ***small additive increases*** and ***multiplicative decreases*** (for mini-batch):
* So, if the ***gradient for the weight at time t*** has the same sign as the ***gradient for the weight at time (t -1)***, with ***t*** *refers to* ***weight******updates***, then when we take that ***product***, it'll be ***positive***.
* Since there is either ***two negative gradients*** or ***two positive gradients***, and then we're going to increase by small *additive amount*.
* If the gradients have opposite signs, we're going to decrease . Since we want to ***damp down*** quickly if it's already big, we're going to ***decrease*** it ***multiplicatively***.
* That ensures that big gains will decay very rapidly if oscillations start.
* ***What would happen if the gradient was totally random?*** On each update of the weights, we pick a random gradient. Then, we'll get an ***equal number*** of ***increases*** and ***decreases*** because it will equally often be the same sign as the previous gradient or the opposite sign.
* We'll get a bunch of *additive 0.05 increases*, and *multiplicative 0.95 decreases*, and they have an equilibrium point which is when the gain is 1.
  + If the ***gain*** is ***bigger*** than 1, the multiplying by 0.95 will reduce it by more than adding 0.05.
  + If the ***gain*** is ***smaller*** than 1, adding 0.05 will increase it more than multiplying by 0.95 decreases it.

|  |  |
| --- | --- |
| * If the gradient is totally random, the gain will hover around 1 when we increase by **plus**  *(additive) half the time* and decrease by **times** *(multiplicative) half the time*. * So, with ***random gradients***, we'll hover around 1. * And if the gradient is ***consistently*** in the ***same direction*** we can get much bigger than 1. * If the ***gradient*** is consistently in ***opposite directions***, which means we're ***oscillating*** across a ***ravine***, we can get much smaller than 1. |  |

* **Tricks for making adaptive learning rates work better**

There's a number of tricks for making the adaptive learning rates work better.

* ***Limit the gains*** to lie in some reasonable range: It's important to limit the size of the gains. A reasonable range is: [0.1, 10] or [.01, 100].
* You ***don't*** want the ***gains*** to get ***huge***, because then you can easily get into an ***instability*** and they ***won't die down*** fast enough, and you'll *destroy all the weights*.
* Use ***full batch*** learning or ***big mini-batches***: The Adaptive Learning Rates was designed for ***full batch learning***. You can also apply it with ***mini batches*** but they better be pretty ***big mini batches***.
* This ensures that ***changes in the sign*** of the gradient are not mainly due to the ***sampling error*** *of a mini-batch*, they really are due to going to the other side of the ravine.
* ***Adaptive learning rates*** can be combined with ***momentum***: There's nothing to prevent you combining adaptive learning rates with momentum. So, Jacob suggests that, instead of using the *agreement in sign* between the ***current gradient*** and the ***previous gradient*** for a weight, you use the *agreement in sign* between the ***current gradient*** for a weight and the ***velocity for that weight***, so the *accumulated gradient*.
* And, if you do that, you get a nice combination of the ***advantages of momentum***, and the ***advantages of adaptive learning rates***.
* ***Adaptive learning rates*** only deal with ***axis-aligned effects***, whereas, ***momentum*** doesn't care about the ***alignment of the axis***. Momentum can deal with these ***diagonal ellipses*** and going in that ***diagonal direction quickly*** which *adaptive learning rates can't do*.

**6.5 Rmsprop: normalize the gradient**

* **RMSPROP:** ***Divide the gradient*** by a ***running average*** of its ***recent magnitude***
* We'll fist discuss a method called RPROP, that is used for ***full batch learning***. It's like Robbie Jacobs method, but not quite the same.
* Then we learn to extend RPROP so that it works for ***mini-batches***. This gives you the advantages of ***RPROP*** and it also gives you the advantage of ***mini-batch learning***, which is essential for *large*, *redundant data sets*.
* The method that we end up with called RMSPROP, is currently most used as a sort of *basic method* for ***learning the weights*** in a ***large*** ***neural network*** with a ***large redundant data set***.
* **rprop: Using only the sign of the gradient**

First we discuss RPROP which is an *interesting way of trying to deal with* the fact that gradients vary *widely* in their magnitudes.

* The magnitude of the gradient can be very ***different*** for ***different weights*** and can change during ***learning***.
* Some gradients can be tiny and others can be huge. And that makes it *hard to choose* a ***single global learning rate***.
* If we're doing ***full batch learning***, we can cope (deal) with this big variations in gradients, by only using the sign of the gradient.
* The ***weight updates*** are all of the ***same magnitude***, i.e. all of the weight updates will be the same size.
* This escapes from plateaus with tiny gradients quickly: For issues like *escaping from plateaus* with very small (tiny) gradients this is a great technique, because even with ***tiny gradients*** we'll take quite ***big steps***.

We couldn't achieve that by just turning up the learning rate because then the ***steps*** we took for ***weights*** that had ***big gradients*** would be much ***too big***.

* rprop: This combines the idea of only using the *sign of the gradient* with the idea of *adapting the step size separately* for *each weight*. So to decide ***how much to change*** your ***weight***, you don't look at the ***magnitude*** of the ***gradient***, you just look at the ***sign of the gradient***. But, you do look at the ***step size*** you decided around for that ***weight***. And, that *step size adopts over time*, again *without* *looking* at the *magnitude of the gradient*.
* ***Increase the step size for a weight multiplicatively if the signs of its last two gradients agree:*** So we ***increase*** the ***step size*** for a weight ***multiplicatively***, for example by factor 1.2 (e.g. times 1.2), if the signs of the last two gradients agree.
* This is like in Robbie Jacobs' *adaptive weights methods* except that we did, gonna do a multiplicative increase here.
* ***Otherwise decrease the step size multiplicatively (e.g. times 0.5):*** If the signs of the last two gradients disagree, we *decrease* the step *size* ***multiplicatively***, and in this case, we'll make that ***more powerful*** than the ***increase***, so that we can die down faster than we grow.
* ***Limit the step sizes to be less than 50 and more than a millionth (Mike Shuster’s advice):*** We need to limit the step sizes. Mike Shuster's advice was to limit them *between 50 and a millionth*. I think it depends a lot on what problem you're dealing with. If for example you have a problem with some tiny inputs, you might need very big weights on those inputs for them to have an effect.
* If you're not dealing with that kind of problem, having an upper limit on the weight changes that's much less than 50 would be a good idea.
* **Why rprop does not work with mini-batches**

So one question is: Why doesn't RPROP work with mini-batches? People have tried it, and find it hard to get it to work. You can get it to work with ***very big mini-batches***, where you use much more ***conservative changes*** to the ***step sizes***. But it's difficult.

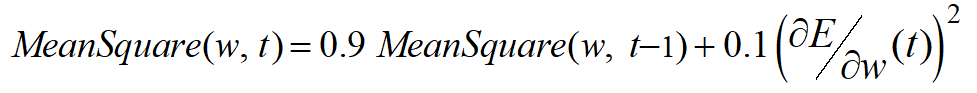
* So the reason it doesn't work is, ***it violates the central idea behind Stochastic Gradient Descent***.The idea behind stochastic gradient descent is that when the ***learning rate is small***, it ***averages the gradients*** over ***successive mini-batches***.
* So consider a weight that gets a gradient of **+0.1** on ***nine mini batches***, and then a gradient of **-0.9** on the ***tenth*** mini batch.
* What we'd like is: those *gradients will roughly average out* so that the ***weight*** *will stay roughly where it is*.
* ***RPROP won't give us that:*** RPROP would ***increment*** the ***weight*** *nine times* by whatever its *current* ***step size*** is, and ***decrement*** it only ***once*** by about the same amount (assuming any adaptation of the step sizes is small on this time-scale).
* And that would make the ***weight*** get much ***bigger***. So the weight would grow a lot.
* We're assuming here that the *step sizes adapt much slower* than the *time scale* of these mini batches.
* ***So the question is:*** can we combine the:
* robustness that you get from rprop by just using the sign of the gradient.
* The efficiency that you get from mini-batches.
* The ***effective averaging*** of *gradients over mini-batches* is what allows *mini-batches to combine gradients in the right way*.
* **rmsprop: A mini-batch version of rprop**

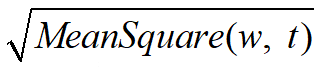
The problems we discussed above lead to a method called RMSPROP, it is a ***mini-batch*** *version of* ***RPROP***.

* RPROP is equivalent to ***using the gradient***, but also ***dividing*** by the ***magnitude*** of the gradient.
* And the reason RPROP has problems with mini-batches is that: we ***divide*** the ***gradient*** by a ***different magnitude*** for ***each mini batch***.
* **RMSPROP:** So the idea is that- we're going to force the ***number we divide by*** to be very similar for adjacent/nearby mini-batches. We do that by ***keeping*** a ***moving average*** of the ***squared gradient*** for ***each weight***.
* means the moving average for weight *w* at time *t*, where ***time*** is an indicator of ***weight updates***. ***Time*** ***increments*** by ***one*** each time we ***update*** the ***weights***.
* The numbers we used, i.e. 0.9 and 0.1 for ***computing moving average*** are just for examples, but they are reasonably sensible examples.
* is the ***value of the Squared Gradient for that Weight*** at time *t*

So the ***mean square*** at time *t* is the ***previous mean square*** times *0.9*, + the ***value of the Squared Gradient for that Weight*** at time *t*, times *0.1*.

* We then take the square root of that mean square (that's why the name RMS: Rooted MeanSquare).
* Then we ***divide*** the ***gradient*** by that RMS, and make an update proportional to that. That makes the learning work much better.
* Notice that we're *not adapting the learning rate separately* for *each connection* here. This is a *simpler method* where we simply, for ***each connection***, keep a ***running average*** of the root-mean-square gradient and divide by that.





* **Further developments of rmsprop**

There are many further developments one could make for RMSPROP.

* ***Combining RMSPROP with Standard Momentum:*** Experiment so far suggests that ***doesn't help as much as*** momentum normally does, and that needs more investigation.
* ***Combining RMSPROP with Nesterov Momentum (Sutskever 2012):*** You could combine our **RMSPROP** with NESTEROV MOMENTUM where you ***first make the jump*** and then make a ***correction***.
* ***Ilya Sutskever*** has tried that and got good results. He's discovered that it works best if the RMS of the recent gradients is used to divide the correction *rather than* the large jump in the *direction* of the ***accumulated corrections***.
* ***Combining RMSPROP with Adaptive Learning Rates for each connection:*** Obviously you could combine ***RMSPROP*** with ***Adaptive Learning Rates*** on ***each connection*** which would make it much more like RPROP. That just needs a lot more investigation.
* ***Other methods related to RMSPROP:*** There is a bunch of other methods related to ***RMSPROP*** that have a lot in common with it.
* *Yann LeCun’s group has a fancy version in “No more pesky learning rates”:* Yann ***LeCun's group*** has an interesting *paper* called ***No More Pesky Learning Rates*** some of the terms in that looked like RMSPROP, but it has many other terms.
* Most of the ***advantage*** that comes from this *complicated method* recommended by ***Yann LeCun's group*** comes from the fact that it's similar to ***RMSPROP***.
* **Summary of learning methods for neural networks**

So, a summary of the ***learning methods*** for ***neural networks***, goes like below:

1. When to use full batch? If you've got a small data-set, say 10,000 cases or less, or a bigger data set without much redundancy, you should consider using a full batch method.

* Redundancy in data: Unnecessary data. ***Data redundancy*** occurs when the ***same information*** is stored in ***multiple places*** in a dataset, either intentionally or unintentionally.
* This ***full batch methods*** adapted from the ***optimization literature*** like:
* Nonlinear Conjugate Gradient or LBFGS, or ***Levenberg Markhart (Marquardt)***. And one advantage of using those methods is *they typically come with a* ***package***. And when you report the results in your ***paper*** you just have to say, *"I used this package and here's what it did".* You ***don't have to justify*** all sorts of ***little decisions***.
* Alternatively you could use the Adaptive Learning Rates, or RPROP, which are both essentially ***full batch methods*** but these were ***developed*** for ***neural networks***.

1. For ***big, redundant datasets*** use ***mini-batches***: If you have a *big redundant data set* it's essential to use ***mini batches***. It's a huge waste not to do that.

* ***Try gradient descent with momentum:*** The first thing to try is just ***Standard Gradient Descent with Momentum***.
* You're going to have to ***choose*** a ***Global Learning Rate****:* You might want to write a little loop to adapt that global learning rate based on whether the gradient has changed sign.
* But to begin with, don't go for anything as fancy as ***Adapting Individual Learning Rates for Individual Weights***.
* ***Try rmsprop (with momentum ?):*** The next thing to try is **rmsprop**. That's very *simple to implement* if you do it ***without momentum***, and in different experiment so far, that seems to work as well as ***Gradient Descent with momentum***, would be better.
* You can also consider all sorts of ways of improving rmsprop by adding ***momentum*** or ***adaptive step sizes*** for *each weight*, but that's still basically *uncharted territory*.
* ***Try LeCun’s latest recipe:*** Finally, you could find out whatever ***Yann Lecun's latest receipe*** is and try that. He's probably the person who's tried the ***most different ways*** of getting Stochastic Gradient Descent to *work well*, and so it's worth keeping up with whatever he's doing.

1. ***Why there is no simple recipe:*** One question you might ask is, *Why is there no simple recipe?* We have been messing around with ***neural nets***, including ***deep neural nets***, for more than 25 years now, and you would think that we would come up with *an agreed way of doing the learning*. There's really ***two reasons*** why there isn't a simple recipe:

* ***Neural Nets differ a lot:***
* Very deep networks (especially ones with narrow bottlenecks in them), are very hard things to optimize and they need ***methods*** that can be ***very sensitive*** to ***very small gradients***.
* Recurrent nets: Recurrent nets are another special case, they're typically ***very hard*** to ***optimize***, if you want them *to notice things that happened a long time in the past* and ***change*** the ***weights*** *based on these things that happened a long time ago*.
* Wide shallow nets: Then there's wide shallow networks, which are quite different in flavor and are used a lot in practice. They often can be ***optimized*** with ***methods*** that are not very accurate, because we *stop the optimization* early before it starts overfitting.

So for these different kinds of networks, there's very different methods that are probably appropriate.

* ***Tasks differ a lot:*** Some tasks require very ***accurate weights***, some don’t.
* Some tasks also has weird properties, like if your ***inputs*** are words, rare words may only *occur* on *one case* in a *hundred thousand*. So it's a very different flavor from what happens if your ***inputs*** are pixels.
* So we really don't have nice clear-cut advice for how to ***train*** a ***neural net***. We have a bunch of rules of thumb, it's not entirely satisfactory, but just think how much better Neural Net will work once we've got this sorted out, and they already work pretty well.