Chapter – 3

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

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

**Introduction to**

**Weights & Back-propagation**

Lectures: Geoffrey Hinton

Learning the weights of a *linear neuron*

The error surface for a *linear neuron*

Learning weights of *logistic output neuron*

The backpropagation algorithm

Using the derivatives from *backpropagation*

**3.1 Learning the weights of a linear neuron**

* **Why the perceptron learning procedure cannot be generalised to hidden layers:**
* The perceptron convergence procedure works by ensuring that every time the *weights* change, they get closer to every “generously feasible” set of *weights* (get closer to a good set of weights).
* That type of *guarantee cannot be extended* to more complex networks. Because in more complex networks when you average two good set of weights, you might get a bad set of weights (the average of two good solutions may be a bad solution).
* So “Multi-Layer” Neural Networks *do not use* the perceptron learning procedure.
* To prove that when they're *learning* something is *improving*, we *don't use* the *same kind of proof at all*.
* They should *never have been called* multi-layer perceptrons.
* The ***learning algorithm*** for a linear neuron is quite like the learning algorithm for a perceptron, but it achieves something different.
* In a perceptron, the weights are always getting closer to a good set of weights.
* In a linear neuron, the outputs are always getting closer to the target outputs.
* **A different way to show that a learning procedure makes progress:**

For ***multilayer NN*** we're gonna need a *different way* to show that the *learning procedure makes progress*.

* Instead of showing the *weights get closer to a good set of weights*, show that the ***actual output values*** get closer the ***target*** ***output values***.
* This can be true even for non-convex problems in which there are many quite *different sets of weights* that work well and ***averaging*** two good sets of weights may give a ***bad set of weights***.
* *In non-convex problems averaging the weights of two good solutions does not give you a good solution.*
* It is not true for PERCEPTRON learning. In PERCEPTRON learning, the outputs as a whole can ***get further away*** from the target outputs even though the *weights are getting* *closer to good sets of weights.*
* The simplest example is a ***linear neuron*** *with a* ***squared error measure***: The ***simplest example of learning*** in which you're making the ***outputs*** get ***closer*** to the ***target outputs*** is learning in a linear neuron with a squared error measure.

|  |  |
| --- | --- |
| * **Linear Neurons (also called *Linear Filters* in electrical engineering):** * The *linear neuron* has a ***real-valued output*** which is a ***weighted sum of its inputs*** * Output **y** (the neuron's estimate of the target value), is the **sum** all weight vector times input vector. * The aim of learning is to minimize the error summed over all training cases. * The error is the squared difference between the target output and the actual output. |  |

* **Why don’t we solve it analytically?**
* Why don't we just solve it analytically? It's straightforward to write down a set of equations (one equation per training case), and to solve for the best set of weights.
* This is the standard engineering approach so why don’t we use it?
* Scientific answer: We want a method that real neurons could use.
* The first answer (and the *scientific answer*) is we'd like to understand what real neurons might be doing, and they're probably not solving a set of equations symbolically.
* Engineering answer: We want a method that can be generalized *to multi-layer, non-linear neural networks*.
* An *engineering answer* is that we want a method that we can then generalize ***to multilayer, nonlinear networks***.
* The *analytic solution relies on* it being ***linear*** and having a ***squared error measure***.
* An iterative method, which we're gonna see next, is usually less efficient, but much easier to generalize to more complex systems.
* **Example of an ITERATIVE method**
* Let's discuss an iterative method for finding the weights of a linear neuron. Suppose that, every day, you get lunch at a cafeteria. And your diet consists entirely of *fish*, *chips*, and *ketchup*.
* Each day, you order *several portions* of *each*, but on ***different days***, it's ***different numbers of portions***.
* The CASHIER only shows you the ***total price of the meal***, the *goal* is to *figure out* what the *price* is for *each portion* of *each kind* of thing.
* After several days, you should be able to figure out the price of each portion.
* Iterative Approach: In the iterative approach, you start with random guesses for the prices of portions.
* And then you adjust these guesses so that you get a better fit to the prices that the cashier tells you (observed prices of whole meals).
* Solving the equations iteratively: For each meal, you get a price and that gives you a linear constraint on the *prices of the individual* *portions*.
* Each meal price gives a linear constraint on the prices of the portions:



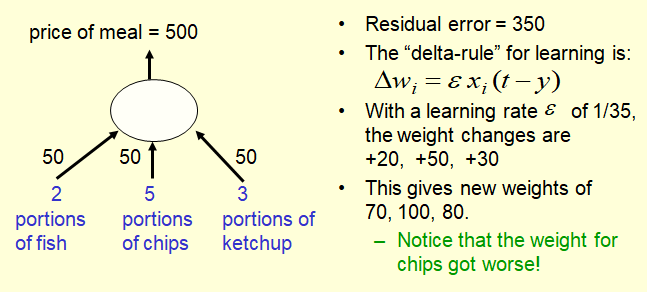
* The price of the whole meal is the sum of number of portion of fish times the cost of a portion of fish , for chips and ketchup .
* The prices of the portions are like the weights of a linear neuron.
* We can think of the whole weight vector as being the *price of a portion* of fish, the *price of a portion* of chips, and the *price of a portion* of ketchup.



* We're going to start with guesses for these prices (weights) and
* Then we're going to adjust the guesses slightly to give a better fit to the prices given by the cashier.

|  |  |
| --- | --- |
| * TARGET VALUE (The true weights used by the cashier): Let's suppose that the true weights that the cashier using to figure out the price, are 150 for a portion of *fish*, 50 for portion of *chips* and a 100 for a portion of *Ketchup*. i.e. the true weight vector is (150, 50, 100). * So that's going to be our target value. * For the meals shown here *(2 fishes, 5 chips, 3 ketchups)*, that will lead to a price of 850. |  |

* **A model of the cashier with arbitrary initial weights**
* A model of the cashier with Arbitrary Initial Weights: We start with guesses, let's say each portion costs ***50***.
* So for the meal with *(2 fishes, 5 chips, 3 ketchups)* we're going to initially think that the price should be **500**.
* But the real price was **800**. That gives us a residual error of **350**.
* Residual Error: The residual error is the difference between what the cashier says and what we think the price should be with our current weights (guessed weights).



* We're then gonna use the "DELTA rule" for **revising** our prices of portions.
* Delta Rule:

= change in a weight,

= learning rate,

= the number of portions of the i-th thing,

= Residual error, the difference between the target **t** and our estimate **y**.

* If we make the learning rate **1/35**, then in our case = () = 10.
* So, for this particular example it is **10**.
* And so, our change in the weight for fish will be . We'll increase that weight by **20**.
* Our change in the weight for chips will be .
* And our change in the weight for ketchup will be .
* That'll give us new weights of .
* Notice, the weight for chips actually got ***worse***.
* There's no guarantee with this kind of learning that the ***individual weights*** *will keep getting better*.
* What's getting better is the difference between what the cashier says and our estimate.
* **Deriving the delta rule**

Now, we're going to derive the delta rule. We start by defining the error measure, which is simply our squared residuals summed over all training cases.

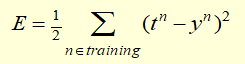
* i.e. the *squared difference* between the target and what the linear neuron predicts. We put a half in front, which will cancel the two, when we differentiate.
* We now differentiate that error measure E with respect to one of the weights,.
* The chain rule says that how the ***error changes as we change a weight***, will be ***how the output changes as we change the weight***, times ***how the error changes as we change the output***.
* is a partial derivative. There's many *different weights* you can *change* to *change the output*. And here, we're just considering the change to weight I ().
* , is actually equal to , because y is just , and

Note that, **n** indicates the **training**.

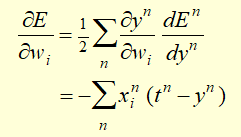
* Learning Rule: So our learning rule is now, we change the weights by an amount that's equal to the learning rate times the derivative of the error with respect to a weight, . We put a minus sign in front because we want the error to go down.

Thus the change in a weight is the *sum of all training cases* of the *learning rate* times the *input value* times the *difference between* the *target* and actual *outputs*. Following is a summary:

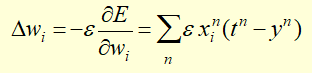
1. Define the error as the squared residuals summed over all training cases:



1. Now differentiate to get error derivatives for weights



1. The batch delta rule changes the weights in proportion to their error derivatives summed *over all training cases*



* **Behavior of the iterative learning procedure**

Now we can ask how does this learning procedure, this delta rule, behave?

* Does the learning procedure eventually get the right answer?
* There may be ***no perfect answer***.
* It may be that we ***gave*** the ***linear neuron*** a *bunch of* ***training cases*** with ***desired answers***. And there's no ***set of weights*** that'll give the ***desired answer***.
* But there's still some *set of weights* that gets the *best approximation* on ***all those training cases***, *minimizes* that *error measure* summed over all training cases.
* And if we make the learning rate ***small enough*** and we learn for ***long enough***, we can *get as close as we like* to that *best answer*.
* How quickly do we get towards the best answer?
* For a linear system, in this kind of ***iterative learning***, the learning can be quite ***slow***.
* If two input dimensions are highly correlated, its very hard to tell how much of the summed weight on both of those input dimensions should be attributed to each input dimension.
* For example, if you always get the same number of portions of *ketchup* and *chips*, we *can't decide* how much of the price is due to the ketchup and how much is used to the chips. It is hard to *decide* how to *divide the price* between ketchup and chips
* And if they're almost always the same, it can *take a long time* for the learning to *correctly attribute* the price to the ketchup and the chips.
* **The relationship between the Online Delta-Rule and the Learning Rule for PERCEPTRONS**
* There's an *interesting relationship* between the delta rule and the learning rule for perceptrons.
* If you use the *online version* of the *delta rule*, where we *change the weights* after each *training case*, it's quite similar to the *PERCEPTRON learning rule*.
* In PERCEPTRON learning, we *increment* or *decrement* the *weight vector* by the *input vector*, but we ***only change the input vector*** when we make an ***error***.
* In the online version of the delta rule, we *increment* or *decrement* the *weight vector* by the *input-vector*. But we scale that by both the residual error and the learning rate.
* And one annoying thing about this is we have to choose a learning rate.
* If we choose a learning rate that's too big, the system will be unstable.
* And if we choose a learning rate that's too small, it will take an unnecessarily long time to, to learn a sensible set of weights.

**3.2 The ERROR SURFACE for a linear neuron**

|  |  |
| --- | --- |
| * **The error surface in extended weight space:**   We can get a nice geometrical understanding of what's happening when we learn the weights of a linear neuron.   * Error surface: This space is very like the weight space that we use to understand PERCEPTRONS, but it has *one extra dimension*. * The error surface lies in the *space* where the *horizontal dimension* correspond to the *weights*. * There's one *vertical dimension* that *corresponds* to the *error*. * So in this space, *points* on the horizontal plane, *correspond to different settings of the weights*. And the *height* corresponds to the *error* that youre making with that ***set of weights***, *summed over all training cases*. |  |

* For a *linear neuron*, the *errors* you make for *each set of weights* define error surface. And this error surface is a Quadratic Bowl.
* i.e. if you take a ***vertical cross-section***, it's always a Parabola. If you take a ***horizontal cross-section***, it's always an Ellipse.
* This is only true for linear systems with a squared error. As soon as we go to a Multilayer Nonlinear Neural Nets, this error surface will get more complicated.
* As long as the weights aren't too big, the error surface will still be smooth, but it may have *many* local minimum.
* **Online versus batch learning**

Using this error surface we can get a picture of what's happening as we do Gradient Descent Learning using the delta rule.

* **What the delta rule does is:** It computes the *derivative* of the *error* with respect to the *weights*.
* If you *change* the weights in proportion to that derivative, that's equivalent to doing *steepest descent* on the *error surface*.
* *Elliptical Contour Lines:* To put it another way, if we ***look*** at the ***error surface from above***, we get *Elliptical Contour Lines*.

|  |  |
| --- | --- |
|  |  |
| * The simplest kind of ***batch learning*** does ***steepest*** ***descent*** on the error surface. | * The simplest kind of ***online learning*** zig-zags around the ***direction of steepest descent***. |

* Batch Learning (picture on the left): The delta rule is gonna take the weight points **(w1, w2)** at right angles (perpendicular) to those elliptical contour lines, as shown in the picture in the right. i.e. This travels perpendicular to the contour lines.
* That's what happens with what's called batch learning, where we get the gradients *summed over all training cases*.
* Online Learning (picture on the right): For online learning, ***after each training case***, we change the ***weights*** *in* ***proportion*** to the ***gradient*** for that ***single training case***.
* That's much more like what we do in PERCEPTRONS.
* You can see, the *change in the weights* moves us towards *one of these constraint planes*.
* So in the *picture* on the *right*, there are ***two training cases***.
* To get the first training case correct, the two weights **(w1, w2)** must lie on one of those blue lines. To get the second training case correct, the two weights must lie on the other blue line.
* So if we *start* at one of those ***red points***, and we *compute* the *gradient* on the *first training case*, the *delta rule* will *move us* *perpendicularly* towards one of those lines. If we then consider the other training case, we'll move perpendicularly towards the other line.
* Convergence: If we alternate between the two training cases, we'll zigzag backwards and forwards, *moving towards the* ***solution point*** which is where those two blue lines intersect.
* That's the set of weights that is correct for both training cases.
* **Why learning can be slow**

Now we're going to look at the error surface for a linear neuron. By understanding the shape of this Error Surface, we can understand a lot about what happens as a linear neuron is learning.

|  |  |
| --- | --- |
| * If that ***ellipse (Contour Line)*** is very elongated, which is gonna happen if the lines that correspond the two training cases is *almost parallel*, then when we look at the gradient, it's going to have a nasty property. * If you look at the red arrow in the picture, the gradient is big in the ***direction*** *in which we* ***don't want to move very far***, and it's small in the ***direction*** in which we ***want to move a long way***. * So the gradient will quickly take us ***across*** the ***bottom*** of that ***ravine***, corresponding to the *narrow axis* of the *ellipse*. * And it will take a long time to take us ***along*** the ***ravine***, corresponding to the *long axis* of the *ellipse*. * It's just the opposite of what we want. We'd like to get a ***gradient*** that's ***small across*** the ravine, and ***big along*** the ravine but that's not what we get. |  |

And so, simple steepest descent, in which you change each weight in proportion to a **learning\_rate error\_derivative**, is gonna have great difficulty, with very elongated surfaces like the one shown in the picture.

* If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum!
* The red gradient vector has a large component along the short axis of the ellipse and a small component along the long axis of the ellipse.
* This is just the opposite of what we want.

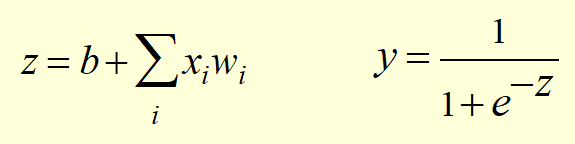
**3.3 Learning weights of logistic output neuron**

To extend the learning rule for a linear neuron to a learning rule for Multilayer Nets of Nonlinear Neurons, we need two steps.

* First, we need to ***extend*** the ***learning rule*** to a ***single nonlinear neuron***. We're going to use logistic neurons, although many other kinds of nonlinear neurons could be used instead.
* Then we generalize it to multi-layer NN.
* **Logistic neurons**

We're now going to generalize the learning rule for a linear neuron to a logistic neuron (which is a non linear neuron).

* These give a *real-valued output* that is a *smooth* and *bounded function* of their *total input*.
* They have *nice derivatives* which make *learning easy*.



* Logit: The ***logit*** is a ***transformation***. The logit transformation transforms a line to a logistic curve. A logit model is often called a *logistic regression model*.

|  |  |
| --- | --- |
| * A logistic neuron, computes its **logit z**, which is its *total input*, which is its *bias* plus the *sum over all its input lines times weights* * It then gives an **output y**, a *smooth nonlinear function* of that logit. |  |

* It is *approximately zero* when **z** is **big** and **negative**, *approximately one* when **z** is **big** and **positive**, and in between, it changes smoothly and nonlinearly.
* So it's easier to get nice derivatives, which make learning easy.
* **The derivatives of a logistic neuron**

To get the derivatives of a logistic neuron with respect to the weight (which is what we need for learning), we first need to compute the derivative of the logit **z** itself, i.e the total input with respect to our weight, and inputs. The derivatives of the logit, z, with respect to the inputs and the weights are very simple.

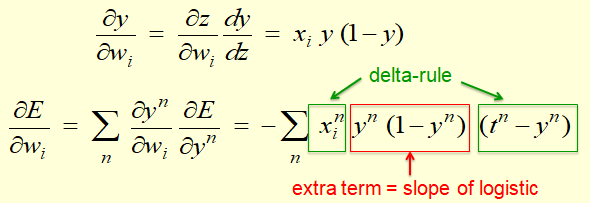
* The **logit z** is just a ***bias*** plus the *sum of all the input lines* of the *value on the input lines* ***times*** the ***weight***.

|  |  |  |
| --- | --- | --- |
| * When we differentiate **z** with respect to , we just get . |  |  |
| * Similarly, the derivative of the logit with respect to is . * The derivative of the output with respect to the **logit z** is also simple if you express it terms of the output. |  | |

* **Chain rule to get the derivatives for Logistic Neuron:**

Now we can use the ***chain rule*** to get the ***derivatives*** needed for *learning the weights* of a *logistic unit (Logistic Neuron)*. To learn the weights we need the derivative of the output with respect to each weight:

* Since, we've got the derivative, the output with respect to the logit and the derivative, the logit with respect to the weight, we can start to figure out the derivative, the output with respect to the weight.
* Then the learning rule for a logistic neuron will be:
* Notice how the *chain rule* is used.
* This derivative is much more like the *delta rule*.



* So, the way the *Error changes* if we change the weight (), is just the ***sum*** of all the row of training cases and of the *value on input line* times the residual, the *difference between the target and the output* , on the actual output of the neuron.
* Notice we've all the terms of the Delta Rule, but we have an extra term
* This extra term , comes from the slope of the logistic function.

So, a slight modification of the Delta Rule gives us the Gradiant Decent Learning Rule for training a logistic unit.

**3.4 The BACKPROPAGATION algorithm**

Learn multiple layers of features: Now we have the preliminaries out the way, we can *get back to the central issue*, which is how to learn *multiple layers of features*.

* Finally we're going to describe the Back-Propagation algorithm, which led to an explosion of interest in neural networks in the 1980s.
* Before we discuss back-propagation, we're going to describe another *very obvious algorithm* that *does not work nearly as well*, but is something that many people think of.
* **Learning with hidden units (again)**

Neural Networks without hidden units are very limited in the *input-output mappings* they can model.

* Now we know how to learn the weights of the logistic units, so we're going to return to the central issue, which is: *how to learn the weights of hidden units*.
* If you add a layer of hand coded features as in a PERCEPTRON, you make the net *much more powerful* but the *new difficulty* is designing the features.
* The learning won't solve the hard problem; you have to solve it by hand.
* What we'd like is a way of finding good features without requiring insights into the tasks or repeated trial and error, where we *guess* *some features and see how well they work*.
* *What we need to do is:* *automate* the *loop* of *designing features* for a particular task and *seeing how well they work*. We'd like the computer to do that loop, instead of having a person in that loop.
* **Learn by perturbing the weights (one weight at a time):**

In this case we randomly perturb one weight and see if it improves performance. If it improves performance of the net, you save that change in the weight (this idea occurs to everyone who knows about evolution).

|  |  |
| --- | --- |
| * Is it a kind of reinforcement learning?: You can think of this as a ***form*** of ***reinforcement learning***. In this case your ***action*** consists of making a ***small change***. * You check whether that *pays off*, and *if it does*, you decide to *perform* that *action*. * But it's very inefficient. Why? Because, just to decide *whether to change one weight*, we need to do ***multiple forward passes*** on a representative ***set*** of ***training cases***. * We have to see if *changing that weight* ***improves things***, and you can't judge that by one training case alone. |  |

* An additional problem with randomly changing weights is that towards the end of learning, any large change in weight will *nearly always make things worse*, because the *weights* have to have the *right relative values* to work *properly*.
* So *towards the end of learning* not only do you have to do a lot of work to *decide* whether each of these *changes helps* but the changes themselves have to be *very small*.
* Relative to this method of randomly changing weight, and seeing if it helps, BACK PROPAGATION is much more efficient.
* It's actually more efficient by a factor of the number of weights in the network, which could be millions.
* **Learning by using perturbations (all weights at a time):**

There are slightly better ways of using perturbations in order to learn. One way is to ***randomly*** perturb all the weights in parallel and then *correlate* the *performance gain* with the *weight changes*.

|  |  |
| --- | --- |
| * That actually ***doesn't really help at all***. The problem is that, we need to do lots and lots of trials with **different random perturbation** of all the weights, in *order to see the effect of changing one weight*, through the *noise created by changing all the other* *weights*. * We need lots of trials on each training case to *"see" the effect* of changing one weight through the noise created by *all the changes* to *other weights*. * So it doesn't help to do it all in parallel. |  |

* A better idea: *Randomly perturb* the *activities* of the *hidden units*.
* Something that does help, is to randomly perturb the activities of the hidden units, instead of perturbing the weight.
* Once we know ***how we want a hidden activity* to *change*** on a ***given training case***, we can *compute* how to *change* the *weights*.
* Once you've decided that *perturbing* the *activity* of a *hidden unit* on a particular training case is going to make things better. You can then compute how to change the weights.
* Since there's many fewer activities than weights, there's less things that you're randomly exploring.
* And this makes the algorithm more efficient.
* But it's still *much less efficient* than Back-Propagation.
* Backpropagation still wins by a *factor* of the *number of neurons*.
* **The idea behind backpropagation**

The idea behind back-propagation is that *we don't know* what the *hidden units* ought to be *doing*. They're called ***hidden units*** because nobody's telling us ***what*** their ***states*** ***ought*** ***to*** ***be***.

* But we can *compute* ***how fast the error changes*** as we ***change*** a ***hidden activity*** on a particular training case.
* So instead of using activities of the hidden units as our *desired states*, we use the *error derivatives* with respect to our *activities*.
* Instead of using *desired activities* to *train* the *hidden units*, use ***error derivatives*** w.r.t. ***hidden activities***.
* Since ***each hidden unit*** can affect *many different* ***output******units***, it can have *many different effects* on the overall error (many separate effects on the error) if we have many output units.
* These affects have to be combined. So that allows us to compute error derivatives for all of the hidden units *efficiently at the same time*.
* Once we've got those error derivatives for the hidden units (*hidden activities*), (i.e. we know *how fast* the error changes as we *changed* the *hidden activity* on that *particular training case*), it's easy to *convert* those *error derivatives for the activities* into *error derivatives for the weights* coming into a *hidden unit*.
* **Sketch of the backpropagation algorithm on a single case**

Let's see how backpropagation works, for a single training case.

* First convert the discrepancy between each output and its target value into an error derivative.
* First we have to *define* the error, and here we'll use the squared difference between the target values of the ***output unit j*** and the actual value that the ***net produces*** for the ***output unit j.***
* We also imagine that, there are ***several output units*** in this case.

|  |  |
| --- | --- |
| * After differentiation we get a familiar expression for ***how the error changes*** as you ***change the activity*** of an ***output unit j***. * We'll use a notation, where the ***index*** on a ***unit*** will tell you: which layer it's in. * Say the output layer has a typical index of **j**, and the layer in front of that, (the hidden layer below it in the diagram), will have a typical index of **i**.   The index will tell you which layer we're in. |  |

* Then compute error derivatives in each hidden layer from error derivatives in the layer above.
* So once we've got the Error-derivative w.r.t the output of one of these *output units*, we then want to use all those Error-derivatives in the output layer to *compute the same quantity* in the hidden layer that comes before the *output layer*.

|  |  |
| --- | --- |
| * Core of back propagation: It is taking error derivatives in one layer and from them *computing* the *error derivatives* in the *layer* that comes *before* that. * So we want to compute from . |  |

* Then use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights.
* Now obviously, when we change the output of ***unit i***, it'll change the activities of all three of those output units, and so *we have to sum* *up* all those *effects*.
* Algorithm: So we have the algorithm that takes error derivatives we've already computed for the top layer and *combines them* using the *same* ***weights*** as we use in the *forward pass* to get *error derivatives* in the *layer below*.
* **Backpropagating**

Let's explain the BACKPROPAGATION Algorithm. You may have to study it for a long time. This is how you backpropagate the ***error derivative w.r.t. the output of a unit***.

* Let's consider an output unit ***j*** on a hidden unit ***i***.
* The *output* of the hidden unit ***i*** will be .
* The *output* of the output unit ***j*** will be .
* And the total input received by the output unit ***j*** will be .
* First we need to convert the ***error derivative w.r.t.*** , into an ***error derivative w.r.t.*** . To do that we use the chain rule.
* We know for logistic units,
* Therefore we get:

So now we've got the ***error derivative w.r.t.*** (where is total input received by the output unit ***j***).

* Now we can compute the ***error derivative w.r.t.*** the output of hidden unit ***i***. It's going to be the sum over all of the three outgoing connections of hidden unit ***i***, of the quantity, .
* The first term is how (the total input to output unit **j**) changes as we change (the output of hidden unit **i**).
* And then we have to multiply that by (the second term) how the error changes as we change (the total input to output unit **j**) which we computed above.
* And as we saw before when studying the logistic unit, is just the weight on the connection .
* So that’s how, we get the error derivative.

w.r.t. the output of hidden unit **i**, is the *sum over* *all the* ***outgoing connections*** *to the* ***layer above*** *of the weigh*t on that connection times for the layer above (output unit **j**).

* Notice, the ***computation*** looks like the ***forward pass***, but we're going in the ***other direction***. For each ***unit i*** in that ***hidden layer***, we compute the sum of a (quantity in the layer above) (the weights on the connections).
* Once we've got (which we calculated above), it's very easy to get the error derivatives for all the weights coming into output-unit **j**. is simply times .
* We've already calculated .
* is *how* *changes as we change the weight on the connection*. is simply the activity of the unit in the layer below .
* So the rule for changing the weight is just multiply the quantity you've computed at a unit: , by the activity ***coming in*** from the layer below. And that gives you the ***error of derivative w.r.t. weight***.

So on this section we've seen how we can start with (in an input-unit) and back propagate to get (for an output-unit), we come back through one layer and computed the same quantity (the derivative of the error w.r.t. the output in the previous layer).

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

* We do that for as many layers as we like. And after we've done that for *all these layers*, we can *compute* how the *error changes* as you *change the weights* on the *connections*. That's the *BACKPROPAGATION algorithm*.

* It's an *algorithm* for taking *one training case*, and computing, efficiently, for *every weight* in the *network*, how the ***error will change*** on that particular training case, as you ***change the weight***.

**3.5 Using the derivatives from backpropagation**

Figuring out how to get the error derivatives for all of the weights in a ***multilayer network*** is the ***key*** to being able to learn ***efficient*** ***neural networks***. Before getting a specific learning procedure, there are some issues need to be addressed. For example:

* We need to decide how often to update the weights.
* Also we need to decide how to prevent the network from over-fitting very badly if we use a large network.
* **Converting error derivatives into a learning procedure**

The Back-Propagation algorithm is an *efficient way* to *compute* the error derivatives dE/dw for each weight on a single training case.

* But that's ***not a learning algorithm***. You have to *specify a number of other things* to get a *proper learning procedure*.
* We need to make lots of other *decisions* about how to use these error derivatives:
* Some of these decisions are about how we're going to optimize, that is "*how we're going to use the* ***error derivatives*** *on the* ***individual cases****, to discover* ***good set of weights***." (discussed in Chapter six).
* Another set of issues is "how do we ensure that the weights that we've learned will generalize well", i.e. how do we make sure they work well on cases (unknown test cases) that we didn't see during training (discussed in Chapter 7).

We now have a very brief overview of these two sets of issues.

* **Optimization issues in using the weight derivatives**

Let's go through a quick overview of these two sets of issues.

* Optimization Issues: Optimization issues are about how you use the weight derivatives.
* Online or Full-batch or mini-batch?: The first question is how often should you update the weights?
* Online-learning: We could try updating the weights ***after each training case***.
* We have to compute the *error derivatives* on *a training case* using *back propagation* and then, you make a *small change* to the *weights*.
* Obviously, this is going to *zigzag around* because on each training case, you'll get *different error derivatives*. But on average, if we make the weight changes small enough, it'll go in the right direction.
* Full batch learning: In this case we'll do a *full sweep* through all of the *training data*, *add together all of the error derivatives* *you get on the individual* *cases*, and then take a ***small step*** in that ***direction***.
* A problem with this is: if we start with a bad set of weights, and we might have a very big training set.
* We don't want to *do all that work of going through the whole training set* in order to fix up some ***weights*** that ***we know*** are ***pretty bad***.
* We only need to look at a few training cases before we get a reasonable idea of what direction we want to move the weights in.
* We don't need to look at a large number of training cases until we get towards the end of learning.
* Mini batch learning: Here we take a small random sample of the training cases and we go in that direction.
* We'll do a *little bit of zigzagging*, not nearly as much *zigzagging* as if we did online (one training case at a time).
* People typically do mini batch learning when they're training big neural networks on big data sets.
* How much we update (discussed in Chapter six): Then there's the issue of how much we update the weights. How big a change we make.
* Fixed learning rate: We can pick some fixed learning rate and then *learn the weights* by changing each weight by the derivative that we've computed times that learning rate ().
* Adapt global learning rate: We can adapt it by oscillating around, if the *error keeps going up and down*, then we'll reduce the learning rate. But if we're making steady progress, we might increase the learning rate.

|  |  |
| --- | --- |
| * Adapt separate learning rate on each connection: We can have a *separate learning rate for each connection* in the network, so that *some weights learn rapidly* and *other weights learned more slowly*. * Don’t use steepest decent: we don't really want to go in the direction of steepest decent at all. * If you look at the figure on the right, when we had a very elongated ellipse, the *direction* of *steepest decent* is almost at *right angles* to the ***direction to the minimum*** that we want to find. * And this is *typical* particularly towards the ***end of learning*** of most ***learning problems***.   So, there are *much better directions to go* in than the *direction* of *steepest decent*. The problem is, it's quite hard to figure out what they are. |  |

* **Overfitting: The downside of using powerful models**

The second set of issues is to do with: '*How* ***well*** *the network* ***generalizes*** *the* ***cases*** *it* ***didn't see during training*** '.

* *The problem here is that:* The ***training data*** contains ***information*** about the regularities ***in the mapping*** from input to output, but it also contains two types of noise.

1. The ***first type of noise*** is that the *target values* may be *unreliable*. And for *neural network*, that's usually only a *minor worry*.
2. The ***second type of noise*** is that the sampling error. There will be accidental regularities just because of the particular training cases that were chosen.

* If we take any particular training set, especially if it's a small one, there will be accidental regularities that are caused by the particular cases that we chose.
* For example, if you show *someone* some *polygons*, if you're a *bad teacher*, you might choose to show them a *square* and a *rectangle*. Those are *both polygons*, but there's *no way for someone* to realize from that, that *polygons* might have *three sides* or *seven sides*.

There's no way for them to understand that the *angles don't have to be right angles*.

If you're *a slightly better teacher*, you might show them a *triangle* and a *hexagon*. But again, from that, *they can't tell* whether *polygons* are always *convex*, and *they can't tell* whether the angles in polygons are always multiples of 60 degrees.

And, however carefully, you choose examples. ***For any finite set of examples, there'll be accidental regularities.***

* When we fit a model, there's no way it can tell the difference between an accidental regularity (for choosing particular samples) and a real regularity (that we'll generalize properly to new cases).
* So, what the model will do is: ***It will fit both kinds of regularity.***

And if you've got a big powerful model, it'll be very good at fitting the sampling error, and that will be a real disaster. That will cause it to generalize really badly.

* The Key point is: When we fit the model, it cannot tell which regularities are real and which are caused by sampling error. So it fits both kinds of regularity.
* If the model is very flexible it can model the sampling error really well. This is a disaster.
* **A simple example of overfitting:**

This is best understood by looking at following example. Here, we've got ***six data points*** shown in black, and we can fit a straight line to them.

* Notice the model has two degrees of freedom and it's fitting the ***six y-values***, given the ***six x values***, or we can fit a polynomial that has six degrees of freedom. By hand, we've drawn in red.
* The polynomial goes through the data points exactly and so it's a much better fit to the data.
* But which model do you trust?
* The complicated model certainly fits the data much better.
* But it's not economical.
* A convincing model is a simple model that explains (fits) a lot of data surprisingly well.
* And the polynomial doesn't do that. It explains these six data points, but it's got six degrees of freedom. So, wherever these data points were, it won't be able to explain them.
* It's not *surprising* that a *complicated model* can fit ***small amount data*** very well and it doesn't make this is a good model.

|  |  |
| --- | --- |
| * So, if you look at the arrow (in the figure), which output value do you predict for this input value? * Well, you'd have to have a lot of faith in the polynomial model in order to predict a value that's outside the range of values in all of the training data you've seen so far. * And I think almost everybody would prefer to predict the blue circle that's on the green line (straight line) rather than the one on the red line. * However, if we had ten times as much data, and *all of these data points lay very close to the* ***red line***, then we would certainly prefer the blue circle in the red line. |  |

* **Ways to reduce OVERFITTING**

There's a *number of ways to reduce over-fitting* that have been developed for neural networks and for many other models.

1. Weight-decay: There's weight decay where you try and keep the weights of the network small, or try and keep many of the weights at zero. And the idea of this is that it will make the model simpler.
2. Weight-sharing: In this method, you make the model simpler, by insisting that many of the weights have the exactly same value as each other.

* You don't know *what the value is,* and you're going to *learn it*. But it has to be *exactly the same* for *many of the weights*. We'll see that in the next chapter, how weight sharing is used.

1. Early stopping: There's *early stopping*, where you make yourself a *fake test set*. As you're *training* the *net*, you *peek* at what's happening on this *fake test set*. Once the performance on the fake test set starts getting worse, you stop training.
2. Model averaging: you train not so different neural nets, and you *average* them together in the hopes that, that will *reduce* the *errors* you're making.
3. Bayesian fitting of neural nets: Just a fancy form of *model averaging*.
4. Dropout: Make your model more robust by randomly emitting hidden units when you're training it.
5. Generative pre-training: these are somewhat more complicated and we'll try to describe towards the end of the course.