Chapter – 9

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

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

**Regularization**

Lectures: Geoffrey Hinton

Overview of ways to *improve generalization*

*Limiting the size* of the *weights*

Using noise as a regularizer

Introduction to the Full Bayesian Approach

The Bayesian interpretation of weight decay

*MacKay's quick and dirty method*

**9.1 Overview of ways to improve generalization**

In this section, we're going to discuss about *improving generalization* by reducing the *overfitting* that occurs when a *network has too* *much capacity* for the amount of data it's given during training.

* We'll discuss various ways of controlling the *capacity* of a network.
* And we'll also discuss how we determine & how to set the *meta parameters* when we use a method for *controlling capacity*. I'll then go on to give an example where we control capacity by *stopping the learning early*.
* **Reminder: Overfitting**
* ***The reason we get over-fitting:*** The training data contains *information about the true regularities* in the *mapping from input to output*. But any finite set of training data also contains *sampling error*.
* There will be *accidental regularities* in the training set, just because of the particular training cases were chosen.
* So when we fit the model to the training set, it can't tell which of the *regularities are real* (it'll exist if we sampled the training set again), and which are caused by the *sampling error*.
* Hence, the model fits both kinds of regularity. And if the model is very flexible, it'll *fit the* ***sampling error*** *really well*, and then it'll ***generalize badly***.
* i.e. if you fitted the model to another training set drawn from the same distribution over cases, it would make different predictions on the test data. This is called “variance”.
* **Use following approaches to prevent OVERFITTING**
* ***Approach 1: Get more data!***
* This method is by far the best. And it's simply to get more data, if *data is cheap* and you have *enough compute power to train* on more data.
* Data has exactly the right characteristics to prevent over fitting. The more of it you have the better.
* ***Approach 2: Use a model that has the right capacity***
* We try to judiciously *limit the capacity of the model* so that it's got *enough capacity* to fit the *true regularities* but *not enough capacity* to fit the *spurious regularities* (if they are weaker) caused by the *sampling error*.
* It's not so easy; there are various approaches to regulate the capacity appropriately.
* ***Approach 3: Average many different models***
* *Use models with different forms:* Another way is to *averaging* together *many different models*. If we average models that have *different forms* and make *different mistakes*, the *average* will do better than the individual models.
* *Train the model on different subsets of the training data:* We could make the models different just by *training* them on *different subsets* of the *training data*. This is a technique called bagging.
* There's also other ways to mess with the training data to make the models as different as possible.
* ***Approach 4: Average the predictions from different weights (Bayesian)***

Use *single neural network architecture*, but average the predictions made by many different weight vectors.

* It is a Bayesian approach; we'll use *single neural network architecture*, but to find many *different sets of weights* that do a good job of predicting the output.
* And then on *test data*, you *average the predictions* made by all those different *weight vectors*.
* **Some ways to limit the capacity of a neural net**

There's many ways to control the capacity of a model.

* ***Architecture:*** Control the capacity of a model by modifying the architecture. We can *limit* the number of *hidden layers*, and the *number of* *units (neurons)* per layer.
* And those control the *number of connections* (i.e. the number of parameters) in the network.
* ***Early stopping:*** Start with *small weights* and *stop* the learning before it *overfits*.
* Assuming that it finds the *true regularities* before it finds the *spurious regularities* in that particular training set we have.
* ***Weight-decay:*** Penalize large weights using penalties or constraints on their squared values or absolute values.
* A very common way to control the capacity of a neural network is to give it a *number of hidden layers or units per layer is a little too large*, but then to *penalize* the *weights* using penalties or constraints using *squared values* (L2 penalty) or *absolute values* (L1 penalty) of the *weights*.
* ***Noise:*** we can control the capacity of a model by adding *noise to the* ***weights***, or by adding *noise to the* ***activities***.

Typically, we use a combination of several of these different capacity control methods.

* **How to choose meta parameters that control capacity**
* Hyperparameters: Hyperparameters *can't be learned using the algorithm* that needs them, but they must be *tuned* before the training (manually or automatically). Hyperparameters are also named as Meta-parameters, Free-parameters or Tuning-parameters.
* To control the model capacity, some Hyperparameters or Meta Parameters for example are the *number of hidden units* or the *size of the weight penalty*.
* For most of the above methods (to limit the capacity of a neural net), there are some *meta-parameters* that we have to set. Like the number of hidden units, or the number of layers, or the size of the weight penalty.
* An obvious **wrong-way** to set those meta-parameters is: to *try lots of different values of one of the meta parameters*, for example, the number of hidden units, and see which gives the *best performance* on the *test set*.
* This is easy to do, but there's something ***deeply wrong*** with that.
* It gives a ***false impression*** of how well the method will work if you give it another test set.
* So the settings that work best for *one particular test set* are ***unlikely to work*** as well on a *new test set* that's drawn from the *same* *distribution* because those *meta-parameters* been *tuned* to that *particular test set*. And that means you get a *false impression* of how well you would do on a *new test set*.
* ***An extreme example:*** Suppose the test set has random answers that *do not depend* on the input or *can't be predicted* from the inputs. (quite a lot of financial data seems to be like that).
* If you choose the model that does best on test set, that will obviously do better than chance on the test set.
* But if we take that model and try it on *new data that's also random*, we can't expect it to do better than chance on that *new test set*.

So by selecting a model, we got a false impression of *how well a model will do on new data* and the question is: *Is there a way around that?*

* **Cross-validation: A better way to choose meta parameters**
* ***Cross validation:*** Cross validation is a better way to *choose* the *meta-parameters*. You start by dividing the total data set into *three subsets*.
* **Training Data:** It is used to train our model, i.e. the *training data* is used for *learning the parameters* of the model.
* **Validation Data:** Validation data is *not* used for *learning (training)* but is used for ***deciding*** what *settings* of the *meta parameters* work best.
* By observing how well the model does on the *validation data*, we decide the *appropriate number* of *hidden units* or an *appropriate size* of *weight penalty*.
* **Test Data:** Test data is used to get a *final, unbiased estimate* of how well the network works. We expect this *estimate* to be *worse than* on the *validation data*.
* After we get our trained model *(with what looks like the best number of hidden units and the best weight penalty)*, we're then going to use the ***test data*** to see how well our ***trained-model*** does.
* We must only *use* that *data once*. And that'll give us an unbiased estimate of how well the network works.
* In general that *estimate* will be a *little worse* than on the *validation data*.

Nowadays in competitions, the people organizing the competitions have learned to hold back *that true test data* and get people to send in predictions so they can see whether they *really can predict* on *true test data*, or whether they're just *over-fitting* to the *validation data* by selecting *meta-parameters* that do particularly *well on the* ***validation data*** but *won't generalize to* ***new test sets***.

* ***N-fold cross-validation:*** One way we can get a better *estimate* of our *weight penalties* or number of *hidden units* or anything else we're trying to fix using the *validation data*, is to ***rotate*** the ***validation set***.
* So, we still use a final *test set* to get our *final unbiased estimate*. But we divide the *other data* into *N equal sized subsets* and we train on all but one (i.e. train on all N-1 subsets) of those N subsets, and use the Nth-subset as a *validation set*.
* The point is: we can *rotate* and a hold back a *different subset* as a *validation set*, and so we can get *many different estimates* of what the *best weight penalty* is, or the *best number of hidden units* is.
* By rotating the validation set, we get *N different estimates* of the *validation error rate*.
* This is called ***N-fold cross-validation***.
* It's important to remember, the N different estimates we get are not independent of one another.

If for example, we were really unlucky and *all the examples of one class* fell into one of those *subsets*, we'd expect to generalize very badly.

* Whether that subset was the *validation subset* or whether it was in the *training data*.
* **Preventing overfitting by early stopping**

If we have ***lots of data*** and a ***big model***, it's very expensive to keep *re-training* it with different sized penalties on the *weights* or different *architectures*.

Early stopping is one particularly easy to use method for *preventing overfitting*.

* It's good when you have a *big model* on a *small computer* and you don't have the time to train a model many different times with different numbers of *hidden units* or different size *weight penalties*.
* What you do is:
* You start with *small weights*, and as the model *trains*, they *grow*.
* You watch the ***performance*** on the ***validation set***. And as soon as it starts getting worse, you *stop training*.
* Now, the *performance* on the *validation set* may fluctuate, particularly if you're *measuring an* ***error rate*** rather than a ***squared error*** or ***cross-entropy error***.
* *It can be hard to decide when performance is getting worse:* So it's hard to decide when to stop and so what you typically do is: keep going until you're sure *things are getting worse* and then go *back* *to the point* at which things *were best*.
* *The capacity of the model is limited because the weights have not had time to grow big:* The reason this controls the capacity of the model, is because *models with small weights* generally don't have as *much capacity*, and the *weights* haven't had time to *grow* big.
* **Why early stopping works**

It's interesting to ask why smaller weights lower the capacity of the network?

|  |  |
| --- | --- |
| * Let's consider a model with some *input units*, some *hidden units*, and some *output units*. * When the weights are very small, if the hidden unit is a logistic units, their total inputs will be close to zero, and they'll be in the middle of their linear range. * With very small weights every hidden unit is in its linear range, i.e. they'll behave very like linear units. |  |

* What that means is, when the weights are small, the whole network is the same as a linear network that maps the inputs straight to the outputs.
* *So even with a large layer of hidden units it’s a linear model:* So, if you multiply that weight matrix **W1** by that weight matrix **W2**, you'll get a *weight matrix* that you can use to *connect* the *inputs* to the *outputs*
* And provided the *weights are small*, a net with a *layer of logistic hidden units* will behave pretty much the same as that *linear net*.
* Provided we also *divide the weights* in the *linear net* by *four*, to take into account the fact that when the *hidden units* there, in that ***linear*** ***region***, and they have a *slope of a quarter*.
* So it's got *no more capacity* than the *linear net*, so even though there's weights, it's really got no more capacity than a network with weights (in which the inputs are directly connected to the outputs).
* As the weights grow, we start using the *non-linear region* of the sequence (the *hidden units* start using their *non-linear ranges* so the capacity grows). And then we start making use of all those parameters.
* So if the network has *6 weights* at the *beginning* of learning and has *30 weights* at the *end* of learning, then we could think of the *capacity* as *changing smoothly* from *6 perimeters* to *30 perimeters* as the *weights get bigger*.

What's happening in ***early stopping*** is: we're *stopping the learning* when it has the *right number of parameters* to do as well as possible on the *validation data*. That is when it's *optimized* the *trade-off* between *fitting* the *true regularities* in the data and *fitting* the *spurious* *regularities* that are just there because of the particular training examples we chose.

**9.2 Limiting the size of the weights**

In this section, we'll discuss about how we can *control capacity of a neural net* by limiting the ***size*** of the ***weights***.

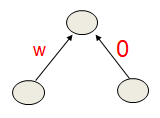
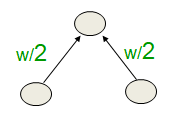
* The standard way to do this is to introduce a **penalty** that prevents the weights from getting **too large**. With the implicit assumption that: *a network with small weights* is somehow *simpler* than a *network with large weights*.
* There are several different *penalty terms* we can use.
* We can also *constrain the weights*, so that the *incoming weight vector* to each of the *hidden units* is *not allowed* to be *longer* than a *certain length*.
* **Limiting the size of the weights**
* The standard way to *limit* the *size of the weights*, is to use an ***L2 weight penalty***, which means that we ***penalize*** the *squared value of the weights*.
* Sometimes it's called ***weight decay*** in the neural network literature, because the ***derivative*** *of that* ***penalty*** acts like a force pulling the weight towards zero.
* This *weight penalty* keeps the *weight small*, unless they have *big error-derivatives* to *counter-act* it.
* The *standard L2 weight penalty* involves adding an *extra term* to the *cost function* that *penalizes* the *squared weights*.
* So if you look at what the ***penalty term*** looks like, as the *weight* *moves away from zero*, you get following ***parabolic cost***.

|  |  |
| --- | --- |
| * If you look at the equation, the ***cost*** that you're *optimizing* is the ***normal error*** that you're trying to *reduce*, plus a term which is the *sum of the* ***squares of the weights***, with a coefficient in front, ***Lambda***, and we *divide by 2* so that when we *differentiate* the *2 is cancelled*. * That ***coefficient*** in front of the *summed* ***squared weights*** is sometimes called the ***weight cost***. That determines how *strong* the *penalty* is. |  |
| * If you differentiate, you can see the *derivative* of this *cost* is just the *derivative of the error* plus something that's to do with the *size of the weight and the value of Lambda*. |  |

* That *derivative* will be zero (i.e. ) when the *magnitude of the weight* is:
* *This keeps the weights small unless they have big error derivatives:* So the only way you can have *big weights*, when you're at a *minimum of the cost function*, is if they also have *big error derivatives*. And this makes the *weights* much *easier to interpret*. You don't have a whole *lot of weights* that are *large* but *aren't doing anything*.
* **The effect of L2 weight cost**

The effect of an ***L2 penalty*** on the *weights* is to *prevent* the network from using *weights that it doesn't need*. This often improves generalization a lot, because it can use those weights that it doesn't really need to fit the sampling error.

* So it helps to stop the network from fitting the sampling error.
* It also makes a smoother model in which the output changes more slowly as the input changes.

* So if the network has *two* very *similar inputs*, when you put in an L2 weight penalty, it prefers to *put* ***half*** *the* ***weight*** on each of those two *similar inputs* rather than all of the weight on one, as shown on the right here.
* If the two inputs are very similar, those two networks will produce very similar outputs.
* But the one with the *halved-weights* will have *much less extreme changes* in its outputs when you change the inputs.
* **Other kinds of (L1) weight penalty**

There are other kinds of weight penalty. For example, an L1 penalty, where the *cost function* is just like V ***shape***.

|  |  |
| --- | --- |
| * ***V-shaped cost-function:*** In the *V-shaped cost-function*, we're penalizing the *absolute values* of the weights. Sometimes it works better to *penalize the* ***absolute values*** of the ***weights***. * This has the nice effect that it drives *many of the weights* to be exactly *zero* and that helps a lot in interpretation. If there's only a *few non-zero weights* left, it's much *easier to understand* what's going on. |  |
| * ***More extreme weight penalties:*** We can also use weight penalties that are *more extreme than L1* where the *gradient* of the *cost function* actually gets *smaller* when the *weight gets really big*. That kind of weight penalty has ***negligible effect*** on ***large weights***. * This allows the network to *keep large weights* without them being *pulled towards zero*. * It's just the *small weights* that'll get *pulled towards zero*. That's how it'll help us getting a ***few large weights***. |  |

* **Weight penalties vs weight constraints**

Instead of putting penalties on the weights, we could actually use weight constraints. That is instead of *penalizing the squared value of each weight separately*, we put a *constraint* on the *maximum squared length of the incoming weight vector* of each *hidden unit* or *output* *unit*.

* When we update the weights, if the ***length*** of that incoming vector gets longer than allowed by the constraint, we simply *scale the vector* *down* by ***dividing*** all the weights by the same amount until its length *fits the allowed length*.
* We usually **penalize** the squared value of each weight separately.
* Instead, we can put a **constraint** on the maximum squared length of the incoming weight vector of each unit.
* If an update violates this constraint, we scale down the vector of incoming weights to the allowed length.
* Using *weight constraints* has a number of advantages over *weight penalties*.
* *Its easier to set a sensible value:* It's much easier to select the *sensible value* for the *squared length* of the *incoming weight vector* than it is to select the *weight penalty*.
* That's because, logistic units have, a *natural scale* to them so we know what a "weight of 1" means.
* *They prevent hidden units getting stuck near zero:* Using weight constraints also *prevents* hidden units *getting stuck near zero* with all their *weights* being *tiny* and *not doing anything useful*.
* Because when all their *weights are tiny*, there's *no constraint* on the weights. So there's nothing preventing them from *growing*.
* *They prevent weights exploding:* Weight constraints also prevent the weight from *exploding*.
* One of the subtle things that weight constraints do is that: when a unit *hits* its *constraint* (hits it’s limit), the *effective weight penalty* on all of its *weights* is determined by the *big gradients*.
* So if some of the *incoming weights* have very *big gradients*, they'll be trying to ***push*** the length of the incoming weight vector ***up***. And that will ***push down*** on all the other weights.
* In effect, if you *think* of it *like a penalty*, the *penalty scales itself* so as to be *appropriate* for the *big weights* and to *suppress* the *small* *weights*.
* This is more effective than a fixed penalty at pushing irrelevant weights towards zero.
* ***Using Lagrange multipliers:*** For those of you who know about *Lagrange multipliers*, the *penalties* are simply the *Lagrange* *multipliers* required to keep the *constraints satisfied*.
* Lagrange multipliers can be used to *handle weight constraints*. When you want to enforce certain constraints on the weights (like keeping their *norm* below a *specific threshold*), you can formulate this as an *optimization problem*.
* The Lagrange multipliers would then *represent the penalties* applied to the objective function to ensure that the *constraints* are *satisfied* during training.
* This approach helps balance the *trade-off* between *minimizing the loss* and *adhering to the weight constraints*.

**9.3 Using noise as a regularizer**

In this section we'll discuss *another way of restricting* the *capacity* of a *neural network*. The idea is ***"adding noise"***, either to the *weights* or to the *activities*.

* Firstly, if we add noise to the inputs in a *simple linear network*, that's trying to minimize the ***squared error***, that's exactly equivalent to imposing an ***L2 penalty*** on the weights of the network.
* Next we'll discuss the uses of noisy weights in more complicated networks.
* Lastly we'll see a recent discovery that extreme noise in the ***activities*** can also be a very good ***regularizer***.
* **L2 weight-decay via noisy inputs**

So let's look at what happens if we add Gaussian noise to the *inputs* to a *simple neural network*.

* The **variance** of the noise gets amplified by the *squared weights* on the *connections*, before going into the *next hidden layer*.
* If we have a very *simple net*, with just a *linear output unit* that's directly *connected* to the *inputs*, the amplified noise then gets *added* to the *output*.

|  |  |
| --- | --- |
| * So if you look at the diagram, we put in an input with additional Gaussian noise that's sampled from a Gaussian with mean and variance . * That additional noise has it's *variance* multiplied by the *squared weight*. It then goes through the linear output unit . And so what comes out of is: the (would have come out before) plus Gaussian noise that has mean and variance .   Input:  Output:   * This *additional variance* makes an *additive contribution* to the *squared error*. You can think of it like **Pythagoras theorem**, that the ***squared error*** is going to be the ***sum*** of the ***squared error*** *caused by* , and this ***additional noise*** , because the *noise* is independent of . * So *minimizing the squared error* tends to *minimize the squared weights* when the *inputs* are *noisy*. |  |

* When we *minimize* the *total squared error*, we'll *minimize* the *squared error* that will come out if it was a *noise-free system*. And in addition, we'll be minimizing that second term.
* i.e. we'd be minimizing the *expected squared value* of that *second term* and the expected squared value is just , so that corresponds to an L2 penalty on with a penalty strength of .

Derivation

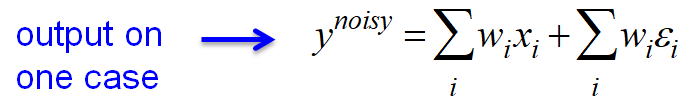
* Following is the derivation of that:
* The *output*, , when we add *noise* to all of the *inputs*, is:
  + : the *output* would have been with *noise-free system* i.e. the sum of all the inputs ,
  + : is the sum of times the noise that we added to each input.
  + And those *noises* are sampled from a *Gaussian* with mean and variance .
* So if we compute the *Expected Squared Difference* between and the *target value* , that's the quantity that's shown on the left-hand side of the equation.
* We're using followed by , i.e. to denote an ***expectation***, so it's not the *error*, it's an *expectation*.
* We're computing the *expectation* of the *things in the square brackets*. In this case, we're computing the *expectation of the squared error* that we'll get with the *noisy system*.
* So if we substitute the equation above for , with , then after calculating the square we get:
* isn't inside of *expectation bracket* because it doesn't involve any ***noise***.
* The second term is the cross product of and the third term is the square of .
* Now that equation simplifies a lot. In fact, it simplifies down to the *normal squared error* plus the expectation of .

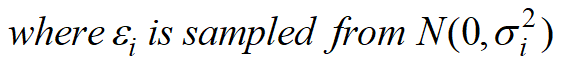
|  |  |
| --- | --- |
|  | The reason it simplifies is because is independent of and is independent of . |

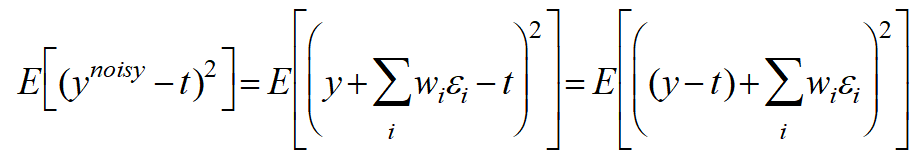
* So if you look at the last term, when we *multiply at that square*, all of the *cross terms* have an *expected value of zero*. Because we're *multiplying* together *two independent things* that are zero mean (because is independent of ).
* If you look at the *middle term*, that also has an *expectation of zero*, because each of the is independent of the residual error .
* Now we can rewrite the *expectation* of i.e. , as simply , because the *expectation* of is just , because that's how we generated .

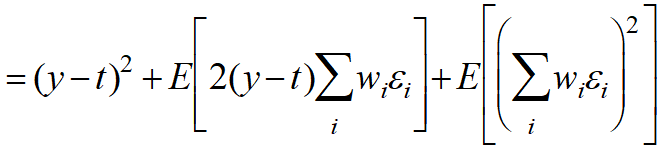
And so we see that the *expected squared error* we get is just the *squared error* we get in the *noise free system* plus this additional term . And that looks just like an L2 penalty on the with thebeing the strength of the penalty.

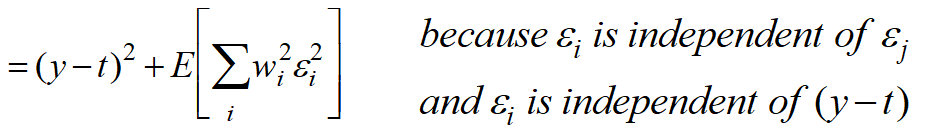
So, is equivalent to an L2 penalty.

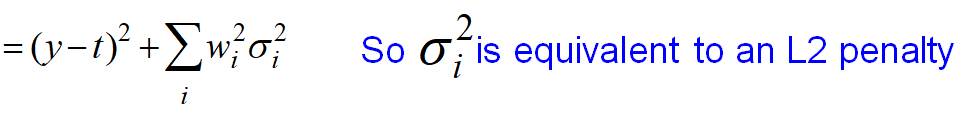












* **Noisy weights in more complex nets**
* In more complex nets, we can *restrict* the *capacity* by adding *Gaussian noise* to the *weights*.
* This is *not exactly equivalent* to *an L2 penalty*. But it seems actually to work better, especially in *Recurrent Networks*.
* *Alex Graves’ recurrent net* that recognizes handwriting, works significantly better if noise is added to the weights.
* **Using noise in the activities as a regularizer**

We can also use *noise* in the *activities* as a *regularizer*. Suppose we use *backpropagation* to train a multilayer neural net composed of *logistic hidden units*.

* What's gonna happen if we make the units *binary & stochastic* on the *forward pass* but then we do the *backward pass* as if we'd done the *normal deterministic forward pass* using the *real values*?
* So we're going to treat a *logistic unit*, in the *forward pass*, as if it's a *stacastic binary neuron*. That is, we compute the output of the **logistic p**, and then we treat that **p** as the probability of outputting a one (i.e. ).
* And in the *forward pass*, you make a *random decision* whether to output a *one* or a *zero* using that *probability*.

|  |  |
| --- | --- |
| * But in the *backward pass*, you use the **real value of p** for *back propagating derivatives* through the *hidden unit*. * This is *not exactly correct*, but it's *close* to being a *correct thing* to do for the stochastic system *if all of the units make small contributions to each unit in the layer above*. |  |
| * When we do this, the ***performance*** on the *training set* is *worse* and *training* is considerably *slower*. It may be several times slower. * But it does significantly *better* on the *test set*. This is currently an unpublished result. |  |

**9.4 Introduction to the Full Bayesian Approach**

In this section, we'll discuss the *Bayesian Approach* to *fitting models*, using a simple *coin tossing* example.

The main idea behind the Bayesian Approach is: instead of looking for the *most likely settings* of the *parameters* of the model, we should consider *all possible settings* of the *parameters* and try to *figure out* for *each of those possible settings*, how *probable* ***it*** is, given the data we observed.

* **The Bayesian framework**

The Bayesian framework assumes that we always have a *prior distribution* for *everything* (we always start with a belief about *how likely* an *event* is).

* This means that for any event, no matter what it is, we have to *assign an initial estimate* of how likely we think it is to happen, even before seeing any data.
* That is, for any *event* we might care to *mention*, we *must have* some *prior probability* that the event might happen.
* The prior might be very vague. So what's happening is, our *data* gives us a *likelihood term*. We combine it with our *prior* and then we get a *posterior*.
* When we see some data, we combine our *prior distribution* with a *likelihood term* to get a *posterior distribution*.
* The *likelihood term* takes into account *how probable* the *observed data* is given the *parameters of the model*.
* The *likelihood term* favors *settings of our parameters* that make the data more likely.
* It can disagree with the prior.
* *With* ***enough data*** *the* ***likelihood terms*** *always wins:* If we get enough data, however unlikely the prior is, the data can overwhelm it. And in the end, with enough data, the *truth will out*. That is, even if your *prior's wrong*, you'll end up with the *right hypothesis*.
* But that may take an *awful lot of data* if you thought that things were very *unlikely* under your *prior*.

Note: Prior vs Posterior

* ***Prior Distribution:*** The prior distribution represents your *initial beliefs* about a parameter or variable *before observing any data*. It reflects what is known or assumed about the parameter based on *previous knowledge or intuition*, without considering *new evidence*.
* Example: If you’re *estimating* the probability of a coin landing *heads*, you might initially assume a 50% chance, reflecting your *prior belief* (assuming the coin is fair).
* ***Posterior Distribution:*** The posterior distribution represents the updated beliefs about a parameter after observing the data. It combines the prior distribution with the new data (likelihood) using Bayes' Theorem to update the understanding of the parameter.
* Example: After *flipping the coin multiple times* and observing more heads than tails, your *posterior distribution* will adjust to reflect the higher likelihood of the coin being *biased toward heads*.

In simple terms, the prior is what you believe before the data, and the posterior is what you believe after seeing the data.

* **A coin tossing example**

Let's use a *coin toss* as an example. Imagine we *don’t know much about coins*, other than the fact that tossing a coin results in either **heads** or **tails**.

* We assume:
* The probability of getting heads is some unknown value **p**.
* The probability of getting tails is **1-p**.
* Each coin toss is an *independent decision*, meaning the *outcome of one toss* doesn’t affect the *outcome of the next*.
* So our coin model has one parameter **p**, which represents the probability of getting heads.
* Now, if we *toss* the coin *100 times* and get *53 heads*, what would be a good *estimate* for p? Most would suggest **p = 0.53** (since 53 out of 100 tosses were heads). But why is that a good estimate?
* The *frequentist approach* (or *maximum likelihood*) suggests choosing the value of p that makes the *observations* most *probable* (53 heads out of 100 tosses). In this case, **p = 0.53** is that value. Let’s go ahead and derive why this is true.

Derivation: coin tossing

* Let's consider is the probability of a particular sequence that contains *53 heads* and *47 tails* and it could be written out by:
* p every time you *toss a head* And 1-p every time you *toss a tail*. And then if we *collect* all the p 's together, and all the 1-p's together, we get , and.
* If we now ask, how does the *probability of observing that data depend on* p, we can differentiate with respect to p, and we get the expression shown below:
* If we then set that derivative to *zero*. We discover that the *probability of the data* is *maximized* by setting p=0.53. So that's *maximum* *likelihood*.

|  |
| --- |
|  |

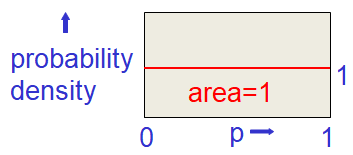
* **Some problems with picking the parameters that are most likely to generate the data**

There are some problems with using *maximum likelihood* to determine the *parameters* of a model.

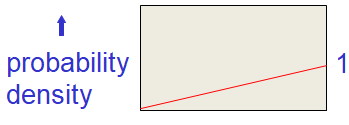
* ***Is p=1 a sensible answer?:*** For example, suppose we *tossed* a coin just once and it landed *heads*.
* It doesn't really make sense to conclude: *"We believe the probability of the coin landing* ***heads*** *in the future is 1"*. That would imply we'd be willing to bet at infinite odds that it will *never land* ***tails***, which seems absurd.
* ***Surely p=0.5 is a much better answer:*** It's sort of intuitively obvious that a better answer is 0.5. But how can we justify that?
* More importantly, we can ask, *"Is it reasonable to give a single answer?"*
* If we don't have much data, we don't know much, so we're *uncertain* about the value of *p*. What we really should do is *avoid giving a single answer* and instead provide a *probability distribution* over *possible answers*.
* Our computations of probabilities will work much better if we take this uncertainty into account.
* A answer like 0.5 is fairly likely, while a answer like 1 is still quite unlikely, especially if we have a *"prior belief"* that coins land ***heads*** about *half the time*."
* **Using a distribution over parameter values**

Let’s go through an example where we begin with a *prior distribution (an initial guess)* about the value of a parameter, like p (the probability of heads). We'll choose a *simple prior distribution*, even if it doesn't perfectly match what we believe about coins.

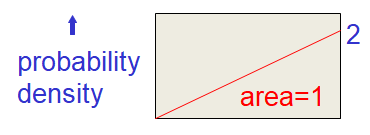
* Then we'll show how that *prior distribution* get *modified by data* if we adopt the Bayesian Approach (i.e. after we look at the data, like the coin toss results).
* We’ll start with a *prior distribution* that assumes *"all the different values of p (the probability of heads) are equally likely"*. This means we believe coins can have *any degree of bias*, and each type of bias is equally possible. For example, some coins might *land heads 50% of the time*, while others might *land heads 100% of the time*, and we think both are equally likely.
* So we start with a *prior distribution over p*. In this case we used a *uniform distribution*.



* Multiply the *prior probability of each parameter value* by the *probability of observing a head given that value*: Now let's observe a *coin landing heads*. So what we do now is: *for each possible value of p (the probability of getting heads), we take its* ***prior probability*** *and we multiply by the* ***probability*** *that we would have observed a head, given that, that value of p is the correct one.*
* So, for example, if we take the value of **p=1** which says *coins come down heads every time* then the *probability of observing a head* would be **1** (there would be no alternative). And if we take the value of **p** to be **0** the *probability of observing a head* would be **0**.
* And if we take it to be **p=0.5**, the *probability of observing head* would be **0.5**.
* So we take that ***red line***, that's our ***prior***, and we multiply each point on that by the *probability of observing a head* according to that hypothesis.



* Now we get the *sloping-line*, that's our ***unnormalized posterior***. It's *unnormalized* because the area under that line doesn't *add up* to **1**. For a probability distribution, the *probabilities of all the alternative events* have to add to **1**.
* ***Re-normalize:*** So the last thing we do is re-normalize it, i.e. *scale up all of the probability densities* so that their integral comes to 1. This gives the *posterior distribution*.



* We *scale everything up*, so the *area under the curve* is 1. And now if we started with the *uniform prior distribution* of a **p**, we end up with this *triangular posterior distribution* of a **p**, having *observed one* ***head***.
* **Let's do it again: Suppose we get a tail**

Now let's do it again. And this time let's suppose we get a ***tail***. So, the *prior distribution* that we start with now is the *posterior* *distribution* we had *after observing* our *one head*.

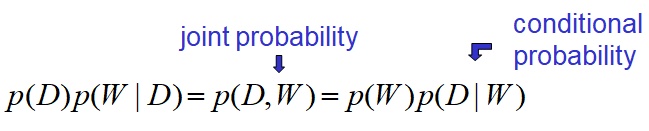
|  |  |
| --- | --- |
| * *Start with a prior distribution over p:* * And now, the ***green line*** shows the *probability* that we will *get a* ***tail*** according to each of those *hypotheses* that correspond to a value of **p**. * So, for example, if **p=1**, the probability we would observe a *tail* is *zero*. |  |
| * *Multiply the* ***prior probability*** *of each parameter value by the* ***probability of observing a tail*** *given that value:* * So we have to multiply our prior by our likelihood term, and now we get a curve like this. |
| * *Then renormalize to get the posterior distribution:* We have to re-normalize to make the *area to be 1*. And that's now a posterior distribution, after having *observed* *one head* and *one* *tail*. * Notice that it's a pretty *sensible distribution*. After observing one of each, we know that p can't be either *zero* or *one*, and it also seems very sensible that the "*most likely* thing" is now in the *middle*. |

|  |  |
| --- | --- |
| * **Lets do it another 98 times**   If we do this again another 98 times, and keep *applying* the *same strategy* of: *multiplying the posterior we had after the last toss, by the likelihood of* ***observing*** *that event,* ***given*** *the various* ***different settings*** *of the parameter* p.   * And let's suppose we get 53 heads and 47 tails in all. Then we'll end up with a curve that looks like this: * That is, after 53 heads and 47 tails we get a very sensible *posterior distribution* that has its *peak at 0.53* (assuming a uniform prior). * It'll have its *mean* at *0.53* because we started with the *uniform prior*. And it'll be fairly *sharply peaked* by *0.53*. * But it'll allow other values like *0.49* as a perfectly reasonable value under this curve. Not quite as likely as *0.53*, but very reasonable. * Whereas a value of 0.25 is extremely unlikely under this curve. |  |

* **Bayes Theorem**

So we can summarize all that with **Bayes Theorem**. Determine the *middle* of folowing equation is the *joint probability* of a set of parameters W, and some data D.

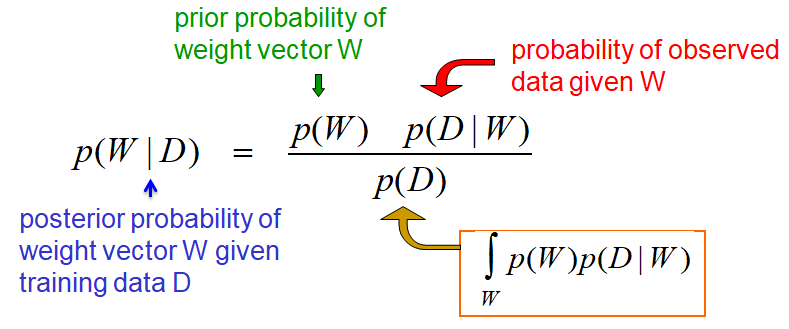
* For *supervised learning*, the data is gonna consist of the target values. So we assume we are given *inputs* and the *data values consist of* *the target values* that are *associated with those inputs* (that what we observe).



* That joint probability can be re-written as the product of an *individual probability* and a *conditional probability*. So on the right, we're written it as , and on the left, w've written it as .

|  |  |
| --- | --- |
| * Now we can divide both sides of the equation by . And this gives us *Bayes Theorem* in it's usual form: * Bayes Theorem says that the *posterior probability* of a *particular value of W, given the data D*, is just the *prior probability* of that *particular* *value of W* times the *probability* given that *particular value of W, that you would have observed the data you observed*. |  |

* And that has to be normalized by the probability of the data which is simply the *integral* over all possible values of , of , i.e. .



* The *bottom line* needs to be the *sum of the top line* over all possible values in order for this to be a *probability distribution that adds to 1*. Because that has *integrated over all possible values* of , it's *not affected* by picking a *particular value* of on the left-hand side.
* So when we're looking for the *best value* of , for example, we can ignore . It doesn't depend on . The other two terms on the right-hand side, however, do depend on .

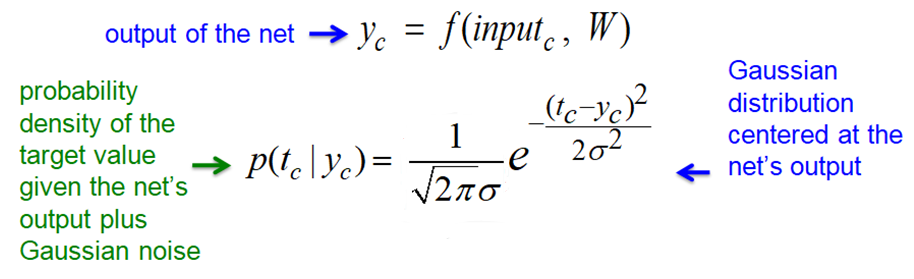
**9.5 The Bayesian interpretation of weight decay**

In this section, we’ll discuss the *Bayesian interpretation* of *weight penalties*. In the Full Bayesian Approach (previous section), we try to compute the *posterior probability* for *every possible setting of the parameters* of a model.

* But there's a much *reduced form* of the *Bayesian approach*, where we simply going to *"look for the single set of parameters that is the best compromise between fitting our prior beliefs (about what the parameters should be like), and fitting the data we've observed"*.
* This simpler *Bayesian* approach, called ***Maximum a Posteriori (MAP) learning***, and it gives us a nice explanation of what's really going on, when we use *weight decay* to control the *capacity* of a model.
* *Maximum a posteriori (MAP) learning* selects a single most likely hypothesis given the data.
* **Supervised Maximum Likelihood Learning**

Let's briefly discuss what’s really happening when we *minimize* the *squared error* during *Supervised Maximum Likelihood Learning*.

* Finding the *weight vector* that *minimizes* the *squared residuals*, i.e. the differences between the *target value* and the *value predicted* by the net, is equivalent to *finding a weight vector* that *maximizes* the *log probability density* of the *correct answer*.
* In order to see this *equivalence*, we have to assume that the *correct answer* is produced by *adding* *Gaussian noise* to the *output* of the neural net. So the idea is:
* We make a *prediction* by first *running the neural net* on the input to get the output.
* Then *adding* some *Gaussain noise*, we ask: "What's the probability that when we do that, we get the correct answer?"

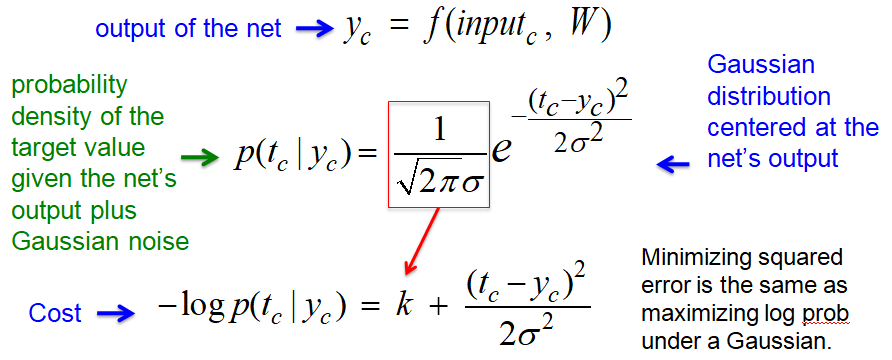


* So the *model's output* is the *center of a Gaussian* and what we're interested in is: having the *target value of high probability* under that *Gaussian*.
* Because the *probability* producing the *value t*, given that the network gives an *output of y* is just the *probability density of t* under a *Gaussian centered at y*.
* Let's suppose that the *output of the neural net* on *training case c* is and this output is produced by applying the to the .
* The *probability* that we'll get the *correct target value* when we add *Gaussian noise* to that output is given by a *Gaussian* *centered on* .

: Probability density of the target value given the net’s output plus Gaussian noise

: Gaussian distribution centered at the net's output

* So we're interested in the *probability density of the target value*, under a *Gaussian centered at the output* of the neural net.
* On the right side, we have that *Gaussian distribution centered at the net's output*, with mean . We also have to assume some *variance*, and that variance will be important later.
* If we now take *logs* and put in a *minus* sign, we see that the *negative log probability density* of the *target value* *given* that the *network* *outputs* , is a constant (that comes from the normalizing term of the Gaussian ) plus (the log of that exponential).



* Therefore, if our *cost function* is the *negative log probability* of getting the *right answer*, that turns into *minimizing* a *squared distance*.
* It's helpful to know that whenever you see a *squared error* being *minimized*, you can make a *probabilistic interpretation* of what's going on, and in that probabilistic interpretation, you'll be ***maximizing*** *the* ***log probability*** *under a* ***Gausian***.

Supervised Maximum Likelihood Learning

*Supervised Maximum Likelihood Learning* is a method where we train a model (like a neural network) to make predictions that match the *observed target values* as closely as possible. In this approach, we assume that the observed *target values* come from a *probability* *distribution* centered around the model’s output with some noise.

* For example, if we assume the target value is normally distributed around the network’s output with *Gaussian noise*, we can express this by saying that the *probability density* of the *target value* , given the network's output , follows a *Gaussian (normal) distribution*. The *Gaussian distribution* is *centered* at with some variance .
* ***Probability Density Function (PDF):*** Given the target and the network’s output , the probability density function can be written as:

where:

: the model's output for input

: the observed target value,

: the variance of the Gaussian noise (assumed constant).

* ***Maximum Likelihood Estimation (MLE):*** In supervised *maximum likelihood learning*, we aim to maximize the likelihood of the observed data. For a dataset with N examples

the *likelihood* of all observations given the network *output* , is the product of the *individual probabilities*:

Taking the *log-likelihood* makes the computation easier and converts the product into a sum:

Simplifying, we get

Since and

To maximize the likelihood, we can equivalently minimize the negative log-likelihood (since maximizing the likelihood is the same as minimizing the negative log-likelihood):

This expression shows that minimizing the negative log-likelihood is equivalent to minimizing the mean squared error (MSE), commonly used in supervised learning, when the noise is Gaussian.

Therefore, *Supervised Maximum Likelihood Learning* under *Gaussian noise* is equivalent to *minimizing the squared error* between the *model’s predictions* and the *observed targets*.

* ***Note:*** When dealing with continuous variables (like the target value y in regression), we use probability density. We describe the *likelihood of different outcomes* using *probability density functions (PDFs)*, rather than *discrete probabilities*.
* A *probability density function* tells us the *likelihood* of *observing a value* within *a small range around a point*, rather than an exact probability of a *single point* (since the probability of any specific point in a continuous distribution is technically zero).
* So, is the *probability density* of observing the *target* given the model’s *output* and the assumed *Gaussian noise*.
* For discrete target variables (like classification), we use probability.
* **MAP: Maximum a Posteriori**

The proper *Bayesian approach*, is to find the *full posterior distribution* over *all possible weight vectors*.

* If we have more than a *handful of weights*, that's hopelessly *difficult* when we have a *non-linear net*.
* Bayesians have a lot of ways (clever tricks) of *approximating* this horrendous distribution, often using Monte Carlo methods. But for the time being, let's try and do something simpler.
* Let's just try to find the *most probable weight vector*. So the *single setting of the weights* that's most probable given the *prior* *knowledge* we have and given the *data*. So what we're going to try and do is:
* Find an optimal value of by starting with some random weight vector, and then adjusting it in the direction that improves the probability of that weight vector given the data i.e. .
* But it will only be a local optimum.

Now, it's going to be easier to work in the *log domain* than in the *probability domain*. So if we want to *minimize a cost*, we better use *negative log probs*.

* **Why we maximize sums of log probabilities**

Just an aside about why we *maximize* sums of *log probabilities*, or *minimize* sums of *negative log probs*, what we really want to do is *maximize the probability* of the data, which is *maximizing the product of the probabilities of producing all the target values* that we *observed on all the different training cases*.

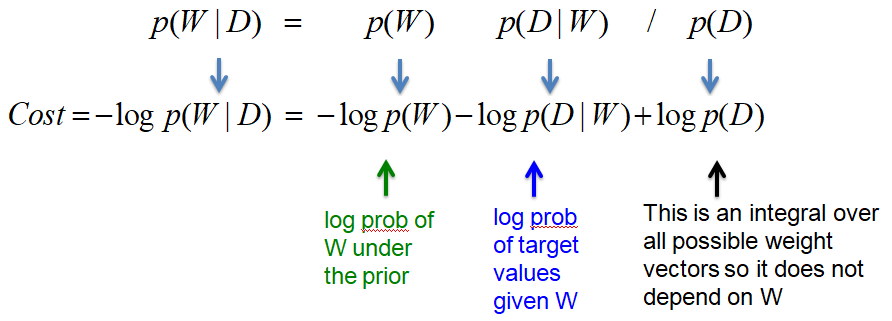
* If we assume that the *output errors* on *different cases* are *independent*, we can write that down as the *product over all the training cases*, of the *probability of producing the target value*, , given the *weights* .
* That is the *product of the probability* of producing , given the *output* that we're going to get from our network, if we give it *input* and it has *weights* .
* The *log function* is *monotonic*, and so it can't change where the maxima are. So instead of *maximizing* a *product of probabilities*, we can *maximize* the *sums of log probabilities*, and that typically works much better on a computer. It's much more stable.
* So we *maximize* the *log probability* of the data D given the weights W, which is simply *maximizing the* ***sum*** *over all* ***training cases*** of the *log probability of the output* for that *training case*, given the *input* and given the *weights*.

|  |
| --- |
| Because the log function is monotonic, it does not change where the maxima are. So we can maximize sums of log probabilities |

* **MAP: Maximum a Posteriori (the math)**

In *Maximum A Posteriori Learning*, we're trying to find the *set of weights* that optimizes the *trade-off* between *fitting our prior* and *fitting* *the data*. So that's Bayes's theorem.

* If we take *negative logs* to get a *cost*, we get that the *negative log* of the *probability of the weights* given the *data*, is the *negative log of the* *prior* term, and the *negative log of the data term*, and an *extra term*.



* So, that last extra term , is an *integral* over all *possible weight vectors*. And so that doesn't affect . So we can *ignore it* when we're *optimizing* .
* The term that *depends on the data* is the *negative log probability of target values given* , and that's our *normal error term*.
* The term that *only depends on*  is the *negative log probability of under its prior*.
* **The log probability of a weight under its prior**

Maximizing the *log probability of a weight* is related to minimizing a *squared distance*, in just the same way as *maximizing* the *log probability of producing correct target value* is related to *minimizing* the *squared distance*.

* *Minimizing the squared weights* is equivalent to *maximizing the log probability of the weights* under a *zero-mean Gaussian prior*.
* Here's a *Gaussian*. It's got a *mean zero*, and we want to *maximize* the *probability of the weights*, or the *log probability of the* *weights*. To do that, we obviously want to be *close to the mean* 0.
* The equation for the Gaussian is just like below, where the *mean is* 0 so we don't have to put it in.

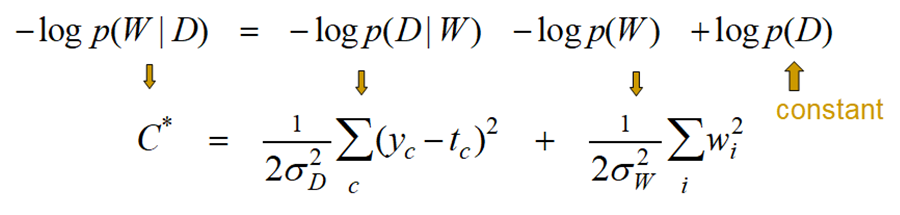
|  |  |
| --- | --- |
|  |  |

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

* So the log probability of is then the *squared weights* () scaled by *twice the variance* (), plus a constant () that comes from the *normalizing term of the Gaussian* and isn't affected when we change .
* **The Bayesian interpretation of weight decay**

Finally we can get to the *Bayesian Interpretation* of *weight decay* or *weight penalties*.

* We're trying to *minimize* the *negative log probability* of the *weights* given the *data* and that involves minimizing a term that ***depends*** *on both* ***data*** *and the* ***weights*** namely how well they fit the targets and the term that depends only on the weights.

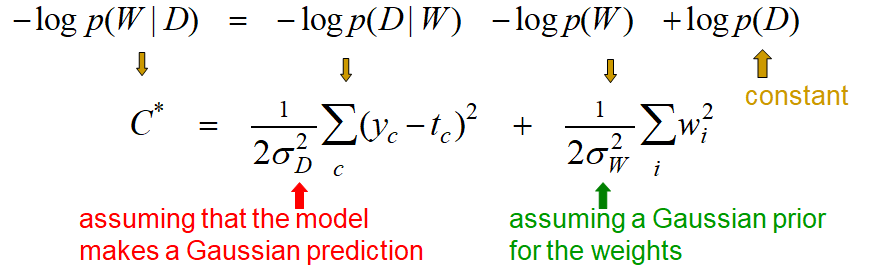


* Cost function is derived from the *log probability* of the *data* given the *weights*, which if we *assume Gaussian noise is added to the output of the model to make the prediction*, then that *log probability* is the *squared distance* between the *output of the net* on the *target value* scaled by *twice the variance* of that *Gaussian noise*.

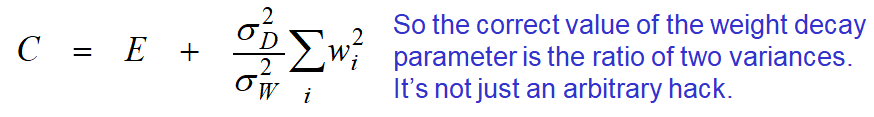
|  |  |
| --- | --- |
| Assuming that the model makes a Gaussian prediction |  |

* Similarly, if we assume we have a *Gaussian prior* for the *weights*, the *log probability of a weight* under the *prior* is the *squared* value of that *weight* scaled by *twice the variance* of the *Gaussian prior*.

|  |  |
| --- | --- |
| Assuming a Gaussian prior for the weights |  |



* Let's take that equation and multiply through by . Then we got a *new cost function* . And the *first term*, when we multiply through, turns into simply the *sum over all training cases* of the *squared difference* between the *output* of the net and the *target*. that's the ***squared error*** that we typically minimize in the neural net.



|  |  |
| --- | --- |
| * The *second term* becomes the ratio of times the sum of the squares of the weights . And so what you see is, the ratio of those two variances is exactly the *weight penalty*. |  |

* So we initially thought of *weight penalties* as just a number *you make up to try* and make things work better. Where you *fix* the value of the *weight penalty* by using a *validation set*.
* But now we see that if we make this GAUSSIAN INTERPRETATION where we have a *Gaussian prior* and we have a *Gaussian* *model* of the *relation* of the *output* of the net to the *target*.
* Then the *weight penalty* is determined by the *variances* of those Gaussians. It's just the *ratio of those variances*. It's not an arbitrary thing at all within this framework.

**9.6 MacKay's quick and dirty method**

In this section, we're going to describe a method developed by David MacKay in the 1990s for *determining* the *weight penalties* to use in a neural network *without using a validation set*.

It's based on the idea that we can *interpret weight penalties* as doing *map estimation* so that the *magnitude* of the *weight penalty* is related to the *tightness* of the *prior distribution over the weights*.

* Mackay showed we can *empirically fit* both the *weight penalties* and the *assumed noise* in the *output of the neural net* to get a ***method*** for *fitting weight penalties* that does not require a *validation set* and therefore, allows us to have *different weight penalties* for *different* *subsets* of the *connections* in a neural network, something that will be very expensive to do using ***validation sets***.
* **Estimating the variance of the output noise**

Here's a simple and practical method developed by David MacKay for making use of the fact that: we can *interpret weight penalties* as the *ratio* of *two variances*.

* After we've learned a model to *minimize squared error* we can find the *best value* for the *output variance (output noise)*. And the best value is found by simply using the *variance* of the *residual errors*.
* The best value is the one that *maximizes the probability* of producing exactly the *correct answers* after adding *Gaussian noise* to the output produced by the neural net.
* The best value is found by simply using the *variance* of the *residual errors*.
* **Estimating the variance of the Gaussian prior on the weights**
* We can also estimate the *variance* in the Gaussian prior of the weight. We have to:
* Start with some *guess* about what this variance should be.
* Then, we do some learning,
* And then we use a very dirty trick called "empirical Bayes".
* We set the *variance* of our *prior* to be the *variance of the weights of the model learned* because that's the variance that will make those weights most likely.
* This really violates a lot of the *presuppositions* of the *Bayesian approach*. We're using the *data* to *decide* what our *prior beliefs* are.
* i.e. use the data itself to decide what your prior is!
* So, once we've *learned* the *weights*, we *fit a zero mean Gaussian* to the *one-dimensional distribution of the* ***learned weights***. And then, we take the *variance* of that *Gaussian*, and we use that for our *prior*.
* *We could easily learn different variances for different sets of weights:* Now, one nice thing about that is: the *different subsets of weights*, like in *different layers*, for example, we could learn *different variances* for the *different layers*.
* We don't need a *validation set* so we can use all of the non-test data for training.
* Because we don't need *validation sets* to determine the *weight penalties* in different layers, we can actually have many *different weight* *penalties*. This would be very *hard* to do with *validation sets*.
* **MacKay’s quick and dirty method of choosing the ratio of the noise variance to the weight prior variance.**

So, here's MacKay's method:

* You start by guessing the *noise variance* and the *weight prior variance*. Actually, all you have to really do is guess their ratio.
* Then, you do some *Gradient Descent Learning*, trying to improve the weights.
* Then, you *reset* the *noise variance* to be the *variance of the residual errors* and
* You *reset* the *weight prior variance* to be the *variance of the distribution of the actually learned weight*.
* And then, you go back around this loop again.