Chapter – 10

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

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

**Model mix, Bayesian method**

Lectures: Geoffrey Hinton

Why it helps to combine models

*Mixtures* of Experts

The idea of Full Bayesian Learning

Making *full Bayesian learning* practical

Dropout

**10.1 Why it helps to combine models**

In this section, we'll explore how *combining multiple models* can improve predictions.

* If we have a *single model*, we have to choose some *capacity* for it.
* If we choose too *little capacity*, it would be able to *fit the regularities* in the *training data*.
* And if we choose *too much capacity*, it won't be able to *fit the sampling error* in the particular training set we have (overfit the sampling error).
* By using many models, we can actually get a better tradeoff between *fitting* the *true regularities*, and *overfitting* the *sampling error* in the data (avoiding overfitting to noise in the data).

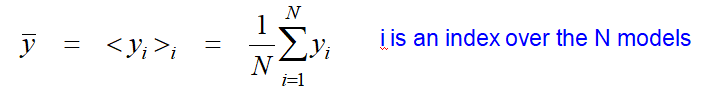
At the beginning of the section, we'll see that when we average models together, we can expect to do better than any single model. This effect is largest when the *models* make very *different predictions* from each other. Toward the end, we'll discuss various techniques to encourage the different models to make very different predictions.

* **Combining networks: The bias-variance trade-off**
* As we've seen before, when we have a *limited* amount of *training data*, we tend to get *overfitting*.
* If we *average* the predictions of many *different models* we can typically *reduce* that *overfitting*.
* This helps most when the models make very *different predictions* from one another.
* For *regression*, the *squared error* can be decomposed into a *bias-term* and a *variance-term*. And that allows us to analyze what's going on.
* The bias term is *big* if the model has too *little capacity* to fit the data. It measures how *poorly* the model *approximates* the *true function*.
* The variance term is *big* if the model has so *much capacity* that it's good at modeling the *sampling error* in our particular training set.
* It's called *variance*, because if we go and get *another training set* of the *same size* from the *same distribution*, our model will *fit differently* to that training set, because it has *different sampling error*.
* And so we'll get *variance* in the way the models *fit* to *different* training *sets*.
* If we *average models* together, what we're doing is we're *averaging* away the *variance*, that allows us to use individual models that have *high capacity* and therefore *high variance*.
* These *high capacity models* typically have *low bias*. So we can get the *low bias* without incurring the *high variance* by using *averaging* to get *rid of the variance*.
* **How the combined predictor compares with the individual predictors**

Let's analyze how an *individual model* compares with an *average of models*.

* On any one test case some *individual predictors* may be *better* than the *combined predictor*. But different individual predictors will be better on different cases.
* If the *individual predictors* *disagree* a lot, the *combined predictor* is typically better than all of the individual predictors when we *average* over *test cases*.
* So we should aim to make the *individual* predictors *disagree*, *without* making them be *poor* predictors (i.e. without making them much worse individually).
* The art is to have individual predictors that make *very different errors* from one another, but are each *fairly accurate*.
* **Combining networks reduces variance**

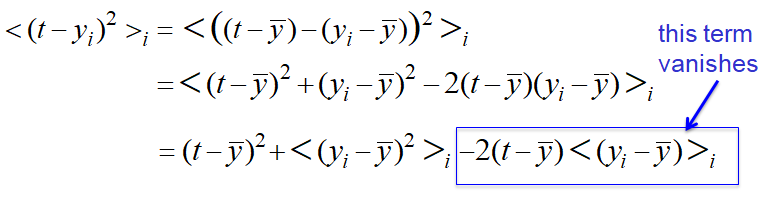
Let's look at the math when we combine networks. We want to compare two *expected squared errors*: *Pick a predictor* at *random* versus use the *average* of all the predictors:



* We're going to compare two *expected squared errors*:
* The *first expected squared error* is the one we get if we *pick one of the predictors* at *random* and use that for making our *predictions*.
* Then we *average overall predictors*, the error we'd expect to get if we followed that *policy*.

|  |  |
| --- | --- |
| * So is the average of what *all the predictors* say, and is what an *individual predictor* says. So is just the *expectation* over all the *individual predictors* of and we're using those *angle brackets* to represent an *expectation*, where the thing that comes after the angle bracket tells you what it's an expectation over. * The angle brackets <> are used to denote the *expected value* or *average* * The subscript i indicates that the expectation is *taken over all predictors i* * represents the value or prediction from an individual predictor i * We can write the same thing as . |  |

* Now, if we look at the *expected squared error* we'd get if we chose a *predictor* at *random*, what we'd have to do is *compare* that *predictor* with the *target*, take the *squared difference*, then *average* that over all predictors (left hand side of the following eqn).
* Note that, will disappear. Since doesn't have an in it anymore, and so we can forget about the *expectation brackets* for that.
* And that's the ***squared error*** we'd get if we compared the *average* of the models with the *target*. And our aim is to show the thing on the left hand side is bigger than that, i.e., by using that *average*, we've *reduced* the *expected squared error*.
* So the extra term we have on the right hand side, is the expectation . And that's just the *variance* of the . It's the *expected squared difference* between and .



* The last term disappears, it disappears because the difference of from we expect to be uncorrelated with the difference between the error that the average of the networks makes on the target.
* And so we're multiplying together two things that are *zero mean* and *uncorrelated* and we expect to get *zero* on *average*.

So the result is that the *expected squared error* we get by *picking a model at random* is greater than the *squared error* we get by *averaging* *the models by the variance of the outputs* of the models. That's how much we win by when we take an average.

* **A picture**

Consider the following picture, along the *horizontal* line, we have the possible values of the *output*, and in this case, all of the different models *predict* a *value* that is *too high*.

* The *predictors* that are *further than average* from t make *bigger* than *average squared errors*, like that bad guy in red, and
* The *predictors* that are *less than (nearer than) the average* distance from t make *smaller* than *average squared errors*.

|  |  |
| --- | --- |
| * The first effect dominates, because we're using *squared error* (and squares work like that). So if you look at the math, let's suppose that the *good guy* and the *bad guy* were *equally far* from the *mean*. So the *average squared error* they make is: * So we get the squared error that the mean of the predictors makes i.e. , plus . We win by *averaging predictors* before we compare them with the *target*. |  |

* That's not always true. It *depends* very much *on* using a *squared error*. For example, if we have a whole *bunch of clocks* and we try and make them more *accurate* by *averaging* them all, that'll be a disaster.

|  |  |
| --- | --- |
| * And it'll be a disaster because the *noise* you expect in *clocks* isn't *Gaussian noise*. * What we expect is that, *many* of them will be very *slightly wrong* and a few of them will have *stopped* or will be *wildly wrong*. And if we *average*, we make sure they are *all significantly wrong*, which is not what we want. |  |

* So don’t try averaging if you want to synchronize a bunch of clocks! The noise is not Gaussian.
* **What about discrete distributions over class labels?**

The same thing applies to the *discrete distribution* as we have our *class labeled probabilities*. Suppose that we have *two models*, one gives the *correct label* of *probability* of , and the other gives the *correct label* of *probability* of .

* Is it better to *pick* one *model* at *random*, or it is it better to *average* those *two probabilities*, and *predict* the *average* of and .
* What if our error measure is the *log probability* of getting the *right answer*? Then, the *log* of the *average* of and is going to be a *better* *bet* than the *log* of plus the *log* of averaged.

|  |  |
| --- | --- |
| * That's most easily seen in a diagram because of the shape of the log function. The black curve is the log, and is in the horizontal axis. The gold colored line, joins to . * We can see that if we first average and together, to get that *average value* at the *blue arrow* is, and then we compute the *log*, we get that *blue dot*. * Whereas if we separately take the , and , and then we *average* those two *logs*, we get the *mid-point* of that *gold line*, which is *below* the *blue dot*. |  |

* **Overview of ways to make predictors differ**

To make this *averaging* be a *big win*, we want our *predictors* to *differ* by a lot. And there's many different ways to make them differ.

* *Getting stuck in different local optima:* We could just rely on a *learning algorithm* that *doesn't work too well*, and get *stuck* in *different local optima* each time. It's not a very intelligent thing to do, but it's worth a try.
* *Model not using neural networks:* We could use *different* kinds of *models*, including ones that are *not neural networks*. Eg:

|  |  |
| --- | --- |
| * Decision trees * Gaussian Process models | * Support Vector Machines * and many others. |

* *Use different neural-network models:* If we really want to use a bunch of different neural-network models, we can make them *different* by using
* *Different number of hidden layers* or
* *Different number of units per layer* or
* *Different types of unit*. Like in some nets you could use *ReLU*, and in other nets you could use *logistic units*.
* Different *types* or *strengths* of *weight penalty*. So you might use *early stopping* for some nets, and an *L2 weight penalty* for others, and an *L1 weight penalty* for others.
* *Different learning algorithms*. For example you could use *full batch* for some, and *mini batch* for others, if your data set is small enough to allow that.
* **Making models differ by changing their training data**

We can also make the models *differ* by *training* the models on *different training data*.

* *Bagging:* Train different models on different subsets of the data. *Bootstrap aggregating* was proposed by ***Leo Breiman*** who also coined the abbreviated term "*bagging*" (*bootstrap aggregating*). *(Breiman developed the concept of bagging in 1994 to improve classification by combining classifications of randomly generated training sets)*.
* In ***bagging*** method, we train *different models* on *different subsets of the data*. We get these *subsets* by *sampling* the *training set* with *replacement*.
* *Bagging gets different training sets by using sampling with replacement:* For example we sampled a training set that had examples A, B, C, D, and E. And we got five examples, but we'll have some *missing* and some *duplicated*. And we train one of our models on that particular training set.

a,b,c,d,e -> a c c d d

* ***Random forests:*** Random forests use lots of *different decision trees* trained using *bagging*., which ***Leo Breiman*** was also involved in inventing. When we train lots of *different decision trees* with *bagging* and then *average* them together, they work much better than *single decision* *tree* by themselves.
* In fact, the ***Kinect box*** uses random forests to convert information about *depth* into information about where your *body* *parts* are. ("*Kinect box*" refers to the *Microsoft Kinect*, a *motion-sensing device* used for *gaming* and *body tracking*. Kinect is a *discontinued* product by Microsoft.)
* *Bagging with neural nets:* We could use *bagging* with *neural nets*, but it's very expensive. If we wanted to train *20 different neural nets* this way, we'd have to get our *20* different *training sets*. And then it would take *20-times long* as training *one net*. Also, at test time, we'd have to run these twenty different nets.
* But that doesn't matter with *decision tress* because they're so *fast to train* and they're also fast to use at *test* time.
* *Boosting:* Another method for making the training data different is to *train* each model on the *whole training set*, but to *weight* the cases *differently*.
* In boosting, we typically we use a *sequence of* fairly *low capacity models*. And we *weight* the *training cases* for each model in the sequence *differently*.
* Boosting *up weights* the cases that the previous *models got wrong* and we *down weight* the case of previous *models got right*.
* So the *next model* in the *sequence* doesn't *waste its time* trying to model cases that are *already correct*. It uses its *resources* to try to deal with *cases* the *other models* are *getting wrong*.
* An early use of boosting, was with neural nets for MNIST, when computer's are actually slower.
* One of the big advantage of boosting was that it focused to *computational resources* on *modeling* the *tricky cases*, And didn't *waste* a lot of time, going over *easy cases* again and again.

**10.2 Mixtures of Experts**

In this section, we're going to talk about the *mixture of experts model* that was developed in the early 1990s. The idea of this model is to train a *number of neural nets*, each of which *specializes* in a different *part* of the *data*.

* Mixture of experts (MoE) is a machine learning technique where multiple *expert networks (learners)* are used to divide a problem space into homogeneous regions. MoE represents a form of ensemble learning.
* That is, we assume we have a *dataset* which comes from a number of *different regimes*, and we *train a system* in which *one neural net* will *specialize* in *each regime*, and a *managing neural net* will look at the input data, and *decide* which *specialist (expert)* to give it to.
* This kind of system, *doesn't make very efficient use of data*, because the data is, fractionated over all these different experts. The model can be inefficient with small datasets because the data is spread across many different experts, which may not have enough data to learn well individually. As a result, the model's performance might suffer when data is limited.
* So with *small datasets*, it can't be expected to do very well.
* But as *datasets grow larger*, this kind of system may prove highly *effective*, as it can make excellent use of extremely large datasets.
* As the dataset grows larger, the model's structure becomes more useful. The large amount of data allows each expert to specialize in a different regime, and the system can make better use of the available data by allocating tasks to the right expert. This allows the model to scale effectively with increasing data.
* ***Experts:*** The model involves training *multiple neural networks*, called "*experts*," where each expert is specialized in handling a particular subset or "*regime*" of the data.
* The ***data*** is assumed to come from *different underlying distributions* or sources, and each expert is designed to learn well on a specific regime of the data.
* ***Managing Network:*** In addition to the experts, there is a *"managing network"* that receives the input and decides which expert should handle the given data point. This is essentially a gating mechanism that directs the flow of data to the appropriate expert.
* **Mixtures of Experts**

In ***boosting***, the *weights* on the models are *not all equal*. But after we *finish training*, each model has the *same weight* for *every test case*. We don't make the *weights* on the *individual models* *depend* on which *particular case* we're dealing with. But in *mixture of experts*, we do.

* Can we do better that just *averaging models* in a way that *does not depend* on the *particular training case*?
* So the idea is that: we can look at the *input* data for a *particular case* during both *training* and *testing* to help us decide which *model* we can *rely* on.
* During training this will allow particular *models* to *specialize* on a *subset* of the *training cases*.
* They *do not learn* on cases for which they are *not picked*. So they can ignore stuff they're not good at modeling. This will lead to *individual models* that are very good at some things and very bad at other things.
* The key idea is to make *each model/expert* focus on predicting the right answer for cases where it's *already doing better* than the other experts.
* This causes specialization.
* **A spectrum of models**

So there's a spectrum of models from very *local models* to very *global models*.

|  |  |
| --- | --- |
| * ***Very local models:*** *Nearest neighbors*, for example, is a very *local model*. * *Very fast to fit:* To fit it, you *just store the training cases*, then if you have to *predict* y from x, you simply find the stored value of x that's *closest* to the test value of x, then you *predict* the value of y that's the same as for the *stored value*. * *Local smoothing would obviously improve things:* The result of that is that the *curve* relating the *input to the output* consists of *lots* of *horizontal lines* connected by *cliffs*. It would clearly make more sense to *smooth* things out a bit. |  |

|  |  |
| --- | --- |
| * ***Fully global models:*** At the other extreme, we have *fully global models*, like fitting one *polynomial* to *all the data*. * *May be slow to fit and also unstable:* They're much *harder to fit* to data, and they may also be *unstable*. i.e, small changes in the data may cause big changes in the model you fit. That's because *each parameter* depends on *all the data*. |  |

* **Multiple local models**

In between those two ends of the spectrum, we have multiple *local models*, which are of *intermediate complexity*. So instead of using a *single global model* or *lots of very local models*, use *several models* of *intermediate complexity*.

* This is good if the *dataset* contains *several* different *regimes* and those different regimes have *different relationships* between *input* and *output*.
* *As an example Financial data depends on the state of the economy:* In *financial data* for example the *state of the economy* has a big effect on determining the mappings between *inputs* and *outputs*, and you might want to have *different models* for *different states of the economy*.
* But we might not know in advance how to decide *what constitutes* different *states* of the *economy*, we need to learn that too.
* *How do we partition the dataset into regimes?* Now the problem is: if we're going to use *different models* for *different regimes*, then how do we *partition* the dataset into these different regimes. We've several ways to do that, let's discuss these in following sub-sections.
* **"Partitioning based on input alone" vs "partitioning based on the input-output relationship"**

In order to fit *different models* to *different regimes* we need to *cluster the training data* into *subsets*, one for each of these regimes (for each local model).

* The aim of the clustering is NOT to find *clusters* of *similar input vectors*. We *don't* want to *cluster* the *data* based on the *similarity of input vectors*.

|  |  |
| --- | --- |
| * All we're interested in is the *similarity* of *input-output mappings*. We want each cluster to have a *relationship* between *input* and *output* that can be *well-modeled by one local model*. * Notice the on the right, four data points fit well with the *red parabola*, and another four fit well with the *green parabola*. * If we *partition* the data based on *input-output mapping* (i.e. fitting a parabola nicely to the data), the *split* happens at the *brown line*. * But if we *partition* based only on *input clustering*, the *split* occurs at the *blue line*, and then if we look to the *left of that blue line* (red parabola and part of the green parabola), we'll be stuck with a subset of data that *can't be modeled nicely* by a *simple model*. |  |

* **An error function that encourages cooperation**

Now we're going to explain an *error function* that *encourages models* to *cooperate*, and then we'll explain an *error function* that *encourages models* to *specialize*.

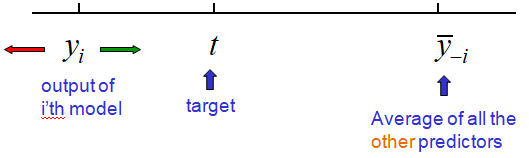
* Also we aim to provide a clear intuition for why these *two functions* have such *different effects*.

|  |  |
| --- | --- |
| * If we want to *encourage cooperation*, we *compare* the *average* of all the *predictors* with the *target* and *train* all the *predictors together* to *reduce* the *difference* between the *target* and their *average* (i.e. reduce discrepancy). * The *error* is the *difference* between the *target* and the *average* of all the predictors of what they predict. Here we're using angle brackets <> to represent *expectation*. * This can *overfit badly*. It makes the model much more *powerful* than *training* each predictor *separately*, because the models will learn to correct the errors made by others. |  |

* **A picture of why averaging models during training causes cooperation not specialization**

So, if we're averaging models during training, and train it so that the average works nicely, we have to consider cases like this.

* On the *right*, we have the *average* of all the models *except* for *model i*. So, that's what everybody else is saying when their votes are averaged together.
* On the *left*, we have the *output* of *model i*.



* Now if we'd like the *overall average* to be *closer* to the *target*, what do we have to do to the *output* of the *i-th model*?
* We have to *move it away* *from* the target. That will take the *overall average* *towards the target*.
* We can see that what's happening is: the *model i* is *learning* to *compensate* for the *errors* made by all the *other models*.

But do we really want to *move* *model i* in the *wrong direction*? Do we really want to *move* the output of *model i* *away* from the *target* value? Intuitively it seems better to move *model i* *towards the target*.

* **An error function that encourages specialization**

Following is an error function that encourages specialization, and it's not very different.

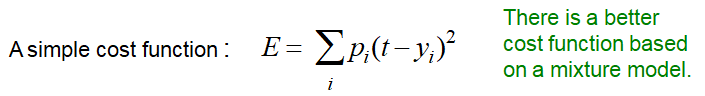
* *Compare each predictor separately with the target:* To encourage specialization, we compare the *output* of each *model* with the *target* separately.
* *Use a "manager" to determine the probability of picking each expert:* We also need to use a *manager* to *determine* the *weight* we put on each of these *models*, which we can think of as the *probability of picking* each *model*, if we have to pick one.

|  |  |
| --- | --- |
| * Now, our error is the *expectation over all the different models* of the *squared error* made by that model times the *probability of picking* that model. * Where the *manager* or *gating-network*, is *determining* that *probability* by looking at the *input* for this particular case. * *Most experts end up ignoring most targets:* If we try to minimize this error, the consequence will be that *most of the experts* will end up *ignoring most of the targets*. Each *expert* will only deal with a *small subset* of the *training cases* and it will learn to perform very well on that small subset. |  |

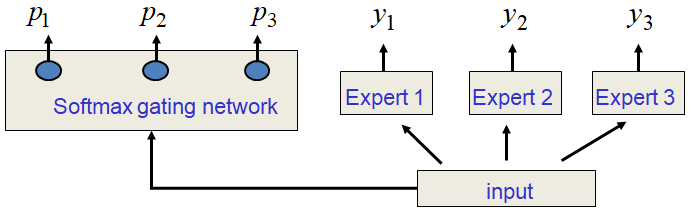
* **The Mixture of Experts architecture (almost)**

Let's briefly discuss the Mixture of Experts (MOE) architecture. In this approach, the *cost function* is defined as the *squared* *difference* between the *output* of each expert and the *target* *averaged* over all the *experts*.

* A simple cost function:
* In this approach, the *weights* used in the *average* are determined by the *manager*. However, there is a more effective cost -function, based on the *mixture model*, which we will discuss later.
* We consider this initial cost function, because it makes the intuition behind the concept easier to explain.



* **How it works:**
* We start with an *input*
* *Multiple experts* process the *input* and make *predictions* based on it.
* Alongside the experts, there is a ***manager*** component.



***The manager's role:***

* The *manager* can consist of multiple layers, with the *final layer* being a *softmax layer*.
* This *softmax* layer outputs a *set of probabilities*, *one for each expert*, indicating the *weight* or *importance* of each expert's prediction for the given input.
* ***Error calculation:*** Using the *probabilities* output by the *manager* and the *predictions* from the *experts*, we compute the value of that *error function*.
* **The derivatives of the simple cost function**

If we look at the derivative of that error function, the *outputs* of the *manager* are determined by the *inputs* to the *softmax group* in the *final layer* of the manager. And then the error is determined by the *outputs of the experts*, and also the *probabilities* output *by the manager*.

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

* If we differentiate that *error* w.r.t the *outputs of an expert*, we get a *signal* for *training that expert* and that *gradient* that we get w.r.t. the *output of an expert* is just the *probability* of *picking* that *expert*, times the difference between what that expert says (output) and the target.
* i.e. if we differentiate w.r.t. the outputs of the experts we get a signal for training each expert.
* So if the *manager decides* that there's a very *low probability of picking* that *expert* for that *particular training case*, the expert will get a very *small gradient*, and the *parameters* inside that expert *won't get disturbed* by that training case. It'll be able to *save its parameters* for *modeling* the *training cases* where the manager gives it a *big probability*.

|  |  |
| --- | --- |
| * We can also *differentiate* w.r.t the *outputs* of the *gating network*, if we do that, we get a *signal* for *training* the *gating net*. And actually what we're gonna do is— differentiate w.r.t, the *quantity* that goes into the *softmax*. That's called the *logit*, that's , * Differentiating w.r.t the *gating network outputs*, calculates how *changes* in the *gating network's outputs* (the *probabilities* assigned to *each expert* by the *softmax layer*) *affect* the *overall* *error*. This provides a training signal to improve the gating network. * But, instead of directly differentiating the gating network's outputs (the probabilities), we compute *gradients* with respect to the *inputs* to the *softmax* function. These inputs are often referred to as "logits" (denoted as ). * By differentiating with respect to logits, we *indirectly train* the gating network. This is because the *logits* determine the probabilities output by the softmax layer, which in turn influence the overall error. |  |

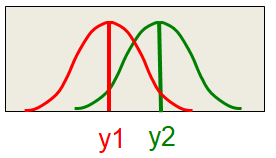
* And if we take the derivative w.r.t the logits , we get the *probability* that, that *expert was picked* times the difference between the *squared error* made by that *expert* and the *average overall experts* when you use the *weighting* provided by the *manager* of the squared error.
* We want to raise for all *experts* that give *less* than the *average squared error* of all the experts (weighted by ).
* If *expert i* makes a *lower squared error* than the *average of the other experts*, then we'll try to *raise* the *probability* () of *expert i*.
* But if *expert i* makes a *higher squared error* than the other experts, we'll try and *lower* its probability. That's what causes the specialization.

This approach ensures that the *gating network* is *updated* effectively to assign *optimal weights* to the *experts* during training.

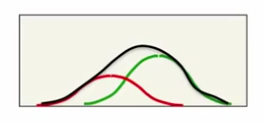
* **A better Cost Function for mixtures of experts (Jacobs, Jordan, Nowlan & Hinton, 1991)**

There exists a *better cost function*. It's a more advanced cost function that is based on mixture models, which we haven't discussed. Again, those will be well explained in *Andrew Ng's coursera* course.

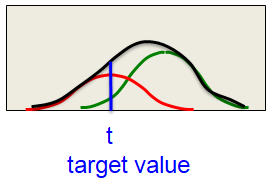
* However, we’ve seen the *interpretation* of *maximum likelihood*, when we're doing *regression*, as the idea that the network is actually making a *Gaussian prediction*. i.e., the network outputs a particular value, say y1, and this can be *interpreted* as a *prediction* that the *target* value follows a *Gaussian distribution* centered around y1 with *unit variance*.
* Think of *each expert* as making a prediction that is a *Gaussian distribution* around its *output* (with variance 1).



* So the *red expert* makes a *Gaussian distribution* of *predictions* around by y1 and the *green expert* makes a *Gaussian distribution* of *predictions* around y2.
* The *manager* then decides *probabilities* for the *two experts* and those *probabilities* are used to *scale down* the *Gaussians*. Those *probabilities* have to *add to 1* (i.e. sum of these probabilities = 1) and they are called *mixing proportions*.
* i.e. we think of the *manager* as deciding on a *scale* for each of these Gaussians. The scale is called a “*mixing proportion*”. e.g {0.4 0.6}

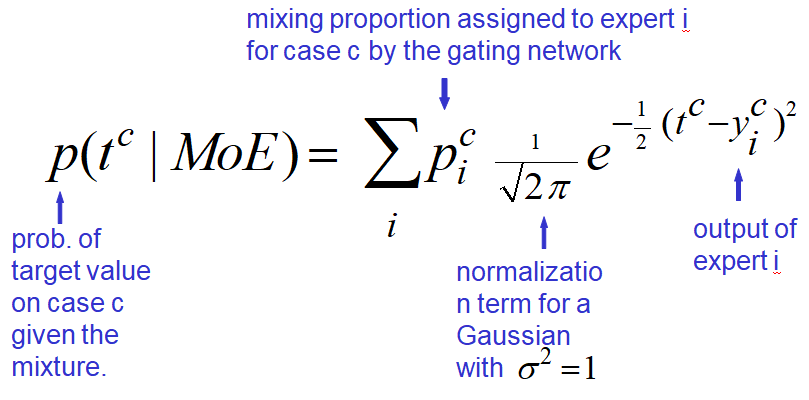


* Once we *scale down the Gaussians* we get to distribution that's no longer a Gaussian, is the *sum* of the *scaled down red Gaussian* and the *scaled down green Gaussian*. And that's the *predictive distribution* from *mixture of experts*.
* Now we want to *maximize* the log probability of the *target value* under that *black curve* (under this mixture of Gaussians model i.e. the sum of the two scaled Gaussians) and remember the *black curve* is just the *sum* of the *red curve* and the *green curve*.



So that leads to the following model for the *probability of a target*, given a *mixture of experts*.

* **The probability of the target under a mixture of Gaussians**
* The probability, is on the left, and it's the *sum over all the experts* , of the *mixing proportion assigned to that expert* by the *manager* or *gating network* times to the power the squared difference between the *target* and the *output* of that expert times , scaled by the normalization term () for a Gaussian with a variance of 1.
* And so our cost function is simply going to be the *negative log* of that *probability* on the left. We're going to try and *minimize* the *negative* *log* of that probability.



Gaussian with a variance of 1

* ***Note:*** The general formula for the probability density function (PDF) of a Gaussian distribution:

For variance 1, i.e.

Here the normalization term is , this ensures that the *integral of the probability density function (PDF) x equals 1*. i.e. ensures that the total probability under the Gaussian curve sums to 1.

**10.3 The idea of Full Bayesian Learning**

In this section, we will revisit the concept of *Full Bayesian Learning* (reccall *Chapter 9: Regularization*) and provide additional insights into how it works. In the following section, we will explore how it can be made practical.

* **Full Bayesian Learning**

In *full Bayesian learning*, we *don't try to find* a *best single setting* of the *parameters* (as in Maximum Likelihood or MAP). Instead, we'll try to find the *full posterior distribution* over *all possible parameter settings*.

* i.e., for *every possible setting*, we want a *posterior probability density*. And all those *densities*, we want to *add up to 1*.
* It's extremely *computationally intensive to compute* this for all, but the simplest models (its feasible for a biased coin).
* So, in the example earlier (*Chapter 9*), we did it for a *biased coin* which just has *one parameter*, which is: "*how biased it is*". But in general, for a neural net, it's impossible.
* After we've computed the *posterior distribution* across *all possible settings of the parameters*, we can then make *predictions* by letting each *different setting of the parameters* make its own *prediction*. And then, *averaging* all those *predictions* together, *weighting* by their *posterior probability*.
* To make predictions, let each different setting of the parameters make its own prediction
* Combine all these predictions by weighting each of them by the posterior probability of that setting of the parameters.
* This is also very computationally intensive.
* The **advantage** of doing this is that— if we use the *full Bayesian approach*, we can *use complicated models* even when we *don't* *have much data*.
* **Overfitting: A frequentist illusion?**

There's a very interesting philosophical point here. We're now used to the idea of ***overfitting***, When you fit a *complicated model* to a *small* *amount of data* (it ends up capturing noise instead of true patterns).

* But that's basically just a *result of not bothering to get* the *full Posterior Distribution* over the *parameters*. i.e. we often ignore the full range of possible parameter values (the posterior distribution) and focus only on a single estimate.
* So, frequentists would say— *"If you don't have much data, you should use a simple model",* because a complex one will overfit*.* And that's true.
* But it's only true if you assume that *"fitting a model"* means— *finding the single best setting of the parameters*.
* If we use the *full posterior distribution* over *parameter settings*, *overfitting disappears*.
* If there's *very little data*, the *full posterior distribution* will typically give you very *vague predictions*, because many *different settings of* *the parameters* that make very *different predictions* will have *significant posterior probability*.
* As you get *more data*, the *posterior probability* will get more and *more focused* on a *few settings* of the *parameters*, and the *posterior* *predictions* will get much *sharper*.

*From a Bayesian viewpoint, overfitting arises because we often simplify the problem by estimating only a single "best" set of parameters (e.g., using maximum likelihood estimation) instead of considering the full posterior distribution over all possible parameters.*

*The full posterior distribution provides a probabilistic description of the parameters, taking into account the uncertainty in the data. By focusing only on a single estimate, we miss this uncertainty, which can lead to overfitting.*

* **Frequentist:** The term frequentist refers to a school of thought in statistics where probabilities are interpreted as the long-run frequency of events. In the frequentist approach:
* In frequentism, ***probability*** represents the ***proportion of times*** an event will occur in repeated, identical experiments. For example, the probability of flipping heads on a fair coin is interpreted as the *fraction of heads* you would observe if you flipped the coin an *infinite number of times*.
* Here, the "proportion of heads" refers to the fraction or percentage of times the coin lands on heads compared to the total number of flips.
* In the frequentist interpretation, the proportion of heads would approach the true probability of flipping heads (e.g., 0.5 for a fair coin) as the number of flips increases toward infinity.
* Parameters (like the mean or variance) are treated as fixed but unknown values. They do not have probabilities because they are not considered random.
* **Inferences are based on the idea of sampling:** given repeated samples from the same population, frequentists make conclusions about the fixed parameter.
* **Confidence intervals** and **hypothesis** testing are central tools, relying on the behavior of the sampling process.

Unlike Bayesian statistics, which treats *parameters* as *random variables* and incorporates ***prior beliefs***, frequentists focus only on the *data at hand*, ***without*** introducing ***prior distributions***.

In short, **frequentism** is about interpreting probability as a long-run frequency and treating parameters as fixed but unknown quantities.

* **A classic example of overfitting**

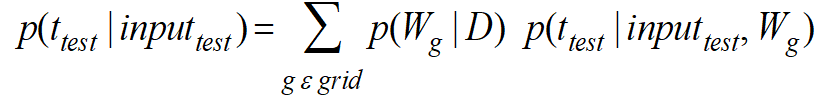
Following is a *classic* example of *overfitting*. We've got *six data points* and we fitted a *fifth order polynomial* and so it should *go exactly through the data*. We also fitted a *straight line* which only has *two degrees of freedom*.

|  |  |
| --- | --- |
| * The question is: which model do we believe? * The model (the polynomial) that has *six coefficients* and fits the data almost perfectly. Or * The model (the straight line) that only has *two coefficients* and doesn't fit the data all that well. * It's obvious that the complicated model (the polynomial) fits better. * But we don't believe it. * It's not economical. * And it also makes silly predictions. |  |
| * *Prediction:* Now let's look at the *blue arrow* in the figure below, and that's the *input value* and we're trying to *predict* the *output* value. * The *red curve (polynomial)* will predict a value that's lower than any of the observed data points, which seems crazy, * Whereas the *green line* will predict a sensible value. * *Using a prior of the fifth-order polynomials:* The situation changes if, instead of directly fitting a fifth-order polynomial, we introduce a reasonable *prior of the fifth order polynomials*. For example, this prior might assume that the coefficients of the polynomial should not be too large. |  |
| * We start with a reasonable prior over all fifth-order polynomials. * Next, we calculate the *full posterior distribution* over all those *fifth-order polynomials*. The image shown represents a sample from this distribution, where *thicker lines* indicate polynomials with *higher probabilities* in the posterior. * Some of the *thinner curves* may *miss* a few *data points* by a significant margin, but overall, they still *fit most of the data points* reasonably well. |  |

* Now, we get much *vaguer*, but much *more sensible predictions*.
* For example, at the point marked by the *blue arrow*, the individual models make very different predictions. However, when these predictions are averaged, the *result* is quite *similar* to the *prediction* made by the *green line*.
* *From a Bayesian prospective*, there's no reason why— *the amount of data we collect* should influence our *prior beliefs* about the *complexity* of the model.
* A true Baysian would say, *"you have prior beliefs about how complicated things might be and just because you haven't collected any data yet, it doesn't mean you think things are much simpler".*
* **Approximating full Bayesian learning in a neural net**

We can *approximate Full Baysian Learning* in a *neural net*, if the *neural net* has *very few parameters*.

* *Creating a Grid Over Parameters:*
* ***The idea is:*** If the *neural net* only has a *few parameters* we could put a *grid* over the *parameter space*, and evaluate at *each grid-point*. So each parameter is restricted to a few possible values.
* Then, we take the cross product of all those values for all the parameters.
* Now we get a set of grid points in the parameter space (representing different possible models).
* *Evaluating Each Grid Point:*
* At each grid point, we check how well the model predicts the data (i.e. if we're doing supervised learning, how well a model predicts the target outputs).
* The posterior probability in that grid-point is the product of:
* How well it *predicts the data* (model fits the likelihood) and
* How likely it is *under the prior*.
* The probabilities are then *normalized* so that they (posterior probability) sum to 1.
* *Advantages:* It is still very expensive, but notice it has some attractive features:
* There's no gradient descent involved.
* No local optimum issues: No risk of getting stuck in local optima since we evaluate all points independently.
* We're not following a path in this space, We're just evaluating a set of points in this space.
* *Making Predictions:*
* After evaluating each grid point we use all of them to make predictions on test data. i.e. Once we've decided on the *posterior probability* to *assign* to *each grid-point*, we then *use them all* to make *predictions* on the *test data*.
* This is also expensive, but it works much better than ML learning when the posterior is vague or multimodal (this happens when data is scarce). When there isn't much data, it'll work much better than *maximum likelihood* or *maximum a posteriori*.



* So, the way we *predict* the *test output, given the test input*, is that we say, the *probability* of the *test output, given the test input* , is the sum overall the grid points of the probability that, that grid-point is a good model:
* Is the sum over all grid-points of the *probability of that grid-point*, given the *data* and given our *prior* , times the probability that we will get that test output, given the input and given the grid-point .
* In other words, we have to take into account, the fact that we might *add noise to the output of the net* before producing the *test answer*.

|  |  |
| --- | --- |
| * **An example of full Bayesian learning**   Here's a picture of full Bayesian learning. We have a little net here,  that has *four weights* and *two biases*.   * If we allowed, nine possible values {-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2} for each of those weights and biases, there would be *grid points* in the *parameter space*. * It's a *big number* but we can *cope* with it. For *each* of those *grid-points*, we compute the *probability* of the *observed outputs* of all the *training cases*. * Multiply the *prior for each grid-point* (which might depend on the values of the weights, for example) by the *likelihood term* and *renormalize* to get the *posterior probability* for *each grid-point*. |  |

* Then we make *predictions* using those *grid-points*, but weigh to each of their predict ions by its posterior probability.
* i.e. we make *predictions* by using the *posterior probabilities* to average the predictions made by the different grid-points.

**10.4 Making full Bayesian learning practical**

In this section, we're going to describe how to make *full Bayesian learning* practical for neural networks that have *thousands*, and perhaps even *millions* of *weights*.

* The method used is a *Monte Carlo approach*, where a *random number generator* *explores* different *weight vectors*, *favoring* those that *reduce* the *cost function*.
* We use the *"random number generator"* to move around the *"space of weight vectors"* in a *random way* (to explore different combinations of weights), but with a *bias* towards going *downhill* in our *cost function*.
* So it’s guided to favor weights that reduce the cost function i.e., weights that make the model perform better
* Sampling from the Posterior: If we do this right, we get a beautiful property, which is that— *"we sample weight vectors in proportion to their probability in the posterior distribution"*.
* And that means— by *sampling a lot of weight vectors*, we can get a *good approximation* to the *full Bayesian method*.

i.e. this process selects *weight vectors* based on their *probability* in the *posterior distribution*. By sampling many such weight vectors, we can approximate full Bayesian learning effectively.

* **What can we do if there are too many parameters for a grid?**
* The number of *grid points* is *exponential* in the number of *parameters*. (i.e. the number of grid points grows exponentially with the number of parameters).
* So, we can't create a *grid* for more than a *few parameters*, as it becomes impossible to handle larger numbers of parameters with this method.
* If there is enough data to make *most parameter vectors* very *unlikely*, only a *tiny fraction* of the *grid points* will make a *significant contribution* to the *predictions*.
* So maybe we can just *focus on evaluating* this *tiny fraction* if we can find it.
* ***Idea:*** An idea that makes *Bayesian learning feasible* is that— it might be sufficient to just *sample weight vectors* according to their *posterior probabilities*. Consider the equation:
* So if you look at this equation, the *probability* that we assigned to a *test output* , given the *input for the test case* and the *training data* , is the *sum over all points in weight space* of the *posterity probability* of that *point in weight space* given the *training* *data*  , times the *probability distribution for the test values* that we predict given that *point in weight space* , and given the *test input* .
* Now *instead* of adding up *all the terms* in that *sum*, we could just *sample terms* from that *sum*.

|  |  |
| --- | --- |
| * What we do is— we *sample* the *weight vectors* in *proportion* to that *probability*. So either we sample them or we don't i.e. they'll get a weight of 1 or 0. |  |

* But the probability of getting a 1 i.e., the *probability being sampled*, will be their *posterior probability*. So that will give us the correct *expected value* for the right hand side.
* It'll have *noise* due to the *sampling* but it'll have the *correct expected value*.
* Let's break down the terms in the equation again:
* : This is the overall probability (or distribution) of the *test output* given the *test input* and all the *training data* . It represents our *final prediction* for the *test case*.
* : Each is a specific set of weights (a *weight vector*) for the neural network. Think of each as one possible model configuration.
* : This is the *posterior probability* of the *weight vector* given the *training data* . It tells us *how likely* that particular set of weights is after considering the data.
* : This is the probability (or *prediction*) for the *test output* given the *test input* , assuming the network uses the specific *weight vector* . It tells us what the network would *predict* if it had *those weights*.
* **Sampling weight vectors**

Here's a picture of what happens in *standard back propagation*. On the right we've drawn the *weight space*. Which of course is very *high* *dimensional* and *unbounded*. And this is a very *simplified picture* of it, but it's the best we can do.

|  |  |
| --- | --- |
| * In this *weight space*, we've drawn some *contours* which are meant to be *contours of equal values* of our *cost function*. And the way back propagation is normally used, is we *start* with some *small* value of the *weights*, and then we *follow* the *gradient*. * In *standard backpropagation* we keep moving the *weights* in the direction that *decreases the cost*. * i.e. we *move downhill* in our *cost function*, in the direction that *increases* the *log-likelihood* plus the *log prior*, *summed* over all *training cases*. * Eventually, we'll either end up at (the weights settle into) *a local minimum* or we'll get *stuck on a plateau*, or we'll just *move so slowly* that we run out of patience. * But the main point of this picture, is that we *follow a path* from an *initial point* to some final, *single point*. |  |

* **One method for sampling weight vectors**

Now if we're using a *sampling method*, what we could do is—

* ***Add Gaussian noise:*** We start at the same place as we did before, but each time we update the weights, we add a bit of *Gaussian noise* so we're *jittering* around.
* So we add some Gaussian noise to the weight vector after each update.
* So the *weight vector* will *never settle down*. It'll keep on *moving around*.
* It'll keep wander over the space, but always *preferring low cost regions* of the weight space. That is, it'll tend to *go downhill* if it can.

|  |  |
| --- | --- |
| * An important question is— *Can we say anything about how often the weights will visit each point in that space (i.e. each possible setting of the weights)?* * So the *red dots* are meant to be *samples* we took of the *weights* as we wandered around the space. * And the idea is— we might save the weights *after every 10,000 steps*. * If we look at those *red dots*, a few of them are in *high cost regions*, because those regions are quite big. * The *deepest minimum* has the *most red dots*, and other minima also have red dots. The dots aren't *right at the bottom of the minimum*, because they're *noisy samples*. |  |

* **The wonderful property of Markov Chain Monte Carlo (MCMC)**

If we add that *Gaussian noise* in just the right way, there's a wonderful property of *Markov Chain Monte Carlo:*

* It's an amazing fact— if we *wandered* around for *long* enough, the *weight vectors* will be *unbiased samples* from the *true posterior distribution over weight vectors*.
* i.e. if we use just the right amount of *noise*, and if we let the *weight vector* *wander* around for *long enough* before we take a sample, we will get an *unbiased sample* from the *true posterior* over weight vectors.
* That is, those *red dots* we saw in the previous sub-section will be *sampled* from the *posterior*, where *weight vectors* are a *highly* *probable* under the *posterior*, a much more *likely* to be represented by a *red dot* than weight vector that is highly improbable.
* This is called *Markov Chain Monte Carlo (MCMC)*.
* MCMC makes it *feasible* to use full *Bayesian learning* with *thousands of parameters*.
* There's a method of *adding* some *Gaussian noise* (by Hinton), but it's not the most efficient. There are more sophisticated methods related MCMC that are more complicated but more efficient.
* By "more efficient", we mean they *don't need* to let the weights *wander* around in the weight space *for so long* before we can *start taking* those *red samples* (samples from the posterior).
* Hinton has mentioned approaches that involve adding Gaussian noise to gradient updates, which are related to MCMC methods. One well-known technique in this area is *Langevin Dynamics* (or its variant, *Stochastic Gradient Langevin Dynamics (SGLD))*. These methods combine *gradient-based optimization* with the injection of *Gaussian noise* to sample from the posterior distribution, effectively blending learning with MCMC sampling.
* ***Gradient Descent with Noise:*** Instead of just moving "downhill" (toward lower cost), you add Gaussian noise to each gradient step. This noise helps *explore the "weight space" randomly*, avoiding getting stuck in bad local minima.
* ***MCMC Sampling:*** Over time, the noise ensures that the *weights* are *sampled proportionally* to their probability in the *posterior* *distribution* (Bayesian property).
* This mimics a *"smart random walk"* through *possible weight values*, guided by both *data (gradients)* and *uncertainty (noise)*.
* ***Why This Matters for Bayesian Learning:***
* It avoids *explicitly calculating* the impossibly *complex posterior* for large neural networks.
* By adding *noise* to gradients, you *approximate sampling* from the *posterior*, making Bayesian learning practical even for models with millions of weights.
* ***Key Papers/Concepts:***
* *Bayesian Learning via Stochastic Gradient Langevin Dynamics (SGLD) (Welling & Teh, 2011)* – Combines stochastic gradient descent with Langevin dynamics (noise injection).
* Hinton’s work on Boltzmann machines and stochastic neural networks also leverages similar ideas (e.g., Gibbs sampling with noise).
* **Full Bayesian learning with mini-batches**

*Full Bayesian learning* can actually be done with *mini batches*.

* When we compute the *gradient* of the *cost function* on a *random mini batch*, we will get an *unbiased estimate* but with *sampling noise*.
* And the idea is— to use that *sampling noise* to provide the *noise* that the *MCMC* method *needs*. It's a very clever idea.
* Welling and his collaborators [*Ahn, Korattikara &Welling— Bayesian Posterior Sampling via Stochastic Gradient Fisher Scoring (ICML 2012)*] showed how they could fairly efficiently get samples from the posterior distribution over weights using mini-batch methods.
* So *full Bayesian learning* is now possible with *lots of parameters*. This should make it possible to use *Full Bayesian Learning* for much *larger networks* where you have to train them with *mini-batch* to have any hope of ever finishing training them.

**10.5 Dropout**

In this section, we're going to describe a new way of *combining* a very large number of *neural network models* *without* having to *separately train* a very large number of models.

* This is a method called ***dropout*** that's recently been very successful in winning competitions.
* For each training case, we *randomly omit* some of the *hidden units*. So, we end up with a *different architecture* for *each training case*. We can think of this as having a *different model* for *every training case*.
* And then, the *question is*—
* How could we possibly *train a model* on *only one training case* and
* How could we *average* all these *models together* efficiently at *test time*?
* The *answer* is that— we use a great deal of *weight sharing*.
* **Two ways to average models**

Let's start by describing *two different ways* of *combining* the *outputs* of *multiple models*.

* ***MIXTURE:*** In a *mixture*, we combine models by *averaging* their *output probabilities*.
* So, if **model A** assigns probabilities of 0.3, 0.2 and 0.5, to three different answers, **model B** assigns probabilities of 0.1, 0.8 and 0.1, the **combined model** simply assigns the *averages of those probabilities*.

|  |  |  |  |
| --- | --- | --- | --- |
| **model A** | 0.3 | 0.2 | 0.5 |
| **model B** | 0.1 | 0.8 | 0.1 |
| **Combined** (average) | 0.2 | 0.5 | 0.3 |

* ***PRODUCT:*** A different way of combining models is to use a *product of the probabilities*. Here, we take a **geometric mean** of their *output probabilities*.
* So, model A and model B again assign the same probabilities as they did before.
* But now, what we do is— we *multiply* each *pair* of probabilities *together* and then take the *square root*. That's the *geometric mean*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **model A** | 0.3 | 0.2 | 0.5 |  |
| **model B** | 0.1 | 0.8 | 0.1 |  |
| **Combined** (average) |  |  |  | / sum |

* Now the *geometric means* will generally *add up* to *less than one*. So, we have to *divide* by the *sum of the geometric means* to *normalize* the *distribution* so that it adds up to 1 again.
* Notice that in a *product*, a *small probability* output by *one model*, has *veto power* over the *other models* (Eg: multiplying by 0.1 gives smaller product).

|  |  |
| --- | --- |
| * **Dropout: An efficient way to average many large neural nets**   ***Paper:*** *Improving neural networks by preventing co-adaptation of feature detectors*  *(https://arxiv.org/abs/1207.0580)*  Now let's describe an efficient way to *average a large number of neural nets* that gives us an *alternative* to doing the *correct Bayesian* thing. This *alternative* probably doesn't work *quite as well* as doing the *correct Bayesian* thing, but it's much more practical.   * So, let's consider a *neural net* with *one hidden layer*, shown on the right. |  |
| * Each time we present a *training example* to it— what we're going to do is *randomly omit* each *hidden unit* with a *probability* of *0.5*. Notice, we *crossed* out (red-cross) *three of the hidden units* here. * And then we *run* the *examples* through the net with those *hidden units absent*. * What this means is that we're *randomly sampling* from different architectures, where is the *number of hidden units*, it's a *huge number of architectures*. |  |

* Of course, all of these architectures *share* *weights* (i.e. whenever we use a *hidden unit*, it's got the *same weight* as it's got in *other architectures*).
* **Dropout as a form of model averaging**
* So, we can think of *dropout* as a *form of model averaging*. We sample from these models. *Most* of the models, in fact, will *never* be *sampled*.
* So only a *few* of the *models* ever get *trained*, and they only get *one training example*.
* And since a model that this sampled only gets one training example— that's a very *extreme* form of *bagging*. The *training sets* are very *different* for the *different models*, but they're *also very small*.
* The *sharing* of the *weights* between all the *models* means that each model is very *strongly regularized* by the *others*. And this is a much *better* *regularizer* than things like *L2* or *L1* *penalties* (these L1, L2 penalties pull the weights toward zero).
* By *sharing weights* with *other models*, a model gets *regularized* by something that's going to tend to *pull the weight* towards the *correct* *value*.
* Dropout doesn't directly pull each weight toward a specific "correct value". Instead, it forces the *weights* to *perform well* across many different *subnetworks* (since neurons are randomly dropped during training). This shared-weight setup encourages the model to learn *robust*, *generalizable features*, effectively regularizing the network and *reducing overfitting*.
* **But what do we do at test time?**

The question still remains— what we do with *test time*?

* We could *sample* *many different architectures* (maybe a hundred), and take the *geometric mean* of the *output distributions*.
* But that would be a lot of work.
* There’s something much simpler we can do, like the following:
* It better to *use all* of the *hidden units*, but we *halve* their *outgoing weights*. So, they have the same *expected effect* as they did when we were *sampling*.
* It turns out that *using all* of the *hidden units* with *half* their *outgoing weights*, exactly computes the *geometric mean* of the predictions that all models would have used, *provided* we're using a *softmax output group*.
* **What if we have more hidden layers?**
* If we have *more* than one *hidden layer*, we can simply use *dropout* at **0.5** in *every layer*.
* At *test time*, we *halve* all the *outgoing weights* of *hidden units*, and that gives us what we call the *"mean net"*.
* So, we use a net that has *all* of the *units* but the *weights* are *halved*.
* When we have *multiple hidden layers*, this is *not* exactly the *same* as *averaging* all the *separate dropped out models*, but it’s a pretty good approximation, and its fast.
* Alternatively, we could run lots of *stochastic models* with *dropout*, and then *average* across those stochastic models. And that would have one advantage over the ***mean net***—
* It would give us an idea of the *uncertainty* in the *answer*.
* **What about the input layer?**
* We can use the same trick there, too: We use *dropout* on the *inputs*, but we use a *higher probability* of *keeping* an input (unit).
* This trick is already used by the "Denoising Autoencoders" developed by *Pascal Vincent*, *Hugo Larochelle* and *Yoshua Bengio* at the University of Montreal, and it works very well.
* **How well does dropout work?**

Well, the record breaking object recognition net developed by Alex Krizhevsky (see lecture 5.4) *would* have broken the record even *without* *dropout*. But it broke a lot more by using dropout.

* In general, if you have a *deep neural net* and it's *overfitting*, then *dropout* will typically *reduce* the number *errors* by quite a lot.
* Any net that requires *early stopping* in order to *prevent* it *overfitting* would do *better* by using *dropout*.
* It would, of course, take *longer* to *train* and it might mean *more hidden units*.
* If you got a *Deep Neural Net* and it's *not overfitting*, you should probably be using a *bigger one* and *using dropout*, that's assuming you have *enough computational power*.
* **Another way to think about dropout**

There's *another way* to think about *dropout*, which is how Hinton originally arrived at the idea. And we'll see it's a bit related to *Mixture of Experts (MOE)*, and—

* What's going wrong when all the *experts cooperate*?
* What's *preventing specialization*?
* If a hidden unit knows— which *other* hidden units are *present*, it can *co-adapt* to the *other hidden units* on the *training* *data*.
* What that means is— the *real signal that's training a hidden unit* is, try to *fix* up the *error* that's *leftover* when all the *other hidden units* have had their *say*. That's what's being *back propagated* to *train the weights* of each hidden unit.
* Now, that's going to cause *complex co-adaptations* between the *hidden units*.
* But these *complex co-adaptations* are likely to go wrong when there's a *change* in the *data*. So, a *new test data*, If you rely on *a complex* *co-adaptation* to get things right on the training data, it's quite likely to not work nearly so well on *new test data*.
* It's like the idea that— a big, *complex conspiracy* involving lots of people is almost certain to go wrong because there's always things you didn't think of. And if there's a large number of people involved, one of them will behave in an unexpected way. And then, the others will be doing the wrong thing.
* It's much better if you want *conspiracies*, to have *lots of little conspiracies*. Then, when *unexpected* things happen, *many* of the little conspiracies will *fail*, but *some* of them will still *succeed*.
* So, by using dropout, we *force a hidden unit* to work with *combinatorially* many *other sets* of *hidden units*.
* If a *hidden unit* has to work well with *combinatorially* many sets of *co-workers*, it is more likely to do something that is *individually* *useful* rather than only useful because of the way *particular other hidden units* are collaborating with it.
* But it is also going to tend to do something that's *individually useful* and is *different* from what other hidden units (co-workers) do.
* It needs to do something that's *marginally useful*, given what its *co-workers tend to achieve*. And that's what's giving *nets with dropout*, their very *good performance*.