Neural Networks for Machine Learning

Lecture 10a
Why it helps to combine models

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Combining networks: The bias-variance trade-off

- When the amount of training data is limited, we get overfitting.
  - Averaging the predictions of many different models is a good way to reduce overfitting.
  - It helps most when the models make very different predictions.
- For regression, the squared error can be decomposed into a “bias” term and a “variance” term.
  - The bias term is big if the model has too little capacity to fit the data.
  - The variance term is big if the model has so much capacity that it is good at fitting the sampling error in each particular training set.
- By averaging away the variance we can use individual models with high capacity. These models have high variance but low bias.

How the combined predictor compares with the individual predictors

- 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 try to make the individual predictors disagree (without making them much worse individually).

Combining networks reduces variance

- We want to compare two expected squared errors: Pick a predictor at random versus use the average of all the predictors:

$$\bar{y} \;=\; \underbrace{< y_i >_i}_{\text{expectation}} \;=\; \frac{1}{N}\sum_{i=1}^{N} y_i$$

i is an index over the N models

$$< (t-y_i)^2 >_i = < \left((t-\bar{y})-(y_i-\bar{y})\right)^2 >_i$$

$$= < (t-\bar{y})^2 + (y_i-\bar{y})^2 - 2(t-\bar{y})(y_i-\bar{y}) >_i$$

this term vanishes

$$= \underbrace{(t-\bar{y})^2}_{\text{squared bias}} + \underbrace{< (y_i-\bar{y})^2 >_i}_{\text{variance}} \boxed{-2(t-\bar{y}) < (y_i-\bar{y}) >_i}$$

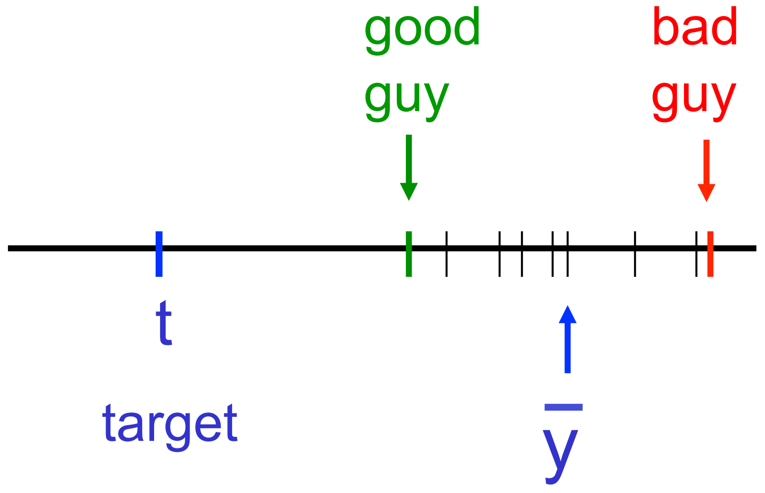
Result: the expected squared error by taking picking a model at random is greater than the squared error of averaging the models by the variance of the models.

A picture

- The predictors that are further than average from t make bigger than average squared errors.
- The predictors that are nearer than average to t make smaller then average squared errors.

- The first effect dominates because squares work like that.

- Don't try averaging if you want to synchronize a bunch of clocks!
  - The noise is not Gaussian.



$$\frac{(\bar{y}-\varepsilon)^2 + (\bar{y}+\varepsilon)^2}{2} = \bar{y}^2 + \varepsilon^2$$

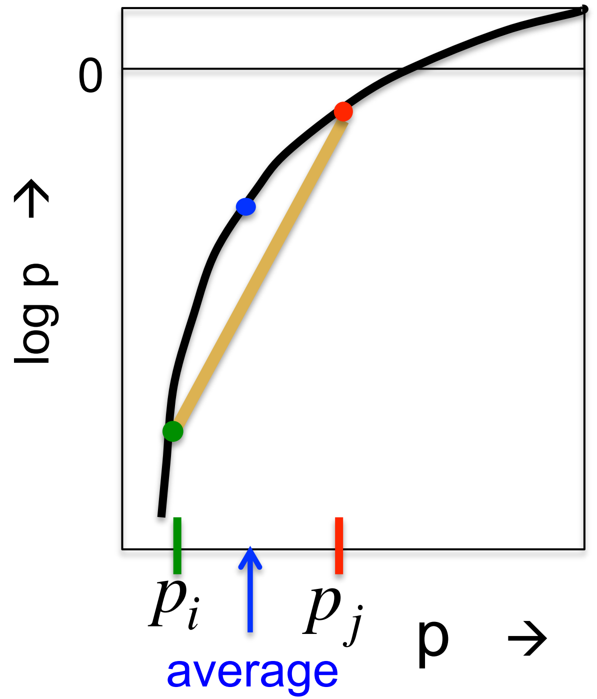
averaged squared error

Some are slightly wrong. A few might be wildly wrong, because they stop.

What about discrete distributions over class labels?

- Suppose that one model gives the correct label probability p_i and the other model gives it p_j
- Is it better to pick one model at random, or is it better to average the two probabilities?

$$\log\left(\frac{p_i + p_j}{2}\right) \geq \frac{\log p_i + \log p_j}{2}$$



Overview of ways to make predictors differ

- Rely on the learning algorithm getting stuck in different local optima.
  - A dubious hack (but worth a try).
- Use lots of different kinds of models, including ones that are not neural networks.
  - Decision trees
  - Gaussian Process models
  - Support Vector Machines
  - and many others.

- For neural network models, make them different by using:
  - Different numbers of hidden layers.
  - Different numbers of units per layer.
  - Different types of unit.
  - Different types or strengths of weight penalty.
  - Different learning algorithms. *full batch for some and mini-batch for others*

Making models differ by changing their training data

- Bagging: Train different models on different subsets of the data.
  - Bagging gets different training sets by using sampling with replacement:
    a,b,c,d,e → a c c d d
  - Random forests use lots of different decision trees trained using bagging. They work well.
- We could use bagging with neural nets but its very expensive.

- Boosting: Train a sequence of low capacity models. Weight the training cases differently for each model in the sequence.
  - Boosting up-weights cases that previous models got wrong.
  - An early use of boosting was with neural nets for MNIST.
  - It focused the computational resources on modeling the tricky cases.

Boosting up-weights the training cases that previous models got wrong and down-weights the cases that 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 trying deal with the cases the other models are getting wrong.

Neural Networks for Machine Learning

Lecture 10b
Mixtures of Experts

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

- developed in the early 1990s

- Idea: To train a number of neural nets each of which specializes in a different part of the data. We assume a data set which comes from a number of different regimes. And we train a system in which one neural net specializes in one regime. And a manager neural net will look into the input data and decide which specialist to give it to.

- Disadvantage: It doesn't make very efficient use of data. Because the data is fractionated over different experts. With small data set, it cannot be expected to do very well.

- Advantage: works well in extremely large data set.

Mixtures of Experts

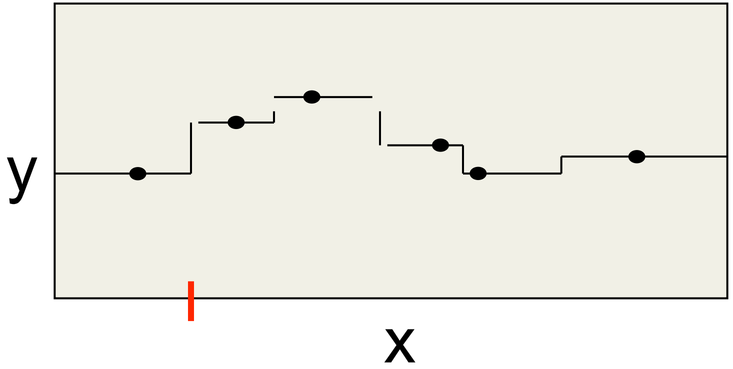
- Can we do better that just averaging models in a way that does not depend on the particular training case?
  - Maybe we can look at the input data for a particular case to help us decide which model to rely on.
  - This may allow particular models to specialize in a subset of the training cases.
  - They do not learn on cases for which they are not picked. So they can ignore stuff they are not good at modeling. Hurray for nerds!
- The key idea is to make each expert focus on predicting the right answer for the cases where it is already doing better than the other experts.
  - This causes specialization.

A spectrum of models

Very local models

- e.g. Nearest neighbors

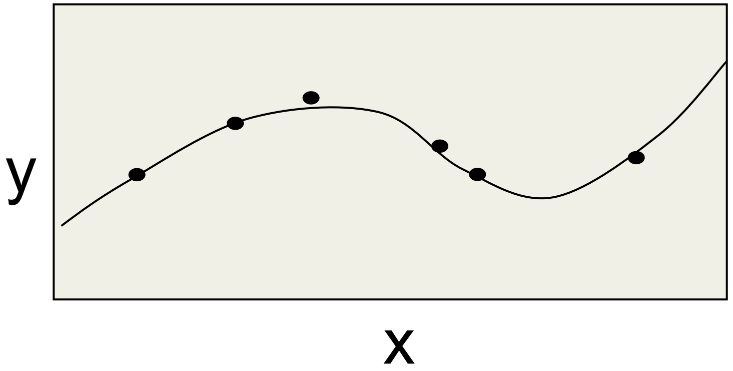
- Very fast to fit
  - Just store training cases
- Local smoothing would obviously improve things.



Fully global models

- e. g. A polynomial

- May be slow to fit and also unstable.
  - Each parameter depends on all the data. Small changes to data can cause big changes to the fit.



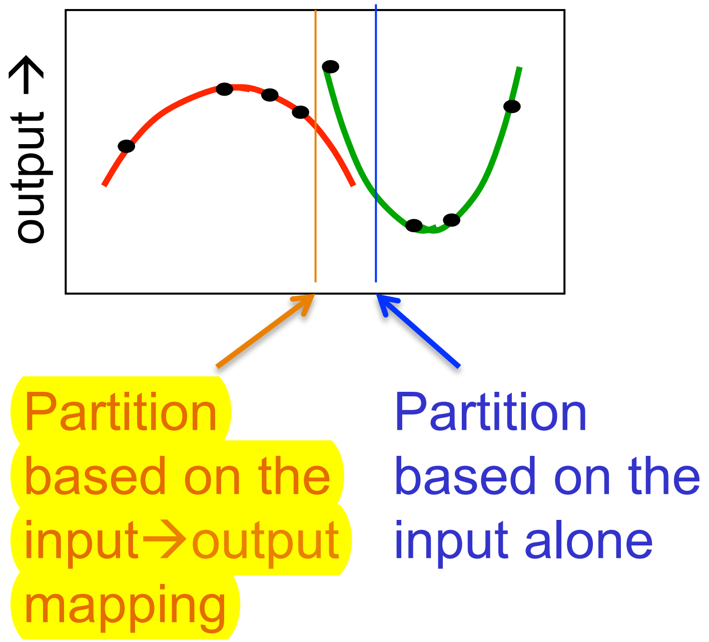
Multiple local models

- Instead of using a single global model or lots of very local models, use several models of intermediate complexity.
  - Good if the dataset contains several different regimes which have different relationships between input and output.
    - e.g. financial data which depends on the state of the economy.
- But how do we partition the dataset into regimes?

Question

Partitioning based on input alone versus partitioning based on the input-output relationship

- We need to cluster the training cases into subsets, one for each local model.
  - The aim of the clustering is NOT to find clusters of similar input vectors.
  - We want each cluster to have a relationship between input and output that can be well-modeled by one local model.


output →
Partition based on the input→output mapping
Partition based on the input alone

An error function that encourages cooperation

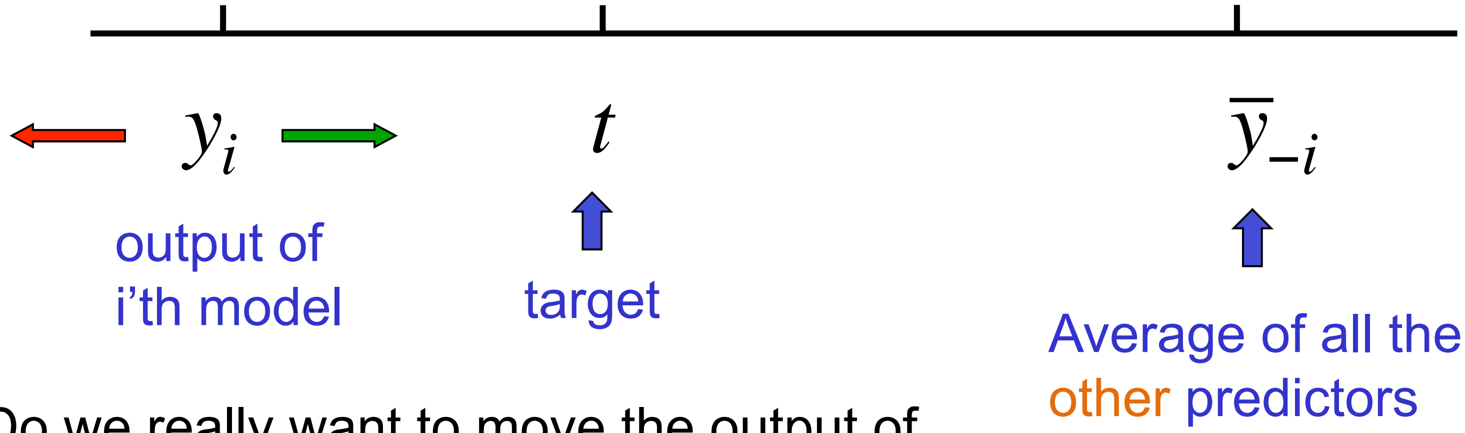
- If we want to encourage cooperation, we compare the average of all the predictors with the target and train to reduce the discrepancy.
  - This can overfit badly. It makes the model much more powerful than training each predictor separately.

Average of all the predictors

$$E = \left(t - \langle y_i \rangle_i\right)^2$$

The model will learn to fix up the errors made by other models.

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



Do we really want to move the output of model i away from the target value?

- Model i is learned to compensate for the errors made by all the other models.
- But do we really want to move the model in the wrong direction (red direction)??
- Intuitively, it seems better to move the model towards the target (green direction).

An error function that encourages specialization

- If we want to encourage specialization we compare each predictor separately with the target.
- We also use a "manager" to determine the probability of picking each expert.
  - Most experts end up ignoring most targets

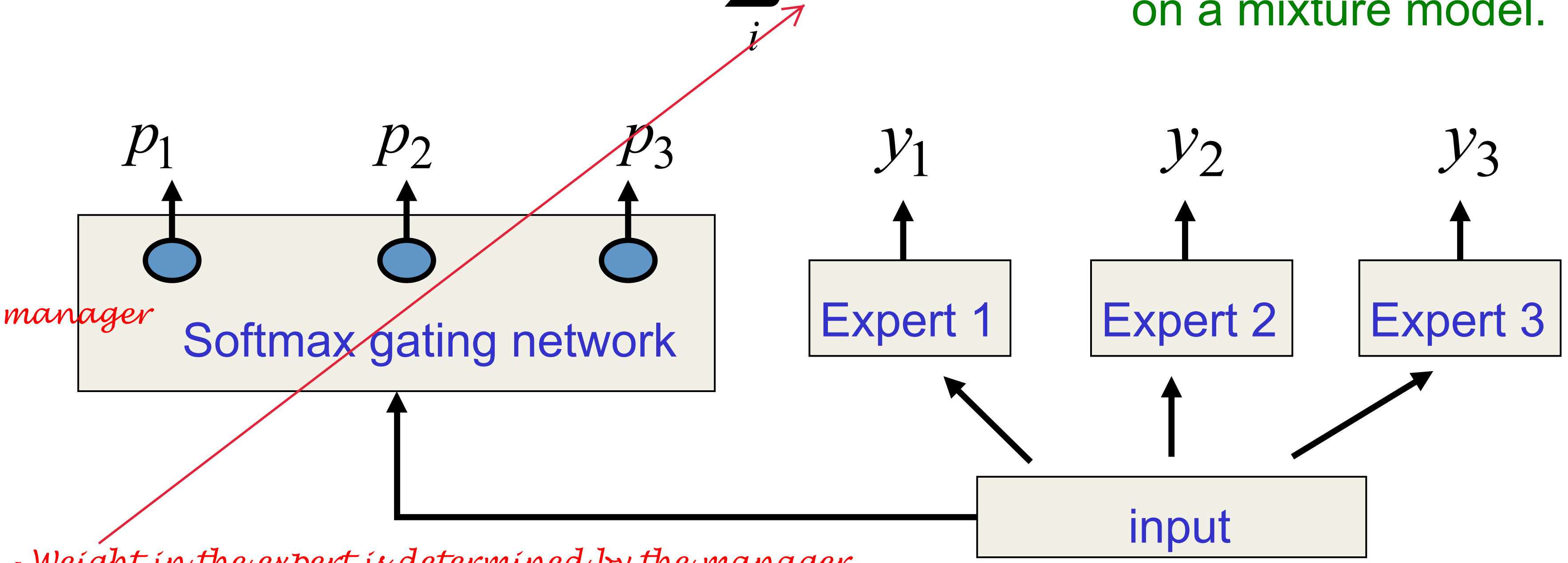
probability of the manager picking expert i for this case

$$E = \Big\langle \overset{\downarrow}{p_i} (t - y_i)^2 \Big\rangle_i$$

The mixture of experts architecture (almost)

A simple cost function : $E = \sum_i p_i (t - y_i)^2$

There is a better cost function based on a mixture model.



- Weight in the expert is determined by the manager.

The derivatives of the simple cost function

- If we differentiate w.r.t. the outputs of the experts we get a signal for training each expert.
- If we differentiate w.r.t. the outputs of the gating network we get a signal for training the gating net.
  - We want to raise p for all experts that give less than the average squared error of all the experts (weighted by p)

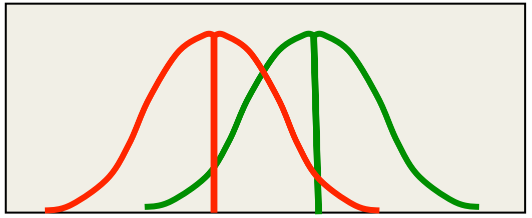
$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \qquad E = \sum_i p_i (t - y_i)^2,$$

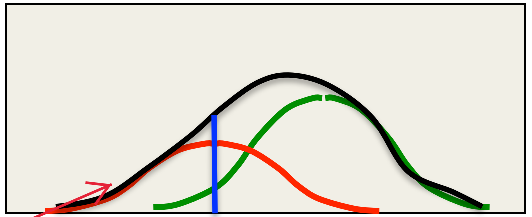
$$\frac{\partial E}{\partial y_i} = p_i (t - y_i)$$

$$\frac{\partial E}{\partial x_i} = p_i \left((t - y_i)^2 - E \right)$$

A better cost function for mixtures of experts (Jacobs, Jordan, Nowlan & Hinton, 1991)

- Think of each expert as making a prediction that is a Gaussian distribution around its output (with variance 1).
- 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}
- Maximize the log probability of the target value under this mixture of Gaussians model i.e. the sum of the two scaled Gaussians.





The black curve is the sum of the red curve and green curve.

The probability of the target under a mixture of Gaussians

mixing proportion assigned to expert i for case c by the gating network

$$p(t^c \mid MoE) = \sum_i p_i^c \, \frac{1}{\sqrt{2\pi}} \, e^{-\frac{1}{2}(t^c - y_i^c)^2}$$

prob. of target value on case c given the mixture.

normalization term for a Gaussian with $\sigma^2 = 1$

output of expert i

Neural Networks for Machine Learning

Lecture 10c
The idea of full Bayesian learning

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Frequentists' MLE: find the best single setting of the parameters.
Bayesians' MAP: compute full posterior distribution over all possible parameter settings.

Full Bayesian Learning

- Instead of trying to find the best single setting of the parameters (as in Maximum Likelihood or MAP) compute the full posterior distribution over all possible parameter settings.
  - This is extremely computationally intensive for all but the simplest models (its feasible for a biased coin). disadvantage
- To make predictions, let each different setting of the parameters make its own prediction and then 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 full Bayesian approach allows us to use complicated models even when we do not have much data. advantage

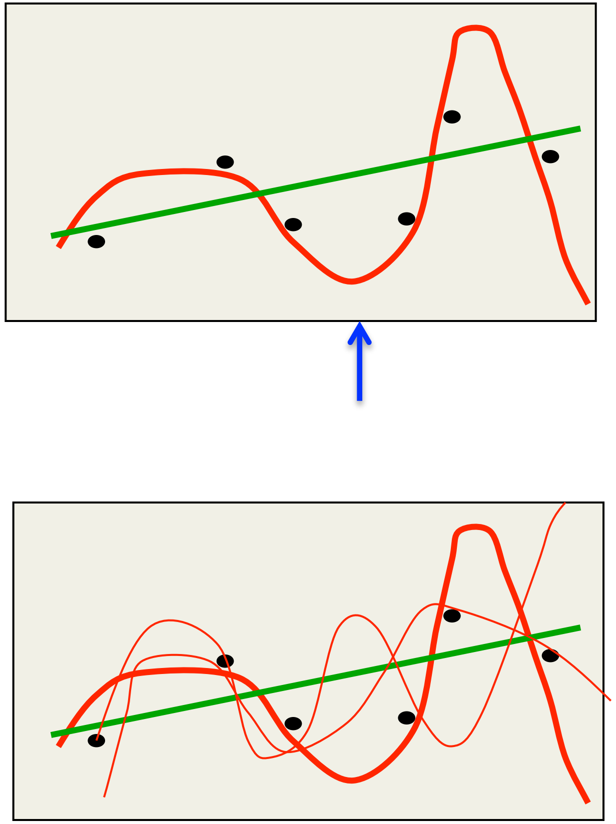
Overfitting: A frequentist illusion?

- If you do not have much data, you should use a simple model, because a complex one will overfit. Frequentist's view
  - This is true.
  - But only if you assume that fitting a model means choosing a single best setting of the parameters.

- If you use the full posterior distribution over parameter settings, overfitting disappears.
  - When there is very little data, you get very vague predictions because many different parameters settings have significant posterior probability.

A classic example of overfitting

- Which model do you believe?
  - The complicated model fits the data better.
  - But it is not economical and it makes silly predictions.
- But what if we start with a reasonable prior over all fifth-order polynomials and use the full posterior distribution.
  - Now we get vague and sensible predictions.
- There is no reason why the amount of data should influence our prior beliefs about the complexity of the model.



Approximating full Bayesian learning in a neural net

- If the neural net only has a few parameters we could put a grid over the parameter space and evaluate p(W | D) at each grid-point.
  - This is expensive, but it does not involve any gradient descent and there are no local optimum issues.
- After evaluating each grid point we use all of them to make predictions on 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).

$$p(t_{test} \mid input_{test}) = \sum_{g \,\varepsilon\, grid} p(W_g \mid D)\ p(t_{test} \mid input_{test}, W_g)$$

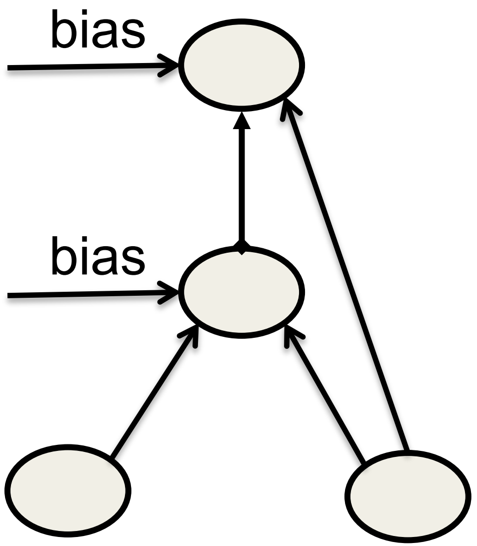
Advantage: no gradient descent, no local optimum issues
Disadvantage: computationally expensive

An example of full Bayesian learning

- Allow each of the 6 weights or biases to have the 9 possible values -2, -1.5, -1, -0.5, 0 ,0.5, 1, 1.5, 2
  - There are 9^6 grid-points in parameter space
- For each grid-point compute the probability of the observed outputs of all the training cases. $p(D|w)$
- Multiply the prior for each grid-point by the likelihood term and renormalize to get the posterior probability for each grid-point.
- Make predictions by using the posterior probabilities to average the predictions made by the different grid-points.

posterior probability for grid point w

$$\boxed{p(w \mid D) = \frac{p(w) \cdot p(D \mid w)}{\sum_{w \in W} p(w) \cdot p(D \mid w)}}$$



A neural net with 2 inputs, 1 output and 6 parameters

Neural Networks for Machine Learning

Lecture 10d
Making full Bayesian learning practical

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Challenge: *Full Bayesian Neural Networks have thousands perhaps millions of weights.*

Solution: *Use Monte Carlo method.*
(1) Use random number generator to move around the space of weight vectors in a random way but with a bias towards going downhill in our cost function.
(2) Sample weight vectors in proportion to their probabilities in the posterior distribution. We get a good approximation to the full Bayesian method.

What can we do if there are too many parameters for a grid?

Challenge

- The number of grid points is exponential in the number of parameters.
  - So we cannot deal with more than a few parameters using a grid.
- If there is enough data to make most parameter vectors very unlikely, only a tiny fraction of the grid points make a significant contribution to the predictions.
  - Maybe we can just evaluate this tiny fraction
- Idea: It might be good enough to just sample weight vectors according to their posterior probabilities.

$$p(y_{test} \mid input_{test}, D) = \sum_i p(W_i \mid D)\; p(y_{test} \mid input_{test}, W_i)$$

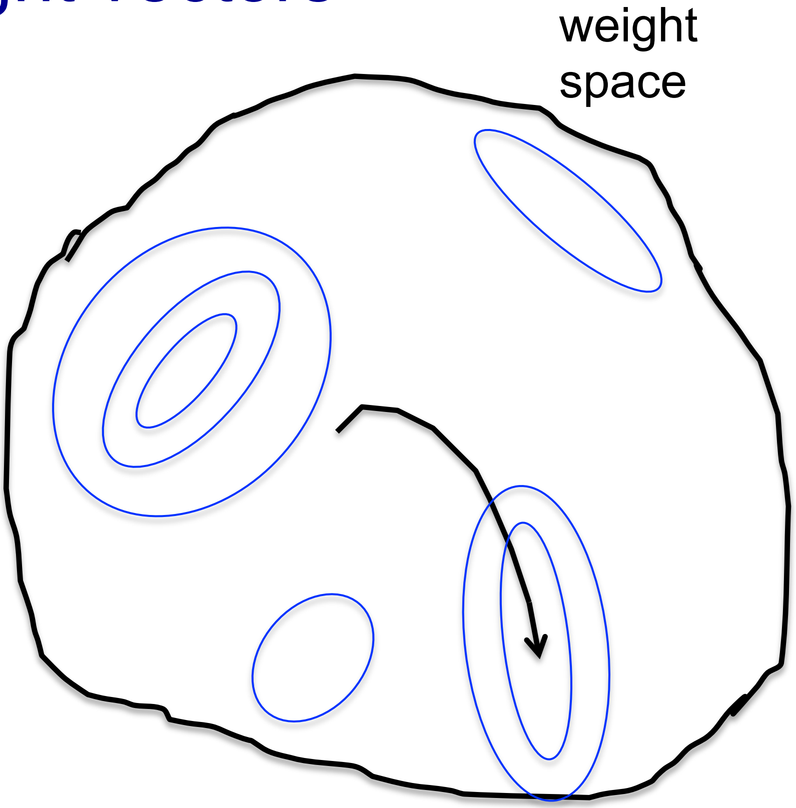
Sample weight vectors with this probability

We will get a weight p(Wi|D) of either 1 or 0. 1 means being sampled, 0 otherwise.

Sampling weight vectors

idea of backpropagation

- In standard backpropagation we keep moving the weights in the direction that decreases the cost.
  - i.e. the direction that increases the log likelihood plus the log prior, summed over all training cases.
  - Eventually, the weights settle into a local minimum or get stuck on a plateau or just move so slowly that we run out of patience.

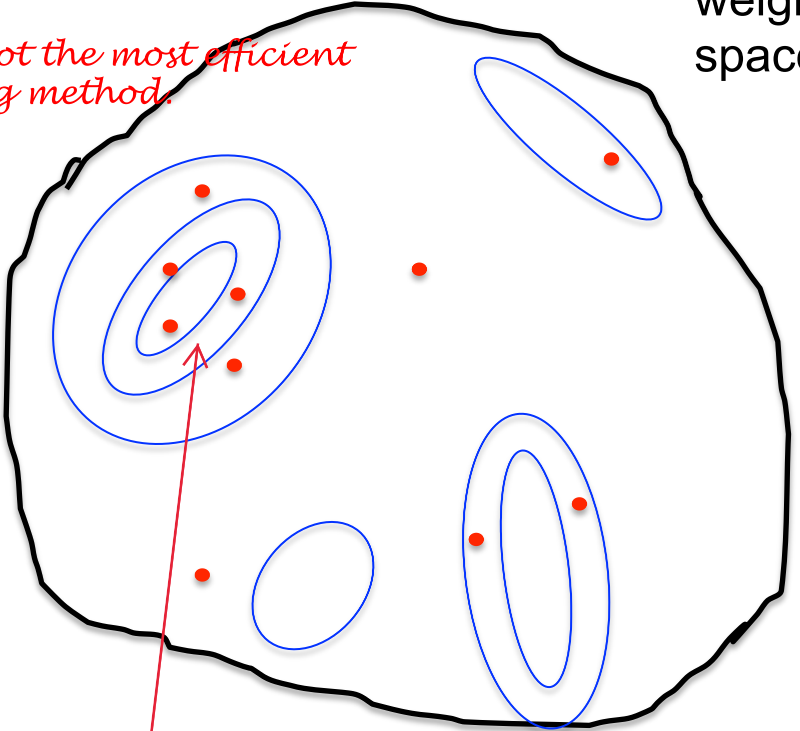


One method for sampling weight vectors

idea of sampling weight vectors

This is not the most efficient sampling method.

- Suppose we add some Gaussian noise to the weight vector after each update.
  - So the weight vector never settles down.
  - It keeps wandering around, but it tends to prefer low cost regions of the weight space.
  - Can we say anything about how often it will visit each possible setting of the weights?



weight space

Save the weights after every 10,000 steps.

The deepest minimum have the most red dots.

The wonderful property of Markov Chain Monte Carlo

- Amazing fact: 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.
  - This is called a "Markov Chain Monte Carlo" method.
  - MCMC makes it feasible to use full Bayesian learning with thousands of parameters.
- There are related MCMC methods that are more complicated but more efficient:
  - We don't need to let the weights wander around for so long before we get samples from the posterior.

Definition: Markov Chain Monte Carlo [Wikipedia]
In statistics, Markov chain Monte Carlo (MCMC) methods are a class of algorithms for sampling from a probability distribution based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. The state of the chain after a number of steps is then used as a sample of the desired distribution. The quality of the sample improves as a function of the number of steps.

Full Bayesian learning with mini-batches

- If we compute the gradient of the cost function on a random mini-batch we will get an unbiased estimate with sampling noise.
  - Maybe we can use the sampling noise to provide the noise that an MCMC method needs!

- Ahn, Korattikara &Welling (ICML 2012) showed how to do this fairly efficiently.
  - So full Bayesian learning is now possible with lots of parameters.

Neural Networks for Machine Learning

Lecture 10e
Dropout: an efficient way to combine neural nets

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Challenge: *Training a large number of neural networks is computationally expensive.*

Solution: *Use dropout.*
For each training case, we randomly omit some of the hidden units. We end up a different architecture for each training case. We can think of it as having a different model for each training case.

How could we train a model with only one training case?
How could we average all these models efficiently at test time?
Use a great deal of weight sharing.

Two ways to average models

Mixture: Arithmetic Mean

- MIXTURE: We can combine models by averaging their output probabilities:

Model A:	.3	.2	.5
Model B:	.1	.8	.1
Combined	.2	.5	.3

Product: Geometric Mean

- PRODUCT: We can combine models by taking the geometric means of their output probabilities:

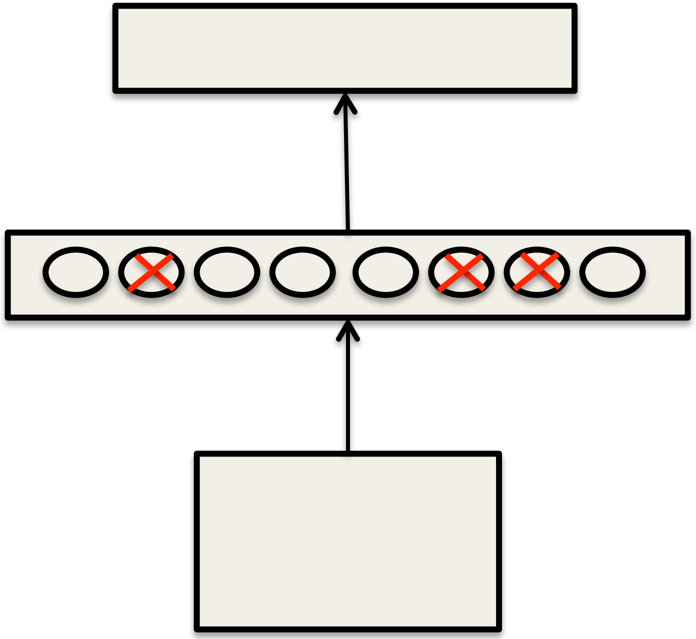
Model A:	.3	.2	.5	
Model B:	.1	.8	.1	
Combined	$\sqrt{.03}$	$\sqrt{.16}$	$\sqrt{.05}$	/sum

Note:
(1) Geometric means will generally add up to less than 1. We have to normalize the distribution afterwards.
(2) In product method, a small probability output has the veto power over the other models.

Dropout: An efficient way to average many large neural nets (http://arxiv.org/abs/1207.0580)

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
  - All architectures share weights.

H = number of hidden units



Dropout as a form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
  - This is as extreme as bagging can get.
- The sharing of the weights means that every model is very strongly regularized.
  - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.

But what do we do at test time?

bad method

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
  - This exactly computes the geometric mean of the predictions of all 2^H models.

good method

What if we have more hidden layers?

- Use dropout of 0.5 in every layer.
- At test time, use the “mean net” that has all the outgoing weights halved.
  - 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, run the stochastic model several times on the same input.
  - This gives us an idea of the uncertainty in the answer.

What about the input layer?

- It helps to use dropout there too, but with 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.

How well does dropout work?

- The record breaking object recognition net developed by Alex Krizhevsky (see lecture 5) uses dropout and it helps a lot.
- If your deep neural net is significantly overfitting, dropout will usually reduce the number of errors by a lot.
  - Any net that uses "early stopping" can do better by using dropout (at the cost of taking quite a lot longer to train).
- If your deep neural net is not overfitting you should be using a bigger one!

Another way to think about dropout

- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
  - But complex co-adaptations are likely to go wrong on new test data.
  - Big, complex conspiracies are not robust.
- 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.
  - But it will also tend to do something that is marginally useful given what its co-workers achieve.