# Dimensions

You may be given the inputs and weights in conforming dimensions, but they may not be organized properly for matrix multiplication. Although the formula for output neuron is: W' \* X, this may not work when you code. Play around by transposing the vectors so that they conform to the multiplication rule. You may need to transpose your final output.

# Logistic/Linear? - binary classification

If the target is 1 and the prediction is 100, the logistic unit will squash this down to a number very close to 1 and so we will not incur a very high cost. With a linear unit, the difference between the prediction and target will be very large and we will incur a high cost as a result, even though we get the classification decision correct.

Answer: Logistic (please verify)

Things to consider: How many functions does each of the following networks learn?

inputs: 10 , hidden units: 0 -> 10 functions

inputs: 10 , hidden units: 10 -> 10 functions

inputs: 10 , hidden units: 1 -> 1 function

inputs: 10 , hidden units: 11 -> 11 functions

In theory, a network with no hidden units may perform better than a network with lesser number of hidden units that the number of inputs.

# Why logistic hidden units, not linear?

With linear hidden units, the output would become a linear function of the inputs, and we typically want to learn nonlinear functions of the input. Therefore, **linear hidden units are a bad choice**.

# ‘Number of hidden units’ dilemma

**Question**. A network with m > n can learn functions that a network with m ≤ n cannot learn. True/False?

Hint: Linear hidden units don't add modeling capacity to the network.

# Averaging the weights to two NN model with same inputs and structures

You can use 1 Feed Forward Network by averaging the weights if it is a linear unit.

You cannot average if it is a logistic hidden unit.

**"bottleneck" layers** - layers with fewer dimensions than the input

**Backpropagation** - rate of change in error (target vs prediction) wrt change in weight

# How to train a n/w for accuracy with equivariance and invariance (L5c)

Huge training set: original + synthetic data

Avoid overfitting: train a dumb n/w on the huge dataset

Consensus model: train different models and vote on the differences

# McNemar test

Calculate significance by comparing the answers that one model got right but the other got wrong

# Full-batch, mini-batch or online learning?

If the dataset is highly redundant, then there is no point in doing a full batch learning because the gradient of the first half of the dataset will be the same as the gradient of its second half. On the other hand, online learning is the other extreme. That is why we stick to mini-batch learning in such cases.

# Shift input value (zero mean)

When using steepest descent, shifting the input values can make a big difference.

Transform each component of the input vector first so that it has zero mean over the entire training set.

- For the hidden untis, use the hyperbolic tangent (2\*logistic - 1) to roughly get the same effect. This makes training faster.

- Do this only if the inputs to the hyperbolic tangent are distributed around zero.

# Common problems in ANN

- If the error stops decreasing, this does not necessarily mean you are stuck in a local minima. You may be stuck at the very end of a plateau if you started with a very big learning rate.

- In classification using squared error or cross entropy, do not make each output using produce an output equal to the proportion of time it should be a 1. This is common. The n/w guesses this quickly and gets stuck on a plateau for a long time.

# How to understand if the n/w is stuck on a plateau

If the error rate starts decreasing fairly rapidly and all of a sudden it's decrease slows down.

# Speed up mini-batch learning

- Use momentum

- Separate adaptive learning rate for each parameter

-- slowly adjust the rate based on empirical measurements. Empirical here means to observe if we are making any progress, or if the gradient is oscillating (it's sign keeps changing)

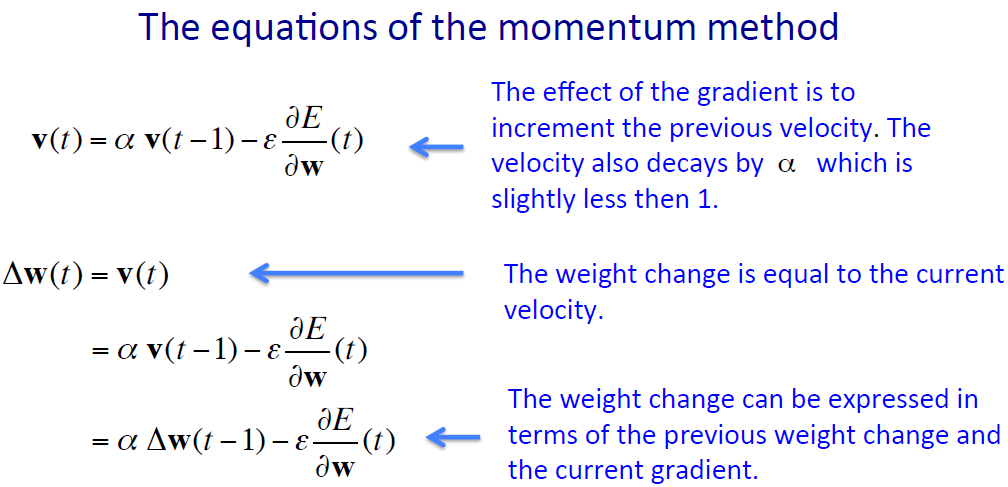
--- If gradient oscillates, reduce the rate. Otherwise, increase it.

- RMSprop

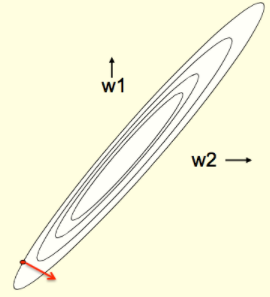
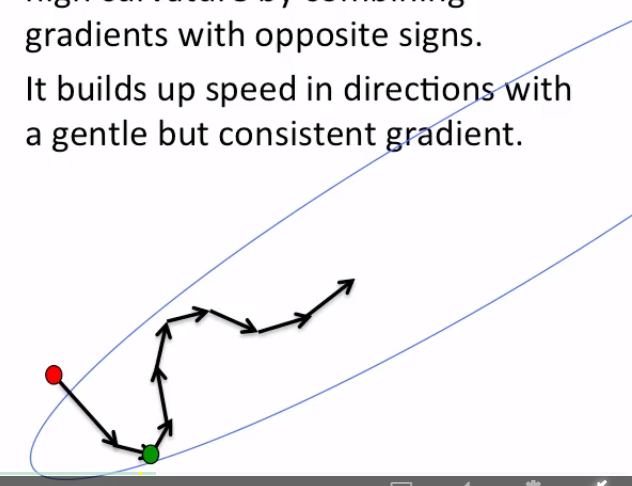
# Momentum

Attenuates or dampens the velocity of the gradient and moves it along the curvature of the error surface. It does so by adding a negative velocity against the direction of the steepest descent across the gradient and then adding the velocity along the curvature. The velocity is dampened using a learning rate which is analogous to viscosity.

Rule of thumb: 0.5



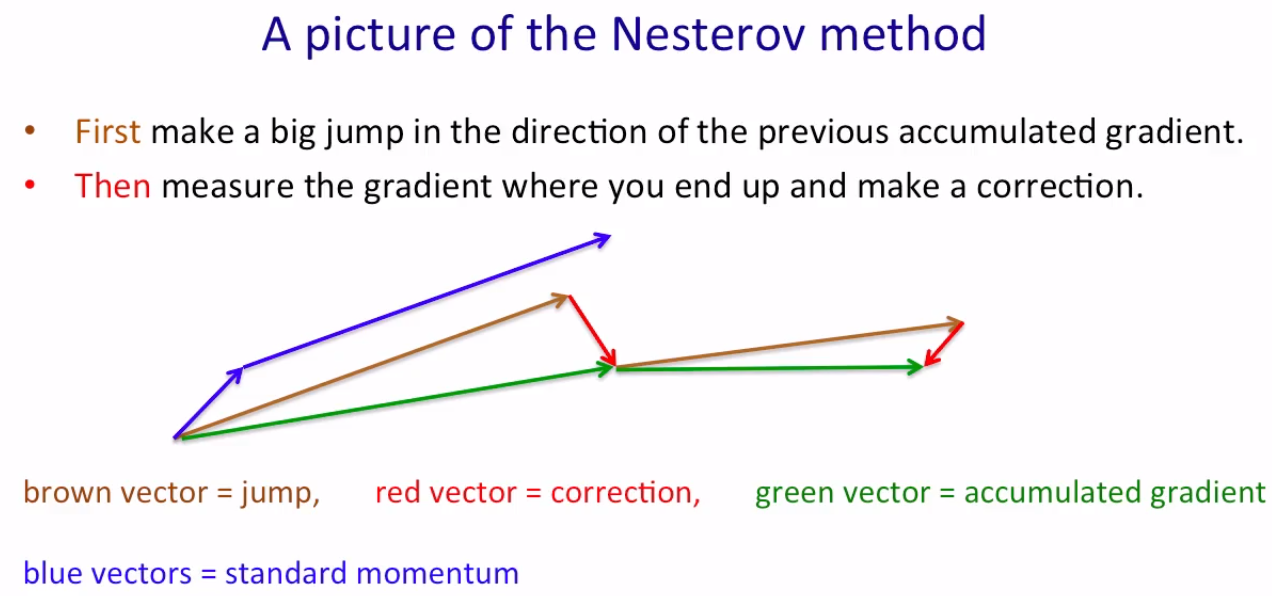
**Figure 1: Equations for momentum** [source: NN for Machine learning by Geoffrey Hinton]

**Figure 2: Steepest descent towards the minima Figure 3: How momentum improves the descent**

More in <https://math.stackexchange.com/questions/2256925/steepest-descent-in-elliptical-error-surface>

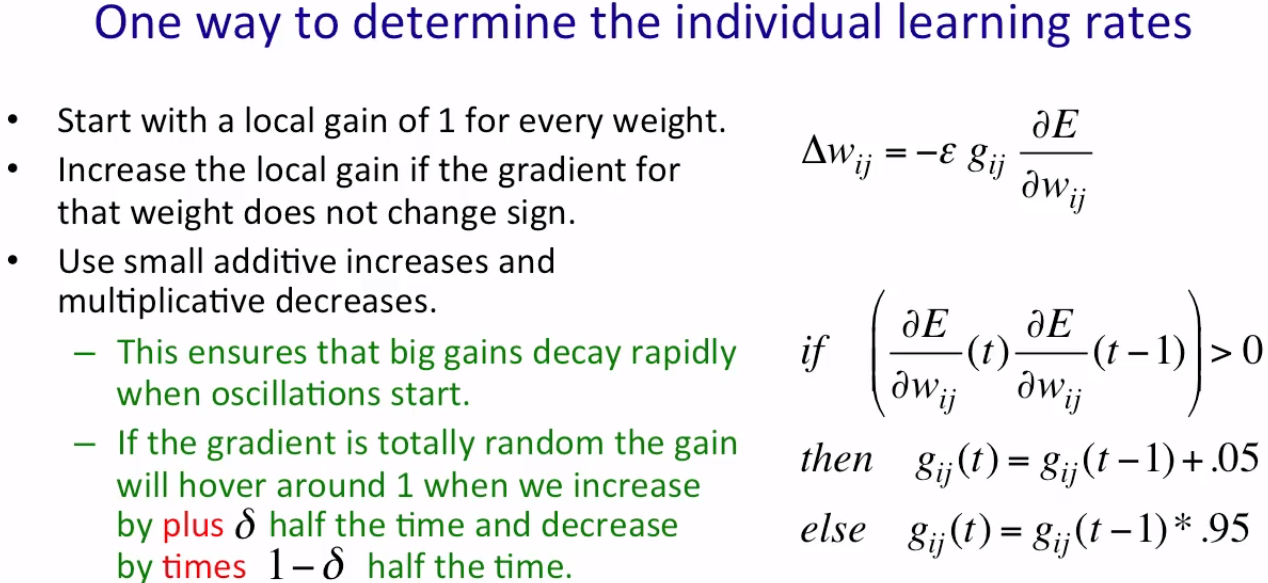
**A better way**: Make a gamble and then make a correction, not the other way around.



Using momentum speeds up learning because

* Directions of consistent change get amplified
* Directions of fluctuations get damped
* Allows using much larger learning rates because learning becomes more stable

# Adaptive learning rate of each connection



**Figure 2: How to calculate the adaptive learning rate by calculating the weight at each point in time**

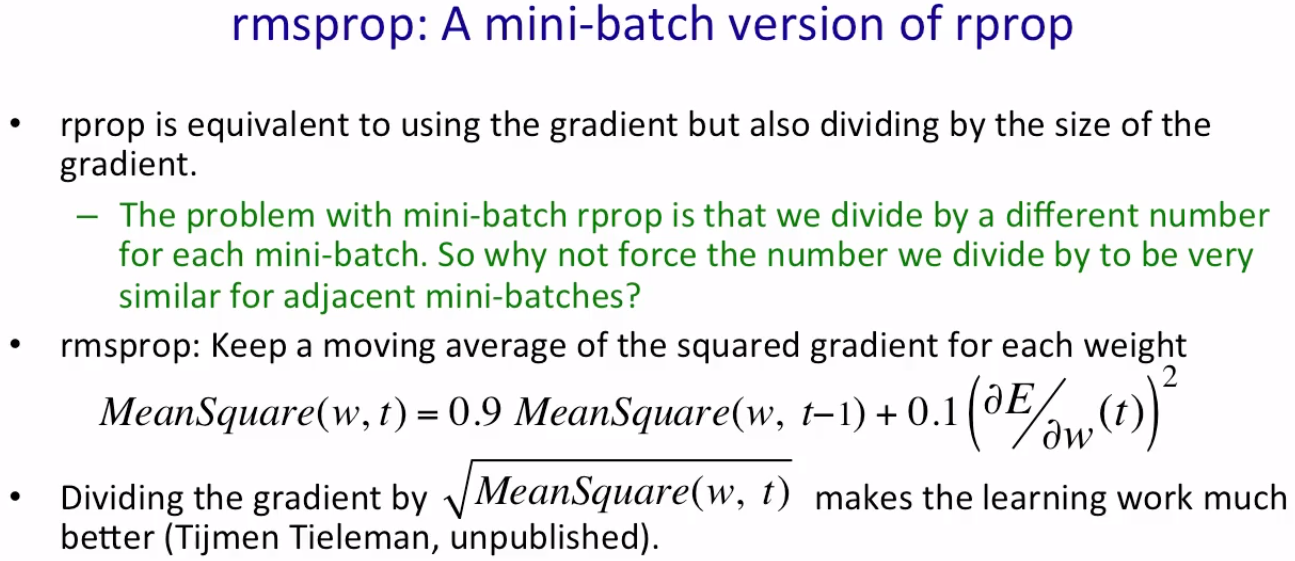
Reasonable range for gains: [0.1, 10], [.01, 100]

Application: full-batch learning, mini-batch learning with large batches

\* You can use this with momentum (Jacobs, 1998)

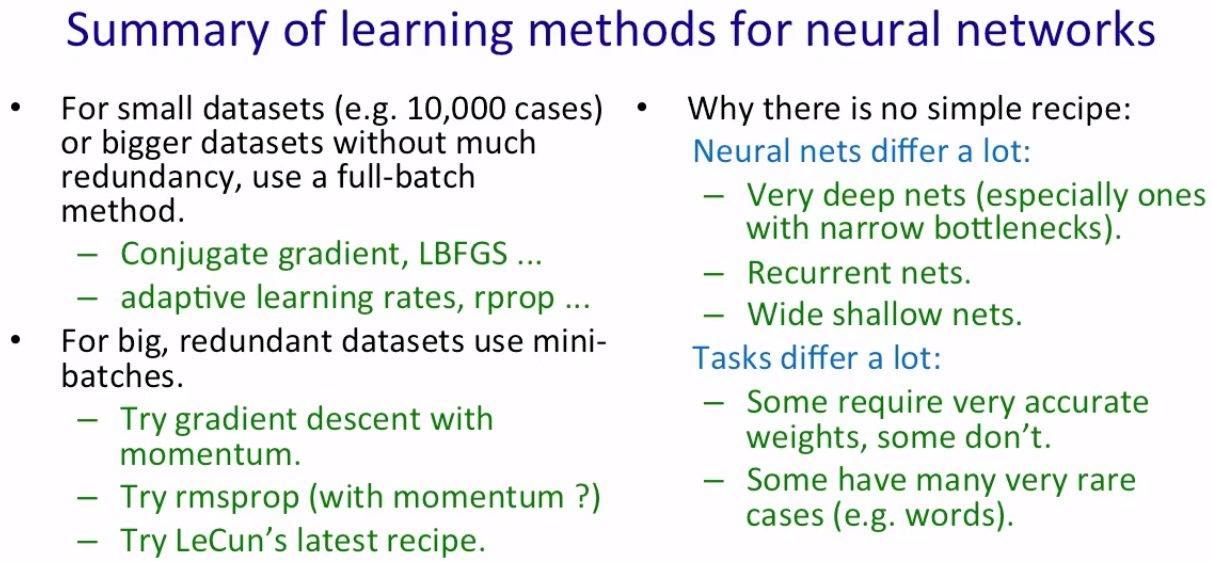
# rmsprop

**Divide the gradient by a running average of its recent magnitude**



Further enhancements

* Combine rmsprop with Nesterov momentum
  + Works best if the rms of the recent gradient is used to divide the correction rather than making a jump in the direction of accumulated corrections



# An approach to developing a Neural Network model

1. **Check for correlated features**
2. **Error surface**:
3. Plot the error surface.
4. Check if it is a very elongated ellipse

- elongated surface: very short minor axis, very long major axis

1. Check the position of the error surface relative to the origin. Even if it is far away from the origin, if the weight vector moves along the axis from which it is closer, gradient descent will converge properly.
2. Similarly, even if the error surface is elongated, if the weight vector moves along the major axis, the algorithm should still converge quite easily. Only if the surface is an elongated ellipse and the weight vector moves along the minor axis, gradient descent will converge rather poorly.
3. **Decide how to update the weights - optimization**
4. how often: online, full batch, mini batch online learning -> performance does not degrade with redundant data
5. how much (learning rate): fixed, global/adaptive
6. **Plan how to avoid overfitting if the network is large - generalization**
7. linear model is great, but it is not feasible all the time
8. polynomial model is the other alternative. But the degree of polynomials and their coefficients should be small enough

**Examples**: weight decay, weight sharing, early stopping, model averaging, Bayesian fitting, dropout, generative pre-training

1. **Eliminate dimension hopping**

Dimension hopping occurs when one can take the information contained in the dimensions of some input, and move this between dimensions while not changing the target. The canonical example is taking an image of a handwritten digit and translating it within the image. The dimensions that contain "ink" are now different (they have been moved to other dimensions), however the label we assign to the digit has not changed. Note that this is not something that happens consistently across the dataset, that is we may have a dataset containing two handwritten digits where one is a translated version of the other, however this still does not change the corresponding label of the digits.

* Information (input) should not jump from one dimension to another i.e.
  + If one neuron stores weights while another stores age of a patient, these neurons should not start swapping variables or using different variables as inputs in the next iteration.

1. **Determine if you should go for full/mini-batch/online learning**
2. If the dataset has no redundancy, then full batch works well because it gives a better estimate of the gradient.
3. If the dataset is highly redundant, then mini-batch learning is good.
4. Online learning: ?
5. **Eliminate class imbalance in mini-batches**

Randomize the dataset so that every mini batch has an equal representation of the classes.

1. **How to guess the learning rate (alpha) in mini-batch learning**

* Start with a random alpha.
* If the error gets worse or oscillates wildly, reduce alpha.
* If the error is falling consistently but very slowly, increase alpha.
* Towards the end of learning, turn down alpha (or if the error stops decreasing). Test this on a validation set and see if the same happens.
  + once this starts working, write a simple program (loop) to automate this.
  + Don't turn down the learning rate too soon or too much because this slows down the learning process.

1. **Scale input values (unit variance) & shift input values (zero mean)**

* Provides zero mean and unit variance for the training set.

1. **Weight initialization**
2. If a hidden unit has a big fan-in, use small weight.
3. If a hidden unit has a small fan-in, use big weight.

Use a learning rate for the weights ??

Rule of thumb:

Scale the learning rate in the same way

1. **PCA**

* Drop the PCs with the smallest
* Divide the remaining principal components by the square roots of their eigenvalues. For a linear neuron, this converts an axis aligned elliptical error surface into a circular one.

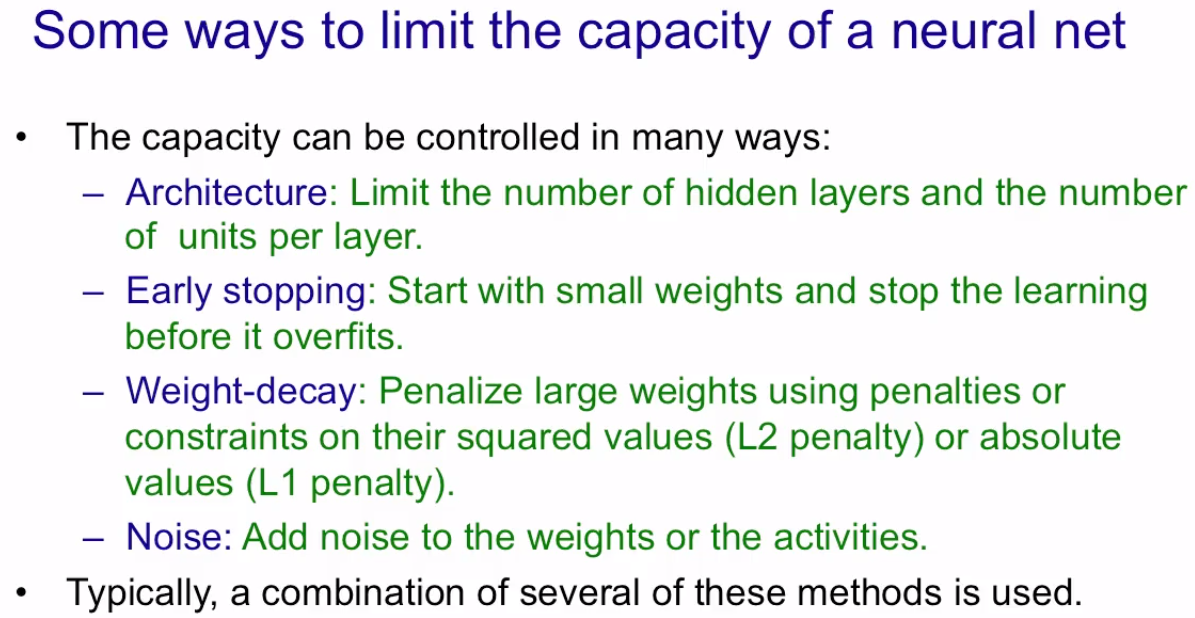
# \*\*\*How to generalize well\*\*\*

**Background**: Training data, especially when sampled e.g. in mini batches, contain sampling errors. This causes accidental irregularities in the dataset. On the contrary, a learning algorithm looks for regularities in the dataset. As a result, it cannot tell which regularities are real and which are caused by the sampling error.

Prevention methods:

1. Get more data
2. Use a model with the right capacity
   * means enough capacity to fit the true regularities but not enough to fit spurious regularities (the weaker ones)
3. Average different models
   * Use models that make different mistakes over the same training data *OR*
   * **Bagging**: train different models on different subsets of the training data
4. Bayesian approach
   * Use different sets of weights that do a good job of predicting the output using the same network model. On test data, average the prediction made by all those sets.

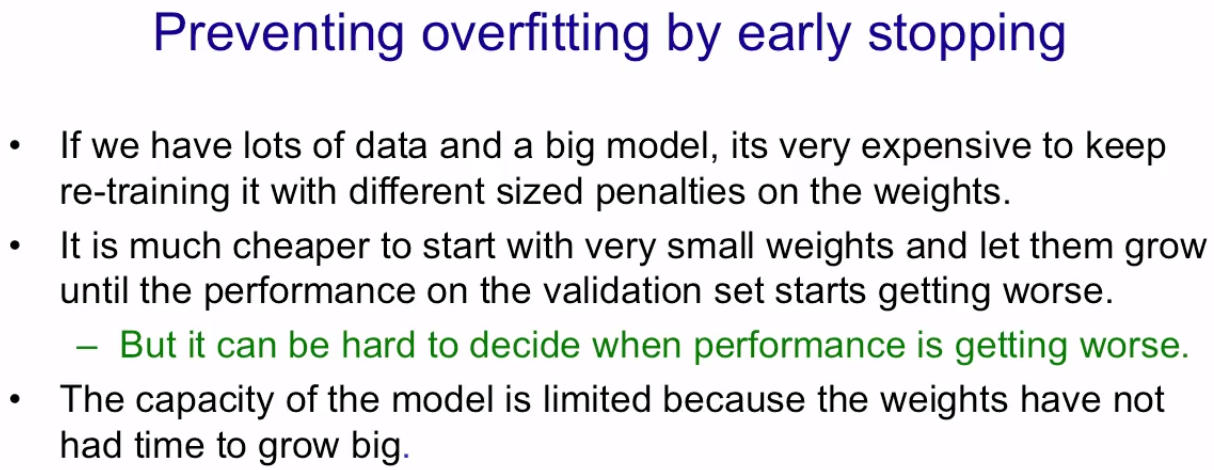
## Use a model with the right capacity



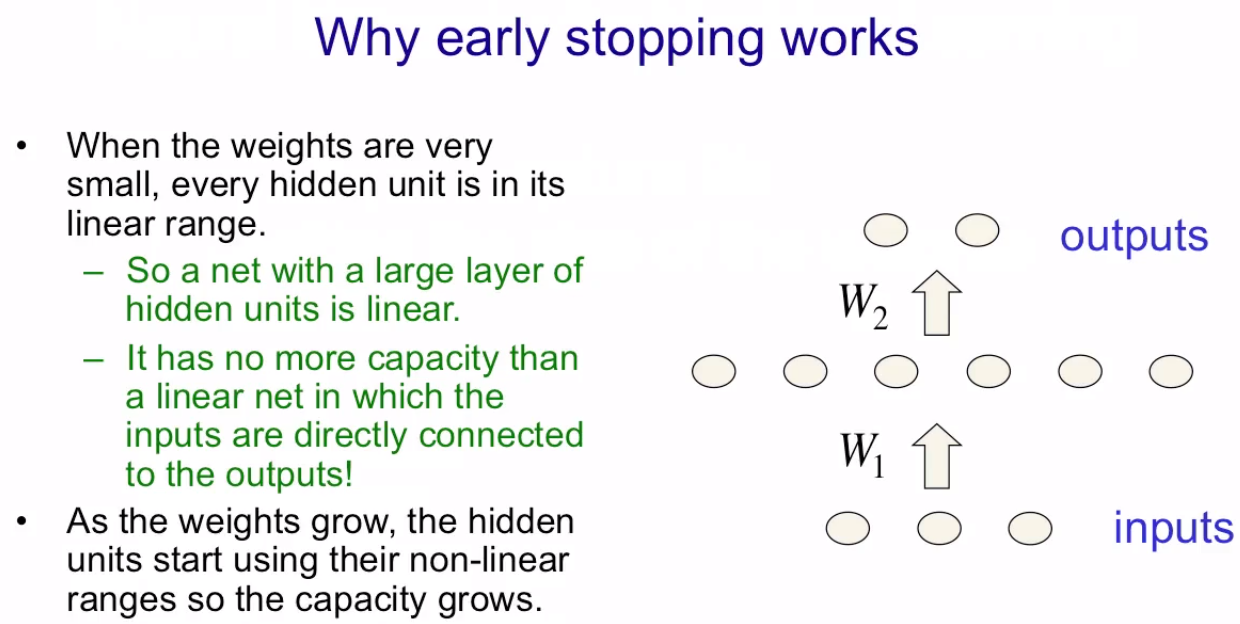
### Use cross-validation

* Be cautious that if you do too well on the validation data, there is a chance that you overfitted your model on this data and it would not generalize well on test data.
* You can use **N-fold cross validation** for better validation of the model, but there is also this chance of getting class imbalance on one or more of the validation sets and therefore training the model badly. This happens because N different sets are not independent of one another.

### Early stopping



[Geoffrey Hinton] “So now I'm going to describe one particularly easy to use method for printing over fitting. 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. And you watch the performance on the validation set. And as soon as it starts to get worse, you stop training. Now, the performance civilization on the set may fluctuate particularly if you're error rate rather than a squared error or presentory error. And so its 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 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. It's interesting to ask why small weights lower the capacity.”



[Geoffrey Hinton] So consider a model with some input units, some hidden units, and some output units. When the weight's very small, if the hidden unit's a logistic units, their total inputs will be close to zero, and they'll be in the middle of their linear range. That is, 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, 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 note. Provided we also divide the weights in the linear note by four, which take into account the fact that when there's 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 in that network I'm showing you there's three six + six two weights, it's really got no more capacity than a network with three two weights. That's the way its grow. We start using the non linear region of the sequence. And then we start making use of all those parameters. So if the network has six 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 six parameters to 30 parameters as the weights get bigger. And 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.

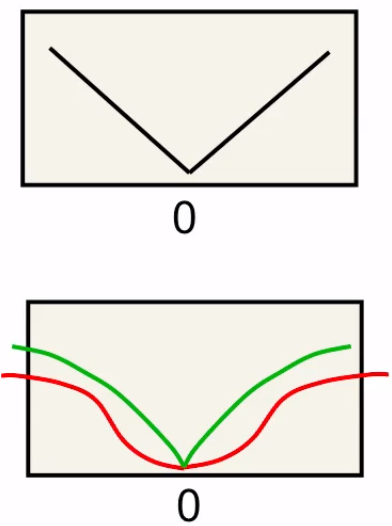
### Limiting size of the weights

#### Penalize the squared value of each weight separately.

* L2 penalty
  + It has the shape of a parabolic bowl.
  + It penalizes large weights.
  + It reduces the gap between the training data and the validation data error.

In short, the L2 weight cost penalizes large weights in the network and improves network generalization to unseen test data.

* L1 penalty
  + It has the shape of a V.
  + It makes many weights zero which makes it easier to interpret what is going on in the network. In short, it helps in debugging the model.
* Penalty with negligible effect on large weights
  + This is technically opposite to L1
  + The gradient of the cost function gets smaller as the weight gets really big. This allows a few large weights to get through.
  + Only small weights get pulled towards 0.



#### Put constraints on weights

Put a constraint on the maximum length of the incoming weight vector of each unit. If an update violates this constraint. we simply scale the weight vector down to its allowable length.

**Advantages over penalizing weights**

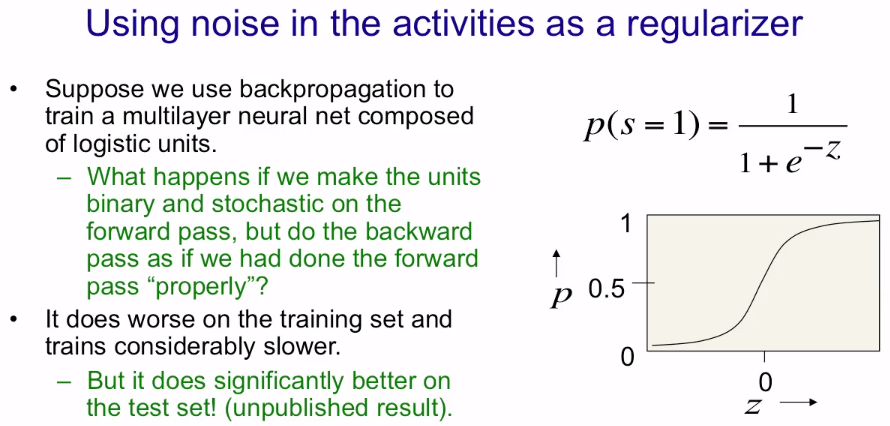
* It's much **easier to select the sensible value** for the squared length of the incoming weight factor than it is to select the weight penalty. That's because, logistic units have, a natural scale to them so we know what a weight of one means.
* Prevents hidden units from getting stuck near zero. – Because when all their weights are tiny, the constraints no them are zero, so nothing prevents them from growing.
* Prevents weights from exploding.
* **Dynamic penalty value**: When a unit hits its limits (e.g. 1 for a logistic unit), the effective penalty of all the weights depends on the big gradient. This means weights with big gradients will try to push the length of the incoming weight vector up. In effect, **the penalty scales itself unlike L1/L2 norms with fixed values** to be appropriate for the big weights and to suppress the small weights. **This is more effective.**

#### Use noise as a regularizer

**Add noise either to the weights or to the activities.**

* Add noise to inputs of a network, e.g. Gaussian noise. – equivalent to L2 decay
* Adding Gaussian noise to the weights of a multi-layer non-linear network improves performance of conventional **recurrent networks** [Alex Graves]

## Regularize backpropagation to train multilayer neural net (unpublished?)



[Geoffrey Hinton] We can also use noise in the activities as a regularizer So suppose we use back propagation to train a multi-layer neural network match with logistic hidden units. What's gonna happen if we make the units binary and 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 stochastic 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. 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 paths, you use the real value of p for back propagating derivatives through the hidden unit. This isn't 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.

## Bayesian approach

Brute force: Instead of looking at the more possible settings of the parameters, you look at all possible settings of the parameters.

### Maximum Likelihood Estimation

Suppose we have a fair coin. We can toss it to get either a head or a tail. Every toss is independent of one another. Let us assume that after 100 tosses, we get 53 heads and 47 tails. What is the probability of getting head?

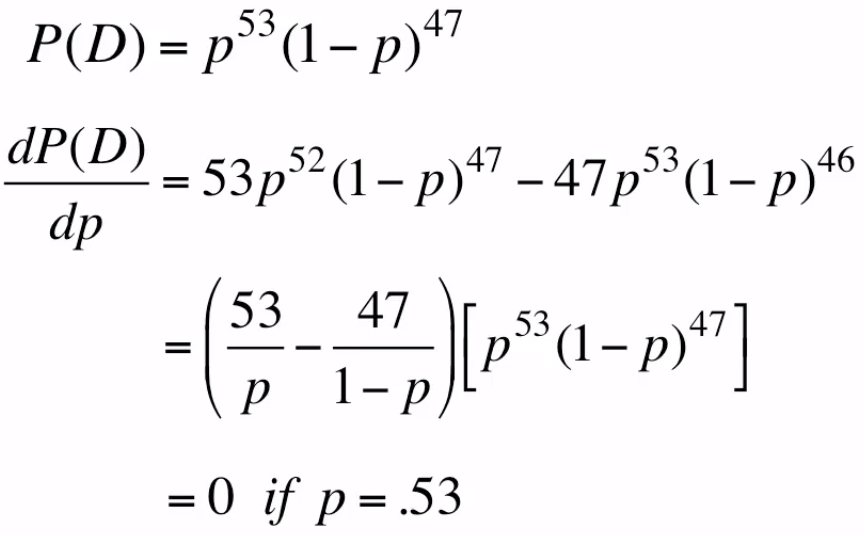
The frequentist answer: p = 0.53. Is it true?

Let us derive this:

How does this probability of observing the data depends on P? i.e. probability, p, of observing a head given the sequence, P, containing 53 heads and 47 tails.

Method:

* Calculate the derivative: the change in the sequence w.r.t the change in observing a head.
* set the derivative to zero



Question: What if we toss the coin only once and get head? Does that infer p(H) = 1?

To a machine, it may be so.

**Essence of Bayesian approach**: You can start with a wrong estimation and use data to gradually make it right. [**incorrect prior >>> data, scale data <<< correct posterior**]

