

Lecture 4: Recurrent Neural Networks + Generalization

Prof. Manolis Kellis

Recurrent Neural Networks (RNNs) + Generalization

1. How do you read/listen/understand/write? Can machines do that?

- Context matters: characters, words, letters, sounds, completion, multi-modal
- Predicting next word/image: from unsupervised learning to supervised learning

2. Encoding temporal context: Hidden Markov Models (HMMs), RNNs

- Primitives: hidden state, memory of previous experiences, limitations of HMMs
- RNN architectures, unrolling, back-propagation-through-time (BPTT), param reuse

3. Vanishing gradients, Long-Short-Term Memory (LSTM), initialization

- Key idea: gated input/output/memory nodes, model choose to forget/remember
- Example: online character recognition with LSTM recurrent neural network

4. Improving generalization

- **More training data**
- **Tuning model capacity**
 - Architecture: # layers, # units
 - Early stopping: (validation set)
 - Weight-decay: L1/L2 regularization
 - Noise: Add noise as a regularizer
- **Bayesian prior on parameter distribution**
- **Why weight decay \Leftrightarrow Bayesian prior**
- **Variance of residual errors**


1a. What do you hear and why?

Context matters

Top-down
processing

THE CAT

Accdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttar in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

 **[FREN] butters** ✓
@NoisyButters

do yall pronounce it data or data?

data	70.3%
data	29.7%

6,504 votes · Final results

7:54 PM · Sep 14, 2018 · [Twitter for iPhone](#)

56 Retweets 485 Likes

7H15 M3554G3
53RV35 7O PR0V3
H0W 0UR M1ND5 C4N
D0 4M4Z1NG 7H1NG5!
1MPR3551V3 7H1NG5!
1N 7H3 B3G1NN1NG
17 WA5 H4RD BU7
N0W, 0N 7H15 LIN3
Y0UR M1ND 1S
R34D1NG 17
4U70M471C4LLY
W17H 0U7 3V3N
7H1NK1NG 4B0U7 17,
B3 PROUD! 0NLY
C3R741N P30PL3 C4N
R3AD 7H15.
PL3453 F0RW4RD 1F
U C4N R34D 7H15.


Phonemic
restoration




Hearing lips and seeing voices
(McGurk, MacDonald, Nature 1976)

<https://youtu.be/PWGeUztTkRA?t=35>

Split class into 4 groups: (1) close your eyes, (2) look left, (3) middle, (4) right



Delayed Auditory Feedback
Stuttering -DAF

ChinJa Music & Audio
Everyone

Contains Ads · Offers in-app purchases
This app is compatible with your device.

Adults: 200 ms delay max disruption.
Children: 500 ms

Delayed typing: Google Docs, zoom
video screen sharing, slow computer

"When we listen to someone talking, the change in our brain's processing from not caring what kind of sound it is to recognizing it as a word happens surprisingly early," said Simon. "In fact, this happens pretty much as soon as the linguistic information becomes available."

When it is engaging in speech perception, the brain's auditory cortex analyzes complex acoustic patterns to detect words that carry a linguistic message. It seems to do this so efficiently, at least in part, by **anticipating what it is likely to hear: by learning what sounds signal language most frequently, the brain can predict what may come next.** It is generally thought that this process -- localized bilaterally in the brain's superior temporal lobes -- involves recognizing an intermediate, phonetic level of sound.

<https://www.sciencedaily.com/releases/2018/11/181129142352.htm>

Recurrent Neural Networks (RNNs) + Generalization

1. How do you read/listen/understand/write? Can machines do that?

- Context matters: characters, words, letters, sounds, completion, multi-modal
- Predicting next word/image: from unsupervised learning to supervised learning

2. Encoding temporal context: Hidden Markov Models (HMMs), RNNs

- Primitives: hidden state, memory of previous experiences, limitations of HMMs
- RNN architectures, unrolling, back-propagation-through-time (BPTT), param reuse

3. Vanishing gradients, Long-Short-Term Memory (LSTM), initialization

- Key idea: gated input/output/memory nodes, model choose to forget/remember
- Example: online character recognition with LSTM recurrent neural network

4. Improving generalization

- **More training data**
- **Tuning model capacity**
 - Architecture: # layers, # units
 - Early stopping: (validation set)
 - Weight-decay: L1/L2 regularization
 - Noise: Add noise as a regularizer
- **Bayesian prior on parameter distribution**
- **Why weight decay \Leftrightarrow Bayesian prior**
- **Variance of residual errors**

2a. Encoding time

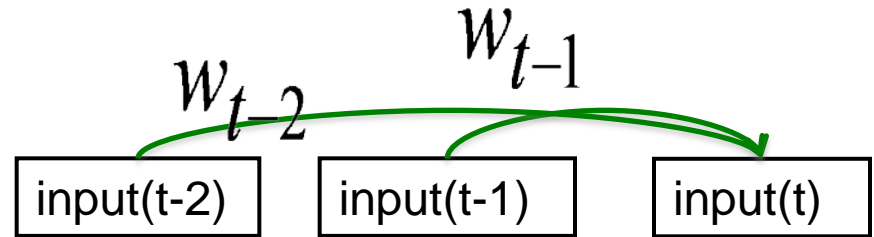
Getting targets when modeling sequences

- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain.
 - *E. g.* turn a sequence of sound pressures into a sequence of word identities.
- When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence.
 - The target output sequence is the input sequence with an advance of 1 step.
 - This seems much more natural than trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image.
 - For temporal sequences there is a natural order for the predictions.
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning.
 - It uses methods designed for supervised learning, but it doesn't require a separate teaching signal.

Memoryless models for sequences

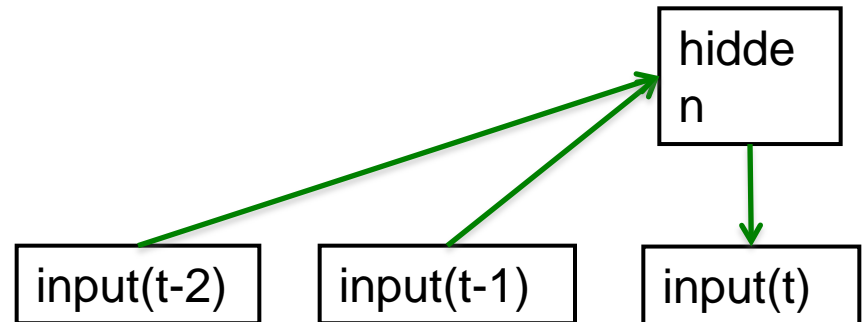
- Autoregressive models

Predict the next term in a sequence from a fixed number of previous terms using “delay taps”.



- Feed-forward neural nets

These generalize autoregressive models by using one or more layers of non-linear hidden units.

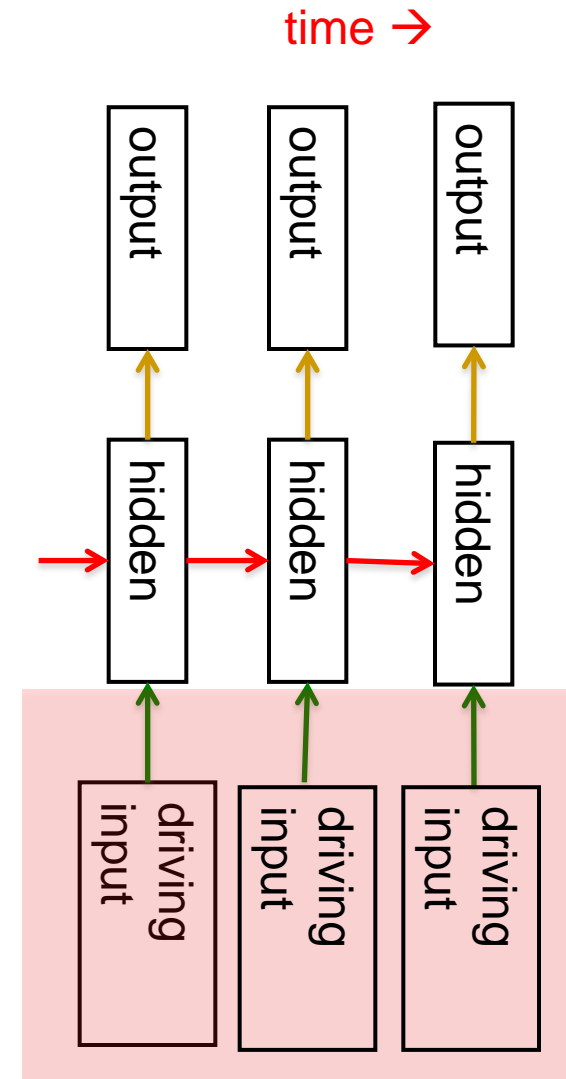


Beyond memoryless models

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
 - It can store information in its hidden state for a long time.
 - If the dynamics is noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state.
 - The best we can do is to infer a probability distribution over the space of hidden state vectors.
- This inference is only tractable for two types of hidden state model.

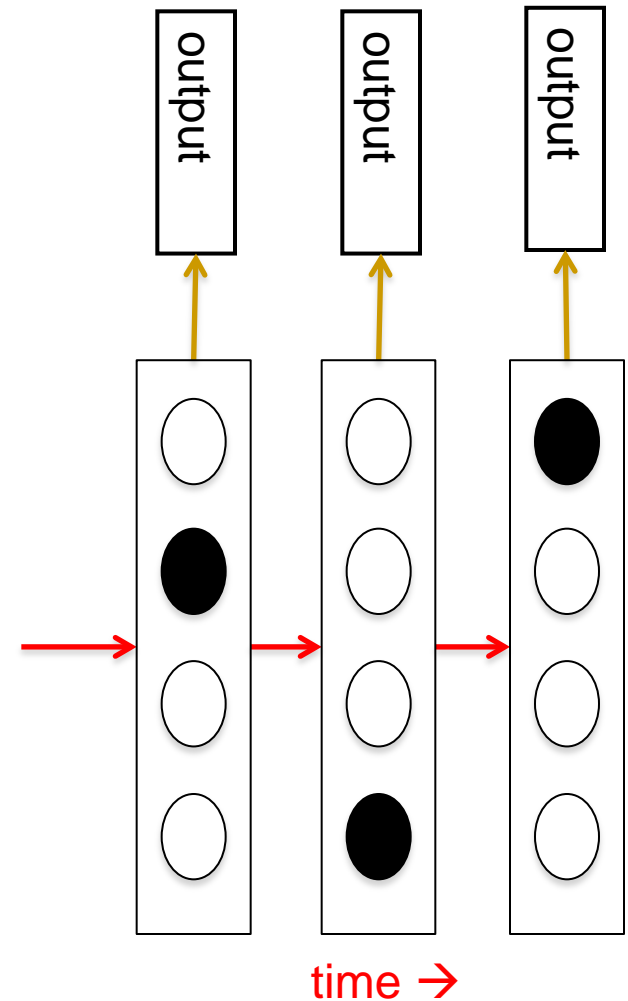
Linear Dynamical Systems (engineers love them!)

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
 - The hidden state has linear dynamics with Gaussian noise and produces the observations using a linear model with Gaussian noise.
 - There may also be driving inputs.
- To predict the next output (so that we can shoot down the missile) we need to infer the hidden state.
 - A linearly transformed Gaussian is a Gaussian. So the distribution over the hidden state given the data so far is Gaussian. It can be computed using “Kalman filtering”.



Hidden Markov Models (computer scientists love them!)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
 - We cannot be sure which state produced a given output. So the state is “hidden”.
 - It is easy to represent a probability distribution across N states with N numbers.
- To predict the next output we need to infer the probability distribution over hidden states.
 - HMMs have efficient algorithms for inference and learning.



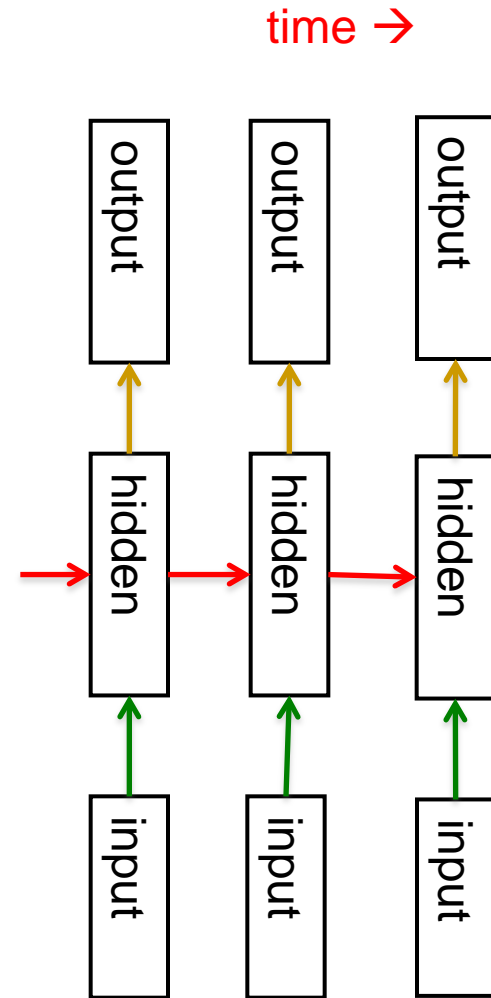
A fundamental limitation of HMMs

- Consider what happens when a hidden Markov model generates data.
 - At each time step it must select one of its hidden states. So with N hidden states it can only remember $\log(N)$ bits about what it generated so far.
- Consider the information that the first half of an utterance contains about the second half:
 - The syntax needs to fit (e.g. number and tense agreement).
 - The semantics needs to fit. The intonation needs to fit.
 - The accent, rate, volume, and vocal tract characteristics must all fit.
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half. 2^{100} is big!

2b. Recursive Neural Networks (RNNs)

Recurrent neural networks

- RNNs are very powerful, because they combine two properties:
 - Distributed hidden state that allows them to store a lot of information about the past efficiently.
 - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.



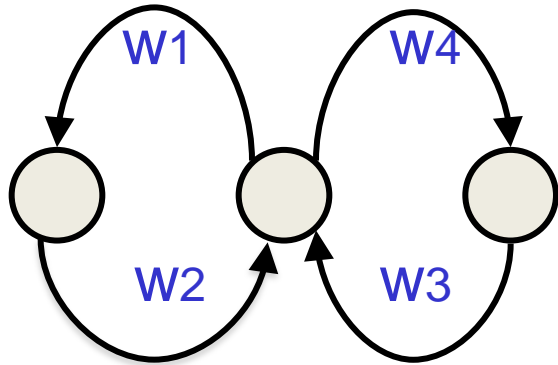
Do generative models need to be stochastic?

- Linear dynamical systems and hidden Markov models are stochastic models.
 - But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data.
- Recurrent neural networks are deterministic.
 - So think of the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or hidden Markov model.

Recurrent neural networks

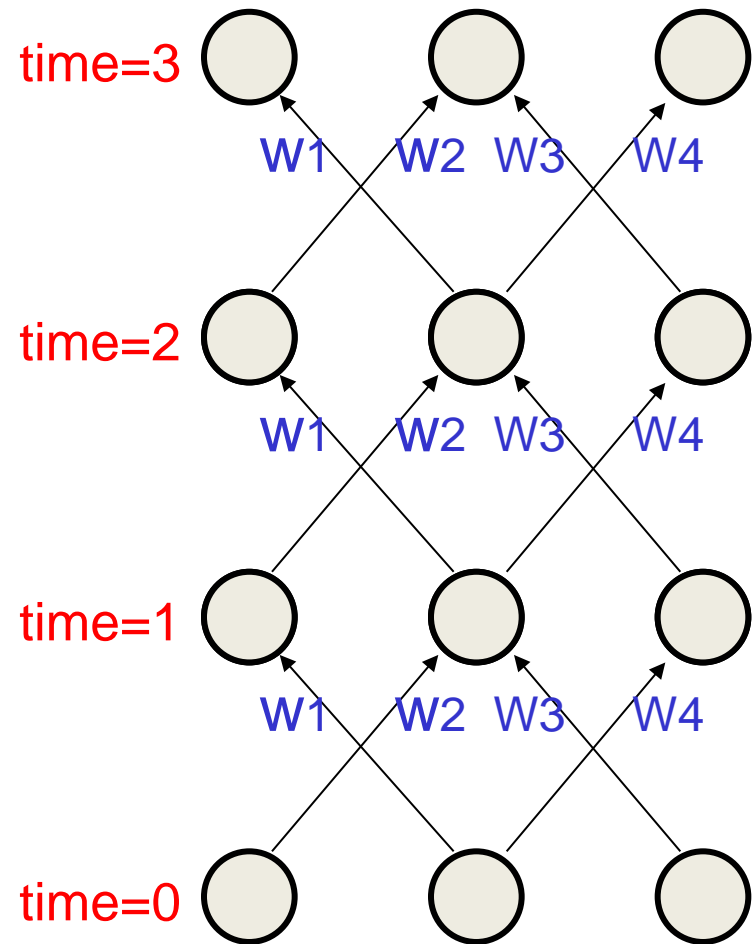
- What kinds of behaviour can RNNs exhibit?
 - They can oscillate. Good for motor control?
 - They can settle to point attractors. Good for retrieving memories?
 - They can behave chaotically. Bad for information processing?
 - RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects.
- But the computational power of RNNs makes them very hard to train.
 - For many years we could not exploit the computational power of RNNs despite some heroic efforts (e.g. Tony Robinson's speech recognizer).

The equivalence between feedforward nets and recurrent nets



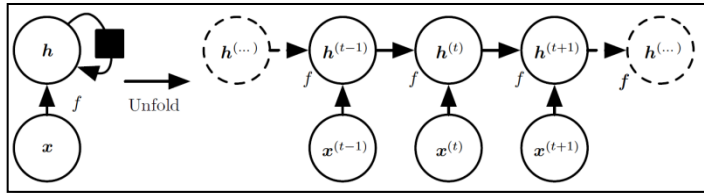
Assume that there is a time delay of 1 in using each connection.

The recurrent net is just a layered net that keeps reusing the same weights.

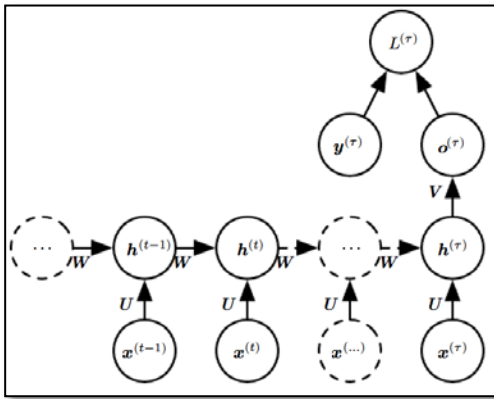


2c. Alternative architectures for RNNs

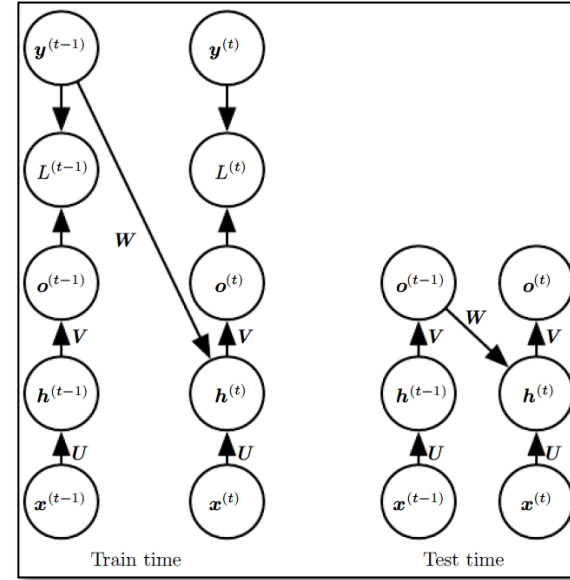
Different RNN remembering architectures



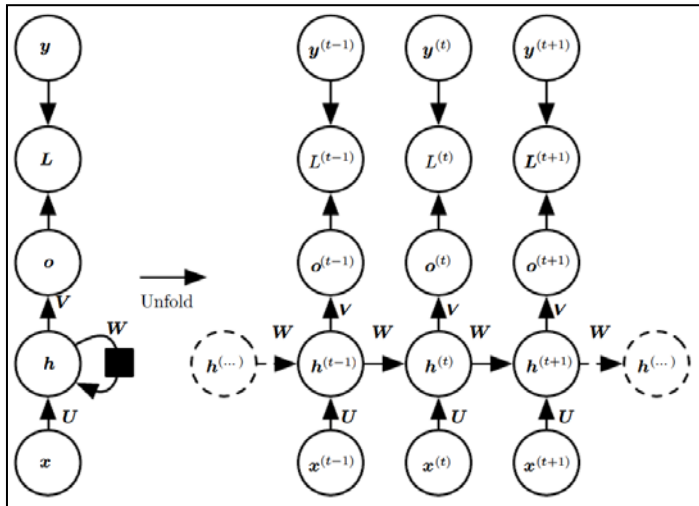
Recurrent network with no outputs



Single output after entire sequence

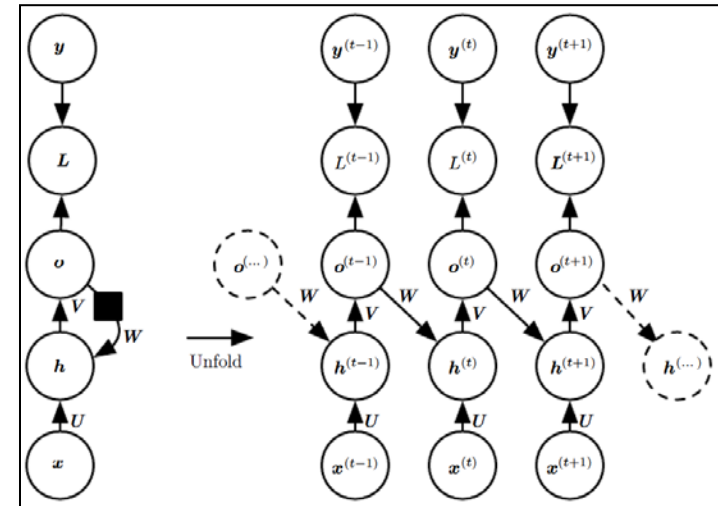


Teacher-forcing: train from \mathbf{y} and \mathbf{x} in parallel



o: output, y: target, L: loss

Memory: $h^{(t-1)} \rightarrow h^{(t)}$



o: output, y: target, L: loss

Memory: $o^{(t-1)} \rightarrow h^{(t)}$. Only train sequentially

2d. Back-propagation through time (BPTT)

Reminder: Backpropagation with weight constraints

- It is easy to modify the backprop algorithm to incorporate linear constraints between the weights.
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
 - So if the weights started off satisfying the constraints, they will continue to satisfy them.

To constrain: $w_1 = w_2$

we need: $\Delta w_1 = \Delta w_2$

compute: $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$

use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ *for* w_1 *and* w_2

Backpropagation through time

- We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.
- We can also think of this training algorithm in the time domain:
 - The forward pass builds up a stack of the activities of all the units at each time step.
 - The backward pass peels activities off the stack to compute the error derivatives at each time step.
 - After the backward pass we add together the derivatives at all the different times for each weight.

Getting targets when modeling sequences

- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain.
 - *E. g.* turn a sequence of sound pressures into a sequence of word identities.
- When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence.
 - The target output sequence is the input sequence with an advance of 1 step.
 - This seems much more natural than trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image.
 - For temporal sequences there is a natural order for the predictions.
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning.
 - It uses methods designed for supervised learning, but it doesn't require a separate teaching signal.

Recurrent Neural Networks (RNNs) + Generalization

1. How do you read/listen/understand/write? Can machines do that?

- Context matters: characters, words, letters, sounds, completion, multi-modal
- Predicting next word/image: from unsupervised learning to supervised learning

2. Encoding temporal context: Hidden Markov Models (HMMs), RNNs

- Primitives: hidden state, memory of previous experiences, limitations of HMMs
- RNN architectures, unrolling, back-propagation-through-time (BPTT), param reuse

3. Vanishing gradients, Long-Short-Term Memory (LSTM), initialization

- Key idea: gated input/output/memory nodes, model choose to forget/remember
- Example: online character recognition with LSTM recurrent neural network

4. Improving generalization

- **More training data**
- **Tuning model capacity**
 - Architecture: # layers, # units
 - Early stopping: (validation set)
 - Weight-decay: L1/L2 regularization
 - Noise: Add noise as a regularizer
- **Bayesian prior on parameter distribution**
- **Why weight decay \Leftrightarrow Bayesian prior**
- **Variance of residual errors**

3a. Remembering for
longer time periods

Four effective ways to increase length of memory

- **Long Short Term Memory**
Make the RNN out of little modules that are designed to remember values for a long time.
- **Hessian Free Optimization:** Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
 - The HF optimizer (Martens & Sutskever, 2011) is good at this.
- **Echo State Networks:** Initialize the input→hidden and hidden→hidden and output→hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
 - ESNs only need to learn the hidden→output connections.
- **Good initialization with momentum**
Initialize like in Echo State Networks, but then learn all of the connections using momentum.

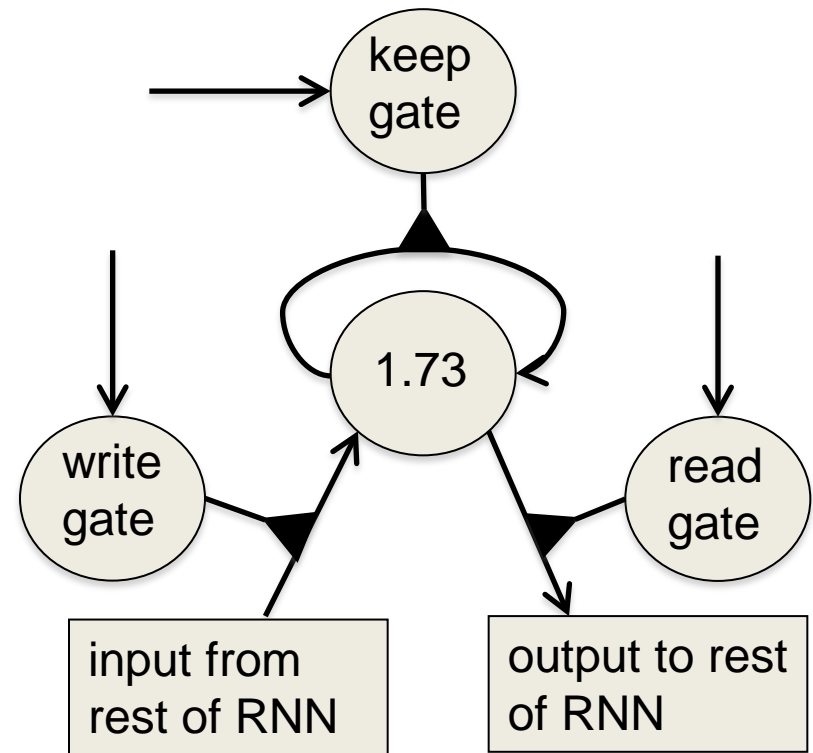
Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.

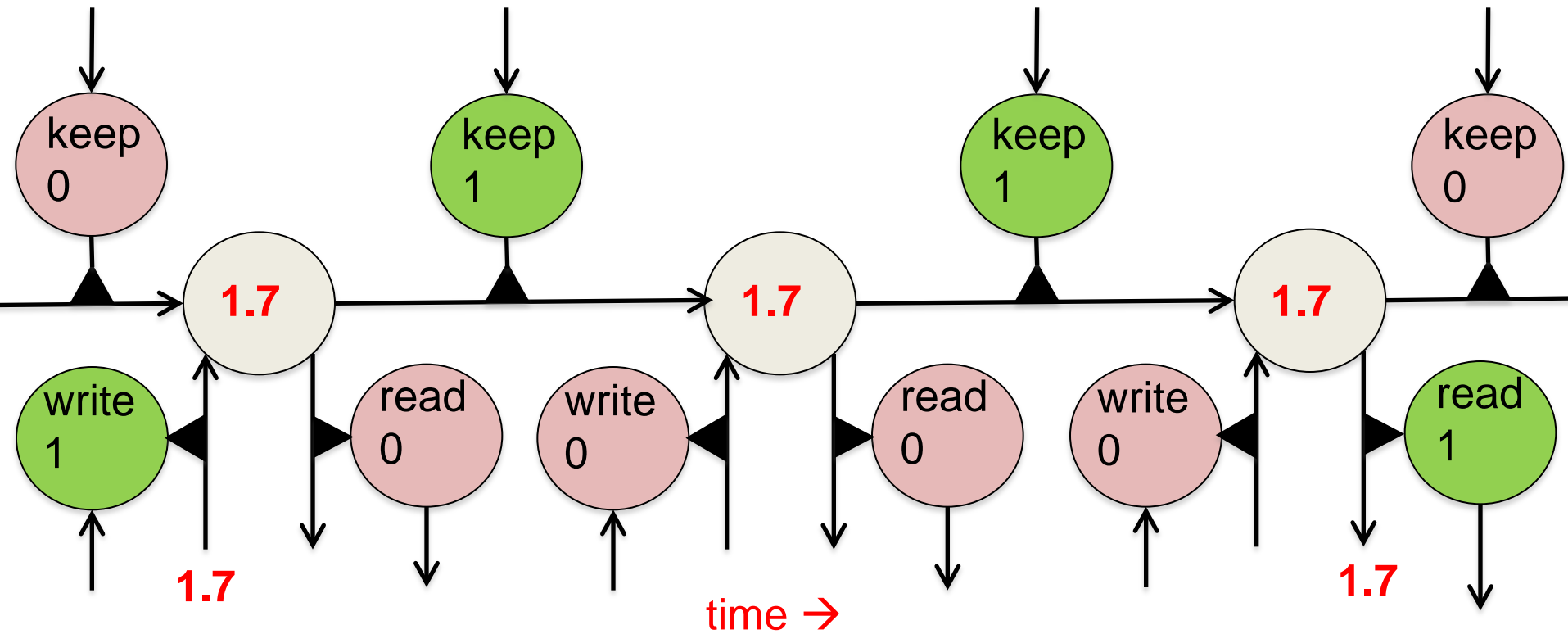
Implementing a memory cell in a neural network

To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistics are have nice derivatives.



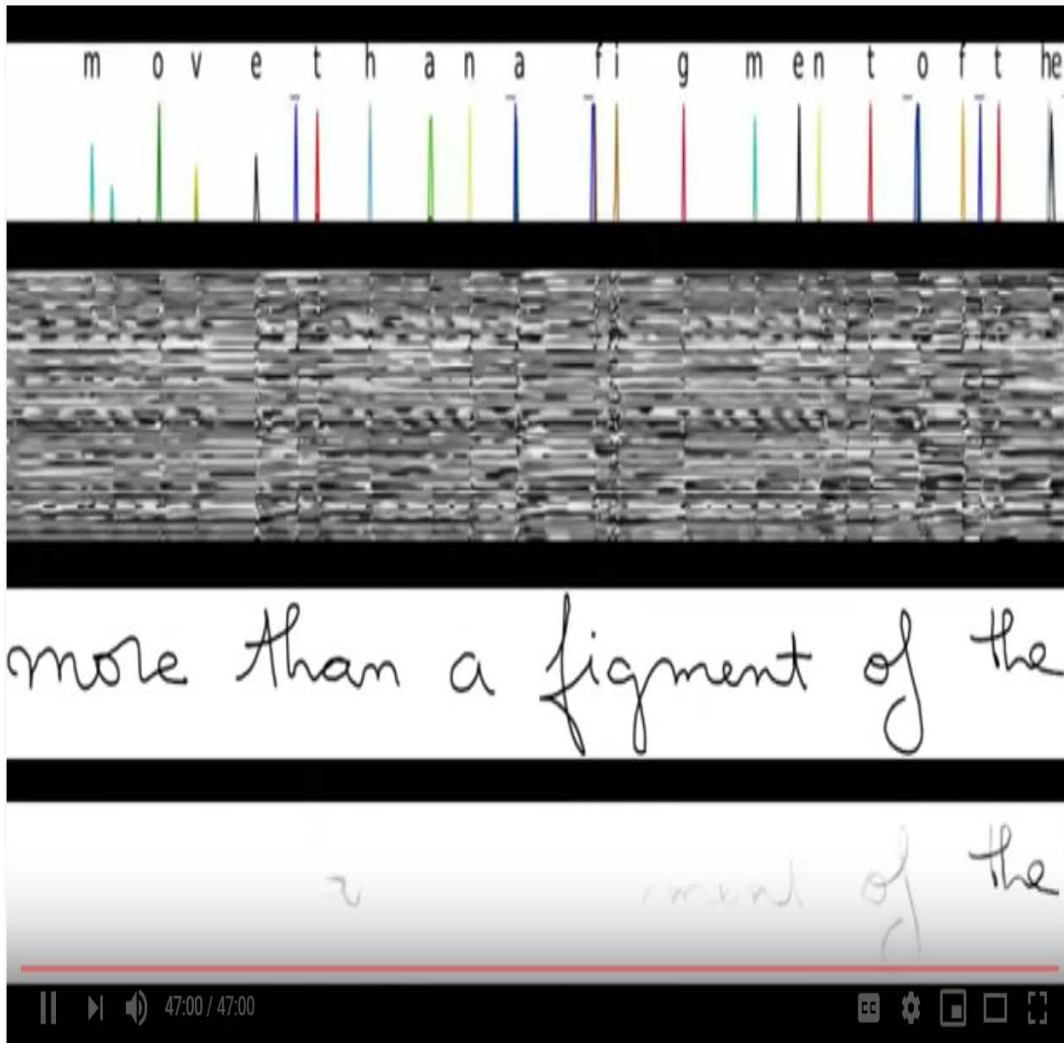
Backpropagation through a memory cell



Reading cursive handwriting

- This is a natural task for an RNN.
- The input is a sequence of (x,y,p) coordinates of the tip of the pen, where p indicates whether the pen is up or down.
- The output is a sequence of characters.
- Graves & Schmidhuber (2009) showed that RNNs with LSTM are currently the best systems for reading cursive writing.
 - They used a sequence of small images as input rather than pen coordinates.

Demonstration of online handwriting recognition by an RNN with Long Short Term Memory (from Alex Graves)



Row 1: Shows when characters are recognized.

- It never revises its output so difficult decisions are more delayed.

Row 2: Shows the states of a subset of the memory cells.

- Notice how they get reset when it recognizes a character.

Row 3: Shows the writing. The net sees the x and y coordinates.

- Optical input actually works a bit better than pen coordinates.

Row 4: Shows the gradient backpropagated all the way to the x and y inputs from the currently most active character.

- This lets you see which bits of the data are influencing the decision.

<https://youtu.be/9T2X6WRUwFU?t=2791>

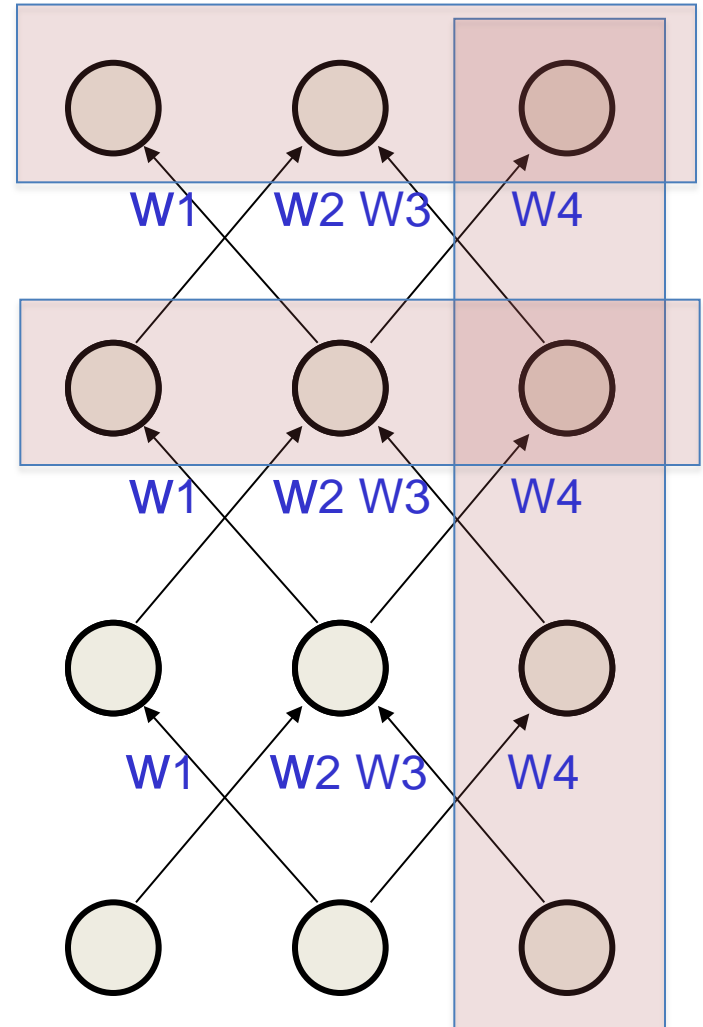
3b. Initialization

Initialization: Dealing with boundary cases

- We need to specify the initial activity state of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
 - Start off with an initial random guess for the initial states.
 - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
 - **Adjust the initial states** by following the negative gradient.

Teaching signals for recurrent networks

- We can specify targets in several ways:
 - Specify **desired final activities** of all the units
 - Specify desired activities of all units for the **last few steps**
 - Good for learning **attractors**
 - It is easy to add in extra error derivatives as we backpropagate.
 - Specify the desired activity of a subset of the units.
 - The other units are input or hidden units.

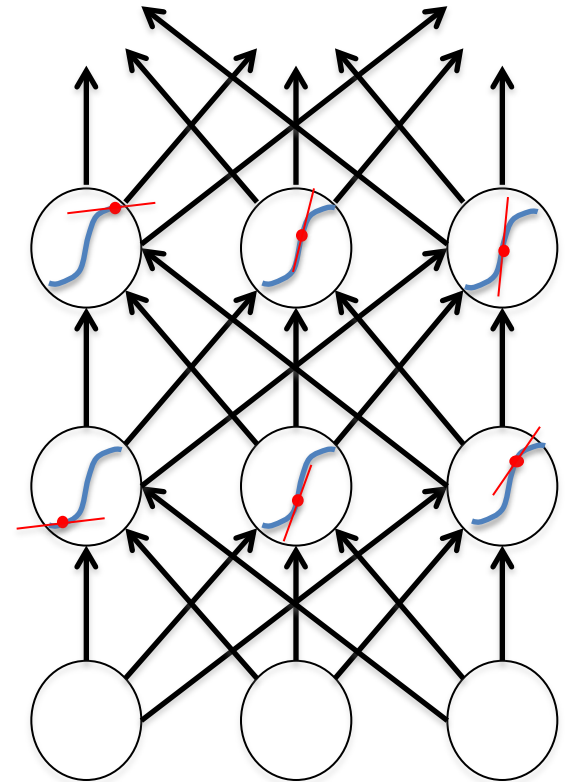


What the network learns

- It learns four distinct patterns of activity for the 3 hidden units. These **patterns** correspond to the nodes in the finite state automaton.
 - Do not confuse units in a neural network with nodes in a finite state automaton. Nodes are like activity vectors.
 - The **automaton** is restricted to be in **exactly one state** at each time. The hidden units are restricted to have exactly **one vector of activity at each time**.
- A recurrent network can **emulate a finite state automaton**, but it is **exponentially more powerful**. With N hidden neurons it has 2^N possible binary activity vectors (but only N^2 weights)
 - This is important when the input stream has two separate things going on at once.
 - A finite state automaton needs to square its number of states.
 - An RNN needs to double its number of **units**.

The backward pass is linear

- There is a big difference between the forward and backward passes.
- In the **forward** pass we use **squashing** functions (like the logistic) to **prevent** the activity vectors from **exploding**.
- The **backward** pass, is completely **linear**. If you double the error derivatives at the final layer, all the error derivatives will double.
 - The forward pass determines the slope of the **linear** function used for backpropagating through each neuron.



The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the **gradients shrink exponentially**.
 - If the weights are big the **gradients grow exponentially**.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.
 - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
 - So RNNs have difficulty dealing with **long-range dependencies**.
 - Can use ideas for residual networks (ResNet), pass info from the input to far away nodes

Recurrent Neural Networks (RNNs) + Generalization

1. How do you read/listen/understand/write? Can machines do that?

- Context matters: characters, words, letters, sounds, completion, multi-modal
- Predicting next word/image: from unsupervised learning to supervised learning

2. Encoding temporal context: Hidden Markov Models (HMMs), RNNs

- Primitives: hidden state, memory of previous experiences, limitations of HMMs
- RNN architectures, unrolling, back-propagation-through-time (BPTT), param reuse

3. Vanishing gradients, Long-Short-Term Memory (LSTM)

- Key idea: gated input/output/memory nodes, model choose to forget/remember
- Example: online character recognition with LSTM recurrent neural network

4. Improving generalization

- **More training data**
- **Tuning model capacity**
 - Architecture: # layers, # units
 - Early stopping: (validation set)
 - Weight-decay: L1/L2 regularization
 - Noise: Add noise as a regularizer
- **Bayesian prior on parameter distribution**
- **Why weight decay \Leftrightarrow Bayesian prior**
- **Variance of residual errors**

4. Improving generalization

Ways to reduce overfitting

- A large number of different methods have been developed.
 - Weight-decay
 - Weight-sharing
 - Early stopping
 - Model averaging
 - Bayesian fitting of neural nets
 - Dropout
 - Generative pre-training
- Many of these methods will be described in lecture 7.

Reminder: Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains sampling error.
 - There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model to the training set it cannot tell which regularities are real and which are caused by sampling error.
 - So it fits both kinds of regularity. If the model is very flexible it can model the sampling error really well.
- If you fitted the model to another training set drawn from the same distribution over cases, it would make different predictions on the test data. This is called “variance”.

Preventing overfitting

- Approach 1: Get more data!
 - Almost always the best bet if data is cheap and you have enough compute power to train on more data.
- Approach 2: Use a model that has the right capacity:
 - enough to fit the true regularities.
 - not enough to also fit spurious regularities (if they are weaker).
- Approach 3: Average many different models.
 - Use models with different forms.
 - Or train the model on different subsets of the training data (this is called “bagging”).
- Approach 4: (Bayesian) Use a single neural network architecture, but average the predictions made by many different weight vectors.

Get more data

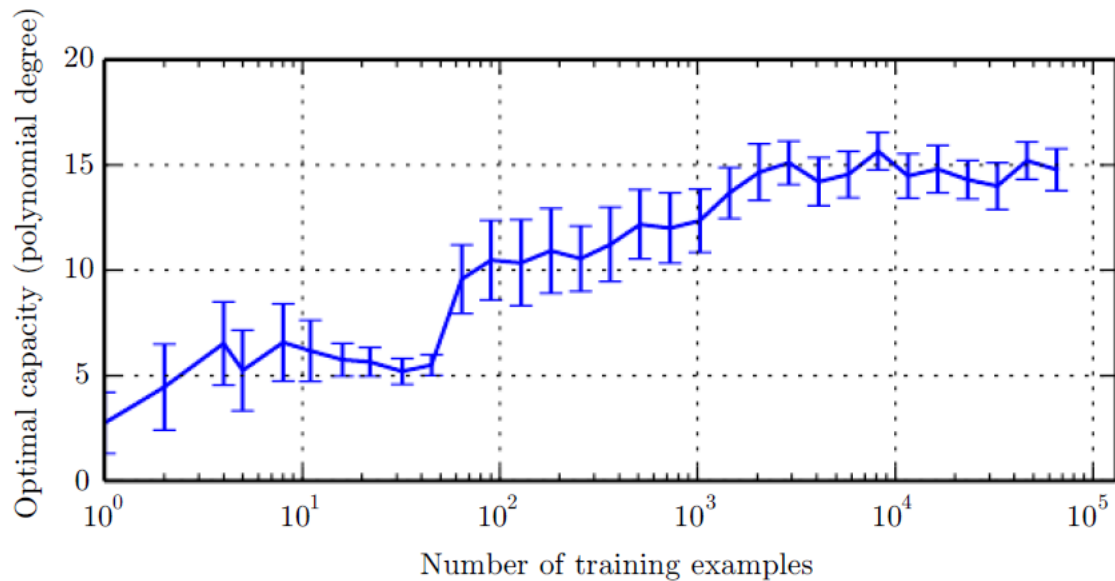
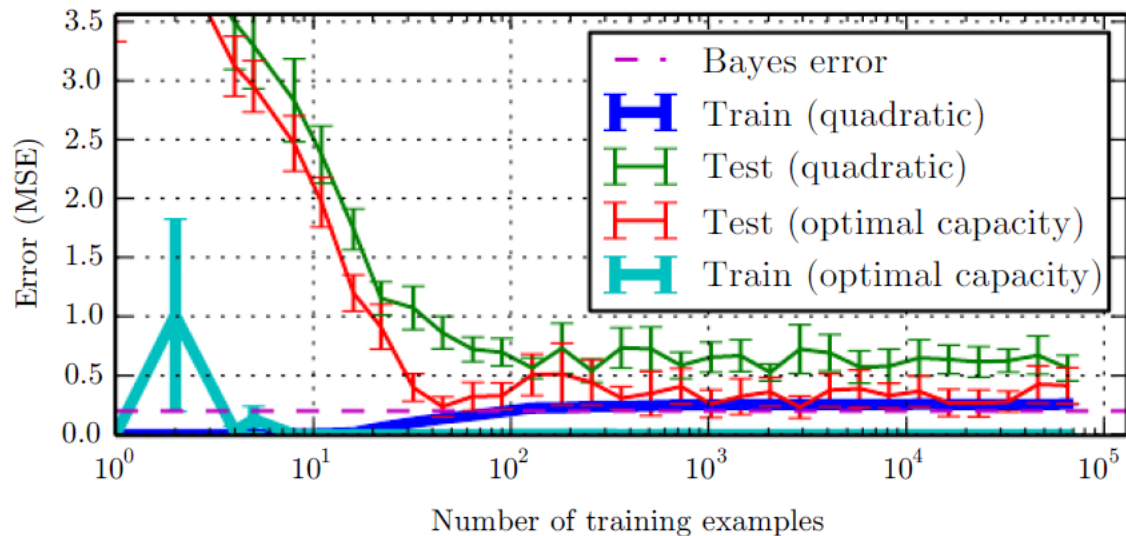


Figure 5.4: The effect of the training dataset size on the train and test error, as well as on the optimal model capacity. We constructed a synthetic regression problem based on adding a moderate amount of noise to a degree-5 polynomial, generated a single test set, and then generated several different sizes of training set. For each size, we generated 40 different training sets in order to plot error bars showing 95 percent confidence intervals. (Top) The MSE on the training and test set for two different models: a quadratic model, and a model with degree chosen to minimize the test error. Both are fit in closed form. For the quadratic model, the training error increases as the size of the training set increases. This is because larger datasets are harder to fit. Simultaneously, the test error decreases, because fewer incorrect hypotheses are consistent with the training data. The quadratic model does not have enough capacity to solve the task, so its test error asymptotes to a high value. The test error at optimal capacity asymptotes to the Bayes error. The training error can fall below the Bayes error, due to the ability of the training algorithm to memorize specific instances of the training set. As the training size increases to infinity, the training error of any fixed-capacity model (here, the quadratic model) must rise to at least the Bayes error. As the training (Bottom) set size increases, the optimal capacity (shown here as the degree of the optimal polynomial regressor) increases. The optimal capacity plateaus after reaching sufficient complexity to solve the task.

4. Improving generalization:

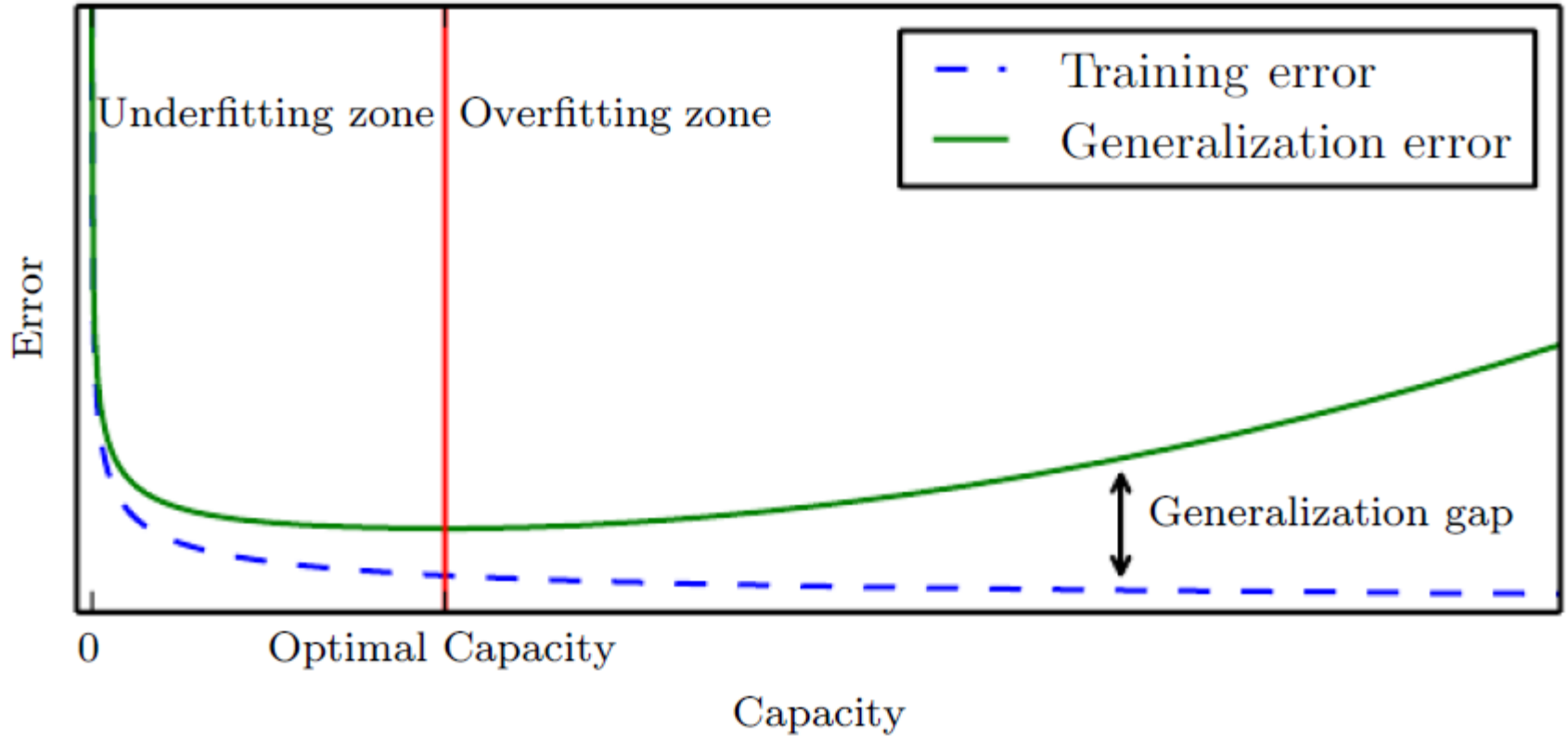
a. Controlling model capacity

- Architecture: # layers, # units
- Early stopping: (validation set)
- Weight-decay: L1/L2 regularization
- Noise: Add noise

Some ways to limit the capacity of a neural net

- The capacity can be controlled in many ways:
 - **Architecture:** Limit the number of hidden layers and the number of units per layer.
 - **Early stopping:** Start with small weights and stop the learning before it overfits.
 - **Weight-decay:** Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
 - **Noise:** Add noise to the weights or the activities.
- Typically, a combination of several of these methods is used.

Effect of model capacity on generalization



Tuning model capacity: Overfitting, underfitting

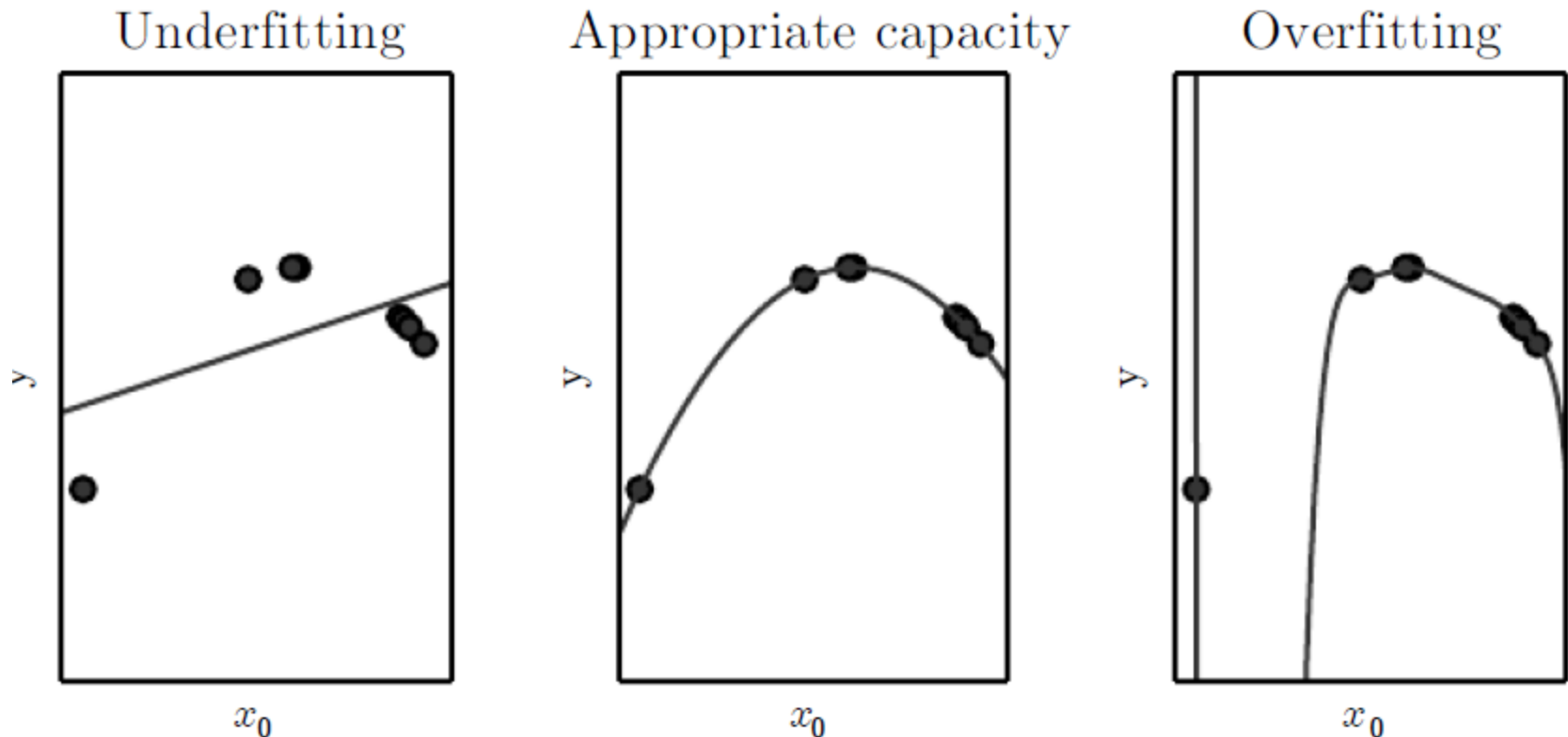


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. (Left) A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. (Center) A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. (Right) A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudoinverse to solve the underdetermined normal equations. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

4a. Architecture hyperparams
(# layers, # units)

How to choose meta parameters that control capacity (like the number of hidden units or the size of the weight penalty)

- The wrong method is to try lots of alternatives and see which gives the best performance on the test set.
 - This is easy to do, but it gives a false impression of how well the method works.
 - The settings that work best on the test set are unlikely to work as well on a new test set drawn from the same distribution.
- An extreme example:
 - Suppose the test set has random answers that do not depend on the input.
 - The best architecture will do better than chance on the test set.
 - But it cannot be expected to do better than chance on a new test set.

Cross-validation: A better way to choose meta parameters

- Divide the total dataset into three subsets:
 - **Training data** is used for learning the parameters of the model.
 - **Validation data** is not used for learning but is used for deciding what settings of the meta parameters work best.
 - **Test data** is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could divide the total dataset into one final test set and N other subsets and train on all but one of those subsets to get N different estimates of the validation error rate.
 - This is called N-fold cross-validation.
 - The N estimates are **not** independent.

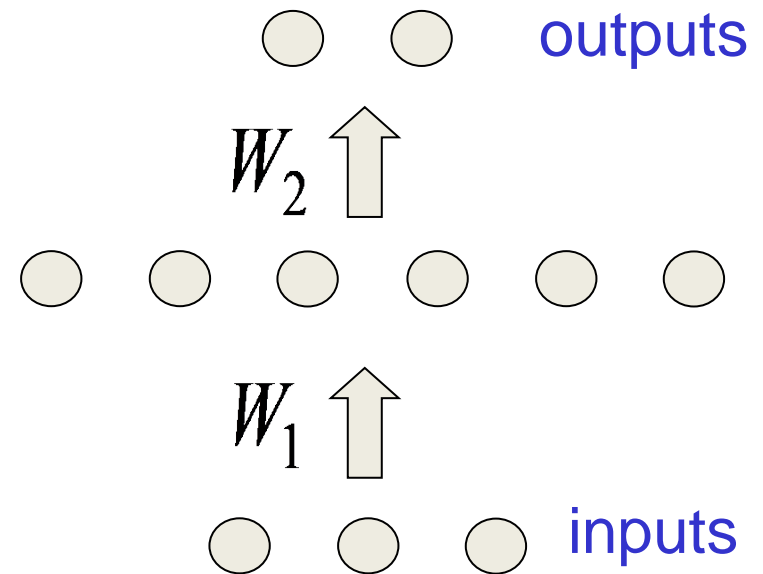
4b. Early stopping
(training/validation/testing)

Preventing overfitting by early stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different sized penalties on the weights or different architectures.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse.
 - But it can be hard to decide when performance is getting worse.
- The capacity of the model is limited because the weights have not had time to grow big.
 - Smaller weights give the network less capacity. Why?

Why early stopping works

- When the weights are very small, every hidden unit is in its linear range.
 - So even with a large layer of hidden units it's a linear model.
 - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



4c. Weight regularization (L1, L2, Elastic Net)

Effect of weight decay

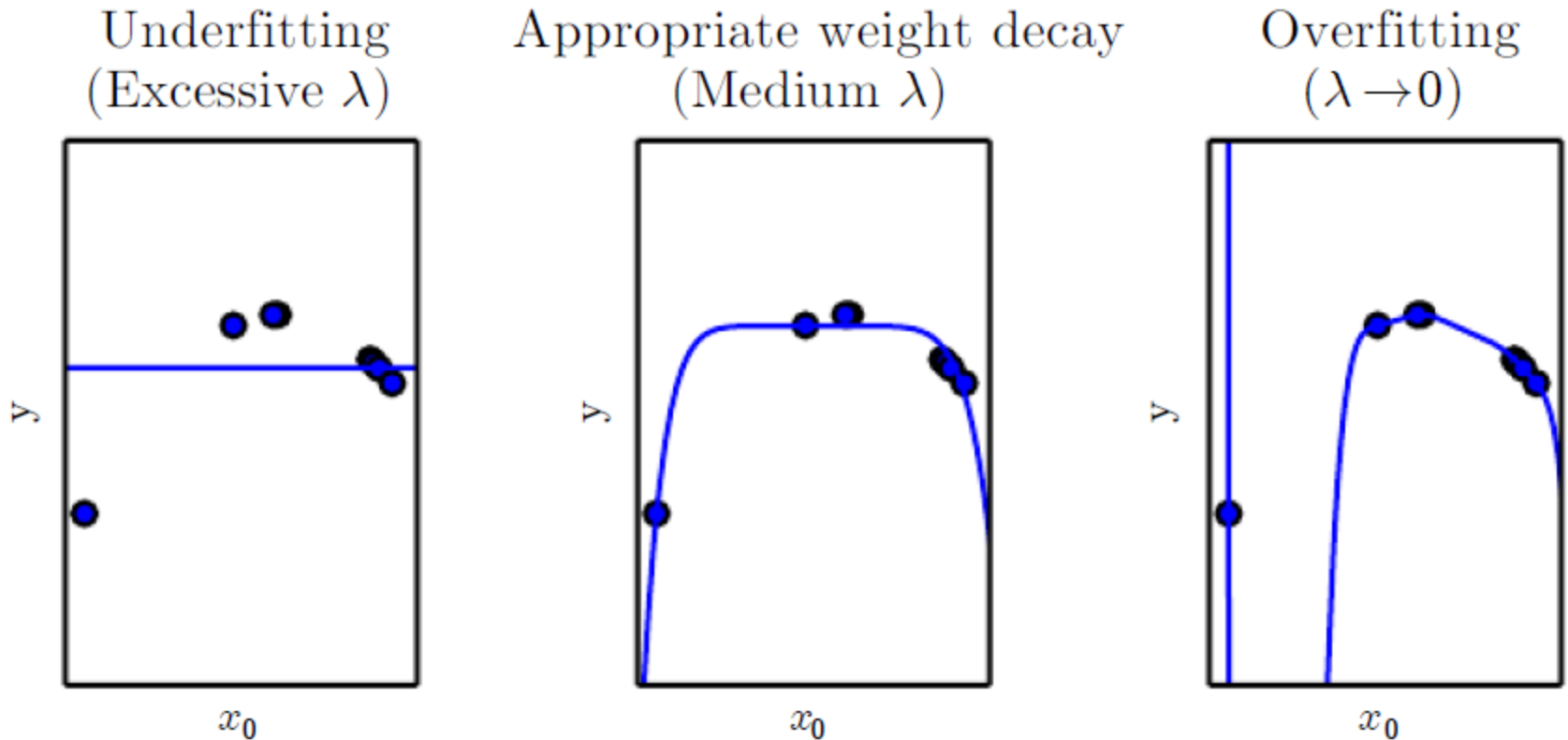


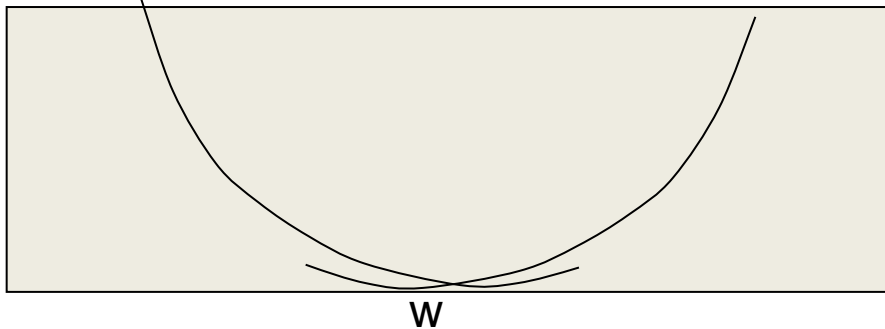
Figure 5.5. We fit a high-degree polynomial regression model to our example training set from figure 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. (Left) With very large λ , we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. (Center) With a medium value of λ , the learning algorithm recovers a curve with the right general shape. Even though the model is capable of representing functions with much more complicated shape, weight decay has encouraged it to use a simpler function described by smaller coefficients. (Right) With weight decay approaching zero (i.e., using the Moore-Penrose pseudoinverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in figure 5.2

Limiting the size of the weights

- The standard L2 weight penalty involves adding an extra term to the cost function that penalizes the squared weights.
 - This keeps the weights small unless they have big error derivatives.

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

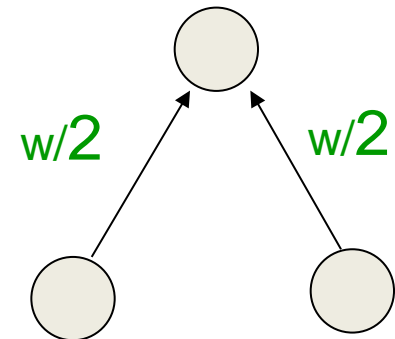
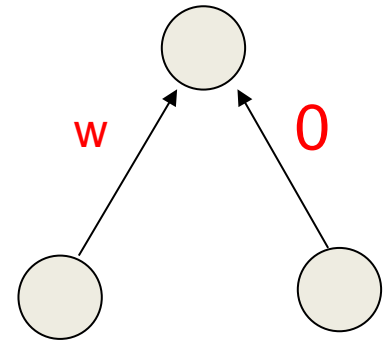
$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$



$$\text{when } \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}$$

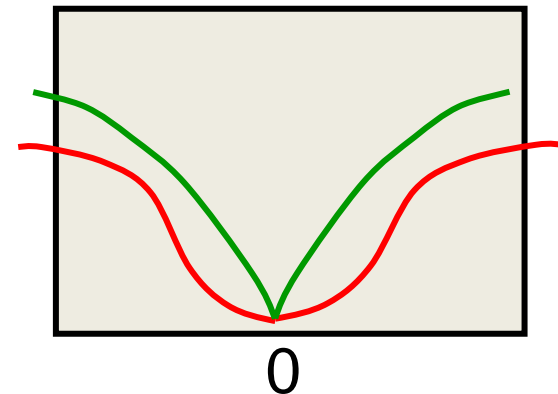
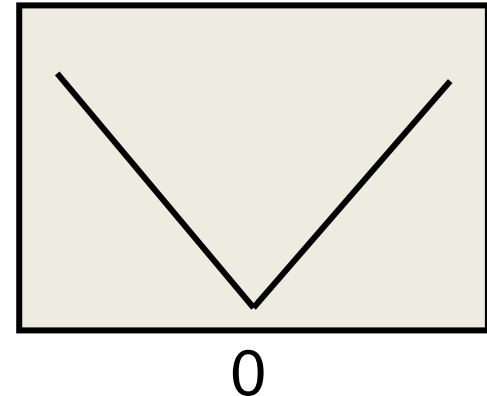
The effect of L2 weight cost

- It prevents the network from using weights that it does not need.
 - This can often improve generalization a lot because it helps to stop the network from fitting the sampling error.
 - It makes a smoother model in which the output changes more slowly as the input changes.
- If the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.



Other kinds of weight penalty

- Sometimes it works better to penalize the absolute values of the weights.
 - This can make many weights exactly equal to zero which helps interpretation a lot.
- Sometimes it works better to use a weight penalty that has negligible effect on large weights.
 - This allows a few large weights.



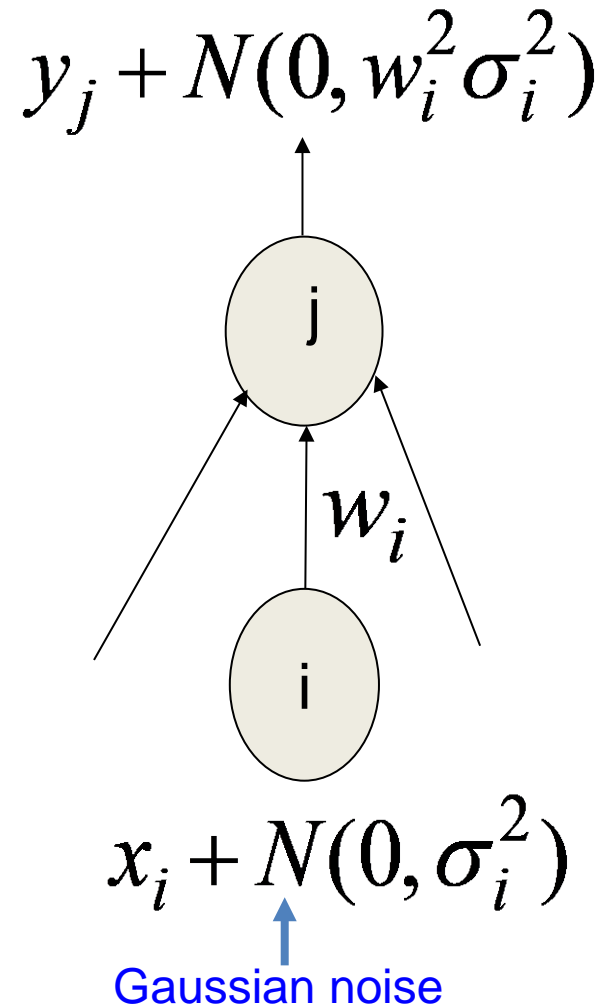
Weight penalties vs weight constraints

- We usually penalize the squared value of each weight separately.
- Instead, we can put a constraint on the maximum squared length of the incoming weight vector of each unit.
 - If an update violates this constraint, we scale down the vector of incoming weights to the allowed length.
- Weight constraints have several advantages over weight penalties.
 - Its easier to set a sensible value.
 - They prevent hidden units getting stuck near zero.
 - They prevent weights exploding.
- When a unit hits it's limit, the effective weight penalty on all of it's weights is determined by the big gradients.
 - This is more effective than a fixed penalty at pushing irrelevant weights towards zero.

4d. Adding noise
(as a regularizer)

L2 weight-decay via noisy inputs

- Suppose we add Gaussian noise to the inputs.
 - The variance of the noise is amplified by the squared weight before going into the next layer.
- In a simple net with a linear output unit directly connected to the inputs, the amplified noise gets added to the output.
- This makes an additive contribution to the squared error.
 - So minimizing the squared error tends to minimize the squared weights when the inputs are noisy.



output on one case $\rightarrow y^{noisy} = \sum_i w_i x_i + \sum_i w_i \varepsilon_i$ where ε_i is sampled from $N(0, \sigma_i^2)$

$$E[(y^{noisy} - t)^2] = E\left[\left(y + \sum_i w_i \varepsilon_i - t\right)^2\right] = E\left[\left((y - t) + \sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y - t)^2 + E\left[2(y - t) \sum_i w_i \varepsilon_i\right] + E\left[\left(\sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y - t)^2 + E\left[\sum_i w_i^2 \varepsilon_i^2\right]$$

*because ε_i is independent of ε_j
and ε_i is independent of $(y - t)$*

$$= (y - t)^2 + \sum_i w_i^2 \sigma_i^2$$

So σ_i^2 is equivalent to an L2 penalty

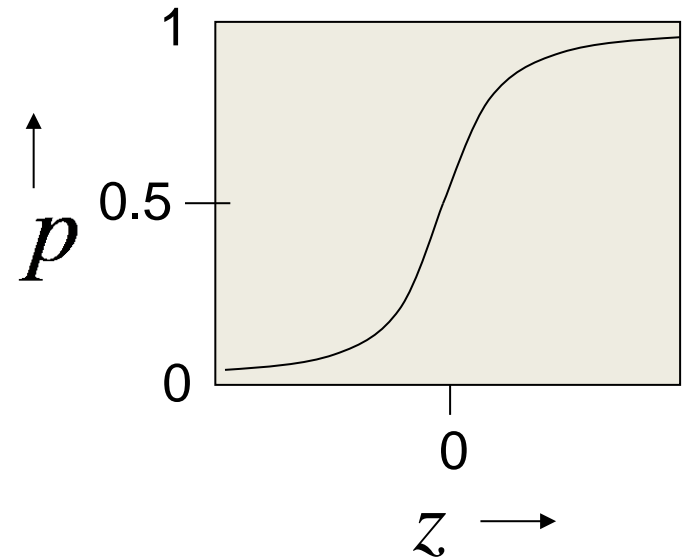
Noisy weights in more complex nets

- Adding Gaussian noise to the weights of a multilayer non-linear neural net is not exactly equivalent to using an L2 weight penalty.
 - It may work better, especially in recurrent networks.
 - Alex Graves' recurrent net that recognizes handwriting, works significantly better if noise is added to the weights.

Using noise in the activities as a regularizer

- Suppose we use backpropagation to train a multilayer neural net composed of logistic units.
 - What happens if we make the units binary and stochastic on the forward pass, but do the backward pass as if we had done the forward pass “properly”?
- It does worse on the training set and trains considerably slower.
 - But it does significantly better on the test set! (unpublished result).

$$p(s = 1) = \frac{1}{1 + e^{-z}}$$



4e. Prior distribution on params
(Bayesian fitting)

The Bayesian framework

- The Bayesian framework assumes that we always have a prior distribution for everything.
 - The prior may be very vague.
 - When we see some data, we combine our prior distribution with a likelihood term to get a posterior distribution.
 - The likelihood term takes into account how probable the observed data is given the parameters of the model.
 - It favors parameter settings that make the data likely.
 - It fights the prior
 - With enough data the likelihood terms always wins.

A coin tossing example

- Suppose we know nothing about coins except that each tossing event produces a head with some unknown probability p and a tail with probability $1-p$.
 - Our model of a coin has one parameter, p .
- Suppose we observe 100 tosses and there are 53 heads.
What is p ?
- The frequentist answer (also called maximum likelihood):
Pick the value of p that makes the observation of 53 heads and 47 tails most probable.
 - This value is $p=0.53$

A coin tossing example: the math

probability of
a particular
sequence
containing 53
heads and 47
tails.

$$\rightarrow P(D) = p^{53} (1-p)^{47}$$

$$\frac{dP(D)}{dp} = 53p^{52} (1-p)^{47} - 47p^{53} (1-p)^{46}$$

$$= \left(\frac{53}{p} - \frac{47}{1-p} \right) \left[p^{53} (1-p)^{47} \right]$$

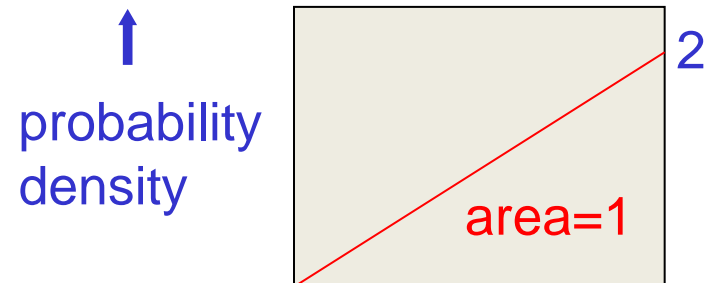
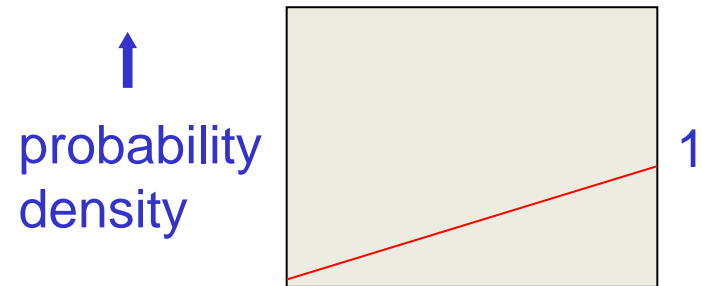
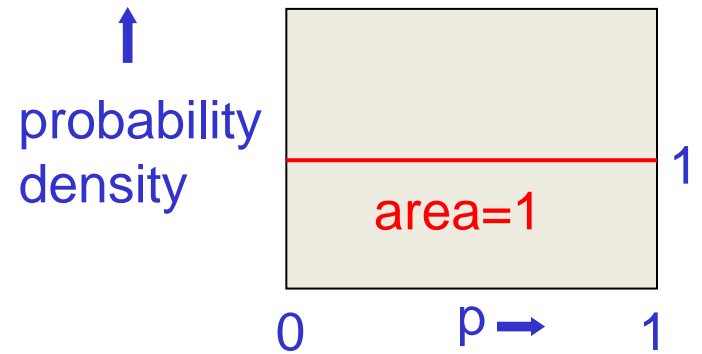
$$= 0 \text{ if } p = .53$$

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

- What if we only tossed the coin once and we got 1 head?
 - Is $p=1$ a sensible answer?
 - Surely $p=0.5$ is a much better answer.
- Is it reasonable to give a single answer?
 - If we don't have much data, we are unsure about p .
 - Our computations of probabilities will work much better if we take this uncertainty into account.

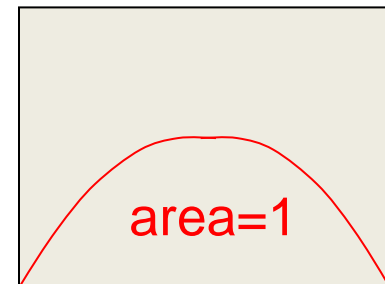
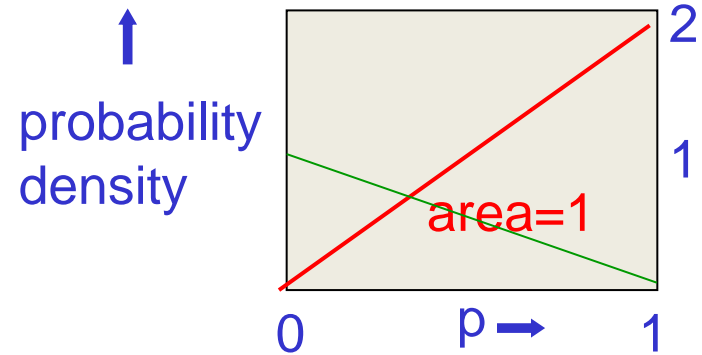
Using a distribution over parameter values

- Start with a prior distribution over p . In this case we used a uniform distribution.
- Multiply the prior probability of each parameter value by the probability of observing a head given that value.
- Then scale up all of the probability densities so that their integral comes to 1. This gives the posterior distribution.



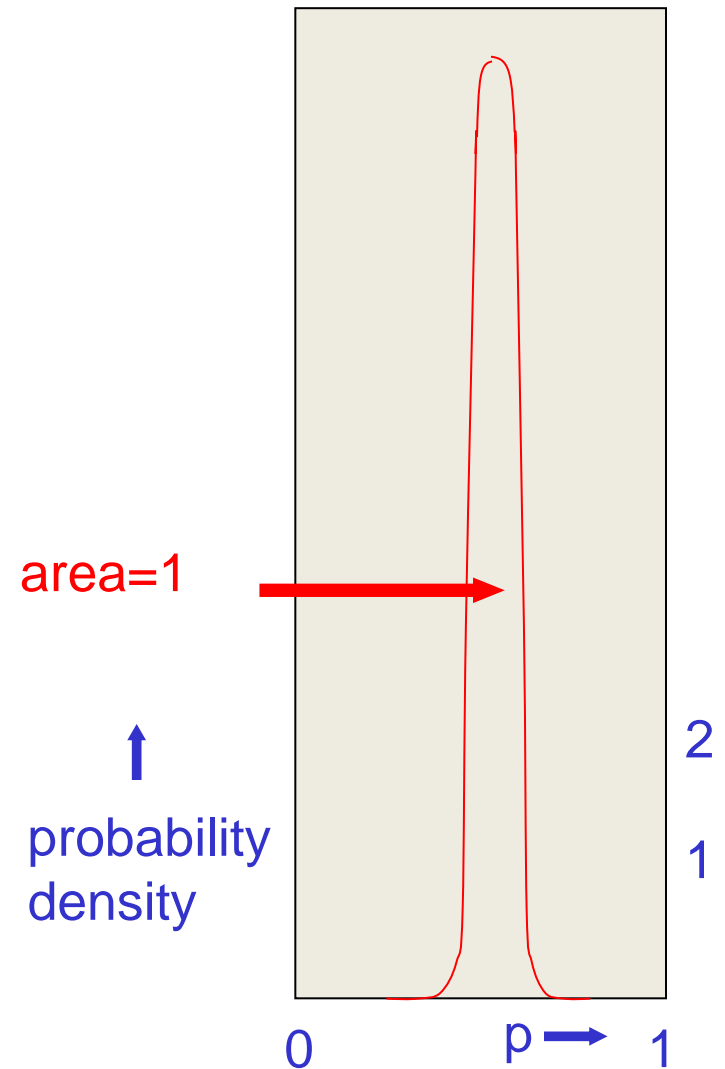
Lets do it again: Suppose we get a tail

- Start with a prior distribution over p .
- Multiply the prior probability of each parameter value by the probability of observing a **tail** given that value.
- Then renormalize to get the posterior distribution.
Look how sensible it is!



Lets do it another 98 times

- After 53 heads and 47 tails we get a very sensible posterior distribution that has its peak at 0.53 (assuming a uniform prior).



Bayes Theorem

joint probability

conditional probability

$$p(D)p(W | D) = p(D, W) = p(W)p(D | W)$$

prior probability of weight vector W

probability of observed data given W

$$p(W | D) = \frac{p(W) p(D | W)}{p(D)}$$

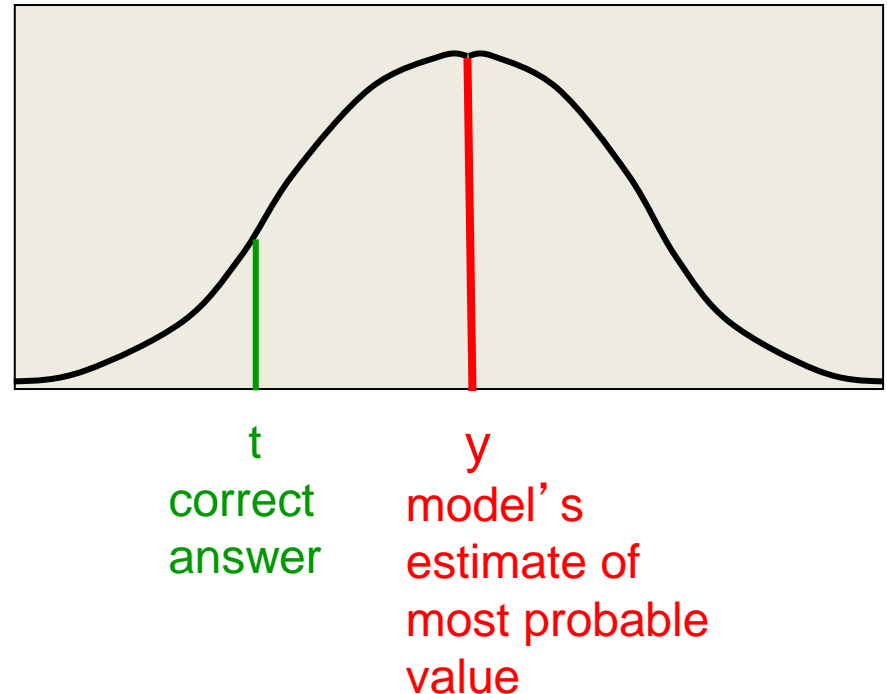
posterior probability of weight vector W given training data D

$$\int_W p(W)p(D | W)$$

4c+4e. Why weight decay is
Bayesian regularization

Supervised Maximum Likelihood Learning

- Finding a weight vector that minimizes the squared residuals is equivalent to finding a weight vector that maximizes the log probability density of the correct answer.
- We assume the answer is generated by adding Gaussian noise to the output of the neural network.



Supervised Maximum Likelihood Learning

output of the net $\rightarrow y_c = f(\text{input}_c, W)$

probability
density of the
target value
given the net's
output plus
Gaussian noise

$$p(t_c | y_c) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(t_c - y_c)^2}{2\sigma^2}}$$

Gaussian
distribution
centered at the
net's output

Cost $\rightarrow -\log p(t_c | y_c) = k + \frac{(t_c - y_c)^2}{2\sigma^2}$

Minimizing squared
error is the same as
maximizing log prob
under a Gaussian.

MAP: Maximum a Posteriori

- The proper Bayesian approach is to find the full posterior distribution over all possible weight vectors.
 - If we have more than a handful of weights this is hopelessly difficult for a non-linear net.
 - Bayesians have all sort of clever tricks for approximating this horrendous distribution.
- Suppose we just try to find the most probable weight vector.
 - We can find an optimum by starting with a random weight vector and then adjusting it in the direction that improves $p(W | D)$.
 - But it's only a local optimum.
- It is easier to work in the log domain. If we want to minimize a cost we use negative log probs

Why we maximize sums of log probabilities

- We want to maximize the **product** of the probabilities of the producing the target values on all the different training cases.
 - Assume the output errors on different cases, c , are independent.

- $p(D | W) = \prod p(t_c | W) = \prod p(t_c | f(\text{input}_c, W))$
Because the log function is monotonic, it does not change where the maxima are. So we can maximize **sums** of log probabilities

$$\log p(D | W) = \sum_c \log p(t_c | W)$$

MAP: Maximum a Posteriori

$$p(W | D) = \frac{p(W) p(D | W)}{p(D)}$$



$$Cost = -\log p(W | D) = -\log p(W) - \log p(D | W) + \log p(D)$$



log prob of
W under
the prior



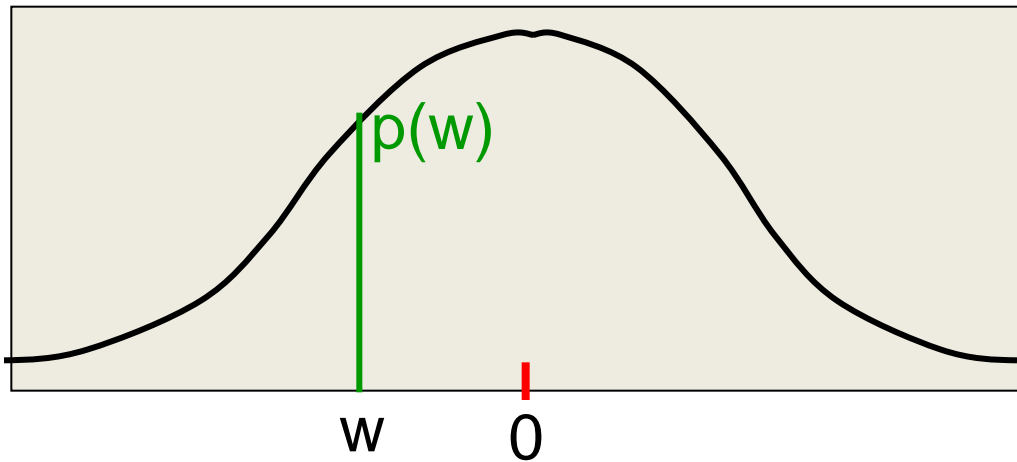
log prob
of target
values
given W



This is an integral over
all possible weight
vectors so it does not
depend on W

The log probability of a weight under its prior

- Minimizing the squared weights is equivalent to maximizing the log probability of the weights under a zero-mean Gaussian prior.




$$p(w) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{w^2}{2\sigma^2}}$$



$$-\log p(w) = \frac{w^2}{2\sigma^2} + k$$

The Bayesian interpretation of weight decay

$$-\log p(W | D) = -\log p(D | W) - \log p(W) + \log p(D)$$



$$C^* = \frac{1}{2\sigma_D^2} \sum_c (y_c - t_c)^2 + \frac{1}{2\sigma_W^2} \sum_i w_i^2$$

assuming that the model makes a Gaussian prediction assuming a Gaussian prior for the weights

constant

$$C = E + \frac{\sigma_D^2}{\sigma_W^2} \sum_i w_i^2$$

So the correct value of the weight decay parameter is the ratio of two variances. It's not just an arbitrary hack.

4f. Variance of residual errors
(MacKay's quick and dirty method)

Estimating the variance of the output noise

- After we have learned a model that minimizes the squared error, we can find the best value for the output noise.
 - The best value is the one that maximizes the probability of producing exactly the correct answers after adding Gaussian noise to the output produced by the neural net.
 - The best value is found by simply using the variance of the residual errors.

Estimating the variance of the Gaussian prior on the weights

- After learning a model with some initial choice of variance for the weight prior, we could do a dirty trick called “empirical Bayes”.
 - Set the variance of the Gaussian prior to be whatever makes the weights that the model learned most likely.
 - i.e. use the data itself to decide what your prior is!
 - This is done by simply fitting a zero-mean Gaussian to the one-dimensional distribution of the learned weight values.
 - We could easily learn different variances for different sets of weights.
- We don't need a validation set!

MacKay's quick and dirty method of choosing the ratio of the noise variance to the weight prior variance.

- Start with guesses for both the noise variance and the weight prior variance.
- While not yet bored
 - Do some learning using the ratio of the variances as the weight penalty coefficient.
 - Reset the noise variance to be the variance of the residual errors.
 - Reset the weight prior variance to be the variance of the distribution of the actual learned weights.
- Go back to the start of this loop.

Recurrent Neural Networks (RNNs) + Generalization

1. How do you read/listen/understand/write? Can machines do that?

- Context matters: characters, words, letters, sounds, completion, multi-modal
- Predicting next word/image: from unsupervised learning to supervised learning

2. Encoding temporal context: Hidden Markov Models (HMMs), RNNs

- Primitives: hidden state, memory of previous experiences, limitations of HMMs
- RNN architectures, unrolling, back-propagation-through-time (BPTT), param reuse

3. Vanishing gradients, Long-Short-Term Memory (LSTM)

- Key idea: gated input/output/memory nodes, model choose to forget/remember
- Example: online character recognition with LSTM recurrent neural network

4. Improving generalization

- **More training data**
- **Tuning model capacity**
 - Architecture: # layers, # units
 - Early stopping: (validation set)
 - Weight-decay: L1/L2 regularization
 - Noise: Add noise as a regularizer
- **Bayesian prior on parameter distribution**
- **Why weight decay \Leftrightarrow Bayesian prior**
- **Variance of residual errors**