Lecture 2 (and more): Improving Neural Network Training

Reading and Research in Deep Learning

James K Baker

Fill Out a Card

- Name
- Department
- Class year (Fresh, Soph, etc.)
- Languages you know (read or speak) and degree of fluency
  - Include dialects of Chinese and regional languages of India
- Do you want to be of the DNN364 coding team (yes, no, maybe)
- Topic preferences for presentation or coding (see next slide)
- Are you interested in a follow-on?
  - Summer employment
  - Open source software project
  - Research project
  - Another course
  - Other
- Other comments

Select Your Preferred Topics

Syllabus	
Generating artificial data	Lewis 4, Nielsen 3
MSE Cost	Nielsen1, 2, Lewis 5
Data preparation	Lewis 4, 5, Brownlee 11, 12
Tuning topology	Brownlee 11,12
Regression	Lewis 5, Brownlee 12
Cross-entropy cost	Nielsen 3
Softmax	Nielsen 3, Lewis 9
Log-likelihood cost	Nielsen 3
Early stopping	Nielsen 3
Holdout	Lewis 7
Cross-validation	
Learning rate	Lewis 5,Nielsen 1, 2, 3
ReLU	Lewis 6,Nielsen 6
Momentum	Lewis 7, Neilsen 6
Nestorov	Lewis 9, Nielsen 6
rmsprop	Lewis 9
adagrad	Lewis 9
adam	
dropout	Lewis 8, Nielsen 3, Brownlee 16
l2 regularization	Lewis 6, Nielsen 3
l1 regularization	Nielsen 3
minibatch size	Nielsen 3
experimenting with hyperparameters	Nielsen 3, Brownlee 9,10
learning rate schedules	Nielsen 3, Bownley 17
Convolutional neural networks	Nielsen 6, Brownlee 18,19
Image augmentation	Brownlee 20
Object Recognition	Brownlee 21
Recurrent neural networks, BPTT, LSTM	Brownlee 23
Time series predcitions	Lewis 5, Brownlee 24
Multiplicative Nodes	
LSTM Time Series	Brownlee 25
Sequence classification	Brownlee 26
Stateful LSTM RNNs	Brownlee 27
Text generation	Brownlee 28
gated units	
autoencoder	
unsupervised training	

If you plan to do research papers as your major project, list your first and second choice of topics for presentation. If a topic is popular, it will be a team presentation to which I will assign extra. List more if you wish to present on more than one topic.

If you plan to be on the DNN364 coding team, list the five topics to which you would be most interested to contribute code. If you also want to volunteer to give a tutorial presentation, list that separately.

- Name
- Department
- Class year (Fresh, Soph, etc.)
- Languages you know (read or speak) and degree of fluency
  - Include dialects of Chinese and regional languages of India
- Do you want to be of the DNN364 coding team (yes, no, maybe)
- Topic preferences for presentation or coding (see next slide)
- Are you interested in a follow-on?
  - Summer employment
  - Open source software project
  - Research project
  - Another course
  - Other
- Other comments

Class Discussion Notes

- For each class session, we need an official note-taker
  - Record questions
    - Also record name of questioner
  - Notes on discussions
    - Also record name of each speaker
  - When you are about to speak, please stand and introduce yourself
    - State your name and what department you are in
    - Later, also state your project team
- I will ask for a volunteer for each class, appoint someone if there is no volunteer

Review from Lecture 1: Derivation of Backprop Equations

- Derivative of cost with respect to $z_{L,j}$
- Derivative of cost with respect to $w_{l-1,i,j}, given\ \delta_{l,j}$
- Derivation of $\delta_{l,j}$
- The backpropagation equations
- Training update equations

Change in Error for Change in an Output Node

- $C = C_{MSE} = (1/n_L)\sum_{j=1}^{n_L}(y_j - o_j)^2$
  - $o_j = a_{L,j}$
  - $\frac{\partial C}{\partial a_{L,j}} = -2(y_j - o_j)/n_L = -2(y_j - a_{L,j})/n_L$
  - By the chain rule,
  - $\frac{\partial C}{\partial z_{L,j}} = \frac{\partial C}{\partial a_{L,j}}\frac{\partial a_{L,j}}{\partial z_{L,j}}$
  - $\frac{\partial C}{\partial z_{L,j}} = -2(y_k - a_{L,j})\frac{\partial a_{L,j}}{\partial z_{L,j}}/n_L$
  - $\frac{\partial C}{\partial z_{L,j}} = -2(y_k - a_{L,j})(\sigma(z_{L,j})(1 - \sigma(z_{L,j}))/n_L$

y is a constant. $\frac{\partial(c-x)^2}{\partial x} = -2(c-x)$

This part will be different for different cost functions, but the rest of the backpropagation will be the same.

This is just the derivative of the sigmoid.

Proceeding Backwards, Layer by Layer

- It is convenient to represent each layer with the input $z_{l,j}$ to each node j rather than by its output $a_{l,j}$
  - In particular, let $\delta_{l,j} = \frac{\partial C}{\partial z_{l,j}}$
- Then $\frac{\partial C}{\partial w_{l,i,j}} = \frac{\partial z_{l,j}}{\partial w_{l,i,j}} \delta_{l,j} = \frac{\partial (\sum_{k=1}^{n_l} a_{l-1,k} w_{l,k,j})}{\partial w_{l,i,j}} \delta_{l,j}$

This is just substituting for $z_{l,j}$.

This equation is true for any l, but we don't yet know $\delta_{l,j}$ for $l < L$.

Note that $w_{l,i,j}$ only appears in one term.

The simple form of this equation is the reason for working with $\delta_{l,j}$.

- $\frac{\partial C}{\partial w_{l,i,j}} = a_{l-1,i} \delta_{l,j}$

- Note that we have already computed $\frac{\partial C}{\partial z_{L,j}} = \delta_{L,j}$
- We just need to compute backward from $\delta_{l,*}$ to $\delta_{l-1,*}$

Working Backward from $\delta_{l,*}$ to $\delta_{l-1,*}$

We already know $\delta_{L,*}$

- For all j, $z_{l,j} = \sum_{k=0}^{n_{l-1}} a_{l-1,k} w_{l,k,j}$, therefore each $a_{l-1,k}$ contributes to $z_{l,j}$ for all j for which $w_{l,k,j} \neq 0$
  - $\frac{\partial C}{\partial a_{l-1,i}} = \sum_{j=1}^{n_l} w_{l,i,j} \delta_{l,j}$

    Note that we are summing over j even though the activation sums over i. This sum results from the fact that $a_{l-1,i}$ contributes to $a_{l,j}$ for all j.
  - $\frac{\partial C}{\partial z_{l-1,i}} = \frac{\partial C}{\partial a_{l-1,i}} \frac{\partial \sigma(z_{l-1,i})}{\partial z_{l-1,i}}$
  - $\delta_{l-1,i} = \frac{\partial C}{\partial z_{l-1,i}} = \sigma(z_{l-1,i})(1 - \sigma(z_{l-1,i})) \frac{\partial C}{\partial a_{l-1,i}}$

    We would be summing over j even if each node j only had one arc coming into it so that $a_{l,j}$ did not have a summation.
  - $\delta_{l-1,i} = a_{l-1,i}(1 - a_{l-1,i}) \sum_{j=1}^{n_l} w_{l,i,j} \delta_{l,j}$

The Backpropagation Equations

- $\delta_{L,j} = -2(y_k - a_{L,j})(\sigma(z_{L,j})(1 - \sigma(z_{L,j}))/n_L$
- $\delta_{l-1,i} = a_{l-1,i}(1 - a_{l-1,i}) \sum_{j=1}^{n_l} w_{l,i,j}\delta_{l,j}$
- $\frac{\partial C}{\partial w_{l,i,j}} = a_{l-1,i}\delta_{l,j}$

Summing Across All the Training Data

- The preceding computations were for a single input-output pair in the training data, but $C(m), a_{l-1,i,j}, (m) and\ \delta_{l,j}(m)$ all vary with the training item m (the weights $w_{l,i,j}$ are held constant for each iteration of training). Therefore, the gradient of the cost function for a training iteration is the sum of the gradient estimates for the individual training items.

- $\frac{\partial C_{Avg}}{\partial w_{l,i,j}} = \sum_{m=1}^{M} \frac{\partial C(m)}{\partial w_{l,i,j}} / M = \sum_{m=1}^{M} a_{l-1,i}(m)\delta_{l,j}(m)/M$

- For each iteration, each $w_{l,i,j}$ is changed by a small amount proportional to the negative of the partial derivative of the cost with respect to the weight
  - $\Delta w_{l,i,j} = -\eta \frac{\partial C_{Avg}}{\partial w_{l,i,j}}$, where η is a parameter that controls the learning rate

Minibatches and Stochastic Gradient Descent

- Summing over all the training date gives us the gradient of the cost on the training data, so the gradient descent procedure will converge
  - However, this is only a statistical estimate of the true cost. The cost is defined as the average of the cost function across all the training data. The average cost on any one minibatch is a random variable that only approximates the cost on the total training set.
- The amount of computation per iteration of training is proportional to the amount of data that we sum together
  - We can get many more iterations of training (i.e. more updates of the estimated weights) by updating the weights after summing over a much smaller batch of training data (called a "minibatch")
- Using a minibatch means that the gradient estimate is only a statistical estimate of the gradient on the training data, so the gradient descent only converges in a statistical sense. Therefore, this procedure is called "stochastic gradient descent".
  - Because there are many more updates per pass through the training set, in practice it usually takes much less computation than batch updating using the whole training set.
- If we are using minibatches, the symbol M will mean the number of examples in the current minibatch. The total number of examples in the training data will be written as M_{Total}.

Initialize the Weights

- Initialize the weights at random
  - Make negative and positive weights equally likely
  - Make the weights large enough so that they matter
    - That is, so that many of the nodes are activated differently for different data examples (so, not all weights equal to zero)
  - Make the weights small enough so that for most data very few of the nodes
    - The more arcs that lead to a node, the more likely it is that random inputs with random weights will saturate the node
      - Make standard deviation of the weights smaller for arcs that go to a node with a large number of incoming arcs
  - Suggestion
    - Choose weights from a Gaussian distribution with mean zero and a standard deviation equal to $\sqrt{n}$, where n is the number of arcs leading to the same node.

Training Algorithm Iteration

1. **Input a set (minibatch) of training examples**
2. **For each training example m,** set $a_{0,i}(m)$ and perform the following steps:
   a. **Feedforward:** For each $l = 1, 2, \ldots, L$ compute
   $$z_{l,j}(m) = \sum_{i=0}^{n_l} w_{l-1,i,j} a_{l-1,i}(m), \quad a_{lj}(m) = \sigma(z_{l,j}(m))$$
   b. **Output error $\delta_{L,j}(m)$:**
   $$\delta_{L,j}(m) = -2(y_j(m) - a_{L,j}(m))(\sigma(z_{L,j}(m))(1 - \sigma(z_{L,j}(m))/n_L$$
   c. **Backpropagate error:** For each l = L-1,L-2,...,2, 1 compute
   $$\delta_{l-1,i}(m) = a_{l-1,i}(m)(1 - a_{l-1,i}(m)) \sum_{j=1}^{n_l} w_{l,i,j} \delta_{l,j}(m)$$
3. **Gradient descent:** For each l = L-1,L-2,...,2, 1 update the weights
   $$w_{l,i,j} \rightarrow w_{l,i,j} - \eta \sum_{m=1}^{M} a_{l-1,i}(m) \delta_{l,j}(m)/M$$

Sum the estimated partial derivatives across all the examples in the minibatch.

Go through all the training data separated into minibatches multiple times. Continue training iterations until a stopping criterion is met. It is recommended that you sample the data in random order.

Refinements will be added to improve learning speed and reliability.

Terminology: Update, Epoch, Iteration

- The learning algorithm updates the estimated parameters once for every mini-batch.
  - We need a name for this. Let's call it an "update".
- The standard implementation runs through many mini-batches until all the training data has been used. This is called an "epoch".
  - From usage in a larger classes of iterative optimization procedures, I often call this an "iteration".
- The learning runs multiple epochs or iterations until some stopping criterion is met
  - There may be thousands of epochs
- Sometimes I forget and call an update an "epoch".

Why we study the sigmoid first

- It came first and is still in wide spread use
  - You need it to understand the original backpropagation papers
  - You need it for papers from the 80s, 90s, and 00s
  - And for some current papers
- It is related to softmax, which is used for the output layer for classification (linear output is for regression)
- Sigmoid itself is also sometimes used for the output layer
  - Especially for binary classification and feature detection
- The [0,1] range fits with probability interpretations
- It is a closer fit to neuron models than is the rectified linear unit (reLU)
- Our textbooks start with it
- It's fun to have a crazy looking derivative $\sigma' = \sigma(1 - \sigma)$

Improving the Learning Performance (Multiple Mini-Lectures)

- There will be a number of techniques for improving the performance of the neural network learning
- Many of these can be applied independently of each other
- Most of these techniques improve the rate of convergence or reduce computation is some other way
  - Rather than improving the accuracy of the final answer
- Because the techniques are relatively independent, this PowerPoint presentation will be broken into a number of mini lectures at convenient stopping points
  - You are encouraged to use this presentation as a reference
  - You can read ahead
    - You should do so if a mini project uses a technique we haven't yet discussed in class

Improvement Techniques

- Holdout and cross-validation
- Cross-entropy cost function
- Softmax
  - Log-likelihood cost function
- Regularization
  - L2
  - L1
  - Dropout
  - Artificial expansion of training data
- Initializing weights
- Heuristics to set hyperparameters
- Other techniques

Problem: Testing our results.

Problem: Slow learning.

Problem: Multi-class learning.

Problem: Overfitting.

Problem: Getting stuck in a local minimum or region of slow learning. (Already covered in Lecture 1)

Problem: Dealing with the many parameters that control the training.

These topics are largely independent of each other. I may discuss them out of order to fit the time available.

The Need to Holdout Validation Data

- In pattern recognition you should never train on your test data
  - Any performance results will be invalid
- Similarly, if during development you make decisions based on performance on training data, you ~~may~~ will be seriously mislead
- You should holdout a substantial amount of data from the training to use for validation testing during development

This principle is extremely important!! Make sure you never forget!

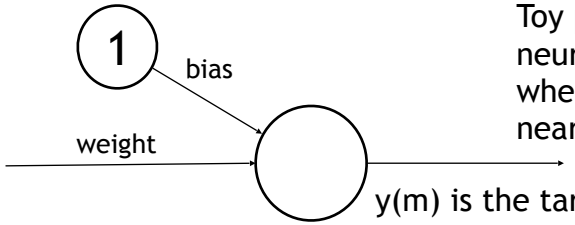
Cross-Validation

- Break the training data into K subsets
- Cross-validation
  - Chose one subset as holdout data
  - Train on the other K-1 subsets and validate on the holdout subset
  - Repeat with each of the K subsets as the holdout

Multiple Holdouts

- If during development you have made significant improvements in performance, you should do validation on a new holdout set that hasn't been used before.
- In fact it is prudent to replicate your results on completely new data (new training data as well as new validation data), if possible.
- This especially important if you are developing a commercial product that will continue to be used and improved for many years.
- Unfortunately, very few R&D groups live up to this standard.

Problem: Sometimes Gradient Descent is Very Slow Even When the Current Answer is Wrong



Toy problem: Train a single neuron to have an output near 1 when it initially has an output near 0.

y(m) is the target (either 0 or 1); a(m) is the activation

Output (or only) layer $\delta_{L,j}(m)$:

$$\delta_{L,j}(m) = -2(y_j(m) - a_{L,j}(m))(\sigma(z_{L,j}(m))(1 - \sigma(z_{L,j}(m))/n_L$$

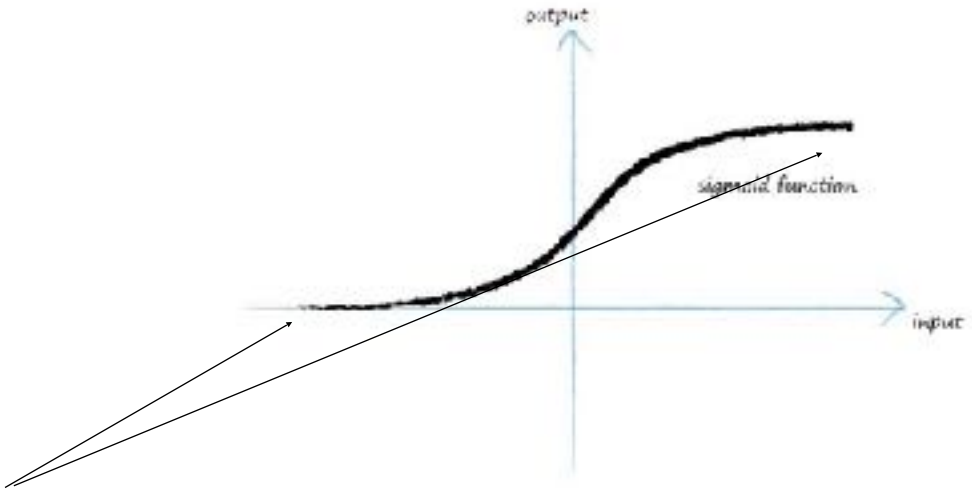
Backpropagate cost: For each l = L-1,L-2,…,2, 1 compute

$$\delta_{l-1,i}(m) = a_{l-1,i}(m)(1 - a_{l-1,i}(m)) \sum_{j=1}^{n_l} w_{l,i,j}\delta_{l,j}(m)$$

The derivatives of the cost function are very small when the activation is close to 0 or 1.

Gradient descent can be slow even in this simple toy problem. In particular, it is slow when the current model is very far from the correct answer, when we would like the learning to be fast and aggressive.

The Derivative of the Sigmoid is Close to Zero When Its Value is Close to Zero or One



Its rate of change saturates at the ends.

This causes the learning to be slow in either of these conditions.

This smooth S-shaped sigmoid function is what we'll be continue to use for making our own neural network. Artificial intelligence researchers will also use other, similar looking functions, but the sigmoid is simple and actually very common too, so we're in good company.

The sigmoid function, sometimes also called the **logistic function**, is

$$y = \frac{1}{1 + e^{-x}}$$

Is the Mean Squared Error a Good Cost Function?

- $C = C_{MSE} = MeanSquaredError = (1/n_L) \sum_{k=1}^{n_{output}} (Error_k)^2$
- The MeanSquaredError is commonly used in statistics
  - It is an appropriate measure for regression (least squares regression) and for problems in which the error is expected to have a Gaussian distribution.
  - However, in classification problems, the target is in the range [0,1] and with sigmoid nodes for the output layer so is the output is also in the range [0,1], so a Gaussian model is not a good fit for the error distribution.
- Can we find a more appropriate cost function that also gives faster training when the activation of a node is at its extremes?

Consider the Logarithm of the Error Rate

- In trying to improve the performance of pattern recognition systems it is reasonable to look at the log of the error rate rather than the absolute error rate
  - As a practical matter for developers, the difficulty of making an improvement tends to be proportional to the relative percentage reduction
  - Typically, it is as difficult to lower an error rate from 2% to 1.5% as it is to lower the error rate from 20% to 15%
  - That is, the difficulty of making improvements is proportional to the logarithm of the ratio of the error rates
- Is there a differentiable function that weights probabilities proportional to the logarithm?
  - Yes, the entropy as defined in information theory

$$Entropy = \sum_{i=1}^{N} p_i \log(p_i)$$

How Can We Use Something Like Entropy as a Measure of Error?

- Cross-entropy is a measure of the deviation of an estimated probability q(x) distribution from the true distribution p(x)
- We have a finite sample of training examples, so the probability distributions are represented by summing across the examples
- For our single neuron example, it is defined as follows:

$$C = -\frac{1}{M}\sum_{m=1}^{M}(y(m)\log(a(m)) - (1 - y(m))\log(1\text{-}a(m)))$$

In practice, a(m) only approaches 0 or 1, so we don't have to worry about log(0).

If a(m) approaches the correct value of y(m) for every m, then either y(m) is 0 and log(1-a(m) approaches log(1) = 0, or y(m) is 1 and log(1-a(m)) approaches 0. In either case the sum approaches 0. On the other hand, C is always positive and has its maximum when y(m) and a(m) always disagree. Thus, cross-entropy, that is C, is a reasonable cost measure.

Cross-Entropy Avoids the Slow Learning

- The derivative of log(x) is 1/x. $a = \sigma(z), so$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{M}\sum_m \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{(1-\sigma(z))}\right)\frac{\partial C}{\partial w_j}$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{M}\sum_m \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{(1-\sigma(z))}\right)\sigma'(z)x_j$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{M}\sum_m \left(\frac{y - \sigma(z)}{\sigma(z)(1 - \sigma(z))}\right)\sigma'(z)x_j$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{M}\sum_m (y - \sigma(z))x_j$$

The dependency on m has been made implicit to make the expressions easier to read.

The learning no longer slows down when $\sigma(z)$ approaches 0 or 1 (except when it is correct).

Making Neuron Activations Simulate Probability Distributions

- A sigmoid neuron is like an event detector
  - The summed input to the neuron z is the evidence in favor of detecting the event
  - The output of the sigmoid function is in the range [0,1] and is somewhat like a probability
  - However, when there is more than one neuron their activations are not like a probability distribution
- Consider the simplest case: two neurons to represent a binary classification
  - Suppose we have separate evidence in favor of each alternative. How do we generalize the sigmoid function to reflect this information?

Binary Classification with Two Neurons

- Having separate evidence for each alternative is like having two neurons
  - Start by assuming that the only evidence that we have is the input to the single neuron, that is z. Then it is natural to use –z as the evidence for the opposite alternative. However, to simplify, we use 2z and -2z.

$$\sigma(2z) = \frac{1}{(1+e^{-2z})};\ \sigma(-2z) = \frac{1}{(1+e^{2z})}$$

$$\sigma(2z) = \frac{e^{z}}{(e^{z}+e^{-z})};\ \sigma(-2z) = \frac{e^{-z}}{(e^{-z}+e^{z})}$$

In this form, it is obvious that the two quantities sum to 1, like a probability distribution.

Binary Classification with Two Neurons that Have Different Evidence

- The obvious generalization is to let each neuron have its own evidence but then normalize to make the sum of the activations equal 1:
- $a_1 = \frac{e^{z_1}}{(e^{z_1}+e^{z_2})}; \quad a_2 = \frac{e^{z_2}}{(e^{z_1}+e^{z_2})};$

- The difference between this model and the sigmoid is that the sigmoid has only one node and implicitly assumes $z_2 = -z$. The full two-node model has twice as many parameters, with the advantages and disadvantages that that entails.

Generalizing to More Than Two Nodes

- There is a obvious generalization which leads to n nodes simulating a probability distribution

- $a_j = \frac{e^{z_j}}{(\sum_{k=1}^{n} e^{z_k})}, for\ j = 1,2, \ldots, n$

- Before this activation function came into common use, classification problems with more than two classes often used the Max function. That is, the class with the maximum activation was selected.
  - $max_j = 1\ z_j \geq\ z_k for\ all\ k, otherwise = 0$
- By comparison, the activation function above is called Softmax.
  - Softmax approximates Max because the largest z often dominates the sum.
  - But Softmax is differentiable.

Softmax

- $a_j = \frac{e^{z_j}}{(\sum_{k=1}^{n} e^{z_k})}, for\ j = 1,2, \dots, n$

- I prefer to think of it as simulating a probability distribution.
- I also think it is the natural generalization of the sigmoid to multiple nodes.
  - Note that a_j is a function not just its own input, so the Softmax appears to be qualitatively different from the sigmoid. However that difference is due to the sigmoid assuming $z_2 = -z_1$ (which removes the dependency of $a_1 on\ z_2$), rather than to a true qualitative difference in the meaning of the function.

Log-Likelihood Cost Function

- In an n-class classification problem when class k is the correct answer, $y_k = 1, y_j = 0 \; for \; j \neq k.$
- The log-likelihood cost function is

$$C = -\log(a_k)$$

$$\frac{\partial C}{\partial a_k} = -\frac{1}{a_k}$$

C only depends on a_k, which looks very simple and straight forward. In fact, how can it be right that the cost should ignore all the rest of the network?

Of course, the cost doesn't ignore the rest of the network because $a_k \; depends \; on \; all \; of \; the \; z_j \; , not \; just \; z_k.$

Because of the summation in the denominator

The Constraint that the Activations Sum to 1 Affects the Derivatives

- $\frac{\partial C}{\partial a_k} = -\frac{1}{a_k}; \ \frac{\partial C}{\partial a_j} = 0, j \neq k$
  - Let $s = \sum_{j=1}^{N_L} e^{z_j}$, $so\ a_k = \frac{e^{z_k}}{s}$
    - $then\ a_k depends\ on\ s\ as\ well\ as\ z_k$
    - s depends on all the z_j, for j=1, 2, …, N_L

Backpropagating the Log-Likelihood

This derivative is zero if j≠k hence the y_j.

- $\frac{\partial a_k}{\partial z_j} = \frac{\frac{\partial e^{z_k}}{\partial z_j} s - \frac{\partial s}{\partial z_j} e^{z_k}}{s^2}$
- $= \frac{y_j e^{z_k} s - e^{z_j} e^{z_k}}{s^2} = \frac{a_k (y_j s - e^{z_j})}{s} = a_k (y_j - a_j)$
- $\frac{\partial C}{\partial z_j} = (-\frac{1}{a_k}) \frac{\partial a_k}{\partial z_j} = a_j - y_j$
- $\frac{\partial C}{\partial w_{L-1,i,j}} = a_{L-1,i} (a_j - y_j)$

This factor only goes to zero with a correct answer, which is when we would like the derivative to approach 0.

Is Cross-Entropy the Same as Log-Likelihood?

- If not, which one is better?
- Softmax with cross-entropy is exactly the same as the two-node model with activations z and –z with log-likelihood.
  - Therefore the difference is entirely due to tying the activations of the two classes to being the negatives of each other, rather than to any difference in the cost function.
    - The two-node model has twice the parameters and therefore can better learn to fit data in which evidence for one class is not the same as evidence against the other
      - For example, consider two target classes selected out of a larger number. Each one would have a compact cluster around it.
    - On the other hand, if the situation really matches the assumption that any data for one class is negative evidence of the other, implicitly forcing the two subnetworks to be negative images of each other may give more robust training, with smoother estimates and more data per parameter

It is not a decision about cost function; it is a decision about whether to tie certain parameters.

Problem: Overfitting

- If you have too many parameters, your model may fit the training data very well but do poorly on new data
  - This is a problem for all pattern recognition, not just neural networks.
  - However, large deep learning networks have billions of parameters
- Possible solutions:
  - Collect more data (may be expensive)
  - Use fewer parameters (lose the ability to model complex interactions)
  - Other techniques
    - Regularization works with fixed network and fixed training data

L2 Regularization

Idea: Add an extra penalty term to the cost to prevent the weights from getting large.

- $C = C_0 + \frac{\lambda}{2M} \sum_{l=0}^{L-1} \sum_{i=1}^{N_l} \sum_{j=1}^{N_{l+1}} (w_{l-1,i,j})^2$

λ controls the relative impact of regularization compared to the regular cost function.

Any regular cost function

This is the regularization term. It makes the learning prefer smaller weights. It is hoped that weights that only fit idiosyncratic data examples will be driven to zero.

Notice that the regularization term only includes the regular connection weights, not the bias.

L2 regularization is also called “weight decay”.

L2 Regularized Gradient Descent

- **1. Gradient descent:** For each l = L-1,L-2,…,2, 1 update the weights

$$w_{l,i,j} \rightarrow w_{l,i,j} - \eta \sum_{m=1}^{M} a_{l-1,i}(m)\delta_{l,j}(m)/M$$

2. **L2 Regularized Gradient descent:** For each l = L-1,L-2,…,2, 1 update the weights

$$w_{l,i,j} \rightarrow w_{l,i,j}(1 - \frac{\eta\lambda}{n}) - \eta \sum_{m=1}^{M} a_{l-1,i}(m)\delta_{l,j}(m)/M$$

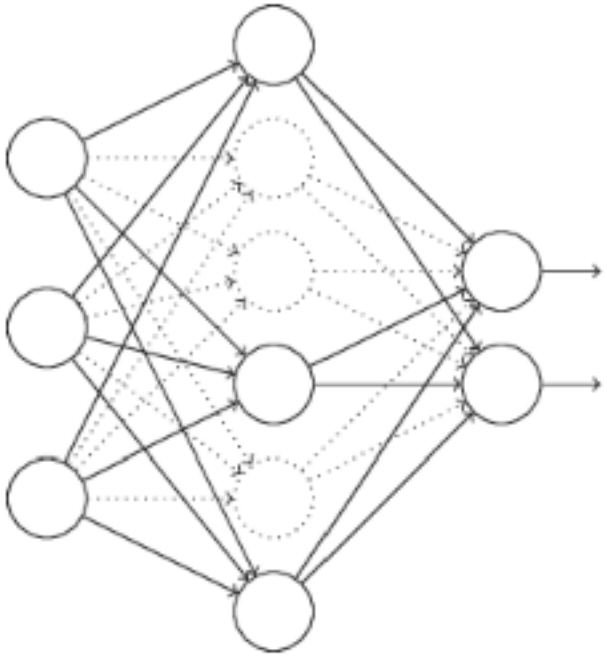
The effect of L2 regularization is to multiply the weight by $(1 - \frac{\eta\lambda}{n})$ before updating.

L1 Regularization

-
  - $c = c_0 + \frac{\lambda}{M}\sum_w |w|$
  - $w_{l,i,j} \rightarrow w_{l,i,j} - \frac{\eta\lambda}{n} sgn(w_{l,i,j}) - \eta \sum_{m=1}^{M} a_{l-1,i}(m)\delta_{l,j}(m)/M$

- L2 shrinks the weight towards 0 by an amount proportional to the weight
  - Shrinks large weights faster
- L1 shrinks the weight towards 0 by a constant amount
  - Drives small weights all the way to zero, making the set of weights more sparse

Dropout



Dropout is often very effective at preventing overfitting even for networks with more parameters than data examples. However, the actual way in which dropout prevents overfitting is somewhat mysterious. It doesn't always help for reasons that match the original justification, or any other particular theory.

Dropout: During the forward activation computation, pretend that some nodes (as many as half) don't exist. Randomly change which nodes are dropped out. Automatically, these nodes also won't participate in backprop.

Obviously, there are fewer parameters participating in any one activation. However, the total number of parameters remains the same, and they are all used to recognize of new data.