Neural Networks: Beyond the textbook!

Brian Hare

Spring 2018

Backpropagation

I'm going to present the basics of backpropagation, giving the relevant formulas and explaining what they mean. I'm not going to go through derivations of any of these, and you don't need to memorize the formulas, let alone apply them; I'm not going to ask you to number-crunch your way through a backpropagation problem. For this course you just need to understand the basic principles, and the ideas behind some of the optimizations. As a practical matter, there are enough predefined libraries out there, you don't need to write your own backpropagation code unless you want to do so for your own education or amusement.

We define the error ε at the output layer L for each node j as

$$\varepsilon_j^L = y_j^L - d_j$$

that is, the value we got at the output layer minus the value that was expected or desired at the output. This gives the error for each node at layer L. We then work backwards through each prior layer, one a time, with

$$\varepsilon_j^l = \sum_{k \in l+1} w_{kj}^{l+1} \delta_j^{l+1}$$

$$\text{where } \delta_j^l = \varepsilon_j^l f'(net)_j^l$$

The notation is convoluted because we have *n* nodes in each layer, each with connections (weights) going both directions, for an iterative algorithm. In the above equations, *net* is the weighted sum of inputs coming into a node. So f(*net*) is the sigmoid function, and f'(*net*) is the derivative of that function. Assume inputs are on the left, and "forward" is moving toward the outputs on the far right. We take the error at the output (the rightmost nodes). Then for each node in the previous layer, for each weight connecting that node to the node to its right, we take the error in that (right) node, multiplied by the derivative of that node's sigmoid. This product is the *local error* of this node (the small sigma). We multiply each local error by the corresponding weight connecting it to the node in the previous (next to the left) layer and sum it up for all weights on that node, to get the net error for that node.

Once errors are computed for layer L-1, we can then propagate backwards through layer L-2, and so on, all the way back to the input. In practice, we have to make some modifications, but this is the basic idea.

Now that we've assigned the error to each node (and each weight connecting them), we adjust each weight by

$$\Delta w_j^l = \eta \delta_j^l y_j^{l-1}$$

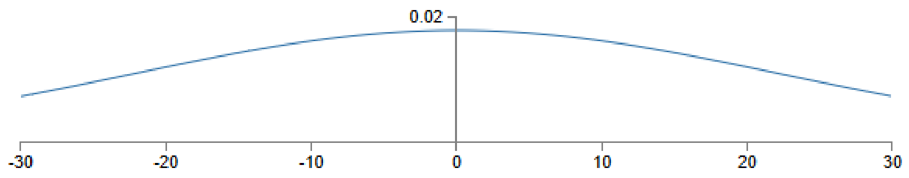
that is, a small value η (the *learning rate*) times the local error times the input at that node. Again, we're not going to worry about how this is derived; it involves finding the partial derivative of the error ε with respect to the weights, followed by some algebraic simplification.

Weights/biases

In general, we hope these would be fairly small. We can be more confident of the results if it's a sum of relatively small terms, rather than delicately balanced near-infinities ($0.01x_1 + 0.2x_2 – 0.31x_3 – 0.025 = 0.721$ rather than $1{,}462{,}092.921x_1 – 9{,}208{,}210x_2 + 20{,}311{,}029x_3 – 5{,}238{,}413.98 = 0.721$, for example). Bear in mind that the sigmoid function is most responsive (changes most rapidly in response to small changes in input) if it's near the midpoint; values get 'compressed' near the ends. Thus we start with small weights; there are a couple of popular choices, based on the size of the network. The basic principle is that the larger the network (the higher the number of connections), the smaller the starting weights should be. Biases diverging to relatively large values are less of a concern.

But of course, we'd like to have a system for setting weights better than "this is what we usually do because this is what we've usually done." Unfortunately, while we have some rules of thumb, the theory is a little light here. That's a recurring theme with neural networks; further discussion is below.

In addition to giving us more confidence in the results, the magnitude of the weights can have major effects on how quickly the network learns, or if it learns at all, particularly as input sizes become larger. Consider a network in which weights and biases are initialized to Gaussians with a variance of 1. For a particular hidden neuron, we have 1000 inputs, each of which is 0 or 1 with equal probability. (This is a special case chosen to illustrate the principle, but the results hold in general.) Then half the inputs will drop out, leaving the neuron's weighted sum to be the sum of 500 Gaussian weights + 1 Gaussian for the bias term. The sum of 501 Gaussian numbers with a variance of 1 is itself a Gaussian, with mean 0 and standard deviation$\sqrt{501} \approx 22.4$.[1]



Recall that for the sigmoid function, input values far from zero—those with an absolute value more than 3 or so—are in a region where the gradient is close to 0; the function is almost flat. This is referred to as the neuron being *saturated*. As can be seen from this graph, most values will produce sums that result in neurons being saturated, and thus exhibiting very slow learning. Making small changes in our weights will have almost no effect on the resulting sigmoid. The sigmoid value for a sum of 5 isn't that far from a sigmoid for a sum of 50. And unless you're working in very high precision, the sigmoid for a sum of 50 is indistinguishable from the sigmoid for a sum of 50,000; the difference between them will be quite small in any event. *As neurons become saturated, learning slows down.*

The solution is to initialize the weights to smaller values as the number of inputs increases. The most common method is to make the standard deviation of the initial weights inversely proportional to the square root of the number of inputs; that is, with a mean of 0 and standard deviation$1/\sqrt{n_{input}}$. Our sum will still be Gaussian, but much more sharply peaked, and much less likely to produce a saturated neuron. Under the same assumptions as before (leaving the biases with a standard deviation of 1), the weighted sum will have a Gaussian distribution with mean 0 and standard deviation$\sqrt{3/2} \approx 1.22$.If the network needs larger weights it can learn that, but we will start with less-saturated sigmoids. This can

[1]This example and image from http://neuralnetworksanddeeplearning.com/chap3.html. The discussion of the cross entropy function is largely taken from the same page.

allow a network to learn much faster (since a significant change in the sums will cause a significant change in the sigmoids) and in some cases can lead to better ultimate results. Another popular option is to initialize the weights uniformly in the interval ± 2.4/L, where L is the fan-in (number of incoming connections) for nodes in that layer. Again, the more connections, the smaller the initial weights. Note that these two methods, although derived via different methods, have similar statistical properties.

We don't need to worry about reducing the initial values for the biases because the bias is unlikely to make the neuron saturate. Of course, using a smaller value, or even initializing biases to 0 and let them be set entirely from learning, are also options. It doesn't seem to make much difference either way.

Implementation note: We've been acting as if biases are a separate property, dealt with separately. And in some implementations they are. But a common implementation method is to number the neurons from 1 to N, and add a neuron w_0, with the convention that the output from any such neuron on any layer is always 1. This allows the bias to be handled as just another weight, with no special handling of the learning required.

So what can go wrong?
In practice, almost everything. Plain vanilla backpropagation as presented above, by itself, hardly ever works. There are several reasons for this.

- Ultimately, each layer of a neural network is doing a very large, very complex set of linear regressions in a very high-dimensional space. Ideally, for regression you want at least 10 representative independent input/output pairs for each parameter. But a large deep network can have thousands to millions of weights, and we seldom have tens to hundreds of millions of inputs available. We can reduce this problem by using a smaller, simpler network. This gives up some potential power but reduces the risk of overtraining. (We don't have the data to unlock the power of a larger network, and are more likely to have problems with it.)
- No matter how much data we have, we always have a subset of reality. With enough data we can draw a very complex decision boundary—but it's still based on a subset of reality, and thus may not generalize as we'd like. It's always incomplete. Overfitting to limited data reveals itself in odd behavior or wildly incorrect classification when presented with unfamiliar data. This can be partially dealt with by *validation* strategies.
  - Part of the data—10-20% or so—is set aside in a lockbox and never seen by the network at all. The rest is divided up, usually about 80/20, with 80% used for training and the other 20% for validation—that is, it's run through the network forward to see how well it's classified, but not used for backpropagation. Seeing how the network does with data that hasn't been used to train it gives a validity check. Ability to fit the training data can always be improved.
  - Generally once the network is starting to overtrain, that is, memorize the specific training data, it's time to stop. This is detected by an increase in the error rate with the validation set. Once it starts to go up, *early stopping* is invoked and training ends.
  - But one good training run isn't enough! The 80/20 split is repeated, the network randomly re-initalized, and the whole process goes again. This allows assessment of whether the earlier results were a lucky (or unlucky) break. This is usually repeated several times:
  - In *bootstrapping*, we train, validate, put everything back, and repeat.
  - In *cross-fold validation*, we divide the data up into *k* sets, and hold each set out as validation data on one training run, and report the average of all *k* networks. Taken to its logical conclusion, this leads to the *leave out one* strategy, where there are as many sets as there are

data inputs. In classification problems, we often make each set contain one example of each category.

    - In either of these cases, what's reported is the average performance on the validation data.

  - The error surface is very high dimensional and has many local minima. Results can vary widely based on starting conditions. Thus we need to do many trials to see how the network 'really' performs. Networks can have similar results from much different topologies, or widely varying results from similar configurations.
  - Once all of this is done, and the network is as good as we can make it, then we take out the test data from the lockbox, see how the network does with it, and report the results as most likely what we'll get with real data. We do not make any further modifications to the network or choose between competing alternatives based on this data; otherwise we're still doing validation. We want our best estimate of how the network will do on 'pristine' data that it's never seen before.

- There's still the *vanishing gradient* problem. Remember that we multiply the local error times the derivative of the gradient. Some basic math will confirm that the sigmoid function has a maximum slope of 0.5. Its derivative is $f(x)(1-f(x))$, which means the derivative hass a maximum slope of 0.25. So by the time we multiply things by 0.25 through several layers, especially if those things are less than 1 themselves, we end up with gradients close to 0. This means almost no learning is occurring. This can be compensated by increasing the step size η as we move backwards through the network.
  - Other causes of the vanishing gradient can include:
  - We've simply hit a plateau, happening to choose a subset of the data that matches pretty well. The network stops learning because it thinks it's done. *Momentum learning* can address this. In momentum learning, we take bigger step sizes with each iteration as long as the sign of the gradient for that weight doesn't change. (Yes, this involves tracking the step size independently for each weight.) It's increased by a fixed amount each iteration, but reduced fairly aggressively (cut in half or so) if the sign of the gradient flips. This corresponds to taking bigger and bigger steps downhill until we overshoot the minimum, then proceed more slowly in the opposite direction—an *adaptive step size*. This helps us move past a shallow local minimum and get to (hopefully) a deeper minimum elsewhere, but it does increase the risk of overshooting past the true minimum.
  - On a plateau, we can bounce from one minimum to another and never stop training. This can be avoided by remembering the previous update and multiplying it by a constant < 1, and adding it to the current update. This provides a decaying exponential average of previous updates, so it remembers some prior steps; if they're all in the same direction, stepsize will gradually increase. Overshooting leads to a reduction in stepsize.
  - One approach to a network that seems to have stopped learning (or to provide a greater variety of input) is to add some noise to the data. This can be done by adding noise to the expected outputs—for example, adding a small Gaussian to the expected value for each output neuron in a classification problem—and then letting it propagate back into the weights. This can help nudge a network out of a local minimum. It's also possible to add noise directly to the input. This is sometimes done on image-recognition networks to generate multiple training images from one base image.

Cross-entropy error function

Most examples we've seen so far use the squared error for all output neurons to compute the error function, which drives learning for the entire network. We're also assuming that a neuron uses a sigmoid function; this will usually be true for all hidden neurons and often for output neurons. But again, remember that the sigmoid function becomes almost flat asymptotically. If a value should have been 0 and was actually close to 0, that's fine; there shouldn't be much adjustment. But if it should have been 0 and was actually close to 1 (i.e. very far off from the correct value), the sigmoid is still almost flat, and the small gradient means little learning will take place. But a neuron that's saturated on the wrong value is exactly when a larger adjustment should be made. The problem is that if we use the squared error for the loss, the partial derivative of the error with respect to the weights depends on the slope of the sigmoid. If the sigmoid is almost flat, the gradient is approximately 0, even if we're on the wrong end of the distribution. For sigmoid or softmax output, we can get better results by using the cross entropy:

$$E = -\frac{1}{n}\sum_{x}[y\ln a + (1-y)\ln(1-a)]$$

where *a* is the sigmoid output, *n* is the number of training items, *y* is the desired output, and the sum is over all training inputs *x*.

Note that if the output value *a* and expected value *y* are close, this will be very close to 0, as we'd expect. But farther away, it will learn faster, because the slope of the cross-entropy error is sensitive only to the magnitude of the error, not the gradient of the underlying sigmoid. Specifically, the partial derivative of the cross entropy with respect to the weights is

$$\frac{\partial E}{\partial w_j} = \frac{1}{n}\sum_{x} x_j(a-y)$$

where *a* is the sigmoid of the weighted sum of each input and the summation is over all inputs. Note that this is proportional to the size of the input and the size of the error, not the slope of the sigmoid. Thus, the farther off our actual value was, the higher the amount of learning. The partial derivative of the cross-entropy with respect to the biases is very similar, and again does not depend on the slope of the sigmoid, only the magnitude of the error.

Learning Rate

The slide deck on backgammon & neural networks mentioned using a learning rate λ of 0.5-0.7 or so. That's dependent on how weights and biases are updated and how the loss (error) function is computed. In general, a "reasonable" numeric value for the learning rate will depend in part on such implementation issues. The idea is that adjustments should be a small fraction of the amount needed to produce the desired output for the training input. Most modern implementations do the actual computations with matrix operations—a matrix of weights, a matrix of biases, a matrix of outputs and an error matrix—and then do one big computation that produces what adjustment should be made to produce the net gradient on all weights & biases. The parameters are then nudged a small amount, no more than a few percent, usually a fraction of a percent, in the direction to reduce the error. The point is to make a *small* adjustment, to nudge the network *slightly*, for each case, so that general principles are learned rather than individual cases memorized. Small steps also reduce the risk of overshooting the minimum and actually making things worse by taking too big a step. The optimum value for the learning rate may depend on the error function, activation function, network size, and other factors; so there's no hard-and-fast rule about what numeric value (0.001? 0.15? 0.5?) it should be. In general, a higher learning rate leads to faster, but less accurate, convergence. Lower learning rates lead to more accurate convergence, but more data may be needed to fully train the network. And there is at least as much art as science in tuning these hyperparameters.

ReLU for output layer
For larger networks, there will be many sigmoids feeding into each output neuron. If the neurons feeding in to a particular neuron are even halfway lit up, the magnitude of the weighted sum for an output neuron can be very large; thus the sigmoid output will be close to 1 or close to 0, and the gradient will be almost flat. Also, once the sum gets past a certain magnitude, large differences in the sum lead to small differences in the output function. Thus, many networks use a Rectified Linear Unit (ReLU) for the output; negative output is coded to 0, positive output left alone. [ReLU(x) = max(0, x).] This has a slope of 1 for output > 0 and thus the gradient will be well-behaved and give us something to work with. Output of 50 and 50,000 are clearly distinguishable, which would not be the case for the sigmoid of those sums. This is particularly important if the network is being used on a regression problem, where the limited output range of a sigmoid (0 to 1, or -1 to 1) may not be wide enough to express the result.

Softmax for output
This leads to a problem, though; or rather, shows the existence of another issue. We have been using the rule of selecting the output with the largest value and calling that the classification. This is called *hot-coding* the output; the maximum is set to 1, everything else is set to 0. The problem with this is that it can throw out useful information. Suppose we have a network with 5 ReLU outputs and we are using hot-coding. Two successive inputs give us the results:

[0.01, 0.0, 8.5, 0.001, 0.002]
[0.01, 0.0, 8.5, 8.4998, 0.002]

Hot-coding will reduce them both to [0, 0, 1, 0, 0], but the first is telling us it's a slam-dunk, the other that it's having a very difficult time distinguishing between two possible categories.

Thus the **softmax** function is used. In this case, we take the exponential of each output, and divide by the sum of the exponentials. [2] This normalizes the values to real numbers in 0-1 that add up to 1, and so can be interpreted as a probability distribution—specifically, the probability that the input falls into a specific category. For this example, the softmax of our first case (rounded to 4 places) is:

[0.0002, 0.0002, 0.9992, 0.0002, 0.0002]

and the second is

[0.0001, 0.0001, 0.4999, 0.4998, 0.0001]

which more accurately reflects the reality of what the network is telling us. For the first case, the network is 99.92% confident of its classification, that is, estimating the probability of that category at 99.92%; for the second, 49.99%.

Softmax can be used anytime we want to interpret the output as a probability distribution; for example, for a game network, we may want to estimate probabilities of White win, Black win, or draw. We could use a separate sigmoid for each as an estimate of the probability, but of course there's no reason to expect the sigmoids to sum to 1.0, meaning we'd have to normalize the results in any event.

For a classification problem, we're primarily interested in the magnitude of the correct output, which should usually be as close to 1 as possible after normalizing; we don't particularly care about the values of other neurons as long as they don't lead us to an incorrect conclusion. For these problems, with

[2]More info: https://en.wikipedia.org/wiki/Softmax_function

softmax output, we can use the log-likelihood cost function. For a given input *x* and desired output *y*, the log-likelihood cost of sigmoid (or normalized) output *a* is

$$C \equiv -\ln a_y$$

Some thought will confirm that when *a* is close to 1, this term will be small; when *a* is small, the cost will be higher. Like the cross-entropy, the partial derivative of the log-likelihood with respect to the weights depends only on the magnitude of the input and of the error. In practice, sigmoid output with a cross-entropy loss function and softmax output with log-likelihood loss function both often work well.

Survivors of Linear Algebra will recognize that softmax can be carried out as a vector operation, with the sum of the exponentials being the norm of the vector. There are definitions for more than one type of norm, of course, with varying computations. The consensus of the sources I've found is that it doesn't much matter which version you use. *But*, the exponential softmax is by far the most commonly used, and so should probably be your default choice unless you have a specific reason to prefer something else. (One advantage of the exponential softmax is that it isn't necessary to recode negative values as 0 before carrying out the operation. Rather than using ReLUs, we can just take the output values—weighted sums plus bias—for each output neuron & normalize.)

While we're on the subject of linear algebra: A *tensor* is, for our purposes, a matrix with an arbitrary number of dimensions. (From a strict mathematical perspective there's more to it than that, but that's "close enough for government work.") Tensor operations can be defined similarly to matrix operations and are defined as long as the dimensions are compatible. TensorFlow, for example, is mostly wrappers around a tensor-algebra package; it's written to take advantage of parallelization where possible, and thus can take advantage of multiple CPUs or GPUs. From this point on we will assume that weights, biases, etc are stored in matrices or tensors, and operations carried out via tensor calculus.

Batch training
Once you've written code for a tensor-based implementation, you might as well take advantage. The computations to adjust weights are fairly expensive; extending dimensions by one case doesn't add much extra work, though. Thus, one approach is to process training data in **mini-batches** of 100 items or so. This produces a tensor consisting of 100 training inputs, 100 sets of intermediate values, 100 expected outputs, and 100 actual outputs. The magic of tensor algebra can then be applied to produce one big adjustment matrix to push the network matrix to produce the gradient averaged across all cases in the mini-batch. The advantage here is that processing a mini-batch of 100 cases may take only 50 times as long as a single case; thus we can process more data in the same time. This is true whether we're doing one big tensor operation or a series of 2-D matrix operations. These adjustments are then multiplied by the learning rate before the adjustment is carried out. Training on randomly-chosen batches also reduces the risk of overtraining.

Training & overtraining
It's not unusual to see the classification rate rise to a relatively high level and then plateau for training data, but to see the error rate slowly increase on test data as training continues. This is an artifact of *overfitting* or *overtraining*; the network is directly encoding cases it's seen rather than learning general principles. Because the degrees of freedom (number of things that can be adjusted independently) is so high, the network develops its own internal representation of the training cases. The most likely (or at least most-often blamed) causes are too many neurons, too little data, or a badly-configured network. There are several ways to address this, separately or in combination. Most of these will produce small

boosts in the accuracy rate for training data and reduce the effects of overtraining on test data. Cumulatively, they can have a large effect.

- Once the accuracy seems to be at a plateau, the hidden layers of the network are scanned searching for the neuron with the lowest root-mean-square[3] (RMS) value of outgoing weights. This is the neuron having the least influence in the network. If its RMS value is well below the average value for the network, remove this neuron from the network permanently. (The exact threshold value can be determined from information theory.) This is then repeated every few thousand cases or so, as long as there are neurons not making contributions. This is mentioned in your text as **optimal brain damage**. It's an interesting idea and it makes sense—if something isn't making a significant contribution to anything, get rid of it—but I can't find much reference to it still being used.
    - In part this is because we're reducing the size of our network; larger networks have the potential to be more powerful.
    - It's also possible that our plateau isn't as good as the network will eventually do, just a pause before further improvement takes place.
    - This method is better suited to smaller networks (a few dozen to a few hundred neurons) than a large, deep network. A deep convolutional network can have thousands to millions of neurons, and plucking out a neuron here and a neuron there isn't going to have much effect. Other methods discussed here have similar effects, achieved via other means.
- Another option is to reduce the effect of overtraining by **reducing the learning rate** over time. Each few epochs (passes through the input data) or so, the learning rate is multiplied by a constant slightly smaller than 1; say, 0.9999. This reduction makes early learning of general principles effective but later fine-tuning less likely to pull the network away from a minimum.
- A third approach, surprisingly effective, is to randomly select a portion of hidden neurons, usually 25-50% of the total, and to not use them for a particular case (or batch). Instead of finding sigmoid output based on their weighted sums, their output is arbitrarily set to 0; and because their output is 0, they do not contribute to the next layer and will not be adjusted via gradient-descent methods. This **dropout** method means that on any given training case (or batch), only part of the network is being trained; this seems to help prevent learning specific cases.
    - Remember, overtraining occurs because we're presenting the same cases for training over & over; dropout ensures that each time we see a case, we're training a differently configured network, and thus not reinforcing the same neurons over and over.
    - Suppose we randomly drop out 50% of our neurons on each training case (or batch). When we run our test cases (cutting all weights in half, since we have twice as many neurons as we trained on), in effect we're assessing the test cases with the average of all of the different networks we trained. It is much less likely that the combined 'average' network will have memorized specific cases.
    - Another way of looking at it is to recall that often overtraining involves multiple neurons—two different neurons nudging things a little high in some cases, offset by another neuron nudging them a little low most of the time, etc. If we're using dropout, then one neuron can't depend on any other neuron being there consistently, and thus is less likely to build up a dependency across multiple neurons.

[3]Square all weights; find the mean (average) of the squares; take the square root of that mean.

- **Regularization**, also known as **weight decay**, is another technique that can reduce overfitting. These are methods of adjusting the learning rate by adding a feature to the cost function based on the average value of the weights.
    - One of the most common, **L2 regularization**, gives the network a preference for smaller weights unless the weight is making a substantial contribution to correct results—the network can have larger weights, but it'd better be worth it. The loss function is increased by a small constant (a hyperparameter) times the sum of squared weights in the network, divided by the size of the training set. This extra term modifies the partial derivative by adding a term to slightly reduce the weight's magnitude, besides the 'normal' term modifying the weight up or down depending on the gradient. This tends to reduce the weight unless increasing it will definitely reduce the loss (so we don't make it *too* small).
        - If the weights are small, then the effects of any 'noise' will also be small. If weights are large, then a small change in input can lead to a large change in output; this isn't what we want. Thus the network with small weights is less likely to learn the effects of any noise in the data—and mistaking noise for data is one cause of overfitting.
        - If applying this repeatedly pushes a node's weight all the way to 0, fine—it apparently wasn't making a contribution and there's no reason to increase it. In some implementations, if all of a node's weights have an absolute value less than some cutoff, usually somewhere around 10^{-6}, it's dropped completely as being not worth computing, saving time for other operations. This is very similar to the optimum brain damage heuristic noted above.
        - L2 regularization is based on the sum of the squared weights.
    - Another method, **L1 regularization**, reduces large-magnitude weights less than L2 regularization and smaller weights more. This concentrates the learning in a relatively small number of important weights and drives others to near 0; that is, it prefers sparse networks. L1 regularization is based on the sum of the absolute value of the weights.
- We can sometimes build up additional training cases by **modifying the training set**. For example, for image data, we can shift each image by a few pixels left, right, up, and down; and rotate the image 10 degrees or so in each direction. If the image doesn't contain text we need to process, it can be flipped left-to-right. We can alter the brightness or contrast. Each of these produces a slightly-different image that's still clearly recognizable and should be classified the same way as the unmodified data, but will appear as a distinct input to the network. Similarly, adding low-level background noise to audio samples may make a more robust training set. (Of course, filtering to remove background noise may prove equally effective.)
- **Batch normalization** is also quite effective. (see below)

Batch normalization

Real-world data is messy. Data may arrive in much different scales: some between 0 and 1, some between -10 and 10, some between 1000 and 100,000. Furthermore, some may be correlated; an economic model has to allow for the fact that the prime interest rate, average mortgage rate, average monthly rent, number of housing starts, inflation, and unemployment aren't independent of each other.

One approach is to rescale (normalize) and decorrelate the data; for example, convert each of the data items to z-scores (standard deviations above or below the mean) and carry out a multiple linear regression and recode each of the above data items as the residuals after controlling for all of the others. That's a lot of work, and assumes we correctly identified which inputs are correlated and which aren't...but wait! If we code everything as a vector, then decorrelating and rescaling can be done via a vector operation. Thus all we need to do is add an extra layer after our input layer, before our first 'real' hidden layer, to decorrelate and rescale the data. (We can get the numbers from our data set as a whole or from the cases in the current batch.) Great. Except, of course, that the output of the first layer may also be correlated, or have wildly different scales (say, half the output within 0.01 +/- 0.005 from 0, the rest all with an absolute value of 0.9 or greater; see earlier comments regarding delicately-balanced elephants). So we add *another* decorrelation and rescaling layer...but the output of the next layer may have the same problem...and before long we've doubled the number of layers in our network. Not a good solution. Even if we don't have that problem, we sometimes find that our weights or biases have a 'skew,' that is, an average value that's nonzero. Is that a quirk of the data? Or is it significant? How to tell? (If our weights have an average below zero, that implies most neurons are bringing values down from a too-high value; should we be preventing the value from being too high in the first place?)

So we normalize each set of weighted sums coming into the neurons. We find the mean$\bar{x}$and standard deviation σ of the weighted sums x for each node, and then find the batch-normalized value

$$\hat{x} = \frac{x - \bar{x}}{\sigma(x)}$$

$$BN(x) = \alpha\hat{x} + \beta$$

where α and β are learnable parameters for each neuron. That's right, we're now doing gradient descent on α and β. Note that if α equals the standard deviation and β equals the mean, we can restore the original *x* values; thus, we do not lose any expressiveness. If we really needed the original *x* value, we can learn that. Also, since we have a vector of β's for each layer, we have our biases, and do not need to store a separate bias for each neuron in addition to this one; we can put our bias vector to work by applying the biases after batch normalization rather than before. We only need calculate the weighted sums for each neuron, and then put the normalization step in before calculating the sigmoid output. It's important to note that this is in addition to, not instead of, the regular training on the weights that's happening via gradient descent. This adds one additional parameter per neuron, which can change during training. *Batch normalization is not done on the output layer*, since we're going to normalize output using softmax anyway; it's only for internal layers producing sigmoid output.

Ensemble methods

Another approach to avoid the risk of overfitting is to recognize that different networks are unlikely to overfit in exactly the same way. One approach is to use ensemble methods, in which a group of networks are all trained at once, and classification is based on consensus among the networks. This is a full-scale version of what dropout training tries to accomplish cheaply. For example, the system used by the U. S. Postal Service to recognize hand-written digits in mail addresses uses 3 networks. The first network trained on the full data set; the second trained only on cases which the first network had trouble classifying; the third trained on cases where the first two disagreed. A 2-out-of-3 protocol is used to conduct the final classification; if the three networks are all different, the item is kicked out as unclassifiable, and turned over to humans for processing. Other ensemble systems train all networks on the same data, using a 2-of-3 or 3-of-5 protocol.

Unstable Gradient Problem

We've been looking at the errors and partial derivatives as functions of the error, without worrying too much about the nuts and bolts of how the error propagates backwards through a network. As the network becomes deeper, a problem starts appearing. The amount of error that propagates backwards is a function of the outputs of the current layer (moving forward) and the weights connecting this layer to the next. Error propagates back from the output layer to the layer forward of the one we're looking at, and then the weights determine how that error is distributed. Of course, the error that propagated to that layer depended on what was ahead of it, and so on. So the amount of error that reaches back to an earlier layer is based on the product of the weights between that layer and the output, and more and more weights are involved the farther back we go.

If those weights are small, relatively close to 0, then the product of several of them will be even closer to 0, and very little error will propagate all the way back to the beginning of the network. If those weights are large, then small differences in the error can lead to large changes in the product, and the gradient is *unstable*. In either case, learning isn't proceeding as we'd like. Either the gradient is almost exactly 0 and no learning is taking place in early layers, or the earliest layers are having their values changed dramatically on each input, and learning doesn't settle down into any particular direction. If the gradient is vanishing (the more common problem), we can increase the learning rate η as we move back; for instance, multiply by 1.3 or so every time we move to a previous layer. This becomes another hyperparameter, and the optimum value of the hyperparameter itself may not be stable.

This is a general problem, and as of this writing there doesn't seem to be a good solution to it. There have been some attempts to use evolutionary methods for training. For example, a suite of 100 networks is initialized randomly, and the same training batch is presented to each. The network that did most poorly, and the two that did best, are identified. (If there are ties, choose with uniform probability from the pool of tied networks.) The worst-performing is eliminated, and the best 2 are fed into a crossover/mutation function that returns a new network, which replaces the network that was removed. This neatly sidesteps the problem of unstable gradients and has been found to work fairly well, but is obviously computationally expensive.

Neural networks for regression

Sometimes rather than classification, we want to carry out *regression*, to estimate f(x) where x is the vector of all the inputs and we don't know the true nature of f(), or even which components of the inputs are relevant in which cases.

In classical multivariate regression (covered in any statistics course), we find which variables have the most influence on f() and include them in the model. We can include squares, crossproducts, etc., by adding them to the data set and treating them as other variables, doing linear regression on them. Of course, since the variables in our model may be correlated among themselves, we have to adjust coefficients (and sometimes what's in our model) based on what variables are added. But eventually we end up with a single model, a set of variables and coefficients for each. Variables that are not included in the model have no effect on our estimate, and the same model is used for all cases. Of course, if we don't have enough data, our model can end up fitting statistical noise.

Suppose instead we use a neural network. Here's how: We can approximate any function as a sum of some other functions, of course. We choose to make a series of Gaussian (normal) curves, of varying magnitudes and widths, located at various points along the number line. A given set of inputs produces a specific set of parameters for our Gaussians, and we can find the sum of the Gaussians to find the estimate for our function. How do we know where to place the Gaussians, and how many of them we need? Gradient descent, of course! We determine the optimum number of Gaussians we need by trial and error. We start with a single Gaussian located at a mean μ with standard deviation σ and height h. We then do gradient descent on μ, σ, and h to improve the fit. Once we're reasonably sure we have a model with k Gaussians that's as good as it's going to get, we go to k+1 Gaussians evenly distributed over the range of the function and try again, continuing until adding another Gaussian gives us no significant improvement.

The advantage of this method is that it keeps all input variables for every case, though the network may be able to learn conditions under which a particular input should have lower weight than usual. In a ‘standard’ multivariate regression, a variable is either in the model or not, and the same model is used for all cases.

All of the various methods of improving performance—regularization, dropout, etc—work for a regression network just as they do for a classification network.

A general discussion of the state of the art
Several times I’ve started an explanation with “the basic idea is...” or “this seems to work because… .” This isn’t just academic circumspection. In some cases, intuition is really the only explanation we have. The mathematical tools are still under development. The discussion of why dropout works, for example, is based more on disciplined speculation than rigorous analysis. We don’t have a good theoretical model to explain why a network of ReLU’s learns faster than a network of sigmoids, but we know empirically that it does. Methods have been tried and found to work, but there’s still an element of seat-of-the-pants flying to some of it. As one researcher put it, “*You have to realize that our theoretical tools are very weak. Sometimes, we have good mathematical intuitions for why a particular technique should work. Sometimes our intuition ends up being wrong.”* Thus, the information presented here may become woefully out of date fairly soon, as new tools are developed and analysis proceeds further. There’s still as much art as science to some of this.

A Selected Videography

How Deep Neural Networks Work
https://www.youtube.com/watch?v=ILsA4nyG7I0&t=23s

Playlist:

- But What *Is* a Neural Network?
- Gradient Descent: How Neural Networks Learn
- What is Backpropagation Really Doing?
- Appendix: Backpropagation Calculus

https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

How Convolutional Neural Networks Work
https://www.youtube.com/watch?v=FmpDIaiMIeA&t=16s

Deep Neural Networks are Easily Fooled
https://www.youtube.com/watch?v=M2IebCN9Ht4&t=30s

Breaking Deep Learning Systems with Adversarial Examples
https://www.youtube.com/watch?v=j9FLOinaG94

Tensorflow and deep learning without a PhD
https://www.youtube.com/watch?v=vq2nnJ4g6N0
Very good, very detailed. Worth it if you have 2 ½ hours.

A friendly introduction to deep learning
https://www.youtube.com/watch?v=BR9h47Jtqyw
Good, but slow-paced. The preliminaries (prior to about 12:30) can be safely skipped if you've been paying attention in class.

A friendly introduction to Convolutional Neural Networks and Image Recognition
https://www.youtube.com/watch?v=2-Ol7ZB0MmU
Good, but slow-paced. May be helpful if you're getting stuck on details.

Developing Neural Networks in Visual Studio
https://www.youtube.com/watch?v=9aHJ-FAzQaE&t=2457s
The discussion of activation functions and error functions starting at about 24:00 is particularly good. The C# code starts around 36:00.

How Google DeepMind Conquered the Game of Go
https://www.youtube.com/watch?v=derC33ODrME
Very good discussion, not particularly technical. Not quite an hour in length.

Appendix:
An extremely brief and extremely conceptual explanation of partial derivatives

Recall from Calculus I that a continuous function *f(x)* has a *derivative* denoted *f'* or *dy/dx* which gives the slope of that function at any point where it is defined. So, for example, for the sigmoid function which is defined for all *x* as

$$f(x) = \frac{1}{1+e^{-x}}$$

we can, after some algebra, determine that

$$f'(x) = f(x)(1 - f(x))$$

(In this quick review, we're not going to go through the details of algebraic manipulation; review your Calc I text if necessary.)

Now suppose we want to generalize this to multiple dimensions; that is, instead of *f(x)*, suppose we have *f(x,y).* Can we still differentiate with respect to *x*? Of course we can; the result is the *partial derivative*, denoted $\partial\, f / \partial\, x$, that shows how *f* changes with respect to changes in *x*, independent of any other variable. To compute this, we assume that *y* is a constant (since changing *x* has no effect on y).

To take an example: Suppose we have a simple node with 2 inputs I_1 and I_2, connected via weights W_1 and W_2, with bias B. The net value passed to the sigmoid function is

$$net = I_1W_1 + I_2W_2 + B$$

The effect on *net* of a change in the weights is given by

$$\frac{\partial net}{\partial W_1} = I_1$$

$$\frac{\partial net}{\partial W_2} = I_2$$

In each case, we treat the variable we're differentiating by as usual, and all other variables are treated as constant (changing W_1 has no effect on W_2 or the bias, or how either affects the total sum).

So the partial derivative is just the one-dimensional derivative for a single variable. The *gradient* is the multidimensional derivative, consisting of all of the partials. If we have *z* = *f(x,y)*, the gradient of z points toward the direction *z* is changing at any (*x*,*y*) point. The gradient is sometimes denoted by ∇z.