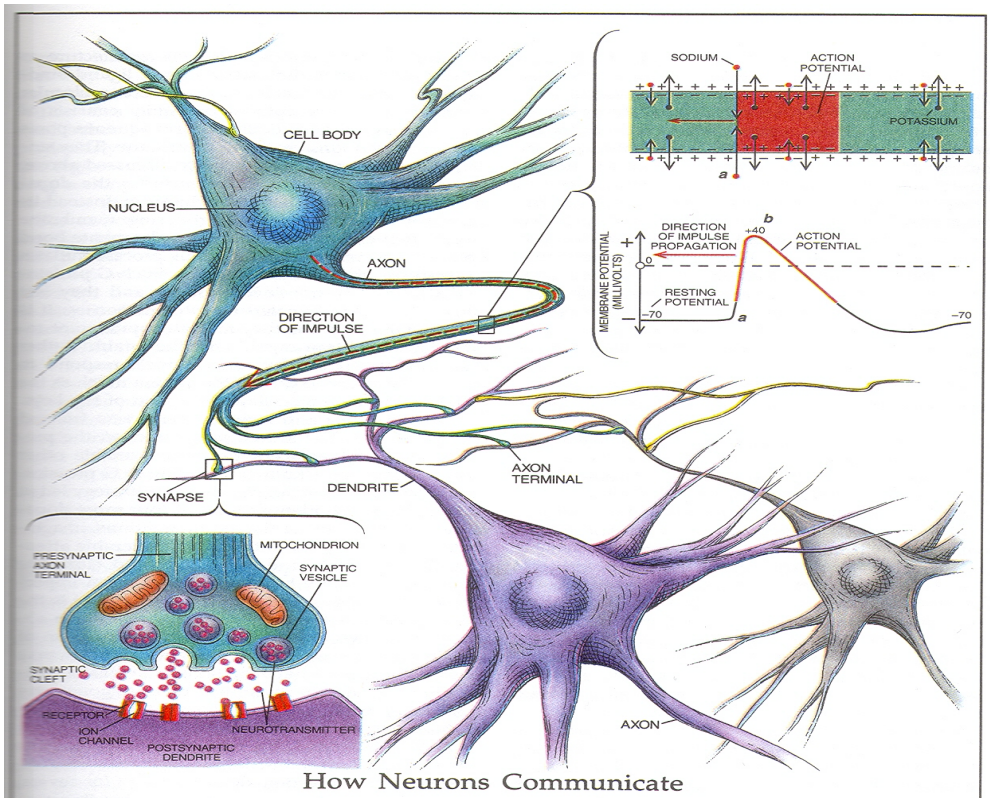
18 Neurobiology; Faster Neural Network Training

NEUROBIOLOGY (cont'd)



neurons.pdf

- Neuron: A cell in brain/nervous system for thinking/communication
- Action potential or spike: An electrochemical impulse fired by a neuron to communicate w/other neurons
- Axon: The limb(s) along which the action potential propagates; "output"
  [Most axons branch out eventually, sometimes profusely near their ends.]
  [It turns out that squids have a very large axon they use for fast propulsion by expelling jets of water. The mathematics of action potentials was first characterized in these squid axons, and that work won a Nobel Prize in Physiology in 1963.]
- Dendrite: Smaller limb by which neuron receives info; "input"
- Synapse: Connection from one neuron's axon to another's dendrite
  [Some synapses connect axons to muscles or glands.]
- Neurotransmitter: Chemical released by axon terminal to stimulate dendrite

[When an action potential reaches an axon terminal, it causes tiny containers of neurotransmitter, called vesicles, to empty their contents into the space where the axon terminal meets another neuron's dendrite. That space is called the synaptic cleft. The neurotransmitters bind to receptors on the dendrite and influence the next neuron's body voltage. This sounds incredibly slow, but it all happens in 1 to 5 milliseconds.]

You have about 10^{11} neurons, each with about 10^4 synapses.

Analogies: [between artificial neural networks and brains]

- Output of unit ↔ firing rate of neuron
  [An action potential is "all or nothing"—all action potentials have the same shape and size. The output of a neuron is not signified by voltage like the output of a transistor. The output of a neuron is the frequency at which it fires. Some neurons can fire at nearly 1,000 times a second, which you might think of as a strong "1" output. Conversely, some types of neurons can go for minutes without firing. But some types of neurons never stop firing, and for those you might interpret a firing rate of 10 times per second as a "0".]
- Weight of connection ↔ synapse strength
- Positive weight ↔ excitatory neurotransmitter (e.g., glutamine)
- Negative weight ↔ inhibitory neurotransmitter (e.g., GABA, glycine) [Gamma aminobutyric acid.]
  [A typical neuron is either excitatory at all its axon terminals, or inhibitory at all its terminals. It can't switch from one to the other. Artificial neural nets have an advantage here.]
- Linear combo of inputs ↔ summation
  [A neuron fires when the sum of its inputs, integrated over time, reaches a high enough voltage. However, the neuron body voltage also decays slowly with time, so if the action potentials are coming in slowly enough, the neuron might not fire at all.]
- Logistic/sigmoid fn ↔ firing rate saturation
  [A neuron can't fire more than 1,000 times a second, nor less than zero times a second. This limits its ability to overpower downstream neurons. We accomplish the same thing with the sigmoid function.]
- Weight change/learning ↔ synaptic plasticity
  [Donald] Hebb's rule (1949): "Cells that fire together, wire together."
  [This doesn't mean that the cells have to fire at exactly the same time. But if one cell's firing tends to make another cell fire more often, their excitatory synaptic connection tends to grow stronger. There's a reverse rule for inhibitory connections. And there are ways for neurons that aren't even connected to grow connections.]
  [There are simple computer learning algorithms based on Hebb's rule. They can work, but they're generally not nearly as fast or effective as backpropagation.]

[Backpropagation is one part of artificial neural networks for which any analogy is doubtful. There have been some proposals that the brain might do something vaguely like backpropagation,[6] but it seems tenuous. Learning in brains is still not well understood.]

[As computer scientists, our primary motivation for studying neurology is to try to get clues about how we can get computers to do tasks that humans are good at. But neurologists and psychologists have also been part of the study of neural nets from the very beginning. Their motivations are scientific: they're curious how humans think, and how we can do the things we do.]

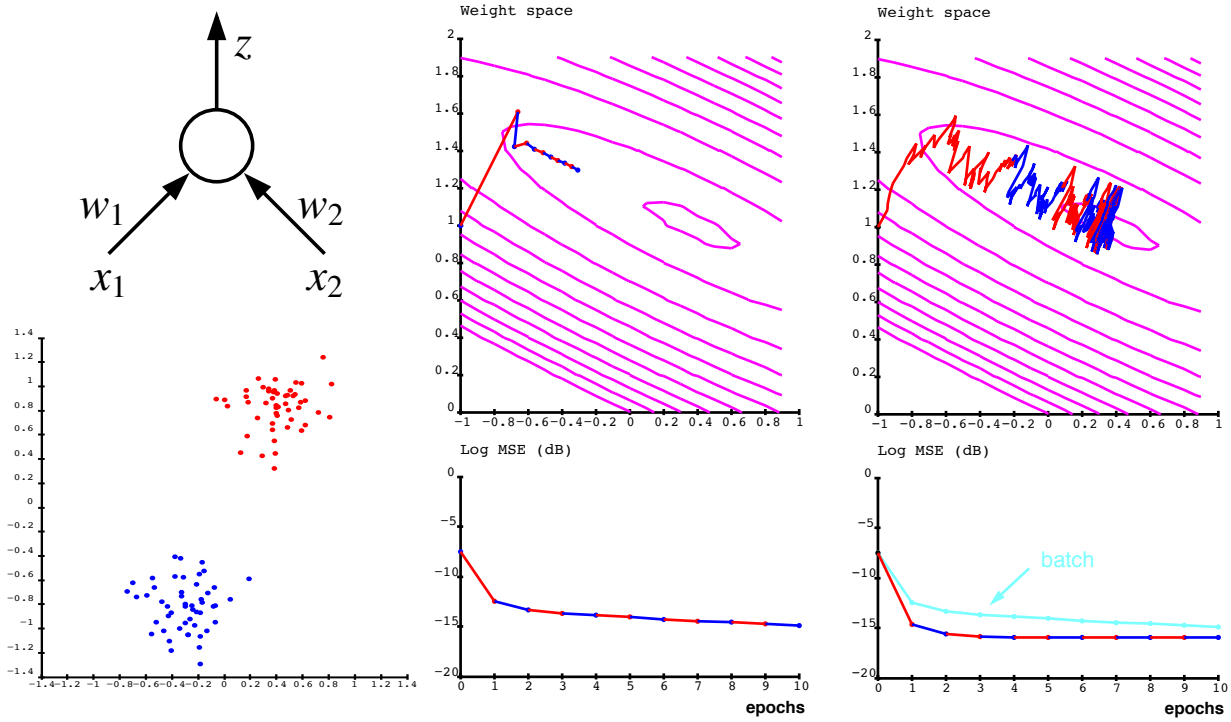
[6] See Lillicrap et al., "Backpropagation and the Brain," *Nature Reviews Neuroscience* **21**, pages 335-–346, April 2020.

HEURISTICS FOR FASTER TRAINING

[A big disadvantage of neural nets is that they take a long, long time to train compared to other classification methods we've studied. Here are some ways to speed them up. Unfortunately, you usually have to experiment with techniques and hyperparameters to find which ones will help with your particular application. I suggest you implement vanilla backpropagation first, usually in combination with stochastic gradient descent and intelligent weight initialization, and experiment with fancy heuristics only after you get that working.]

(1) ReLUs. [To fix the vanishing gradient problem, as described in the previous lecture.]

(2) Stochastic gradient descent (SGD): faster than batch on large, redundant data sets.
[Whereas batch gradient descent walks downhill on one cost function, stochastic descent takes a very short step downhill on one point's loss function and then another short step on another point's loss function. The cost function is the sum of the loss functions over all the sample points, so one batch step is akin to n stochastic steps and does roughly the same amount of computation. But if you have many different examples of the digit "9", they contain much redundant information, and stochastic gradient descent learns the redundant information more quickly—often *much* more quickly. Conversely, if the data set is so small that it encodes little redundant information, batch gradient descent is typically faster.]



batchvsstochmod.pdf (LeCun et al., "Efficient BackProp") [Left: a perceptron with only two weights trained to minimize the mean squared error cost function, and its 2D training data. Center: batch gradient descent makes only a little progress each epoch. Epochs alternate between red and blue. Right: stochastic descent decreases the error much faster than batch descent. Again, epochs alternate between red and blue.]

One epoch presents every training point once. Training usually takes many epochs, but if sample is huge [and carries lots of redundant information], SGD can take less than one epoch.

(3) SGD with minibatches.
Choose a minibatch size b; e.g. 256.
Repeatedly perform gradient descent on the sum of the loss functions of b randomly chosen points.

[Although we perform gradient descent on a minibatch of training points all at once, we don't call it *batch* gradient descent. We still call it *stochastic* gradient descent.]

Advantages [compared to SGD done just one point at a time]:

- Less "bouncy"; usually converges more quickly.
  [SGD bounces around wildly. Minibatches reduce the variance of the steps by a factor of $\sqrt{b}$ while maintaining the advantages of SGD.]
- Can use parallelism, vectorization, GPUs efficiently.
  [The backpropagation computations are fully independent from one training point to another, so it's very easy to compute gradients for multiple points in parallel or through vectorization.]
- Better speed because of memory hierarchy.
  [You should lay out the activations for the training points in the minibatch next to each other in memory. With the right memory layout and minibatch size, your use of the caches and memory hierarchy can be very efficient. Performing SGD on 64 training points might be almost as fast as performing SGD on one. The bottleneck in neural network training is memory latency, not arithmetic.]

[Minibatches nearly always work faster than processing just one training point at a time. They are standard in implementations of neural network training.]

Typically, we shuffle training pts, partition into $\lceil n/b \rceil$ minibatches.
An epoch presents each minibatch once. [Reshuffling for each epoch is optional.]
[It is important to randomize well so your minibatch is a representative subsample of the training points. Sometimes practitioners store each class in a separate list, shuffle each class separately, and build minibatches with a proportional number of training points from each class.]

[Be forewarned that the best learning rate ϵ will be different for different values of the minibatch size b, and there isn't always a predictable relationship between b and the best ϵ.]

(4) To choose learning rate ϵ, use a small random subsample of training data.
[Practitioners have found that the size of the training set has only a weak effect on the best choice of ϵ. So use a subsample to quickly estimate a good learning rate, then apply it to your whole training set. This is very easy to do, and it can save you a lot of time!]

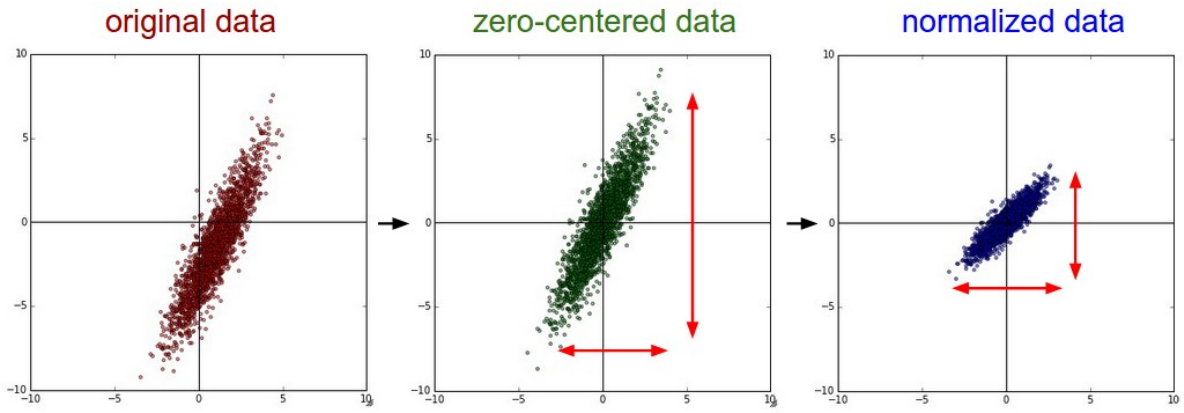
(5) Emphasizing schemes. [Neural networks learn the most redundant examples quickly, and the most rare examples slowly. This motivates emphasizing schemes, which repeat the rare examples more often.]

- Stochastic: present examples from rare classes more often, or w/bigger ϵ.
- Batch: examples from rare classes have bigger losses.

[Emphasizing schemes are a natural way to incorporate an asymmetric loss function. Suppose you're developing a test for a rare disease, and you have a small number of examples of people with the disease, but many examples of people without it. The classifier might decide to always return negative, "no disease," because it can achieve 99% accuracy that way. So we'd like to impose larger losses on examples with positive labels. For batch gradient descent, we make their losses bigger in the cost function. For SGD there are two options: use a bigger loss for rare examples, equivalent to presenting them with a larger learning rate ϵ; or present them more often. It's not clear which of these two options will train faster, as those extra presentations take more computation but shorter steps have better convergence properties.]

[Emphasizing schemes can also be used to emphasize misclassified training points, like the Perceptron Algorithm does, but that can backfire if those points are bad outliers.]

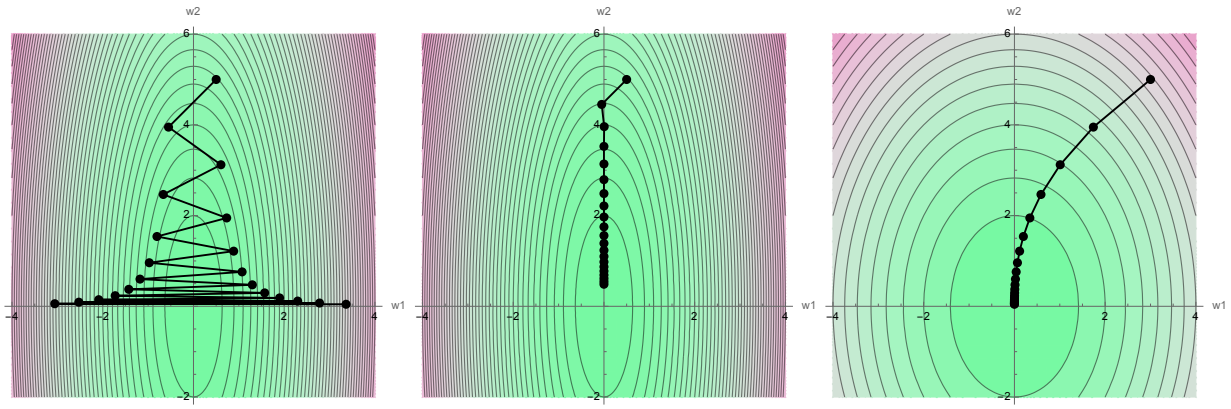
(6) Normalizing the training pts.
   – Center each feature so mean is zero.
   – Then scale each feature so variance ≈ 1.



normalize.jpg [A 2D example of normalizing points.]

[Remember that the power of neural networks comes from the nonlinearity of the activation function, and the nonlinearity of a sigmoid or ReLU unit falls where the linear combination of values coming in is close to zero. Centering makes it easier for the first layer of hidden units to be in the nonlinear operating region.]

[Neural networks are an example of an optimization algorithm whose cost function tends to have better-conditioned Hessians if the input features are normalized, so it may converge to a local minimum faster.]



illcondition105.pdf, illcondition055.pdf, goodcondition.pdf

[Recall these illustrations from Lecture 5. Gradient descent on a function with an ill-conditioned Hessian matrix can be slow because a large step size diverges in one direction (left) while a small step size converges slowly in another direction (center). Normalizing the data might improve the conditioning of the Hessian (right), thus speeding up gradient descent. Moreover, if you use ℓ_2-regularization, normalization makes it penalize the features more equally.]

[You could go even further and whiten the data, as we discussed in Lecture 9, but whitening takes $\Theta(nd^2+d^3)$ time for n training points with d features, so it takes too much time if d is very large; whereas normalizing takes $\Theta(nd)$ time.]

[Remember that whatever linear transformation you apply to the training points, you *must* later apply the same linear transformation to the test points you want to classify!]

(7) Initializing weights. [Proper initialization of weights is very important, especially for deep networks. Consider this carefully for Homework 6!]

[Recall that we initialize a neural network with random weight values to break the symmetry between hidden units. If we make those random values too small, they might never grow enough, especially if the network is deep. If we make them too large, we may cause the exploding gradient problem in ReLUs, or the vanishing gradient problem in sigmoid units. Here are some rules of thumb for initializing random weights.]

Consider the variance of each unit's output, given random weights.
Principle: output of unit should have same variance as each of its inputs.[7]

For a unit with fan-in η (not counting bias term), initialize each incoming edge to ...
[The fan-in of a unit is the number of connections entering the unit.]
For a ReLU unit, a weight in $\mathcal{N}(0, 2/\eta)$ or $\mathcal{U}\left(-\sqrt{6/\eta},\ \sqrt{6/\eta}\right)$.
[This is called He initialization, after Kaiming He.]
For a sigmoid unit, make it $\mathcal{N}(0, 12.8/\eta)$ or $\mathcal{U}\left(-\sqrt{38.4/\eta},\ \sqrt{38.4/\eta}\right)$.
For a linear or tanh unit, make it $\mathcal{N}(0, 1/\eta)$ or $\mathcal{U}\left(-\sqrt{3/\eta},\ \sqrt{3/\eta}\right)$.
[This initialization is sometimes called Xavier initialization, but it isn't quite what Xavier Glorot originally proposed. A tanh unit is very similar to a sigmoid unit, but its output is centered at zero, whereas sigmoid outputs are centered at 0.5. I don't recommend you use either sigmoid or tanh units as hidden units, but if you do, the tanh is preferable for that reason.]

[The reason we divide by the fan-in is because the more inputs a unit has, the greater its incoming signal is. So we must make the weights smaller to match the unit's output variance to the variance of each input.]

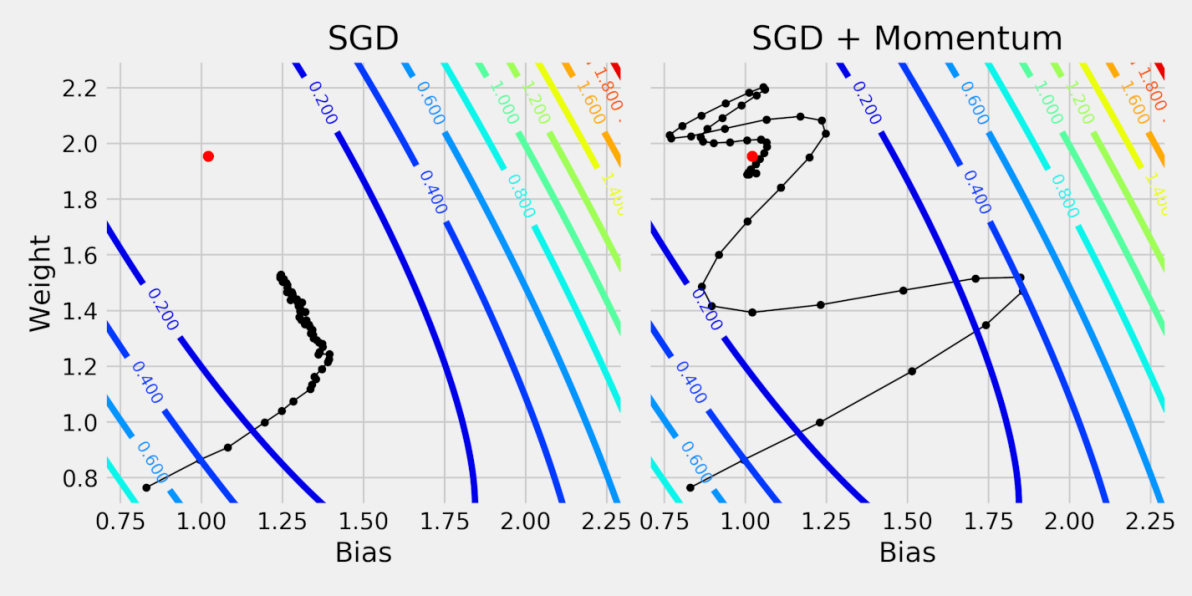
Set bias terms to zero. [Bias terms can too easily overpower signals coming from earlier layers. For ReLU units, some people suggest setting the bias terms to a small positive constant so they're more likely to be turned on at first, but other people say it gives worse performance in practice.]
Linear output unit: set bias term to the mean label.
Sigmoid output unit: set bias term so default output is the mean label.
[E.g., if 90% of your training points are in class C, set the bias so the sigmoid output defaults to 0.9. Andrej Karpathy says that if you don't initialize the output unit biases, the first few minibatches are largely wasted learning the mean labels.]

(8) Momentum. Gradient descent changes "momentum" m slowly. [The intuition is that if you've taken many steps in roughly the same direction, you should go faster in that direction.]

$m \leftarrow -\epsilon' \nabla J(w)$
repeat
    $w \leftarrow w + m$
    $m \leftarrow \beta m - \epsilon \nabla J(w)$

Good for both batch & stochastic. Choose hyperparameter $\beta < 1$.
[Here, J is the cost for the minibatch, which could be anything from a single training point to the whole training set. The hyperparameter β specifies how much momentum persists from iteration to iteration.]
[I've seen conflicting advice on β. Some researchers set it to 0.9; some set it close to zero. Geoff Hinton suggests starting at 0.5 and slowly increasing it to 0.9 or higher as the gradients get small.]
[If β is large, you should usually choose ϵ small to compensate, but you might still use a large ϵ' in the first line so the initial velocity is reasonable.]

[7] For an explanation of these suggestions, see Siddharth Krishna Kumar, "On Weight Initialization in Deep Neural Networks."



sgdmomentumgodoy.png (Daniel Godoy) [Left: 50 steps of SGD (with 16-point mini-batches) don't get very close to the minimum (red). Right: 50 steps with momentum do get close to the minimum, but overshoot it several times.]

[A problem with momentum is that once it gets close to a good minimum, it overshoots the minimum, then oscillates around it. But it often gets us close to a good minimum sooner. We see both phenomena above.]

pretzelwaterpark.jpg [How I imagine a neural network's cost function. It does not resemble a parabolic bowl. The downhill paths from the start to the local minima are sinuous. The swimmers here employ gradient descent with momentum with great success.]