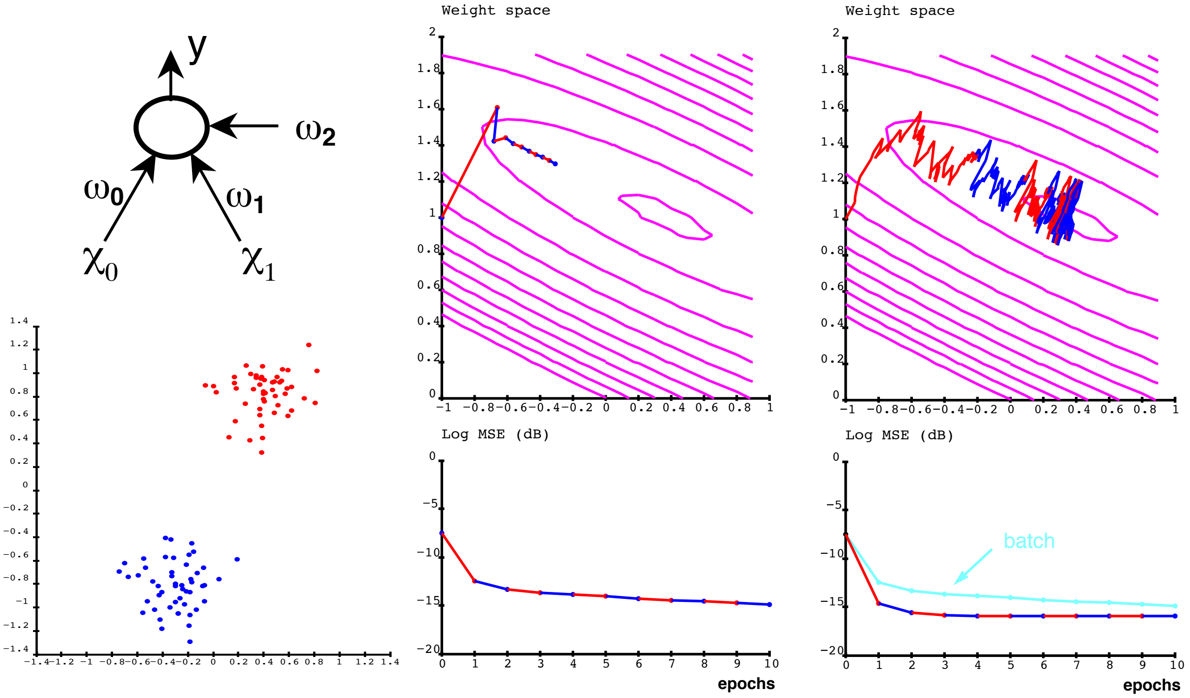
19 Better Neural Network Training; Convolutional Neural Networks

[I'm going to talk about a bunch of heuristics that make gradient descent faster or prevent it from overfitting. I suggest you implement vanilla stochastic backpropagation first, and experiment with these heuristics only after you get that working.]

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.]

- Fix vanishing gradient problem. [As described in the previous lecture.]
- Stochastic gradient descent: 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 behaves similarly to n stochastic steps and takes roughly the same amount of time. 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. Conversely, if the data set is so small that it encodes little redundant information, batch gradient descent is typically faster.]

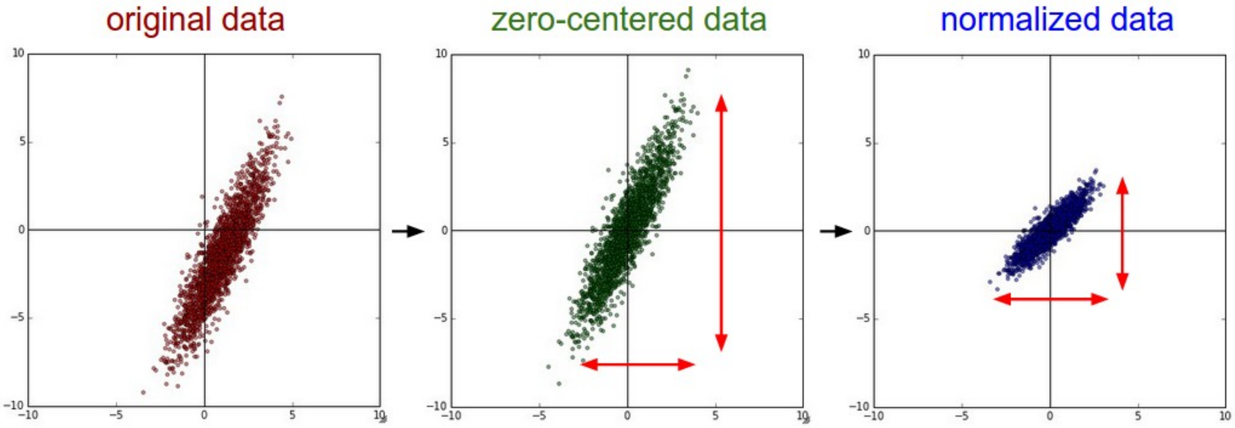


batchvsstoch.pdf (LeCun et al., "Efficient BackProp") [Left: a perceptron with only three weights, and its 2D training data. Center: batch gradient descent makes only a little progress each epoch. (Epochs alternate between red and blue. Note that the bias term ω_2 is not plotted.) Right: stochastic descent decreases the error much faster than batch descent.]

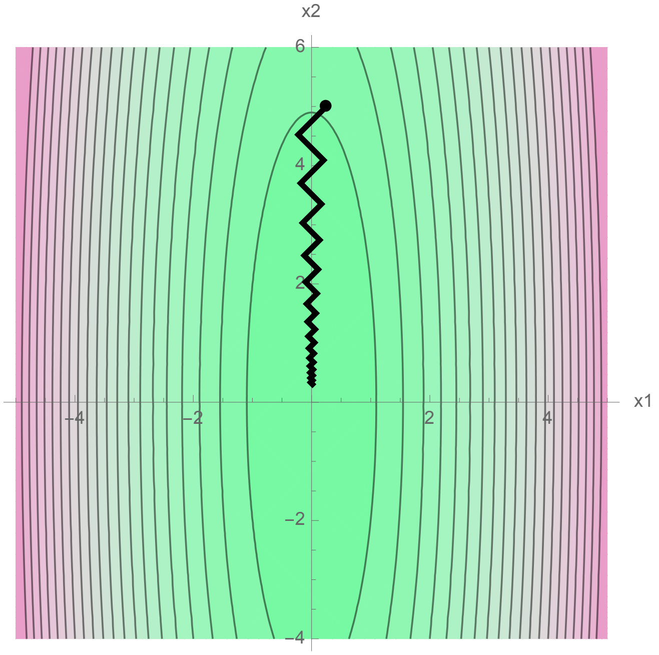
One epoch presents every training point once. Training usually takes many epochs, but if sample is huge [and carries lots of redundant information] it can take less than one.

- Normalizing the data.
  - Center each feature so mean is zero.
  - Then scale each feature so variance ≈ 1.

    [The centering step seems to make it easier for hidden units to get into a good operating region of the sigmoid or ReLU. The scaling step makes the objective function better conditioned, so gradient descent converges faster. The scaling step also makes ℓ_2-regularization treat the features more equally.]



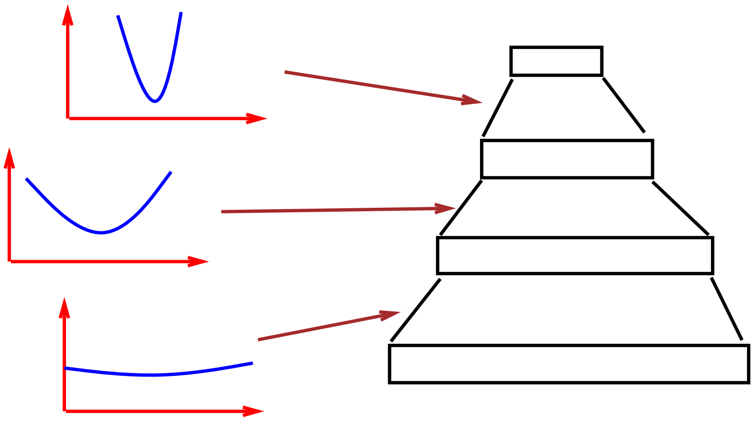
normalize.jpg [A 2D example of data normalization.]



illcondition.pdf [Skewed data often leads to an objective function with an ill-conditioned (highly eccentric) Hessian. Gradient descent in these functions can be painfully slow, as this figure shows. Normalizing helps by reducing the eccentricity. Whitening reduces the eccentricity even more, but it's more expensive. Another thing that helps with elongated troughs like this is momentum, which we'll discuss shortly. It eventually gets us moving fast along the long axis.]

- "Centering" the hidden units helps too.
  Replace sigmoids with $\tanh \gamma = \dfrac{e^{\gamma} - e^{-\gamma}}{e^{\gamma} + e^{-\gamma}} = 2\, s(2\gamma) - 1.$
  [This function ranges from -1 to 1 instead of from 0 to 1.]
  [If you use tanh units, don't forget that you also need to change backprop to replace s' with the derivative of tanh, $(1 - \tanh^2 \gamma)$. Also, good output target values change to roughly 0.8 and -0.8.]

- Different learning rate for each layer of weights.
  Earlier layers have smaller gradients, need larger learning rate.



curvaturelayers.pdf [In this illustration, the inputs are at the bottom, and the outputs at the top. The derivatives tend to be smaller at the earlier layers.]

- Emphasizing schemes.
  [Neural networks learn the most redundant examples quickly, and the most rare examples slowly. So we try to emphasize the uncommon examples.]
  - Present examples from rare classes more often, or w/bigger ϵ.
  - Same for misclassified examples.

  [Be forewarned that emphasizing schemes can backfire if you have really bad outliers.]

- Second-order optimization.
  [Unfortunately, Newton's method is completely impractical, because the Hessian is too large and expensive to compute. There have been a lot of attempts to incorporate curvature information into neural net learning in cheaper ways, but none of them are popular yet.]
  - Nonlinear conjugate gradient: works well for small nets + small data + regression.
    Batch descent only! → Too slow with redundant data.
  - Stochastic Levenberg Marquardt: approximates a diagonal Hessian.
    [The authors claim convergence is typically three times faster than well-tuned stochastic gradient descent. The algorithm is complicated.]

- Acceleration schemes: RMSprop, Adam, AMSGrad.
  [These are quite popular. Look them up online if you're curious.]

- Momentum. Gradient descent changes "velocity" Δw slowly.

  ```
  $\Delta w \leftarrow -\epsilon \nabla J(w)$
  repeat
      $w \leftarrow w + \Delta w$
      $\Delta w \leftarrow -\epsilon \nabla J(w) + \beta \Delta w$
  ```

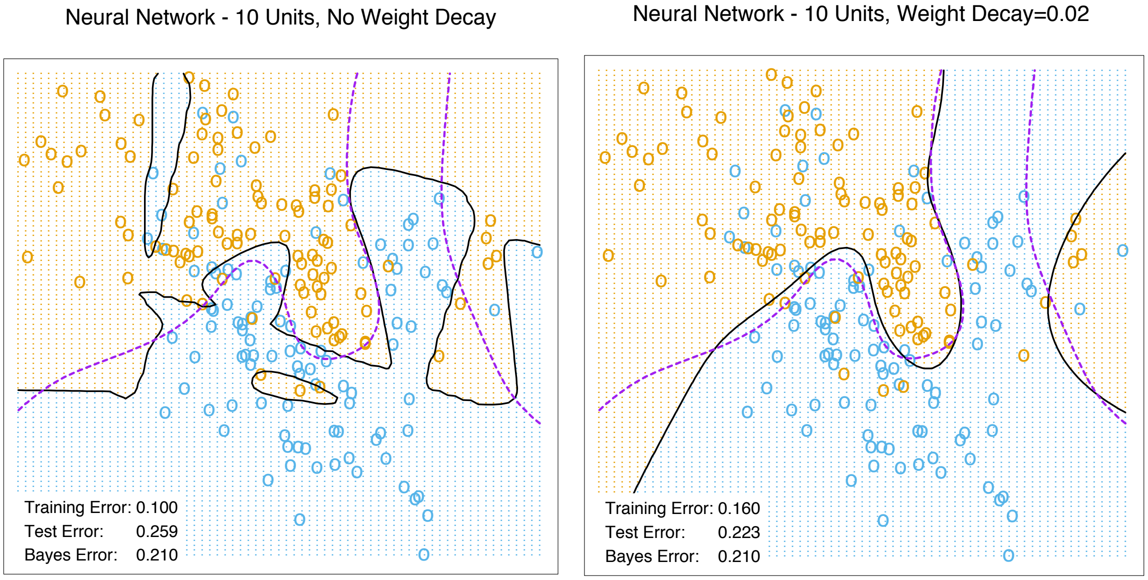
  Good for both batch & stochastic. Choose hyperparameter $\beta < 1$.
  [The hyperparameter β specifies how much momentum persists from iteration to iteration.]
  [I've seen conflicting advice on β. Some researchers set it to 0.9; some set it close to zero. Geoff Hinton suggests starting at 0.5 and slowly 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.]
  [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, even if the last bit of accuracy comes more slowly.]

Heuristics to Avoid Overfitting

- Number of hidden units.
  [Unfortunately, the relationship between overfitting and the number of units per hidden layer is complicated. If there's too few, you can't learn well, of course. But empirically we often see a phenomenon called "double descent." As the number of units per hidden layer increases, we see the usual U-shaped curve of test error as the bias reduces and the variance increases. But when there are enough weights that the training error reaches zero, we sometimes see a second descent where the test error starts to decrease again! This effect seems to be related to properties of gradient descent, and it might not appear if you use acceleration methods like Adam or second-order methods.]

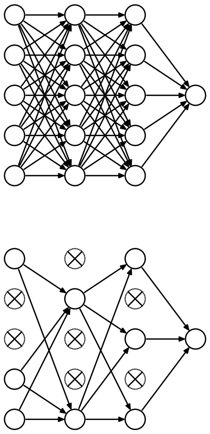
- Ensemble of neural nets. Bagging + random initial weights.
  [We saw how well ensemble learning works for decision trees. It works well for neural nets too. The combination of bagging and random initial weights helps ensure that each neural net comes out differently. Obviously, ensembles of neural nets are slow.]

- ℓ_2 regularization, aka <u>weight decay</u>.
  Add $\lambda \|w\|^2$ to the cost/loss fn, where w is vector of all weights.
  [w includes all the weights in matrices V and W, rewritten as a vector.]
  [We do this for the same reason we do it in ridge regression: penalizing large weights reduces overfitting by reducing the variance of the method.]
  [With a neural network, it's not clear whether penalizing the bias terms is bad or good. If you penalize the bias terms, regularization has the effect of drawing each ReLU or sigmoid unit closer to the center of its operating region. I would suggest to try both ways and use validation to decide whether you should penalize the bias terms or not.]
  Effect: $\Delta w_i = -\epsilon \dfrac{\partial J}{\partial w_i}$ has extra term $-2\epsilon\lambda\, w_i$
  Weight w_i decays by factor $1 - 2\epsilon\lambda$ if not reinforced by training.



weightdecayoff.pdf, weightdecayon.pdf (ESL, Figure 11.4) Write "10 hidden units + softmax + cross-entropy loss". [Examples of 2D classification without (left) and with (right) weight decay. Observe that the second example better approximates the Bayes optimal boundary (dashed purple curve).]

– Dropout emulates an ensemble in one network.



dropout.pdf

[During training, we temporarily disable a random subset of the units, along with all the edges in and out of those units. It seems to work well to disable each hidden unit with probability 0.5, and to disable input units with a smaller probability. We do stochastic gradient descent and we frequently change which random subset of units is disabled. The authors claim that their method gives even better generalization than ℓ_2 regularization. It gives some of the advantages of an ensemble, but it's faster to train.]

[Recall Karl Lashley's rat experiments, where he tried to make rats forget how to run a maze by introducing lesions in their cerebral cortexes, and it didn't work. He concluded that the knowledge is distributed throughout their brains, not localized in one place. Dropout is a way to strengthen this effect in neural networks.]

CONVOLUTIONAL NEURAL NETWORKS (ConvNets; CNNs)

[Convolutional neural nets have driven a big resurgence of interest in neural nets in the last decade. Often you'll hear the buzzword deep learning, which refers to neural nets with many layers. All the best image recognition networks are deep and convolutional. In 2018, the Association for Computing Machinery gave the Alan M. Turing Award to Geoff Hinton, Yann LeCun, and Yoshua Bengio for their work on deep neural networks.]

Vision: inputs are large images. 200 × 200 image = 40,000 pixels.
If we connect them all to 40,000 hidden units → 1.6 billion connections.
Neural nets are often overparametrized: too many weights, too little data.

[As a rule of thumb, if you have hugely many weights, you want a huge amount of data to train them. A bigger problem with having billions of weights is that the network becomes very slow to train or even to use.]

[Researchers have addressed these problems by taking inspiration from the neurology of the visual system. Remember that early in the semester, I told you that you can get better performance on the handwriting recognition task by using edge detectors. Edge detectors have two interesting properties. First, each edge detector looks at just one small part of the image. Second, the edge detection computation is the same no matter which part of the image you apply it to. So let's apply these two properties to neural net design.]

ConvNet ideas:

(1) Local connectivity: A hidden unit (in early layer) connects only to a small patch of units in previous layer.

[This improves the overparametrization problem, and speeds up both training and classification considerably.]

(2) Shared weights: Groups of hidden units share same set of weights, called a mask aka filter aka kernel.
[No relationship to the kernels of Lecture 16.] We learn several masks.
[Each mask operates on every patch of image.]
Masks × patches = hidden units in first hidden layer.
If one mask learns to detect edges, *every* patch has an edge detector.
[Because the mask that detects edges is applied to every patch.]
ConvNets exploit repeated structure in images, audio.
Convolution: the same linear transformation applied to different patches of the input by shifting.

[Shared weights improve the overparametrization problem, because shared weights means fewer weights. It's a kind of regularization.]

[But shared weights have another big advantage. Suppose that gradient descent starts to develop an edge detector. That edge detector is being trained on *every* part of every image, not just on one spot. And that's good, because edges appear at different locations in different images. The location no longer matters; the edge detector can learn from edges in every part of the image.]

[In a neural net, you can think of hidden units as features that we learn, as opposed to features that you code up yourself. Convolutional neural nets take them to the next level by learning features from multiple patches simultaneously and then applying those features everywhere, not just in the patches where they were originally learned.]

[By the way, although local connectivity is inspired by the human visual system, shared weights obviously don't happen in biology.]

[Show slides on computing in the visual cortex and ConvNets, available from the CS 189 web page at https://people.eecs.berkeley.edu/~jrs/189/lec/cnn.pdf . Sorry, readers, there are too many images to include here. The narration is below.]

[Neurologists can stick needles into individual neurons in animal brains. After a few hours the neuron dies, but until then they can record its action potentials. In this way, biologists quickly learned how some of the neurons in the retina, called retinal ganglion cells, respond to light. They have interesting receptive fields, illustrated in the slides, which show that each ganglion cell receives excitatory stimulation from receptors in a small patch of the retina but inhibitory stimulation from other receptors around it.]

[The signals from these cells propagate to the V1 visual cortex in the occipital lobe at the back of your skull. The V1 cells proved harder to understand. David Hubel and Torsten Wiesel of the Johns Hopkins University put probes into the V1 visual cortex of cats, but they had a very hard time getting any neurons to fire there. However, a lucky accident unlocked the secret and ultimately won them the 1981 Nobel Prize in Physiology.]

[Show video HubelWiesel.mp4, taken from YouTube: https://www.youtube.com/watch?v=IOHayh06LJ4]

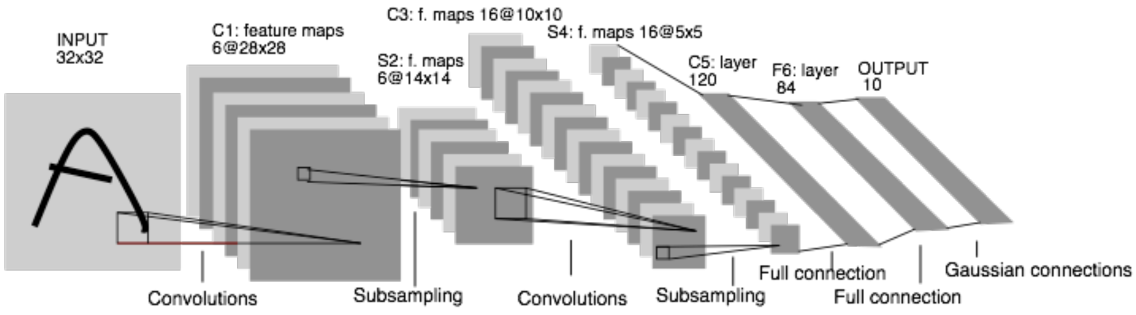
[The glass slide happened to be at the particular orientation the neuron was sensitive to. The neuron doesn't respond to other orientations; just that one. So they were pretty lucky to catch that.]

[The simple cells act as line detectors and/or edge detectors by taking a linear combination of inputs from retinal ganglion cells.]

[The complex cells act as location-independent line detectors by taking inputs from many simple cells, which are location dependent.]

[Later researchers showed that local connectivity runs through the V1 cortex by projecting certain images onto the retina and using radioactive tracers in the cortex to mark which neurons had been firing. Those images show that the neural mapping from the retina to V1 is retinatopic, i.e., locality preserving. This is a big part of the inspiration for convolutional neural networks!]

[Unfortunately, as we go deeper into the visual system, layers V2 and V3 and so on, we know less and less about what processing the visual cortex does.]



LeNet5.png Architecture of LeNet5.

[ConvNets were first popularized by the success of Yann LeCun's "LeNet 5" handwritten digit recognition software. LeNet 5 has six hidden layers! Hidden layers 1 and 3 are convolutional layers in which groups of units share weights. Layers 2 and 4 are pooling layers that make the image smaller. These are just hardcoded max-functions with no weights and nothing to train. Layers 5 and 6 are just regular layers of hidden units with no shared weights. A great deal of experimentation went into figuring out the number of layers and their sizes. At its peak, LeNet 5 was responsible for reading the zip codes on 10% of US Mail. Another Yann LeCun system was deployed in ATMs and check reading machines and was reading 10 to 20% of all the checks in the US by the late 90's. LeCun is one of the Turing Award winners I told you about earlier.]

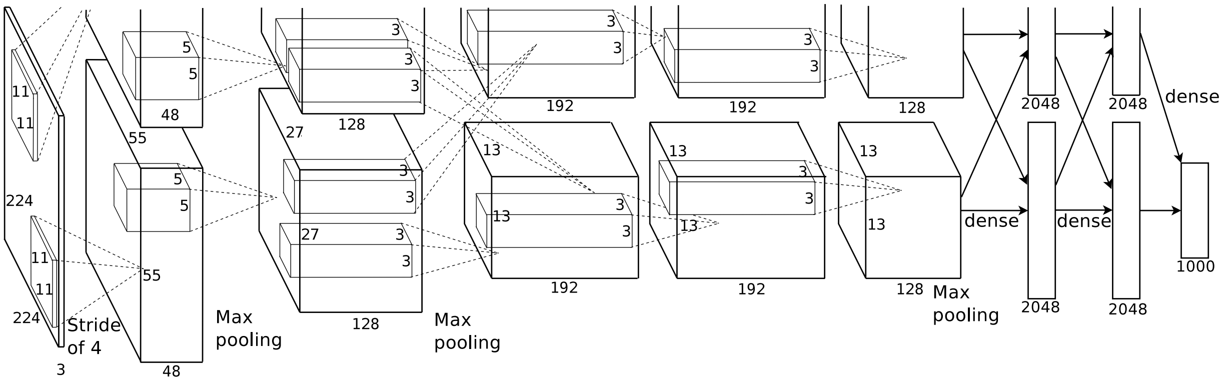
[Show Yann LeCun's video LeNet5.mov, illustrating LeNet 5.]

[When ConvNets were first applied to image analysis, researchers found that some of the learned masks are edge detectors or line detectors, similar to the ones that Hubel and Wiesel discovered! This created a lot of excitement in both the computer learning community and the neuroscience community. The fact that a neural net can naturally learn the same features as the mammalian visual cortex is impressive.]

[I told you two lectures ago that neural net research was popular in the 60's, but the 1969 book *Perceptrons* killed interest in them throughout the 70's. They came back in the 80's, but interest was partly killed off a second time in the 00's by . . . guess what? By support vector machines. SVMs work well for a lot of tasks, they're much faster to train, and they more or less have only one hyperparameter, whereas neural nets take a lot of work to tune.]

[Neural nets are now in their third wave of popularity. The single biggest factor in bringing them back is probably big data. Thanks to the internet, we now have absolutely huge collections of images to train neural nets with, and researchers have discovered that neural nets often give better performance than competing algorithms when you have huge amounts of data to train them with. In particular, convolutional neural nets are now learning better features than hand-tuned features. That's a recent change.]

[One event that brought attention back to neural nets was the ImageNet Image Classification Challenge in 2012. The winner of that competition was a neural net, and it won by a huge margin, about 10%. It's called AlexNet, and it's surprisingly similarly to LeNet 5, in terms of how its layers are structured. However, there are some new innovations that led to their prize-winning performance, in addition to the fact that the training set had 1.4 million images: they used ReLUs, dropout, and GPUs for training.]



alexnet.pdf Architecture of AlexNet.

[If you want to learn more about deep neural networks, there's a whole undergraduate class at Berkeley just on that topic: CS 182.]