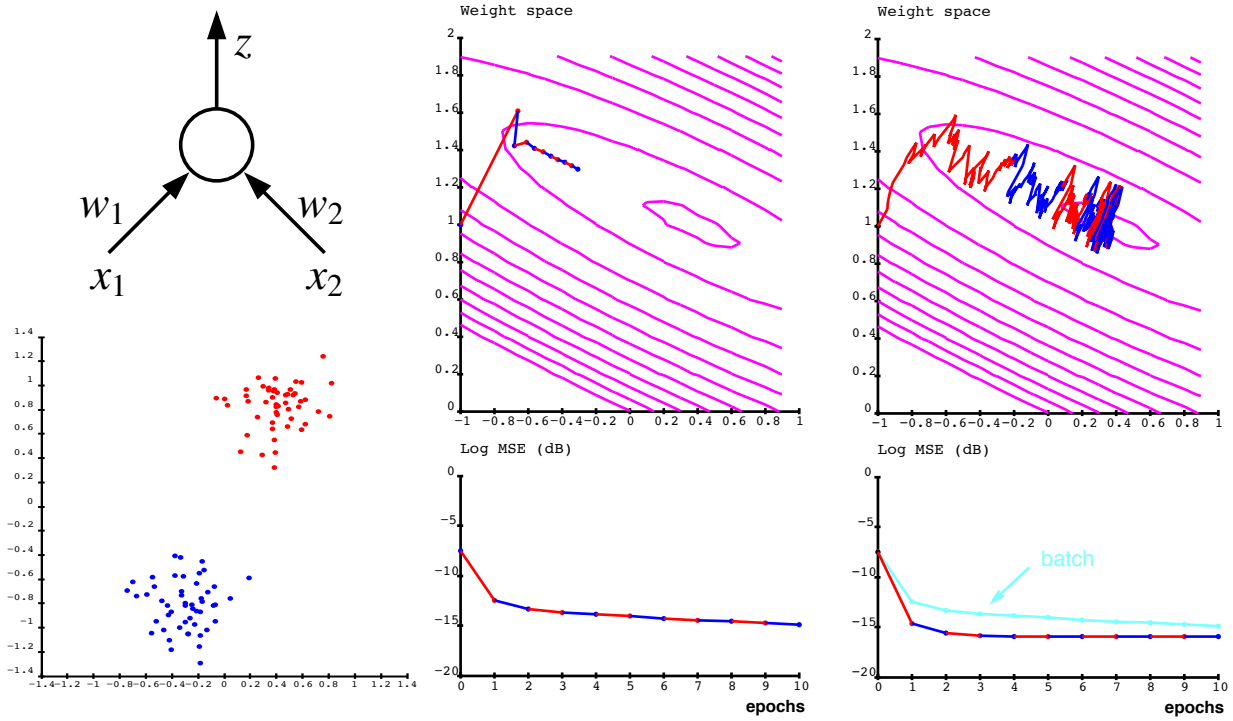
19 Better Neural Network Training; Convolutional Neural Networks

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 experiment with fancy heuristics only after you get that working.]

(1) Fix vanishing gradient problem. [As described in the previous lecture.]

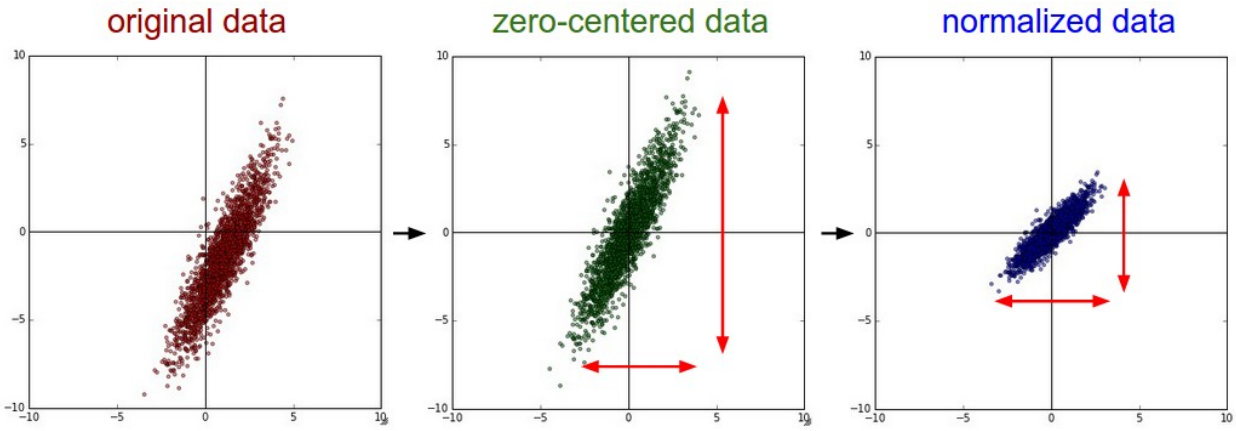
(2) Stochastic gradient descent (SGD): faster than batch on large, redundant data sets.
[Whereas batch gradient descent walks downhill on one cost function, stochastic descent takes a very short step downhill on one point's loss function and then another short step on another point's loss function. The cost function is the sum of the loss functions over all the sample points, so one batch step 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—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.]

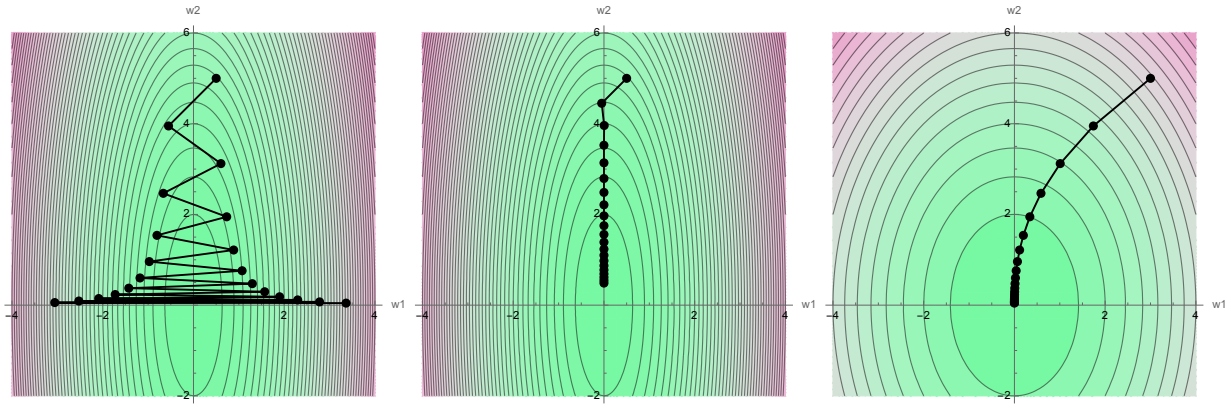
One epoch presents every training point once. Training usually takes many epochs, but if sample is huge [and carries lots of redundant information], SGD can take less than one epoch.

(3) 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.]

[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.]

[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.]

[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!]

(4) Use a small random subsample of training data to choose learning rate ϵ [or adaptive learning schedule]. [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 all your training points.]

(5) Different learning rate for each layer of weights. [Commonly we find that there are large variations in the magnitudes of the gradients in different layers of edges. A learning rate that's just right for one layer might be much too small to make progress in another layer. It looks like the illustrations of ill-conditioning on the previous page, but each axis represents a different layer of the network.]

(6) 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.
- Can do the same for high-loss examples. [E.g., perceptron alg. presents only misclassified examples.]
- Batch: cost fn is a weighted average of training pts.
  [Examples from rare classes can be more heavily weighted.]

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

(7) Acceleration schemes: Adam, AdaGrad, AMSGrad, RMSprop, momentum.
[These variations of stochastic gradient descent are quite popular. They speed up stochastic gradient descent by using adaptive learning rates and descent directions that aren't the direction of steepest descent. Look them up online if you're curious.]

DOUBLE DESCENT

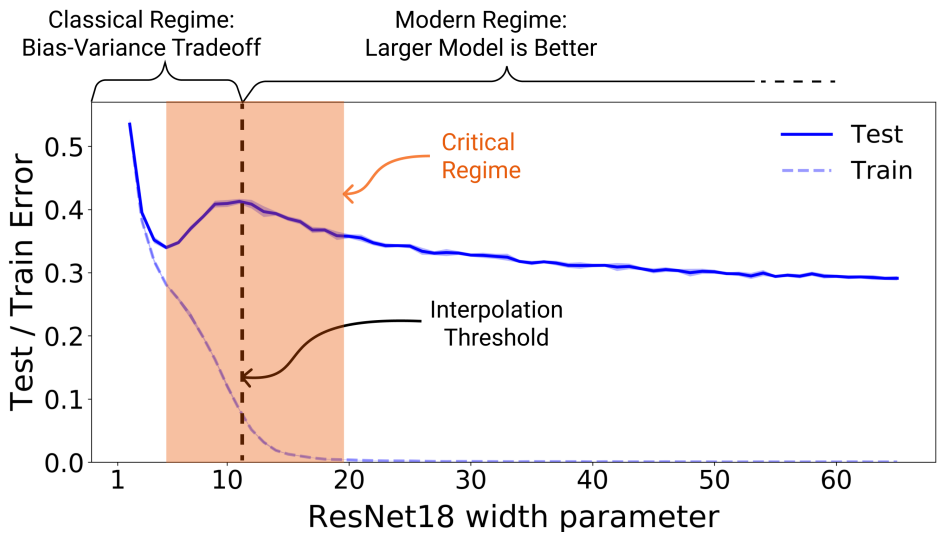
[Early neural network researchers sometimes struggled with their networks falling into bad local minima and failing to achieve low training errors. But with experience and greater computational power, we've discovered that these problems can usually be solved simply by adding more units to every hidden layer. We call this "making the network wider." Sometimes you also have to add more layers. But if your layers are wide enough, and there are enough of them, a well-designed neural network can typically output exactly the correct label for every training point, which implies that you're at a *global* minimum of the cost function.]

Hidden layers are wide enough + numerous enough ⇒ network output can interpolate the label ($z = y$) for every training pt ⇒ find global minimum of cost fn.

[Last lecture, I pointed out that if you use sigmoid or softmax output units, you can't set the labels to exactly 1 or 0 and achieve interpolation, as sigmoid and softmax outputs are strictly between 0 and 1; but with labels like 0.1 and 0.9, interpolating the labels is a realistic goal! And the linear output units used for regression can interpolate arbitrary numbers. If you fall into a bad local minimum, your network is too small.]

[One reason it took so long to make this discovery is that researchers believed that having too many weights in a neural network would cause overfitting. It turns out that's only half true. Empirically, we observe a phenomenon called "double descent," illustrated below.]

[Consider the solid blue curve, showing the test error as the width of a network increases. The horizontal axis is the number of units per hidden layer. As that number increases, at left the test error exhibits the classic U-shaped bias-variance tradeoff. But when we pass the point where the network is interpolating the labels and continue to add more weights, we typically see a second "descent," where the test error starts to decrease again and ultimately gets even lower than before! The peak in the middle of the curve tends to be larger when there is more noise in the labels. Observe that the test error continues to fall even after the training error is zero. The takeaway is, "bigger models are better."]



doubledescent.pdf (Nakkiran et al., "Deep Double Descent") [A classic double descent curve (solid blue) for test error. The horizontal axis indicates the number of units in each hidden layer of a residual neural network used for image recognition, and the vertical axis measures the the test error (solid curve) and training error (dashed curve).]

[The currently accepted explanation for double descent, per Nakkiran et al., is that "at the interpolation threshold ... the model is just barely able to fit the training data; forcing it to fit even slightly-noisy or mis-specified labels will destroy its global structure, and result in high test error. However for over-parameterized models, there are many interpolating models that fit the training set, and SGD is able to find one that 'absorbs' the noise while still performing well on the distribution."]

[Double descent has also been observed in decision trees and even in linear regression where we add random features to the training points (thereby adding more weights to the linear regression model).]

HEURISTICS FOR BETTER GENERALIZATION

[Classic methods for preventing overfitting, such as ℓ_2 regularization and ensembles of learners, sometimes help neural networks to generalize better to points they haven't been trained on.]

(1) ℓ_2 regularization, aka weight decay.

Add $\lambda \|w\|^2$ to the cost/loss fn, where w is vector of all weights in network.

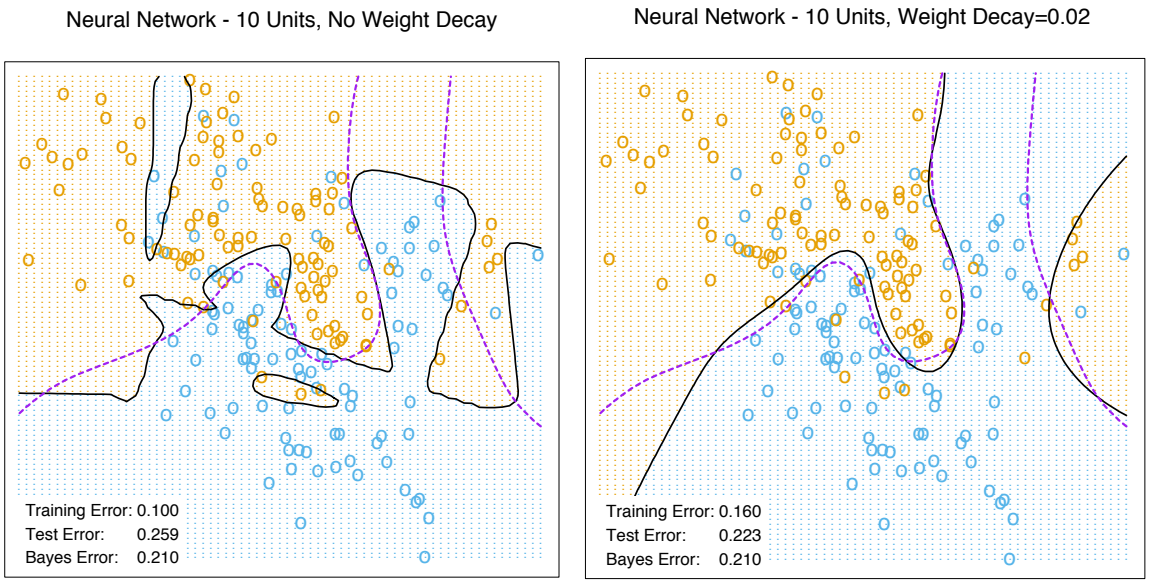
[w includes all the weights in all the weight matrices, rewritten as a vector.]

[We regularize for the same reason we do it in ridge regression: we suspect that overly large weights are spurious.]

[With a neural network, it's not clear whether penalizing the bias terms is bad or good. Penalizing the bias terms has the effect of drawing each ReLU or sigmoid unit closer to the center of its nonlinear operating region. I would suggest to try both ways and use validation to decide whether you should penalize the bias terms or not. Also, you could try using a different hyperparameter for the bias terms than the λ you use for the other weights.]

Effect: step $\Delta w_i = -\epsilon \dfrac{\partial J}{\partial w_i}$ has extra term $-2\epsilon\lambda\, w_i$

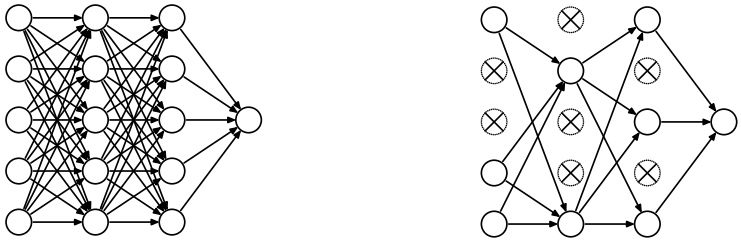
Weight w_i decays by factor $1 - 2\epsilon\lambda$ if not reinforced by training.



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 in the second example, the decision boundary (black) better approximates the Bayes optimal boundary (dashed purple curve).]

(2) 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.]

(3) Dropout emulates an ensemble in one network.



dropout1.pdf, dropout2.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 better generalization than ℓ_2 regularization. It gives some of the advantages of an ensemble, but it's faster to train. I'm leaving out many important details; see Hinton et al., "Improving neural networks by preventing co-adaptation of feature detectors."]

[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 images. 200×200 image = 40,000 pixels.
If we connect them all to 40,000 hidden units $\rightarrow$ 1.6 billion connections.
Vision nets can be overparametrized: too many weights.

[As we saw in our discussion of double descent, this can give you excellent generalization to new examples. The problem with having billions of weights is that the network becomes very slow to train or even to use.]

[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. 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 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 $\times$ 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 are a kind of regularization, because shared weights means fewer weights. It's unlikely that a weight will become spuriously large if it's used in many places.]

[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 wherever they are.]

[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, local connectivity was inspired by the human visual system, as well as by techniques used in image processing. However, shared weights 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]

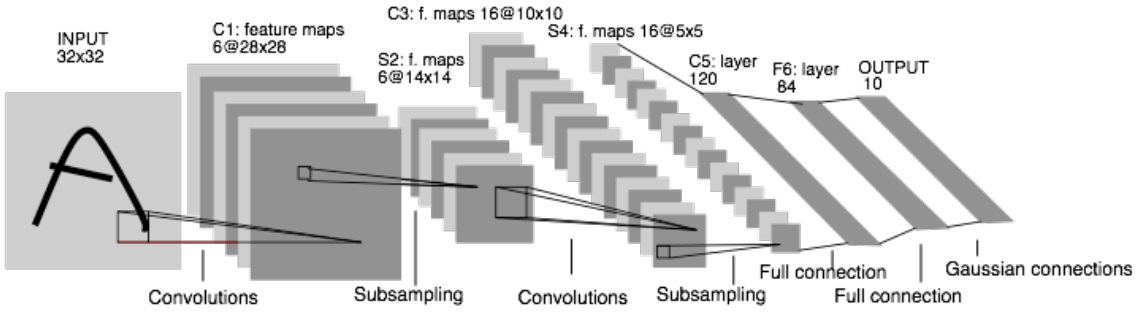
[The glass slide happened to be at the particular orientation the neuron was sensitive to. The neuron doesn't respond to other orientations; just that one. So they were pretty lucky to catch that.]

[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.]

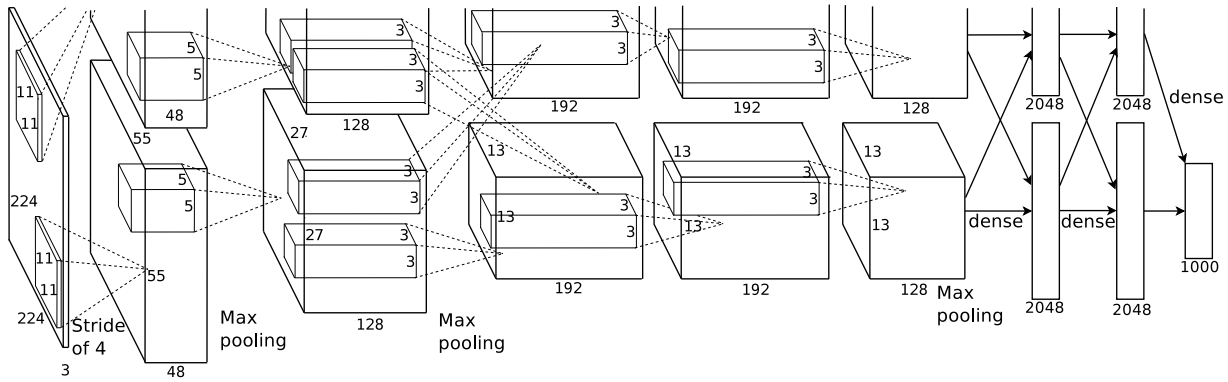
[Show Yann LeCun's video LeNet5.mov, illustrating LeNet 5.]

[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 artificial intelligence 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.]

[The 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.]