Neural Networks

36-462/662, Spring 2022

5 April 2022 (Lecture 21)

Contents

From Logistic Regression to Multi-Layer Neural Networks

Let's look again at the probability which is output by logistic regression (rather than the class label). It's

$$\mathbb{P}\left(Y = 1 | \vec{X} = \vec{x}\right) = \frac{e^{b+\vec{x}\cdot\vec{w}}}{1+e^{b+\vec{x}\cdot\vec{w}}} \tag{1}$$

This takes an input vector $\vec{x} = (x_1, x_2, \ldots x_p)$ and calculates a weighted sum of its components, $\vec{x}\cdot\vec{w} = \sum_{j=1}^{p} x_j w_j$. It then adds on a constant. This can give us a number, call it η, anywhere between $-\infty$ and $+\infty$. We then calculate $e^{\eta}/(1+e^{\eta})$. This squashes the number back into the range $[0,1]$. To speak like an engineer, the output p is very nearly linear in η when $\eta \approx 0$, but when η is very large ($\to \infty$) or very small ($\to -\infty$), the output **saturates**. This kind of behavior is typical of lots of nonlinear devices, and in

particular it serves as a crude caricature of how nerve cells or **neurons** behave. This led to the idea that if we can compute simple things, like classification, with one device of this sort, maybe we can compute more complicated functions with many of them wired together.

Formally, a **feed-forward artificial neural network** is defined by having **layers** of such devices, where the output of one layer provides the input to the next. The convention is to count the input variables as the first layer, so logistic regression is an example of a two-layer neural network. If we add an **intermediate** or **hidden** layer with m nodes between the inputs and the outputs, one has a model like so:

$$
\begin{aligned}
t_k &= f_h(\alpha_k^{(1)} + \sum_{l=1}^{p} x_l w_{lk}^{(1)}) && (2)\\
&= f_h(\eta_k(\vec{x})) && (3)\\
s_j &= f_o(\alpha_j^{(2)} + \sum_{k=1}^{r} t_k w_{kj}^{(2)}) && (4)\\
&= f_o(\xi_j(\vec{x})) && (5)
\end{aligned}
$$

Here we have an input vector $\vec{x}$, of dimension p, and an output vector $\vec{s}$, of dimension q. t_k is the output of the k^{th} hidden node (there are m of them). f_h is the **activation**, **response** or **squashing** function of the hidden nodes, and f_o is the activation function of the output nodes. The weights $w_{lk}^{(1)}$ and $w_{kj}^{(2)}$ control how strongly the output of k is influenced by j. Similarly the biases $\alpha^{(1)}$ and $\alpha^{(2)}$ set base-line levels of activation for each node[1]. I will write the over-all output of the neural network on inputs $\vec{x}$ as $\phi(\vec{x}; \alpha, \mathbf{w})$. It is, I hope, clear how we could extend this notation to multiple intermediate or hidden layers, which will become important later.

Using a three-layer neural network thus deciding on the number m of hidden units, what the activation functions should be, and on what the weights should be.

Take m as given for now; we'll come back to that.

The activation functions are almost always chosen to be monotonic, and if we want to be able to estimate efficiently they should also be smooth. (We'll come back to why in a little bit.) Otherwise, appropriate choices depends on what we want to do. Common activation functions are linear (the identity function),

$$f(\eta) = \eta \ , \tag{6}$$

"rectified linear" (a.k.a. "ReLU" for "rectified linear unit"),

$$f(\eta) = \eta \mathbb{1}_{\{\eta > 0\}} \tag{7}$$

logistic

$$f(\eta) = e^{\eta}/(1 + e^{\eta} \ , \tag{8}$$

and the hyperbolic tangent

$$f(\eta) = \tanh \eta = \frac{e^{\eta} - e^{-\eta}}{e^{\eta} + e^{-\eta}} \tag{9}$$

Most often, f_h is the hyperbolic tangent, and f_o is linear for regression problems, or logistic for classification problems[2].

So: fix the number of neurons, fix the activation functions, and "all" that's left to find are the weights. This is called **training**.

[1] It's common to introduce an extra input variable, call it x_0 or t_0, which is always equal to 1. Then we can replace the off-sets $\alpha_k^{(1)}$ and $\alpha_j^{(2)}$ with weights $w_{0l}^{(1)}$ and $w_{0j}^{(2)}$. Also, some people like to allow layer-skipping connections, directly from inputs to outputs; this raises no point of principle but complicates the notation a bit. Finally, some people like to number the units consecutively across all the layers, so that one just needs a single weight matrix w, rather than two. Again, this doesn't raise any point of principle.

[2] Another way to approach classification is to have one output node for each class, with a linear activity function, and then take the probability of class j to be $e^{s_j} / \sum_{j=1}^{q} e^{s_j}$; this ensures that probabilities sum to 1.

Gradient Descent and Backpropagation

People usually train neural networks by gradient descent. Remember that we have training data $(\vec{x}_1, y_1), \ldots (\vec{x}_n, y_n)$, and a loss function ℓ. If we set all the weights and biases, the empirical risk will be

$$\hat{R}(\alpha, \mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \phi(\vec{x}_i; \alpha, \mathbf{w})) \tag{10}$$

(The empirical risk is implicitly a function of the training data, but I am suppressing that in the notation for now.)

If we want to optimize this, we're going to need to take derivatives with respect to the parameters.

Output-layer derivatives

Taking the derivative with respect to parameters for the output layer is straightforward, if long-winded, using the chain rule:

$$\frac{\partial \hat{R}}{\partial w^{(2)}_{jk}} = \frac{1}{n}\sum_{i=1}^{n} \left.\frac{\partial \ell(y_i, a)}{\partial a_j}\right|_{a=\phi(\vec{x}_i;\alpha,\mathbf{w})} \left.\frac{df_o}{du}\right|_{u=\xi_j(\vec{x}_i)} t_k(\vec{x}_i) \tag{11}$$

This is a mouthful, so let's think it through for a simple case, specifically regression with a squared-error loss function. Because we're predicting a single real number, we have *one* output node, hence $q = 1$. The derivative $\frac{\partial \ell(y_i,a)}{\partial a} = 2(y_i - a)$, a.k.a. (twice) the residual. So in the regression case, we'd get

$$\frac{\partial \hat{R}}{\partial w^{(2)}_{j1}} = \frac{1}{n}\sum_{i=1}^{n} \text{residual}_i \left.\frac{df_o}{du}\right|_{u=\xi(\vec{x}_i)} t_j(\vec{x}_i) \tag{12}$$

The first factor in the sum is "how does the loss change (on this data point) if the output is increased slightly?" The second factor in the sum is "how much does the output change if the total input to that node is increased slightly?"[3] The third factor is "how much does the total input to the node change if the weight we're interested in is increased slightly?"

If we are using a different loss function, all that changes is that the first factor is no longer (twice) the residual, but some other slope saying "how much would the loss on this data point increase if the output of this final-layer neuron increased a little?" You can, if you like, think of such a slope as a sort of generalized residual. If we've got multiple output-layer neurons, we need to take the derivatives for each of them, but that's a calculation we can do "in parallel" for all of them.

(I will leave it as a character-building exercise to take the derivatives with respect to the biases $\alpha^{(2)}_k$.)

Hidden-layer derivatives

Now suppose we want to take the derivative with respect to one of the weights going into the hidden layer, $w^{(1)}_{lk}$. We use the chain rule again, and we recognize that changing $w^{(1)}_{lk}$ is only going to change the loss if doing so changes the activation of an output node, which is only going to happen if doing so changes the

[3] Also, notice that this is the only place where the shape of the activation function appears explicitly. If the activation function was linear, this derivative would always be 1. If the activation function is "rectified linear", the derivative is always 0 (below the threshold) or 1 (above it); this simplicity is one of the reasons ReLU neural networks are popular. If we use a logistic or hyperbolic-tangent activation function we need to do more calculation, but the derivative of $\tanh u$ is (exercise!) $\left(\frac{2}{e^u + e^{-u}}\right)^2$ which is easy enough. Further exercise: find the derivative of the logistic activation function.

total input to an output node. So we get, from the chain rule,

$$\begin{aligned}\frac{\partial \hat{R}}{\partial w_{lk}^{(1)}} & \\ = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{q} \left.\frac{\partial \ell(y_i, a)}{\partial a_j}\right|_{a=\phi(\vec{x}_i;\alpha,\mathbf{w})} \left.\frac{df_o}{du}\right|_{u=\xi_j(\vec{x}_i)} w_{jk}^{(2)} \left.\frac{df_h}{dv}\right|_{v=\eta_k(\vec{x}_i)} x_{il}\end{aligned} \tag{13}$$

Again, this is a mouthful, but again, we can make some sense of it by taking it piece by piece.

The first factor in each term being averaged is the generalized residual for that data point. The second factor is how much the output of each final-layer unit changes as we change its total input. The third factor is how much the total input to final-layer unit j changes in response to a small change in the output of hidden-layer unit k. We need to sum these three factors, multiplied togehter, over all final-layer units, because we're contemplating a change to how the hidden-layer unit k works, which could affect any or all of the final-layer units. (If we're doing regression or binary classification we only have one final-layer unit, so that sum over j would go away.) We then weight this by how much the output of unit k changes in response to a small change in its total input, and then by how much a small change in the weight unit k gives to feature l will change the total input.

You can imagine extending this if there were multiple hidden or intermediate layers. Doing so will illuminate why people who are into neural networks are also into automatic differentiation.

The backpropagation trick

Something to notice about what we've just done is that a *lot* of the work for the hidden-layer units is also needed for the final-layer units.

Let's say that for each final-layer unit j, we calculate

$$b_{ij} = \left.\frac{\partial \ell(y_i, a)}{\partial a_j}\right|_{a=\phi(\vec{x}_i;\alpha,\mathbf{w})} \left.\frac{df_o}{du}\right|_{u=\xi_j(\vec{x}_i)} \tag{14}$$

which is the generalized residual for that unit, times the slope of the unit's activation function: "how much would increasing the output of j change the loss, *times* how rapidly does the output change when we change the total input?"

Similarly, for each hidden-layer unit k, we calculate

$$b_{ik} = \sum_{j=1}^{q} b_{ij} w_{jk} \left.\frac{df_h}{dv}\right|_{v=\xi_k(\vec{x}_i)} \tag{15}$$

That is, we sum the b_{ij} weighted by w_{jk}, and we multiply everything by the derivative of k's output with respect to its total input.

Now

$$\frac{\partial \hat{R}}{\partial w_{jk}^{(2)}} = \frac{1}{n}\sum_{i=1}^{n} b_{ij} t_j(\vec{x}_i) \tag{16}$$

but

$$\frac{\partial \hat{R}}{\partial w_{lj}^{(1)}} = \frac{1}{n}\sum_{i=1}^{n} b_{ik} x_{il} \tag{17}$$

We can extend this to even more layers, if we need to, by defining b_{il} similarly: the sum of the b_{ik} over the neurons in the next layer, times he weights, times the slope of the activation function.

This idea was introduced by Rumelhart, Hinton, and Williams (1986), under the name of **back-propagating error** or **back-propagation**, and what I have called the b_{ij}, b_{ik}, b_{il} are the **back-propagation signals**. These provide an extremely handy way of calculating the derivatives we need for optimizing the weights[4].

Gradient descent and stochastic gradient descent

Here's a classic way of doing gradient descent for neural networks:

- Start with some initial weights (and biases) for all the units.
- Set $t = 1$
- Until the weights stabilize, or $t = t_{max}$:
  - Cycle through the data points from 1 to n
    - For each data point i, compute the back-propagation signals b_{ij} and then b_{ik} using the current weights
    - Adjust the weights: $w_{jk} \rightarrow w_{jk} - \gamma_t b_{ij} t_j(\vec{x}_i)$, $w_{lk} \rightarrow w_{lk} - \gamma_t b_{ik} x_{il}$, where the **gains** γ_t gradually decrease towards zero
    - $t \rightarrow t_+ 1$

You can, of course, imagine all kinds of variations here. For instance, maybe we can save a lot of time by calculating the back-propagation signals once on each pass through the data, rather than adjusting after each data point; maybe we change the order in which we visit data points on each pass.

One particularly important variation is that we might not visit_every_ data point on each cycle. In particular, we might randomly sample a smaller number of data points r, perhaps even just $r = 1$. As we saw when we looked at such stochastic gradient descent in general, *in expectation* this gives the correct derivatives, and that's still true when we use back-propagation to calculate the derivatives. This can be *extremely* useful when the data size is very large.

This combination of back-propagation to efficiently calculate derivatives, and gradient descent or stochastic gradient descent, is not the only possible way to train neural networks, but it's overwhelmingly the most popular.

How many layers? How many neurons per layer?

Two-layer neural networks, a.k.a. "perceptrons"

Two-layer neural networks, also called "perceptrons" after Rosenblatt (1958), are pretty straightforward:

$$s_j = f_o(\alpha_j + \sum_{l=1}^{p} x_l w_{jl}) \tag{18}$$

We take a linear combination of the input features and apply a single nonlinearity to them (e.g., a logistic transformation). These are what econometricians would call **single-index models**. (What's that single index?) Because f_o is monotonic, all the classifications we can realize by thresholding the output are ones we could achieve *without* the transformation, by applying a different threshold to the linear form $\alpha_j + \sum_{l=1}^{p} x_l w_{jl}$.

So: two-layer neural networks are just a slight variant on linear models. In particular, as classifiers they really are just linear classifiers.

[4]While Rumelhart, Hinton, and Williams (1986) made the technique a permanent part of the neural-network literature, and introduced the handy name "back-propagation", there is a complicated history of independent re-inventions, of greater or lesser generality, going back to the 1960s at least. (Since it's just an application of the chain rule, in principle it could have been discovered at any time since the early 18th century.) The reference works given under "Further Reading" below discuss some of this history.

Three-layer neural networks, a.k.a. multi-layer perceptrons

Three-layer neural networks are vastly more powerful than two-layer neural networks. In particular, with the right choice of smooth, nonlinear activation functions (logits and hyperbolic tangent very much included), we have (roughly) the following result:

> Let $g : \mathbb{R}^p \mapsto \mathbb{R}^q$ be any smooth function, and pick any limited domain $D \in \mathbb{R}^p$. Then, by taking m large enough, we can always find a three-layer network with m nodes and weights $\mathbf{w}$ so that $\max_{\vec{x} \in D} \|g(\vec{x}) - \phi(\vec{x}; \mathbf{w})\| \leq \epsilon$.

That is, by throwing enough hidden units at the problem, we can approximate an arbitrary (smooth) function arbitrarily closely. We say that three-layer neural networks are **universal approximators**.

Now, many other combinations of basic functions can also be universal approximators, in this sense. The most obvious are polynomials (a result known as **Weierstrass's theorem**), but one can also use combinations of sines and cosines, and many other function families; the general result in mathematical analysis which covers this is what's called the **Stone-Weierstrass* theorem. (Indeed, most proofs of the result I just gave use the Stone-Weierstrass theorem.) But it's still remarkable that adding just one layer increases the power of the models so remarkably, from "basically linear" to "basically anything".

Deriving new features, and cautions about interpretation

The final layer of the neural network is doing a (basically) linear operation on the output of the hidden layer. (Or, with even more layers, on the output of the next-to-final layer.) So it is very tempting to think that what's happening is that the hidden layer is computing new features from the input variables, and then the final layer is applying a linear method to those new features. Thus training the neural network is in some sense discovering new features, or learning a new representation of the data.

This is, I think, basically right, but it requires the caution that it's very hard to interpret those new features.

Specifically, one is tempted to try to give some human, understandable story about what each hidden-layer unit is interpreting. One common procedure for coming up with such a story about hidden-layer unit k is to take the trained network, and then look at the input vectors $\vec{x}_i$ which maximize t_k. Then one tries to figure out what those input vectors have in common, and so what unit k is representing.

The difficulty here is that unit k doesn't really do *anything* on its own, but only in concert with all the other hidden-layer units. The hidden layer as a whole provides a representation of the data, but it's generally a **distributed representation**, not (necessarily) localized to particular units. Neuroscientists studying biological networks long ago coined the term "grandmother cell fallacy" to describe the difficulty: The fact that you can recognize your grandmother's face (and distinguish it from other people's faces, from her hands, from non-faces, etc.) means that the biological neural network in your brain must, somehow, represent your grandmother's face. It does not follow that there is one particular neuron which is activated when you see her face!

So, as a point of theory, we shouldn't *expect* individual hidden-layer units to represent humanly-meaningful features, even if the hidden layer acts as a distributed representation. Concretely, Szegedy et al. (2013) did the experiment of creating random weights Z_k and looking at the inputs which maximizes $\sum_{k=1}^{m} Z_k t_k(\vec{x})$ in trained neural networks that worked well at image classification[5]. The images that maximized the output of random combinations of hidden-layer units were just as coherent, and just as easy to tell stories about and interpret, as those which maximized the activation of individual hidden-layer units[6]. Again, this is compatible with the hidden layer *as a whole* being a distributed representation of the data, in terms of new features.

[5] Actually, the networks they looked at were "deep" networks with many more than three layers, but the idea is the same.

[6] Zhu, Rogers, and Gibson (2009), in a series of experiments which deserve to be much better known than they are, showed that humans — or at least undergrads at UW-Madison — have a pretty impressive ability to come up with stories that let them interpret, and memorize, quite long collections of literally-random dictionary words. It would be very interesting to know how big a collection of random images from the data sets used to train these neural networks could be "interpreted", and memorized, by graduate students and professors of machine learning.

Auto-encoders, dimension reduction, embeddings

The universal approximations result say that with *large enough* hidden layers, we can come arbitrarily close to any function we like. This often leads to situations where $m > p$, so we expand out into a higher-dimensional space. (This should remind you of the kernel trick.) There are particularly interesting situations, however, where $m < p$, so we try to **encode** or **compress** the relevant aspects of the input into a smaller number of dimensions. This doesn't always work — there isn't a guarantee that a good compression exists — but when we can pull it off we've done dimension reduction.

An extremely interesting example of the bottleneck situation is when the output dimension and the input dimensions are equal, $q = p$, and we use a loss function which simply rewards reproducing the input, *but* the hidden layer has fewer units, $m < p$. A network which can do this is called an **auto-encoder**. This will be possible when the original high-dimensional input is actually highly redundant, and, to a good approximation, has a much smaller number of dimensions, *and* we can find a way of recovering the input from that compressed representation. This should remind you of PCA; you can think of auto-encoders as being like a nonlinear sort of PCA, or PCA as being a linear auto-encoder.

When we *can* make $m \ll p$ and still get good results, we sometimes refer to the hidden layer as a **bottleneck** which the information needs to pass through. Exactly what the connections are here to the "information bottleneck method" introduced by Tishby, Pereira, and Bialek (1999) is an active subject of investigation, but many researchers strongly suspect that this is a big part of what's going on in neural networks.

If we have an input x which the low-dimensional hidden layer transforms into a point $\vec{t}(x) \equiv (t_1(x), t_2(x), \ldots t_m(x)) \in \mathbb{R}^m$, we sometimes refer to $\vec{t}(x)$ as an **embedding** of x into the vector space $\mathbb{R}^m$. Once we've represented the input, whatever it might be, as a point in a vector space, we can do all kinds of things to it, like do similarity search, or analogy-completion tasks (cf. Kearns and Roth (2019), pp. 57–63). Again, you can think of this as one way of generalizing what we did with PCA.

An outstanding example of successfully-applied neural network embeddings is provided by word embeddings, beginning with Mikolov et al. (2013). The basic idea here is to try to predict the occurrence of a word in text as a function of the other words around it, with each word being represented by a not-too-high-dimensional vector. The original paper on this `word2vec` technique was sufficiently opaque to lead to a series of follow-up papers explaining what, exactly, was going on (like Goldberg and Levy (2014)), but it also worked really well on lots of natural-language-processing tasks. In particular, it worked much better than previous methods of embedding words in vector spaces based on PCA (Baroni, Dinu, and Kruszewski 2014). Since Mikolov et al. (2013) appeared, neural word embedding models have gotten much more elaborate, and there has been a flurry of effort to devise similar sorts of embedding methods for other kinds of data, e.g., nodes in graphs.

Random features

Remember that when we looked at random features, we looked at models of the form

$$s(\vec{x}) = \sum_{k=1}^{m} w_k f(\vec{x}; \omega_k) \tag{19}$$

where f was a *fixed* nonlinear function, and the parameters ω_k were drawn randomly from some distribution ρ, leaving only the weights w_k to be adjusted to minimize the risk. In particular, we got a lot of good results with random Fourier features, so $f(\vec{x}; \omega) = \cos(\vec{x} \cdot \nu + \delta)$, breaking the $(p+1)$-dimensional ω into a p-dimensional "wave vector" ν and a scalar "phase" δ. However, there was nothing magical (as opposed to convenient) to using random Fourier features, as opposed to, say, random hyperbolic tangent features.

We introduced random feature models as a way of efficiently approximating kernel machines when the number of training data points grew large — instead of dealing with an $n \times n$ kernel matrix, we could deal with just an $m \times m$ covariance matrix of the random features, and get nearly the same results. We can see now that *another* way of understanding random feature models is as three-layer neural networks, where the weights connecting the input features to the hidden layer are randomly generated independently of the data, *and*

then fixed during training. Instead, only the weights connecting the hidden layer to the output layer are optimized, eliminating the need for back-propagation.

Deep neural networks and deep learning

A "deep" neural network, as opposed to a shallow one, is simply one with more than three layers — the more layers it has, the deeper it is. "Deep learning" is simply fitting a deep neural network, which is, overwhelmingly, done by the combination of back-propagation and stochastic gradient descent sketched above. (See more below.)

Over the last ten years or so, deep neural networks have proved *extremely* effective at a lot of tasks relating to image classification, speech recognition, and natural language processing. This is actually puzzling, because nobody has managed to get equivalent performance out of shallow-but-wide three-layer networks, even though we know, mathematically, that deep networks have no more expressive power than shallow ones. (Once you're already a universal approximator, there's not really much room to expand your scope.) This poses a lot of puzzles for statistical learning.

Not just back-prop and SGD...

I said a moment ago that deep neural network training is *mostly* just back-propagation and stochastic gradient descent. The "mostly" conceals a lot of bells and whistles which have been added to that basic contraption, either to speed up training or to improve the performance of the learned machine. We could easily spend the better part of a course going over all these modifications. It is not at all clear *which* of these tweaks actually improve training, how much they improve training, or *why* they improve training. While there are people actively working on all of those problems, I think it's fair to say that they have not achieved the level of clarity and consensus that it'd be worth discussing the details here. But if you are going to work with these models on an industrial scale, then I am afraid you are going to have to plunge into those details — but they also change very rapidly, so, again, I am not going to go over ideas which will soon be obsolete.

Choice of number of layers and number of units per layer

These are different model classes; use cross-validation.

(Or any other good way of estimating generalization error.)

Beyond feed-forward networks

The fact that I called the models I've described "feed-forward" should suggest that there is a "feed-back" alternative. There are many. The most common are **recurrent** neural networks, where the inputs to units in early layers include the outputs of units in later layers. Feed-forward networks are just ways of specifying functions, mappings from $\mathbb{R}^p$ to $\mathbb{R}^q$. Recurrent networks aren't functions, or, if you like, aren't always the *same* function, because the history of previous inputs matters. They describe dynamical systems or automata[7], rather than just functions. Training them is harder; I defer to references under "Further Reading".

Some history and some puzzles

Artificial neural network models, almost as we know them, originated with McCulloch and Pitts (1943) — that is, with biologists and psychologists using a deliberately-simplified model of how biological neurons

[7] Specifically, what automata-theorists call **transducers**, which map sequences of inputs to sequences of outputs.

interact with each other to show that processes in the brain could implement abstract logic[8].

The next turn towards what we call neural networks came with Rosenblatt (1958), whose perceptrons were, in our terms, two-layer neural networks used as linear classifiers for various pattern-recognition problems[9]. This was greeted by the press with stories about the arrival of "thinking machines", worries about what people would do when jobs were automated away, domination by artificial intelligence, etc.

There was *also* serious scientific research into the possibilities and limitations of these models. In particular, Minsky and Papert (1969), recognizing that perceptrons could only do linear classification, established conclusive that there were lots of pattern-recognition tasks they just could not do, no matter how much data they got or how they were trained. This more or less killed the field for almost two decades.

The revival of interest in neural networks, at the time also called "connectionism", came mostly from psychology. The two important ideas were the recognition that adding one or more intermediate layers could drastically expand the expressive power of these models, *and* that the intermediate layers could act as distributed representations. In the late 1980s and the early 1990s, there was a *lot* of interest, both scientific and popular, in the genuinely-impressive results which people got on various artificial problems, especially when using back-propagation (Rumelhart, Hinton, and Williams 1986). There were again all kinds of stories about thinking machines, "naturally intelligent systems"[10], technological unemployment, artificial intelligence, etc. Interest in neural networks actually played key role in the development of "statistical learning" as a field (e.g., Anthony and Bartlett (1999)), which is why one of our leading conferences is called "Advanced in Neural Information Processing Systems".

Neural networks nonetheless faded from popularity for about two decades between the mid-1990s and the early-2010s, for two reasons.

1. *As predictive models*, it turned out that people were able to get much higher performance using *different* kinds of models. In particular, kernel and support vector machines were, for a long time, the state-of-the-art, random forests had (and have) their partisans, etc.
2. Even when neural networks could generalize in the statistical sense, i.e., predict well on new data points drawn from the same distribution, they did not find it easy to *generalize* in ways that humans find easy. If instead of using an 8×8 board to play checkers one introduces a 9×9 board, a human player will be a bit hesitant, but not totally at a loss of what to do; a neural network would need to be completely retrained. More generally, neural networks have found it *extremely* hard to incorporate abstract concepts, and general rules using those abstractions. It's clear that abstractions and rules *can* be implemented using neural networks, but it wasn't clear how to *learn* them, and back-propagation, starting from pretty un-structured starting points, seemed to be very bad at doing so. (Marcus (2001) remains a very good statement of this line of criticism.)

The second issue is a real one for cognitive science, psychological modeling and artificial intelligence, but less important for statistical learning. Nonetheless, the combination of the two put neural networks into eclipse, again[11].

All of this changed around 2012–2013, when people began publishing extremely impressive *statistical* results

[8] If you look at some of the classics of late-19th century psychology, like James (1890), you will see lots of diagrams which *look* an awful lot like pictures of neural networks. These were speculations about some sort of "activity" or "energy" or "current" flowing through distinct regions of the brain, accompanying, or implementing, processes of thought. It was only around 1900 that the great anatomist Santiago Ramon y Cajal proved that the nervous system is divided into distinct cells, which he named "neurones", rather than being (as some had held) a single continuous mass. The idea that the nervous system works by neurons sending *something* from cell to cell, at junctions called "synapses", and that synapes work in one direction only, was established by the great physiologist Charles Sherrington, who at least sometimes referred to the nervous system as a "network". (Sherrington (1906) is still a remarkably insightful work.) It wasn't until the 1920s that it became clear that neurons sent electrical impulses *to* their synapses, but the transmission at the synapse is chemical. It then became clear that a neuron's electrical "spike" was only triggered if the total stimulation it received crossed some threshold. This is what McCulloch and Pitts (1943) modeled in a deliberately simplified way, and what turned into the activiation function.

[9] I am simplifying, and in particular glossing over the extremely important work done by Hebb (1949), Hayek (1952) and the now-neglected but actually very-influential Ashby (1960).

[10] The title of one of the best of the semi-popular books at the time, Caudill and Butler (1990), which I still actually recommend.

[11] When I started teaching the predecessor to this course in 2006 and included neural networks, the feedback was that I was wasting the students' time with an obsolete technique, and by 2009 I'd dropped the topic.

using deep neural networks on image classification and natural-language-processing tasks. There were a couple of reasons why the new models worked better than their predecessors of 15–20 years before.

1. New architectures. In particular, for image-processing tasks, it turned out to be extremely important to use "convolutional" neural networks[12], rather than unstructured ones.
2. Improved training. In particular, the bells and whistles added on to back-propagation and stochastic gradient descent that I alluded to.
3. Lots of data. Truly startingly large data sets are now available, and can be used in the training process, at least if you have a *lot* of computing power.

The result over the last decade has been a flood of work on deep learning, accompanied, in the popular press, by stories about thinking machines, fears about technological unemployment, and speculations about how general artificial intelligence is right around the corner[13]. Some of this results in extremely impressive and widely-applied technologies, some of it amounts to re-discovering the wheel (as shown by, e.g., Dacrema, Cremonesi, and Jannach (2019)), and some of it poses genuine unsolved scientific problems.

Three Interesting Aspects of Deep Learning

This section is long, and optional (in 2022), but hopefully interesting.

The recent revival of multilayer neural networks has a lot of specific engineering accomplishments to point to, as well as a lot of more ambiguous feats of engineering. But there are also, I think, three specific *scientifically* interesting phenomena it has revealed[14]:

1. **Adversarial examples**, where tiny, humanly-imperceptible perturbations to the inputs result in drastic changes to the output;
2. **Interpolation**, where the models generalize to new data (from the same distribution) despite *perfectly* fitting the training data, even in cases where the model capacity is high enough to memorize random labels;
3. **Alienness** and **cheating**, where the models achieve high performance at matching human classification despite evidently using features which are very different from the ones human beings attend to, and often getting to high performance by using features that seem to us like accidents or tricks.

It is not clear how, or whether, these phenomena are related. It's not even clear how much they are specific to deep neural networks, as opposed to (say) highly-parameterized systems trained by gradient descent, or (say) any sort of high-dimensional classifier with good accuracy over many classes.

Adversarial Examples

were first noticed, and named, by Szegedy et al. (2013); the phenomenon is simply illustrated with one of their examples. Here are some images, from the standard "ImageNet" dataset (Deng et al. 2009), which were

[12]Mathematically, the convolution of two functions f and g on the same domain is another function $f * g$, where $(f * g)(r) = \int f(u)g(r-u)du$. On a discrete domain, we'd have instead $(f * g)(r) = \sum_u f(u)g(r-u)$. In a convolutional neural network, we have inputs $x(r)$, where the discrete coordinates r represent palces in a sequence or pixels on a grid, etc. The **kernel** (sorry, but that's what it's called) function $g(h)$ is chosen so that it goes to zero when $\|h\|$ is too big. So we started with $x(r)$, and get $t(r) = \sum_u x(r)g(r-u)$, which is a kind of local average or local weighted sum of the original input features. The kernel function might have some weights or other parameters buried inside it, but if so we apply the *same* parameters everywhere along the original data. (We typically hold the "support" of the kernel, the range of values h where $g(h) \neq 0$, fixed during training.) This means we do the *same* transformation_ to the original features everywhere. Convolutional transformations were known already to be a very good way of discriminating textures in images and detecting edges. The successes of deep-learning image-classifiers came from using multiple layers of convolution.

[13]So far as I am aware, however, the previous episodes of excitement did not include cults devoted to placating the future AI god. Whether this counts as progress or not depends, I guess, on how entertaining you find such expressions of the myth-making impulse, and the irony of calling it "rationalism".

[14]"Adversarial examples" and "interpolation" are pretty standard names in the literature; "alienness" and "cheating" are not, but there isn't a good single phrase for that phenomenon.

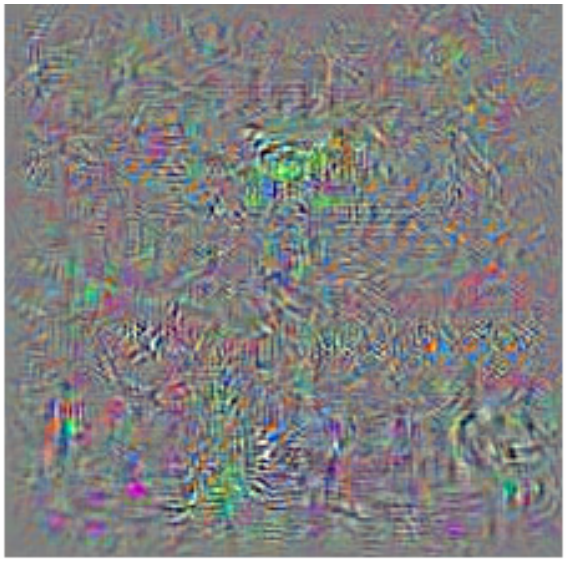
all correctly classified, with very high confidence, by a then-leading system:

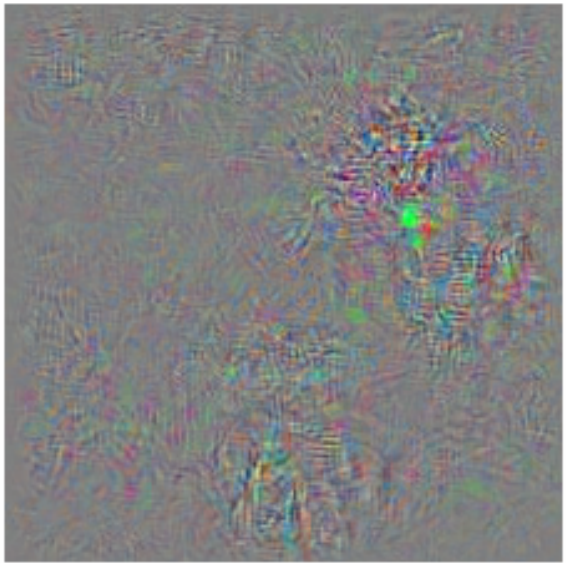

SCHOOL BUS

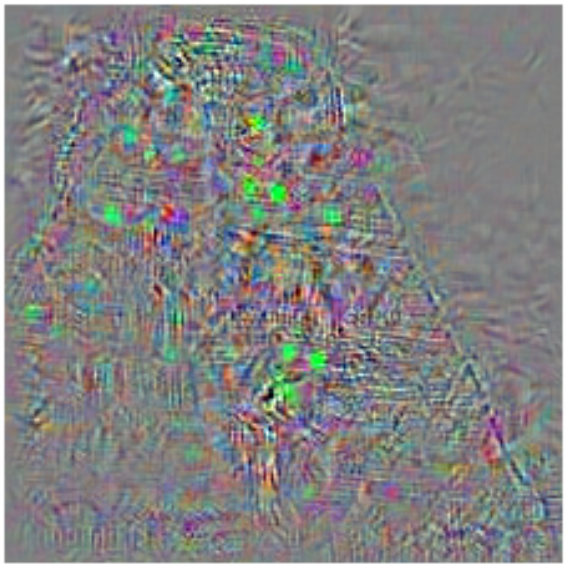
Here are (believe it or not) *different* images, which were all classified as (believe it or not) ostriches:


SCHOOL BUS

Here is the *difference* between the two images, exaggerated by a factor of 10:







This isn't just a bizarre glitch where this one system happens to do badly on this one image. It turns out that we can usually take those correctly-classified images, and by adding a humanly-imperceptible perturbation, get something which is, with high probability, classified as belonging to basically any class we want.

Now you might suppose that what's going on here is that image are very high-dimensional, but n for the ImageNet dataset is *merely* 3.2 million, so the classifier learns to do well on a little "island" around each image but has no idea what to do when we step off one, and so it says something basically random. But if that were true then these classifiers should be vulnerable to random noise, too. Yet adding a *larger* amount of IID Gaussian noise *doesn't* send things haywire in the same way (Szegedy et al. 2013, 6–7).

In computer science, people often talk about "the Adversary" deliberately crafting inputs or examples to get bad performance out of a method or algorithm. It's a vivid way of reasoning about how badly things can go wrong, and ultimately of obtaining worst-case performance guarantees for algorithms[15]. Szegedy et al. (2013) called their carefully-tweaked images "adversarial examples" because they're what an Adversary might do to fool the classifier — find the smallest perturbation which achieves a desired wrong label with at least a specified confidence. The name has stuck.

One reason the name has stuck is that adversarial examples turn out to be ubiquitous in deep neural networks. You can usually eliminate any particular one you find, but there are (seemingly) always others. And it's not just image classifiers: adversarial examples are also known for text classification (Gao, n.d.), audio-to-text and other sorts of audio signal processing (Taori et al. 2019), reinforcement learning algorithms that play games or control robots (Gleave et al. 2020), et cetera. This ubiquity has, potentially, implications for deploying these systems in the real world. The fact that one can *print* adversarially-perturbed images and have the print-outs fool image classifiers (Brown et al. 2017) at the very least doesn't *sound* good for, say, self-driving cars. But here we should focus on the scientific issues that these examples raise. (If nothing else, our understanding of the scientific issues will shae our assessment of the practical implications.)

The thing to focus on, I think, is that ubiquitous, hard-to-eliminate presence of these examples. This suggests that it's worth looking for some very general, equally-ubiquitous aspect or property of very large neural networks, operating on high-dimensional inputs, trained by gradient descent, which explains why adversarial examples exist. (It *could* be that there's some very specific explanation for adversarial examples in every different network, and those specifics can't be usefully abstracted and generalized, but that'd be a *lot* of coincidences.) It might even be that the relevant explanation isn't about *neural networks* specifically, but (say) any sort of high-dimensional classifier trained by gradient descent, and we just happened to notice the phenomenon first in deep learning.

Nobody has an explanation for adversarial examples which accounts for all the known facts and commands general consent among knowledgeable experts. There are intriguing suggestions about the role of computational constraints (Bubeck, Price, and Razenshteyn 2019), about un-intuitive properties of the geometry of high-dimensional spaces (Shamir et al. 2019), and, relatedly, about a property of high-dimensional distributions called "concentration of measure"[16]. But the truth is we just do not know why this happens.

[15]The rhetoric might owe something to the lingering influence of Wiener (1964), and, behind that, the cunning and malicious demon of Descartes (1637).

[16]The last of these is worth a footnote. To illustrate "concentration", consider a radius-1 sphere in p dimensions. What fraction of its volume is within a distance ϵ of its surface? Well, the fraction of its volume which *isn't* is the fraction of its volume inside the smaller sphere of radius $1-\epsilon$; that fraction has to be $(1-\epsilon)^p$. So the fraction of volume within ϵ of the surface is $1-(1-\epsilon)^p$. But this $\to 1$ exponentially fast as $p \to \infty$, no matter how small we make ϵ. Even though the *exact* surface of the sphere has volume 0, in high dimensions most of the volume is in fact "concentrated" exponentially close to the surface. This is true not just for sphere but also for cubes and for pretty much any geometric shape you can easily describe. Probabilistically, if we have a distribution that's independent over p variables, then unless the distribution is *really* weird, there's a set of very small volume (like the surface of a sphere) which the probability concentrates on (Boucheron, Lugosi, and Massart 2013). More exactly, a distribution is "concentrated" when, if a set has *any* positive probability, it can be expanded by a very small distance to get a set with at least 1/2 of the total probability. This phenomenon of "concentration" is particularly clean for distributions of many independent random variables, but it also holds if the dependence between variables is not too strong (Kontorovich and Raginsky 2017). The suggested relevance of this to adversarial examples is, roughly, that the set of images classified as (say) "trucks" needs to have positive probability, as does the set classified as (say) "pandas". So there would need to be a very small expansion of the classified-as-cars set with probability $> 1/2$, and a very small expansion of the classified-as-pandas set with probability $> 1/2$, which means that the two expansions need to overlap, and so two small steps take us from trucks to pandas. If the argument in this form seems sketchy, you're not wrong, but see (Mahloujifar et al. 2019).

Generative Adversarial Networks

Back in the 1980s and 1990s there was a lot of interest in ideas like "actor-critic" learning architectures, co-evolving classifiers and sets of examples, etc.[17] With the rise of statistical learning, much of this work faded into obscurity. In particular, if your goal is "low risk", and you think of "risk" as "expected loss under the *same* distribution", it's hard to know what to make of the idea of evolving training problems to be hard for your classifier.

The discovery of adversarial examples has nonetheless led to a recent re-invention of ideas like this, though usually without any recognition of the precedents. These are **generative adversarial networks** (GANs), named by Goodfellow et al. (2014). The idea is to train up *two* networks, a generator and a discriminator, say G and D. The job of the discriminator D is easily described: it gets fed cases which are either from the real data or are simulation output from the generator, and it needs to say which is which; one typically uses a log loss, and balances the number of real data cases against the number of generated ones. The generator needs to come up with stuff that fools the classifier, and gets rewarded conversely.

We can think of this as a game with two players, and ask about equilibria, where neither player can improve their pay-off by changing their strategy on their own. (The learning process won't *necessarily* converge on an equilibrium, but equilibria will *generally* be fixed points of the learning process.) One equilibrium, emphasized by Goodfellow et al. (2014), is when the distribution produced by G is the same as the distribution of the data, and the discriminator assigns probability $(1/2, 1/2)$ to every example. In that case, the GAN will have learned to mimic the distribution perfectly. But there's little reason to think that GAN training will *typically* approach this.

Interpolation, and Generalizing Despite Memorizing

Throughout the course, I have emphasized that merely memorizing the training data is generally a very bad idea. Partly this is because the training examples are of no *intrinsic* interest, being a product of accidents of sampling and sheer irreproducible noise in the data-generating process. But still more it's because it doesn't, usually, help us to generalize to new data.

Now there was already an important caveat to those warnings, which I know occurred to some of you: nearest neighbors, kNN with $k = 1$, does very little *other than* memorize the training data. But we saw in Lecture 11 that, under some reasonable-sounding assumptions about continuity, the asymptotic risk of nearest neighbors is (at most) twice the risk of the *optimal* decision rule. That's asymptotically as $n \to \infty$, but since it *is* true in the limit, it must also be true that at finite n, nearest neighbors has *some* ability to generalize.

So 1NN gives us an example of a learning procedure which (i) always memorizes, but (ii) can generalize *if* the data-source has the right properties[18], so (iii) in-sample performance and true risk can be incredibly different.

I bring this up in the context of neural networks because it turns out that very large, "over-parameterized" neural networks turn out to show the same combination of (i), (ii) and (iii). In a series of remarkably simple yet convincing experiments, Zhang et al. (2021) took networks that were known to do well on standard image-classification data sets, and *randomly* re-assigned the class labels of the images, so that there was, in fact, no connection between the image and the label. The networks succeeded in memorizing the new labels, but of course their out-of-sample performance could be no better than chance. (This is what 1NN would do in this situation.) But, again, with the correct labels, these same networks trained to memorization *do* generalize. So these architectures have the *capacity* to memorize noise, but still generalize. In fact, it is fairly common, in applications, to train large neural networks to the point of memorization and beyond, and till get good performance under cross-validation or on testing sets.

[17] I won't give a huge collection of references here, but will just mention Rosin and Belew (1997) as an example of a large literature I happen to have handy a quarter-century later.

[18] If the data-generating process is such that the optimal rule always does the *opposite* of what nearby points do (like a checkerboard...), then nearest neighbors will work poorly. See Thornton (2000) for an elaboration of this point, and some thought-provoking implications about the limitations of similarity-based learning.

Exactly what is going on is not clear. In addition to 1NN, something analogous to this was already known to happen with boosting (which we looked at in Lecture 13), where the most plausible (though not certain) explanation was that boosting increases the margin, and high-margin classification implies generalization (Schapire et al. 1998). Spurred by trying to understand these puzzles, it has *also* turned out that something similar can happen with high-dimensional linear models.

To appreciate this last point, imagine we're doing linear regression of Y on $\vec{X}$, but $\dim \vec{X} = p > n$. The OLS estimate of the regression coefficients requires inverting $\mathbf{x}^T\mathbf{x}$, but this inverse doesn't exit when $p > n$. So let's back up a step in OLS's chain of logic. Minimizing the MSE gives us an equation for $\widehat{\beta}$,

$$\mathbf{x}^T\mathbf{x}\widehat{\beta} = \mathbf{x}^T\mathbf{y}$$

and the impulse is of course to get $\widehat{\beta}$ by itself by multiplying both sides with $(\mathbf{x}^T\mathbf{x})^{-1}$ from the left. Again, we can't do that when the inverse doesn't exist, but that doesn't mean that the equation for $\widehat{\beta}$ doesn't have solutions. Rather, what happens is that there are *infinitely many* solutions[19].

When there are infinitely many equally good potential solutions, we're free to pick one however we like, and many people have thought it sounded sensible to pick the solution with the smallest norm, the imaginatively-named **min-norm solution**:

$$\tilde{\beta} = \underset{b:\mathbf{x}^T\mathbf{x}b=\mathbf{x}^T\mathbf{y}}{\operatorname{argmin}} \|b\|$$

That is, we take the shortest vector which solves the equation. If there's a unique $\hat{\beta}$, then of course $\tilde{\beta} = \hat{\beta}$, but in the high-dimensional case this still gives us a well-behaved prescription for finding *a* coefficient vector (see exercises).

For many, many distributions, if increase the dimension p of $\vec{X}$ but use the min-norm $\tilde{\beta}$, we get a generalization error that *decreases* in p once $p > n$. It won't always, but it *can*.

Now, in regression, it is *always* true that

$$\begin{aligned}
\mathbb{E}\left[(Y - \hat{\mu}(X))^2 | X = x\right] &= \mathrm{Var}\left[Y|X=x\right] + \left(\mathbb{E}\left[(\mu(x) - \hat{\mu}(x)\right]\right)^2 + \mathrm{Var}\left[\hat{\mu}(x)\right] && (20)\\
&= (\text{system noise}) + (\text{squared bias of estimate}) + (\text{variance of estimate}) && (21)
\end{aligned}$$

In these over-parameterized regimes, we've driven bias towards zero. But, under these distributions of data, the variance of the estimate is also not exploding. What's happening with high-dimensional linear models, for instance, is that selecting the *minimum* norm solution is, implicitly, imposing stronger and stronger constraints on that norm as p grows. (It's very close to doing ridge regression and increasing the strength of the penalty as p grows.)

Many people *suspect* that something like this is at work with deep neural networks. That is, some combination of their architecture and their training process meshes with the kind of tasks we give them, so that some implicit regularization happening on *those* problems, even though they *could* memorize any old random noise. But *exactly* what's going on here, or how we could characterize it to know when neural networks will be reliable, is still unknown (Belkin 2021).

Alien Cheaters, or, What the Frog's Eye Tells the Frog's Brain, or, What Is It Like to Be a Convolutional Neural Network?

Here is an image that a leading-in-2015 neural network confidently classifies as a cheetah:

[19] Similarly, even if $p < n$, if two (or more) columns of $\mathbf{x}$ are linearly dependent, i.e., there is collinearity among the regressors, the difficulty isn't that there are no solutions to $\widehat{\beta}$. The difficulty is that we can "trade off" coefficients among the linearly-related regressors, without changing any of the predictions, so there are actually infinitely many equally good coefficient vectors.

Here is another image which the same neural network is confident is also a cheetah (Nguyen, Yosinski, and Clune 2015):

If you look at the second image and, without prompting, also see a cheetah, you are a very unusual primate and your brain should be promptly (but humanely!) studied by neuropsychologists for the good of science. For most of us, the second image looks nothing like a cheetah.

Now, the network in question, like almost all the others, was trained to imitate human judgments and evaluations on standard image repositories[20]. Those systems manage to do this successfully. But when we step outside those data sets, it becomes clear that the way the systems are processing images must be very different from what we East African Plains Apes do. The scientific questions are thus:

- *what* features do these systems use,
- *why* do they use *those* features, and
- *is* using those features a good idea?

[20]The photograph data sets were mostly collected by scraping the Web. Recht et al. (2019) went through the interesting exercise of trying to replicate the best-known of these, ImageNet, by repeating the published data-collection protocol. (Even that was hard.) They then examined the performance of various published neural networks trained on the original ImageNet on their replication data. Unsurprisingly, all of them did worse on the replication did. Very surprisingly (at least to me), the increase in average loss was about the same for all systems!

To put the last point a little science-fictionally[21], have we (unintentionally) invented devices which process the world in an alien but equally effective way, or have we made something superficially impressive but ultimately stupid?

To dive into this, let's start by examining how the second image above was created. Nguyen, Yosinski, and Clune (2015) began with a random image, and then optimizing the predicted probability of belonging to the desired class. (Specifically, they used an "evolutionary" optimization algorithm, keeping multiple candidates around, making random "mutations" to them, and copying the ones which did the best job of fooling the image-classification network.) What this tells us is that there is a region (or set of regions) in image space where the classification network assigns very high confidence (say, $> 99.6\%$) to the image being a cheetah (or whatever). This region includes actual pictures of cheetahs, but also certain images of video snow, weird blobs, etc., etc. If we were to ask human beings to classify images the same way, the region where a human subject would say "yes, that's definitely a cheetah" would have some overlap with the machine, around the actual pictures, but *not* in the wilder regions of the image space. On the other hand, normal human beings from my culture immediately recognize these as depictions of cheetahs, though it's not at all clear a neural network trained on a collection of *photographs* would do so:

This suggests two plausible conclusions:

1. Human beings and current neural networks classify based on very different sets of visual features (or at least using very different weightings on whatever features might be common to both);
2. Human beings and neural networks both give high-confidence classifications to many images which objectively, on a pixel-by-pixel level, are *very* different from prototypical photographs of members of the class.

Point (2) is particularly suggestive: maybe we're looking at a general property of classifiers in high-dimensional spaces, or something like that. But this is, currently, even less well understood than the existence of adversarial examples. In this context, though, adversarial examples are another indication that whatever features neural networks use are very different from the features humans are sensitive to, since objectively-small and humanly-indistinguishable changes clearly move images across neural network classification boundaries.

The mere fact that neural networks use very different visual features than we do isn't necessarily bad. At least, it's bad if we want to use these neural networks as models of human visual perception, but not necessarily bad as *engineering*. (A car is a very bad model of a horse, but it doesn't run any worse on that account, and making cars move more like horses wouldn't improve the driving experience.) But there are very worrying indications that, at least as currently designed and trained, neural networks end up using visual features which are, in various ways, bad for their intended applications.

Here[22] is an image classified[23] by a production system as "sheep":

There are, indeed, no sheep here. But in training sets, there are also few (no?) pictures of green mossy mountain scenes *without* sheep (or at least other animals) for visual interest. Whatever features the neural network uses links this image to others in the training set, and it spits out the appropriate classification for them. It's learned to associate the label "sheep" not with the animals, or even (to be cautious) photographic traces of animals, but with certain visual features which were, in the training set, statistically-reliable *signs* of the animals[24].

There is good reason to believe that in this case the signs are things like the text of the meadows. I mentioned earlier that modern image classifiers rely heavily on "convolutional" network layers, which apply the same

[21] John Campbell (1910–1971), an influential early editor of science fiction, was supposed to tell his authors "Write me a story about a creature that thinks *as well* as a man, but not *like* a man". (You will not be the first to suggest simply writing a story about a woman.)

[22] I learned of this example from Janelle Shane, at [https://www.aiweirdness.com/do-neural-nets-dream-of-electric-18-03-02/], which provides more informative mis-classifications; I strongly recommend Shane's site if the material in these notes is at all interesting.

[23] Actually, as you can read in the image, this system doesn't just classify, it provides automatically-generated captions and tags. This is *basically* an elaboration on classification. (It's a little more complicated; the neural network has learned to associate bits of text with images by mapping them to a common latent space.)

[24] The connection between signs and probability is very old, and well-explored in Hacking (1975).

Figure 1: http://publicdomainvectors.org/photos/cheetah-cartoon-publicdomain.jpg

Figure 2: https://publicdomainvectors.org/photos/CHEETAH.png

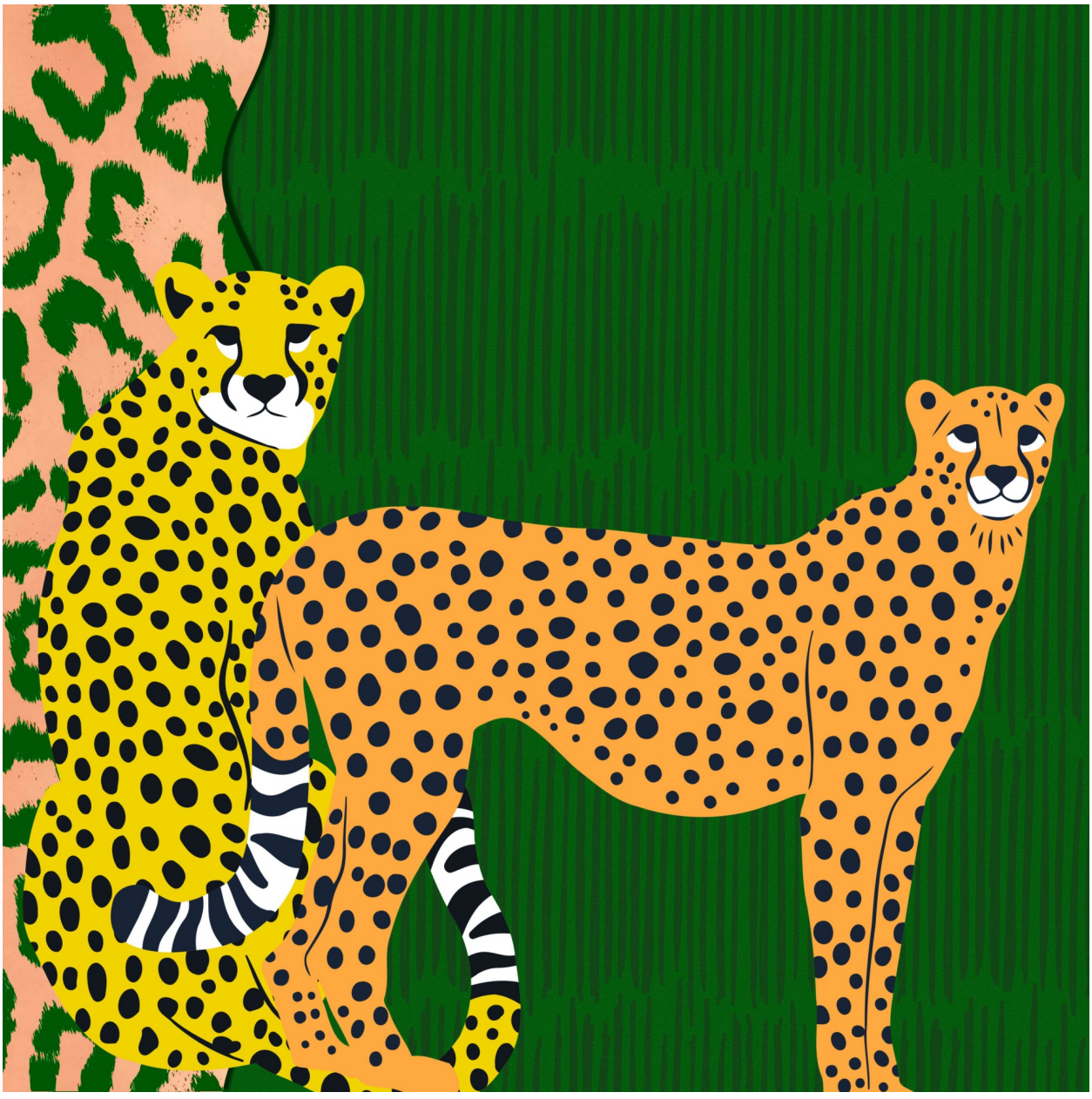
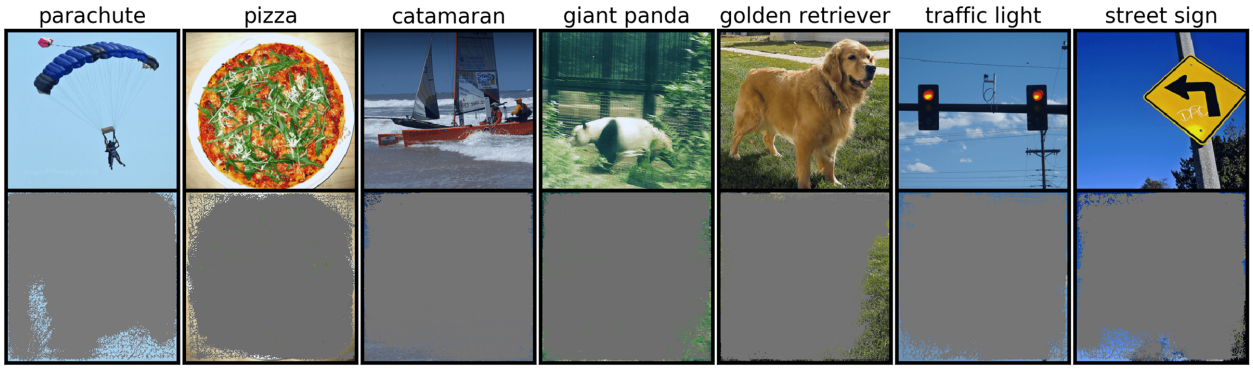


Figure 3: https://www.publicdomainpictures.net/pictures/450000/velka/image-16505217656FP.jpg

Figure 4: https://www.aiweirdness.com/content/images/public/images/e3c62683-f530-49cf-aed2-c2868855f656_2000x1364.jpg

transformation locally over the whole of the image. Convolutional filters are very good at detecting rather local features of images like texture, but are intrinsically ill-suited to notice more global properties. Take an image, chop it into little squares, shuffle the squares around, and we get something human beings see as very weird (and perhaps turn into a jig-saw puzzle); a convolutional neural network will however "see" very little difference[25]. If it's learned to associate one label with "lots of fuzzy green", well, there's still a lot of fuzzy green...

If this was just one goof, well, who cares, that's only a mildly-amusing anecdote. More serious is Carter et al. (2021), which explored the issue of what features image classifiers really use by seeing how much of an image could be "masked" without altering classification performance. Results on leading systems looked like this:



That is, the images are being classified correctly (with at least 90% confidence) using only thin strips at the borders. Again, this is enough to get at predominant textures, which seem to be most of what this (convolution-reliant) system uses. But this should be very worrisome when we consider any sort of real-world use. Even if we're not worried about someone deliberately trying to fool the system[26], the sheer, inhuman strangeness of the system ought tio make us doubt that we can anticipate how it will work in the wild.

Now in fact there is *very* good evidence that neural networks frequently seize upon features which are predictive in their training context, but *only* in their training context, in "high-stakes" situations. A compelling (even disturbing) one comes from Zech et al. (2018), which re-examined the ability of neural networks to diagnose diseases from chest x-rays. It turned out that x-rays from different diseases had been (predominantly) collected using different types of machines (portable vs. fixed) and at different hospitals; that the machine and hospital information was printed on the x-ray image; and that the neural network had learned, in training, to give a lot of weight to *that* part of the image. So the neural network learned to be accurate, but in a useless way.

Or, more exactly, in a way which will continue to be accurate only if the environment in which the neural network is used continues to work the same way as the training environment. If something (other than the neural network!) continues shunting patients with different diseases to different hospitals, and hospital information keeps being added to x-ray images in the same way as before, the neural network will keep working just fine. The doctors asked for a system to do diagnosis (a real-world problem). The data analysts translated that into a tractable statistical problem: come up with a learning process which will find a decision rule with low risk in *that* environment. They solved that problem, they ran through the learning process, and they got a rule which delivered *exactly* on that promise. The snag is that "diagnose reliably" is *not* the same as "classify accurately in a given environment". If we are unhappy that the rule doesn't also have low risk in other, different environments that the learning process was never exposed to, we have no one to blame but ourselves.

It should, perhaps, not be surprising that we see behavior like this from neural networks. One of the classic early papers about biological neural computation was titled "What the frog's eye tells the frog's brain" (Lettvin et al. 1959). By very careful experimentation, they showd that the frog's retinas contained

[25]There are many papers documenting the importance of texture for convolutional neural networks, but I will just mention Geirhos et al. (2019) and Brendel and Bethge (2019) (the latter did a version of the chop-into-chunks-and-shuffle experiment).

[26]But why wouldn't we be? If the system is worth building for a real-world application, it'll also be worth somebody's time to try to subvert. "What gets measured gets manipulated" and "Some people cheat" are two very sound design principles...

neural networks which detected the presence, location and velocity of small dark rapidly-moving blobs, and transmitted that information along the optic nerve to the frog's brain. In the frog's ecological context, that was information about flies and other insects the frog could catch and eat with its tongue. In the lab, those small dark jittery blobs could be all kinds of things, and in environment with (say) many other little dark flying bits (gravel? metal shavings?) this part of the percentual system mightn't work very well for a hungry frog, or at least might need to be supplemented with other information.

We have developed neural networks which (at their best) work like *that*. They are very little like intelligence, as we experience conscious rational thought, or even the forms of unconscious thought underlying (say) grammar. Rather, they are much closer to unconscious, automatic perception. Experience and evolution has tuned our perceptions to work well in a certain range of environments, it is (as the psychologist Gerd Gigerenzer puts it) "ecological rational". But *outside* those environments, our automatic perceptions can go horribly awry.

Some would argue that a great deal of human intelligence is also only "ecologically rational". (Gigerenzer, for one, makes a strong case for this (Gigerenzer 2000).) We look at the mistakes of a neural network and laugh, or swear not to use something so stupid. But our own stubborn visual illusions seem to us mere quirks which in no way detract from our dignity as rational creatures. Perhaps we're right about this; perhaps this is mere chauvinism, and a truly alien intelligence, "vast, cool, and unsympathetic" (as the poet says), would draw no distinction between us and our artifacts[27].

There is however one straightforward way in which it is bad for neural networks to use weird-to-us features. We want to deploy those neural networks in *our* world, in a physical and social environment we have grown up in and have built to work with *our* perceptions, modes of thought and bodies. If our machines do not work well in *that* range of environments, they are badly engineered *for us*, and we should designed systems which work better *for us*. The alternative is to redesign our world for the convenience of our machines. That seems both stupid, and inconsistent with "the human use of human beings" (Wiener 1954) — which doesn't mean it won't be tried.

Before closing this section, I want to mention two things. First, it is very plausible that there is some connection between this issue, that of using strange and apparently fragile features, and the other issues I highlighted, those of adversarial examples and interpolative generalization. Plausible, but by no means established! Second, it is, again, quite unclear how far this issue is peculiar to deep neural networks. If anyone has tried to comparable studies on, say, support vector machines or random forests, I haven't seen it. (I haven't tried to do it myself!) But it might be that we're looking at phenomena which are fairly generic to large-scale learning systems, rather than specific to neural networks.

Further reading

Goodfellow, Bengio, and Courville (2016) is, deservedly, the standard text on deep learning, though algorithmic details advance rapidly, and are hard even for specialists to keep up with.

From the last burst of interest in neural networks, Churchland and Sejnowski (1992) is still good, and recently brought back into print. Ripley (1996) is full of good ideas about statistical applications, and has the unusual distinction of being written by a statistician deeply involved in the development of R.

On distributed representations, Abbott and Sejnowski (1998) is a now-classic volume of important, and generally readable, papers. Churchland and Sejnowski (1992) is also a good reference on this topic. That the meaning of the activity of neural networks might only reside in the global configuration, and not in individual cells, was more or less explicit already in the work of some of the pioneers, such as Hayek (1952) and Hebb (1949) (and, at a different level of biological organization, Luria (1973)).

[27] One might even imagine those vast, cool, unsympathetic intellects being, in some distant future, remote descendants of our current neural networks, carrying pattern-recognition modules optimized for ImageNet around as traces of an evolutionary past as distant for them as our own blood's mimicy of the chemistry of prehistoric seas. But I imprinted on Arthur C. Clarke as a boy, and am launching a flight of fancy, and we should return to our actual machines.

As discussed above, our understanding of how models with the capacity to memorize noise can also generalize to new data is still very incomplete. I recommend the review/discussion paper by Belkin (2021), written by one of the leaders in the field and with excellent references to earlier work. (If this intrigues you, you might also try your hand at the exercises.)

Exercises

I recognize that it's a bit perverse that all the exercises in notes on neural networks relate to over-parameterized linear models.

1. *Min-norm linear regression, part 1.* Consider the ordinary-least squares problem $\mathbf{x}^T\mathbf{x}\vec{b} = \mathbf{x}^T\curvearrowright$, where $p > n$. Let's say that any vector $\vec{b}$ which satisfies this equation is an "OLS solution". Recall (from linear algebra) that the collection of vectors $\vec{v}$ where $\mathbf{a}\vec{v} = 0$ is called the **null space** of the square matrix $\mathbf{a}$.
   a. Show that if $\vec{v}_1$ and $\vec{v}_2$ are both in the null space of $\mathbf{a}$, then $c_1\vec{v}_1 + c_2\vec{v}_2$ is also in the null space. (This justifies the "space" part of the name "null space".)
   b. Show that the dimension of the null space of $\mathbf{x}^T\mathbf{x}$ is at least $p - n$. (Some people call this number the "nullity" of the matrix.) Can you given an example of an $\mathbf{x}$ where the dimension of the null space is larger than $p - n$? (We're assuming $p > n$ here so don't try $p = 1$, $n = 1000$ or anything silly like that.)
   c. Show that if $\vec{b}$ is an OLS solution, and $\vec{v}$ is in the null space of $\mathbf{x}^T\mathbf{x}$, then $\vec{b} + \vec{v}$ is also an OLS solution. Use this to explain why the space of OLS solutions has dimension $p - n$ (at least).
   d. Show that the vector at the origin of p-dimensional space, $\vec{0}$, has a unique projection on to the space of OLS solutions. Explain why the norm of the projection of $\vec{0}$ is the minimum value of the norm in the whole space of OLS solutions. *Hint*: for any vector $\vec{v}$, $\|\vec{v}\| = \text{distance}(\vec{v}, \vec{0}) = \text{distance}(\vec{0}, \vec{v})$. (Why?)
2. *Min-norm linear regression, part 2.* Suppose that $p > n$, and we look for the minimum norm solution:

$$\tilde{\beta} = \underset{b:\mathbf{x}^T\mathbf{x}b=\mathbf{x}^T\mathbf{y}}{\text{argmin}} \|b\|$$

   a. Explain why

$$\tilde{\beta} = \underset{b:\mathbf{x}b=\mathbf{y}}{\text{argmin}} \|b\|$$

   b. (Harder) Explain why

$$\tilde{\beta} = \mathbf{x}^T(\mathbf{x}\mathbf{x}^T)^{-1}\mathbf{y}$$

   The quantity $\mathbf{x}^T(\mathbf{x}\mathbf{x}^T)^{-1}$ is often written as $\mathbf{x}^\dagger$, and called[28] the **Moore-Penrose pseudo-inverse**, or **Moore-Penrose generalized inverse**, of $\mathbf{x}$.
3. *Interpolation in the linear model and risk.* Consider the situation where $Y = Z + \epsilon$, $Z \sim \text{Unif}(-10, 10)$ and $\epsilon \sim \mathcal{N}(0, 1)$. However, we do not get to see Z, but instead get to observe $\vec{X}$ where $\vec{X}_j = \sin(\omega_j Z + \phi_j)$. The frequencies ω_j are also $\mathcal{N}(0, 1)$, and the phases ϕ_j are $\sim \text{Unif}(0, 2\pi)$. The coordinates of $\vec{X}$ are thus random Fourier features, as in a previous lecture. Notice that while there is a simple relationship between Y and Z, there is no *exact* linear relationship between Y and $\vec{X}$, though one can come increasingly close by using more and more random features, say d.
   a. Write a function which will generate n IID (Z, Y) pairs from the specification above. It should return a two-column data frame.
   b. Write a function which takes in a two-column, n-row data (Z, Y) data frame, as from (a), and a two-column, d-row array (or data frame) of ω_j and ϕ_j, and returns the $(d+1)$-column, n-row data frame of $(\vec{X}, Y)$ pairs.

[28] More pedantically, the Moore-Penrose pseudo-inverse of a $n \times p$ matrix $\mathbf{a}$ is the (unique) $p \times n$ matrix $\mathbf{a}^\dagger$ such that $\mathbf{a}\mathbf{a}^\dagger\mathbf{a} = \mathbf{a}$ and $\mathbf{a}^\dagger\mathbf{a}\mathbf{a}^\dagger = \mathbf{a}^\dagger$ (so $\mathbf{a}$ and $\mathbf{a}^\dagger$ "cancel out" in matrix products), and where $\mathbf{a}\mathbf{a}^\dagger$ and $\mathbf{a}^\dagger\mathbf{a}$ are both symmetric. If $p = n$ and $\mathbf{a}$ is invertible, then $\mathbf{a}^\dagger = \mathbf{a}^{-1}$. If $n < p$ but $\mathbf{a}$ has full rank (=linearly-independent rows), then $\mathbf{a}^\dagger = \mathbf{a}^T(\mathbf{a}\mathbf{a}^T)^{-1}$. If $n > p$ but $\mathbf{a}$ has full rank (=linearly-independent columns), then $\mathbf{a}^\dagger = (\mathbf{a}^T\mathbf{a})^{-1}\mathbf{a}^T$, which should look familiar.

c. Write a function which takes a data frame as returned by (b), and a vector of d coefficients, and returns the MSE of using those coefficients, applied to the $\vec{X}$ columns, to predict the $\vec{Y}$ column. (You can assume the intercept term is zero.)
d. Write code which, given a data frame as returned by (c), *either* finds the OLS coefficients, if $n > d$, *or* finds the min-norm solution. In doing the latter, you may find the `ginv` function from the `MASS` package helpful.
e. Generate a training set with $n = 100$ and $d = 1000$. (Make sure to save the ω, ϕ values for the features in their own array.) Calculate the in-sample MSE of linear models fitted to the first p features, for $p \in 1 : d$.
f. Generate an evaluation set with $n = 100000$ and the same $d = 1000$ random Fourier features. For each $p \in 1 : d$, calculating the MSE of linear models fitted to the first p features on the training set on this evaluation set.
g. Plot in-sample and generalization MSE as a function of p. What happens?
h. Plot the (L_2) norm of the estimated coefficients vectors as a function of p. What happens?

References

Abbott, Laurence F., and Terrence J. Sejnowski, eds. 1998. *Neural Codes and Distributed Representations: Foundations of Neural Computation*. Cambridge, Massachusetts: MIT Press.

Anthony, Martin, and Peter L. Bartlett. 1999. *Neural Network Learning: Theoretical Foundations*. Cambridge, England: Cambridge University Press.

Ashby, W. Ross. 1960. *Design for a Brain: The Origins of Adaptive Behavior*. 2nd ed. London: Chapman; Hall.

Baroni, Marco, Georgiana Dinu, and German Kruszewski. 2014. "Don't Count, Predict! A Systematic Comparison of Context-Counting Vs. Context-Predicting Semantic Vectors." In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics [Acl14]*, 238–47. Baltimore, Maryland: Association for Computational Linguistics. http://www.aclweb.org/anthology/P/P14/P14-1023.

Belkin, Mikhail. 2021. "Fit Without Fear: Remarkable Mathematical Phenomena of Deep Learning Through the Prism of Interpolation." *Acta Numerica* 30:203–48. https://doi.org/https://doi.org/10.1017/S0962492921000039.

Boucheron, Stéphane, Gábor Lugosi, and Pascal Massart. 2013. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford: Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199535255.001.0001.

Brendel, Wieland, and Matthias Bethge. 2019. "Approximating CNNs with Bag-of-Local-Features Models Works Surprisingly Well on ImageNet." In *International Conference on Learning Representations 2019 [Iclr 2019]*. OpenReview.net. https://openreview.net/forum?id=SkfMWhAqYQ.

Brown, Tom B., Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. 2017. "Adversarial Patch." In *Machine Learning and Computer Security Workshop*, edited by Jacob Steinhardt, Bo Li, Chang Liu, Nicolas Papernot, Percy Liang, and Dawn Song. http://arxiv.org/abs/1712.09665.

Bubeck, Sébastien, Eric Price, and Ilya Razenshteyn. 2019. "Adversarial Examples from Computational Constraints." Edited by Kamalika Chaudhuri and Ruslan Salakhutdinov. PMLR. https://proceedings.mlr.press/v97/bubeck19a.html.

Carter, Brandon, Siddhartha Jain, Jonas Mueller, and David Gifford. 2021. "Overinterpretation Reveals Image Classification Model Pathologies." In *Advances in Neural Information Processing Systems 34 [Neurips 2021]*, edited by M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, and J. Wortman Vaughan, 15395–15407. Curran Associates. https://proceedings.neurips.cc/paper/2021/hash/8217bb4e7fa0541e0f5e04fea764ab91-Abstract.html.

Caudill, Maureen, and Charles Butler. 1990. *Naturally Intelligent Systems*. Cambridge, Massachusetts: MIT Press.

Churchland, Patricia S., and Terrence J. Sejnowski. 1992. *The Computational Brain*. Cambridge, Massachusetts: MIT Press.

Dacrema, Maurizio Ferrari, Paolo Cremonesi, and Dietmar Jannach. 2019. "Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches." In *Proceedings of the 13th Acm Conference on Recommender Systems (Recsys 2019)*. https://doi.org/10.1145/3298689.3347058.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. "ImageNet: A Large-Scale Hierarchical Image Database." In *2009 IEEE Conference on Computer Vision and Pattern Recognition [Cvpr]*, 248–55. IEEE. https://doi.org/10.1109/CVPR.2009.5206848.

Descartes, René. 1637. *Discours de La Méthode Pour Bien Conduire Sa Raison, et Chercher La Vérité Dans Les Sciences*. Leiden.

Gao, Hang anf Tim Oates. n.d. "Universal Adversarial Perturbation for Text Classification." E-print, arxiv:1910.04618. https://doi.org/https://doi.org/10.48550/arXiv.1910.04618.

Geirhos, Robert, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. 2019. "ImageNet-Trained CNNs Are Biased Towards Texture; Increasing Shape Bias Improves Accuracy and Robustness." In *International Conference on Learning Representations 2019 [Iclr 2019]*. OpenReview.net. https://openreview.net/forum?id=Bygh9j09KX.

Gigerenzer, Gerd. 2000. *Adaptive Thinking: Rationality in the Real World*. Evolution and Cognition. Oxford: Oxford University Press.

Gleave, Adam, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. 2020. "Adversarial Policies: Attacking Deep Reinforcement Learning." In *Eighth International Conference on Learning Representations [Iclr 2020]*. https://arxiv.org/abs/1905.10615.

Goldberg, Yoav, and Omer Levy. 2014. "`word2vec` Explained: Deriving Mikolov et Al.'s Negative-Sampling Word Embedding Method." Electronic preprint, arxiv:1402.3722. https://arxiv.org/abs/1402.3722.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: MIT Press.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. "Generative Adversarial Nets." In *Advances in Neural Information Processing Systems 27 [Nips 2014]*, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, 2672–80. Red Hook, New York: Curran Associates. https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

Hacking, Ian. 1975. *The Emergence of Probability: A Philosophical Study of Early Ideas About Probability, Induction and Statistical Inference*. Cambridge, England: Cambridge University Press.

Hayek, Friedrich A. 1952. *The Sensory Order: An Inquiry into the Foundations of Theoretical Psychology*. Chicago: University of Chicago Press.

Hebb, D. O. 1949. *The Organization of Behavior: A Neuropsychological Theory*. New York: Wiley.

James, William. 1890. *Principles of Psychology*. Henry Holt.

Kearns, Michael J., and Aaron Roth. 2019. *The Ethical Algorithm: The Science of Socially Aware Algorithm Design*. Oxford: Oxford University Press.

Kontorovich, Aryeh, and Maxim Raginsky. 2017. "Concentration of Measure Without Independence: A Unified Approach via the Martingale Method." In *Convexity and Concentration*, edited by Eric Carlen, Mokshay Madiman, and Elisabeth M. Werner, 161:183–210. IMA Volumes in Mathematics and Its Applications. New York: Springer. https://arxiv.org/abs/1602.00721.

Lettvin, J. Y., H. R. Maturana, W. S. McCulloch, and W. H. Pitts. 1959. "What the Frog's Eye Tells the Frog's Brain." *Proceedings of the IRE* 47:1940–51. https://doi.org/10.1109/JRPROC.1959.287207.

Luria, Aleksandr R. 1973. *The Working Brain: An Introduction to Neuropsychology*. New York: Basic Books.

Mahloujifar, Saeed, Xiao Zhang, Mohammad Mahmoody, and David Evans. 2019. "Empirically Measuring Concentration: Fundamental Limits on Intrinsic Robustness." In *Advances in Neural Information Processing Systems 32 [Neurips 2019]*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 5209–20. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/46f76a4bda9a9579eab38a8f6eabcda1-Abstract.html.

Marcus, Gary F. 2001. *The Algebraic Mind: Integrating Connectionism and Cognitive Science*. Learning, Development, and Conceptual Change. Cambridge, Massachusetts: MIT Press.

McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 5:115–33.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." In. http://arxiv.org/abs/1301.3781.

Minsky, Marvin, and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, Massachusetts: MIT Press.

Nguyen, Anh, Jason Yosinski, and Jeff Clune. 2015. "Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Images." In *2015 IEEE Conference on Computer Vision and Pattern Recognition (Cvpr 2015)*, 427–36. IEEE. https://doi.org/10.1109/CVPR.2015.7298640.

Recht, Benjamin, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. 2019. "Do ImageNet Classifiers Generalize to ImageNet?" In *Proceedings of the 36th International Conference on Machine Learning [Icml 2019]*, edited by Kamalika Chaudhuri and Ruslan Salakhutdinov, 5389–5400. PMLR. http://proceedings.mlr.press/v97/recht19a.html.

Ripley, Brian D. 1996. *Pattern Recognition and Neural Networks*. Cambridge, England: Cambridge University Press.

Rosenblatt, Frank. 1958. "The Perceptron: A Probablistic Model for Information Storage and Organization in the Brain." *Psychological Review* 65:386–408. https://doi.org/10.1037/h0042519.

Rosin, Christopher D., and Richard K. Belew. 1997. "New Methods for Competitive Coevolution." *Evolutionary Computation* 5:1–29. https://doi.org/10.1162/evco.1997.5.1.1.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323:533–36. https://doi.org/10.1038/323533a0.

Schapire, Robert E., Yoav Freund, Peter Bartlett, and Wee Sun Lee. 1998. "Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods." *Annals of Statistics* 26:1651–86. https://doi.org/10.1214/aos/1024691352.

Shamir, Adi, Itay Safran, Eyal Ronen, and Orr Dunkelman. 2019. "A Simple Explanation for the Existence of Adversarial Examples with Small Hamming Distance." E-print, arxiv:1901.1086. http://arxiv.org/abs/1901.1086.

Sherrington, Charles. 1906. *The Integrative Action of the Nervous System*. New Haven, Connecticut: Yale University Press.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. "Intriguing Properties of Neural Networks." In. http://arxiv.org/abs/1312.6199.

Taori, Rohan, Amog Kamsetty, Brenton Chu, and Nikita Vemuri. 2019. "Targeted Adversarial Examples for Black Box Audio Systems." In *2019 IEEE Security and Privacy Workshops (SPW)*, edited by Konrad Rieck, Battista Biggio, and Nikolaos Vasiloglou, 15–20. IEEE. https://doi.org/10.1109/SPW.2019.00016.

Thornton, Chris. 2000. *Truth from Trash: How Learning Makes Sense*. Cambridge, Massachusetts: MIT Press.

Tishby, Naftali, Fernando C. Pereira, and William Bialek. 1999. "The Information Bottleneck Method." In *Proceedings of the 37th Annual Allerton Conference on Communication, Control and Computing*, edited by B. Hajek and R. S. Sreenivas, 368–77. Urbana, Illinois: University of Illinois Press. http://arxiv.org/abs/physics/0004057.

Wiener, Norbert. 1954. *The Human Use of Human Beings: Cybernetics and Society*. 2nd ed. Garden City, New York: Doubleday.

———. 1964. *God and Golem, Inc.: a Commentary on Certain Points Where Cybernetics Impinges Upon Religion*. Cambridge, Massachusetts: MIT Press.

Zech, John R., Marcus A. Badgeley, Manway Liu, Anthony B. Costa, Joseph J. Titano, and Eric K. Oermann. 2018. "Confounding Variables Can Degrade Generalization Performance of Radiological Deep Learning Models." *PLoS Medicine* 15:e1002683. https://doi.org/10.1371/journal.pmed.1002683.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. "Understanding Deep Learning (Still) Requires Rethinking Generalization." *Communications of the ACM* 64:107–15. https://doi.org/10.1145/3446776.

Zhu, Xiaojin, Timothy Rogers, and Bryan Gibson. 2009. "Human Rademacher Complexity." In *Advances in Neural Information Processing Systems 22*, edited by Y. Bengio, D. Schuurmans, John Lafferty, C. K. I. Williams, and A. Culotta, 2322–30. http://papers.nips.cc/paper/3771-human-rademacher-complexity.