MACHINE LEARNING, MATH, AND GENERALIZATION

HIKARI SORENSEN

Contents

Preface

There is an idea in machine learning that I consider perhaps the most important (or at least, for me the most compelling) idea one could work on: that of generalization of models. What's substantial, crucial, inevitable, about this idea is that it's, well, *general*. It's a notion fundamental not only to machine learning, but also to mathematics and indeed more broadly to any field that seeks to take information and compress it in some more concise codification. In machine learning the question of generalization presents itself in the form, "does my learning model make correct predictions on data it has never seen?". In pure mathematics, it might look like, "does this statement/hypothesis/lemma/theorem address all of the instances of the thing I want to use it to talk about?", or even more abstractly (for lack of a better word, despite the clash in usage), "what is the scope of this abstraction?". The natural sciences, statistics, computer science - really any field of theory - have their analogous questions.

My goal is to address in some part the specific matter of generalization of machine learning models, and to present both a conceptual and practical (that is, implementable) framework that I think could be a step toward advancing how we understand and improve the generalizability of neural networks. I first present my way of thinking about neural networks in a (quasi-)mathematical, highly abstract framework (part of whose exposition is a justification for using such formalism, besides the fact that that's just how I happen to think of it). Much of this is to present how I came to the particular more concrete ideas regarding neural network generalization, and to describe what I think could be an important next step in neural network research. I then put the abstract framework in the more concrete context of neural network implementation, and discuss in brief the most recent research addressing similar ideas, and how my proposal could extend current cutting-edge techniques. I conclude with an outline for an experimental research agenda to test the proposed techniques, as well as a discussion of other interesting implications of the ideas I propose.

Part 1. Machine Learning For The Working Mathematician

0.1. **Justifying abstract nonsense.** For all its success in predictive modeling tasks, deep learning as it is today is still widely considered a hodge-podge of ad hoc methods and architectures that work in practice but have relatively little principled, formal motivation. This is unsatisfying not only to theoreticians, for whom lack of formalism is disturbing, but also to the machine learning community as a whole because it makes the success of deep learning difficult to understand; as a result, there is often little grounding for how to move forward. I here attempt to formally characterize the goals of machine learning and the role neural networks play in this general framework. This formalism gives a view of neural networks that addresses the variety of seemingly ad hoc heuristics and training methods and incorporates them into a larger, more structured image. The goal of this theoretical understanding is to lend a broader perspective on what neural networks are in the context of machine learning, with the hope that such insights will inspire new approaches for moving forward.

1. The abstract nonsense

1.1. **Supervised learning.** I here consider the case of supervised learning. We have some set of observations of a system in the form of a dataset $D = \{(x_t, y_t),\ t \in T\}$ that consists of associations between features $x_t \in X$ and corresponding labels $y_t \in Y$, for T a finite set. We would like to understand the nature of the relationship between points $x \in X$ and corresponding $y \in Y$, and in particular if we assume that there is some underlying generative process described by a "true" function $F : X \to Y$, we would like to find, or at least approximate, this F. Now, if we had access to all (in general, infinitely many) pairs (x, y) of the system, we would know F by knowing how it behaves on every $x \in X$. However, the problem machine learning faces is that we do not have access to all pairs (x, y), and in fact have access only to relatively small, finite subsets of the infinite space $\mathcal{D} = X \times F(X)$. Even worse, the data we do have tends to be noisy, so that in general if according the true distribution $x \mapsto F(x)$, our observation is of the form $(x_t, F(x_t) + \epsilon_t)$. Our task then is to produce, using finite and possibly noisy data $D \subset \mathcal{D}$, some function $\tilde{F}$ that approximates the true F.

Because of the uncertainty inherent in any approximation of an infinite space given only finite, possibly noisy samples, in practice the best we can hope to understand is a distribution over candidate functions of their "good"-ness or "reasonableness" (for some notion of "good" or "reasonable") as approximations of F. Training a machine learning model approximates such a distribution $\mathrm{Dist}_F : \mathrm{Fun}(X, Y) \to \mathbb{R}$ by sampling functions f from the space $\mathrm{Fun}(X, Y)$ of all functions $X \to Y$ and assigning a score reflecting its "good"-ness.

To ask what it is for a machine learning model to *learn* is to ask the following:

(1) Given some sampled set of functions $\mathcal{S} \subset \mathrm{Fun}(X, Y)$, what is $\mathrm{Dist}_F(\mathcal{S})$? In particular, how do we choose Dist_F, given $\mathcal{S}$?

(2) How do we sample $\mathcal{S}$ to begin with? How good is that sample?

1.2. **Loss functions.** Let $X_T = \{x_t \mid (x_t, y_t) \in D\}$ and similarly $Y_T = \{y_t \mid (x_t, y_t) \in D\}$. Now given arbitrary $f \in \mathrm{Fun}(X, Y)$, we can evaluate $f(x_t)$ for $x_t \in X_T$. For some choice of metric $d_Y : f(X_t) \times Y_T \to \mathbb{R}$, we can evaluate for every x_t

$$d_Y(f(x_t), y_t) \in \mathbb{R}.$$

This choice of metric, along with a method for aggregating or summing the distances $d_Y(f(x_t), y_t)$ over all of $t \in T$ induces a map

$$\begin{tikzcd}
\mathrm{Fun}(X,Y) \arrow[r, "\text{apply } d_Y"] \arrow[dr, "\mathcal{L}"'] & \mathrm{Fun}(T,\mathbb{R}) \arrow[d, "\sum"] & & f \arrow[r, mapsto] \arrow[dr, mapsto, "\mathcal{L}"'] & d_{Y,f} \arrow[d, mapsto, "\sum"] \\
& \mathbb{R} & \rightsquigarrow & & S(d_{Y,f}) = \mathcal{L}(f)
\end{tikzcd}$$

that sends f to $d_{Y,f} : T \to \mathbb{R}$, $d_{Y,f}(t) = d_Y(f(x_t), y_t)$, then to $\sum(d_{Y,f})$ where $\sum : \mathrm{Fun}(T, \mathbb{R}) \to \mathbb{R}$ is a form of summation or aggregation over the distances $d_Y(f(x_t), y_t)$ for all $t \in T$. What we recover from the resulting composition (diagonal arrow) is a map

$$\mathcal{L} : \mathrm{Fun}(X, Y) \to \mathbb{R},$$

which we know as the familiar loss function involved in training a neural network (or any other model). As a concrete example, we can characterize the L2 or minimum squared error loss as the choice of metric $d_Y(f(x_t), y_t) = f(x_t) - y_t$ (standard Euclidean metric) and choice of aggregation $\sum(d_{Y,f}) = \sum_{t \in T} d_{Y,f}(t)^2$ (sum of squares). Other LP losses, like the L1 or L-∞ losses, have the same metric but with different choices of $\sum$. The cross-entropy loss, on the other hand, chooses for a metric $d_Y(f(x_t), y_t) = -y_t \cdot \log(p(f(x_t)))$, where $\cdot$ is the standard dot product and p maps $f(x_t)$ to a probability distribution (for example, neural network classifiers often choose p to be the softmax function). The aggregation function in the cross-entropy loss is a simple summation over the index set T.

1.3. **Computable functions and neural network architectures.** We are now equipped with a function (a distribution) $\mathcal{L} : \mathrm{Fun}(X, Y) \to \mathbb{R}$ that reflects how well $f \in \mathrm{Fun}(X, Y)$ fits the data D. $\mathcal{L}$, however, is over an infinite space of functions $X \to Y$. In practice, however, we do not have access to all of $\mathrm{Fun}(X, Y)$[1]: we may only choose functions that are representable by computational models, and the architectures of these models impose important structural constraints on the subset of $\mathrm{Fun}(X, Y)$ from which we may sample functions.

[1]in fact, we only have access to a subset of measure zero!

1.3.1. *Neural network-computable functions.* From a mathematical standpoint, a neural network is a family of functions parametrized by the network's weights.[2] That is, given a particular network architecture with $|W|$ trainable weights, each point $\theta \in \mathbb{R}^{|W|}$ specifies a function $f_\theta \in \mathrm{Fun}(X, Y)$ computed by that neural network with the choice of parameters θ. In this way, a neural network is a subset $P \simeq \mathbb{R}^{|W|} \subset \mathbb{R}^\infty$ corresponding to a set of functions $N_P \subset \mathrm{Fun}(X, Y)$ expressible by $\theta \in P$.[3] We thus have the following:

$$\begin{tikzcd}
\theta \in P \simeq \mathbb{R}^{|W|} \arrow[d, "N"'] \arrow[dr, "L"] & \\
f_\theta \in \mathrm{Fun}(X, Y) \arrow[r, "\mathcal{L}"'] & \mathbb{R}
\end{tikzcd}$$

where now $L : P \to \mathbb{R}$ is the loss of parametrizations of a particular neural network.

1.3.2. *Choosing a map from parameters to functions.* Now, what the above diagram makes clear is that given a particular network topology, the computed loss $L(\theta)$ corresponding to a particular parametrization of a network is influenced by two separate factors: the choice of loss function $\mathcal{L}$, but also the particular map $N : P \to \mathrm{Fun}(X, Y)$. That is, even given a fixed network topology, the factoring of the map L into $\mathcal{L} \circ N$ suggests that not only can $\mathcal{L}$ be chosen, but also that the map N from $\mathbb{R}^{|W|}$ to functions can be chosen. Now, having chosen P (corresponding to having a fixed arrangement of nodes in layers), we are no longer free to change the parameter space (i.e., change how the information in each network layer depends on the previous layer). However, there is freedom in how to transform information in the network without reference to P. That is, the values of the network at the nodes can be transformed individually, elementwise. We can do this in an unparametrized fashion by applying some function $\mathbb{R} \to \mathbb{R}$ to the data in every layer. In practice, this is implemented in the form of a choice of activations in a network.

Mathematically, the effect of choosing different activations between each layer is to choose different maps $N : P \to \mathrm{Fun}(X, Y)$. In particular, if $\mathcal{P} = \{P \mid P \text{ parametrizes a neural network}\}$, we have a map

$$\{\text{configurations of activations}\} \times \mathcal{P} \to \mathrm{Fun}(P, \mathrm{Fun}(X, Y)), \qquad (\Theta, P) \mapsto N_\Theta,$$

where Θ is some arrangement of activations within the network parametrized by P.

1.3.3. *Choosing N.* The above discussion regarding the choice of $N : P \to \mathrm{Fun}(X, Y)$ assumed a fixed choice of $P \in \mathcal{P}$. Of course, there is great variability in what P might be, corresponding to the variability of neural network architectures. This variation in the choice of P occurs along a

[2]If we think of weight updates along a gradient as being approximately continuous, it in fact forms (approximately) a smoothly-varying family of functions. In particular, the process of training defines a homotopy between the function f_{init} specified by a randomly initialized network and the function $f_{trained}$ specified by the same network after training.
[3]Here, a particular subset P corresponds to a particular network topology given by an arrangement of nodes in layers, in a way that specifies the connectivity of data throughout the network. Activations are not here considered part of a network topology, since, as elementwise operations, they do not change connectivity of the network, and moreover they are not learnable parameters.

number of different axes. One of the primary ways in which P can vary is in the structure of the parameter space. If a network has k layers such that each layer has m_i nodes, $i \in \{1, ..., k\}$, then we can identify P as $P \simeq \mathbb{R}^{\sum_{i=1}^{k-1} m_i \times m_{i+1}}$. Different choices of arrangements of nodes in layers will result in different identifications of P with some high-dimensional real vector space.

P can also vary at the level of constraints that dictate connectivity of the network. For instance, constraining weights between layers to take the form of specific matrices results in different ways in which information is communicated layer to layer. We can put convolutional neural networks into context here as an example of a choice of constraint on the matrix form of weights: convolution corresponds to a linear map whose matrix is constrained to be doubly block circulant [Goodfellow et al., 2016, 329]. In general, parameter sharing can be encoded by constraining blocks of entries within a matrix to be equal, while different patterns of connectivity between layers result from forcing certain matrix entries to be 0. Practically speaking, what this suggests is to explore the use of different matrix forms as constraints on the parameter space, as a way of endowing $N_P \subset \text{Fun}(X, Y)$ with structure; in just the same way that convolutional neural networks benefit from the added structure of assuming dependence of features only on local data, as well as translation equivariance (both of which emerge from the use of doubly block circulant matrices) so the use of other matrix forms might result in networks with special properties.

In the section above, we characterized choosing a neural network architecture − in practice a choice involving overwhelmingly many different variables, and a process still approached as an experimental science − as being fundamentally the specification of two mathematical objects: a parameter space $P \subset \mathbb{R}^\infty$, which can be endowed with more or less structure, and a map $N : P \to \text{Fun}(X, Y)$ that associates choices of parameters with functions. It is via these identifications that we can specify a loss $L : P \to \mathbb{R}$ as a function of network parameters.

1.4. **Training as sampling from N_P.**

We return now to the question of choosing a distribution over functions $\text{Dist}_F : \text{Fun}(X, Y) \to \mathbb{R}$. The previous section leaves at our disposal a distribution $L : P \to \mathbb{R}$ over the parameter space P of a given network, which induces a distribution $\mathcal{L}_P : N_P \to \mathbb{R}$, where $N_P = N(P) \subset \text{Fun}(X, Y)$. Intuitively, $\mathcal{L}_P$ should in some way inform Dist_F, since a "good" function in terms of Dist_F (i.e., one that well-approximates the "true" F) should also be reasonably good over N_P (i.e., well-models our observed data). However, $\mathcal{L}_P$ is severely limited in its ability to determine how well $f \in N_P$ will approximate F, since it is informed by only finitely many examples $(x_t, y_t) \in D$ which, moreover, are probably noisy. Indeed, $\mathcal{L}_P$ will tend to concentrate on $f \in N_P$ that fit D (and, in particular, its noise) extremely well, at the expense of their ability to approximate F on values of x not seen in the data. The phenomenon of overfitting is the result of having $\mathcal{L}_P$ inform Dist_F too heavily.

Thus, we can incorporate $\mathcal{L}_P$ as a term (but not the only term, lest we overfit!) in computing Dist_F, as a measure of function "good"-ness, on any given f. Moreover, we can also use $\mathcal{L}_P$ as a prior over how we sample f from N_P. That is, in order to actually produce the distribution Dist_F, we must define its values on some set of functions $\mathcal{S}$, and because we cannot compute Dist_F over all of the infinite space N_P (it is computationally infeasible even to compute it on all but samples from small local neighborhoods in N_P!), we would like at least to sample as many "good" functions as possible; the value of $\mathcal{L}_P$ at any given f (in particular, its gradient) gives us a good heuristic for how to sample nearby functions.

Thus we are now faced with the task of producing Dist_F, using $\mathcal{L}_P$ as a guide. We can characterize different neural network training techniques as being one of the following:

(1) methods to, for any given $f \in N_P$, define the value $\mathrm{Dist}_F(f)$ (which depends on $\mathcal{L}_P(f)$ in some way)
(2) methods to produce a sample set $\mathcal{S} \subset N_P$ on which Dist_F is defined.

1.4.1. *Type (1): Regularization.* The most immediate way to reduce the dependence of Dist_F on $\mathcal{L}_P$ is to add other terms to the computation, so that Dist_F becomes a function not only of $\mathcal{L}_P$ but also of other variables. Different forms of Lp[4] regularization, like the familiar L1 and L2 regularizations, make Dist_F also a function of P itself, skewing Dist_F in favor of functions f_θ with parametrizations θ that lie closer to the origin in P (i.e., those with lower-magnitude weights). This builds into the definition of Dist_F a form of Occam's Razor: the prior that the true F is probably a fairly smooth, "well-behaved" function.

Adversarial training also performs regularization [Goodfellow et al., 2014], though by a different mechanism than that of Lp regularization: while the latter encourages smoothness of f_θ indirectly by putting a prior on values of θ, the former manipulates f directly, forcing it to be smooth by assigning the same label y_t to a number of different points $x_{t,i} = x_t + \epsilon_i$ which can be interpreted as samples from a ball of radius ϵ around x_t, where ϵ is the maximum perturbation norm of adversarial examples. In terms of Dist_F, the effect is to have a prior that favors $f \in N_P$ for which $\mathcal{L}_P$ is constant over a neighborhood of f. That is, adversarial training can be seen as adding a term to $\mathrm{Dist}_F(f)$ that depends on the smoothness of $\mathcal{L}_P$ in a neighborhood of f. This added term is, however, only implicit in the use of adversarial examples, which is in practice implemented by modifying the data rather than by augmenting a loss as in Lp regularization. What this interpretation of adversarial training suggests, however, is that it might be worth exploring ways to do regularization by explicitly adding terms to $\mathrm{Dist}_F(f)$ that depend on the values of the loss on nearby functions $f' \in U \subset N_P$ a neighborhood of f.

1.4.2. *Type (2): Optimization.* We now come to the matter of choosing the actual set $\mathcal{S} \in N_P$ of functions for which we compute $\mathrm{Dist}_F(f) \in \mathbb{R}$ in the search for a "good" $\tilde{F}$; the function we end up choosing as $\tilde{F}$ depends not only how we evaluate the "good"-ness of functions, but also on the set of functions we are able to evaluate. As previously mentioned, the gradient of $\mathcal{L}_P$ at any given

[4](excuse the notation!)

$f \in N_P$ provides information about how to sample the next function in a way that minimizes the loss. If we have differentiable regularization terms for Dist_F at any given f, we may even use the gradient of Dist_F to inform sampling this way. This is, of course, the general framework according to which gradient-based methods of training operate.

Within this gradient-based framework, however, there are a number of training methods that affect sampling. The initialization of the network, in general done randomly, dictates which region of N_P we sample from, while the learning rate, usually thought of as a training hyperparameter, determines how close in N_P our sampled functions are, and thus the density of sampling in any given neighborhood of N_P. Various optimization methods affect how we sample: momentum-based methods, including Nesterov accelerated gradient [Nesterov, 1983], make the direction of sampling at every step dependent on the direction of sampling at the last step; meanwhile, methods that employ an adaptive learning rate (e.g., Adagrad [Duchi et al., 2011], Adadelta [Zeiler, 2012], RMSprop [Ruder, 2016]) make the density of sampling dependent on either previous sampling densities, previous sampling directions, or on absolute position in N_P. If we think of sampling as moving through the space N_P with some direction and velocity, optimization methods become a description of the "physics" of the function space, given that we move through the space according to the heuristic of following a gradient.

2. Even more abstract nonsense

What we have established is a formalization for the training process for a particular network. Now, we can zoom out a bit and return to the notion of generalization by laying out the abstraction hierarchy surrounding a neural network.

We can frame supervised learning in the following way: we have a task $T \in \mathbf{Task}$, the set of all possible tasks. This has a natural[5] correspondence with a dataset $\mathcal{D} = (X, Y) \in \mathbf{DSet}$, the set of all datasets. Note that in the first section, we were considering some finite subset $\mathcal{D} \supset D = \{(x_t, y_t),\ t \in T\}$[6] as the training dataset for a particular network. Now, for each task, we choose a model in **Models** as the method for learning the task. The set of neural network architectures **Net** (previously denoted simply as N, but in this context I'll use **Net** because it's more consistent with the rest of my labelings in this section) is a subset of **Models**, and is in direct correspondence with the set **NetPSpaces** of parameter spaces that parameterize neural networks. We choose an architecture $N_P \in \mathbf{Net}$ to train, which corresponds directly to a choice of parameter

[5]Sort of; the dataset in general implies a task, and a task (specified to the appropriate level of abstraction) also a dataset - however, in principle this isn't *necessary*, in that you could take a cat/dog image dataset and specify the task to be identifying birds. It makes no sense, but in principle it's "possible".

[6]note the use of T as both referring to a task and serving as an index set. This was somewhat by accident, and doesn't *exactly* line up contextually between the two uses, but it actually ends up working out, in that we end up having the association of a finite set of data points with a particular learning task, e.g., "identify cats vs. dogs". There's an abuse of notation going on, but as a heuristic it works.

space $P \in \mathbf{NetPSpaces}$. Since neural networks are parameterized by their real-valued weights, we have that P is isomorphic to $\mathbb{R}^n$, where n is the number of trainable weights in the neural network.

$$\begin{tikzcd}
\textbf{Models} \arrow[d, phantom, "\cup"] & \textbf{PSpaces} \arrow[d, phantom, "\cup"] \\
\textbf{Net} \arrow[r] \arrow[d, phantom, "\Cup"] & \textbf{NetPSpaces} \arrow[d, phantom, "\Cup"] \\
N_P \arrow[r, mapsto, "\sim"] \arrow[d, "\sim"] & P \simeq \mathbb{R}^n \arrow[d, equal] \\
N_* \arrow[r, "\sim"] \arrow[d, phantom, "\Cup"] & P \arrow[d, phantom, "\Cup"] \\
N_{trained} \arrow[r, mapsto, "\sim"] \arrow[d, mapsto, "\mathrm{predict}_{x_{test}}"] & p_{trained} \\
\hat{y} = N_{trained}(x_{test}) &
\end{tikzcd}$$

An isomorphic, but slightly conceptually different, way of considering N is as N_*, the space of all possible fixed-weight networks with architecture N. This is, equivalently, the space of all possible parameterizations of a network with architecture N, which we have identified with $P \simeq \mathbb{R}^n$. Now, any given fixed-weight network - e.g. a fully trained network (along with the corresponding locally optimal parameterization) - is an element $N_{trained} \in N_*$. Being, as it is, specified by its parameterization (assuming some given architecture), it is in direct correspondence with some $p_{trained} \in P$.

$N_{trained}$ specifies, for any $x \in X$, some prediction $\hat{y} \in \hat{Y}$. In particular, it is a function

$$N_{trained} : X \to \hat{Y}$$

that serves as the approximating function $\tilde{F}$ of F as discussed in section 1. In the diagram above we have defined the map $\mathrm{predict}_{x_{test}}$ such that

$$\mathrm{predict}_{x_{test}}(N_{trained}) = N_{trained}(x_{test}).$$

3. Generalization

We return now to the notion of generalization. To say that we want models to generalize is to say that we want to approximate F as well as possible for (in principle observable) X as large as possible, given training data $D = \{(x_t, y_t),\ t \in T\} \subset (X, Y)$ as small as possible. Simply stated as though it may seem, this basic setup is essentially what characterizes all of supervised learning's goals. We can frame the generalization ability of a model as the size of a subset of *unseen* examples $X_{test} \subset X$ over which it can approximate $F : X \to Y$ well (that is, within some error). More concretely, we think of a model as generalizing well if it is able to correctly classify many unseen examples. One way to increase this ability is to train multiple different nets, initialized differently, and then for any given unseen example, use the aggregate information of many separate models (e.g., in the form of an average) to make a single prediction. This sort of *ensemble* method increases model generalizability by reducing the effect of (non-systemic) error in any one given net.

Here, we gain robustness by averaging over *predictions*. Notice that if we return to the diagram

$$\begin{tikzcd}
\textbf{Models} & \textbf{PSpaces} \\
\cup & \cup \\
\textbf{Net} \arrow[r] & \textbf{NetPSpaces} \\
\rotatebox{90}{$\in$} & \rotatebox{90}{$\in$} \\
N \arrow[r, mapsto, "\sim"] \arrow[d, "\sim"] & P \simeq \mathbb{R}^n \arrow[d, equal] \\
N_* \arrow[r, "\sim"] & P \\
\rotatebox{90}{$\in$} & \rotatebox{90}{$\in$} \\
N_{trained} \arrow[r, mapsto, "\sim"] \arrow[d, mapsto, "\text{predict}_{x_{test}}"] & p_{trained} \\
\hat{y} = N_{trained}(x_{test}) &
\end{tikzcd}$$

ensemble modeling corresponds to averaging over objects that occupy the lowest "rung" of this abstraction ladder (predictions). With this in mind, and now having made our levels of abstraction explicit, we can ask: what if, instead of averaging over the bottom rung, we average over objects one level of abstraction up? Indeed, in general, taking an average over objects at a lower level of abstraction, and then abstracting, yields something different than abstracting first, then taking an average over those more abstract objects.

So we are faced with asking the question, if you can average over predictions of nets, **can you average over nets themselves?**. In particular, what does this even mean, and is it useful? It turns out that since I first asked this question, [Izmailov et al., 2018] experimented with doing this, to surprising success. I address their method in the next section.

Part 2. Averaging and the alignment problem

(Or, 'almost getting to the point - but first, a problem, plus some other people who ran into the same problem')

4. Averaging Over Neural Networks

It is with this motivation of generalizability via averaging that we finally come to the interesting part of all of this nonsense. In order to have a coherent notion of average, we first need a coherent notion of distance or *metric* between objects in N_*. At first glance, it might seem as though we might be able to simply use the Euclidean metric, especially considering that we have identified $N_* \simeq P \simeq \mathbb{R}^n$, the last of which is endowed with the Euclidean metric. However, while this serves as a perfectly reasonable metric in general, it turns out not to be the notion of distance that we're looking for and that is meaningful to us in machine learning. The naive Euclidean metric fails to account for the symmetries induced by the structure of a neural network: in any given layer, nodes can be permuted arbitrarily without changing the function the network computes. Treating a network simply as a point in $\mathbb{R}^n$ doesn't account for these permutations of nodes in layers as inducing equivalence classes of points (nets).

4.1. **Node Alignment.** A way to state this issue is that given any two arbitrary $N_1, N_2 \in N_*$, there is no a priori guarantee that the nodes are "aligned" in a way that makes taking their Euclidean distance meaningful in the context of learning. On the other hand, consider the following: it's generally recognized that training a neural network corresponds to tracing out a smooth path (from an arbitrary initialization to a local optimum) over a loss landscape. But if this is true, then it must be the case that any given point on the path must have its nodes aligned with its infinitesimal neighbor along the path (otherwise a permutation of nodes would cause a discontinuity between nets *interpreted as points*, contradicting the idea that training traces a smooth path). Since this holds across the entire path, we arrive at the following

Hypothesis: two nets can be thought of as "aligned" or "alignable" in their nodes if they are connected by a smooth path between them over a loss surface.

Recent research actually has much to recommend this idea: [Garipov et al., 2018] discovered that nearby local optima are connected in parameter space via paths of near-constant loss (refuting the previously widely-held assumption that local minima were generally isolated). Moreover, they developed a procedure known as *fast geometric ensembling* (FGE) for finding these paths, and then using samples from those paths in an ensemble model. They found that the ensembles built using networks sampled along the paths had particularly good generalization ability. [Izmailov et al., 2018] then built upon this observation, noticing that FGE typically finds models for ensembling that lie on the periphery of an optimum (that is, lie along the rim of a basin in the loss surface). They used the intuition that if points on a surface lie around the rim of a basin, then their average should be a point somewhere around the bottom of the basin in order to justify averaging the *weights* of the network rather than taking a standard enseble with an average over predictions (see Figure 1, taken from their paper). They present this averaging of weights as a technique they call *stochastic weight averaging* (SWA).

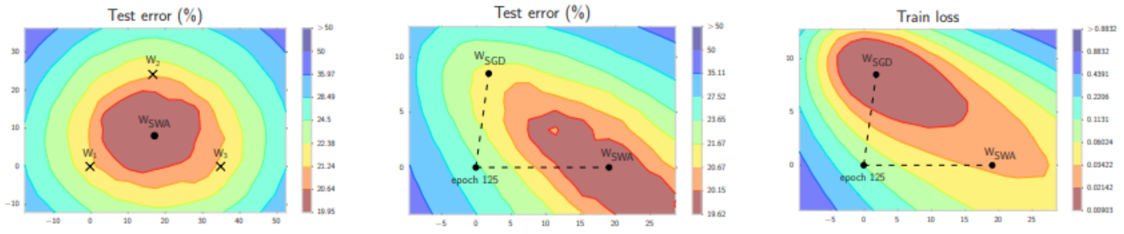

Figure 1: Illustrations of SWA and SGD with a Preactivation ResNet-164 on CIFAR-100[1]. **Left**: test error surface for three FGE samples and the corresponding SWA solution (averaging in weight space). **Middle** and **Right**: test error and train loss surfaces showing the weights proposed by SGD (at convergence) and SWA, starting from the same initialization of SGD after 125 training epochs.

SWA seems to be able to produce models that compete with the current state-of-the-art, but that achieve their high performance at a much lower training cost than is required of its competitors. That is to say, the idea of weight averaging seems to be a good one. As Izmailov et al. recognize in their paper, however, SWA is extremely limited because it only works if the weights to be averaged correspond to points that lie around a basin in the loss surface. In particular, it only works for a local cluster of points, clearly connected by some smooth path. The locality of these points ensures that their nodes are "aligned properly". Unfortunately, in general, this doesn't seem to be the case.

In general, we are faced with the problem of having two discrete, distinct objects, and wanting to compare them, but being unable to because the information/structure/data associated with each object individually does not a priori "match up" or align with the information of the other. This seems to be a fairly common phenomenon in mathematics - in the absence of a "choice of basis" (abstractly speaking), there is no way to guarantee that two arbitrary objects, even if they are of the same type, are comparable. In some way, "drawing a smooth path", or more generally, doing smooth interpolation, between objects, is a way of forcing objects to adhere to the same "choice of basis" or "system of orientation".

Many failed experiment ideas later...

Now, it turns out that trying to work with the idea of "smooth path" when it comes to extremely high-dimensional computations that are, moreover, not even actually over the reals but are only some approximation thereof, is very hard. It's hard to even try to figure out where to start, or what "smooth path" actually even means when it comes to an actual implementation in code. (I learned this the hard way.) If, however, the goal is ultimately to force networks to adhere to "the same choice of basis", then perhaps we ought to pursue such a thing directly.

Part 3. Symmetries and "a choice of basis", or unique representation

(Or, 'now actually getting to the point')

5. Symmetries

As mentioned previously, a fully connected deep neural network is an object with many symmetries. More specifically that in any given hidden layer of the network, any permutation of the nodes within the layer amounts to the same network as the original, without permutation. That is, the fully-connectedness of the network allows for permutations within hidden layers without changing the function the network computes.

Our current standard way of representing neural networks (or parameters, thereof) - as collections of matrices, in a way that treats the nodes of hidden layers as having an ordering - do not, however, account for any of the permutation symmetries of hidden layers. Thus, even though the family of networks produced by the permutations of a single hidden layer all are "the same network" (or isomorphic), in that they compute the same function, we treat them as though they are distinct objects; we consider the corresponding points in parameter space as being different when we should in fact identify them with each other, collapsing all permutations into a single "point". The absence of this identifying structure, with which the parameter space P is induced by the neural network structure, but which is abandoned when we treat P as $\mathbb{R}^n$ (as is current standard practice) is what makes the naive notion of weight averaging node-wise incoherent.

Thus we want to consider all networks (or their representations in parameter space) to be unique modulo permutations within hidden layers. This such uniqueness modulo hidden layer permutations we might call "functional uniqueness". Note that also the redundancy in not identifying these permutations makes the parameter search space in some ways artificially large. If we were able to account for permutations of this kind as yielding the same network, then we could shrink our search space to that containing only "functionally unique" points; that is, if we accounted for all permutations within all m hidden layers $h_1, ..., h_m$, our search space would be $\mathbb{R}^N/(S_{h_1}, ..., S_{h_m})$: the original search space, $\mathbb{R}^N$, modulo the symmetric (permutation) groups S_{h_i} for $i \in [1, m]$ generated by the nodes in h_i.

Of course, as theoretically straightforward as this is, the problem is precisely that in practice, we do not have a good way of identifying points that are separate in $\mathbb{R}^N$ but that are functionally equivalent (and so collapse together when "modding out" by symmetries). What this requires is that each functionally unique point have a *unique representation* (with respect to the parameter space); or, stated otherwise, each equivalence class of functionally equivalent points must have a unique representative. It's this that we do not currently have. In fact, it seems strange even to think of a particular set of neural networks as being identified with a unique representation - exactly which point should we pick as the representative? Is there any way of doing this without choosing completely arbitrarily?

6. Choosing representatives

I shall here try to provide a way to choose such a representative. Given any tuple, if we identify all its permutations as being equivalent, then an obvious choice of unique representative for the equivalence class of permutations is the permutation in which the elements are sorted in increasing order. So given an equivalence class of neural network parameterizations, we can pick as its unique representative the point whose vector representation is such that within each hidden layer, the nodes are are listed in increasing order of their value - that is, the sub-vector corresponding to the values of a given hidden layer is an increasing sequence. Unfortunately, this still isn't quite right - the elements of the "vector"/"tuple" corresponding to a given hidden layer are not (in general) numbers, but are vectors themselves. In general, each layer is associated with a weight *matrix* (so that indeed, each *node* in a layer is associated with a weight vector), rather than with a weight *vector* for an entire layer. In order to use an (increasing) ordering of nodes in hidden layers as a way to specify a unique representative, we need to be able to specify an order on vectors. There are multiple different ways of doing this. Abstractly speaking, since real vectors have no inherent ordering, but real numbers do, it makes sense to try to map each vector $v_{i,j}$ (corresponding to the parameters of layer i, node j) to some real number, $f(v_{i,j})$, where $f : \mathbb{R}^{n_i} \to \mathbb{R}$, and n_i is the number of nodes in hidden layer i. We would then sort by $f(v_{i,j})$, which then acts as a kind of "proxy value", or statistic, by which vectors inherit the ordering of $\mathbb{R}$. There are multiple different possible candidates for the map f. If for now we let $v_{i,j} = (x_1, ..., x_k)$, we might have:

(1) $f(v_{i,j}) = x_1$. That is, we sort by the first coordinate. In the case of a tie, we sort those using $f_2(v_{i,j}) = x_2$, by the second coordinate. In the case of a tie there, we use the third coordinate, etc.
(2) $f(v_{i,j}) = \|v\|$, for some vector norm. The obvious first candidate here would be the standard magnitude, or Euclidean norm.
(3) $f(v_{i,j}) = \text{mean}(x_1, ..., x_k)$
(4) f is a linear functional that takes some particular $v^* \in \mathbb{R}^{n_i}$ as an argument
(5) f is some other kind of functional taking some particlar $v^* \in \mathbb{R}^{n_i}$ as an argument.

These are just the obvious first few to try.

Next steps from here: future work will explore concrete implementation of this idea.

Again, what has been established here is a possible way to endow the space of neural networks with the notion of distance, and as a corollary, the notion of average. If networks are represented in this universal way, then now the Euclidean distance could be used without inconsistency. If this notion of metric over neural networks works in practice, then the reprecussions for areas like metalearning and transfer learning may be great. The following are a few ideas regarding the application of the notion of neural network distance beyond simply averaging networks within some given task.

7. Generalizing Across Tasks

Suppose now this notion of metric over N_*. If that's the case, then taking a distance between nets makes sense *independent of task*. That is, if we have two networks with the same architecture, we should be able to talk about their distance whether they've been trained on the same task or on different ones. This, of course, now motivates a discussion of transfer learning.

7.1. **Some Remarks on Training Practices.** In general, when training models on data, we start completely from scratch every time we encounter a new task. That is, we frame different learning tasks as being separate problems, and by and large consider the information learned by training on one dataset to be irrelevant to learning a different dataset. This, however, with only a moment's consideration seems absolutely preposterous. Consider just the set of tasks corresponding to classifying images of a fixed size: the set of images that are structured enough to be recognizable and meaningful to humans forms an extremely low-dimensional (relatively speaking) subspace of the set of all possible images of that fixed size. Any two tasks involving humanly recognizable images, no matter how apparently dissimilar (e.g., classifying cats vs. dogs and classifying trees vs. road signs), share an incredible amount of information and are much closer together in the whole data space than each is to an arbitrary random noise image.

The immediate conclusion to draw is that given a fixed net architecture (and fixed image size, for e.g.), a new task should always be initialized with the weights of a net that has been previously trained on some other task. This seems certainly like it should be better than a random initialization - but can we do better?

7.2. **Smoothing the Transfer in Transfer Learning.** The problem with using a net trained on one task as an initialization for training on a different task in many ways parallels the problem we encountered when trying to find some notion of distance between neural networks: we're trying to "jump" discretely from one object to another, with no guarantee that the information/structure from one scenario matches that of the target scenario. Here, what transferring tasks given a fixed network corresponds to is to hold a point still but change the loss landscape around it. In particular, even if our fixed point (network) sat in a local minimum in its trained loss landscape, there is no guarantee that that same point occurs anywhere good in the target task's loss landscape - in fact, we have no a priori information at all about what the value of the loss will be for that fixed point in the new loss landscape.

What's worse, we have no transferred information about how to improve the loss, since the information of the prior task suggests that movement in any direction *increases loss* (w.r.t the prior task). Again, we face a jump discontinuity kind of problem.

Can we do something here like we did for node-alignment previously? Can we "trace a smooth path" through the space of loss landscapes so that we can make the two discrete landscapes we're considering "align" nicely? Indeed, topology suggests to us the notion of "smooth deformation" of surfaces, or a smooth interpolation between the two loss landscapes. In practice, this might look like doing some smooth interpolation of images in the training datasets.

Next steps from here: future work will explore concrete implementation of this idea.

The upshot of doing smooth interpolation of loss landscapes is illustrated by the following image: consider some loss landscape, at some local minimum of which rests a ball. This corresponds to the loss landscape of the initial task and the net trained on it (the ball) as being a point lying in a local minimum. Now consider the landscape changing discretely underneath the ball so that now the ball rests at some point on a hill or even local maximum. This discrete jump of the loss landscape is what occurs if we simply initialize a the target task's network with the weights of the prior task's trained net. Now imagine the loss landscape as some kind of blanket that starts in the position of the first landscape, and that morphs smoothly to take the form of the target landscape. Because the ball is always rolling downward (i.e., the network is always doing gradient descent in training), seeking the minimum at any given moment, the smooth deformation of one landscape to another ensures that even if the landscapes of two tasks differ, a net trained on a dataset that smoothly interpolates between the two tasks will not get stuck in a suboptimal valley, and will always maintain information for how to improve its loss.

Part 4. More advantages of unique representation, and other more abstract thoughts

8. Shrinking the search space

As mentioned previously, another advantage of establishing some unique representation is that it enables us to shrink the search space by identifying seemingly different networks that are actually equivalent. We want to be able to enforce a search rule that amounts to something like "only look for neural netork parameterizations such that the parameterization that corresponds to any given hidden layer occurs as an increasing sequence (for whatever choice of f and order we pick)". Depending on the choice of f, this translates to the point-in-parameter-space view as specifying either a direction, a subspace, or a "directed" sequence of subspaces in which we should search for (local) optima. Computationally, enforcing an increasing order of elements amounts to enforcing that for any choice of x_1, the vector of differences $(x_2 - x_1, x_3 - x_2, ..., x_k - x_{k-1})$ is non-negative.

Thus we might try to reformulate the training optimization problem as one of finding the strictly non-negative difference vector for which the original vector $(x_1, ..., x_n)$ is a part of (locally) optimal parameterization of the neural network.

It is this reformulation of the problem as optimizing over difference vectors that achieves our initial goal of shrinking our search space: instead of a high-redundancy search for local optima over $\mathbb{R}^N$, we now have a search over unique parameterizations via (local) optimization of difference vectors in $\mathbb{R}^{N^-}_{\geq 0}$, where $N^- \leq N$ (since a k-dimensional regular vector corresponds to a $(k-1)$-dimensional difference vector, along with some starting x_1).

This experiment is still, however, entirely *in cognito* (as opposed to *in silico*) - it remains to be seen which of the above choices of f (if any) admit a translation of the problem into one of difference vectors and, moreover, one that is computationally reasonable and is compatible with some version of the (largely gradient-based) optimization methods we currently have.

Next steps from here: future work will explore whether this can work in our learning framework, and whether it's really even worth it.

9. Ambitious and abstract hopes and dreams and stuff

Looking for unique representations can easily be motivated by concrete and optimization-based reasons. However, my looking for some notion of "unique representation" is also part of a much broader abstract project. There's a sense in which one of the biggest problems right now (maybe the biggest one, depending on how you look at it) in deep learning is that all of our representations (encodings) of networks are non-generalizable. Indeed, there's the well-known problem of how well neural networks generalize with respect to a learning task, but all of that is lying on top of the problem of the representation *of the network itself* being such that there's currently no meaningful way in which any binary operation of two networks (even with an identical architecture, and differing only in parameterization) could ever make sense (that is, aside from all of what I've just presented). The data, or information, of the network is fundamentally tied to the network itself, even at a strictly "syntactic" level (at the level of parameter values and arrangements, and as opposed to a "semantic" level at which we identify a parameterized network as a function). I think this is largely because we use ordered objects to represent information that is at some level "unordered" (at least insofar as the information carried by nodes in hidden layers is node-permutation-invariant). Because we treat two functionally equivalent networks (that differ only by some hidden layer permutation) as different networks (in representation), given any parameterized network we have no way of extracting the information of the weights (in some sense, "what's learned by the network") in any way that generalizes beyond arbitrary permutation of hidden layers. In this sense, our representations of a network is given with respect to a "basis" induced by the enumeration of nodes in layers, and what we would like is a representation of a network that is "basis-free". Put another way, we would like to separate what's learned by the network from the network itself. In practice, this requires a way to uniquely (and *universally*, as a system across all neural network architectures) represent functionally unique networks. And the representation I outlined above is perhaps a good choice. But there is still something dissatisfying about *needing* a basis, period. One should be able to speak of the information of a network without necessarily having to do that with respect to some arbitrary encoding[7].

10. Next steps, and to be continued

The main crucial next step is to try to implement all of this. It's entirely possible that none of this will actually work in implementation (or will be computationally expensive, or perhaps some other notion of "not working"). Of course, *in theory*, it should work...

The more general theoretical project (around trying to extract the *information* of networks, independent in some sense from networks themselves - and perhaps via some universal representation) is something I intend to pursue as an approach to transfer learning, and perhaps more broadly as a way to approach the exchange of information from one network to another in a way that preserves "what is learned" by the donor network in the context of the recipient network.

[7]On the other hand, I do tend to think this about essentially everything ("there should never be arbitrary choices, ever!")

References

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159.

[Garipov et al., 2018] Garipov, T., Izmailov, P., Podoprikhin, D., Vetrov, D., and Wilson, A. G. (2018). Loss surfaces, mode connectivity, and fast ensembling of dnns.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. The MIT Press.

[Goodfellow et al., 2014] Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and harnessing adversarial examples.

[Izmailov et al., 2018] Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., and Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization.

[Nesterov, 1983] Nesterov, Y. (1983). A method for solving the convex programming problem with convergence rate $O(1/k^2)$.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms.

[Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method.