# Consistency and robustness of neural networks

Alexa Aucoin[1]
[1]Department of Mathematics, University of Arizona

February 10, 2021

**Abstract**

**Developing tools to make neural networks robust and consistent across a wide range of data is of significant interest to data scientists. This comprehensive exam paper presents detailed examples of neural networks which suffer from data inconsistency or fragiility and discusses some possible avenues to recitfying these short-comings. In particular, we first look at a large set of primate brain activity data and highlight how simple classification networks fail to produce consistent results across the entire data set. We then present possible techniques and architectures thoughtfully chosen to improve the generalizability of a network trained using this data. We then shift our focus to the problem of robustness by exploring residual networks subject to adversarial attacks. Drawing from a dynamical systems perspective, many have used the addition of noise to increase the robustness of a trained network. This chosen noise is often small but arbitrary and, in large systems, can increase the complexity of a network significantly. We discuss a potential choice of noise which is informed by the training data set and perhaps simpler to implement. This work is ongoing and future directions are also discussed at the end of the paper.**

## 1 Introduction

Neural networks (NNs) have become increasingly popular as a tool to discover key patterns and correlates in data. They have been shown to be effective in a variety of classification and regression tasks, however, how these complex sytems work remains opaque. Training these networks often requires large data sets and, once trained, the network itself can be

quite fragile. This fragility can come from internal factors like variability in the representations within training data, or by more malicious means, like adversarial attacks from outside hackers. Developing tools to make NNs robust and consistent across a wide range of data is therefore of significant interest to data scientists.

NNs are prone to overfitting and in practice can ehibit poor generalizability on unseen data. This means the network does sufficiently well at completeing a task on the training data, but fails to achieve the same accuracy on new data from test and validation sets. Empirical evidence has shown, however, that deep neural networks (DNNs) are, in fact, able to successfully generalize but realizing such generalizability is heavily dependent on the choices of network design. With complex structure and many more parameters than data, this seems to contrast what we expect from over parameterization. Much work has been done to quantify the trade-off between network complexity and generalizability, and to develop best-practices when it comes to choosing network design [14] [10] [11] and [7]. These design choices include, but are not limited to choices of input and batch regularization, network architecture, loss functions, and optimization algorithms. Even with an extensive body of research on neural networks, how these complex systems learn still remains opaque. In this paper, we attempt to address yet another generalization issue with neural networks. In Section 3, we provide an example of experiemntal neuroscience data with significant variability in how the data is representated and discuss the challenges this variablility poses on network consistency. (Here, the term consistency refers to the networks ability to perform with similar accuracy on experimental data which is recording on different trial days.) We also present an example of such a variable data set and present possible regularization techniques and network architectures to improve upon this inconsistency.

In addition to issues of generalization, trained neural networks also remain vulnerable to adversarial attacks. This is a significant concern in networks designed to learn security- or safety-critical tasks. Such tasks occur in applications to malware detection, facial recognition and autonomous vehicles, which have been utilizing neural networks with increasing popularity due to the recent success of neural networks on natural language processing and image processing [8], [6] and [2]. In Section 4, we present a resdiual network framework for thinking about the robustness problem. Using a dynamical systems technique, we reformulate the adversarial attack problem on residual networks into an ODE and show how added noise has been used to improve network robustness [13]. We also give an elementary arguement for a choice of simple added noise informed by the training data. In the Section 5, we discuss future directions of this work. In particular, we present the potential for connecting these two projects to inform the design of more consistent and robust neural networks.

## 2 Background

Artificial neural networks are a collection of processing nodes (neurons) that are connected to one another through weights. The neurons are so named because they loosely mimic the computations done by the human brain. Each neuron accepts a weighted sum of inputs which is added to the neuron's bias. The bias can be thought of as the analog to the threshold value of a biological neuron. This sum, of the weighted input and bias, is then passed through an activation function which determines if and to what strength the neuron outputs information about the input signal to subsequent neurons. Mathematically, the computation done by the $i$-th neuron can be represented as

$$\hat{y}_i = a_i(\sum_{j=1}^{N} w_{ij}x_j + b_i) \tag{1}$$

where $a$ is the chosen activation function, $N$ is the number of input units, and $x, \hat{y}, w$, and $b$ are the input, output, weights, and bias respectively. These neurons are often organized into layers, starting with the input layer, followed by any number of hidden layers, and ending with an output layer. A hidden layer consists of hidden neurons which are defined by their choice of linear or non-linear activation function $a$. A visual of the computation done by a sigmoidal hidden unit is shown in Figure 1.

There are many choices of network design which determine how information propagates through the network. Between two layers, the units may be fully connected, where each neuron in a single layer accepts input from each neuron in the preceeding layer, or pooled, where each neuron only accepts input from a subset of neurons in the preceeding layer. Fully connected layers are very common, especially in the last layers of a classification network as they allow the network to share information from all sources. Pooling is most often used in image processing tasks as it allows for large, complex input images to be simplified into smaller representations by using convolutional layers. The direction of the connections between units is also a choice made during the design phase of a network algorithm. Feed-forward networks are networks which allow information to propogate only from one layer to the next, while recurrent neural networks (RNNs) allow for more complicated flows of information (see Figure 2). In an RNN, a hidden layer can have one or many feedback loops. These loops are backwards connection from one layer to itself or to any number of the preceeding layers. As one might expect, this makes RNN more difficult to train, but allows for more sophisticated information to be stored such as memory. Choices related to network design and connectivity are often informed by the type of input data and the desired learning task.

The goal of these artificial networks is to approximate a function $f$ which maps a set of input data $x$ to an output $y$. This mapping is defined by the set of parameters, $\theta$, which includes all of the weights and biases in the network. The process of updating the parameters $\theta$ to more accurately (defined with respect to a chosen loss function) map input data to an output is called learning. For the purposes of this paper, we will consider

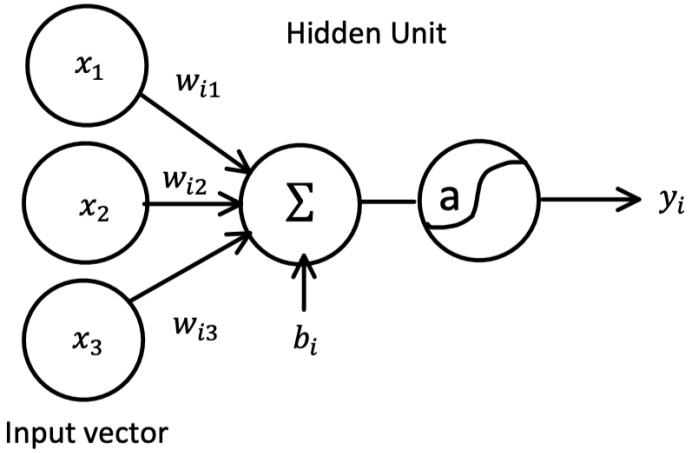



Figure 1: An expanded visualization of the parameters and computations associated with a single hidden unit $h_i$. It accepts as input a vector of length three and assigns associated weights to each connection with an input neuron. Then the unit computes the weighted sum of the inputs plus the bias term $b_i$. This sum gets passed through a sigmoidal activation function $a$ before returning output $y$. The set of parameters $\theta = (w_i j, b_i)$ can all be updated by the network during learning.

only supervised learning tasks, but interested readers may refer to [3] for details on unsupervised learning algorithms. In supervised learning, each observation in the training set, $x$, is associated (or labelled) with a desired output, $y$, called a target. The network learns to approximate the mapping $f$ by examining sample observations of the desired mapping. We call this collection of pairs $(x, y)$ the training data.

During a supervised learning task, an example from the training set is given as input to the network. The sample observation $x$ is propogated through the network and outputs an approximation $\hat{y}$. The accuracy of the network is then given by the distance between the approximated target $\hat{y}$ (outputted by the network) and the true target $y$. This distance is determined by a loss function, which is chosen during the design of the network. As with network design, choice of loss function will also be influenced by the desired learning task. For example, (binary) cross entropy, is often used for classification tasks while mean squared error (MSE) is commonly chosen for regression tasks.

The process of updating the weights $\theta$ to better approximate $f$ is done through an optimization algorithm which drives the parameters towards a local minimum of the cost function. Algorithms such as gradient descent and stochastic gradient descent are often used for the choice of optimizer. During opitimization, the gradient of the loss funcion with respect to $\theta$ is computed by a process called backpropogation. Readers can refer to

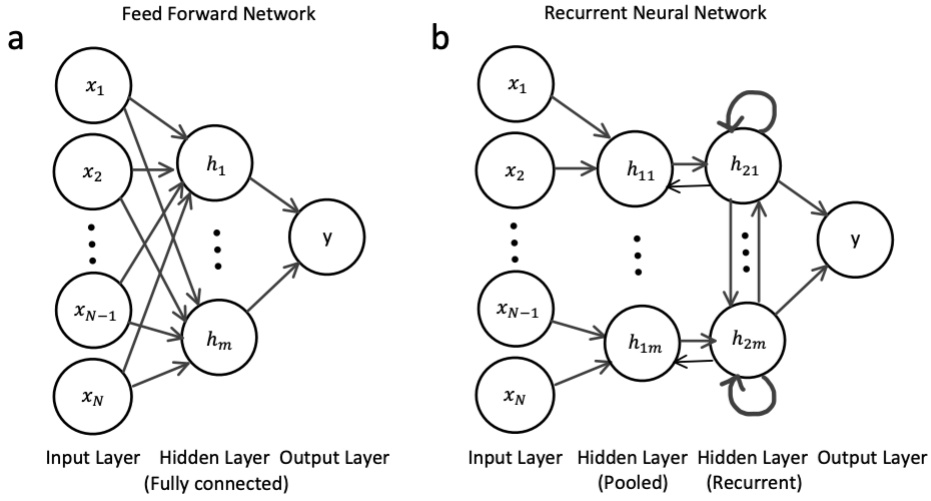



Figure 2: Simple examples of feed-forward and recurrent neural networks. In panel a, we present an example architecture for a feed-forward neural network. The hidden layer is fully connected, meaning each hidden unit has a connection from each input neuron. Within each hidden unit, a squashed linear mapping occurs (see Fig. 1 and Eq. 1) and their outputs are used to produce the arbitrary network output $y$.

[3] and many other sources for details on how backpropagation simply efficiently compute the gradient.

Learning is said to be complete when presenting the network with more observations from the training data does not signiciantly improve network accuracy. When learning is complete, the network parameters are fixed and can now be used to predict output values for new observations outside of the training set. Due to the large number of network parameters, it is possible for the network to experience over-fitting. This means the network performs very well for observations within the training data set, but exhibits poor accuracy for new data. Overfit networks suffer from a lack of generalizability. To mitigate issues of generalizability, it is common to set aside a set of validation data. This is data that is held separate from the training set and periodically passed through the network to check validate accuracy on new data.

There are many validation strategies used during training that can help mitigate overfitting. However,it no is not the only aspect of NN training that can lead to a lack of generalizability. Variability in the representations of the training data can also limit a network's generalizability. Careful choices of data processing, network architecture, and training are therefore an essential part of designing a network that is useful on never-before-seen data. However, the analysis of NNs remains a black-box mystery and guidance on the "best" network design remains context dependent and highly reliant on experienced intuition. In the following section, we present a concrete example of an experimental data set with significant variability in the representations along with some strategies to design reliably accurate NNs for this training set.

# 3 Designing a consistent neural network

To highlight how variability in training set representations creates issues with network consistency and design, we consider a large set of neural recording data. The data was collected from a primate experiment conducted by Katalin Gothard in the Gothard Lab at University of Arizona [4].

## 3.1 Example data from a neuroscience experiment

The experiment is broken up into five time blocks: 3 blocks of machine delivered puffs and 2 blocks of human touch. For this paper, we are particularly interested in the block of neural data recorded during the human touch stimulus. During a touch block, a human handler (with whom the monkey has a positive repore) touches the monkeys face in a sweeping gesture 10 times across the left upper muzzle (LUM), and then 10 times accross the left brow (LBR). This grouping of touches, reffered to as a touch trial, is repeated 4 times in each touch block for a total of 80 touches. The duration of the touch is approximately 1 second long, with 4 seconds between each touch.

The two brain regions recorded during these experiments are 3b and the amygdala. 3b is a portion of the primary somatosensory cortex responsible for decoding sensory inputs related to touch. This region on the brain is well studied and convenient because the neurons in 3b have a very structured receptive field. One can think of a neuron's receptive field as the portion of the input space that the neuron will respond to. In other words, it is the group of inputs that drive that neuron's spiking activity. The receptive field in 3b is so nicely structured that one can create a map of the face where each region corresponds to a group of neurons in 3b. This is a particularly nice feature as it makes designing experiments and analyzing data much more straightforward.

The amygdala is the part of the brain that is responsible for processing our emotions. Almost all information is passed through the amygdala at some point or another. For example, information on motor functions need to be sent to the amygdala so it can anticipate changes in sensory stimulus. The amygdala is constantly receiving and decoding information about our surrounding environment from all brain areas in order to inform our emotional responses. Because of its complex input, the amygdala is generally not well understood or even often studied. Unlike 3b, the amygdala does not have a well-structured receptive field and responds to a variety of inputs. By working with the data collected at the Gothard lab, we have a unique opportunity to study and learn from spiking activity in the amygdala.

During the experiment a probe is placed within the monkey's brain and records electric pulses transmitted by the neurons in 3b and the amygdala. These electrical pulses are referred to as spikes and are thought to contain important information about changes in the environment that may impact the body. Through a complex inverse problem, the electric pulses recorded by the probe are is assigned to a particular neuron. The methodology for this inverse problem is beyond the scope of this paper, but an interested reader can refer to the associated dissertation [5]. Once all spike times are associated with a particular neuron, we obtain the brain activity of the primate as a series of spike times, referred to as spike trains, for each recorded neuron.

During the experiment, the monkey's heartrate is also recorded. Changes in the primate heartrate can be used as in indicator that the monkey undergoing a change in emotional state due to the presence of a stimulus. This is particularly useful as we expect that the human touch stimulus is an enjoyable one for the primate. An important feature of the heartrate data for this experiment was the drop in the primate's baseline heartrate during a touch trial. It is reasonable then, to expect there exists a correlation between the recorded spiking activity and the underlying presence (or lack) of stimulus. This provides resonable evidence that a sufficiently designed neural network could learn to infer the underlying stimulus from spatiotemporal spiking data. Additionally, machine learning techniques have recently been successful in analyzing/modelling experimental spike trains obtained from more well-studied regions like V1, hippocampus, olfactory systems [9] [1] [12]. There is reason to believe it may also be useful in understanding how information is encoded or decoded in 3b and even more complex regions like the amygdala. In

particular, we would like to train a simple classification network to detect the difference between spike activity that is recorded during the presence of a touch stimulus versus latent (baseline) spike activity.

## 3.2 Variability within the data

On any given day of experimental data, we have $N$ ordered lists of spike times, where $N$ is the number of individual 3B cells recorded (Fig. 3 A). To prepare the experimental data to be fed through a feed-forward network, we first discretize the time of the experiment into blocks of 5ms. For each time block of 5ms, and for each $N$, we count the number of spikes occuring in that 5ms window. This yields a vector of spike counts of length $N$. We then append 5 of these vectors together, which yields a matrix of spike counts of length $N$ by 5 corresponding to a 25ms window in the experiment (Fig. 3 B) and are then flattened into a $5N$ column vector so they may be used an input into a network (Fig. 3 C). We do this for the duration of the experiment, and assign each matrix one of two labels: stimulus or latent. Matrices labelled as stimulus correspond to spiking activity during 25ms of active touch stimulus, while matrices lablled as latent correspond to spiking activity which occured entirely during a period of latency (no stimulus present). These latent times can occur between touches or between experimental blocks. Since these periods of latent were longer and more frequent, there are much more latent data points than there are stimulus data points. To ensure that both labelled classes were represented equally, we randomly sample from the latent data with replacement until we had an equal number of data points in both classes. The labelled matrice of latent and stimulus data make up our training data set.

The variability in the data comes from pooling together data recorded on different days of the experiment. Between different days of experimental recordings, the probe is left in the primate brain and it is very common for the probe to shift during this time. As such, we can expect that the spiking activity is recorded from a new population of neurons for each day of the experiment. Additionally, due to the senstivity of the patches on the probe, the number of neurons recorded on any given day is also not guarenteed. This creates a significant problem when trying to use the spike data across days as input into the same neural network. Certainly we can expect that the function mapping the spike data to an underlying stimulus remains similar across days, but on any given day we have significant variability in the number of cells recorded. In the remainder of this section, we attempt to address the following questions: How do we design a network to accept variable-size input data while being reasonably confident it will generalize to new data from the same experiment? Can we design such a network that is reasonably complex without over-manipulation of the data?

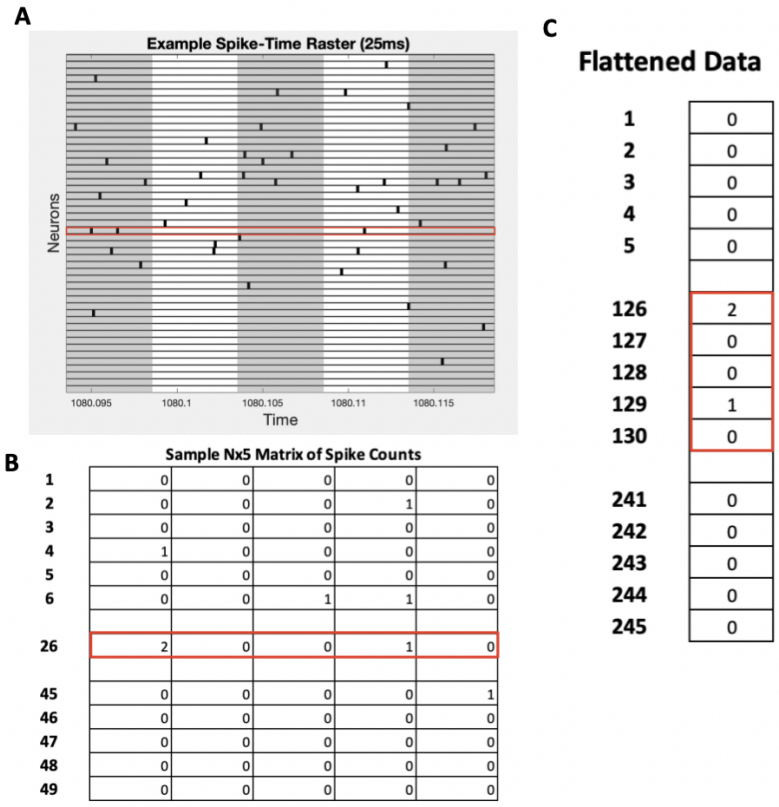



Figure 3: Visualization of the input data processing for the an example of a 25ms touch stimulus observation. Panel (A) is a raster plot of the spike times for $N = 49$ neurons. The alternating shaded regions of the graph represent a time window of 5ms. The entries corresponding to spike times for neuron 26 is highlighted throught the panels as a visual example. Panel $B$ shows the corresponding $N$ by 5 spike count matrix before it is flattened. Panel C shows the flattened matrix to be used as input into the neural networks described in this paper.

| Algorithm | Accuracy | $\lambda$ |
|---|---|---|
| Logistic Regression | 71.63% | 0.007 |
| SVM | 71.14% | 0.025 |

Table 1: Results of two simple supervised learning algorithms for a single days of touch data where $N = 49$. Validation accuracy shown is the best achieved out of range of regularization strengths $\lambda \in [10^{-6}, 10^{-0.5}]$.

### 3.3 Network inconsistencies on variable neural data

Before we tackle the variability across days, we first test the performance of simple neural networks on data from a single day of the primate experiment. This is a useful task for exploring the kinds of network design choices that result in an a efficient, accurate neural network for this kind of experimental data. We start first with two simple supervised learning algorithms: logistic linear regression and support vector machine (SVM) with a linear kernel. These linear algorithms are very easy to implement and their training is well-studied. As such they are a natural starting point before applying more complex learning algorithms. For both algorithms we tested a range of values for $\lambda$, the regularization strength. This is a tunable hyperparameter that helps to ensure generalizability of data by encouraging network sparsity. Larger $\lambda$ leads to stronger penalties for large parameter values in $\theta$. Table 1 summarizes the best validation accuracy achieved for each algorithm.

These simple linear algorithms perform both perform similarly, and certainly leave much room for improvement. This suggests our data may not be linearly separable. To introduce some nonlinearity and model complexity, we trained a three-layer long-short term memory (LSTM) network. Details about the architecture of LSTMs can be found in [3], but the important feature of the recurrent LSTM network is the ability to identify order dependencies in a sequence by storing information about past inputs. This allows for more context based classification decisions which is potentially relevant in the context of neural recordings. The shallow LSTM is also implemented fairly easily. A visualization of the LSTM network architecture is given in Figure 4

To avoid over-fitting during training of the LSTMs, we implemented two standard techniques: dropout and bootstrapping. Dropout is a regularization method in which neurons in the network are randomly excluded from the network during training. This is achieved by setting their activations and weights to 0. This introduces sparsity into the network and potentially prevents over-fitting. Bootstrapping does not introduce network sparsity, but rather increases the size of the training set by sampling the training data with replacement. Empirical evidence shows that providing the network with more samples from the input distribution allows for more reliable estimators about the distribution. Thus, bootstrapping is useful in the potential prevention of over-fitting. Table 2 shows the

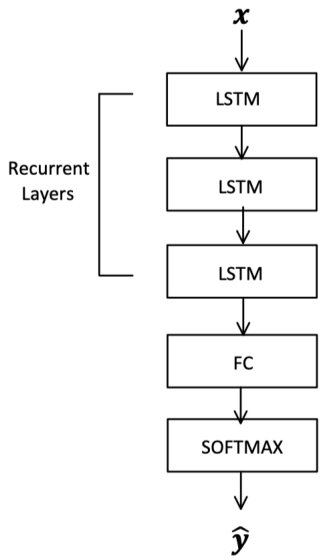



Figure 4: Layer architecture for the three-layer LSTM classification network used on a single day of spike-train data.

| Algorithm | Accuracy |
|---|---|
| LSTM (plain) | 68.42% |
| Dropout LSTM | 64.98% |
| Bootstrap LSTM | 95.22% |

Table 2: Validation accuracy of three-layer LSTM trained on a single days of touch data where $N = 49$ with no regularization techniques, dropout and bootstrapping.

results for the LSTM training exercise. Although the plain LSTM and dropout LSTM algorithms preformed worse than the linear classification algorithms, the significant victory here is the validation accuracy achieved on single day with bootstrapping.

The results of this groundwork provides evidence that relatively straight-forward neural networks can successfully learn on data from a single day of the experiment. Recall that our aim is to design a network that can be reliably extended to classify experimental data from any given day. This necessarily means addressing the issue of non-uniform input size across days.

## 3.4 Methods and potential algorithms

One possible way ensure our algorithm is accruate across days is to be able to train one network on a training set consisting of data from across all days of the experiemnt. This is difficult with the data we currently have since the shape of the input space changes across

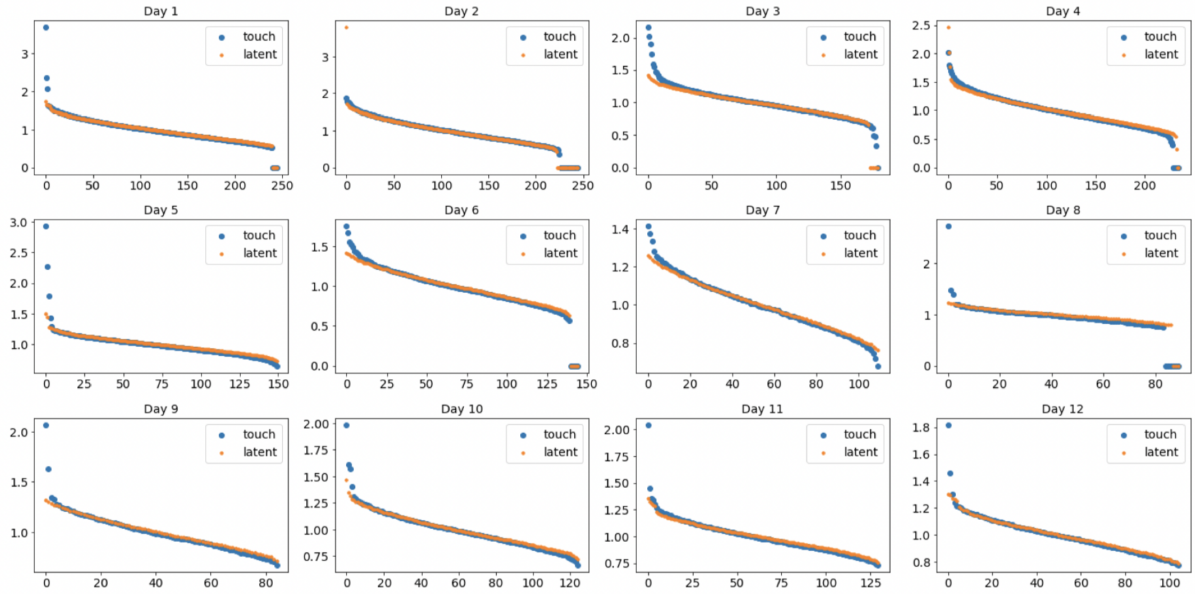



Figure 5: The explained variance coefficients (eigenvalues) for latent and touch data across all experimental days. Results how qualitatively similar behavior across days, with no indication of a natural cut-off point for the low dimensional projection.

days. One way to rectify this is to project each days data onto a subspace of prescribed size $k$. In our experimental case, this can be thought of as picking a $k$ dimensional subset of the $N$ recorded neurons. This can consist of choosing $k$ indivual neurons or $k$ linear combinations of neurons. Applying this dimensional reduction for each day results in a uniform size of input data across days. Of course, there is always a trade-off between dimensional simplicity and accuracy. In projecting onto this lower dimensional subspace, we are losing potentially important information. Principal component analysis or PCA is a natural way to project data onto a lower dimensional subspace while also preserving the features of the data with the most variability. This is particularly desirable in applications to machine learning where variability is seen as containing crucial information. PCA can potential transform non-separable data into separable data. This is a potential benefit as separability in the input data space allows for greater success in classification tasks.

We start by computing the PCA for each of the twelve days' data. To compute PCA for a given day, we must first standardize the statistics for each neuron. This is necessary since PCA is particularly sensitive to variance across input features. In this case, the input features are the spike counts for a particular neuron in a single time bin. Thus, there is a total of $5N$ input features. For ease of computation, we flatten each $N \times 5$ input matrix to be a single vector of length $5N$. We then compute the standardized data $\tilde{x}$ by

$$\tilde{x_i} = x_i - \mu_i / \sigma_i$$

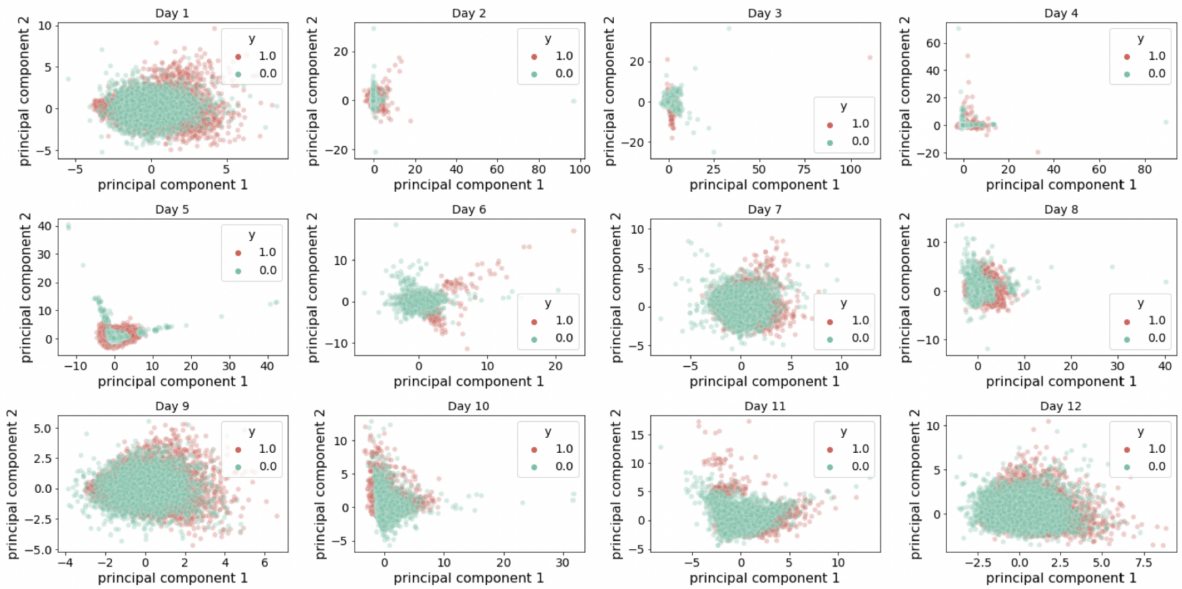



Figure 6: The projection onto the first two principapl components for latent and touch data across all experimental days. Results how variable the data is across days, and that the data is highly non-separable.

where $\mu_i$ and $\sigma_i$ are the mean and standard deviation for the $i$-th input feature respectively. Once standardized, we compute the covariance matrix for $\tilde{x}$ and its eigenvector and eigenvalues. The eigenvectors of the convariance matrix are the principal components, the direction along which the data has the most variability. Their associated eigenvalues can be thought of as a coefficient which quanitfies the amount of variability in that direction. By dropping the last $N - k$ principal components and projecting the data along the remaining subspace, we can transform the data across days while losing the least amount of variability. This allows us to have a uniform input size across days.

Nevertheless, PCA is still useful to us. By using PCA to transform the existing experiemental data, we can create a training set of uniform input size that represents multiple days of experimental data. This allows us the freedom to implement standard network designs. PCA also equips us with the means to transform completely new data into a space consistent with our current training set. This is important as we want to be able to handle experimental data with an input size not previously seen before.

There are some concerns with using PCA as a way to preprocess the training data that are worth discussing. As mentioned above, there is the concern with dimensional simplicity at the expense of accuracy. With no obvious threshold cut-off for principal component variability and the significant difference in the number of components across days, there is a legitimate concern that the PCA may be throwing away too much information

embedded within the data. Additionally, the PCA must be computed on each days data individually. This means there is really guarentee that the $k$-dimensional projections of data from different days are encoded with the same feature information.

It is also important to mention that there is little to no biological or real-world meaning that can be interpreted by the principal components. Using PCA largely means giving up the hope for a tractable and practical understanding of any classification results obtained from this data. If the long term goal is to use machine learning to inform our hypthosis about the computational mechanisms in the brain, PCA is not necessarily the top choice. Nevertheless, the computation is quick and simple to implement and classification algorithms using the PCA data are worth exploring.

An alternative way to make the input shape uniform across days comes from an idea inspired by Generative Adversarial Networks (GANs). A GAN consists of two competeing sub-networks, a Generator and a Discriminator, and is best understood by a more colloquial analogy. Consider the problem of a banker and a criminal trying to create a counterfeit dollar bill. In this analogy, the buyer plays the role of the Discriminator network and the criminal plays the role of the Generator network. The criminal will first generate some counterfeit money and try to deposit the it at the local bank. The banker will examine the bill counterfeit against real bills, point out the discrepancies, and send the criminal home. The criminal, now equipped with the knowledge of what went wrong, tries to improve his counterfeit bill and try again. This will continue until the criminal makes a counterfeit bill so good that the discriminator cannot tell the different between the fake and a real bill. Training for a GAN network follows the following steps:
**Generator:**

1. Create samples of latent space data using random noise.
2. Forward propagate noisy data through the generator to create "fake" data.
3. Pass the generated data through the discriminator and get predictions.
4. Calculate the discriminator's output loss on the generated data with the data falsely labelled as "real". (This step is crucial as we want an error in the discriminators classification to *positively* impact the weights of the generator.)
5. Backpropagate through the generator **only**.

**Discriminator:**

1. Pass a batch of real data to the discriminator.
2. Pass a batch of generated data to the discriminator.

3. Average the loss from Steps 1 & 2.

4. Backpropagate through the discriminator only.

The key concept in GANs is that the competing networks work together to learn the key features of the input data. Error from the discrimintors loss is back-propogated through the network and used to update the weights of the generator. By competing against the discrimintor, the generator learns how to map a sample from one distribution space (often a random distribution) to something that mimics, quite accurately, a sample from the desired distribution space. We propose a modified GAN framework to rectify the dimensional variability in our experimental data.

Our modified GAN framework will be made up of three types of networks: an encoder, a decoder, and a discriminator. It will often be useful to consider a specific encoder and decoder together as one pair. We refer to this pairing as an autoencoder. In this framework, the autoencoder plays the role of the generator. Suppose we train a separate autoencoder for each day of experimental data. The first part of the autoencoder, the encoder, will accept as input a vector of spike counts of length $N \times 5$ and ouput an encoded representation of the data of specified length $k$. The second part of the autoencoder, the decoder, will accept as input the encoded $k$-dimensional representation of the data and output a reconstructed version of the original $N \times 5$-length vector of spike counts. The third part of the network is a single discriminator, this discriminator will accept as input the $k$-dimensional encoded data and produce an output which classifies what day the representation comes from.

The training of our modified GAN network will work as follows:

**Initial start-up:**

1. Train an autoencoder to encode and decode data from Day 1 of the experiment. Training will stop when the autoencoder can accurate reconstruct the original data from the $k$-dimensional encoded representation.

**Generator:**

1. Initialize a new autoencoder.

2. Generate $k$-dimensional encoded data.

3. Pass the $k$-dimensional encoded data through the discriminator and get predictions.

4. Calculate the discriminator's output loss on the new day's encoded data with the data falsely labelled as "old". (This step is crucial as we want an error in the discriminators classification to *positively* impact the weights of encoder.)

5. Forward propagate the encoded data through the corresponding decoder.

6. Calculate the reconstruction loss.

7. Take a convex combination of the discrimination loss and reconstruction loss.

8. Backpropagate through the autoencoder.

**Discriminator:**

1. Pass a batch of old days' data to the discriminator.

2. Pass a batch of new day's data to the discriminator.

3. Average the loss from Steps 1 & 2.

4. Backpropagate through the discriminator only.

5. Repeat Generator and Discriminator steps for each additional day of data.

Using this framework, we design a network to learn the best $k$-dimensional representation of the data rather than force our choice of low dimensional representation. Using the same discriminator across days ensures that the $k$-dimensional representations are all encoded in a similar way, while the individual decoders enforce that not too much data is lost by the dimensional reduction. This is a potentially useful way to make the input dimensions consistent across all experimental days as it overcomes two of the main concerns of the PCA approach. Intrepetability of the $k$-dimensional representation using the GAN approach is still unclear, as it relies on the analysis of a group of trained networks, which is still a topic of interest that is not well understood.
The concern with using this approach is mainly due to the fact that it is fairly complex. It requires training a new network for each new day of data, which can potentially be time consuming. Though there are some techniques in transfer learning may help ameliorate this. Additionally, it is unclear whether the overall accuracy of all data encoders will converge to a shared minimum. Early implementations of this network architecture have been successful in creating encoded data for the first 5 days or so, but additional days lead to a overall blowup of training accuracy. Work to remedy this issue and develop a stable training algorithm is still on going.

In the next section, we shift our focus from possible approaches to improve generalizability to ones aimed at improving robustness of trained neural networks. Both are important obstacles of interest in the application of machine learning to large data sets which we hope to address in future work.

# 4 Robustness of Residual Networks

Residual networks (ResNets) were first introduced in 2015 to address the issue of exploding/vanishing gradients and have sinced made training ultra-deep networks (networks over

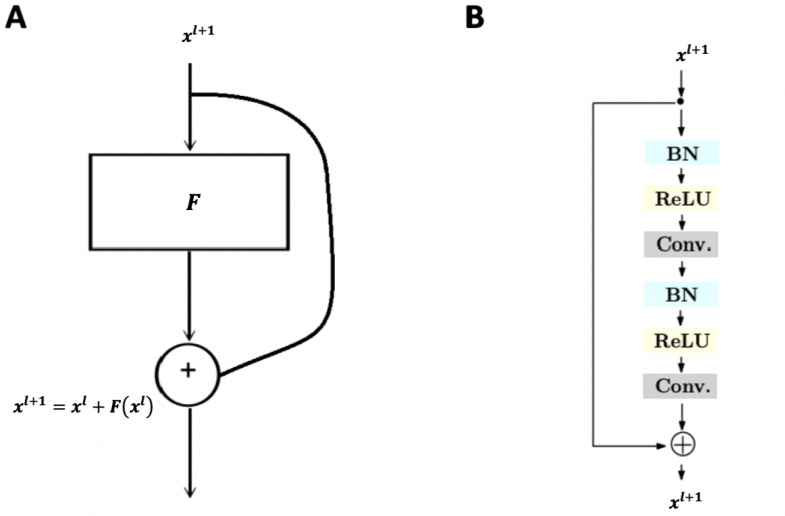



Figure 7: Panel A is a visualization of the skip connections in a residual layer. Here, $x^l$ represents the input into the $l$-th layer and $x^{l+1}$ its output. This is useful notation as residual networks are usually designed as a forward-propagating sequence of residual blocks. An example of the pre-activation residual block used for this discussion is given in Panel B. These figures were modified from figures in the 2019 paper by Wang et. al. "ResNets Ensemble via the Feynman-Kac Formalism to Improve Natural and Robust Accuracies"[13].

1000 layers) pragmatic. In previous discussion, we considered a mapping, $y = f(x, \theta)$ from input data $x$ to target $y$ with parameters $\theta$, to be learned by a feed-forward network. Assuming that the input and output have the same dimension, we can ask that the network instead learn the residual mapping $r(x) = f(x) - x$. This is achieved by placing a skip connection which takes the input layer and to the output layer, bypassing the intermediate layers of the model. A visualization of this skip connection is given in (Fig 7 A) Thus, the true mapping to be learned by the network will be $F(x) = r(x) + x$. While the network should be able to approximate both functions asymptotically in theory, the residual network formulation allows for a better conditioned learning task.

The problem of exploding/vanishing gradients is called the degradation problem. As more layers are added to neural networks, the network will begin to converge, then become saturated, and then abrubtly lose accuracy with the addition of more layers. This degradation of accuracy suggests that, in the presence of many nonlinear layers, the network may have difficulty in approximating near identity mappings.

In the residual learning framework, we assume that mapping between input and output is near-identity. In the case where the identity mapping is optimal, the network can drive the weights towards zero to approximate an identity mapping. In practical cases,the

identity mappings will likely not be the optimal mapping, but nevertheless the residual learning framework has been shown to precondition the learning problem better, with learned residual functions having much smaller responses in general [13]. Ensembles of residual networks, which consist of a group of residual networks trained simultatneously, have been shown to be extremely accurate on practical image recognition tasks.

## 4.1 Fragility under adversarial attacks

Although the residual learning framework made training DNNs pragmatic, these deep networks remain vulnerable to adversarial attacks and there is a reluctance to apply DNNs to practical applications where security and/or safety is crucial. This includes but is not limited to automonomous vehicles, robotics, malware detection etc.. REFERENCES
Adversarial attacks can fall into two distinct categories: white-box attacks and blind attacks. In white-box attacks, the attacker has acces to all of the information about a network. This includes access to the models architecture, weights, loss function, and gradients with respect to input. Many of the white-box adversarial attacks perturb specific inputs to the model while some attacks also include perturbing the model parameters. Blind attacks are more challenging to implement as the attacker has very limited access to information about the model. Blind attacks often solely focus on perturbing input data into the model to force a degradation in the model's performance.
For this paper, we consider white-box type adversarial attacks on an ensemble of residual networks. The task of this ensemble is to correctly classify images from the CIFAR10 data set. The two adversarial attacks we consider, Fast Gradient Sign Method (FGSM) and its iterative counterpart (IFGSM). Both methods exploit the loss function to create new adversarial images. The adversarial images $x_{adv}$ are created by perturbing a true input images $x$ in the direction that maximizes the loss function $L$ with respect to the aversarial image $x_{adv}$. Mathematically, this is calculated by solving the following optimization problem

$$\begin{cases} \max\limits_{x_{adv}} L(F(x_{adv}, \theta), y) \\ ||x_{adv} - x||_\infty < \epsilon. \end{cases} \tag{2}$$

For linearized loss function, the solution is

$$x_{adv} = x + \epsilon \text{sign}(\nabla_x L(x, y)). \tag{3}$$

Geometrically speaking, these attacks move the input image $x$ closer to the decision boundary between its target class $y$ and some other class. One option to combat these types of adversarial attacks is to regularize the decision boundaries associated with the classification tasks. That is, if we can make the decision boundaries less sensitive to small perturbations, we can mitigate the effects of these white-box attacks. To help conceptualize this idea, we reformulate the neural network classification problem into a neural ODE.

## 4.2 ResNets as neural ODEs

We consider a residual network that is made up of a series of pre-activated residual blocks (Fig 7 B). Each block can be thought of as a layer that computes a residual mapping. The $l$-th residual block take in input $x_l$ and computes the mapping

$$x_{l+1} = x_l + F(x_l, \theta_l)$$

where $x_0$ is a sample from the training set $T \subset \mathbb{R}^d$ and the parameters $\theta_l$ are learned through back-propogation of the loss function. Here $F(x_l, \theta_l)$ is a pre-activation residual block and can be written as

$$F(x_l, \theta_l) = \theta_l^{C2} \otimes \sigma(\theta_l^{BN2} \odot \theta_l^{C1} \otimes \sigma(\theta_l^{BN1} \odot x_l))$$

where superscript BN are batch normalization layers, supercript C are convolutional layers and $\odot, \otimes$ represent that batch normalization and convolution operators respectively. The activation function $\sigma$ is an application of the rectified linear unit (ReLU). ReLU is the linear piecewise function

$$\sigma(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0. \end{cases}$$

In a supervised learning task, each element $x \in T$ will have an associated label $y$. We can represent the full ResNet as

$$\begin{cases} x_{l+1} = x_l + F(x_l, \theta_l), \quad l = 0, \ldots, L-1 \\ x_0 = x \\ \hat{y} = f(x_L) \end{cases} \tag{4}$$

where L is the number of residual blocks, $f(x) = \text{softmax}(\theta \cdot x)$ and $\hat{y}$ is the network's prediction of true target label $y$ for the initial input $x$.

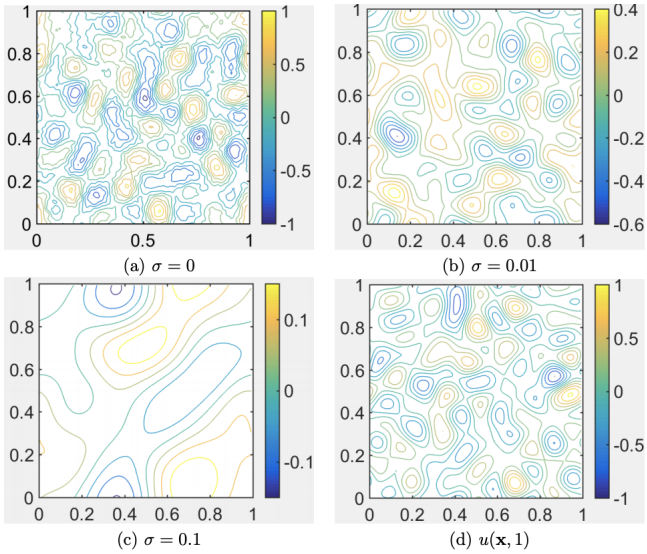
It is helpful to view the training of the ResNet as a an ODE. To do so, we will introduce a temporal discretization where "time" refers to the layer depth. Let $\Delta t_l = l/L$ for $l = 0, 1, \ldots, L$ and $x_l = x(t_l)$, then Eq. 4 becomes

$$\begin{cases} x(t_{l+1}) = x(t_l) + F(x(t_l), \theta(t_l)), \quad l = 0, \ldots, L-1 \\ x(0) = x \\ \hat{y} = f(x(1)). \end{cases} \tag{5}$$

Multiplying the $F(x(t_l), \theta(t_l)$ term by $\Delta t / \Delta t$ and defining $\tilde{F} = \frac{1}{\Delta t} F$ we obtain the forward Euler discretization

$$\frac{dx(t)}{dt} = \tilde{F}(x(t), w(t)), \; x(0) = x. \tag{6}$$

How noise regularizes the decision boundary of network (transport eq?)



(a) $\sigma = 0$ (b) $\sigma = 0.01$ (c) $\sigma = 0.1$ (d) $u(\mathbf{x}, 1)$

Figure 8: (a),(b),(c) are solutions to the convection-diffusion equation at $t = 0$ with different values of the diffusion coefficient $\sigma$. (d) is the terminal condition for the PDE. This figure was adopted from the 2019 paper by Wang et. al. "ResNets Ensemble via the Feynman-Kac Formalism to Improve Natural and Robust Accuracies"[13].

### 4.3 Added noise to improve robustness

Successful improvements to training with the addition of added noise Gaussian noise is added at each residual block. Is this complexity necessary? Can we achieve similar or better accuracy with simpler additive noise informed by the training data?

## 5 Discussion and Future Work

Plan for dissertation research...

# References


[1] Andrea Banino et al. "Vector-based navigation using grid-like representations in artificial agents". In: *Nature* 557.7705 (2018), pp. 429–433. DOI: 10.1038/s41586-018-0102-6.

[2] Ronan Collobert and Jason Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning". In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, 160–167. ISBN: 9781605582054. DOI: 10.1145/1390156.1390177. URL: https://doi.org/10.1145/1390156.1390177.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] *Gothard Lab*. URL: http://www.gothardlab.org/.

[5] Patrick Greene. "A Bayesian Approach to Spike Sorting of Neural Data via Source Localization". PhD thesis. The University of Arizona, 2018.

[6] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[7] Elad Hoffer, Itay Hubara, and Daniel Soudry. *Train longer, generalize better: closing the generalization gap in large batch training of neural networks*. 2018. arXiv: 1705.08741 [stat.ML].

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*, p. 2012.

[9] Ivan Lazarevich, Ilya Prokin, and Boris Gutkin. *Neural activity classification with machine learning models trained on interspike interval series data*. 2018. arXiv: 1810.03855 [q-bio.NC].

[10] Jocelyn Sietsma and Robert J.F. Dow. "Creating artificial neural networks that generalize". In: *Neural Networks* 4.1 (1991), pp. 67–79. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90033-2. URL: https://www.sciencedirect.com/science/article/pii/0893608091900332.

[11] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (June 2014), pp. 1929–1958.

[12] Charles F. Stevens. "What the fly's nose tells the fly's brain". In: *Proceedings of the National Academy of Sciences* 112.30 (2015), pp. 9460–9465. ISSN: 0027-8424. DOI: 10.1073/pnas.1510103112. eprint: https://www.pnas.org/content/112/30/9460.full.pdf. URL: https://www.pnas.org/content/112/30/9460.

[13] Bao Wang et al. *ResNets Ensemble via the Feynman-Kac Formalism to Improve Natural and Robust Accuracies*. 2019. arXiv: 1811.10745 [cs.LG].

[14] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *CoRR* abs/1611.03530 (2016). arXiv: 1611.03530. URL: http://arxiv.org/abs/1611.03530.