Flexibility and robustness of neural networks

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

February 17, 2021

Abstract

Developing tools to make neural networks robust and flexible 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 fragility and discusses some possible avenues to rectifying 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 flexibility 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 the potential for a choice of noise which is informed by the training data. 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 systems 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 flexible across a wide range of data is therefore of significant interest to data scientists.

NNs are prone to overfitting and in practice can exhibit poor generalizability on unseen data. This means the network does sufficiently well at completing 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 [17, 13, 14] and [8].

Related to the issue network generalizability is the lack of network flexibility. Here we use the term flexibility to describe a network's ability to accept and learn from data representations of different sizes or number of features. Currently, networks are designed to accept inputs of a specified size and the number of input features informs the size of a network's hidden layers. This poses a challenge for developing networks that can successfully learn accross data with non-uniform input size and features. There is little guidance for how to develop flexibile neural networks without transforming the input data significantly. In Section 3, we provide an example of experiemntal neuroscience data which exhibits representation variability across different experimental days. We discuss the challenges this poses on designing a network that can generalize across days and present possible techniques and network architectures which provide more input flexibility.

In addition to a lack of flexibility, 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 their recent success learning natural language processing and image processing tasks [9], [7] and [3]. In Section 4, we present a residual 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 [16]. We also give an elementary argument 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. The 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.

Architecture. There are many choices of network architecture which determine how

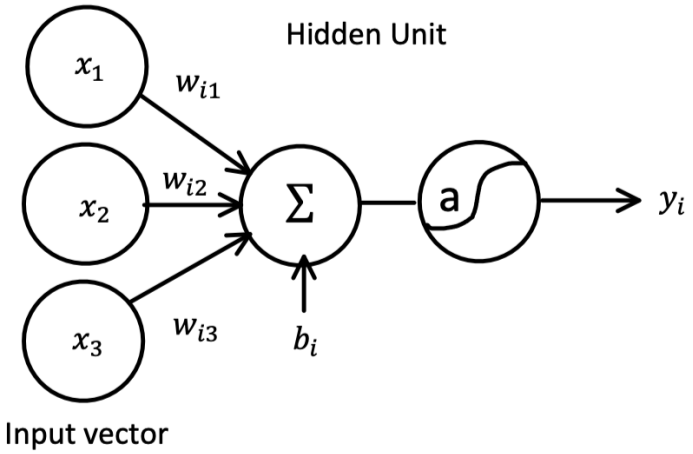


Figure 1: An expanded visualization of the parameters and computations associated with a single *sigmoidal* hidden unit h_i. It accepts as input a vector of length three and assigns a weight to each connection with an input neuron. The unit computes the weighted sum of the inputs plus the bias term b_i.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

information propagates through the network. Between any two layers, the units may be

fully connected, where each neuron in a single layer accepts input from each neuron in the preceding layer, or *pooled*, where each neuron only accepts input from a subset of neurons in the preceding 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. This is often achieved by using *convolutional* layers which act like a filter that extracts important features fromt the input.

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 propagate 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 connections from one layer to itself or to any number of the preceding layers. As one might expect, this makes RNNs more difficult to train, but allows for more sophisticated flows of information to be stored such as network memory. Choices related to network design and connectivity are often informed by the type of input data and the desired learning task.

Training 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, θ, which includes all of the weights and biases in the network. The process of updating the parameters θ to more accurately map input data to an output is called learning. For the purposes of this paper, we will consider only supervised learning tasks, but interested readers may refer to [4] 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 propagated 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 θ to better approximate f is done through an optimization algorithm which drives the parameters towards a local minimum of the loss function. Algorithms such as gradient descent and stochastic gradient descent are often used for the choice of optimizer. During optimization, the parameteres of the network are updated to minimize the loss function. To do so, it is necessary to compute the gradient of the loss function with respect to θ. This is computed by a process called

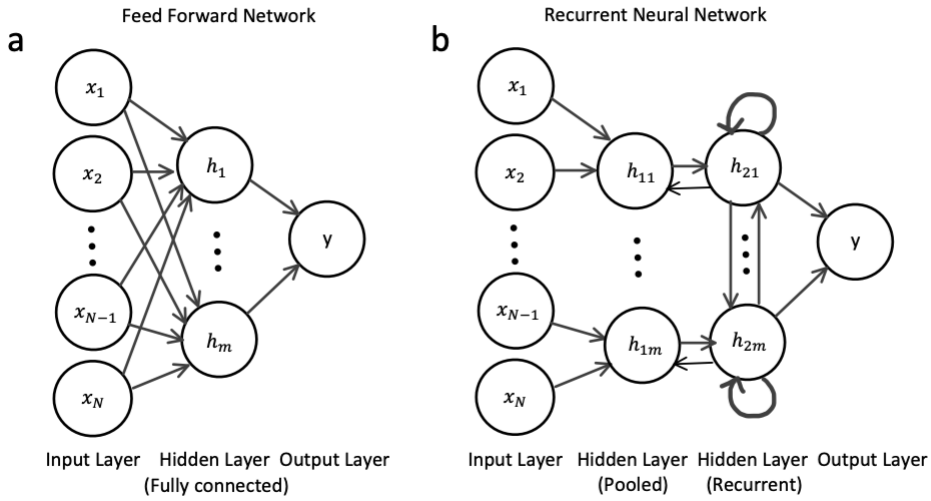


Figure 2: Simple examples of feed-forward and recurrent neural networks. In panel **a**, we present an example architecture for a feed-forward neural network. The hidden layer is fully connected, meaning each hidden unit has a connection from each input neuron. Within each hidden unit, a squashed linear mapping occurs (see Fig. 1 and Eq. 1) and their outputs are used to produce the arbitrary network output $\hat{y}$. In panel **b**, we present an example arhitecture for a recurrent neural network. The first hidden layer is a pooled layer where each hidden unit only has connections to two input neurons. The second hidden layer is a recurrent layer where each unit has a backward connection to a unit in the previous hidden layer, a lateral connection to the units within the same layer, and a feedback look connection to itself.

back-propagation. Readers can refer to [4] and many other sources for details on how backpropagation simply efficiently compute the gradient of the loss function.

Learning is said to be complete when presenting the network with more observations from the training data does not significantly improve network accuracy. When learning is complete, the network parameters are fixed and can 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 often 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 accuracy on new data.

There are many validation strategies used during training that can help mitigate overfitting. However, it 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. Neural networks are rigid and are designed to accept inputs of a specified size. For data sets that have variability in size or number of input features, the data is often preprocessing to fit into the network. Networks trained on this transformed data can exhibit good accuracy on some of the input representations and poor accuracy on others. 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 choices 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 flexible and reliable NNs for this training set.

3 Designing a flexible 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 and her team in the Gothard Lab at University of Arizona [5].

3.1 Example data from a neuroscience experiment

The neuroscience experiment simultaneously observes the behavior and neural activity of macaque monkeys during a tactile stimulus. The goal of observing both autonmics and neural activity during stimulus is to find neural correlates of behavior. The experiment is broken up into five experimental 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 rapport) touches the monkeys face in a sweeping gesture 10 times across the left upper muzzle (LUM), and then 10 times across the left brow (LBR). The group of touches (refered to as a touch trial) is repeated four 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 area 3b of the somatosensory cortex and the amygdala. 3b is the portion of the primary somatosensory cortex responsible for decoding sensory inputs related to touch. This region of 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 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 likely responsible for processing our emotions and behavior. The amygdala is constantly receiving and decoding information about our surrounding environment from all sensory areas of the brain 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 neural activity in the amygdala.

During the experiment a probe 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 [12, 6]. Once all spike times are associated with a particular neuron, we obtain the brain activity of the primate as a series of spike times for each recorded neuron. These vectors of spike times are called spike trains.

During the experiment, the monkey's heart rate is also recorded. Changes in the primate heart rate can be used as in indicator that the monkey undergoing a change in emotional state due to the presence of a stimulus. An important feature of the heart rate data for this experiment was the drop in the primate's baseline heart rate during a touch trial. It is reasonable then, to expect there exists a correlation between the experimental spike trains and the underlying presence (or lack) of stimulus.

Machine learning techniques have recently been successful in analyzing/modeling ex-

perimental spike trains obtained from more well-studied regions like V1, hippocampus, olfactory systems [10, 1, 15]. 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. Given the success of NN on spike train data from other brain regions and the indication of emotional response in the monkey's heartrate, we would like to train a neural network using the spike train data from this experiment. 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 occurring in that 5ms window. This yields a vector of spike counts of length N. We then append 5 of these vectors together, resulting in a matrix of spike counts of size $N \times 5$ corresponding to a 25ms window in the experiment (Fig. 3 B). This matrix is 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 vector one of two labels: touch stimulus or latent. Vectors labelled as stimulus correspond to spiking activity during 25ms of active touch stimulus, while vectors labelled as latent correspond to spiking activity which occurred entirely during a period of latency (no stimulus present). These latent times can occur between individual touches or between experimental blocks. Since latent periods 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. Roughly 80% of these labelled vectors of latent and stimulus data make up our training data set.

The variability in the training data comes using data recorded on different days of the same experiment. Between different days of experimental recordings, it is very common for the probe to shift slighlty. 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 sensitivity of the patches on the probe, the number of neurons recorded on any given day is also not guaranteed. This creates a significant problem when trying to use the spike data across days as input into the same neural network as the size of a network's input is generally rigid. 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

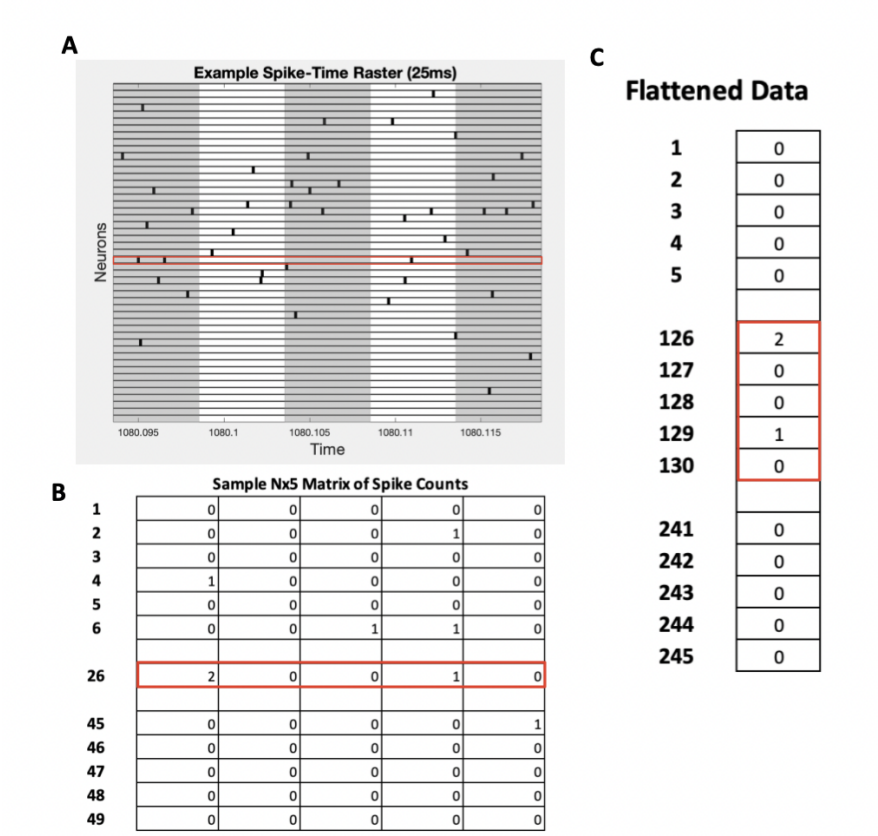


Figure 3: Visualization of the creation of a "touch" input vector. Panel **A** is a raster plot of spike trains for $N = 49$ neurons. The alternating shaded regions of the graph represent a time window of 5ms. Panel **B** shows the corresponding $N \times 5$ spike count matrix before it is flattened. Panel **C** shows the flattened matrix to be used as input into the neural networks described in this paper. The entries corresponding to spike times for neuron 26 are highlighted throughout the panels as a visual example.

Algorithm	Accuracy	λ
Logistic Regression	71.63%	0.007
SVM	71.14%	0.025

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

to address the following questions: How do we design a network to accept variable-size input data while being reasonably confident it will perform comparably on new data from the same experiment? Can we design such a network that is reasonably complex without over-manipulation of the data?

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 which network design choices result in an 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 λ, the regularization strength. This is a tunable hyper-parameter that helps to ensure generalizability of data by encouraging network sparsity. Larger λ leads to stronger penalties for large parameter values in θ. Table 1 summarizes the best validation accuracy achieved for each algorithm.

These simple linear algorithms perform both perform similarly ($\approx 71\%$), 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 [4], 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

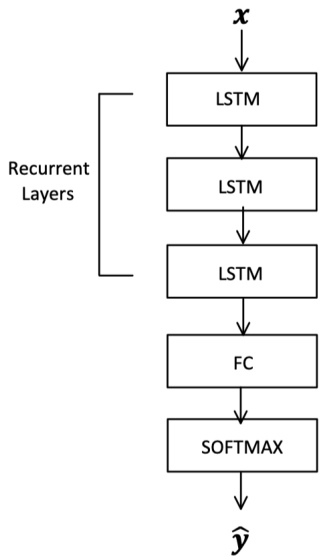


Figure 4: Layer architecture for the three-layer LSTM classification network used on a single day of spike-train data.

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.

replacement. Empirical evidence shows that providing the network with more samples from the input distribution allows for more reliable estimators about the distribution of the data. Thus, bootstrapping is useful in the potential prevention of over-fitting. Table 2 shows the results for the LSTM training exercise. Although the plain LSTM and dropout LSTM algorithms preformed worse than the linear classification algorithms ($\approx 65 - 68\%$), the significant victory here is the validation accuracy achieved on single day with bootstrapping ($\approx 95\%$).

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.

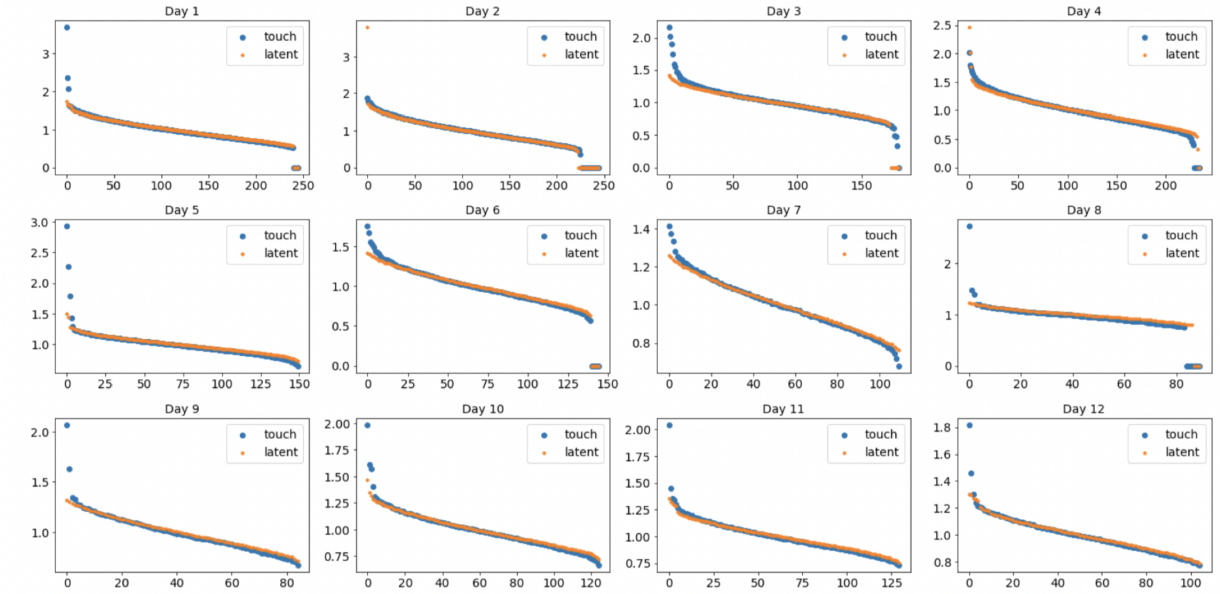


Figure 5: The explained variance coefficients (eigenvalues) for latent and touch data across all experimental days. Results show qualitatively similar behavior across days, with no indication of a natural cut-off point for the low dimensional projection.

3.4 Methods and potential algorithms

PCA. One possible way ensure our algorithm is accurate across days is to be able to train one network on a training set consisting of data from across all days of the experiment. This is difficult with the data we currently have since the shape of the input space changes across 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 individual neurons or k linear combinations of neurons. Applying this dimensional reduction to 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 information crucial to distinguishing between classification classes. In some cases, PCA can 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

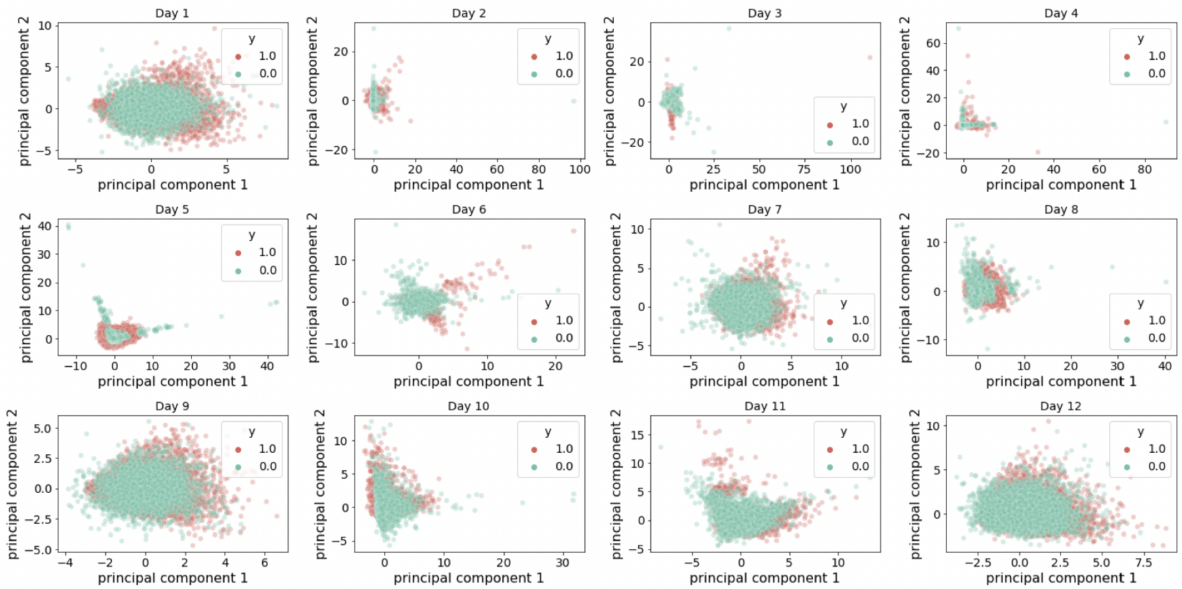


Figure 6: The projection onto the first two principal components for latent and touch data across all experimental days. Results show how variable the data is across days, and that the data is highly non-separable.

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. Using the input vectors of length $5N$, we compute the standardized data $\tilde{x}$ by

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

where μ_i and σ_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 covariance matrix are called the *principal components* which describe the direction along which the data has the most variability. Their associated eigenvalues can be thought of as a coefficient which quanitifies the amount of variability in that direction. Ideally, we can list the eigenvalues in descending order and there will be a clear number k such that all the $k+1$-st eigenvalue and below are significantly smaller than the first k. By dropping the last $N-k$ principal components and projecting the data onto the remaining k-dimensional subspace, we transform the data across days into the same *dimensional* space while losing the least amount of variability.

By using PCA to transform the existing experimental data, we 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 no guarantee 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 hypothesis 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.

GAN An alternative way to make the input shape uniform across days comes from an idea inspired by Generative Adversarial Networks (GANs). A GAN (Figure 7 *left*) consists of two competing 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

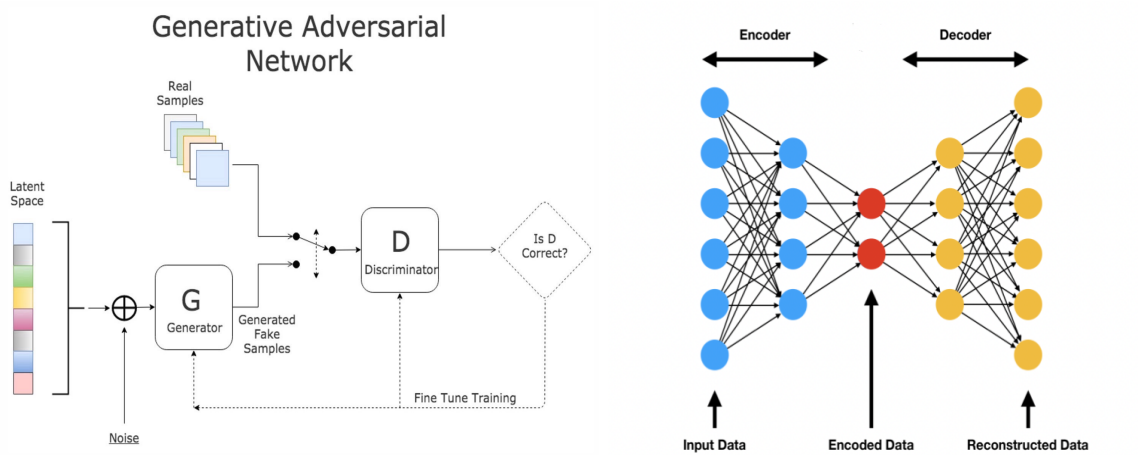


Figure 7: General Adversarial Network (GAN) architectures (left). Autoencoder architecture (right). Photos borrowed from https://medium.com/analytics-vidhya/a-review-of-generative-adversarial-networks-part-1-a3e5757a3dc2 and https://www.compthree.com/blog/autoencoder/ respectively.

labelled as "real". (This step is crucial as we want an error in the discriminator's 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 discriminator's loss is back-propagated through the network and used to update the weights of the generator. By competing against the discriminator, 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 and ensure that the embedded information is consistent across experiemntal days.

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. This pairing as called an *auto-encoder* and is often used to learn to find low dimensional features of high-dimensional input data (Figure 7*right*). In this framework, the auto-encoder plays the role of the generator. The first part of the auto-encoder, the encoder, will accept as input a vector of spike counts of length $N \times 5$ and output an encoded representation of the data of specified length k. The second part of the auto-encoder, the decoder, will accept as input the encoded k-dimensional representation of the data and output a reconstructed version of the original $5N$-length vector of spike counts. We will train a separate auto-encoder for each day of experimental data. The third network type 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 auto-encoder to encode and decode data from Day 1 of the experiment. Training will stop when the auto-encoder can accurately reconstruct the original data from the k-dimensional encoded representation.

Generator:

1. Initialize a new auto-encoder.
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 auto-encoder.

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. Interpretability 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. 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 global minimum. Early implementations of this network architecture have been successful in creating encoded data for the first five days or so, but any additional day leads to an overall blow-up of training accuracy. Work to remedy this issue and develop a stable training algorithm is still on going.

Both the PCA and GAN approaches provide us with a way to train a single network on data with variable input size. The PCA approach transforms the data before it is passed through a network, while the *GAN* approach makes the transformation a part of the networks learning. Both approaches have their advantages and disadvantages and work to implement them is on going. We hope to quanitfy the successes of each approach in more detail in the future.

4 Robustness of Residual Networks

In this section, we shift our focus from possible approaches to improve network flexibility to ones aimed at improving robustness of trained neural networks. To examine neural network sensitivity to perturbations, it is helpful to consider a specific type of feed-forward networks called residual networks. Residual networks (ResNets) were first introduced in 2015 to address the issue of exploding/vanishing loss function gradients and have since made training ultra-deep networks (networks over 1000 layers) pragmatic. In previous discussion, we considered a mapping, $y = f(x, \theta)$ from input data x to target y with parameters θ, 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

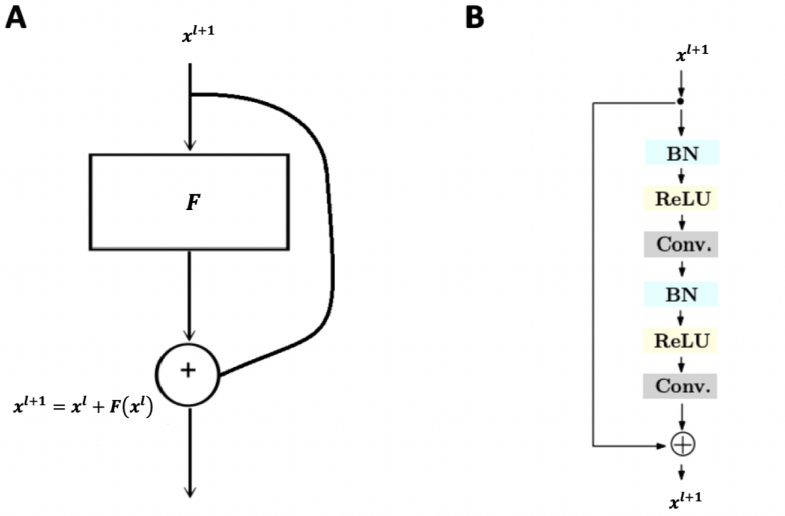


Figure 8: Panel A is a visualization of the skip connections in a residual layer. Here, x^l represents the input into the l-th layer and x^{l+1} its output. This is useful notation as residual networks are usually designed as a forward-propagating sequence of residual blocks. An example of the pre-activation 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"[16].

the input of a layer and conencts it directly to the output of that layer, bypassing the intermediate hidden layers of the model. A visualization of this skip connection is given in (Fig 8 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.

In the residual learning framework, we assume the 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 to perturbations in general [16].

4.1 Fragility under adversarial attacks

Although the residual learning framework provides a better conditioned learning problem for DNNs, these deep networks remain vulnerable to adversarial attacks. Developing robust algorithms is of significant importance to practical applications where security

and/or safety is crucial (autonomous cars, malware detection, facial recognition etc.). The 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 to force a degradation in the model's performance. Insight into how to perturb input images to yieled the highest chance of misclassification comes from attacking a related (*oracle*) model in a white-box fashion.
For this paper, we consider blind 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), exploit the loss function of an oracle model 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. 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. This makes it likely that the model, and other related models, will misclassify the adversarial image.
Some robust training algorithms have been designed to help minimize the risk of adversarial attack with significant success (see [11]). However, these robustly trained residual networks exhibit low accuracy on clean input images compared to their natural counterparts. This further serves to motivate the discovery of new regularization techniques to improve model robustness.

One option to improve network accuracy 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 8 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 θ^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 subscript BN are batch normalization layers, subcript C are convolutional layers and $\odot, \otimes$ represent that batch normalization and convolution operators respectively. The activation function σ is an application of the rectified linear unit (ReLU).
In a supervised learning task, each element $x \in T$ will have an associated label y. We can represent the full ResNet as

$$\begin{cases} x^{l+1} = x^l + F(x^l, \theta^l), \quad l = 0, \dots, L-1 \\ x_0 = x \\ \hat{y} = f(x^L) \end{cases} \tag{4}$$

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.

We can introduce a temporal discretization where "time" refers to the layer depth by letting $\Delta t_l = l/L$ for $l = 0, 1, \dots, L$ and $x_l = x(t_l)$, then Eq. 4 becomes

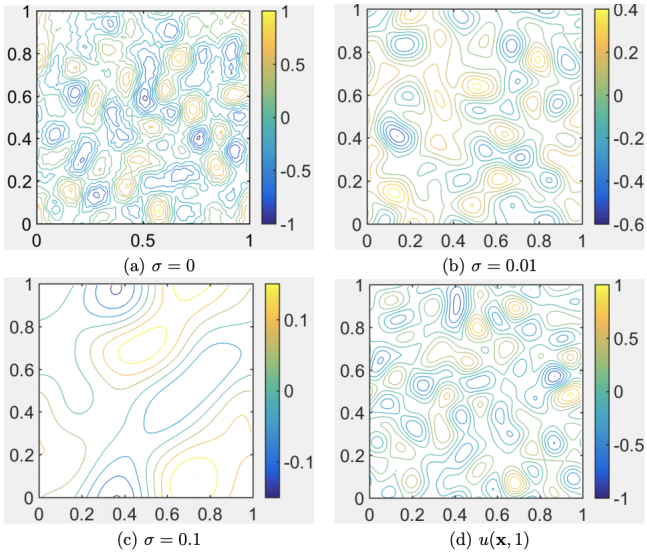
$$\begin{cases} x(t_{l+1}) = x(t_l) + F(x(t_l), \theta(t_l)), \quad l = 0, \dots, L-1 \\ x(0) = x \\ \hat{y} = f(x(1)). \end{cases} \tag{5}$$

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 approximation to the following ODE

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

Finally, if we let $u(x, t)$ be constant along Eq. 6 then we find $u(x, t)$ satisfies the following convection-diffusion equation

$$\begin{cases} \dfrac{d}{dt}(u(x(t), t) = \dfrac{u}{t}(x, t) + \tilde{F}(x, w(t)) \cdot \nabla u(x, t) = 0 \\ u(x, 1) = f(x) \end{cases} \tag{7}$$



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

Figure 9: (a),(b),(c) are solutions to the convection-diffusion equation (8) at $t = 0$ with different values of the diffusion coefficient σ. (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"[16].

Reformulating the residual network training problem into a transport equation problem allows us to consider the network classification problem in a more familiar setting. In particular, we can associate the level-sets of $u(x, 0)$ in the transport with the decision boundaries of the classification network. One approach to to regularizing the level sets of this partial differential equation is the addition of a noise term.

Adding a diffusivity term $\frac{1}{2}\sigma^2\Delta u$ to (7), where we have freedom to choose the diffusivity constant σ^2, is one way to add smoothness to the level sets of $u(x, 0)$. The new system becomes

$$\begin{cases} \dfrac{d}{dt}(u(x(t), t) = \dfrac{u}{t}(x, t) + \tilde{F}(x, w(t)) \cdot \nabla u(x, t) + \frac{1}{2}\sigma^2\Delta u \\ u(x, 1) = f(x) \end{cases} \tag{8}$$

. In Fig. 9, we show how the level sets in (8) change as we strengthen σ. We see the level sets become more regular as σ increases. Since the level-sets of $u(x, 0)$ represent the classification boundaries of the transport equation, this method of regularization should also improve the overall performance of the classifier [16].

Model	Acc_{nat}	Acc_{rob} (FGSM)	Acc_{rob} (IFGSM)
ResNet20	75.11	50.89	46.03
En_5ResNet20	80.52	58.92	51.54

Table 3: Table of accuracy for naturally (Acc_{nat}) and robustly (Acc_{nat}) trained residual networks with added noise.

4.3 Added noise improves robustness of ResNets

Using the above discussion as motivation, we improve the ResNet accuracy by adding an additional hyper-parameter into the network. After each residual block (Fig. 8 A), we add a noise term of the form $\mathcal{N}(0, \sigma^2 I)$ where $\sigma^2 = a\sqrt{\text{Var}(x^l + F(x^l))}$ and a is a tunable parameter. This choice of σ is motivated by [16]. Using these noisy residual blocks, we test the accuracy of two ResNet models on the CIFAR10 data. The first ResNet model is a single ResNet20 model which conists of (among other layers) nine residual blocks. The second network, En_5ResNet20 is an ensemble of five ResNet20 models which are jointly trained and averaged. We first train both models with no noise and use their natural accuracy on clean images and their robust accuracy under FGSM and IFGSM attacks as a baseline for accuracy performance. We then add noise into each of the skip connection of the residual layers and check for accuracy improvement. The results of this test are summarized in Table 3.

Implementation of the noisy ResNets shows that the added noise term does in fact improve over all accuracy, with accuracy under adversarial attacks improving by $2-3\%$. These results are consistent with those of [16] with minor discrepancies due to differences in computational hard/software. This exercise provides empircal evidence that adding noise is a viable way to improve robustness in neural networks, even under adversarial attacks. There is a natural statistical relationship between the input and output data. We propose that rather than have a noise term learned by the network, we could make a choice for *sigma* that preserves these statistics. We seek to develop a formal definition for this choice in the near future.

5 Discussion and Future Work

In this paper we presented two concrete examples of neural networks that suffer from a lack of flexibility and robustness. We discussed the challenges of designing flexible classification networks using experimental data recorded across many experimental trials. Specifically, we highlighted how differences in data collected on experimental trial days leads to both an implementation and a generalization issue for neural networks. We then presented a couple of possible approaches for designing networks that would perform consistently across the experimental trials.

On robustness of neural networks under adversarial attacks, we presented the formulation for viewing a neural network as a neural ODE (as outlined in [2, 16]). Specifically, we approximated the neural network decision boundaries by level-sets of the transport equation. We then introduced a diffusion term to regularlize the level-sets of the transport equation and used this analogy to add a tunable amount of gaussian noise into two different ResNet models. Our implmentation of the noisy ResNets showed that added noise improved the overall accuracy of the robust and natural model. These results were consistent with the findings of [16], though difficult to implement.
In the future, we seek to rigorously formulate a simplified, informed choice for the added Gaussian noise. We hope to exploit the known statistics of the training set to help achieve this. Additionally, it would prove interesting to marry the two ideas in this paper to create one more consistent and robust network model. After developing a working network model for the experimental data, and once our informed choice of noise is solidified, we hope to how a noisy trained network may help to mitigate the effects of experimental variability on the touch data classification task.

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] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG].

[3] 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, pp. 160–167. ISBN: 9781605582054. DOI: 10.1145/1390156.1390177. URL: https://doi.org/10.1145/1390156.1390177.

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

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

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

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

[8] 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].

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

[10] 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].

[11] Aleksander Madry et al. *Towards Deep Learning Models Resistant to Adversarial Attacks*. 2019. arXiv: 1706.06083 [stat.ML].

[12] Hernan Gonzalo Rey, Carlos Pedreira, and Rodrigo Quian Quiroga. "Past, present and future of spike sorting techniques". In: *Brain Research Bulletin* 119 (2015). Advances in electrophysiological data analysis, pp. 106–117. ISSN: 0361-9230. DOI: https://doi.org/10.1016/j.brainresbull.2015.04.007. URL: https://www.sciencedirect.com/science/article/pii/S0361923015000684.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.