Overview and TensorFlow Implementation of *Diet Networks: Thin Parameters for Fat Genomics*[1]

Leonard Strnad
ljstrnadiii@gmail.com

Department of Mathematical and Statistical Sciences

University of Colorado at Denver

Committee:
Audrey Hendricks, Ph.D
audrey.hendricks@ucdenver.edu
Erin Austin, Ph.D
erin.e.austin@ucdenver.edu
Dan Connors, Ph.D
Dan.Connors@ucdenver.edu

Abstract

The goal of this Master's project is twofold, to understand how to design a deep learning model with high dimensional data, $p >> n$, and implementing *Diet Networks: Thin Parameters for Fat Genomics*[1] in TensorFlow to replicate the results. This is an expository paper explaining the mechanics of *Diet Networks*, how to build the model in TensorFlow, and how to train the model with a Google Cloud Platform (GCP) instance with a GPU using Docker. Accompanying this expository paper are few tutorials on deep learning with TensorFlow. These tutorials will be given in a series of 2-3 talks based on `https://ljstrnadiii.github.io/` that acts as a good resource for people interested in Deep Learning, Docker, and TensorFlow.

Introduction

The motivation behind *Diet Networks: Thin Parameters for Fat Genomics*[1] is to build a better genetic ancestry prediction model using genomic data and deep learning. Most current approaches use projection techniques such as PCA.[2] Predicting ancestry is important when performing genomic based inference on an individual.

Diet Networks uses deep learning to build a more robust model for ancestry prediction. Most machine learning algorithms tend to suffer from very high dimensional data. Other methods avoid this problem by projecting the very high dimensional data into a much lower dimensional manifold using linear methods. Neural networks or deep learning give us the ability to essentially non-linearly project data into a much lower dimensional manifold. The main concern, however, is that the dimension of genomic data and number of observations tends to differ by three orders of magnitude. In order to deal with this problem, *Diet Networks* uses an auxiliary network that learns an embedding on the transpose of the data. The output of this auxiliary network is then used to construct the first weight matrix in such a way that the learnable parameters of this method are four orders of magnitude fewer. The benefit of this is that learning is faster and there is less vulnerability to overfitting the model.

This paper reviews the architecture and methods discuss in *Diet Networks* and discusses the details of an implementation [1] in TensorFlow that corresponds to this paper. Model selection is used to choose which model performs the best. The model selection process considers models built from different activation functions, weight initialization, and optimizers. In addition to the implementation, there are corresponding tutorials [2] that help introduce newcomers to deep learning and TensorFlow using Docker as a TensorFlow working environment.

Neural Network Architectures

Neural Networks are biologically inspired learning algorithms. They were first introduced in the late 50s by an American psychologist, Frank Rosenblatt.[3] They have come a long way and are now considered some of the most powerful and robust learning algorithms. They are used in the context of supervised and unsupervised learning. In addition to the supervised setting of classification and regression, they can even be used in an unsupervised context from complex distributions. Generative Adversarial Networks[4] and Variational Autoencoders[5] are an example of these structures. This section will discuss the strength and flexibility of neural networks and how they are constructed.

Neural networks can be constructed in many ways. They can be represented as a computational graph that is directed and acyclic. If the network architecture is acyclic, we call that network structure a feed forward network. There are also cyclic architectures, which are referred to as recurrent neural networks and were introduced in the 80s by John Hopfield.[6] These are not the focus of the paper and we focus only on feed-forward networks.

[1] https://github.com/ljstrnadiii/Dietnet
[2] https://ljstrnadiii.github.io

Figure 1: (Left) A single-layer Neural Network representation of a Linear Discriminant function. (Right) A single-layer Neural Network representation of multiple linear discriminant function. The index runs from 0 to p which implies $p+1$ inputs and weights per hidden node. If we let p be the dimension of the data, then x_0 and w_0 can be multiplied together to generate the bias term. In order for this bias term to be constant, we let $x_0 = 1$. Furthermore, the index of the output layer starts at 1 to emphasize there are c classes and there is no bias term.

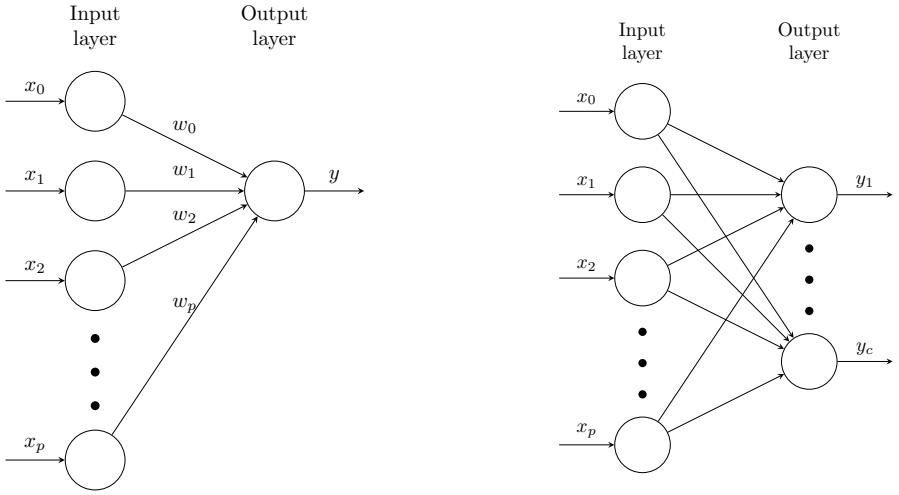
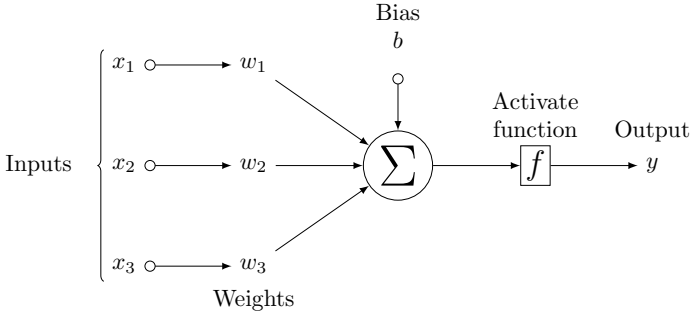


Figure 2: A detailed diagram of the computational flow of a node in a neural network. The dot product of x and W is passed to an activation function, f, and yields an output, y. Note: the bias term, b, and its weight, w_0, get multiplied also.



Linear Regression and Softmax Regression

This section assumes familiarity with linear regression and softmax or multinomial regression. Within the set of neural network architectures we can create a linear regression model. It is insightful to review this architecture as it will provide a more intuitive understanding of neural networks, how they are represented and mathematically defined. The left image in figure 1 is a representation of an architecture that takes p dimensional input x, has weights $w_1, ..., w_p$ and an output y. Let's be a bit more explicit about what is happening in the final node that outputs y. Figure 2 provides a diagram of what is occurring in this computational node.

Each dimension of x is multiplied by a corresponding weight, w_i, then these products are summed and passed to a function, f, to generate an output, y. This function is often referred to as the activation function. Activation functions are functions applied to the input of a node to define the corresponding output. There are many different activation function used including the rectified linear unit (ReLU),[7] hyperbolic tangent (tanh), sigmoid, softmax, radial basis, and identify function among many others. Each one has its own strength. The literature of neural networks tends to assume familiarity with these activation functions and often refers to them by their shortened names.

The diagram in figure 2 only displays x with dimension 3, but the idea extends to dimension p. Alternatively, we can represent the diagram as a dot product of x and w passed to an activation function, f:

$$y = f(x^t \dot{w} + b) \tag{1}$$

In the context of linear regression f is the identity function. The other term, b, is called the bias term and is used to shift the corresponding hyperplane. One perspective of neural nets is that we want to construct a hypothesis space of functions that contains a good approximation to our underlying distribution or function we are estimating. In the context of linear regression, we must find the parameters w and b that construct our optimal hyperplane to fit our linear data. The w parameter gives us flexibility in the slope of our hyperplanes and the bias, b, gives us the ability to shift it around to fit.

Next, we want a framework to learn these parameters. Instead of assuming our error is gaussian as assumed in standard linear regression, we can assume that our input is gaussian and find the parameters using maximum likelihood estimation. Since maximizing the likelihood gives the same critical points as maximizing the log likelihood, we can maximize the log likelihood. Recall that the log likelihood brings the L2 norm down from the exponent in the gaussian distribution, which lends itself to a mean squared error loss function. This is why the L2 loss is used in regression; it is a consequence of maximum likelihood.

In regression, the objective is to find w s.t.

$$w* = \arg\min_w \sum_{i=1}^{N} (y'_i - x_i^t w)^2 \tag{2}$$

or in matrix notation

$$w* = \arg\min_w (\mathbf{y} - \mathbf{X}\mathbf{w})^t (\mathbf{y} - \mathbf{X}\mathbf{w}) \tag{3}$$

where the bias has absorbed into the weights by appending 1 to each x_i and y'_i is the observed responses making this a supervised learning task. Also, note that w is a vector in both cases.

The usual approach that is taught in linear regression is to solve for w or β analytically even though we are told that software does not actually solve the problem that way. The common procedure used with neural networks involves training the model by finding the optimal weights and bias using an optimizing procedure called back-propagation.[8] This is an iterative approach to learning parameters. There are many versions of back-propagation methods, but the main idea is that it uses the chain rule on each nodes activation function to compute the gradient and iteratively walks the parameters towards a solution. Many back-propagation algorithms have

been proposed. Modern back-propagation algorithms use an adaptive learning rate. In short, the term 'adaptive' comes from the idea the learning rates decreases as time iterations increase. This means that smaller and smaller steps are taken in the direction of steepest ascent or descent. These methods are very robust and tend to find local optima better than the typical fixed learning rate methods. These algorithms or training procedures are also referred to as optimizers. They are called optimizers because they are the algorithm used to optimize the objective or loss function.

Softmax Regression is a generalization to logistic regression. The objective is to predict the class of x. Again, this section assumes familiarity with softmax regression or multinomial regression. The idea is to demonstrate how we can perform softmax regression with neural networks by discussing the architecture, mathematical representation and parameter learning procedure.

For an arbitrary dimensional vector x we would like to output probabilities of belonging to a particular class. The neural network structure that corresponds to this is the right diagram in Figure 1. The outputs will be probabilities for each possible class and we determine the class of x by choosing the class with the maximum probability. The difference between this architecture and the last is that there are many more weights to learn and multiple outputs. In the regression example, we had a vector for weights to learn for the output node. In this case, we have a vector of weights for each output node. This implies that we have a matrix of weights to learn, $\mathbf{W}$, that is of size $(p \times c)$ where c is the number of classes. Furthermore, our activation function, f, is taken to be the softmax function defined as

$$\sigma(\mathbf{z})_j = \frac{\exp^{\mathbf{z_j}}}{\sum_{i=1}^{c} \exp^{\mathbf{z_i}}} \tag{4}$$

We define $\mathbf{z}_j$ by multiplying an observation, x, which is a $(p \times 1)$ with the weights that point to the j^{th} node in the output layer:

$$\mathbf{z}_j = x^t w_{(j,\cdot)} \tag{5}$$

A geometric interpretation is that the input, $\mathbf{z}_j$, to the activation of the j^{th} node in the output layer constructs a hyperplane. There is then a hyperplane that corresponds to each class and it is mapped to a feature space by the softmax function. That is, these hyperplanes get mapped to a "sigmoidal" plane that is normalized such that for a given x the outputs sum to one. This gives us the ability to interpret the output as the posterior probability of the class given the observation. The basic idea is that we learn a weight matrix,$\mathbf{W}$, which constructs discriminant hyperplanes, such that when mapped by the softmax it generates posterior probabilities of being in a particular class.

Similarly, we use maximum likelihood estimation to estimate the parameters, $\mathbf{W}$. Again, for simplicity we absorb the biases like before. The multiclass problem lends itself to a multinomial distribution that is used to derive a likelihood function. We then arrive at the cross-entropy loss function. Again, this section assumes familiarity to softmax regression. The optimal parameters $\mathbf{W}$ are also found using back-propagation.

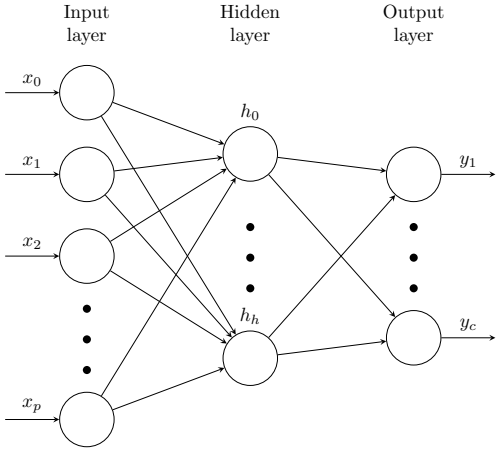
This approach works very well when the data is linearly separable and when the dimension of the data, p, is not too large. As p increases we have more parameters to learn, which lends itself to the vulnerability of being overfit. Furthermore, when data is not linearly separable we may have to use a different approach. The next section discusses a broader class of neural network architectures that include hidden layers as depicted in Figure 3.

Hidden Layers and Representation Learning

A very prominent theme in advanced statistical methods or machine learning is the idea of using kernels or activation functions to map the data into a new feature space. The goal is to learn a linear combination a new coordinate system or feature space to perform regression or discrimination. Mapping data into a new feature space can help learn a linear representation or linear discriminant function in this new learned space. This approach is used in successful statistical methods such a Support Vector Machines, Kernel Ridge Regression, Boltzman Machines, Gaussian Processes, and many others. The underlying method is often referred to as the kernel method. The resulting feature space is a consequence of which kernel or activation function is used.

The goal behind neural networks is to construct a large hypothesis space of functions that contains a good approximation to the underlying function that represents the deterministic behavior of the underlying process that generates the data. The free parameters that construct this hypothesis space are the weights that are learned. Now, consider a neural network with a hidden layer as depicted in Figure 3. Each of the nodes in the hidden layer has an activation function. Each edge also has an associated weight. We see there are then two weight matrices to learn: one from the input to the hidden and the other from the hidden layer to the output.

Figure 3: An example of a neural network with one hidden layer. There are two layers of weights. One between input and hidden and another between hidden and output.



Similarly, the nodes in the hidden layer can be represented by Figure 2. However, instead of the output of y we output h_i for each node that also has a corresponding weight. If we use a generic activation, ϕ, we can express the output of each node as

$$h_j = \phi(x^t w_{(j,\cdot)}) \tag{6}$$

and if we use the same activation function for each node in the hidden layer we can represent the hidden layer as a vector

$$\mathbf{h} = \phi(x^t \mathbf{W}). \tag{7}$$

We can now think of the elements of $\mathbf{h}$ as new coordinates for the observation x. In other words, $\mathbf{h}$ is a new representation of x mapped to a feature space determined by $\mathbf{W}$ and the activation function. Again, the bias is absorbed into x and $\mathbf{W}$ as before. There is also another matrix of

weights between the hidden layer and the output. Let's call this matrix $\mathbf{V}$. This matrix has size $(n_h \times c)$ where n_h is the number of hidden nodes and c is the number of output nodes. Furthermore, let's assume that the final activation function for the output is g. Then we have that our hypothesis space of functions can be represented as

$$\mathcal{H} = \{f(x) | f(x) = g(\phi(x^t \mathbf{W})^t \mathbf{V}), \mathbf{W} \in \mathbb{R}^{p \times n_h}, \mathbf{V} \in \mathbb{R}^{n_h \times c}\} \tag{8}$$

where f can be a vector function if $c > 1$. The final activation function, g, operates on a linear combination of this new representation of x. The idea behind finding a new representation is to vary $\mathbf{W}$ and $\mathbf{V}$ before passing it to a pointwise application of the activation function. The back-propagation procedure will not be able to exhaust all possible $\mathbf{W}$ and $\mathbf{V}$. Thus, the learning procedure only searches a subset of this hypothesis space of functions. The choice of the optimization or back-propagation algorithm often leads to different results. Some algorithms are better at exploring this function space than others. Getting stuck in local optima is one of the main concerns in training a neural network and this is the reason it can be important to use a few different optimizers when considering different models for model selection.

If we have a non-linearly separable data set and the hidden layer has found a new representation such that the data is linearly separable in that new feature space, then we can let g be the softmax function and essentially perform nonlinear softmax regression. Parameter estimation using maximum likelihood is similar as before in that we can assume the hidden representation is either normal, which leads to a mean squared error loss, or multinomial, which leads to a cross entropy loss to minimize.

A remarkable theorem that demonstrates the power of neural networks is called the universal approximation theorem.[9] It essentially states states that a neural network with a finite number of hidden units in a single hidden layer with a sigmoid activation function is capable of approximating any continuous function on a compact subset of $\mathbb{R}^n$. However, it is not always useful to simply increase the number of hidden nodes. Increasing the number of hidden nodes is the same as increasing the dimension of the feature space. One of the main assumptions made in machine learning is the manifold assumption. The manifold assumption is the assumption that the data approximately lie on manifold of much lower dimension than the input space. A more appropriate method to increase the flexibility of the model is to add more hidden layers. This is the idea that puts the "deep" in deep learning. Usually, the depth refers to the number of hidden layers and width refers to the number of hidden nodes in each layer.

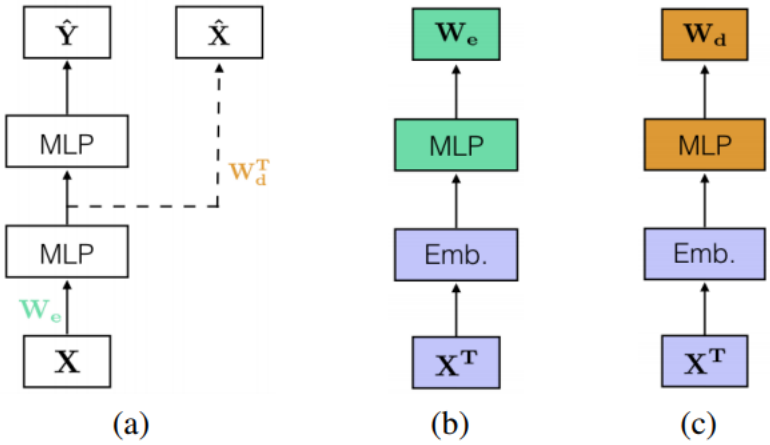
The immediate concern with the single/multiple hidden layer model discussed is the number of learned parameters. The weight matrix $\mathbf{W}$ and $\mathbf{V}$ are of size $(p \times n_h)$ and $(n_h \times c)$. There is an obvious concern if p, the dimension of the data, is large. Data that exhibits large p could be image data, audio data, genomic data and more. Convolutional Neural Networks are often used for image and audio data. Hinton demonstrated some of the first success with convolutional nets and image classification.[10] Genomic data, however, does not have the obvious spatial structure that images have, which motivates convolutional neural nets. Additionally, genomic data tends to be very high dimensional. As expressed in the previous section, the data we would like to work with has about 120,000 dimensions and only 2500 observations. *Diet Networks*[1] proposes a way to reduce the number of free parameters. This has many benefits including being less prone to overfitting and having fewer number of parameters to save for inference. The next section

reviews this method.

Diet Networks

Diet Networks: Thin Parameters for Fat Genomics[1] has focused on the problem of reducing the number of free parameters in the context of genomic data. There have been other approaches that focus on reducing the number of free parameters in a similar way including Bengio et al., 1991; Schmidhuber, 1992; Gomez & Schmidhuber, 2005; Stanley et al., 2009; Denil et al., 2013; Andrychowicz et al., 2016. However, these approaches did not focus on genomic data. The goal of *Diet Networks* is to predict ancestry at the subpopulation level, which has 26 regions. In the title, "thin parameters" refers to the number of free parameters and "Fat Genomics" refers to the large dimension of genomic data. As discussed before, the genomic data focused on here is 2500 observations each of which have dimension of approximately 120,000. As discussed in the previous section, we see that the first layer in a neural network would then have to have a weight matrix of size $(p \times n_h)$, where n_h is the number of nodes in the first hidden layer. If we consider n_h to be 100, then the number of free parameters just for the first layer would be approximately 12,000,000. Although neural networks can do a good job projecting very high dimensional data into a lower dimensional manifold in a nonlinear way, a large number of parameters can cause a few problems. These problems include taking longer to converge as the parameter space is much larger and can lead to overfitting and poor generalization. *Diet Networks*[1] proposes a method to significantly reduce the number of free parameters. First, we review the data and the corresponding preprocessing.

Figure 4: (a) is the discriminative network; (b) is the encoding auxiliary network; (c) is the decoding network. Image taken from Diet Network: Thin Parameters for Fat Genomics[1]



Data: 1000 Genomes Project

The goal of the 1000 genomes project[11] was to find the most genetic variants that occur with a frequency of at least 1% in the populations studied. They claim it was the first project to have sequenced the genomes of a large number of people from many different populations. Sampling from many subjects across many regions along with genotype imputation gave the project the ability to determine a sample's genotype. In addition, each subject has been sequenced with a coverage of 4X. Coverage in DNA sequencing is the number of reads for a specific nucleotide in the reconstructed DNA sequence. The main idea is that the more coverage there is in reconstruction the fewer sequencing errors there will be. The final dataset generated from this sequencing is a sequence of variations that occur at a specific place in the genome. An additional requirement of these variations, called Single-Nucleotide Polymorphisms (SNPs), is that they have to occur at some frequency within a population. This allows the variations to be somewhat unique to the population even though the frequency is as low as 1%. These SNPs are represented as a ternary inputs and are prepared as one-hot encodings for the model.

The data are 315,345 variants or SNPs preprocessed such that there is a minor allele frequency of at least 5% in 3450 individuals. There are 26 populations. *Diet Networks* plots the counts of each population to show they are roughly of equal proportion. This labeled data is often referred to as the phenotypic data. They also only include SNPs that are in approximate linkage equilibrium such that $r^2 < .5$. This leads to a subset of SNPs that are less correlated. The paper also breaks the data up for training by using cross validation with 5 folds. This particular implementation, however, only uses a train, test and validation approach. In *Diet Networks*, they specify the exactly how to download and preprocess the data using Plink.[12]

The Structure of *Diet Networks*

In order to mitigate the problem of having to learn a large number of parameters, *Diet Networks* proposes to use two auxiliary networks for predicting the weight matrix of the first layer. Figure 4 shows the general structure of the Diet Network. The middle network structure outputs a matrix, $\mathbf{W}_e$, that is the weight matrix of the first layer in the discriminative network shown on the left. The MLP suggests that we can have a multiple layer perceptron at each of these blocks in the diagram. This means there can be multiple hidden layers in these blocks. For simplicity, we will assume that each of the MLPs have a single hidden layer and have n_h nodes.

The middle network in Figure 4 is referred to as the encoding auxiliary network. The idea is to learn an embedding on the feature space of the data, $\mathbf{X}$. That is, the embeddeding layer is a new representation of $\mathbf{X}^t$ or of the feature space of the data. This embedding can be constructed or learned end-to-end. The paper proposes a few different methods to build an embedding layer. One method includes simply using a hidden layer to learn a representation of the feature space. This method can be learned in an end-to-end training fashion. This means we can learn this embedding when training the whole network. The method that will be the focus of the later discussed implementation is the per class histogram. This embedding allows us to use a priori knowledge of the data.

The per class histogram embedding is a matrix whose elements represent the frequency of the support of each SNP for each class. Recall that each SNP can have three values: $\{0, 1, 2\}$. The embedding matrix, $\mathbf{E}$, is a $(p \times 78)$ matrix where each row's elements are the frequency of $\{0, 1, 2\}$ occurring for each of the 26 classes, which gives 78 columns. This is a representation of the co-occurrences of the feature space of the data. This is the embedding that shows the most success.

This embedding matrix is constructed from the training and validation data and not the test data.

After the embedding layer in the encoding auxiliary network there is a MLP. Let's assume that this MLP just contains a single hidden layer. Since this hidden layer takes the embedding layer as input that implies that the weight matrix, $\mathbf{W}'_e$ of this layer is of size $(78 \times n_h)$, where n_h is the width of the hidden layer. After this MLP layer, the encoding auxiliary network outputs a matrix, $\mathbf{W}_e$. Therefore, when we pass the embedding matrix, $\mathbf{E}$, to the MLP layer we will get

$$\mathbf{W}_e = \mathbf{E}\mathbf{W}'_e \tag{9}$$

and we know that $\mathbf{E}$ and $\mathbf{W}'_e$ have sizes $(p \times 78)$ and $(78 \times n_h)$. This implies that the output matrix, $\mathbf{W}_e$ is of size $(p \times n_h)$. Assuming that the first hidden layer in the discriminative network also has n_h hidden nodes, this resulting matrix can then be used as the weight matrix of the first hidden layer in the discriminative network. Notice, however, that there are only $78 \times n_h$ free parameters instead of $p \times n_h$. Since the dimension of the genomic data used is $p = 120,000$, this method significantly reduces the number of free parameters. For example, if $n_h = 100$ then we have reduced the number of free parameters at this first hidden layer in the discriminative network from 12,000,000 to 7800.

The right network in Figure 4–the decoder auxiliary network–is used to predict a weight matrix to help reconstruct the data. The paper does not specify why it is helpful to add a reconstruction network, $\hat{\mathbf{X}}$, but simply demonstrates that it gives better results. If we assume that the first MLP in the discriminative network is a single hidden layer with n_h nodes then its output will be of size $(n \times n_h)$. Therefore, in order to output $\hat{\mathbf{X}}$, which is size $(n \times p)$, the reconstruction weight matrix $\mathbf{W}^t_d$ should be $(n_h \times p)$. This further implies that the output of the decoder auxiliary network, $\mathbf{W}_d$, should be $(p \times n_h)$. Similarly, the decoder auxiliary network learns a weight matrix, $\mathbf{W}'_d$ such that

$$\mathbf{W}_d = \mathbf{E}\mathbf{W}'_d. \tag{10}$$

This is the same size as the output of the encoding auxiliary network. In fact, the paper suggests these two matrices can be the same. This option would also save a lot of the memory in weight matrices in addition to greatly reducing the number of free parameters as with the encoding auxiliary network.

The training process involves minimizing two loss functions. The classification network lends itself to the cross entropy loss function and a common loss function for reconstruction is the mean squared error. Therefore the total loss minimized is

$$\mathcal{L} = \mathcal{H}(y, \hat{y}) + \lambda ||\mathbf{X} - \hat{\mathbf{X}}||^2 \tag{11}$$

where $\mathcal{H}$ is the cross entropy loss function and λ is a tunable parameter.

Results

In the paper, the architecture of the network includes only a single hidden layer in each of the MLP blocks. So, there are in total two hidden layers between the input $\mathbf{X}$ and the output $\mathbf{Y}$. Furthermore, there is only a single hidden layer in both of the auxiliary networks. Each of these hidden layers have 100 units. Additionally, the model was trained using stochastic gradient descent with an adaptive learning rate and norm constraints were places on the gradients. The models were trained with and without reconstruction. Also, 5 fold cross validation was used for the results of the *Diet Networks* paper.

The model and embedding focused on in the TensorFlow implementation is the per class histogram embedding with reconstruction. The results presented in *Diet Networks* suggest that the per class histogram embedding with reconstruction has much less variance in the model and lowest mean misclassification than the other embeddings considered. They report an accuracy of $92.59 \pm .45\%$. The accuracy reported is the mean frequency of correct genome classification $\pm$ one standard deviation of the 5 fold cross validation test data. In addition to those results, the per class histogram has the lowest number of free parameters.

The details of how the model was trained to get these results are left to the reader to determine. The learning rate, specific optimization algorithm, weight initialization, regularization, and activation functions used are not specified in the *Diet Networks* paper. The choice of these hyperparameters is critical for convergence and getting the model to learn. If the learning rate is off by a decimal the model may have a very difficult time converging. If the weights are initialized without much consideration convergence may take much longer or the model may get stuck in local optima. Lastly, different activation functions make the model expressive in different ways. Choosing these specific details is critical in building a model that will be optimal. The section on the TensorFlow implementation discusses these details and provides specific details of the model structure to duplicate the results of the per class histogram model with reconstruction.

TensorFlow and *Diet Networks*

Diet Networks does not discuss the specific details of their implementation. They specify the architecture that is discussed in the results section above. Although the details of the width, depth and general structure are discussed, many details are omitted. These details include specifying the activation functions used, weight initialization, learning rate, optimization algorithm, regularization, and whether to use batch norm or not. However, they do share [3] their implementation, but the model is built in Theano.[13] Another effort [4] has been made using TensorFlow to implement *Diet Networks*, from which this implementation took some inspiration. Before discussing the details of the implementation, we motivate the use of TensorFlow to build the deep learning model.

TensorFlow

The website for TensorFlow [5] makes a helpful introduction.

> TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

[3] https://github.com/adri-romsor/DietNetworks
[4] https://github.com/gokceneraslan/dietnet
[5] https://www.TensorFlow.org/

The ability to compute using a graphical processing unit (GPU) significantly speeds up training and inference time. TensorFlow has made it relatively simple to run the models on GPUs. Additionally, the Python API used to create these TensorFlow models is excellent and well documented [6]. TensorFlow also has libraries that abstract common operations so that users do not have to repeat themselves. There are abstractions for fully connected networks, convolutional networks and recurrent neural networks among others. They have also abstracted common features such as adding a regularization constraint on the weights of these abstracted layers, adding batch norm, specifying a distribution to initialize weights, etc. The main goal of TensorFlow is to make it easy for a researcher to make different architectures in a simple fashion to test and develop different models on top of a production grade framework. Google uses TensorFlow in their back-end and they even make it simple to take to production with TensorFlow Serving [7].

This paper is written alongside a TensorFlow implementation [8]. In addition to the implementation, there are tutorials at [9], which introduce TensorFlow concepts, discuss different neural network architectures and motivate the need for a model like *Diet Networks*. Additionally, there are setup instructions to set up a TensorFlow coding environment using Docker on a linux instance using Google Cloud Platform.

TensorFlow Implementation

This implementation uses two hidden layers in the discriminative network and each of the auxiliary networks. There is batch normalization[14] after the first layer in the auxiliary network, and a dropout[15] layer before the final fully connected layer. The first layer uses the ReLU activation function and the last is set up to consider ReLU or tanh for model selection. The discriminative network performs batch norm and drop out in the first layer similar to the auxiliary network. The final layer before the last softmax layer also has batch norm and dropout. The auxiliary and discriminative network are built to test multiple activation functions and weight initialization for model selection. A diagram of the structure is shown in figure 5.

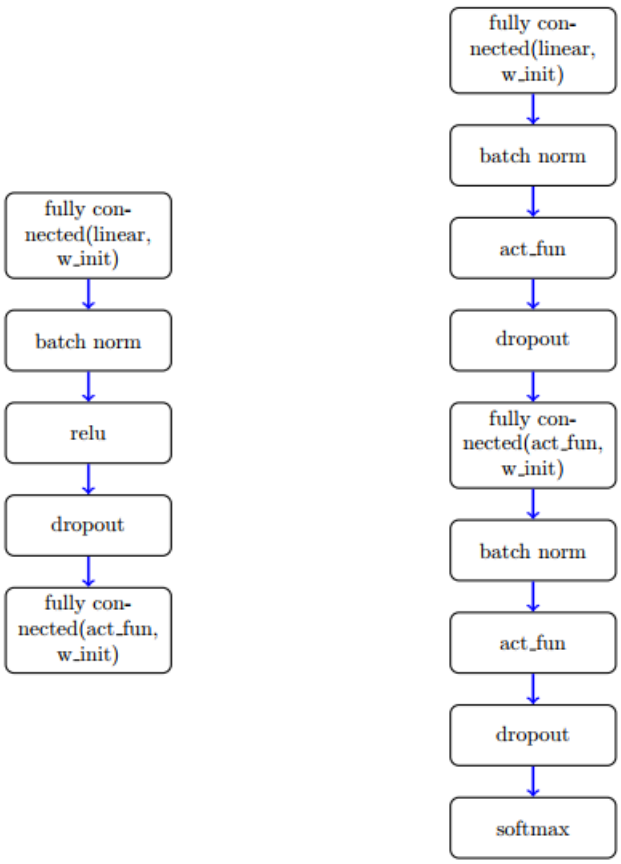
The approach to model selection in this paper is to train many models with the different configurations just discussed. As these models are training, the loss function and accuracy are saved and plotted. The loss function is the cross entropy between the correct and predicted labels. The accuracy is the frequency the model correctly classifies the ancestry that corresponds to the given genomic data. The accuracy is assessed on the train, validation and test data set, however, this review only discusses the testing data set accuracy.

Since the learning process is iterative, each model is a set of trained models at different steps or epochs. A very basic approach is to choose the model based on the accuracy performance on the test data set. For simplicity, the best model here is determined by the maximum test data set accuracy over all epochs or steps. This is a crude approach to ensure the model is not overfit.

In order to find a good model, we train multiple models with different activation functions, weight initializations and learning rates. TensorFlow offers a visualization tool for learning called tensorboard [10]. Tensorboard is a webapp that allows a researcher to train multiple models and visualize the loss, gradient, accuracy, histogram of weights over epochs, etc. It allows the

[6] https://www.TensorFlow.org/api_docs/
[7] https://www.TensorFlow.org/serving/
[8] https://github.com/ljstrnadiii/DietNet
[9] https://ljstrnadiii.github.io/
[10] https://www.TensorFlow.org/get_started/summaries_and_tensorboard

Figure 5: The left graph provides the detailed structure of the auxiliary network. The right graph describes the discriminative network. Everywhere there is w_init (weight initialization) and act_fun (activation function) we leave open for model selection.



researcher to perform a hyperparameter search to see which model performs the best. Deep Learning models get a lot of attention, but training them successfully and performing model selection can be arduous. This section demonstrates model selection using tensorboard.

The specific combinations of models considered here includes weight normalization using the uniform and normal distribution; a standard deviation of .1 and .01; tanh and ReLU activation functions, the Adam[16] and RMS optimizers; and learning rates of .001 and .0001. This will generate 32 models for consideration. In addition to training all of these models, experimentation has lead to certain regularization values in gradient clipping and L2 regularization of the weights. Ideally we would add this into our model selection process, but computational resources are limited. Lastly, the final model selection procedure usually includes choosing the correct epoch to save the model. This paper does not cover the best epoch to choose, but it is worth mentioning as it is an important part of model selection. Early stopping is a regularization method used to choose the best epoch for increasing generalization.

The accuracy on the test data set of the 32 models is shown as a function of a training step in the plot in Figure 6. The accuracy shown is the smoothed or moving average of the correct ancestry classification in the test data set. The x axis represents a step. Each step considers a batch of data and a sufficient number of steps or batches will present the model with the entire training data set. One pass on the entire training data set is known as an epoch. Batch training is the standard practice for training deep neural networks.

Figure 6: Accuracy of 32 different models on test data set. The models are constructed from a standard deviation of .1 and .01; normal and uniform weight initializations; tanh and ReLU activation functions; Adam and RMS optimizers; .001 and .0001 learning rates. The y-axis is the accuracy of the entire test data and the x-axis is a step in the iterative learning process.

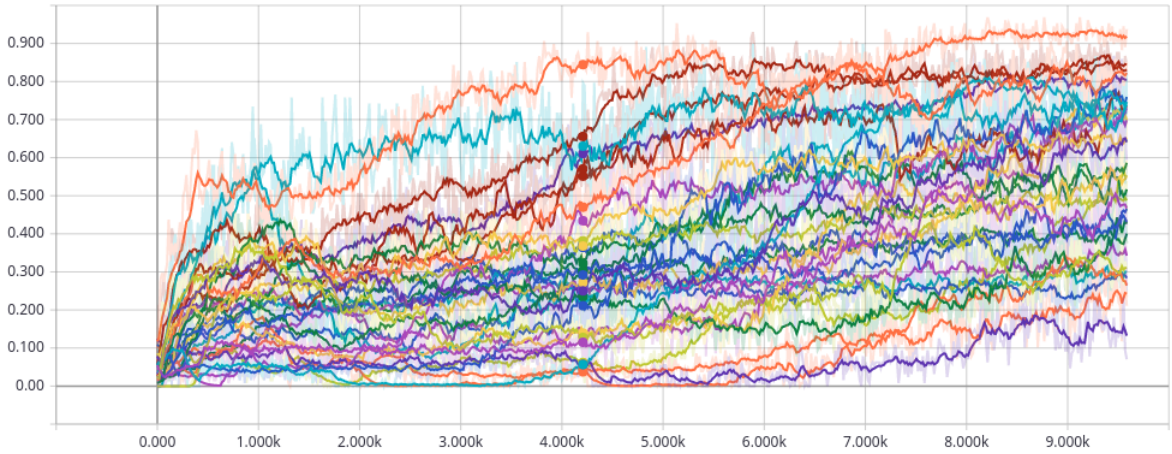
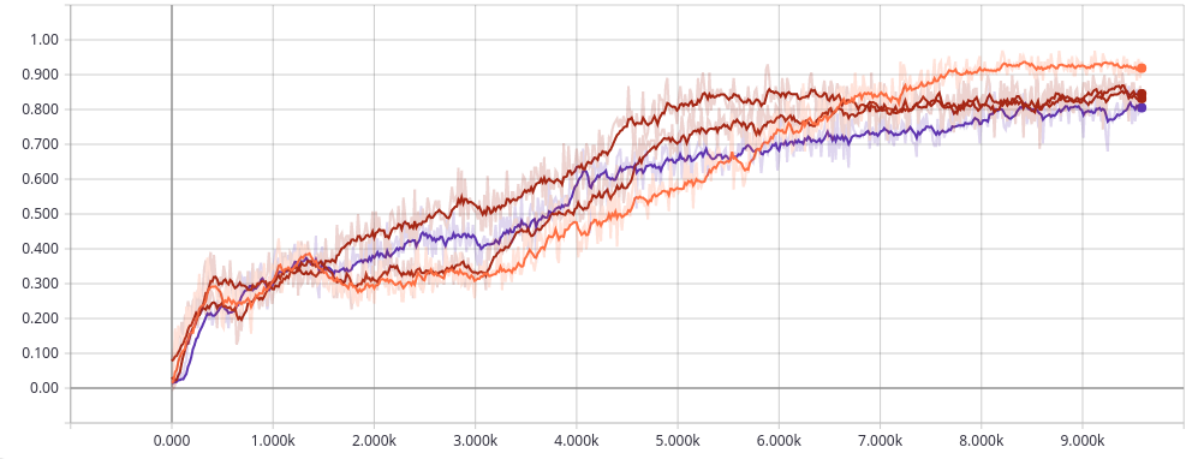


Table 1: Displays the best 4-subset of the first parameter search attempt. Max accuracy is the maximum accuracy over all the steps or epochs.

weight init	sd	activation function	optimizer	learning rate	Max Accuracy
normal	.01	tanh	Adam	.0001	.93
normal	.1	tanh	Adam	.0001	.86
uniform	.01	tanh	Adam	.0001	.84
uniform	.1	tanh	Adam	.0001	.83

Figure 7: Accuracy of the top 4 models from Figure 6. It is based on a smoothed plot or moving average of the accuracy. The y-axis is the accuracy on the test data and the x-axis is the step number.



The plot in Figure 6 is actually interactive in tensorboard. We can select subsets to hide all other models. The top four models that achieved the highest test data set accuracy are listed in Table 1. A common trend is that the Adam optimizer with the tanh activation function is superior to the RMS optimizer and ReLU. This exemplifies the importance of optimizers.

The best subset shown in Figure 7 suggests that the combination of using the tanh activation function with the Adam optimizer with a learning rate of .0001 is a good place to start. Considering the best performing model in the *Diet Networks* paper achieved 7.44% misclassification error, this implementation seems to achieve similar results and has successfully reproduced one of *Diet Networks* results using the histogram embedding method. One important thing to note is that the test error reported in this implementation is likely to be an overestimate of the generalization error. This implementation's maximum test accuracy does match the results of *Diet Networks*, however, they are likely to have a more accurate estimate of the generalization error since they are implementing 5-fold cross validation.

Future work would include using cross validation like *Diet Networks* to get a sense of variance of the model and increase generalization. Additionally, it would be worth while to explore other models with varying regularization parameters, learning rates that are closer to .0001, and drop out probabilities.

Conclusion

Deep learning is a powerful machine learning algorithm. There are many variations that extend from unsupervised learning to sequence models. The universal approximation theorem proves the strength of these algorithms. The application of deep learning to the genomic data problem presented in *Diet Networks* is very promising. There are many different frameworks to build deep learning algorithms, but this paper reviews the success of an implementation in TensorFlow. The benefit of TensorFlow is that if offers a visualization tool, tensorboard, to display learning. It

makes model selection very simple, however, finding the right class of models to train can be challenging. Figure 6 shows how many models are unsuccessful. Considering each of these models takes about an hour to train on a Tesla K80 GPU, model selection can be limited to resources.

Future research using this TensorFlow implementation should explore different learning rates and weight initialization using the Adam optimizer with the tanh activation function. Of course other models could be considered, but these are the models that seem to be the most promising in this implementation. Also, exploring different regularization parameters may lead to a model that performs better with generalization. The code for this TensorFlow implementation can be found here [11] and a series of tutorials and setup instructions of how to use Docker to host a TensorFlow coding environment can be found here [12]

[11] https://github.com/ljstrnadiii/DietNet
[12] https://ljstrnadiii.github.io

References

[1] Adriana Romero, Pierre Luc Carrier, Akram Erraqabi, Tristan Sylvain, Alex Auvolat, Etienne Dejoie, Marc-André Legault, Marie-Pierre Dubé, Julie G Hussin, and Yoshua Bengio. Diet networks: Thin parameters for fat genomic. *arXiv preprint arXiv:1611.09340*, 2016.

[2] Peristera Paschou, Elad Ziv, Esteban G Burchard, Shweta Choudhry, William Rodriguez-Cintron, Michael W Mahoney, and Petros Drineas. Pca-correlated snps for structure identification in worldwide human populations. *PLoS genetics*, 3(9):e160, 2007.

[3] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[5] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[6] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[7] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[8] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[9] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Nayanah Siva. 1000 genomes project, 2008.

[12] Shaun Purcell, Benjamin Neale, Kathe Todd-Brown, Lori Thomas, Manuel AR Ferreira, David Bender, Julian Maller, Pamela Sklar, Paul IW De Bakker, Mark J Daly, et al. Plink: a tool set for whole-genome association and population-based linkage analyses. *The American Journal of Human Genetics*, 81(3):559–575, 2007.

[13] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, 2016.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[15] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[16] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.