Predicting the Presence of Heart Disease

Greg Pennisi
The College of William & Mary
Williamsburg, Virginia
gfpennisi@email.wm.edu

ABSTRACT

Given a dataset containing a variety of heart-related medical readings from 180 anonymous patients, a multilayer perceptron was trained to predict how likely heart disease was present in the given patient. The first iteration of the project used a feedforward neural network, but later transitioned to a multilayer perceptron in order to keep the implementation as simple as possible, thereby maximizing the chance of building a useful network. Many challenges posed themselves, such as overfitting and feature representation, but were ultimately solved to an acceptable degree. The final version of the network is currently placed 202nd out of 929 entrants in the competition, and has a binary log loss of 0.4095. This value was obtained from submitting the results of the network into the drivendata competition.

INTRODUCTION

The source for both the data and inspiration for this project is the Machine Learning with a Heart dataset from drivendata.org. This dataset contains relevant parameters to assist in the task of predicting heart disease, such as the patient's sex and age. It also contains several medical readings for each patient, such as resting blood pressure, resting EKG results, and the patient's type of chest pain. The motivation for this problem stems from the fact that heart disease is the leading cause of death for both men and women in the United States. Furthermore, early detection is an extremely effective way to prevent heart disease-related deaths, so a model that can predict heart disease would be an extremely valuable life-saving tool.

While this was initially approached as a classification problem, the final implementation transitioned to regression, resulting in non-binary outputs ranging from 0 to 1 in a unimodal space.

Each input vector contains thirteen relevant data points, along with a value marking whether or not the patient has heart disease. Supervised learning was the obvious approach, given that the dataset is paired with the correct classification for each input vector.

The components of learning consisted of the data set (each patient, their associated medical readings, and whether or not they have heart disease), the target function f which correctly predicts whether or not a patient has heart disease given the input vector of medical readings, the learning algorithm, and the hypothesis set. Given the relatively straightforward nature of the problem, a multilayer perceptron was used for implementation. A type of gradient descent called Adam[1] was used as the learning algorithm, since PyTorch was used to build the neural network. PyTorch contains a large variety of learning algorithms, and after much testing, Adam appeared to be the most useful algorithm for this project.

DATASET

The dataset consists of 180 input vectors. Each of these input vectors contains a randomly generated patient ID (to maintain anonymity), thirteen features, and a binary label where 0 indicates a lack of heart disease, while 1 marks its presence. The full list of features taken directly from the project description[2] is shown here:

(1) `slope_of_peak_exercise_st_segment` (type: int): the slope of the peak exercise ST segment, an electrocardiography read out indicating quality of blood flow to the heart
(2) `thal` (type: categorical): results of thallium stress test measuring blood flow to the heart, with possible values `normal, fixed_defect, reversible_defect`
(3) `resting_blood_pressure` (type: int): resting blood pressure
(4) `chest_pain_type` (type: int): chest pain type (4 values)
(5) `num_major_vessels` (type: int): number of major vessels (0-3) colored by flourosopy
(6) `fasting_blood_sugar_gt_120_mg_per_dl` (type: binary): fasting blood sugar > 120 mg/dl
(7) `resting_ekg_results` (type: int): resting electrocardiographic results (values 0,1,2)
(8) `serum_cholesterol_mg_per_dl` (type: int): serum cholesterol in mg/dl
(9) `oldpeak_eq_st_depression` (type: float): oldpeak = ST depression induced by exercise relative to rest, a measure of abnormality in electrocardiograms
(10) `sex` (type: binary): 0: female, 1: male
(11) `age` (type: int): age in years
(12) `max_heart_rate_achieved` (type: int): maximum heart rate achieved (beats per minute)
(13) `exercise_induced_angina` (type: binary): exercise-induced chest pain (0: False, 1: True)

As seen above, every feature contains a numerical value except for `thal`. During early stages of implementation, this feature was converted into a one-hot vector, but this resulted in the need for every other feature to be converted to three dimensions, so `thal` was ultimately dropped in order to ensure a working project could

[1] https://arxiv.org/abs/1412.6980
[2] https://www.drivendata.org/competitions/54/machine-learning-with-a-heart/page/109/

be made. Also, while not mentioned in the above list, each input vector also contained a unique patient ID in the form of a string. This ID was always dropped, as it was not needed for the overall design strategy. Instead of using IDs to pair inputs and their labels, simple indexing was used.

A third CSV file was also present, containing 80 unlabeled input vectors. This data set was only used to test out-of-sample error by running it through the network and submitting the results into the drivendata competition in order to receive a binary log loss score.

Data Preparation

The data was obtained in the form of a CSV file. The pandas[3] library was used to read in the CSV values. This decision was made in large part due to the ease of dropping specific features in-place using this format; a simple `.drop(paramters)` command on the respective data table allowed for quick testing of different drops.

Once the data tables were tailored to the desired set of features, the numpy[4] library was used to extract the values from the table. The reasoning behind this decision is that PyTorch ultimately needs the data to be stored in the tensor format before it can be passed to the network, and PyTorch is equipped with commands that allow for quick conversion from a numpy array to a tensor object. The resulting tensor object was then stored in a standard Python array. PyTorch appears to contain its own commands for the storage and iteration of tensors, but learning to work with those appeared to not be worth the effort, as iterating through a standard Python array was sufficient to get the network up and running.

As alluded to above, the input vectors themselves lacked their associated labels. All of these labels were stored in a separate CSV file, which was loaded in and processed very similarly to the steps described in the previous paragraph. The design of the overall code was built with this separation in mind. Since each input vector and its associated label was stored on the same line number in its associated CSV file, indexing appeared to be the easiest and fastest way to pair each input with its appropriate label. As a result, the training data and training labels were stored in two separate Python arrays, each containing 180 tensors, where `training_data[i]`'s corresponding label could be found at `training_labels[i]`.

The testing data was prepared using the same methods, and was only utilized when there was a need to check out-of-sample error. This was done via submission of the network's output into the contest.

Feature Dropping

Brief exploration was made into feature dropping. Due to the ease of in-place dropping with the pandas library, many input features were dropped in order to observe the changes in the in-sample error. Unfortunately, while several features (and combinations of features) were dropped, there was no discernible change in the in-sample error. As a result, all features (except for `thal`, explained above) were kept when training the final version of the network.

ARCHITECTURE

All iterations of this project were built using the PyTorch library, due to its approachability and Pythonic syntax. The initial version of this project was built as a Feedforward Neural Network. At its core, this decision was the result of an inadequate understanding of the various types of neural networks. While much time was spent adjusting this network, it was never able to generate any meaningful output; the closest it ever came was a prediction of 0 for every data point. After discussion with the TA, it was clear that a multilayer perceptron was capable of completing this task. Furthermore, as simplicity is a good rule of thumb when designing these networks, a multilayer perceptron was the clear choice, since this architecture was shown to be perfectly capable of solving this problem and making meaningful predictions.

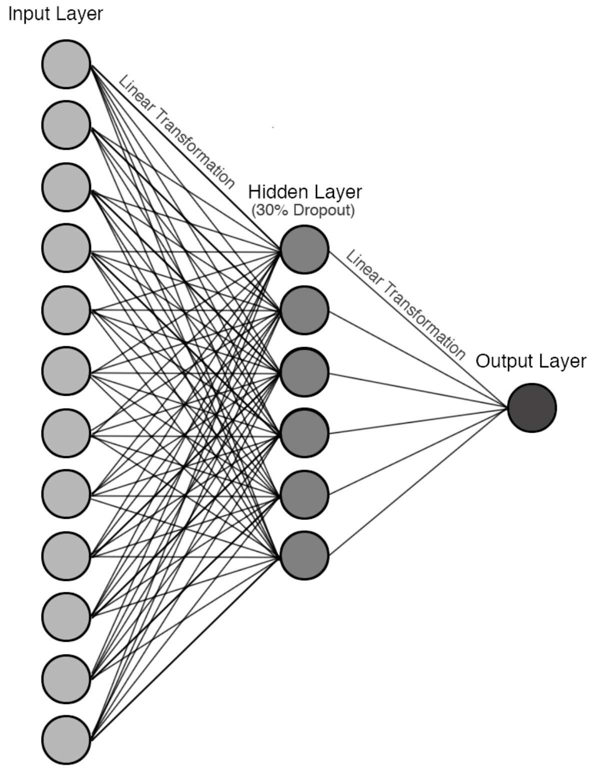

Figure 1: The final network architecture.

As seen in Figure 1, the final version of the network consists of an input layer of twelve features, one hidden layer condensed to six nodes, and an output layer condensed to one node. For each forward pass, linear transformations are applied in order to condense the twelve input values to six, and then those six values to one. The hidden layer utilizes the ReLU[5] activation function on the incoming data. Following this, a dropout rate of 30% is applied to the hidden layer, before those values are passed to the output layer using another linear transformation.

[3] https://pandas.pydata.org/

[4] http://www.numpy.org/

[5] https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning

Backpropagation

For backpropagation, the first implementation used a vanilla Stochastic Gradient Descent algorithm. However, in-sample error would routinely get stuck at a certain value, usually somewhere around 0.10. The hyperparameters learning rate and momentum were tweaked to try to fix this issue. The learning was initialized at 0.001 and was adjusted several times, with larger values resulting in errors or very high in-sample error, and lower values showing little changes of in-sample error, so the value 0.001 ended up unchanged. Intuitively, momentum appeared to be the better candidate to fix this problem, since a higher momentum could "propel" the algorithm out of these error "pits." After trying various values for momentum, however, no notable improvement was observed. Momentum was initialized to 0.4. Similar to the learning rate, high values of momentum showed very high in-sample error, while low values of momentum showed similar behavior to the original 0.4, with the exception of getting stuck at higher in-sample error values. After the exploration of these adjustments, and their various combinations, the network was still unable to reliably avoid falling into these error "pits." After changing the learning algorithm to Adam, however, the network no longer showed any signs of getting stuck at a certain value. The in-sample error would continue to decrease as more epochs were run, showing that these error "pits" were no longer an issue.

OVERFITTING

From the performance perspective, overfitting was by far the most difficult issue to grapple with. From the moment the network began running without errors, the in-sample error would routinely be driven down to values ranging from 5% to 0.0005%. These outputs immediately raised suspicion, as it was extremely unlikely that these error values accurately tracked out-of-sample error. This hypothesis was confirmed when a network with extremely low in-sample error was scored with a binary log loss that was greater than 1. A low in-sample error paired with a high out-of-sample error is a textbook example of overfitting, so it was clear that this issue would need to be addressed in order to improve the network to the point of generating useful predictions.

This overfitting issue was expected from the start, as the data set only contains 180 points, which is far toward the smaller side of the full spectrum of data sets used for machine learning. The most immediate cure for overfitting is typically to increase the size of the data set, but this was out of the question, as the full data set was already being used. Other methods of reducing overfitting were researched, such as dropout, batching, and changing the number of hidden layers and their size.

Number and Size of Hidden Layers

The knowledge gained from reading[6] about the best ways to choose the number and size of hidden layers was unable to be used to improve the network, as the original numbers used happened to fall right in the middle of the recommended ranges for a network being used to solve this particular problem.

Despite

Batching

Batching was researched briefly, but this pursuit was dropped due to the potential complexity of implementation. At the time of researching batching, the network was already in a working state, and producing reasonably useful predictions. A combination of aiming for architectural simplicity and fear of improper implementation lead to the decision to ignore batching and instead pursue other fixes for the overfitting problem.

Dropout

Dropout[7], put simply, is the process of temporarily "turning off" nodes in the hidden layer. On each iteration of training, each hidden node has the same chance of being "turned off" in that iteration. In PyTorch, this probability is defined upon creation of a `Drop` object, where the inputted decimal value is equal to the likelihood of a node being dropped (0 means no dropping, 0.5 means 50% likelihood, etc).

Dropout was fortunately very easy to implement, which allowed for efficient testing. Dropout was initialized with a value of 0.5, or 50%. Introduction of dropout resulted in unpredictable behavior; while the network would appear to work as intended on some runs, the majority of runs showed erratic behavior, with the in-sample error value randomly bouncing up and down, showing no signs of convergence. This value was adjusted to 0.7, which simply amplified the aforementioned erratic behavior. These observations lead to the adjustment down to 0.3, which was used in the final version of the network, as it had proven to be the most useful value. Compared to the same network without dropout, after the same number of epochs, the dropout network had a noticeably higher range of in-sample error values (~0.05 ± 0.03) compared to the non-dropout network (~0.005 ± 0.01).

Dropout proved to be the most useful adjustment, though the improvement was marginal, as the binary log loss dropped from 0.41958 to 0.40950 (a drop of about 0.01). While this slight change may appear to be the result of random chance, all outputs from the non-dropout architecture were producing scores of 0.41958 ± 0.001. Since this improvement was an order of magnitude higher than these random fluctuations, it was concluded that dropout was very likely the direct cause of said improvement.

MEASURING ERROR

The loss functions tested were Smooth L1 Loss[8] and Mean Squared Error[9]. It was difficult to discern which of these two loss functions was more useful, as all versions of the network were wrought with overfitting. Furthermore, out-of-sample error could only be measured by submitting the network's predictions on the testing set into the contest. There is a cap of three submissions per day, since that happens to be the daily limit of this contest. Given these restrictions, submissions had to be made only when there was strong suspicion of improvement. Very often, despite this suspicion, the score would show little or no improvement. As mentioned previously, the contest's error output was in the form of binary log

[6] https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

[7] https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

[8] https://pytorch.org/docs/stable/nn.html#smoothl1loss

[9] https://pytorch.org/docs/stable/nn.html#mseloss

loss.

Mean Squared Error tended to produce in-sample error values that were higher, and therefore likely more reasonable and accurate, than Smooth L1 Loss. Furthermore, the MSE's outputs between epochs was generally more consistent than Smooth L1's outputs, so MSE was ultimately used as the loss function in the final version of the network.

RESULTS

The best performing network was scored by the competition with a binary log loss of 0.4095, which was the lowest loss out of all submissions from various versions of this project. The worst performing network was scored by the competition with a binary log loss of 1.02377. This submission was generated by a version of the network that had yet to successfully address any of the overfitting problems discussed previously, so this abysmal score is not at all surprising.

DIFFICULTIES AND REFLECTION

As referenced throughout, there were many issues encountered along the way to producing a network that was able to make smart observations given this medical data.

Working with PyTorch

While the PyTorch library is beginner-friendly compared to most other machine learning libraries, the process of getting the project off the ground was still quite laborious. A large amount of time was spent reading through documentation and tutorials on the PyTorch website before the first attempt at implementation was made. Despite this preparation, the first several attempts at implementation were wrought with errors. In hindsight, this was in large part due to a poor understanding of tensor objects, and how they are the necessary data type to be passed to the network. Fortunately, after further research (primarily in the form of medium articles, stackoverflow posts, and YouTube tutorials), a strong enough understanding was gained, and progress began to inch forward. A key turning point was the transition from a Feedforward architecture to a multilayer perceptron, at the suggestion of the TA. This is due to the relative simplicity of this new architecture, which lead to less potential errors.

Using a Contest Dataset

A large obstacle was the noticeable lack of information regarding the performance of a given version of the network. During development, in-sample error outputs between epochs were the only indicator of the network's performance. Due to the high degree of overfitting, these outputs often shed very little insight into the actual performance of the network, which could only be measured via a submission into the contest. As a result of this lack of transparency, a large amount of intuition and guessing was present in the optimization process.

Having gone through the entire process of designing and training a network, it is now clear that a sufficiently large data set will likely make the entire process easier. This knowledge will be kept in mind when choosing a new dataset to work with for future projects.

Data Representation

There are a variety of features in this dataset, which raised concern since many of the features were represented as integers. For example, the `sex` feature is a simple binary value, but it is viewed by the network as an integer. The `resting_blood_pressure` feature is an integer value, and is considerably larger than the binary features in the dataset. Due to this, there were fears that the binary values would be overshadowed by the large integer values. It is unclear whether or not this ended up being the case, as the network was able to generate seemingly smart outputs given the testing set. For future projects, research on input representation will be necessary in order to determine whether or not this is a real issue.