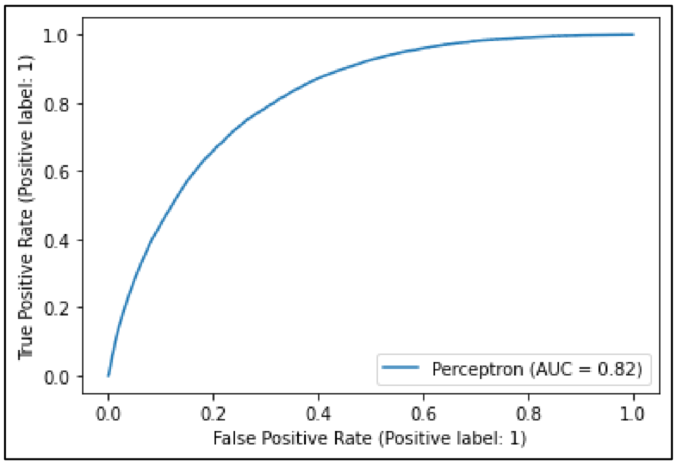
Tomisin Adeyemi - FML HW 4

1. Build and train a Perceptron (one input layer, one output layer, no hidden layers and no activation functions) to classify diabetes from the rest of the dataset. What is the AUC of this model?

I used the scikit learn Perceptron class to build my perceptron model. Because the dataset is small enough for the specific train/test/split and random seed combination to matter, I designed a wrapper class called optimizePerceptron that takes in a list of test sizes and random states and uses an approach similar to GridSearchCV to find the combination that results in the highest AUC.

I decided to use scikit learn's perceptron model, rather than building a model in pytorch, because it was relatively simple to just use Scikit learn's implementation. The choice of test sizes and random states were mostly just arbitrary, I picked values that I had seen used often.

The optimal test_size and state for the perceptron was found to be 0.4 and 10 respectively, leading to an AUC of 0.82, graphed below:



An AUC of 0.82 is pretty good and shows that a Perceptron model is relatively good at classifying diabetes.

Feedforward neural network preamble

For questions 2 to 5, I designed different classes and functions to aid the process of using the FeedForward Network. Here is a summary of them:

- `get_split(X, y, test_size, seed)`: wrapper around scikit learn's train_test_split function that also sets the PyTorch seed.
- `get_weights(y)`: uses Scikit Learn's compute_class_weight class to compute the appropriate class weights, specifically because of the imbalanced nature of the dataset.
- `LoadData(Dataset)`: this inherits the PyTorch Dataset class and is used to load any relevant data to a format compatible with Pytorch.
- `FeedforwardNeuralNetModel(nn.Module)`: this is my FeedForward Neural Network class. It takes in the input dimensions, hidden dimensions, output dimensions, number of hidden layers and activation functions as parameters. This way, I can easily test how each of these parameters affect the output of the model.

- `train(model, epoch, criterion, optimizer, train_loader), test(model, criterion, optimizer, test_loader, plot=False)`: I specified different train and test functions for the Diabetes detection/BMI detection questions, as they both used different loss functions.
- `get_train_test_loader(X, y, test_size, seed, binary=True, batch_size=1024)`: converts regular X and y data from csv file to loader compatible with pytorch.

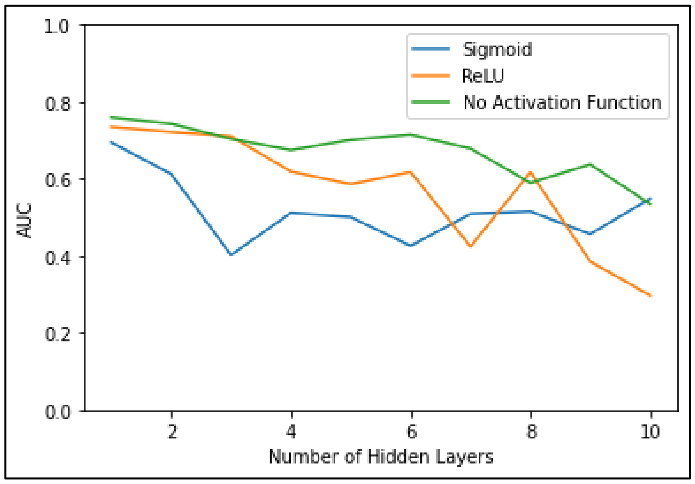
Commentary on network input parameters

- number of input dimensions: because there are 22 columns in the diabetes dataset, and 1 of those columns was predicted, there are 21 input dimensions for the neural network.
- number of hidden *dimensions*: varied depending on the question, explained further in each question.
- activation function: I experimented with Sigmoid, ReLU and no activation function.
- number of output dimensions: 2 for diabetes prediction (can either be 0 or 1), and 1 for BMI prediction (could be a range of (possibly continuous) values).
- number of hidden layers: depended on the question, explained further in each question.
- number of epochs: differed depending on the question, but the main factor that influenced this was the amount of time it took for training.
- learning rate: 0.001 is used for most of the questions.
- batch size: I used a default batch size of 1024, but experimented with different batch sizes in question 5.

2. Build and train a feedforward neural network with at least one hidden layer to classify diabetes from the rest of the dataset. Make sure to try different numbers of hidden layers and different activation functions (at a minimum ReLU and sigmoid). Doing so: How does AUC vary as a function of the number of hidden layers and is it dependent on the kind of activation function used (make sure to include "no activation function" in your comparison). How does this network perform relative to the Perceptron?

Because of test size and seed concerns, I first checked which test sizes and seed combos gave the best AUC for each activation function. I used 100 hidden dimensions to give the model the opportunity to learn a lot in just one epoch. I found that test sizes of 0.1 worked best for all activation functions. This can be since the model has the most data to train itself on (90%), or the smaller test size gives room for error. This could also mean that the model is overfitting. After finding the optimal test size and random states, I used these values to train a feedforward neural network for each of the activation functions, using 1 to 10 hidden layers. I increased the number of epochs to 2 to aid the training process, increasing it to more than 2 would have been computationally inefficient on my machine.

The highest AUC for all activation functions was from using just 1 hidden layer, with AUC of 0.69, 0.73 and 0.76 for the Sigmoid, ReLU and no activation function respectively. A graph of the AUC against the number of hidden layers, is shown below:



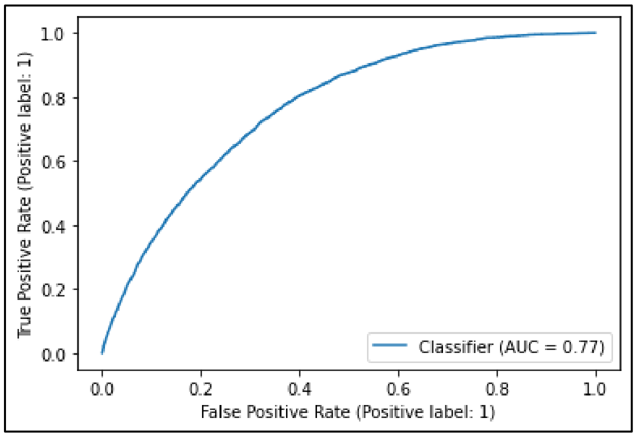
There is generally no improvement in performance as the number of hidden layers increases, although for the Sigmoid function it seems to increase but not to the same level of performance as just using 1 layer. Increasing the number of hidden layers past 10 would have been too computationally expensive, so it is safe to say it is not worth increasing the layers any further. This probably means that 1 layer is enough to learn about the relationships in this dataset, increasing the layers beyond that probably gives room for overfitting or confusing the model. The highest AUC, 0.76 for the model with no activation function and 1 hidden layer, is a slight decrease in performance from the perceptron model, which reiterates that hidden layers are probably not needed to predict diabetes.

3. Build and train a "deep" network (at least 2 hidden layers) to classify diabetes from the rest of the dataset. Given the nature of this dataset, is there a benefit of using a CNN or RNN for the classification?

Generally, there is no added benefit of using a CNN or RNN to predict diabetes, as a CNN is used to take advantage of nearness/spatial locality in data, i.e. for images, and an RNN is used to take advantage of sequential data/temporal locality, i.e. audio data. Neither of these features are shown in this dataset, so it would add unnecessary complexity and computationally expensive layers to our model training, which as shown in question 2 above, is not very helpful.

The model in this question builds upon the best performing models in question 2. I increased the number of epochs to 5 and the number of hidden dimensions to 200, because only 1 network is being trained in this question. I also increased the learning rate from 0.001 to 0.01 to compensate for the increase in computational resources, but I later found that this increase does not really affect the results. I decided to use no activation function, as the functions I used did not improve performance. Lastly, I used 3 hidden layers to build the "deep" network, because it is the minimum number layers required for the network to be deep, and using too many layers may not be worth the computational resources.

The AUC of this model is 0.77, slightly better than the best performing model in question 2. This is most likely due to the increase in hidden dimensions rather than the increase in number of layers, for reasons explained previously. The graph is shown below:

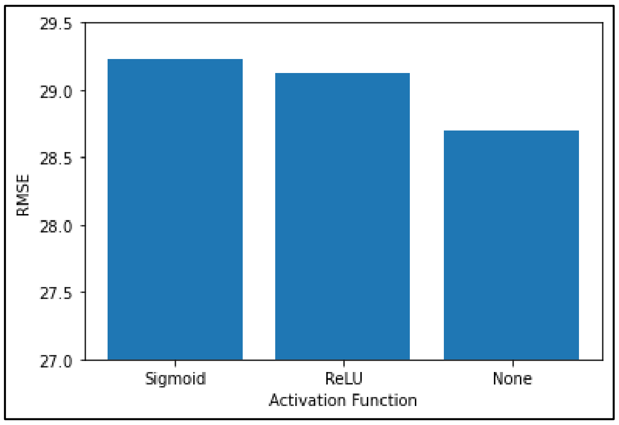


An AUC of 0.77 is generally good, but could be achieved without as much computational resources as this network used (show in previous homework using just machine learning models)

4. Build and train a feedforward neural network with one hidden layer to predict BMI from the rest of the dataset. Use RMSE to assess the accuracy of your model. Does the RMSE depend on the activation function used?

First, I wrote new train and test functions that used MSELoss as the loss function, then computed the square root of that to find the RMSE. Then, I found the optimal train test split for each activation function. The optimal test size and seed was the same for all activation functions: 0.1 and 1234 respectively. After I computed the RMSE for each activation function, using the optimal test size and seed. I used 150 hidden layers for finding the best activation function, which yielded better results (but took more time to train) than when I used 100 hidden layers to find the optimal test size and random state.

The RMSE for the Sigmoid, ReLU and no activation function was 29.22, 29.12 and 28.67 respectively. This is shown on the graph on the right:

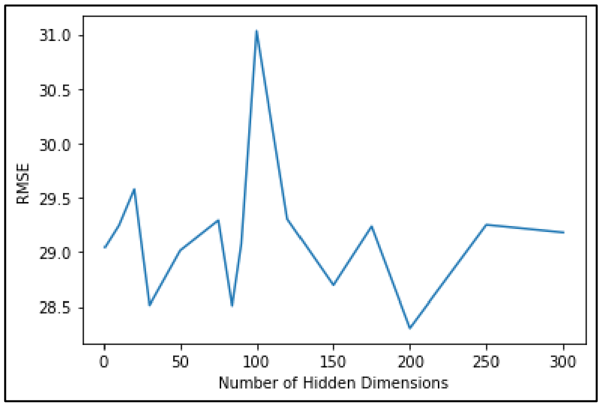


Given that using no activation function yielded the best results, even though the feature predicted now is BMI, shows that non-linearity is probably not needed for this dataset.

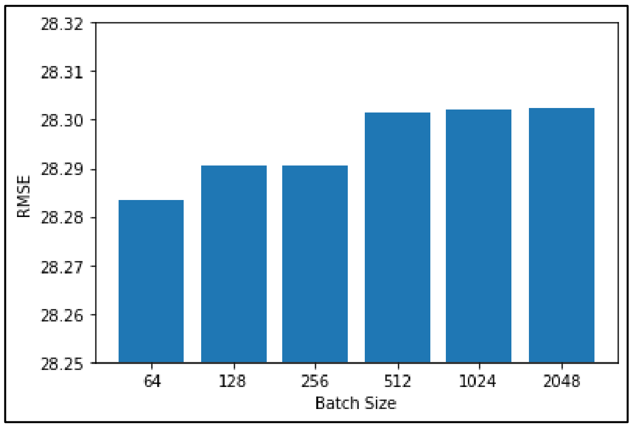
5. Build and train a neural network of your choice to predict BMI from the rest of your dataset. How low can you get RMSE and what design choices does RMSE seem to depend on?

Use #4 as a baseline, I progressively tuned different hyperparameters and picked the best performing value for each hyperparameter. Just as a reminder, here are the values I started out with, as a continuation from number 4: `input_dim = 21, hidden_dim = 150, output_dim = 1, num_hidden = 1, num_epochs = 2, learning_rate = 0.001, test_size = 0.1, seed = 1234, activation = None`. I tried tuning 8 different hyperparemeters:

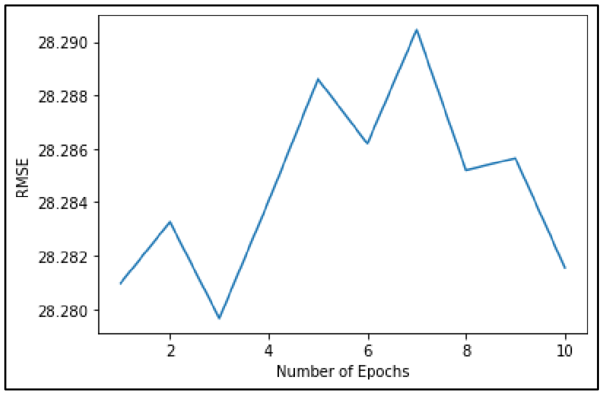
1. number of hidden layers – this did not reduce the RMSE, this is expected given the previous questions.
2. number of hidden dimensions – I tried a different number of hidden dimensions and using 200 dimensions decreased the RMSE to 28.302. There is really no specific way to determine the number of hidden dimensions that work best, it is usually dependent on the nature of the dataset and classification/regression task. Hence, it is not surprising that the graph on the right is a bit all over the place.



3. learning rate - I tried learning rates from 0.001 to 1, but the RMSE did not change at all. I decided to just stick with a value of 0.1.



4. batch size – I tried different batch sizes, and as expected, smaller batch sizes took less time to run. The best performing batch size turned out to be the smallest one I tested, which was 64. This resulted in the RMSE dropping to 28.283. This is illustrated in the graph on the left. This may be because a smaller batch size allows the model to learn more intricate patterns in the dataset.
5. Number of epochs – the change in RMSE for the number of epochs seems to be somewhat random. I was not able to test for more than 10 epochs as that would have taken too much time. However, I suspect that the randomness is because there isn't that much for the model to learn at each epoch, especially given that the number of hidden dimensions is 200. Either way the number of epochs that resulted in the lowest RMSE was 3 epochs, with an RMSE of 28.2797. The graph is illustrated on the right.



6. Momentum - I experimented with changing the momentum parameter in the optimizer function I used (SGD), but this did not change anything.
7. Activation function - like before, using no activation function yielded better results than trying to use any other activation function. As explained before, this probably means that non-linearity is not needed to understand the complexity of this dataset,
8. Regularization – I tried to regularize by modifying the weight_decay (L2 regularization) parameter of the optimizer, but as expected, this penalty only increased the RMSE.

Overall, the lowest RMSE I could achieve was **28.2797**, anything lower than that would have either been too computationally expensive, or an overfit on the train data.

Extra credit b) Write a summary statement on the overall pros and cons of using neural networks to learn from the same dataset as in the prior homework, relative to using classical methods (logistic regression, SVM, trees, forests, boosting methods). Any overall lessons?

Pros:

- Neural networks can learn more complex relationships than just regular machine learning methods, but not every dataset is very complex. This dataset for example, does not necessitate the level of complexity that neural networks give.
- Now (after a lot of pain!!) I know how to program neural networks, and I have learnt that it is highly dependent on the use case: if a regular machine learning model can do it, you probably do not need a neural network.

Cons:

- There are a lot more hyperparameters to think about, so it can make model development more difficult.
- Neural networks use a lot more computational resources than regular machine learning models, and the change in performance did not compensate for that. The regular machine learning models in the prior homework performed better, showing that a neural network is probably not appropriate for this task.