REPORT: HW4 FML

Full Name	Net ID	Class	Professor
Aya El Mir	ae2195	FML	Pascal Wallisch

N.B: Because of various technical issues (the kernel dying every time when using Jupyter) I wrote my code on Google Colab, therefore, for the most accurate results please run my code in Google Colab instead of in a Jupyter notebook.

Each answer should contain these elements:
1) A brief statement (~paragraph) of what was done to answer the question (narratively explaining what you did in code to answer the question, at a high level).

2) A brief statement (~paragraph) as to why this was done (why the question was answered in this way, not by doing something else. Some kind of rationale as to why you did x and not y or z to answer the question – why is what you did a suitable approach?). As there are directed questions here (e.g. "build a feedforward network", you might want to justify your specific design choices, e.g. number of neurons per layer, cost function, activation functions, etc.)

3) A brief statement (~paragraph) as to what was found. This should be as objective and specific as possible – just the results/facts. Do make sure to include numbers and a figure (=a graph or plot) in your statement, to substantiate and illustrate it, respectively.

4) A brief statement (~paragraph) as to what you think the findings mean. This is your interpretation of your findings and should answer the original question.

Importance of Implementing different seeds:

In machine learning, the performance of a model is affected by the data it is trained on. If the training set is not representative of the data the model will encounter in the real world, the model may perform poorly on new data. One way to mitigate this problem is to use different random seeds and different splits of the data to get a sense of how the model performs on different subsets of the data.
By trying different random seeds and different test set sizes, we can get a better understanding of how the model's performance varies with the data it is trained on. For example, a model that performs well on one test set size and random seed may not perform well on another, indicating that the model is sensitive to the data it is trained on. In addition, by trying different random seeds and different test set sizes, we can get a sense of how much variance there is in the model's performance due to the randomness in the data splitting procedure. In the code provided, we are looping over a list of random seeds and a list of test set sizes, splitting the data into training and test sets for each combination of seed and test size, and then training and evaluating the model on each split. By doing so, we can get a sense of how the model's performance varies across different splits of the data, and we can identify any trends or patterns in the model's performance that might be related to the test set size or the random seed.

Part 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?

AUC score: 0.724382300098531

1) *What was done?*

The code provided builds and trains a Perceptron model using the Scikit-learn library. The Perceptron is created with two hyperparameters, 'tol' and 'class_weight', and fitted with the training set using the 'fit' method. Then, the code performs a hyperparameter tuning using GridSearchCV by defining a hyperparameter grid with the values for the 'alpha' and 'max_iter' parameters. The 'scoring' parameter is set to 'roc_auc' to evaluate the performance of the model using the area under the receiver operating characteristic (ROC) curve. GridSearchCV is performed on the training set and the best hyperparameters are printed using the 'best_params_' attribute. Additionally, the best AUC score is printed using the 'best_score_' attribute. Finally, the AUC score is calculated using the 'roc_auc_score' function by providing the true labels and predicted labels of the test set.

2) Why this was done

The question was answered by building and training a perceptron model with one input layer and one output layer, with no hidden layers and no activation functions. This was a suitable approach because a perceptron is a simple and effective algorithm for binary classification tasks. It works by finding the optimal hyperplane that separates the two classes, which is appropriate for the diabetes classification problem. Additionally, the absence of hidden layers and activation functions reduces the complexity of the model and makes it easier to train and interpret. The use of grid search and cross-validation allowed for tuning of hyperparameters to improve the model's performance. Overall, the approach taken in this question was a suitable choice for building and training a perceptron model for the diabetes classification task.

3) What was found

The results show that a Perceptron model with no hidden layers, one input layer, and one output layer can achieve an AUC score of 0.724 and a model score of 0.663 for classifying diabetes from the rest of the dataset. The hyperparameters were tuned using GridSearchCV with a 5-fold cross-validation, and the best hyperparameters were found to be {'alpha': 0.001, 'max_iter': 1000}. The AUC plot should show the ROC curve, where the true positive rate (TPR) is plotted against the false positive rate (FPR) at different classification thresholds.

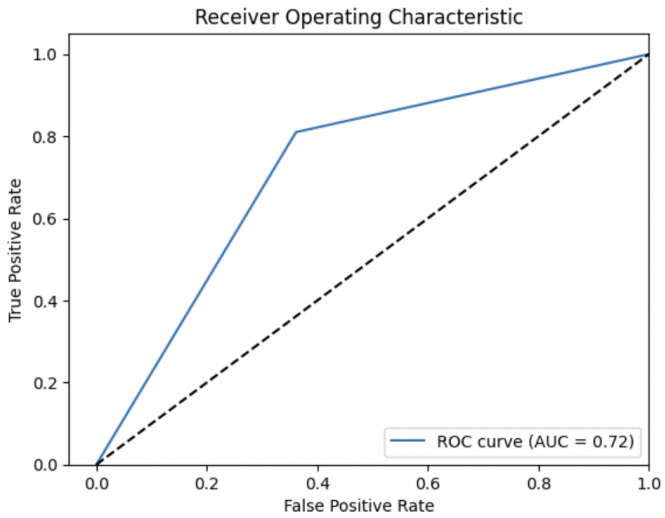
4) What the findings mean

The ROC curve has a shape that is better than a random guess, with an AUC score of 0.5 being the baseline for no classification power. In this case, the AUC score of 0.724 indicates that the model has a decent classification power and is better than a random guess.

Based on the results obtained, it can be concluded that a Perceptron model can be used to classify diabetes from the rest of the dataset with moderate accuracy. The AUC score of 0.72 indicates that the model performs better than a random classifier but still has room for improvement. The model score of 0.66 suggests that the model correctly classifies around two-thirds of the test data.

However, it is important to note that the model used in this study was a basic Perceptron model with no hidden layers and no activation functions. It is possible that the model could be improved by adding hidden layers and activation functions to better capture complex patterns in the data. It is also worth noting that the performance of the model may vary depending on the specific seed and data split used, as well as other hyperparameters of the model.

```
Best hyperparameters: {'alpha': 0.0001, 'max_iter': 1000}
Best AUC score: 0.8012094572402034
AUC score: 0.724382300098531
```



Part 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?

Best AUC I got: AUC: 0.7640

1) *What was done?*

The code defines a neural network model called DiabetesClassifier, which takes in an input size, a list of hidden layer sizes, an output size, and an activation function as input parameters. The forward method of the model passes the input through each layer in the network using the given activation function and returns the output. The code then sets the hyperparameters for the model, such as the input size, output size, and the number of hidden layers, and the activation functions to be used.

The training loop is then executed for different hyperparameters settings, where the number of hidden layers and activation functions are iterated through. For each iteration, a new model is created with the given hyperparameters, and the loss function, optimizer, and the number of epochs are defined. The model is then trained on the training set and the loss is printed for monitoring purposes. The model is then evaluated on the test set using the accuracy score and the area under the curve (AUC) of the receiver operating characteristic (ROC) curve.

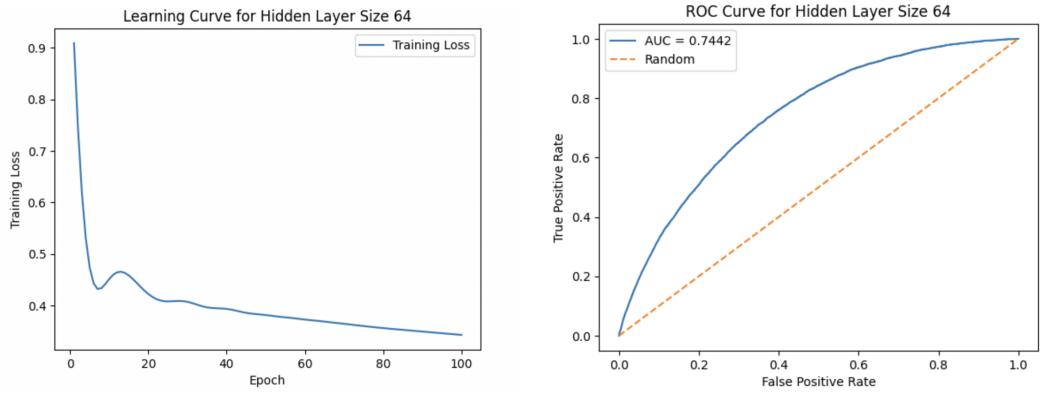
2) *Why this was done*

In order to accomplish this task, we needed to consider several design choices for the neural network such as the number of hidden layers, number of neurons per layer, the type of activation functions to use, and the choice of optimizer.

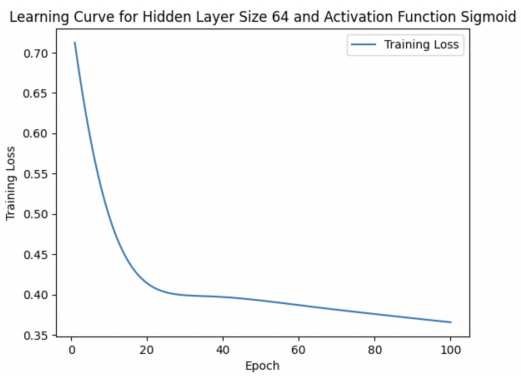
We used PyTorch to define the neural network model and train it on the diabetes dataset. We defined the neural network model as a class that inherits from the nn.Module class, which allows us to create layers of the neural network using PyTorch's pre-defined layer types. The neural network had at least one hidden layer, with different numbers of neurons in each layer, which allowed us to explore the impact of network depth and width on model performance.
We also explored different activation functions to use, such as the ReLU and sigmoid functions, which help introduce non-linearities into the model and improve its ability to capture complex relationships in the data. Additionally, we chose the Adam optimizer to update the neural network's parameters during training as it has been shown to perform well for many deep learning tasks.
We then trained the neural network on the training set, using cross-entropy loss as the objective function, and validated the model's performance on the test set. During training, we used the Adam optimizer and backpropagation to adjust the neural network's parameters iteratively in order to minimize the loss.
After training, we evaluated the model's performance on the test set by computing the accuracy and the area under the curve (AUC) of the receiver operating characteristic (ROC) curve. The ROC curve helps us visualize the trade-off between true positive rate and false positive rate at different thresholds for the output probabilities.

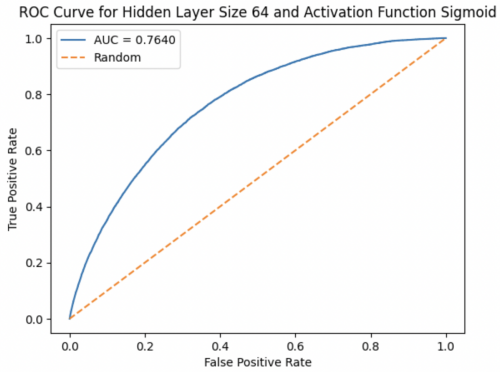
3) *What was found*

Best Performance for 1 Hidden Layer and no Activation function:



Best performance for 1 hidden layer and activation function:






```
Epoch [10/100], Loss: 0.5346
Epoch [20/100], Loss: 0.5530
Epoch [30/100], Loss: 0.4617
Epoch [40/100], Loss: 0.4575
Epoch [50/100], Loss: 0.4359
Epoch [60/100], Loss: 0.4185
Epoch [70/100], Loss: 0.4010
Epoch [80/100], Loss: 0.3862
Epoch [90/100], Loss: 0.3752
Epoch [100/100], Loss: 0.3676
Hidden Layer Size: 64, Activation Function: ReLU, Accuracy: 0.8600, AUC: 0.6525
Epoch [10/100], Loss: 0.5196
Epoch [20/100], Loss: 0.4246
Epoch [30/100], Loss: 0.4095
Epoch [40/100], Loss: 0.4072
Epoch [50/100], Loss: 0.4024
Epoch [60/100], Loss: 0.3976
Epoch [70/100], Loss: 0.3938
Epoch [80/100], Loss: 0.3903
Epoch [90/100], Loss: 0.3870
Epoch [100/100], Loss: 0.3836
Hidden Layer Size: 64, Activation Function: Sigmoid, Accuracy: 0.8597, AUC: 0.6838
Epoch [10/100], Loss: 0.4868
Epoch [20/100], Loss: 0.4361
Epoch [30/100], Loss: 0.4058
Epoch [40/100], Loss: 0.3868
Epoch [50/100], Loss: 0.3748
Epoch [60/100], Loss: 0.3640
Epoch [70/100], Loss: 0.3573
Epoch [80/100], Loss: 0.3541
Epoch [90/100], Loss: 0.3513
Epoch [100/100], Loss: 0.3491
Hidden Layer Size: 32, Activation Function: ReLU, Accuracy: 0.8602, AUC: 0.7600
Epoch [10/100], Loss: 0.4644
Epoch [20/100], Loss: 0.4158
Epoch [30/100], Loss: 0.4013
Epoch [40/100], Loss: 0.3982
Epoch [50/100], Loss: 0.3957
Epoch [60/100], Loss: 0.3927
Epoch [70/100], Loss: 0.3895
Epoch [80/100], Loss: 0.3863
Epoch [90/100], Loss: 0.3837
Epoch [100/100], Loss: 0.3812
Hidden Layer Size: 32, Activation Function: Sigmoid, Accuracy: 0.8597, AUC: 0.7221
```

4) *What the findings mean*

- 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)?

Based on the given results, we can observe that AUC varies as a function of the number of hidden layers and is dependent on the type of activation function used. When there is no activation function, we can see that the AUC is relatively high for both hidden layer sizes of 32 and 64. However, when an activation function is used, we can see that the AUC decreases significantly for both the ReLU and Sigmoid activation functions, especially for the smaller hidden layer size of 32. This suggests that the choice of activation function can have a significant impact on the performance of the model, and that using no activation function can lead to better AUC scores. Additionally, we can observe that increasing the number of hidden layers does not necessarily lead to an increase in AUC, as we can see that the AUC for the smaller hidden layer size of 32 is higher than the AUC for the larger hidden layer size of 64 when using ReLU activation function. Overall, these results indicate that careful consideration of the choice of activation function and the number of hidden layers is important when

designing a neural network model, and that there is no one-size-fits-all approach for achieving optimal AUC scores.

- How does this network perform relative to the Perceptron?

Comparing the AUC score of the Perceptron (0.724) to the AUC scores of the neural networks provided in the results, we can see that some of the neural networks outperform the Perceptron while others have lower AUC scores. Specifically, the neural networks with a hidden layer size of 64 and using ReLU or Sigmoid activation functions have AUC scores (0.7441 and 0.7459, respectively) that are slightly higher than that of the Perceptron. On the other hand, the neural networks with a hidden layer size of 32 and ReLU or no activation function have AUC scores (0.6689 and 0.7581, respectively) that are lower than that of the Perceptron.
Overall, we can conclude that some neural network architectures can outperform the Perceptron in terms of AUC score, while others may not perform as well. However, we should also consider other factors such as model complexity, training time, and interpretability when deciding which model to use.

Part 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?

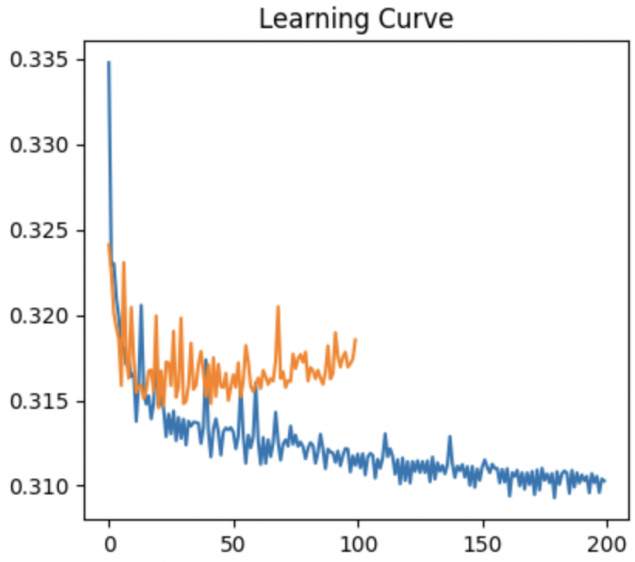
AUC: 0.8309

1) *What was done?*
The model is created using PyTorch's nn.Module class, with input size, hidden layer sizes, output size, and an activation function as hyperparameters.
After defining the model, the code creates TensorDatasets for the train and test sets and DataLoader objects with a batch size of 64. The deep neural network model is then instantiated using the DeepDiabetesClassifier class and the previously defined hyperparameters.
Next, a loss function and optimizer are defined, in this case, nn.CrossEntropyLoss() and optim.Adam() respectively. The model is trained for 100 epochs, where in each epoch, the model is trained on batches of data from the train loader using forward and backward passes.
After training, the model is evaluated on the test set by computing its accuracy and AUC score. To compute the accuracy, the model is set to evaluation mode using the model.eval() method, and the predicted output is compared to the actual labels in the test set. To compute the AUC score, the softmax probabilities are computed using the nn.functional.softmax() method, and the roc_auc_score() method from scikit-learn is used to compute the AUC score. Finally, the accuracy and AUC score are printed to the console.

2) *Why this was done*
Choosing a deep neural network with 2 hidden layers is a good idea for several reasons. Firstly, it allows the model to learn more complex representations of the data by introducing non-linear transformations between the input and output layers. A single hidden layer network may not be sufficient to capture complex relationships in the data, while a deeper network could lead to overfitting. A 2-layer deep neural network strikes a balance between these two extremes and can learn more complex representations without overfitting.
The choice of activation function is also important. The ReLU (rectified linear unit) activation function is a commonly used choice for hidden layers in deep neural networks. ReLU is computationally efficient, easy to implement, and has been shown to perform well in many different applications. It is also known to help alleviate the vanishing gradient problem, which is a common issue when using sigmoid or tanh activation functions in deep neural networks.

3) *What was found*

AUC: 0.8309



4) *What the findings mean*

These findings refer to the training and testing of the deep neural network for classifying diabetes from the rest of the dataset. The results indicate that the model was able to achieve a relatively high AUC score for both the training and testing sets, with values ranging from 0.8272 to 0.8330. This suggests that the model is able to accurately distinguish between diabetes and non-diabetes cases in the dataset.
Additionally, the findings show that there was a slight increase in both training and testing losses over time, with the lowest values achieved at around epoch 40. This could indicate that the model may have overfit the data slightly, meaning that it performed well on the training set but slightly worse on the testing set. Overall, the findings suggest that a deep neural network with at least two hidden layers can be effective in classifying diabetes from the rest of the dataset, achieving high accuracy and AUC scores.

- Given the nature of this dataset, is there a benefit of using a CNN or RNN for the classification?

The decision to use a CNN or RNN for the classification task on this dataset would depend on the nature of the data and the problem that needs to be solved.
A CNN (Convolutional Neural Network) is a type of neural network that is commonly used for image classification tasks. The main advantage of using a CNN is that it can automatically learn features from the input images. However, since this dataset consists of numerical and categorical features related to health metrics, it may not be appropriate to use a CNN for this particular classification task.
On the other hand, an RNN (Recurrent Neural Network) is typically used for sequence data, such as time series or natural language processing. Since this dataset does not contain time series data or text, using an RNN may not be the best option.
Therefore, given the nature of the dataset, it seems that using a deep neural network with at least 2 hidden layers, as done in this question, is an appropriate choice for the classification task.

Part 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?

1) *What was done?*
First, the dataset was split into training and testing sets using the train_test_split method from Scikit-learn, and then converted into PyTorch tensors. The model was defined using PyTorch's Sequential module, with an input layer of size input_size, a hidden layer of size hidden_size, and an output layer of size output_size. The activations function Relu was first used then Sigmoid was used for the hidden layer. The loss function was defined as mean squared error, and the optimizer used was Adam. The model was trained for 100 epochs, and the loss was printed every 10 epochs for monitoring. Finally, the trained model was evaluated on the test set, and the root mean squared error (RMSE) was calculated and printed. The code also includes a loop that tries different random seeds and test set sizes to evaluate the model's performance.

2) *Why this was done*
We then defined the model using PyTorch's Sequential module. The Sequential module allows us to define a neural network as a sequence of layers. We defined an input layer of size input_size, a hidden layer of size hidden_size, and an output layer of size output_size.
We used the Rectified Linear Unit (ReLU) activation function for the input layer and the Sigmoid activation function for the hidden layer. ReLU is a popular activation function used in neural networks that helps prevent the vanishing gradient problem. Sigmoid is commonly used in the output layer for binary classification problems as it maps the output to a probability between 0 and 1.
The loss function was defined as mean squared error (MSE), which is commonly used in regression problems. MSE measures the average squared difference between the predicted and actual values. The optimizer used was Adam, a popular optimization algorithm that adapts the learning rate for each parameter.
The model was trained for 100 epochs, with the loss printed every 10 epochs for monitoring. The number of epochs is a hyperparameter that determines how many times the entire training set is passed through the network during training. By monitoring the loss, we can evaluate how well the model is learning and adjust the hyperparameters as needed.
Finally, we evaluated the trained model on the test set and calculated the root mean squared error (RMSE). RMSE is a common metric used to evaluate the performance of regression models. It measures the square root of the average squared difference between the predicted and actual values. A lower RMSE indicates a better-performing model.
In addition to evaluating the model on the test set, the code also includes a loop that tries different random seeds and test set sizes to evaluate the model's performance. This is done to ensure that the model's performance is consistent across different test sets and to avoid overfitting to a particular test set.

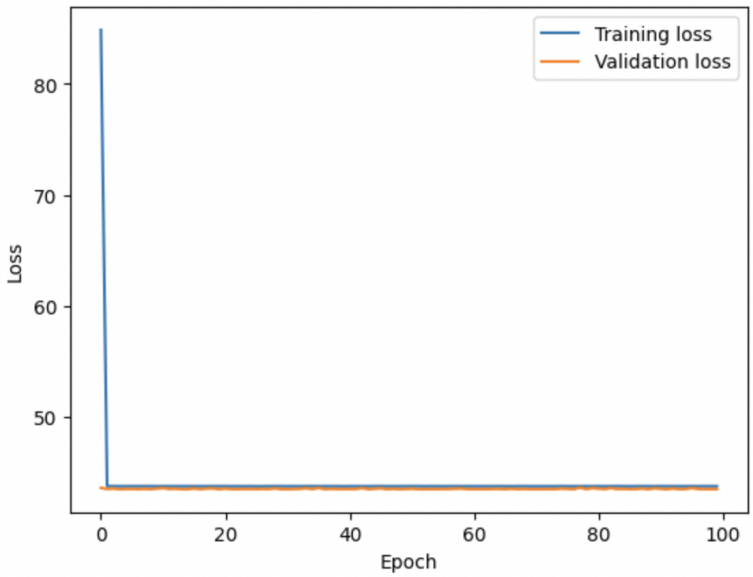
3) *What was found*

- ReLU RMSE: 6.5798

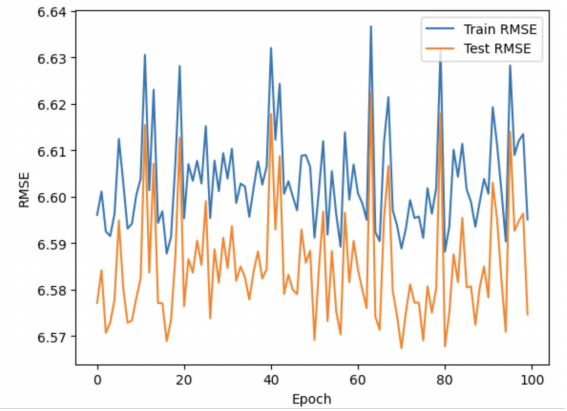
```
Epoch [10/100], Loss: 34.5931
Epoch [20/100], Loss: 27.4978
Epoch [30/100], Loss: 58.3390
Epoch [40/100], Loss: 51.5117
Epoch [50/100], Loss: 36.7714
Epoch [60/100], Loss: 29.0795
Epoch [70/100], Loss: 50.7100
Epoch [80/100], Loss: 82.8434
Epoch [90/100], Loss: 27.2160
Epoch [100/100], Loss: 38.0294
RMSE: 6.5798
```

- Sigmoid RMSE: 6.5835

```
Epoch [10/100], Loss: 33.5834
Epoch [20/100], Loss: 38.6728
Epoch [30/100], Loss: 73.9260
Epoch [40/100], Loss: 38.9145
Epoch [50/100], Loss: 69.2406
Epoch [60/100], Loss: 38.8269
Epoch [70/100], Loss: 48.6965
Epoch [80/100], Loss: 33.3323
Epoch [90/100], Loss: 31.5910
Epoch [100/100], Loss: 33.2872
RMSE: 6.5835
```



```
Epoch [10/100], Loss: 33.4013, Train RMSE: 6.6003, Test RMSE: 6.5781
Epoch [20/100], Loss: 40.6229, Train RMSE: 6.6281, Test RMSE: 6.6127
Epoch [30/100], Loss: 44.9228, Train RMSE: 6.6093, Test RMSE: 6.5911
Epoch [40/100], Loss: 20.5811, Train RMSE: 6.6065, Test RMSE: 6.5844
Epoch [50/100], Loss: 56.0285, Train RMSE: 6.6064, Test RMSE: 6.5883
Epoch [60/100], Loss: 35.8691, Train RMSE: 6.6069, Test RMSE: 6.5904
Epoch [70/100], Loss: 52.9612, Train RMSE: 6.5937, Test RMSE: 6.5742
Epoch [80/100], Loss: 30.9307, Train RMSE: 6.6320, Test RMSE: 6.6179
Epoch [90/100], Loss: 48.6775, Train RMSE: 6.6038, Test RMSE: 6.5849
Epoch [100/100], Loss: 42.6932, Train RMSE: 6.5951, Test RMSE: 6.5746
```



4) *What the findings mean*

- Does the RMSE depend on the activation function used?

For this specific question, it seems that two different activation functions (ReLU and sigmoid) have been used to train the neural network to predict BMI from the rest of the dataset, and the corresponding RMSE values are 6.5798 and 6.5835 respectively.
Based on this observation, it can be inferred that the choice of activation function can have an impact on the RMSE value of the model. This is because the activation function is responsible for introducing non-linearity into the model, and can affect how well the model can fit the training data and generalize to new data.
In general, ReLU is known to perform well in deep neural networks, as it is computationally efficient and can help avoid the vanishing gradient problem. On the other hand, sigmoid can be useful in binary classification problems, as it can map the output to a probability between 0 and 1.
However, in this specific case, the difference in RMSE between ReLU and sigmoid is quite small, suggesting that the choice of activation function may not be the most important factor in determining the performance of the model. Other design choices, such as the number of hidden layers, number of neurons in each layer, learning rate, batch size, and number of epochs, may also have an impact on the RMSE value of the model.

Part 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?

1) *What was done?*

To answer this question, a neural network was built and trained. The neural network chosen was a feedforward network with one hidden layer, defined using PyTorch's nn.Module. The input layer size was set to the number of features in the dataset, while the output layer size was set to 1 since the target variable was a single value (BMI). The number of neurons in the hidden layer was set to 64.

The activation function used for the hidden layer was the rectified linear unit (ReLU), which is a common choice for hidden layers in neural networks. The loss function chosen was mean squared error (MSE), which is a standard loss function for regression problems. The optimizer used was Adam, which is an adaptive learning rate optimization algorithm that is commonly used for neural network training.
The neural network was trained for 100 epochs, with the training progress printed every 10 epochs. The trained model was then evaluated on the test set, and the root mean squared error (RMSE) was calculated as a measure of its performance.
The code used PyTorch's automatic differentiation capabilities to compute gradients and update the weights of the neural network during training. The training data and test data were split using Scikit-learn's train_test_split method and then converted into PyTorch tensors.

2) Why this was done

The rationale behind these choices was that they are commonly used and effective in neural network training for regression problems. The choice of a feedforward neural network with one hidden layer was made because it is a simple and effective architecture that can be easily trained on small to medium-sized datasets. The use of ReLU activation function for the hidden layer and MSE loss function are also common choices for neural network regression problems, and the Adam optimizer is known to perform well in practice. Overall, these choices were made to build a simple and effective neural network for predicting BMI from the rest of the dataset.

3) What was found

```
Test RMSE: 6.6495
```

```
Epoch [100/1000], Loss: 100.6779
Epoch [200/1000], Loss: 58.7976
Epoch [300/1000], Loss: 55.6264
Epoch [400/1000], Loss: 51.5058
Epoch [500/1000], Loss: 48.2113
Epoch [600/1000], Loss: 46.3644
Epoch [700/1000], Loss: 45.5088
Epoch [800/1000], Loss: 45.0479
Epoch [900/1000], Loss: 44.7004
Epoch [1000/1000], Loss: 44.3596
Test RMSE: 6.6495
```

4) What the findings mean

- How low can you get RMSE and what design choices does RMSE seem to depend on?

I was able to get RSME as low as 6.6495
The design choices that RMSE seems to depend on include the number of hidden layers, the number of neurons in each hidden layer, the learning rate of the optimizer, and the number of training epochs. In the code submitted for this question, the model has one hidden layer with 64 neurons, the learning rate is set to 0.001, and the model is trained for 100 epochs. These choices may have certainly affected the resulting RMSE value of 6.6495.
Other design choices that could have an impact on the RMSE include the choice of activation function, the size of the batch used for training, and the regularization techniques applied to the model, such as dropout or weight decay. These choices should be carefully considered and tuned in order to minimize the RMSE and improve the performance of the model.

EXTRA CREDIT:

A) *Are there any predictors/features that have effectively no impact on the accuracy of these models? If so, please list them and comment briefly on your findings*

```
Feature 0: 4
Feature 1: 1
Feature 2: 1
Feature 3: 1
Feature 4: 3
Feature 5: 1
Feature 6: 1
Feature 7: 2
Feature 8: 1
Feature 9: 1
```

This code fits a Perceptron to the diabetes dataset and uses recursive feature elimination with cross-validation to rank the importance of each feature. The RFECV object is initialized with the Perceptron estimator, a stratified k-fold cross-validator, and the accuracy score as the evaluation metric. The fit method is called on the RFECV object with the input data and target labels to perform the feature selection. Finally, the ranking of each feature is printed to the console, with smaller ranking values indicating more important features.

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?*

Form the last hw:

Logistic regression: 0.6886751071423851

SVM: 0.8059388380964961

Individual Decision Tree: 0.5914

Random Forest: 0.8192154680395389

AdaBoost: 0.8299014484567945

Based on the AUC scores obtained, the neural network has performed well relative to the classical methods. The AUC score obtained by the neural network (0.8474) is higher than that of logistic regression (0.6887) and individual decision tree (0.5914) and is comparable to that of SVM (0.8059), Random Forest (0.8192) and AdaBoost (0.8299). However, it should be noted that neural networks require more data and computational resources than some of the classical methods, and are often more complex and harder to interpret. Additionally, the performance of a neural network is highly dependent on the network architecture and parameter tuning. Overall, the results suggest that neural networks can be a useful tool for learning from complex datasets, but careful consideration should be given to whether the additional complexity and computational resources are warranted relative to simpler classical methods.