Sam Hopkins

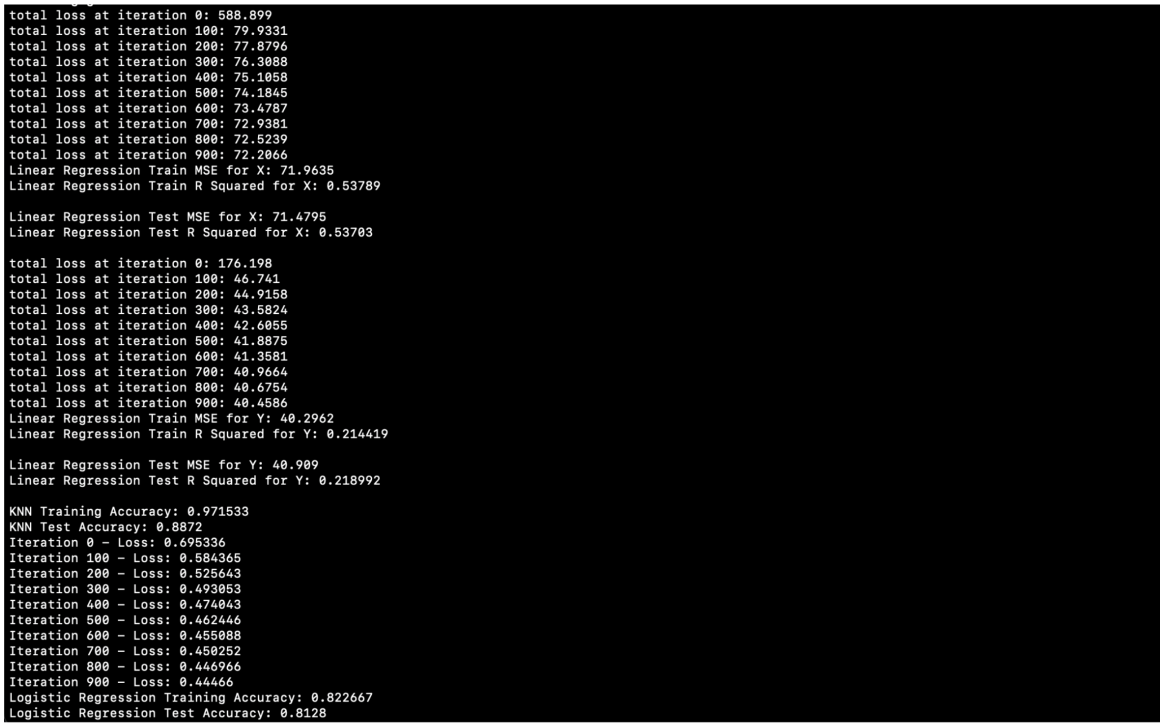
PREDICTING VEHICLE POSITION AND LINE OF SIGHT STATUS USING CHANNEL STATE INFORMATION

For any regression or classification task, there are a number of different approaches that could be used. Oftentimes, the answer to the question of which approach to take can be solved by first trying many, and using only the models that provide the best performance. Which models these are can help to reveal the underlying relationship between the inputs and the target outputs. Predicting the position of a car, and whether or not it is in line of sight, using channel state information from a remote radio unit, may sound difficult. However, with the help of certain models and programming languages, it is surprisingly easy. In this project, I attempted to solve this problem using linear regression and a CNN for predicting vehicle positions, and logistic regression, KNN and a CNN for predicting LoS/nLoS (whether or not a vehicle is in line of sight).

Three variations of the original dataset were used for different approaches to solving the problem. The first, which was used for the CNN approach, was the original dataset with its original dimensions (nSamples x 1632 x 4). The second, which was used when I attempted to solve the problem with a multilayer perceptron, was the same as the first, except that the values of all the subcarriers for each antenna were concatenated into one array of features for each sample (nSamples x 6528). The third, which was used for linear/logistic regression and KNN, contained only the mean value for each antenna as features for each sample (nSamples x 4). I used min/max normalization of each feature in all variations.

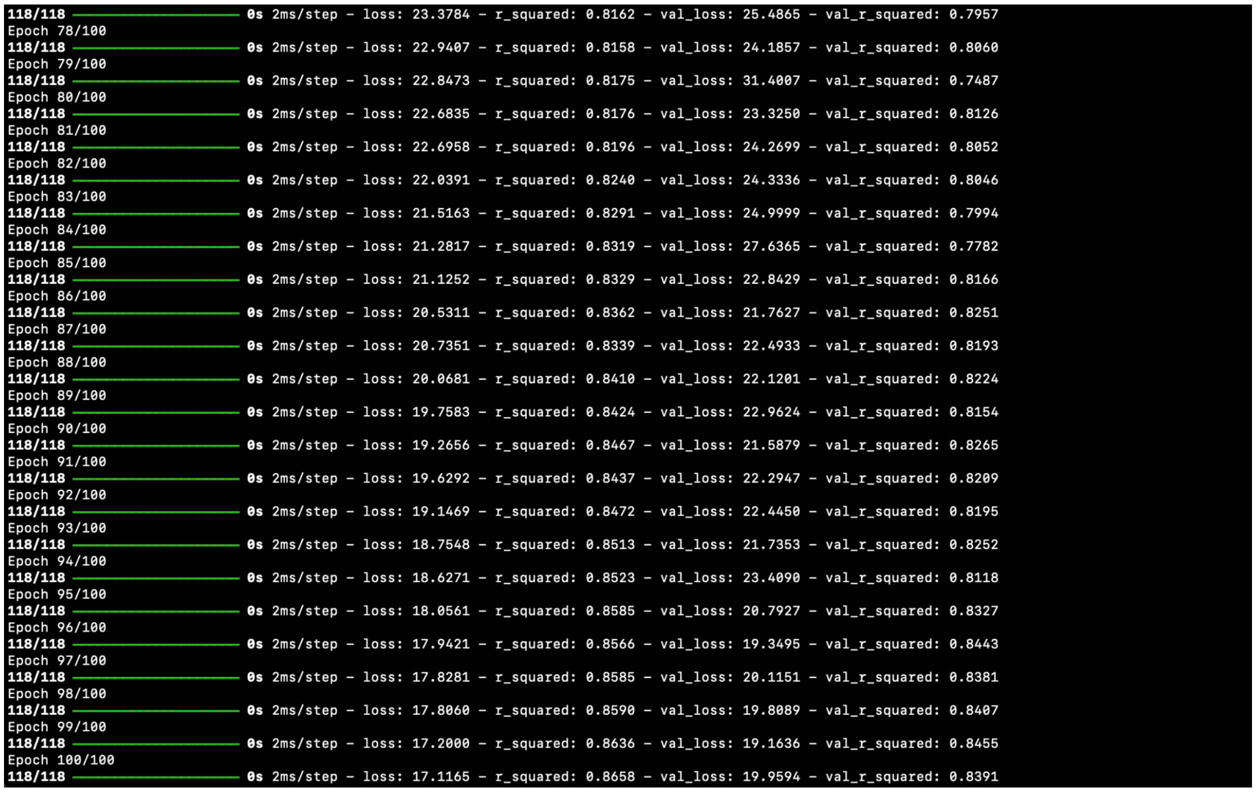
For the X and Y position, I tried fitting a linear regression model for each coordinate separately. Neither one of these models are good at approximating either coordinate. With 1000 training iterations and a learning rate of 0.5, the model for X position obtained both a training and testing R^2 value around 0.53. For Y position, these values were even worse – both around 0.21. This suggested early on that the relationship between the channel state information for a given car and the X or Y position is very nonlinear, and a more complex model, such as a neural network, would likely do a much better job of fitting the data.

For the classification problem, logistic regression and KNN were tried. I obtained a training accuracy of 0.82 and test accuracy of 0.81 using a logistic regression model that was trained with 1000 iterations and a learning rate of 0.01. For KNN (k=7), I obtained a training accuracy of 0.97, and a test accuracy of 0.89. It should be noted that my implementation of KNN was manually done and included no cross-validation like some other approaches might use, so the true maximum testing accuracy for KNN could be slightly higher. The success of logistic regression indicates that the LoS_nLoS classification problem has a more linear mapping from inputs to outputs than the problem of finding X and Y position. For all machine learning approaches above, manually coded c++ headers were used.

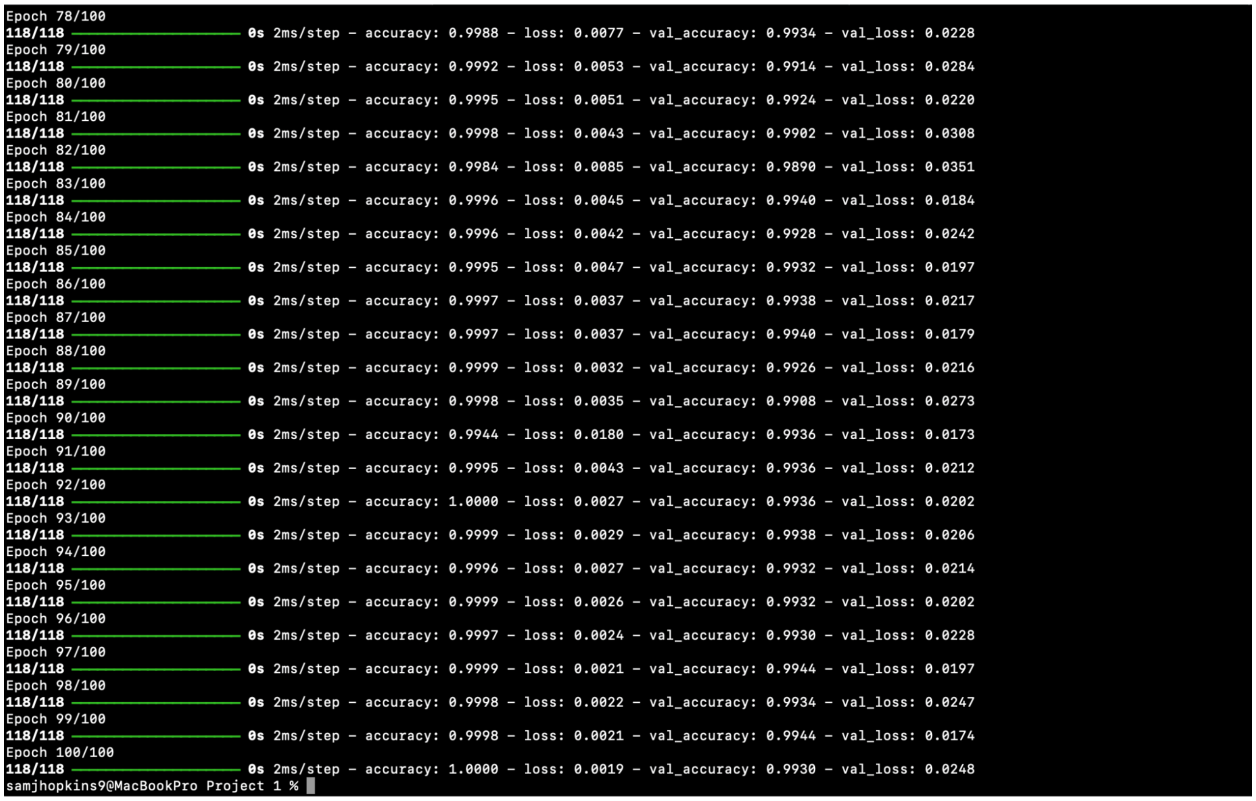


```
total loss at iteration 0: 588.899
total loss at iteration 100: 79.9331
total loss at iteration 200: 77.8796
total loss at iteration 300: 76.3088
total loss at iteration 400: 75.1058
total loss at iteration 500: 74.1845
total loss at iteration 600: 73.4787
total loss at iteration 700: 72.9381
total loss at iteration 800: 72.5239
total loss at iteration 900: 72.2066
Linear Regression Train MSE for X: 71.9635
Linear Regression Train R Squared for X: 0.53789

Linear Regression Test MSE for X: 71.4795
Linear Regression Test R Squared for X: 0.53703

total loss at iteration 0: 176.198
total loss at iteration 100: 46.741
total loss at iteration 200: 44.9158
total loss at iteration 300: 43.5824
total loss at iteration 400: 42.6055
total loss at iteration 500: 41.8875
total loss at iteration 600: 41.3581
total loss at iteration 700: 40.9664
total loss at iteration 800: 40.6754
total loss at iteration 900: 40.4586
Linear Regression Train MSE for Y: 40.2962
Linear Regression Train R Squared for Y: 0.214419

Linear Regression Test MSE for Y: 40.909
Linear Regression Test R Squared for Y: 0.218992

KNN Training Accuracy: 0.971533
KNN Test Accuracy: 0.8872
Iteration 0 - Loss: 0.695336
Iteration 100 - Loss: 0.584365
Iteration 200 - Loss: 0.525643
Iteration 300 - Loss: 0.493053
Iteration 400 - Loss: 0.474043
Iteration 500 - Loss: 0.462446
Iteration 600 - Loss: 0.455088
Iteration 700 - Loss: 0.450252
Iteration 800 - Loss: 0.446966
Iteration 900 - Loss: 0.44466
Logistic Regression Training Accuracy: 0.822667
Logistic Regression Test Accuracy: 0.8128
```

Using a CNN, I was able to obtain a training R^2 of around 0.87 and a test R^2 around 0.84 for the X and Y prediction problem. The CNN predicts both values simultaneously. For classification, the accuracy is even better, approaching 1.00 for both. Both results are after 100 training epochs. With more epochs, better results were obtained. I used a 2-layer convolutional neural network, with a single dense layer. A single convolutional layer first, which achieved similar performance to the machine learning models outlined above. When I added a second layer, the results improved to their current standard, and there was no need to add more layers. Tanh, sigmoid, and reLu were all tried as activation functions in the convolutional layers and dense layer. The choices which clearly produced the best results were reLu in both of the convolutional layers, and sigmoid in the dense one. I tried training the model using convolution over the subcarriers for each antenna separately, and over the antennae for each subcarrier separately. Convolution over the antennae for each subcarrier produced more accurate models and trained far faster, though I had to use small kernel sizes since the maximum number of antennae corresponding to each subcarrier index is four. The first layer has 16 filters, with a kernel size of 1. The second layer also has 16 filters, with a kernel size of 4. Increasing the kernel size beyond this point produced dimensionality errors, but the model still seems to be highly effective. The number of filters was something I adjusted until it seemed to reach an optimal value – lower values produced slightly lower accuracies, but higher values increased training time. The stride length was 1, and no padding or pooling was used.



```
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 23.3784 - r_squared: 0.8162 - val_loss: 25.4865 - val_r_squared: 0.7957
Epoch 78/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 22.9407 - r_squared: 0.8158 - val_loss: 24.1857 - val_r_squared: 0.8060
Epoch 79/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 22.8473 - r_squared: 0.8175 - val_loss: 31.4007 - val_r_squared: 0.7487
Epoch 80/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 22.6835 - r_squared: 0.8176 - val_loss: 23.3250 - val_r_squared: 0.8126
Epoch 81/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 22.6958 - r_squared: 0.8196 - val_loss: 24.2699 - val_r_squared: 0.8052
Epoch 82/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 22.0391 - r_squared: 0.8240 - val_loss: 24.3336 - val_r_squared: 0.8046
Epoch 83/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 21.5163 - r_squared: 0.8291 - val_loss: 24.9999 - val_r_squared: 0.7994
Epoch 84/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 21.2817 - r_squared: 0.8319 - val_loss: 27.6365 - val_r_squared: 0.7782
Epoch 85/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 21.1252 - r_squared: 0.8329 - val_loss: 22.8429 - val_r_squared: 0.8166
Epoch 86/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 20.5311 - r_squared: 0.8362 - val_loss: 21.7627 - val_r_squared: 0.8251
Epoch 87/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 20.7351 - r_squared: 0.8339 - val_loss: 22.4933 - val_r_squared: 0.8193
Epoch 88/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 20.0681 - r_squared: 0.8410 - val_loss: 22.1201 - val_r_squared: 0.8224
Epoch 89/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 19.7583 - r_squared: 0.8424 - val_loss: 22.9624 - val_r_squared: 0.8154
Epoch 90/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 19.2656 - r_squared: 0.8467 - val_loss: 21.5879 - val_r_squared: 0.8265
Epoch 91/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 19.6292 - r_squared: 0.8437 - val_loss: 22.2947 - val_r_squared: 0.8209
Epoch 92/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 19.1469 - r_squared: 0.8472 - val_loss: 22.4450 - val_r_squared: 0.8195
Epoch 93/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 18.7548 - r_squared: 0.8513 - val_loss: 21.7353 - val_r_squared: 0.8252
Epoch 94/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 18.6271 - r_squared: 0.8523 - val_loss: 23.4090 - val_r_squared: 0.8118
Epoch 95/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 18.0561 - r_squared: 0.8585 - val_loss: 20.7927 - val_r_squared: 0.8327
Epoch 96/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 17.9421 - r_squared: 0.8566 - val_loss: 19.3495 - val_r_squared: 0.8443
Epoch 97/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 17.8281 - r_squared: 0.8585 - val_loss: 20.1151 - val_r_squared: 0.8381
Epoch 98/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 17.8060 - r_squared: 0.8590 - val_loss: 19.8089 - val_r_squared: 0.8407
Epoch 99/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 17.2000 - r_squared: 0.8636 - val_loss: 19.1636 - val_r_squared: 0.8455
Epoch 100/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 17.1165 - r_squared: 0.8658 - val_loss: 19.9594 - val_r_squared: 0.8391
```



```
Epoch 78/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9988 - loss: 0.0077 - val_accuracy: 0.9934 - val_loss: 0.0228
Epoch 79/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9992 - loss: 0.0053 - val_accuracy: 0.9914 - val_loss: 0.0284
Epoch 80/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9995 - loss: 0.0051 - val_accuracy: 0.9924 - val_loss: 0.0220
Epoch 81/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss: 0.0043 - val_accuracy: 0.9902 - val_loss: 0.0308
Epoch 82/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9984 - loss: 0.0085 - val_accuracy: 0.9890 - val_loss: 0.0351
Epoch 83/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9996 - loss: 0.0045 - val_accuracy: 0.9940 - val_loss: 0.0184
Epoch 84/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9996 - loss: 0.0042 - val_accuracy: 0.9928 - val_loss: 0.0242
Epoch 85/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9995 - loss: 0.0047 - val_accuracy: 0.9932 - val_loss: 0.0197
Epoch 86/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss: 0.0037 - val_accuracy: 0.9938 - val_loss: 0.0217
Epoch 87/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss: 0.0037 - val_accuracy: 0.9940 - val_loss: 0.0179
Epoch 88/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9999 - loss: 0.0032 - val_accuracy: 0.9926 - val_loss: 0.0216
Epoch 89/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss: 0.0035 - val_accuracy: 0.9908 - val_loss: 0.0273
Epoch 90/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9944 - loss: 0.0180 - val_accuracy: 0.9936 - val_loss: 0.0173
Epoch 91/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9995 - loss: 0.0043 - val_accuracy: 0.9936 - val_loss: 0.0212
Epoch 92/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0027 - val_accuracy: 0.9936 - val_loss: 0.0202
Epoch 93/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9999 - loss: 0.0029 - val_accuracy: 0.9938 - val_loss: 0.0206
Epoch 94/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9996 - loss: 0.0027 - val_accuracy: 0.9932 - val_loss: 0.0214
Epoch 95/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9999 - loss: 0.0026 - val_accuracy: 0.9932 - val_loss: 0.0202
Epoch 96/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9997 - loss: 0.0024 - val_accuracy: 0.9930 - val_loss: 0.0228
Epoch 97/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9999 - loss: 0.0021 - val_accuracy: 0.9944 - val_loss: 0.0197
Epoch 98/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss: 0.0022 - val_accuracy: 0.9934 - val_loss: 0.0247
Epoch 99/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss: 0.0021 - val_accuracy: 0.9944 - val_loss: 0.0174
Epoch 100/100
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0019 - val_accuracy: 0.9930 - val_loss: 0.0248
samjhopkins9@MacBookPro Project 1 %
```

The code for this project consists of five files. Project1.sh is the main shell script containing all python code for reading, preprocessing, and fitting models for data (for the CNN approach), c++ code for the same (for the KNN and Regression approaches), and commands to write and run both programs. The python portion of this file also includes code to write the tensor data to a text file, which is then read into c++. I wrote four header files in c++ to manually implement the tasks of reading, storing, preprocessing, and fitting models for data. These include implementations of linear/logistic regression, KNN, and multilayer perceptron.

ProcessingFunctions.h includes the DataFrame class, which was created to store 2d and 3d matrices of data and includes methods to read text files, retrieve rows, columns, and 2d images corresponding to given indices, as well as methods to normalize the matrices using min/max normalization. This header file also contains methods to calculate the dot product and euclidean distance between two vectors, various activation functions and their derivatives, and loss/performance metrics such as MSE, R Squared and Accuracy. MachineLearning.h includes LinearRegression, LogisticRegression, and KNN classes, each of which contain a constructor and a predict method. Both regression classes include a train method as well; the KNN.predict method simply predicts the target values for a given input vector using the distances from values in the dataset provided in the constructor; “training” the model is simply predicting using the training dataset as input and adjusting k to minimize the error without overfitting. The train functions for both regression functions first initialize the weights and biases randomly, iterates over every sample in the dataset, makes a prediction using a weighted sum of the inputs plus the bias (and an activation function if logistic regression), calculates the prediction error, and updates the weights accordingly by subtracting the product of the error, the input associated with the given weight and the learning rate from each weight.

Here is where I will go more into how I attempted to construct a multilayer perceptron. While it did not end up generating remotely useful predictions relative to other models, and is somewhat limited in its capabilities, I still think it is worth delving into since it helps to understand part of what is happening inside of a neural network model, which can feel very mysterious when using pre-built packages. In MultilayerPerceptron.h I included a Neuron class, a Layer class, and an MLP class. The Neuron class stores weights and a bias, a boolean denoting whether or not to use an activation function in the neuron, and two vectors which store a value for each sample in the training dataset: one for the weighted sums, and one for the outputs after passing through the activation function, both of which are used in backpropagation. The constructor initializes the weights and bias randomly using He initialization, storing a single bias and a number of weights corresponding to the input dimension, which is given as a parameter. The layer class simply initializes and stores a vector of neurons with a given input dimension, of a given length. The MLP class stores the input DataFrame, a vector of inputs for each layer in the network for each sample in the dataset, a vector of Layers as hidden layers, and a single Neuron as the output layer. The forward and backward pass functions are designed for a single sample; the train function calls these for all samples in the dataset however many times are specified in the constructor. The forward pass function first computes the output of each hidden layer and stores it as the input for the next layer, then computes the predicted value using the single Neuron in the output layer. The backward pass function computes the error between the predicted and observed values for a given sample, backpropagates the error through every neuron in every hidden layer, and subtracts the product of the backpropagated error, the input with respect to a given weight and the learning rate from that weight to update it. This product (without the learning rate) is equivalent to the partial derivative of SSE loss with respect to the given weight. I have included a written proof of the gradient update rule I coded in the project folder. The file containing the proof, titled "BackpropagationProof_ComplexityAnalysis.pdf" includes a proof of the backpropagation algorithm I used, some explanation of the notation, time complexity analysis for different stages of training a multilayer perceptron, and a sketch of the network.

ConvolutionalNeuralNet.h includes only a ConvolutionalLayer class and a CNN class. The convolve function in the ConvolutionalLayer class was written specifically for the problem in this project and uses a one-dimensional filter to convolve over each antenna separately. In hindsight, this was the wrong approach, but I had not experimented with convolution approaches yet at this point in my work. The ConvolutionalLayer class is relatively similar to the Neuron class in MLP.h in its initialization of weights and biases. It also includes the convolve and flatten functions. The CNN class is very similar to the MLP in structure; it only includes separate vectors for the convolutional layers and fully connected layers, and their respective inputs for each sample. The portion of the backward pass function which is supposed to backpropagate the error through the convolutional layers is unfinished; it was at this point that I decided to use tensorflow instead of manually creating my own neural networks.

After experimenting with many different kinds of models, it seems that using a convolutional neural network is the best approach to both the regression and classification problems. Simple linear regression could not provide meaningful results for the X or Y position, while logistic regression and KNN provided somewhat and considerably better results respectively, but still performed worse than the CNN did. In terms of efficiency, the CNN was excellent, training about 3 epochs per second. The linear and logistic regression models were also fast, training about 100 iterations per second. The KNN model was much slower, taking about 3 minutes to run. The slowest part of the program, however, was reading the text data into c++, which took about 4 minutes. This is a strong disadvantage of using c++ to create models for data which was formatted using python.

The fact that convolving over all antennae for each subcarrier index produced a highly accurate model could imply that the true features which correspond to a given position or LoS state are independent of which antenna they appear on, only that they appear on an antenna. The relative success of the classification models to the regression models suggest that the channel state information can reveal whether or not a car is in line of sight with near perfect accuracy, and can predict its position with very good accuracy. It would be interesting to see if including LoS/nLoS as a parameter in the regression model would produce better results.