Neural Networks, Regression, and Spectroscopy: Column Densities in IllustrisTNG

Group C: Annelia Anderson, Nethra Rajavel, Jacob Morgan

Introduction:

The field of astronomy has been inundated with data for the last two or more decades. Large all sky surveys and other automated observations help to create petabytes of data per year. Although this is a boon to the field, it raises questions about how to handle such a large amount of information. Crowdsourcing projects such as Galaxy Zoo are popular. However, data production has begun to outpace this as well.

Data mining spectral features from large surveys has become important. Machine learning algorithms are used to identify and classify separate astronomical objects found in the spectra of patches of sky. We want to do something similar, but aimed at regression.

The circumgalactic medium refers to a region outside the stellar population of galaxies. Observations show that although the region has no stars, it does have a large amount of gas, metal, and dust. Current models of galaxy formation posit the CGM to be an important reservoir of star forming material. Thus what it is made of, how fast it cools, how dense it is, etc are all of interest.

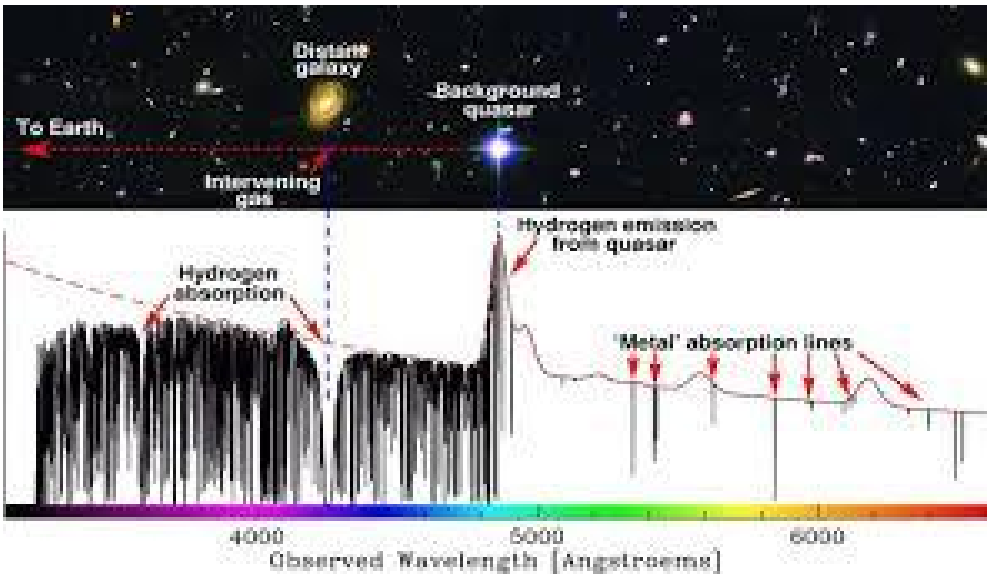


Figure 1: Two galaxies are required to observe the CGM in this manner; the galaxy with the gas of interest, and a more distant galaxy hosting a quasar which shines brightly through the nearer system. The gas created spectral features as the light travels through it, leaving information about its state.

However, the gas in the region is very diffuse, creating almost no light. The CGM is rarely observed directly. Instead, we rely on “backlighting” from quasars (see Figure 1). However,

these quasars only provide a tiny sampling of all the gas in the CGM. Observers then rely on complex rules relating spectral lines and gas properties to constrain a model of the gas.

One important feature of interest is the column densities of certain elements or ions. In this paper, we focus on hydrogen, carbon, nitrogen, oxygen, and iron. It should be noted here that in theory, the column density of ions undergoing some line transition can be directly calculated by the strength and width of the line. This means that an observer with access to every line for some ion could find the true column density. However, observers never have every line. Instead, they use knowledge of the overall density/temperature of the region and relationships between lines to constrain a model of the gas. Our model also uses only the lines found in 900-2000 angstrom, but is expected to find the column densities not just for an ionic state, but several whole elements. Then our network must learn and use the complex spectral relations internally.

Despite the large amount of publicly available spectral data, we opted to simulate our own. This gives us access to the “true” state of the gas, so we can directly compare our “observations” (predicted values) with the “true” values at the end.

To generate our data, we use simulated gas in the Illustris TNG100 simulation. This hydrodynamical simulation is ~100Mpc on a side at z=0, which is a large enough volume to support thousands of galaxies. The simulation records gas properties such as bulk velocity, temperature, and metal content. We then use a python package called yt to organize this data into sightlines. Essentially, yt is a visualization tool we use to create thin cylinders of gas particles. It is here that we calculate the column densities of our elements (our y-values). Then, we use the package Trident to convert the properties of gas in the cylinder to spectral properties (our x-values). We also record information about the position of the line of sight and the host galaxy. Details on the data collection process are below.

Data Production and Preparation:

We compile a list of 4496 halos in the TNG simulation. A halo consists of a central subhalo (“main galaxy”) and satellite subhalos. For each, we fire a simulated “quasar beam” through the gas along each principal axis of the simulation. We use impact parameters between 5-20 ckpc/h. This probes up to the outer CGM of small halos, but hits the central galaxy in some larger ones. Each instance in our data is a single array of spectral data and information about the host halo and LOS. The spectral data consists of 110001 relative flux values for a wavelength range 900-2000 Å. The halo data are physical parameters of the simulations, consisting of virial radius, stellar half-mass radius, impact parameter, halo mass, and the stellar circularity fraction for the central galaxy. Together, this leaves us with an array of 110006 values for the X-component. The data types are combined into a single array for simplicity, and because a neural network will be able to find the importance of features automatically.

The y-component of our data is an array of column densities along the LOS. We chose to track the column densities of hydrogen, carbon, nitrogen, oxygen, and. We also track neutral hydrogen separately. These elements were chosen because HCNO are all commonly used to observe the circumgalactic medium. In Figure 2 we see a depiction of gas in a simulated CGM (grey) along with ions that are commonly found at that density and temperature. Similarly, our wavelength range was chosen to be similar to the COS G130M spectrograph, and our redshift (z~0.15) was chosen so that the UV lines given off by HCNO fall in this range. We also track iron because it is associated with supernovae, and both it and neutral hydrogen are strong indicators of star formation.

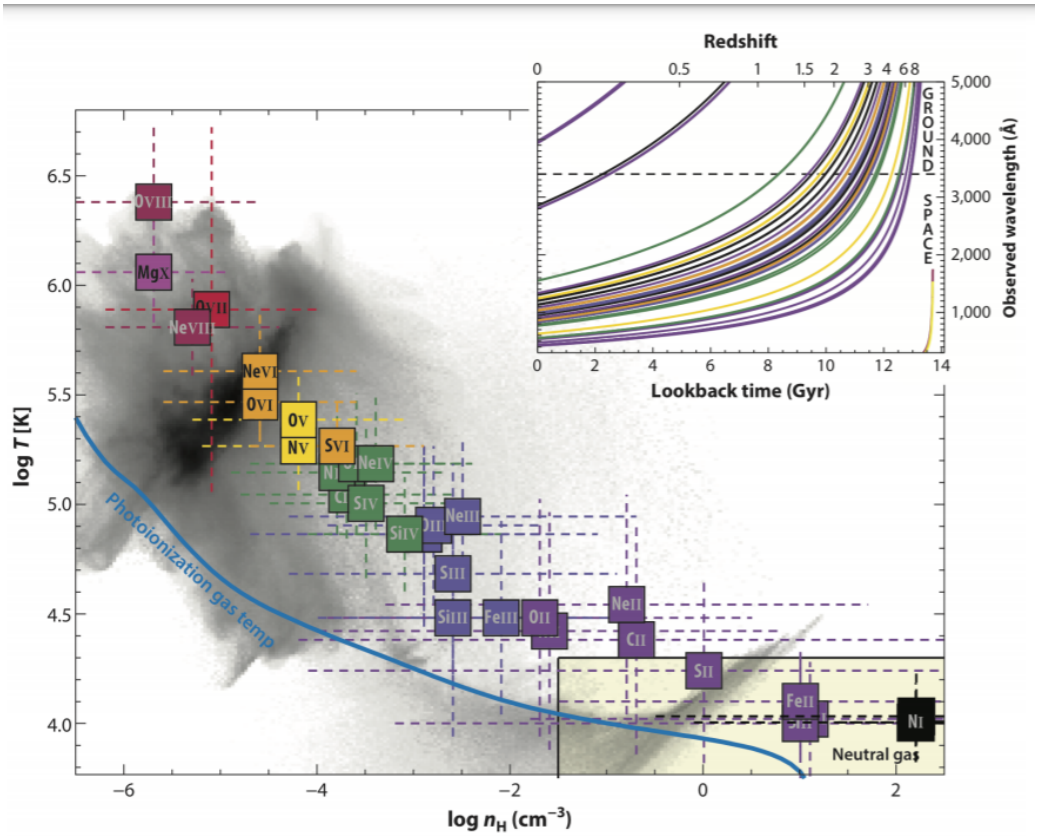


Figure 2: Phase plot of gas from the EAGLE simulation with commonly observed ions overlaid. The dotted lines represent the approximate range of density/temperature the ion is found at. Lower ions are purple, higher are red.

Both the X and y components were centered and scaled using StandarScaler. Then the data was prepared separately for two different runs:

1) No further preparation
2) Took the logarithm of the column densities

The logarithm was taken so the model could work with values that were closer numerically.

Batch Loading:
Because the files are so large, they need to be loaded in batches to prevent runtime crashes in Colab. However, using the maximum number of files per batch is preferred, because the files in a batch are trained together. We were able to use 10 files at a time for the model using log10, but only 5 at a time for the other.
This was essentially manual mini-batch training. Five (or ten) random files were loaded at a time, the data was scaled and shuffled. By using the random seed argument in sklearn, we were able to ensure the same random validation data is created when using test_train_split for: A) scaling, B) autoencoder training, and C) model training. Ten percent of each file batch is allocated to a total validation set, that was appended to in each iteration. Testing data is separated beforehand to ensure purity.

Model:

Because the goal is to predict multiple column densities, we built a multi-output regression model. It is made of two separately trained models: a stacked autoencoder and a simple neural network. The autoencoder is trained first to perform dimensionality reduction while learning the important features and relationships in the spectral and halo data. Next, only the encoder is extracted and used as the base layers in the neural network that is trained to predict column densities. The architecture of each model is shown below.

Autoencoder:

Encoder:
Dense: 250 neurons, activation = leaky_relu
Batch Normalization
Dense: 40 neurons, activation = linear
Batch Normalization

Decoder:
Batch Normalization
Dense: 250 neurons, activation = linear
Dense: 110006, activation = sigmoid

Hyperparameters:
loss = mean squared error
optimizer = SGD (lr=0.01, momentum = 0.95)
epochs = 50
batch size = 32

Final Model:

Encoder (trained)

Batch Normalization
Dropout (rate = 0.2)
Dense: 6 neurons , activation = linear

Hyperparamters:
loss = mean squared error
optimizer = SGD (lr=0.01, momentum = 0.95)
epochs = 70
batch size = 32

Leaky relu was used after we encountered the dead neuron problem in early models. The alpha that we chose for leaky relu was 0.01. The other activation functions used are standard to the ones used in an autoencoder architecture. The hyperparameters that we tuned were the number of neurons on the first and last layer of the autoencoder, number of epochs, batch size, learning rate and optimizer. After trial and error, a small batch size of 32 seemed to give us low loss values. Similarly, 250 neurons in the first layer seemed to give us better results than a smaller value of 100 neurons. Decreasing the learning rate did not help our model because of which we let it remain at 0.01. We tried a different optimizer other than SGD such as adam.This ended up requiring a lot more computational power even though adam did much better while training. Therefore we stuck to using SGD. We increased the epochs to 70 so that the model had enough time to train while also not overfitting on the data. We found the model worked better with more epochs, but did not increase this above 70 as we were concerned with overfitting.

Results:

We are mainly using loss and plots of true vs predicted column density values to evaluate the models. The true vs predicted column density plots can help us understand how the predicted data correlates with the true data. Better predictions are indicated by values near the line. Six plots are made, for each elements' column density (H, C, N, O, Fe, and neutral hydrogen). The loss is estimated by the mean squared error since this is a regression problem. Our initial run with the autoencoder model as described in the Model section but with 100 neurons instead of 250 produced a train loss of ~0.1 and test loss of ~0.15 showing that our model is indeed overfitting. For this case, the plots of the true vs predicted column densities for H and neutral hydrogen were the only one that fitted well with all the points along the line.

One of the main observations made here was that the predictions of C, N and O looked fairly similar implying that they may implicitly depend on the same features. This is in fact expected since C, N and O are produced by stars of a similar mass. But our initial model was not able to predict the column densities for them as well as it did for H and neutral hydrogen because C,N and O had column density data points over a great range of values. To combat this issue we considered scaling them down using a logarithmic function. This helped our output greatly. The true vs predicted plots for H, neutral hydrogen, C, N and O greatly improved with all (or most) of

the points falling on the line. But we do observe a trailing off/ tail sort of pattern in our pattern near the lower limits of our data indicating that there is less data near the lower limits. Hence we expect this.

In general predicting iron column densities was the most difficult. Despite the values for iron having a similar range as carbon (log(N)~8-17/cm^2) for the majority of sightlines, it is the only element for which there are sightlines with none. This seems to have caused a variety of problems, some of which can “bleed into” predictions of other elements!

Our initial, non-log model performed badly on iron and high column densities for all elements. This is because the vast majority of instances have low column densities-- however, it is the sightlines with high values that are interesting. We tried taking the logarithm of our y-values before any other data processing to alleviate the difference in scale between high and low values (~10^{22} vs 10^{13}). It was then we discovered that certain LOS had absolutely no iron-- we replaced these zero values with log(N)=-100. This caused our model to fit high column densities well, but each element besides iron had a “tail” at low column density where the model became inaccurate. Additionally, the model became terrible at predicting iron values. Figure 2 below shows an iron and carbon plot from this model. Specifically, it would predict values for iron far above any it had seen in training or validation (log(N)~22 vs true values of log(N)~17). The badly predicted iron values would make for good predictions of hydrogen. It seems the model can, “mix up” elements, with the bad iron fit being connected to the bad fits at low column densities for other elements. However, this problem *seemed* fixed in our final model, which uses a replacement value of log(N)=-1. See Figure 4. The most important problem we solved in this project was recognizing the important differences between completely null values and low values.

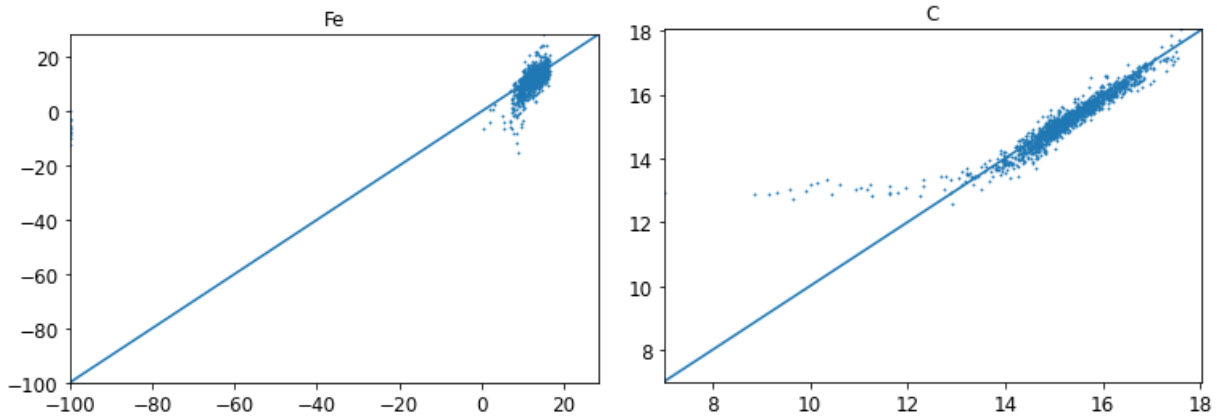


Figure 2: True values of log(N) (x-axis) vs predicted (y-axis) for our log-model using 10^{-100} as a stand in for 0 for iron.The scale is due to manually setting the value of iron so low. It makes the plot hard to see, but a better view mostly shows a blob anyways. We see that this causes not just an overprediction of iron, but also of other elements at low column densities.

When our log model initially performed badly, we decided to make two runs of our models-- one with the log function and one without. This would help us compare how much the model improved after scaling with the log function. The evaluation of our models can be seen in Table 1. And Figures 3-4.

Loss	No log	With log
Autoencoder loss	0.9427	1.0155
Training Loss	0.0447	0.055
Val Loss	0.10805	0.055
Test Loss	0.17	0.18

Table 1.

As expected, we observe two things: (1) Run 1 seems to be overfitting a lot compared to the other. (2) The one with the Log seems to have a much smaller training and validation loss. This can only indicate that the log function has helped in decreasing overfitting and also helped increase our model's performance overall.
It can be noted that our autoencoders do have high loss values than we would like them to be for both the runs. Maybe a reduced autoencoder loss could have given us an overall better performance with the test loss too.

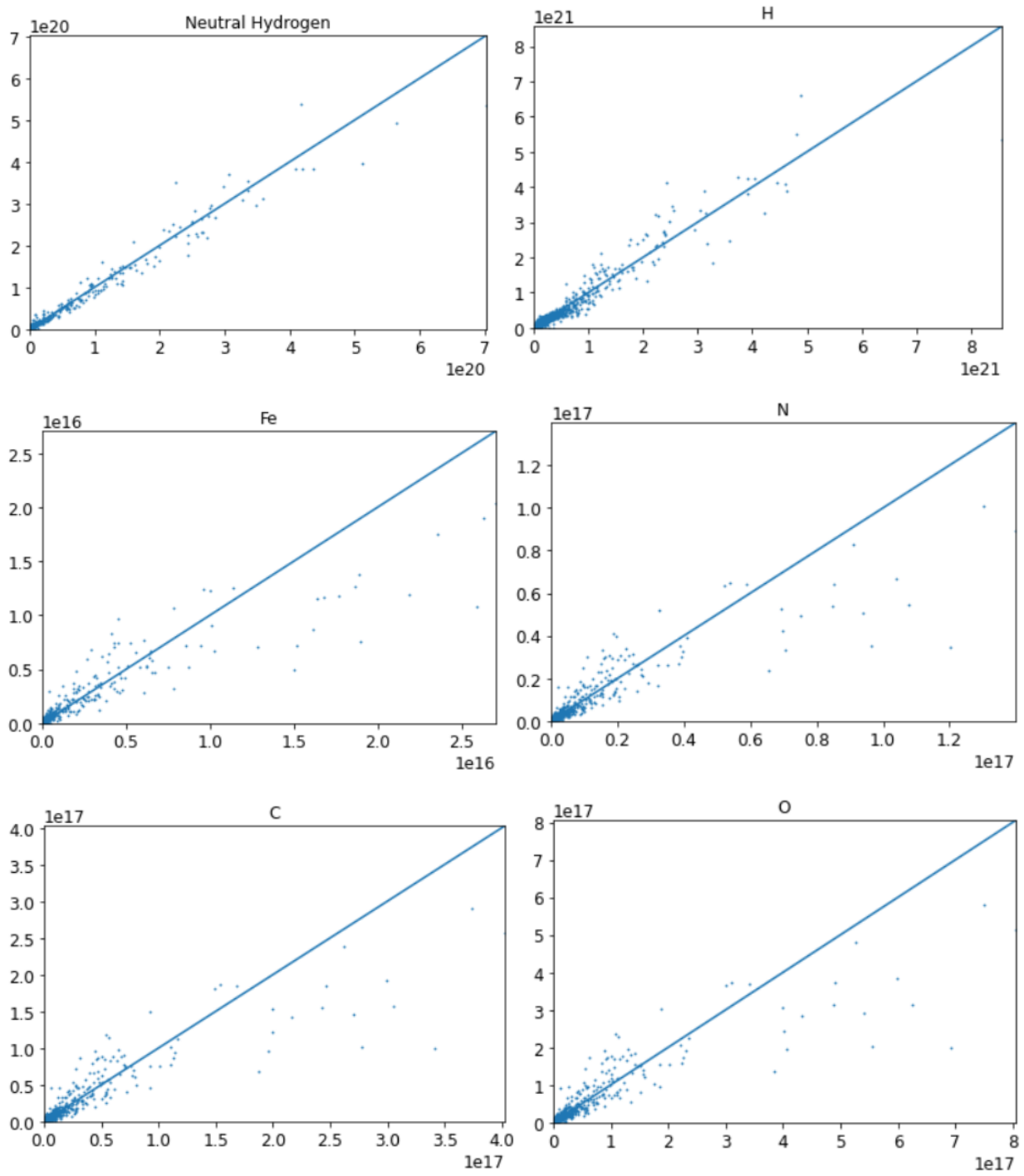


Figure 3 : These plots represent the true vs predicted plots from Run 1: no log

Similar to our initial run, Figure 3 shows that higher order values in C, N, O and Fe are not fitting as well as the lower order ones but H and HI are fitting pretty well. Even though H and HI are predicted well they're not predicted as well as we would like them to.

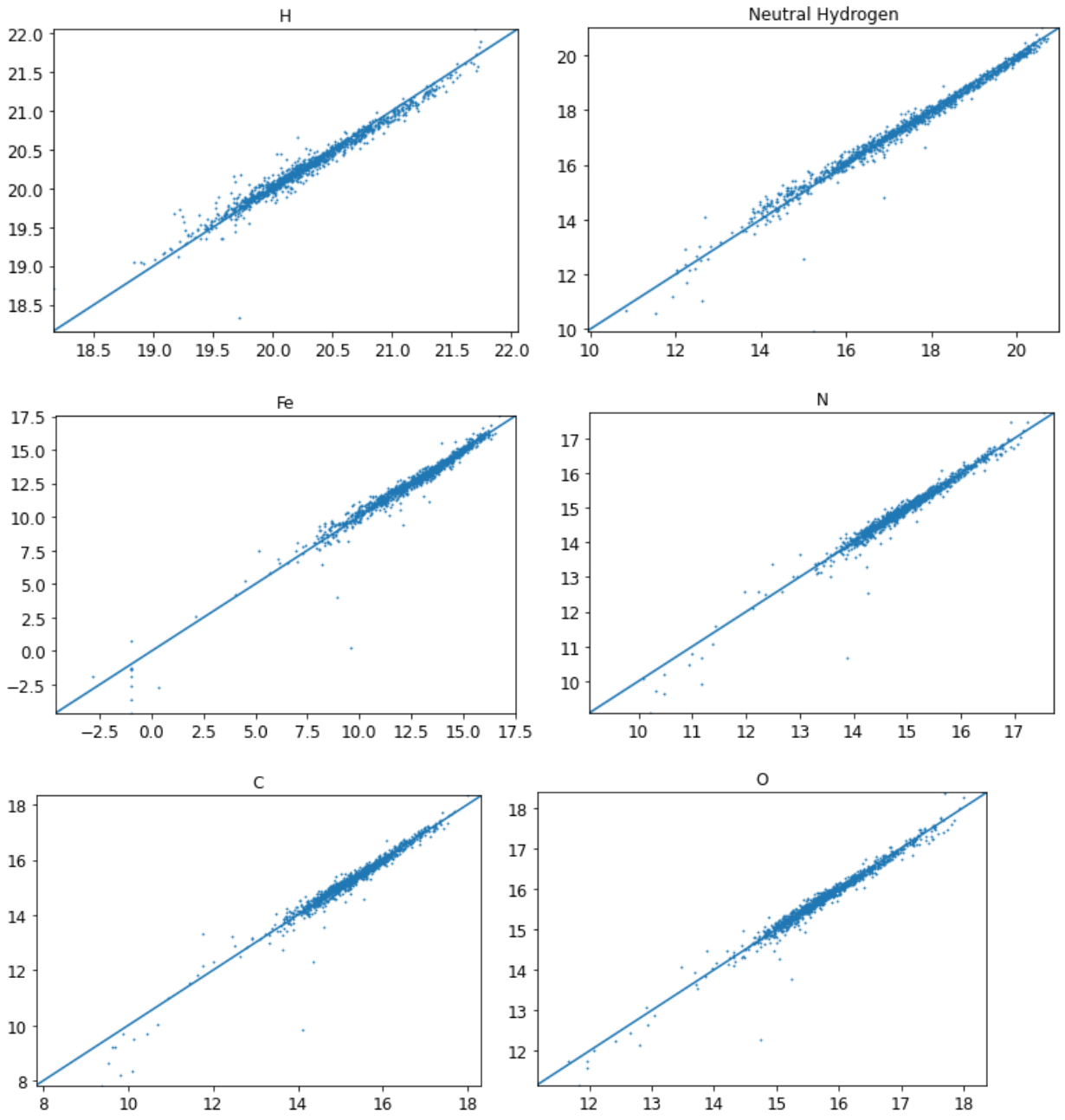
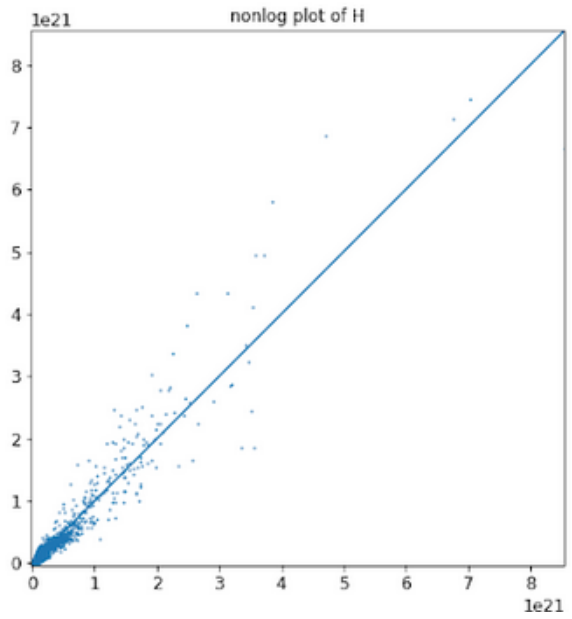
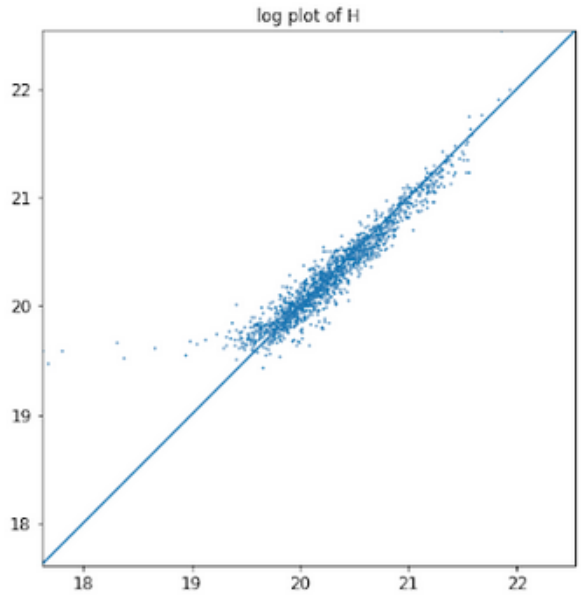


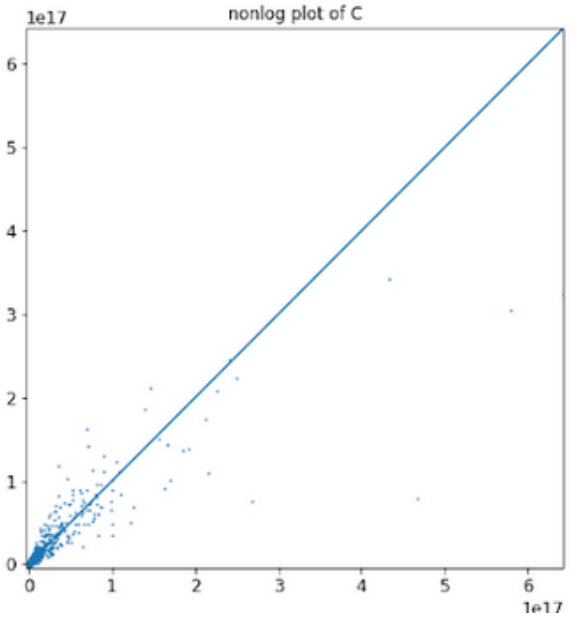
Figure 4: These plots represent the true vs predicted plots from Run 3: with log

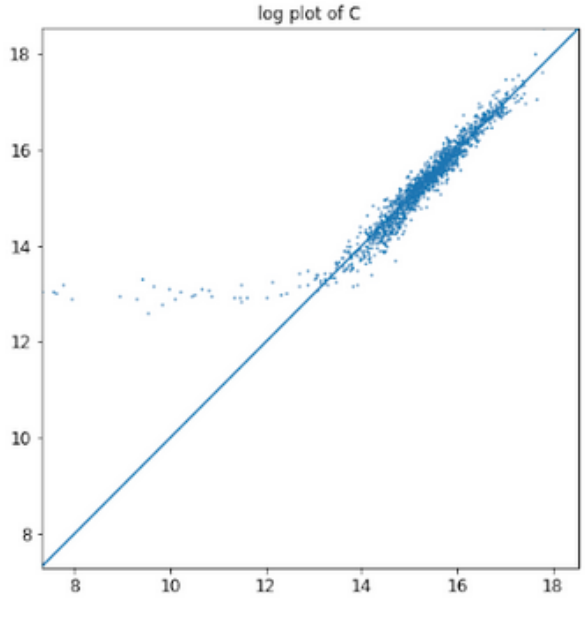
The plots in Figure 4. All show how well all the column densities are predicted. This is a drastic improvement from the previous plots of just good prediction for 2 out of 6 column density ions.

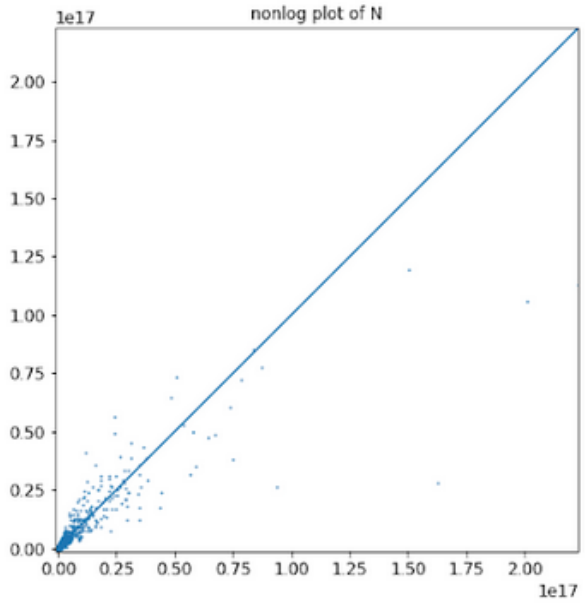
However….the final test data was not fit nearly as well as the validation data. Iron is still better than in any previous model, but the low column density “tails” are all back. The training, validation, and test losses are listed in Table 1. A comparison of our final log and nonlog models on the test data are below in Figure 5.

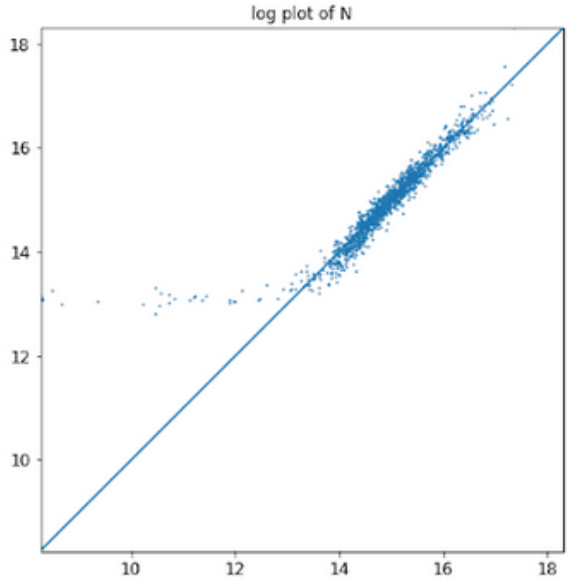

nonlog plot of H
1e21
0
1
2
3
4
5
6
7
8
1e21

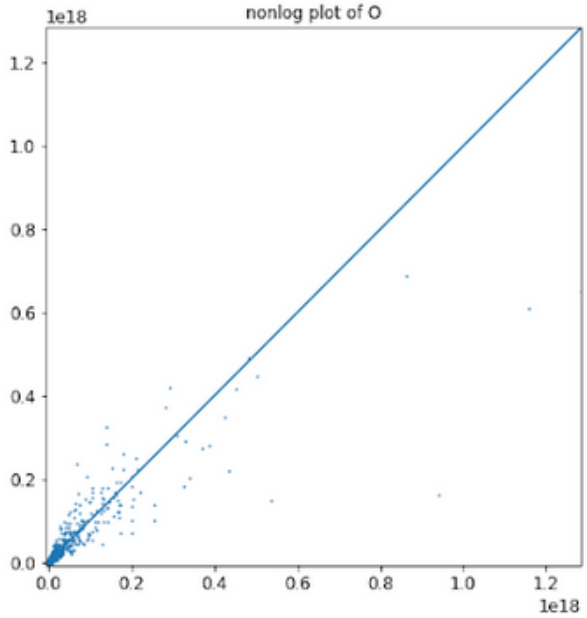

log plot of H
18
19
20
21
22

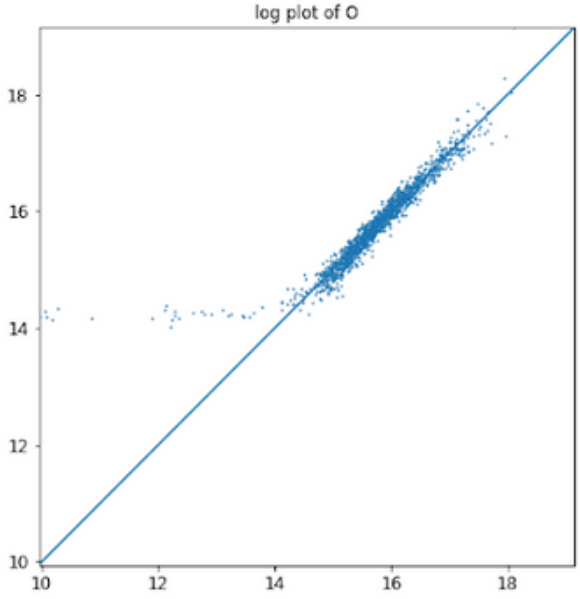

nonlog plot of C
1e17
0
1
2
3
4
5
6
1e17


log plot of C
8
10
12
14
16
18


nonlog plot of N
1e17
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
1e17


log plot of N
18
16
14
12
10
10 12 14 16 18


nonlog plot of O
1e18
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0 1.2
1e18


log plot of O
18
16
14
12
10
10 12 14 16 18

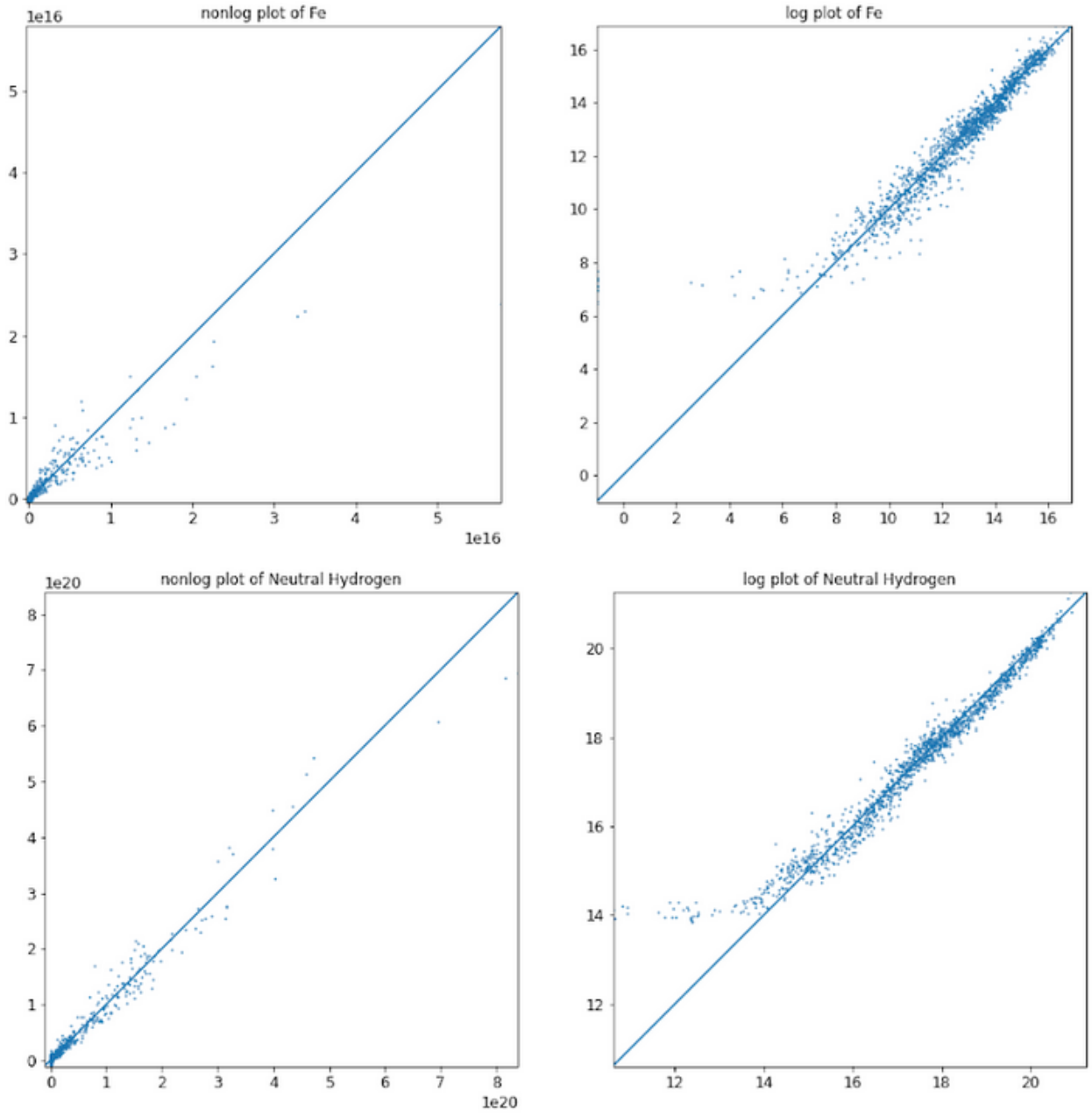


Figure 5: Comparison of our two final models. Although the loss is "better" for the nonlog model (test loss of 0.17 vs 0.18), the log model predicts the high column densities of more interesting systems better. The ability of our nonlog model to get the loss so low likely comes from the number of low column density sightlines providing smaller values of loss when predicted badly.

Conclusion:

To conclude, we are able to produce a working neural network model that can predict column densities of H, C, N, O, Fe and HI well compared to the uncertainties quoted in observations. It is unfortunate that the “low column density tail” re-occurs in the test data. We did not have the opportunity to tune our zero-replacement hyperparameter for the log model-- it’s possible that log(N)=-1 is still too small, being ~9 orders of magnitude lower than any column densities actually found. Given that the low column density tail first disappeared in the validation step after changing this value, but reappeared in the test data while still fitting iron well, it seems likely that a change from say, log(N)=-1 to log(N)=2-3 may alleviate the problem more generally. I believe we would have discovered the same problem in the validation step eventually, but in truth we were only able to try our final model on 2 different random validation sets. Given more time, this is the first thing we would do, followed by an analysis of the spectral data without line of sight or halo data. Although observers use those properties, it may be possible to estimate the column densities from lines alone.

Link to codes:
No log:
https://colab.research.google.com/drive/1o_vIwzM3t7316XpqWKwTXrFpH4iIMuKD?usp=sharing
With log:
https://drive.google.com/file/d/1pj_UmpkptHu9TwCSlBygDBh-9avlKKRe/view?usp=sharing
Run on test set:
https://drive.google.com/file/d/1NLArb0tFcS26NL8Bgnf2w5kXLkKaww0C/view?usp=sharing