Quick Notes about Machine Learning on Intensity Maps

Daniel N. Pfeffer

Abstract

The idea of this project is to use machine learning on intensity maps to determine the luminosity function of the underlying halos.

1 Jump to Section

2 Intensity Maps Background

Intensity mapping is done by looking at a given emission line. Whatever is being traced should be emitting this line at any location where the tracer is located. Having a higher density of the tracer would cause an increased intensity of whatever line is being looked at. As the light travels to Earth it will get redshifted based on where it was originally emitted. By looking at a range of frequencies one can get 3D spatial information (maps) about whatever tracer is being looked at.

George has code to generate different halo catalogs quickly and has done so to make (as of the time of writing this part) 161 halo catalogs. Each of these catalogs can be converted into smaller subfields as well as rotated to produce more catalogs. With another code of George's one can convert these halo catalogs (or regions in them) into intensity maps. We want lots of intensity maps so that we can do machine learning on the maps to determine the underlying luminosity function.

3 Machine Learning Background

I'm feeling lazy right now and don't want to fully flesh this out yet.

Machine learning can be used for lots of tings if you throw enough data at it.

Neural networks are supposed to represent how brains and neurons work. It is trained for a specific task and each neuron has it's own weights. This gets very memory intensive for large networks because there can be lots of neurons. A way around this is to use convolutional neural networks (CNN). A CNN has filters that convolve with layers of input or neuron output and each layer has the same filter which saves on memory. A quick Google showed these links that explain CNNs both in depth (http://cs231n.github.io/convolutional-networks/) and at a surface level (https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050).

4 Intensity Mapping CNN (Actually on N not dN/dL)

As of right now the idea is to use a CNN on simulated intensity maps to determine the luminosity function of the underlying halos that made the simulated intensity maps. George has code to make the halo catalogs and another code to convert the catalogs into intensity maps. I've made code to split up the catalogs into smaller subfields to match possible experiments. I also have code that will rotate the halo catalogs before making subfields so that we have more subfields to train out network with. George's code limlam mocker (llm) (the code that converts catalogs into intensity maps) was modified by me to also give out the luminosity function of the underlying halos. The llm can also use different underlying halo luminosity relations to generate different maps and luminosity functions.

As of the basic training right now I am not doing anything to split up the maps into training, validation and test sets. I'm just seeing the general results of different things right now.

4.1 log dN/dL

Originally the CNN was trained on converting an intensity map into $\log dN/dL$ instead of just dN/dL. This was done originally to prevent having such a large range of output values could be hard to train.

As of right now I have a trained network that is 4 layers that trained for 100 epochs of 400 maps apiece. Each layer is a 3D convolution with kernel size 5 and stride 1 followed by a max pooling 3D of size 2 and stride 2. The first layer has 32 filters, followed by 64, 128 and 256. Following the convolutional layers the network is flattened and then has a dense layer with 1000 neurons. The final layer is another dense layer with a neuron for each point in the luminosity function we want. Currently I take 50 points of the luminosity function. The loss function was just the mean square error function.

Should get plots for accuracy and loss as a function of time, but forgot to add that functionality to the CNN when I first ran it. The network took under 30 hours to train. Figs. 1 to 3 show the result of the CNN on some random map. In all luminosity bins before around $L = 10^6 L_{sun}$ the CNN luminosity function is within a couple of percent of the underlying one. After $L = 10^6 L_{sun}$ the ratio of CNN luminosity function to the given luminosity function jumps up to around 1.6. The specific values change depending on what map is used, but they all show the issue at large luminosities.

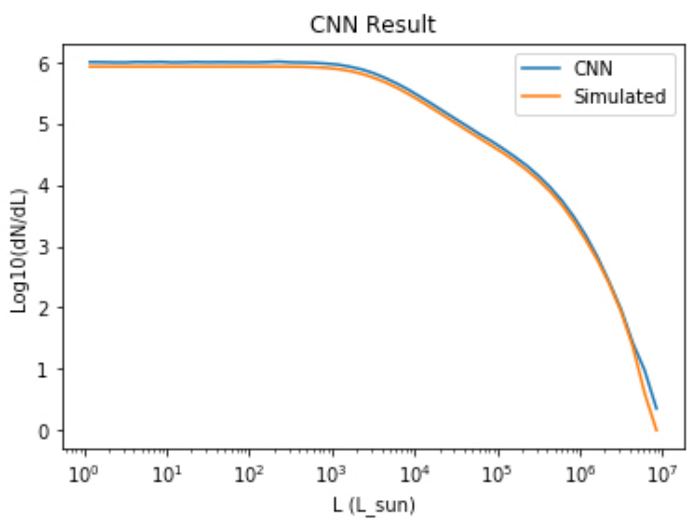

Figure 1: Plot showing the comparison of the output of the 4 layer CNN to the expected result of the underlying luminosity function that made the intensity map.

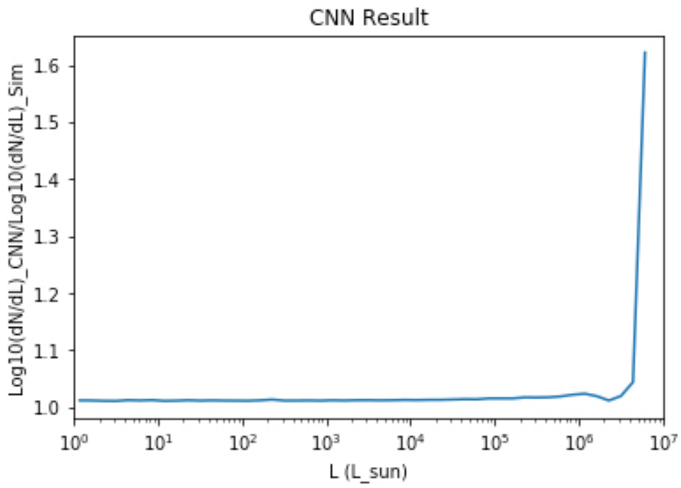

Figure 2: Plot showing the ratio of the CNN luminosity function over the underlying luminosity function.

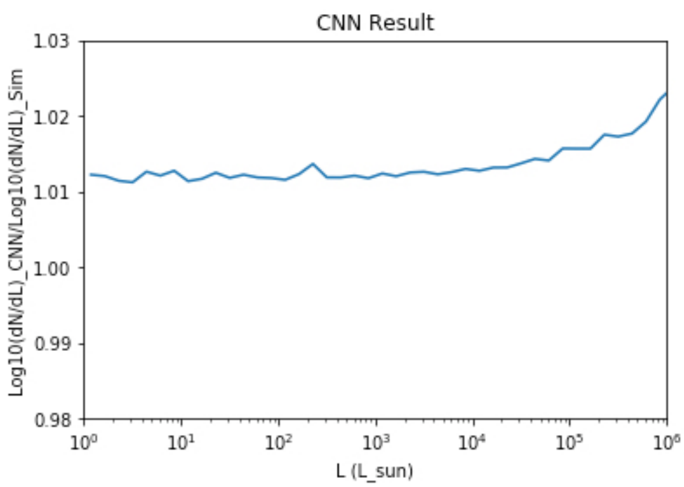

Figure 3: Zoom in of Figure 2 showing the ratio of values before $L = 10^6 L_{sun}$.

We are probably interested in the actual values of things and not the log value so I checked the accuracy of the values instead of the log values. The same set of plots but without logs are shown in Figs. 4 to 6. What can be seen is that the error increases by at least an order of magnitude. The error is still only around 20%, but is still much larger then in the log case. It can be shown (but I'm too lazy to type it out now) that the ratio of the unloged values is actually given by

$$\frac{dN}{dL}_{\text{CNN}} / \frac{dN}{dL}_{\text{sim}}(L) = 10^{\log\left(\frac{dN}{dL}_{\text{sim}}(L)\right)*y(L)} \tag{1}$$

where

$$y = \log\left(\frac{dN}{dL}_{\text{CNN}}\right) / \log\left(\frac{dN}{dL}_{\text{sim}}\right)(L). \tag{2}$$

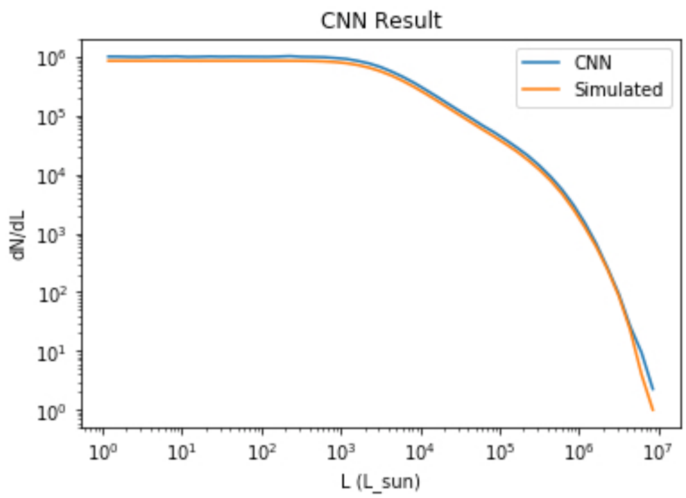

Figure 4: Plot showing the comparison of the output of the 4 layer CNN to the expected result of the underlying luminosity function that made the intensity map.

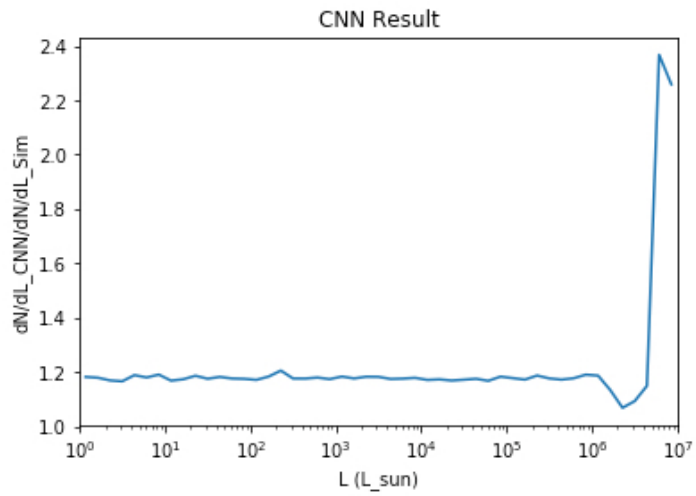

Figure 5: Plot showing the ratio of the CNN luminosity function over the underlying luminosity function.

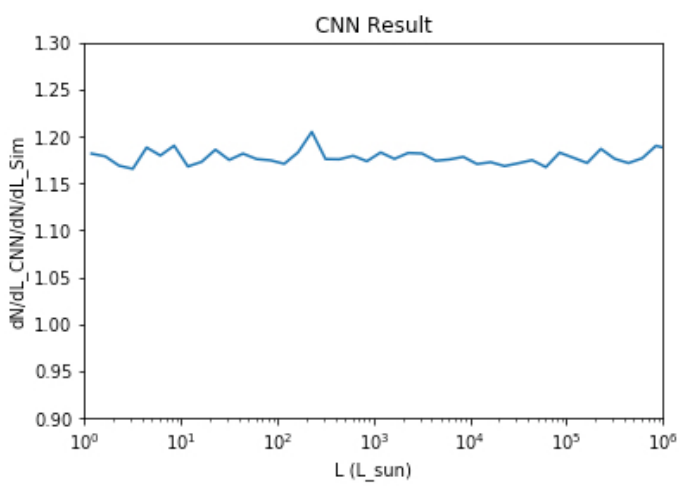

Figure 6: Zoom in of Figure 2 showing the ratio of values before $L = 10^6 L_{sun}$.

4.2 4 Layer dN/dL

Seeing that finding the log luminosity value gives errors around 20%, I tried to see if we could find the actual luminosity function directly. The first network was 4 layers and was trained for 100 epochs of 400 maps apiece. Each layer is a 3D convolution with kernel size 5 and stride 1 followed by a max pooling 3D of size 2 and stride 2. The first layer has 32 filters, followed by 64, 128, 256 512. Following the convolutional layers the network is flattened and then has a dense layer with 1000 neurons. The final layer is another dense layer with a neuron for each point in the luminosity function we want. Currently I take 50 points of the luminosity function. The loss function was the mean log square error function. Using mlse instead of mse makes it so that it doesn't get stuck trying to fit the region of low L and ignore the higher L regions where dN/dL is smaller.

I forget how long the network took to train, but it was under 48 hours. Looking at Figs. 7 and 8 one can see that the training did not go as well as it could. There are discontinuities in the outputted luminosity function, sharp spikes and nothing after $L \approx 10^6 L_{\rm sun}$. I'm no expert, but we would want something better then that. Figs. 9 and 10 show the training history of the loss function and another metric as a function of epoch while training 4 layer network. The shape of Fig. 10 does look like what one would expect from a loss function.

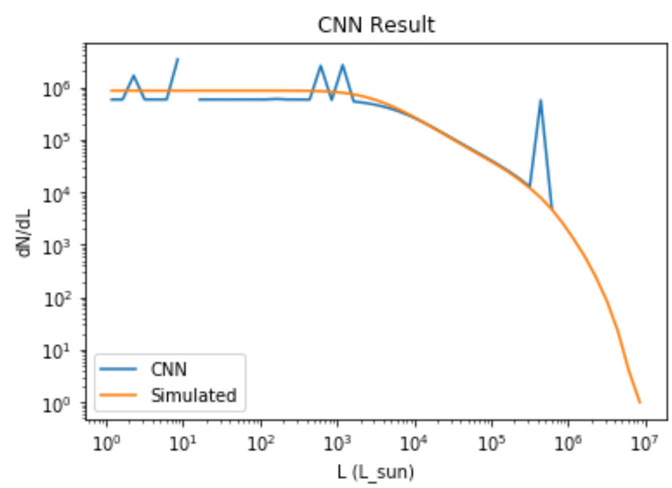

Figure 7: Plot showing the comparison of the output of the 4 layer CNN to the expected result of the underlying luminosity function that made the intensity map.

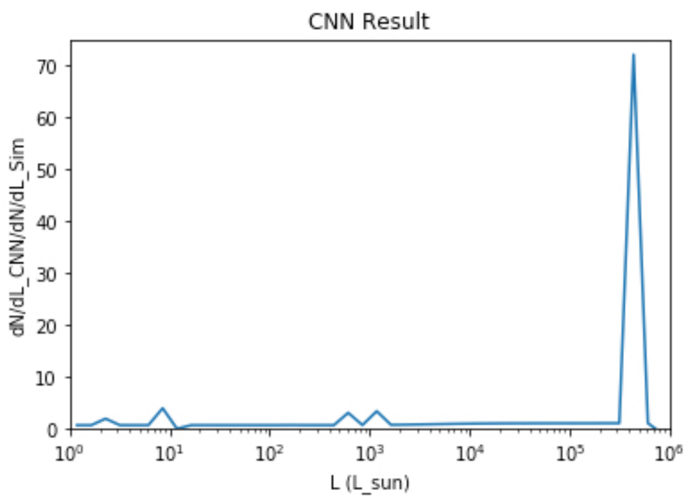

Figure 8: Plot showing the ratio of the CNN luminosity function over the underlying luminosity function.

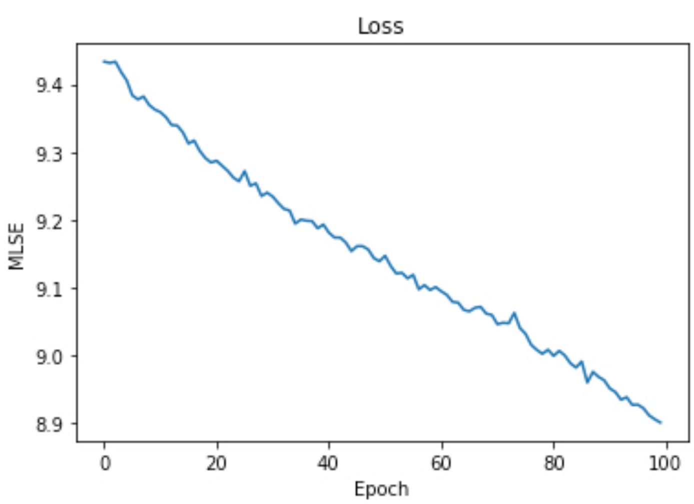

Figure 9: Plot showing loss history of the 4 layer CNN that was trained on the full luminosity function. The loss function that was used was the mean log squared error

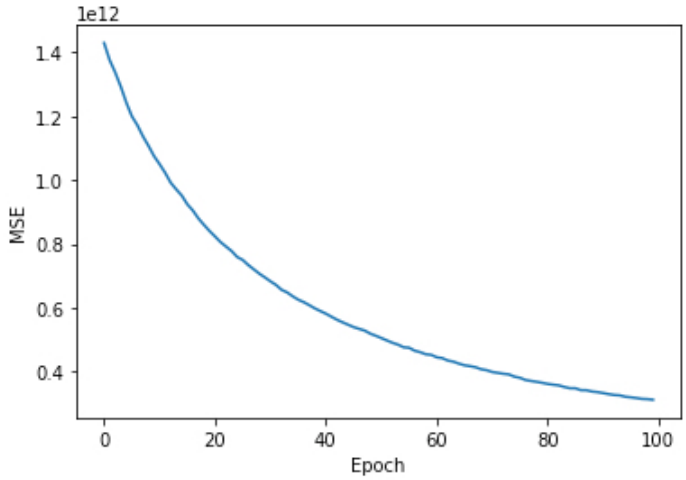

Figure 10: Plot showing history of the mean squared error metric as a function of epoch.

4.3 5 Layer dN/dL

I tried making a 5 layer network, but it was much slower and only got around 30 epochs in 48 hours. It was way too slow to train. I saved every 20 epochs so not everything was wasted. Fig. 11 shows how "good" the semi-trained network is and it is pretty bad. This needs more time. I'm trying to run it for longer now and see if it will hopefully turn out better then the 4 layer network.

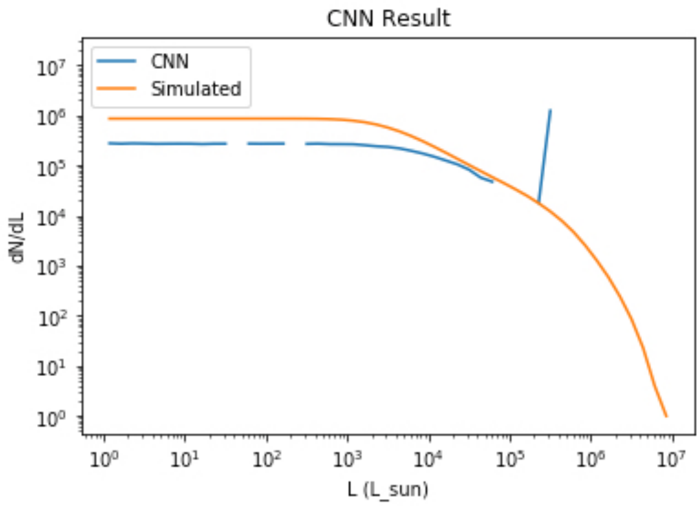

Figure 11: Plot showing the comparison of the output of the 5 layer CNN to the expected result of the underlying luminosity function that made the intensity map.

5 Intensity Mapping CNN (On dN/dL this time)

The previous shown work was done using the number count instead of the luminosity function.

$$N = \int_{L_*}^{\inf} \phi dL \tag{3}$$

Using the actual luminosity function instead of the number count gives worse results. I believe this is due to the fact that instead of something that is monotonically decreasing we have to worry about more features. I think if we converted this to a Fourier space analog and looked at power at different scale we would find more power at lower scales when looking at ϕ rather then N. In Fig. 12 we see for a given map the difference between the normalized number counts and normalized luminosity function. The luminosity function has more features. These extra features require more training on the part of the CNN. Training on number counts will be faster and produce better results then the luminosity function.

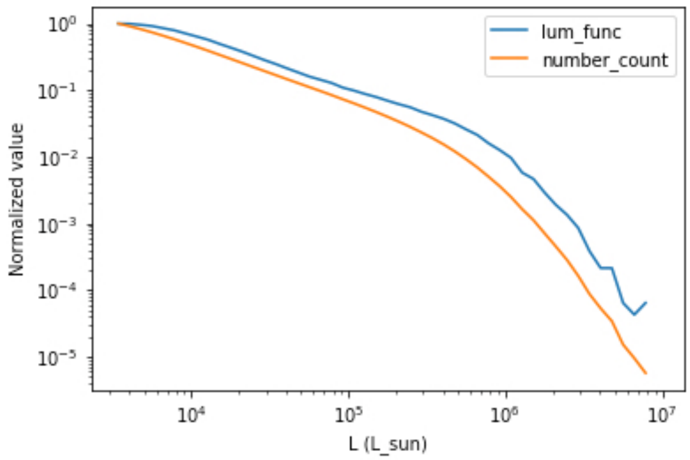

Figure 12: Figure showing the differences between normalized N and ϕ. The lum func line is $\phi(L)$ and number count is $N(L)$. The main thing to notice is that there are more features in the ϕ curve then the N curve.

While training models I get stupid errors that I don't really know every once and a while. It is some stupid cuda error which makes it very hard to debug and I can't reproduce them. Just going to throw one of them here so it is in the record. Line breaks were added by me.

```
F tensorflow/stream_executor/cuda/cuda_dnn.cc:521]
Check failed: cudnnSetTensorNdDescriptor(handle_.get(),
elem_type, nd, dims.data(), strides.data()) ==
CUDNN_STATUS_SUCCESS (3 vs. 0)
batch_descriptor: {count: 0 feature_map_count: 16
spatial: 252 252 96  value_min: 0.000000
value_max: 0.000000 layout: BatchDepthYX}
```

There were also issues with trying to train on ϕL. I don't know what the network was doing, but it would just start getting nulls, nans or infs for output pretty quickly or it would just give garbage in the end. It is probably something I'm doing, but maybe the architecture doesn't like the shape of ϕL as seen in Fig. 13. it is more complicated then either N or ϕ.

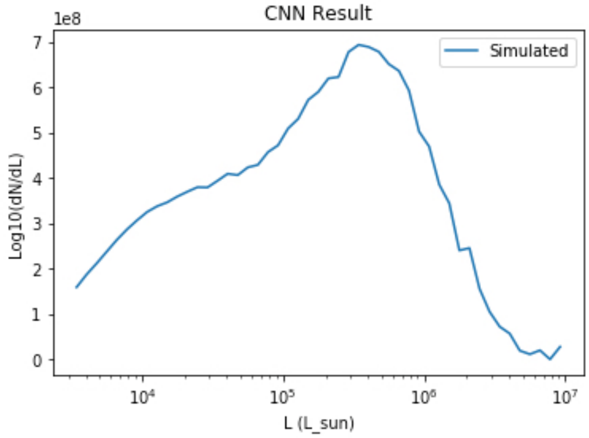

Figure 13: A generic look at what ϕL v.s. L should look like.

5.1 2D v.s. 3D

When designing the CNN we have a choice between doing 2D or 3D convolutions. 3D makes things much slower and requires more space. A comparison between 2D and 3D models can be seen in Fig. 14. From the figure it is clear that the 2D model beats the 3D one and the longer running 2D model does best. The 3D model just doesn't look good. I can't get it to actually work for the log values. I don't have a model for 5 layers due to it failing at some point while running.

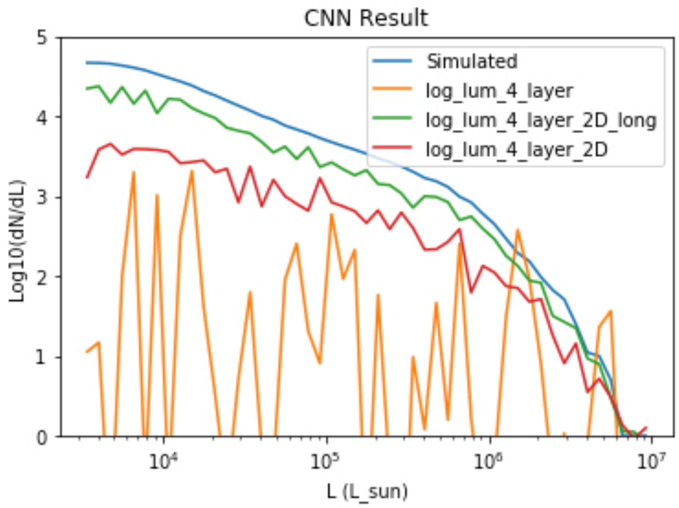

Figure 14: A comparison between three different models. For two of them the difference between them was that one was with 3D convolutions and the other with 2D ones. Their epochs were roughly the same number of evaluations and they had the same number of epochs. The long model was run with more evaluations per epoch rather then more epochs.

It seems, at least for the log values, that it is better to go with the 2D convolutions. It is faster, more space efficient and gives better results.

We can compare the loss history of the 2D and 3D models in Figs. 15 and 16 to try and see what's going on. The history for the 3D one is garbage. The validation loss never goes down. I don't fully understand what is happening. Naively this would mean that it is memorizing the training data and not knowing the validation data. The issue with this is that if I run the model on data that should have been training data it still returns garbage. The 2D model is better. The validation data does improve, but it stalls out around halfway through and doesn't improve much anymore. It does show learning though.

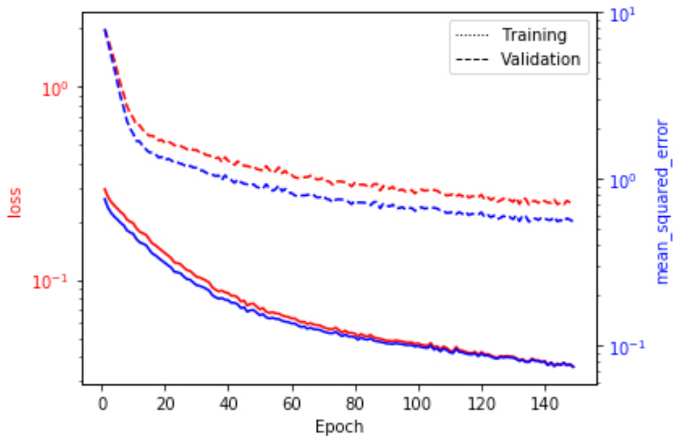

Figure 15: Loss history of the training of the 4 layer, 2D and log valued model. Note that training data is actually the solid line not what the legend says.

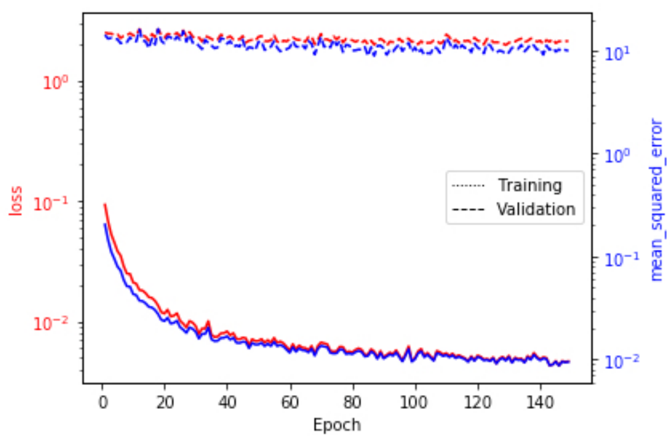

Figure 16: Loss history of the training of the 4 layer and log valued model. Note that training data is actually the solid line not what the legend says.

I don't have models in both 2D and 3D for testing against ϕ or ϕL to see how they differ between the dimensions.

5.2 What is actually working?

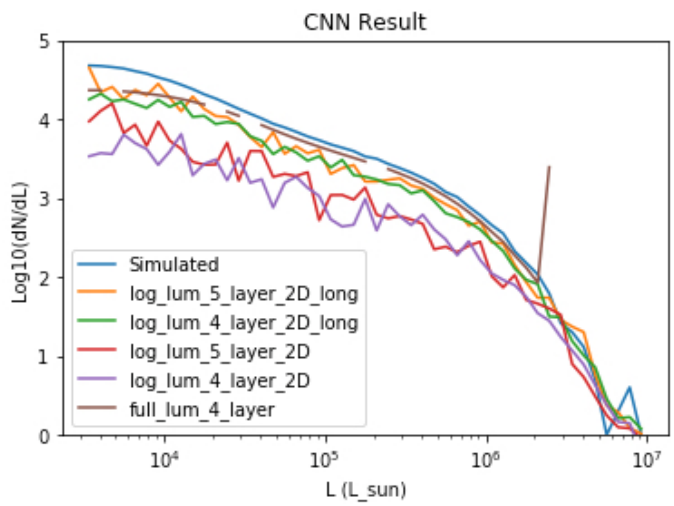

Figure 17: Comparison of the output from a few different models.

Now lets try to figure out what is working and what we should be continuing on with. Fig. 17 shows the output from a few different models for a single map. Note that not every model was trained on log data, but it is presented in log space now. A few things become apparent by looking at the curves. First is that the longer training models did better then the less trained models. This is seen comparing the 4 and 5 layer 2D log data v.s. the long version of those. Who could have guessed this? The next thing to notice is that the 4 layer network trained on the full luminosity function isn't bad. I would have expected this to be awful. The range of values it needs to get is very large, but it was one of the best trained models. It does have some holes in it's range which we saw earlier for a similar model in Fig. 7. I don't know why those things are appearing, but they do. Maybe more training will help?

More models were tested then what is in the figure, but the best were shown.

In Fig. 18 we can look at the ratio of model output to expected output as a function of L. In this figure we see that the 4 layer full luminosity function model is best at times with the 5 layer 2D log value one behind that and the 4 layer version of the previous model slightly worse off.

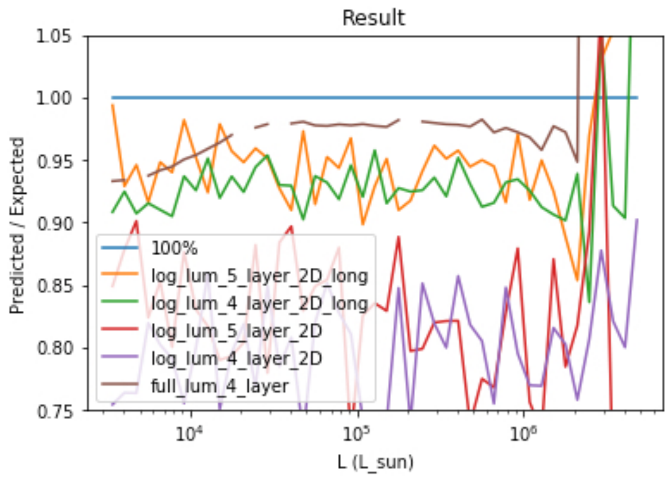

Figure 18: Comparison of the ratio of output to expected value from a few different models.

Again it is useful to look at the training history. When comparing the 4 layer and 5 layer histories in Figs. 19 and 20 we see that there is a big gap between the training in validation for 4 layers, but not for 5 layers. It might be that the network isn't big enough with 4 layers to learn as fast as it can. The 5 layer network learns very nicely and doesn't appear to be leveling out yet when the training ended. I don't know how, but somehow training in 3D gave a terrible history which can be seen in Fig. 21. The training and validation loss data are mostly the same after about 40 epochs, but it oscillates which is weird. The mse error improves for some reason even though it isn't being trained on that and the loss isn't really improving.

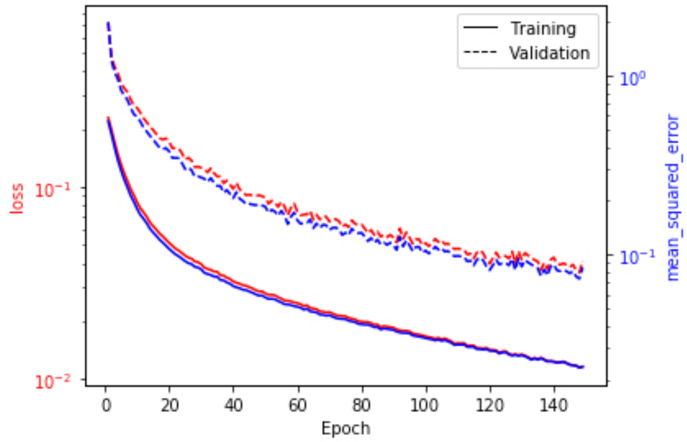

Figure 19: Loss history of the training of the 4 layer 2D log valued model.

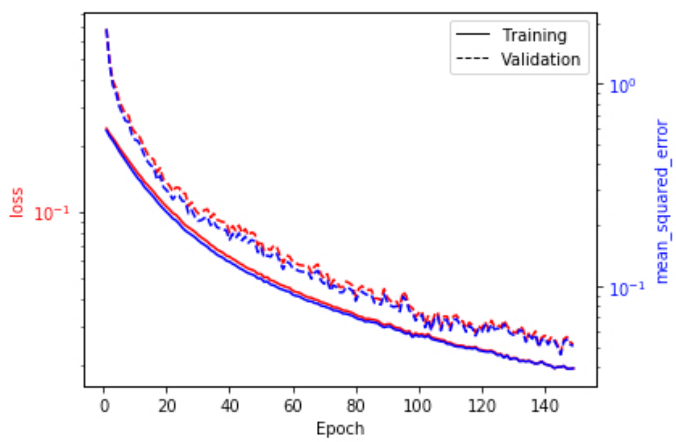

Figure 20: Loss history of the training of the 5 layer 2D log valued model.

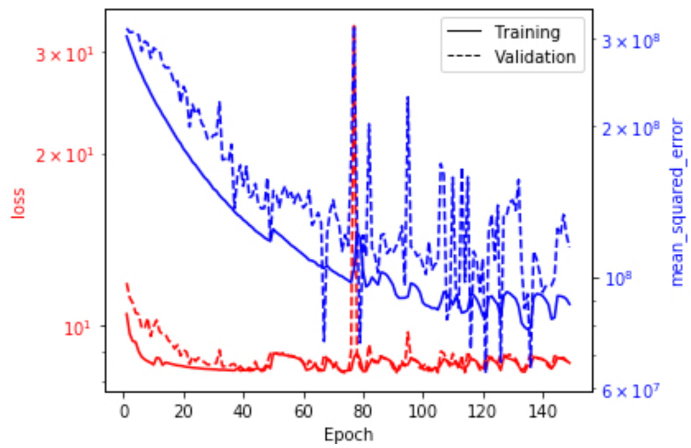

Figure 21: Loss history of the training of the 4 layer full valued model.

6 Finalizing the architectures

6.1 No Noise

The architecture of the model needs to be finalized. The last things to look at are really

1. log v.s. normal map input values
2. ϕ v.s. N
3. How many layers and filters can the gpu handle
4. Any changes to hyper parameters?

Note that I know realized that whenever I have been using ϕ in the second batch of testing I have only been using dN. I've only been considering the differential amount of sources in a bin. Because I am using log bins this should be the same as $dN/d\log L$ up to a constant amplitude difference.

Figs. 22 to 25 show the difference between models that vary, number of layers, number of base filters, log input v.s. normal inputs, dN v.s. N and kernel size for convolutions. The base number of filters was 32 and was used by everything but the most_filters model. The most_filters model used 128 filters, but ran out of memory when training for the second time so only trained half as long as the other models. Because there is a convolution layer (that acts as a pooling layer) between normal convolution layers there is a max number of layers. After 5 layers the remaining image is only 4x4 so convolutions using our original 5x5 filter just gets all information at once. The way around this is to use a kernel size of 3x3 instead. The 6 layer model uses 3x3 kernels. There was also a test of a 5 layer model with a 3x3 kernel.

The output for the model trained on N gives weird results when displaying $\log_{10} dN$ because it gets negative values that don't log well. What we see in these figures is that more filters and more layers are better then less although they end up at pretty much the same accuracy. We also see that the model trained on N is good at predicting N. Models are generally better at predicting N and with less noise then dN.

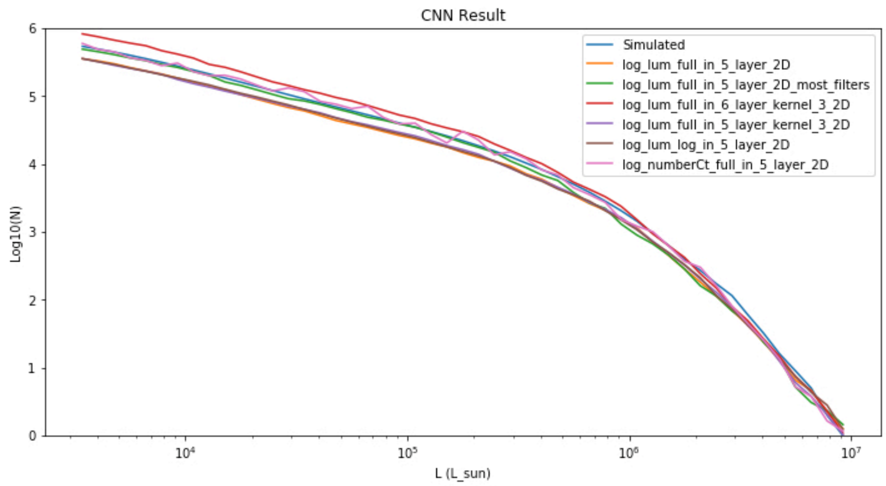

Figure 22: Output of different CNN architectures for N.

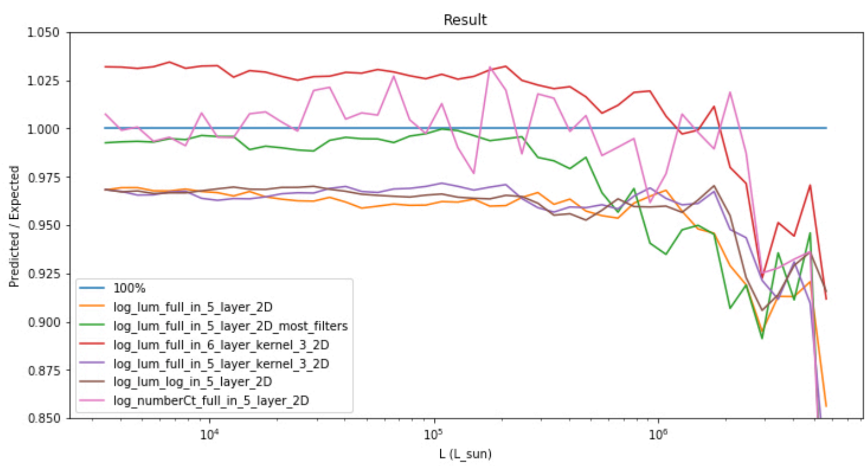

Figure 23: Same as Fig. 22, but showing ratio of CNN output to underlying value instead of raw values.

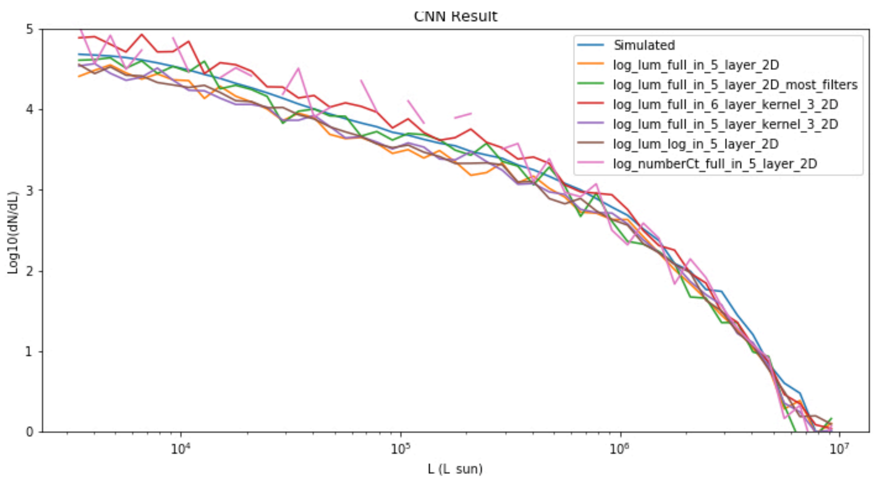

Figure 24: Output of different CNN architectures for dN.

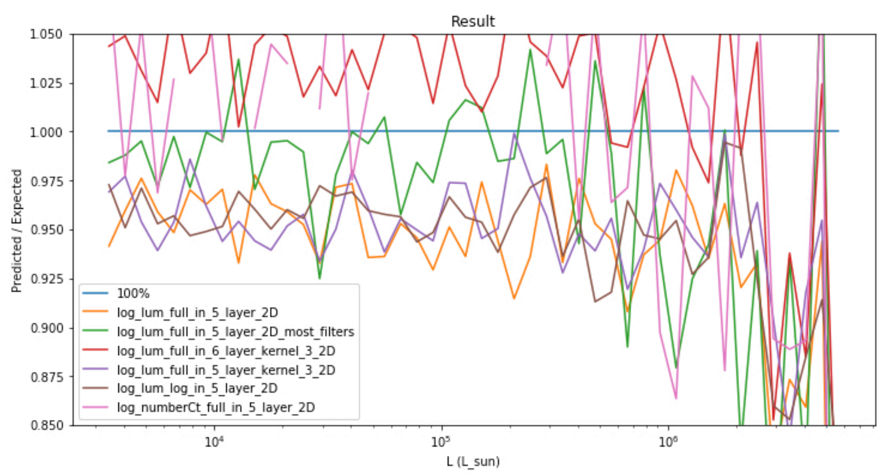

Figure 25: Same as Fig. 24, but showing ratio of CNN output to underlying value instead of raw values.

The loss from training and validation from a few models are shown in Figs. 26 to 29. The validation loss being less than the training loss is interesting. This could be due to dropout while training. While training 20% of the neurons don't fire as a way to help prevent over-fitting. It also means that the CNN isn't at it's best while training. Validation data is taken without dropout so it can be better. The weird discontinuities at the 100th epoch are due to the model being trained again. I only had enough time to train for 100 epochs before time out so I had to resume and in doing so it did something. One should be careful about comparing the log luminosity models to the number count models because they are looking at different things so different values are to be expected.

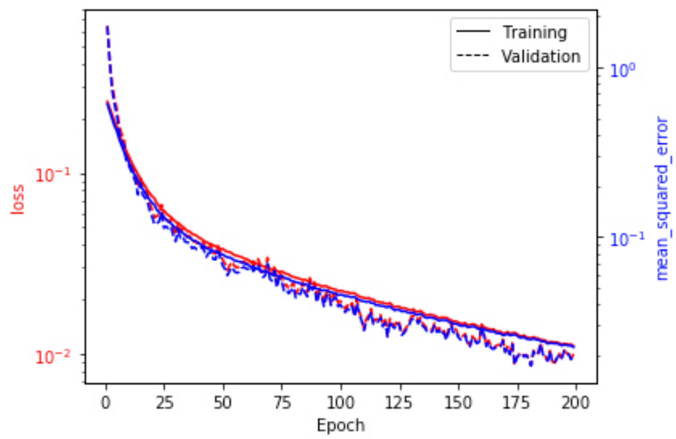

Figure 26: Loss history of the training of the log_lum_full_in_5_layer_2D model.

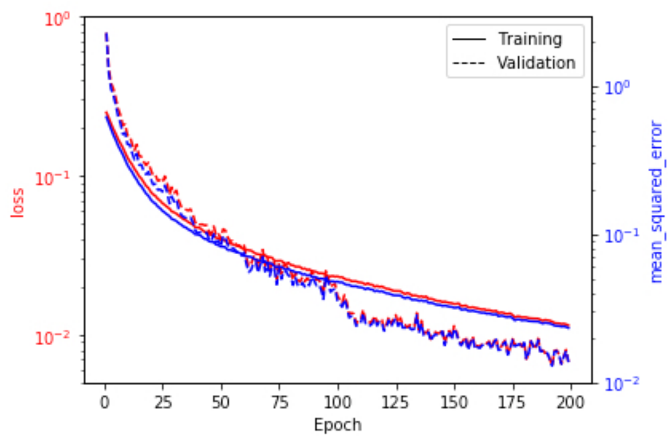

Figure 27: Loss history of the training of the log_numberCt_full_in_5_layer_2D model.

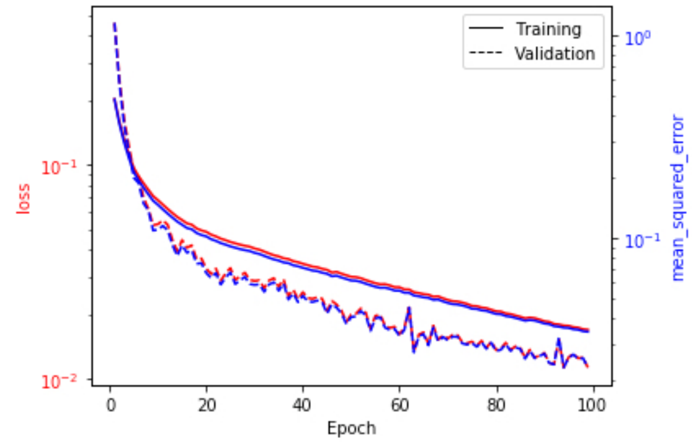

Figure 28: Loss history of the training of the log_lum_full_in_5_layer_2D_most_filters model.

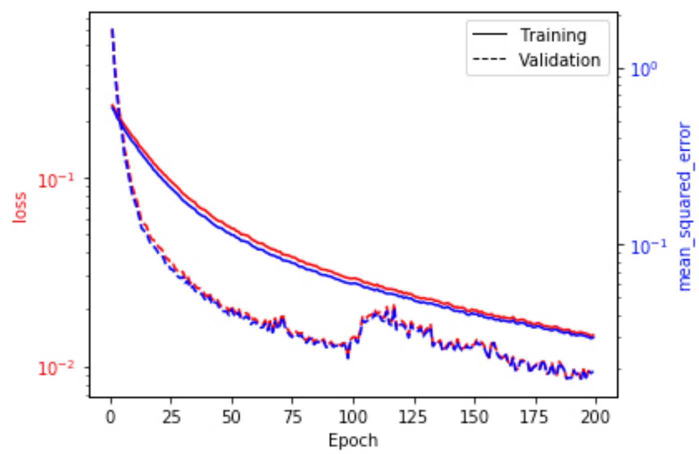

Figure 29: Loss history of the training of the log_lum_full_in_6_layer_kernel_3_2D model.

6.2 Adding Noise

Testing models on perfect information is all good and dandy, but we want to test our models against more realistic scenarios. In https://arxiv.org/abs/1808.07487 they try to get the luminosity function (ϕL) from intensity maps using an MCMC and power spectra. They also include an 11 μK white noise to their maps so it would be good to compare to their work. Figs. 30 to 33 are similar to Figs. 22 to 25 except the map (it is the same exact map) was loaded with 11 μK of white noise.

What we see is that most models get worse as in they predict less N or dN for all luminosities, but the 6 layer one gets much better because it originally over predicted values. Increasing the noise by a factor of 10 to 100 μK does not change model predictions that much. There is some variation because the noise varies whenever the map is loaded, but adding noise makes the 6 layer model better. All of the other models bunch together. This was not training on noise or anything, this is just taking the models we trained on perfect data and just putting noise into the maps when we want a prediction.

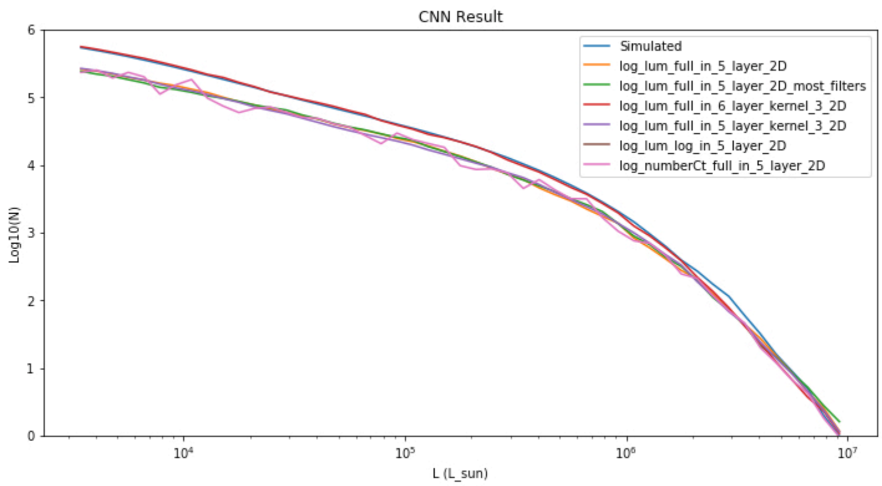

Figure 30: Output of different CNN architectures for N with 10 μK white noise.

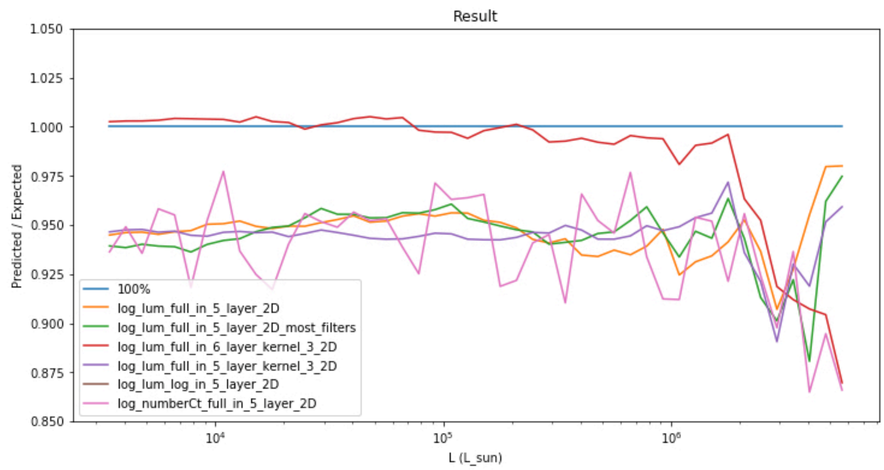

Figure 31: Same as Fig. 30, but showing ratio of CNN output to underlying value instead of raw values.

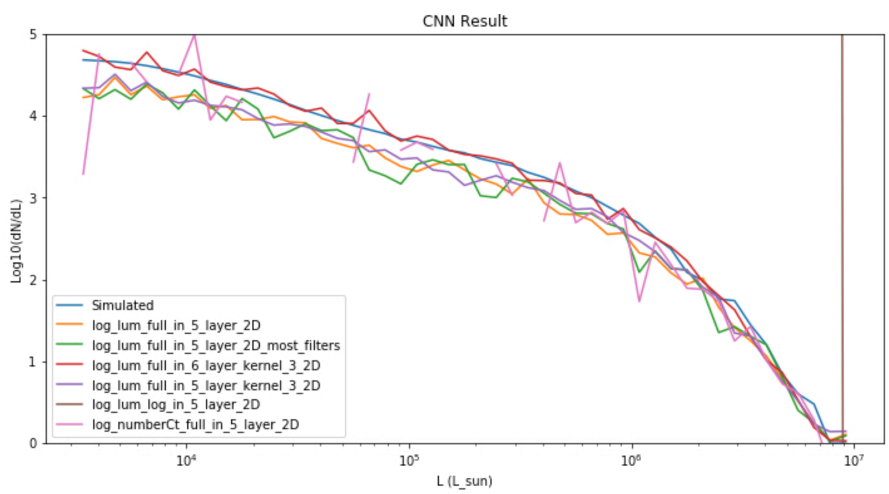

Figure 32: Output of different CNN architectures for dN with 11 μK white noise.

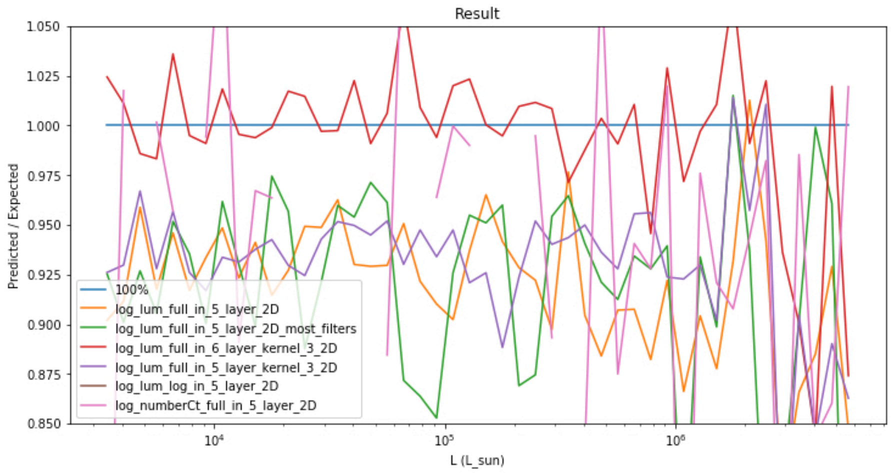

Figure 33: Same as Fig. 32, but showing ratio of CNN output to underlying value instead of raw values.

7 Compare to COMAP power spectrum luminosity method work

The work in Section 6 points to wanting to have more layers and more filters. There was no reason to go beyond 6 layers because of the fact that we would not be able to convolve anything in the 7th layer. More than 64 starting filters would lead to memory issues so we decided to only test 32 and 64 initial filters. Hyperparameters were still not messed with. Figs. 34 to 37 show the results of 200 epochs of training 6 layer models with a kernel size of 3 with starting size of 32 or 64 filters as well as training with noise or not. The plots show the usual things of comparing the output for N and dN v.s. the underlying values and then looking at the ratio of predicted to underlying. What we see is that the 64 filters doesn't do the best. Noise actually improves the 32 filter model, but doesn't do much for the 64 filter models. It seems like 32 filters is a good amount to have.

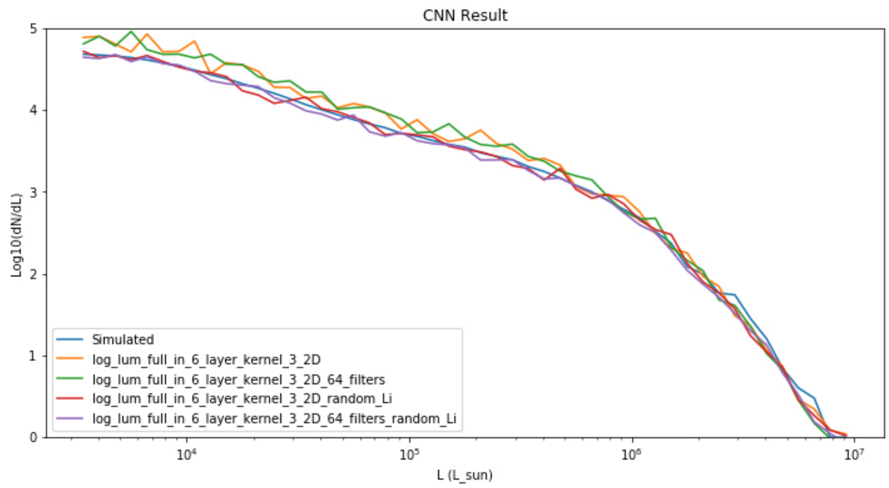

Figure 34: Output of different CNN architectures for dN.

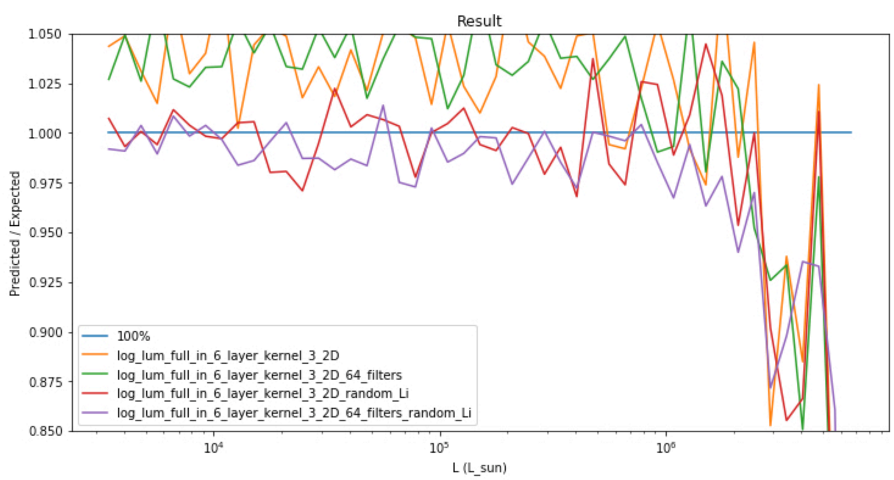

Figure 35: Same as Fig. 34, but showing ratio of CNN output to underlying value instead of raw values.

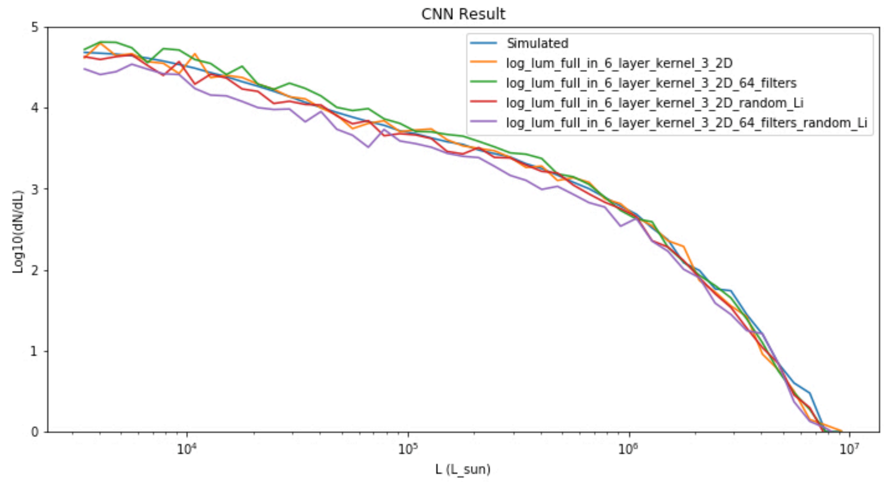

Figure 36: Output of different CNN architectures for dN with 11 μK white noise.

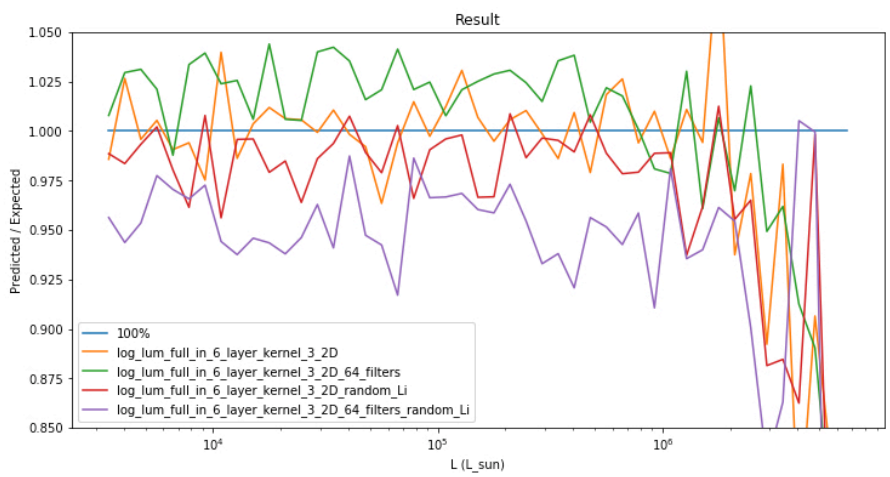

Figure 37: Same as Fig. 36, but showing ratio of CNN output to underlying value instead of raw values.

In Figs. 38 to 41 we see the training history for the models being looked at in this section. For the non-noisy models the plot is messed up. Epochs 100-175 for them are from a different run. Do to how I programmed things and the fact that I used the same names it did something stupid. Those epochs are an artifact and did not affect how the models trained this time. It only had 200 epochs of training and I didn't feel like trying to fix the plots and make them better.

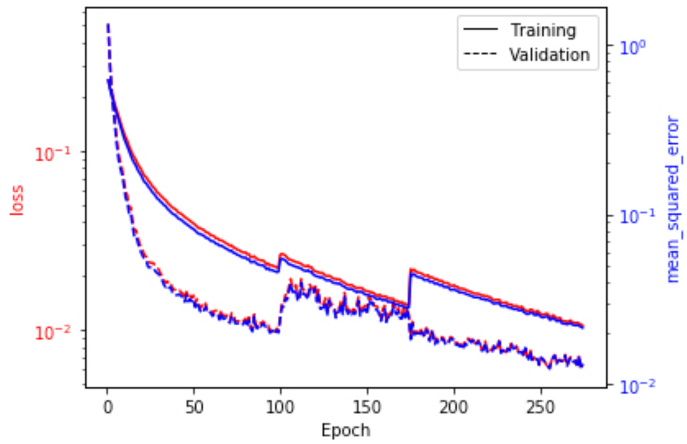

Figure 38: Loss history of the training of the log_lum_full_in_6_layer_kernel_3_2D model. See the above paragraph about epochs 100-175.

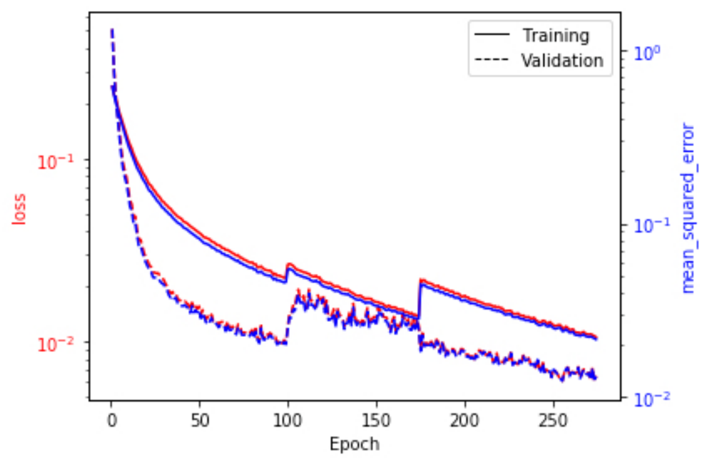

Figure 39: Loss history of the training of the log_lum_full_in_6_layer_kernel_3_2D_64_filters model. See the above paragraph about epochs 100-175.

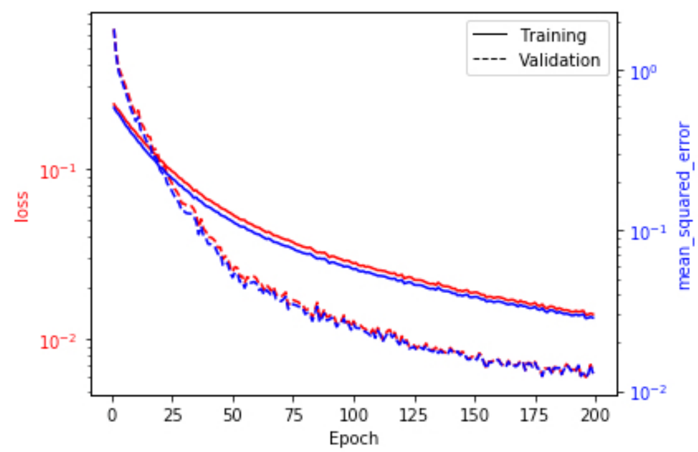

Figure 40: Loss history of the training of the log_lum_full_in_6_layer_kernel_3_2D_random_Li model.

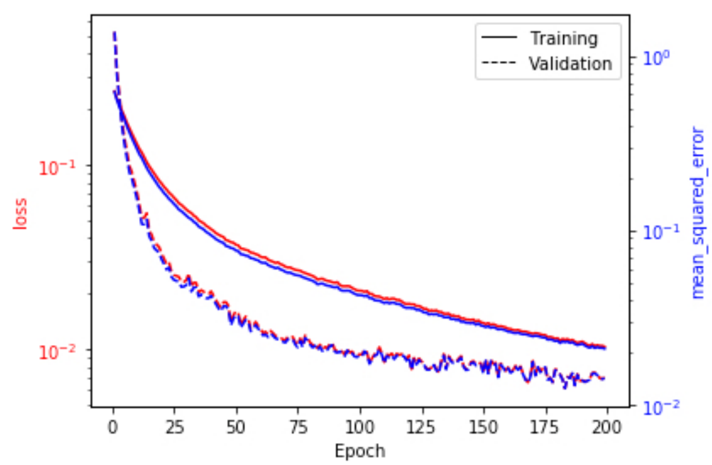

Figure 41: Loss history of the training of the log_lum_full_in_6_layer_kernel_3_2D_64_filters_random_Li model.

Looking at this it seems like 11 μK of noise doesn't do much to these models, but training on it does help a bit. Increasing the amount of noise by a factor of 3 does impact things though. I should remember to put in a figure for this.

At some point we should actually compare to the power spectrum and and temperature bin method of determining the underlying luminosity function. Figure 42 is the money plot from arxiv:1808.07487 which shows how accurate the power spectrum method is. One of the issues with this is that the luminosity function being used in the power spectrum paper is a density rather then the raw number count being used in this work so far. If we were working with real valued data instead of log data this wouldn't be an issue because a ratio gets rid of scaling, but that is not true in log space.

$$\frac{x \times a}{y \times a} = \frac{x}{y} \tag{4}$$

$$\frac{\log(x \times a)}{\log(y \times a)} = \frac{\log x + \log a}{\log y + \log a} \neq \frac{\log x}{\log y} \tag{5}$$

This means that converting from a density to number counts or vice versa will change the expected accuracy of the technique. Figure 43 shows the magnitude of a random map from the CNN training. It has values that are more then 6 orders of magnitude greater then the ones used in the power spectrum method. In Eq. (5) this would equate to having $a = 10^6$. The $\log a$ terms will dominate in some regions of the ratio compared to the $\log x$ and $\log y$ terms. The $\log x$ and $\log y$ terms represent the CNN prediction for the luminosity function and the underlying value and in density space will have values between -1 and -6.

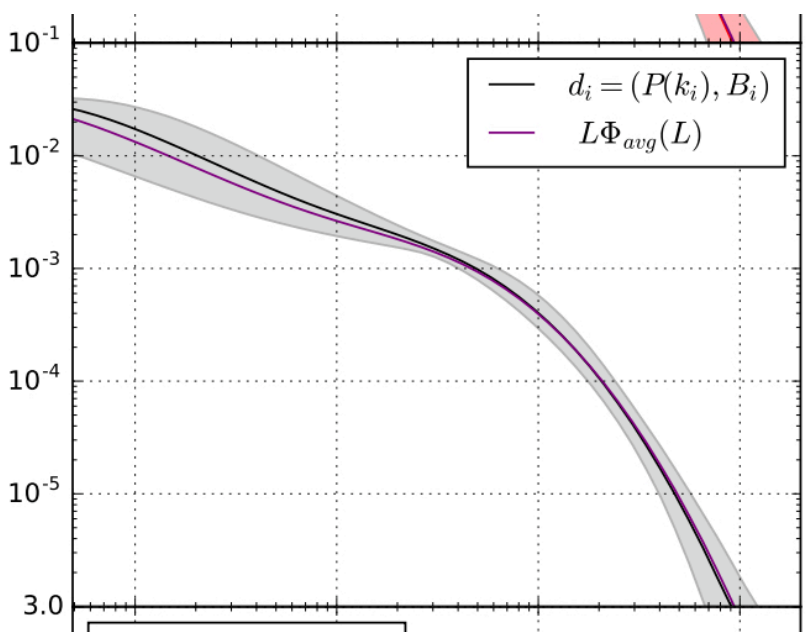

Figure 42: Figure from arxiv:1808.07487 showing the accuracy of the power spectrum luminosity function prediction technique. The y-axis is a luminosity function density instead of raw number count and the x-axis is just luminosity with the first major tick being $10^4 L_{sun}$. The grey shaded region is the 95% credibility interval of the power spectrum method while the purple line is the underlying luminosity function.

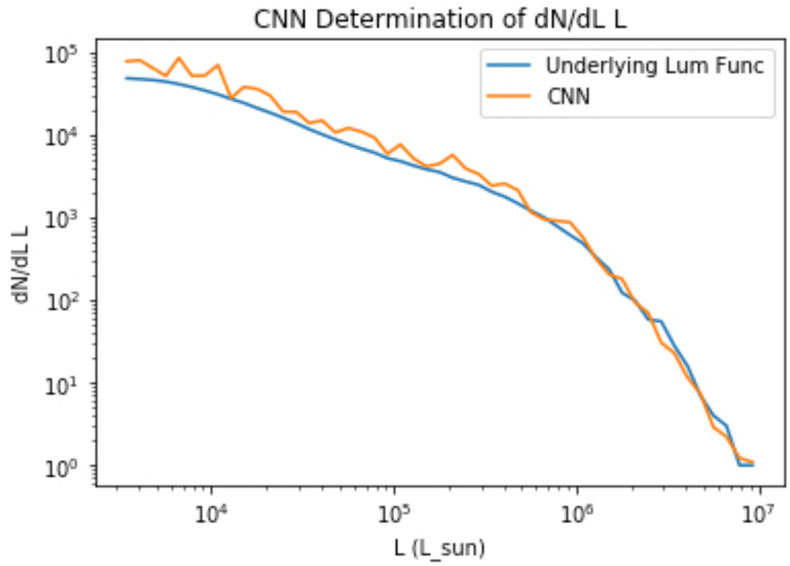

Figure 43: Figure showing the magnitude of luminosity functions used while training the CNNs. Notice that the y-axis if Fig. 42 is about 6 orders of magnitude smaller.

We must also figure out how to display our results in a statistical manor like that of the ones shown in Fig. 42. To do this we need the variance (with respect to the underlying luminosity function) of our CNN models. To calculate this I run a model on 40 different maps. For each luminosity I calculate the variance of the CNN luminosity functions with respect to the underlying function. From this I then calculate the standard deviation (std) and can get the 95% confidence interval using twice the std. When comparing the accuracy of the CNN by eye before I never thought about the 95% confidence interval used in the power spectrum paper so it really doubles the inaccuracy I was observing. Figs. 44 and 45 show the results for this with and without noise as well as the confidence interval for the power spectrum method. We can see that the CNN method is not as good as we believed. When comparing the correct things the CNN method is not the clear better choice. It is better at lower luminosities up until around $few \times 10^5$ and stays worse until around 10^6 where the two methods are roughly comparable again.

At a glance both methods seem comparable. Further in there are some differences. The CNN method is slow to train, but fast to get a luminosity function from a given map after training. I'm not sure about the speed of the power spectrum method. Running an MCMC is not usually the fastest thing around. The power spectrum method also takes into account a prediction of the underlying luminosity function. A different underlying function then the predicted one would probably cause issues. The CNN needs to train on simulated maps, so using maps made from multiple models will reduce accuracy, but I believe it shouldn't be as bad as with the power spectrum. This should be tested later. It hasn't been better, but the CNN might be better with foregrounds. This really needs to be tested.

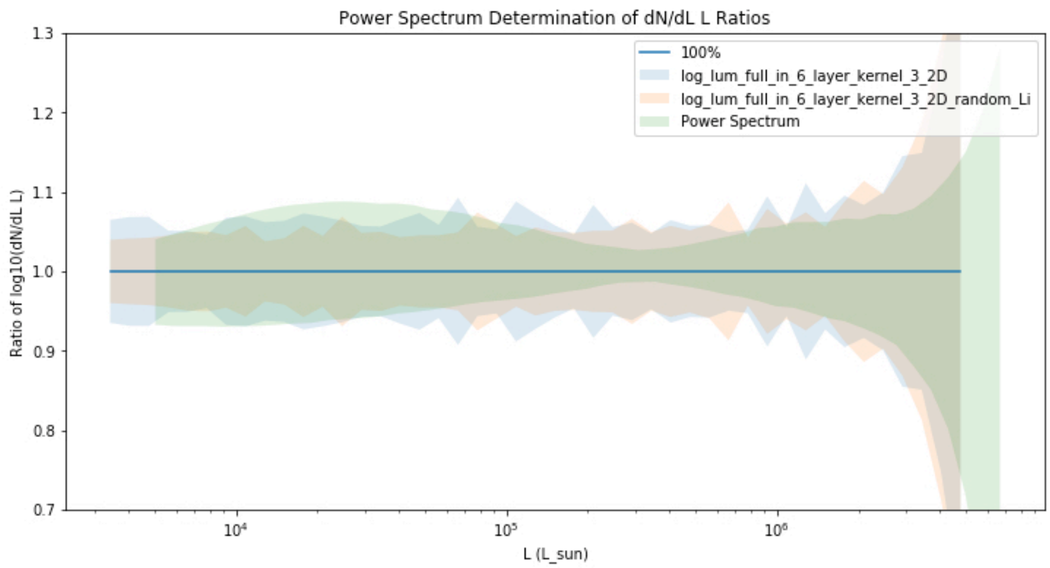

Figure 44: Comparison of different methods to determine the underlying luminosity function of an intensity map. The shaded regions are the 95% credibility interval or the 2 σ error bounds.

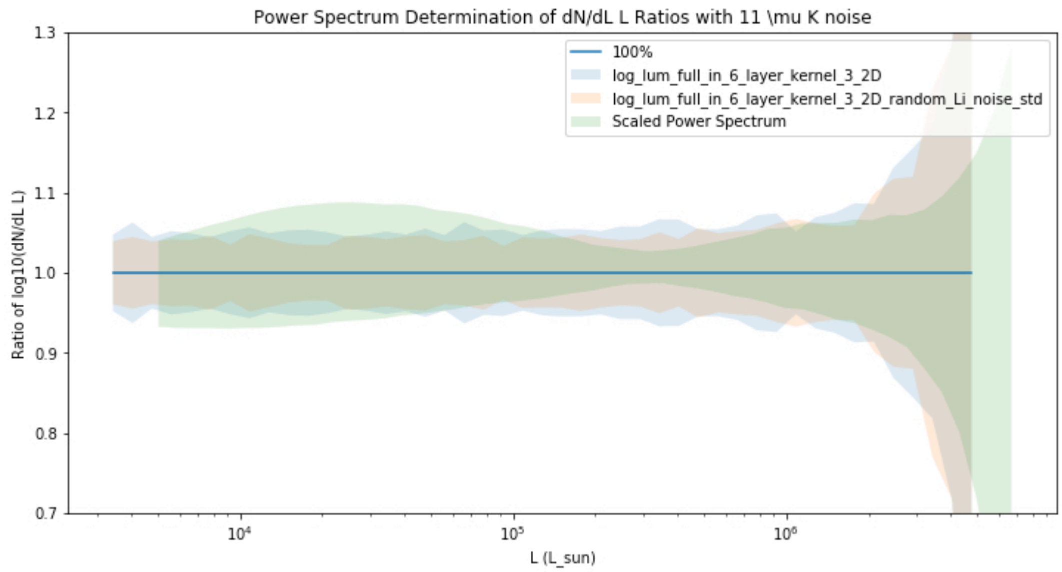

Figure 45: Same as Fig. 44, but with 11 μK white noise.

8 Training and Testing On Many Models

This stuff should be a second topic after testing with just white noise and seeing how good it is

Once we determine what architectures we are using we must determine what we want to train and test our networks on. As of right now we have 3 luminosity models with uncertainty on parameters. The only noise currently implemented is white noise.

We'll need to see what maps we want to train on. Some possible things to train on are as follows.

1. All base luminosity models (Just a single model at a time)
2. All luminosity models with random values used for parameters in each map (Just a single model at a time)
3. Combined models with base parameters
4. Combined models with random parameters
5. All previous scenarios with noise added

From this we need to determine which model works best in each scenario that matters. My naive guess would be models trained on noise will be better with noise and models without noise would be better for future experiments with less noise.

9 Things to do for Dan in no particular order

1. ~~Figure out what a good frequency bin size is~~ COMAP can handle around 500 different frequency bins so using 100 is a lower limit.

2. ~~Make maps and luminosity functions. Have done this for some maps and have the ability to do this for more.~~

3. ~~Add in different halo luminosity relations to llm~~

4. ~~Do some test runs with something basic~~

5. ~~Make an actual CNN and try training it for an extended period of time~~

6. ~~Get GPUs working correctly~~

7. ~~Train on actual luminosity function instead of log(luminosity function). Doesn't give the best results. See Section 4.2 (and this was done on N not ϕ).~~

8. ~~Get CNN to record loss and metric as it trains~~

9. ~~Make lots of maps on MARCC to train with. This will happen soon.~~

10. ~~Use validation data to see that the model is actually learning and not memorizing.~~

11. ~~Ability to train on log input data~~

12. ~~Ability to make noisy maps when producing the maps, or to add noise to maps after the fact~~

13. ~~See how many filters / layers the gpus can handle for 2D convolutions~~

14. ~~Figure out what architecture we actually want to use and start the actual training and testing. Log input or not, ϕ v.s. N, # of layers, # of filters.~~ 6 layers, 32 initial filters, full input, log output, dN/dL with log spacing of L.

15. Put info in about visualizing layers? Can do this, but do we care right now?

16. Foregrounds?