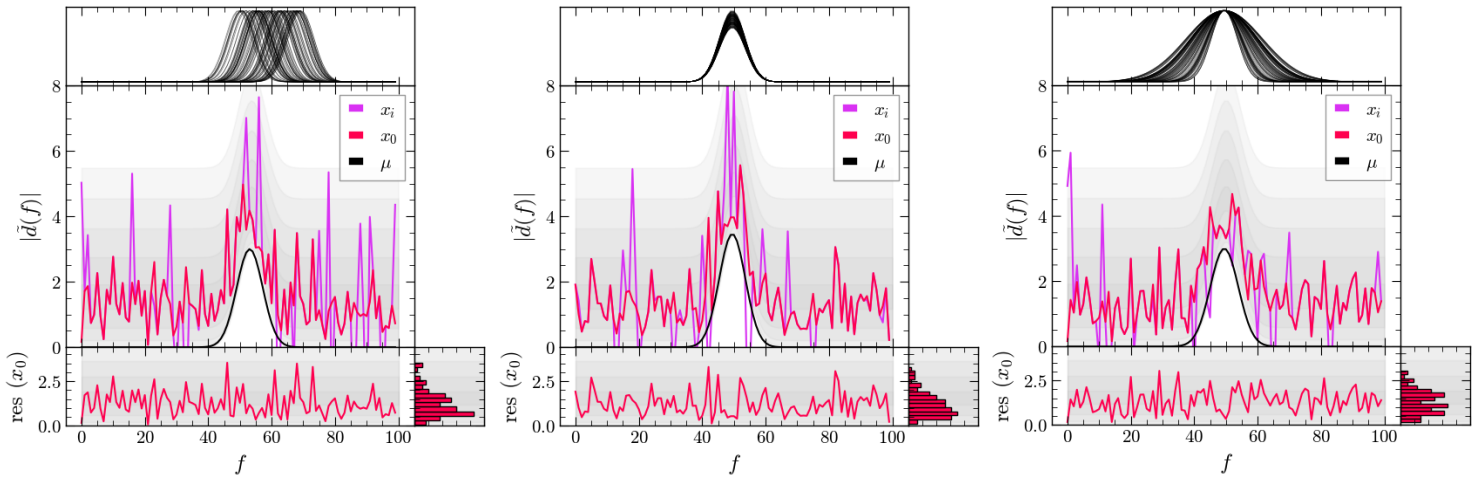
Network Architecture Update – For Noemi

code appendices: `https://github.com/james-alvey-42/gw-mist/tree/main/notebooks/week7/for_noemi`

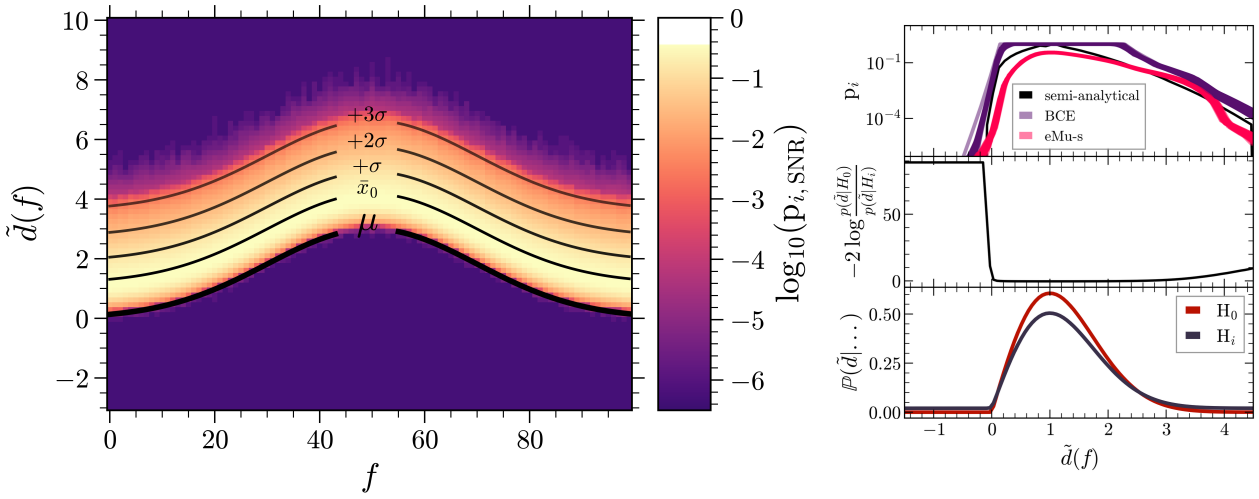
We have been working towards designing a network that best describes binwise distortions on top of a Gaussian mu with H0 noise generated by

```
Noise = torch.abs(torch.complex(torch.randn(x_shape), torch.randn(x_shape)))
```

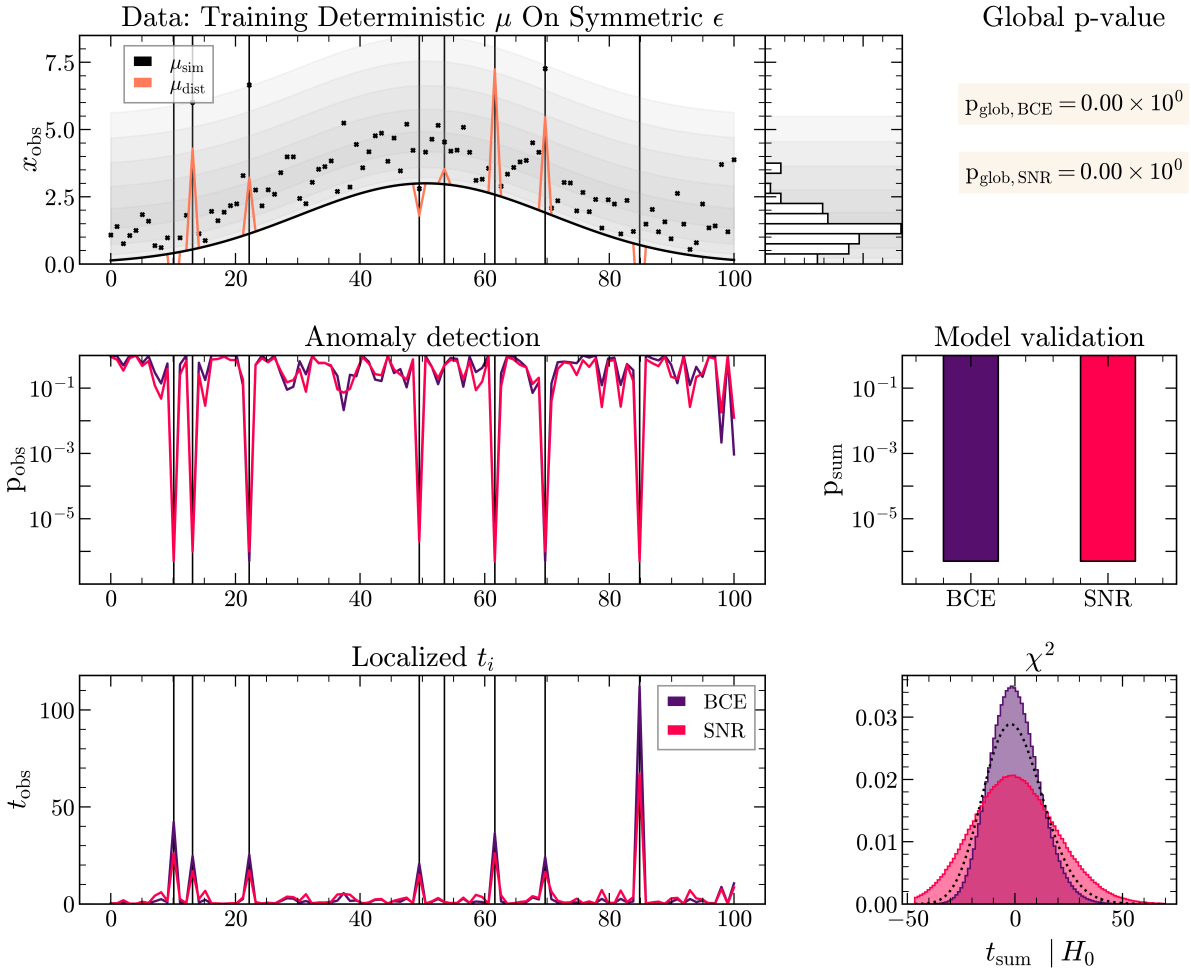
We have tested this with both a deterministic mu and with a stochastic mu within which we vary the amplitude, mean, and std by a small amount from the generator. The generator function can be found in this python documentation – it is a variation on the `Simulator_Additive` class from mist's sourcecode. For the purposes of the results in this document the simulator is kept in 'complex' mode (as opposed the the 'gw' mode or the 'white' mode). The below plots show the various forms of stochasticity the network is trained on when in stochastic mode.



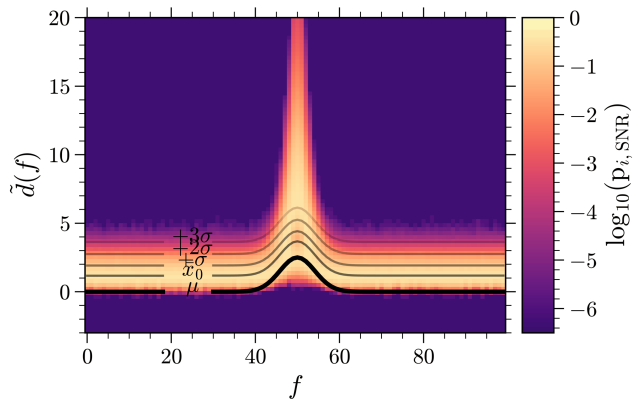
Training methods have seen various success. We train on a 'dynamic' pipeline: the network learns mu and epsilon simultaneously, which is much better at learning the data than a 'static' pipeline, where we train a network on mu and then one on epsilon, and treat the mu network like a static fitter/remover to the data. This works best when the data is not stochastic – the network learns the following p-values:

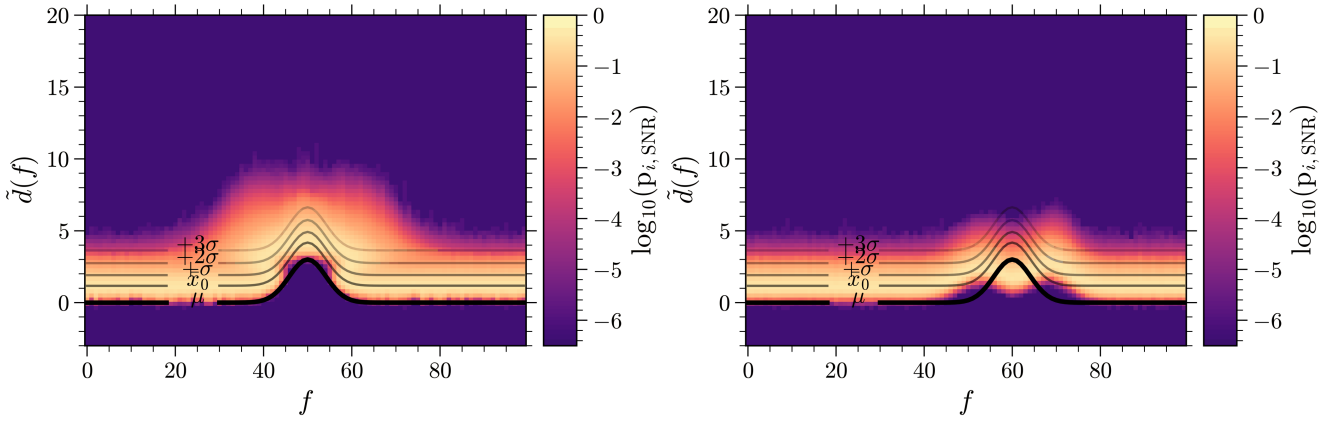


Which when used as a test on the generator produces the following distributions (the model converges within 20 epochs)

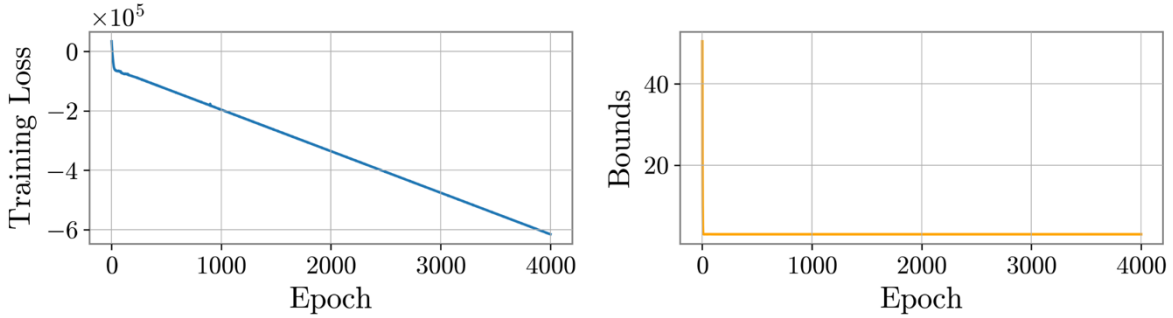


The difficulty then becomes, however, when training the stochastic mu. When training as above on the stochastic signal, the model learns the varying mean and std stochasticities well (these are best viewed as gifs in the same folder that the sourcecode for this document is in to show the variation in mu), but fails to learn variations in the amplitude, where the pvalue blows up (to a d(f)>50) even once converged. This network has a separate loss calculation for the mu-net and epsilon-net which is combined into a single loss that the network returns.

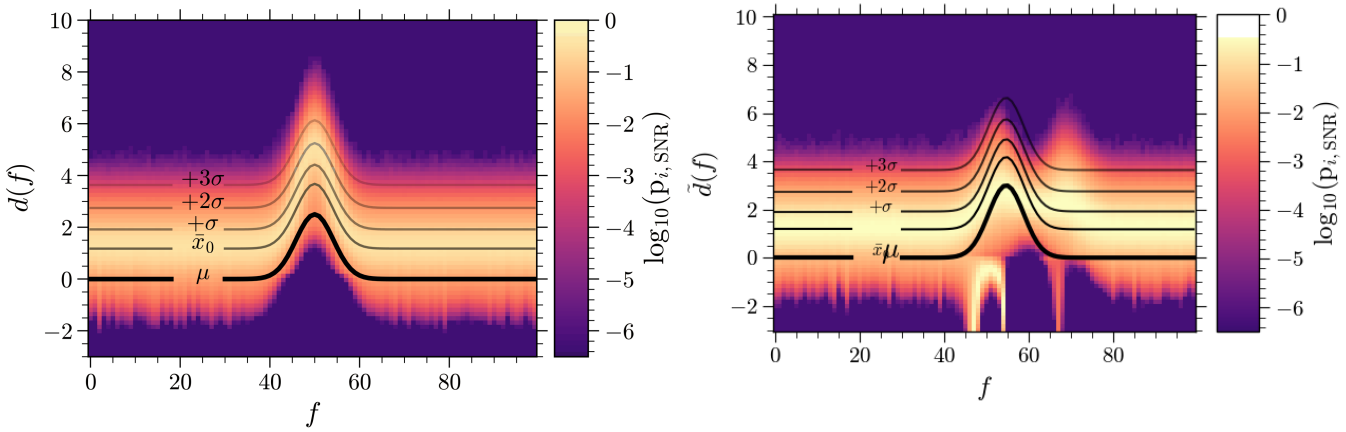




The alternative was decided to split the `network.logvariance` into a `network.logvariance_epsilon` and `network.logvariance_mu` for the separate network losses. This model notably failed to converge on a stable solution: the loss function for all three runs when left extensively followed

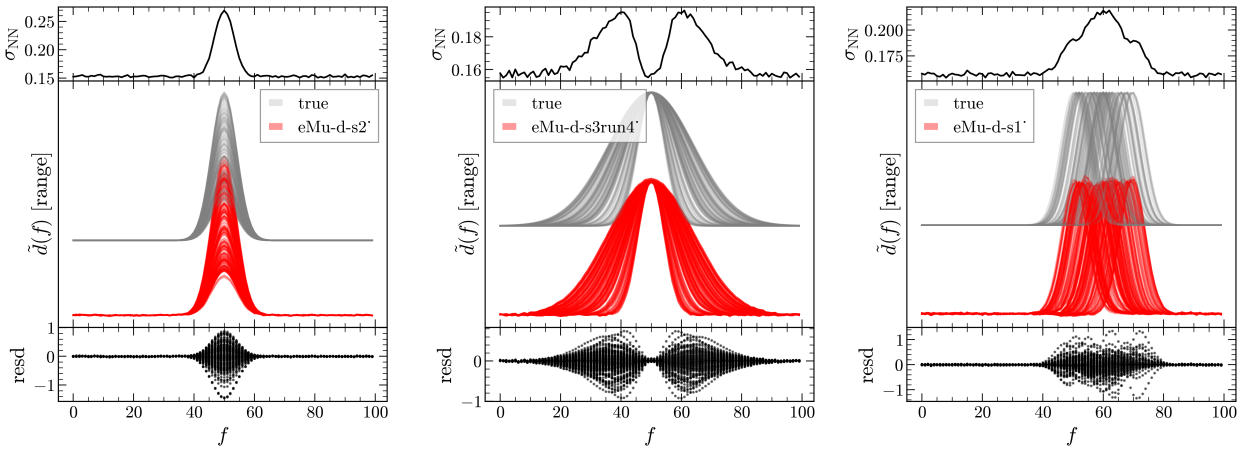


And when stopped in the initially stable zone (20 to 50 epochs) seemed to constrain the data well for the varying amplitude, but failed for varying mean and standard deviation:

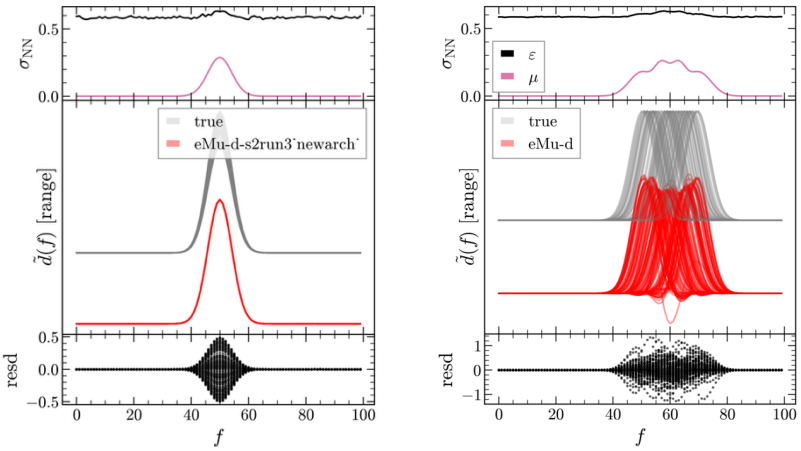


We are mainly wondering why the networks fail to constrain either the changes in the underlying signal amplitude or the changes in the gaussian mean/sigma and/or why splitting the network variances into individually learned parameters causes the network to fail to converge (even after ~4000 epochs). We also tried a purely mean-squared loss for the network,

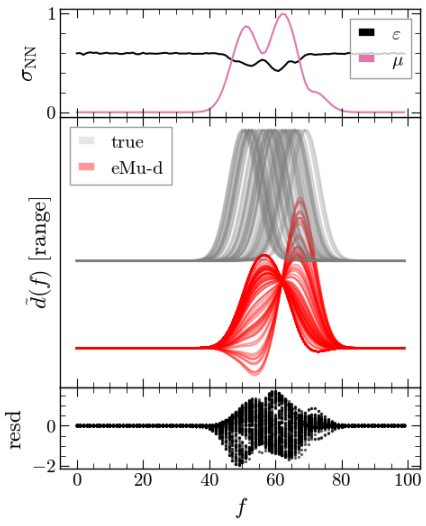
which converged, but produced similar crazy errors like in the figures above. You may also find my plots of the network's learned mu useful, firstly for the single-variance case:



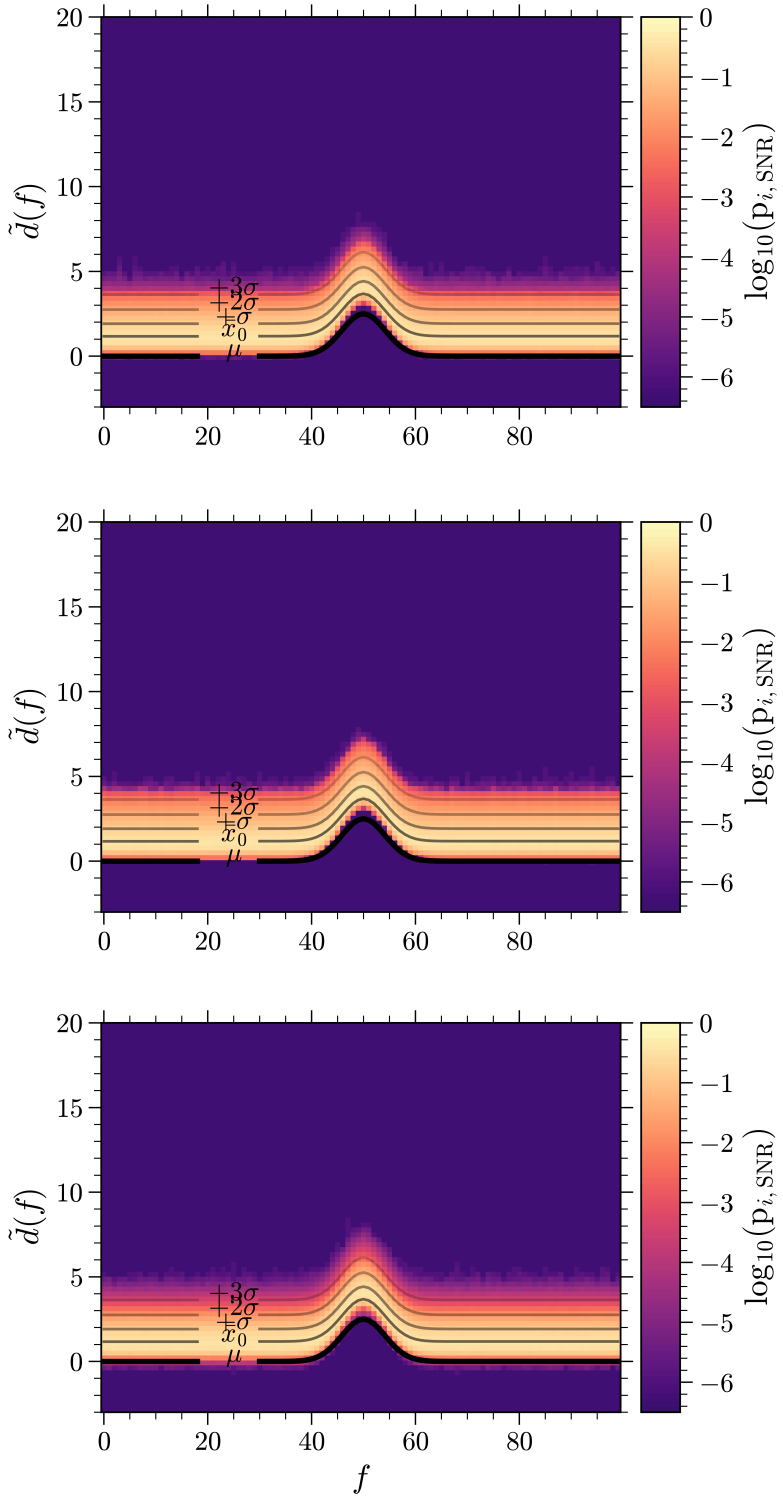
And then in examples of the separate variance cases:

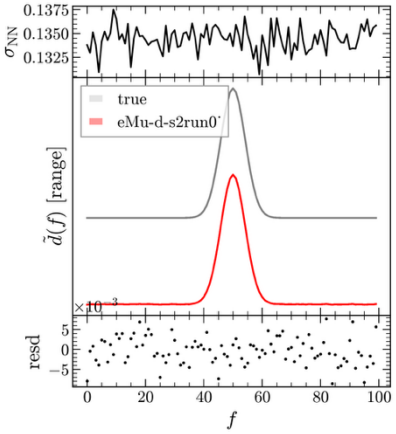


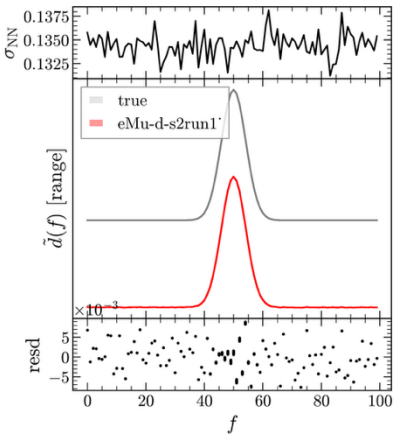
Note that we also tried with a network architecture that made a combination of the two variance parameters when calculating the loss on epsilon as to try and remove the degeneracy we see above; this network had a loss function that did not converge and looked like:

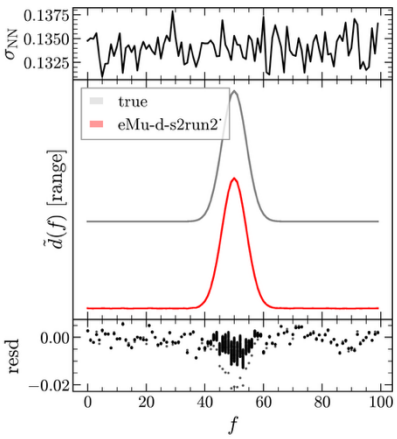


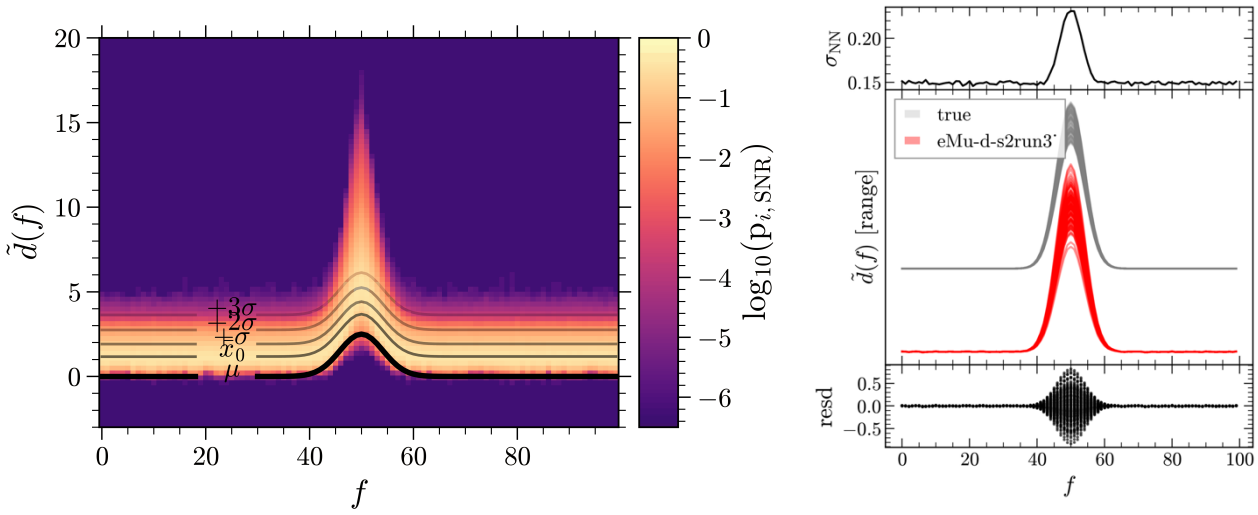
As a point of testing we also tried to see what scale distortion would cause the glitches observed in the figures above – the following plots range in stochasticity from an amplitude of 0.001 up to 1 (ignore the gif on the first image – it was a mistake!).











Ultimately this all may be pushing the stochasticity too far given that the GW model doesn't vary nearly as much as these mu models do, but equally it would be really good if you had any ideas as to why the different network architectures are causing the network to have very different p-value predicting capabilities. I also don't think the gaussian loss method is doing us any favours given that the underlying noise is not symmetric. As a NB 1) I have tried increasing the complexity of the MLP I'm using to calculate mu and this doesn't have much of an effect on how well the network is predicting mu, and 2) this dynamic method of predicting the mu does not work if I get the MLP to produce a 3-output of theta instead – the network almost always converges on a mu of 0 in these cases (although this method works well in the static case where a network can easily be trained to guess theta, although not to the accuracy at which the bin-wise predictor works).

Thanks so much!

Tom.