AM160 Final Project

Arjun Dhamrait

March 2025

The code for this project can be found on colab. When you made the announcement, I split it up into 2 parts and cleaned it up overall. The code for each part is listed under each part.

0.1 Data Loading

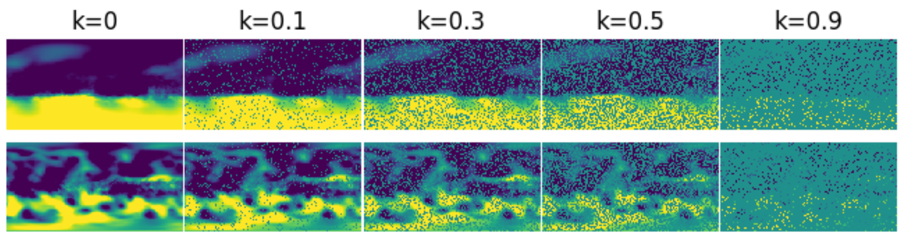
For both parts, the data is imported from mounting google drive and extracting the data from there. It's pretty self-explanatory if you'd like to import some new inputs to just upload the data files into your drive under a folder in the root of your drive named "AM160_Final_data", and follow the naming conventions for that year. For each part, data is standardized before doing anything with it, as seen in the "Data loading/standardization" section. I standardized instead of normalized because I like tanh more than sigmoid...

1 Denoising VAE

The code for this problem can be found here. This code includes both a derivative model and the actual best model I made: the direct model. If I'm going to be graded on performance, I'd like to be graded against the direct model implementation. For both of these implementations, I used an ELBO loss function , (MSE + a KL divergence term) and it worked pretty well!

1.1 Sparsification

The purpose of this part was to implement a VAE that takes sparse weather data and produces an estimate of the actual weather data. Here is an example of the sparsification function I used:



I tried to implement this two ways, first with a model that just tries to fill in points where the data is sparse (like a derivative model for a differential equation), and also a model that simply outputs the unsparsified data. The second model turned out to be much better, however I included both in this report :)

1.2 Model hyperparameters

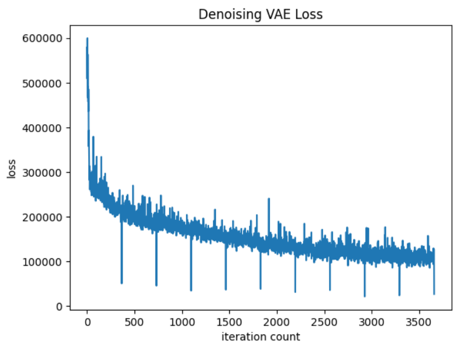
The VAE I used for all problems in this final was the same: 2 2-d convolutional layers to two linear layers to get the latent space mean and variance for the encoder, and a linear layer to get from latent space into 2 2-d convolutional layers to get the output! I programmed the latent space to be variable in length. For the denoising VAE, I included a conditioning in the decoder on the sparsity level for my final product, however I learned this didn't matter too much. For both models here, the latent dimension was 200.

1.3 Derivative model

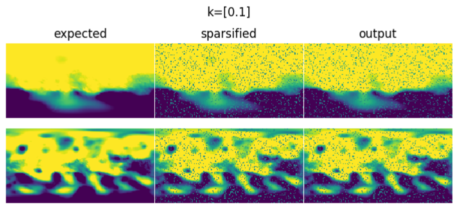
To train the model, I used 10 epochs with batch sizes of 16. For each batch, I chose a random k to sparsify the data. The loss function was MSE loss + KL divergence. The expected output of the model was filled-in values at the sparse points, so in order to get a desparsified output

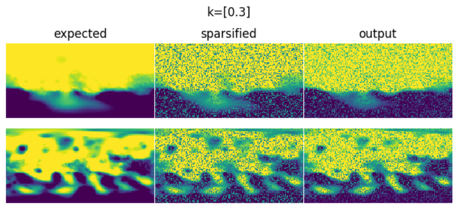
$$x_{\text{desparsified}} = M(x_{\text{sparsified}}) + x_{\text{sparsified}}$$

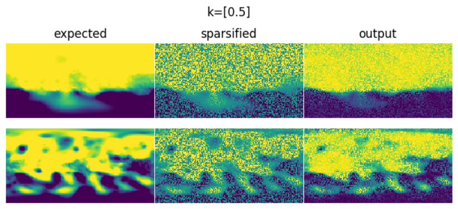
Here is a loss graph:

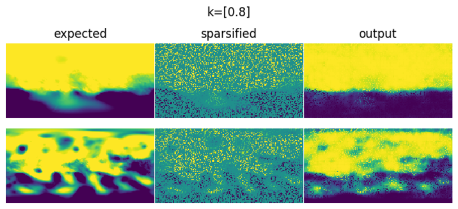


Here is some chosen output comparisons at different k values:

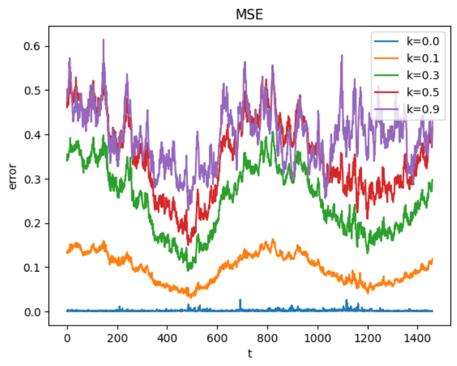




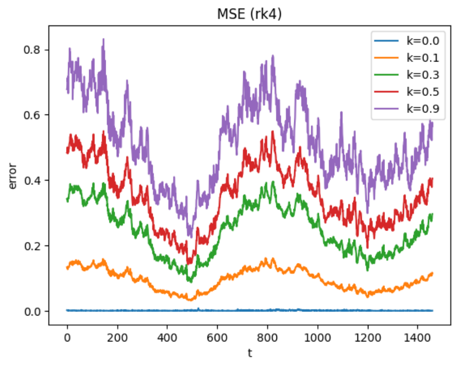




And an error graph over t for different k-values.



Just for fun, I wanted to see if I could use an rk4 step to get better results instead of just adding the model's output (which would be an euler step). It was not better :(

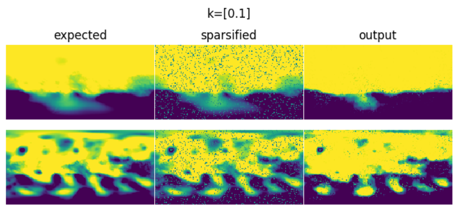


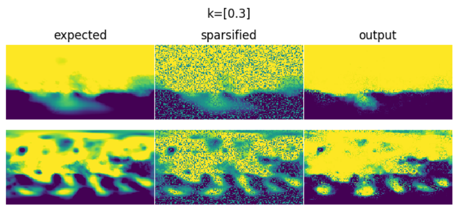
1.4 Direct model

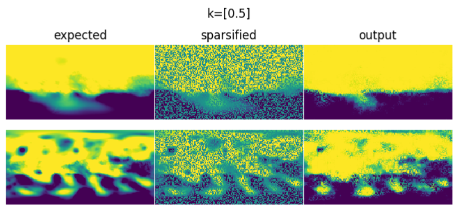
Instead of doing the derivative model, the simpler direct model is better for this case! Here, we cirectly calculate the desparsified data. Additionally, we conditioned the model on the k-value.

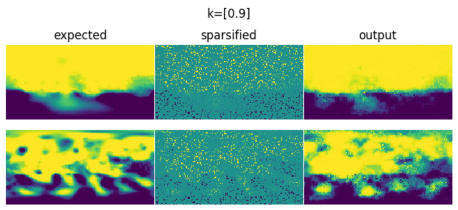
$$x_{\text{desparsified}} = M(x_{\text{sparsified}}, k)$$

The output is much better:

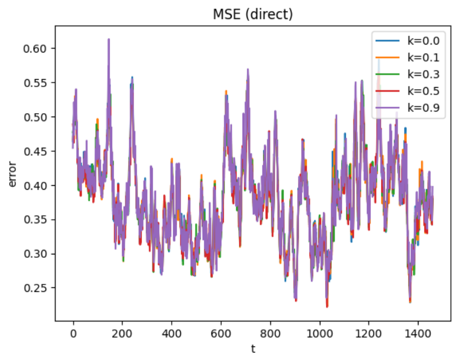








Interestingly, the error doesn't change much with differing k values any more! This means the conditioning really works!



2 Autoregressive Generative VAE

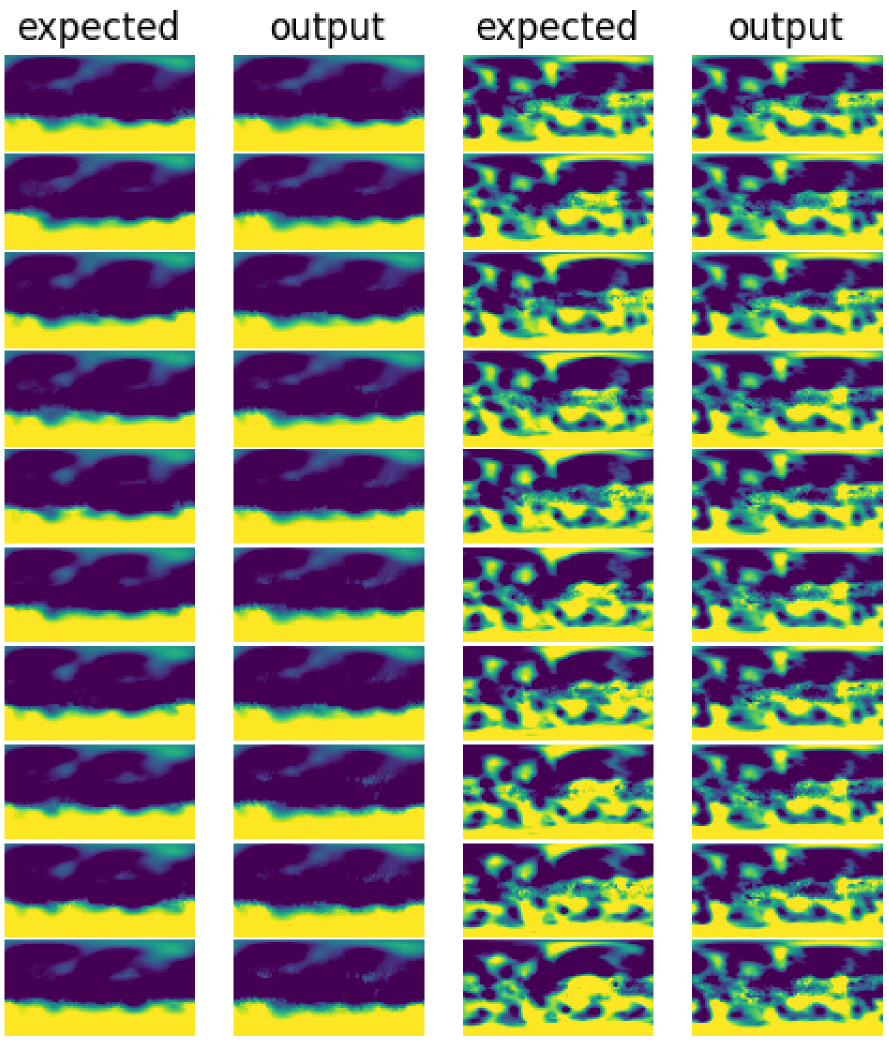
For this problem, I accidentally implemented an incorrect model that ended up working better than the actual autoregressive VAE. For the incorrect model, I was not concatenating x_{t-1} to the latent space of the decoder, instead I was inputting it into the VAE overall. I fixed this, and made a model that can take a variable number of previous timesteps, and trained that model in this colab. This ended up doing slightly worse, as I talked about in the ”comparison” section, but it is the model that was asked for on the final so It's the model I will use :) Here is the colab.

2.1 Incorrect implementation

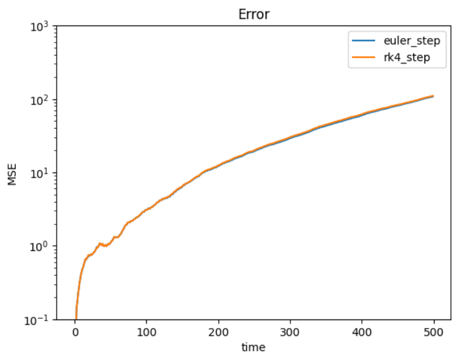
If you'd like to see the incorrect model in action it's in the main colab, under part 2 in the section titled "Random Conditioning + Ensemble". It turns out that my VAE model works pretty well for generation without doing anything to the latent space, with very fast (¡2min) training times! Additionally, prediction times are pretty fast too. To create the generative model, I used the literally the exact same model as the desparsification VAE, just increased the latent dimension to 1000 instead of 200. The decoder was conditioned on a random variable r which was only 1-dimensional, so that I could get generational output after. I used a derivative model, where the model predicts the change in state between timesteps, as trying to just output the next step wasn't working very well at all (you can see this other implementation in the colab too, it wasn't very good).

$$\frac{dx}{dt}|_{x_{t+1}} = M(x_t)$$

Because the decoder is conditioned on a random r, we can take an ensemble of samples at every timestep and find the mean of that ensemble to get the mean output! This made the iteration much more stable. For the step function, I implemented both rk4 and euler steps, however both were just about equivalent so if I were to actually use this model I'd use the euler step (much less overhead). Iterating for 500 steps was still super fast! Only taking $\tilde{3}$0s despite 10 samples per rk4 step and 4 rk4 steps per full step (40 calls to the model for every step)! Here is the euler-step output plotted for the first 10 timesteps:

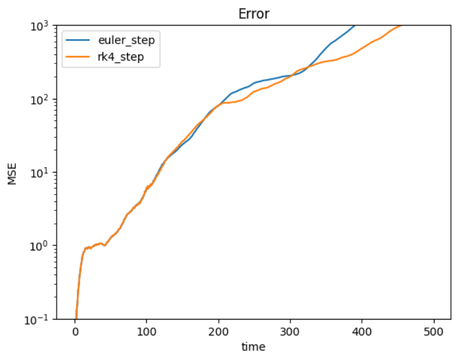


The MSE graph was pretty nice too! < 1 MSE up to 50 iterations. That's almost 2 weeks of output! Pretty sweet....

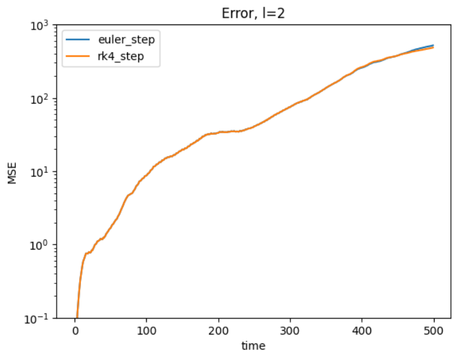


2.2 Correct implementation

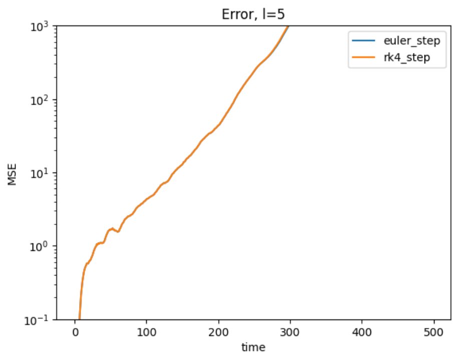
Again, here is the colab. The correct implementation was to train just the decoder, given noise concatenated with the previous timestep for the input to the decoder. Like above, I chose to use a derivative model, predicting the change in the next timestep. When only training the model using the last timestep, we get slightly dissappointing results:



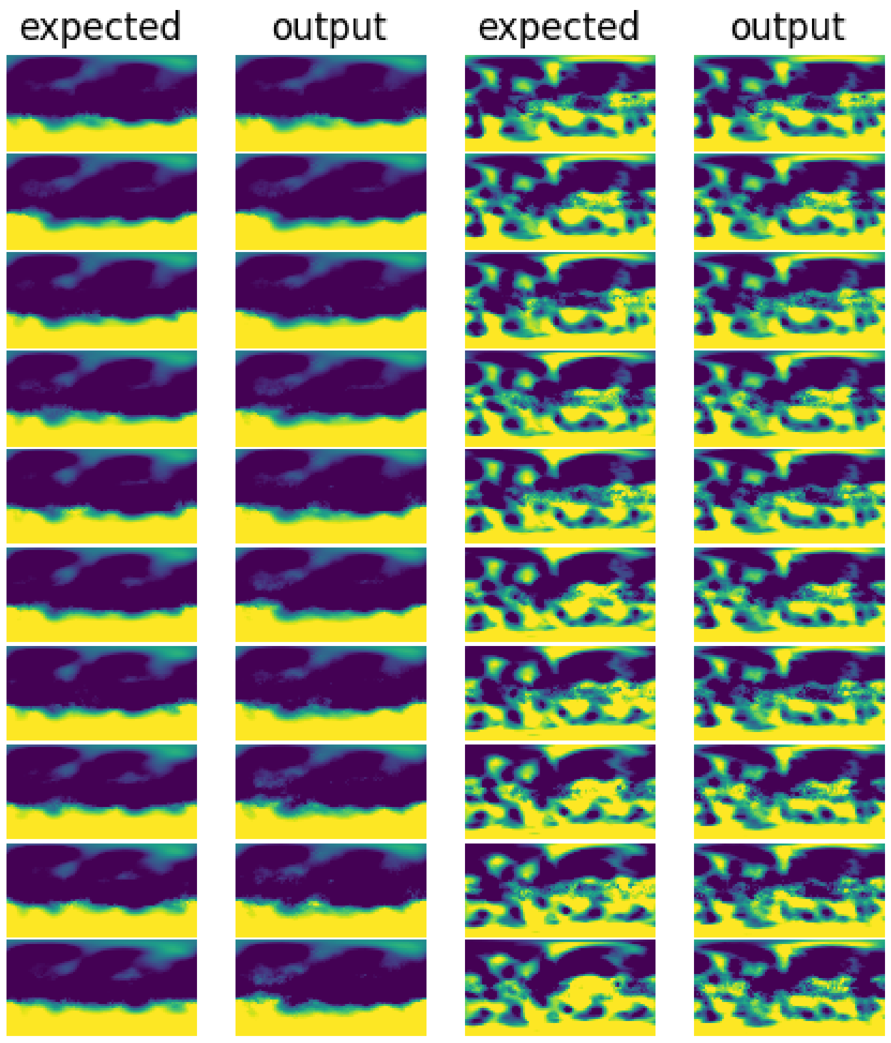
however, the beauty of this model is you can add more previous timesteps easily! Here's the error plot with the previous 2 timesteps as input:



And here's for 5:



As you can see, none of these error plots show better performance than the incorrect implementation above... I bet I could get the performance of the $l = 5$ case to be better if I had more resources to train it, however it was already taking around 5 minutes to train the model so I skipped that. Because for $l = 2$ the euler step is just about equivalent to the rk4 step, here's a comparison of the output with euler step:



2.3 Comparison

As you can see from the graphs, the ”incorrect” implementation where you input x_{t-1} into a full VAE and concatenate noise to the latent dimension to create an ensemble was actually better than concatenating x_{t-1} to random noise in the latent space and decoding that! Not only does it have better error rates, it is faster to train and use! If I were to guess why, I don’t have enough parameters in my correct implementation model to train it well. Unfortunately, when I tried to increase the number of parameters I was running out of memory... Additionally, looking at both implementations it is clear that overall rk4 and euler step is nearly identical, with rk4 being slightly better usually. I’d say, for these models it doesn't make sense to use an rk4 step because it's 4x the work for really negligible impacts.

3 Forward Diffusion Process

$$\begin{aligned}
x(t) &= \sqrt{1-\beta_t}x(t-1) + \sqrt{\beta_t}N(0,I) \\
&= \sqrt{1-\beta_t}(\sqrt{1-\beta_{t-1}}x(t-2) + \sqrt{\beta_{t-1}}N(0,I)) + \sqrt{\beta_t}N(0,I) \\
&= \sqrt{1-\beta_t}\sqrt{1-\beta_{t-1}}x(t-2) + (\sqrt{1-\beta_t}\sqrt{\beta_{t-1}} + \sqrt{\beta_t})N(0,I) \\
&\dots \\
&= x(0)\prod_{k=1}^{t}\sqrt{1-\beta_k} + N(0,I)\sum_{k=1}^{t}\sqrt{\beta_k}\prod_{j=k+1}^{t}\sqrt{1-\beta_j}
\end{aligned}$$

We know that

$$\lim_{t\to\infty}\prod_{k=1}^{t}\sqrt{1-\beta_k} = 0$$

Additionally, we can see that

$$\sum_{k=1}^{t}\sqrt{\beta_k}\prod_{j=k+1}^{t}\sqrt{1-\beta_j}$$

always exists even as $t \to \infty$, so $x(t)$ is the sum of gaussians when t is very large. The sum of gaussians is also gaussian, so $x(t)$ approaches a gaussian.