SAiDL Spring Assignment 2024

I have attempted the Pruning and Sparsity, Reinforcement Learning and Computer Vision from the Induction Assignment. The following is a detailed report of my attempt.

Pruning and Sparsity

I started with the pruning task, as the idea (from quick Google searches) seemed exciting and proceeded to dive into the course material. As someone interested in the intersection of systems and ML, I had previously gone through some modules from the TinyML and Efficient Deep Learning Computing course, namely Knowledge Distillation and Distributed Training.

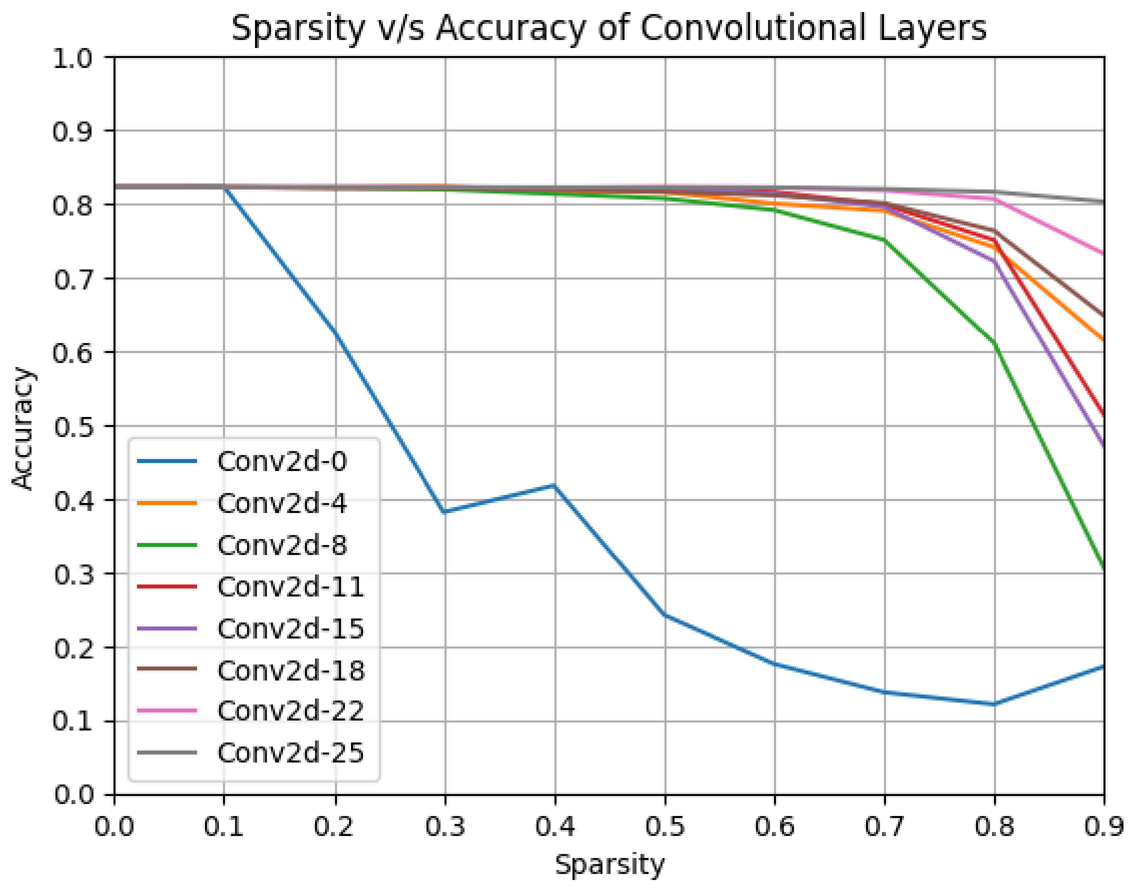
Base Model Implementation

Once I understood the technique, I picked the VGG11 (passed the size threshold of 35 MB) and trained it on CIFAR-10 to get the base model.

Sensitivity Scan

After identifying the convolutional layers of the model, I performed a sensitivity scan on them individually by training the model from scratch and pruning the layer with sparsity ranging from 10% to 90% in increments of 10%. The pruning performed was a fine-grained L1 (magnitude-based) method.

I plotted the sparsity vs accuracy of each layer in the below plot to identify optimal sparsity values for each layer.



Fine-grained Pruning

I picked the following sparsity values for each layer by observing the drop-off in accuracy from the graph.

Conv2d	0	4	8	11	15	18	22	25
Sparsity	0.1	0.7	0.5	0.7	0.7	0.7	0.8	0.9

Once again, I trained the VGG11 model to an accuracy of 87% and pruned each convolutional layer with the chosen sparsity value, which caused the accuracy to drop to 41%.
Then, I proceeded to fine-tune the pruned model again up to an accuracy of 88%.

The compression ratio of the model through pruning is 23%.

I have also plotted the distribution of weights in the conv layers before and after pruning.

Reinforcement Learning

I started the RL task with basic RL knowledge (MCC and TD) through some David Silver lectures and a half implementation of easy21 in Jax. This task was my first foray into Deep Reinforcement Learning, and I learned a lot through this task.

Deep Reinforcement Learning

I started by revising some concepts in DRL and reading the paper to understand the core idea. ICM aims to give the agent an intrinsic curiosity reward signal to promote exploration.

I set off to implement the DQN algorithm for the Hopper-v4 environment first. I went through the Gymnasium tutorials (which were really helpful) and the documentation to take note of the tools I'll need later (like the FrameStackObservationV0 env wrapper).

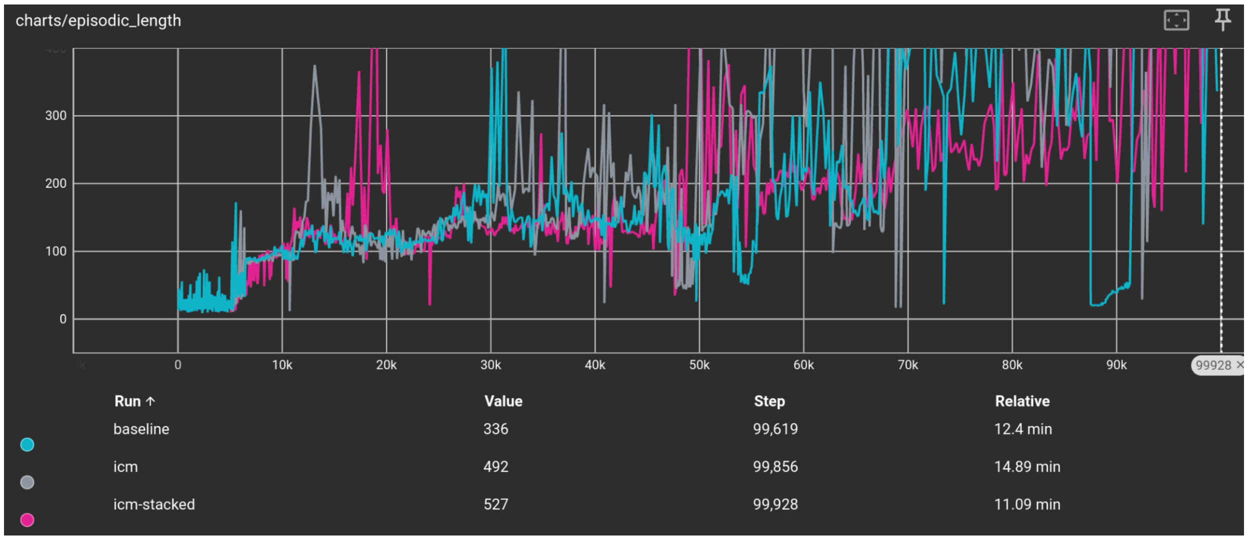
However, I soon ran into trouble since the Hopper-v4 had a continuous action space, which is not ideal for integrating with a DQN (it was a good exercise in understanding DQN's in-depth). Then, I decided to go with a SAC model after getting advice from a PoC and implemented a basic version. I borrowed some of the infrastructure for plotting (using tensorboard) and logging from CleanRL's SAC implementation (since I ran out of time before midsems).

Intrinsic Curiosity Module

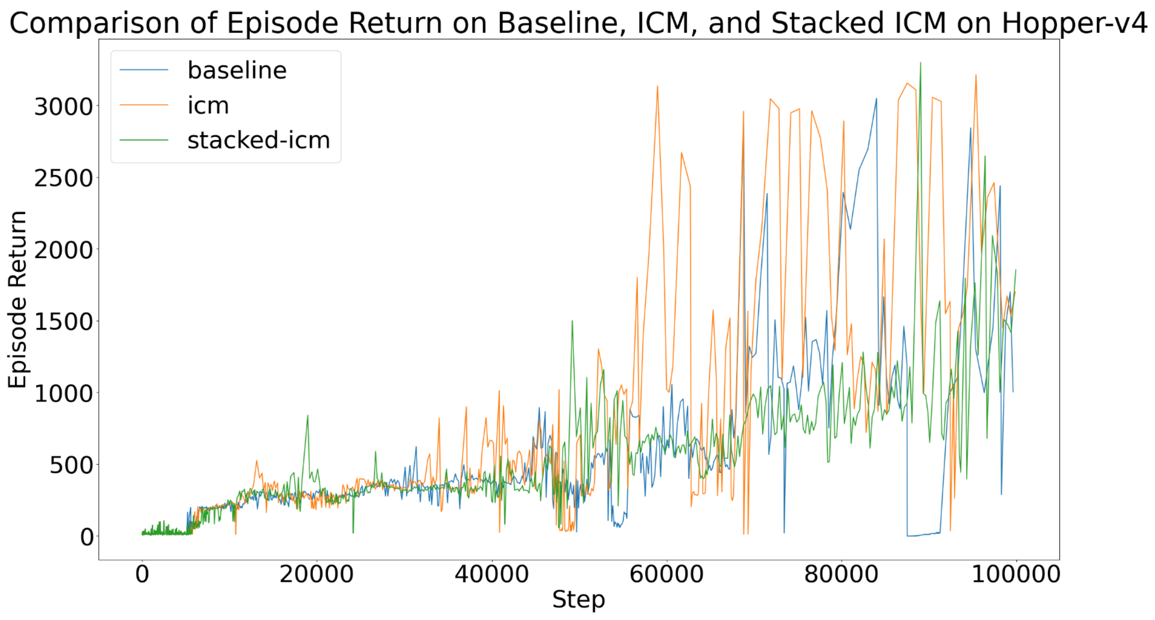
The ICM model implementation was pretty straightforward. The forward model predicts the next state given the previous state (or stack of previous states) and action, whereas the inverse model predicts the action given the previous and next states.

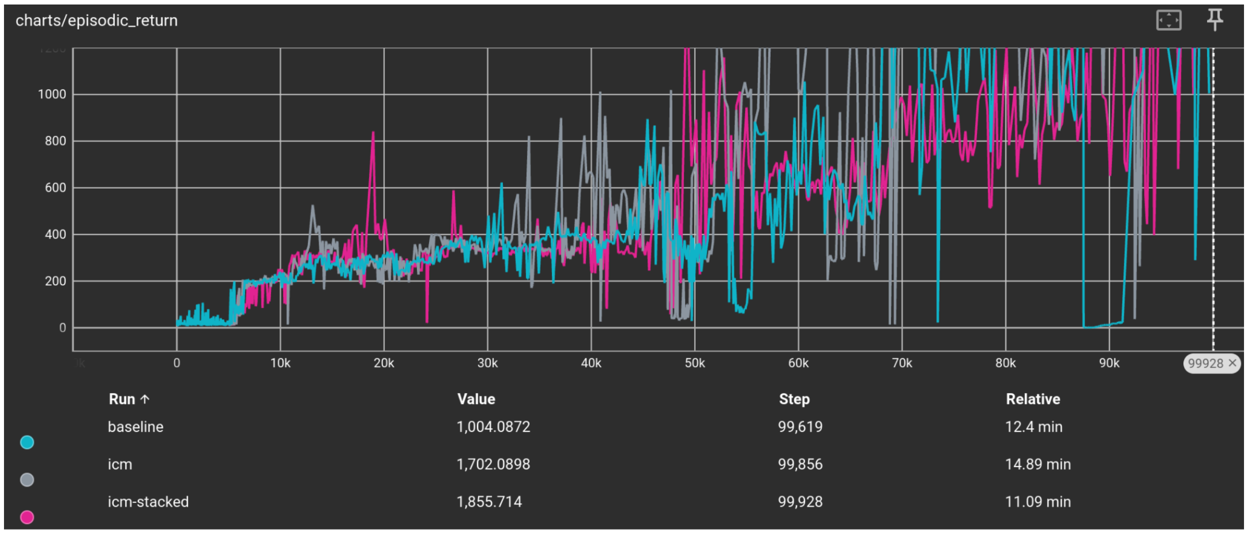
Since the Hopper-v4 observation is a size 11 vector, I decided to go with a simple MLP for both the forward and backward models of the ICM Module. I calculated the ICM Loss and Intrinsic Reward as described in the paper and trained the agent.

I also implemented an ICM with a modified forward model, which uses a stack of previous frames' encodings and the current action instead of just the current frame's encoding to compute the loss. It can be seen as performing marginally better than the single-frame ICM.

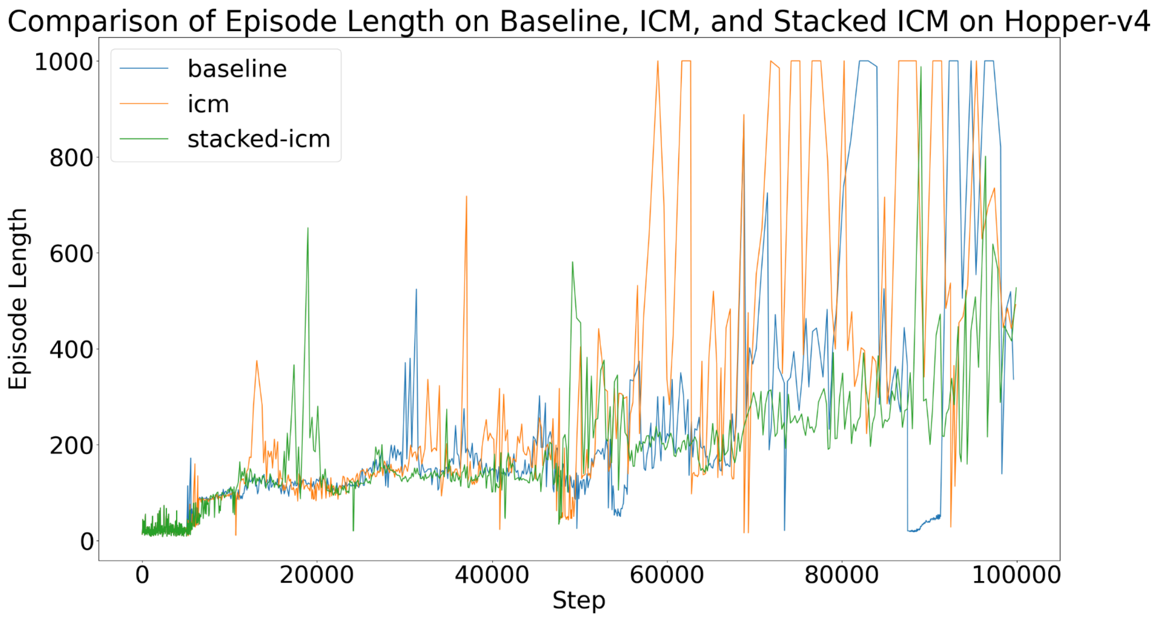


	Run ↑	Value	Step	Relative
●	baseline	336	99,619	12.4 min
●	icm	492	99,856	14.89 min
●	icm-stacked	527	99,928	11.09 min





	Run ↑	Value	Step	Relative
●	baseline	1,004.0872	99,619	12.4 min
●	icm	1,702.0898	99,856	14.89 min
●	icm-stacked	1,855.714	99,928	11.09 min



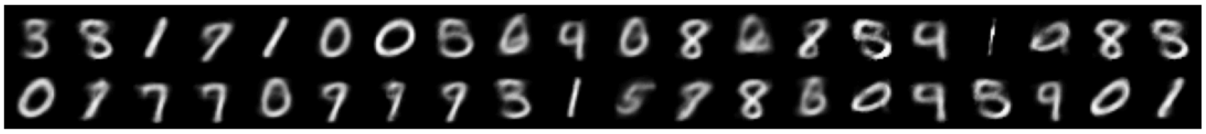
Computer Vision

I picked the computer vision task since I have previously worked with CNNs and GANs but have yet to delve into VAEs and thought it would be a good opportunity for the same.

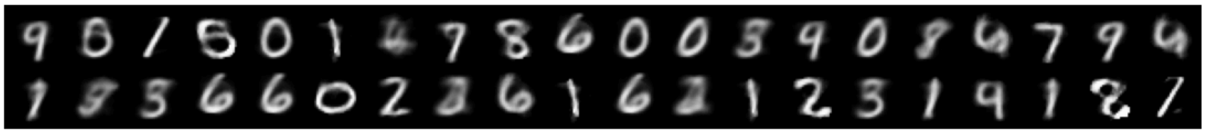
I reviewed the attached articles and videos to understand how VAEs extend the AutoEncoder architecture. Once I had the necessary knowledge, I implemented a simple Full Connected VAE with two latent dimensions and trained it on the MNIST dataset.

I trained two variations of the models, one with Normal(1,0) and the other with Gaussian(1,2), to compare them.

FC VAE Normal(1,0) Latent Dim = 2



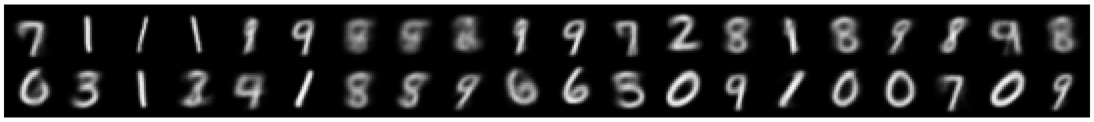
FC VAE Gaussian(1,2) Latent Dim = 2



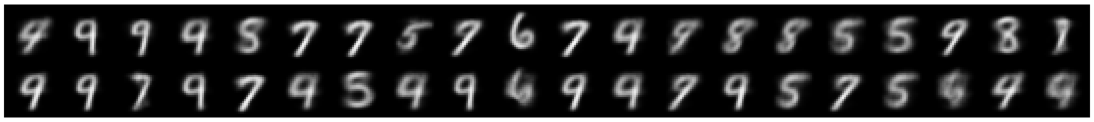
The generations from both seem qualitatively similar, except that the distorted ones from the Gaussian set seemed more distorted than the ones from the Normal set.

I quickly realised that using convolutional blocks might improve the generation quality of the VAEs, so I implemented a convolutional VAE.

CNN VAE Normal(1,0) Latent Dim = 2



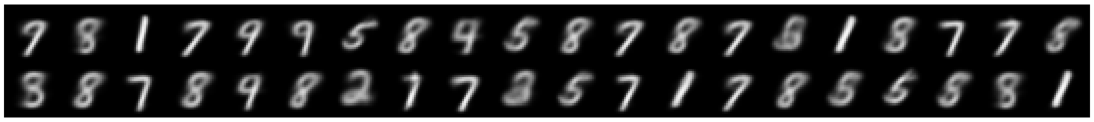
CNN VAE Gaussian(1,2) Latent Dim = 2



But the quality has stayed the same(the training was only done for a few epochs, fixed for both cases).

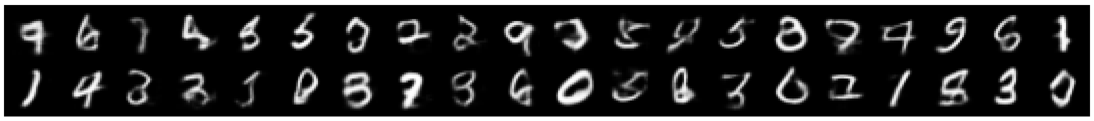
Further research led me to BetaVAE, which scaled up the KL Divergence loss during training and was shown to improve the generation quality. I also found that the latent dimensions might need to be increased from 2 for BetaVAEs, which required a bit of reworking.

CNN BetaVAE Gaussian(1,2) Latent Dim = 2

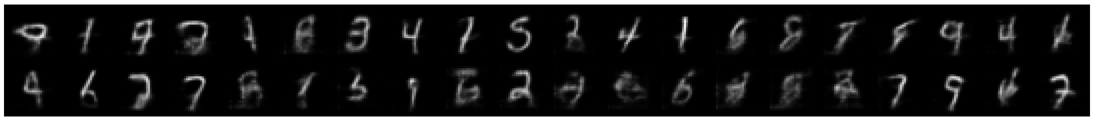


I increased the latent dimensions from 2 to 16 and repeated the experiments.

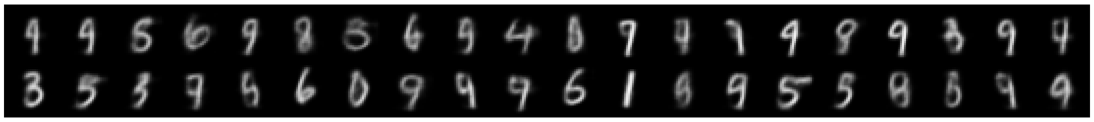
CNN VAE Normal(1,0) Latent Dim = 16



CNN VAE Gaussian(1,2) Latent Dim = 16



CNN BetaVAE Gaussian(1,2) Latent Dim = 16



The images generated with 16 latent dimensions appear sharper/thinner/(lighter than the previous smudged ones. We can observe a significant improvement in using BetaVAEs over VAE for the Gaussian(1,2) model.