zj722, 02215217

1. Network Training

In deep learning, using various optimization and regularization techniques can significantly enhance model performance. In this Task, I implemented some techniques and compared the outcomes against a baseline model that did not use any optimization or regularization methods. The performance of the baseline model are provided in Appendix 10.

To improve performance, I firstly applied **Data Augmentation** using a combination of random zoom, rotation, and translation. By adjusting the intensity of these transformations, I evaluated both the overall benefits and the effects of more aggressive augmentation settings on the model's performance. Under **moderate augmentation setting**, the model achieved higher validation accuracy and lower validation loss compared to the baseline. Surprisingly, the test result are even better than the training, because model may learn additional useful features from augmented data sets. Additionally, the gap between training and validation curves remained stable over epochs, indicating that augmentation also helped reduce overfitting. The performance are shown in appendix 11.

When using **aggressive augmentation**, performance was still better than the baseline but slightly worse than the moderate version. Heavier transformation on dataset can deviate from the original data, sometimes even disappearing the important features and misguide the model from training. Performance of model are shown in appendix 12.

By introducing **Dropout layer (rate 0.3)** after each fully connected layer, The loss and accuracy plot become smoother with a reduced overfitting. This better generalization is due to Dropout layer can prevent model from relying too much on specific features. Performance of model are shown in appendix 13.

Batch Normalization(BN) , in principle, can accelerate training by redistribute the output of each convolution layer into active region before pass into activation layer. However in my case, BN worse the performance and the reason for this is maybe from hyperparameter tuning. For example, if the learning rate is too high, the redistribution from BN can amplify oscillations, preventing stable convergence. Result is shown in appendix14.

The same behavior happens when **combining dropout with BN** The validation curve is oscillating and the gap between train and validation stays roughly constant rather than widening. In other words, BN plus Dropout is preventing overfitting on training data (so the gap doesn't grow), but the chosen hyperparameter may not allow the model to achieve higher validation accuracy. Result are shown in appendix15

I also assessed the importance of initialization, I changed the **initialization method** from random to set all initial neurons to zero. This result in a graph where training accuracy and loss curve to stay at a specific level (10%). The reason for this is by setting initial weight all to zeros, the forward propagation result is always zero, also, the backpropagation gradient update is also zero. So the model is actually not "learning" from training. Result shown in appendix 16

Lastly i assessed the effect on **Learning rate** , as expected, higher learning rate will result in faster convergence of training but also introduce more oscillations, shown in figure below 1

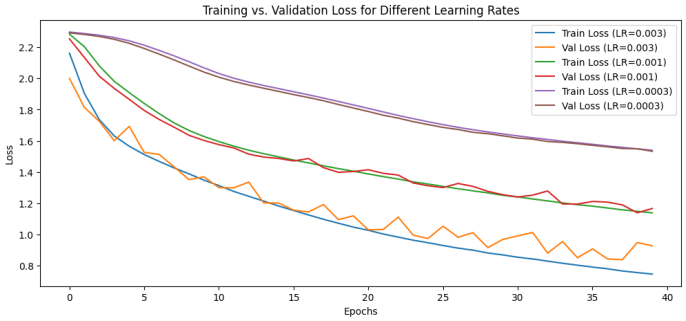

Figure 1: loss for different Learning Rate

The table 1 summarize the test accuracy of each of above methods.

Method	Validation Accuracy
Baseline (No Aug.)	75%
Moderate Data Augmentation 1	80%
Aggressive Data Augmentation 2	75%
Dropout (0.3)	80%
BN	75%
BN + Dropout	65%
Zero Initialization	10%

Table 1: Best validation accuracy across different methods.

2. CNN structures

Training methods can affect both the training time and the accuracy of the resulting model. I evaluated three different methods to train VGG-16 on Tiny-ImageNet dataset, and the corresponding accuracy and training time are presented in Table 2 (GPU used: A100).

When training VGG-16 from scratch, the training stopped after only five epochs, with both training and test accuracies remaining very low. This poor performance was due to inadequate hyperparameter tuning (e.g. learning rate and weight initialization). In this case, early stopping was triggered by the lack of improvement during the first few epochs, preventing the model from learning and converging to better performance. However, finding out the proper hyperparameter is challenging for such large model, So btter to use alternative training methods.

Second method i used is transfer learning. It converged at the 13th epoch. I leveraged a pre-trained VGG-16 model by freezing its convolutional base and replacing its dense layers with two new fully connected layers containing 1025 and 512 neurons respectively. As shown in Table 2, both accuracy and training time improved. This is because the model starts from a good pre-trained

initialization, which results in fast convergence(few epochs) and higher accuracy on the target dataset. The training time for each epoch is faster than train from scratch, because the model only need to update the fully connected layers which is relatively easy.

Finally, I assessed the fine-tuning method, which also converged around the 13th epoch, yielded the highest accuracy. In fine-tuning, all layers of the network are trainable. This allows the pre-trained weights to be adjusted specifically for the Tiny-ImageNet dataset, capturing its unique features more effectively than other two methods.

However, a clear issue with both transfer learning and fine-tuning is that they build upon a model already effective for image tasks, so further training may lead to overfitting, where training accuracy is far higher than test accuracy as shown in 2. In our experiments, fine-tuning has most overfitting, as its update all weights to fit the model more closely to the specific dataset. Moreover, compared to transfer learning, fine-tuning takes longer time for each epoch because it updates all parameters in the network rather than just the newly added fully connected layers.

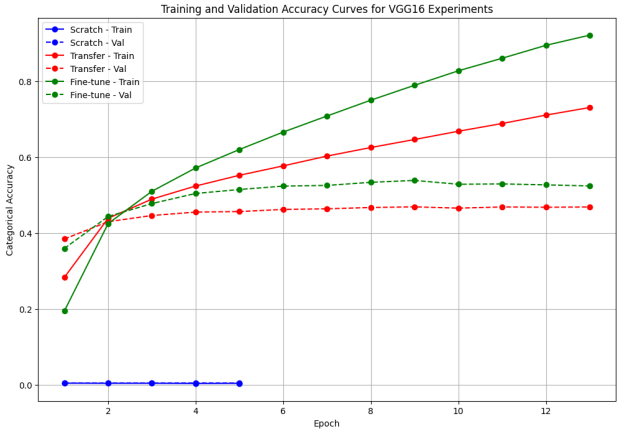

Figure 2: VGG-16 moedel performance with different training methods

Finally, to determine the best model for this task, I fine-tuned ResNet50, which achieved slightly better accuracy than VGG-16.

Method	Train Accuracy	Test Accuracy	Mean Time per epoch(s)	Time /image(ms)
VGG16				
scratch	0.5 %	0.5%	147.51	0.9897
Transfer	74%	46.5%	8.01	0.5218
Fine-tune	87.02%	52.7%	19.05	0.2516
ResNet50				
Fine-tune	97.2%	54.6%	18.8	0.2436

Table 2: Training result from different method

3. RNN

3.1. Task 1

RNN is a network structure that does not rely solely on the current input to generate its output, instead it can take historical inputs into account, allows generating output based on both current and past inputs.

A crucial hyperparameter that affects performance of RNN is window size, Which determines how many historical context the model can considers.

In theory, using a small window size may limit the model's ability to capture long-term trends, leading to more fluctuating predictions. Conversely, a large window size tends to produce smoother predictions but may fail to capture short-term rapid changes. Thus, selecting an appropriate window size according to characteristics of the dataset is important.

For our dataset, which exhibits a 12-month periodic data. I experimented window sizes of 1, 3, 6, and 12 months and compared the predictions from RNN with actual results (see Figure 3).

For a window size of 12, as expected, the model produced the smoothest predictions but failed to capture and follow the short-term changes, yielding the worst performance. In contrast, using a window size of 1 and 3, the model reacted more quickly to short-term changes and achieved the similar best performance in my tests. However, the potential drawback of a 1 month window is its more susceptibility to noise, potentially missing gradual long-term trends. So the 3-month window would be more trustworthy configuration for this dataset.

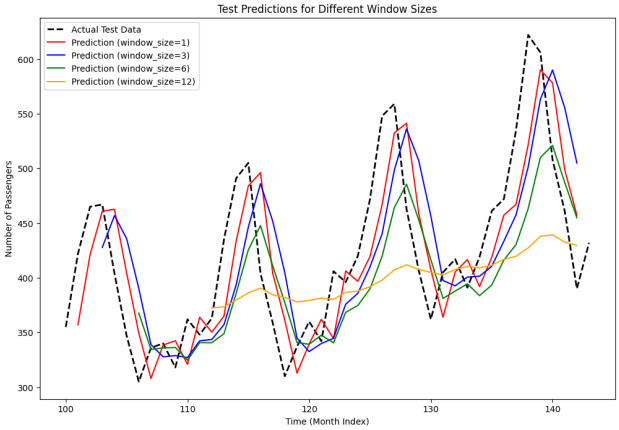

Figure 3: prediction based on different windowsize

3.2. Task 2

Model	**Test Accuracy (%)**	**Train Accuracy (%)**	**Example Review 1 prediction (neg)**	**Example Review 2 prediction (pos)**
Embedding_Model	0.8520	0.9093	0.34546518	0.34546518
LSTM_Model	0.8500	0.9938	0.1836538	0.4182204
LSTM_glove_Model	0.8558	0.9485	0.16760278	0.7029551

Table 3: Training result from different models

In this task, I evaluated the impact of an LSTM module and embeddings on a text classification model's accuracy. I trained three different models and present their results in Table 3. The table includes test accuracy and scores on two example sentences:

Example 1: "the movie is boring and not good"

Example 2: “the movie is good and not boring”

The embedding_model uses only an embedding layer with dimension 1 combined with a global average pooling layer. This setup limits performance on capture semantic nuances or other relations between words. For example, the difference between ”good” and ”boring” might not be explicitly shown in a scalar representation. Also, by relying solely on average pooling, the model disregards word order. Since both example sentences contain the same words, despite in a different order, the model outputs identical Scores.

The lstm_model uses a 300-dimensional trainable embedding layer followed by an LSTM. A higher embedding dimension allows the model to capture detailed semantic information for each word. E.g. model can distinguish between words ”good” and ”boring” in semantic level. The LSTM layer improves the long term memory, captures the order and context of words. This enables the model to understand that the phrase ”not boring” has a different sentiment than ”boring” by itself. As a result, the model assigns different scores to the two example sentences, reflecting its ability to distinguish based on context.

The lstm_glove_model, maintains the same LSTM architecture as lstm_model but initializes the embedding layer with pre-trained GloVe embeddings. GloVe embeddings provide high quality word embeddings. With GloVe embeddings as a starting point, the LSTM has a stronger foundation for modeling long-term dependencies and contextual meanings. This leads to best overall performance among three models.

The training and validation accuracy plots are present in appendix 17 18 19

3.3. Task3

In this task, I evaluate how temperature affects generated text and measured by the BLEU score, and compare the performance of word-level versus character-level models.

Temperature redistribute the predicted token. For low temperature, the probability distribution over the next token becomes very peaked, so the model tend to chooses the most likely token. As a result, the generated text is highly predictable and close to the training data. This leads to a higher BLEU score because the n-gram overlap with the reference text is larger.

In contrast, for low temperature, the distribution becomes flatter, giving less likely tokens a greater chance from being selected. This produces more creative text, but it can also lead to less coherent or grammatically correct output. Consequently, the BLEU score tends to decrease at higher temperatures.

Also, BLEU score usually higher in word level model, this is because for character level model. n-gram overlaps are computed over characters, so even small changes (e.g. a missing or extra letter) can affect the score.

However, there are some benefits of character level model which can not be reflected by the BLEU score. Since it operate on individual characters, it can generate words that were never seen during training. so the output are usually more creative.

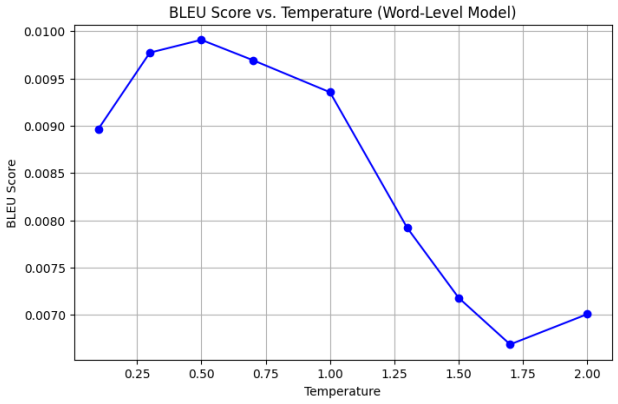

Figure 4: word-level score with temperature

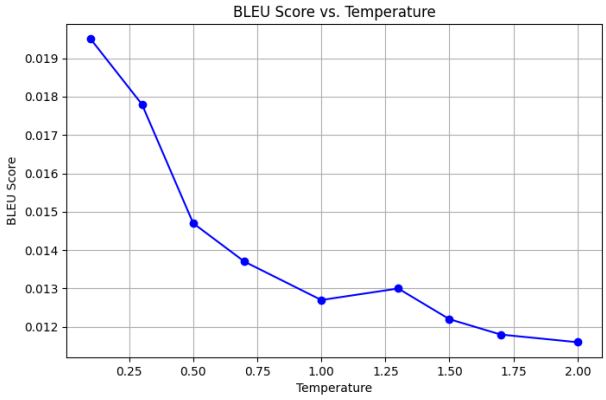

Figure 5: Character level score with temperature

4. Autoencoder

4.1. Task 1

In this task, I assessed the effect of different structure on performance of Autoencoders. Specifically PCA, neural network with non-linear activation and CNN architecture.

By using **MLP with non-linear activation functions**. The network is able to capture the non-linear features of the input due to non-linear activation function. The structure of the network is shown in appendix Figure 20.

This structure has neurons in each hidden layer of the encoder and decoder changes gradually. For the encoder, this helps avoid losing useful features too quickly. For decoder this helps reconstructing input by successively expanding latent representations back to the original 32×32 size. Each layer uses a ReLU activation function to capture non-linear features.

Additionally, **Batch Normalization (BN)** is applied before each activation layer to prevents exploding or vanishing gradients and speeds up training. There is no BN layer before the first layer because the input data is already normalized to the range [0, 1]. Also **Dropout layer** is applied in encoder to avoid overfitting.

Because MNIST is a relatively simple dataset; using too many dropout could potentially destroy important features. So I chose to use only one dropout layer.

The second Autoencoder utilizes a **CNN-based architecture**, following a similar idea as before. In the encoder, I used four convolutional blocks. In each block, a **Batch Normalization (BN)** layer is applied before the ReLU activation function. The number of filters in each block is gradually increased, ensuring that the encoder captures detailed features from the input.

In the decoder, I begin by adding a fully connected layer that maps the 1×10 dimensional bottlenect back to a matrices of dimension $4 \times 4 \times 128$ this guarantees that there are 2048 data points available to represent and reshape bottleneck into feature map. Next, using Conv2DTranspose layers with a stride of 2 to gradually doubled the dimension (e.g from 4×4 to 8×8). while the number of filters is gradually decreased to reduce the number of channels. This process ultimately reconstructs the image to the original size of $32 \times 32 \times 1$. Structure of network are illustrate in appendix 21

Table 4 shows that the autoencoder, which captures non-linear features, is crucial for achieving low MSE. When using convolution layer connect with a classifier, this structure can result in accuracy about 95%. In contrast, PCA, which only captures linear features, yields a higher MSE, and even with a classifier, its accuracy is only around 80%.

Method	Test MSE	Train MSE	Test Acc	Train Acc
Non-Conv	0.008019	0.007398	0.9131	0.9110
CNN	0.8139	0.007672	0.9475	0.9465
PCA	0.0256	0.0258	0.8140	0.8097

Table 4: Train and Test MSE and accuracy for each structure.

4.2. Task2

Loss function used during training also affect the performance of result model, as it can directly determine how model is updated. I did experiment with four different loss functions.

MSE heavily penalizes large pixel-level deviations, yielding moderate performance but not best overall reconstruction.

MAE use absolute error, so compare to MSE, penalized error lighter, so it may not force errors to be extremely small, so training process become smoother and the reconstruction often looks more natural.

SSIM focuses on structural similarity rather than exact pixels, so it has higher MSE but can produce perceptually better images.

MS-SSIM refines SSIM across multiple scales, so model trained based on both global and local features, this achieving better pixel match.

Loss function	Valid MSE	Train MSE
MSE	0.0058	0.0060
SSIM	0.0075	0.0072
MS-SSIM	0.0055	0.0053
MAE	0.0055	0.0054

Table 5: Train and Test MSE for different loss function.

Below are denoising result from using MAE and MS-SSIM error function. which along with previous discussion. Result for MSE and SSIM are shown in appendix 2223

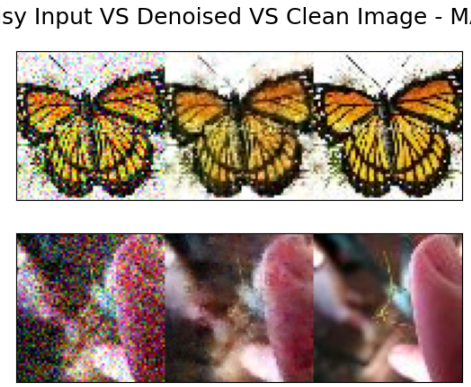

Figure 6: MAE denoising

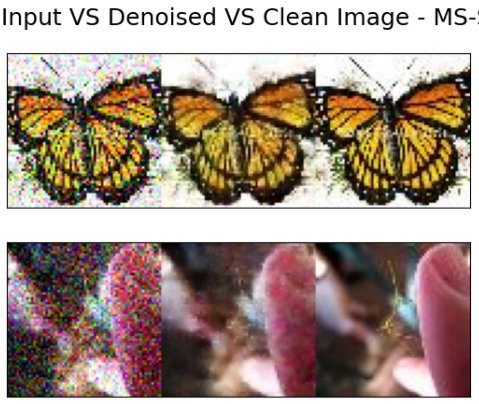

Figure 7: MS-SSIM denoising

5. VAE_GAN

5.1. Task 1

This section focusing on generative model, which aims to learn the data distribution from a given set of training samples so it can generate new outputs that fit that distribution.

VAE are optimized by an objective function with two term, one encourages the model to reconstruct the input more accurately, second (KL divergence) encourages the encoded distributions for all samples to be close to a common prior $\log P_\theta(z)$. To examine

the importance of the KL divergence, I trained the same VAE both with and without the KL term, then measured performance using MSE and the Inception Score(IS). The results are shown in Table6.

When with KL layer, MSE is slightly higher, because the model does not focus solely on perfectly reconstructing each input. Instead, it also ensures that latent "structure" learned by model align with the input sample's distribution. Moreover, IS is higher, indicating more diverse and realistic samples that follow the learned distribution rather than simply overfitting to training samples.

A different generative model structure is GAN, it contains two part, generator and discriminator. It use an adversarial loss, training a discriminator to distinguish between real and fake from generated data from generator, this pushes the generator to create sharper and more realistic images. As an result GANs often achieve a higher Inception Score (IS) since the generated samples are more realistic. However, this adversarial loss training can easily failed, which requires a careful hyperparameter tuning and training to converge model to desired result.

Model	MSE	Inception score
VAE with KL	0.0122	7.2024
VAE without KL	0.0106	5.9113
GAN	-	8.2219

Table 6: MSE and IS for Two VAE and a GAN model

5.2. task2

In this section, I used both a cGAN and a UNet autoencoder to colorize the same black-and-white images, then compare their results.

cGAN and the UNet achieve similar MAE values on the test set (0.0458 vs. 0.0449, respectively), However, they produce visually different outputs. Theoretically, cGAN training encourages its generator to produce images with sharper details and more realistic colors to misguide discriminator. In contrast, the UNet trained solely with MAE loss focuses on minimizing pixel-wise errors, without caring output is close to real or not.

As shown in figure 24 in appendix, Despite the resolution is too low to see detailed difference, it is clear that the cGAN's colorization aligns more closely to realistic, particularly on the "animal's body" part, Hence, the cGAN is better result compare with uNet autoencoder.

6. RL

In this task, im using Q-learning combined with ϵ-greedy or Softmax to implement reinforcement learning.

Under an ϵ -greedy strategy, the agent explores more randomly at the beginning of training, then increasingly shifts toward a greedy policy as ϵ decays, often leading to a rapid initial performance boost followed by a plateau. By contrast, with a Softmax strategy, the degree of exploration is controlled by a temperature parameter, where a higher temperature promotes more uniform exploration and a lower temperature makes the policy more greedy. This can result in a smoother learning curve, slightly slower convergence compared to a purely ϵ-greedy approach.

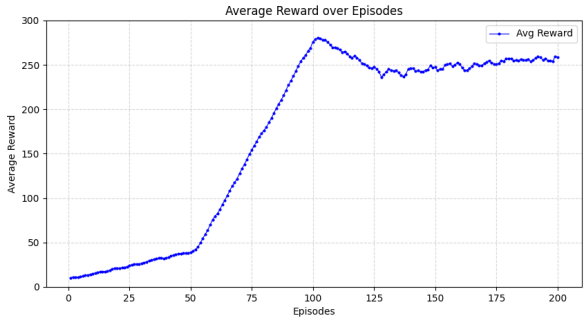

Figure 8: Q-learning with ϵ-gready

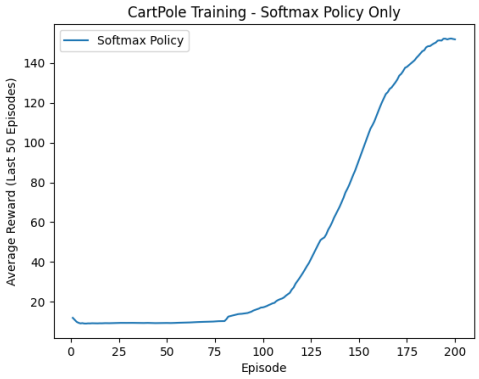

Figure 9: Q-learning with softmax, Temperature 0.05

References

7. Appendix

7.1. Network Training

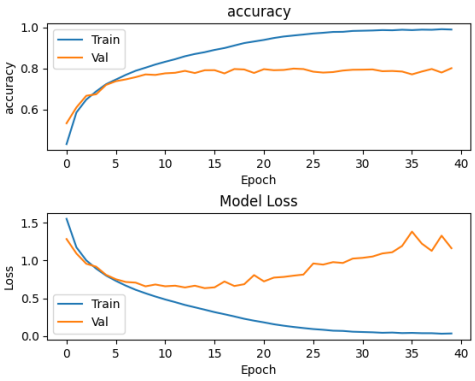

Figure 10: Baseline model performance

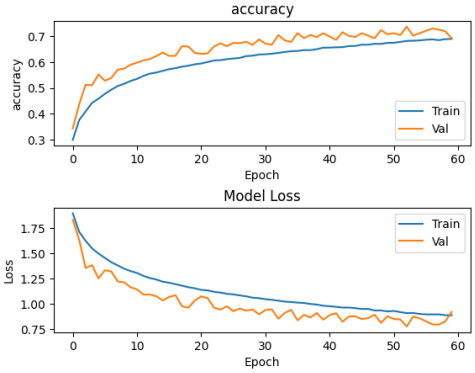

Figure 11: Moderate data augmentation

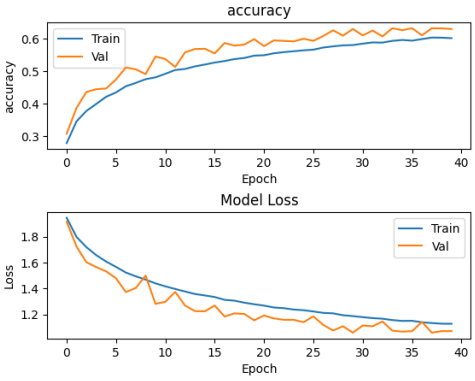

Figure 12: Aggressive data augmentation

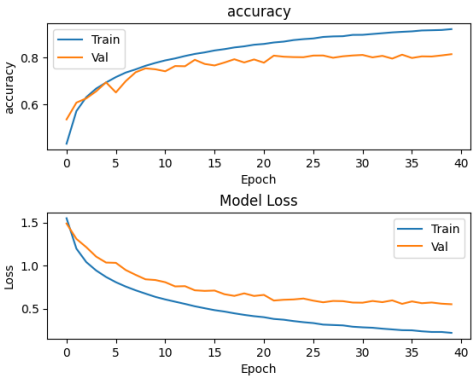

Figure 13: Model with Dropout (0.3)

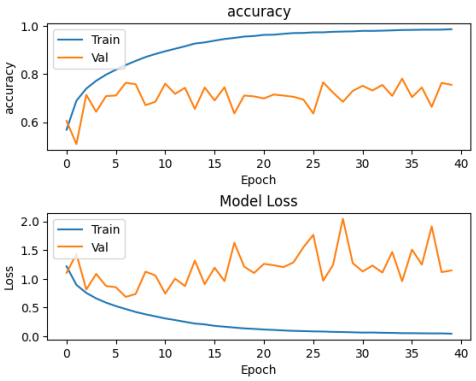

Figure 14: Model with BN

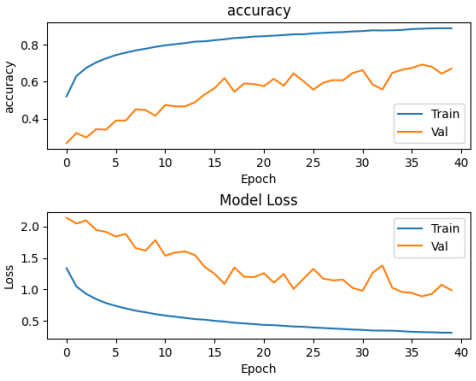

Figure 15: Model with BN+Dropout(0.3)

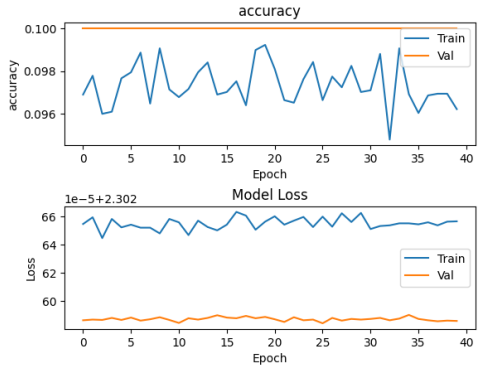

Figure 16: Model with zero initialization

7.2. Additional train and validation plot for RNN task2

.

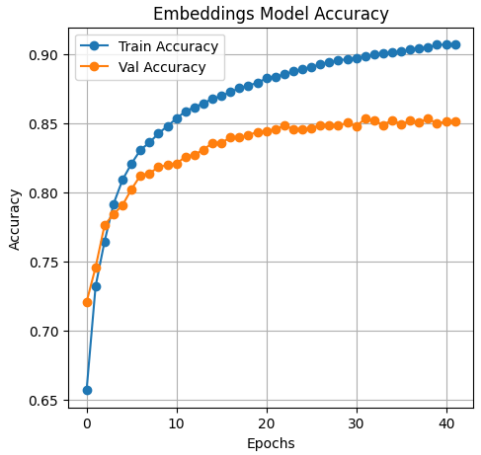

Figure 17: Train and valid accuracy for embedding_model.

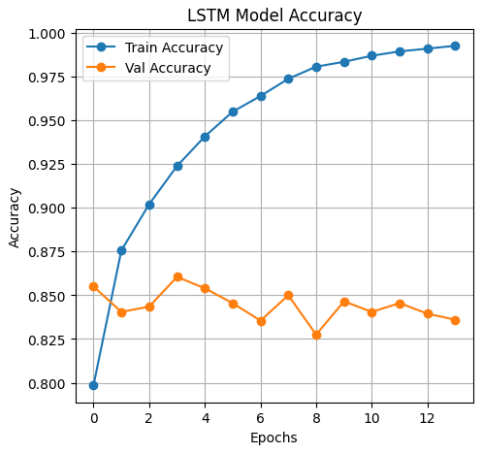

Figure 18: Train and valid accuracy for lstm_model.

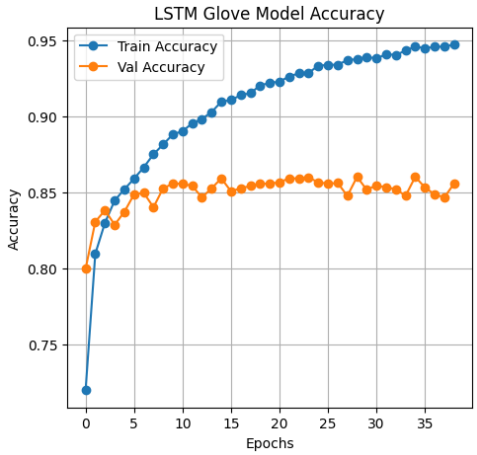

Figure 19: Train and valid accuracy for lstm_glove_model.

7.3. Autoencoder Task1

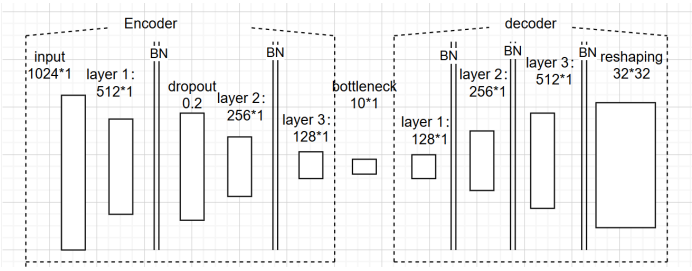

Figure 20: Non linear autoencoder MLP

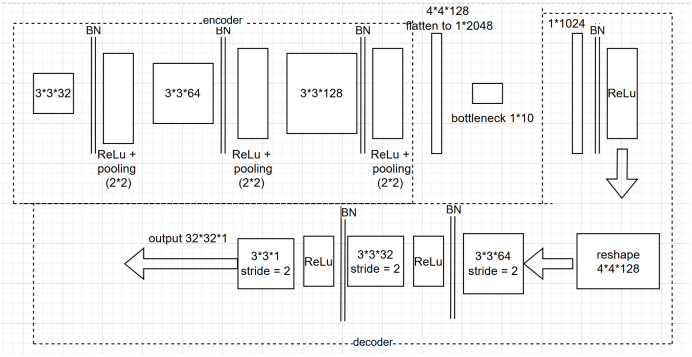

Figure 21: Non linear autoencoder CNN

7.4. Autoencoder Task2

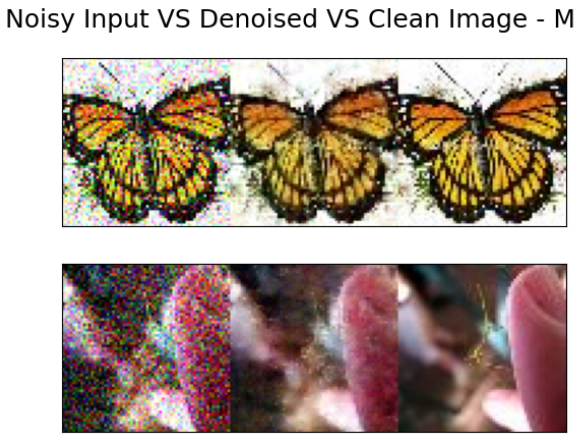

Figure 22: MSE denoising

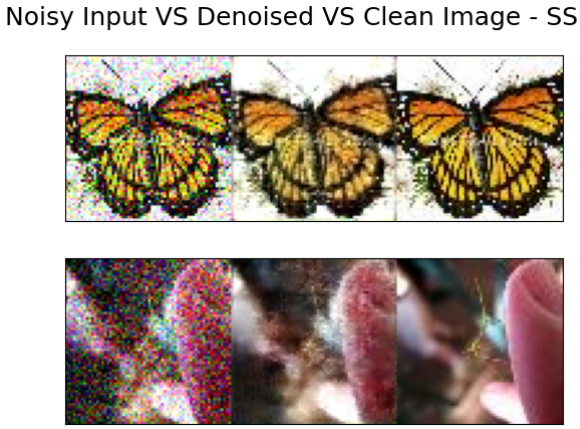

Figure 23: SSIM denoising

7.5. VAE_GAN Task 2

Figure 24: Coloring result

7.6. RL DQNAgent modification

In the provided DQNAgent code, the $replay$ function calculates the target by taking the maximum $Q - value$ predicted for $next\_state$, reflecting the Q-learning update, while the act method uses an ϵ-greedy strategy. To implement Softmax, the $Qvalues$ should changed from $model.predict$ into a provided $softmax$ function. changing temperature parameter controlling the degree of exploration. Other remains unchanged.