Adam Horsler, ah2719, 01738592

3. Network Training

Task 3.1 The small data augmentation uses a 0.05 random zoom and rotation, as well as a 0.01 random translation (width and height). The large data augmentation uses a 0.2 random zoom, 0.3 random rotation and 0.2 random translation (width and height). The results show that the best performing model in terms of validation accuracy was with no data augmentation, and the worst performing model had large data augmentation.This is because data augmentation introduces artificial variances in the dataset, which changes the probability distribution of it. This change is not done in the validation (test) set, thus the model has learnt a perturbed distribution to the test set, and so performs slightly worse.
Dropout + Batch Normalisation produced the best validation accuracy. This is because dropout helps the model to learn by introducing different connections to the network due to the randomness of which nodes are dropped[4]. In addition, it helps prevent overfitting by removing a fraction of the model complexity during training. Batch normalisation allows for faster and better convergence by using the batch's 1st and 2nd statistical moments between layers. For the models used, dropout was added after the 1st, 2nd and 4th convolution layers, and batch normalisation was added after every convolution layer.
The zeros kernel initialisation method was considerably worse than other initialisers. This is likely due to the model getting stuck in a local minimum, as the accuracy did not improve with training.

Model	Best Val Accuracy
No data augmentation	0.7979
Small data augmentation	0.7862
Large data augmentation	0.6621
Dropout	0.8086
Batch Norm	0.7932
Dropout and Batch Norm	0.8153
Zeros Kernel Initialisation	0.100

Table 1: Network Training Strategies and their Best Validation Accuracy

The SGD learning rate of $3e^{-3}$ produced the best results in training and validation, while $1e^{-3}$ produced the worst results. As $3e^{-3}$ is the largest of the learning rates, this shows that a lower learning rate causes the model to learn slower, as the losses are still consistently decreasing even at epoch 40. However, increasing the learning rate too far will result in unstable training as the model can fail to converge.

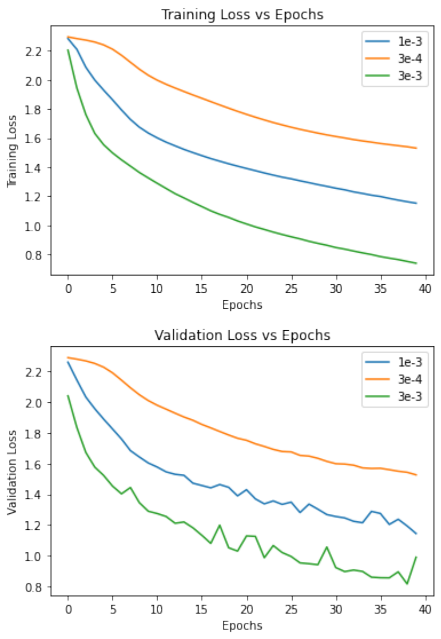

Figure 1: Training and Validation Loss Curves for different SGD Learning rates.

4. Common CNN Architectures

Task 4.1 The Training and Validation accuracy plots show that VGG16 trained from scratch performed the best. This is because the model was able to learn using all layers. Fine tuning and transfer learning methods had most of their layers trained on a similar but different dataset, thus the weights learnt corresponding to a slightly different feature space, resulting in worse performance. Transfer learning performed significantly worse in the validation accuracy as all convolution layers were frozen in this method, compared to the last convolutional layer being unfrozen in the fine tuning method. This last convolutional layer therefore had a critical role in learning the high-level feature space.

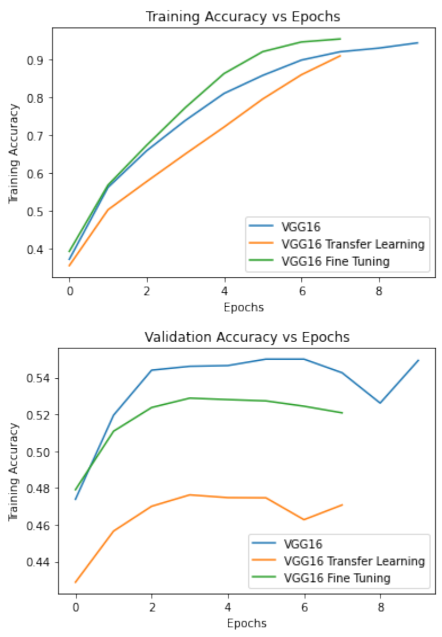

Figure 2: VGG16, Transfer Learning and Fine Tuning Training and Validation Plots

Fine tuning and transfer learning had significantly lower training times due to most of the model weights being frozen, reducing training complexity, so less computation is required.

Model	Best Test Accuracy	Inference Time	Epochs
VGG16	0.5490	62.4	10
VGG16 (Fine Tuning)	21.26	10	
VGG16 (Transfer Learning)	23.80	10	

Table 2: VGG16 Results and Inference Times. Device Used: /device:GPU:0

5. RNN

Task 5.1 The figure show the effect of different window sizes on training and testing.

The window size training and test MSE's are shown in the table. Window sizes of 1, 3 and 5 all produce similar training and test MSE's. This shows that there is not much different in performance between these model sizes.
10, 20 and 30 window sizes all showed decreasing training MSE in ascending order of window size. However, there was a significant different between the 20 window size and rest of the models' test MSE, as the 20 window size experienced a drastic increase in test MSE (over 3x the test MSE of 10). This is likely due to the 20 window size model learning extra noise not incorporated from other window sizes, simply due to the nature of the dataset. The 30 window size likely did not learn this noise as it was able to filter it out due to its larger window size, while the window sizes below 20 didn't have a window size big enough to experience this noise.

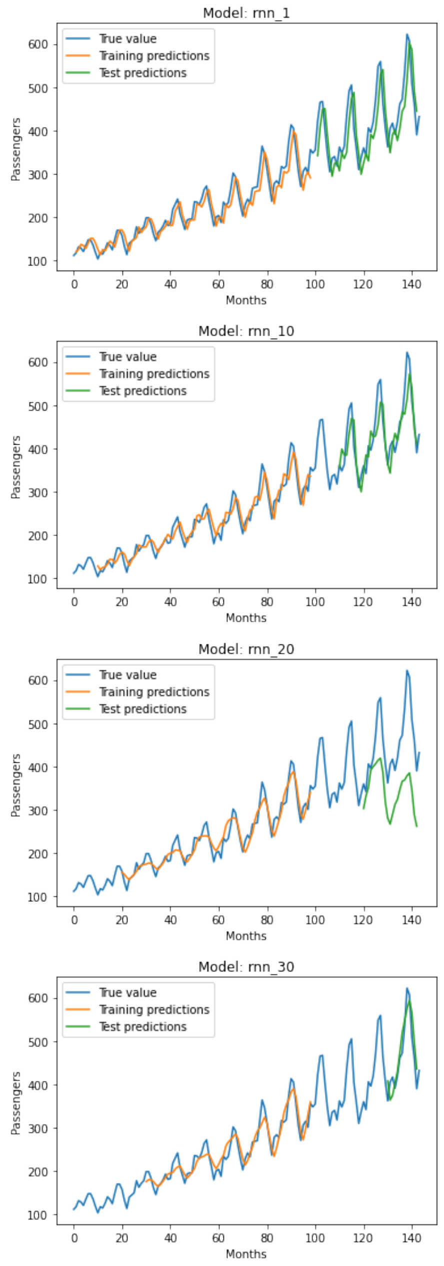

Figure 3: Training and Validation Plot of different window sizes

Task 5.2 From the table of results, all 3 models had similar test accuracy. This shows that they were all able to learn the input feature space and word embeddings.

Window Size	Training MSE	Test MSE
1	24.26	51.87
3	24.42	53.77
5	24.90	56.61
10	21.54	42.46
20	19.21	123.55
30	20.75	36.36

Table 3: Window Size Results

However, the sentiment analysis provided significantly different results. The input sentence contained "good" - which is in general a positive word, and "boring", which is in general a negative word. However, the sentence was structured such to switch the negation (by using a "not" word), thus adding a level of confusion to the review. In this instance, LSTM had the best results for both positive and negative predictions. This is due to LSTM being able to learn the specific word embeddings specifically from the dataset, whereas the GLoVe model has pre-learnt word embeddings. This resulted in the GLoVe model having an incorrect prediction for the positive sentence. The embeddings model gave the same result for both sentiment predictions as it is not able to learn the negation due to its low feature space - dimensionality 1.

Model	Best Test Accuracy	Negative Sentiment Prediction	Positive Sentiment Prediction
Embeddings	0.8522	0.3636	0.3636
LSTM	0.8388	0.0320	0.7755
LSTM GLoVe	0.8663	0.0469	0.4703

Table 4: RNN Model Results and Sentiment Predictions

Task 5.3 The BLEU score is an attempt to measure the syntactic and grammatical sense of generated text against a set of high quality reference translations. For the character model, as temperature increases, the generated text becomes less human-like and more useless. The BLEU score reflects this as it decreases when temperature increases, as shown in the figure. However, while a temperature of 0 achieved the best BLEU score, an inspection of the generated text shows that the same sentence was repeated, and other text was similar. This shows a lack of variance in the model generating text, as a low temperature (e.g. T = 0) results in choosing the option with the highest probability from an output probability vector. On the other hand, a higher temperature (e.g. T = 2) produced more varying outputs, but in general making less sense. Thus, a temperature somewhere between these extremes may be optimal.
The word level temperature graph shows a different shape, as the optimal temperature for BLEU was 1, compared to 0 for the character level. The word level model benefits more from a higher temperature due to the lower feature space for words compared to characters. It is more likely a sentence will make more sense when only predicting the next word compared to predicting the next character. Thus, the character model often produced incorrect words, whereas the word model is guaranteed to produce correct words.
Both models had the problem of producing the same sentence multiple times when T = 0.

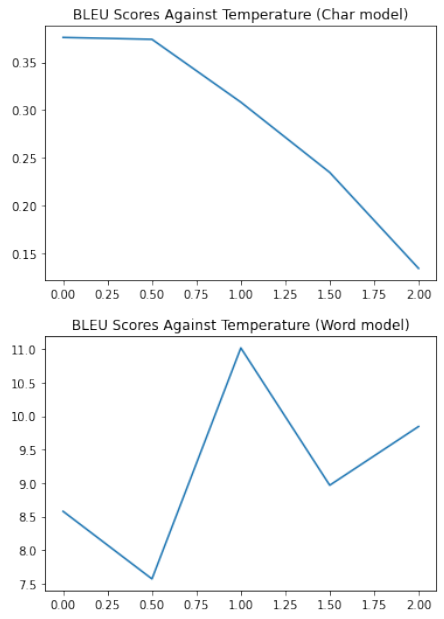

Figure 4: Character and Word Level BLEU Scores vs Temp

6. Autoencoders

Task 6.1 All autoencoder model architectures are shown in the Appendix. The autoencoder and convolutional autoencoder produced a significant improvement on MSE compared to PCA, by a magnitude of at roughly 10x. The convolutional autoencoder had an MSE of 0 to 4 decimal places, a significant improvement on the standard autoencoder. Although PCA uses maximal variance to compute the dimensions with the most information[1], thus minimising information loss during dimensionality reduction, it does so through the limitation of linear functions. However, autoencoders using non-linear activation functions are able to learn more complex functions related to the input data, thus improving its performance as it doesn't have this PCA limitation.
In addition, PCA features are linearly uncorrelated as they are projections onto an orthogonal basis[2]. The data may however have linear correlations, which autoencoders are able to learn, further improving their performance.
These performance improvements are evident in the

classifier accuracy, with the convolutional autoencoder (no skip connections) resulting in an accuracy of roughly 10% higher than PCA. It can be noted that PCA still achieves a high classification accuracy of 80%.
The autoencoder and convolutional autoencoder both incorporate skip connections. This helps the models learn as initial features learnt can be easily passed through the network using these skip connections. Without skip connections, there is a risk of the low-level features from the initial layers being lost as the dimensionsality of the network decreases. However, this causes the model to not create an accurate latent space, as evident in the autoencoder and convolutional autoencoder classification accuracies which were significantly worse than PCA and no skip connections.

Model	Training MSE	Validation MSE	Training Classifier Accuracy	Validation Classifier Accuracy
PCA	0.0258	0.0256	0.8100	0.8144
Autoencoder	0.0031	0.0030	0.3980	0.3996
Conv Autoencoder	0.0000	0.0000	0.1477	0.1527
Conv Autoencoder (no skip connections)	0.0185	0.0192	0.9256	0.9286

Table 5: Autoencoder Results

7. Varational Encoders (VAE) and Generative Adverserial Networks (GAN)

Task 7.1 KL-Divergence quantifies the distance between 2 probability distributions. The VAE model can therefore maximise both good reconstruction of the input sample as well as similar representation for each input[3]. This is in contrast to autoencoders, which do not have sensible representations for data in between the learnt feature spaces. The MSE score for no KL-Divergence is therefore lower as this causes the model to behave like a standard autoencoder, which seeks to just learn the most important features of the input data.
However, the KL-Divergence loss enabled the VAE model to have a better Inception Score (IS), as IS seeks to measure the quality of generated images. The GAN model performed best on IS score as the VAE model has a problem of producing less sharp samples when compared to a GAN model.

Task 7.2 The UNet autoencoder model had a lower MAE score than the cGAN. This is because the UNet model's job is similar to that discussed in 7.1. The cGAN, however, learns to generate seemingly 'real' data.
The generated pictures illustrate that the cGAN produces

Model	MSE	Inception Score
VAE	0.0114	7.486
VAE (no KL Divergence)	0.0106	6.080
GAN	-	8.249

Table 6: VAE and GAN Results

more realistic images, whereas the UNet model has learnt to minimise MAE loss by using much less colour, i.e. RGB value closer to black and white.

Model	MAE
UNet	0.0449
cGAN	0.0458

Table 7: UNet and cGAN MAE Score

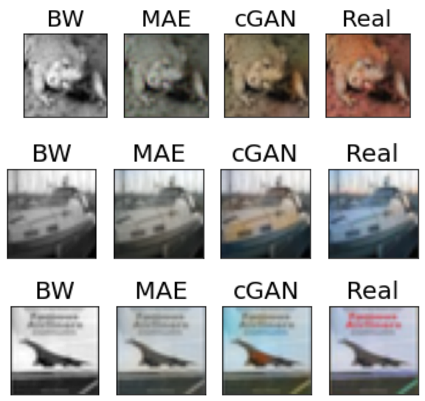

Figure 5: UNet vs GAN Visualisation Results

8. Reinforcement Learning (RL)

Task 8.1 The figure shows that the softmax function was not as effective as the epsilon-greedy policy when comparing last 50 average episode rewards. This is likely due to the added hyper parameter when using the softmax policy - the temperature. This adds more complexity to the model as the hyper parameter must also be optimised. However, it should be noted that both models should eventually converge to the same optimal functional approximation of the Q-function.

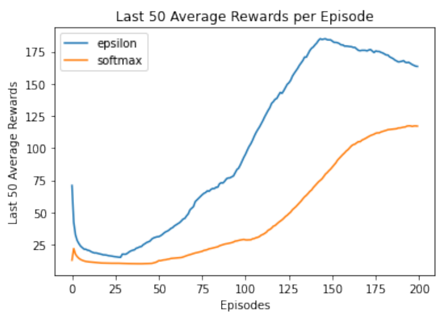

Figure 6: Last 50 Average Rewards Plot

References

[1] K. M. . C. Ciliberto. Data representation and learning. page 27.

[2] K. M. . C. Ciliberto. Data representation and learning. page 25.

[3] K. M. . C. Ciliberto. Generative models. page 21.

[4] K. M. . C. Ciliberto. Network training. page 14.

9. Appendix

9.1. Appendix A: Task 3.1 Data Augmentation

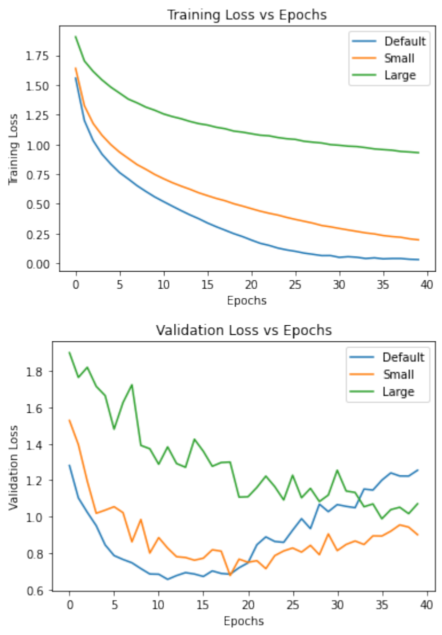

Figure 7: Data Augmentation Training and Validation Loss Plots

9.2. Appendix B: Task 3.1 Dropout and Batch Norm

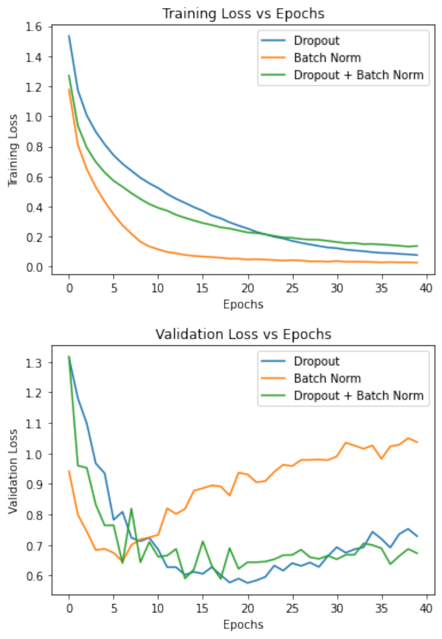

Figure 8: Data Augmentation Training and Validation Loss Plots

9.3. Appendix C: Task 5.2 RNN Models

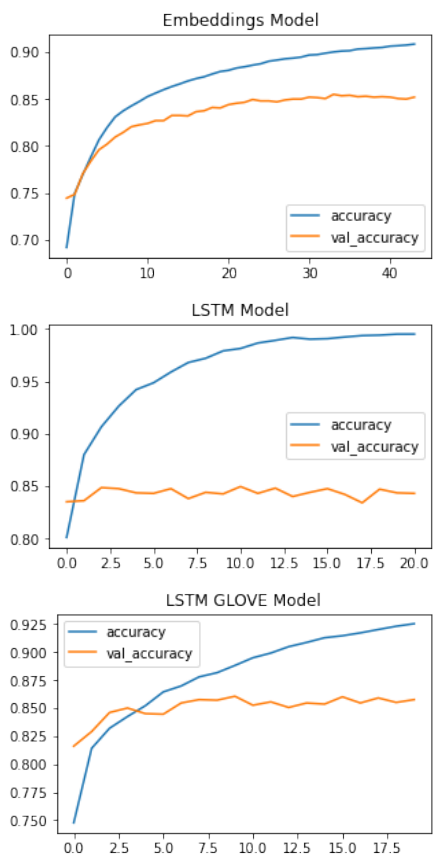

Figure 9: Different RNN Models and their Training and Validation Plots

9.4. Appendix D: Task 6.1 Autoencoder Model Architectures

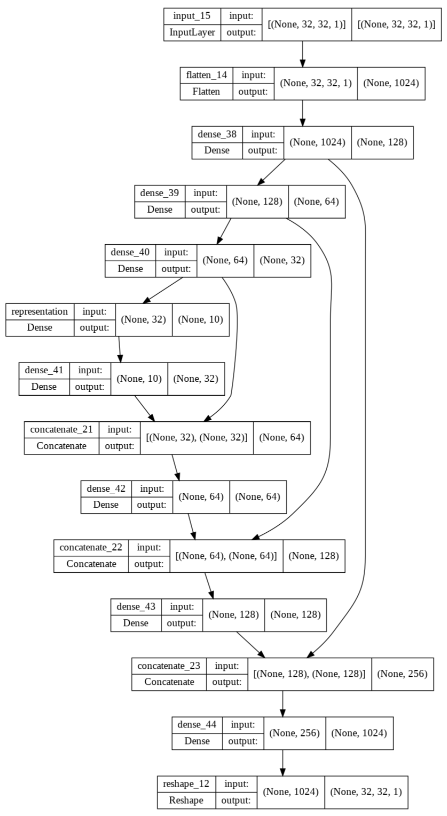
input_15 InputLayer input: output: [(None, 32, 32, 1)] [(None, 32, 32, 1)]
flatten_14 Flatten input: output: (None, 32, 32, 1) (None, 1024)
dense_38 Dense input: output: (None, 1024) (None, 128)
dense_39 Dense input: output: (None, 128) (None, 64)
dense_40 Dense input: output: (None, 64) (None, 32)
representation Dense input: output: (None, 32) (None, 10)
dense_41 Dense input: output: (None, 10) (None, 32)
concatenate_21 Concatenate input: output: [(None, 32), (None, 32)] (None, 64)
dense_42 Dense input: output: (None, 64) (None, 64)
concatenate_22 Concatenate input: output: [(None, 64), (None, 64)] (None, 128)
dense_43 Dense input: output: (None, 128) (None, 128)
concatenate_23 Concatenate input: output: [(None, 128), (None, 128)] (None, 256)
dense_44 Dense input: output: (None, 256) (None, 1024)
reshape_12 Reshape input: output: (None, 1024) (None, 32, 32, 1)

Figure 10: Autoencoder Model Architecture

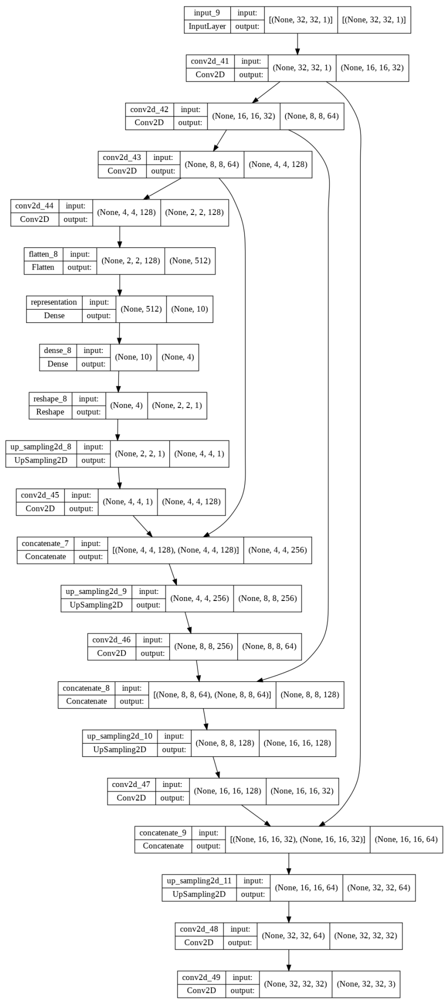
input_9 | input: | [(None, 32, 32, 1)] | [(None, 32, 32, 1)]
InputLayer | output:
conv2d_41 | input: | (None, 32, 32, 1) | (None, 16, 16, 32)
Conv2D | output:
conv2d_42 | input: | (None, 16, 16, 32) | (None, 8, 8, 64)
Conv2D | output:
conv2d_43 | input: | (None, 8, 8, 64) | (None, 4, 4, 128)
Conv2D | output:
conv2d_44 | input: | (None, 4, 4, 128) | (None, 2, 2, 128)
Conv2D | output:
flatten_8 | input: | (None, 2, 2, 128) | (None, 512)
Flatten | output:
representation | input: | (None, 512) | (None, 10)
Dense | output:
dense_8 | input: | (None, 10) | (None, 4)
Dense | output:
reshape_8 | input: | (None, 4) | (None, 2, 2, 1)
Reshape | output:
up_sampling2d_8 | input: | (None, 2, 2, 1) | (None, 4, 4, 1)
UpSampling2D | output:
conv2d_45 | input: | (None, 4, 4, 1) | (None, 4, 4, 128)
Conv2D | output:
concatenate_7 | input: | [(None, 4, 4, 128), (None, 4, 4, 128)] | (None, 4, 4, 256)
Concatenate | output:
up_sampling2d_9 | input: | (None, 4, 4, 256) | (None, 8, 8, 256)
UpSampling2D | output:
conv2d_46 | input: | (None, 8, 8, 256) | (None, 8, 8, 64)
Conv2D | output:
concatenate_8 | input: | [(None, 8, 8, 64), (None, 8, 8, 64)] | (None, 8, 8, 128)
Concatenate | output:
up_sampling2d_10 | input: | (None, 8, 8, 128) | (None, 16, 16, 128)
UpSampling2D | output:
conv2d_47 | input: | (None, 16, 16, 128) | (None, 16, 16, 32)
Conv2D | output:
concatenate_9 | input: | [(None, 16, 16, 32), (None, 16, 16, 32)] | (None, 16, 16, 64)
Concatenate | output:
up_sampling2d_11 | input: | (None, 16, 16, 64) | (None, 32, 32, 64)
UpSampling2D | output:
conv2d_48 | input: | (None, 32, 32, 64) | (None, 32, 32, 32)
Conv2D | output:
conv2d_49 | input: | (None, 32, 32, 32) | (None, 32, 32, 3)
Conv2D | output:

Figure 11: Convolutional Autoencoder Model Architecture

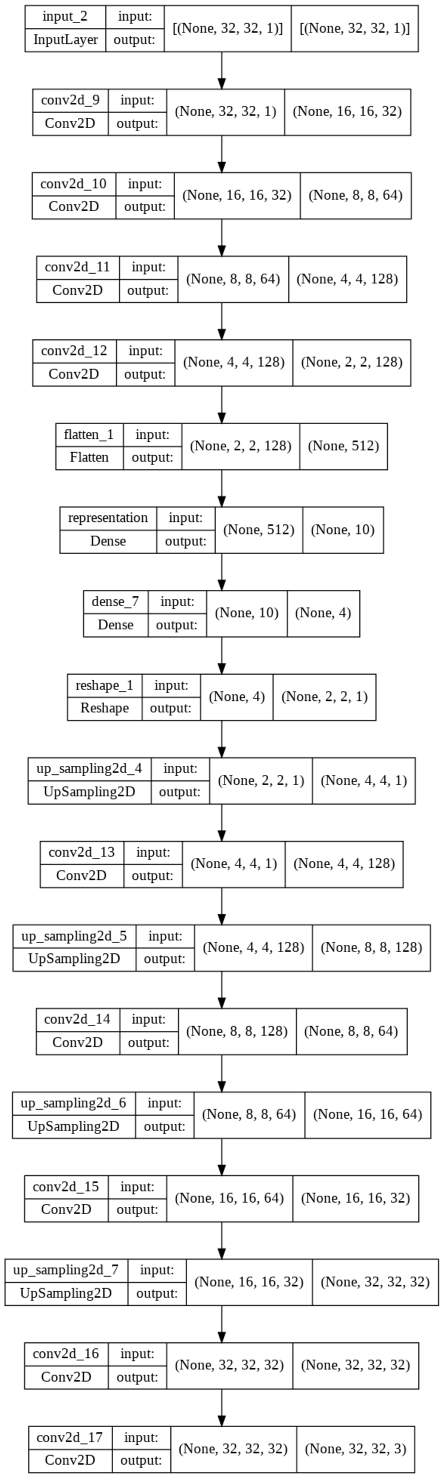
input_2 | input: | [(None, 32, 32, 1)] | [(None, 32, 32, 1)]
InputLayer | output:
conv2d_9 | input: | (None, 32, 32, 1) | (None, 16, 16, 32)
Conv2D | output:
conv2d_10 | input: | (None, 16, 16, 32) | (None, 8, 8, 64)
Conv2D | output:
conv2d_11 | input: | (None, 8, 8, 64) | (None, 4, 4, 128)
Conv2D | output:
conv2d_12 | input: | (None, 4, 4, 128) | (None, 2, 2, 128)
Conv2D | output:
flatten_1 | input: | (None, 2, 2, 128) | (None, 512)
Flatten | output:
representation | input: | (None, 512) | (None, 10)
Dense | output:
dense_7 | input: | (None, 10) | (None, 4)
Dense | output:
reshape_1 | input: | (None, 4) | (None, 2, 2, 1)
Reshape | output:
up_sampling2d_4 | input: | (None, 2, 2, 1) | (None, 4, 4, 1)
UpSampling2D | output:
conv2d_13 | input: | (None, 4, 4, 1) | (None, 4, 4, 128)
Conv2D | output:
up_sampling2d_5 | input: | (None, 4, 4, 128) | (None, 8, 8, 128)
UpSampling2D | output:
conv2d_14 | input: | (None, 8, 8, 128) | (None, 8, 8, 64)
Conv2D | output:
up_sampling2d_6 | input: | (None, 8, 8, 64) | (None, 16, 16, 64)
UpSampling2D | output:
conv2d_15 | input: | (None, 16, 16, 64) | (None, 16, 16, 32)
Conv2D | output:
up_sampling2d_7 | input: | (None, 16, 16, 32) | (None, 32, 32, 32)
UpSampling2D | output:
conv2d_16 | input: | (None, 32, 32, 32) | (None, 32, 32, 32)
Conv2D | output:
conv2d_17 | input: | (None, 32, 32, 32) | (None, 32, 32, 3)
Conv2D | output:

Figure 12: Convolutional Autoencoder (no skip connections) Model Architecture