**COSC420 A1**

The folders associated with this file contain training data for the code files attached. The file labelled saved contains training data for part 1 of the assignment: classification and regression, and the files labelled diff\_saved and diff2\_saved respectively contain each attempt at diffusion detailed in the document below.

The dataset the model is to be conducted on is the small NORB dataset, which contains 96x96 pixel grey-scale images of 50 toys classified into 5 categories: four-legged animal, human figure, airplane, truck, or car. These images are taken by two cameras under varied lighting, angles, and elevation.

The initial data set consisted of 24300 sets of size 96x96 images, each set containing 2 images of the same classified object. The dataset is of shape [24300, 96, 96, 2].

The parameters of the images recorded – the labels as such, include the object instance, the angle from which the image was taken, the label of the image (the category under which the image falls), the elevation of the camera, and the lighting conditions.

As each of the 24300 image sets contain 2 images taken from separate, the image sets contain 2 slightly different representations of the same data. Due to this, it could be hypothesised that using both images in each set can reduce the generalisability of the model as similar representations of data. On the other hand, this multi representation can function to provide some variability in the data, as well as a greater quantity of data. Due to the insignificant amount of detail accessible about the given dataset, it is not reasonable to assume that any 2 images of a set are similar enough to average them out to represent their category equally and accurately.

For the sake of this paper, we will be separating the image set photos in a way so that the images can be evaluated individually, as to aid in the learning of the model. Flattening this axis effectively creates a new dataset of shape [48600, 96, 96, 1], doubling the size of the dataset for training, testing, and validation (if it is to be utilized).

**Part 1**

**Classification Network for Multi-class, Single-label Categorical Classification**

See classification.py

The purpose of this model is to determine the image label (category) from a given image.

Data pre-processing:

As the input values are in the range [0 – 255], the values need to be normalised to values in range [0 – 1] so that the TensorFlow and Keras framework initially small weights (and through extension weight update calculations) can have an effect that is non-negligible, allowing for the model to train.

Determining hyperparameters.

While the dataset is limited during testing for computational and temporal reasons, an increase in representative data will not harm a model’s generalisation. Variations of the model, in terms of number of layers, layer types, size of layers etc, are used to determine a suitable and accurate set of hyperparameters.

During testing, the following hyperparameters were methodically experimented with:

List index randomisation, number of validation data relative to training data, number of network layer, type, size, activation and other attributes of network layers, the introduction of data flattening, batch normalisation and weight decay, image augmentation – whitening, dimension shifting and image flipping, optimizer and loss function.

The purpose for this augmentation is to provide variation of epochs so the same image rarely is reused. This improves the generalisation of the model.

Tests were conducted with 800 data over 100 epochs. The baseline network consists of two layers each of Conv2D and MaxPooling2D, one dense sigmoid layer, and a dense SoftMax output layer. All layers are very small. Network evaluation is conducted with 100 test images.

Baseline accuracy was recorded at: 0.630.

|  |  |
| --- | --- |
| **Function** | **Test Accuracy : Difference** |
| Dropout : Rate = 0.5 | 0.680 : +8% |
| Batch Normalisation | 0.760 : +13% |
| Weight Decay : Rate = 0.1 | 0.650 : +2% |

|  |  |
| --- | --- |
| **Layer : Number of Layers** | **Test Accuracy : Difference** |
| Conv2D : 1 | 0.520 : -11% |
| Conv2D : 2 | +0% |
| Conv2D : 3 | 0.720 : +9% |
| **Conv2D : 4** | **0.750 : +12%** |
| Conv2D : 5  --------------------------------------------------------------- | 0.640 : +1%  --------------------------------------------------------------- |
| MaxPooling2D : 1 (Pool Size = 2) | 0.660 : +3% |
| MaxPooling2D : 2 | +0% |
| MaxPooling2D : 3 | 0.690 : +6% |
| **MaxPooling2D : 4** | **0.730 : +10%** |
| MaxPooling2D : 5  --------------------------------------------------------------- | 0.660 : +3%  --------------------------------------------------------------- |
| Dense : 1 | +0% |
| **Dense : 2** (128, 128) | **0.690 : + 6%** |
| Dense : 3 | 0.620 : -1% |
| Dense : 2 (128, 512) | 0.670 : +4% |

The introduction of data augmentation resulted in a model accuracy of 74% with the otherwise standard stated conditions, despite such a small data sample size. This increase of 11% shows a well worth while increase in model prediction accuracy. Due to this as well as the little amount of variation between images, it has been found that the effect of data augmentation, while beneficial, it is only minimally so (in combination with other generalisation methods).

Contrary to individual testing results, full scale application of certain regularisation techniques produced varying results. The purpose of reserving validation testing data is to measure the validity of a model, representing how the model will predict actual data – determining if the model is over fitting training data, or if the model is representative of the actual testing data.

Using weight decay regularization (L2) resulted in a testing accuracy score of only around 20% (equal to random guessing), while using batch normalisation resulted in a slight accuracy decrease of around 2-3% (both techniques were altered until they provided the greatest possible accuracy).

Chart, line chart

Description automatically generatedGraphical user interface, chart

Description automatically generatedEffect of batch normalisation (left) on validation accuracy appears to be beneficial purely for training data, perhaps it is the case that using the applied 10,000 data points is insufficient for batch normalisation to provide aid, but due to this negative effect, it has been elected that batch normalisation will be left out of this model. As can be seen in the second graph, a weight decay kernel optimizer appears detrimental to the model under the given circumstances as well.

As can be seen in the above graph, removing the application of batch normalisation and weight decay has substantially improved the validation accuracy consistency. The dropout layer, however, effectively reinforces generalisation through the random deactivation of a layer’s neurons, encouraging the model to not rely on the output of few neural nodes and pathways.

The activation function chosen for all layers was ReLU due to the non-linear nature of the desired model, as well as the function reliably returning consistently smaller error values. Sigmoid appeared to at times skew the error, while tanh appeared to do so however bilaterally.

Text

Description automatically generated After testing different architectures, methods, activation functions, parameters, and structure orders (have also been tested concurrently), the model architecture that will be used is as such:

Note that the parameters of the Dense ‘ReLU’ layers is different from the optimal showcased in testing, this is because in combination with other introduced model hyperparameters, this layout results in the best average results. Accuracy was used as the metric as it best reflects the relation to the desired result (100%).

The architecture of the model also took the computational power required into consideration. Unlike many architectures tested enroute to determining one suitable for the task, this model contains only around 500,000 trainable parameters.

Tests determining the optimization algorithm averages were conducted with small sample sizes of only 2500 as the effect of adding more training data works to decrease generalisation error. Therefore, the differences found in effectiveness found for this particular task will translate to a greater sample size.

The stochastic gradient descent algorithm, with respect to learning rates, has average validation accuracy as follows.

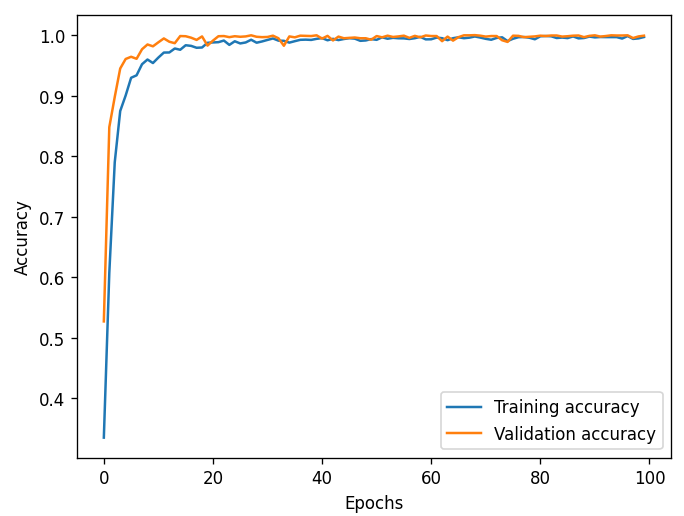
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Learning Rate** | 0.01 | 0.05 | 0.1 | 0.2 |
| **Accuracy** | ~20% | ~83.5% | ~85% | ~77% |

The Adagrad algorithm returned very similar if not, slightly improved values, with some decreasing the accuracy. The average difference fell within the range -1.5% to +4.5%. This shows an improvement but was however insignificant compared to other tested optimizers.

The RMSprop algorithm consistently resulted in validation accuracy of over 85% with an average accuracy of 88.9%. Comparing this to the Adam optimizer, which returned an average accuracy of 86.1%, it was found that the theoretically slower RMSprop optimization algorithm surprisingly performed better on average.

As classes data belong to are mutually exclusive, sparse categorical cross entropy is the loss function utilised for this section of the assignment.

A picture containing graphical user interface

Description automatically generatedFor epoch determining, 10,000 data points were passed through the training over 100 epochs.

Above are the Training and Validation accuracy and loss plotted against one another. Only minimal learning occurs after around 40 epochs. Due to call back functions, this epoch number only specifies the maximum number of data repetitions, the best performing model weights (on validation data) will be saved regardless of epoch. Therefore, training will be conducted until the validation loss begins to rise again from its lowest value. The implementation of early stopping could have mitigated this issue with a great enough range, however doing so wouldn’t allow for much variation.

Text

Description automatically generatedAfter training the model over the full unravelled training dataset 50 times, the best performing version of the model is saved, and achieves 88% accuracy on the unravelled test data, on the categorisation task.

This is a peculiar result as the validation accuracy achieved over the course of training was considerably, consistently higher than the test accuracy. After testing the model on a random permutation of image to label indices (and consulting with the course coordinator ensuring that this was allowable), a much stronger relationship between the validation and testing data accuracy could be seen. This hints towards an uneven distribution of category representation throughout the dataset with the supplied order of image indices – the validation data (retrieved from the training data) is ill representative of the testing data.



****

Shown above are the accuracy and loss values for training, validation, and then testing. The top set of values shown utilises the plain data as supplied in the training/testing dataset; the bottom set of values utilises completely randomised sets of training/testing data of exactly the same size as before.

As can be seen, despite a decreased validation and training accuracy, the testing accuracy is significantly increased.

Chart, histogram

Description automatically generatedTraining over 300 epochs shows that overtraining begins only around 150 epochs. Due to the best weights (relative to validation accuracy) being recorded and restored, overshooting the optimal as such allows for the small amount of randomness present in all models.

Text

Description automatically generated with low confidenceFinal training results in the loss and accuracy testing values below:

As is shown from the accuracy results, the data augmentation and generalisation techniques used have prevented the over fitting of the training data.

Text

Description automatically generatedThe best-found weights produce the training and testing accuracy as shown:

**Regression Network for Camera Elevation**

See regression.py

The purpose of this model is to predict the elevation from which a given image was taken.

For this model, the same, unravelled, randomised dataset, will be utilised for the same reasoning.

In theory, the modular pattern recognition capabilities of convolutional network layers have less value in such a prediction procuring model. For this among other reasoning, the final model will likely differ in structure to the previous model.

The loss function used for this network is mean square error (mse) due to the continuous regression nature of the network. Along with this, the metric used to determine network accuracy is mean absolute error, also due to the network’s regression nature.

The Adam optimization algorithm was chosen due to an apparent continuation of learning, for the longest out of the optimization algorithms. Along with this, while the range of final error may not extend as low as other algorithms, it was more consistently shown to provide an accurate model, with error rates on small scale tests ranging from 0.4 to 0.7. Implementation of which resulted in an error difference of -0.09 over the next best optimizer, RMSprop. Other algorithms consistently produced error rates exceeding RMSprop, regardless of learning rate.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Optimizer | Adam | RMSprop | Adagrad | SGD |
| Mean Error | **0.5500** | 0.5600 | 0.5970 | 0.6000 |

Testing has shown that implementing 4 basic convolutional layers results in the smallest recorded mean absolute error value at an average of 1.521 followed by 3 layers with a value of 1.598. Introducing 5 layers adds too many parameters resulting in slower learning and over training, reducing the accuracy metric further. Testing around the size of the convolutional layers is displayed below. Note that sticking to convention, all following layers are equal in size if not greater than the previous. As well as this, each layer utilises the previous layer size that allows for the lowest error regardless of best performing size on the previous layer – when the layer was considered to be the last (size carried over highlighted in bold). This table shows the mean absolute error of testing results on a model with layer quantity sizes depicted, with no dense layers outside of output. However, max pooling layers are placed after each convolutional layer (when added) for benefits discussed in the first section of this assignment.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Layer : Size | 16 | 32 | 64 | 128 | 256 |
| 1 | 1.4741 | **1.3966** | 1.4225 | 1.4558 | 1.4334 |
| 2 | /////////////// | 1.3313 | **1.3137** | 1.2828 | 1.3008 |
| 3 | /////////////// | /////////////// | 1.1655 | **1.1231** | 1.1480 |
| 4 | /////////////// | /////////////// | /////////////// | 1.1221 | **1.0858** |

Many of the data and other precursor functions from the categorization network have been carried over as the optimization and generalization supplied by these have been shown through testing to benefit the regression network as well.

Previous testing has shown that using 2 densely connected layers results in the best learning, the testing of these layer size values is depicted below, with the layer size shown across the x axis (top), and the layer number across the y axis (side). The table shows the mean absolute error returned by a model implementing layers with sizes as shown. Similar to the above table, the best acting layer (in terms of following layer performance) is utilised for the following layer (and highlighted in bold).

|  |  |  |  |
| --- | --- | --- | --- |
| Layer : Size | 128 | 256 | 512 |
| 1 | **1.2801** | 1.2997 | 1.3550 |
| 2 | 1.1889 | 1.1300 | **1.1298** |

Note testing was conducted with otherwise basic structure. The testing shows that layers of sizes 128 and 512 work in collaboration to provide the best accuracy. The introduction of more layers resulted in severe overfitting and therefore reduced accuracy.

Testing with batch size yielded the following results:

|  |  |  |  |
| --- | --- | --- | --- |
| Batch Size | 1 (Default) | 16 | 32 |
| Mean Test Error | 1.0800 | **1.0486** | 1.1119 |

All testing conducted on this network was over limited data to initially find patterns, before expanding the data utilized to determine validity.

Data augmentation parameters used include whitening, width and height shift, and horizontal flipping. Individual application of any of the parameters resulted in little benefit, but all used together resulted in well worth while data generalization. Increasing the parameter values any further resulted in exaggerated data that was of no benefit, while any less allowed for over training of repeated data. The application of data augmentation, with values as below, results in a mean absolute error decrease of on average ~0.2400, corresponding to an error reduction of ~1.2 degrees.

Displayed below is the effect individual augmentation settings have on the absolute error of the model.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Augmentation: | Whitening | Width Shift | Height Shift | Horizontal Flip |
| Error Change: | -0.0840 | -0.0355 | -0.0811 | -0.0402 |

In determining the minimum number of epochs to run, the model with all the above adaptations is run over 300 epochs to determine where overfitting begins to occur. Due to the excess of data, it was found that this model actually continued training and over training as such did not occur until much later, due to this, the model is only trained until the error change becomes stagnant.

Other generalization methods (batch normalization and l2 normalization) were found to have similar effects as shown in section one of this report and so have not been repeated here. Meanwhile, once again dropout was found to decrease validation error by an average of 0.026.

Cropping of the images was experimented with and extremely consistently increased the error of the model. The more the image was cropped, the more profound the error became. For example, reducing the images from size 96 x 96 to size 64 x 64, while strongly decreasing model training time, increased the error by an average of 0.15 to 0.2, corresponding to between 0.75 and 1 degree.

The resulting model from the testing shown above is as below:

This final model produced during testing an average error of +/- 0.3335, or around 1.6675 degrees.

Interestingly, as the majority of tests were conducted on a relatively smaller sample of the dataset, the model continued training later than was suggested from testing, this indicated a larger model may be able to better conduct the regression on this task. To test this theory, a model with 7 convolutional layers of sizes 64 (x3), 128 (x2), and 512 (x2), along with 3 densely connected layers of sizes 512, 128 and 512 respectively, was shown to provide a testing mean absolute accuracy of 0.2555. Layers were added until generalisation begun making the model unable to ever determine an accurate answer, and the sizes of which were fine tuned afterwards until the error bottomed out. The dense layers followed a similar process, except changing the order of which resulted in the model reaching a smaller minimum error, at a faster rate (In testing, validation mae decreased by as much as 0.05 with the application of this dense format).

Graphical user interface, application

Description automatically generated

This model also shows the continuous decrease of error despite the reuse of data.

While the initially determined hyperparameters certainly reach a satisfactory accuracy, it is worth noting that it appears to be possible to increase the accuracy to an arbitrary degree with deeper, more complex networks, trading off computational efficiency for accuracy. However, a simpler model has more room for error, and therefore has better generalization.

**Part 2:**

**Diffusion**

**\*\*\*Please Note\*\*\*** : Multiple diffusion code files exist to display testing conducted, most important files are highlighted in bold, files contain as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File Name** | **Diffusion.py** | Diffusion2.py | Diffusion3.py | **Diffusion4.py** |
| **Content** | GAN | Unet Testing | Noise Testing | Final Diffusion |

See Diffusion.py

The goal for this section of the assignment is to create a model or set of models, which when provided with random noise can procure an image reminiscent of those from the smallnorb dataset.

One way a network like this would work is to have one model that adjusts random noise input values to fit certain parameters, this model is called the generator model, the output of this model is then passed into another model, the discriminator model, which is trained to recognise if an image is a part of the original dataset, the certainty of this second model (either positive of negative) is then used to produce a learning parameter (loss value) used to train the initial model. These models form a UNet, a neural network that starts and ends with large data shape, but bilaterally compress data representations with corresponding size data filters. This form of algorithm is called a generative adversarial network.

It is important to note that this section of the assignment is heavily inspired by a tutorial article on deep convolutional generative adversarial networks by the TensorFlow website, available at: <https://www.tensorflow.org/tutorials/generative/dcgan>. It is therefore of very little surprise that hyperparameters chosen for the purpose of this report differ little to those in the article, due to the reputability of the articles’ network creators.

While utilising an autoencoder would be effective in creating a differentiator model to return a learning parameter for an image generator model, I found that it may be more practical to train both model types in tandem.

The loss for both of the models was calculated though binary cross entropy as both models’ accuracy is determined through output similarity to desired category: original image or fake image.

Unfortunately, training this network as above required both models to train simultaneously meaning the loss value recorded at any epoch value with any hyperparameter set will likely have a similar value. The discriminator section of an autoencoder can however address this issue through providing a non-biased value predicting the likelihood of an image originating from the actual dataset.

For optimizer function, due to the unknown and potentially changing parameters encoded into either model, the Adam algorithm was used, and was found to be most consistent when initialised with a small learning rate.

For testing, a modified version of regression network portrayed above was used as a loss evaluating autoencoder.

The training images had their pixel values normalised to values between -1 and 1 so the image has the same range of values as a normal distribution.

The below table shows the loss of the generator model with batch sizes (y axis) and noise input quantities (x axis).

|  |  |  |  |
| --- | --- | --- | --- |
| **Batch Size : Noise Size** | 100 | 500 | 1000 |
| 64 | 1.8512 | 1.9153 | 1.7000 |
| 128 | **1.2520** | 1.6830 | 1.8250 |
| 256 | 1.4728 | 1.7700 | 2.1030 |

Below the depth of the UNet (or rather number of convolutional layers per model) is shown against the loss of the generator model.

|  |  |
| --- | --- |
| 2 layers | 1.4100 |
| 3 layers | **1.2180** |
| 4 layers | 1.7056 |

The above table shows that having 4 or more layers in a model, while increasing the model capacity, also increased the complexity, and an output of similar value can only be produced much later on in training. Meanwhile, less than 3 layers showed an inability to encode sufficient information required to portray a smallnorb image.

It was found that leakyReLU and batch normalization layers provided the greatest learning of the generator model, while leakyReLU and dropout layers provided the best for the discriminator model.

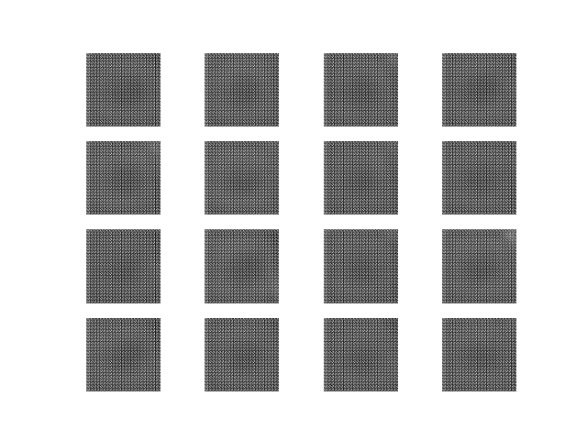
Batch normalisation served to average values, while leaky relu layers then worked to normalise value distribution. The effect of introducing these layers together reduced the loss by an average of between 0.2 and 0.6. Individually the layers would be inconsistent on whether they increase or decrease the loss.

Following this, the use of leaky relu and dropout layers allowed equally strengthened connections between layers, resulting in a more consistent output and lower loss. Increasing the dropout occurrence chance over 0.3 showed to decrease the range of loss by an average of between 0.2 and 0.3, while setting it just below 0.3 increased the range, but lowered the average loss decrease to between 0.1 and 0.4. Set at 0.3, dropout displayed the most consistent, highest loss decrease at an average of 0.28. Replacing these relu layers was found to more often than not produce a dead neuron disallowing back propagation through the passing of an empty output.

The number of epochs was chosen to be only 50 as after which diffusion slowed drastically.

Below are pictured the same 16 images after 1, 25, and 50 epochs. The loss at each epoch was ~5.120, ~2.200, and ~1.2180. The progression between the images is evident, however, the rate of change, along with the rate of change of the loss, decreases over time .

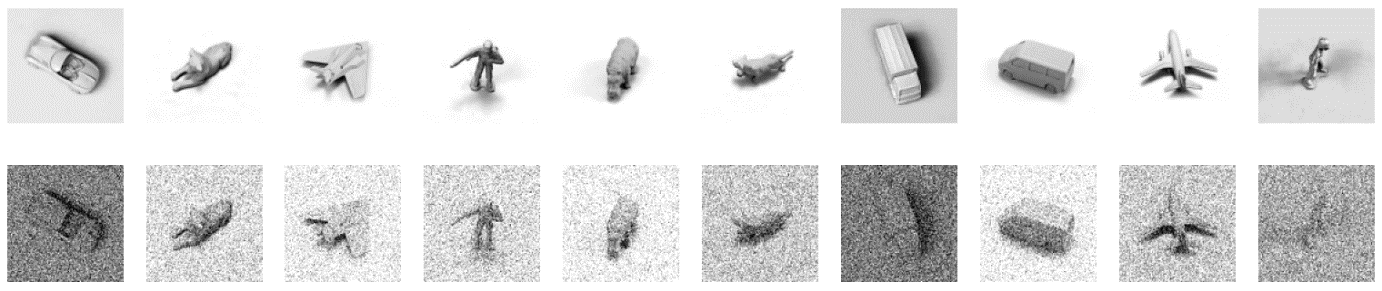
A picture containing white

Description automatically generatedA close up of a person's face

Description automatically generated with low confidence

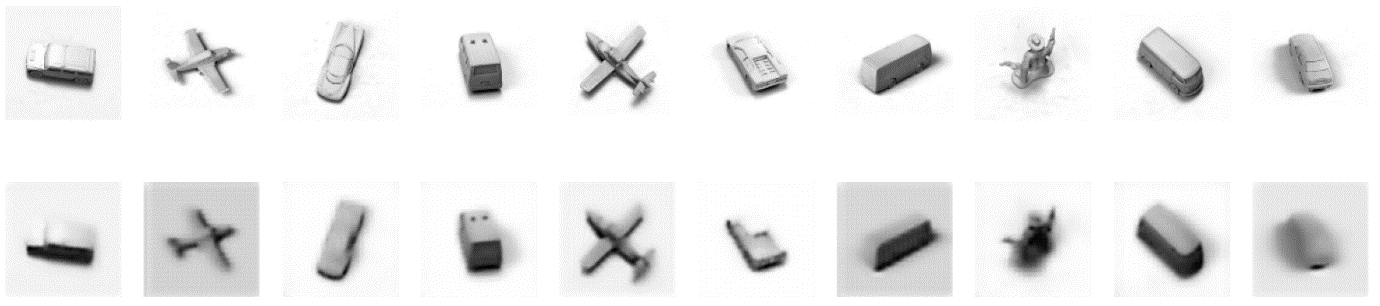
Attempt 2 at diffusion: - see file diffusion2.py

After reading up on autoencoders and their potential for diffusion application (primarily here: <https://pyimagesearch.com/2022/02/21/u-net-image-segmentation-in-keras/>), it was determined that there was another, potentially better method of noise reduction.

In this method, an autoencoder is created but trained with inputs of noise and targets of an actual image.

The above image shows a comparison between the original image and a slightly noisy version of the image.

The purpose of first testing the autoencoder with noise reduction is to ensure functionality.

The above image shows the results of passing the previously shown noisy images through the model, proving its functionality for encoding and decoding data correctly.

The training data is passed through the network with multiple different levels of noise, the testing of which is shown below with average loss via evaluation on a pure noise input.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **# Levels** | 1 | 2 | 3 | 4 | 5 |
| **Loss** | 0.89 | 0.76 | 0.54 | **0.49** | 0.56 |

The below table depicts results showing the amount of noise at each level (of 4 levels as determined above), that allows for the loss on average to be at its minimum.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Level** | 1 | 2 | 3 | 4 |
| **Noise Value** | 0.1 | 0.2 | 0.5 | 1.0 |

In deciding the number of layers of the model, models with 2, 4 and 6 (UNet) layers were compared. From this testing, while it was found the loss remained similar, the time required to train the models certainly did, because of this, a model layout composing of only 2 layers (either side of the UNet) was chosen.

Without a batch size set (i.e. batch size 1), the model was prone to overfitting and being biased towards particular outputs, the found batch size that reduced this while retaining sufficient representation quantity was 128 samples per batch.

The kernel size 3 x 3 was chosen as it would preserve the most data allowing for more thorough representational understanding.

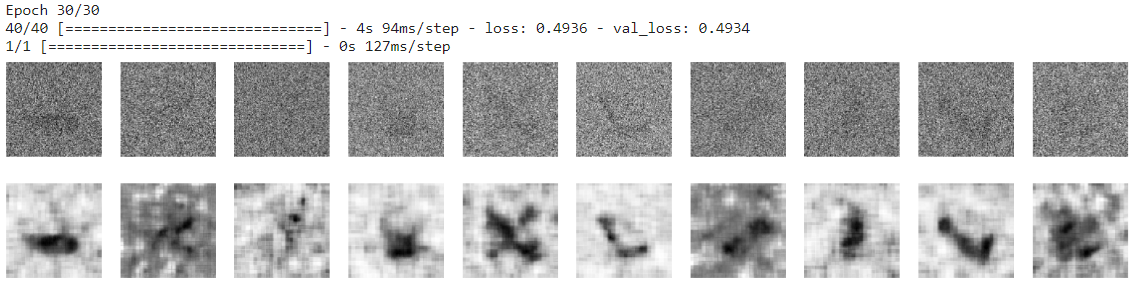
Below is an example of the network part way through training with 80% noise applied to inputs and the determined outputs.

Qr code

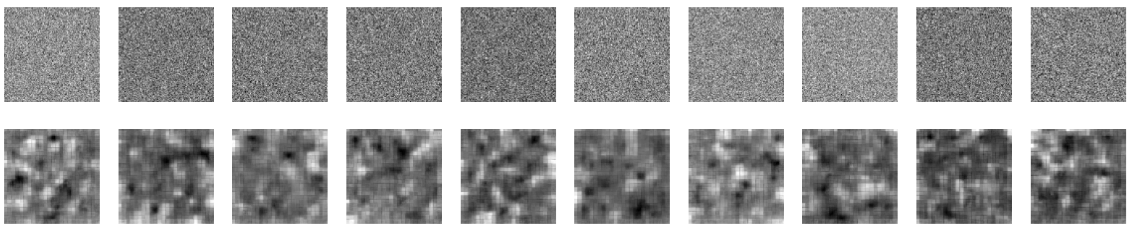
Description automatically generated

Over training, after only the first few epochs of each section, the loss begins to plateau while improvements are still being made. For this reason, 30 epochs per section have been used for the purpose of training to ensure thorough encoding of representations. With four separate sections representing varying levels of noise and 30 epochs of training at each level, the model performs 120 networks.

Adding skip connections to the network had no effect on the loss or visual standing of the outcome. As well as this, a sigmoid activation function on the output layer was required to retain a value between 0 and 1. Changing the activation function from ReLU for the hidden layers had no effect other than increasing the loss and decreasing the amount of coherence offered by the model.



After training, the model is able to diffuse the noise surprisingly well with a random normally distributed area of noise coating an original data image with an average loss value of only 0.49. However, when the trained model is then passed a random non data masking array, the results are much less coherent (below).



Unfortunately, this model appears to produce poor results when given random input with zero rhyme or reason, even simply having the background image masked completely by noise allows for much better results with evidently some amount of the image making it through the noise update.

Attempt 3

After this failed attempt, another model (found under diffusion3.py) was constructed.

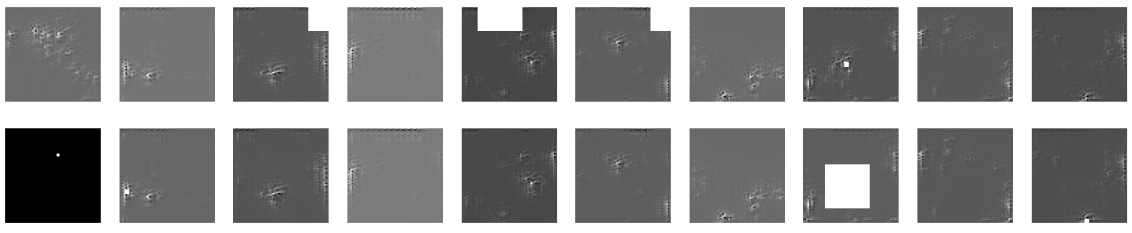
A similar model was tried that functioned by generating two datasets, one representing data to be input into the network: sequentially added noise per noise layer creating a test set of size noise layers times the original arrays size. The other dataset represented the output data: the image before the noise was added – effectively the previous value of the input, emulating time step minus one. With these datasets, the model would attempt to map the noisy input to the slightly less noisy output image. In theory, with enough representations, these denoising steps appear relatively small and create a trivial task for the network, removing noise to leave behind a rough average of the data set images, creating a new image.

In creation of the datasets inputs and outputs, multiple noising methods were tested including: applying increasing levels of a gaussian noise filter over layers, applying strength variations of a single gaussian filter per image, and applying layer-wise noise addition with noise remaining constant between images for a given layer of noise strength.

In the diffusion3.py file diffusion implementation attempt, there is code for implementing all 3 of these approaches. The model defined also utilises a unet incorporating skip connections as was shown to be beneficial during training.

The model also was tested using different size unets, output activation functions, layer types, generalisation features, optimizers, loss functions, noise splits, batch sizes, and model types. Unfortunately, due to limited time, results of which are not displayed here, features listed above that performed the greatest were kept and can be found in the implementation.

The application of the method applying consistent strength gaussian noise differing between layer and image was found to be the most consistent, while still not producing the desired results. While the loss was able to be reduced to an incredibly minimal degree at 1.54e-04, utilising the model to predict noise transformations over the decided number of layers served to only produce incoherence.



The images above show the result of training a model with 1000 layers of light noise then providing said model with random noise input and passing the result through the model 1000 times for emulation of the noise layers from training. It was also found that less layers had little effect, noise was more evident with less layers, but with more layers the images did not begin to resemble actual data any more than with lesser layers.

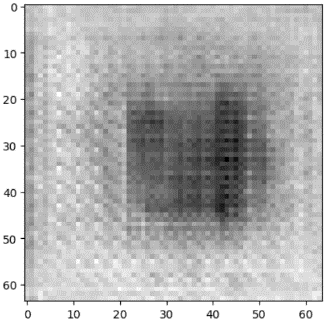
**Functioning Diffusion:** (See file diffusion4.py)

Adding noise was also tested via altering the standard deviation of gaussian noise, with sigma (Σ) denoting the filter’s standard deviation, and x the level at which the layer is. The equation for this calculation is as below:

Σ = √x / (2 \* √(2 \* log­10))

A picture containing chart

Description automatically generatedA picture containing chart

Description automatically generatedThis method of adding noise to an image as it was being passed through the model is shown in file diffusion4.py as a function. Again, however, this method was unable to produce very good results as is shown below:

Pictured are the results of training a model with few (5), an intermediate amount (15) and many (100+) layers of noise implemented. During testing of this method, it was found that the loss of the model would consistently bottom out around 0.0075, regardless of the batch utilised. As can be seen in the central image in the figure above, around 15 layers used during training results in the best image relative to the smallnorb dataset. Determining the number of epochs over which to train consisted of watching the loss of the function and find where it stopped decreasing, this point is often where overtraining begins to occur, in this case, around 20 epochs.

The model architecture was found to have a large effect on the loss of the model. Using a linear model, the loss reached its lowest at around 0.048, while with the same otherwise hyperparameters, using a unet implementation reached a loss low of 0.0075. Adding too many layers resulted in excess training required to learn the introduced unnecessary parameters, while having to few model layers resulted in the model not being able to properly represent the smallnorb dataset. Following this understanding, the number of layers was increased until the loss began to reach a barrier, only later and later into training, this occurred after the implementation of 2 downscaling blocks, 1 bottleneck block, and 2 upscaling blocks. The choosing of numbers of filters for each of these blocks was influenced by both testing and common convention. The number of filters per block found to result in the smallest loss values were in order: 32, 64, 128, 64, and 32.

While blocks are reminiscent of single layers, they are made up of many layers. The best results were found with downscaling blocks incorporating 2 same size convolutional layers alongside a max pooling layer, the bottleneck block performed best when made up of 2 same size convolutional layers, and the upscaling blocks when made up of 2 same size convolutional layers following a transposing convolutional layer.

Due to the regression nature of the desired model, the loss function of the model was chosen as mean squared error, with the optimizer function as adam due to the large amount of variables influencing loss gradients.

The size of the images was reduced for the sake of retaining computational efficiency.

Due to time restrictions, this was regretfully all the testing that was able to be conducted on this model. The trained network is loaded for regression4.py, with the file containing the other tested noise application methods discussed previously.

It is worth noting that it would have been trivial to copy and paste a diffusion model found online and adapt it to the smallnorb dataset, however, due to lack of understanding and familiarity with the probability equations, it is assumed that the assigned task was to produce a model from scratch without utilising some non-aforementioned calculations.

**Final note:**

Apologies for the confusing layout of files and data supplied, testing required many different layouts and would have been a mess of a single file. Files have been saved in such a way that running will not begin training automatically but rather showcase the model and its functionality. Training is run with a initially deactivated function in each of the models.

**References:**

Generative Adversarial Networks:

[**https://www.tensorflow.org/tutorials/generative/dcgan**](https://www.tensorflow.org/tutorials/generative/dcgan)

UNets in image segmentation:

[**https://pyimagesearch.com/2022/02/21/u-net-image-segmentation-in-keras/**](https://pyimagesearch.com/2022/02/21/u-net-image-segmentation-in-keras/)