Analyzing the Effectiveness of CNNs and GANs in Supersampling MRI Scans

Alan Hencey
Tickle College of Engineering
University of Tennessee
Knoxville, Tennessee
ahencey@vols.utk.edu

Tanner Fry
Tickle College of Engineering
University of Tennessee
Knoxville, Tennessee
tfry2@vols.utk.edu

Racheal Dylewski
Tickle College of Engineering
University of Tennessee
Knoxville, Tennessee
rdylewsk@vols.utk.edu

***Abstract*—Advances in areas of image supersampling and image super-resolution have yielded high performing results for many applications. In this paper, we attempt to compare different supersampling deep learning models to MRI images of brain tumors in order to enlarge and greatly improve the resolution of the initial image scan. We do this by presenting two different deep learning models, including a convolutional neural network (CNN) and a generative adversarial network (GAN) for image supersampling. We evaluate the reconstructive performance of the two models and test the effect of different hyper-parameters on the accuracy of the model.**

***Index Terms*—supersampling, CNN, GAN**

I. Introduction

Image supersampling techniques have been explored in many areas to recover a larger in size high-resolution (HR) image from a low-resolution (LR) image. Most popularly, this field has been studied for applications in areas such as video surveillance, graphical rendering, and medical imaging. In this study, we recognize the need for image up-scaling in the medical field for tumor recognition. Many systems have attempted to define cancerous tumors based on tissue analysis methods. However, these systems can be limited by the quality of MRI imaging due to low-resolution and noisy output. We attempt to take images of MRI brain tumor scans, upscale the images, and increase the resolution.

For general supersampling, the LR image data is generally downscaled from the original HR image. From a high-resolution image, a degradation function can be applied to clearly and easily produce a LR image. The inverse of this poses an ill-posed problem, as the degradation function used to produce a LR image is not known in the training processes. In this study, we compare the supersampling output from two models in order to compare the effectiveness of the proposed solutions. This includes a convolutional neural network (CNN) tested on a variety of hyper-parameters, including a comparison of loss functions. The loss configurations include the traditionally used mean-squared error, structural similarity index, combined loss accounting for both the mean-squared error and the structural similarity index, and a sequential combination of the two. In addition to this, generative adversarial networks (GAN) have become increasingly popular in the realm of image generation and reconstruction. We compare the performance of a specialized GAN on the desired task in order to evaluate the best performing model.

II. Previous Work

Convolutional Super-Resolution

Prediction-based methods were among some of the early methods used to tackle image-resolution problems. These approaches proved to be a fast, yet oversimplified approach to the problem [7]. To combat the lack of fine detail and inability to remove defects of traditional algorithms, convolutions neural networks have seen success in this task domain. In the past, Convolutional Neural Networks (CNNs) have performed well in medical imaging tasks, such as disease classification and tumor segmentation [8]. Dong et al. proposed a Super-Resolution Convolutional Neural Network (SRCNN) that learns an end-to-end mapping between low- and high-resolution images. SRCNN consists of upscaling low-resolution (LR) images to desired high-resolution (HR) image size, extraction of a set of feature maps from up-scaled images, a non-linear mapping of LR to HR patches, followed by reconstruction of the final image [9]. The model tests 2-4 layered CNN and their performance on super-resolution tasks, proving that a CNN can be used to learn mapping from LR to HR images in an end-to-end fashion.

SRCNN presented by Dong et al. can be limited in that it relied on context of small image regions, training converges slowly, and works only for a single scale. Kim et al. found that increasing network depth showed significant improvement to accuracy with by presenting a 20 layer model that only learned residuals and used very high learning rates [10]. Deep networks can be more difficult to train, however, they also have the potential to greatly improve accuracy by modeling mappings of very high complexity. Another progressive adaptation of SRCNN has been to adaptively increase resolution by implementing convolution with pooling as a downsampling approach, followed by deconvolution as the upsampling. LapSRN proposed a downsampling approach using a Laplacian pyramid; transposed convolutions are used for upsampling to the finer level [11]. DBPN proposed an iterative up and downsampling approach to exploit high resolution features [12].

SSNet and SSNet-M proposed supersampling methods to predict the output with a larger size than the target HR image, and then adaptively down-sampled the over-sampled output image through convolutions. Introduced though this technique is a depth wise separable convolution, recursive depth wise convolution, and a super-sampling block [13].

Loss Functions

In the past, most CNNs use a pixel-wise Mean Square Error (MSE) as the loss function to optimize the network. MSE is interpreted as the Maximum Likelihood Estimator of the mean of a conditional Gaussian distribution. Zhao et al. found that in the context of superresolution, denoising, and JPEG deblocking, a network trained with the l_1 cost function can outperform a network trained with l_2 loss function even in the context of l_2 loss. In addition, it is concluded that MSE loss does not guarantee better performance in terms of Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) [14]. Wang et al. found that SSIM has had better performance in predicting perceived images over MSE [15].

Johnson et al. found that per-pixel losses, an approach used by Dong et al, do not capture perceptual differences between the output image and ground-truth. Recent works show that high-quality images can be generated using perceptual loss functions based on high-level image feature representations extracted from pre-trained CNN instead of pixel-by-pixel loss [16].

Xu et al. proposed a method of super-resolution using a combined three losses with assigned weights for optimization that outperformed existing models on benchmark datasets [17].

GAN Super-Resolution

Despite breakthroughs in SRCNN as a fast way to product high-resolution images, there remains the problem of achieving finer details, especially when upscaling to much larger images. In the first proposed super-resolution generative adversarial network (SRGAN) [18], a perceptual loss function that consisted of adversarial loss and content loss motivated by pixel-wise similarity was proposed. The SRGAN deployed a deep residual network with skip-connections that augments the content loss function with an adversarial loss function.

Image resolution has presented especially challenging in recovering texture details. Success has been found in using several different loss functions, primarily where one can differentiate the supersampling, which is done by applying a smaller upscaling factor and combining an anti-alias approach to GANs [18].

III. Technical Approach

Problem

The goal of this project is to develop and compare CNN and GAN models with various setups for image supersampling of MRI scans. Furthermore, we plan to develop a model that can upscale an image for 4x the detail effectively. For both networks, we plan to test various hyper-parameters related to both the network architecture and to the training and evaluation of model performance. Additionally, for the CNN we plan to investigate different loss function setups that are commonly found for image supersampling, and a method not as commonly found from our research of existing information related to image supersampling in deep learning.

CNN

The first model analyzed in this project is the convolutional neural network (CNN). A CNN is a neural network architecture commonly used in deep learning for different forms of image processing, and is loosely inspired by the organization of neurons on the human visual cortex [6], which allows for the CNN to be efficient at pattern detection in images. CNNs typically take an input image and try to reduce the size by performing feature extraction and pooling to keep track of the most important characteristics of the input images for the network to perform well at the task at hand. CNNs typically include some combination of the following computational layers: convolutional layer, pooling layer, and fully connected layers. Non-computational layers such as flatten or reshape layers can also be used.

Convolutional layers are used to extract features from the incoming data using filters called kernels. These extracted features are the most important attribute of the CNN when it comes to supersampling images, because the network will learn how attributes from a downsampled image transform to a higher pixel area to produce the supersampled images. The hyper-parameters tested in this project related to convolutional layers include: number of features to extract, kernel sizes, activation functions, and padding. For this project, only 2D convolutional layers are used in the CNN.

Pooling layers are used to reduce the dimensions of convolutional layer outputs, which help speed up the training of the network by only keep the most influential features extracted from the convolutional layer. The hyper-parameters tested in this projected related to pooling layers include: pool size, and padding. For this project, only 2D max pooling layers were used in the CNN.

Fully connected layers are the dense layers of a neural network. Each input to the layer connects to each neuron of the layer, and each input at each neuron has a weight associated to it. Also, each fully connected layer will include a bias in this project. The number of neurons, and activation function of the fully connected layer are the only hyper-parameters tested in this project.

Flatten and reshape layers are used for transforming the structure of data. Flatten layers can be used to transformed multi-dimensional matrices to a single dimension by appending the data across the dimensions into one array. Reshape layers can be used in a similar fashion except they can be used to transform from multi-dimensional matrices to multi-dimensional matrices where dimensions can be in a different ordering.

For monitoring the learning of CNNs, different loss functions can be used based on the overall task of the network. For image upscaling, there are various loss functions that can be used that all have their own learning problems they attempt to solve. In this project, four different loss setups will be analyzed for the CNN: mean squared error (MSE), structural similarity index measure (SSIM), a weighted sum of both SSIM and MSE (referred to as ”SSIM and MSE” in this project), and lastly a more unique approach where the network will first train with SSIM until convergence then train with MSE until convergence (refereed to as ”SSIM then MSE” in this project). MSE in image supersampling can be used to represent a loss that is looking at the per-pixel loss because it looks at each pixel and is taking the difference between the true and predicted pixel values. SSIM was first proposed in a 2004 IEEE paper, *Image Quality Assessment: From Error Visibility to Structural Similarity* [5]. The idea behind SSIM is to provide a learning method more aligned with the human visual perceptions system that is highly capable of identifying structural information [5]. To do this, SSIM uses three features to analyze the difference between two images: luminance, contrast, and structure [5]. The resulting values from SSIM fall in the range of -1 and 1, where 1 indicates very similar images, and -1 represents very different images.

The goal of analyzing the different losses described is to see which loss setup performs better between pixel-wise comparison, structural analysis, and a combination of the two losses. The weighted combination of the losses is not a new concept, and can be found in articles on super sampling [21], where the idea is to use a custom loss function that can account for various image attributes by weighting different losses focused on different goals and summing them all together for a single value. In this case, the SSIM and MSE loss function is used to try and account for both structural and pixel-wise learning in the CNN simultaneously. From seeing weighted sums of different loss functions, an idea emerged: can a CNN also be trained on different loss functions separately to account for the different goals of each loss? This allows for sequential loss training rather than simultaneous loss training. The idea is that first, the network will train on learning attributes of the structure, using SSIM, for supersampling the images, then once the structural learning has finished, the network will switch over to a pixel-wise comparison loss function, MSE in this case, to try and get the coloring/shading fine-tuned.

GAN

The second model type analyzed in this project is the generative adversarial network (GAN). GANs use two separate networks, one to generate the expected output from a latent space, and one to classify whether or not an input comes from the original dataset or has been generated by the first network. These networks are called the generator and the discriminator respectively and each match the structure of convolutional neural networks described previously. In our project, the generator is a convolutional neural network whose input is the downscaled 25 x 25 MRI images and whose output is a 50 x 50 image that attempts to mimic the original 50 x 50 image. The discriminator is also a convolutional neural network, and its input consists of a combination of the original 50 x 50 images and the generated 50 x 50 images. Its output is a single floating point number which represents the probability that the given input was ”valid”, meaning it comes from the original dataset.

The process of training a GAN is different than the process used to train a more traditional convolutional neural network. The two networks are trained at the same time, where the generator is constantly learning to maximize the probability that the discriminator makes a mistake and the discriminator is constantly learning to tell the difference between generated images and images from the dataset [19]. The models are trained at the same time because if one was trained extensively before the other began training, then the other model might never catch up to the one that was trained first, so training at the same time attempts to keep the models in balance. The training of the generator and the discriminator works in a series of steps. First, the generator is used to generate super-resolution images from the 25 x 25 downscaled images. These generate images are then combined with an equal number of the original 50 x 50 images and fed into the discriminator, which predicts the validity of each of the samples it was given. The discriminator uses the binary cross-entropy loss function to calculate its error and update its weights. Then, to train the generator, just the generated images are fed into the discriminator with the ground truth values set to ”valid”, because that is the goal of the generator, and the error is once again calculated using the binary cross-entropy loss function. This error is then fed into the generator model, and used to update its weights. This process repeats, training each model concurrently one batch at a time.

The idea of using a GAN to generate super-resolution images is based on the idea that the discriminator can be trained to act as a better measure of loss for a generated image than a traditional loss function could, in turn causing the generator to become more accurate with its predictions. This specific use-case for a GAN was proposed in 2017 and is called SRGAN, super-resolution generative adversarial network [18]. In this project, we compare the results of our SRGAN to the results obtained from using the SRCNN with the various loss functions described above.

IV. Dataset and Implementation

Dataset

The dataset used in this project comes from a Kaggle project that was designed for labeling brain tumor images [1]. This dataset contains 3064 total MRI scans from 233 patients containing three different brain tumor types: meningioma (708 instances), glioma (1426 instances), and pituitary (930 instances) [1]. The original data from this project includes the image data, class labels, patient IDs, border of tumor location, and masks for the tumors in each image. From this data, croppings of the tumors based on the border and masking information are extracted and used as the data for this project.

When extracting the croppings of the tumor in each image, the croppings are extracted in two different scales: 25 x 25 pixels, and 50 x 50 pixels. The 25 x 25 croppings are treated as the x data and act as the downscaled images the models take as an input, while the 50 x 50 croppings are treated as the y data and act as the supersampled images the models try to upscale the input to. The 2D pixel arrays for all croppings were saved using matplotlib, and using the ”bone” color mapping. Matplotlib saves image data in four channels: R, G, B, and A. A represents Alpha, or transparency in this case [3]. Additionally, saving with the ”bone” color mapping keeps the pixel values normalized between 0 and 1.

For splitting the dataset, an 80/10/10 split is used where 80% of the data is used for training, 10% is used for validation, and 10% is used for testing. Instead of splitting the data randomly amongst the training/validation/testing sets, a more fair approach was taken where an equal percentage of each class of tumor is split into the training/validation/testing sets. Since there are three different classes of tumors, 80% of each class will go to the training, and 10% of each class will go into the validation/testing sets. This was done to ensure that the models are seeing an equal distribution of the different tumor types while training in case any features being extracted for upscaling are unique to the respective tumor classes.

CNN

For the CNN, various network hyper-parameters were tested including: layer types and their respective hyper-parameters, optimizers, learning rates, decays, momentum (if applicable), batch sizes, losses described in section III, and early stopping mechanisms.

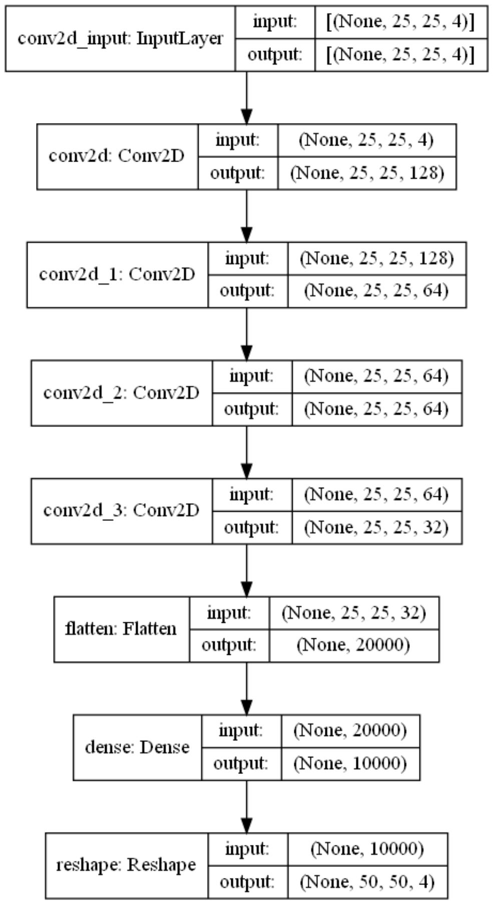

conv2d_input: InputLayer
input: [(None, 25, 25, 4)]
output: [(None, 25, 25, 4)]
conv2d: Conv2D
input: (None, 25, 25, 4)
output: (None, 25, 25, 128)
conv2d_1: Conv2D
input: (None, 25, 25, 128)
output: (None, 25, 25, 64)
conv2d_2: Conv2D
input: (None, 25, 25, 64)
output: (None, 25, 25, 64)
conv2d_3: Conv2D
input: (None, 25, 25, 64)
output: (None, 25, 25, 32)
flatten: Flatten
input: (None, 25, 25, 32)
output: (None, 20000)
dense: Dense
input: (None, 20000)
output: (None, 10000)
reshape: Reshape
input: (None, 10000)
output: (None, 50, 50, 4)

Fig. 1. CNN architecture.

Starting with the network architecture, the architecture hyper-parameters tested include: layer types, number of filters, kernel sizes, number of neurons in dense layers, activation functions, pool sizes, and padding. We found that not including any max pooling layers helped preserve the feature extractions needed for image upscaling, thus none were used in the final implementation. The final implementation of the CNN includes 4 convolutional layers, a flatten layer, a fully connected layer, and a reshape layer. Figure 1 showcases the sequential design and dimensionality of each layer of the CNN implementation. The first convolutional layer include 128 features, a kernel size of 10, relu activation, and padding set to same. The second convolutional layer includes 64 features, a kernel size of 5, relu activation, and padding set to same. The third convolutional layer includes 64 features, a kernel size of 5, relu activation, and padding set to same. The fourth convolutional layer includes 32 features, a kernel size of 3, relu activation, and padding set to same. Then there is a flatten layer to flatten the output from the last convolutional layer to feed into the fully connected layer. The fully connected layer has 10,000 neurons, and uses the sigmoid activation function. Then the 10,000 outputs from the fully connected layer is reshaped into a (50,50,4) matrix to represent the supersampled image output.

Moving on to the learning and evaluation hyper-parameters tested, the optimizers tested include stochastic gradient decent (SGD) and adam. Learning rates tested ranged from 0.1 to 0.000001. Decay values tested ranged from 0.01 to 0.0000001. Momentum values tested ranged from 0 to 0.9. Batch sizes tested ranged from 8 to 128. Loss functions tested include the loss functions described in section III which include: MSE, SSIM, SSIM and MSE, and SSIM then MSE. Lastly, early stopping callbacks were tested with different patient values ranging from 3 to 5. Patient values in Keras callback functions for early stopping represent the number of epochs with no improvement relative to the validation loss, after which training will be stopped [4]. The combination of network hyper-parameters tested that provided the best results are the following: adam optimizer, 0.00001 learning rate, 0.000001 decay, momentum not applicable, 32 batch size, SSIM then MSE loss, and a patience of 5 for early stopping. The evaluation to conclude that the SSIM then MSE loss provided the best results is showcased in V. For the SSIM and MSE combined loss, the MSE gets weighted at 40% of the loss, and SSIM gets weighted at 60% of the loss.

The time to train the CNN differs based on the loss functions used due to them each causing the network to converge at different points. For the MSE network, the network finished training in 39 minutes. The SSIM network finished training in 36 minutes. The SSIM and MSE network finished training in 28.5 minutes. Lastly, the SSIM then MSE network finished training in 52.5 minutes.

GAN

The GAN consists of two networks, the generator and the discriminator. The final implementation of the generator consists of 16 layers, but only 5 of the layers contained trainable parameters. The first layer is a convolutional layer with 64 features, a kernel size of 3, a stride of 1, and padding same. The second layer is a batch normalization layer with momentum of 0.9. The third layer simply applies the relu activation function to the output of the previous layer. The fourth layer is a dropout layer that drops 40% of the input units from the previous layer. The fifth layer is another convolutional layer with 32 features, a kernel size of 5, a stride of 1, and padding same. The sixth layer is a batch normalization layer with a momentum of 0.9. The seventh layer is a relu activation layer. The eighth layer is an upsampling layer. The ninth layer is a convolutional layer with 16 features, a kernel size of 5, a stride of 1, and padding same. The tenth layer is a batch normalization layer with momentum of 0.9. The eleventh layer is a relu activation layer. The twelfth layer is a convolutional layer with 8 features, a kernel size of 5, a stride of 1, and padding same. The thirteenth layer is a batch normalization layer with a momentum of 0.9. The fourteenth layer is a relu activation layer. The fifteenth layer is a convolutional layer with 4 features, a kernel size of 5, a stride of 1, and padding same. The final layer is a sigmoid activation layer. The structure of the generator is shown in figure 2.

The final implementation of the discriminator is composed of four modules of layers, followed by a flatten layer and a fully connected layer with a single output. Each of the modules consists of three layers. The first is a convolutional layer with 8, 16, 32, and 64 features respectively, a kernel size of 5, a stride of 1, and padding same. This is followed by a leaky relu layer with an alpha of 0.2. The third layer in each module is a dropout layer with a dropout rate of 40%. The discriminator consists of four of these modules where in each consecutive module the number of features in the convolutional layer is doubled. The final two layers of the network are a flatten layer and a fully connected layer with just one neuron that uses the sigmoid activation function. The structure of the discriminator is seen in figure 3.

The time to train the GAN network was 20 hours.

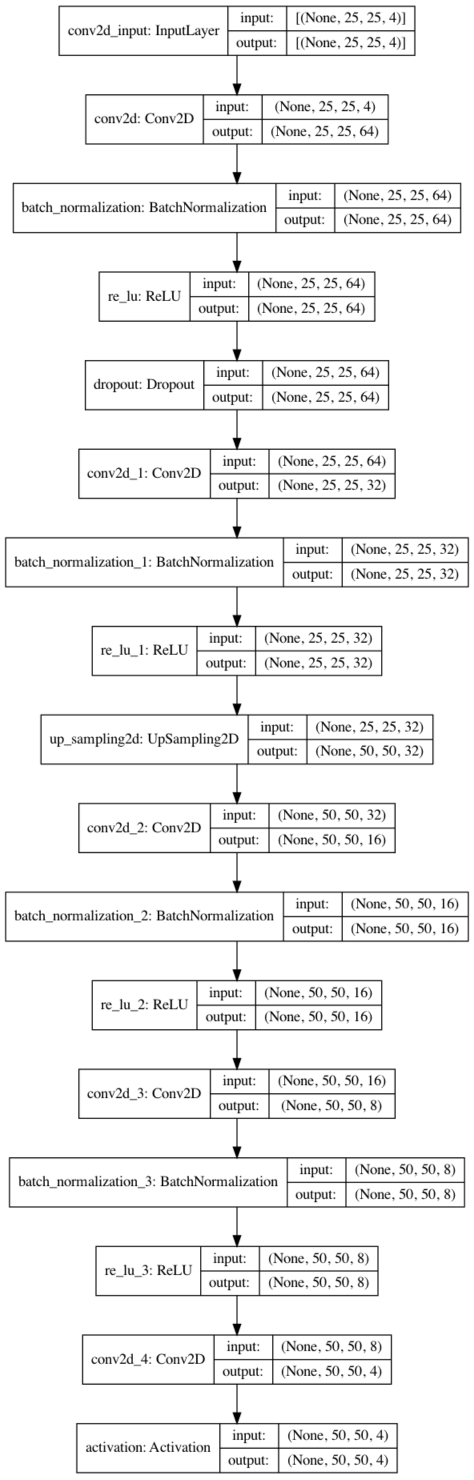
Hardware

All training was performed on a Windows 10 computer using an i7-8700k @ 3.8GHz CPU, 16GB DDR4 3200MHz RAM, and a GTX 1080 GPU. The Tensorflow/Keras environment was setup to train with the GPU, and take advantage of CPU multiprocessing.

V. Experiments and Results Analysis

CNN

For analyzing the performance of the CNN models created and tested in this project, the training and validation losses were tracked throughout their training until convergence. Converge was triggered based on the early stopping callback function described in section IV. For the MSE model, it converged in 263 epochs. The SSIM model converged in 242 epoch. The SSIM and MSE model converged in 195 epochs. The SSIM then MSE model converged in 324 epochs. The training and validation losses over epochs during training is shown in figure 4 and figure 5 respectively.



conv2d_input: InputLayer
input: [(None, 25, 25, 4)]
output: [(None, 25, 25, 4)]
conv2d: Conv2D
input: (None, 25, 25, 4)
output: (None, 25, 25, 64)
batch_normalization: BatchNormalization
input: (None, 25, 25, 64)
output: (None, 25, 25, 64)
re_lu: ReLU
input: (None, 25, 25, 64)
output: (None, 25, 25, 64)
dropout: Dropout
input: (None, 25, 25, 64)
output: (None, 25, 25, 64)
conv2d_1: Conv2D
input: (None, 25, 25, 64)
output: (None, 25, 25, 32)
batch_normalization_1: BatchNormalization
input: (None, 25, 25, 32)
output: (None, 25, 25, 32)
re_lu_1: ReLU
input: (None, 25, 25, 32)
output: (None, 25, 25, 32)
up_sampling2d: UpSampling2D
input: (None, 25, 25, 32)
output: (None, 50, 50, 32)
conv2d_2: Conv2D
input: (None, 50, 50, 32)
output: (None, 50, 50, 16)
batch_normalization_2: BatchNormalization
input: (None, 50, 50, 16)
output: (None, 50, 50, 16)
re_lu_2: ReLU
input: (None, 50, 50, 16)
output: (None, 50, 50, 16)
conv2d_3: Conv2D
input: (None, 50, 50, 16)
output: (None, 50, 50, 8)
batch_normalization_3: BatchNormalization
input: (None, 50, 50, 8)
output: (None, 50, 50, 8)
re_lu_3: ReLU
input: (None, 50, 50, 8)
output: (None, 50, 50, 8)
conv2d_4: Conv2D
input: (None, 50, 50, 8)
output: (None, 50, 50, 4)
activation: Activation
input: (None, 50, 50, 4)
output: (None, 50, 50, 4)

Fig. 2. GAN generator architecture.

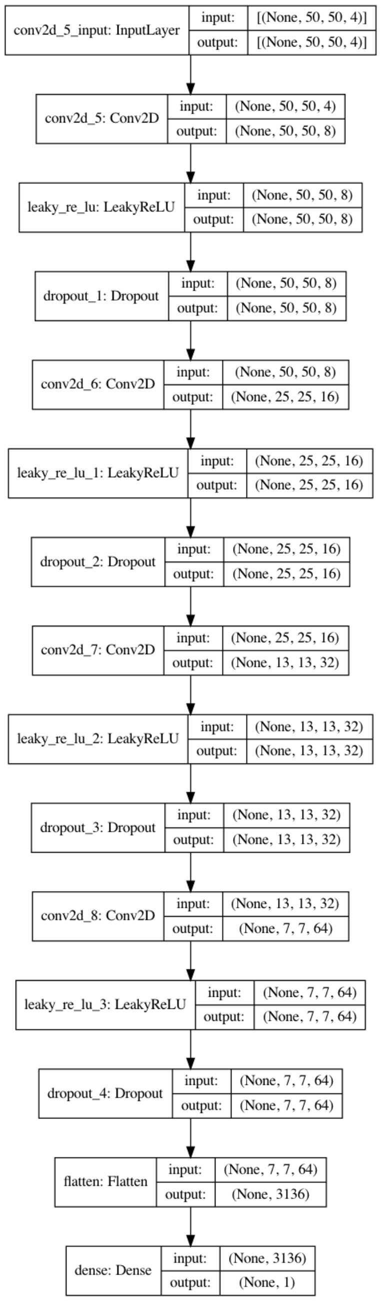


Fig. 3. GAN discriminator architecture.

Looking at the training losses, all models have good learning curves showcasing the models are actually learning over the epochs. For the SSIM then MSE loss curve, the steep drop after 300 epochs comes from the model switching from learning with the SSIM loss to the MSE loss. The loss curve for the SSIM then MSE model shows that the SSIM loss training does the majority of training, then the MSE continues to fine tune the model. The SSIM and MSE loss does not seem to level out its curve before stopping, and seems like it could benefit from further training, but the stopping occurs due to the validation losses not getting better, which keeps the model

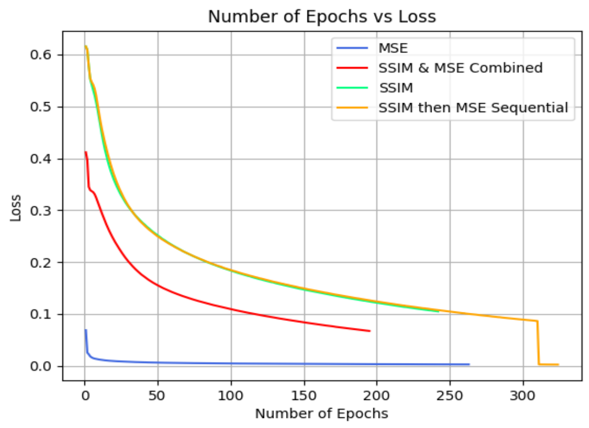


Fig. 4. CNN training loss over epochs.

from over-fitting on the training data. The SSIM only loss shows a similar curve to the SSIM then MSE model while its only using SSIM, however does not train as long. This is most likely due to the random weight generation causing the model to have a different starting point, and like the SSIM and MSE combined loss, seems like it could train a little longer, but stops early due to the validation losses. The MSE loss curve seems quickly drops then even out and slowly decrease in loss over time, but these small changes in loss are exaggerated due to the scaling of the other losses, and the relative range of MSE loss values.

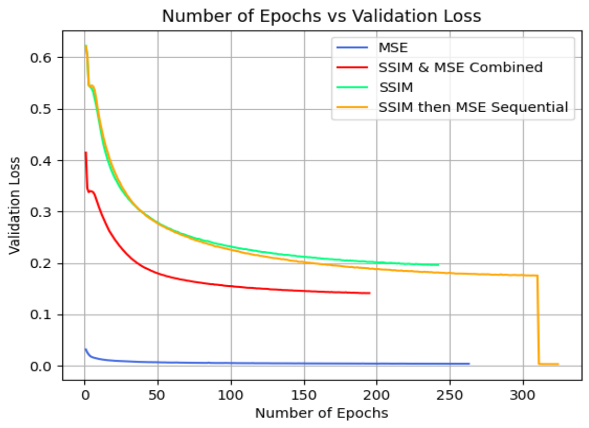


Fig. 5. CNN validation loss over epochs.

Moving on to the validation losses, the validation loss plot provides similar trends to the training loss values, showing that the model was doing a decent job of not over-fitting to the training samples too much, and keeping the model generalizing well. Obviously the training losses are a bit lower compared to the validation losses, but this is to be expected since the training data is the data directly influencing the updates made to the model while training. Some loss lines begin to even out sooner than in the training loss plot, showing that the model

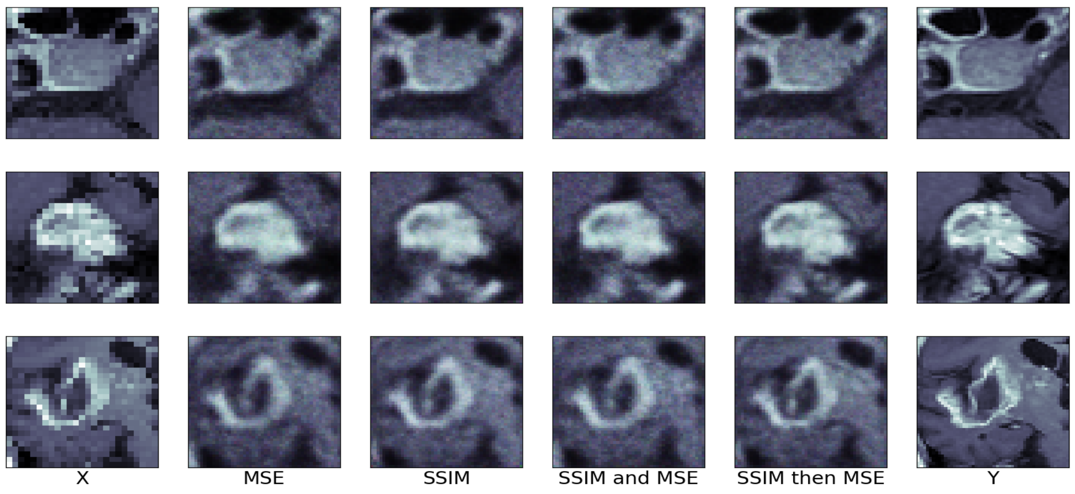


Fig. 6. CNN visual results on testing samples.

is beginning to start over-fitting on the training data if the training were to continue, which is why the early stopping callback function was put in place and chosen to monitor the validation loss.

To showcase a visual representation of the CNN performance, 3 testing samples were chosen and displayed for each model. In figure 6, each row represents a testing sample, and each column represent where the image came from. The first column represents X, or the input / downscaled 25x25x4 image. The second column represents the MSE model's prediction. The third column represents the SSIM model's prediction. The fourth model represents the SSIM and MSE model's prediction. The fifth column represents the SSIM then MSE model's prediction. Lastly, the sixth column represents the Y or true / original supersampled 50x50x4 image.

Looking at the overall visual results, the SSIM then MSE model seems to provide the best results. The predicted images from this model have small advantages over the other models in both detail and coloring / shading. Looking at the first row representing the first image from the testing set, one area the SSIM then MSE model provides slightly better detail at is in the top of the image. The top of the image has some small white connecting lines in the area that wrap around black areas of the image. In the true image, the white lines are most clearly defined, and looking at the model predictions, the SSIM then MSE model seems to slightly do better than the rest at defining these lines. The other models have some form of lines defined in this area, but either are darker or more fuzzy on the actual outline.

Similar results are shown in the second testing sample. Looking at the true image, there are many dark defined lines in the center area. The SSIM then MSE seems to again provide slightly better results in the detail of these lines and the coloring / shading. The MSE model in this sample shows the least amount of detail in this area, not capturing much of the lining structure. The SSIM does a better job as this, showing that its structural loss function does a better job at detailing. Then the SSIM and MSE model also does a fair job, with similar results to the SSIM model. However, the SSIM then MSE model provides slightly better definition of the lining in this area, and even does a good job with the coloring and shading, showing that this setup was capable of both learning the structure of the image and the coloring better than the other models tested.

Looking at the last testing sample, this testing sample has a lot of small lining detail throughout the image. Analyzing the different dark linings around the center of the image, again the SSIM then MSE model seems to do the best compared to the other models. It does a better job at defining the linings in some areas, and provides good coloring / shading of the areas. These results are similar to the ones discussed for sample two, where the MSE does the worst, in that the details of the structure of the image are the most blurry. Then the SSIM model does a better job at defining areas of the structure, with similar performance provided with the SSIM and MSE model.

Overall, the CNN does a relatively good job at upscaling the 25 x 25 images to 50 x 50 images. The SSIM then MSE model seems to provide the best results compared to the other models testes. This is very interesting since this method of training a model is somewhat uncommon, and was an idea we came up with. This setup does seem to use SSIM to learn more about the structuring of the image, providing better results than just MSE, then having the model train with MSE after SSIM has converged, fine tunes the model even more to provide the best results.

All models do lack the ability to extract very fine details that

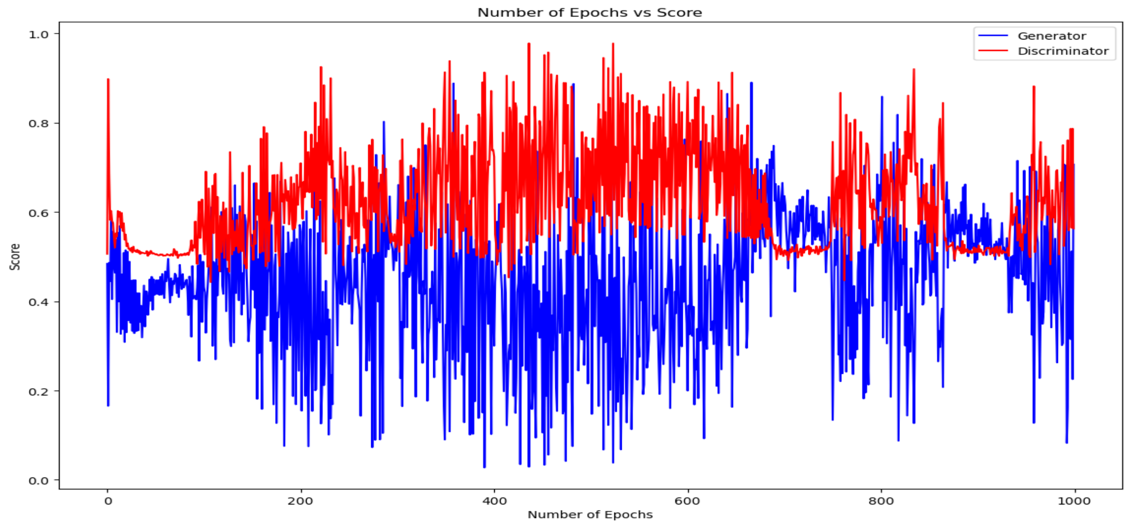


Fig. 7. GAN score over epochs.

are present in the true images, and the coloring's / shading's aren't as smooth. This could partly be due to the original images being at such a small scale compared to the true images being used, with only 1/4 of the imaging information. Additionally, the models themselves could just not be large enough to extract the necessary fine details and coloring / shading. The limited loss functions tested could also play a role, since there are various other loss functions that could be used in image supersampling to combat some of these issues.

GAN

To analyze the performance of the GAN model, we used a custom scoring metric based on the overall accuracies of the generator and the discriminator [20]. The score for the generator is calculated by taking the mean of the probabilities corresponding to the discriminator's output for the generated images of the validation set. The score for the discriminator is calculated by taking the mean of the probabilities that the input images from the generator and the ground truth images of the validation set belong to the correct class. These scores should indicate which of the two networks is outperforming the other and by how much. The equations for the generator and discriminator scores can be found in figures 8 and 9.

$$\text{scoreGenerator} = \text{mean}(\widehat{Y}_{\text{Generated}})$$

Fig. 8. GAN generator scoring metric.

The GAN was trained for 1000 epochs and the scores of its generator and discriminator are shown in figure 7. The results indicate that the discriminator remained slightly better at detecting generated images than the generator is at fooling the discriminator.

$$\text{scoreDisriminator} = \frac{1}{2}\text{mean}(\widehat{Y}_{\text{Real}}) + \frac{1}{2}\text{mean}(1 - \widehat{Y}_{\text{Generated}})$$

Fig. 9. GAN discriminator scoring metric.

To showcase a visual representation of the GAN performance, 3 testing samples were chosen and displayed. In figure 10, each column displays the x, generator, and y images while the rows are the three different samples.

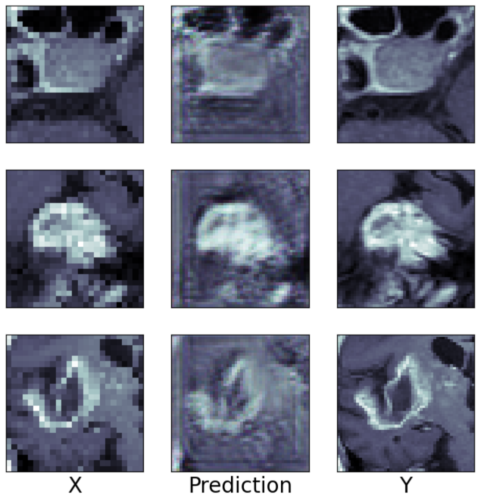


Fig. 10. GAN visual results on testing samples.

Comparing these results to the results of the CNN models, these results do not look as convincing. This could be contributed to the network needing to train for longer, or maybe the binary cross-entropy loss function for the generator could have been supplemented with the MSE or SSIM loss functions. However, from the results that we obtained, the GAN generated the worst results out of all of the models that we tested.

VI. Conclusion

In this work, we have presented two different models for supersampling brain tumor MRI scans and evaluated the results. The task was best modeled by the CNN, showing the greatest reconstructive accuracy in the form of the clearest qualitative image results. Of the CNN model, we concluded that the SSIM loss followed by MSE performed the best. While the GAN did not perform as well and have more issues with convergence, other GAN models have performed well doing similar tasks. In the future, redesigning of the implementation and training including testing more loss functions and sequential training techniques could be beneficial to its performance. The GAN could also be improved by supplementing its adversarial loss with a perceptual loss. In addition, more training iterations could improve CNN performance. Overall, the CNN model in particular was able to achieve a fair amount of success that can be applied to and improved to achieve adequate image restoration in medical imaging.

References

[1] Awsaf, "Brain Tumor .npy," Kaggle, 18-Apr-2020. [Online]. Available: https://www.kaggle.com/awsaf49/brain-tumor. [Accessed: 03-May-2021].

[2] awsaf49, "Brain Tumor Visualization," Kaggle, 18-Apr-2020. [Online]. Available: https://www.kaggle.com/awsaf49/brain-tumor-visualization. [Accessed: 03-May-2021].

[3] "Image tutorial," Image tutorial - Matplotlib 3.4.1 documentation. [Online]. Available: https://matplotlib.org/stable/tutorials/introductory/images.html. [Accessed: 03-May-2021].

[4] K. Team, "Keras documentation: EarlyStopping," Keras. [Online]. Available: https://keras.io/api/callbacks/early_stopping/. [Accessed: 03-May-2021].

[5] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.

[6] G. Leite, "An introduction to Convolutional Neural Networks for image upscaling using Sub-pixel CNNs," Medium, 02-Dec-2020. [Online]. Available: https://guipleite.medium.com/an-introduction-to-convolutional-neural-networks-for-image-upscaling-using-sub-pixel-cnns-5d9ea911c557. [Accessed: 04-May-2021].

[7] R. Keys, "Cubic convolution interpolation for digital image processing," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 29, 1981.

[8] Javeria Amin, Muhammad Sharif, Mussarat Yasmin, Steven Lawrence Fernandes, "Big data analysis for brain tumor detection: Deep convolutional neural networks." Future Generation Computer Systems, Volume 87, 2018, Pages 290-297, ISSN 0167-739X,

[9] C. Dong, C. C. Loy, K. He, and X. Tang. "Image super-resolution using deep convolutional networks." IEEE Transactions on Pattern Analysis and Machine Intelligence, 38(2):295–307, 2016.

[10] Jiwon Kim, Jung Kwon Lee, Kyoung Mu Lee; "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition" (CVPR), 2016, pp. 1646-1654

[11] Muhammad Haris, Gregory Shakhnarovich, and Norimichi Ukita. "Deep back-projection networks for super-resolution". In CVPR, pages 1664–1673, 2018. 2

[12] Wei-Sheng Lai, Jia-Bin Huang, Narendra Ahuja, and MingHsuan Yang. "Deep laplacian pyramid networks for fast and accurate super-resolution". In CVPR, pages 624–632, 2017. 2

[13] K. -W. Hung, Z. Zhang and J. Jiang, "Real-Time Image Super-Resolution Using Recursive Depthwise Separable Convolution Network," in IEEE Access, vol. 7, pp. 99804-99816, 2019, doi: 10.1109/ACCESS.2019.2929223.

[14] H. Zhao, O. Gallo, I. Frosio and J. Kautz, "Loss Functions for Image Restoration With Neural Networks," in IEEE Transactions on Computational Imaging, vol. 3, no. 1, pp. 47-57, March 2017, doi: 10.1109/TCI.2016.2644865.

[15] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.

[16] Johnson J., Alahi A., Fei-Fei L. (2016) "Perceptual Losses for Real-Time Style Transfer and Super-Resolution." In: Leibe B., Matas J., Sebe N., Welling M. (eds) Computer Vision – ECCV 2016. ECCV 2016. Lecture Notes in Computer Science, vol 9906. Springer, Cham.

[17] Xu, Jinchang 'I&' Zhao, Yu 'I&' Dong, Yuan 'I&' Bai, Hongliang. (2017). "Fast and Accurate Image Super-Resolution Using a Combined Loss." 1093-1099. 10.1109/CVPRW.2017.147.

[18] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi; "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition" (CVPR), 2017, pp. 4681-4690

[19] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," arXiv.org, 10-Jun-2014. [Online]. Available: https://arxiv.org/abs/1406.2661. [Accessed: 04-May-2021].

[20] "Monitor GAN Training Progress and Identify Common Failure Modes," MathWorks. [Online]. Available: https://www.mathworks.com/help/deeplearning/ug/monitor-gan-training-progress-and-identify-common-failure-modes.html. [Accessed: 04-May-2021].

[21] "An Introduction to Super-Resolution using Deep Learning," BeyondMinds, 18-Mar-2021. [Online]. Available: https://beyondminds.ai/blog/an-introduction-to-super-resolution-using-deep-learning/. [Accessed: 04-May-2021].

VII. Appendix A: Code Design

All code, images used in this paper, and links to our pre-trained models can be found at https://github.com/TannerFry/CS525-Final-Project.

extract_crops.py

Script used for extracting the croppings of the tumor areas from the original Kaggle dataset. Code is based on the "Brain Tumor Visualization" notebook tied to the project on Kaggle [2]. The script will loop through all images in the original dataset, use the mask and bounding box information to extract the crops in a specified image size. The script has two functions: **get_bounding_box**, and **crop_to_bbox**. The **get_bounding_box** function will take the mask information, and extract the bounding box location. Then the **crop_to_bbox** will take the bounding box location, and use it to extract the are of the tumor from the original image, and return the cropped image. First the script extracts the croppings in 25x25x4 sizes, and saves them to a folder called "downscaled_cropped_images", then the script extracts the croppings in 50x50x4 sizes, and saves them to a folder called "original_cropped_images". To run the script, run with python3, no command line arguments, and make sure the original data from Kaggle is the local directory.

Usage: ”python3 extract_crops.py”.

balance_data.py

Script to provide support functions to create dataset of balanced classes (same number of tumor instances of each class). There are two functions in this script: **balance_data**, and **balance_distribution**. Both functions are used to read all of the data for the project and split them equally based on their classes into the training/validation/testing sets. The only difference between the two is one can be used with already uncompressed/cropped data, and one can be used with the original compressed .npy files from Kaggle. This file does not get run directly.

CNN.py

Script uses to create, train, and evaluate the CNN models. The script has 7 functions: **read_data**, **train**, **evaluate**, **SSIM_and_MSE**, **SSIM**, **setup_model**, and **main**. The **read_data** function handles reading in the data and splitting amongst training/validation/testing sets using the ***balance_data.py*** script. The **SSIM_and_MSE** and **SSIM** functions are two custom loss functions used for the CNN. **setup_model** creates the CNN model based on the architecture described in this project. **train** is in charge of training the CNN model based on the loss function the user specifies in the command line arguments, and saving the model and loss histories once the model has converged. **evaluate** is in charge of running all the testing data through the saved models and saves all the testing predictions to a folder. Lastly, **main** handles the input arguments and proper function calling. The user can either specify if they want to train a model or evaluate a model. If the user wants to train, they must specify the supported loss function to use. If the user wants to evaluate, the user must specify the path to the model to load and the loss function that was used for training it.

Usage:

python3 CNN.py train <MSE | SSIM | SSIM_and_MSE | SSIM_then_MSE>

python3 CNN.py evaluate model_path <MSE | SSIM | SSIM_and_MSE | SSIM_then_MSE>

GAN.py

Script used to create, train, and evaluate the GAN model. The script has 10 functions: **read_data**, **make_generator**, **make_discriminator**, **make_gan**, **score_generator**, **score_discriminator**, **plot_scores**, **train_gan**, **evaluate**, and **main**. If the user wants to train a GAN, they must specify the number of epochs they wish the GAN to train for. If the user wants to evaluate a stored GAN model, they must specify the path to the saved generator model to load, and then evaluate it.

Usage:

python3 GAN.py train epochs

python3 GAN.py evaluate model_path

CNN_plot_losses.py

Script to produce loss plots for the CNN based on validation loss as well as training loss. This script has two functions: **val_loss_plot**, and **loss_plot**. Both will produce loss vs. epoch plots for training and validation for the CNN based on all different loss models setup. The script uses the history files found produced when training the CNN models, so it expects all history .txt files to be present in the CNN_models folder. When you run the script, it will produce all the graphs for both training and validation loss automatically.

Usage: python3 CNN_plot_losses.py

plot_visual_predictions.py

Script uses generate the visual testing samples shown in this paper. The script has 6 functions: **read_data**, **CNN_results**, **GAN_results**, **SSIM_and_MSE**, **SSIM**, and **main**. The **read_data**, **SSIM_and_MSE**, and **SSIM** are the exact same functions as described in the ***CNN.py*** script. The user specifies which model to generate the visual graph for: CNN or GAN. Once specif ed using the input arguments, the respective results function will be called: **CNN_results** or **GAN_results**. Within each function, 3 testing samples are picked, and run through the each model for predictions, and placed in a plot with the input and true images.

Usage: ”python3 plot_visual_predictions.py <CNN | GAN >”.

VIII. Appendix B: Workload Distribution

Tanner: In charge of cropping extractions from Kaggle dataset, CNN model implementation and testing, and CNN related information in sections III, IV, and V.

Racheal: In charge of compiling literature review of previous work. Also in charge of creating helper scripts, such as plots and dataset balancing. In charge of sections I, II, VI, and the abstract.

Alan: In charge of GAN model implementation and testing, and GAN related information in sections III, IV, and V.