DeepPaint: A Tool for Image Inpainting

Kushagr Gupta
Stanford University
kushagr@stanford.edu

Suleman Kazi
Stanford Unversity
sbkazi@stanford.edu

Terry Kong
Stanford Unversity
tckong@stanford.edu

Abstract

This project focuses on solving the inpainting problem using both Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) approaches. Popular methods of inpainting include Adobe Photoshop's Content Aware Fill, which do not use neural networks. The goal of this project was to explore successful architectures and use them as a foundation in creating our own. The best architectures that we came up with are a CNN with Sigmoid Euclidean Loss and a simplified PixelRNN.

1. Introduction

The goal of inpainting is to fill in a portion of an image that is either corrupted or unwanted. The applications of inpainting include: removing noise from an image, undoing deterioration (in historical or old photos), or to simply add or remove elements from an image. It can also be incorporated indirectly in compression wherein some percentage of original image can be transmitted and the whole can be reconstructed on the other end using a pretrained neural network.

There are many approaches to the inpainting problem, so we have decided to focus on models whose primary objective is to determine the best texture for a selected region. The texture of a region that must be reconstructed is something a talented painter could ascertain by looking at different areas of the painting. Like a talented painter, we preferred architectures that would learn how to paint missing textures by looking at different areas of the image. This contrasts other models, such as generative adversarial networks[1], whose objective is to minimize the probability of making a mistake. In the end, the upshot for both generative adversarial nets and our proposed architectures is the same.

Also, there has been recent success with two-dimensional RNNs in [2] and [3], which is encouraging news and are the motivating architectures for the PixelRNN presented in this report.

For both the CNN and RNN, the input is an $(N \times N \times 3)$ RGB image, where N is the width and the height of the image. Since we present multiple CNN architectures in our report, the output is either a reconstruction of the entire RGB image, or a reconstruction of a small patch of the image $(n \times n \times 3)$. The output of the RNN is the entire RGB image.

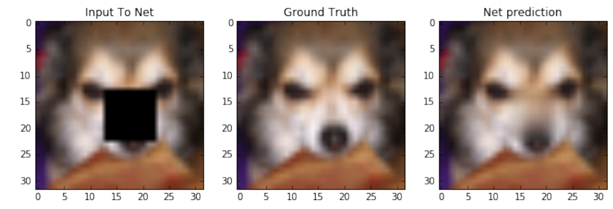


Figure 1: Example of Image Inpainting using our CNN

2. Related Work

Interesting work has been done to solve the inpainting problem using Neural Network approaches spanning from CNNs to RNNs. [4] follows a blind inpainting approach wherein they segment the corrupted image into small subimages and feed that through a network of five conv relu layers followed by a MSE calculation i.e. the euclidean loss function. The positive aspects of this approach are that it works well in removing small and local distortions but one of the limitations is that there needs to be a separate network for each type of corruption.

[5] uses an exemplar-based inpainting approach based on techniques explored in [6] and [7] for texture synthesis and inpainting and obtains amazing results in terms of filling the corrupted region with background information. This method is not able to fill in the ground truth, although this can be solved using a neural network approach given sufficient amount of training data and enough parameters to learn variability of data. [8] presents a novel approach to image denoising and blind inpainting combining sparse coding and deep neural nets using pretrained encoders and does a good job in removing overlaid text and small corrupted pixels but doesn't present any discussion for larger corruptions.

[2] and [3] covers RNN and LSTM approaches to solve the inpainting problem by building a generative model that exploits the spatial structure of the entire image. They define a conditional distribution to learn every pixel value with respect to other pixels in the image and the different channels (RGB) and the RNN uses a sequence of previous pixels seen in the image to learn a dependency. They cover both a row and diagonal RNN/LSTM which differ in how their receptive fields grow; for row RNNs the receptive field grows upward and for diagonal RNNs the receptive field grows diagonally. The positive aspects of this approach are that it can be used to paint large regions, but as a caveat, it takes an incredibly long time to train as compared to a CNN approach due to the RNN/LSTM's sequential nature.

3. Methods

The image inpatining problem can generally be classified into two types:

1. **Non-Blind Inpainting:** In non-blind image inpainting the location of the pixels needed to be painted are known a priori and provided to the network in some form.
2. **Blind Inpainting:** In the blind image inpainting problem no information about the location of the regions that are corrupted or need to be filled in is given to the algorithm or network and the network must automatically locate as well as fill in these regions. This technique is useful in cases in which the areas that need to be filled in are not well defined for examples removing scratches and age-related wear from vintage photos. Blind inpainting is a harder problem to solve than non-blind inpainting.

In our project we try both the non-blind and blind inpainting methods and find that networks trained to do non-blind inpainting perform much better than those trained to do blind inpainting.

Loss Functions:

We experiment with two types of loss functions in our project, both of these are computed pixel-wise for the region to be inpainted.

1. **Euclidean Loss:** The euclidean loss is the equivalent to the L2 loss function. It is commonly used in regression problems. The pixel-wise squared euclidean distance between the area to be inpainted and the ground truth is considered to be the loss:

$$Loss_{euclidean} = \sum_{i=1}^{N} (x_i - x_{true_i})^2$$

Where N is the number of pixels in the region to be painted, x_i is the i^{th} pixel in the output of the network and x_{true_i} is the ground truth value of the pixel at the same location.

2. **Softmax Loss:** The softmax loss is sometimes called the negative log-likelihood loss. It is commonly used in classification problems. For the softmax loss, the loss is computed pixel-wise where the classes are the possible RGB triples a particular pixel can take on. As an example in an 8-bit 3 channel (RGB) image we have 255 possible values for the intensities of each of the three channels which means that the number of possible classes is 255^3, in order to reduce the large number of classes we quantize the image (for details please see dataset section). The softmax loss is calculated as:

$$Loss_{softmax} = \sum_{i=1}^{N} L_i$$

Where:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Table 1: Different CNN Architectures

Blind Softmax
5 x [Conv-ReLu] - [FC] - [Softmax Loss]

Non-Blind Softmax
6 x [Conv-ReLu] - [FC] - [Softmax Loss]

Non-Blind Euclidean w/o Sigmoid
10 x [Conv-ReLu] - [FC] - [Euclidean Loss]

Non-Blind Euclidean with Sigmoid
10 x [Conv-ReLu] - [FC] - [sigmoid] - [Euclidean Loss]

3.1. CNN Architecture

After conducting a literature review we came up with a few architectures that we wanted to test. These differ in the number of parameters, number of layers, and in what they output depending on whether they use a softmax or a euclidean loss and whether they are blind or non-blind (For the exact details, e.g. number of parameters, please see the CNN Setup section). We use a number of conv-ReLU[1] sandwich layers followed by a fully connected (FC) layer, followed by a sigmoid layer, and finally followed by a softmax loss or euclidean loss. The high-level difference between the networks with euclidean loss and softmax loss is that the output of the euclidean networks is a continuum of RGB triples, whereas with the softmax networks the output for each pixel is the correct class (discrete), and we have to convert the class back to the RGB values for that class.

Using repeated conv layers with a small filter size allows us to increase the effective receptive field for each neuron

[1]A conv layer is the same as a convolutional layer

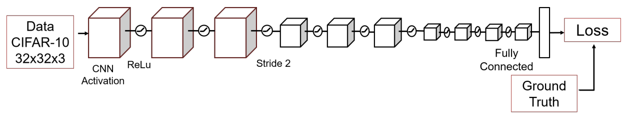


Figure 2: General skeleton of CNN networks used

whilst keeping the computational expense manageable. The ReLU activations introduce non-linearities and offer sparse activation and efficient gradient back-propagation. The FC layer at the end of the network introduces additional parameters and also allows us to reshape the output of the conv layers into the desired shape.[2].

As shown in the results section, using the *Blind Softmax* architecture does not perform well. We hypothesize this is due to the fact that the network must reconstruct every pixel in the image since it is not given information about the regions it must reconstruct; so in this sense, the problem is harder because there is a tougher objective to satisfy.

The *Non-Blind Softmax* architecture also does not give visually pleasing results and introduces patterns of black pixels in the reconstructed region. This might be due to the fact that we have a large number of classes for the softmax loss (even after we quantize the image we have 512 possible pixel classes) so the network is not able to classify them correctly given the amount of training data we have.

The best performing network we have (and the one we used as our final network and trained on the full CIFAR -10 dataset) was the *None-Blind Euclidean with Sigmoid* architecture. In early experiments we had the fully connected layer's output going directly to the euclidean loss but we found that some pixels saturated in the output. We hypothesize that this could be due to the fact that the FC layer output was unbounded and had no guarantee of being within the [0,1] normalized range for pixel values; we fixed this problem by adding a sigmoid layer after the FC layer which squashes the FC layer output to the required range before passing it to the loss.

3.2. RNN Architecture

The RNN architecture implemented for this project is based on Google Deepmind's Multi-scale PixelRNN[2]. The novelty in this approach is how new hidden states are computed. Since an image is not a one-dimensional data structure, it does not have a linear ordering. For this reason, any RNN that operates on an image must specify its traversal on the input image. We first present an overview of the architecture and then we list the key differences from this architecture and the one presented in [2]. We will call the our RNN architecture the *Simplified PixelRNN* from this point forward.

[2]The desired shape output shape for the non-blind nets is the size of the corrupted patch and for the blind nets is the size of the input image.

The general form of the Simplified PixelRNN is:

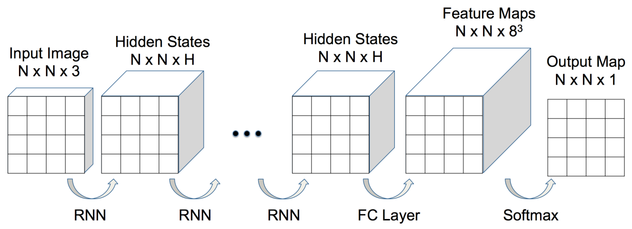


Figure 3: The general form of the Simplified PixelRNN Architecture. The ellipsis indicates that any number of RNN layers can be added in order to scale the model up. The blocks represent the activation layers while the arrows represent the layers.

We also provide an intuition on the function of each layer in this architecture. Since the cornerstone of this architecture is the RNN, we can have any number of RNN layers, where more RNN layers means that the model is more expressive. The FC Layer functions as a way to map the final RNN activation layer to classes so that we can take the softmax. The classes in this case are the different colors a pixel can take on. For this reason, the Simplified PixelRNN is a generative model since it attempts to generate the complete color distribution over all images. The only detail we have not mentioned is the update for the hidden states.

3.2.1 RNN Hidden Update

The choice for the traversal over the image will be row-by-row in the Simplified PixelRNN. This has two implications: 1. This means that the receptive field for a given pixel is always above that pixel (it is in fact triangular as we will later show) 2. The RNN can only predict pixels that are below a selected row if the rows above that are completely given. There is a hyperparameter, k, that controls how many previous input pixels and previous hidden states are used in the calculation of the new hidden state. We have seen that $k = 3$ works very well in [2], so we will fix this in the Simplified PixelRNN.

Figure 4 illustrates how a new hidden state is calculated.

The update equation for the Simplified PixelRNN is thus:

$$h_{(i,j)} = tanh(W_x x + W_h h + b),$$

$$x = \begin{bmatrix} x_{(i-1,j-1)} \\ x_{(i-1,j)} \\ x_{(i-1,j+1)} \end{bmatrix}, h = \begin{bmatrix} h_{(i-1,j-1)} \\ h_{(i-1,j)} \\ h_{(i-1,j+1)} \end{bmatrix}$$

As the update equation and Figure 4 illustrate, the receptive field for any pixel in the output is triangular with the triangle beginning at an output pixel and opening upward.

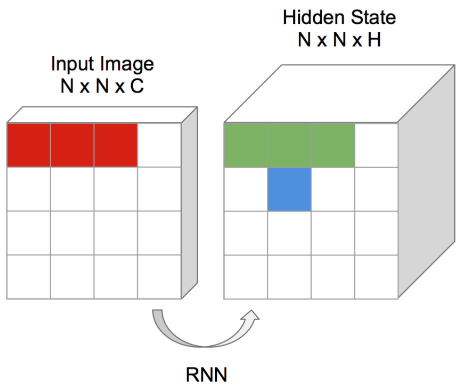


Figure 4: Hidden state update illustration from input to hidden states. The blue square is the location of the new hidden state. The green squares are the ”previous” hidden states that the new hidden state uses for its update. The red squares are the pixels in the input that the new hidden state uses for its update

Since the Simplified PixelRNN is a multi-layered RNN, it can have many of the layers defined above. The only constraint is that at a particular layer, l, the currently updated hidden state in layer l can only be computed once the triangular region above that pixel on earlier layers have been computed.

A minor detail when updating the hidden states is that the value of a pixel outside of the input activation layer and the current activation layer is assumed to be 0. This way all pixels are well-defined.

3.2.2 Simplified PixelRNN Backpropagation

We do not provide the equations for the backpropagation of the simplified PixelRNN since they are almost exactly the same as the standard RNN. The one thing to note when backpropagating is that the RNN backpropagation looks slightly different depending on whether it is the final RNN layer or not. The difference is that if we are currently backpropagating into the last RNN layer, only the gradient that contributes to the pixel at index (i, j) in the current RNN layer is the (i, j)th gradient in the (FC) layer above it. Whereas if we are backpropagating into an RNN layer that is not the last RNN layer, the pixel at index (i, j) in the current RNN layer receives gradients from the $(i+1, j-1)$, $(i+1, j)$, and $(i+1, j+1)$ indices in the (RNN) layer above it.

3.2.3 Differences from DeepMind’s PixelRNN

We list the main of differences between the Simplified PixelRNN and the the PixelRNN in [2].

1. The Simplified PixelRNN outputs all three of the color channels at each iteration while the PixelRNN outputs the color of one particular channel at each iteration

2. The Simplified PixelRNN does not use any information about the current pixel (This is the main reason this model is named ”Simplified”). Since the PixelRNN updates color channels one at a time (in the order of red, green, then blue), it may use some of the color channels of the current pixel. For example, while predicting the green channel of the current pixel, the PixelRNN would use the red channel, but not the blue channel since the blue channel has not been predicted yet.

4. CNN Datasets and Features

For the CNN, the CIFAR-10 data set was used, consisting of 60000 images of size 32x32x3. We generated a training set of 50,000 images and validation and test sets consisting of 5000 images each.

1. **Blind Inpainting:** For this approach we added three 5x5 masks at random locations in the data set, as can shown in 5, which became our input to the network. Ground truth in this case were the true images themselves.

2. **Non-Blind Inpainting:** One spatially fixed 10x10 mask was used to generate the corrupted input data set. Ground truth in this case was the small 10x10 true patch that was removed from the image, as shown in 6.

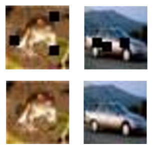


Figure 5: Illustration of Input and Ground Truth for blind network

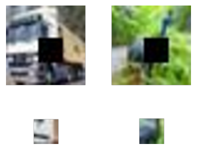


Figure 6: Illustration of Input and Ground Truth for Non-blind network

After generating the input and ground truth we converted the datasets into LMDB format which could be easily read in caffe. For the softmax loss we quantized the color space into 8 bins in each channel, thus giving a total of 512 labels. We then converted the ground truth images into corresponding labels, since the softmax layer uses these labels to calculate loss and do back propagation. Similarly during testing time output of FC layer i.e. the labels were converted back to images for visualization. For the euclidean loss, however, quantization wasn’t carried out and the images were fed as it is.

Other preprocessing steps involved scaling the dataset to lie in $[0, 1]$. We didn't remove the mean image from data as [2] didn't do that operation either.

5. RNN Dataset and Features

The Simplified PixelRNN also uses CIFAR-10, but it only uses one class: frogs. For training, the entire uncorrupted image is passed through the network, and the output is an array of integers which map to RGB triples. The ground truth used to compute the loss is the entire image. Normally an RNN uses the input shifted by one as the ground truth to calculate the loss, but the Simplified PixelRNN does this internally.

At testing time, an image with only the upper half is given and the PixelRNN will output the entire color map, which can then be converted to RGB triples through a mapping.

6. Experimentation Setup

The following sections describe the experiments done and the setup for both the CNN and RNN architectures along with the observations made about each.

6.1. CNN Setup

We used the BVLC caffe library [11] and compiled the GPU version of it on one of our machines in Linux (Ubuntu) environment. Various architectures were tried and evaluated to see which perform the best for the task at hand by creating a separate prototxt file for each network.

6.1.1 Networks

We started off with a *Blind Softmax* network consisting of five Convolutional and ReLu (activation) layers made up of 3x3 filters followed by a fully connected (FC) layer with 512 outputs corresponding to the eight bit quantized color space feeding into a softmax loss function to solve the blind inpainting problem wherein the location and size of corruption is unknown. We realized that since the whole image is reconstructed in this case, we would need a large number of parameters and the time to train the network would be prohibitively large given limited computational power. Thus, we decided to stick to solving the non-blind problem which yielded much more promising results.

The second network *Non-Blind Softmax* had six conv layers wherein the first 5 layers had 64 filters of size 3x3 and the last one had 3 filters of size 3x3. Each conv Layer was followed by a ReLu layer and finally a FC layer with 512 outputs which went to the softmax loss layer. Initial results obtained didn't give decent performance which could be due to the fact that we were trying to construct a generative model in the pixel space, which would have required much more training data and time.

In order to improve the performance we modified our loss function from softmax to euclidean(L2) loss in order to make it into a regression problem. Our network i.e. the *Non-Blind Euclidean w/o Sigmoid* had 10 Conv ReLu layers with filter size of 3x3, followed by a FC layers mapping to 300 outputs and a euclidean loss layer. We trained two models, one having 64 filters in each layer and every third conv layer having stride 2. The other network had 16 filters in first 3 layers, 32 in next 3 and 64 in the last four with every third layer having stride 2. Input to the network was a 32x32x3 image which after passing through the conv layers gave an output of size 4x4x64 which was fed to the FC layer that gave a vectorized output of size 300 corresponding to the 10x10x3 ground truth for the corrupted region.

In the final network *Non-Blind Euclidean with Sigmoid* we also added a sigmoid layer just before the loss function in order to reduce color saturation as it helps in squashing the outputs of the FC layer within the range [0,1].

6.1.2 Solver

Initially we used the default SGD solver but as it took a huge number of iterations to converge we shifted to the Adam solver which converged roughly 10x times faster. Since we wanted to learn aggressively during the initial phase while switching to smaller learning rates periodically the network periodically reduced the learning rate by 80% of its value every 50000 iterations.

[4] uses a batchsize of 1 on their entire dataset. Our dataset consisted of 50000 training images so we chose a batch size of 50 in order to be able to train faster as compared to just training on one image while still allowing the loss to "wiggle" while the weights are updated. Our batchsize is on the same order as mentioned in [2] and it helps in regularizing the weights and prevents overfitting the network. Along with this we also used a weight decay parameter to explicitly prevent the network from overfitting and tried various values ranging few orders of magnitude and finally decided on $\lambda = 0.005$.

6.2. RNN Setup

The Simplified PixelRNN architecture was built using the CS231N homework code as a template. This decision was made in the interest of time since we were unsure whether we could finish testing both the CNN architecture in caffe and the RNN with a GPU-capable package like Torch.

The Simplified PixelRNN is only able to run in CPU mode since all of the code uses python and numpy. Python is a language that is quite slow at running explicit for loops, which means that for an RNN, which is sequential by nature, it is not the preferred environment. This penalty in speed allowed us to only train on a small number of layers and a small training set. Despite this inconvenience, we still observed very promising results.

All networks were trained with a 2.3GHz Intel Core i7. Each network took about 12 hours to train.

There were two networks we trained for a substantial amount of time.

1. Simplified PixelRNN with 1 RNN Layer on 50 frog images from CIFAR-10 with a batchsize of 10. The hidden dimension was 1024. The learning rate was $5e-4$. There is a learning rate decay applied 0.95 after each epoch of training.

2. Simplified PixelRNN with 2 RNN Layers on 64 frog images from CIFAR-10 with a batchsize of 8. The hidden dimension was 256. The learning rate was $5e-3$. There is a learning rate decay applied 0.99 after each epoch of training.

Since it was infeasible to perform cross-validation given the time-constraint, we decided to use [2] as a baseline for hyperparameter choices. The batchsizes of 10 and 8 were chosen since they prevented the Simplified PixelRNN from overfitting to the training data. The reason only frog images were used as training was because we believed the texture and colors between from images of frogs were likely to be very similar, which would yield better results despite our small training set.

All of the hyperparameters were hand curated by refining the next best set of hyperparameters iteratively. This was done simply because an exhaustive search over all hyperparameters was infeasible in terms of time.

6.2.1 Hidden Dimension

The choices of the hidden dimension hyperparameter for the two networks may seem odd at first glance, but there is a natural explanation. For the Simplified PixelRNN with 1 RNN Layer, to ensure that the model was expressive enough, we needed to increase the hidden dimension to a 1024 (note that this is eight times what is used in [2]). However, for the model with 2 RNN layers, a hidden dimension of 1024 was too large and led to overfitting since the second RNN layer made the model much more expressive and removed the need for a very high hidden dimension.

6.2.2 Solver

Both of the trained nets using the Adam update. Adam was a much better choice than vanilla gradient descent because Adam's convergence rate is much faster than vanilla gradient descent and this implementation of the Simplified PixelRNN was starved of computational resources.

7. Results

The results of inpainting are difficult to evaluate because they are very subjective. A particular network may paint a patch in a way that does not match the ground truth, but it can still look natural. Thus, the metric for evaluating performance should be the pixel-wise error, depending on the loss function, and not whether or not an image is perfectly reconstructed according to the ground truth.

For networks that use a euclidean loss (CNN only), the metric of performance is the total L2 loss over the mini-batch. As for the networks with a softmax loss layer (CNN and RNN), the metric is the negative log likelihood over the mini-batch.

We present some results for both our CNN and RNN implementations below more results can be seen in the appendix.

7.1. CNN Results

Blind Softmax Results: Only one example from this network is shown since it does not perform well and experiments on this were stopped after we were unable to achieve better results.

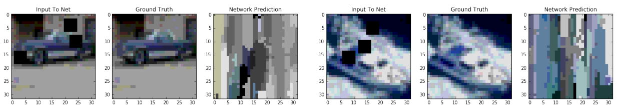


Figure 7: Test images for Blind Softmax architecture

Non-Blind Softmax Results: These are results using the *Non-Blind Softmax* architecture. As mentioned earlier, this architecture introduces jarring pixel artifacts.

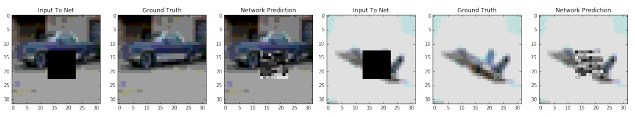


Figure 8: Test images for the Non-Blind Softmax architecture

Non-Blind Euclidean Loss Network w/o Sigmoid Results: These results are similar to our final chosen architecture except for the over saturated pixels, which were fixed with the sigmoid.

Non-Blind Euclidean Loss Network With Sigmoid Results: These results are similar to our final chosen architecture except for the over saturated pixels which were fixed with the sigmoid activation before loss. This network performs the best amongst all CNN's in our experiments.

CNN Loss Curves: With the numerous experiments tried for each type of network with parameter training it is

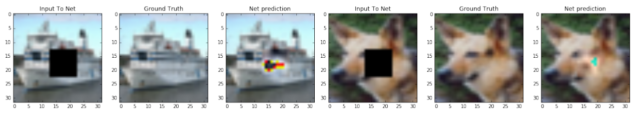


Figure 9: Test images for non blind euclidean without sigmoid

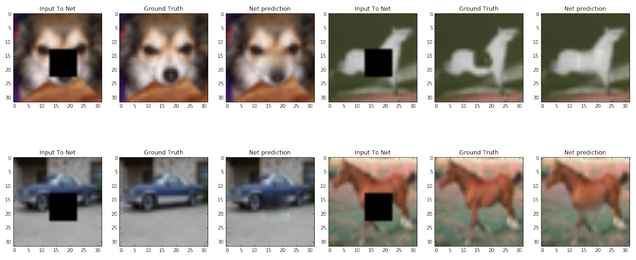


Figure 10: Test images for non blind Euclidean with sigmoid

not possible to display all of the loss curves that we generated. For the sake of brevity a typical loss curve obtained for best CNN: *Non-Blind Euclidean Loss with Sigmoid* in Figure 11.

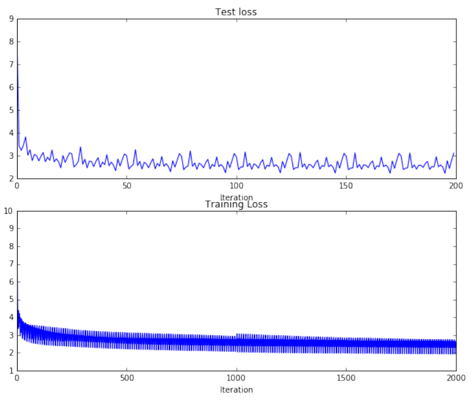


Figure 11: Loss curves for Non-Blind Euclidean Loss with Sigmoid. The train loss plot is the loss over the training set at each iteration. The test loss plot is the loss after every 100 training iterations.

7.2. RNN Results

To illustrate how well each Simplified PixelRNN model performed, we included both training loss curves in Figure 12. There is an aggressive drop in the loss at beginning which is the result of the network greatly improving its performance over the random initialization. The tail end of the loss curves seem to flatten out, but the network is in fact still learning. This is due to the fact that the Simplified PixelRNN uses a softmax loss layer. Because the softmax considers the argmax over all classes, a class's value only needs to inch itself ahead of the current largest class in order to become the largest. If that margin between the largest class and the new largest class is small, the loss may not change very much, but the change in the output class would be very noticeable in the result.

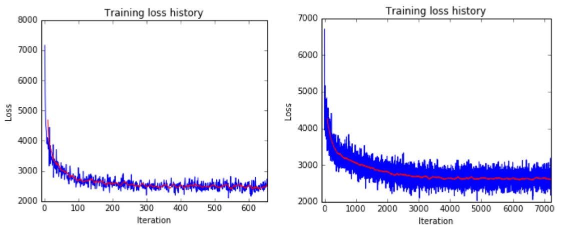
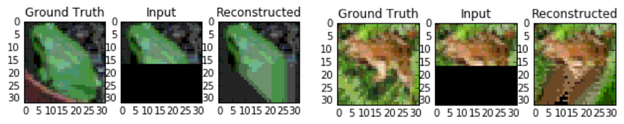


Figure 12: Simplified PixelRNN Training Loss curves. The left plot is for the network with 1 RNN layer and the right plot is for the network with 2 RNN layers. The blue curve is the loss at each iteration and the red curve is the average loss over the last 10 iterations (for the left) and 100 iterations (for the right).

We provide some results to both Simplified PixelRNN networks to qualify their performance and highlight their limitations in Figure 13.



(a) 1 RNN Layer (b) 2 RNN Layers

Figure 13: A test result from both Simplified PixelRNNs

Figure 13(a) illustrates a problem with only using 1 RNN layer. The pattern observed in most training and testing images when 1 RNN layer was used was that most frog images were painted with little to no texture. There also seemed to be a tendency to paint downwards using the color in the row above. While this is a very discouraging result, what is almost amazing is how well the Simplified PixelRNN works with 2 RNN layers.

What is surprising is that the Simplified PixelRNN with 2 RNN layers uses an order of magnitude less variables than the one with 1 RNN layer, but still performs better. In Figure 13(b) we can see that the network paints the frog with much more texture. Also, rather than painting downward, it seems to understand that frogs have legs so it should not continue painting downwards with the same color as the one above. We include more results for the Simplified PixelRNN with 2 RNN layers in the appendix.

Another thing to note is that there is a tendency in the 2 RNN layer Simplified PixelRNN to paint to the left. This is the result of only using 64 training examples where many of the training images all had this pattern.

8. Conclusion

In this project we tried out a number of different architectures and methods for blind as well as non blind image inpainting. We found blind image inpainting to be a much harder problem than non blind painting. During the course of the project we were exposed to Neural Net frameworks like Caffe and also came up with our own implementation from scratch for the Simplified PixelRNN. We gained practical experience with the training and fine tuning of neural networks and managed to produce reasonably good results.

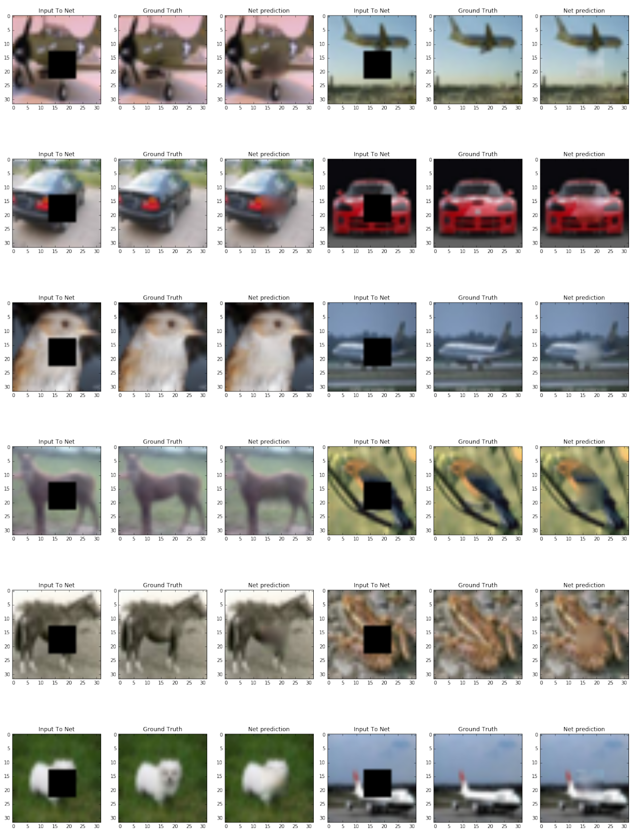
9. Future Work

There are a large number of possibilities that can be considered for further work after this project. We list some possibilities below:
1. At the moment we only test and train on the 32 x 32 CIFAR-10 dataset, the networks can also be trained and tested on higher resolution images or data sets like IMAGENET.
2. We only consider operating in the RGB space, but we could also try converting the image to the HSV space before trying to paint them. This might lead to better results because of a different separation of the Hue Saturation and Value leading to less jitter in the loss for the euclidean CNN architecture.
3. The simplified PixelRNN can be coded up into a GPU package to enable faster training and allow us to train it with a larger dataset with more images.
4. The RNN layers of the Simplified PixelRNN can be changed to LSTM layers.
5. At the moment we get our best results using non-blind inpainting methods. For the CNN the region to be painted is fixed (10x10 pixels from the center), different types of corrupted regions can be tried and the blind inpainting problem can be explored further to give better results.
6. A mask could be passed into the CNN architectures to allow the size of corrupted region to be learned.

10. Appendix

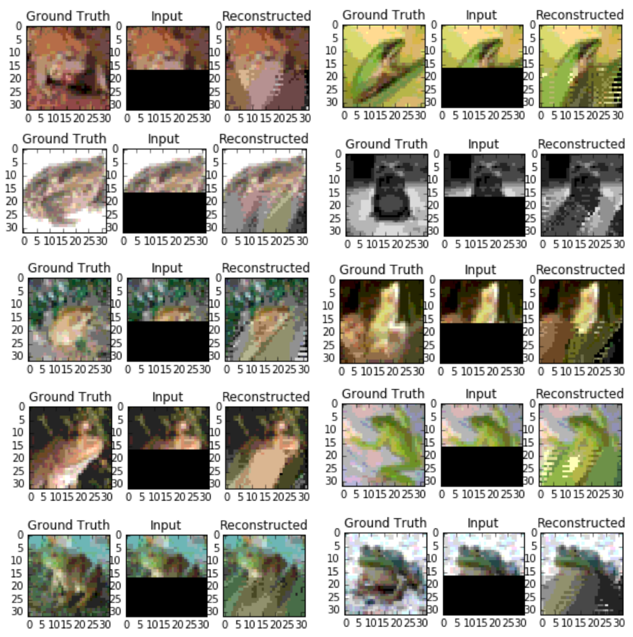
10.1. More CNN Results:

Note: All of the CNN results shown in the appendix are test images for our final architecture: NonBlind Euclidean loss with a sigmoid layer before loss



10.2. More RNN Results:

Note: All of the RNN results shown in the appendix are test images.



References

[1] Goodfellow, I, Pouget-Abadie, J., Mirza, M., Xu, B., Warde- Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In NIPS , 2014.

[2] Oord, Aaron van den, Kalchbrenner, Nal, and Kavukcuoglu, Koray. Pixel recurrent neural networks. arXiv preprint arXiv:1601.06759 , 2016.

[3] Theis, Lucas and Bethge, Matthias. Generative image mod- eling using spatial LSTMs. In Advances in Neural Infor- mation Processing Systems , 2015

[4] Blind inpainting using the fully convolutional neural network, Nian Cai, et al. , Springer 2015

[5] Criminisi, Antonio, Patrick Prez, and Kentaro Toyama. ”Region filling and object removal by exemplar-based image inpainting.” Image Processing, IEEE Transactions on 13.9 (2004): 1200-1212.

[6] Xu, Zongben, and Jian Sun. ”Image inpainting by patch propagation using patch sparsity.” Image Processing, IEEE Transactions on 19.5 (2010): 1153-1165.

[7] Fadili, Mohamed-Jalal, J-L. Starck, and Fionn Murtagh. ”Inpainting and zooming using sparse representations.” The Computer Journal 52.1 (2009): 64-79.

[8] Xie, Junyuan, Linli Xu, and Enhong Chen. ”Image denoising and inpainting with deep neural networks.” Advances in Neural Information Processing Systems. 2012.

[9] Bertalmio, Marcelo, Andrea L. Bertozzi, and Guillermo Sapiro. ”Navier-stokes, fluid dynamics, and image and video inpainting.” Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. Vol. 1. IEEE, 2001.

[10] Van Den Oord, Aron, and Benjamin Schrauwen. ”The student-t mixture as a natural image patch prior with application to image compression.” The Journal of Machine Learning Research 15.1 (2014): 2061-2086.

[11] Y. Jia, ”Caffe: An open source convolutional architecture for fast feature embedding,” gttp://caffe.berkeleyvision.org/,2013 2013