Image Inpainting with Context Encoders on MNIST

Jeffrey H. Hu and Jose I. Velarde

December 14, 2016

Abstract

We implement a novel machine learning approach to image inpainting introduced by Pathak et al in TensorFlow and explore its performance on the MNIST handwritten digits database. Image inpainting, the process of recovering lost or corrupted parts of an image, is typically restricted to recovering small defects. We use convolutional neural networks in an encoder-decoder scheme to extract semantic information from the image and use it to reconstruct hypotheses for large missing regions.

1 Introduction

Through this work, we explore the effectiveness of machine learning techniques at tackling the problem of image inpainting. Image inpainting is the process of reconstructing lost or corrupted parts of an image. Typical algorithms in this domain redraw the missing regions by interpolating pixels from their immediate context. This works well for small defects but is hopeless for recovering large portions of an image.

Pathak presented a novel approach for inpainting large regions by first using a convolutional neural network to achieve a semantic understanding of the image. The world around us is extremely varied but also highly structured. Similar to how a human artist could rely on this structure to paint a plausible hypothesis for a missing region, the goal is to train a neural network to regenerate missing regions with good semantic and pixel-wise coherency.

In this paper, we implement in TensorFlow the context encoder scheme proposed by Pathak. By gathering datasets of images and then randomly blacking out certain regions, we train a model to recover the original images. The network learns how to convolve an image down to a compact feature representation and use it for upsampling to form a hypothesis. Theoretically, the learned feature representation contains all the information necessary to regenerate the full image.

Due to limited time and computing power, we choose to use the MNIST handwritten digit database for its small image size and convenience.

2 The Model

The context encoder model uses a simple encoder-decoder scheme. The encoder converts the single channel, gray-scale, 28x28 input image into some m channel, nxn feature representation. If we connected this to a fully connected hidden layer and then 10 output units, we have the classic architecture for classifying digits. Instead, we connect the feature representation to the decoder via a channel-wise fully connected layer. This outputs an image with the same dimensions as the input image. By use this image with its pixel-wise L2 loss from the original, we can train for reconstruction accuracy.

2.1 Input

The model takes an image and a mask as input. A mask is a 28x28 array of ones with one square or rectangle of zeros. To black out a portion of an image, we simply element-wise multiply the image with the mask. In this paper, we randomly generate masks that black out anywhere from 1/8 to 1/4 of the pixels anywhere in the image. In this manner, we can iterate on a single image in the training set dozens of times with different masks.

2.2 Encoder

The encoder takes the masked image and uses k convolutional layers to arrive at a feature representation. For this application, we found that 3 convolutional layers, each with 16 feature maps, works well. The feature representation we obtain from this is 16x28x28. Pooling layers were intentionally omitted because they actually hurt performance. Classification CNNs use pooling layers to help recognize features with location invariance. For reconstruction however, location information is very important to help match a hypothesis with the provided context.

2.3 Channel-wise Fully Connected Layer

To connect the encoder and decoder, we could directly feed the feature representation from one into the other. In that case however, the filter activations from one side would not propagate and contribute to the regeneration of pixels on the other side of the image. The model needs a way to synthesize the activations, similar to how classification CNNs end with fully connected layers to synthesize the activations of their filters.

Another option is to fully connect the output layer of the encoder to the input layer of the decoder. Connecting m nxn feature maps however, results in an explosion of parameters. The m^2n^4 parameters largely negate the gains of using CNNs and dominate the total number of parameters that need training.

Pathak designed a neat compromise. A channel-wise fully connected layer fully connects encoder and decoder nodes that belong to the same feature map. This construction requires only mn^4 parameters to train and achieves much better performance in practice.

2.4 Decoder

The decoder architecture is identical to the encoder except in reverse. If there are k layers, the input to ith layer of the decoder has the same shape, filter size, and number of feature maps as the output to the $(k-i+1)$th layer of the encoder. Deconvolution can be understood as convolution with a fractional stride length or as the transpose of convolution. TensorFlow provides a convenient operation for this called conv_2d_transpose.

2.5 Loss Function

Let x be the input image and M be the mask. If we represent the model as function F, our pixel-wise L2 loss function can be expressed as follows:

$$L(x) = ||(1 - M) \odot (x - F(M \odot x))||^2 \tag{1}$$

where $\odot$ is the element-wise product operation.

Notice that we mask the pixel differences to only preserve losses on the missing region. This way, only the parameters from the context that contribute to the missing region are updated during backpropagation. We want to train for features that exist in the context and that are important for reconstructing missing regions and care much less about the parameters that preserve the context.

3 Results

3.1 Recovering an Image

To test our implementation, we started with a training set consisting of one image with one mask placed in 100 different locations in the image, yielding 100 training samples. We then trained the model to regenerate the original image. This should be a trivial problem as the model will likely have seen every pixel unmasked. Thankfully, as shown in Figure 1, the recovered image was perfect.

The recovery was so good in fact that, as seen in Figure 2a, even when the mask covered the entire input, the 7 was fully recovered! This is clearly not the behavior we were looking for. It turned out the target image was imprinting itself onto the bias variables in the encoder's convolution layers. Once we removed these variables, the 7 went away. Figure 2b demonstrates to proper performance of the network.

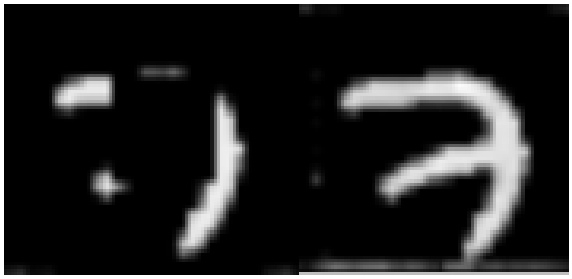
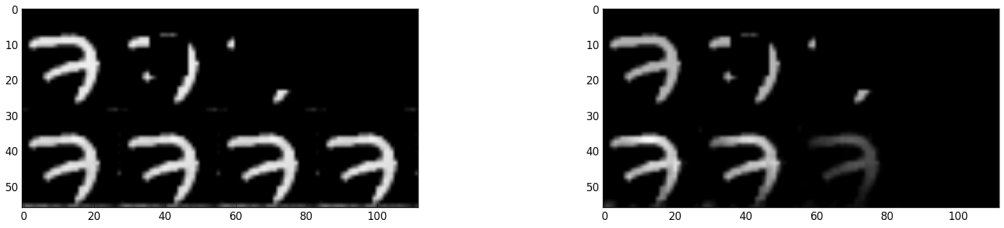


Figure 1: Perfect recovered 7 from a network exclusively trained on that 7.



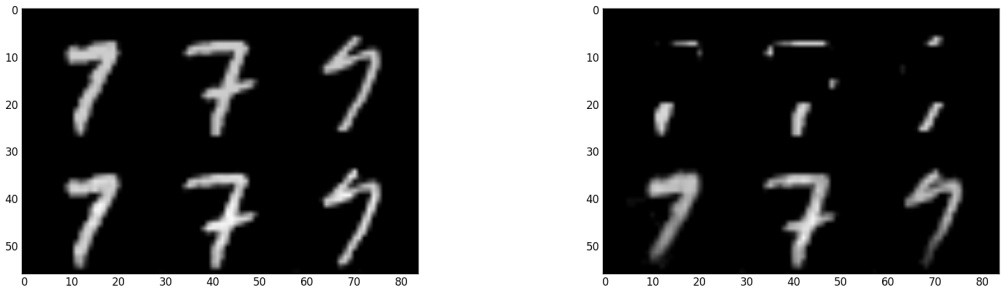
(a) With bias term.

(b) Without bias term.

Figure 2: Reconstruction of 7 with varying mask sizes by a context encoder trained on that one image.

3.2 Recovering a Number

We then tested to see if the model could learn to recover a single number. With a training set of 40 different handwritten 7s, we ran 200 iterations with batch size equal to the training set size. Masks were randomly generated for each iteration of each image.



(a) Mask size 0

(b) Mask size 12

Figure 3: Context encoder trained for inpainting 7s for 200 iterations.

The results were extremely encouraging. All the 7s, including the ones with a line in the middle, were correctly reconstructed with minimal blurring. The average loss per pixel in the missing region on the test set was less than 0.03. The black and white pixels have values ranging from 0 to 1, so that meant we achieved an average per pixel loss of less than 3%.

3.3 Recovering all Numbers

We were finally ready to inpaint for all numbers. Using twenty samples for each number in the training set, we trained this model for 500 iterations. Once again, batch sizes were the same as the training set size and random masks were generated as needed. The results are displayed below in Figure 4.

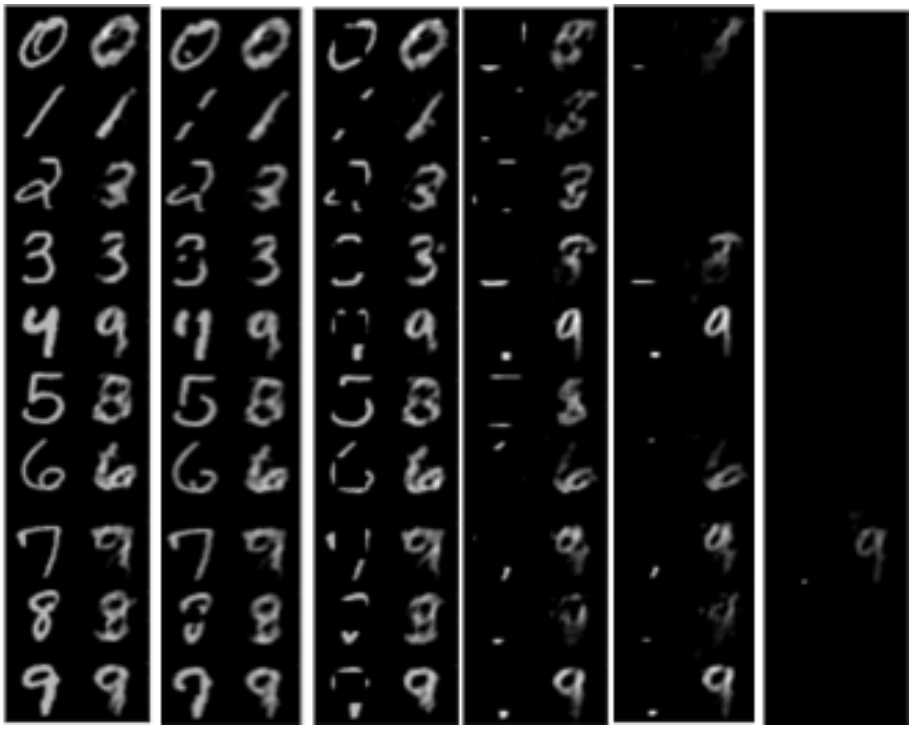
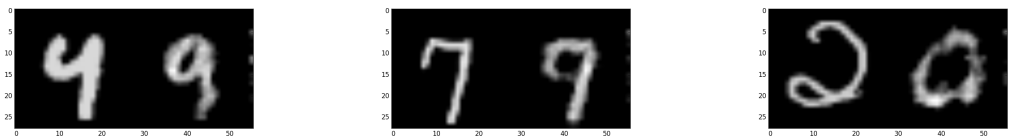


Figure 4: Context encoder trained for inpainting all digits for 500 iterations. Inputs are on the left. Outputs are on the right. Mask size increases from left to right.

The model does achieve some success in filling in the masks but is far from perfect. In the next few subsections, we share our observations and strategies for mitigating some of the problems that we encountered.

3.3.1 Determining Mask Position

As you can see in the first column of Figure 4, even when there is no mask, the reconstruction is not perfect. Common mistakes include 5s getting redrawn as 8s, and 4s and 7s turning into 9s. This is because the context encoder takes a masked digit as input and in many situations, an unmasked 4 is indistinguishable from a masked 9. Recall that when training, masks are not centered.



(a) 4 mistaken for 9 (b) 7 mistaken for 9 (c) 2 mistaken for 0

Figure 5: Unmasked digits mistaken for different masked digits.

In order to fix this, we need to give the neural network not just the masked image but also information regarding the location of the mask. This however, is deceptively challenging. Imagine two identical images. One is the result of a mask and the other is unmasked. They will pass through

the context encoder with the exact same activations. In order for both to recover their original images, knowledge of the location of the mask will need to trigger some activation otherwise would not have been triggered. Unfortunately, there does not seem to be an natural way to combine the mask and the masked image for the context encoder's CNNs.

A solution we did implement was to output a learned element-wise linear combination of the decoder reconstruction and the masked x. Training however, seemed to give a large amount of weight to masked x which marginally improved performance and did not address the underlying problem.

Our conclusion was that calling this a problem may have been misleading. As shown in our loss function, we train for reconstruction accuracy within a mask. Inaccuracies outside the mask will be copied over when we merge the masked x with our model's output and no one will be the wiser.

3.3.2 Mask Size vs Brightness

Figure 4 reveals another problem by demonstrating a correlation between the brightness of reconstructed images and mask size. The hypotheses gets darker because the decoder, which aggregates activations in the feature representation to reconstruct an image, receives fewer activations as more and more of the image gets covered up. This results in a dimmer image and although the dimness can be indicative of uncertainty, our goal is to construct a plausible hypothesis of the missing region. Predicted image brightness should be invariant to mask size.

One way we tackled this problem was by normalizing the prediction of context encoder just prior to it being sent to loss function. L2 loss generally causes neural networks to prefer the average of two possibilities, resulting in a grey image. By scaling the entire prediction so pixel max value was one, we forced the context encoder to rapidly improve its image reconstruction or face huge losses from blurry guesses. The normalization forced the encoder to improve at the task of reconstruction without hedging its bets with blurriness. The complete results are shown below in Figure 6.

Figure 6: Context encoder trained to output predictions that have their largest value normalized to 1. Outputs are on the right. Mask size increases from left to right. Predictions are much brighter and more confident.

3.3.3 Mixing Hypotheses

Optimizing this architecture boils down to a fight against blurriness. Normalization inadvertently mitigates the issue but Figure 7 reveals a deeper problem. When faced with an masked image

that could be one of two numbers, the model prefers to output an image that is a mix of the two numbers.

It is well known that the L2 loss function has a preference for average values and these average values could be causing the model to mix multiple hypotheses into one image. We however, believe that no pixel-wise loss function can solve the problem of mixing hypotheses. Between possible translations and different styles of numbers, numbers like 7 and 9 look too similar to create a loss function that penalizes one heavily enough to persuade a model to strongly prefer the other. The only way to solve this problem is to create a loss function that directly penalizes guesses that do not look like numbers. Thus we looked into adversarial loss.

Figure 7: Context encoder trained for 500 iterations on all digits is unable to decide between a 7 or a 9.

3.4 Adversarial Loss

Pathak implements adversarial loss based on Generative Adversarial Networks(GANs). GANs involve two neural networks trained simultaneously with alternating batch gradient descent. One network is a generator, our context encoder, and the other is a classifier. At every step, the classifier is presented with two images, one image is generated by the generator and the other is randomly sampled from the data. If the classifier correctly identifies the one produced by the generator, the generator receives negative feedback and the classifier receives a positive feedback. If the classifier chooses incorrectly, the signals are reversed. In this manner, the classifier gets better at distinguishing the generator's images and the generator gets better at fooling the classifier. The advantage of this approach is that we let a neural network learn and assign loss to the features that keep our model's output from looking realistic.

The loss from the GAN is weighted with the L2 reconstruction loss to give the total loss function

$$L = \lambda L_{rec} + (1 - \lambda) L_{adv} \tag{2}$$

where λ is a tuned hyperparameter.

Unfortunately due to time constraints, we could not implement this network.

4 Conclusions and Future Work

Image inpainting provides an interesting demonstration of a CNN's ability to extract semantic information. This has important applications in computer vision on problems like occlusion. Pathak already demonstrated the generalizability of the model on many different datasets and we had our own host of interesting problems studying MNIST. In the end, the generated images were not the crisp images we hoped for but we learned a great deal about the model and deconvolution, and had a great time coming up with solutions to the MNIST specific problems we identified.

If we continue this project, we would like to explore the effect of pooling on the network. We mentioned in passing that pooling is detrimental to performance, but it also decreases the number of parameters. It would be interesting to see how much pooling is possible while still producing acceptable outputs. We would also of course, fully implement adversial loss.

Extending the model to datasets with larger images with more computing power is always interesting problem but a real extension of this project would be investigating the model's potential application to high resolution images without incurring the corresponding computational cost. Conceivably, one neural network could learn characteristics and features that generalize well in high resolution images while as second neural network learns a feature representation of the

image in terms off the learned characteristics. The network would decode using the same learned characteristics and the whole model could be trained using masks and adversarial loss. Learning this feature representation would be an interesting way of bringing high dimensional images down to manageable size.

5 Division of Labor

Jeffrey Hu implemented the convolutional neural network for the encoder decoder scheme and optimized the model architecture. Jose Velarde wrote the code that manipulated and masked the images, and evaluated the results. The problem solving and the writing of this paper was split evenly.

References

[1] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. Efros. *Context Encoders: Feature Learning by Inpainting.* In CVPR, 2016.