Pixel Recurrent Neural Networks

Heramb Pendyala, Pratheek Keerthi, Akshata Chalke
Department of Computer Science
Old Dominion University
Norfolk, VA 23508
hpend001@odu.edu, pkeer001@odu.edu, achal001@odu.edu

Abstract

The paper combines a variety of techniques to generate images. The models introduced in the paper improve the image generation using combination of new techniques and models.

Pixel-RNN uses a novel architecture with recurrent layers and residual connections which helps predicts pixels of an image, across its vertical and horizontal axes. The architecture also models the joint distribution of pixels as a product of conditional distributions of horizontal and diagonal pixels. The model achieves state-of-the-art in the generation of natural images.

1 Introduction

The main problem addressed in this paper was to build a generative model that is scalable, easy to handle and provides reliable results for any kind of task in hand. Generative models are a subset of unsupervised learning and are a powerful way of learning from any kind of data distribution. Generative models aim to learn from the real data distribution and be able to generate new data. Three of the most commonly used efficient approaches are Variational Autoencoders(VAE), Generative Adversarial Networks(GAN) and Auto-regressive Models. VAE used to encode an input image to a much smaller dimensional representation which can store latent information about the input data. But a disadvantage of VAE is that it generates blurry images. On the other hand, GAN generates very sharp images when compared to VAEs. GANs don't work with any density estimation like VAE. Instead, it is based on an approach to find the Nash equilibrium between the networks. However, the disadvantage of GAN is that they are very hard to train. The paper uses autoregressive models, they have introduced a deep neural network that predicts the pixels of an image.

Some of the advantages of Auto-regressive models are: 1. These models have the advantage of returning explicit probability densities (unlike GANs), making it straightforward to apply in domains such as compression and probabilistic planning and exploration 2. Training a GAN requires finding the Nash equilibrium. Since there is no algorithm present which does this, training a GAN is unstable as compared to PixelRNN or PixelCNN. 3. It's hard to learn to generate discrete data for GAN, like text.

This method models the probability of every pixel and encodes the complete set of dependencies in an image. All the samples generated by this model are sharp ad coherent. The resulting pixelRNN is composed of up to twelve, fast two-dimensional LSTM layers. These LSTMs are of two types, the first one is the Row LSTM and the second one is the Diagonal BiLSTM.

2 Models Proposed

These are the models proposed in the paper:

1. PixelCNN
2. PixelRNN
    - Row LSTM
    - Diagonal BiLSTM
3. Multi-scale PixelRNN

Pixel Recurrent Neural Networks uses many different techniques to generate a natural-looking image. It models the distribution of image data using techniques that include LSTM cell, and sequentially infer the pixels in an image to generate an image or complete a given incomplete image.

2.1 Pixel Generation

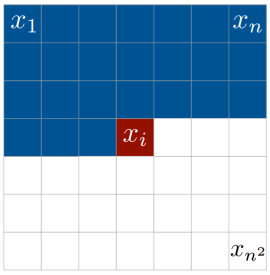

Figure 1: Pixel generation

The image is traversed from left-to-right and top-to-bottom along with the image. The intensity value of the pixel that is being generated is dependent on the pixels before it. In the above $N * N$ image the intensity of xi is based on the conditional probabilities of all the previous pixels.

$$x = p(x_i|x_1, x_2, ..., x_i - 1)$$

We calculate the joint probability of an image x by multiplying all conditional probabilities of the image together, as so:

$$p(x) = \prod_{i=1}^{n} p(x_i|x_1, x_2, ..., x_i - 1)$$

The probability of an image x is a conditional probability of all the pixels. We learn these conditional probabilities through a series of special convolutions that capture this context around a given pixel. First, we sample the first pixel and this is sent through the RNN as an input and then it will predict the next pixel. This procedure is iterated several times until we get all the pixels.

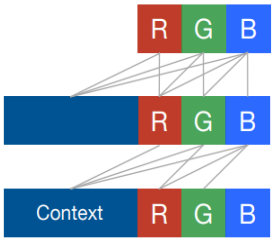

Figure 2: Pixel generation with color channels

We know that an image is composed of pixels and every pixel has a color. These colors are nothing but different compositions of colors Red, Green and Blue. While generating an image it is very important to keep this color composition in mind. For the same reason, the pixel generation uses three channels (Red, Green and Blue). The pixels on every channel are generated one at a time. Each pixel xi is dependent on the pixel value before it and the pixel value on the channel before it. We can rewrite the above distribution formula as:

$$p(x_i, R|x < i)p(x_i, G|x < i, x_i, R)p(x_i, B|x < i, x_i, R, x_i, G)$$

While training and evaluation the pixel values distribution is computed in parallel and image generation is done in a sequence.

2.2 Model Architecture

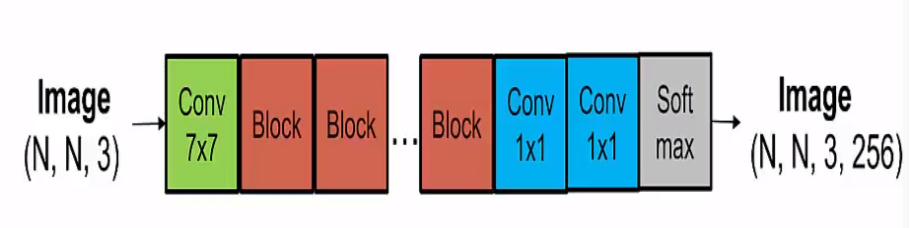

Figure 3: Typical model architecture

The above image shows the typical architecture used for all the models in this paper. We start off with an image of height = N, width = N and three channels (RGB). Then it goes through a 7x7 convolution layer followed by several residual blocks which can be convolution layers or LSTMs based on the model we are working on. Then it goes through two 1x1 convolutional layers and the final output is a softmax layer, it has an image of the same size as the input 256 values for each channel.

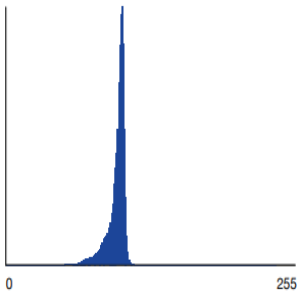

Figure 4: Softmax distribution

The model has the softmax layer as its last layer. The softmax layer has 256 units, each representing one of the possible values that pixel can have from 0 to 255. Each value represents the probability of generating that pixel's intensity. The model then chooses the maximum value of this distribution and choose that value as the pixel intensity. Since we can now know the conditional probability of our pixel value, to get the appropriate pixel value we use a 256-way softmax layer. Output of this layer can take any value ranging from 0–255 i.e our pixel value can vary from 0–255.

2.3 PixelRNN

RNNs have been shown to be extremely efficient in handling sequence problems. An effective approach to model pixel recurrent neural network is by using probabilistic density models (like Gaussian or Normal distribution) to quantify the pixels of an image as a product of conditional distributions. This method turns the modeling problem into a sequence problem where the next pixel value is predicted based on all the previously generated pixel values. To process these non-linear and long term dependencies between pixel values and distributions we need an expressive sequence model like Recurrent Neural Network(RNN).

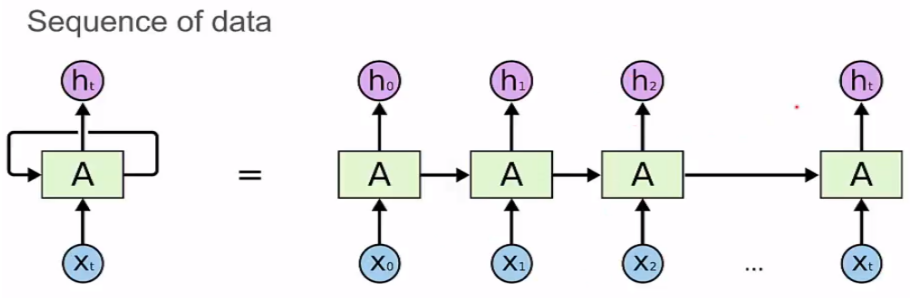

Figure 5: Pixel recurrent neural network

RNN is good with the sequential data – in our case the input that we pass through RNN is sequence of pixels that make up the image.

X0,X1,...,Xt is the input that goes through the RNN (A in picture) and generates an output which is the hidden state vectors represented as h0,h1. . . ht. These outputs are used further for the production of pixels. Hidden states represent all pixels that come before it. For example h2 represents pixels x2, x1 and x0.

2.3.1 Image generation in PixelRNN

First we sample a pixel, then the sample pixel is sent through the RNN which generates hidden state, from the hidden state we add on a soft max layer of 256 and then it predicts next pixel. Similarly we send this generated pixel as input

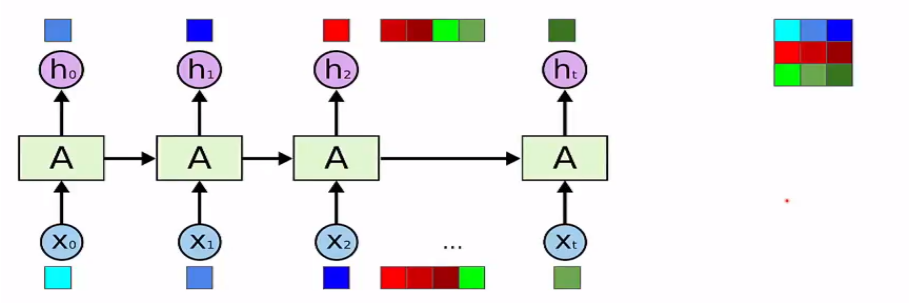

Figure 6: Image generation using PixelRNN

to the next layer and this process goes on until you generate a vector with all the pixels and at the end we just reshape those pixels to make up an image.

However, pixelRNNs can not efficiently handle long-range dependencies. So they introduce LSTMs. The resulting RNNs are composed of up to twelve, fast two-dimensional Long Short-Term Memory (LSTM) layers. These layers use LSTM units in their state and adopt a convolution to compute at once all the states along one of the spatial dimensions of the data. LSTM networks adds gates to RNN so that it can have a long term memory. LSTM has 4 gates for control to see how much information is allowed through.

$$i = \sigma(x_i U^i + h_{i-1} W^i)$$

$$f = \sigma(x_i U^f + h_{i-1} W^f$$

$$o = \sigma(x_i U^o + h_{i-1} W^o$$

$$g = tanh(x_i U^g + h_{i-1} W^g$$

And states ci and hi hold the information about all the time steps up till now.

$$C_i = C_{i-1} \emptyset f + g \emptyset i$$

$$h_i = tanh(C_i) \emptyset O$$

As LSTM is a very computationally expensive task, The paper tries to enhance the parallelization in the LSTM by computing input-to-state component for entire 2-dimensional map, for this a convolution is used to follow row-wise orientation of LSTM itself. This convolution is masked to include only valid context. so the new hidden and cell states are obtained as below:

$$[\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] = \sigma(\mathbf{K}^{ss} \circledast \mathbf{h}_{i-1} + \mathbf{K}^{is} \circledast \mathbf{x}_i)$$
$$\mathbf{c}_i = \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{i}_i \odot \mathbf{g}_i$$
$$\mathbf{h}_i = \mathbf{o}_i \odot \tanh(\mathbf{c}_i)$$

2.3.2 Row LSTM

Hidden state(i,j) = Hidden state(i-1,j-1)+ Hidden state(i-1,j+1)+ Hidden state(i-1,j)+ p(i,j)

The first layer in this model is a 7x7 convolution that uses the mask of type A. It is followed by the input to state layer which is a 3x1 convolution that uses mask of type B and a 3x1 state to state convolution layer which is not masked. The feature map is then passed through a two 1x1 convolution layers consisting of ReLU and mask type B.

The final layer of the architecture is the 256-way softmax layer. The model processes the image row by row and top to bottom computing features of the entire row all at once. It only captures a triangular region above the pixel. However it isn't able to capture the entire previously generated region.

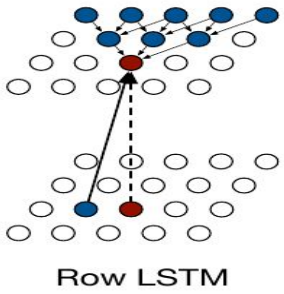

Figure 7: Row LSTM receptive field

2.3.3 Diagonal BiLSTM

LSTM cells used by the main variant of PixelRNNs capture this conditional dependency across dozens or hundreds of pixels. The paper has implemented a novel spatial bi-directional LSTM cell, the Diagonal BiLSTM, to capture the desired spatial context of a pixel.

The receptive field of this layer encompasses the entire available region. The processing goes on diagonally. It starts from the top corner and reaches the opposite corner while moving in both directions.

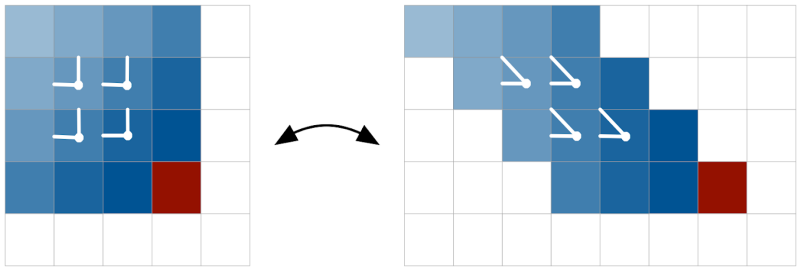

Figure 8: Diagonal BiLSTM receptive field

To aid in capturing context before the first layer of the network, we mask the input image so that for a given pixel xi we are predicting, we set the values of all pixels yet to be traversed, xj,ji,xj,ji, to 0, to prevent them from contributing to the overall prediction. In subsequent LSTM layers, we perform a similar mask, but no longer set xi to 0 in the mask. We then skew the image, so that each row is offset by one from the row above it, as shown above. We can then perform a series of k x 1 convolutions on the skewed image using the Diagonal BiLSTM cells.

This enables us to efficiently capture the preceding pixels in the image to predict the upcoming one. LSTM cells also capture a potentially unbounded dependency range between pixels in their receptive field. However, this comes at a high computational cost, as the LSTM requires "unrolling" a layer many steps into the future. Residual connections (or skip connections) are also used in these networks to increase convergence speed and propagate signals more directly through the network.

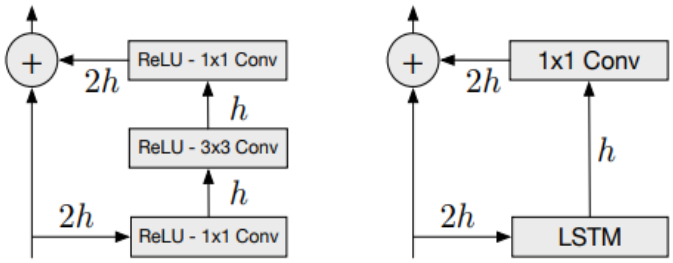

Figure 9: Residual blocks for a PixelCNN (left) and PixelRNNs.

2.4 Masked convolutions

The features for each input position in every layer are split into three parts, each one corresponding to a color(RGB). Masks are an important part of network

which maintain the number of channels in the network. For computing the values of G channel we need the value of the R channel along with values of all previous pixels.

Similarly, B channel requires information of both R and G channels. To restrict the network to adhere to these constraints we apply masks to convolutions.We use two types of masks :

1. Type A: This mask is only applied to the first convolutional layer and restricts connections to those colors in current pixels that have already been predicted.

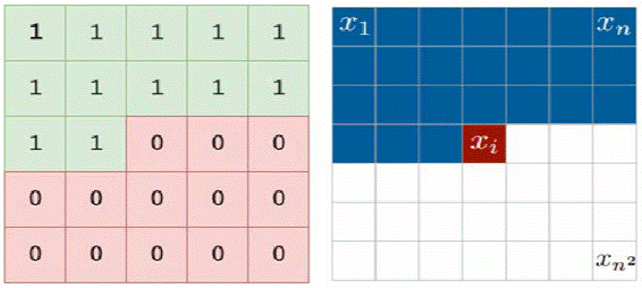

Figure 10: Masking

2. Type B: This mask is applied to other layers and allows connections to predicted colors in the current pixels.

2.5 PixelCNN

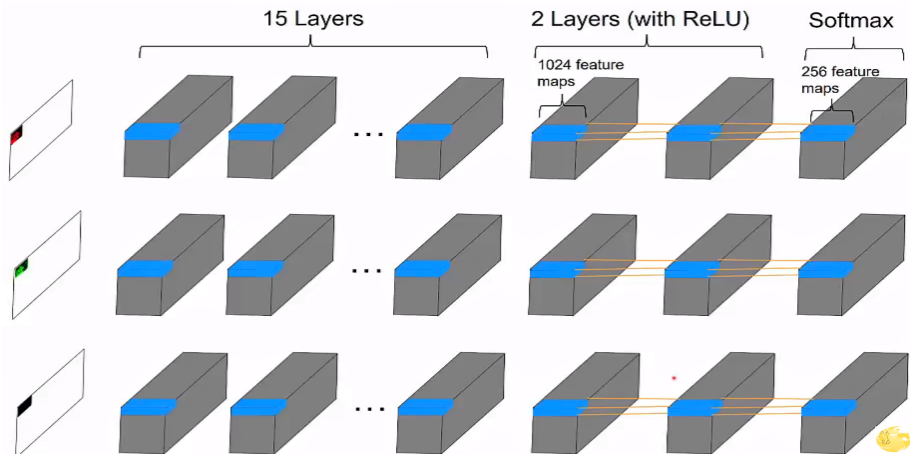

Figure 11: PixelCNN architecture

PixelCNNs are very easy to train when compared to PixelRNNs, PixelRNN is very slow as each state needs to be computed sequentially. This can be overcome by using convolutional layers and increasing the receptive fields. PixelCNN includes convolutional layers that capture a bounded receptive field and computes the features for all the pixel positions at once. Masks are introduced in this model to stop the model from violating the conditional dependencies.

In the PixelCNN architecture, the first layer is a 7x7 convolution that uses mask A and the rest are 3x3 convolution that uses mask B. Then the feature maps generated in the first layer are passed through several layers consisting of ReLU activation and 1x1 convolution The final layer of this is a 256-way softmax layer.

Though the training process is faster in PixelCNN, it is not feasible for generating an image. As image generation in PixelCNN is very slow and it has a very small receptive field. PixelCNN architecture :

2.6 Project Implementation

A faster method to solve the problem faced by the above model would be, replacing the LSTM cell with a series of convolutions to capture a large and bounded receptive field. We can implement the convolution operation by using the masks as needed:

Implementing Model

- Setting up hyperparameters
- Setting up Network (convolutional layers)
- Optimization(RMSProp)
- Training model

We have used the weights initialization scheme to create the convolution kernel. After that, a mask is applied to the image to restrict the focus of the kernel to the current context. Finally, convolution is applied to the image with an optimal activation function.

2.6.1 Datasets

The model is trained on the MNIST dataset, then the model is trained

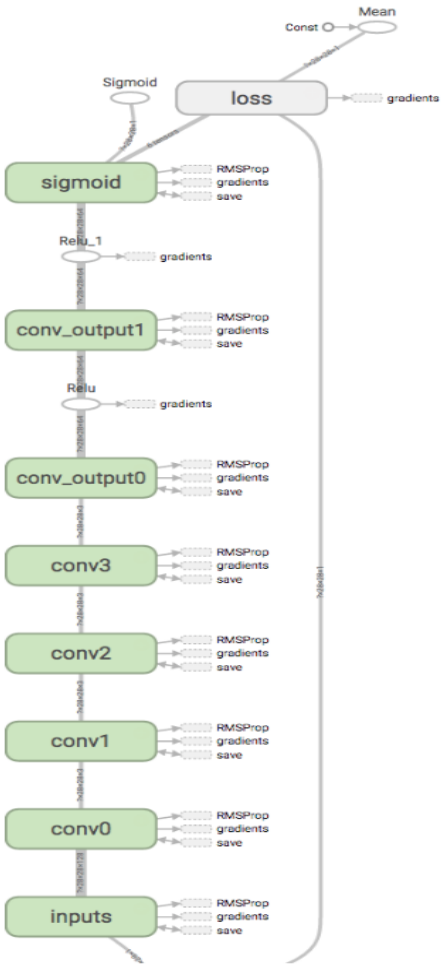

Figure 12: model architecture

to generate the handwritten digits that don't appeain the dataset. The dataset was downloaded from http://yann.lecun.com/exdb/mnist/ The pixelRNN is also modeled to complete the partially occluded images by predicting the rest of the pixels.

2.6.2 Features of the Network

Let's construct the PixelCNN model. First, we set up input placeholder. We'll be feeding batches of training images into the model through this.

Next, we construct the masked convolutional layers. These layers apply a series of convolutions to the image, where each filter is masked to only account for pixels in the region of interest. These are the pixels above and to the left of the pixel in the center of the mask, which follows the PixelRNN generative model assumptions. Also of note is that the receptive field of the PixelCNN model grows linearly with the depth of these convolutional stacks.

Along with the convolutional layer, PixelRNNs also uses residual connections. These connections copy the output from the previous layer and attach it with the output of the deeper layer. This helps in preserving the information learned from all the previous layers in the model. Figure 12, shows the model architecture that was implemented. For our model, these residual connections looks something like this :

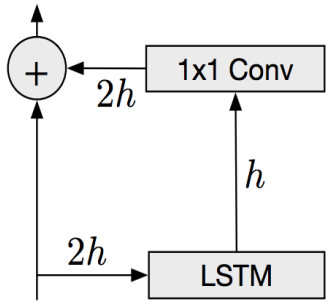

Figure 13: residual block for pixelRNN

These residual connections allow the model to increase the depth and gain accuracy and keeps making the model easy to optimize at the same time. The last layer applies a sigmoid activation function to the input. The output of this layer is a value between 0 and 1 this value is nothing but the normalized pixel intensity. The final architecture of our model looks like this. We can construct the network using this architecture and the convolution operations.

Class pixelRNN

Get inputs and reshape them(batch size = 100)
Add 1 7x7 convolution Layer(transform input to vector)
Add Hidden layers (3 with 1x1 convolution layer)
Add Output layer (2 1x1 convolution layer)
Add a Sigmoid Layer(generate pixel based on distribution)
return pixel values

Here a 7x7 convolution is being applied to the image with the initial mask A which helps remove the self-connection to the pixel that is being connected. Next, a series of 1x1 convolutions are applied to the image. After this a series of 1x1 convolutions using ReLU are constructed.
Finally, a final convolution is applied with a sigmoid activation function this produces a series of pixel predictions for the image.

input of main recurrent layers:

-Generate 7x7 convolution layer with mask A
-Returns feature maps for the image pixel
-Generate 1x1 convolution with mask B (input feature maps from 7x7 convolution layer)
-Output the final convolution
-Pass the output through sigmoid activation function

2.6.3 Training Procedure

hyperparameters =

size of a batch = 100
dimesion of hidden states of LSTM or Conv layers = 16
length of LSTM or Conv layers = 7
dimesion of hidden states of output Conv layers = 32
the length of output Conv layers = 2

For training the network, we are supplying mini-batches of images and predicting each pixel in parallel using out network using the hyperparameters mentioned above. We minimized the cross entropy between our predictions and binary pixel values of the images. This objective is optimized using an RMS optimizer with a learning rate of 0.001, selected using grid search. We use a batch size of 100 and 16 hidden units for each convolution.

The training time for MNIST dataset on a normal quad-core PC with a RAM of 8GB, for pixelCNN was around 4 hrs, and 20 mins on a RAM of 16 GB.

2.6.4 Image generation and occlusion completion

After training, the model is used to generate sample images using generative model. And the model is also able to infer the remaining pixel values of an occluded image. we have a method that generates an occluded image given a normal image and another method that is called when we need to generate a complete image from an occluded one. The training and generating methods are similar to model.fit and model.transform from scikit-learn library.

```
def generate(occluded images, dimensions):
    Images = run CNNmodel.predict(occluded images, dimensions)
    return Images

def generate occlusions(images, dimensions):
    Generate a sample of occluded image
    Mask the remaining pixels from height/2 to 0s.
    return samples
```

Figure 14: Input image and output image

The above code can successfully finish an occluded image. The output still has a lot of discrepancies in the generated images.

2.6.5 Experimentation and results

From the above algorithms and methods we were able to build a pixelRNN model, that was able to complete the occluded images that were passed as an input to the model. Below are some of the images.

From our implementation using PixelRNN, we have generated a training loss of 0.133 and testing loss of 0.134 after 20 epochs.

Figure 15: The first image on the top-left corner is the occluded image(input), the other three images are the outputs after certain epochs

Accuracy - Train : 86.7, Test : 86.6

2.6.6 Conclusion

This paper helped us understand the concepts of generative models like image generation, pixel recurrent neural networks, convolution neural networks, LSTMs etc and how they are implemented. We have used python for coding and a framework called Tensorflow. Though it already has packages readily available to do some of the tasks required for our project, we implemented them from scratch to have a better understanding of the models.

We have implemented a faster and a simple version of PixelRNN architecture called PixelCNN which just relies on series of masked convolutions.The model gives us a testing accuracy of 86.6 and the paper has a accuracy of 81.30 for pixelCNN.

2.6.7 Future Scope

As we were only able to create one of the proposed models, we would like to learn more about generative models and try to implement other models like

ROW LSTM, diagonal LSTM. This model was implemented using a very simple data-set that can learn the distribution of MNIST images. In future, we would like this model to work with images compose of multiple color channels, like CIFAR10 and imageNet.

The implementation of other models like Row LSTM and Diagonal BiLSTM is computationally expensive even on a state of art GPU. Convolution based architectures run approximately 20 times faster that the Diagonal BiLSTM.

References

- https://arxiv.org/pdf/1601.06759.pdf
- https://towardsdatascience.com/auto-regressive-generative-models-pixelrnn-pixelcnn-32d192911173
- https://gist.github.com/shagunsodhani/e741ebd5ba0e0fc0f49d7836e30891a7
- https://mandroid6.github.io/2018/01/20/PixelRNN-CNN/
- https://github.com/singh-hrituraj/PixelCNN-Pytorch