Deep Recurrent Attentive Writer
CmpE597 Final Project

Mine Melodi Çalışkan [1]
Master Student
Department of Computational Science and Engineering
Bogazici University
Bebek, Istanbul 34432 Turkey

May 28, 2018

[1] minemelodicaliskan@gmail. com

Introduction

In this project deep recurrent attentive writer (DRAW) [1] is implemented on *Quick!, Draw* data set [2]. Draw network is a recurrent auto encoder that uses attention mechanisms. Attention mechanism focus on a small part of the input data in each time step, and iteratively generates image that is closer to the original image. The network is trained with stochastic gradient descent and the objective function is variational upper bound on the log likelihood of the data.

Architecture

Draw network augments the encoder and decoder with recurrent networks. The encoder determines a distribution over latent variables that contains salient information about the input data and the decoder takes samples from this distribution and uses them to condition its distribution. The inference and generation defined by a sequential process. An attention mechanism is added over the input to define this sequential process. Attention mechanism restricts the input region read by the encoder and the output region written by the decoder. LSTMs are used for the encoder and decoder. To generate images, samples are picked from the latent layer based on a prior. These samples are then fed into the decoder. That is repeated for several time steps until the image is finished.

Data Generation

The encoder receives at each time step the image and the output of the previous decoding step. The hidden layer in between encoder and decoder is a distribution $Q(Z_t|h_t^{enc})$ which is a diagonal gaussian. The mean and standard deviation of that gaussian is derived from the encoder's output vector with a linear transformation. Using a gaussian instead of a bernoulli distribution enables the use of the reparameterization trick [4]. The reparametrization trick enables backpropagation algorithm straightforward. At every time step encoder and decoder generates a vector, and the vector generated by the decoder is added to image *canvas*. Fig. 1 show the architecture of the Draw.

Training steps can be summarized as following steps:

1. Encoding:
   (i) $\hat{x} = x - \sigma(c_{t-1})$
   (ii) $r_t = read(x_t, \hat{x}_t, h_{t-1}^{dec})$
   (iii) $h_t^{enc} = RNN^{enc}(h_{t-1}^{enc}, [r_t, h_{t-1}^{dec}])$
2. Sampling: $z_t \sim Q(Z_t|h_t^{enc})$
3. Decoding: $h_t^{enc} = RNN^{dec}(h_{t-1}^{dec}, z_t)$
4. Output: $c_t = c_{t-1} + write(h_t^{dec})$

where $\sigma(x)$ is the logistic sigmoid function and the latent distribution is taken as a diagonal gaussian $\mathcal{N}(Z_t|\mu_t, \sigma_t)$ where

$$\mu_t = W(h_t^{enc})$$
$$\sigma_t = \exp^{W(h_t^{enc})}$$

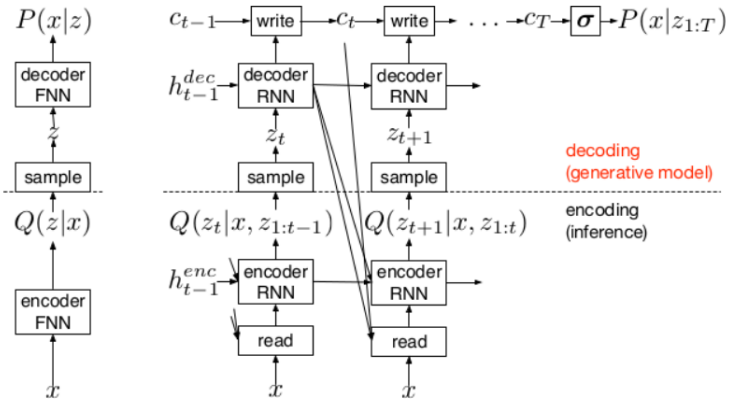


Figure 1: Architecture of the Draw. [1]

Attention

In draw a selective attention model is used to decide which parts to read of the image. These are called *glimpses*. Each glimpse is defined by its center (x, y), its stride δ, its gaussian variance σ^2 and a scalar multiplier γ. Stride value determines zoom level, scalar multiplier scales the intensity of the glimpse results and the higher the gaussian variance the more blurry is the result. These parameters are calculated based on the decoder output using a linear transformation. For an $N \times N$ glimpse it takes $N \times N$ grid of gaussian filters. The center pixel of each gaussian is then used as the respective output pixel of the glimpse. The glimpse parameters are generated from the decoder output in both cases.

Given an image of size $A \times B$, we place $N \times N$ grid of gaussian filters positioned on image with grid center (g_X, g_Y) with stride δ and the mean location of the filter is given as:

$$\mu_X^i = g_X + (i - \frac{N}{2} - 0.5)\delta \tag{1}$$

$$\mu_Y^i = g_Y + (i - \frac{N}{2} - 0.5)\delta \tag{2}$$

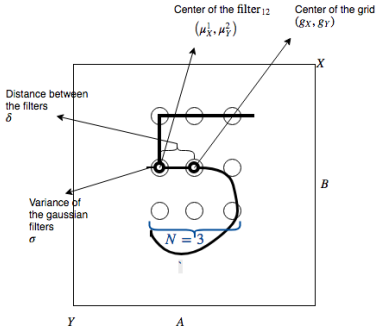


Figure 2: $N \times N$ grid of gaussian filters.

The read attention parameters are determined by a linear transformation of h_{dec} as follows:

$$(\tilde{g}_X, \tilde{g}_Y, \log \sigma^2, \log \tilde{\delta}, \log \gamma) = W(h^{dec})$$

Here log terms of σ, γ computed to guarantee $\sigma = \exp(\log(\sigma))$ and $\sigma = \exp(\log(\gamma))$ to be positive.

$$g_X = \frac{A+1}{2}(\tilde{g}_X + 1)$$

$$g_Y = \frac{B+1}{2}(\tilde{g}_Y + 1)$$

$$\delta = \frac{\max(A, B) - 1}{N - 1}\tilde{\delta}$$

where A, B are width and height of the image respectively.

Using the parameters above and normalization factors Z_X and Z_Y, the horizontal and vertical filter bank matrices are computed.

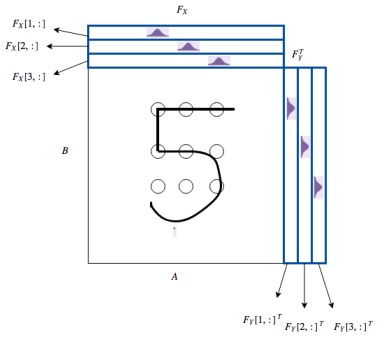
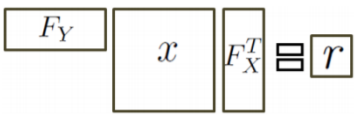


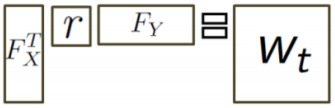
Figure 3: Horizontal and vertical filters.

1. Horizontal filter bank matrix:

$$F_X[i, a] = \frac{1}{Z_X} \exp(-\frac{(a - \mu_X^i)^2}{2\sigma^2})$$



(a) Read attention.



(b) Write attention.

2. Vertical filter bank matrix:

$$F_Y[j,b] = \frac{1}{Z_Y}\exp(-\frac{(b-\mu_Y^j)^2}{2\sigma^2})$$

In the above equations (i,j) is a point in attention patch, and (a,b) is a point in input image. The extracted value for attention grid pixel (i,j) is the convolution of the image with a $2D$ gaussian whose x and y components are the respective i and j rows in F_X and F_Y.

Read attention and write attention can be formulated as follows:

1. Read: $read(x,\hat{x},h_{t-1}^{dec}) = \gamma[F_Y x F_X^T, F_Y \hat{x} F_X^T]$
2. Write: $write(h_t^{dec}) = \frac{1}{\hat{\gamma}}[\hat{F}_Y^T w_t \hat{F}_X]$

Loss Functions

Generative Loss

The loss function of the decoder is the negative log likelihood of the image given in the final canvas matrix : $-\log D(x|c_t)$, where x is the input image and c_t is the final output image of the autoencoder. D is a bernoulli distribution if the image is binary.

Latent Loss

The loss function of the latent layer is the Kullback-Leibler (KL) divergence [5] between that layer's gaussian distribution and a prior, summed over the timesteps.

$$L_z = \sum_{t=1}^{T} KL(Q(Z_t|h_t^{enc})\|P(Z_t))$$

where Z_t = latent layer, h_t^{enc} = result of encoder and a prior $P(Z_t)$.

The KL-divergence loss term measures the difference between the distribution of the latent vector z, to that of an independent and identically distributed gaussian vector with zero mean and unit variance. Optimizing for this loss term

Figure 5: Sample figure of Quick! Draw data set.

allows to minimizing following difference. Assuming $P(Z_t)$ to be $\mathcal{N}(0,1)$, and using the result in [4]:

$$L_z = \frac{1}{2}\left(\sum_{t=1}^{T} \mu_t^2 + \sigma_t^2 - \log \sigma_t^2\right) - \frac{T}{2}$$

Total Loss

The total loss is the expectation value of the sum of latent and generative loss.

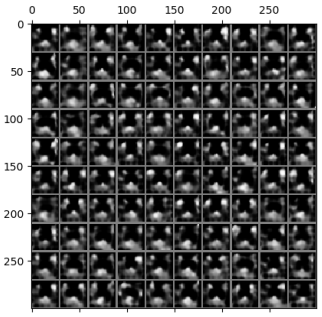
Experiments

Data

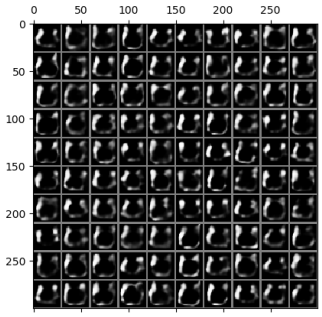
Experiments are based on two categories and their mix from Quick! Draw dataset [2] which is constructed from 50 million vector drawings obtained from Quick, Draw! [3], an online game where the players are asked to draw objects belonging to a particular object class in less than 20 seconds. Fig.5 shows some samples from the data set. For the model I use preprocessed version as Numpy bitmaps, and then converted each image into MNIST data set format.

Results

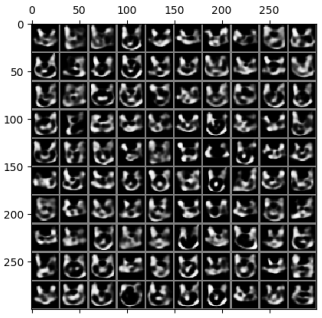
The results were reported in Table 1. Maximum iteration step and initial learning rate are fixed to 10^4 and $1e-3$ respectively. As an optimizer RMSprop is used, and gradients are clipped to avoid overshooting. Loss results are calculated on data set both the reconstruction loss and the latent loss. So, for cat category decreasing batch size from 100 to 64 improved the reconstruction score from 166.68 to 146.50. We observed that increasing the fixed number of time steps for image generation T usually gives better results, e.g for dog data set the first result using $T = 10$ was 171.14 for reconstruction loss, doubling T decreased this error sufficiently so that generated sequential image looks similar to the original image. Fig. 7 shows 4 generation steps for dog images. It can be observed from Fig. 8a that this modification also increased the speed of



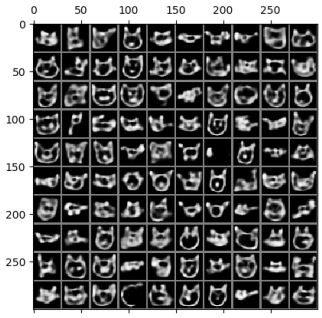
(a) Data generation at time step t = 4.



(b) Data generation at time step t = 6.



(c) Data generation at time step t = 7.



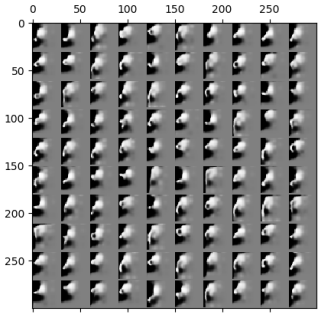
(d) Data generation at time step t = 9.

Figure 6: Generated samples for cat category in Quick! Draw data set.

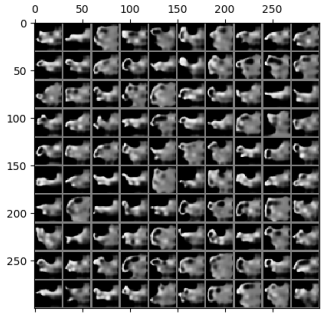
learning. Smaller patch size for read attention decreased the performance, an example result can be seen in Table 1 for “Cat & Dog” category.

Conclusion

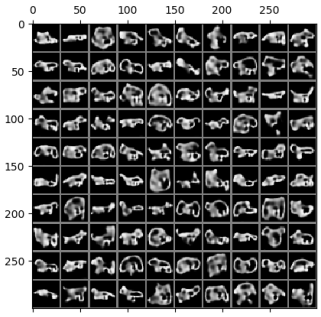
In this project we implemented a recurrent neural network model for image generation, DRAW for Quick! Draw data set. We applied solutions to the difficulties we encountered by tuning hyper parameters of the model so that final results show promising results in the sense of “ natural” sequential image generation and also with respect to final reconstructed image.



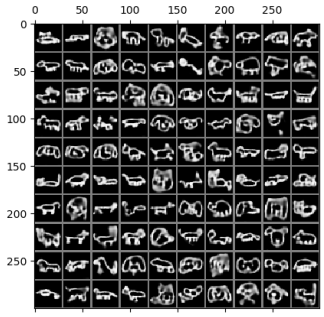
(a) Data generation at time step t = 4.



(b) Data generation at time step t = 8.

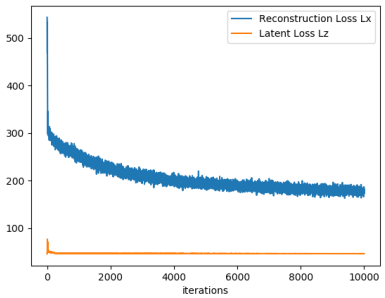


(c) Data generation at time step t = 15.

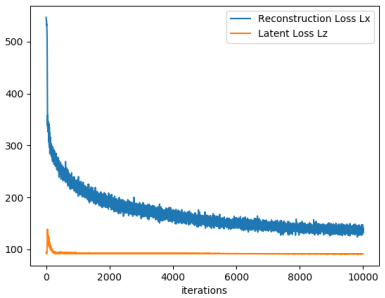


(d) Data generation at time step t = 20.

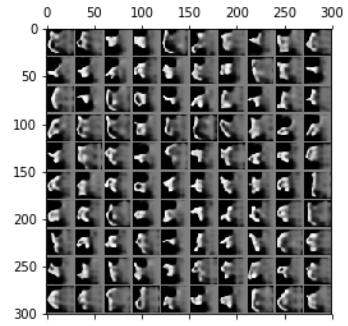
Figure 7: Generated samples for dog category in Quick! Draw data set.



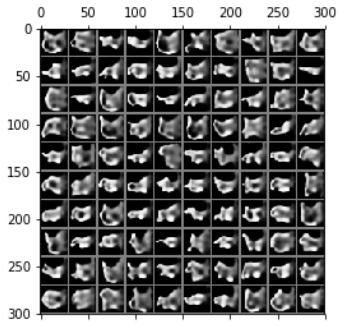
(a) Reconstruction and latent loss using 10 iteration for dog category during training.



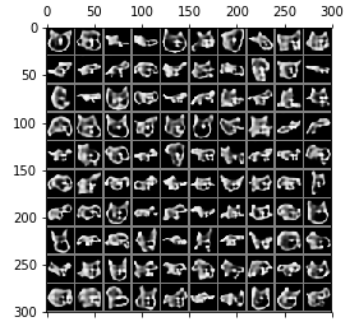
(b) Reconstruction and latent loss using 20 iteration for dog category during training.



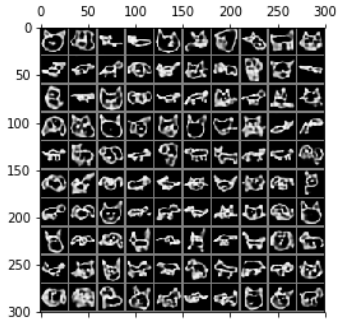
(a) Data generation at time step t = 7.



(b) Data generation at time step t = 9.



(c) Data generation at time step t = 15.



(d) Data generation at time step t = 20.

Figure 9: Generated samples for cat and dog categories together in Quick! Draw data set.

Category	Lx	Lz	read	write	z size	T	batch
Cat	**146.501862**	92.220596	5	5	10	20	64
Cat	166.688004	91.576004	5	5	10	20	100
Dog	171.142306	92.418405	5	5	10	10	100
Dog	**158.912208**	91.2177324	5	5	10	20	100
Cat& Dog	181.984634	91.815247	2	5	10	20	100
Cat& Dog	**141.932007**	91.562057	5	5	10	20	100

Table 1: Experimental Hyper-parameters and Test Losses.

Appendices

Source code

```
import tensorflow as tf
from tensorflow.python.client import device_lib
import os
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
tf.app.flags.DEFINE_string('f', '', 'kernel')
flags = tf.app.flags


tf.flags.DEFINE_string("folder", "", "")
FLAGS = tf.flags.FLAGS
```

Data Preprocessing

```
class Dataset:

    def __init__(self,data):
        self._index_in_epoch = 0
        self._epochs_completed = 0
        self._data = data
        self._num_examples = data.shape[0]
        pass


    @property
    def data(self):
        return self._data

    def next_batch(self,batch_size,shuffle = True):
        start = self._index_in_epoch
        if start == 0 and self._epochs_completed == 0:
            idx = np.arange(0, self._num_examples)
            np.random.shuffle(idx)
             # get list of `num` random samples
            self._data = self.data[idx]

        # go to the next batch
        if start + batch_size > self._num_examples:
            self._epochs_completed += 1
            rest_num_examples = self._num_examples - start
            data_rest_part = self.data[start:self._num_examples]
            idx0 = np.arange(0, self._num_examples)  # gets all
                possible indexes
            np.random.shuffle(idx0)
            self._data = self.data[idx0]

            start = 0
            self._index_in_epoch = batch_size - rest_num_examples #
                avoid the case where the #sample is not integer
                factor of the batch size i
            end =  self._index_in_epoch
            data_new_part =  self._data[start:end]
            return np.concatenate((data_rest_part, data_new_part),
                axis=0)
        else:
```

```
            self._index_in_epoch += batch_size
            end = self._index_in_epoch
            return self._data[start:end]
```

Train test split

```
import numpy as np
from sklearn.model_selection import train_test_split
import h5py
mypath = "cat/"
txt_name_list = []
for (dirpath, dirnames, filenames) in os.walk(mypath):
    if filenames != '.DS_Store':
        txt_name_list.extend(filenames)
        break
x_train = []
x_test = []
y_train = []
y_test = []
xtotal = []
ytotal = []
slice_train = int(80000/len(txt_name_list))  #setting value to be
    80000 for the final dataset
i = 0
seed = np.random.randint(1, 10e6)

##Creates test/train split with quickdraw data
for txt_name in txt_name_list:
    txt_path = mypath + txt_name
    xx = np.load(txt_path)
    try:
        xx = xx.astype('float32') / 255.    ##scale images
    except AttributeError:
        pass
    try:
        y = [i] * len(xx)
        np.random.seed(seed)
        np.random.shuffle(xx)
        np.random.seed(seed)
        np.random.shuffle(y)
        xx = xx[:slice_train]
        y = y[:slice_train]
        if i != 0:
            xtotal = np.concatenate((xx,xtotal), axis=0)
            ytotal = np.concatenate((y,ytotal), axis=0)
        else:
            xtotal = xx
            ytotal = y
        i += 1
        x_train, x_test, y_train, y_test = train_test_split(xtotal,
             ytotal, test_size=0.2, random_state=42)
    except TypeError:
        pass
```

Hyper parameters setting

```
A,B = 28,28 # image width,height
img_size = B*A # the canvas size
enc_size = 256 # number of hidden units / output size in LSTM
dec_size = 256
read_n = 5 # read glimpse grid width/height
write_n = 5 #  write glimpse grid width/height
read_size = 2*read_n*read_n
write_size = write_n*write_n
z_size = 10 # QSampler output size
T = 20 # Data generation sequence length
batch_size = 100 # training minibatch size
train_iters = 10000
learning_rate = 1e-3 # learning rate for optimizer
eps = 1e-8 # epsilon for numerical stability



DO_SHARE = None

# input (batch_size * img_size)
x = tf.placeholder(tf.float32,shape=(batch_size,img_size))

# Qsampler noise
e = tf.random_normal((batch_size,z_size), mean=0, stddev=1)

# encoder Op
lstm_enc = tf.contrib.rnn.LSTMCell(enc_size, state_is_tuple=True)

# decoder Op
lstm_dec = tf.contrib.rnn.LSTMCell(dec_size, state_is_tuple=True)
```

Functions

```
#affine transformation
def linear(x, output_dim):
    #x.shape = (batch_size, num_features)
    w = tf.get_variable("w", [x.get_shape()[1], output_dim])
    b = tf.get_variable("b", [output_dim], initializer=tf.
        constant_initializer(0.0))
    return tf.matmul(x,w)+b

def filterbank(gx, gy, sigma2,delta, N):
    grid_i = tf.reshape(tf.cast(tf.range(N), tf.float32), [1, -1])
    mu_x = gx + (grid_i - N / 2 - 0.5) * delta # Eq (1)
    mu_y = gy + (grid_i - N / 2 - 0.5) * delta # Eq (2)
    a = tf.reshape(tf.cast(tf.range(A), tf.float32), [1, 1, -1])
    b = tf.reshape(tf.cast(tf.range(B), tf.float32), [1, 1, -1])
    mu_x = tf.reshape(mu_x, [-1, N, 1])
    mu_y = tf.reshape(mu_y, [-1, N, 1])
    sigma2 = tf.reshape(sigma2, [-1, 1, 1])
    Fx = tf.exp(-tf.square(a - mu_x) / (2*sigma2))
    Fy = tf.exp(-tf.square(b - mu_y) / (2*sigma2)) # batch x N x B
    # normalize, sum over A and B dims
    Fx=Fx/tf.maximum(tf.reduce_sum(Fx,2,keepdims=True),eps)
    Fy=Fy/tf.maximum(tf.reduce_sum(Fy,2,keepdims=True),eps)
    return Fx,Fy
```

```
def attn_window(scope,h_dec,N):
    with tf.variable_scope(scope,reuse=DO_SHARE):
        params=linear(h_dec,5)
    gx_,gy_,log_sigma2,log_delta,log_gamma=tf.split(params,5,1)
    gx=(A+1)/2*(gx_+1)
    gy=(B+1)/2*(gy_+1)
    sigma2=tf.exp(log_sigma2)
    delta=(max(A,B)-1)/(N-1)*tf.exp(log_delta) # batch x N
    return filterbank(gx,gy,sigma2,delta,N)+(tf.exp(log_gamma),)


#Read attention
def read_attn(x,x_hat,h_dec_prev):
    Fx,Fy,gamma=attn_window("read",h_dec_prev,read_n)
    def filter_img(img,Fx,Fy,gamma,N):
        Fxt=tf.transpose(Fx,perm=[0,2,1])
        img=tf.reshape(img,[-1,B,A])
        #F_Y x F_X^T
        glimpse=tf.matmul(Fy,tf.matmul(img,Fxt))
        glimpse=tf.reshape(glimpse,[-1,N*N])
        return glimpse*tf.reshape(gamma,[-1,1])
    x=filter_img(x,Fx,Fy,gamma,read_n) # batch x (read_n*read_n)
    x_hat=filter_img(x_hat,Fx,Fy,gamma,read_n)
    return tf.concat([x,x_hat], 1) # concat along feature axis

read = read_attn


def encode(state,input):
    """
    run LSTM
    state = previous encoder state
    input = cat(read,h_dec_prev)
    returns: (output, new_state)
    """
    with tf.variable_scope("encoder",reuse=DO_SHARE):
        return lstm_enc(input,state)


# Q-Sampler (Variational Autoencoder)
def sampleQ(h_enc):
    """
    sample zt ~ normrnd(mu,sigma) with reparametrization trick for
        normal distribution:
    z ~ mu + normrnd(0,1) * sigma
    """
    with tf.variable_scope("mu",reuse=DO_SHARE):
        mu=linear(h_enc,z_size)
    with tf.variable_scope("sigma",reuse=DO_SHARE):
        logsigma=linear(h_enc,z_size)
        sigma=tf.exp(logsigma)
    return (mu + sigma*e, mu, logsigma, sigma)


def decode(state,input):
    with tf.variable_scope("decoder",reuse=DO_SHARE):
```

```
        return lstm_dec(input, state)

# Write attention
def write_attn(h_dec):
    with tf.variable_scope("writeW",reuse=DO_SHARE):
        w=linear(h_dec,write_size) # batch x (write_n*write_n)
    N=write_n
    w=tf.reshape(w,[batch_size,N,N])
    Fx,Fy,gamma=attn_window("write",h_dec,write_n)
    Fyt=tf.transpose(Fy,perm=[0,2,1])
    wr=tf.matmul(Fyt,tf.matmul(w,Fx))
    wr=tf.reshape(wr,[batch_size,B*A])
    #gamma=tf.tile(gamma,[1,B*A])
    return wr*tf.reshape(1.0/gamma,[-1,1])


write = write_attn

#Main Loss Function
def binary_crossentropy(t,o):
    return -(t*tf.log(o+eps) + (1.0-t)*tf.log(1.0-o+eps))
```

State variables

```
cs=[0]*T # sequence of canvases
mus,logsigmas,sigmas=[0]*T,[0]*T,[0]*T # gaussian params generated
    by SampleQ

# initial states
h_dec_prev=tf.zeros((batch_size,dec_size))
enc_state=lstm_enc.zero_state(batch_size, tf.float32)
dec_state=lstm_dec.zero_state(batch_size, tf.float32)
```

Draw model

```
# construct the unrolled computational graph
for t in range(T):
    c_prev = tf.zeros((batch_size,img_size)) if t==0 else cs[t-1]
    x_hat = x-tf.sigmoid(c_prev) # error image
    r=read(x,x_hat,h_dec_prev)
    h_enc,enc_state = encode(enc_state,tf.concat([r,h_dec_prev], 1)
        )
    z,mus[t],logsigmas[t],sigmas[t] = sampleQ(h_enc)
    h_dec,dec_state = decode(dec_state,z)
    cs[t] = c_prev+write(h_dec) # store results
    h_dec_prev = h_dec
    DO_SHARE = True
```

Error

```
# reconstruction term collapsed down to a single scalar value
x_recons=tf.nn.sigmoid(cs[-1])

# compute binary cross entropy, sum across feautes and take the
    mean across minibatches
```

```
Lx=tf.reduce_sum(binary_crossentropy(x,x_recons),1) #
    reconstruction term
Lx=tf.reduce_mean(Lx)

#KL divergence
kl_terms=[0]*T
for t in range(T):
    mu2=tf.square(mus[t])
    sigma2=tf.square(sigmas[t])
    logsigma=logsigmas[t]
    kl_terms[t]=0.5*tf.reduce_sum(mu2+sigma2-2*logsigma,1)-.5 #
        each kl term is (1xminibatch)
KL=tf.add_n(kl_terms) # this is 1xminibatch, corresponding to
    summing kl_terms from 1:T
Lz=tf.reduce_mean(KL) # average over minibatches

#Total cost
cost=Lx+Lz
```

Optimizer

```
optimizer=tf.train.RMSPropOptimizer(learning_rate)
grads=optimizer.compute_gradients(cost)
for i,(g,v) in enumerate(grads):
    if g is not None:
        grads[i]=(tf.clip_by_norm(g,5),v) # clip gradients
train_op=optimizer.apply_gradients(grads)
```

Training

```
train_data = Dataset(x_train)

fetches=[]
fetches.extend([Lx,Lz,train_op])
Lxs=[0]*train_iters
Lzs=[0]*train_iters
sess=tf.InteractiveSession()


# saves variables learned during training
saver = tf.train.Saver()
tf.global_variables_initializer().run()


for i in range(train_iters):
        xtrain=train_data.next_batch(batch_size) # xtrain is (
            batch_size x img_size)

        feed_dict={x:xtrain}
        results=sess.run(fetches,feed_dict)
        Lxs[i],Lzs[i],_=results
        if i%100==0:
                print("iter=%d : Lx: %f Lz: %f" % (i,Lxs[i],Lzs[i])
                    )
```

```
canvases=sess.run(cs,feed_dict) # generate  examples
canvases=np.array(canvases) # T x batch x img_size

out_file=os.path.join(FLAGS.folder,"draw_datacat102010055.npy")
np.save(out_file,[canvases,Lxs,Lzs])
print("Outputs saved in file: %s" % out_file)
```

Reconstruction

```
import matplotlib
import sys
import numpy as np
matplotlib.use('Agg')
import matplotlib.pyplot as plt


def xrecons_grid(X,B,A):
        """
        plots canvas for single time step
        X is x_recons, (batch_size x img_size)
        features = BxA images

        """
        padsize=1
        padval=.5
        ph=B+2*padsize
        pw=A+2*padsize
        batch_size=X.shape[0]
        N=int(np.sqrt(batch_size))
        X=X.reshape((N,N,B,A))
        img=np.ones((N*ph,N*pw))*padval
        for i in range(N):
                for j in range(N):
                        startr=i*ph+padsize
                        endr=startr+B
                        startc=j*pw+padsize
                        endc=startc+A
                        img[startr:endr,startc:endc]=X[i,j,:,:]
        return img

if __name__ == '__main__':
        prefix='datacat102010055'
        out_file='draw_datacat102010055.npy'
        [C,Lxs,Lzs]=np.load(out_file)
        T,batch_size,img_size=C.shape
        X=1.0/(1.0+np.exp(-C)) # x_recons=sigmoid(canvas)
        B=A=int(np.sqrt(img_size))
        if interactive:
                f,arr=plt.subplots(1,T)
        for t in range(T):
                img=xrecons_grid(X[t,:,:],B,A)
                if interactive:
```

```
                        arr[t].matshow(img,cmap=plt.cm.gray)
                        arr[t].set_xticks([])
                        arr[t].set_yticks([])
                else:
                        plt.matshow(img,cmap=plt.cm.gray)
                        imgname='%s_%d.png' % (prefix,t)
                        plt.savefig(imgname)
                        print(imgname)
        f=plt.figure()
        plt.plot(Lxs,label='Reconstruction Loss Lx')
        plt.plot(Lzs,label='Latent Loss Lz')
        plt.xlabel('iterations')
        plt.legend()
        if interactive:
                plt.show()
        else:
                plt.savefig('%s_loss.png' % (prefix))
```

Testing

```
test_data =  Dataset(x_test)
xtest=test_data.next_batch(100)
feed_dict_t={x:xtest}
results_t=sess.run(fetches_t,feed_dict_t)
Lxs_t,Lzs_t,_=results_t

print("Lx: %f Lz: %f" % (Lxs_t,Lzs_t))

canvases=sess.run(cs,feed_dict_t)
canvases=np.array(canvases) # T x batch x img_size

out_file=os.path.join(FLAGS.folder,"draw_datacatdog102010055t.npy")
np.save(out_file,[canvases,Lxs_t,Lzs_t])
print("Outputs saved in file: %s" % out_file)
```

Bibliography

[1] Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., & Wierstra, D. *DRAW: A Recurrent Neural Network For Image Generation.* Google DeepMind, arXiv:1502.04623v2 [cs.CV] 20 May 2015.

[2] Ha, D., & Eck, D. *A Neural Representation of Sketch Drawings.* ICLR 2018, arXiv:1704.03477

[3] Jongejan, J., Rowley, H., Kawashima, T., Kim, J., & Fox-Gieg, N. *The Quick, Draw!.* `https://quickdraw.withgoogle.com/`

[4] Kingma,D. P., & Welling, M. *Auto-Encoding Variational Bayes.* Proceedings of the 2nd International Conference on Learning Representations (ICLR), ArXiv:1312.6114v10, 1 May 2014.

[5] Kullback, S. *The Kullback-Leibler Distance.* The American Statistician, Vol. 41, No. 4. (1987), pp. 340-341.