Deep Learning Homework 4

Name: Yifan Shen

SN: 15015762

Start date: *13th March 2018*

Due date: *29th March 2018, 11:55 pm*

How to Submit

When you have completed the exercises and everything has finsihed running, click on 'File' in the menu-bar and then 'Download .ipynb'. This file must be submitted to Moodle named as **studentnumber_DL_hw4.ipynb** before the deadline above.

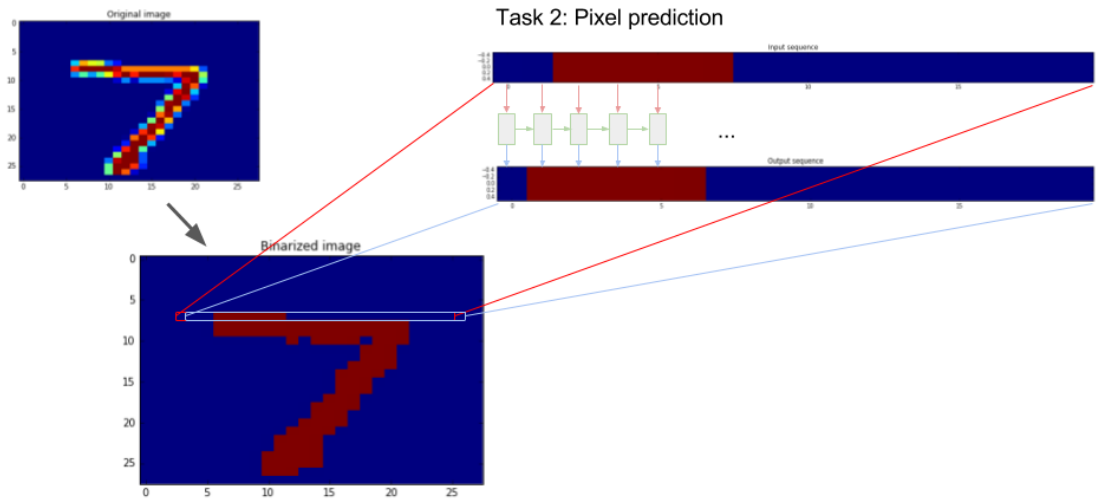
Also send a **sharable link** to the notebook at the following email: ucl.coursework.submit@gmail.com. You can also make it sharable via link to everyone, up to you.

Please compile all results, all plots/figures and all answers to the understanding/analysis results questions into a PDF. Name convention: **studentnumber_DL_hw4.pdf**. Do not include any of the code (we will use the notebook for that).

Page limit: 15 pg .

PART 1: MNIST as a sequence (follow-up from last assignment)

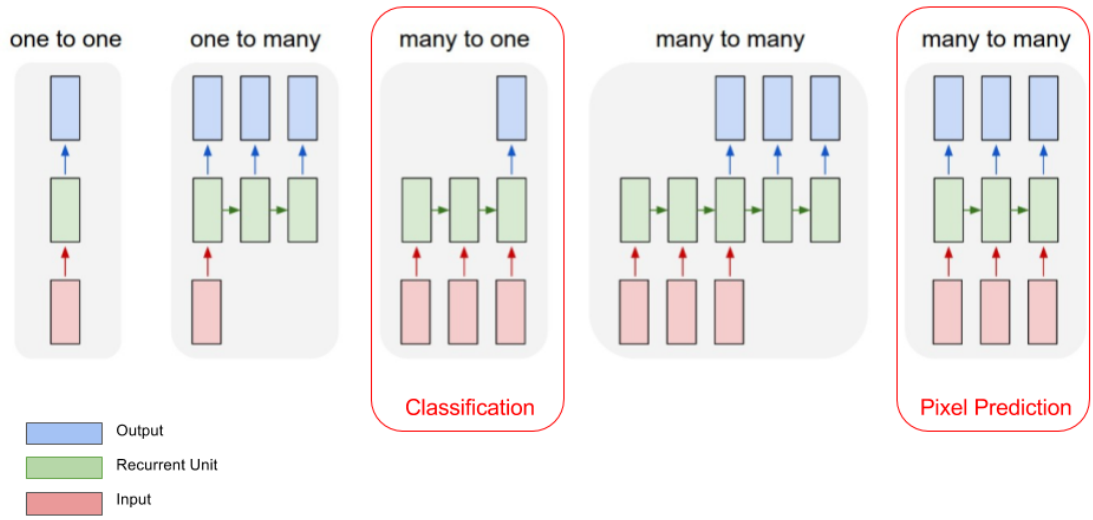
In this assignment we will be using the MNIST digit dataset (https://yann.lecun.com/exdb/mnist/). The dataset contains images of hand-written digits ($0 - 9$), and the corresponding labels. The images have a resolution of 28×28 pixels. This is the same dataset as in Assignment 1, but we will be using this data a bit differently this time around. Since this assignment will be focusing on recurrent networks that model sequential data, we will be looking at each image as a sequence: the networks you train will be "reading" the image one row at a time, from top to bottom (we could even do pixel-by-pixel, but in the interest of time we'll do row-by-row which is faster). Also, we will work with a binarized version of MNIST -- we constrain the values of the pixels to be either 0 or 1. You can do this by applying the method `binarize`, defined below, to the raw images.



- We take the MNIST images, binarise them, and interpret them as a sequence of pixels from top-left to bottom-right. ("Task 2" refers to the next homework, wherein you will be using the sequence for pixel prediction).

Recurrent Models for MNIST

As discussed in the lectures, there are various ways and tasks for which we can use recurrent models. A depiction of the most common scenarios is available in the Figure below. In this assignment and the following one we will look at two of these forms: **many-to-one** (sequence to label/decision) and the **many-to-many** scenario where the model receives an input and produces an output at every time step. You will use these to solve the following tasks: i) pixel prediction and ii) in-painting.



- (Figure adapted from Karpathy's The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness)). You will be implementing variants of *many-to-one* for classification (in this homework), and *many-to-many* for prediction (in the next homework).

Task 1: (Next) Pixel prediction (35 pts)

In this part, you will train a **many-to-many** recurrent model: at each time t, the model receives as input a pixel value x_t and tries to predict the next pixel in the images x_{t+1} based on the current input and the recurrent state. Thus, your output function is now a probability over the value of pixel x_{t+1} -- which can be either 0 or 1 (black or white).

$$\hat{p}(x_{t+1}|x_{1:t}) = g(x_t, h_t, c_t)$$

Once we get to observe the actual value of x_{t+1} at the next time-step, we can compute the cross-entropy between our predicted probability $\hat{p}(x_{t+1}|x_{1:t})$ and the observed value (pixel in the image). We can (and will) do that for every time-step prediction within a sequence. This will provide us with the training signal for optimizing the parameters of the mapping g and the recurrent connections -- remember these are shared!, they do not change with t.

Optimization

Use the Adam optimizer (with default settings other than the learning rate) for training.

[Optional] Sometimes dropout has been shown to be beneficial in training recurrent models, so feel free to use it or any other form of regularization that seems to improve performance. It might be also worth trying out batch-normalization. Reference (https://arxiv.org/pdf/1603.09025.pdf).

Models: Your models will have the following structure:

1. [(Red Block)] The *input* (current binarised row of pixels) can be fed directly into the recurrent connection without much further pre-processing.
2. [(Blue Block)] The *output* (probabilities over the activation of the pixel) is produced by looking at the last output of the recurrent units, transforming them via an affine transformation.
3. [(Green Block)] For the *recurrent* part of the network, please implement and compare the following architectures:

   - LSTM with 32 units. **[15 pts]**

     OR
   - GRU with 32 units. **[15 pts]**

Your network should look like: \begin{equation} \textrm{Input} \Rightarrow \textrm{RNN cell} \Rightarrow \textrm{Relu} \Rightarrow \textrm{Fully connected} \Rightarrow \textrm{Relu} \Rightarrow \textrm{Fully connected} \Rightarrow \textrm{Output} \end{equation} You might find the function `tf.nn.dynamic_rnn` useful.

Hyper-parameters

For all cases train the model with these hyper-parameter settings:

- *num_epochs*=5, *learning_rate*=0.001, *batch_size*=256, *fully_connected_hidden_units=64*

With these hyper-parameters you should give you a good perfomance on both GRUs/LSTMs. It is worth noting that in 5 epochs the model has yet converged, but in the interest of time (the training should have $\approx$ 1h). That being said, feel free to try other settings, there are certainly better choices, but please report the results with these exact hyper-parameters and/or train for longer -- the models should still improve (convergence is achieved around 25-30 epochs).

Tasks:

1) Implement and train the previously described model (choose either GRU **or** LSTM). Please report the *cross-entropy* on the *test set* and *training set* of the models trained. Use the `plot_summary_table` method below to format the table. Provide the learning curves (both training and testing loss) -- choose appropiate reporting interval here (at least 20 points).

2) Using the previously trained model, visualize the 1-step predictions, 10-step predictions, one row prediction (28 steps) and filling out the image (fill out all the pixels using the recurrent model).

- **Generate a small in-painting dataset.** Sample 100 images from your test set. Mask/Remove the last 300 pixels (roughly 10 rows and a half).
- **Predict missing parts and compare with GT**. Given the above generated partial sequences as input to your train models, generate the continuation of these masked images (for the next 1, 10, 28, 300 pixels). Report the cross-entropy of your in-paintings for the trained model at beginning of training(0 epochs), after 1 epoch and at the end of training. Discuss the results: contrasting long/short time prediction; compare these with the cross-entropy of the ground truth images. For multiple steps in-paintings, average the loss over 10 samples. **[10 pts]**

- **Visualize completing the image**. Pick out 3 examples from your in-painting dataset to visualize the resulting images -- this can be done at random, but should include *a successful example, failure example and one that displays high variance between samples.* For each example picked, please provide 5 samples for the last three scenarios (10, 28, 300 pixels) and 1 for the 1-pixel prediction -- total 16 samples/exampls. The samples should be generated recursively by sampling the generative process provided by the trained recurrent connections. Total number of in-painting to report: 16 samples x 3 examples = 48 **[10 pts]**

Task 2: Using pixel-to-pixel: In-painting task (25 pts)

Using the models trained in the previous section, please in-paint the missing pixels in the following datasets:

- One-pixel missing (https://github.com/dianaborsa/compgi22_dl_cw4/blob/master/one_pixel_inpainting.npy)
- Window of 2x2 pixels missing (https://github.com/dianaborsa/compgi22_dl_cw4/blob/master/2X2_pixels_inpainting.npy)

This is similar to Task 1.b, but now you have information not only about the past(previous pixels in the image) but also future (pixels that come after your predictive target)

Results

1) Provide the formula used to compute the probability over the missing pixel and respectively for the missing patch **[5+5 pts]**

2) Visualize the most probable in-painting, according to your model. How does this compare to the ground truth? (Compare cross-entropy between your most probable sample and the ground truth). Explain the difference. It is enough to include just one example per task/dataset. **[10 pts]**

In-painting data

We provide two datasets (one corresponing to the one-pixel in-painting tasks and the other one with a 2x2 patch missing). The datasets are available on git (https://github.com/dianaborsa/compgi22_dl_cw4/blob/master/). Links are available in the description and code is provide below to load the dataset and visualize. Both datasets have 1000 sampled images from MNIST(test). Both dataset sets have the same simple structure: cropped images and their ground truth (GT). in this second task, you will consider the copped images and use your pixel-to-pixel model, try to predict the missing pixel/patches.

Imports and utility functions (do not modify!)

In [0]:
```
#@title Import libraries
#@test {"output": "ignore"}

# Import useful libraries.
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data


# Binarize the images
def binarize(images, threshold=0.1):
  return (threshold < images).astype('float32')

# Import dataset with one-hot encoding of the class labels.
def get_data():
  return input_data.read_data_sets("MNIST_data/", one_hot=True)

# Placeholders to feed train and test data into the graph.
# Since batch dimension is 'None', we can reuse them both for train and eval.
def get_placeholders():
  x = tf.placeholder(tf.float32, [None, 783])
  y = tf.placeholder(tf.float32, [None, 783])
  return x, y

# Generate summary table of results. This function expects a dict with the
# following structure: keys of 'LSTM' (or 'GRU') and the values for each key are a
# list of tuples consisting of (test_loss, test_accuracy), and the list is
# ordered as the results from 0 epoch (beginning of training), 1 epoch, 5 epochs (or end of training):
# {
#  'LSTM': [(loss,acc), (loss, acc), (loss, acc)]
# }
def plot_summary_table(experiment_results):
  # Fill Data.
  cell_text = []
  columns = ['(Beginning - 0 epochs)', '(Mid-training - 1 epoch)', '(End of training - 5 epochs)']
  for k, v in experiment_results.items():
    rows = ['Test loss', 'Test accuracy']
    cell_text=[[],[]]
    for (l, _) in v:
      cell_text[0].append(str(l))
    for (_, a) in v:
      cell_text[1].append(str(a))

    fig=plt.figure(frameon=False)
    ax = plt.gca()
    the_table = ax.table(
      cellText=cell_text,
      rowLabels=rows,
      colLabels=columns,
      loc='center')
    the_table.scale(2, 8)
    # Prettify.
    ax.patch.set_facecolor('None')
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.text(-0.73, 0.9, k, fontsize=18)


def plot_learning_curves(training_loss, testing_loss):
  plt.plot(training_loss)
  plt.plot(testing_loss, 'g')
  plt.legend(['Training loss', 'Testing loss'])
  plt.show()
```

In [0]:
```
def showImage(arr, missing = 'end'):
    full_plot = np.zeros((784))
    # len(arr) = 783
    if missing == 'end':
        full_plot[-len(arr):] = arr
    elif missing == 'begin':
        full_plot[:len(arr)] = arr
    full_plot_reshape = full_plot.reshape((28,28))
    plt.imshow(full_plot_reshape, cmap="gray")
```

Train Models

Generate summary table of results. This function expects a dict with the following structure: keys of 'LSTM' (or 'GRU') and the values for each key are a list of tuples consisting of (test_loss, test_accuracy), and the list is ordered as the results from 0 epoch (beginning of training), 1 epoch, 5 epochs (or end of training) i.e. expected dictionary (final performace only):

```
{
  'LSTM': [(loss,acc), (loss, acc), (loss, acc)]
}
```

In [78]:
```
mnist = get_data()
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

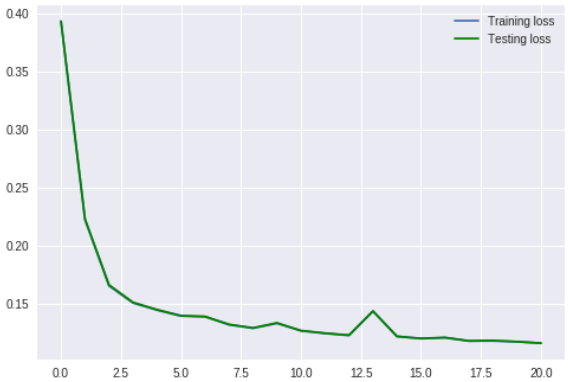
In [0]:
```
class lstmModel(object):
    def __init__(self, batch_size = 256, num_rnn_layers=1, num_units=32,
                 fully_connected_hidden_units=64, num_epochs=5, learning_rate=0.01):
        self.batch_size = batch_size
        self.num_rnn_layers=num_rnn_layers
        self.num_units = num_units
        self.fully_connected_hidden_units = fully_connected_hidden_units
        self.num_epochs = num_epochs
        self.learning_rate = learning_rate
        self.setGraph()

    def setGraph(self):
        with tf.device('/device:GPU:*'):
          tf.reset_default_graph()
          self.X, self.Y = get_placeholders()
          self.rnnCell()
          self.linearLayer(self.rnn_output)
          self.loss()
          self.train()
          self.accuracy()
          self.pixelPrediction()

    def rnnCell(self):
        RNN_layers = [tf.contrib.rnn.BasicLSTMCell(size) for size in ([self.num_units] * self.num_rnn_layers)]
        self.cell = tf.contrib.rnn.MultiRNNCell(RNN_layers)
        x_rnn_input=tf.reshape(self.X,[-1,783,1])
        rnn_output, _ = tf.nn.dynamic_rnn(self.cell, x_rnn_input, dtype=tf.float32)
        # rnn_output: 256 * 783 * 32
        self.rnn_output = tf.reshape(rnn_output, [-1, self.num_units])
        # self.rnn_output: (256 x 783) * 32

    def linearLayer(self, linear_layer_input):
        hid_1_output = tf.contrib.layers.fully_connected(linear_layer_input, self.fully_connected_hidden_units, tf.nn.relu)
        hid_2_output = tf.contrib.layers.fully_connected(hid_1_output, self.fully_connected_hidden_units, tf.nn.relu)
        model_output = tf.contrib.layers.fully_connected(hid_2_output, 1, tf.nn.sigmoid)
        self.linear_layer_output = tf.reshape(model_output,[-1,1]) # 256 * 783

    def loss(self):
        self.train_result = tf.reshape(self.linear_layer_output, [-1,783])
        self.loss = -tf.reduce_mean(
            self.Y * tf.log(self.train_result) + (1 - self.Y) * tf.log(1 - self.train_result))

    def train(self):
        self.optimizer = tf.train.AdamOptimizer().minimize(self.loss)

    def accuracy(self):
        train_result_bin = tf.round(self.train_result) # 256 * 783
        self.accuracy = tf.reduce_sum(tf.cast(tf.equal(train_result_bin, self.Y),dtype=tf.int32))/(self.batch_size * 783)

    def pixelPrediction(self):
        self.test_input = tf.placeholder(tf.float32, [self.batch_size, 1]) # 256 * 1
        self.test_initial_state = self.cell.zero_state(self.batch_size, dtype = tf.float32)
        self.cell_output, self.state_output = self.cell(self.test_input, self.test_initial_state)
        self.linearLayer(self.cell_output)
        self.pixel_prediction = self.linear_layer_output

```

In [87]:
```
Model = lstmModel(num_epochs=5)
experiment_results = {'LSTM':[]}
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  saver = tf.train.Saver(tf.global_variables())
  i = 0
  training_loss = []
  testing_loss = []
  # for plotting learning curves
  train_x_full = binarize(mnist.train.images[0:11000, :-1])
  train_y_full = binarize(mnist.train.images[0:11000, 1:])
  test_x_full = binarize(mnist.test.images[:,:-1])
  test_y_full = binarize(mnist.test.images[:,1:])
  while mnist.train.epochs_completed < Model.num_epochs:
    i+=1
    x, _ = mnist.train.next_batch(Model.batch_size)
     # for batch training
    x_bin=binarize(x) # 256 * 784
    x_train = x_bin[:,:-1] # 256 * 783
    y_train = x_bin[:,1:] # 256 * 783
    _, loss_run, acc_run = sess.run([Model.optimizer, Model.loss, Model.accuracy],
                {Model.X: x_train, Model.Y: y_train})
    store_iteration = [1,215,1075] # at 0,1,5 epochs
    if i in store_iteration:
      print('iteration:', i, 'epochs_completed:', mnist.train.epochs_completed,
           'loss:', loss_run, 'batch_accuracy',acc_run)
      experiment_results['LSTM'].append((loss_run, acc_run))

    if i%50 == 0:
      train_los = sess.run(Model.loss, {Model.X: train_x_full, Model.Y: train_y_full})
      test_los = sess.run(Model.loss, {Model.X: test_x_full, Model.Y: test_y_full})
      training_loss.append(train_los)
      testing_loss.append(test_los)
      print(i)

  path='./dl4'+str(mnist.train.epochs_completed)+'.ckpt'
  save_path = saver.save(sess, path)
  print("Model saved in path: %s" % save_path)


```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
Model saved in path: ./dl45.ckpt
```

Results

In [134]:
```
# plot learning curve:
plot_learning_curves(training_loss, testing_loss)
```



In [135]:
```
1 plot_summary_table(experiment_results)
```

LSTM	(Beginning - 0 epochs)	(Mid-training - 1 epoch)	(End of training - 5 epochs)
Test loss	0.69256914	0.14796627	0.11355774
Test accuracy	0.8283894077266922	0.9367766203703705	0.9539880667305237

Pixel prediction

Generate a small in-painting dataset.

Sample 100 images from your test set. Mask/Remove the last 300 pixels (roughly 10 rows and a half).

In [0]:
```
def pixelPredict(num_epochs = 5, num_next_pixel = 300):
    Model = lstmModel(batch_size=100)
    eval_mnist = get_data()
    dataset = binarize(eval_mnist.test.images[0:100])
    final = dataset[:,:-300]
    mnist = get_data()
    with tf.Session() as sess:
        saver = tf.train.Saver(tf.global_variables())
        path ='./dl4' + str(num_epochs) + '.ckpt'
        saver.restore(sess, path)

        bin_test = final
        test_x = bin_test[:,:-1]
        test_y = bin_test[:,1:]

        np_state_output = sess.run(Model.test_initial_state)
        final = in_painting_dataset
        for i in range(0,783 - 300):
            test_input = test_x[:,i].reshape((-1,1))
            feed = {Model.test_input:test_input, Model.test_initial_state:np_state_output}
            pixel_prediction, np_state_output = sess.run([Model.pixel_prediction, Model.state_output],feed_dict=feed)

        for i in range(0,num_next_pixel):
            feed = {Model.test_input:pixel_prediction, Model.test_initial_state:np_state_output}
            pixel_prediction, np_state_output = sess.run([Model.pixel_prediction, Model.state_output],feed_dict=feed)
            pixel_prediction = [np.random.choice(2, p=[1 - p, p]) for p in pixel_prediction[:,0]]
            pixel_prediction = np.asmatrix(pixel_prediction).reshape(-1,1)

            final = np.concatenate((final, pixel_prediction),axis=1)

        concat_zeros = np.zeros((100, 300 - num_next_pixel ))
        final = np.concatenate((final, concat_zeros), axis = 1)

        return final
```

Predict missing parts and compare with the ground truth.

Given the above generated partial sequences as input to your train models, generate the continuation of these masked images (for the next 1, 10, 28, 300 pixels).

In [0]:
```
def calculateCorssEntropy(orignal, predict):
  predict_softmax = np.exp(predict) / np.sum(np.exp(predict)) # softmax
  return - np.sum((orignal * np.log(predict_softmax)) + (1 - orignal) * np.log(1 - predict_softmax))
# calculate cross entropy the 10 samples
def crossEntropyForPrediction(generated_images):
  mnist=get_data()
  original_images = mnist.test.images[0:100]
  num_pixels_list = [1, 10,28,300]
  cross_entropy_store = {}
  for num_pixels in num_pixels_list:
    cross_for_pixel = 0
    for which_run in range(10):
      generated_pixels = np.asarray(generated_images[str(which_run+1)][:,:484 + num_pixels])
      original_pixels = np.asarray(original_images[:,:484 + num_pixels])
      cross_each_image = 0
      for j in range(100):
        cross_each_image += calculateCorssEntropy(original_pixels[j,:], generated_pixels[j,:])
      cross_for_pixel += cross_each_image
    cross_for_pixel = cross_for_pixel/((784) * 100 * 10)
    cross_entropy_store[str(num_pixels)] = cross_for_pixel
  return cross_entropy_store
```

In [90]:

```python
generated_images_epochs5 = {}
for nth_sample in range(10):
  generated_images_epochs5[str(nth_sample+1)] = pixelPredict(num_epochs = 5)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
```

In [91]:
```
cross_entropy_store_epochs5 = crossEntropyForPrediction(generated_images_epochs5)
cross_entropy_store_epochs5
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Out[91]:
```
{'1': 0.4301752861585081,
 '10': 0.4551913692648021,
 '28': 0.46817838707376186,
 '300': 0.7370442306493591}
```

In [92]:

```python
generated_images_epochs1 = {}
for nth_sample in range(10):
  generated_images_epochs1[str(nth_sample+1)] = pixelPredict(num_epochs = 1)


```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
restore model
INFO:tensorflow:Restoring parameters from ./dl41.ckpt
```

In [93]:
```
1 cross_entropy_store_epochs1 = crossEntropyForPrediction(generated_images_epochs1)
2 cross_entropy_store_epochs1
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Out[93]:
```
{'1': 0.43021156617570433,
 '10': 0.45575205205458086,
 '28': 0.4693656443628442,
 '300': 0.7427415140534898}
```

In [94]:

```python
generated_images_epochs0 = {}
for nth_sample in range(10):
  generated_images_epochs0[str(nth_sample+1)] = pixelPredict(num_epochs = 0)

```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
restore model
INFO:tensorflow:Restoring parameters from ./dl40.ckpt
```

In [95]:
```
cross_entropy_store_epochs0 = crossEntropyForPrediction(generated_images_epochs0)
cross_entropy_store_epochs0
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

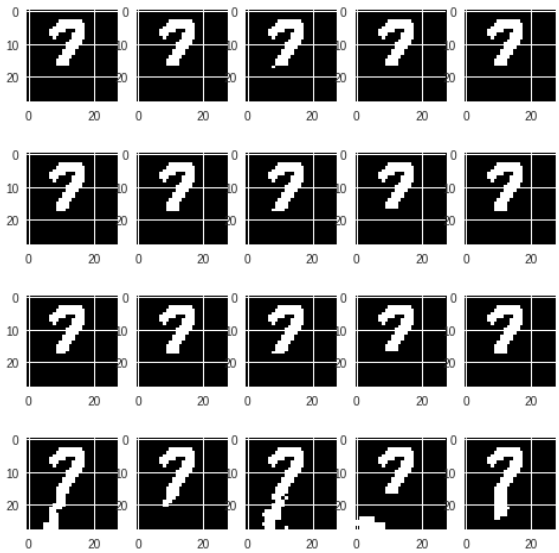
Out[95]:
```
{'1': 0.43029152368942114,
 '10': 0.456389046478028,
 '28': 0.4712260912478111,
 '300': 0.7520377629290987}
```

Answer: The cross entropy loss follows: higher loss with more pixels and higher loss with small number of training epochs.

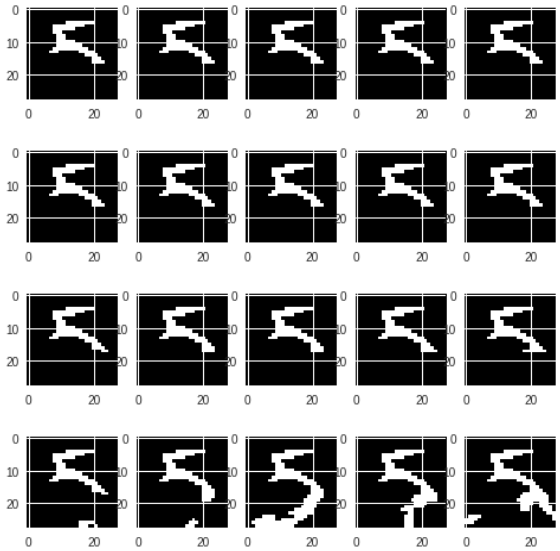
Visualize completing the image.

Pick out 3 examples from your in-painting dataset to visualize the resulting images -- this can be done at random, but should include \textit{a successful example, failure example and one that displays high variance between samples.}

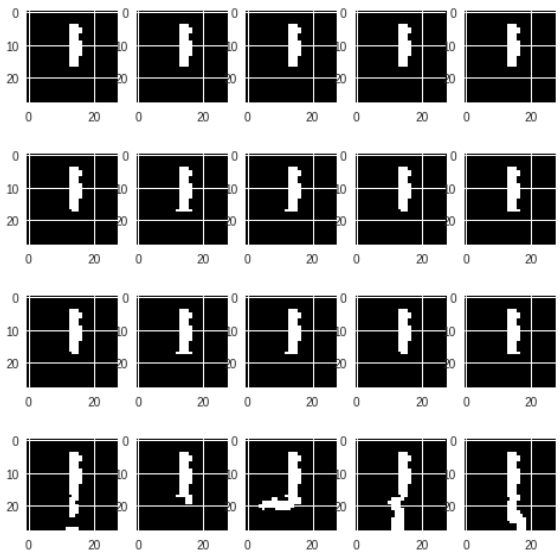
In [148]:
```
pixel_for_vis = [1,10,28,300]
# sub_plot_location = [161, 162, 163, 164, 165, 166]
sub_plot_location = [151, 152, 153, 154, 155]
# We use generated_images_epochs5 to plot:
# first image:
for img_idx in [1]:
  plt.figure()
  image_list = [generated_images_epochs5[str(which_run)][img_idx] for which_run in range(1,6)]
  for px in pixel_for_vis:
    append_zeros = np.zeros((1, 300 -  px))
    for i, loc in enumerate(sub_plot_location):
      plt.subplot(loc)
      showImage(np.concatenate((image_list[i-1][:,: 484 +  px], append_zeros), axis = 1),'None')
    plt.show()

```



In [149]:
```
for img_idx in [15]:
  plt.figure()
  image_list = [generated_images_epochs5[str(which_run)][img_idx] for which_run in range(1,6)]
  for px in pixel_for_vis:
    append_zeros = np.zeros((1, 300 -  px))
    for i, loc in enumerate(sub_plot_location):
      plt.subplot(loc)
      showImage(np.concatenate((image_list[i-1][:,: 484 +  px], append_zeros), axis = 1),'None')
    plt.show()
```



In [158]:
```
for img_idx in [14]:
  plt.figure()
  image_list = [generated_images_epochs5[str(which_run)][img_idx] for which_run in range(1,6)]
  for px in pixel_for_vis:
    append_zeros = np.zeros((1, 300 -  px))
    for i, loc in enumerate(sub_plot_location):
      plt.subplot(loc)
      showImage(np.concatenate((image_list[i-1][:,: 484 +  px], append_zeros), axis = 1),'None')
    plt.show()
```



In-painting Task

In-painting data

We provide two datasets (one corresponing to the one-pixel in-painting tasks and the other one with a 2x2 patch missing). The datasets are available on git (https://github.com/dianaborsa/compgi22_dl_cw4/blob/master/). Links are available in the description and code is provide below to load the dataset and visualize. Both datasets have 1000 sampled images from MNIST(test). Both dataset sets have the same simple structure: cropped images and their ground truth (GT). in this second task, you will consider the copped images and use your pixel-to-pixel model, try to predict the missing pixel/patches.

Formula used in this task:

for one pixel: Suppose the missing pixel is x_k and there are T pixels all together

$$\mathbb{P}(x_k|x_{1:k-1}, x_{k+1:T}) = \frac{\mathbb{P}(x_{1:T})}{\mathbb{P}(x_{1:k-1}, x_k = 0, x_{k+1:T}) + \mathbb{P}(x_{1:k-1}, x_k = 1, x_{k+1:T})}$$

for two pixel: We consider the $2X2$ missing pixels toghther. Note all possibilities are from the set:

$$ combination = [0000,0001,0010,0011, \cdot \cdot \cdot ,1110,1111]$$

denote the missing part and remaining part as:

$$x_{2X2} \mbox{ and }x_{rest}$$ then we have $$\mathbb{P}(x_{2X2}|x_{rest}) = \frac{\mathbb{P}(x_{1:T})}{\sum_{x_{2X2} \in combination}\mathbb{P}(x_{2X2}, x_{rest})})$$ where $\sum_{x_{2X2} \in combination}$ means summing over all combinations in the combination set above

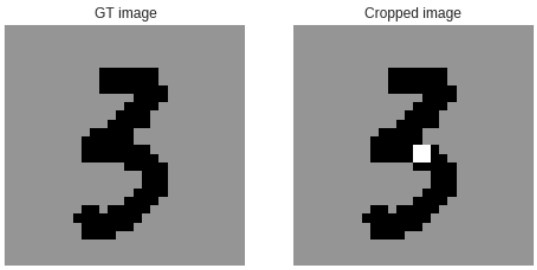
In [95]:
```
# Downloading the inpainting datasets
!git clone https://github.com/dianaborsa/compgi22_dl_cw4.git
```

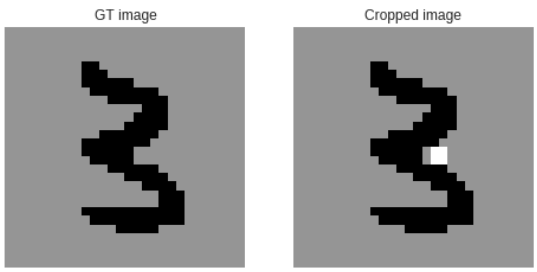
```
Cloning into 'compgi22_dl_cw4'...
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 16 (delta 6), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (16/16), done.
```

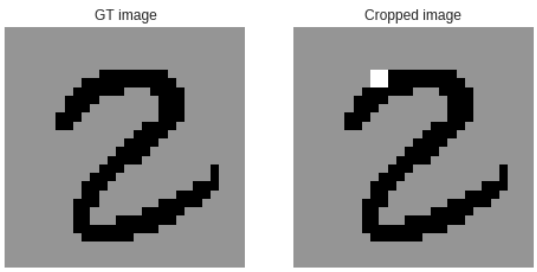
In [0]:
```
# Load the dataset (2X2)
dataset = np.load('compgi22_dl_cw4/2X2_pixels_inpainting.npy')
```

In [128]:
```
# checking loading
images    = dataset[0]
gt_images = dataset[1]

nSamples, ndim = gt_images.shape
print('Loaded dataset has {} samples: cropped + GT'.format(nSamples))

# randomly visualize a few samples
for SampleID in np.random.randint(nSamples,size=3):
  plt.figure()
  plt.subplot(1,2,1)
  plt.imshow(np.reshape(gt_images[SampleID],(28,28)), interpolation='None',vmin=-1, vmax=1)
  plt.title("GT image")
  plt.grid(False)
  plt.axis('off')
  plt.subplot(1,2,2)
  plt.imshow(np.reshape(images[SampleID],(28,28)), interpolation='None',vmin=-1, vmax=1)
  plt.title("Cropped image")
  plt.grid(False)
  plt.axis('off')
```

```
Loaded dataset has 1000 samples: cropped + GT
```





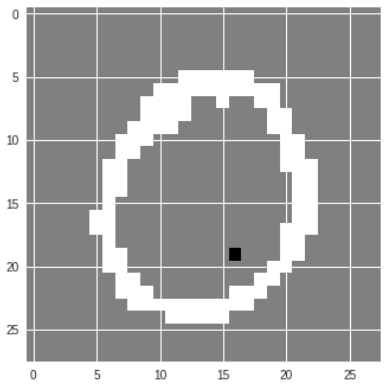


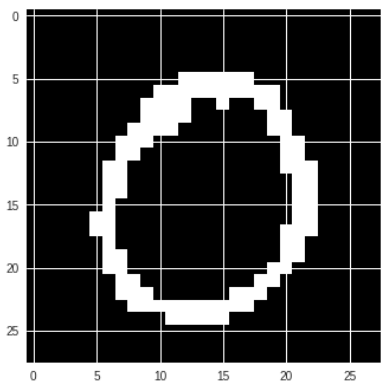
One-pixel prediction

In [188]:

```
Model = lstmModel(batch_size=2)
dataset = np.load('compgi22_dl_cw4/one_pixel_inpainting.npy')
log_joint = np.zeros((2,1))
images = dataset[0]
nSamples = images.shape[0]
combination = ['0', '1']
randomSampleID = np.random.randint(nSamples)
gt_images = dataset[1]
sample=images[randomSampleID]
# possibilities:
location = np.where(sample == -1)[0]
sample_set = np.zeros((2,784))
tmp_1 = np.copy(sample)
tmp_2 = np.copy(sample)
tmp_1[location] = 0.
tmp_2[location] = 1.
sample_set[0,:] = tmp_1
sample_set[1,:] = tmp_2
with tf.Session() as sess:
  saver = tf.train.Saver(tf.global_variables())
  print('restore model')
  path='./dl45.ckpt'
  saver.restore(sess, path)
  np_state_output=sess.run(Model.test_initial_state)
  for i in range(0,783):
    test_input=sample_set[:,i].reshape((-1,1))
    feed={Model.test_input:test_input, Model.test_initial_state:np_state_output}
    pixel_prediction, np_state_output = sess.run([Model.pixel_prediction, Model.state_output],feed_dict=feed)
    log_joint += np.log(pixel_prediction)*sample[i+1]+np.log(1-pixel_prediction)*(1-sample[i+1])
  prob=np.exp(log_joint)/np.sum(np.exp(log_joint))
  painted_sample=np.copy(sample)

  if prob[0] > prob[1]:
    paint_graph = tmp_1
  elif prob[0] < prob[1]:
    paint_graph = tmp_0
  # orignal picture
  plt.imshow(sample.reshape((28,28)), cmap="gray")
  plt.show()
  # inpainted picture
  plt.imshow(paint_graph.reshape((28,28)), cmap="gray")
  plt.show()
```

```
restore model
INFO:tensorflow:Restoring parameters from ./dl45.ckpt
```





Two-by-two patch prediction

PART 2: Learning multiple tasks with LSTM-s (40 pts)

(Credits to Pedro Ortega for insipring the task and insights behind it)

Task Description

Consider the following generative model:

- We have 3 symbols that will be generates from a multinomial/categorical distribution, with parameters $\textbf{p}=(p_1, p_2, p_3)$: symbol 1 is generated with probability p_1, symbol 2 is generated with probability p_2, symbol 3 with probability p_3. $$X \sim Categorical(3, \textbf{p})$$
- The parameter vector $\textbf{p}$ is drawn from a Dirichlet prior: $$\textbf{p} \sim Dirichlet(\alpha)$$

We are going to use the above to generate sequences (a continuous stream of data/observations), in the following way:

- Step 1: We sample $\textbf{p}$ from the prior
- Step 2: Given this $\textbf{p}$, for $T-1$ time-steps we will generate i.i.d observations by sampling one of the 3 symbols from the categorical distribution induced by $\textbf{p}$ $$ (X_1, X_2, \cdots X_{T-1}) , s.t. X_i \sim Categorical(3, \textbf{p})$$
- Step 3: At the end of the sequence we append a fourth symbol (RESET): $(X_1, X_2, \cdots X_T, RESET)$
- Step 4: Return to Step 1 and resample $\textbf{p}$.
- Repeat this 'forever'.

This will give rise to a continuous stream of data, of the form: $ x_1, x_2, \cdots x_{T-1}, RESET, x_{T+1}, x_{T+2}, \cdots x_{2T}, RESET, \cdots, x_{kT+1}, x_{kT+2}, \cdots x_{(k+1)T}, RESET, \cdots$.

Note: Data generation is provided for you in the cell below. (You just need to call the minibatch function to get a sequence of this form).

Model

We are going to consider an simple LSTM (32 units hidden state) and present this (generated) sequence of data as an input. Similar to the pixel-to-pixel model, at each time step the LSTM will receive one bit of information (gets to observe the symbol recorded at this time step) and need to output the probability distribution for the next symbol. Thus, at time t the LSTM get as input the symbol x_t and will return a probability over the next state $P(x_{t+1}| x_{t}, LSTM_{t-1})$.

Questions

1) **Without running any experiments (5 pts)**, try to think about the following scenarios and answer these questions:

- Consider we generate the data with $Dirichlet(\alpha)$, where $\alpha = (10.,1.,1.)$. What do you think the LSTM model will learn, if anything? Remember we are effectively changing the prediction task, every time we are resampling the probability vector $\textbf{p}$. *Hint: Think about the distribution over $\textbf{p}$ that this prior induces.*

Answer: The prior $\alpha = (10.,1.,1.)$ will induce to a high value of p_1 (very close to one), and small values of p_2 and p_3 (close to zero). And the RNN model is very stable in this case.

- What if we consider a more uniformative prior, like $\alpha=(1.1, 1.1, 1.1)$?

Answer: The prior $\alpha=(1.1, 1.1, 1.1)$ will induce a $\mathbf{p}$ which is close to $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. Therefore the sequence will be harder for our RNN model to predict.

- How does this (learning ability and generalization) depend on the lenght of the tasks T and the unrolling length on the LSTM? It might be helpful to consider the two extremes: i) $T=1$ (we reset the task at every time step). What should the model learn in this case?, ii) $T=\infty$ (we sample the task once and keep it forever). What should the model learn in this case? (Answer this for both previous priors)

Answer: 1. $T = 1$, prior $\alpha = (10.,1.,1.)$, the learning and generalization ability will be good and very similar. As $\mathbf{p}$ will always be close to $(1,0,0)$ and the model resets every time. 2. $T = 1$ prior $\alpha=(1.1, 1.1, 1.1)$ the learning generalization ability will be quite bad, as it is very hard for the model to learn the latent structure as it updated every time. The model will also be quite bad at generalization due to randomness of the prior. 3. $T=\infty$, prior $\alpha = (10.,1.,1.)$, the learning and generalization ability will be great, as the task is very straight forward with $\mathbf{p}$ is almost $(1,0,0)$ and do not have to reset. 4. $T=\infty$, prior $\alpha = (1.1,1.1,1.1)$, the learning ability will be good with sufficient unrolling length. However generalization ability will not be as good, since our model could give a poor result with a very different $\mathbf{p}$ caused by the $\alpha=(1.1, 1.1, 1.1)$ prior.

- Does this increase or descrease the complexity of the prediction problem? What about the ability to generalize to unseen $\textbf{p}$?

Answer: For larger T, the complexity of prediction problem is increased. And the prediction/generalization ability on unseen $\textbf{p}$ is quite bad.

2) **Time to check your intuitions (10 pts)**

Implement a similar LSTM model as in PART 1. This will take as input a one-hot description of the obsevation ($[1,0,0,0]$ for symbol 1. $[0,1,0,0]$ for symbol 2, $[0,0,1,0]$ for symbol 3, $[0,0,0,1]$ for the RESET symbol). This input is fed into a 32-unit LSTM and LSTM output is processed as before: $\textrm{Relu} \Rightarrow \textrm{Fully connected} \Rightarrow \textrm{Relu} \Rightarrow \textrm{Fully connected} \Rightarrow \textrm{Output}$. The model will be trained, as before, by cross-entropy on predicting the next symbol. You will notice that the setup is really similar to the previous tasks, so feel free to re-use whenever appropiate.

Train the following models:

- T = 5, and T=20 with the data generated from a Dirichlet with $\alpha = (10.,1.,1.)$. Unrolling length for the LSTM = 100. Minibatch size = 64. (M1, M2)
- T = 5, and T=20 with the data generated from a Dirichlet with$\alpha = (1.3, 1.3, 1.3)$ Unrolling length for the LSTM = 100. Minibatch size = 64. (M3, M4)

Train the models for 1000 iterations (1000 minibatches). Record the training and testing preformance (every 10-20 iterations). Plot the curves over training time. What do you observe? (Is this curve smooth? Do any of them plateau?). **[2x5 pts]**

3) **Analysis results (10 pts)**

In this section, we will investigate what the models have actaully learnt. For this we will generate a few test sequences:

- *Test sequence 1*: generate a test sequence that changes tasks every T=5 steps from a a Dirichlet with $\alpha = (10.,1.,1.)$.
- *Test sequence 2*: generate a test sequence that changes tasks every T=5 steps from a a Dirichlet with $\alpha = (1.3,1.3,1.3)$.
- *Test sequence 3*: generate a test sequence that 'changes tasks' every T=5 steps, but keep sampling according to the same probability vector $\textbf{p}=(1,0,0)$ (You can use any of the extreme here).

i) Test the preformance of M1 and M3 and these test sequences. In addition plot the actual prediction the models do (probability of symbols over time). This should give you more insight in what the model does. Does this correspond or contradict your previous intuitions? **[5 pts]**

ii) Repeat the same procedure for task length $T=20$ and models M2 and M4. What do you observe? How do M2 and M4 compare to each other and how to their compare to M1 and M3 (the models trained on the shorter task length). **[5 pts]**

4) **Comparison to the Bayesian update (15 pts)**

Going back to the generative process in the task description. For a given prior, for each the mini-tasks (selecting/sampling a $\textbf{p}$), one could compute the Bayesian posterior at each time step. We start with a prior and every time we observe a symbol with update our posterior over the parameters $\textbf{p}$ given the data. We do this every time step, till we reach the RESET symbol which marks the end of the task. Then we start again, from the prior.

i) Derive the posterior update for each time step. (Hint: since the two distribution are conjugates or each other, the posterior has a closed form). **[3 pts]**

Answer: Let $\textbf{x}$ be the current symbol, whose i-th element x_i is binary. From the Bayes formula, the posterior is:

$$\begin{split} \mathbb{p}(\mathbf{p} \mid \mathbf{x}) & \propto \mathbb{p}(\mathbf{x} \mid \mathbf{p})\mathbb{p}(\mathbf{p} \mid \mathbf{\alpha}) \\ & \propto \prod_{i=1}^{3}p_i^{\mathbb{1}(x_i=1)} \prod_{i=1}^{3}p_i^{\alpha_i-1} \\ & = \prod_{i=1}^{3}p_i^{\mathbb{1}(x_i=1) + \alpha_i - 1} \end{split}$$

Where the posterior follows $Dirichlet \big(\mathbb{1}(x_i=1) + \alpha_i\big)$, where $\mathbb{1}(\cdot)$ is the indication function.

ii) Implement this posterior update and use it to infer the probabilities over the next symbol, for the previously generated test sequences. This will tell you, what the inferred probabilities would be, if we knew the structure of the model, the prior and that the reset symbol means the tasks has finished and we should reset our estimate to the prior. (For test sequence 1 and 2, use the prior that generated them, for test sequence 3 compute the updates starting from both priors) **[5 pts]**

iii) Compare this to what the LSTM predictions are. What do you observe? What are the failure cases -- can you explain why this might happen? (For test sequence 1 and 2, use the prior that generated them, for test sequence 3 compute the updates starting from both priors). **[7 pts]**

For this analysis, only consider $T=20$ and respectively models M2 and M4.

5) **Play (not for credit, just for fun)**

Visualize the hidden state of the LSTM. Look at the eigenvalues: How many of these are actual revelant? What do they correspond to?

In [0]:
```
#@title Generate data function

n_symbols = 3
def get_data_per_task(number_samples_per_task=10, p=None, alpha=None):
  if p == None:
    # sample task
    if alpha == None:
      p = np.random.dirichlet((1.3, 1.3, 1.3), 1)[0]
    else:
      p = np.random.dirichlet(alpha, 1)[0]

  p = np.append(p, [0])
  sample = np.random.multinomial(1, p, size = number_samples_per_task)

  sample = np.append(sample,[[0,0,0,1]], axis=0)
  return sample


def get_data(ntasks, nsamples_per_task, p=None, alpha=None):
  sample = []
  for task_id in range(ntasks):
    sample.append(get_data_per_task(number_samples_per_task = nsamples_per_task, p=p, alpha=alpha))
  return np.concatenate(sample)


def get_minibatch(batch_size, ntasks, nsamples_per_task, p=None, alpha=None):
  sample = get_data(batch_size*ntasks, nsamples_per_task, p=p, alpha=alpha)
  return np.reshape(sample, [batch_size, ntasks*(nsamples_per_task+1), n_symbols+1])

```

Training your recurrent model

In [0]:
```
class task2Model(object):
    def __init__(self, batch_size = 64, num_rnn_layers=1, num_units=32,
                 fully_connected_hidden_units=64, learning_rate=0.01):
        self.batch_size = batch_size
        self.num_rnn_layers=num_rnn_layers
        self.num_units = num_units
        self.fully_connected_hidden_units = fully_connected_hidden_units
        self.learning_rate = learning_rate
        self.setGraph()

    def setGraph(self):
        with tf.device('/device:GPU:*'):
          tf.reset_default_graph()
          self.X = tf.placeholder(tf.float32, [None, 99 ,4])
          self.Y = tf.placeholder(tf.float32, [None, 99, 4])
          self.rnnAndLinear()
          self.lossAndTrain()

    def rnnAndLinear(self):
        RNN_layers = [tf.contrib.rnn.BasicLSTMCell(size) for size in ([self.num_units] * self.num_rnn_layers)]
        self.cell = tf.contrib.rnn.MultiRNNCell(RNN_layers)
        rnn_output, _ = tf.nn.dynamic_rnn(self.cell, self.X, dtype=tf.float32)
        # rnn_output: 64 * 4 * 32
        self.rnn_output = tf.reshape(rnn_output, [-1, self.num_units])
        # self.rnn_output: (64 x 4) * 32
        hid_1_output = tf.contrib.layers.fully_connected(self.rnn_output, self.fully_connected_hidden_units, tf.nn.relu)
        hid_2_output = tf.contrib.layers.fully_connected(hid_1_output, self.fully_connected_hidden_units, tf.nn.relu)
        self.logits = tf.contrib.layers.fully_connected(hid_2_output, 4, None)


    def lossAndTrain(self):
        y_reshape = tf.reshape(self.Y,(-1,4))
        self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_reshape,logits=self.logits))
        self.optimizer = tf.train.AdamOptimizer().minimize(self.loss)

```

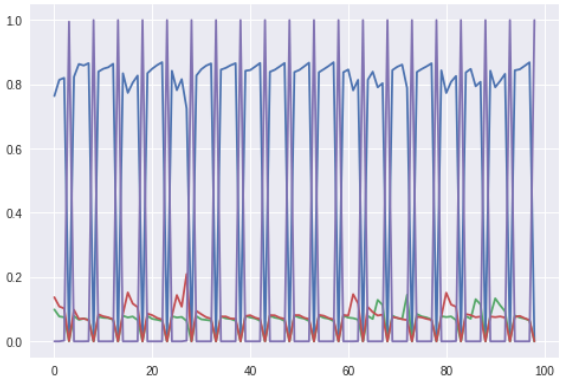
In [0]:

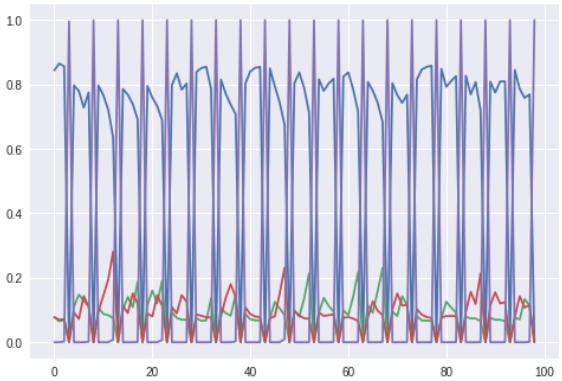
```
def task2Experiment(n,T,a,p=None):
    Model=task2Model()
    print(' ============= start generating data ============')
    train_data_list = [get_minibatch(batch_size=64, ntasks=n, nsamples_per_task=(T-1), p=None, alpha=a) for i in range(0,1000)]
    train_data = np.concatenate(train_data_list)
    print('train_data:', train_data.shape)
    test_data=get_minibatch(batch_size=10000, ntasks=n, nsamples_per_task=(T-1), p=None, alpha=a)
    print(' ============ data generating finished ============= ')
    test_list=[]
    train_list=[]
    new_test=[]
    new_test.append(get_minibatch(batch_size=1, ntasks=n, nsamples_per_task=(T-1), p=None, alpha=(10.,1.,1.)))
    new_test.append(get_minibatch(batch_size=1, ntasks=n, nsamples_per_task=(T-1), p=None, alpha=(1.3,1.3,1.3)))
    new_test.append(get_minibatch(batch_size=1, ntasks=n, nsamples_per_task=(T-1), p = (1.,0.,0.)))
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for iteration in range(0,1000):
            X_mb = train_data_list[iteration][:,:-1,:]
            Y_mb = train_data_list[iteration][:,1:,:]
            _, loss_v = sess.run([Model.optimizer, Model.loss], {Model.X: X_mb, Model.Y: Y_mb})

            if iteration%20==0:
                train_ls = sess.run(Model.loss, {Model.X: train_data[:,:-1,:], Model.Y: train_data[:,1:,:]})
                train_list.append(train_ls)
                test_ls = sess.run(Model.loss, {Model.X: test_data[:,:-1,:], Model.Y: test_data[:,1:,:]})
                test_list.append(test_ls)

        for num,i in enumerate(new_test):
            prediction=sess.run(Model.logits, {Model.X: i[:,:-1,:], Model.Y: i[:,1:,:]})
            prediction_reshape = np.exp(prediction)/np.sum(np.exp(prediction),axis=1).reshape(99,1)
            for k in range(4):
                plt.plot(prediction_reshape[:,k])
            plt.show()
    return train_list, test_list
```

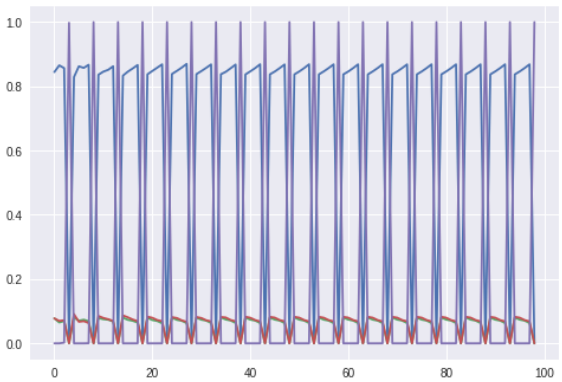
Analysing your recurrent model

In [174]:
```
1 train_list_1, test_list_1 = task2Experiment(n=20,T=5,a=(10.,1.,1.),p=None)
```

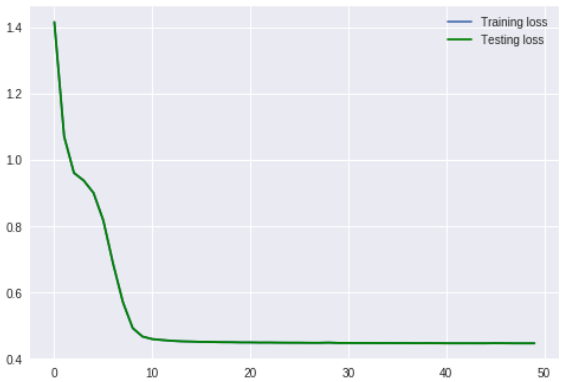
```
 ============= start generating data ============
train_data: (64000, 100, 4)
 ============ data generating finished =============
```





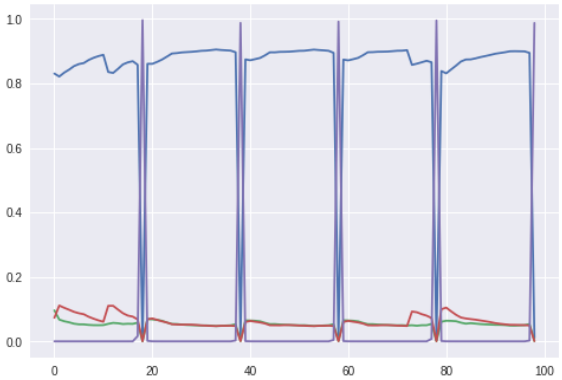


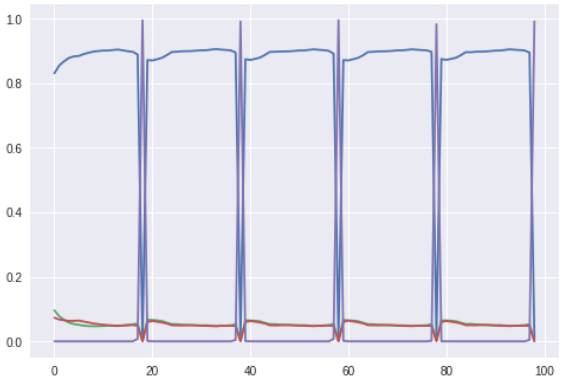
In [175]:
```
1 plot_learning_curves(train_list_1, test_list_1)
```



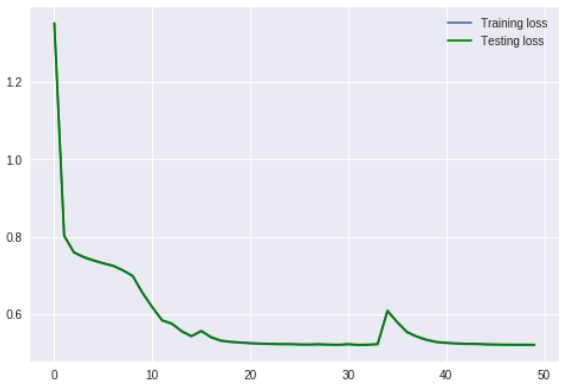
In [188]:
```
1 train_list_2, test_list_2 = task2Experiment(n=5,T=20,a=(10.,1.,1.),p=None)
```

```
 ============= start generating data ============
train_data: (64000, 100, 4)
 ============ data generating finished =============
```



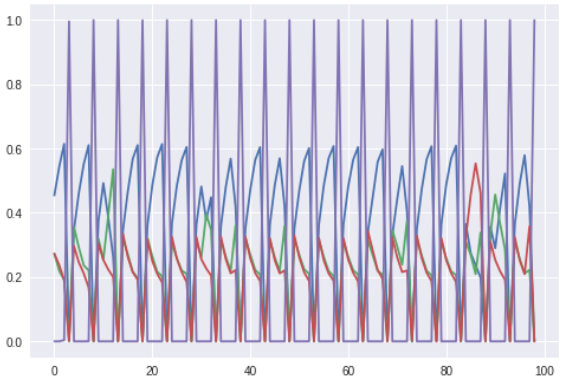


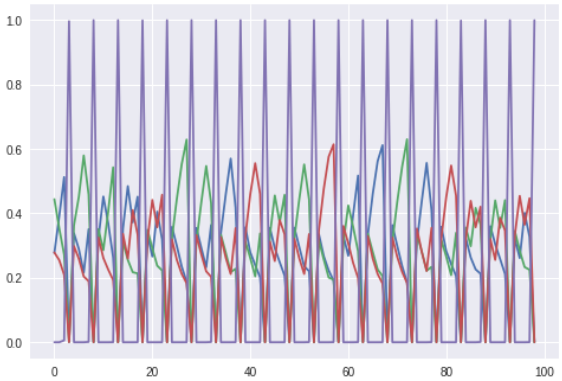
In [189]:
```
1 plot_learning_curves(train_list_2, test_list_2)
```

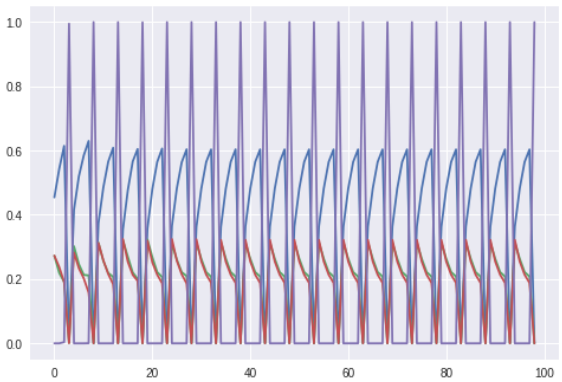


In [190]:
```
1 train_list_3, test_list_3 = task2Experiment(n=20,T=5,a=(1.3,1.3,1.3),p=None)
```

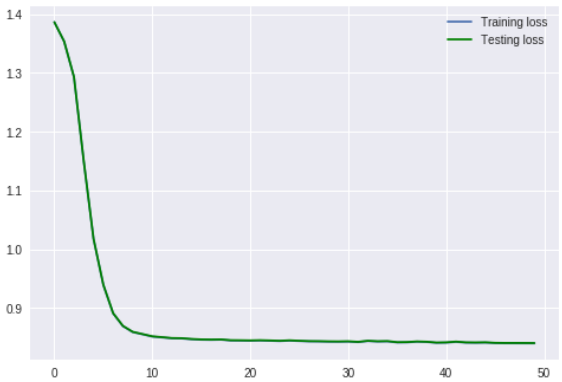
```
 ============= start generating data ============
train_data: (64000, 100, 4)
 ============ data generating finished =============
```





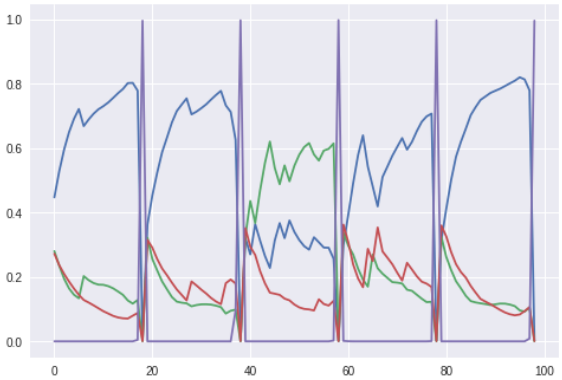


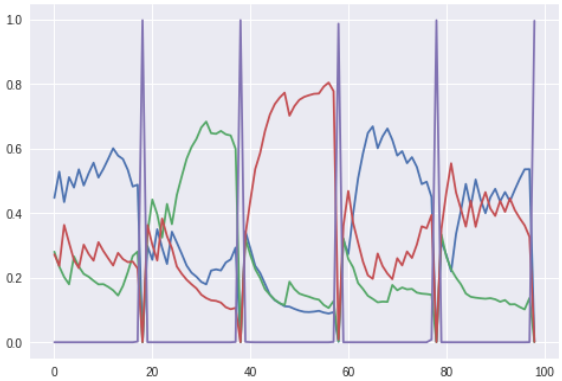
In [191]:
```
1 plot_learning_curves(train_list_3, test_list_3)
```

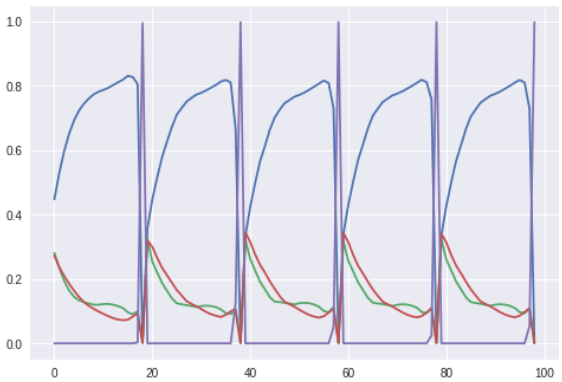


In [192]:
```
1 train_list_4, test_list_4 = task2Experiment(n=5,T=20,a=(1.3,1.3,1.3),p=None)
```

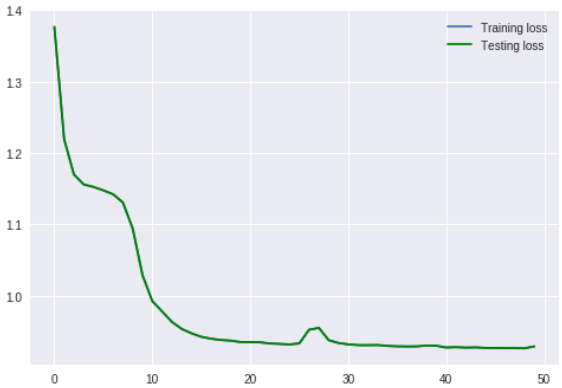
```
 ============ start generating data ============
train_data: (64000, 100, 4)
 ============ data generating finished =============
```







In [193]:
```
1 plot_learning_curves(train_list_4, test_list_4)
```



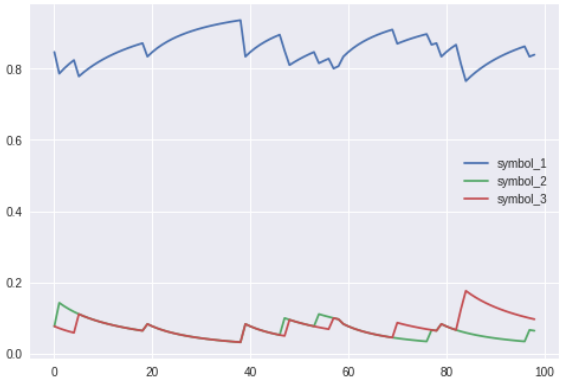
Note that the learning curves have overlapped

Bayesian Updates

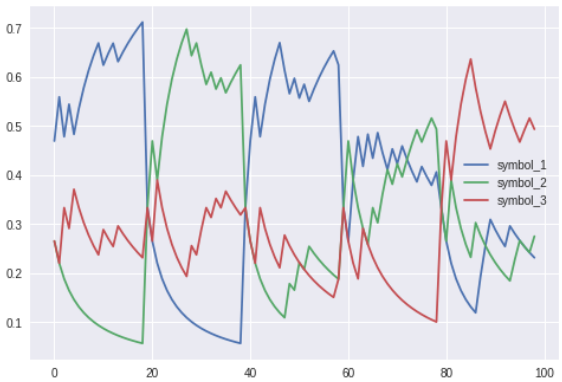
In [0]:
```
# Implement Bayesian update (as if you knew the 'right' prior and model)

def bayesianUpdates(sample_data, alpha):
    result=[]
    alpha_copy = np.copy(alpha)
    sample_data_reshape = sample_data.reshape(100,4)
    for i in range(0,99):
        sample = sample_data_reshape[i]
        if sample[-1] == 1:
            alpha_copy = np.copy(alpha)
            result.append(alpha_copy / np.sum(alpha_copy))
        elif sample[-1] == 0:
            alpha_copy + =sample[:-1]
            result.append(np.copy(alpha_copy / np.sum(alpha_copy)))
    posterior = np.asmatrix(result)
    return posterior
```

In [0]:
```
alpha_1 = np.array([10.,1.,1.])
alpha_2 = np.array([1.3,1.3,1.3])

test_1 = get_minibatch(1, 5, 19, alpha = (10.,1.,1.))
test_2 = get_minibatch(1, 5, 19, alpha = (1.3,1.3,1.3))
test_3 = get_minibatch(1, 5, 19, p = (1.,0.,0.))
```

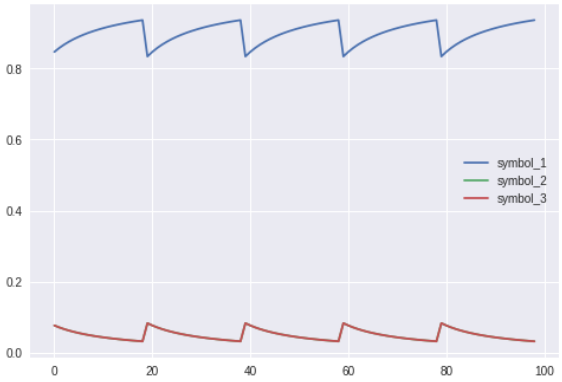
In [217]:
```
bayesian_result = bayesianUpdates(test_1, np.array([10.,1.,1.]))
for k in range(3):
  plt.plot(bayesian_result[:,k], label = 'symbol_' + str(k+1))
plt.legend()
plt.show()
```



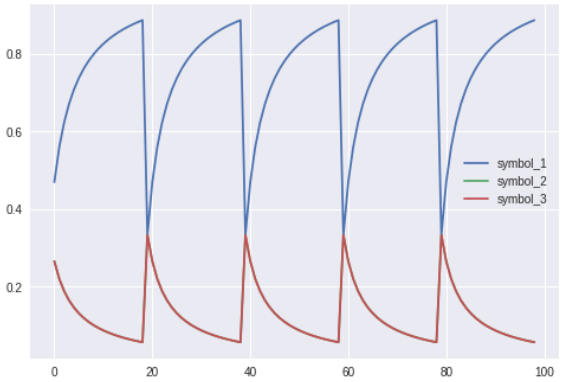
In [218]:
```
bayesian_result = bayesianUpdates(test_2, np.array([1.3,1.3,1.3]))
for k in range(3):
  plt.plot(bayesian_result[:,k], label = 'symbol_' + str(k+1))
plt.legend()
plt.show()

```



In [219]:
```
bayesian_result = bayesianUpdates(test_3, np.array([10.,1.,1.]))
for k in range(3):
  plt.plot(bayesian_result[:,k], label = 'symbol_' + str(k+1))
plt.legend()
plt.show()
```



In [220]:
```
bayesian_result = bayesianUpdates(test_3, np.array([1.3,1.3,1.3]))
for k in range(3):
  plt.plot(bayesian_result[:,k], label = 'symbol_' + str(k+1))
plt.legend()
plt.show()
```



In [0]:
```
1
```

In [47]:

```
class painting(object):
    def __init__(self, batch_size=256, num_rnn_layers=1, num_units=32,
                 fully_connected_hidden_units=64, num_epochs=5, learning_rate=0.01,gradient_clip=False, model="lstm"):

        self.batch_size = batch_size
        self.num_rnn_layers=num_rnn_layers
        self.num_units = num_units
        self.fully_connected_hidden_units = fully_connected_hidden_units
        self.num_epochs = num_epochs
        self.learning_rate = learning_rate
        self.gradient_clip=gradient_clip
        self.model = model
        self.build_model()

    def rnn_input(self):
        self.X, self.Y = get_placeholders()

    def rnn_layer(self):

        if self.model == "lstm":
            RNN_layers = [tf.contrib.rnn.BasicLSTMCell(size) for size in ([self.num_units] * self.num_rnn_layers)]
        elif self.model == "gru":
            RNN_layers = [tf.contrib.rnn.GRUCell(size) for size in ([self.num_units] * self.num_rnn_layers)]
        else:
            raise Exception("Model error")

        self.cell = tf.contrib.rnn.MultiRNNCell(RNN_layers)

        x_rnn_input=tf.reshape(self.X,[-1,783,1])
        rnn_output, final_state = tf.nn.dynamic_rnn(self.cell, x_rnn_input, dtype=tf.float32)
        self.rnn_output = tf.reshape(rnn_output, [-1, self.num_units])

    def ff_layer(self,ff_layer_input):
        with tf.variable_scope("feedforward", reuse=tf.AUTO_REUSE):
            h1 = tf.contrib.layers.fully_connected(ff_layer_input, self.fully_connected_hidden_units, tf.nn.relu)
            h2 = tf.contrib.layers.fully_connected(h1, self.fully_connected_hidden_units, tf.nn.relu)
            prob = tf.contrib.layers.fully_connected(h2, 1, tf.nn.sigmoid)
            self.ff_layer_output=tf.reshape(prob,[-1,1])

    def loss(self):
        self.train_result=tf.reshape(self.ff_layer_output,[-1,783])
        self.loss = -tf.reduce_mean(
            self.Y * tf.log(self.train_result) + (1 - self.Y) * tf.log(1 - self.train_result))

    def train(self):
        if self.gradient_clip:
            t_vars = tf.trainable_variables()
            grad_clip = 5
            grads, _ = tf.clip_by_global_norm(tf.gradients(self.loss, t_vars), grad_clip)
            optimization = tf.train.AdamOptimizer(self.learning_rate)
            self.optimizer = optimization.apply_gradients(zip(grads, t_vars))
        else:
            self.optimizer = tf.train.AdamOptimizer().minimize(self.loss)

    def next_pixel_prediction(self):
        self.test_input=tf.placeholder(tf.float32, [self.batch_size, 1])

        self.test_initial_state= self.cell.zero_state(self.batch_size, dtype=tf.float32)
        self.cell_output, self.state_output=self.cell(self.test_input, self.test_initial_state)
        self.ff_layer(self.cell_output)
        self.pixel_prediction=self.ff_layer_output

    def build_model(self):
        with tf.device('/device:GPU:*'):
            tf.reset_default_graph()
            with tf.name_scope("rnn_input"):
                self.rnn_input()

            with tf.name_scope("rnn_layer"):
                self.rnn_layer()

            with tf.name_scope("ff_layer"):
                self.ff_layer(self.rnn_output)

            with tf.name_scope("loss"):
                self.loss()

            with tf.name_scope("train"):
                self.train()

            with tf.name_scope("test"):
                self.next_pixel_prediction()
```

In [5]:

```
#@title Import libraries
#@test {"output": "ignore"}

# Import useful libraries.
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data

# Binarize the images
def binarize(images, threshold=0.1):
  return (threshold < images).astype('float32')

# Import dataset with one-hot encoding of the class labels.
def get_data():
  return input_data.read_data_sets("MNIST_data/", one_hot=True)

# Placeholders to feed train and test data into the graph.
# Since batch dimension is 'None', we can reuse them both for train and eval.
def get_placeholders():
  x = tf.placeholder(tf.float32, [None, 783])
  y = tf.placeholder(tf.float32, [None, 783])
  return x, y

# Generate summary table of results. This function expects a dict with the
# following structure: keys of 'LSTM' (or 'GRU') and the values for each key are a
# list of tuples consisting of (test_loss, test_accuracy), and the list is
# ordered as the results from 0 epoch (beginning of training), 1 epoch, 5 epochs (or end of training):
# {
#  'LSTM': [(loss,acc), (loss, acc), (loss, acc)]
# }
def plot_summary_table(experiment_results):
  # Fill Data.
  cell_text = []
  columns = ['(Beginning - 0 epochs)', '(Mid-training - 1 epoch)', '(End of training - 5 epochs)']
  for k, v in experiment_results.items():
    rows = ['Test loss', 'Test accuracy']
    cell_text=[[],[]]
    for (l, _) in v:
      cell_text[0].append(str(l))
    for (_, a) in v:
      cell_text[1].append(str(a))

    fig=plt.figure(frameon=False)
    ax = plt.gca()
    the_table = ax.table(
      cellText=cell_text,
      rowLabels=rows,
      colLabels=columns,
      loc='center')
    the_table.scale(2, 8)
    # Prettify.
    ax.patch.set_facecolor('None')
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.text(-0.73, 0.9, k, fontsize=18)



def plot_learning_curves(training_loss, testing_loss):
  plt.figure()
  plt.plot(training_loss)
  plt.plot(testing_loss, 'g')
  plt.legend(['Training loss', 'Testing loss'])

```

In [6]:

```
def show_imgae(partial,position='end'):
    show=np.zeros((28*28))
    length=len(partial)
    if position=='end':
        show[-length:]=partial
    elif position=='begin':
        show[:length]=partial
    plt.imshow(show.reshape((28,28)), cmap="gray")
    plt.show()
```

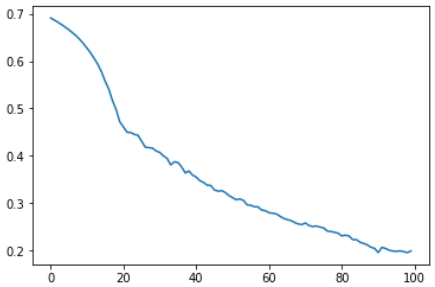
In [47]:

```
class painting(object):
    def __init__(self, batch_size=256, num_rnn_layers=1, num_units=32,
                 fully_connected_hidden_units=64, num_epochs=5, learning_rate=0.01,gradient_clip=False, model="lstm"):

        self.batch_size = batch_size
        self.num_rnn_layers=num_rnn_layers
        self.num_units = num_units
        self.fully_connected_hidden_units = fully_connected_hidden_units
        self.num_epochs = num_epochs
        self.learning_rate = learning_rate
        self.gradient_clip=gradient_clip
        self.model = model
        self.build_model()

    def rnn_input(self):
        self.X, self.Y = get_placeholders()

    def rnn_layer(self):

        if self.model == "lstm":
            RNN_layers = [tf.contrib.rnn.BasicLSTMCell(size) for size in ([self.num_units] * self.num_rnn_layers)]
        elif self.model == "gru":
            RNN_layers = [tf.contrib.rnn.GRUCell(size) for size in ([self.num_units] * self.num_rnn_layers)]
        else:
            raise Exception("Model error")

        self.cell = tf.contrib.rnn.MultiRNNCell(RNN_layers)

        x_rnn_input=tf.reshape(self.X,[-1,783,1])
        rnn_output, final_state = tf.nn.dynamic_rnn(self.cell, x_rnn_input, dtype=tf.float32)
        self.rnn_output = tf.reshape(rnn_output, [-1, self.num_units])

    def ff_layer(self,ff_layer_input):
        with tf.variable_scope("feedforward", reuse=tf.AUTO_REUSE):
            h1 = tf.contrib.layers.fully_connected(ff_layer_input, self.fully_connected_hidden_units, tf.nn.relu)
            h2 = tf.contrib.layers.fully_connected(h1, self.fully_connected_hidden_units, tf.nn.relu)
            prob = tf.contrib.layers.fully_connected(h2, 1, tf.nn.sigmoid)
            self.ff_layer_output=tf.reshape(prob,[-1,1])

    def loss(self):
        self.train_result=tf.reshape(self.ff_layer_output,[-1,783])
        self.loss = -tf.reduce_mean(
            self.Y * tf.log(self.train_result) + (1 - self.Y) * tf.log(1 - self.train_result))

    def train(self):
        if self.gradient_clip:
            t_vars = tf.trainable_variables()
            grad_clip = 5
            grads, _ = tf.clip_by_global_norm(tf.gradients(self.loss, t_vars), grad_clip)
            optimization = tf.train.AdamOptimizer(self.learning_rate)
            self.optimizer = optimization.apply_gradients(zip(grads, t_vars))
        else:
            self.optimizer = tf.train.AdamOptimizer().minimize(self.loss)

    def next_pixel_prediction(self):
        self.test_input=tf.placeholder(tf.float32, [self.batch_size, 1])

        self.test_initial_state= self.cell.zero_state(self.batch_size, dtype=tf.float32)
        self.cell_output, self.state_output=self.cell(self.test_input, self.test_initial_state)
        self.ff_layer(self.cell_output)
        self.pixel_prediction=self.ff_layer_output

    def build_model(self):
        with tf.device('/device:GPU:*'):
            tf.reset_default_graph()
            with tf.name_scope("rnn_input"):
                self.rnn_input()

            with tf.name_scope("rnn_layer"):
                self.rnn_layer()

            with tf.name_scope("ff_layer"):
                self.ff_layer(self.rnn_output)

            with tf.name_scope("loss"):
                self.loss()

            with tf.name_scope("train"):
                self.train()

            with tf.name_scope("test"):
                self.next_pixel_prediction()
```

In [8]:
```
def run():
    mymodel = painting()
    mnist = get_data()

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        saver = tf.train.Saver(tf.global_variables())

        iteration=0
        loss_list=[]
        while mnist.train.epochs_completed < mymodel.num_epochs:
            iteration+=1
            x_mb, _ = mnist.train.next_batch(mymodel.batch_size)
            b_x_mb=binarize(x_mb)
            x_mb=b_x_mb[:,:-1]
            y_mb=b_x_mb[:,1:]

            _, loss_v = sess.run([mymodel.optimizer, mymodel.loss], {mymodel.X: x_mb, mymodel.Y: y_mb})
            loss_list.append(loss_v)

            if iteration%100==0:
                plt.plot(loss_list)
                plt.show()

        path='./tmp/dl14'+str(mnist.train.epochs_completed)+'.ckpt'
        save_path = saver.save(sess, path)
        print("Model saved in path: %s" % save_path)
```

In [48]:
```
run()
```

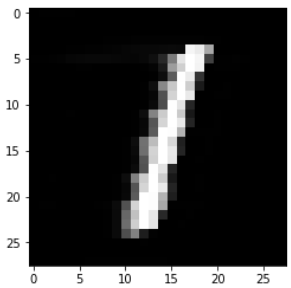
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```



In [50]:
```
def test_model():
    mymodel = painting()
    mnist = get_data()
    with tf.Session() as sess:
        saver = tf.train.Saver(tf.global_variables())
        print('restore model')
        path='./model_store/dl45.ckpt'
        saver.restore(sess, path)
        b_test=binarize(mnist.test.images)
        test_x=b_test[:,:-1]
        test_y=b_test[:,1:]
        y_show=sess.run(mymodel.train_result,{mymodel.X: test_x, mymodel.Y: test_y})
        show_imgae(y_show[2])
```

In [51]:
```
test_model()
```

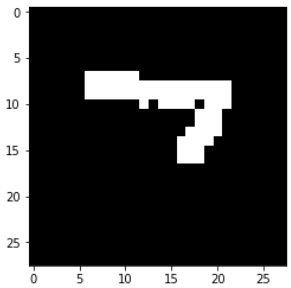
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./tmp/dl45.ckpt
```

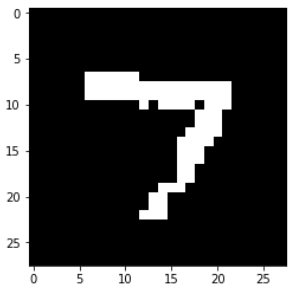


In [52]:
```
def pixel_prediction_task():
    mymodel = painting(batch_size=100)
    eval_mnist=get_data()
    dataset=binarize(eval_mnist.test.images[0:100])
    in_painting_dataset=[]
    for i in dataset:
        in_painting_dataset.append(i[:-300])
    in_painting_dataset=np.asarray(in_painting_dataset)
    mnist = get_data()
    with tf.Session() as sess:
        saver = tf.train.Saver(tf.global_variables())
        print('restore model')
        path='./model_store/dl45.ckpt'
        saver.restore(sess, path)

        b_test=in_painting_dataset
        test_x=b_test[:,:-1]
        test_y=b_test[:,1:]

        np_state_output=sess.run(mymodel.test_initial_state)
        w=in_painting_dataset
        show_imgae(final_painting[0],position='begin')
        for i in range(0,483):
            test_input=test_x[:,i].reshape((-1,1))
            feed={mymodel.test_input:test_input, mymodel.test_initial_state:np_state_output}
            pixel_prediction, np_state_output=sess.run([mymodel.pixel_prediction, mymodel.state_output],feed_dict=feed)

        for i in range(0,300):
            feed={mymodel.test_input:pixel_prediction, mymodel.test_initial_state:np_state_output}
            pixel_prediction, np_state_output=sess.run([mymodel.pixel_prediction, mymodel.state_output],feed_dict=feed)
            pixel_prediction=[np.random.choice(2, p=[1 - p, p]) for p in pixel_prediction[:,0]]
            pixel_prediction=np.asmatrix(pixel_prediction).reshape(-1,1)

            final_painting=np.concatenate((final_painting, pixel_prediction),axis=1)

        plt.imshow(final_painting[0].reshape((28,28)), cmap="gray")
        plt.show()
```

In [54]:
```
pixel_prediction_task()
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
restore model
INFO:tensorflow:Restoring parameters from ./tmp/dl45.ckpt
```





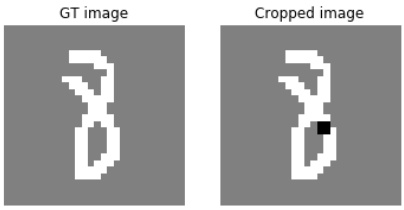
In [13]:
```
# Downloading the inpainting datasets
!git clone https://github.com/dianaborsa/compgi22_dl_cw4.git
```

```
fatal: destination path 'compgi22_dl_cw4' already exists and is not an empty directory.
```

In [14]:
```
# Load the dataset (2X2)
dataset = np.load('compgi22_dl_cw4/2X2_pixels_inpainting.npy')
```

In [56]:
```
# checking loading
images    = dataset[0]
gt_images = dataset[1]

nSamples, ndim = gt_images.shape
print('Loaded dataset has {} samples: cropped + GT'.format(nSamples))

# randomly visualize a few samples
for SampleID in np.random.randint(nSamples,size=1):
    plt.figure()
    plt.subplot(1,2,1)
    plt.imshow(np.reshape(gt_images[SampleID],(28,28)), interpolation='None',vmin=-1, vmax=1, cmap="gray")
    plt.title("GT image")
    plt.grid(False)
    plt.axis('off')
    plt.subplot(1,2,2)
    plt.imshow(np.reshape(images[SampleID],(28,28)), interpolation='None',vmin=-1, vmax=1, cmap="gray")
    plt.title("Cropped image")
    plt.grid(False)
    plt.axis('off')
    plt.show()
```

```
Loaded dataset has 1000 samples: cropped + GT
```



In [114]:
```
def per(n):
    result=[]
    for i in range(1<<n):
        s=bin(i)[2:]
        s='0'*(n-len(s))+s
        result.append(s)
    return result
```

```
['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000', '1001', '1010', '1011', '1100', '1101', '1110', '1111']
```

In [166]:
```
print(per(1))
```

```
['0', '1']
```

In [190]:
```


def build_enumerator(sample,mode):
    if mode=='one':
        combination=per(1)
    elif mode=='patch':
        combination=per(4)
    location=np.where(sample==-1)[0]
    sample_set=[0]
    result=[]
    for i in combination:
        tmp=np.copy(sample)
        for num,j in enumerate(location):
            tmp[j]=int(i[num])
        result.append(tmp)
    sample_set=np.asmatrix(result)

    ## test:
#     for i in location:
#         print(sample[i])
#     for i  in result:
#         for j in location:
#             print (i[j])
#    print(np.shape(sample_set))
    return sample_set
```

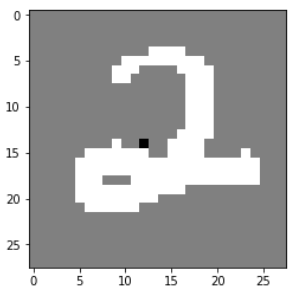
In [199]:

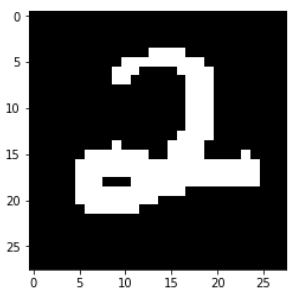
```
def pixel_inpainting(mode='one'):
    if mode=='one':
        mymodel = painting(batch_size=2)
        dataset = np.load('compgi22_dl_cw4/one_pixel_inpainting.npy')
        log_joint=np.zeros((2)).reshape(2,1)
    elif mode=='patch':
        mymodel = painting(batch_size=16)
        dataset = np.load('compgi22_dl_cw4/2X2_pixels_inpainting.npy')
        log_joint=np.zeros((16)).reshape(16,1)
    else:
        raise Exception("Model error")

    images    = dataset[0]
    gt_images = dataset[1]
    sample=images[0]
    sample_set=build_enumerator(sample,mode)
    print('sample_set:',np.shape(sample_set))
    with tf.Session() as sess:
        saver = tf.train.Saver(tf.global_variables())
        print('restore model')
        path='./model_store/dl45.ckpt'
        saver.restore(sess, path)

#         print(np.shape(sample_set[:,0].reshape((-1,1))))
        np_state_output=sess.run(mymodel.test_initial_state)
        for i in range(0,783):
            test_input=sample_set[:,i].reshape((-1,1))
            feed={mymodel.test_input:test_input, mymodel.test_initial_state:np_state_output}
            pixel_prediction, np_state_output=sess.run([mymodel.pixel_prediction, mymodel.state_output],feed_dict=feed)
            log_joint+=np.log(pixel_prediction)*sample[i+1]+np.log(1-pixel_prediction)*(1-sample[i+1])
        prob=np.exp(log_joint)/np.sum(np.exp(log_joint))
        print('log_joint:',log_joint)
        print('prob',prob)
        painted_sample=np.copy(sample)
        location=np.where(sample==-1)[0]
        if mode=='one':
            paint_pixel=per(1)[np.argmax(np.array(prob))]
#             print(paint_pixel)
        if mode=='patch':
            paint_pixel=per(4)[np.argmax(np.array(prob))]
#             print(paint_pixel)

        for num,i in enumerate(location):
            painted_sample[i]=int(paint_pixel[num])

        plt.imshow(sample.reshape((28,28)), cmap="gray")
        plt.show()
        plt.imshow(painted_sample.reshape((28,28)), cmap="gray")
        plt.show()

```

In [197]:

```
pixel_inpainting(mode='one')
```

```
sample_set: (2, 784)
restore model
INFO:tensorflow:Restoring parameters from ./tmp/dl45.ckpt
log_joint: [[-92.36534959]
 [-95.70196214]]
prob [[ 0.9656637]
 [ 0.0343363]]
0
```





In [200]:
```
1 pixel_inpainting(mode='patch')
```

```
sample_set: (16, 784)
restore model
INFO:tensorflow:Restoring parameters from ./tmp/dl45.ckpt
log_joint: [[-55.47631782]
 [-51.7958291 ]
 [-63.15168292]
 [-60.13713465]
 [-60.86023732]
 [-57.29839357]
 [-67.51671798]
 [-64.39266136]
 [-71.32512059]
 [-67.45864211]
 [-78.65284252]
 [-75.47843574]
 [-71.45256035]
 [-67.31310425]
 [-80.21810664]
 [-76.70012997]]
prob [[  2.44845206e-02]
 [  9.71197481e-01]
 [  1.13637821e-05]
 [  2.31592545e-04]
 [  1.12378978e-04]
 [  3.95889664e-03]
 [  1.44481506e-07]
 [  3.28528315e-06]
 [  3.20511796e-09]
 [  1.53120834e-07]
 [  2.10598281e-12]
 [  5.03595881e-11]
 [  2.82161421e-09]
 [  1.77108977e-07]
 [  4.40219471e-13]
 [  1.48425016e-11]]
```