DRAW Experiment Results Bella Godiva 20214805

1. Cluttered MNIST classification (Quantitative experiment)

a. DRAW

```
Epoch 47                Training Loss: 2.1274484783411025               Validation Loss: 7.001629670461019
Epoch 48                Training Loss: 2.1202082163095475               Validation Loss: 7.2932664553324384
Epoch 49                Training Loss: 2.1106819766759872               Validation Loss: 6.787922382354736
Epoch 50                Training Loss: 2.097898497581482                Validation Loss: 6.882350444793701
```

In [57]:
```
model_test = DrawModel(T,A,B,z_size,read_size,write_size,dec_size,enc_size,attention)
model_test.cuda()
model_test.load_state_dict(torch.load('/Users/ivy2021/Documents/DRAW/class_model/saved_model.pth'))
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data, labels in test_loader:
        bs = data.size()[0]
        data = data.view(bs, -1)
        labels = torch.squeeze(labels)
        if torch.cuda.is_available():
            data, labels = data.cuda(), labels.cuda()
        # calculate outputs by running images through the network
        outputs = model_test(data)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        predicted.cuda()
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
<ipython-input-52-be383057d346>:112: UserWarning: torch.range is deprecated and will be removed in a future release because its
behavior is inconsistent with Python's range builtin. Instead, use torch.arange, which produces values in [start, end).
  ai = torch.ones((delta.size()[0],A)).to('cuda:0') * torch.range(0,A-1).to('cuda:0')
<ipython-input-52-be383057d346>:113: UserWarning: torch.range is deprecated and will be removed in a future release because its
behavior is inconsistent with Python's range builtin. Instead, use torch.arange, which produces values in [start, end).
  bj = torch.ones((delta.size()[0],B)).to('cuda:0') * torch.range(0,B-1).to('cuda:0')

Accuracy of the network on the 10000 test images: 21 %
```

- 60x60 Cluttered MNIST is used (60k training samples, 10k validation samples, 10k test samples)
- Attention patch: 12x12, glimpses (time sequence t): 8, trained for 50 epochs
- I only use 20k train samples, 3k validation samples, 3k test samples because when I attempted to use the whole number of samples the kernel died during training, which shows that the computation or memory needed is too much for the kernel to handle and hence only 20k train samples are used.
- With 20k training samples and 50 epochs, **DRAW only manages to achieve 21% accuracy**. This training took around one hour. We can see that the loss is still decreasing fine until the last epoch, so perhaps when trained with longer epochs, it will achieve higher accuracy. Also, when trained with 10k training samples and 20 epochs, it only achieves 12%. So, if it can be trained with more than 20k samples, the performance might increase too.
- I think training takes a really long time because of the read attention algorithm where the method I use for creating the gaussian filters despite its simplicity in being understood, is not memory efficient and slow.

b. Convolution 2 layers (comparison)

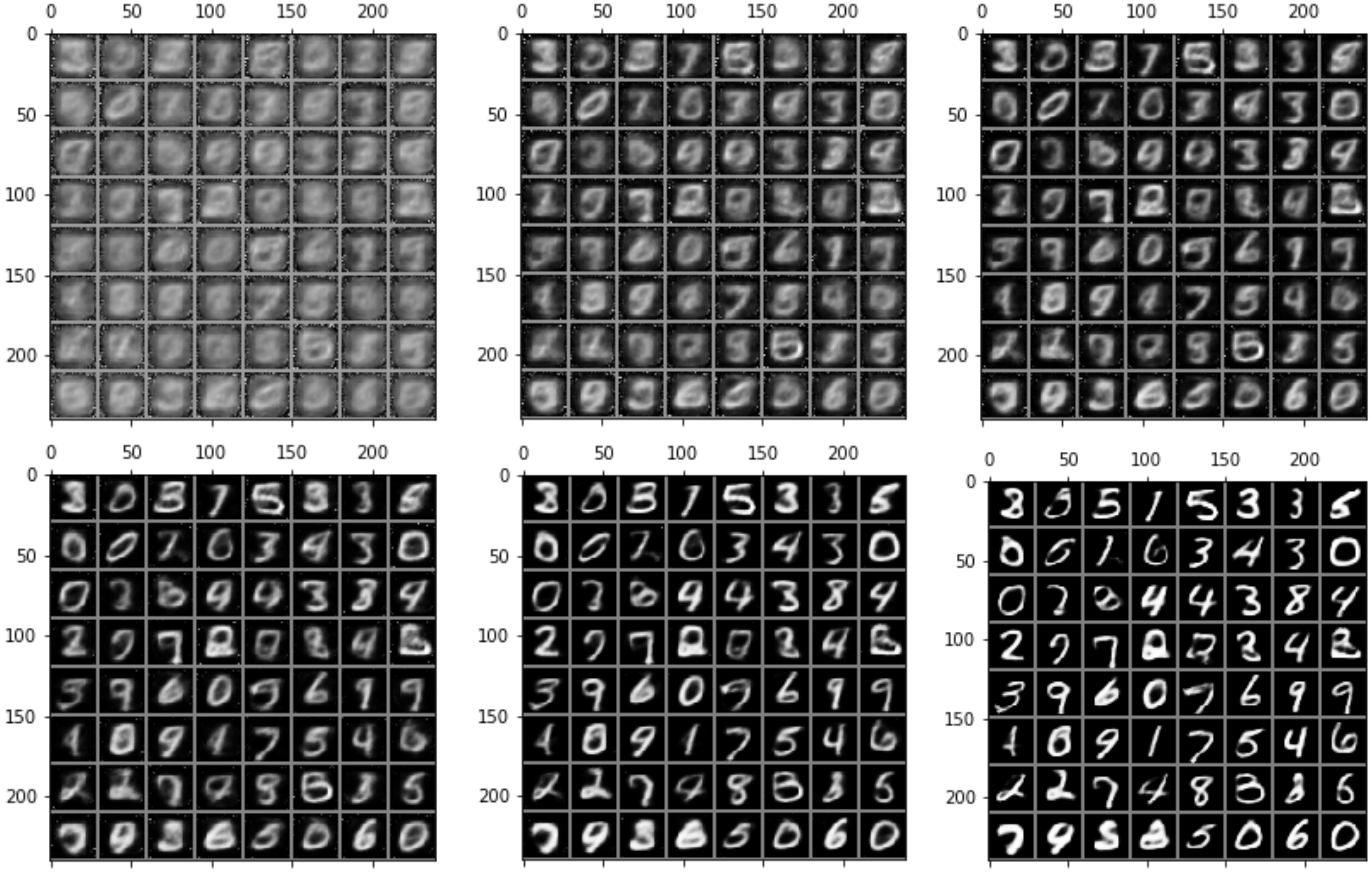
[33]:
```
model_test = Net()
model_test.load_state_dict(torch.load('/Users/bellagodiva/Downloads/DRAW/class_model/save
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data, labels in test_loader:
        bs = data.size()[0]
        data = data.view(bs, 1,60,60)
        labels = torch.squeeze(labels)
        if torch.cuda.is_available():
            data, labels = data.cuda(), labels.cuda()
        # calculate outputs by running images through the network
        outputs = model_test(data)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 53 %
```

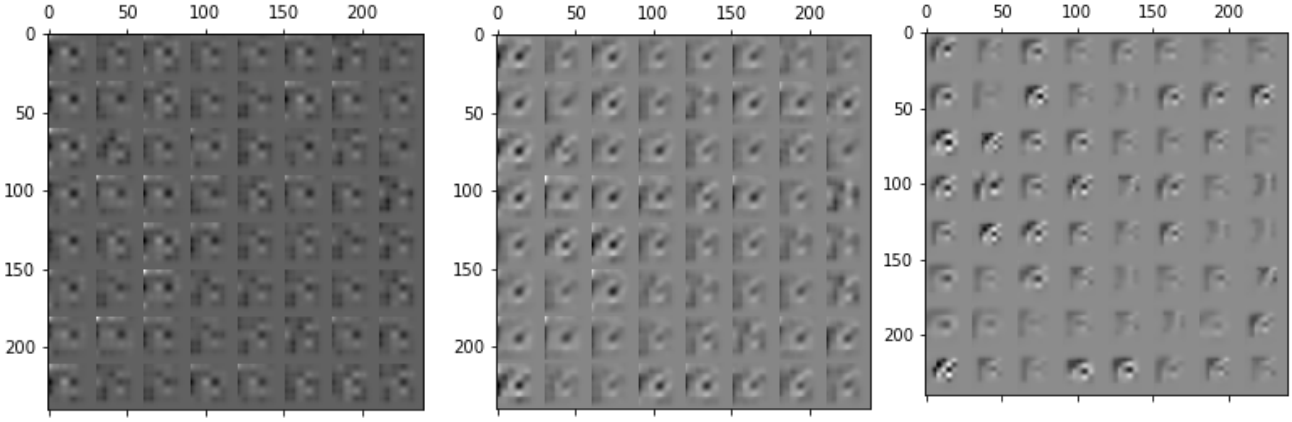
- 60x60 Cluttered MNIST is used
- one layer of 6 5 × 5 filters with stride 1, followed by a fully connected layer with 256 units with RELU after each layer, trained for 20 epochs
- Use all samples (60k train samples, 10k validation samples, 10k test samples)
- **Convolution 2 layer manages to achieve accuracy of 53%** with a much shorter training time needed compared to DRAW

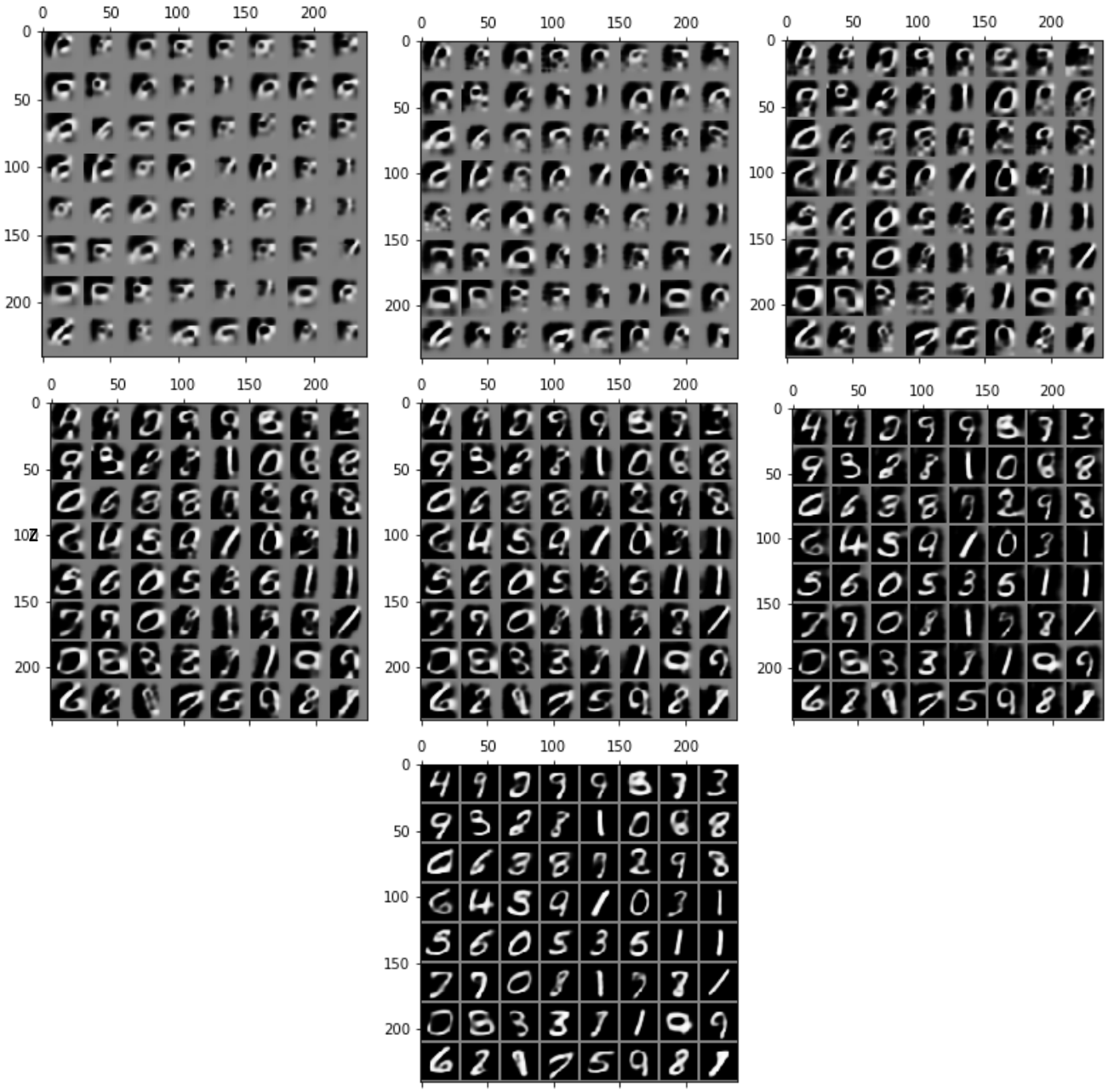
2. Image Generation (Main Result)

MNIST (without attention)



MNIST (with attention**)**

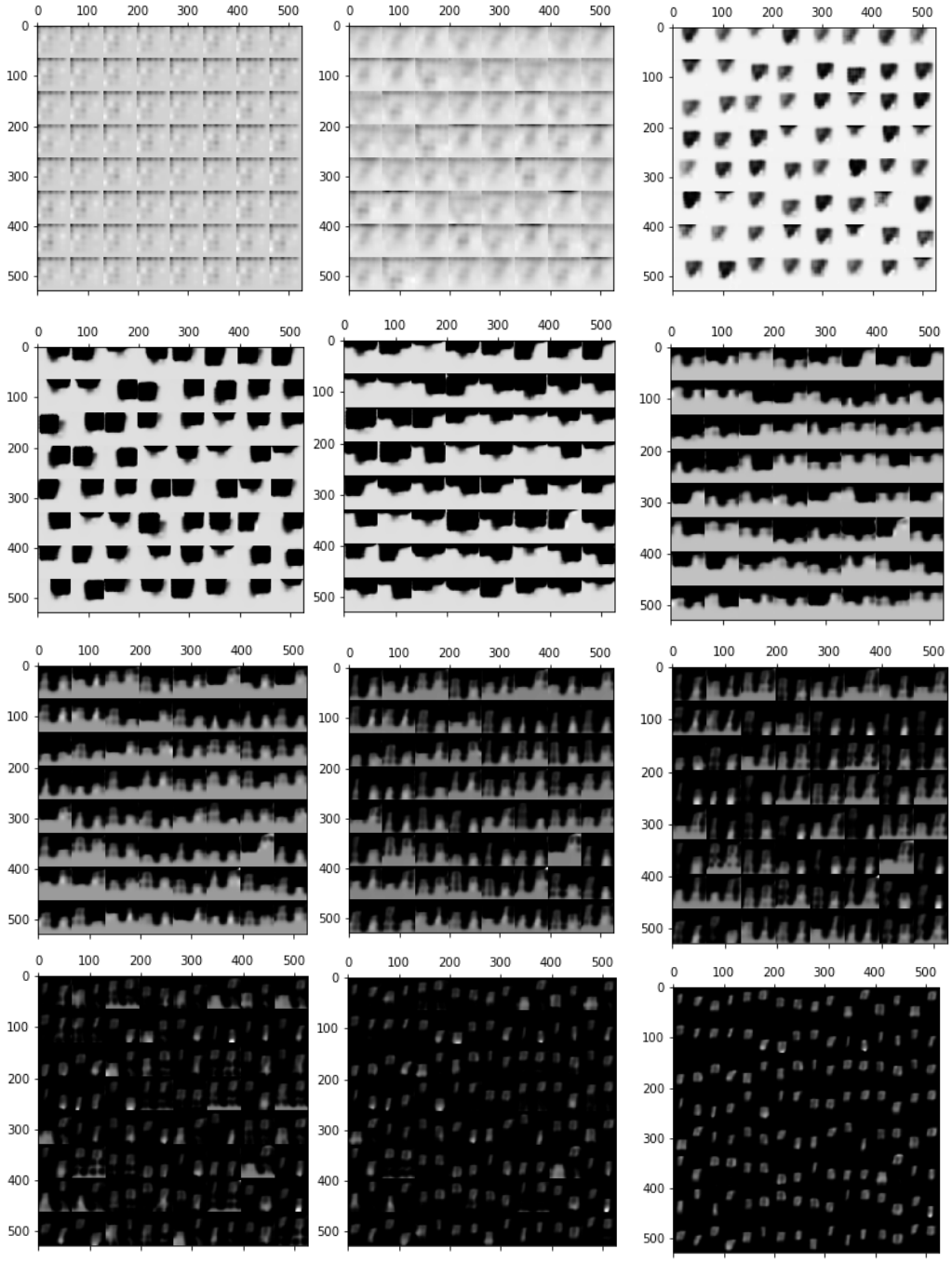




- 28x28 MNIST is used (60k training samples)
- Attention patch: 5x5, glimpses (time sequence t): 10, trained for 20 epochs
- **With and without attention, DRAW manages to successfully generates good quality MNIST handwritten digits.**
- Difference in how DRAW without attention and with attention generate images can be observed from the above generated samples. In DRAW without attention, we can notice that the digits can be identified since first step but in a blurry form. Then, it progressively becomes clearer and clearer. This is because without attention, the network studied the image as a whole and generates image as a whole(28x28). On the other side, we can see how the images generated by DRAW with attention could not be identified during the

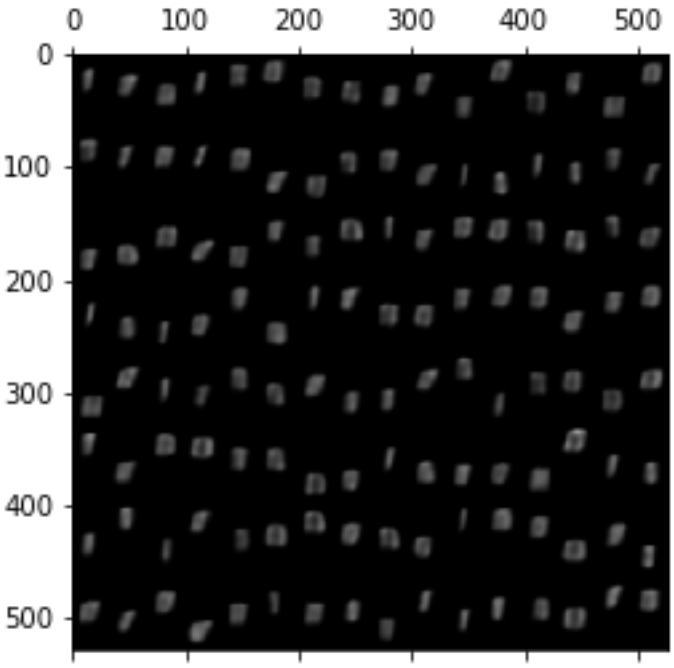
early steps and it progressively add parts of the digits until it builds an identifiable digit in the end. This is because DRAW with attention learn image by small patches from different locations of the image and generates image in small patches and add all those patches to form a whole image.

Double MNIST (with attention)



Training sample

Generated sample



- 64 x 64 DoubleMNIST is used (64k training samples)
- Attention patch: 5x5, glimpses (time sequence t): 15, trained for 20 epochs
- **The generated images are not illegible**. We can only identify two white things with varying positions with black background on all generated images. At least, it is successful in learning the shape pattern, it means there is hope to improve on this.
- To improve on the performance, attention patch and glimpses can be enlarged as pixels of training samples are larger, however this would take a really long time to train.

3 Discussion Points

- How will size of patch (N) and number of iterations (glimpses) affect network's performance?

To improve on the performance, attention patch and glimpses can be enlarged as pixels of training samples are larger, however this would take a really long time to train.

- When generating image, how would the progression of the generated image looks different for algorithm with and without attention? (slide 14)

In DRAW without attention, we can notice that the digits can be identified since first step but in a blurry form. Then, it progressively becomes clearer and clearer. This is because without attention, the network studied the image as a whole and generates image as a whole(28x28). On the other side, we can see how the images generated by DRAW with attention could not be identified during the early steps and it progressively add parts of the digits until it builds an identifiable digit in the end. This is because DRAW with attention learn image by small patches from different locations of the image and generates image in small patches and add all those patches to form a whole image.

- Besides LSTM, what other networks can we use for encoder and decoder?

Any other RNN networks for example GRU