“New Method on Efficiently Sampling using Neural Network with Real NVP”

Durham NC- April 20, 2022 – Nowadays scientists or statisticians are becoming familiar with relying on neural network structure to obtain high accuracy models, and it feels like both supervised modeling and unsupervised modeling are near at hand. Nevertheless, people are still facing many obstacles when implementing unsupervised learning of probabilistic models due to the difficulty of tractability during training. The method we implement in this project, which is designed for unsupervised learning, not only helps on building tractable learning, but can also be used on sampling from input data. The functionality of real-valued non-volume preserving (real NVP) transformations on sampling is critical when we don’t have enough training datasets especially for neural network which normally requires larger volume of inputs than other machine learning algorithms. Another functionality of this transformation is to determine the underlying probability distribution for image generation. We can even further implement this transformation together with Generative adversarial networks.

We attempt to sample from KMINST and USPS dataset. What we do with those traditional datasets are implementing different type of neural networks to predict the label, while in this project, we will be generating digits or hiragana that are different from the original dataset.

The main body of the project will be divided into three parts: Implementing Normal Batch Transformation, Implementing Real NVP Transformation, Generating Sampling Outputs. As the result turns out, the high resolution for generated outputs and the similarity between the original dataset has demonstrated the success of our implementation.

FAQ

April 23, 2022

1 What is the motivation of this project?

We are motivated by the challenge of learning probabilistic distribution as well as generating samples from the underlying distribution. Our goal is to construct a network architecture that allows tractable learning, sampling, inference, and evaluation. Inspired by the work of Dinh, Laurent et al. [1], we implement normal batch transformation and real-valued non-volume preserving (real NVP) transformation to learn from traditional datasets and to generate digits and hiragana that are different from the original ones.

2 What kind of problems in deep learning does the realNVP solve?

The Real NVP method can be implement in unsupervised learning to make the learning process tractable. In addition, this method can be used to solve the problem of lacking of training data as well. We can augment the training data set by sampling from the input data. Moreover, we can apply the Real NVP to determine and generate image from the underlying probability distribution of the training data.

We can also build the neural network to approximate conditional distribution by sampling images given different labels.

3 Are there any potential pitfalls in this project?

We have performed different activation function and obtained various result, as it turns out, we believe the RealNVP algorithm is sensitive to the choice of activation functions.

4 Is there anything to be improve in this project?

The Real NVP network architecture's performance can be further improved. Due to the complexity of the original image in the KMNIST dataset and the activation we imposed in the network, the generated images have darker background and are not very clear in shape. Therefore, we might incorporate some more sophisticated techniques in image modelling and try different activations when designing the Real NVP network architecture in our future work to generate images of better quality. We can also include multi-channel squeezing (if we are doing color images) and multiscale structures. Where the squeezing operatin refers to dividing the image into subsquares and perform reshaping.

5 What tricks / approaches are used to produce feasible and tractable inference on highly nonlinear models in high-dimensional continuous spaces?

A powerful class of bijective functions are defined, and they enable exact and tractable density evaluation and inference. By using the change of variable formula and the fact that the determinant of the triangular matrix produced by the bijective function can be easily computed, feasible and tractable result is obtained.

6 How can samples from the resulting distribution be generated?

This method takes advantages of the change of variable formula and a latent variable. Suppose x is the the sample from resulting distribution that we are after, and Z is a latent variable, given a bijection function f from x to z, we can sample z in the latent space and then evaluate its inverse image which gives us the sample x in the original space.

7 The change of variable formula requires the computation of Jacobian matrix, which is impractical for modeling arbitrary distributions in high-dimensional domain. How is the problem overcomed?

Since the determinant of a triangular matrix could be easily computed (the product of diagonal terms), we could solve this problem by stacking multiple layers of simple bijective functions. These bijective functions are chosen to be simple to invert itself. These are called affine coupling layer.

8 We have implemented batch normalization here. What are some advantages for using batch normalization?

It facilitates the propagation of the training signal. The effects of batch normalization act like a linear rescaling function on the data: $x \mapsto \frac{x-\tilde{\mu}}{\sqrt{\tilde{\sigma}^2+\epsilon}}$ with Jacobian determinant $(\prod_i(\tilde{\sigma}_i+\epsilon))^{-\frac{1}{2}}$,[2]easily computed using the previously discussed property. This allows deeper layers of coupling layer and alleviates the instability when training conditional distributions with a scale parameter thorough gradient-based approach.

9 What is the structure of our implemented neural network?

Our neural network don't have multi-channel structure, so it can only functioning on black-white images. Our structures of neural network for each layer is composed of coupling layers connected with batch norm. We take checkerboard pattern mask for our affine coupling layers.

10 What is the different type of maskings?

There are two types of masking that can be used on exploiting the correlation inside of image structure. One is called spatial checkerboard patterns and the other is called channel-wise mask-

ing. We only use checkerboard patterns here. Inside of the NVP paper, the author uses both masking methods and alternating between them which can help with forward transformation. The author assigns 1 to checkerboard masking when the sum of spatial coordinates is odd, and 0 otherwise [1]. In our implementation, we have changed the masking to a more straight forward way by flatten the input, and masking half part of the input each time.

11 What are some assumptions for NVP

• The model of interest is non-linear and exists in continuous spaces
• We obtain the maximum likelihood by optimizing the log-likelihood, which is further calculated by the change of variable formula
• Real NVP does not rely on a fixed form reconstruction cost which rewards some components over the others

12 What is the advantage of not applying a fixed form reconstruction cost?

The sampling is processed very efficiently since it is paralleled over input dimensions, resulting in globally coherent and sharp samples

13 What are the properties of the scale and translation functions defined in each affine coupling layer?

Since computing the Jacobian and inverse of scale and translation functions are not involved in computing the Jacobian determinant and inverse of the coupling layer, the scale and translation functions can be randomly complex and expensive to invert.

14 What is the limitation of coupling layers and how do we overcome the problem?

The forward transformation of the coupling layers results in parts of the components unchanged. However, this limitation can be overcome by constructing alternating coupling layers so that the unchanged components in one coupling layers can be updated in the next transformation. The Jacobian determinants of the function in the alternating strategy remains tractable with easily computed inverse.

References

[1] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp, 2016.

[2] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation, 2017.

561_Project

April 23, 2022

[1]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
import math
from mpl_toolkits.mplot3d import Axes3D
import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.init as init
import torch.nn.functional as F
import torchvision
import os
import seaborn as sns
import copy

from torch.distributions.multivariate_normal import MultivariateNormal
```

[2]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

1 BatchNorm and NVP Implementation

[3]:
```
class BatchNorm(nn.Module):
    def __init__ (self, dim, momentum=0.9, eps=1e-5):
        super().__init__()
        self.eps = eps
        self.gamma = nn.Parameter(torch.zeros(1, dim))
        self.beta = nn.Parameter(torch.zeros(1, dim))
        self.momentum = momentum

        self.register_buffer('running_mean',torch.zeros(dim))
        self.register_buffer('running_var',torch.ones(dim))


    def forward(self,x):
        if self.training:
            self.mean = x.mean(dim=0)
```

```
            self.running_mean.mul_(self.momentum).add_(self.mean.data*(1-self.
↪momentum))
            self.var = x.var(dim=0,unbiased=False)
            self.running_var.mul_(self.momentum).add_(self.var.data*(1-self.
↪momentum))

            mean = self.mean
            var = self.var
        else:
            mean = self.running_mean
            var = self.running_var

        # According to section 3.7 NVP & MAF EQN22
        x_hat = (x-mean)/torch.sqrt(var+self.eps)
        u = x_hat * torch.exp(self.gamma) + self.beta

        # According to MAF EQN23
        log_det = self.gamma - 0.5 * torch.log(var+self.eps)

        return u, log_det


    def reverse(self,u):
        if self.training:
            mean = x.mean(dim=0)
            var = x.var(dim=0,unbiased=False)
        else:
            mean = self.running_mean
            var = self.running_var

        x_hat = (u-self.beta)*torch.exp(-self.gamma)
        x = x_hat * torch.sqrt(var+self.eps)+mean

        log_det = 0.5 * torch.log(var + self.eps) - self.gamma

        return x, log_det
```

[5]:
```
class Affine_Coupling(nn.Module):
    def __init__(self, input_size, hidden_size, n_hidden, mask,␣
↪cond_label_size=None):
        super().__init__()
        # initiate the mask
        self.register_buffer('mask', mask)
        s_net = [nn.Linear(input_size, hidden_size)]
        for _ in range(n_hidden):
            s_net += [nn.ReLU(), nn.Linear(hidden_size, hidden_size)]
        s_net += [nn.ReLU(), nn.Linear(hidden_size, input_size)]
```

```
        # NN to train the S parameter
        self.s_net = nn.Sequential(*s_net)
        # NN to train the T parameter
        self.t_net = copy.deepcopy(self.s_net)

    def forward(self, x):
        # input x according to masks
        mx = x * self.mask
        s = self.s_net(mx)
        t = self.t_net(mx)
        u = mx + (1 - self.mask) * (x - t) * torch.exp(-s)
        log_abs_det_jacobian = - (1 - self.mask) * s
        return u, log_abs_det_jacobian

    def reverse(self, u):
        # reverse operation
        mu = u * self.mask
        s = self.s_net(mu)
        t = self.t_net(mu)
        x = mu + (1 - self.mask) * (u * s.exp() + t)

        log_abs_det_jacobian = (1 - self.mask) * s

        return x, log_abs_det_jacobian

class RealNVP(nn.Module):
    def __init__(self, n_blocks, input_size, hidden_size, n_hidden):
        super().__init__()

        # initiate mean and variance as 0 and 1 for the distribution we want to␣
 ↪transform to
        self.register_buffer('base_dist_mean', torch.zeros(input_size))
        self.register_buffer('base_dist_var', torch.ones(input_size))

        # initiate Affine Coupling Layer and BatchNorm
        modules = []
        for _ in range(n_blocks):
            mask = torch.ones(input_size).int()
            m_length = 0
            i = 0
            mask_set = []
            while True:
                i += 1
                m_length += math.ceil(input_size * (1 / 2)**i)
                tmp = copy.deepcopy(mask)
                tmp[:m_length] = 0
                mask_set.append(tmp.int())
```

```
                modules += [
                    Affine_Coupling(input_size, hidden_size, n_hidden, tmp.
↪int()),
                    BatchNorm(input_size),
                ]
                if torch.sum(tmp.int()) == 1:
                    break
            for m in reversed(mask_set):
                mask_sym = torch.flip(m, [0])
                modules += [
                    Affine_Coupling(input_size, hidden_size, n_hidden, mask_sym.
↪int()),
                    BatchNorm(input_size),
                ]

        self.net = nn.Sequential(*modules)


    @property
    def base_dist(self):
        # Base distribution
        return torch.distributions.Normal(self.base_dist_mean, self.
↪base_dist_var)

    def forward(self, x):
        # Forward Pass, reshape x because x could be a 3d tensor
        x = x.view(x.shape[0], -1)
        log_det_sum = 0
        for module in self.net:
            x, log_det = module(x)
            log_det_sum = log_det_sum + log_det
        return x, log_det_sum

    def reverse(self, u):
        # reverse transform
        log_det_sum = 0
        for module in reversed(self.net):
            u, log_det = module.reverse(u)
            log_det_sum = log_det_sum + log_det
        return u, log_det_sum
```

2 Apply on KMNIST and USPS Dataset

[10]:
```
batch_size_train = 1000
train_set = torchvision.datasets.KMNIST('KMNIST/raw/train-images-idx3-ubyte',␣
 ↪train=True, download=True,
                              transform=torchvision.transforms.Compose([
                                torchvision.transforms.Resize(16),
                                torchvision.transforms.ToTensor()
                              ]))

train_loader_KMNIST = torch.utils.data.DataLoader(train_set,␣
 ↪batch_size=batch_size_train, shuffle=True)
```

```
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-images-
idx3-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-images-
idx3-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/train-images-
idx3-ubyte.gz
```

```
  0%|          | 0/18165135 [00:00<?, ?it/s]
```

```
Extracting KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/train-images-
idx3-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw

Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-labels-
idx1-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/train-labels-
idx1-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/train-labels-
idx1-ubyte.gz
```

```
  0%|          | 0/29497 [00:00<?, ?it/s]
```

```
Extracting KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/train-labels-
idx1-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw

Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-images-
idx3-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-images-
idx3-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/t10k-images-
idx3-ubyte.gz
```

```
  0%|          | 0/3041136 [00:00<?, ?it/s]
```

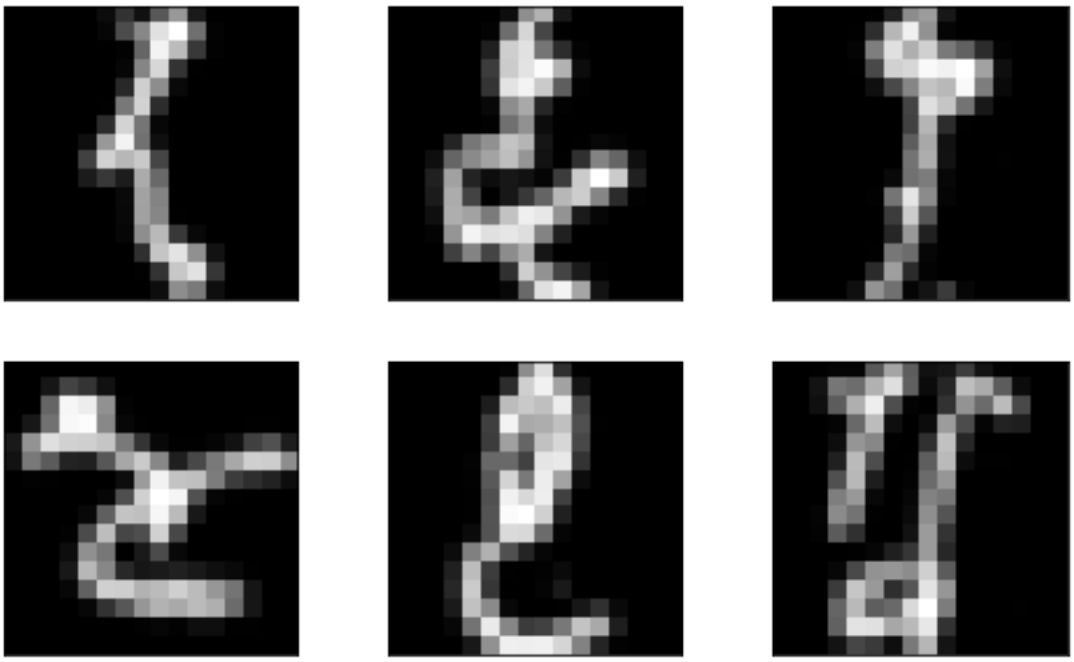
```
Extracting KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/t10k-images-
idx3-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw

Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-labels-
idx1-ubyte.gz
Downloading http://codh.rois.ac.jp/kmnist/dataset/kmnist/t10k-labels-
idx1-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/t10k-labels-
idx1-ubyte.gz
```

```
  0%|          | 0/5120 [00:00<?, ?it/s]
```

```
Extracting KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw/t10k-labels-
idx1-ubyte.gz to KMNIST/raw/train-images-idx3-ubyte/KMNIST/raw
```

[11]:
```
examples = enumerate(train_loader_KMNIST)
batch_idx, (example_data,_) = next(examples)
print(example_data.shape)
fig = plt.figure()
for i in range(min(example_data.shape[0],6)):
    plt.subplot(2,3,i+1)
    plt.tight_layout()
    plt.imshow((example_data[i][0]), cmap='gray', interpolation='none')
    plt.xticks([])
    plt.yticks([])
    fig
```

```
torch.Size([1000, 1, 16, 16])
```

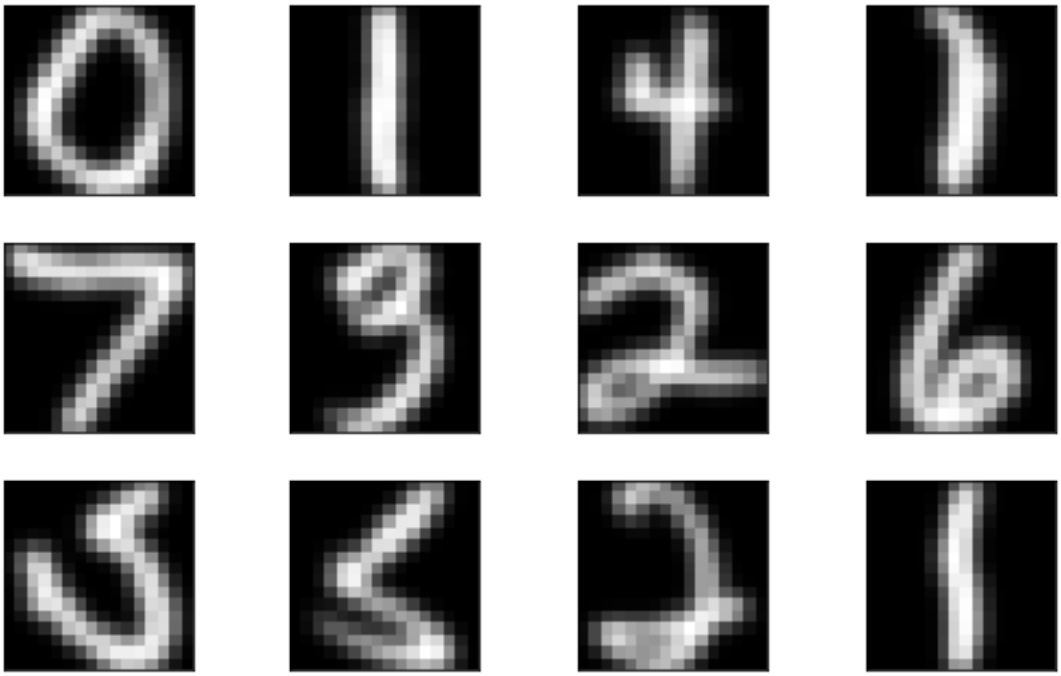


[12]:
```
batch_size_train = 1000
train_set = torchvision.datasets.USPS('USPS', train=True, download=True,
                              transform=torchvision.transforms.Compose([
                                torchvision.transforms.Resize(16),
                                torchvision.transforms.ToTensor()
                              ]))
```

```
train_loader_USPS = torch.utils.data.DataLoader(train_set,␣
 ↪batch_size=batch_size_train, shuffle=True)
```

```
Downloading
https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass/usps.bz2 to
USPS/usps.bz2

  0%|          | 0/6579383 [00:00<?, ?it/s]
```

[21]:
```
examples = enumerate(train_loader_USPS)
batch_idx, (example_data) = next(examples)
fig = plt.figure()
for i in range(12):
    plt.subplot(3,4,i+1)
    plt.tight_layout()
    plt.imshow((example_data[0][i][0]), cmap='gray', interpolation='none')
    plt.xticks([])
    plt.yticks([])
    fig
```



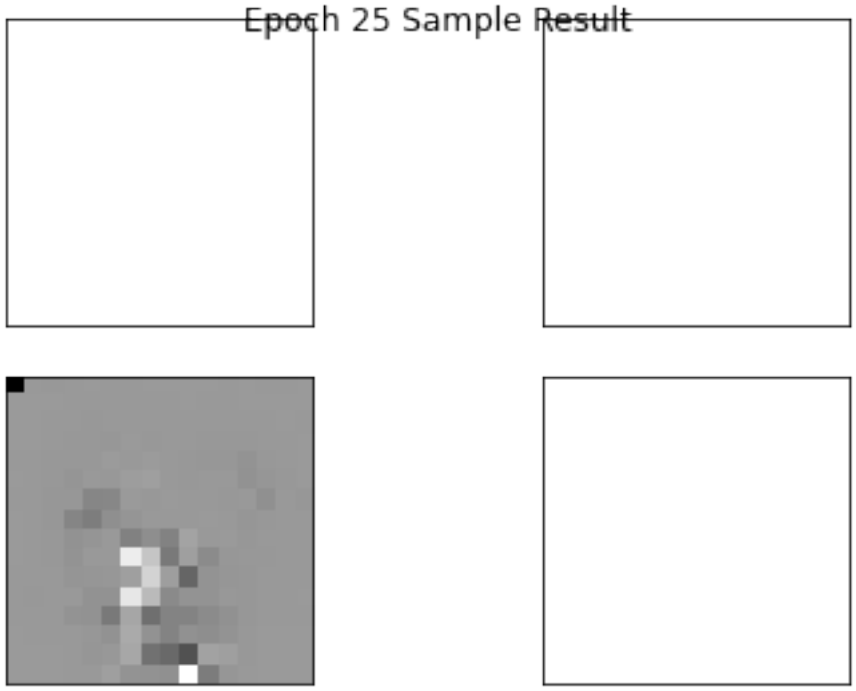
[34]:
```
hidden_dim = 256
input_dim = train_set[0][0].shape[1]*train_set[0][0].shape[2]
nblock = 5
realNVP = RealNVP(nblock,input_dim,hidden_dim,3).to(device)
```

```
optimizer = optim.Adam(realNVP.parameters(), lr = 0.001)


Epochs = 80
realNVP.train()
for epoch in range(Epochs):
    for idx, (data, _) in enumerate(train_loader_KMNIST):
        x = data.to(device)
        z, log_det = realNVP(x)
        loss = -torch.sum(realNVP.base_dist.log_prob(z) + log_det, dim=1).mean(0)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (epoch+1)%25 == 0:
        print(f"Trained Epoch {epoch+1}: loss = {loss.item()}")
        realNVP.eval()
        distrib = MultivariateNormal(torch.zeros(input_dim), torch.
 ↪eye(input_dim))
        z_gen = distrib.rsample(sample_shape=torch.Size([4])).to(device)
        x_gen, _ = realNVP.reverse(z_gen.reshape(z_gen.shape[0], 256))
        x_gen = x_gen.reshape(x_gen.shape[0], 16, 16)
        plt.figure()
        for i in range(4):
            plt.subplot(2,2,i+1)
            plt.tight_layout()
            plt.imshow((x_gen[i].cpu().detach().numpy()), cmap='gray',␣
 ↪interpolation='none')
            plt.xticks([])
            plt.yticks([])
        plt.suptitle(f"Epoch {epoch+1} Sample Result")
        plt.show()
    realNVP.train()
```

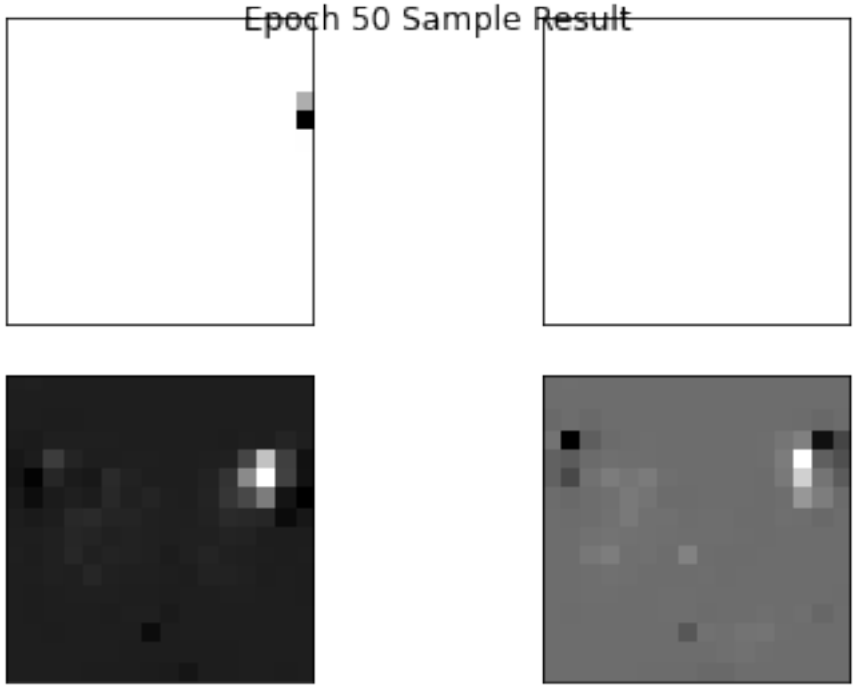
```
Trained Epoch 25: loss = -843.0462646484375
```

```
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:452: UserWarning:
Warning: converting a masked element to nan.
  dv = np.float64(self.norm.vmax) - np.float64(self.norm.vmin)
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:459: UserWarning:
Warning: converting a masked element to nan.
  a_min = np.float64(newmin)
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:464: UserWarning:
Warning: converting a masked element to nan.
  a_max = np.float64(newmax)
<string>:6: UserWarning: Warning: converting a masked element to nan.
/usr/local/lib/python3.7/dist-packages/matplotlib/colors.py:993: UserWarning:
Warning: converting a masked element to nan.
```

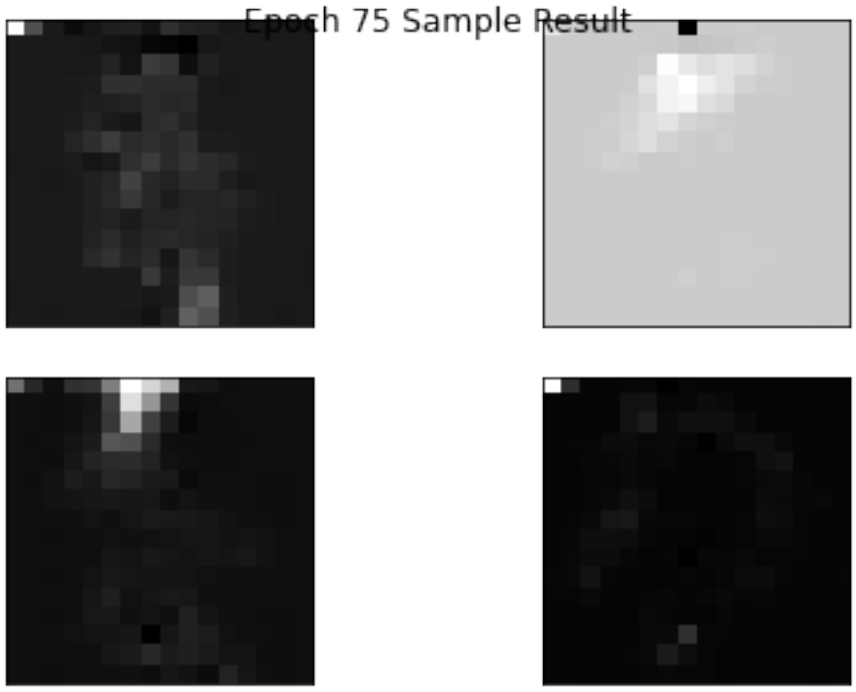
```
data = np.asarray(value)
```




```
Trained Epoch 50: loss = -1048.078125
```



```
Trained Epoch 75: loss = -1071.4827880859375
```

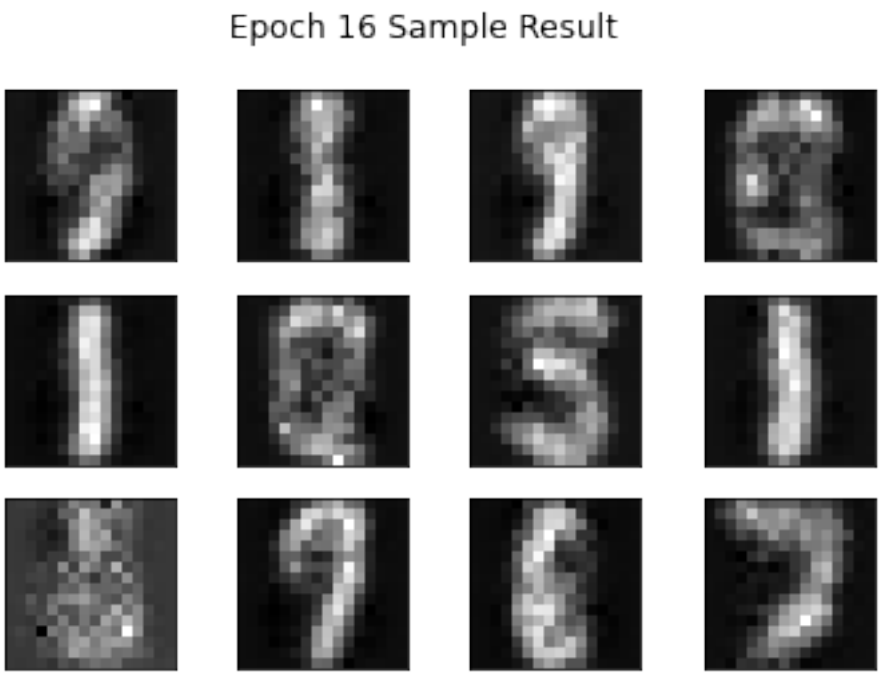


[33]:
```
torch.cuda.empty_cache()
```

[32]:
```
hidden_dim = 256
input_dim = train_set[0][0].shape[1]*train_set[0][0].shape[2]
nblock = 3
realNVP = RealNVP(nblock,input_dim,hidden_dim,3).to(device)
optimizer = optim.Adam(realNVP.parameters(), lr = 0.001)


Epochs = 80
realNVP.train()
for epoch in range(Epochs):
    for idx, (data, _) in enumerate(train_loader_USPS):
        x = data.to(device)
        z, log_det = realNVP(x)
        loss = -torch.sum(realNVP.base_dist.log_prob(z) + log_det, dim=1).mean(0)

        optimizer.zero_grad()
        loss.backward()
```

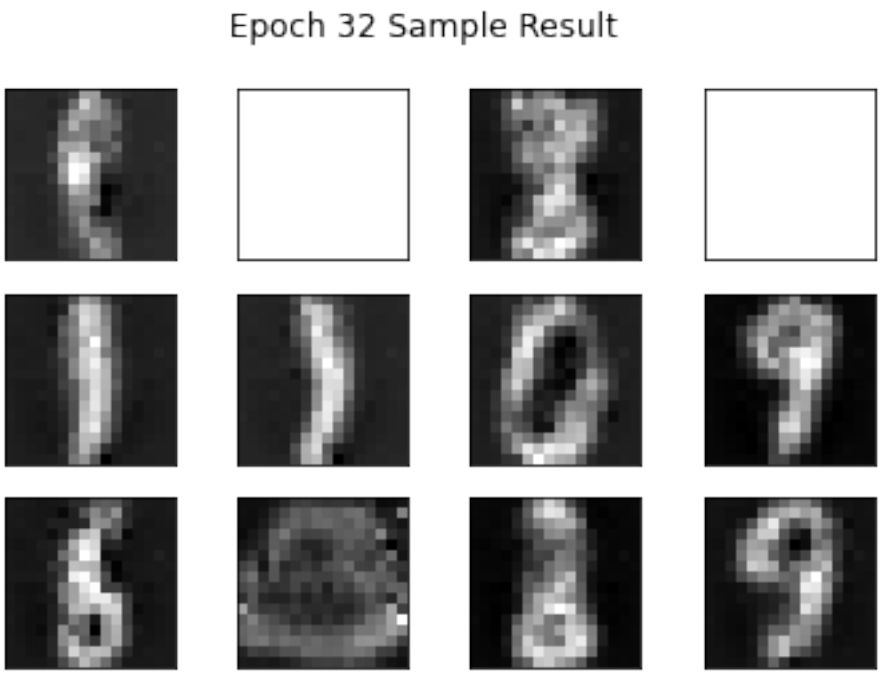
```
        optimizer.step()

    if (epoch+1)%16 == 0:
        realNVP.eval()
        print(f"Trained Epoch {epoch+1}: loss = {loss.item()}")
        distrib = MultivariateNormal(torch.zeros(input_dim), torch.
↪eye(input_dim))
        z_gen = distrib.rsample(sample_shape=torch.Size([12])).to(device)
        x_gen, _ = realNVP.reverse(z_gen.reshape(z_gen.shape[0], 256))
        x_gen = x_gen.reshape(x_gen.shape[0], 16, 16)
        plt.figure()
        for i in range(12):
            plt.subplot(3,4,i+1)
            plt.imshow((x_gen[i].cpu().detach().numpy()), cmap='gray',␣
↪interpolation='none')
            plt.xticks([])
            plt.yticks([])
        plt.suptitle(f"Epoch {epoch+1} Sample Result")
        plt.show()
    realNVP.train()
```

```
Trained Epoch 16: loss = -679.2488403320312
```



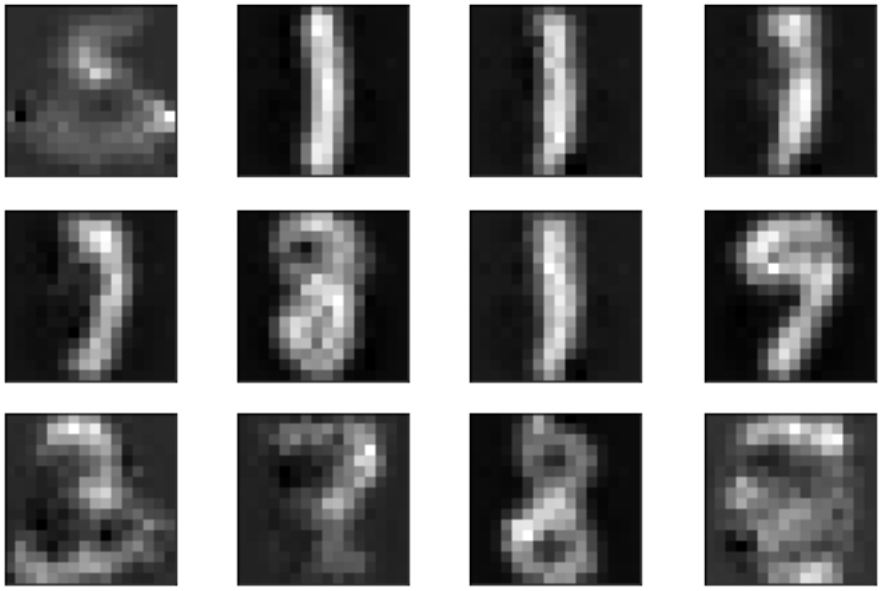

```
Trained Epoch 32: loss = -769.5353393554688
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:452: UserWarning:
```

```
Warning: converting a masked element to nan.
  dv = np.float64(self.norm.vmax) - np.float64(self.norm.vmin)
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:459: UserWarning:
Warning: converting a masked element to nan.
  a_min = np.float64(newmin)
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:464: UserWarning:
Warning: converting a masked element to nan.
  a_max = np.float64(newmax)
<string>:6: UserWarning: Warning: converting a masked element to nan.
/usr/local/lib/python3.7/dist-packages/matplotlib/colors.py:993: UserWarning:
Warning: converting a masked element to nan.
  data = np.asarray(value)
```



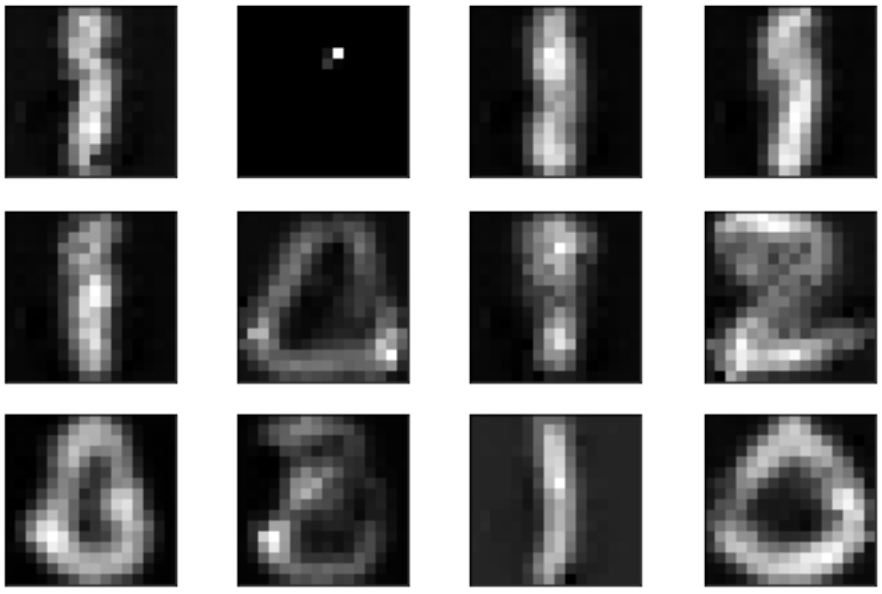

```
Trained Epoch 48: loss = -836.2598876953125
```

Epoch 48 Sample Result

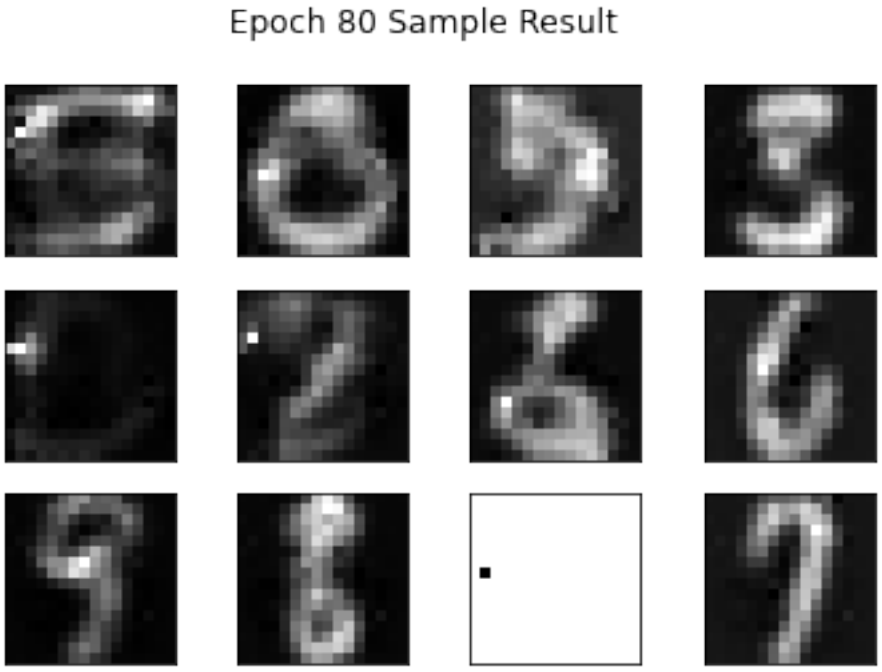


```
Trained Epoch 64: loss = -926.8760986328125
```

Epoch 64 Sample Result



```
Trained Epoch 80: loss = -961.7860107421875
```



We inference the code from the following Github Repository:
https://github.com/kamenbliznashki/normalizing_flows
https://github.com/zhongyuchen/generative-models
https://github.com/tensorflow/probability/tree/v0.15.0/tensorflow_probability/python/bijectors
https://github.com/chrischute/real-nvp/blob/master/models/real_nvp/real_nvp.py