I. INTRODUCTION

The objective of this assignment is to be familiarized with the working principles of variational inference and to use this method for doing approximate inference in a real model. We have used the MNIST [1] dataset to work with our model. Our model is composed of two parts: Encoder and Decoder. The encoder takes an image as input and encodes it into a latent variable while the decoder is designed to decode this variable and generate the same image. Our model is also capable of generating meaningful images using the latent variable. ADAM optimization technique was implemented to optimize the parameters of our model.

II. METHOD AND IMPLEMENTATION

The implementation of the variational autoencoder was done in python language. The lab provided us with a framework in which we were required to implement a few key methods and the ADAM algorithm. Our work can be divided into two tasks: Method implementation and ADAM implementation.

A. Method Implementation

In this section, we have implemented four methods: *sample_latent_variables_from_posterior()*, *bernoulli_log_prob()*, *compute_KL()* and *vae_lower_bound()*. The first method *sample_latent_variables_from_posterior()* implements the reparameterization technique from the lab literature. This method generates latent variable samples from its distribution. We are considering a single Monte Carlo sample per each data point x_i in the batch B. Importantly, we can now separate the randomness in each latent variable z^i from the parameters of $\boldsymbol{q_\phi(z|x_i)}$, which will depend on $\boldsymbol{\phi}$. $\boldsymbol{q_\phi(z|x_i)}$ is the posterior approximation of the latent variable z. We get z^i by using (1).

$$\boldsymbol{z_j^i = \mu_j^\phi(x_i) + \sqrt{v_j^\phi(x_i)}\epsilon_i^j} \tag{1}$$

Where $\boldsymbol{\mu_j^\phi(x_i)}$ and $\boldsymbol{v_j^\phi(x_i)}$ demotes mean and variance of latent variable distribution respectively and $\boldsymbol{\epsilon_i^j \sim N(0,1)}$. In Fig. 1 we can see our python implementation of the reparameterization trick.

```python
# This implements the reparametrization trick

def sample_latent_variables_from_posterior(encoder_output):

    # Params of a diagonal Gaussian.

    D = np.shape(encoder_output)[-1] // 2
    mean, log_std = encoder_output[:, :D], encoder_output[:, D:]

    # TODO use the reparametrization trick to generate one sample from   per each batch datapoint
    # use npr.randn for that.
    # The output of this function is a matrix of size the batch x the number of latent dimensions
    z=[[0]*batch_size]*latent_dim

    var=(np.exp(log_std))**2

    epsilon = npr.randn(batch_size,latent_dim)

    z = mean + np.sqrt(var) * epsilon

    return z
```

Fig. 1. Python implementation of *sample_latent_variables_from_posterior()*

Next, we proceed to design the method called *bernoulli_log_prob()*. It computes the log probability of the generator output. This method actually computes the conditional distribution $P_\theta(x|z)$ using (2).

$$\boldsymbol{p_\theta(x|z) = \prod_{j=1}^{D} x_j \sigma\left(f_j^\theta(z)\right) + (1 - x_j)(1 - \sigma\left(f_j^\theta(z)\right))} \tag{2}$$

where $\boldsymbol{f_j^\theta(z)}$ represents the j-th dimensional output of a deep neural network with parameters
(weights and biases) θ and $\boldsymbol{\sigma(.)}$ is the sigmoid activation function which is given by

$$\boldsymbol{\sigma(x) = \frac{1}{1+\exp(-x)}} \tag{3}$$

Therefore, the distribution of x given z is simply a product of D Bernoulli random variables with activation probability equal to $\boldsymbol{\sigma\left(f_j^\theta(z)\right)}$ for j=1,……,D..Fig. 2 demonstrates the python adaption of this method.

```python
# This evlauates the log of the term that depends on the data

def bernoulli_log_prob(targets, logits):

    # logits are in R
    # Targets must be between 0 and 1

    # TODO compute the log probability of the targets given the generator output specified in logits
    # sum the probabilities across the dimensions of each image in the batch. The output of this function
    # should be a vector of size the batch size

    log_prob = np.sum(targets * np.log(sigmoid(logits)) + (1 - targets) * np.log(1-sigmoid(logits)),axis=1)


    return log_prob
```

Fig. 2. Python implementation of *bernoulli_log_prob()*.

The *compute_KL()* method is designed to compute the Kullback–Leibler (KL) divergence between the posterior approximation q(z|x) and prior p(z). In our model since both distributions are Gaussian in nature, we use (4) to compute the KL divergence.

$$\boldsymbol{KL\left(q_{\phi}(z|x_i)\middle|p(z)\right) = \sum_{j=1}^{L} \frac{1}{2}(v_j^{\phi}(x_i) + \mu_j^{\phi}(x_i)^2 - 1 - log\mu_j^{\phi}(x_i))} \qquad (4)$$

Where L is the dimensionality of the latent variable z.

In Fig. 3 the python implementation of (4) can be seen.

```
# This evaluates the KL between q and the prior

def compute_KL(q_means_and_log_stds):

    D = np.shape(q_means_and_log_stds)[-1] // 2
    mean, log_std = q_means_and_log_stds[:, :D], q_means_and_log_stds[:, D:]
    var=np.exp(log_std)**2
    # TODO compute the KL divergence between q(z|x) and the prior (use a standard Gaussian for the prior)
    # Use the fact that the KL divervence is the sum of KL divergence of the marginals if q and p factorize
    # The output of this function should be a vector of size the batch size
    KL = np.sum(0.5 * (var + (mean ** 2) - 1 - np.log(var) ) , axis=-1)

    return KL
```

Fig. 3. Python implementation of *compute_KL()*.

Finally, we had implemented *vae_lower_bound()*. With this method we compute the lower bound, $\boldsymbol{L(x_i, \theta, \phi)}$ of our generative model $\boldsymbol{\log(p_{\theta}(x_i))}$. The lower bound $\boldsymbol{L(x_i, \theta, \phi)}$ is given in (5).

$$\boldsymbol{L(z, \theta, \phi) = E_{q_{\phi(z|x)}}\left[log\frac{p_{\theta}(x|z)p(z)}{q_{\phi}(z|x)}\right]} \qquad (5)$$

In Fig. 4 the python implementation of (5) is given.

```
# This evaluates the lower bound

def vae_lower_bound(gen_params, rec_params, data):

    # TODO compute a noisy estiamte of the lower bound by using a single Monte Carlo sample:

    # 1 - compute the encoder output using neural_net_predict given the data and rec_params
    # 2 - sample the latent variables associated to the batch in data
    #     (use sample_latent_variables_from_posterior and the encoder output)
    # 3 - use the sampled latent variables to reconstruct the image and to compute the log_prob of the actual data
    #     (use neural_net_predict for that)
    # 4 - compute the KL divergence between q(z|x) and the prior (use compute_KL for that)
    # 5 - return an average estimate (per batch point) of the lower bound by substracting the KL to the data dependent term
    pred=neural_net_predict(rec_params,data)

    latent_var=sample_latent_variables_from_posterior(pred)

    reconstruct=neural_net_predict(gen_params,latent_var)

    log_prob=bernoulli_log_prob(data,reconstruct)

    kld=compute_KL(pred)

    estimate=np.mean(log_prob-kld)


    return estimate
```

Fig. 4. Python implementation of *vae_lower_bound()*.

B. ADAM Implementation

In this section, we have implemented the ADAM algorithm for stochastic optimization of the lower bound on the log marginal likelihood. ADAM is straightforward to implement and is computationally efficient. The ADAM algorithm is displayed in Fig. 5. This algorithm only requires access to a noisy estimate of the gradients of the objective. In Fig. 6 the python adaption of the algorithm is portrayed.

Algorithm 1: g_t indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Fig. 5. The ADAM algorithm for optimizing stochastic objectives.

Usually, ADAM is used to minimize the objective function. However, in our case, we are maximizing the lower bound on the log marginal likelihood. In other words, we are maximizing our objective function. Therefore during update of the parameters, instead of subtracting the adjustment residual, we are adding it to the previous parameter.

```
# ADAM parameters
t = 1

# TODO write here the initial values for the ADAM parameters (including the m and v vectors)
# you can use np.zeros_like(flattened_current_params) to initialize m and v
beta1=0.9
beta2=0.999
eps=10**-8
m=np.zeros_like(flattened_current_params)
v=np.zeros_like(flattened_current_params)


# We do the actual training

for epoch in range(num_epochs):

    elbo_est = 0.0

    for n_batch in range(int(np.ceil(N / batch_size))):

        batch = np.arange(batch_size * n_batch, np.minimum(N, batch_size * (n_batch + 1)))

        # TODO Use the estimated noisy gradient in grad to update the paramters using the ADAM updates

        print(t)
        grad = objective_grad(flattened_current_params)

        m= beta1*m + (1-beta1)*grad

        v=beta2*v+(1-beta2)*(grad**2)

        m_hat=m/(1-(beta1**t))

        v_hat = v/(1-(beta2**t))
        prev_param=flattened_current_params

        flattened_current_params=flattened_current_params+(learning_rate*m_hat)/(np.sqrt(v_hat)+eps)

        elbo_est += objective(flattened_current_params)

        t=t+1
```

Fig. 6. Python implementation of *ADAM Algorithm*

III. RESULTS

A. TASK 3.1

Our first task is generate 25 images from the generative model. This should be done by drawing z from the prior, to then generate x using the conditional distribution $p_\theta(x|z)$. We set the pixel intensity of the image equal to the activation probability. We did not binarize the images. The result is saved into a file using the function *save_images.* The result is shown in Fig. 7.

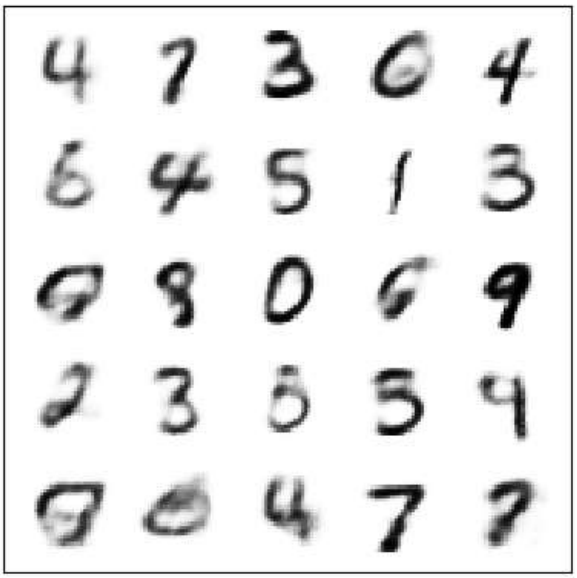

Fig. 7. Images generated from the generative model by using z from the prior

B. TASK 3.2

In this task we were required to generate 10 image reconstructions using the recognition model and then the generative model. We choose the first 10 images from the test set. The reconstructions are obtained by generating z using $\boldsymbol{q_\phi(z|x)}$ and then generating x again using $p_\theta(x|z)$. Like the previous task, we set the pixel intensity of the image equal to the activation probability to avoid binarization of the images. The result is saved into a file using the function *save_images.* The result is shown in Fig. 8.

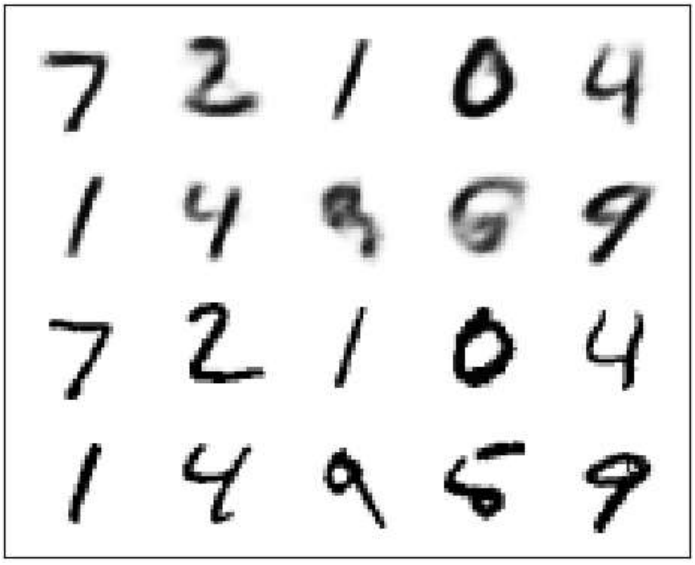

Fig. 8. Images generated using the recognition model and then generative model (First two rows reconstructions, second two rows original images)

C. TASK 3.3

Our final task is to generate 5 interpolations in the latent space from one image to another. We did this by considering the first and second image in the test set, the third and fourth image in the test set and so on. The interpolations is obtained by finding the latent representation of each image. As the latent representation we considered only the mean of predictive model Consider only the mean of the predictive model $\boldsymbol{q(z|x)}$, disregarding the variance.

If we have the latent representation of the first image z_1 and the second image z_2, then the interpolation between these representations will be $\boldsymbol{z^s_{mix} = z_1 s + (1 - s) z_2}$ for $\boldsymbol{s \, \epsilon [0, 1].}$ We have considered a grid of 25 values in the interval [0,1]. Computing the $\boldsymbol{z^s_{mix}}$, we generate the corresponding images using $p_\theta(x|z)$. As usal we did not binarize the images. We set the activation probability as the pixel intensity and saved the images using the function named *save_images*. The results are shown in Fig. 9, 10, 11, 12, and 13.

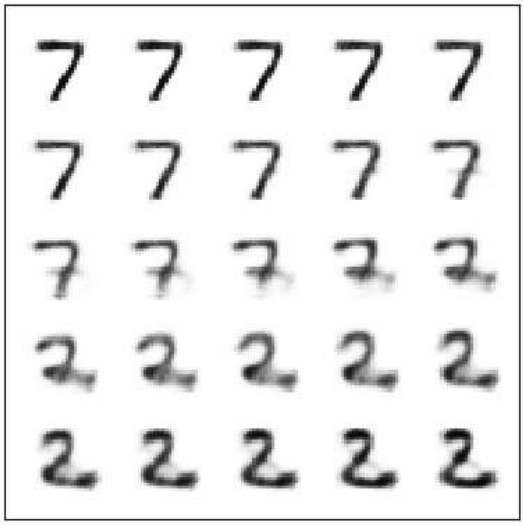

Fig. 9. Interpolated Image (first and second image in the test set)

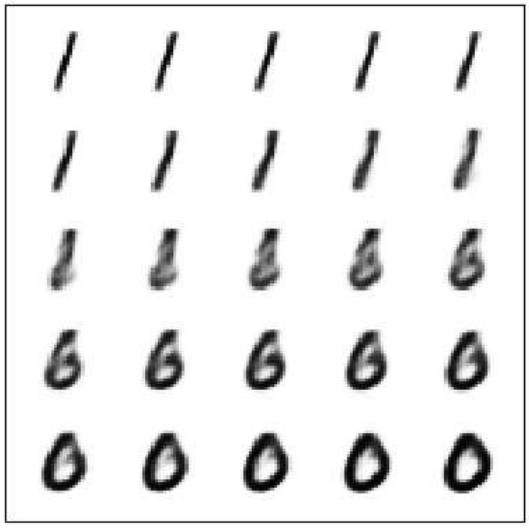

Fig. 10. Interpolated Image (third and fourth image in the test set)

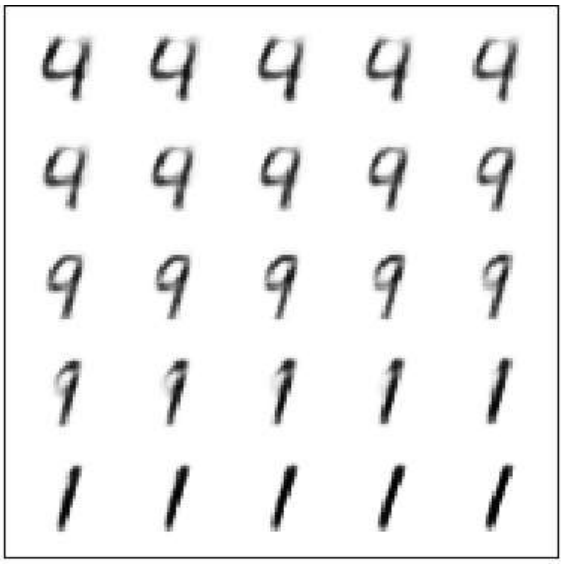

Fig. 11. Interpolated Image (fifth and sixth image in the test set)

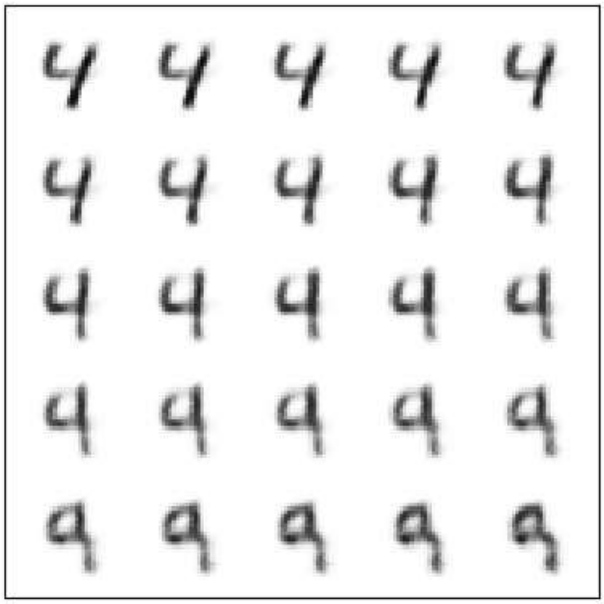

Fig. 12. Interpolated Image (seventh and eighth image in the test set)

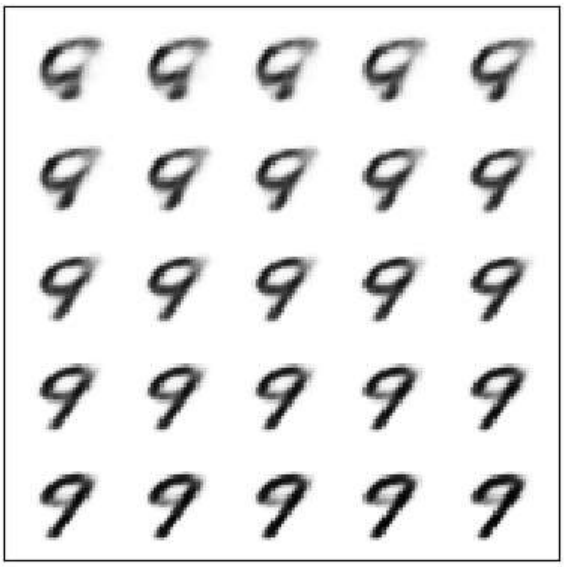

Fig. 13. Interpolated Image (ninth and tenth image in the test set)

IV. CONCLUSION

This assignment is a practical implementation of variational inference bayes. This work can be used as role model to develop more complex systems to generate reliable data in domains with small datasets.

REFERENCE

[1] MNIST dataset, http://yann.lecun.com/exdb/mnist/