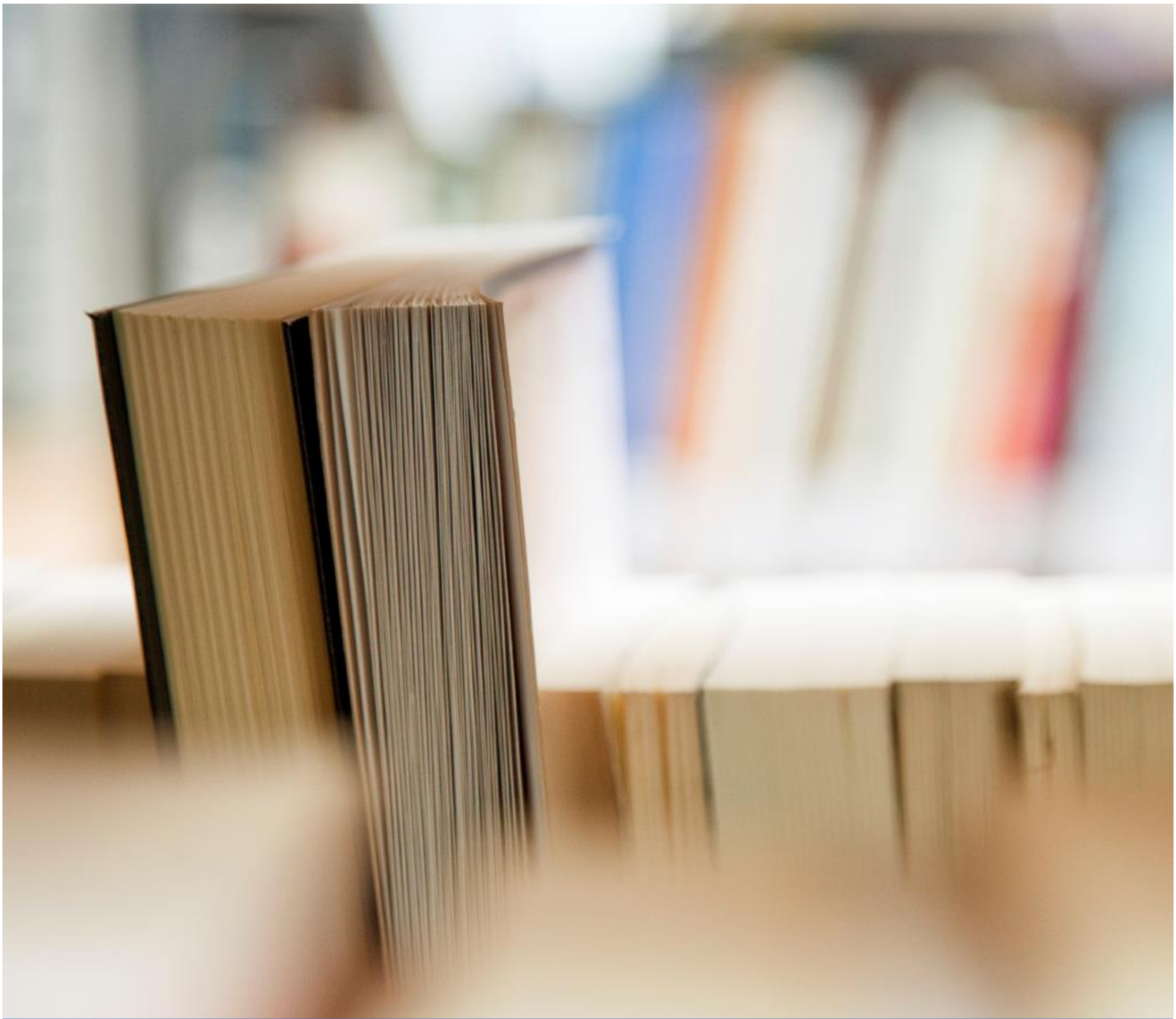

Made IT Task Report

GEN AI for MNIST

Muhammad Mehroz

1/19/13

Report

In this report, I present the design, implementation, and evaluation of a neural network model for MNIST digit image generation. The goal is to create a generative model that takes a digit label (0 to 9) as input and generates realistic handwritten digit images corresponding to that label.

I am submitting both tensorflow and numpy files.

Design Decision

As It is conditional GenAI problem, as we want generate images based on specific conditions.

I opted for 2 design methods

1. Using Simple TensorFlow
2. Using Numpy

TensorFlow:

- Generator :
  - It will generate images using Conv2D, Dense, Up sampling layer using Random Noise
  - Its goal is to produce data that closely resembles real examples.

- Discriminator
  - The discriminator acts as a judge or critic.
  - It evaluates the authenticity of generated data compared to real data.
  - Given an input (either real or generated), the discriminator predicts whether it is real or fake.

- Training Loop
  - We have to make balance between discriminator and generator
  - The learning rate of generator will be higher then discriminator

Sources:

[The Discriminator | Machine Learning | Google for Developers]

[What is Generative AI? | A Comprehensive Generative AI Guide | Elastic]

[Generative AI | NIST]

[GitHub - imehar/mnist-generator: Generative Adversarial Networks (GAN) implementation of MNIST Dataset]

https://numpy.org/doc/stable/dev/internals.code-explanations.html?highlight=index

Numpy

```
 Load MNIST dataset
def load_mnist():
    from tensorflow.keras.datasets import mnist
```

```python
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train, x_test = (x_train.astype('float32') - 127.5) / 127.5,
(x_test.astype('float32') - 127.5) / 127.5
    x_train = x_train.reshape((-1, *img_shape, 1))
    x_test = x_test.reshape((-1, *img_shape, 1))
    return x_train, y_train, x_test, y_test

# Generator
def generate_noise(batch_size):
    return np.random.normal(0, 1, (batch_size, latent_dim))

def generate_labels(batch_size):
    return np.random.randint(0, num_classes, batch_size)

def define_generator():
    model = dict(
        dense1=np.random.randn(latent_dim, 128 * 7 * 7) * 0.1,
        batch_norm1=np.ones((128 * 7 * 7,)),
        batch_norm2=np.ones((128,)),
        conv2d1=np.random.randn(3, 3, 128, 128) * 0.1,
        batch_norm3=np.ones((64,)),
        conv2d2=np.random.randn(3, 3, 64, 1) * 0.1
    )
    return model



def batch_norm(x, mean, variance, epsilon=1e-5):
    # Reshape mean and variance to match the shape of x along the batch
dimension
    mean_broadcasted = np.broadcast_to(mean, x.shape)
    variance_broadcasted = np.broadcast_to(variance, x.shape)

    # Apply batch normalization
    normalized_x = (x - mean_broadcasted) / np.sqrt(variance_broadcasted +
epsilon)
    return normalized_x


def generate_images(generator, noise, labels):
    z = np.dot(noise, generator['dense1'])
    z_mean = np.mean(z, axis=0, keepdims=True)
    z_var = np.var(z, axis=0, keepdims=True)
    z = z.reshape((-1, 128, 7, 7))

    # Batch normalization for z
    z = batch_norm(z, z_mean, z_var)

    z = np.maximum(0, z)
```

```python
    z = np.repeat(z[:, np.newaxis, :, :], 2, axis=1)
    z = np.repeat(z[:, :, :, np.newaxis], 2, axis=3)

    z = np.dot(z, generator['conv2d1'])
    z_mean = np.mean(z, axis=(0, 1, 2), keepdims=True)
    z_var = np.var(z, axis=(0, 1, 2), keepdims=True)

    # Batch normalization for z
    z = batch_norm(z, z_mean, z_var)

    z = np.maximum(0, z)
    z = np.dot(z, generator['conv2d2'])
    z_mean = np.mean(z, axis=(0, 1, 2), keepdims=True)
    z_var = np.var(z, axis=(0, 1, 2), keepdims=True)

    # Batch normalization for z
    z = batch_norm(z, z_mean, z_var)

    z = np.tanh(z)
    return z

# Discriminator
def define_discriminator():
    model = dict(
        conv2d1=np.random.randn(3, 3, 2, 32) * 0.1,
        conv2d2=np.random.randn(3, 3, 32, 64) * 0.1,
        dense1=np.random.randn(7 * 7 * 64, 512) * 0.1,
        dense2=np.random.randn(512, 1) * 0.1
    )
    return model

def discriminator_predict(discriminator, images, labels):
    z = np.concatenate((images, labels), axis=3)
    z = np.dot(z, discriminator['conv2d1'])
    z = np.maximum(0.2 * z, z)
    z = np.dot(z, discriminator['conv2d2'])
    z = np.maximum(0.2 * z, z)
    z = z.reshape((z.shape[0], -1))
    z = np.dot(z, discriminator['dense1'])
    z = np.maximum(0.2 * z, z)
    z = np.dot(z, discriminator['dense2'])
    z = 1 / (1 + np.exp(-z))
    return z

# Training
def train(generator, discriminator, x_train, y_train):
    losses = {'G': [], 'D': []}
    for epoch in range(epochs):
```

```
        idx = np.random.randint(0, x_train.shape[0], batch_size)
        real_imgs, labels = x_train[idx], y_train[idx]

        noise = generate_noise(batch_size)
        fake_imgs = generate_images(generator, noise, labels)

        d_loss_real = np.mean(-np.log(discriminator_predict(discriminator,
real_imgs, labels) + 1e-9))
        d_loss_fake = np.mean(-np.log(1 - discriminator_predict(discriminator,
fake_imgs, labels) + 1e-9))
        d_loss = 0.5 * (d_loss_real + d_loss_fake)

        z = generate_noise(batch_size)
        labels = generate_labels(batch_size)
        g_loss = np.mean(-np.log(discriminator_predict(discriminator,
generate_images(generator, z, labels), labels) + 1e-9))

        losses['G'].append(g_loss)
        losses['D'].append(d_loss)

        if epoch % save_interval == 0:
            print(f"Epoch {epoch}, Losses (D, G): {d_loss}, {g_loss}")
            results = generate_images(generator, z_test, labels_test)
            plot_image(results[:16], labels_test[:16], 4, 4)
    return losses

# Plotting
def plot_image(images, labels, rows, cols):
    fig = plt.figure(figsize=(8, 8))
    for i in range(1, cols * rows + 1):
        img = images[i - 1]
        ax = fig.add_subplot(rows, cols, i)
        ax.title.set_text(labels[i - 1])
        plt.imshow(img.reshape(img_shape), cmap='gray')
        plt.axis('off')
    fig.tight_layout()
    plt.show()
```

```python
iface = gr.Interface(
    fn=generate_digit_image,
    inputs=gr.Number( label="Enter a digit (0-9)"),
    outputs=gr.Image(type="numpy", label="Generated Image"),
    live=True
)# Launch the Gradio app
iface.launch()
```

```
Running on local URL:  http://127.0.0.1:7866

To create a public link, set `share=True` in `launch()`.
```