DEEP LEARNING

**DINH HOANG VIET PHUONG – 301123263**

LAB 04 REPORT

Several essential libraries are imported to facilitate the development of a deep learning model. These libraries include time, for timing operations; numpy, a fundamental package for scientific computing with Python; tensorflow, an open-source library for numerical computation and large-scale machine learning; and matplotlib.pyplot, for creating static, interactive, and animated visualizations in Python. The code also utilizes tensorflow.keras.utils.plot\_model, a tool to visualize the model architecture, and tensorflow.keras's layers and Model, which are fundamental for building and specifying the architecture of neural networks.

A screen shot of a computer program

Description automatically generated

A.

This code snippet focuses on utilizing TensorFlow, a popular open-source library for machine learning, to load and manage the Fashion MNIST dataset. Initially, it imports the TensorFlow library, essential for leveraging its vast array of functionalities. Following this, the Fashion MNIST dataset, a widely used dataset comprising 70,000 grayscale images across 10 fashion categories, is loaded. This dataset is divided into two parts: the training set, containing 60,000 images, and the test set, with 10,000 images. The code then proceeds to organize these datasets into two dictionaries: ds1\_phuong for the training data and ds2\_phuong for the test data. Each dictionary stores both the images and their corresponding labels, facilitating easy access and manipulation of the dataset for training and evaluating machine learning models. This organization is particularly useful for streamlined model development and evaluation processes.

A screen shot of a computer program

Description automatically generated

B.

Data normalization and examination of dataset dimensions are performed for the Fashion MNIST dataset previously loaded. The normalization process adjusts the pixel values of images in both ds1\_phuong and ds2\_phuong dictionaries to a range between -1 and 1. This is achieved by first converting the image data type to float32 for precision, then dividing by 127.5 (half of 255, the maximum pixel value) and subtracting 1. Normalization is a crucial pre-processing step in machine learning as it ensures that the model learns and converges faster during training by providing data on a consistent scale.

After normalization, the code displays the shape of the images in both dictionaries. The training set (ds1\_phuong['images']) contains 60,000 images, and the test set (ds2\_phuong['images']) comprises 10,000 images. Each image is represented in a 28x28 pixel format, which indicates that all images are grayscale and of the same size. This uniformity in data shape and the normalization process are vital for the effective training of neural network models, as they rely on consistent input dimensions and data scales for optimal performance.

A screen shot of a computer program

Description automatically generated

This code snippet further manipulates the previously normalized Fashion MNIST dataset by specifically filtering out images of pants, which are designated by the class label 1. To achieve this, it applies a filtering condition on the label’s arrays within both ds1\_phuong and ds2\_phuong dictionaries to extract only the images corresponding to pants from the training and test datasets, respectively. These extracted images are stored in pants\_images\_ds1 for the training set and pants\_images\_ds2 for the test set.

Subsequently, the filtered images from both datasets are combined into a single new dataset named dataset\_phuong using the np.concatenate function, which merges them along their first axis (maintaining the image dimensionality). This operation results in a consolidated dataset specifically composed of pants images from both the original training and test sets.

Finally, the shape of this newly created dataset is displayed, revealing that dataset\_phuong contains 7,000 images, each of 28x28 pixels. This indicates that, across the combined original training and test sets, there were a total of 7,000 images labeled as pants, showcasing how data manipulation techniques can be utilized to create specialized datasets for targeted analyses or training specific machine learning models focused on classes.

A screen shot of a computer program

Description automatically generated

The displayed code block is designed to showcase a visual representation of the first 12 images from the dataset\_phuong, which contains grayscale images of pants. The matplotlib.pyplot library is used for plotting these images. A figure of size 8x8 inches is created to ensure the images are of a reasonable size when displayed. In a 4x3 grid, each of the first 12 images from dataset\_phuong is plotted, allowing a batch of images to be viewed simultaneously. This grid format is helpful for visual inspection or presentation purposes. The images are displayed in grayscale, which is the format of the Fashion MNIST dataset. Additionally, the x and y ticks, which are the markers on the axes, are removed to keep the focus solely on the image content.

A screenshot of a computer screen

Description automatically generated

The code snippet provided demonstrates the preparation steps to convert the filtered dataset dataset\_phuong, which consists of pants images from the Fashion MNIST dataset, into a TensorFlow dataset suitable for training a model. It uses TensorFlow's data API to accomplish this.

Initially, the tf.data.Dataset.from\_tensor\_slices method is used to create a TensorFlow dataset object from the dataset\_phuong array. This method is effective for slicing the array into a dataset of individual items, making it amenable to further data pipeline transformations.

Next, the dataset is shuffled using the shuffle method with a buffer size equal to the number of images (7000), ensuring that the data is well-randomized. Shuffling is important to prevent the model from learning anything from the order of the samples, which could potentially lead to overfitting on the training data sequence.

Finally, the dataset is batched using the batch method with a specified batch size of 256. Batching is an essential step in preparing data for training as it defines the number of samples that will be propagated through the network in one forward/backward pass, which is a compromise between computational efficiency and the stochastic nature of the training process.

A screen shot of a computer

Description automatically generated

C.

The model starts with a dense (fully connected) input layer that does not use biases and takes a vector of size 100 as input. This layer is followed by a reshape layer that restructures the dense layer's output into a 4D tensor, suitable for transposed convolution operations, essential for generating images in GANs.

Subsequent layers include batch normalization, which stabilizes learning by normalizing the input to each activation function, and LeakyReLU activation functions, which allow for a small, non-zero gradient when the unit is not active, potentially helping to maintain a healthy gradient flow.

There are multiple Conv2DTranspose layers, often called deconvolutional layers, which perform the opposite operation of a conventional convolution. Instead of mapping multiple input activations into a single output, they map a single input activation into a larger spatial output - effectively 'upscaling' the image data.

The final Conv2DTranspose layer uses a 'tanh' activation function and expands the data to the final image size of 28x28 pixels with 1 channel (grayscale), which matches the size of the input images of the Fashion MNIST dataset.

The model summary would list the details of each layer, including the output shapes which confirm the transformation of the input vector into a 2D image. The plot\_model function call at the end would save this visual representation to a PNG file, showcasing the architecture of the generator model. This visualization helps in understanding the data flow and the transformations applied at each layer, which is critical for debugging and explaining the model architecture.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A black and white diagram

Description automatically generated

D.

Initially, a sample noise vector is created using tf.random.normal([1, 100]), which generates a 1D tensor of 100 random values drawn from a standard normal distribution. This noise vector serves as a random input for the generator model.

Subsequently, the generator model named generator\_model\_phuong takes this noise vector as input and generates an image. The training=False parameter is set during inference to ensure that the model's behavior is consistent with how it was during training, specifically not updating weights or batch normalization parameters.

Finally, the generated image is displayed using matplotlib.pyplot.imshow with a grayscale colormap, and the axes are turned off to focus solely on the image content. Since the generator model is untrained, the output is expected to be random pixel values that do not resemble a coherent image of pants or any other recognizable object. The result is typically a pattern of noise, as shown in the image, and through the training process, the generator would learn to produce more structured and realistic images.

A screenshot of a computer screen

Description automatically generated

E.

The model begins with a convolutional layer (Conv2D) that reduces the image dimensions while increasing the depth to 64 feature maps and applies a LeakyReLU activation function, allowing a small gradient when the unit is inactive. This is followed by a dropout layer to prevent overfitting by randomly setting a fraction of input units to 0 during training.

Another convolutional layer follows, further reducing the spatial dimensions and increasing the depth to 128 feature maps, with another LeakyReLU activation and dropout layer. This sequence of layers effectively captures complex features in the input image and helps the discriminator make a decision.

Afterward, the model flattens the 3D feature maps into a 1D vector, which is then fed into a dense layer with a single neuron that outputs a scalar value representing the discriminator's judgment of the image's authenticity.

The summary of this model would include the specifics of each layer, such as the number of parameters and the shape of the output after each layer. The plot\_model function would save a visual diagram of the model's structure, including the shape of the outputs and the names of the layers, to a file named 'discriminator\_model\_phuong.png'. This visualization assists in understanding the sequence and structure of the operations the model performs on the input data.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A black and white diagram

Description automatically generated

F.

The loss function used here is binary cross-entropy with logits, a common choice for classification problems where the classes are mutually exclusive. The 'from\_logits=True' argument indicates that the outputs of the models are unnormalized, meaning they are not passed through a sigmoid function, which converts them to probabilities.

The Adam optimizer is chosen for both the generator and the discriminator. The Adam optimizer is an extension to stochastic gradient descent that has been empirically shown to handle sparse gradients on noisy problems very well, which is often the case in GAN training.

Two loss functions are defined for the GAN training process:

* generator\_loss calculates the loss for the generator model. The generator aims to produce images that the discriminator will classify as real. Therefore, the function compares the discriminator's predictions on the generated images (fake\_output) to a tensor of ones, since the ideal scenario for the generator is that all its outputs are classified as real (labelled as 1).
* discriminator\_loss computes the loss for the discriminator model, which has a twofold goal: to correctly classify real images as real (real\_output compared to ones) and generated images as fake (fake\_output compared to zeros). The total discriminator loss is the sum of the loss for real and fake images.

By training both the generator and discriminator using these loss functions and optimizers, the GAN aims to reach a point where the generator produces indistinguishable images, and the discriminator becomes proficient in detection, resulting in a minimax game equilibrium.

A computer screen shot of a computer program

Description automatically generated

The training\_step function orchestrates a single training iteration for a Generative Adversarial Network. It generates noise and uses it to produce synthetic images with the generator. The discriminator evaluates these alongside real images, and losses for both models are computed. Gradients of these losses are then calculated and applied to update the model weights through the optimizers. The function concludes by returning the generator and discriminator losses, key indicators of the GAN's learning progress.

A screenshot of a computer program

Description automatically generated

G.

This code sets up and executes the training process for a GAN model. With a batch size of 256 and noise dimensionality of 100, the model trains for 10 epochs. The loop records the duration of each epoch, printing it out and appending it to a list epoch\_times for later analysis.

Loss functions for both the generator and discriminator are defined within the setup, and a custom training\_step function is employed to execute the training for each batch of images from train\_dataset\_phuong.

Post-training, it computes the average duration per epoch, and then it extrapolates to estimate the total training time for 100 epochs using 70,000 samples. The calculations indicate the model takes a significant amount of time to train for the given number of epochs, with the estimated total time reaching over 310,000 seconds.

Note that the time calculation assumes that the average time per epoch remains consistent regardless of the number of epochs, which may not account for factors like hardware performance variations or the increasing model complexity over time.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

H.

This code block describes the process of generating a grid of images using a trained generator model from a Generative Adversarial Network (GAN):

* Sample Vector Creation: It begins by generating 16 random sample vectors with a predefined dimension (noise\_dim), following a standard normal distribution.
* Image Generation: These vectors are fed into the generator model to produce 16 synthetic images. The training=False argument ensures that the model operates in inference mode, not affecting its internal states.
* Normalization: The pixel values of the generated images are normalized from the [-1, 1] range (as output by the generator using the tanh activation function) to the [0, 255] range, suitable for image display. This is done by scaling the pixel values up by 127.5 and then adding 127.5. The images are then cast to the uint8 data type, which is the appropriate format for displaying images with pixel values between 0 and 255.

Display: A 4x4 grid of images is set up using matplotlib's subplot function, and the normalized images are displayed in grayscale without axes. This visual representation is rendered using plt.show().

Through this process, one can visually inspect the quality of images generated by the GAN, which is often a crucial step in evaluating the performance of generative models.

A computer screen shot of a program

Description automatically generated

A collage of x-ray images

Description automatically generated