Lab Sheet 6: Pixel CNNs and Autoencoders

Expected Lab Duration: ~1 hours

6.1 - Introduction to CNNs

Convolutional Neural Networks (CNNs)

CNNs are a class of deep learning models, primarily used for image processing, but also for other types of input such as audio. A CNN is designed to automatically and adaptively learn spatial hierarchies of features from the input data.

The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.

The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Pixel CNNs

Pixel CNNs are a variant of CNNs, but with a twist. While CNNs are great for image classification tasks, Pixel CNNs are designed for image generation tasks.

The idea behind Pixel CNNs is to model the distribution of image data sets by assigning a probability to each possible image. The way they do this is by factorizing the joint image distribution as a product of conditionals, and then modeling each conditional distribution with a convolutional neural network.

In simpler terms, Pixel CNNs generate images pixel by pixel. For each pixel, they consider all the pixels to the top and to the left of the current pixel (i.e., the context) to generate the pixel value. This is done using a masked convolution where the mask ensures that the future pixels (to the bottom and right of the current pixel) do not contribute to the present pixel's value.

This results in a network that is able to generate diverse and sharp images, one pixel at a time. However, because of this sequential nature, Pixel CNNs can be slower to train and generate images compared to some other image generation models.

6.2.1 - Hands on Pixel CNN!

Problem Statement: Image Generation using Pixel CNNs

The goal of this exercise is to design and implement a Pixel CNN model to generate images. You will be working with an open-source dataset for this task. The aim is not just to implement the model, but also to understand the intricacies and details of how Pixel CNNs work.

Dataset

For this exercise, we will use the MNIST dataset (again😛). It's a simple yet powerful dataset containing grayscale images of handwritten digits (0-9). The dataset is relatively small and should be manageable for this exercise.

Tasks:

Data Preparation: Download and load the MNIST dataset. Perform any necessary preprocessing steps such as normalization and reshaping of the images. Split the dataset into training and validation sets.

Model Design: Design a Pixel CNN model. Remember that unlike traditional CNNs, Pixel CNNs generate images pixel by pixel, taking into account all the pixels to the top and to the left of the current pixel.

Training: Train your Pixel CNN model on the MNIST dataset. Make sure to use a suitable loss function and optimizer. Also, remember to save the model parameters after training.

Image Generation: Use the trained Pixel CNN model to generate new images. Start with a blank image and fill it in pixel by pixel, using the model to get the next pixel value given the current context.

Evaluation: Evaluate the performance of your model. This can be tricky as it's not as straightforward as comparing predictions to ground truth. You might want to visually inspect the generated images. Another option is to use a separate discriminator network to distinguish between real and generated images.

6.2.2 - Solving with python 🐍

Note: To make the model better than the standard approach, we have used multiple convolutional layers in the Pixel CNN model. This allows the model to learn more complex patterns in the data.

Please also note that this is a simplified example and may not produce high-quality images. Training PixelCNNs can be quite slow due to their autoregressive nature, and generating high-quality images often requires a lot of computational resources and time. Don't worry we will learn about technologies like Variational Autoencoders (VAEs) & Generative Adversarial Networks (GANs) later in this course as they can often produce good results more quickly.

```
import numpy as np
from keras.layers import Input, Conv2D, Activation, BatchNormalization
from keras.models import Model
from keras.datasets import mnist
```

Custom Masked Convolution Layer

```
class MaskedConv2D(Conv2D):
    def __init__(self, mask_type, *args, **kwargs):
        super(MaskedConv2D, self).__init__(*args, **kwargs)
        self.mask_type = mask_type
        assert mask_type in {'A', 'B'}, "mask_type must be 'A' or 'B'."

    def build(self, input_shape):
        super(MaskedConv2D, self).build(input_shape)
        self.mask = np.ones(self.kernel.shape.as_list())
        self.mask[self.kernel_size[0] // 2, self.kernel_size[1] // 2 +
(self.mask_type == 'B'):, :, :] = 0.
        self.mask[self.kernel_size[0] // 2 + 1:, :, :, :] = 0.

    def call(self, inputs):
        self.kernel.assign(self.kernel * self.mask)
        return super(MaskedConv2D, self).call(inputs)
```

Load MNIST dataset

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Normalize and reshape the data

```
x_train = x_train.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
```

Design a more complex PixelCNN model

```
inputs = Input(shape=(28, 28, 1))
x = MaskedConv2D('A', 64, 7, padding='same')(inputs)
x = BatchNormalization()(x)
x = Activation('relu')(x)

for _ in range(5):  # Add more layers
    x = MaskedConv2D('B', 64, 7, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

x = MaskedConv2D('B', 1, 7, padding='same')(x)
model = Model(inputs=inputs, outputs=x)
```

Compile and train the model

```
model.compile(optimizer='adam', loss='mse')  # Use MSE loss
model.fit(x_train, x_train, epochs=5, batch_size=128)  # Train for more
epochs
```

VISUALISATION

```
import matplotlib.pyplot as plt

# Generate images
n = 10  # Number of images to display
generated_images = model.predict(x_train[:n])

# Reshape and re-scale the images
generated_images = generated_images.reshape((n, 28, 28))
generated_images = (generated_images * 255).astype(np.uint8)

# Plot the images
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
```

```
    plt.imshow(x_train[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(generated_images[i])
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Task: Exercise

Add code to visualize the generated images. Then try to add more layers to your network. Report how this affected your image generation, is adding more layers always helpful in problems like these? What are the other improvements that can be done to generate better images? Write down your observations in the jupyter notebook.

(Optional):

You can read about more standard metrics of how the generated images of a model are evaluated. (For example: Inception Score (IS) and Frechet Inception Distance (FID). These methods compare the statistics of generated images to those of real images in the feature space of a pre-trained model (like InceptionV3)).

6.3 - Introduction to Autoencoders! 🤖

Reference articles:

1. https://www.geeksforgeeks.org/auto-encoders/
2. https://www.ibm.com/topics/autoencoder

Autoencoders are a specialized class of artificial neural networks designed for unsupervised learning. They can learn efficient representations of input data without the need for labels. The essential principle of an autoencoder is to compress and effectively represent input data.

The architecture of an autoencoder includes an encoder, a decoder, and a bottleneck layer:

Encoder: The input layer takes raw input data. The hidden layers progressively reduce the dimensionality of the input, capturing important features and patterns. These layers compose the encoder.

Bottleneck Layer (Latent Space): This is the final hidden layer, where the dimensionality is significantly reduced. This layer represents the compressed encoding of the input data.

Decoder: The bottleneck layer takes the encoded representation and expands it back to the dimensionality of the original input. The hidden layers progressively increase the dimensionality and aim to reconstruct the original input. The output layer produces the reconstructed output, which ideally should be as close as possible to the input data.

The loss function used during training is typically a reconstruction loss, measuring the difference between the input and the reconstructed output. Common choices include mean squared error (MSE) for continuous data or binary cross-entropy for binary data. During training, the autoencoder learns to minimize the reconstruction loss, forcing the network to capture the most important features of the input data in the bottleneck layer.

Autoencoders are useful for a variety of tasks, including dimensionality reduction, feature learning, anomaly detection, data compression, image denoising, and facial recognition. Certain types of autoencoders, like variational autoencoders (VAEs) and adversarial autoencoders (AAEs), adapt autoencoder architecture for use in generative tasks, like image generation or generating time series data.

6.4.1 - Generating using Autoencoders!

Problem Statement: Generative Modeling of Handwritten Digits using Autoencoders

In the field of machine learning, generative models are powerful tools for creating new data instances that resemble your training data. Autoencoders, in particular, have shown promising results in tasks such as image denoising and anomaly detection. In this project, we aim to explore the use of autoencoders as a generative model.

Dataset:

We will use the MNIST dataset again. 😴

Approach:

We will train an autoencoder on the MNIST dataset. The autoencoder will learn to compress each input image into a low-dimensional representation, and then reconstruct the original image from this representation. Once the autoencoder is trained, we can use the decoder part of the autoencoder as a generative model. By feeding the decoder random vectors sampled from the same distribution as the encoded representations, the decoder will generate new images that resemble the original handwritten digits.

Expected Outcome:

At the end, we expect to have a generative model that can produce realistic images of handwritten digits. We will evaluate the quality of the generated images with visual inspection.

Challenges:

One of the main challenges in this is ensuring that the autoencoder learns a useful representation of the data. If the autoencoder simply learns to copy its input to its output, the encoded representations will not capture the underlying structure of the data, and the generative model will produce poor results. To overcome this, we may need to experiment with different architectures, loss functions, and regularization techniques.

This problem statement provides a clear direction for using autoencoders as a generative model and sets the stage for an interesting machine learning project. It also leaves room for further exploration and extension, such as experimenting with different types of autoencoders (e.g., variational autoencoders) or applying the generative model to other types of data.

6.4.2 - Coding Autoencoders! 👩‍💻

This code first loads and preprocesses the MNIST dataset. It then defines an autoencoder with one hidden layer in the encoder and decoder. The autoencoder is trained to reconstruct the input images, and the decoder part of the autoencoder is used to generate new images. The generated images are visualized at the end.

The innovation here is the use of a standard autoencoder for a generative task, which is not the most common use case for autoencoders. This shows the versatility of autoencoders and their potential for creative applications. Note that this is a simple example and the quality of the generated images can be improved with more complex models and training techniques.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
```

Load MNIST dataset

```
(x_train, _), (x_test, _) = mnist.load_data()
```

Normalize data

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

Flatten the 28x28 images into vectors of size 784

```
x_train = x_train.reshape((len(x_train), -1))
x_test = x_test.reshape((len(x_test), -1))
```

Define the size of encoded representations

```
encoding_dim = 32
```

Define the input layer

```
input_img = Input(shape=(784,))
```

Define the encoder layers

```
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(encoding_dim, activation='relu')(encoded)
```

Define the decoder layers

```
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)
```

Create the autoencoder model

```
autoencoder = Model(input_img, decoded)
```

Create the encoder model

```
encoder = Model(input_img, encoded)
```

Create the decoder model

```
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers-3
decoder_layer = autoencoder.layers-2
decoder_layer = autoencoder.layers-1
decoder = Model(encoded_input, decoder_layer)
```

Compile the autoencoder model

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Train the autoencoder

```
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

The encoded representations of the MNIST digits that the autoencoder has learned during training might not follow a normal distribution. Therefore, when you generate new points from a normal distribution, these points might not correspond to meaningful representations in the learned manifold, resulting in images that don't look like digits.

Generate new images

```
# Use the encoder to transform the test set into the encoded space
encoded_imgs = encoder.predict(x_test)
```

```
# Use the decoder to generate new images from the encoded representations
generated_images = decoder.predict(encoded_imgs)

# Reshape and visualize the generated images
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(generated_images[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Exercises: Just run the code above and make observations particularly in matters of whether you can generate completely new images that the model has never seen before using this implementation. Why or Why not? Write down your observations in the jupyter notebook.

Since VAEs are covered in lectures, does it explicitly model the distribution of the encoded representations and allows you to sample from this distribution to generate new images? Would it solve our current problem? We can implement VAEs in the next lab sheet!