Lab Sheet 4: Image generation

Expected Lab Duration: ~2 hours

2.1 - Image Generation with Fully Visible Sigmoid Belief Networks

Objective

So, what are we going to achieve in this lab?

Well, we're diving into the world of Fully Visible Sigmoid Belief Networks (FVSBNs). These are super cool because they can generate images - imagine creating your own handwritten digits!

Here's what you'll be able to do by the end of this lab:

1. **Get the hang of FVSBNs**: You'll understand what these networks are, how they're structured, and why they're used to generate images.
2. **Play with the MNIST Dataset**: This is a popular dataset used for image recognition tasks. You'll get to load it, visualize it, and understand why it's so widely used.
3. **Code your own FVSBNs**: Using Python and TensorFlow, you'll build your own FVSBN. You'll learn how to define the network, choose the right parameters, and train your model using the MNIST dataset.
4. **Generate your own images**: Once your FVSBN model is trained, you'll use it to generate new images. It's like having your own digital artist!
5. **Evaluate your work**: You'll learn how to check how well your model performs, understand its strengths and weaknesses, and discuss how it could be improved.

Task 1: Introduction to Fully Visible Sigmoid Belief Networks

What are Fully Visible Sigmoid Belief Networks?

FVSBNs are a type of **generative model** that are used in the field of machine learning[1]. They are a variant of Sigmoid Belief Networks (SBNs), which are directed graphical models[2].

In an SBN, each node represents a binary variable that can take on the values 0 or 1. The probability of a node taking on a value is determined by a sigmoid function applied to a linear combination of its parent nodes[3].

FVSBNs extend this concept by making all nodes visible, hence the name “Fully Visible”. This means that every node in the network contributes to the output, and the state of any node can be determined by looking at the states of its parent nodes.

How do FVSBNs work?

The working of FVSBNs is based on the concept of **probabilistic graphical models**. Each node in the network represents a binary variable, and the edges represent the relationships between these variables[2].

The probability of a node taking on a certain value is determined by a sigmoid function applied to a linear combination of its parent nodes[3]. This allows the network to model complex, non-linear relationships between variables.

Training an FVSBN involves learning the weights and biases that define the relationships between nodes. This is typically done using a method such as **stochastic gradient descent**[1].

Why are FVSBNs used in image generation?

FVSBNs are particularly useful for tasks like image generation because they can model complex, high-dimensional data. By training an FVSBN on a dataset of images, the network can learn to generate new images that are similar to the ones it was trained on[2].

The ability of FVSBNs to model the dependencies between pixels makes them well-suited to this task. Each pixel in an image can be represented by a node in the network, and the relationships between pixels can be modeled by the edges[2].

I hope this gives you a good understanding of Fully Visible Sigmoid Belief Networks. They’re a powerful tool in the field of machine learning, particularly for tasks involving high-dimensional data like image generation. Happy learning! 🚀

Task 2: Understanding the MNIST Dataset

The **MNIST dataset** (Modified National Institute of Standards and Technology database) is a large collection of handwritten digits[1]. It is a very popular dataset in the field of image processing and is often used for benchmarking machine learning algorithms[1].

Here are some key details about the MNIST dataset:

- It contains a total of **70,000 images** of handwritten digits from 0 to 9[1].
- Each image is **28 x 28 pixels** in size[1].
- The dataset is already divided into training and testing sets[1].
- The training set contains **60,000 images**, and the testing set contains **10,000 images**[1].

This dataset provides a baseline for testing image processing systems and is commonly used in the field of machine learning and computer vision[1]. It's available in various libraries like Keras[1] and TensorFlow[2]. The images have been size-normalized and centered in a fixed size of 28×28 pixels[3].

Task 3: Implementing FVSBNs

This code uses the PyTorch library to create a Fully Visible Sigmoid Belief Network (FVSBN) and trains it on the MNIST dataset. After training, it generates new images.

Please note that this is a simple example and doesn't include some important aspects such as model evaluation and fine-tuning. Also, FVSBNs are not typically the first choice for image generation tasks. More complex models like Variational Autoencoders (VAEs) or Generative Adversarial Networks (GANs) might be more suitable for such tasks.

Imports

```
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader
```

Load MNIST dataset

```
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

Define the FVSBN model

```
class FVSBN(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(FVSBN, self).__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.hidden_layer = nn.Linear(hidden_dim, input_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.input_layer(x))
        return self.sigmoid(self.hidden_layer(x))
```

Initialize the FVSBN model

```
model = FVSBN(784, 512) #adjust
```

Define the loss function and optimizer

```
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Train the model

```
for epoch in range(5):
    for images, _ in train_loader:
        images = images.view(images.shape[0], -1)
        output = model(images)
        loss = criterion(output, images)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Generate new images

```
with torch.no_grad():
    new_images = model(torch.randn(64, 784))
```

#TODO : Plot the image using matplotlib

2.2 - Image Generation with Neural Autoregressive Density Estimation (NADE)

We will have similar objectives as the previous section!

Task 1: Introduction to Neural Autoregressive Density Estimation (NADE)

You can refer to your course slides. However, note that an autoregressive NADE implementation will be complicated. For sake of simplicity and time constraints we proceed with a simpler implementation in the next page.

Task 2: Implementing NADE

```
import torch
from torch import nn
from torchvision import datasets, transforms
from torch.autograd import Variable
```

Load MNIST dataset

```
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True,
train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
shuffle=True)
```

Define the NADE model

```
class NADE(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(NADE, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, input_dim)

    def forward(self, x):
        h = torch.relu(self.fc1(x))
        return torch.sigmoid(self.fc2(h))

    def generate(self, batch_size):
        samples = torch.zeros(batch_size, self.input_dim)
        with torch.no_grad():
            for i in range(self.input_dim):
                h = torch.relu(self.fc1(samples))
                prob = torch.sigmoid(self.fc2(h))
                samples[:, i] = torch.bernoulli(prob[:, i])
        return samples
```

Initialize the model and optimizer

```
model = #todo
#todo: choose your optimiser, learning rate and network parameters
```

Training loop

```
for epoch in range(100):  # loop over the dataset multiple times
    running_loss = 0.0
    print(epoch)
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, _ = data
        inputs = inputs.view(inputs.shape[0], -1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = nn.BCELoss()(outputs, inputs)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss
/ 2000))
            running_loss = 0.0

print('Finished Training')
```

```
import matplotlib.pyplot as plt

# Generate a new image
new_image = model.generate(1)
# Reshape the image to 28x28 pixels
new_image = new_image.view(28, 28)
# Convert tensor to numpy array
new_image = new_image.detach().numpy()
# Display the image
plt.imshow(new_image, cmap='gray')
plt.show()
```