Assignment 4

1 Practice: Model Deployment

Use the CIFAR-10 dataset to train an Energy Model and a Diffusion Model and add it to the API implemented as part of Module 6 Class Activity. Commit your code to GitHub.

2 Theory: Building Blocks of a Diffusion Model

Answer the following questions.

Question 1

Given a timestep t, describe how the sinusoidal embedding is computed. Write the mathematical formula for the i-th dimension of the embedding.

Question 2

If the embedding dimension is 8, write out the values of the sinusoidal embedding vector for $t = 1$. Assume max period of 10000.

Question 3

How does positional encoding in transformers relate to sinusoidal time embeddings in diffusion models? What is the key difference in how they are used?

Question 4

If the input image is of size 64x64, and you apply 3 downsampling blocks with stride-2 convolutions, what is the spatial resolution at the bottleneck?

Question 5

If the model receives an image x_t and timestep t, what does the UNet output, and how is that used to compute the loss.

3 Theory: Building Blocks of an Energy Model

So far in this course, we have relied on PyTorch (or TensorFlow) to automatically adjust the parameters of our neural networks to minimize a given loss function. Under the hood these frameworks need to calculate the gradient vector (consisting of derivatives of the loss with respect to model parameters or in the case of Langevin Sampling, with respect to the inputs).

In this section, you will practice some of these computations. If you are familiar with Multivariable Calculus you can easily verify these simple calculations, if not do not worry - our goal is to understand the mechanics of how the gradients are tracked, not the precise derivative formulas.

Question 6: Basic Gradient Calculations

When we define vectors or tensors in PyTorch, we use the `torch.tensor` class. If a tensor is created with `requires_grad=True`, PyTorch automatically constructs a *computational graph* as operations are performed on it. This graph keeps track not only of the results of each operation, but also of how each output depends on the inputs (more precisely, the gradient).

As a result, when we call `.backward()`, PyTorch efficiently computes gradients using the chain rule, even for complex compositions of functions.

Run the following code:

```
import torch

# Create a tensor with requires_grad=True
x = torch.tensor([2.0], requires_grad=True)

# Define a simple function y = x² + 3x
y = x**2 + 3 * x

# Backpropagate
y.backward()

# Print the gradient
print("x.grad =", x.grad)
```

a) What is the expected gradient of $y = x^2 + 3x$ with respect to x?

b) What happens if you set `requires_grad=False` on `x`?

c) If we create `torch.tensor` without specifying the `requires_grad` flag will the gradients be tracked? (Consult the documentation for `torch.tensor`)

Question 7: Introduce Weights

Let's replace the coefficients in the previous question with variables.

```
# Create a tensor with requires_grad=True
x = torch.tensor([2.0], requires_grad=True)
w = torch.tensor([1.0, 3.0])

# Define a simple function y = x² + 3x
```

```
y = w[0] * x**2 + w[1] * x

# Backpropagate
y.backward()

# Print the gradient
print("x.grad =", x.grad)
```

a) When we train neural networks we are interested in gradients with respect to the weights w. Add `print("w.grad =", w.grad)` to the above code. What is the result and why?

b) Modify the code to calculate and print the gradient with respect to w.

c) If we create `torch.tensor` without specifying the `requires_grad` flag will the gradients be tracked? (Consult the documentation for `torch.tensor`)

Question 8: Breaking the Graph

```
x = torch.tensor([1.0], requires_grad=True)
y = x * 3
z = y.detach()
w = z * 2

w.backward()  # This will raise an error
```

Why does this fail? Fix the code to still use `z` but allow gradients to flow back to `x`.

Question 9: Gradient Accumulation

```
x = torch.tensor([1.0], requires_grad=True)

y1 = x * 2
y1.backward()
print("After first backward: x.grad =", x.grad)

y2 = x * 3
y2.backward()
print("After second backward: x.grad =", x.grad)
```

What is happening with `x.grad`? How do you avoid this issue?