Lab 15

Neural Networks with Pytorch

What are Neural Networks?

Neural networks are a class of machine learning models inspired by the structure of the human brain. They consist of layers of nodes (neurons) where each node performs a computation based on its input and passes the result to the next layer. Neural networks are powerful for modeling complex patterns in data, especially in fields like computer vision, natural language processing, and speech recognition.

What is Pytorch?

PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It is known for its flexibility, dynamic computation graph, and easy-to-debug code, making it popular in both research and industry. PyTorch provides intuitive APIs for defining and training neural networks. Here’s why PyTorch is a great choice:

- **Easy to Learn:** Its Pythonic nature makes it simple to pick up.
- **Flexibility:** Allows customizations at every step.
- **Community Support:** A vast community ensures plenty of resources for troubleshooting and learning.
- **Integration:** Works seamlessly with other libraries and tools.

PyTorch Primer

Tensors

Tensors are the core data structures in PyTorch. They are similar to NumPy arrays but have additional features for GPU acceleration and autograd.

```
import torch
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = torch.randn(2, 3)
```

You can perform operations like:

```
z = x + 2
```

This adds `2` to every element in the tensor `x`. If `x = tensor([1.0, 2.0, 3.0])`, then `z = tensor([3.0, 4.0, 5.0])`.

```
a = y @ y.T
```

This performs **matrix multiplication** between `y` and its transpose `y.T`. The `@` operator is shorthand for matrix multiplication. If `y` is a 2x3 tensor, `y.T` will be 3x2, and the result `a` will be a 2x2 matrix.

Autograd

PyTorch tracks operations on tensors to compute gradients automatically. This is crucial for training.

```
z = x.sum()
z.backward()
print(x.grad)  # prints gradient of z w.r.t x
```

Building Models

Models are defined by subclassing `nn.Module`.

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(10, 20)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(20, 1)

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
```

Loss Functions

- `nn.MSELoss()` for regression
- `nn.BCELoss()` for binary classification
- `nn.CrossEntropyLoss()` for multi-class classification

Optimizers

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

This line creates an Adam optimizer that will update the parameters of `model`. The `model.parameters()` call returns all the learnable weights of the model. The `lr=0.001` sets the learning rate, which controls how much the parameters are updated at each step. Adam is an adaptive learning rate optimizer that works well in many deep learning applications.