Deep Learning

Autograd - Model Architecture

Vo Nguyen Thanh Nhan

February 2026

Contents

Chapter 1

Autograd in Pytorch

1.1 Definition

Autograd (or Automatic Differentitate) automates the computation of backward passes in neural networks. It is a system that records all operations performed on tensors to create a dynamic Computational Graph, allowing for the automatic calculation of gradients using the Chain Rule.
There are 22 steps for it to work:

- **Forward:** Here the Autograd will record the calculation (e.g `y = w*x + b` and each resulting tensor has a `grad_fn` (e.g `grad_fn=<AddBackward0>`)
- **Backward:** When `y.backward()` is called, Autograd traverses the graph in reverse order and computes the derivative with the respect to the parameter (e.g `x.grad()`)

1.2 Coding

The following code is an example of steps in Linear Regression:

```
import torch

# 1. Setup Data
x_train = torch.tensor([1.0, 2.0, 3.0])
y_true = torch.tensor([3.0, 5.0, 7.0])

# 2. Initialize Parameters (Starting Point)
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
```

```python
learning_rate = 0.01

# 3. The Loop (Iterative Learning)
for epoch in range(100): # Running for 100 iterations
    # --- Step A: Forward Pass ---
    y_pred = w * x_train + b


    # --- Step B: Calculate Scalar Loss ---
    loss = torch.mean((y_pred - y_true)**2)

    # --- Step C: Backward Pass (Autograd) ---
    # This computes gradients for w and b
    loss.backward()

    # --- Step D: Optimization (Weight Update) ---
    with torch.no_grad():
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad

        # IMPORTANT: Zero the gradients for the next iteration
        w.grad.zero_()
        b.grad.zero_()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}: Loss = {loss.item():.4f}, w = {w.item():.2f}, b = {b.item():.2f}")
```

Note: You will need `requires_grad= True` for the Autograd to record and calculate the derivatives after that.

```
Epoch 10: Loss = 0.5788, w = 1.29, b = 2.03
Epoch 20: Loss = 0.2130, w = 1.45, b = 2.07
Epoch 30: Loss = 0.1707, w = 1.51, b = 2.07
Epoch 40: Loss = 0.1597, w = 1.53, b = 2.05
Epoch 50: Loss = 0.1519, w = 1.55, b = 2.03
Epoch 60: Loss = 0.1447, w = 1.56, b = 2.00
Epoch 70: Loss = 0.1379, w = 1.57, b = 1.98
Epoch 80: Loss = 0.1314, w = 1.58, b = 1.95
Epoch 90: Loss = 0.1252, w = 1.59, b = 1.93
Epoch 100: Loss = 0.1193, w = 1.60, b = 1.91
```

Figure 1.1: Outputs

Chapter 2

Model Architecture for DL

2.1 Definition

Model architecture is the structural design of a neural network

2.1.1 Structure Encapsulation

We use the `__init__` phase to declare layers which represents a specific mathematical operation and its associated storage

- **Linear Layers (Fully Connected):** These are the building blocks that perform transformations
- **Regularization Layers** Components which don't have learnable parameters are architectural choices made to improve model's generalization

2.1.2 Functional Mapping

This makes the architecture works

- **Sequential and Non-sequential**
- **Dimensional Transformation**

2.2 Coding

The following code will illustrate an example of Model Architecture:

```python
import torch.nn as nn
import torch.nn.functional as F

class MNISTClassifier(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=128, output_dim=10):
        """
        Constructor: Defines the model architecture.
        """
        super().__init__() # Initializes the base nn.Module

        # Define Fully Connected (Linear) Layers
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

        # Regularization layer
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        """
        Forward Pass: Defines the computational graph.
        """
        # Step 1: Reshape 2D image (28x28) to 1D vector (784)
        x = x.view(x.size(0), -1)

        # Step 2: Linear transformation followed by Non-linear
        ↪  activation (ReLU)
        x = F.relu(self.fc1(x))

        # Step 3: Apply Dropout to prevent overfitting
        x = self.dropout(x)

        # Step 4: Final linear layer to produce output Logits
        logits = self.fc2(x)
        return logits
# Customizing the architecture: input 784, hidden 256, output 10
model = MNISTClassifier(input_dim=784, hidden_dim=256, output_dim=10)
```