Lec17: Neural Networks

Introduction

- (Deep) Neural Network is a really flexible and powerful machine learning model that
  - Approximate any reasonable function in theory
  - “Learns the pattern from data” incredibly well in practice, particularly in hard areas like cognition, perception
- Two computational modes:
  - Training: go through training examples, update model parameters (usually driven by reducing a certain loss function)
  - Inference: given new input, compute output (prediction, next token, classification, regressions, probability, etc…)
- Usually very hungry for data+compute
  - Modern LLM pretty much demands to be run on accelerators (GPUs, TPUs, …)

Agenda

- To study how a neural network works, in particular its computational structure and their acceleration on GPU…
- We study a relatively simple but representative example: digit classifier on the MNIST dataset.
- Moving from higher abstraction -> lower abstraction
  - PyTorch version
  - Python+numpy version (pytorch free) + Math
  - C version (without dependencies)
  - C+CUDA version (accelerated on GPU!)

Neural Network Background

- 10,000-ft bird view:
  A particular class of mathematical function
  f(x;w) = y,
  x is input vector
  w is the model parameters
  y is output vector
- Training:
  given (x0,y0), (x1,y1), …, update w to make f() better predict y
- Inferencing:
  given x, compute f(x; w) = y

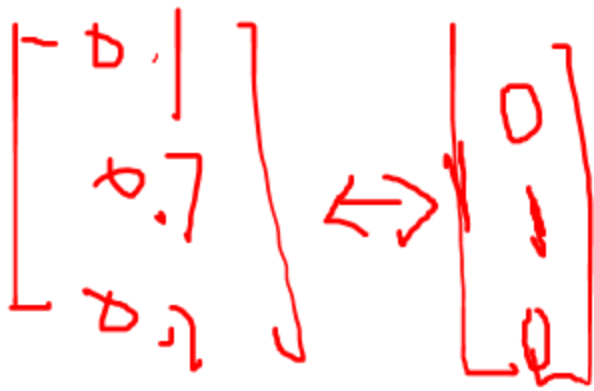
Function f(x;w)

- It's constructed in **layers**
  input x -> (layer 1) -> output z1 -> (layer 2) -> output z2 -> ...
  -> output z
- There are many different layers:
  - Linear layer: $\mathrm{f(x)} := \ x \cdot W + b = z$
    W is weights, a matrix of size (#input, #output), b is vector bias.
  - Convolutional layer, recurrent layer, embedding layer...
  - Activation layer:
    ReLU: $f(x) = \max(x, 0)$
    Sigmoid, Softmax, ...
  - Normalization layer (to stabilize training): BatchNorm, LayerNorm,
  - Attention layer:

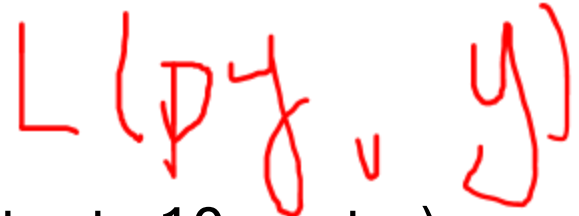
MNIST Dataset

- MNIST contains 60,000 training images and 10,000 testing images of handwritten digits.
- The dataset comprises grayscale images of size 28×28 pixels.

Simple Multilayer Perceptron (MLP)



- Input x: 784 (28*28) pixels(floats)
- Three layers: input -> hidden -> output
  - Linear layer1: 784*256 (a matrix that maps 784-vector to 256-vector
    $z1=x*W1+b1$
  - ReLU layer: 256->256
    $a1=\text{ReLU}(z1)$
  - Linear layer2: 256*10 (a matrix that maps 256 –vector to 10-vector)
    $z2=a1W2+b2$



Softmax + Cross-Entropy Loss:

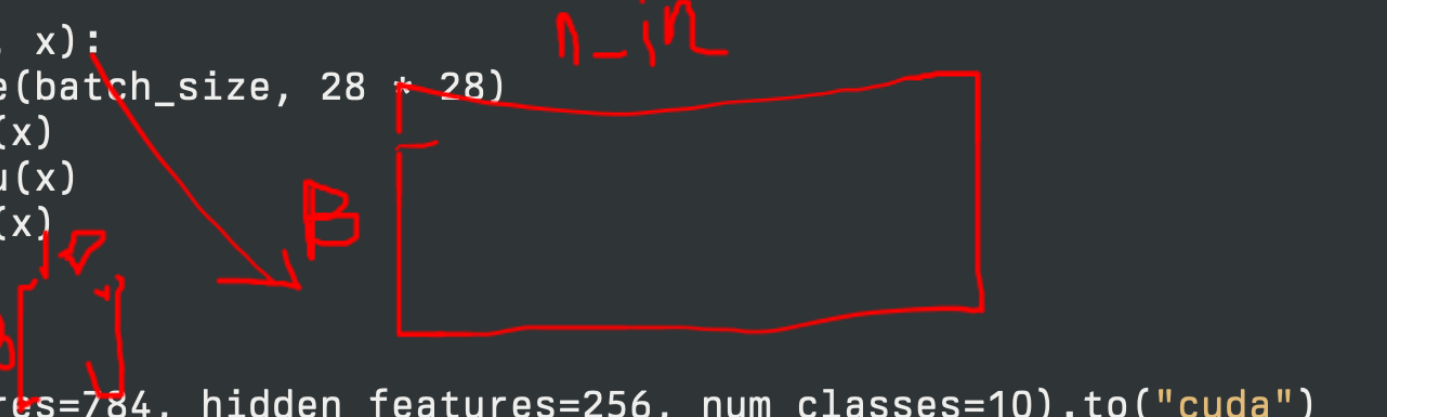
$$L=-\sum y\log(\text{softmax}(z2))$$

PyTorch MLP: Network

```python
class MLP(nn.Module):
    def __init__(self, in_features, hidden_features, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_features, num_classes)

    def forward(self, x):
        x = x.reshape(batch_size, 28 * 28)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x


model = MLP(in_features=784, hidden_features=256, num_classes=10).to("cuda")
# model = torch.compile(model)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```



PyTorch MLP: Train

```python
# Training the model
def train(model, criterion, optimizer, epoch):
    model.train()
    running_loss = 0.0

    for i in range(iters_per_epoch):

        optimizer.zero_grad()
        data = train_data[i * batch_size : (i + 1) * batch_size].to("cuda")
        target = train_labels[i * batch_size : (i + 1) * batch_size].to("cuda")

        start = time.time()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        end = time.time()
        running_loss += loss.item()
        if i % 100 == 99 or i == 0:
            print(f"Epoch: {epoch+1}, Iter: {i+1}, Loss: {loss}")
            print(f"Iteration Time: {(end - start) * 1e3:.4f} sec")
            running_loss = 0.0
```

PyTorch: Evaluate

```python
# Evaluation function to report average batch accuracy using the loaded test data
def evaluate(model, test_data, test_labels):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()

    total_batch_accuracy = torch.tensor(0.0, device=device)
    num_batches = 0

    with torch.no_grad():
        for i in range(len(test_data) // batch_size):
            data = test_data[i * batch_size : (i + 1) * batch_size].to(device)
            target = test_labels[i * batch_size : (i + 1) * batch_size].to(device)
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            correct_batch = (predicted == target).sum().item()
            total_batch = target.size(0)
            if total_batch != 0:  # Check to avoid division by zero
                batch_accuracy = correct_batch / total_batch
                total_batch_accuracy += batch_accuracy
                num_batches += 1

    avg_batch_accuracy = total_batch_accuracy / num_batches
    print(f"Average Batch Accuracy: {avg_batch_accuracy * 100:.2f}%")
```

Python MLP without PyTorch

- What actually does PyTorch do?
  - Layers are just functions mapping input vector -> output vector
  - .backward(): automatically computing gradients using chain rules
  - Train: iteration of computing gradient and update parameters using some variant of stochastic gradient descent
- For our simple MLP we can derive and hardcode the gradient computing rule, the forward pass of the network, and the training iterations with SGD using nothing other than Numpy linear algebra

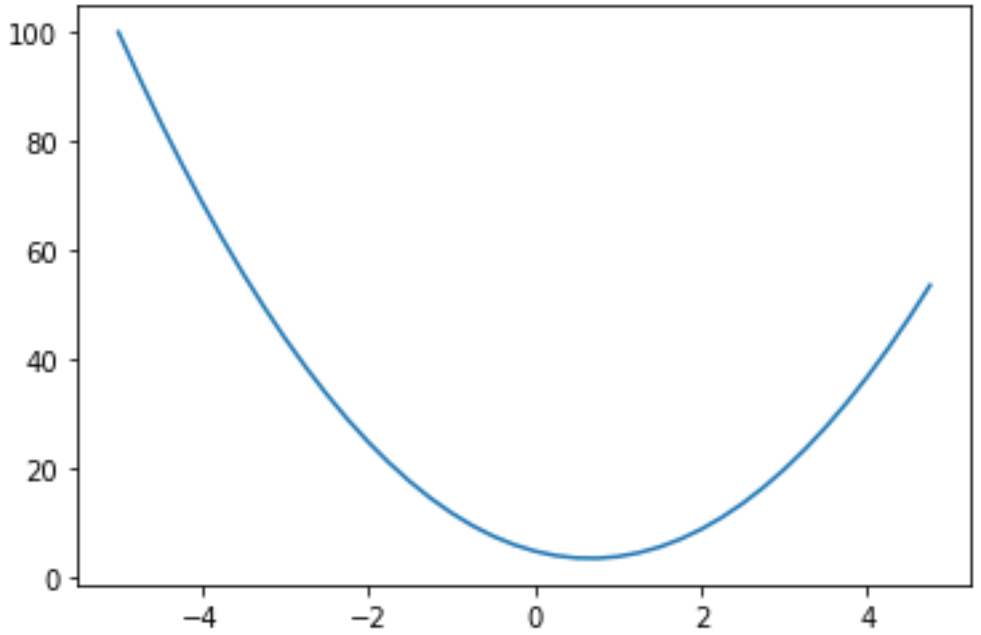
Training: Back-propagation

- Recall training:
  f(x; w) = y
  Optimize w to minimize (cross-entropy) loss function over a training data set (x0,y0), (x1, y1), …, (xn, yn):

$$\min_{w} \sum_{i} L(f(x_i; w), y_i)$$

- Take one training point (x0, y0). How to update w so that loss is lower?
- Intuition: In calculus, we learn that gradient (derivative) at w is the **slope** that the function **increases** around w

def L(w): **return** 3***w****2 **-** 4***w** **+** 5



Gradient & Back-propagation

- Intuitively, the f gradient of w at point x0

$$\nabla_w f(x0, w)$$

  points to the **direction** in which the scalar function f(x0,w) increases the fastest;
- If we want to **reduce** f(x0,w), we would go tiny step in opposite direction of the gradient g: $= \nabla_w f(x0, w)$:

$$\delta w = t * g$$

- $f(x0; w - \delta w) \approx f(x0; w) - (\delta w, g) = f(x0; w) - t\left|\left|g\right|\right|^2$
- So in summary, to minimize the loss function at point (x0,y0), one would update parameters w a tiny amount in the $\nabla L_w(x0; w)$
- But how to compute $\nabla L_w(x0; w)$ for our MLP?
  Chain rule, and layer by layer backwards (that's where back-propagation got its name)

Chain rule

General Chain Rule Principle

For a composite function:

$$L = f(g(h(x)))$$

The gradient of L w.r.t. x is:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

Chain rule applied to the 3-layer net

Forward Pass:

$$
\begin{aligned}
Z_1 &= XW_1 + b_1 && \text{(Layer 1)} \\
A_1 &= \text{ReLU}(Z_1) && \text{(Activation)} \\
Z_2 &= A_1W_2 + b_2 && \text{(Layer 2)} \\
A_2 &= \text{Softmax}(Z_2) && \text{(Output)} \\
L &= \text{CrossEntropy}(A_2, y) && \text{(Loss)}
\end{aligned}
$$

General Chain Rule Principle

For a composite function:

$$L = f(g(h(x)))$$

The gradient of L w.r.t. x is:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

Backward Pass (Gradient Flow):

1. **Output Layer Gradient:**

$$\frac{\partial L}{\partial Z_2} = A_2 - y \quad \text{(Softmax + CrossEntropy derivative)}$$

2. **Layer 2 Weights & Bias:**

$$\frac{\partial L}{\partial W_2} = A_1^T \frac{\partial L}{\partial Z_2}, \quad \frac{\partial L}{\partial b_2} = \sum \frac{\partial L}{\partial Z_2}$$

3. **Gradient to Layer 1 Output:**

$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial Z_2} W_2^T$$

4. **Layer 1 Activation Gradient:**

$$\frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial A_1} \odot \text{ReLU}'(Z_1)$$

5. **Layer 1 Weights & Bias:**

$$\frac{\partial L}{\partial W_1} = X^T \frac{\partial L}{\partial Z_1}, \quad \frac{\partial L}{\partial b_1} = \sum \frac{\partial L}{\partial Z_1}$$

Layer-by-layer Back-propagation

- For an linear layer,
  z = xW + b
  the backward() should compute the following gradients:
  given gradient_out (∂L/∂z), computes gradients for parameters W, b, and for gradient_in(∂L/∂x):

For a linear layer $Z = XW + b$, the gradients are:

- $\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Z}$
- $\frac{\partial L}{\partial b} = \sum \frac{\partial L}{\partial Z}$ (summed over the batch)
- $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} W^T$

Numpy version

- https://github.com/Infatoshi/mnist-cuda/blob/master/python/c-friendly.py

References:

- https://github.com/Infatoshi/mnist-cuda/tree/master
- https://www.kaggle.com/code/scaomath/simple-neural-network-for-mnist-numpy-from-scratch/notebook
- Back-propagation: Andrej Karpathy, MicroGrad: https://www.youtube.com/watch?v=VMj-3S1tku0&t=579s
- Pure CUDA implementation to reproduce GPT-2(124M) model: https://github.com/karpathy/llm.c/discussions/481 https://www.youtube.com/watch?v=l8pRSuU81PU