Intro to Deep Learning Assignment 2 (Part 2)

Tsinghua University
Allan Li (李盼加乐), 2025403473

Part 1: Objective Questions

1. A
2. B
3. A
4. A
5. B, D
6. Two shortcomings of using an MLP are as follows:
    1. First, an MLP treats a given image as a 1D vector of pixel values. The issue with this is that it doesn't capture the relationships of nearby pixels that might combine together to form a new edge/texture, because the model doesn't "know" which pixels might be adjacent to each other. Thus, it cannot naturally capture patterns like corners or shapes that might be core to the distinction of two different objects.
    2. Secondly, in an image, each input pixel is connected to every neuron in the following layer. Thus, the number of parameters grow exponentially with image size. 10,000 pixels (100x100 image) connected to 100 hidden weights already needs 1 million weights. This might make MLPs unsuitable for image recognition because it is sensitive to overfitting, while also requiring a lot of computation power and time to train the model.

Part 2: Programming Practice

Results & console output can be found in `programming_practice.ipynb`

2.1

```
A = torch.rand(5, 5)
B = torch.rand(5, 5)
C = torch.zeros_like(A)

addition = A + B
subtraction = A - B
ele_multiplication = A * B
mat_multiplication = torch.matmul(A, B)
```

2.2

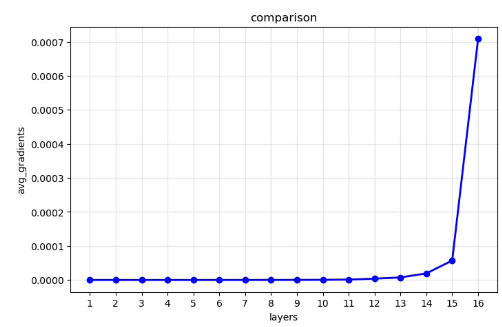
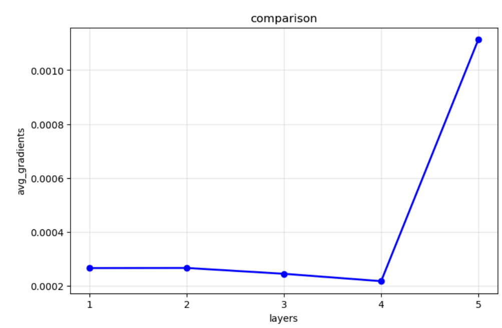
```
A = torch.rand(6, 4)
A_reshaped = A.reshape(24)
A_subset = A[1:5, 1:3]
```

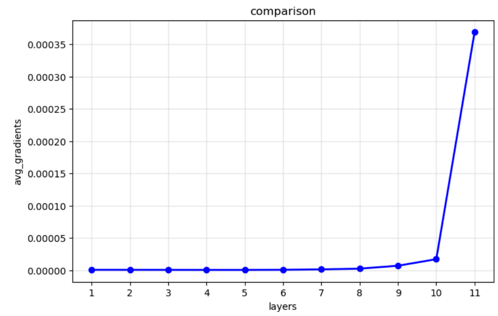
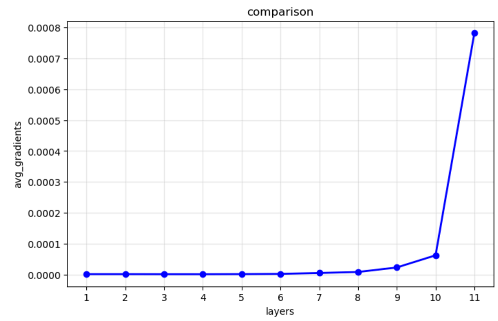
2.3

```
A = torch.rand(3, 3, requires_grad=True)
B = torch.tensor([[1., 0., 2.],
                  [1., 1., 0.],
                  [0., 3., 1.]])
C = A * B
S = C.sum()
S.backward()
```

2.4: Observing Gradients

The following table summarizes differences in gradients:

Model Adjustment	Gradient Analysis	Gradient Visulization
Base Model (No Changes)	Base Model.	
Decrease Layer Count (to 4)	Gradients increase overall. With less layers each layer has a larger impact on reducing the overall loss	

Model Adjustment	Gradient Analysis	Gradient Visulization
Increase Hidden Layer Count (to 512)	Wider layers cause the gradients to become relatively more uniform and stable (compared to base model)	
Decrease Learning Rate (to 0.0001)	Generally similar to the base model - somewhat more polarized gradients.	

Summary of results:

- Generally speaking, the gradients for all models were highest near the latter layers, with a massive increase in the last layer compared to all other layers. This makes sense since they are the last layer before the results, so changing them would heavily influence the results of the model
- Decreasing Layer count and increasing the hidden layer/making layers wider generally caused less polarization between the former and latter layers.
- Learning rate generally didn't change the gradient uniformity, but extreme values of learning rates on both sides would likely make gradients more polarized because the model instability increases

Simple Linear Regression Model

```
torch.manual_seed(42)
N = 256
x = torch.linspace(-2, 2, N).unsqueeze(1)
true_m, true_b = 3.0, 0.7
y = true_m * x + true_b + 0.2 * torch.randn_like(x)

class SimpleLinear(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(1, 1)
```

```python
    def forward(self, x):
        return self.fc(x)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimpleLinear().to(device)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.05)

model.train()
data, target = x.to(device), y.to(device)

optimizer.zero_grad()
output = model(data)
loss = criterion(output, target)
loss.backward()
optimizer.step()

m_learned = model.fc.weight.item()
b_learned = model.fc.bias.item()
print(f"MSE Training loss after 1 epoch: {loss.item()}")
print(f"Learned params: w={m_learned}, b={b_learned}")

save_path = "weights.pt"
torch.save(model.state_dict(), save_path)
print(f"Saved weights to: {save_path}")

new_model = SimpleLinear().to(device)
new_model.load_state_dict(torch.load(save_path, map_location=device))
new_model.eval()

with torch.no_grad():
    test_x = torch.tensor([[-1.5], [0.0], [1.5]]).to(device)
    test_y_hat = new_model(test_x)
print(f"True params: w={true_m}, b={true_b}")
print("Predictions for x=[-1.5, 0.0, 1.5]:",
test_y_hat.squeeze().cpu().numpy())
print(f"Actual y values: {true_m * test_x.squeeze().cpu().numpy() + true_b}")
```

Results:

- MSE Training loss after 1 epoch: 10.019256591796875
- Learned params: w=0.8872767686843872, b=-0.5553232431411743
- True params: w=3.0, b=0.7
- Predictions for x=[-1.5, 0.0, 1.5]: [-1.8862385 -0.55532324 0.77559197]

- Actual y values: [-3.8 0.7 5.2]