Analysis for hw1

Related Path

- ipynb file:https://github.com/RuiqiTang/Deep_Learning_2025_Spring_Lesson_Homworks_Submission/blob/main/HW1_submission/hw1_sol.ipynb
- saved parameters:https://github.com/RuiqiTang/Deep_Learning_2025_Spring_Lesson_Homworks_Submission/blob/main/HW1_submission/best_model.npz
- Detaset downloaded from: http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

Load dataset

- load train_dataset & test_dataset
- compose transformation on picture datasets
- set batch_size=64
- Transform `train_dataset` and `test_dataset` into DataLoader type.

```
from torch.utils.data import DataLoader, SubsetRandomSampler

# 定义数据变换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # 归一化到[-1, 1]
])

# 加载数据集
train_dataset = torchvision.datasets.CIFAR10(
    root=r'nn_hw1',
    train=True,
    transform=transform,
    download=True
)
test_dataset = torchvision.datasets.CIFAR10(
    root=r'nn_hw1',
    train=False,
    transform=transform,
    download=True
)

# 划分训练集和验证集
indices = np.arange(len(train_dataset))
np.random.shuffle(indices)
split = int(0.9 * len(train_dataset))
train_indices, val_indices = indices[:split], indices[split:]

# 创建DataLoader
batch_size = 64
```

```
train_loader = DataLoader(train_dataset, batch_size=batch_size,
sampler=SubsetRandomSampler(train_indices))
val_loader = DataLoader(train_dataset, batch_size=batch_size,
sampler=SubsetRandomSampler(val_indices))
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Define Net structure

- The Neural Network includes 3 layers
  - Input Layer-> Hidden Layer 1(size=hidden_size[0])
  - Hidden Layer 1->Hidden Layer 2(size=hidden_size[1])
  - Hidden Layer 2->Output Layer(size=output_size)
- Use `W = np.random.randn(n_in, n_out) * np.sqrt(2./n_in)` to keep that the var of the output of each layer as `2./n_in` to avoid gradient vanishing/exploding.
- Forward process:

```
Z₁ = X·W₁ + b₁     # 线性变换
A₁ = σ(Z₁)         # 激活函数

Z₂ = A₁·W₂ + b₂
A₂ = σ(Z₂)

Z₃ = A₂·W₃ + b₃    # 无激活函数（直接输出logits）
```

- Backpropogation:
  - Use cross-entropy loss plus L2 regularization.

```
L = -1/m Σ y·log(ŷ) + λ/2(||W₁||² + ||W₂||² + ||W₃||²)
```

  - Calculate the gradients of the hidden layer:

```
# Output Layer
dZ₃ = (probs - one_hot_y)/m   # 推导自交叉熵导数
# Use Softmax
probs = exp(Z₃ - max(Z₃)) / sum(exp(Z₃ - max(Z₃)))
∂L/∂W₃ = A₂ᵀ·dZ₃ + λW₃
∂L/∂b₃ = sum(dZ₃, axis=0)

# The second hidden layer
dA₂ = dZ₃·W₃ᵀ
dZ₂ = dA₂ ⊙ σ'(Z₂)    # Hadamard积
∂L/∂W₂ = A₁ᵀ·dZ₂ + λW₂
∂L/∂b₂ = sum(dZ₂, axis=0)

# The first
dA₁ = dZ₂·W₂ᵀ
```

```
dZ₁ = dA₁ ⊙ σ'(Z₁)
∂L/∂W₁ = Xᵀ·dZ₁ + λW₁
∂L/∂b₁ = sum(dZ₁, axis=0)
```

- Use SGD:

```
W = W - η·∂L/∂W
b = b - η·∂L/∂b
```

```python
class ThreeLayerNN:
    def __init__(self, input_size, hidden_sizes, output_size, activations):
        self.params = {
            'W1': np.random.randn(input_size, hidden_sizes[0]) * np.sqrt(2. / input_size),
            'b1': np.zeros(hidden_sizes[0]),
            'W2': np.random.randn(hidden_sizes[0], hidden_sizes[1]) * np.sqrt(2. / hidden_sizes[0]),
            'b2': np.zeros(hidden_sizes[1]),
            'W3': np.random.randn(hidden_sizes[1], output_size) * np.sqrt(2. / hidden_sizes[1]),
            'b3': np.zeros(output_size)
        }
        self.activations = activations

    def forward(self, X):
        self.cache = {}
        # Layer 1
        Z1 = X.dot(self.params['W1']) + self.params['b1']
        A1 = self._activate(Z1, self.activations[0])
        self.cache['Z1'], self.cache['A1'] = Z1, A1
        # Layer 2
        Z2 = A1.dot(self.params['W2']) + self.params['b2']
        A2 = self._activate(Z2, self.activations[1])
        self.cache['Z2'], self.cache['A2'] = Z2, A2
        # Output layer
        Z3 = A2.dot(self.params['W3']) + self.params['b3']
        self.cache['Z3'] = Z3
        return Z3

    def _activate(self, Z, activation):
        if activation == 'relu':
            return np.maximum(0, Z)
        elif activation == 'sigmoid':
            return 1 / (1 + np.exp(-Z))
        elif activation == 'tanh':
            return np.tanh(Z)
        else:
            raise ValueError("Unsupported activation")

    def backward(self, X, y, reg_lambda):
        m = X.shape[0]
        grads = {}
```

```
        # 获取缓存
        Z1, A1 = self.cache['Z1'], self.cache['A1']
        Z2, A2 = self.cache['Z2'], self.cache['A2']
        Z3 = self.cache['Z3']
        W3 = self.params['W3']

        # 计算softmax梯度
        probs = np.exp(Z3 - np.max(Z3, axis=1, keepdims=True))
        probs /= np.sum(probs, axis=1, keepdims=True)
        one_hot = np.eye(10)[y]
        dZ3 = (probs - one_hot) / m

        # 第三层梯度
        grads['W3'] = A2.T.dot(dZ3) + reg_lambda * self.params['W3']
        grads['b3'] = np.sum(dZ3, axis=0)

        # 第二层梯度
        dA2 = dZ3.dot(W3.T)
        dZ2 = dA2 * self._activate_deriv(Z2, self.activations[1])
        grads['W2'] = A1.T.dot(dZ2) + reg_lambda * self.params['W2']
        grads['b2'] = np.sum(dZ2, axis=0)

        # 第一层梯度
        dA1 = dZ2.dot(self.params['W2'].T)
        dZ1 = dA1 * self._activate_deriv(Z1, self.activations[0])
        grads['W1'] = X.T.dot(dZ1) + reg_lambda * self.params['W1']
        grads['b1'] = np.sum(dZ1, axis=0)

        return grads

    def _activate_deriv(self, Z, activation):
        if activation == 'relu':
            return (Z > 0).astype(float)
        elif activation == 'sigmoid':
            s = 1 / (1 + np.exp(-Z))
            return s * (1 - s)
        elif activation == 'tanh':
            return 1 - np.tanh(Z)**2
        else:
            raise ValueError("Unsupported activation")
```

Train & Test Results

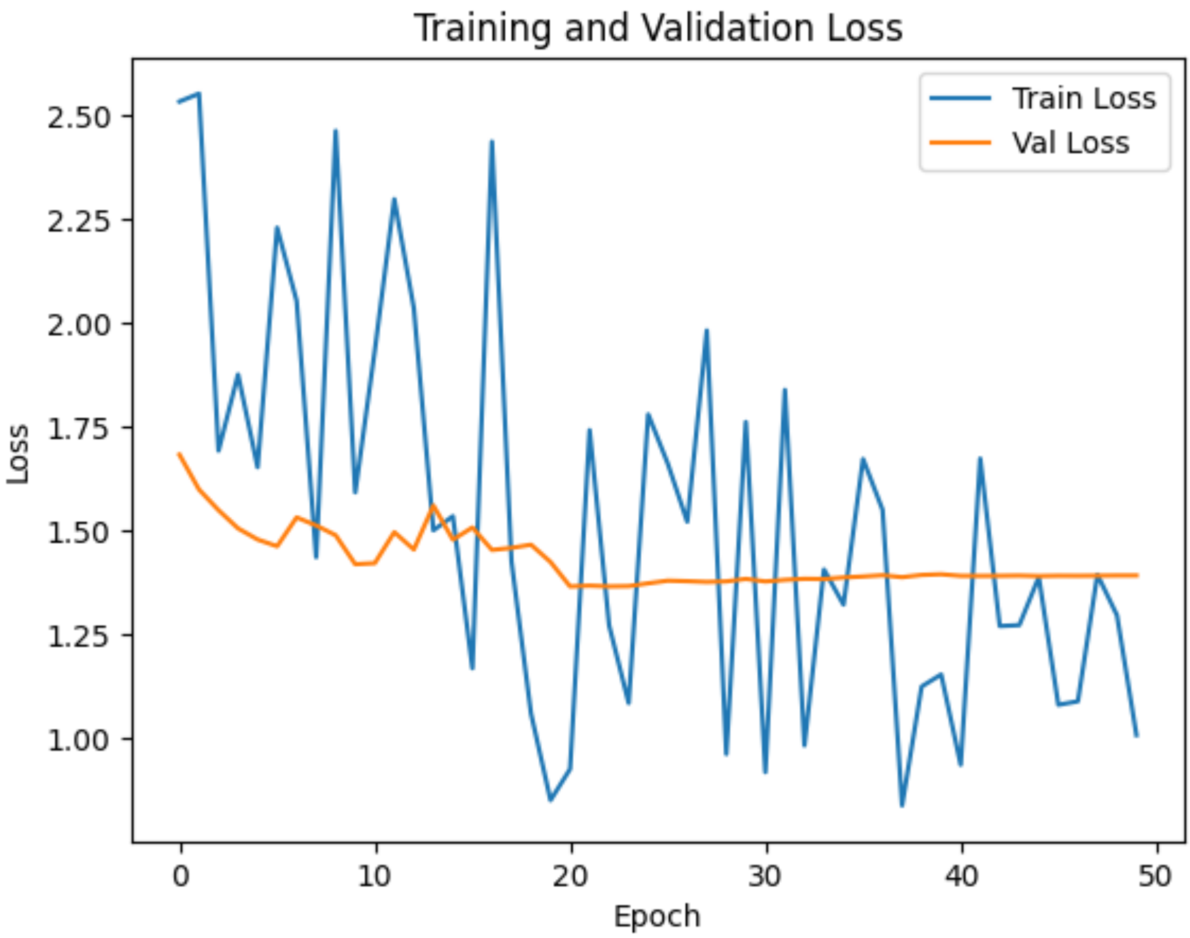
Best Params

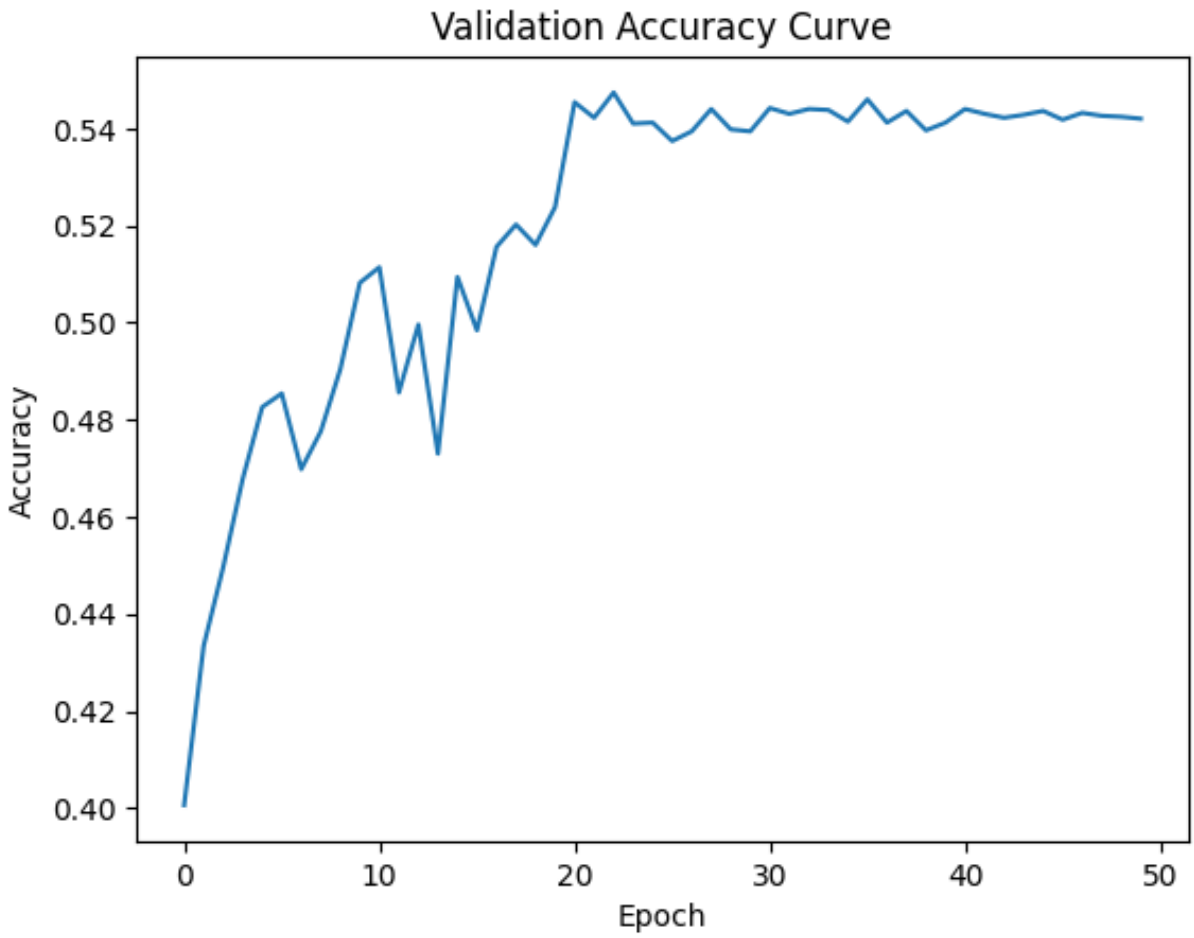
- size of hidden layer 1:512
- size of hidden layer 2:256
- lr: 1e-2
- reg: 1e-3

Test Accuracy

- Test Accuracy: 0.5421
- Test Loss: 1.3361

Visualization





Validation Accuracy Curve
0.40
0.42
0.44
0.46
0.48
0.50
0.52
0.54
Accuracy
0
10
20
30
40
50
Epoch