HW1

2021320071_허천윤

[2]:
```
import torch
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
y = torch.ones_like(x) * 2
result = x + y
print(result)
```

```
tensor([[3, 4, 5],
        [6, 7, 8]])
```

[4]:
```
import pandas as pd from sklearn.preprocessing
import StandardScaler data = {'feature1': [1, 2, 3],
'feature2': [4, 5, None]} df = pd.DataFrame(data)
df.fillna(df.mean(), inplace=True) # Handle missing
values scaler = StandardScaler() scaled_data =
scaler.fit_transform(df) print(scaled_data)
```

```
[[-1.22474487 -1.22474487]
 [ 0.          1.22474487]
 [ 1.22474487 0.        ]]
```

[7]:
```
from sklearn.linear_model import LinearRegression
X = torch.tensor([[1], [2], [3],
[4]]) y = torch.tensor([2, 4, 6,
8]) model = LinearRegression()
model.fit(X.numpy(), y.numpy())
print(model.coef_,
model.intercept_)
```

```
[2.] 0.0
```

[9]:
```
import torch
import torch.nn as nn
import torch.optim as optim

class LinearRegressionModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, output_dim)
```

```
    def forward(self, x):
        return self.linear(x)

class LinearRegression:
    def __init__(self, input_dim, output_dim, learning_rate=0.01):
        self.model = LinearRegressionModel(input_dim, output_dim)
        self.criterion = nn.MSELoss()
        self.optimizer = optim.SGD(self.model.parameters(), lr=learning_rate)

    def train(self, X_train, y_train, epochs=100):
        for epoch in range(epochs):
            self.model.train()
            self.optimizer.zero_grad()
            outputs = self.model(X_train)
            loss = self.criterion(outputs, y_train)
            loss.backward()
            self.optimizer.step()
            if (epoch+1) % 10 == 0:
                print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

    def predict(self, X):
        self.model.eval()
        with torch.no_grad():
            return self.model(X)
```

[10]:
```
import numpy as np

class LinearRegressionFromScratch:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        # Initialize parameters
        self.weights = np.zeros(X.shape[1])
        self.bias = 0

        # Gradient Descent
        for _ in range(self.epochs):
            y_pred = self.predict(X)
            dw = (2 / X.shape[0]) * np.dot(X.T, (y_pred - y))
            db = (2 / X.shape[0]) * np.sum(y_pred - y)
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
```

```python
        return np.dot(X, self.weights) + self.bias
```

[ ]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

class SoftmaxRegressionModel(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(SoftmaxRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, num_classes)

    def forward(self, x):
        return self.linear(x)

class SoftmaxRegression:
    def __init__(self, input_dim, num_classes, learning_rate=0.01):
        self.model = SoftmaxRegressionModel(input_dim, num_classes)
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = optim.SGD(self.model.parameters(), lr=learning_rate)

    def train(self, X_train, y_train, epochs=100):
        for epoch in range(epochs):
            self.model.train()
            self.optimizer.zero_grad()
            outputs = self.model(X_train)
            loss = self.criterion(outputs, y_train)
            loss.backward()
            self.optimizer.step()
            if (epoch+1) % 10 == 0:
                print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

    def predict(self, X):
        self.model.eval()
        with torch.no_grad():
            outputs = self.model(X)
            return torch.argmax(outputs, dim=1)
```

[ ]:
```
4.2 The Image Classification Dataset
Image classification datasets often consist of images labeled with categories.
 ↪Common datasets include MNIST, CIFAR-10, and ImageNet.

Discussion Points:
Preprocessing: Resize, normalize, and augment images.
Splitting: Divide the dataset into training, validation, and test sets.
```

[ ]:
```python
class SimpleNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes):
```

```
        super(SimpleNNModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, num_classes)

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

[ ]:

```
class SoftmaxRegressionFromScratch:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        # Initialize parameters
        self.weights = np.zeros((X.shape[1], len(np.unique(y))))
        self.bias = np.zeros(len(np.unique(y)))

        for _ in range(self.epochs):
            logits = np.dot(X, self.weights) + self.bias
            probs = self.softmax(logits)
            loss = self.cross_entropy(probs, y)
            grads = self.compute_gradients(X, y, probs)
            self.weights -= self.learning_rate * grads['dw']
            self.bias -= self.learning_rate * grads['db']

    def softmax(self, logits):
        exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))
        return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

    def cross_entropy(self, probs, y):
        m = y.shape[0]
        log_likelihood = -np.log(probs[range(m), y])
        return np.sum(log_likelihood) / m

    def compute_gradients(self, X, y, probs):
        m = y.shape[0]
        grads = {}
        one_hot_y = np.eye(probs.shape[1])[y]
        d_logits = (probs - one_hot_y) / m
        grads['dw'] = np.dot(X.T, d_logits)
        grads['db'] = np.sum(d_logits, axis=0)
        return grads
```

```python
    def predict(self, X):
        logits = np.dot(X, self.weights) + self.bias
        probs = self.softmax(logits)
        return np.argmax(probs, axis=1)
```

[8]:
```python
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(2, 3)  # Hidden layer
        self.output = nn.Linear(3, 1)  # Output layer

    def forward(self, x):
        x = torch.relu(self.hidden(x))
        return self.output(x)

model = MLP()
print(model)
```

```
MLP(
     (hidden): Linear(in_features=2, out_features=3,
                                          bias=True)
(output): Linear(in_features=3, out_features=1,
bias=True) )
```

[ ]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim):
        super(MLP, self).__init__()
        self.layers = nn.ModuleList()

        # Input layer
        self.layers.append(nn.Linear(input_dim, hidden_dims[0]))
        self.layers.append(nn.ReLU())

        # Hidden layers
        for i in range(len(hidden_dims) - 1):
            self.layers.append(nn.Linear(hidden_dims[i], hidden_dims[i+1]))
            self.layers.append(nn.ReLU())

        # Output layer
        self.layers.append(nn.Linear(hidden_dims[-1], output_dim))

    def forward(self, x):
        for layer in self.layers:
```

```
            x = layer(x)
        return x

# Example usage
input_dim = 784  # Example input dimension (e.g., 28x28 images)
hidden_dims = [128, 64]  # Example hidden layer dimensions
output_dim = 10  # Example number of output classes

model = MLP(input_dim, hidden_dims, output_dim)
print(model)
```

[11]:
```
# Forward pass
def forward_pass(model, X):
    return model(X)

# Example usage# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Example training loop
def train(model, X_train, y_train, epochs=100):
    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        outputs = forward_pass(model, X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
        if (epoch+1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

# Example usage
y_train = torch.randint(0, output_dim, (5,))  # Random labels
train(model, X, y_train)

X = torch.randn(5, input_dim)  # Batch of 5 examples
outputs = forward_pass(model, X)
print(outputs)
```

```
---------------------------------------------------------------------
------
NameError                              Traceback (most recent call last)
Cell In[11], line 22
     19print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f} 21
     # Example usage
---> 22 y_train = torch.randint(0, output_dim, (5,)) # Random labels
     23 train(model, X, y_train)
     25 X = torch.randn(5, input_dim) # Batch of 5 examples
')
```

```
NameError: name 'output_dim' is not defined
```

```
[ ]:
```