DAT341 / DIT867 Applied machine learning
Assignment 4

Chunqiu Xia Chunqiu@student.chalmers.se
Yicheng Li yicheng@student.chalmers.se
Wenjun Tian wenjunt@chalmers.se

March 2, 2025

[90]:
```
import pandas as pd
import numpy as np
import torch
```

1 Task 1

Loading the synthetic dataset. The input data $D = \{(\vec{x_i}, y_i)\}_{i=1}^{n}$ looks like:

$$X = \begin{pmatrix} \vec{x_1} \\ \vec{x_2} \end{pmatrix}, Y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

[91]:
```
# You may need to edit the path, depending on where you put the files.
data = pd.read_csv('data/a4_synthetic.csv')

X = data.drop(columns='y').to_numpy()
Y = data.y.to_numpy()
```

Training a linear regression model for this synthetic dataset.

[92]:
```
np.random.seed(1)

w_init = np.random.normal(size=(2, 1))
b_init = np.random.normal(size=(1, 1))

# We just declare the parameter tensors. Do not use nn.Linear.
w = torch.tensor(w_init, requires_grad=True) # col vector W = (w_1; w_2)
b = torch.tensor(b_init, requires_grad=True) # scalar

eta = 1e-2
# SGD optimizer with a learning rate of eta
# Parameters include W and b
opt = torch.optim.SGD([w, b], lr=eta)

for i in range(10):
```

```
    sum_err = 0

    for row in range(X.shape[0]):
        x = torch.tensor(X[[row], :]) # row vector X_i = (x1, x2)
        y = torch.tensor(Y[[row]])

        # Forward pass.
        # compute predicted value for x
        y_pred = w.T @ x.T + b
        # compute squared error loss
        err = (y - y_pred) ** 2

        # Backward and update.
        # compute gradients and then update the model.
        opt.zero_grad() # Get rid of previously computed gradients.
        err.backward() #Compute the gradients.
        opt.step() #Update the model.

        # For statistics.
        sum_err += err.item()

    mse = sum_err / X.shape[0]
    print(f'Epoch {i+1}: MSE =', mse)
```

```
Epoch 1: MSE = 0.799966113082318
Epoch 2: MSE = 0.017392390107906882
Epoch 3: MSE = 0.009377418010839892
Epoch 4: MSE = 0.009355326971438456
Epoch 5: MSE = 0.009365440968904255
Epoch 6: MSE = 0.009366989180952537
Epoch 7: MSE = 0.009367207398577986
Epoch 8: MSE = 0.009367238983974492
Epoch 9: MSE = 0.009367243704122532
Epoch 10: MSE = 0.009367244427185763
```

2 Task 2, 3, 4

2.1 Computation Node

Definition of computation nodes is as follows.

Overall structure For the tensor calculation `z = x + y`, tensor `z` holds a `AdditionNode` with `left = x, right = y`.

Backward function In the `backward()` for a certain kind of `Node`, we calculate the following:

$$l\_grad = \frac{\partial Loss}{\partial x} = \frac{\partial Loss}{\partial z}\frac{\partial z}{\partial x} = grad\_output@\frac{\partial z}{\partial x}$$

`r_grad` is simliar as above.

We get `grad_output` from the function input, and calculate $\frac{\partial z}{\partial x}$ according to the computation type.

After we calculate the gradient for `x, y`, we invoke `propagate()` to continue backwarding to the deeper layer.

[4]:
```
class Node:
    def __init__(self, left, right):
        # left: Tensor
        self.left = left
        # right: tensor | int (for power only)
        self.right = right

    def backward(self, grad_output):
        raise NotImplementedError('Unimplemented')

    # Invoke backward() for left&right tensors(operands).
    def propagate(self, l_grad, r_grad):
        self.left.backward(l_grad)
        # when powering, we don't need a derivative w.r.t. the exponent.
        if isinstance(r_grad, np.ndarray):
            self.right.backward(r_grad)

    def __repr__(self):
        return str(type(self))


class AdditionNode(Node):
    def backward(self, grad_output):
        l_grad = grad_output
        r_grad = grad_output
        self.propagate(l_grad, r_grad)

class SubstractionNode(Node):
    def backward(self, grad_output):
        l_grad = grad_output
        r_grad = -grad_output
        self.propagate(l_grad, r_grad)

class MatMulNode(Node):
    def backward(self, grad_output):
        l_grad = grad_output @ self.right.data.T
        r_grad = self.left.data.T @ grad_output
        self.propagate(l_grad, r_grad)

class PowerNode(Node):
    def backward(self, grad_output):
```

```
        base, exp = self.left.data, self.right
        par_der = exp * (base ** (exp - 1))
        l_grad = grad_output @ par_der
        self.propagate(l_grad, None)
```

2.2 Tensor

We construct our own `Tensor` as follows.

Overall structure A `tensor` has a `np.ndarray` for storing data, a `Node` called `grad_fn` to record computation, and `requires_grad | grad` to store the gradient of loss function w.r.t. the tensor.

Backward function The `backward()` for `Tensor` looks like:

```
if tensor x is a calculated value:
    Back propagate grad_ouput.
else:
    Exit recursion. Store grad if needed
```

At the very being of the backward, we want calculate $\frac{\partial Loss}{\partial Loss}$ as the `grad_output`. Note that the value of a loss function is a scalar. Thus $\frac{\partial Loss}{\partial Loss} = 1$. For convenient computation, we turn it into a 1x1 matrix.

Arithmetic operation For every operation, we create the corresponding computation `Node`.

[9]:
```
class Tensor:

    # Constructor. Just store the input values.
    def __init__(self, data, requires_grad=False, grad_fn=None):
        #data: ndarray
        self.data = data
        self.shape = data.shape
        #grad_fn: Node | None
        self.grad_fn = grad_fn
        self.requires_grad = requires_grad
        #grad: None | np.ndarray
        self.grad = None

    # So that we can print the object or show it in a notebook cell.
    def __repr__(self):
        dstr = repr(self.data)
        if self.requires_grad:
            gstr = ', requires_grad=True'
        elif self.grad_fn is not None:
            gstr = f', grad_fn={self.grad_fn}'
        else:
            gstr = ''
        return f'Tensor({dstr}{gstr})'
```

```
    # Extract one numerical value from this tensor.
    def item(self):
        return self.data.item()

    # YOUR WORK WILL BE DONE BELOW

    # For Task 2:

    # Operator +
    def __add__(self, right):
        # Call the helper function defined below.
        return addition(self, right)

    # Operator -
    def __sub__(self, right):
        return substraction(self, right)

    # Operator @
    def __matmul__(self, right):
        return matrix_multiplication(self, right)

    # Operator **
    def __pow__(self, right):
        # NOTE! We are assuming that right is an integer here, not a Tensor!
        if not isinstance(right, int):
            raise Exception('only integers allowed')
        if right < 2:
            raise Exception('power must be >= 2')
        return power(self, right)


    # Backward computations. Will be implemented in Task 4.
    def backward(self, grad_output=None):
        # We first check if this tensor has a grad_fn: that is, one of the
        # nodes that you defined in Task 3.
        if self.grad_fn is not None:
            # If grad_fn is defined, we have computed this tensor using some␣
↪operation.
            if grad_output is None:
                # This is the starting point of the backward computation.
                # This will typically be the tensor storing the output of
                # the loss function, on which we have called .backward()
                # in the training loop.

                # Generally the value of a loss function is a scalar.
                # Thus Loss/Loss = 1.
```

```
                # For convenient computation, we turn it into a 1x1 matrix
                self.grad_fn.backward(np.eye(1))
            else:
                # This is an intermediate node in the computational graph.
                # This corresponds to any intermediate computation, such as
                # a hidden layer.
                self.grad_fn.backward(grad_output)
        else:
            # If grad_fn is not defined, this is an endpoint in the computational
            # graph: learnable model parameters or input data.
            if self.requires_grad:
                # This tensor *requires* a gradient to be computed. This will
                # typically be a tensor that holds learnable parameters.
                self.grad = grad_output
            else:
                # This tensor *does not require* a gradient to be computed. This
                # will typically be a tensor holding input data.
                pass


# A small utility where we simply create a Tensor object. We use this to
# mimic torch.tensor.
def tensor(data, requires_grad=False):
    return Tensor(data, requires_grad)

# We define helper functions to implement the various arithmetic operations.

# This function takes two tensors as input, and returns a new tensor holding
# the result of an element-wise addition on the two input tensors.
def addition(left, right):
    new_data = left.data + right.data
    grad_fn = AdditionNode(left, right)
    return Tensor(new_data, grad_fn=grad_fn)

def substraction(left, right):
    new_data = left.data - right.data
    grad_fn = SubstractionNode(left, right)
    return Tensor(new_data, grad_fn=grad_fn)

def matrix_multiplication(left, right):
    new_data = left.data @ right.data
    grad_fn = MatMulNode(left, right)
    return Tensor(new_data, grad_fn=grad_fn)

def power(left, right): # left = base, and right = exp
    new_data = left.data ** right
```

```
    grad_fn = PowerNode(left, right)
    return Tensor(new_data, grad_fn=grad_fn)
```

2.3 Sanity Checks for Task 2, 3, 4

Some sanity checks for Task 2.

[95]:
```
# Two tensors holding row vectors.
x1 = tensor(np.array([[2.0, 3.0]]))
x2 = tensor(np.array([[1.0, 4.0]]))
# A tensors holding a column vector.
w = tensor(np.array([[-1.0], [1.2]]))

# Test the arithmetic operations.
test_plus = x1 + x2
test_minus = x1 - x2
test_power = x2 ** 2
test_matmul = x1 @ w

print(f'Test of addition: {x1.data} + {x2.data} = {test_plus.data}')
print(f'Test of subtraction: {x1.data} - {x2.data} = {test_minus.data}')
print(f'Test of power: {x2.data} ** 2 = {test_power.data}')
print(f'Test of matrix multiplication: {x1.data} @ {w.data} = {test_matmul.
 ↪data}')

# Check that the results are as expected. Will crash if there is a␣
 ↪miscalculation.
assert(np.allclose(test_plus.data, np.array([[3.0, 7.0]])))
assert(np.allclose(test_minus.data, np.array([[1.0, -1.0]])))
assert(np.allclose(test_power.data, np.array([[1.0, 16.0]])))
assert(np.allclose(test_matmul.data, np.array([[1.6]])))
```

```
Test of addition: [[2. 3.]] + [[1. 4.]] = [[3. 7.]]
Test of subtraction: [[2. 3.]] - [[1. 4.]] = [[ 1. -1.]]
Test of power: [[1. 4.]] ** 2 = [[ 1. 16.]]
Test of matrix multiplication: [[2. 3.]] @ [[-1. ]
 [ 1.2]] = [[1.6]]
```

Sanity check for Task 3.

[96]:
```
x = tensor(np.array([[2.0, 3.0]]))
w1 = tensor(np.array([[1.0, 4.0]]), requires_grad=True)
w2 = tensor(np.array([[3.0, -1.0]]), requires_grad=True)

test_graph = x + w1 + w2

print('Computational graph top node after x + w1 + w2:', test_graph.grad_fn)
```

```
assert(isinstance(test_graph.grad_fn, AdditionNode))
assert(test_graph.grad_fn.right is w2)
assert(test_graph.grad_fn.left.grad_fn.left is x)
assert(test_graph.grad_fn.left.grad_fn.right is w1)
```

```
Computational graph top node after x + w1 + w2: <class '__main__.AdditionNode'>
```

Sanity check for Task 4.

[97]:
```
x = tensor(np.array([[2.0, 3.0]]))
w = tensor(np.array([[-1.0], [1.2]]), requires_grad=True)
y = tensor(np.array([[0.2]]))

# We could as well write simply loss = (x @ w - y)**2
# We break it down into steps here if you need to debug.

model_out = x @ w
diff = model_out - y
loss = diff ** 2

loss.backward()

print('Gradient of loss w.r.t. w =\n', w.grad)

assert(np.allclose(w.grad, np.array([[5.6], [8.4]])))
assert(x.grad is None)
assert(y.grad is None)
```

```
Gradient of loss w.r.t. w =
 [[5.6]
 [8.4]]
```

An equivalent cell using PyTorch code. Your implementation should give the same result for `w.grad`.

[98]:
```
pt_x = torch.tensor(np.array([[2.0, 3.0]]))
pt_w = torch.tensor(np.array([[-1.0], [1.2]]), requires_grad=True)
pt_y = torch.tensor(np.array([[0.2]]))

pt_model_out = pt_x @ pt_w
pt_model_out.retain_grad() # Keep the gradient of intermediate nodes for␣
 ↪debugging.

pt_diff = pt_model_out - pt_y
pt_diff.retain_grad()

pt_loss = pt_diff ** 2
pt_loss.retain_grad()

pt_loss.backward()
```

```
pt_w.grad
```

[98]:
```
tensor([[5.6000],
        [8.4000]], dtype=torch.float64)
```

3 Task 5

We could make class SGD heritat from class Optimizer. Then, according to Gradient Descent formula,

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

for each parameter, if its gradient is not None, update the parameter according to the formula.

[1]:
```
class Optimizer:
    def __init__(self, params):
        self.params = params

    def zero_grad(self):
        for p in self.params:
            p.grad = np.zeros_like(p.data)

    def step(self):
        raise NotImplementedError('Unimplemented')


class SGD(Optimizer):
    def __init__(self, params, lr):
        super().__init__(params)
        self.lr = lr

    def step(self):
        for p in self.params:
            if p.grad is not None:
                p.data -= self.lr * p.grad
```

For each epoch, every sample undergoes forward propagation, loss computation, backpropagation, and parameter updating, while the losses of all samples are accumulated to compute and output the average loss (MSE) at the end of the epoch.

[100]:
```
data = pd.read_csv('data/a4_synthetic.csv')
X = data.drop(columns='y').to_numpy()
Y = data.y.to_numpy()

np.random.seed(1)
w_init = np.random.normal(size=(2, 1))
b_init = np.random.normal(size=(1, 1))

w = tensor(w_init, requires_grad=True)
```

```
b = tensor(b_init, requires_grad=True)

eta = 1e-2
opt = SGD([w, b], lr=eta)

for i in range(10):

    sum_err = 0

    for row in range(X.shape[0]):
        x = tensor(X[[row], :])
        y = tensor(Y[[row]])

        y_pred = x @ w + b
        err = (y - y_pred) ** 2

        opt.zero_grad()
        err.backward()
        opt.step()

        sum_err += err.item()

    mse = sum_err / X.shape[0]
    print(f'Epoch {i+1}: MSE =', mse)
```

```
Epoch 1: MSE = 0.7999661130823179
Epoch 2: MSE = 0.017392390107906875
Epoch 3: MSE = 0.009377418010839892
Epoch 4: MSE = 0.009355326971438458
Epoch 5: MSE = 0.009365440968904258
Epoch 6: MSE = 0.009366989180952535
Epoch 7: MSE = 0.009367207398577987
Epoch 8: MSE = 0.00936723898397449
Epoch 9: MSE = 0.009367243704122534
Epoch 10: MSE = 0.009367244427185761
```

As expected, the result is `almost identical` to that of task 1.

These results indicate that the mean squared error (MSE) dropped dramatically from epoch 1 to epoch 2 and then quickly converged to a very low value (around 0.009367) from epoch 3 onward. This suggests that the model rapidly learned the underlying relationship in the data and reached convergence, with the training error stabilizing at a low level.

4 Task 6

In this part, we leveraged the custom Tensor and automatic differentiation framework developed in Questions 1-4, and extended it with additional operations to implement a binary cross-entropy loss function. We then built a two-layer feedforward neural network, trained it using our custom

SGD optimizer, selected the best model based on validation performance, and finally evaluated its accuracy on the test set.

[2]:
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split
```

To update the Tensor class, add a **transposed tensor** `T` that supports gradient propagation, and convert the method into a class property so that it can be easily accessed.

[10]:
```
class Tensor:
    def __init__(self, data, requires_grad=False, grad_fn=None):
        self.data = data
        self.shape = data.shape
        self.grad_fn = grad_fn
        self.requires_grad = requires_grad
        self.grad = None

    def __repr__(self):
        dstr = repr(self.data)
        if self.requires_grad:
            gstr = ', requires_grad=True'
        elif self.grad_fn is not None:
            gstr = f', grad_fn={self.grad_fn}'
        else:
            gstr = ''
        return f'Tensor({dstr}{gstr})'

    def item(self):
        return self.data.item()

    @property
    def T(self):
        return Tensor(self.data.T, requires_grad=self.requires_grad,␣
↪grad_fn=TransposeNode(self))

    def __add__(self, right):
        return addition(self, right)

    def __sub__(self, right):
        return substraction(self, right)

    def __matmul__(self, right):
        return matrix_multiplication(self, right)

    def __pow__(self, right):
        if not isinstance(right, int):
```

```
            raise Exception('only integers allowed')
        if right < 2:
            raise Exception('power must be >= 2')
        return power(self, right)

    def __mul__(self, right):
        return multiply(self, right)

    def __neg__(self):
        return neg(self)

    def backward(self, grad_output=None):
        if self.grad_fn is not None:
            if grad_output is None:
                self.grad_fn.backward(np.eye(1))
            else:
                self.grad_fn.backward(grad_output)
        else:
            if self.requires_grad:
                self.grad = grad_output

def tensor(data, requires_grad=False):
    return Tensor(data, requires_grad)
```

In the computation graph node definition, a new node is added to support the **negation (unary minus) operation**, which is necessary because when a tensor is negated, the operation must be recorded so that during backpropagation, its gradient is multiplied by -1 to ensure correct gradient propagation.

[11]:
```
class NegNode(Node):
    def backward(self, grad_output):
        self.left.backward(-grad_output)


class TransposeNode(Node):
    def __init__(self, left):
        super().__init__(left, None)
    def backward(self, grad_output):
        self.left.backward(grad_output.T)
    def __repr__(self):
        return "TransposeNode"
```

4.1 Extension for `Tensor` and `Node`

Implementing **scalar (element-wise) multiplication**, the **tanh** and **sigmoid** activation functions, and the **log** function, each with its corresponding backward propagation node. These operations are then combined to form the binary cross-entropy loss function for a single sample, enabling proper gradient propagation during backpropagation.

[ ]:
```
class MultiplyNode(Node):
    def backward(self, grad_output):
        l_grad = grad_output * self.right.data
        r_grad = grad_output * self.left.data
        self.propagate(l_grad, r_grad)

class TanhNode(Node):
    def backward(self, grad_output):
        local_grad = 1 - np.tanh(self.left.data) ** 2
        self.left.backward(grad_output * local_grad)

def tanh(x):
    new_data = np.tanh(x.data)
    grad_fn = TanhNode(x, None)
    return Tensor(new_data, requires_grad=x.requires_grad, grad_fn=grad_fn)

class SigmoidNode(Node):
    def backward(self, grad_output):
        s = 1 / (1 + np.exp(-self.left.data))
        local_grad = s * (1 - s)
        self.left.backward(grad_output * local_grad)

def sigmoid(x):
    new_data = 1 / (1 + np.exp(-x.data))
    grad_fn = SigmoidNode(x, None)
    return Tensor(new_data, requires_grad=x.requires_grad, grad_fn=grad_fn)

class LogNode(Node):
    def backward(self, grad_output):
        local_grad = 1 / self.left.data
        self.left.backward(grad_output * local_grad)

def log(x):
    new_data = np.log(x.data)
    grad_fn = LogNode(x, None)
    return Tensor(new_data, requires_grad=x.requires_grad, grad_fn=grad_fn)

# Binary Cross-Entropy Loss
def bce_loss(pred, target):
    one = tensor(np.ones_like(pred.data))
    l1 = target * log(pred)
    l2 = (one - target) * log(one - pred)
    loss = -(l1 + l2)
    return loss
```

4.2 Model training with hyperparameter search

The structure of the feedfoward neural network we used is the same as the one mentioned in the instructions:

1. Feature learning: 1 hidden layer, tanh activation, with `hidden_dim` neurons.
2. Classification: Logistic regression, cross-entropy loss with SGD optimization, learning rate `eta`.

Other hyperparameters that are not related to network structure:

1. Training for `max_epoch = 50` epochs. Early stopping with `patience = 1` (number of epochs to tolerate validation accuracy decrease).

In the following part, we search in a hyperparameter space, and find the best `(hidden_dim, eta)`. As a result, we reached the best validation accuracy of `80.00%` at `(14, 0.0001)`, and the corresponding test accuracy is `82.22%`

[113]:
```
import math
# Load data, and pre-processing.
a4data = pd.read_csv('data/raisins.csv')
X = scale(a4data.drop(columns='Class'))
Y = (a4data.Class == 'Besni').to_numpy().astype(int)

# train:test:validation = 8:1:1
# Train-test split
Xtrain, Xtemp, Ytrain, Ytemp = train_test_split(X, Y, random_state=114514,␣
↪test_size=0.2, stratify=Y)
# Test-validation split
Xtest, Xval, Ytest, YVal = train_test_split(Xtemp, Ytemp, random_state=1919810,␣
↪test_size=0.5, stratify=Ytemp)

# Neural network structure:
# 1. One hidden layer with tanh activation.
# 2. Logistic regression binary classifer (output: probability of positive).
def forward(x):
    z1 = x @ W1 + b1
    a1 = tanh(z1)
    z2 = a1 @ W2 + b2
    y_prob = sigmoid(z2)
    return y_prob

# Used for validation and test after training
def predict_and_evaluate(input, output):
    correct = 0
    for i in range(len(input)):
        x_sample = tensor(input[i:i+1, :])
        y_pred = forward(x_sample)
        # Classify input based on predicted probability of positive.
```

```
        pred_label = 1 if y_pred.data[0, 0] >= 0.5 else 0
        if pred_label == output[i]:
            correct += 1
    accuracy = correct / len(output)
    return accuracy

# Hyper-parameter search space
learning_rates = [1e-2, 1e-3, 1e-4, 1e-5]
hidden_dims = [3, 7, 14, 21, 35]
max_epoch = 50
patience = 1
np.random.seed(1919810)

# Searching Result
best_config = None
best_accuracy = 0.0
best_model = None

# Searching hyper-parameter space
for hidden_dim in hidden_dims:
    for eta in learning_rates:
        # Dim of input x and output y
        input_dim = 7
        output_dim = 1

        # Randomly initialied W/b for input layer -> hidden layer
        W1_init = np.random.randn(input_dim, hidden_dim)
        b1_init = np.random.randn(1, hidden_dim)
        # Randomly initialied W/b for hidden layer -> output layer
        W2_init = np.random.randn(hidden_dim, output_dim)
        b2_init = np.random.randn(1, output_dim)

        W1 = tensor(W1_init, requires_grad=True)
        b1 = tensor(b1_init, requires_grad=True)
        W2 = tensor(W2_init, requires_grad=True)
        b2 = tensor(b2_init, requires_grad=True)

        # Optimizer: SGD
        opt = SGD([W1, b1, W2, b2], lr=eta)
        accuracy = 0
        accuracy_decrease_count = 0
        # Train model with `max_epoch` epochs with early stopping.
        for epoch in range(max_epoch):
            loss = None
            # Iterate all training data
            for i in range(len(Ytrain)):
                x_sample = tensor(Xtrain[i:i+1, :])
```

```
                y_target = tensor(Ytrain[i:i+1].reshape(1, 1))
                y_pred = forward(x_sample)
                loss = bce_loss(y_pred, y_target)
                opt.zero_grad()
                loss.backward()
                opt.step()
            # Calculate the validation accuracy when using the given␣
↪hyper-parameter
            # (hidden-dim, learning rate eta).
            accuracy_epoch = predict_and_evaluate(Xval, YVal)
            # Early Stopping
            if accuracy_epoch < accuracy:
                accuracy_decrease_count += 1
                if accuracy_decrease_count > patience:
                    break
            else:
                accuracy = accuracy_epoch
        # Update the best hyper-parameter and related info
        if accuracy >= best_accuracy:
            best_accuracy = accuracy
            best_config = {"hidden_dim": hidden_dim, "learning_rate": eta}
            best_model = [W1.data.copy(), b1.data.copy(), W2.data.copy(), b2.
↪data.copy()]
        print(f"(hidden_dim={hidden_dim}, lr={eta}): Accuracy = {accuracy:.6f}␣
↪at epoch: {epoch}")

print("\nBest Hyperparameters:")
print(best_config)
print("Best Overall Validation Accuracy:", best_accuracy)

# Test model accuracy
W1, b1, W2, b2 = best_model
accuracy = predict_and_evaluate(Xtest, Ytest)
print(f"\nTest accuracy with the best hyperparameters: {accuracy}")
```

```
(hidden_dim=3, lr=0.01): Accuracy = 0.688889 at epoch: 2
(hidden_dim=3, lr=0.001): Accuracy = 0.500000 at epoch: 49
(hidden_dim=3, lr=0.0001): Accuracy = 0.766667 at epoch: 8
(hidden_dim=3, lr=1e-05): Accuracy = 0.711111 at epoch: 49
(hidden_dim=7, lr=0.01): Accuracy = 0.744444 at epoch: 2
(hidden_dim=7, lr=0.001): Accuracy = 0.755556 at epoch: 8
(hidden_dim=7, lr=0.0001): Accuracy = 0.800000 at epoch: 16
(hidden_dim=7, lr=1e-05): Accuracy = 0.688889 at epoch: 6
(hidden_dim=14, lr=0.01): Accuracy = 0.777778 at epoch: 4
(hidden_dim=14, lr=0.001): Accuracy = 0.777778 at epoch: 31
(hidden_dim=14, lr=0.0001): Accuracy = 0.800000 at epoch: 3
(hidden_dim=14, lr=1e-05): Accuracy = 0.600000 at epoch: 49
```

```
(hidden_dim=21, lr=0.01): Accuracy = 0.777778 at epoch: 2
(hidden_dim=21, lr=0.001): Accuracy = 0.711111 at epoch: 7
(hidden_dim=21, lr=0.0001): Accuracy = 0.700000 at epoch: 36
(hidden_dim=21, lr=1e-05): Accuracy = 0.377778 at epoch: 11
(hidden_dim=35, lr=0.01): Accuracy = 0.688889 at epoch: 4
(hidden_dim=35, lr=0.001): Accuracy = 0.600000 at epoch: 11
(hidden_dim=35, lr=0.0001): Accuracy = 0.777778 at epoch: 49
(hidden_dim=35, lr=1e-05): Accuracy = 0.655556 at epoch: 18

Best Hyperparameters:
{'hidden_dim': 14, 'learning_rate': 0.0001}
Best Overall Validation Accuracy: 0.8

Test accuracy with the best hyperparameters: 0.8222222222222222
```

4.3 Conclusion for task 6

First, through the experiments in Task 6, we demonstrated the feasibility of building a neural network model using a custom `Tensor` with automatic differentiation framework. Then, after systematically searching over various hyperparameter combinations—including hidden layer size and learning rate—we found that the best configuration is a hidden layer size of `14` with a learning rate of `0.0001`, achieving a peak validation accuracy of `80.00%`, and the corresponding test accuracy is `82.22%`