Assignment_4

February 28, 2025

1 Assignment 4: Software for neural network training

By: Tim Bakkenes, Elvira Moberg and Agnes Magnusson

[22]:
```
import pandas as pd
import numpy as np
import torch
```

2 Task 1

Loading the synthetic dataset.

[23]:
```
# 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.

[24]:
```
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, dtype=torch.float, requires_grad=True)
b = torch.tensor(b_init, dtype=torch.float, requires_grad=True)

eta = 1e-2
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], :], dtype=torch.float)  # added dtype=torch.
↪float to ensure same format
        y = torch.tensor(Y[[row]], dtype=torch.float)     # added dtype=torch.
↪float to ensure same format

        # Forward pass.
        y_pred = x @ w + b
        err = (y - y_pred)**2

        # Backward and update.
        opt.zero_grad()
        err.backward()
        opt.step()

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

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

```
Epoch 1: MSE = 0.7999662647869263
Epoch 2: MSE = 0.017392394159767264
Epoch 3: MSE = 0.009377418162580966
Epoch 4: MSE = 0.009355327616258364
Epoch 5: MSE = 0.009365440349979508
Epoch 6: MSE = 0.009366988411857164
Epoch 7: MSE = 0.009367207068114567
Epoch 8: MSE = 0.009367238481529512
Epoch 9: MSE = 0.009367244712136654
Epoch 10: MSE = 0.009367244620257224
```

3 Task 2

[25]:
```
class Tensor:

    # Constructor. Just store the input values.
    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

    # 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:
```

```python
            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')
        else:
            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:
```

```python
                # 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.

                #initialize grad output to 1s
                grad_output = np.ones_like(self.data)

                self.grad_fn.backward(grad_output)
            else:
                # This is an intermediate node in the computational graph.

                # This corresponds to any intermediate computation, such as
                # a hidden layer.

                #call the backward function recursively
                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.

                #save grad_output in self.grad
                self.grad = grad_output

            else:
                # This tensor *does not require* a gradient to be computed. This
                # will typically be a tensor holding input data.

                #terminate the recursion
                return


# 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):
```

```python
    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 = SubtractionNode(left, right)
    return Tensor(new_data, grad_fn=grad_fn)

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

def power(left, right):
    new_data = left.data**right
    grad_fn = PowerNode(left, right)
    return Tensor(new_data, grad_fn=grad_fn)

def tanh(x):
    new_data = np.tanh(x.data)
    grad_fn = TanhNode(x)
    return Tensor(new_data, grad_fn=grad_fn)

def sigmoid(x):
    # Ensure x.data is a numpy array
    data = np.asarray(x.data)
    new_data = 1 / (1 + np.exp(-data))
    grad_fn = SigmoidNode(x)
    return Tensor(new_data, grad_fn=grad_fn)

def BCELoss(y_pred, y_true):
    #using sigmoid to convert y_pred from logits to probabilities
    y_prob = sigmoid(y_pred)
    new_data = -y_true.data*np.log(y_prob.data) - (1-y_true.data)*np.
 ↪log(1-y_prob.data)
    grad_fn = BCELossNode(y_prob, y_true)
    return Tensor(new_data, grad_fn=grad_fn)
```

Some sanity checks.

[27]:
```python
# 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]]
```

Create some tensors and make sure that you can compute the arithmetic operations that you used in the linear regression examples above.

4 Tasks 3 and 4

For each node, the derivative of the correspronding function is calculated and passed backwards, enabling backpropagation.

[28]:
```
class Node:
    def __init__(self):
        pass

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

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


class AdditionNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right
```

```
    def backward(self, grad_output):
        self.left.backward(grad_output)
        self.right.backward(grad_output)

class SubtractionNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def backward(self, grad_output):
        self.left.backward(grad_output)
        self.right.backward(-grad_output)

class MatrixMultiplicationNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def backward(self, grad_output):
        self.left.backward(grad_output @ self.right.data.T)
        self.right.backward(self.left.data.T @ grad_output)

class PowerNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def backward(self, grad_output):
        self.left.backward(grad_output*self.right*(self.left.data**(self.
↪right-1)))

class TanhNode(Node):
    def __init__(self, x):
        self.x = x

    def backward(self, grad_output):
        self.x.backward(grad_output*(1-(np.tanh(self.x.data)**2)))

class BCELossNode(Node):
    def __init__(self, y_prob, y_true):
        self.y_prob = y_prob
        self.y_true = y_true

    def backward(self, grad_output):
        derivative = (self.y_prob.data - self.y_true.data)/(self.y_prob.
↪data*(1-self.y_prob.data))
```

```
        self.y_prob.backward(grad_output*derivative)

class SigmoidNode(Node):
    def __init__(self, x):
        self.x = x

    def backward(self, grad_output):
        self.x.backward(grad_output*(self.x.data*(1-self.x.data)))
```

Sanity check for Task 3.

[29]:
```
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.

[30]:
```
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`.

[31]:
```python
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
```

[31]:
```
tensor([[5.6000],
        [8.4000]], dtype=torch.float64)
```

5 Task 5

[32]:
```python
class Optimizer:
    def __init__(self, params):
        self.params = params

    def zero_grad(self):
        for param in self.params:
            param.grad = np.zeros_like(param.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 param in self.params:
            if param.grad is not None:
                if param.grad.shape[0] != param.data.shape[0]:
```

```
                    #sum gradients for bias, then apply gradient descent
                    grad_b = np.sum(param.grad, axis=0, keepdims=True)
                    param.data -= self.lr * grad_b
                else:
                    #gradient descent on each weight
                    param.data -= self.lr * param.grad
```

Testing if the same results are achieved with the code form task 1

[33]:
```
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 = 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]])

        # Forward pass.
        y_pred = x @ w + b
        err = (y - y_pred)**2
        #err = err.mean()

        # Backward and update.
        opt.zero_grad()
        err.backward()
        opt.step()

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

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

```
Epoch 1: MSE = 0.7999661130823178
Epoch 2: MSE = 0.017392390107906875
```

```
Epoch 3: MSE = 0.009377418010839892
Epoch 4: MSE = 0.009355326971438456
Epoch 5: MSE = 0.009365440968904256
Epoch 6: MSE = 0.009366989180952533
Epoch 7: MSE = 0.009367207398577986
Epoch 8: MSE = 0.009367238983974489
Epoch 9: MSE = 0.009367243704122532
Epoch 10: MSE = 0.009367244427185763
```

6 Task 6

[34]:
```
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split

# You may need to edit the path, depending on where you put the files.
a4data = pd.read_csv('data/raisins.csv')

X = scale(a4data.drop(columns='Class'))
Y = 1.0*(a4data.Class == 'Besni').to_numpy()

np.random.seed(0)
shuffle = np.random.permutation(len(Y))
X = X[shuffle]
Y = Y[shuffle]

Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, random_state=0,␣
↪test_size=0.2)
```

Initialize the model structure with one hidden layer using tanh as activation function

[35]:
```
class RaisinCLF:
    def __init__(self, input_dim, hidden_dim, output_dim):
        #initialize weights with small random values, and biases with zero.␣
↪Requires_grad=true to enable backpropagation
        self.w1 = tensor(np.random.randn(input_dim, hidden_dim) * 0.01,␣
↪requires_grad=True)
        self.b1 = tensor(np.zeros((1, hidden_dim)), requires_grad=True)
        self.w2 = tensor(np.random.randn(hidden_dim, output_dim) * 0.01,␣
↪requires_grad=True)
        self.b2 = tensor(np.zeros((1, output_dim)), requires_grad=True)

    def forward(self, x):
        """forward pass through network"""

        #computing hidden layer with linear transformation and tanh activation
        hidden = tanh(addition(matrix_multiplication(x, self.w1), self.b1))
```

```
        #computing output layer with linear transformation
        output = addition(matrix_multiplication(hidden, self.w2), self.b2)
        return output

    def get_params(self):
        return [self.w1, self.b1, self.w2, self.b2]
```

[36]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

def train_classifier(model, X, Y):
    # hyperparameters
    lr = 1e-6
    n_epochs = 100
    val_size = 0.2

    Xtrain, Xval, Ytrain, Yval = train_test_split(X, Y, test_size=val_size,␣
 ↪random_state=0)

    optimizer = SGD(model.get_params(), lr=lr)

    #stores training and validation accuracy
    history = []

    #convert data to tensor
    Xtrain_t = tensor(Xtrain)
    Ytrain_t = tensor(Ytrain)

    max_acc = 0

    for epoch in range(n_epochs):

        loss_sum = 0

        outputs = model.forward(Xtrain_t)

        loss = BCELoss(outputs, Ytrain_t)

        #initilize gradient to 0
        optimizer.zero_grad()

        #computing gradient
        loss.backward()

        #apply gradient descent
        optimizer.step()
```

```
        #compute average loss
        loss_sum += loss.data.mean()

        train_pred = predict(model, Xtrain).flatten()
        val_pred = predict(model, Xval).flatten()

        train_acc = accuracy_score(Ytrain.flatten(), train_pred)
        val_acc = accuracy_score(Yval.flatten(), val_pred)

        max_acc = max(max_acc, val_acc)

        history.append((train_acc, val_acc))

        if (epoch+1) % 5 == 0:
            print(f'Epoch {epoch+1}: loss = {loss_sum:.4f}, train acc =␣
 ↪{train_acc:.4f}, val acc = {val_acc:.4f}')

    print(f'Max validation accuracy achieved: {max_acc:.4f}')
    return history


def predict(model, x):
    Xt = tensor(x)
    scores = sigmoid(model.forward(Xt).data)
    y_guess = (scores.data > 0.5).astype(int)
    return y_guess
```

[37]:
```
input_dim = 7
hidden_dim = 128
output_dim = 1

model = RaisinCLF(input_dim=input_dim, hidden_dim=hidden_dim,␣
 ↪output_dim=output_dim)

history = train_classifier(model, Xtrain, Ytrain.reshape(-1, 1))

y_guess = predict(model, Xtest)

accuracy = accuracy_score(Ytest, y_guess)
print(f'Test accuracy: {accuracy:.4f}')
```

```
Epoch 5: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
Epoch 10: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
Epoch 15: loss = 0.6921, train acc = 0.8576, val acc = 0.8125
Epoch 20: loss = 0.6921, train acc = 0.8576, val acc = 0.8194
Epoch 25: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
Epoch 30: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
Epoch 35: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
```

```
Epoch 40: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
Epoch 45: loss = 0.6921, train acc = 0.8576, val acc = 0.8194
Epoch 50: loss = 0.6921, train acc = 0.8576, val acc = 0.8194
Epoch 55: loss = 0.6921, train acc = 0.8576, val acc = 0.8194
Epoch 60: loss = 0.6921, train acc = 0.8576, val acc = 0.8125
Epoch 65: loss = 0.6921, train acc = 0.8576, val acc = 0.8125
Epoch 70: loss = 0.6921, train acc = 0.8576, val acc = 0.8125
Epoch 75: loss = 0.6921, train acc = 0.8559, val acc = 0.8125
Epoch 80: loss = 0.6921, train acc = 0.8576, val acc = 0.8125
Epoch 85: loss = 0.6921, train acc = 0.8576, val acc = 0.8194
Epoch 90: loss = 0.6921, train acc = 0.8594, val acc = 0.8194
Epoch 95: loss = 0.6921, train acc = 0.8611, val acc = 0.8194
Epoch 100: loss = 0.6921, train acc = 0.8611, val acc = 0.8264
Max validation accuracy achieved: 0.8264
Test accuracy: 0.8333
Epoch 100: loss = 0.6921, train acc = 0.8611, val acc = 0.8264
Max validation accuracy achieved: 0.8264
Test accuracy: 0.8333
```