Machine Learning Homework 07

Author: Jesus Arturo Sol Navarro

1 Problem 3

$$a + \log\left(\sum_{i=1}^{N} e^{x_i - a}\right) = a + \log\left(e^{-a}\sum_{i=1}^{N} e^{x_i}\right) = a - a + \log\left(\sum_{i=1}^{N} e^{x_i}\right) = \log\left(\sum_{i=1}^{N} e^{x_i}\right)$$

2 Problem 4

$$\frac{e^{x_i - a}}{\sum_{i=1}^{N} e^{x_i - a}} = \frac{e^{x_i} e^{-a}}{e^{-a}\sum_{i=1}^{N} e^{x_i}} = \frac{e^{x_i}}{\sum_{i=1}^{N} e^{x_i}}$$

3 Problem 5

3.1 Affine layer

```python
class Affine:
def forward(self, inputs, weight, bias):
    """Forward pass of an affine (fully connected) layer.

    Args:
        inputs: input matrix, shape (N, D)
        weight: weight matrix, shape (D, H)
        bias: bias vector, shape (H)

    Returns
        out: output matrix, shape (N, H)
    """
    self.cache = (inputs, weight, bias)
    #############################################################
    # TODO
    # Your code here
    # Backward computation using chain rule:
    # d_inputs = d_out * W^T
    # (N,H) @ (H,D) -> (N,D)
    out = np.dot(inputs, weight) + bias
    #############################################################
    assert out.shape[0] == inputs.shape[0]
    assert out.shape[1] == weight.shape[1] == bias.shape[0]
    return out

def backward(self, d_out):
    """Backward pass of an affine (fully connected) layer.

    Args:
        d_out: incoming derivaties, shape (N, H)

    Returns:
        d_inputs: gradient w.r.t. the inputs, shape (N, D)
        d_weight: gradient w.r.t. the weight, shape (D, H)
        d_bias: gradient w.r.t. the bias, shape (H)
    """
    inputs, weight, bias = self.cache
    #############################################################
    # TODO
    # Your code here
    # Backward computation using chain rule:
    # d_inputs = d_out * W^T
    # (N,H) @ (H,D) -> (N,D)
    d_inputs = np.dot(d_out, weight.T)

    # d_weight = X^T * d_out
    # (D,N) @ (N,H) -> (D,H)
    d_weight = np.dot(inputs.T, d_out)

    # d_bias = sum of d_out across batch dimension
```

```
        # sum((N,H), axis=0) -> (H,)
        d_bias = np.sum(d_out, axis=0)
        ############################################################
        assert np.all(d_inputs.shape == inputs.shape)
        assert np.all(d_weight.shape == weight.shape)
        assert np.all(d_bias.shape == bias.shape)
        return d_inputs, d_weight, d_bias
```

3.2 ReLU layer

```
    class ReLU:
    def forward(self, inputs):
        """Forward pass of a ReLU layer.

        Args:
            inputs: input matrix, arbitrary shape

        Returns:
            out: output matrix, has same shape as inputs
        """
        self.cache = inputs
        ############################################################
        # TODO
        # Your code here
        # ReLU forward: max(0, x)
        out = np.maximum(0, inputs)

        ############################################################
        assert np.all(out.shape == inputs.shape)
        return out

    def backward(self, d_out):
        """Backward pass of an ReLU layer.

        Args:
            d_out: incoming derivatives, same shape as inputs in forward

        Returns:
            d_inputs: gradient w.r.t. the inputs, same shape as d_out
        """
        inputs = self.cache
        ############################################################
        # TODO
        # Your code here
        # ReLU backward: derivative is 1 where input > 0, 0 elsewhere
        d_inputs = d_out * (inputs > 0)

        ############################################################
        assert np.all(d_inputs.shape == inputs.shape)
        return d_inputs
```

3.3 CategoricalCrossEntropy layer

```
    class CategoricalCrossEntropy:
    def forward(self, logits, labels):
        """Compute categorical cross-entropy loss.

        Args:
            logits: class logits, shape (N, K)
            labels: target labels in one-hot format, shape (N, K)

        Returns:
            loss: loss value, float (a single number)
        """
        ############################################################
        # TODO
        # Your code here
        # Compute softmax probabilities
        # Subtract max for numerical stability
        shifted_logits = logits - np.max(logits, axis=1, keepdims=True)
```

```
        exp_logits = np.exp(shifted_logits)
        probs = exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

        # Compute cross entropy loss
        # Add small epsilon to avoid log(0)
        eps = 1e-15
        probs = np.clip(probs, eps, 1 - eps)
        loss = -np.sum(labels * np.log(probs)) / logits.shape[0]

        #############################################################
        # probs is the (N, K) matrix of class probabilities
        self.cache = (probs, labels)
        assert isinstance(loss, float)
        return loss

    def backward(self, d_out=1.0):
        """Backward pass of the Cross Entropy loss.

        Args:
            d_out: Incoming derivatives. We set this value to 1.0 by default,
                since this is the terminal node of our computational graph
                (i.e. we usually want to compute gradients of loss w.r.t.
                other model parameters).

        Returns:
            d_logits: gradient w.r.t. the logits, shape (N, K)
            d_labels: gradient w.r.t. the labels
                we don't need d_labels for our models, so we don't
                compute it and set it to None. It's only included in the
                function definition for consistency with other layers.
        """
        probs, labels = self.cache
        #############################################################
        # TODO
        # Your code here
        # Gradient of cross entropy with respect to logits
        # When combined with softmax, this simplifies to (probs - labels)
        d_logits = (probs - labels) * d_out / labels.shape[0]

        #############################################################
        d_labels = None
        assert np.all(d_logits.shape == probs.shape == labels.shape)
        return d_logits, d_labels
```

3.4 Logistic regression (with backpropagation)

```
    class LogisticRegression:
    def __init__(self, num_features, num_classes, learning_rate=1e-2):
        """Logistic regression model.
        Gradients are computed with backpropagation.


        The model consists of the following sequence of opeartions:

        input -> affine -> softmax
        """
        self.learning_rate = learning_rate

        # Initialize the model parameters
        self.params = {
            "W": np.zeros([num_features, num_classes]),
            "b": np.zeros([num_classes]),
        }

        # Define layers
        self.affine = Affine()
        self.cross_entropy = CategoricalCrossEntropy()

    def predict(self, X):
        """Generate predictions for one minibatch.

        Args:
            X: data matrix, shape (N, D)
```

```
        Returns:
            Y_pred: predicted class probabilities, shape (N, D)
            Y_pred[n, k] = probability that sample n belongs to class k
        """
        logits = self.affine.forward(X, self.params["W"], self.params["b"])
        Y_pred = softmax(logits, axis=1)
        return Y_pred

    def step(self, X, Y):
        """Perform one step of gradient descent on the minibatch of data.

        1. Compute the cross-entropy loss for given (X, Y).
        2. Compute the gradients of the loss w.r.t. model parameters.
        3. Update the model parameters using the gradients.

        Args:
            X: data matrix, shape (N, D)
            Y: target labels in one-hot format, shape (N, K)

        Returns:
            loss: loss for (X, Y), float, (a single number)
        """
        # Forward pass - compute the loss on training data
        logits = self.affine.forward(X, self.params["W"], self.params["b"])
        loss = self.cross_entropy.forward(logits, Y)

        # Backward pass - compute the gradients of loss w.r.t. all the model parameters
        grads = {}
        d_logits, _ = self.cross_entropy.backward()
        _, grads["W"], grads["b"] = self.affine.backward(d_logits)

        # Apply the gradients
        for p in self.params:
            self.params[p] = self.params[p] - self.learning_rate * grads[p]
        return loss
```

```
# Specify optimization parameters
learning_rate = 1e-2
max_epochs = 501
report_frequency = 50
log_reg = LogisticRegression(num_features=D, num_classes=K)
for epoch in range(max_epochs):
    loss = log_reg.step(X_train, Y_train)
    if epoch % report_frequency == 0:
        print(f"Epoch {epoch:4d}, loss = {loss:.4f}")
```

Epoch 0, loss = 2.3026
Epoch 50, loss = 0.2275
Epoch 100, loss = 0.1599
Epoch 150, loss = 0.1306
Epoch 200, loss = 0.1130
Epoch 250, loss = 0.1009
Epoch 300, loss = 0.0918
Epoch 350, loss = 0.0846
Epoch 400, loss = 0.0788
Epoch 450, loss = 0.0738
Epoch 500, loss = 0.0696
test set accuracy = 0.953

3.5 Feed-forward neural network

```
    def xavier_init(shape):
    """Initialize a weight matrix according to Xavier initialization.

    See pytorch.org/docs/stable/nn.init#torch.nn.init.xavier_uniform_ for details.
    """
    a = np.sqrt(6.0 / float(np.sum(shape)))
    return np.random.uniform(low=-a, high=a, size=shape)
```

3.6 Implement a two-layer ‘FeedForwardNeuralNet‘ model

```python
class FeedforwardNeuralNet:
def __init__(self, input_size, hidden_size, output_size, learning_rate=1e-2):
    """A two-layer feedforward neural network with ReLU activations.

    (input_layer -> hidden_layer -> output_layer)


    The model consists of the following sequence of opeartions:

    input -> affine -> relu -> affine -> softmax

    """
    self.learning_rate = learning_rate

    # Initialize the model parameters
    self.params = {
        "W1": xavier_init([input_size, hidden_size]),
        "b1": np.zeros([hidden_size]),
        "W2": xavier_init([hidden_size, output_size]),
        "b2": np.zeros([output_size]),
    }

    # Define layers
    ############################################################
    # TODO
    # Your code here
    self.affine1 = Affine()
    self.relu = ReLU()
    self.affine2 = Affine()
    self.loss = CategoricalCrossEntropy()
    ############################################################

def predict(self, X):
    """Generate predictions for one minibatch.

    Args:
        X: data matrix, shape (N, D)

    Returns:
        Y_pred: predicted class probabilities, shape (N, D)
        Y_pred[n, k] = probability that sample n belongs to class k
    """
    ############################################################
    # TODO
    # Your code here
    # Forward pass through the network (without loss computation)
    h1 = self.affine1.forward(X, self.params["W1"], self.params["b1"])
    h1_relu = self.relu.forward(h1)
    logits = self.affine2.forward(h1_relu, self.params["W2"], self.params["b2"])

    # Convert logits to probabilities using softmax
    exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    Y_pred = exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

    ############################################################
    return Y_pred

def step(self, X, Y):
    """Perform one step of gradient descent on the minibatch of data.

    1. Compute the cross-entropy loss for given (X, Y).
    2. Compute the gradients of the loss w.r.t. model parameters.
    3. Update the model parameters using the gradients.

    Args:
        X: data matrix, shape (N, D)
        Y: target labels in one-hot format, shape (N, K)

    Returns:
        loss: loss for (X, Y), float, (a single number)
    """
    ############################################################
    # TODO
```

```python
        # Your code here
        # Forward pass
        h1 = self.affine1.forward(X, self.params["W1"], self.params["b1"])
        h1_relu = self.relu.forward(h1)
        logits = self.affine2.forward(h1_relu, self.params["W2"], self.params["b2"])
        loss = self.loss.forward(logits, Y)

        # Backward pass
        d_logits, _ = self.loss.backward()
        d_h1_relu, d_W2, d_b2 = self.affine2.backward(d_logits)
        d_h1 = self.relu.backward(d_h1_relu)
        d_X, d_W1, d_b1 = self.affine1.backward(d_h1)

        # Update parameters using gradient descent
        self.params["W1"] -= self.learning_rate * d_W1
        self.params["b1"] -= self.learning_rate * d_b1
        self.params["W2"] -= self.learning_rate * d_W2
        self.params["b2"] -= self.learning_rate * d_b2

        ############################################################
        return loss
```

```python
H = 32  # size of the hidden layer

# Specify optimization parameters
learning_rate = 1e-2
max_epochs = 501
report_frequency = 50
model = FeedforwardNeuralNet(
    input_size=D, hidden_size=H, output_size=K, learning_rate=learning_rate
)
model = FeedforwardNeuralNet(
    input_size=D, hidden_size=H, output_size=K, learning_rate=learning_rate
)
```

Epoch 0, loss = 13.3519
Epoch 50, loss = 0.6245
Epoch 100, loss = 0.3687
Epoch 150, loss = 0.2667
Epoch 200, loss = 0.2092
Epoch 250, loss = 0.1729
Epoch 300, loss = 0.1473
Epoch 350, loss = 0.1273
Epoch 400, loss = 0.1120
Epoch 450, loss = 0.0997
Epoch 500, loss = 0.0894
test set accuracy = 0.936