

# exercise\_07\_notebook

December 6, 2021

```
[1]: import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.metrics import accuracy_score

from scipy.special import softmax

[2]: X, y = load_digits(return_X_y=True)
# Convert a categorical vector y (shape [N]) into a one-hot encoded matrix
# → (shape [N, K])
Y = label_binarize(y, classes=np.unique(y)).astype(np.float64)

np.random.seed(123)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.25)

[3]: N, K = Y.shape # N - num_samples, K - num_classes
D = X.shape[1] # num_features
```

Remember from the tutorial: 1. No for loops! Use matrix multiplication and broadcasting whenever possible. 2. Think about numerical stability

```
[4]: import nn_utils # module containing helper functions for checking the
# → correctness of your code
```

## 0.1 Task 1: Affine layer

Implement forward and backward functions for Affine layer

```
[5]: 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)
    #####
    out = inputs.dot(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 derivatives, 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
    #####
    d_inputs = d_out.dot(weight.T)
    d_weight = inputs.T.dot(d_out)
    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

```

```

[6]: affine = Affine()
     nn_utils.check_affine(affine)

```

All checks passed succesfully!

## 0.2 Task 2: ReLU layer

Implement forward and backward functions for ReLU layer

```
[7]: 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
        #####
        b = inputs > 0
        out = b.astype(inputs.dtype) * 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
        #####
        b = inputs > 0
        d_inputs = b.astype(inputs.dtype) * d_out
        #####
        assert np.all(d_inputs.shape == inputs.shape)
        return d_inputs
```

```
[8]: relu = ReLU()
nn_utils.check_relu(relu)
```

All checks passed successfully!

### 0.3 Task 3: CategoricalCrossEntropy layer

Implement forward and backward for CategoricalCrossEntropy layer

```
[9]: 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
    N, _ = logits.shape
    #softmax
    logits_shifted = logits - np.max(logits, axis=1, keepdims=True)
    denominator = np.sum(np.exp(logits_shifted), axis=1, keepdims=True)
    probs = np.exp(logits_shifted) / denominator
    #applying the CE to the result of softmax
    ce_terms = labels * (logits_shifted - np.log(denominator))
    ce_loss_per_instance = - np.sum(ce_terms, axis = 1)
    loss = 1 / N * np.sum(ce_loss_per_instance)
    #####
    # 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
    #####
    N, _ = probs.shape
    d_logits = 1/N * (probs - labels)*d_out

    #####
    d_labels = None

```

```

    assert np.all(d_logits.shape == probs.shape == labels.shape)
    return d_logits, d_labels

```

```

[10]: cross_entropy = CategoricalCrossEntropy()
      nn_utils.check_cross_entropy(cross_entropy)

```

All checks passed succesfully!

## 1 Logistic regression (with backpropagation) — nothing to do in this section

```

[11]: 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 operations:

          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

```

```

[12]: # Specify optimization parameters
learning_rate = 1e-2
max_epochs = 501
report_frequency = 50

```

```

[13]: log_reg = LogisticRegression(num_features=D, num_classes=K)

```

```

[14]: 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
```

```
[15]: y_test_pred = log_reg.predict(X_test).argmax(1)
      y_test_true = Y_test.argmax(1)
```

```
[16]: print(f'test set accuracy = {accuracy_score(y_test_true, y_test_pred):.3f}')
```

```
test set accuracy = 0.953
```

## 2 Feed-forward neural network (with backpropagation)

```
[17]: 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)
```

### 2.1 Task 4: Implement a two-layer FeedForwardNeuralNet model

You can use the LogisticRegression class for reference

```
[18]: 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 operations:

          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
    #####
    self.affine1 = Affine()
    self.relu = ReLU()
    self.affine2 = 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
    """
    #####
    hidden1 = self.affine1.forward(X, self.params['W1'], self.params['b1'])
    h1_act = self.relu.forward(hidden1)
    hidden2 = self.affine2.forward(h1_act, self.params['W2'], self.
→params['b2'])
    Y_pred = softmax(hidden2, 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)
    """
    #####

```



```

        #estimating prediction and corresponding loss
        hidden1 = self.affine1.forward(X, self.params['W1'], self.params['b1'])
        h1_act = self.relu.forward(hidden1)
        hidden2 = self.affine2.forward(h1_act, self.params['W2'], self.
→params['b2'])
        loss = self.cross_entropy.forward(hidden2, Y)

        grads = {}
        d_hidden2, _ = self.cross_entropy.backward()
        d_inputs, grads['W2'], grads['b2'] = self.affine2.backward(d_hidden2)
        d_relu = self.relu.backward(d_inputs)
        _, grads['W1'], grads['b1'] = self.affine1.backward(d_relu)

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

```

[19]: `H = 32 # size of the hidden layer`

```

# Specify optimization parameters
learning_rate = 1e-2
max_epochs = 501
report_frequency = 50

```

[20]: `model = FeedforwardNeuralNet(input_size=D, hidden_size=H, output_size=K, ↵  
→learning_rate=learning_rate)`

[21]: `for epoch in range(max_epochs):  
 loss = model.step(X_train, Y_train)  
 if epoch % report_frequency == 0:  
 print(f'Epoch {epoch:4d}, loss = {loss:.4f}')`

```

Epoch    0, loss = 8.5876
Epoch   50, loss = 0.6002
Epoch  100, loss = 0.3517
Epoch  150, loss = 0.2510
Epoch  200, loss = 0.1975
Epoch  250, loss = 0.1631
Epoch  300, loss = 0.1401
Epoch  350, loss = 0.1231
Epoch  400, loss = 0.1098
Epoch  450, loss = 0.0989
Epoch  500, loss = 0.0897

```

```
[22]: y_test_pred = model.predict(X_test).argmax(1)  
      y_test_true = Y_test.argmax(1)
```

```
[23]: print(f'test set accuracy = {accuracy_score(y_test_true, y_test_pred):.3f}')
```

```
test set accuracy = 0.938
```