Q2

March 19, 2021

1 2.1

[ ]:
```
from utils import load_data, load, save, display_plot
import numpy as np
from matplotlib import pyplot as plt
```

[ ]:
```
def init_nn(num_inputs, num_hiddens, num_outputs):
    """ Initializes neural network's parameters.
    :param num_inputs: Number of input units
    :param num_hiddens: List of two elements, hidden size for each layers.
    :param num_outputs: Number of output units
    :return: A dictionary of randomly initialized neural network weights.
    """
    W1 = 0.1 * np.random.randn(num_inputs, num_hiddens[0])
    W2 = 0.1 * np.random.randn(num_hiddens[0], num_hiddens[1])
    W3 = 0.01 * np.random.randn(num_hiddens[1], num_outputs)
    b1 = np.zeros((num_hiddens[0]))
    b2 = np.zeros((num_hiddens[1]))
    b3 = np.zeros((num_outputs))
    model = {
        "W1": W1,
        "W2": W2,
        "W3": W3,
        "b1": b1,
        "b2": b2,
        "b3": b3
    }
    return model


def softmax(x):
    """ Computes the softmax activation function.
    :param x: Inputs
    :return: Activation of x
    """
    return np.exp(x) / np.exp(x).sum(axis=1, keepdims=True)
```

```python
def nn_forward(model, x):
    """ Runs a forward pass.
    :param model: Dictionary of all the weights.
    :param x: Input to the network.
    :return: Dictionary of all intermediate variables.
    """
    z1 = affine(x, model["W1"], model["b1"])
    h1 = relu(z1)
    z2 = affine(h1, model["W2"], model["b2"])
    h2 = relu(z2)
    y = affine(h2, model["W3"], model["b3"])
    var = {
        "x": x,
        "z1": z1,
        "h1": h1,
        "z2": z2,
        "h2": h2,
        "y": y
    }
    return var

def nn_backward(model, err, var):
    """ Runs the backward pass.
    :param model: Dictionary of all the weights.
    :param err: Gradients to the output of the network.
    :param var: Intermediate variables from the forward pass.
    :return: None
    """
    dE_dh2, dE_dW3, dE_db3 = affine_backward(err, var["h2"], model["W3"])
    dE_dz2 = relu_backward(dE_dh2, var["z2"])
    dE_dh1, dE_dW2, dE_db2 = affine_backward(dE_dz2, var["h1"], model["W2"])
    dE_dz1 = relu_backward(dE_dh1, var["z1"])
    _, dE_dW1, dE_db1 = affine_backward(dE_dz1, var["x"], model["W1"])
    model["dE_dW1"] = dE_dW1
    model["dE_dW2"] = dE_dW2
    model["dE_dW3"] = dE_dW3
    model["dE_db1"] = dE_db1
    model["dE_db2"] = dE_db2
    model["dE_db3"] = dE_db3
    return
```

[ ]:
```python
def affine(x, w, b):
    """ Computes the affine transformation.
    :param x: Inputs (or hidden layers)
    :param w: Weight
    :param b: Bias
```

```
    :return: Outputs
    """
    y = x.dot(w) + b
    return y


def affine_backward(grad_y, x, w):
    """ Computes gradients of affine transformation.
    Hint: you may need the matrix transpose np.dot(A, B).T = np.dot(B, A) and␣
 ↪(A.T).T = A
    :param grad_y: Gradient from upper layer
    :param x: Inputs from the hidden layer
    :param w: Weights
    :return: A tuple of (grad_h, grad_w, grad_b)
        WHERE
        grad_x: Gradients wrt. the inputs/hidden layer.
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    ######################################################################
    # TODO:                                                              #
    # Complete the function to compute the gradients of affine           #
    # transformation.                                                    #
    ######################################################################
    grad_x = grad_y @ w.T
    grad_w = x.T @ grad_y
    grad_b = np.sum(grad_y, axis = 0)
    ######################################################################
    #                         END OF YOUR CODE                           #
    ######################################################################
    return grad_x, grad_w, grad_b
```

```
[ ]: def relu(x):
    """ Computes the ReLU activation function.
    :param z: Inputs
    :return: Activation of x
    """
    return np.maximum(x, 0.0)

def relu_backward(grad_y, x):
    """ Computes gradients of the ReLU activation function wrt. the unactivated␣
 ↪inputs.
    :param grad_y: Gradient of the activation.
    :param x: Inputs
    :return: Gradient wrt. x
    """
    ######################################################################
```

```
    # TODO:                                                               #
    # Complete the function to compute the gradients of relu.             #
    #######################################################################
    ydx = np.float32(relu(x)!=0)
    grad_x = grad_y * ydx
    #######################################################################
    #                          END OF YOUR CODE                           #
    #######################################################################
    return grad_x
```

[ ]:
```
def nn_update(model, eta):
    """ Update NN weights.
    :param model: Dictionary of all the weights.
    :param eta: Learning rate
    :return: None
    """
    #######################################################################
    # TODO:                                                               #
    # Complete the function to update the neural network's parameters.   #
    # Your code should look as follows                                    #
    # model["W1"] = ...                                                   #
    # model["W2"] = ...                                                   #
    # ...                                                                 #
    #######################################################################
    model["W1"] = model["W1"] - eta * model["dE_dW1"]
    model["W2"] = model["W2"] - eta * model["dE_dW2"]
    model["W3"] = model["W3"] - eta * model["dE_dW3"]
    model["b1"] = model["b1"] - eta * model["dE_db1"]
    model["b2"] = model["b2"] - eta * model["dE_db2"]
    model["b3"] = model["b3"] - eta * model["dE_db3"]
    #######################################################################
    #                          END OF YOUR CODE                           #
    #######################################################################
    return
```

[ ]:
```
def train(model, forward, backward, update, eta, num_epochs, batch_size):
    """ Trains a simple MLP.
    :param model: Dictionary of model weights.
    :param forward: Forward prop function.
    :param backward: Backward prop function.
    :param update: Update weights function.
    :param eta: Learning rate.
    :param num_epochs: Number of epochs to run training for.
    :param batch_size: Mini-batch size, -1 for full batch.
    :return: A tuple (train_ce, valid_ce, train_acc, valid_acc)
        WHERE
        train_ce: Training cross entropy.
```

```
        valid_ce: Validation cross entropy.
        train_acc: Training accuracy.
        valid_acc: Validation accuracy.
    """
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
        target_test = load_data("toronto_face.npz")
    rnd_idx = np.arange(inputs_train.shape[0])

    train_ce_list = []
    valid_ce_list = []
    train_acc_list = []
    valid_acc_list = []
    num_train_cases = inputs_train.shape[0]
    if batch_size == -1:
        batch_size = num_train_cases
    num_steps = int(np.ceil(num_train_cases / batch_size))
    for epoch in range(num_epochs):
        np.random.shuffle(rnd_idx)
        inputs_train = inputs_train[rnd_idx]
        target_train = target_train[rnd_idx]
        for step in range(num_steps):
            # Forward pass.
            start = step * batch_size
            end = min(num_train_cases, (step + 1) * batch_size)
            x = inputs_train[start: end]
            t = target_train[start: end]

            var = forward(model, x)
            prediction = softmax(var["y"])

            train_ce = -np.sum(t * np.log(prediction)) / float(x.shape[0])
            train_acc = (np.argmax(prediction, axis=1) ==
                         np.argmax(t, axis=1)).astype("float").mean()
            print(("Epoch {:3d} Step {:2d} Train CE {:.5f} "
                   "Train Acc {:.5f}").format(
                epoch, step, train_ce, train_acc))

            # Compute error.
            error = (prediction - t) / float(x.shape[0])

            # Backward prop.
            backward(model, error, var)

            # Update weights.
            update(model, eta)

        valid_ce, valid_acc = evaluate(
```

```
            inputs_valid, target_valid, model, forward, batch_size=batch_size)
        print(("Epoch {:3d} "
               "Validation CE {:.5f} "
               "Validation Acc {:.5f}\n").format(
            epoch, valid_ce, valid_acc))
        train_ce_list.append((epoch, train_ce))
        train_acc_list.append((epoch, train_acc))
        valid_ce_list.append((epoch, valid_ce))
        valid_acc_list.append((epoch, valid_acc))
    display_plot(train_ce_list, valid_ce_list, "Cross Entropy", number=0)
    display_plot(train_acc_list, valid_acc_list, "Accuracy", number=1)

    train_ce, train_acc = evaluate(
        inputs_train, target_train, model, forward, batch_size=batch_size)
    valid_ce, valid_acc = evaluate(
        inputs_valid, target_valid, model, forward, batch_size=batch_size)
    test_ce, test_acc = evaluate(
        inputs_test, target_test, model, forward, batch_size=batch_size)
    print("CE: Train %.5f Validation %.5f Test %.5f" %
          (train_ce, valid_ce, test_ce))
    print("Acc: Train {:.5f} Validation {:.5f} Test {:.5f}".format(
        train_acc, valid_acc, test_acc))

    stats = {
        "train_ce": train_ce_list,
        "valid_ce": valid_ce_list,
        "train_acc": train_acc_list,
        "valid_acc": valid_acc_list
    }

    return model, stats


def evaluate(inputs, target, model, forward, batch_size=-1):
    """ Evaluates the model on inputs and target.
    :param inputs: Inputs to the network
    :param target: Target of the inputs
    :param model: Dictionary of network weights
    :param forward: Function for forward pass
    :param batch_size: Batch size
    :return: A tuple (ce, acc)
        WHERE
        ce: cross entropy
        acc: accuracy
    """
    num_cases = inputs.shape[0]
    if batch_size == -1:
```

```
        batch_size = num_cases
    num_steps = int(np.ceil(num_cases / batch_size))
    ce = 0.0
    acc = 0.0
    for step in range(num_steps):
        start = step * batch_size
        end = min(num_cases, (step + 1) * batch_size)
        x = inputs[start: end]
        t = target[start: end]
        prediction = softmax(forward(model, x)["y"])
        ce += -np.sum(t * np.log(prediction))
        acc += (np.argmax(prediction, axis=1) == np.argmax(
            t, axis=1)).astype("float").sum()
    ce /= num_cases
    acc /= num_cases
    return ce, acc


def check_grad(model, forward, backward, name, x):
    """ Check the gradients.
    """
    np.random.seed(0)
    var = forward(model, x)
    loss = lambda y: 0.5 * (y ** 2).sum()
    grad_y = var["y"]
    backward(model, grad_y, var)
    grad_w = model["dE_d" + name].ravel()
    w_ = model[name].ravel()
    eps = 1e-7
    grad_w_2 = np.zeros(w_.shape)
    check_elem = np.arange(w_.size)
    np.random.shuffle(check_elem)
    # Randomly check 20 elements.
    check_elem = check_elem[:20]
    for ii in check_elem:
        w_[ii] += eps
        err_plus = loss(forward(model, x)["y"])
        w_[ii] -= 2 * eps
        err_minus = loss(forward(model, x)["y"])
        w_[ii] += eps
        grad_w_2[ii] = (err_plus - err_minus) / 2. / eps
    np.testing.assert_almost_equal(grad_w[check_elem], grad_w_2[check_elem],
                                   decimal=3)
```

[ ]:

```
def main():
    """ Trains a neural network.
    :return: None
```

```
    """
    model_file_name = "nn_model.npz"
    stats_file_name = "nn_stats.npz"

    # Hyper-parameters. Modify them if needed.
    num_hiddens = [16,80]
    eta = 0.01
    num_epochs = 1000 # Number of iterations
    batch_size = 100

    # Input-output dimensions.
    num_inputs = 2304
    num_outputs = 7

    # Initialize model.
    model = init_nn(num_inputs, num_hiddens, num_outputs)

    # Uncomment to reload trained model here.
    # model = load(model_file_name)

    # Check gradient implementation.
    print("Checking gradients...")
    x = np.random.rand(10, 48 * 48) * 0.1
    check_grad(model, nn_forward, nn_backward, "W3", x)
    check_grad(model, nn_forward, nn_backward, "b3", x)
    check_grad(model, nn_forward, nn_backward, "W2", x)
    check_grad(model, nn_forward, nn_backward, "b2", x)
    check_grad(model, nn_forward, nn_backward, "W1", x)
    check_grad(model, nn_forward, nn_backward, "b1", x)

    # Train model.
    model, stats = train(model, nn_forward, nn_backward, nn_update, eta,
                         num_epochs, batch_size)

    # Uncomment if you wish to save the model.
    save(model_file_name, model)

    # Uncomment if you wish to save the training statistics.
    save(stats_file_name, stats)
```

```
[ ]: if __name__ == "__main__":
         main()
```

2 2.5

[ ]:
```
model_file_name = "nn_model.npz"
inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
    target_test = load_data("toronto_face.npz")
model = load(model_file_name)
var = nn_forward(model, inputs_test)
prediction = softmax(var["y"])

pred_conf = np.max(prediction, axis = 1)
uncertain = pred_conf <= sorted(pred_conf)[4]
uncertain_img = inputs_test[uncertain]
uncertain_class = target_test[uncertain]
uncertain_pred = np.argmax(prediction[uncertain], axis = 1)+1

for i in range(5):
    plt.figure()
    plt.imshow(np.reshape(uncertain_img[i],[48,48]), cmap = "gray")
    plt.axis("off")
    true_class = np.argmax(prediction[uncertain], axis = 1)[i]+1
    pred_class = uncertain_pred[i]
    plt.title(f"True Class: {true_class}, Predicted Class: {pred_class}")
    plt.show()
```