Assignment_5

March 12, 2023

1 Imports

[ ]:
```
import numpy as np
import os
from scipy.io import loadmat
```

2 Data

[ ]:
```
DATA_DIR = "data"
X_train = loadmat(os.path.join(DATA_DIR, "train_images.mat"))["train_images"]
X_test = loadmat(os.path.join(DATA_DIR, "test_images.mat"))["test_images"]
y_train = loadmat(os.path.join(DATA_DIR, "train_labels.mat"))["train_labels"]
y_test = loadmat(os.path.join(DATA_DIR, "test_labels.mat"))["test_labels"]
```

[ ]:
```
def one_hot(y, n_classes):
    """
    Converts a vector of labels into a one-hot matrix.

    Parameters
    ----------
    y : array_like
        An array of shape (m, ) that contains labels for X. Each value in y
        should be an integer in the range [0, n_classes).

    n_classes : int
        The number of classes.

    Returns
    -------
    one_hot : array_like
        An array of shape (m, n_classes) where each row is a one-hot vector.
    """
    if len(y.shape) > 1:
        raise ValueError("y should be a vector")
    m = y.shape[0]
    one_hot = np.zeros((n_classes, m))
```

```
    for i in range(m):
        one_hot[y[i], i] = 1
    return one_hot
```

2.1 Preprocessing

[ ]:
```
# Making the data compatible with the model
X_train = X_train.T
X_test = X_test.T
X_train = X_train.reshape(-1, 28,28,1)
X_test = X_test.reshape(-1, 28,28,1)
X_train.shape, X_test.shape
```

[ ]: ((1000, 28, 28, 1), (1000, 28, 28, 1))

[ ]:
```
# Normalizing the data
X_train = X_train / X_train.max() - 0.5
X_test = X_test / X_test.max() - 0.5
```

[ ]:
```
# One hot encoding the labels
y_train = np.squeeze(y_train)
y_test = np.squeeze(y_test)
y_train = one_hot(y_train, 10)
y_test = one_hot(y_test, 10)
y_train = y_train.T
y_test = y_test.T

y_train.shape, y_test.shape
```

[ ]: ((1000, 10), (1000, 10))

3 Problem 1

3.1 Implementations

[ ]:
```
def zero_pad(X, pad):
    """
    Pad with zeros all images of the dataset X. The padding is applied to the␣
↪height and width of an image,
    as illustrated in Figure 1.

    Argument:
    X -- python numpy array of shape (m, n_H, n_W, n_C) representing a batch of␣
↪m images
    pad -- integer, amount of padding around each image on vertical and␣
↪horizontal dimensions
```

```
    Returns:
    X_pad -- padded image of shape (m, n_H + 2 * pad, n_W + 2 * pad, n_C)
    """

    #(  1 line)
    # X_pad = None
    # YOUR CODE STARTS HERE

    X_pad = np.pad(X, ((0,0), (pad, pad), (pad, pad), (0,0)), mode="constant",␣
↪constant_values = (0,0))
    # YOUR CODE ENDS HERE

    return X_pad

# GRADED FUNCTION: conv_single_step

def conv_single_step(a_slice_prev, W, b):
    """
    Apply one filter defined by parameters W on a single slice (a_slice_prev)␣
↪of the output activation
    of the previous layer.

    Arguments:
    a_slice_prev -- slice of input data of shape (f, f, n_C_prev)
    W -- Weight parameters contained in a window - matrix of shape (f, f,␣
↪n_C_prev)
    b -- Bias parameters contained in a window - matrix of shape (1, 1, 1)

    Returns:
    Z -- a scalar value, the result of convolving the sliding window (W, b) on␣
↪a slice x of the input data
    """

    s = np.multiply(a_slice_prev,W)
    Z = np.sum(s)
    b = np.squeeze(b)
    Z = Z + b
    # YOUR CODE ENDS HERE

    return Z

# GRADED FUNCTION: conv_forward

def conv_forward(A_prev, W, b, hparameters):
    """
    Implements the forward propagation for a convolution function
```

```
    Arguments:
    A_prev -- output activations of the previous layer,
        numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    W -- Weights, numpy array of shape (f, f, n_C_prev, n_C)
    b -- Biases, numpy array of shape (1, 1, 1, n_C)
    hparameters -- python dictionary containing "stride" and "pad"

    Returns:
    Z -- conv output, numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache of values needed for the conv_backward() function
    """

    # YOUR CODE STARTS HERE
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = W.shape

    stride = hparameters["stride"]
    pad = hparameters["pad"]

    n_H = int((n_H_prev + 2*pad - f)/stride) + 1
    n_W = int((n_W_prev + 2*pad - f)/stride) + 1

    Z = np.zeros((m, n_H, n_W, n_C))

    A_prev_pad = zero_pad(A_prev, pad)

    for i in range(m):
        a_prev_pad = A_prev_pad[i]
        for h in range(n_H):
            vert_start = stride * h
            vert_end = vert_start  + f

            for w in range(n_W):
                horiz_start = stride * w
                horiz_end = horiz_start + f

                for c in range(n_C):

                    a_slice_prev = a_prev_pad[vert_start:vert_end,horiz_start:
↪horiz_end,:]

                    weights = W[:, :, :, c]
                    biases  = b[:, :, :, c]
                    Z[i, h, w, c] = conv_single_step(a_slice_prev, weights,␣
↪biases)
```

```
    cache = (A_prev, W, b, hparameters)

    # YOUR CODE ENDS HERE

    # Save information in "cache" for the backprop
    cache = (A_prev, W, b, hparameters)

    return Z, cache


def conv_backward(dZ, cache):
    """
    Implement the backward propagation for a convolution function

    Arguments:
    dZ -- gradient of the cost with respect to the output of the conv layer␣
 ↪(Z), numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache of values needed for the conv_backward(), output of␣
 ↪conv_forward()

    Returns:
    dA_prev -- gradient of the cost with respect to the input of the conv layer␣
 ↪(A_prev),
               numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    dW -- gradient of the cost with respect to the weights of the conv layer (W)
          numpy array of shape (f, f, n_C_prev, n_C)
    db -- gradient of the cost with respect to the biases of the conv layer (b)
          numpy array of shape (1, 1, 1, n_C)
    """


    # YOUR CODE STARTS HERE
    (A_prev, W, b, hparameters) = cache

    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    (f, f, n_C_prev, n_C) = W.shape

    stride = hparameters["stride"]
    pad = hparameters["pad"]

    (m, n_H, n_W, n_C) = dZ.shape

    dA_prev = np.zeros(A_prev.shape)
    dW = np.zeros(W.shape)
    db = np.zeros(b.shape) # b.shape = [1,1,1,n_C]
```

```
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)

    for i in range(m):                       # loop over the training examples

        # select ith training example from A_prev_pad and dA_prev_pad
        a_prev_pad = A_prev_pad[i]
        da_prev_pad = dA_prev_pad[i]

        for h in range(n_H):                   # loop over vertical axis of the␣
↪output volume
            for w in range(n_W):               # loop over horizontal axis of␣
↪the output volume
                for c in range(n_C):           # loop over the channels of the␣
↪output volume

                    # Find the corners of the current "slice"
                    vert_start = stride * h
                    vert_end = vert_start + f
                    horiz_start = stride * w
                    horiz_end = horiz_start + f

                    # Use the corners to define the slice from a_prev_pad
                    a_slice = a_prev_pad[vert_start:vert_end,horiz_start:
↪horiz_end,:]

                    # Update gradients for the window and the filter's␣
↪parameters using the code formulas given above
                    da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :]␣
↪+= W[:,:,:,c] * dZ[i, h, w, c]
                    dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
                    db[:,:,:,c] += dZ[i, h, w, c]

        # Set the ith training example's dA_prev to the unpadded da_prev_pad␣
↪(Hint: use X[pad:-pad, pad:-pad, :])
        if pad:
            dA_prev[i, :, :, :] = da_prev_pad[pad:-pad, pad:-pad, :]
        else:
            dA_prev[i, :, :, :] = da_prev_pad

    # YOUR CODE ENDS HERE

    # Making sure your output shape is correct
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db
```

```
# GRADED FUNCTION: pool_forward

def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implements the forward pass of the pooling layer

    Arguments:
    A_prev -- Input data, numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    hparameters -- python dictionary containing "f" and "stride"
    mode -- the pooling mode you would like to use, defined as a string ("max"␣
 ↪or "average")

    Returns:
    A -- output of the pool layer, a numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache used in the backward pass of the pooling layer, contains the␣
 ↪input and hparameters
    """

    # Retrieve dimensions from the input shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve hyperparameters from "hparameters"
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Define the dimensions of the output
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Initialize output matrix A
    A = np.zeros((m, n_H, n_W, n_C))

    # YOUR CODE STARTS HERE
    for i in range(m):
        a_prev_slice = A_prev[i]
        for h in range(n_H):
            vert_start = stride * h
            vert_end = vert_start + f

            for w in range(n_W):

                horiz_start = stride * w
                horiz_end = horiz_start + f
```

```python
                for c in range (n_C):

                    a_slice_prev = a_prev_slice[vert_start:vert_end,horiz_start:
↪horiz_end,c]

                    if mode == "max":
                        A[i, h, w, c] = np.max(a_slice_prev)
                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_slice_prev)
                    else:
                        print(mode+ "-type pooling layer NOT Defined")

    # YOUR CODE ENDS HERE

    # Store the input and hparameters in "cache" for pool_backward()
    cache = (A_prev, hparameters)

    # Making sure your output shape is correct
    #assert(A.shape == (m, n_H, n_W, n_C))

    return A, cache


def create_mask_from_window(x):
    """
    Creates a mask from an input matrix x, to identify the max entry of x.

    Arguments:
    x -- Array of shape (f, f)

    Returns:
    mask -- Array of the same shape as window, contains a True at the position␣
↪corresponding to the max entry of x.
    """
    # (1 line)
    # mask = None
    # YOUR CODE STARTS HERE
    mask = (x == np.max(x))

    # YOUR CODE ENDS HERE
    return mask

def distribute_value(dz, shape):
    """
    Distributes the input value in the matrix of dimension shape

    Arguments:
```

```
    dz -- input scalar
    shape -- the shape (n_H, n_W) of the output matrix for which we want to␣
 ↪distribute the value of dz

    Returns:
    a -- Array of size (n_H, n_W) for which we distributed the value of dz
    """

    (n_H, n_W) = shape
    average = np.prod(shape)
    a = (dz/average)*np.ones(shape)

    # YOUR CODE ENDS HERE
    return a



def pool_backward(dA, cache, mode = "max"):
    """
    Implements the backward pass of the pooling layer

    Arguments:
    dA -- gradient of cost with respect to the output of the pooling layer,␣
 ↪same shape as A
    cache -- cache output from the forward pass of the pooling layer, contains␣
 ↪the layer's input and hparameters
    mode -- the pooling mode you would like to use, defined as a string ("max"␣
 ↪or "average")

    Returns:
    dA_prev -- gradient of cost with respect to the input of the pooling layer,␣
 ↪same shape as A_prev
    """

    (A_prev, hparameters) = cache

    stride = hparameters["stride"]
    f = hparameters["f"]

    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape

    dA_prev = np.zeros(A_prev.shape)

    for i in range(m): # loop over the training examples

        # select training example from A_prev ( 1 line)
```

```
        a_prev = A_prev[i,:,:,:]

        for h in range(n_H):                    # loop on the vertical axis
            for w in range(n_W):                # loop on the horizontal axis
                for c in range(n_C):            # loop over the channels (depth)

                    # Find the corners of the current "slice" (4 lines)
                    vert_start  = h * stride
                    vert_end    = h * stride + f
                    horiz_start = w * stride
                    horiz_end   = w * stride + f

                    # Compute the backward propagation in both modes.
                    if mode == "max":

                        # Use the corners and "c" to define the current slice␣
↪from a_prev (1 line)
                        a_prev_slice = a_prev[ vert_start:vert_end, horiz_start:
↪horiz_end, c ]

                        # Create the mask from a_prev_slice (1 line)
                        mask = create_mask_from_window( a_prev_slice )

                        # Set dA_prev to be dA_prev + (the mask multiplied by␣
↪the correct entry of dA) (1 line)
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end,␣
↪c] += mask * dA[i, h, w, c]

                    elif mode == "average":

                        # Get the value da from dA (2 line)
                        da = dA[i, h, w, c]

                        # Define the shape of the filter as fxf (1 line)
                        shape = (f,f)

                        # Distribute it to get the correct slice of dA_prev. i.
↪e. Add the distributed value of da. (1 line)
                        dA_prev[i, vert_start: vert_end, horiz_start:␣
↪horiz_end, c] += distribute_value(da, shape)


    # YOUR CODE ENDS HERE

    # Making sure your output shape is correct
    assert(dA_prev.shape == A_prev.shape)
```

```python
    return dA_prev

def dense_forward(A_prev, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of␣
 ↪previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of␣
 ↪previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation␣
 ↪parameter
    cache -- a python dictionary containing "A_prev", "W" and "b" ; stored for␣
 ↪computing the backward pass efficiently
    """

    #Z = np.dot(W, A_prev) + b
    Z = np.dot(W, A_prev.T) + b
    # assert(Z.shape == (W.shape[0], A_prev.shape[1]))
    cache = (A_prev, W, b)

    return Z, cache

def dense_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer␣
 ↪(layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current␣
 ↪layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation␣
 ↪in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the␣
 ↪previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape␣
 ↪as W
    db -- Gradient of the cost with respect to b (current layer l), same shape␣
 ↪as b
```

```
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = np.dot(dZ, A_prev) / m
    db = np.sum(dZ, axis=1, keepdims=True) / m
    dA_prev = np.dot(W.T, dZ)

    # assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db

def initialize_W_b(neurons, output_shape):
    W = np.random.randn(neurons, output_shape)
    b = np.zeros((neurons, 1))
    return W, b

def initialize_kernel(kernel_size, input_channels, output_channels):
    kernel = np.random.randn(kernel_size, kernel_size, input_channels,␣
 ↪output_channels) * np.sqrt(
        2 / (kernel_size * kernel_size * input_channels)
    )
    bias = np.zeros((1, 1, 1, output_channels))
    return kernel, bias

def softmax_forward(z):
    z -= np.max(
        z, axis=0, keepdims=True
    )  # axis=0 means coloumn z is the shape of (n_l, batch_size), axis=0 means␣
 ↪the max value of each column b/c we are giving input as (n_l, batch_size)
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=0, keepdims=True), None

def softmax_backward(z):
    # we have calculated the dA/dz in the loss_prime itself,
    # that returns (dJ/dA)*(dA/dz) itself so no need to take the derivative of␣
 ↪activation here
    return z

def loss(y_true, y_pred):
    m_samples = y_pred.shape[-1]

    cost = -np.sum(y_true * np.log(y_pred + 1e-10))   # shape = (batch_size,)

    cost = cost / m_samples
```

```python
    return cost

def loss_backward(y_true, y_pred):
    m_samples = y_pred.shape[-1]

    # this is little bit different from the else loss_prime,
    # this return the (dJ/dA)*(dA/dz) so we don't need to find the derivative␣
↪of sofmax_prime
    cost_prime = (y_pred - y_true) / m_samples

    return cost_prime


def flatten_forward(A_prev):
    """
    Implement the forward propagation for a flatten layer

    Arguments:
    A_prev -- activations from the previous layer (or input data): (size of␣
↪previous layer, number of examples)

    Returns:
    A -- flatten output of the activation function, also called␣
↪"post-activation" value
    cache -- a python dictionary containing "A_prev"; stored for computing the␣
↪backward pass efficiently
    """

    # ( 1 line)
    # A = None
    # YOUR CODE STARTS HERE
    A = A_prev.reshape(A_prev.shape[0], -1)
    # YOUR CODE ENDS HERE

    # Store input shape in "cache" for the backprop
    cache = A_prev.shape

    return A, cache

def flatten_backward(dA, cache):
    """
    Implement the backward propagation for a flatten layer

    Arguments:
```

```
    dA -- gradient of the cost with respect to the flatten output (of the␣
↪current layer l)
    cache -- cache of values needed for the flatten_backward(), output of␣
↪flatten_forward()

    Returns:
    dA_prev -- gradient of the cost with respect to the activation (of the␣
↪previous layer l-1), same shape as A_prev
    """

    # Retrieve information from "cache"
    (A_prev_shape) = cache

    # Reshape dA to A_prev_shape
    dA_prev = dA.reshape(A_prev_shape)

    return dA_prev
```

3.2 Train

[ ]:
```
kernels_shape = (3,3, 1, 9)
b_shape = (1, 1, 1, 9)
kernel_W = np.random.randn(*kernels_shape)
kernel_b = np.random.randn(*b_shape)
W, b = initialize_W_b(10, 13*13*9)



hparameters = {"pad" : 0,
                "stride": 1}

max_pool_hparameters = {"stride" : 2,
                        "f": 2}
mode = "max"
```

[ ]:
```
def train_one(X_train, y_train, lr, parameters):
    #Forward Propagation
    kernel_W = parameters["kernel_W"]
    kernel_b = parameters["kernel_b"]
    W = parameters["W"]
    b = parameters["b"]

    A_prev = X_train
    A_prev, cache_conv = conv_forward(A_prev, kernel_W, kernel_b, hparameters)
    A_prev, cache_pool = pool_forward(A_prev, max_pool_hparameters, mode)
    A_prev, cache_flatten = flatten_forward(A_prev)
```

```
    A_prev, cache_dense = dense_forward(A_prev, W, b)
    A_prev, cache_softmax = softmax_forward(A_prev)

    # Backward pass
    dA_prev = loss_backward(y_train, A_prev)
    dA_prev = softmax_backward(dA_prev)
    dA_prev, dW, db = dense_backward(dA_prev, cache_dense)
    W -= lr * dW
    b -= lr * db
    dA_prev = flatten_backward(dA_prev, cache_flatten)
    dA_prev = pool_backward(dA_prev, cache_pool)
    dA_prev, dW, db = conv_backward(dA_prev, cache_conv)
    kernel_W -= lr * dW
    kernel_b -= lr * db

    params = {
        "kernel_W": kernel_W,
        "kernel_b": kernel_b,
        "W": W,
        "b": b
    }
    loss_val = loss(y_train, A_prev)
    accuracy_val = accuracy(y_train, A_prev)
    return params, loss_val, accuracy_val

def accuracy(y_true, y_pred):
    y_pred = np.argmax(y_pred, axis=0)
    y_true = np.argmax(y_true, axis=0)
    return np.sum(y_pred == y_true) / len(y_true)
```

[ ]:
```
def random_sample(X, y, n):
    index = np.random.choice(len(X), n, replace=False)
    return X[index], y[index]
```

[ ]:
```
parameters = {
    "kernel_W": kernel_W,
    "kernel_b": kernel_b,
    "W": W,
    "b": b
}
epochs = 2
losses = []
X, y = random_sample(X_train, y_train, 101)
for i in range(epochs):
    # parameters, loss_val, acc = train_one(X_train, y_train, 0.01, parameters)
    parameters, loss_val, acc = train_one(X, y.T, 0.01, parameters)
    losses.append(loss_val)
```

```python
    print(f"Epoch: {i+1}, Loss: {loss_val:.4f}, Accuracy: {acc:.4f}")
```

```
Epoch: 1, Loss: 19.0564, Accuracy: 0.1683
Epoch: 2, Loss: 18.3659, Accuracy: 0.1485
```

[ ]:
```python
def predict(X, parameters):
    kernel_W = parameters["kernel_W"]
    kernel_b = parameters["kernel_b"]
    W = parameters["W"]
    b = parameters["b"]

    A_prev = X
    A_prev, cache_conv = conv_forward(A_prev, kernel_W, kernel_b, hparameters)
    A_prev, cache_pool = pool_forward(A_prev, max_pool_hparameters, mode)
    A_prev, cache_flatten = flatten_forward(A_prev)
    A_prev, cache_dense = dense_forward(A_prev, W, b)
    A_prev, cache_softmax = softmax_forward(A_prev)
    return A_prev
```

[ ]:
```python
train_acc = accuracy(y_train.T, predict(X_train, parameters))
test_acc = accuracy(y_test.T, predict(X_test, parameters))
```

[ ]:
```python
print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
```

```
Train Accuracy: 0.1090, Test Accuracy: 0.0930
```