Exercise 4: Solution

Linear Regression

Linear Model – Forward

```python
def forward(self, X):
    """
    Performs the forward pass of the model.

    :param X: N x D array of training data. Each row is a D-dimensional point.
    :return: Predicted labels for the data in X, shape N x 1
             1-dimensional array of length N with housing prices.
    """
    assert self.W is not None, "weight matrix W is not initialized"
    # add a column of 1s to the data for the bias term
    batch_size, _ = X.shape
    X = np.concatenate((X, np.ones((batch_size, 1))), axis=1)
    # save the samples for the backward pass
    self.cache = X
    # output variable
    y = None
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass and return the output of the model.      #
    ########################################################################
    y = X.dot(self.W)
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return y
```

Remark: the input variable X is saved in self.cache, such that the backward pass can access the input variable.

Linear Model – Backward

```python
def backward(self, y):
    """
    Performs the backward pass of the model.

    :param y: N x 1 array. The output of the forward pass.
    :return: Gradient of the model output (y=X*W) wrt W
    """
    assert self.cache is not None, "run a forward pass before the backward pass"
    dW = None
    ###############################################################################
    # TODO:                                                                       #
    # Implement the backward pass. Return the gradient wrt W, dW                  #
    # The data X are stored in self.cache.                                        #
    ###############################################################################
    dW = self.cache
    ###############################################################################
    #                             END OF YOUR CODE                                #
    ###############################################################################
    self.cache = None
    return dW
```

The backward pass can access the input X via the variable self.cache

L1-Loss – Forward

```python
def forward(self,y_out, y_truth):
    """
    Performs the forward pass of the L1 loss function.

    :param y_out: [N, ] array predicted value of your model.
           y_truth: [N, ] array ground truth value of your training set.
    :return: [N, ] array of L1 loss for each sample of your training set.
    """
    result = None
    #########################################################################
    # TODO:                                                                 #
    # Implement the forward pass and return the output of the L1 loss.     #
    #########################################################################
    result = np.abs(y_out - y_truth)
    #########################################################################
    #                       END OF YOUR CODE                                #
    #########################################################################

    return result
```

L1-Loss – Backward

```python
def backward(self,y_out, y_truth):
    """
    Performs the backward pass of the L1 loss function.

    :param y_out: [N, ] array predicted value of your model.
           y_truth: [N, ] array ground truth value of your training set.
    :return: [N, ] array of L1 loss gradients w.r.t y_out for
              each sample of your training set.
    """
    gradient = None
    ###########################################################################
    # TODO:                                                                   #
    # Implement the backward pass. Return the gradient wrt y_out              #
    # hint: you may use np.where here.                                        #
    ###########################################################################
    gradient = y_out - y_truth

    zero_loc = np.where(gradient==0)
    negative_loc = np.where(gradient<0)
    positive_loc = np.where(gradient>0)

    gradient[zero_loc] = 0
    gradient[positive_loc] = 1
    gradient[negative_loc] = -1
    ###########################################################################
    #                            END OF YOUR CODE                             #
    ###########################################################################
    return gradient
```

MSE-Loss – Forward

```
def forward(self,y_out, y_truth):
    """
    Performs the forward pass of the MSE loss function.

    :param y_out: [N, ] array predicted value of your model.
            y_truth: [N, ] array ground truth value of your training set.
    :return: [N, ] array of MSE loss for each sample of your training set.
    """
    result = None
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass and return the output of the MSE loss.    #
    ########################################################################
    result = (y_out - y_truth)**2
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return result
```

MSE-Loss – Backward

```python
def backward(self,y_out, y_truth):
    """
    Performs the backward pass of the MSE loss function.

    :param y_out: [N, ] array predicted value of your model.
           y_truth: [N, ] array ground truth value of your training set.
    :return: [N, ] array of MSE loss gradients w.r.t y_out for
              each sample of your training set.
    """
    gradient = None
    ############################################################################
    # TODO:                                                                    #
    # Implement the backward pass. Return the gradient wrt y_out               #
    ############################################################################
    gradient = 2*(y_out - y_truth)

    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    return gradient
```

Logistic Regression

Classifier – Sigmoid

```python
def sigmoid(self, x):
    """
    Computes the ouput of the sigmoid function

    :param x: input of the sigmoid, np.array of any shape
    :return: output of the sigmoid with same shape as input vector x
    """
    out = None
    ########################################################################
    # TODO:                                                                #
    # Implement the sigmoid function, return out                           #
    ########################################################################
    out = 1 / (1 + np.exp(-x))
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return out
```

Classifier – Forward

```python
def forward(self, X):
    """
    Performs the forward pass of the model.

    :param X: N x D array of training data. Each row is a D-dimensional point.
    :return: Predicted labels for the data in X, shape N x 1
             1-dimensional array of length N with classification scores.
    """
    assert self.W is not None, "weight matrix W is not initialized"
    # add a column of 1s to the data for the bias term
    batch_size, _ = X.shape
    X = np.concatenate((X, np.ones((batch_size, 1))), axis=1)
    # save the samples for the backward pass
    self.cache = X
    # output variable
    y = None
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass and return the output of the model. Note  #
    # that you need to implement the function self.sigmoid() for that      #
    ########################################################################
    y = X.dot(self.W)
    y = self.sigmoid(y)
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return y
```

Classifier – Backward

```python
def backward(self, y):
    """
    Performs the backward pass of the model.

    :param y: N x 1 array. The output of the forward pass.
    :return: Gradient of the model output (y=sigma(X*W)) wrt W
    """
    assert self.cache is not None, "run a forward pass before the backward pass"
    dW = None
    ###########################################################################
    # TODO:                                                                   #
    # Implement the backward pass. Return the gradient wrt W, dW              #
    # The data X is stored in self.cache. Be careful with the dimensions of   #
    # W, X and y and note that the derivative of the sigmoid fct can be       #
    # expressed by sigmoids itself (--> use the function self.sigmoid() here) #
    ###########################################################################
    X = self.cache

    # dz/dW, where z = X * W
    dW = X

    # dsigmoid/dz, where z = X * W
    dz = y * (1 - y)

    # dy/dW = dsigmoid/dz * dz/dW
    dW*= dz
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dW
```

Keep the dimensions of the arrays in mind:
X: [N, D]
y: [N, 1],
dW should be of shape [N, D] as it contains a gradient of W w.r.t. each samples (N: number of samples).
The average over all samples is taken in the solver step.

BCE – Forward

```python
def forward(self,y_out, y_truth):
    """
    Performs the forward pass of the binary cross entropy loss function.

    :param y_out: [N, ] array predicted value of your model.
            y_truth: [N, ] array ground truth value of your training set.
    :return: [N, ] array of binary cross entropy loss for each sample of your training set.
    """
    result = None
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass and return the output of the BCE loss.    #
    ########################################################################
    result = -y_truth * np.log(y_out) - (1-y_truth) * np.log(1-y_out)
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return result
```

BCE – Backward

```python
def backward(self,y_out, y_truth):
    """
    Performs the backward pass of the loss function.

    :param y_out: [N, ] array predicted value of your model.
           y_truth: [N, ] array ground truth value of your training set.
    :return: [N, ] array of binary cross entropy loss gradients w.r.t y_out for
              each sample of your training set.
    """
    gradient = None

    ##########################################################################
    # TODO:                                                                  #
    # Implement the backward pass. Return the gradient wrt y_out             #
    ##########################################################################
    gradient = - (y_truth / y_out) + (1-y_truth)/(1-y_out)
    ##########################################################################
    #                           END OF YOUR CODE                             #
    ##########################################################################
    return gradient
```

Optimization

Solver – Step

```python
def _step(self):
    """
    Make a single gradient update. This is called by train() and should not
    be called manually.
    """
    model = self.model
    loss_func = self.loss_func
    X_train = self.X_train
    y_train = self.y_train
    opt = self.opt
    ##########################################################################
    #   TODO:                                                                #
    #   Get the gradients dhat{y} / dW and dLoss / dhat{y}.                  #
    #   Combine them via the chain rule to obtain dLoss / dW.                #
    #   Proceed by performing an optimizing step using the given             #
    #   optimizer (by calling opt.step() with the gradient wrt W)            #
    #                                                                        #
    #   Hint: don't forget to divide number of samples when computing the    #
    #   gradient!                                                            #
    ##########################################################################
    model_forward, model_backward = model(X_train)
    loss, loss_grad = loss_func(model_forward, y_train)

    grad = loss_grad.T.dot(model_backward) / loss_grad.shape[0]

    opt.step(grad.T)
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################
```

Model and loss_func return (forward, backward) when called, cf. __call__() in their base classes.

Mind the dimensions of all elements. In particular, we want to update W (via opt.step()) with an array of the same shape, i.e., [1, D]

Optimizer – Step

```
class Optimizer(object):
    def __init__ (self, model, learning_rate=5e-5):
        self.model = model
        self.lr = learning_rate

    def step(self, dw):
        """
        :param dw: [D+1,1] array gradient of loss w.r.t weights of your linear model
        :return weight: [D+1,1] updated weight after one step of gradient descent
        """
        weight = self.model.W
        #########################################################################
        # TODO:                                                                 #
        # Implement the gradient descent for 1 step to compute the weight      #
        #########################################################################
        weight -= self.lr * dw
        #########################################################################
        #                         END OF YOUR CODE                              #
        #########################################################################
        self.model.W = weight
```

Review of Exercise 4

https://docs.google.com/forms/d/e/1FAIpQLSdHos6ShLizDpHdXocfkCm2zVxrt3lh5YVJnfXyrFaI9Gt1zA/viewform?usp=sf_link

Questions? Moodle

☺