Exercise 4: Solution

Loss: BCE - Forward method

```
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
```

Loss: BCE - Backward method

```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
```

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 method

```
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 method

```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 sigmoid itself                                          #
    ########################################################################

    X = self.cache
    N, _ = X.shape

    # 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 the output w.r.t. W for each sample (N: number of samples). The average over all samples is taken in the solver step.

Optimization

Optimizer: Step method

```python
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
```

Solver: Step method

```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]

Feedback Exercise 4

https://docs.google.com/forms/d/e/1FAIpQLSdQ1MGokyD-aaALcvUBlPYFrWbQL7akP-Z0Ov7awDnciqbiOw/viewform

Questions? Piazza ☺