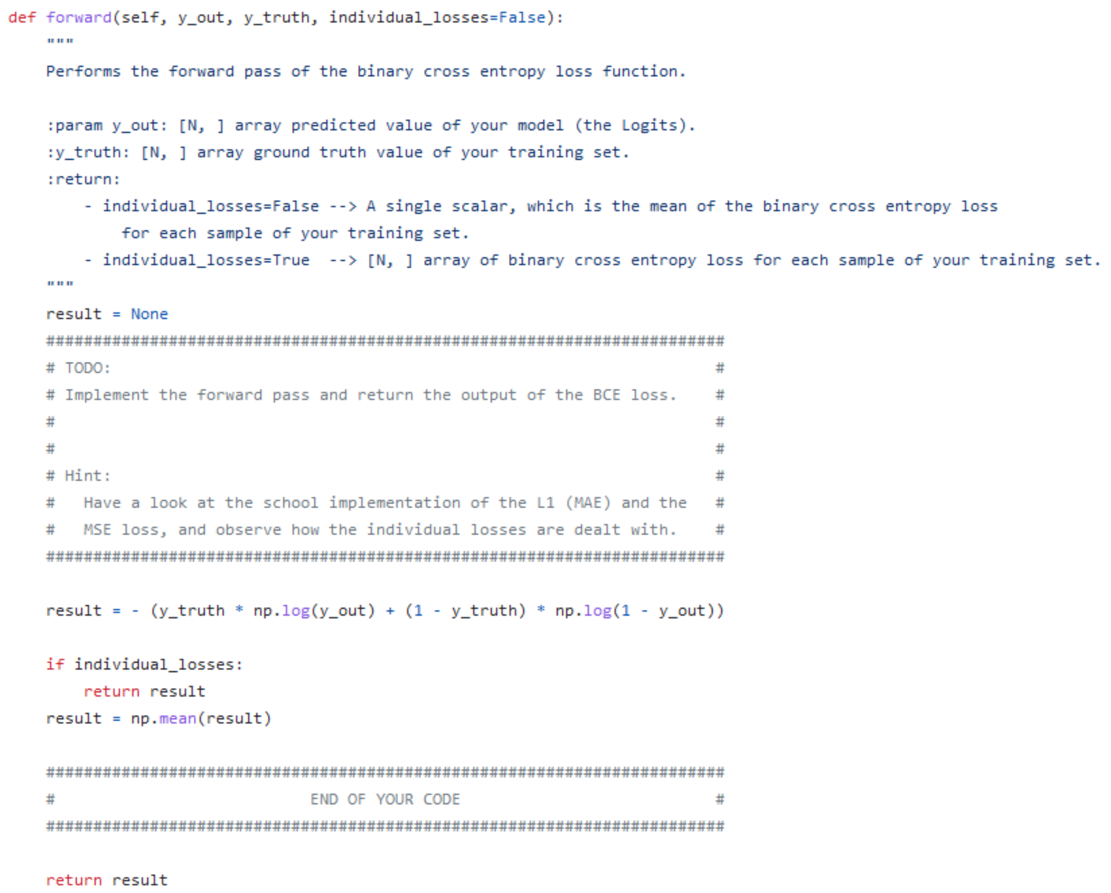
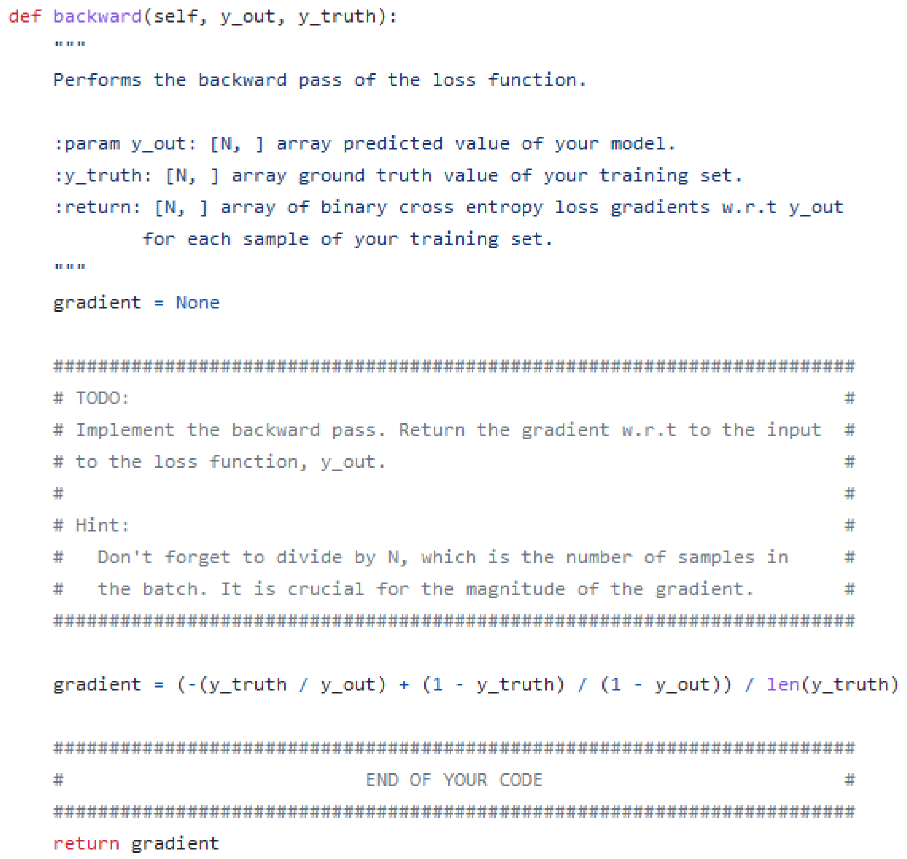
Exercise 4: Solution

Loss: BCE – Forward method



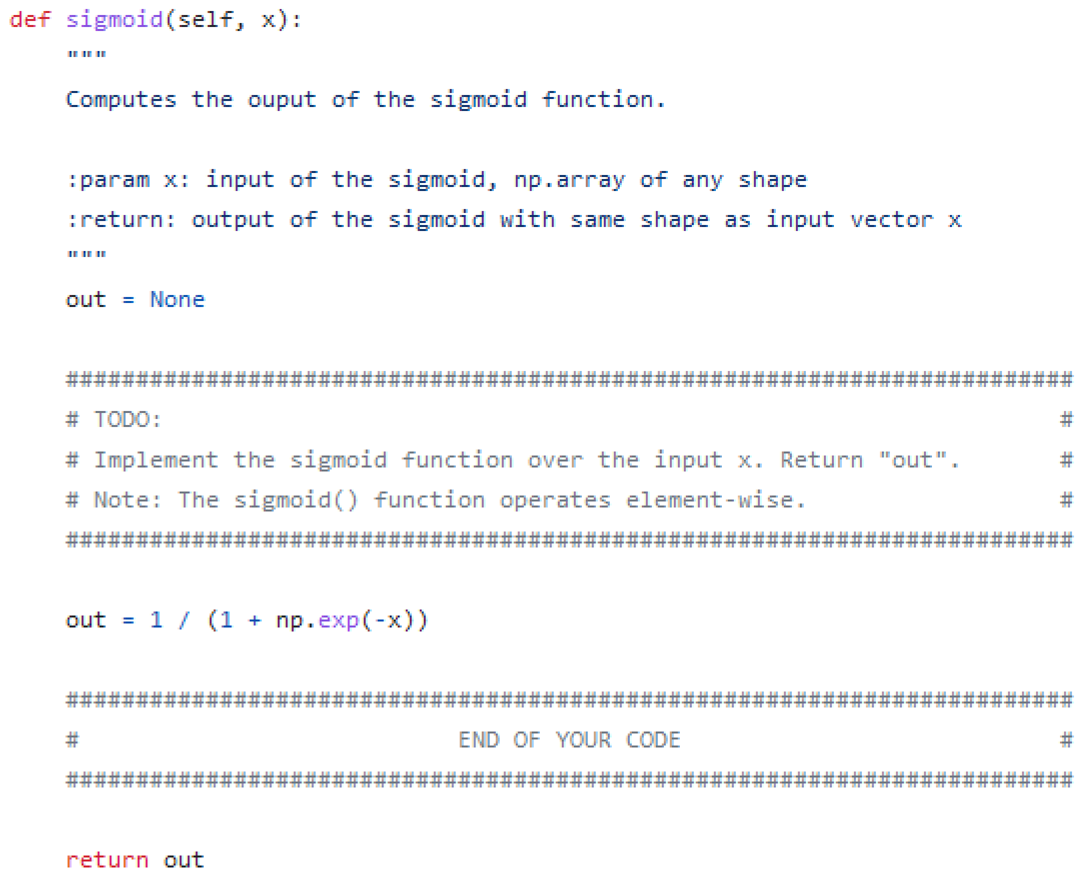
```
def forward(self, y_out, y_truth, individual_losses=False):
    """
    Performs the forward pass of the binary cross entropy loss function.

    :param y_out: [N, ] array predicted value of your model (the Logits).
    :y_truth: [N, ] array ground truth value of your training set.
    :return:
        - individual_losses=False --> A single scalar, which is the mean of the binary cross entropy loss
            for each sample of your training set.
        - individual_losses=True  --> [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.    #
    #                                                                      #
    #                                                                      #
    # Hint:                                                                #
    #   Have a look at the school implementation of the L1 (MAE) and the   #
    #   MSE loss, and observe how the individual losses are dealt with.    #
    ########################################################################

    result = - (y_truth * np.log(y_out) + (1 - y_truth) * np.log(1 - y_out))

    if individual_losses:
        return result
    result = np.mean(result)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return result
```

Loss: BCE – Backward method



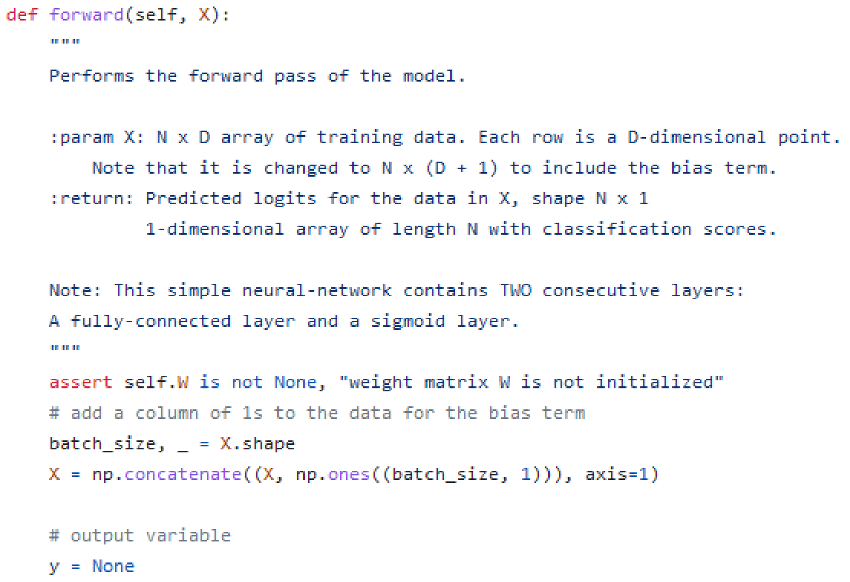
```
def backward(self, y_out, y_truth):
    """
    Performs the backward pass of the loss function.

    :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 w.r.t to the input  #
    # to the loss function, y_out.                                         #
    #                                                                      #
    # Hint:                                                                #
    #   Don't forget to divide by N, which is the number of samples in     #
    #   the batch. It is crucial for the magnitude of the gradient.        #
    ########################################################################

    gradient = (-(y_truth / y_out) + (1 - y_truth) / (1 - y_out)) / len(y_truth)

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

Classifier: Sigmoid

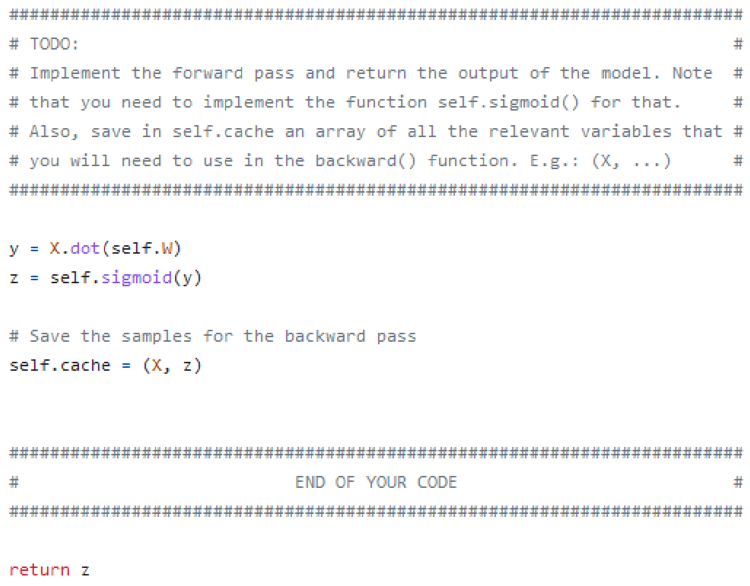


```python
def sigmoid(self, x):
    """
    Computes the ouput of the sigmoid function.

    :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 over the input x. Return "out".       #
    # Note: The sigmoid() function operates element-wise.                  #
    ########################################################################

    out = 1 / (1 + np.exp(-x))

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return out
```

Classifier: Forward method

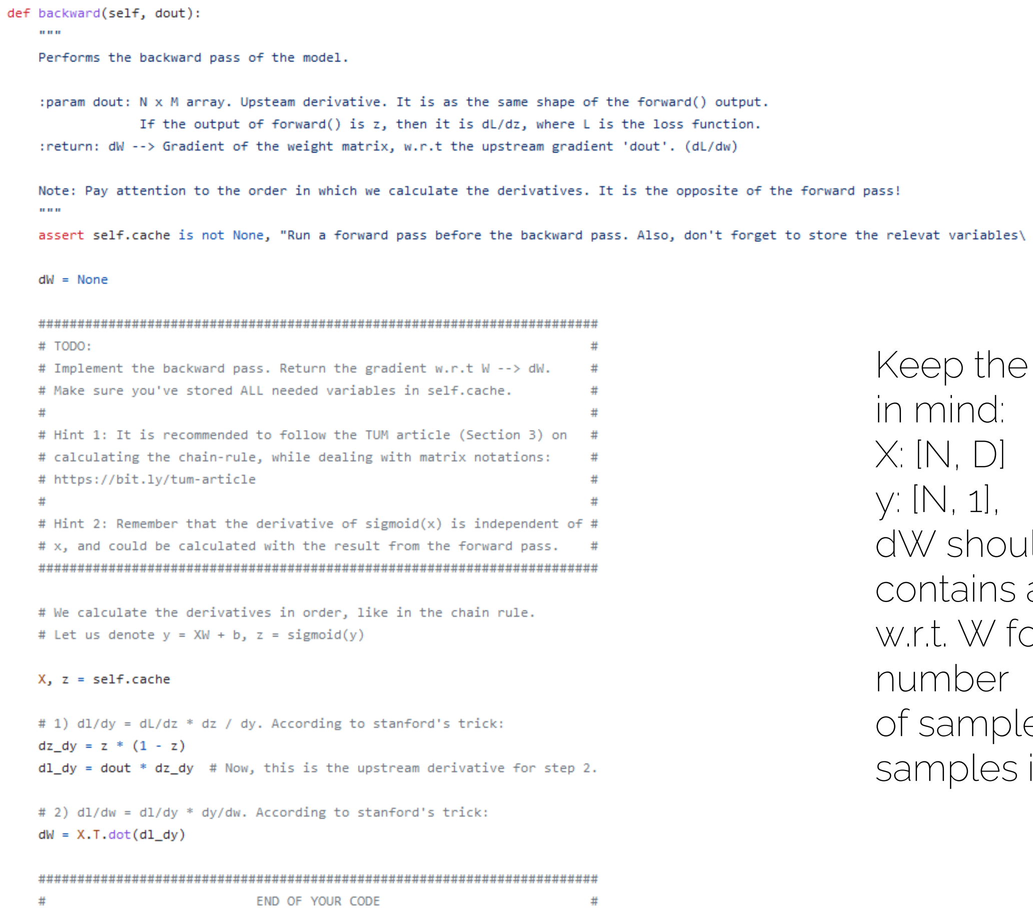


```python
def forward(self, X):
    """
    Performs the forward pass of the model.

    :param X: N x D array of training data. Each row is a D-dimensional point.
        Note that it is changed to N x (D + 1) to include the bias term.
    :return: Predicted logits for the data in X, shape N x 1
             1-dimensional array of length N with classification scores.

    Note: This simple neural-network contains TWO consecutive layers:
    A fully-connected layer and a sigmoid layer.
    """
    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)

    # output variable
    y = None
```



```python
########################################################################
# TODO:                                                                #
# Implement the forward pass and return the output of the model. Note  #
# that you need to implement the function self.sigmoid() for that.     #
# Also, save in self.cache an array of all the relevant variables that #
# you will need to use in the backward() function. E.g.: (X, ...)      #
########################################################################

y = X.dot(self.W)
z = self.sigmoid(y)

# Save the samples for the backward pass
self.cache = (X, z)


########################################################################
#                           END OF YOUR CODE                           #
########################################################################

return z
```

Classifier: Backward method

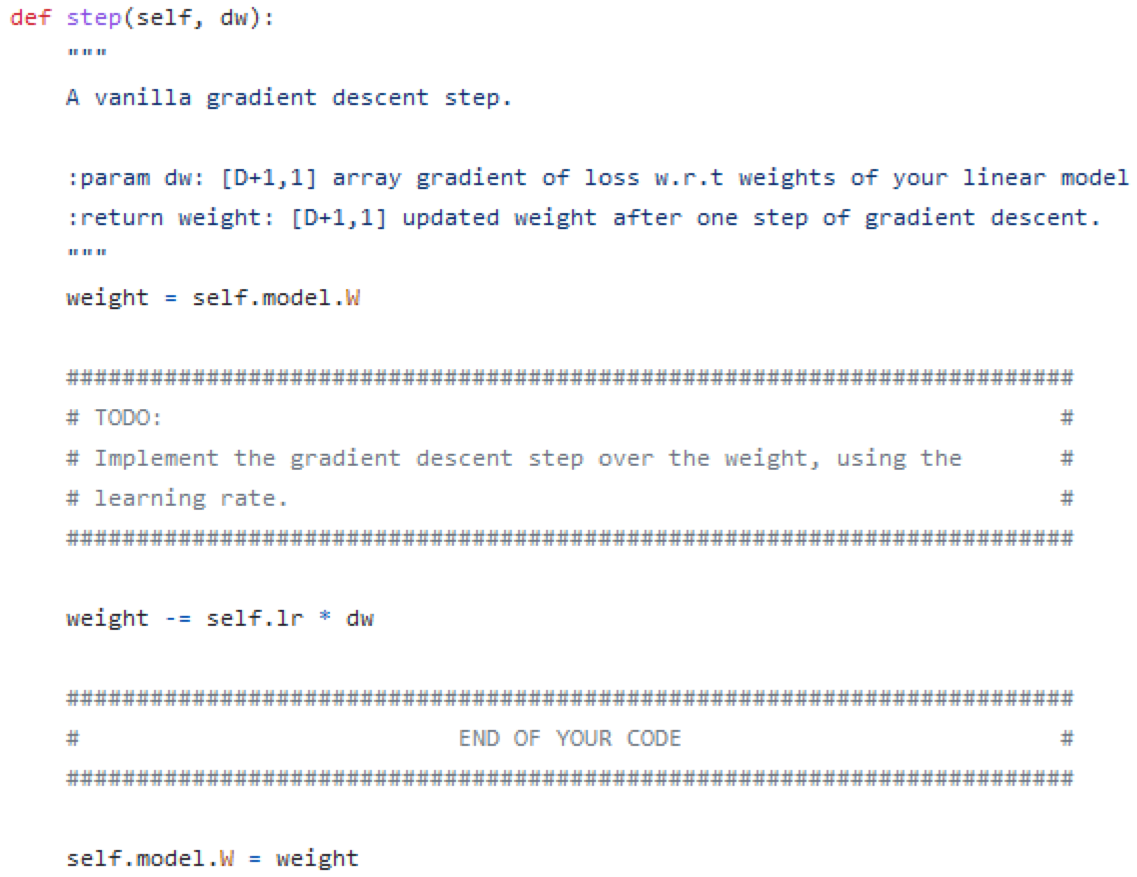


```
def backward(self, dout):
    """
    Performs the backward pass of the model.

    :param dout: N x M array. Upsteam derivative. It is as the same shape of the forward() output.
                 If the output of forward() is z, then it is dL/dz, where L is the loss function.
    :return: dW --> Gradient of the weight matrix, w.r.t the upstream gradient 'dout'. (dL/dw)

    Note: Pay attention to the order in which we calculate the derivatives. It is the opposite of the forward pass!
    """
    assert self.cache is not None, "Run a forward pass before the backward pass. Also, don't forget to store the relevat variables\

    dW = None

    ########################################################################
    # TODO:                                                                #
    # Implement the backward pass. Return the gradient w.r.t W --> dW.     #
    # Make sure you've stored ALL needed variables in self.cache.          #
    #                                                                      #
    # Hint 1: It is recommended to follow the TUM article (Section 3) on   #
    # calculating the chain-rule, while dealing with matrix notations:     #
    # https://bit.ly/tum-article                                           #
    #                                                                      #
    # Hint 2: Remember that the derivative of sigmoid(x) is independent of #
    # x, and could be calculated with the result from the forward pass.    #
    ########################################################################

    # We calculate the derivatives in order, like in the chain rule.
    # Let us denote y = XW + b, z = sigmoid(y)

    X, z = self.cache

    # 1) dl/dy = dL/dz * dz / dy. According to stanford's trick:
    dz_dy = z * (1 - z)
    dl_dy = dout * dz_dy  # Now, this is the upstream derivative for step 2.

    # 2) dl/dw = dl/dy * dy/dw. According to stanford's trick:
    dW = X.T.dot(dl_dy)

    ########################################################################
    #                           END OF YOUR CODE                           #
```

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.

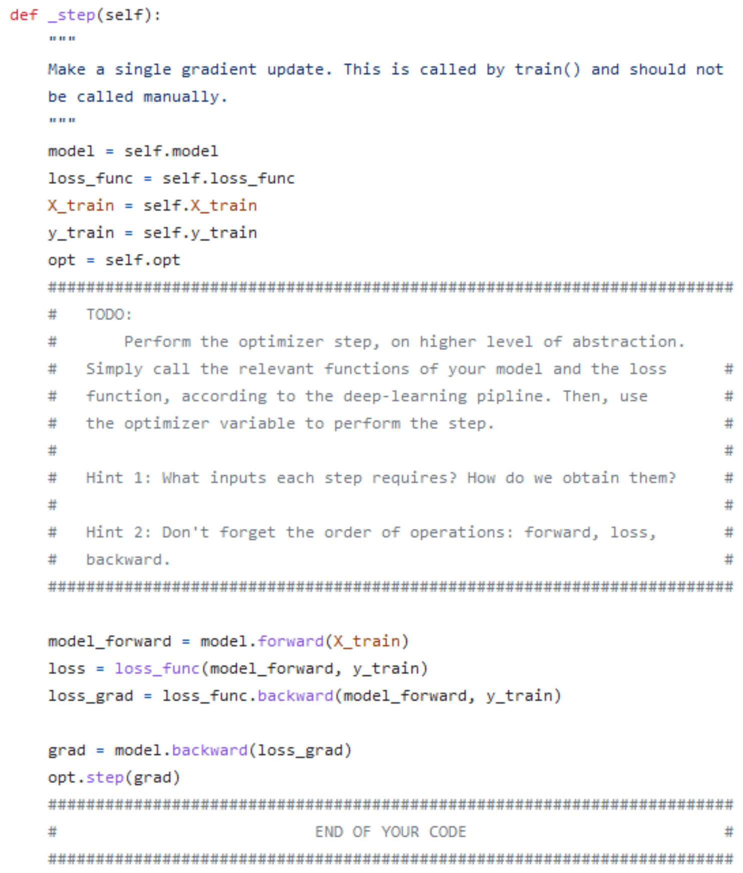
Optimization

Optimizer: Step method



```python
def step(self, dw):
    """
    A vanilla gradient descent step.

    :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 step over the weight, using the       #
    # learning rate.                                                       #
    ########################################################################

    weight -= self.lr * dw

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    self.model.W = weight
```

Solver: Step method



```
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:
    #       Perform the optimizer step, on higher level of abstraction.
    #   Simply call the relevant functions of your model and the loss      #
    #   function, according to the deep-learning pipline. Then, use       #
    #   the optimizer variable to perform the step.                        #
    #                                                                      #
    #   Hint 1: What inputs each step requires? How do we obtain them?     #
    #                                                                      #
    #   Hint 2: Don't forget the order of operations: forward, loss,       #
    #   backward.                                                          #
    ########################################################################

    model_forward = model.forward(X_train)
    loss = loss_func(model_forward, y_train)
    loss_grad = loss_func.backward(model_forward, y_train)

    grad = model.backward(loss_grad)
    opt.step(grad)
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

Model and loss_func return forward when called, cf. __call__() in their base classes. For loss gradient use backward method.

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]

Questions? Piazza