HW3 Review

April 3, 2026

Huihan Li

Problem 1: Multi-class Perceptron (16pts)

Recall that a linear model for a multiclass classification problem with C classes is parameterized by C weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C \in \mathbb{R}^d$. In class, we derived the solution for multiclass logistic regression by minimizing the multiclass logistic loss. In this problem, you will derive the multiclass perceptron algorithm in a similar way. Specifically, the multiclass perceptron loss on a training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times [C]$ is defined as

$$F(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C) = \frac{1}{n} \sum_{i=1}^{n} F_i(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C), \quad \text{where } F_i(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C) = \max \left\{ 0, \max_{y \neq y_i} \left(\boldsymbol{w}_y^{\mathrm{T}} \boldsymbol{x}_i - \boldsymbol{w}_{y_i}^{\mathrm{T}} \boldsymbol{x}_i \right) \right\}.$$

1.1 (8pts) To optimize this loss function, we need to first derive its gradient. Specifically, for each $i \in [n]$ and $c \in [C]$, write down the partial derivative $\frac{\partial F_i}{\partial \boldsymbol{w}_c}$ (provide your reasoning). For simplicity, you can assume that for any i, $\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}_i, \ldots, \boldsymbol{w}_C^{\mathrm{T}}\boldsymbol{x}_i$ are always C distinct values (so that there is no tie when taking the max over them, and consequently no non-differentiable points needed to be considered).

1.2 (4pts) Similarly to the binary case, multiclass perceptron is simply applying SGD with learning rate 1 to minimize the multiclass perceptron loss. Based on this information, fill in the missing details in the repeat-loop of the algorithm below (your solution cannot contain implicit quantities such as $\nabla F_i(\boldsymbol{w})$; instead, write down the exact formula based on your solution from the last question).

Algorithm 1: Multiclass Perceptron

---

1 **Input:** A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$

2 **Initialization:** $\boldsymbol{w}_1 = \cdots = \boldsymbol{w}_C = \boldsymbol{0}$

3 **Repeat:**

1.3 (4pts) At this point, you should find that the parameters $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C$ computed by Multiclass Perceptron are always linear combinations of the training points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, that is, $\boldsymbol{w}_c = \sum_{i=1}^{n} \alpha_{c,i} \boldsymbol{x}_i$ for some coefficients $\alpha_{c,i}$. Just as we saw for kernelized linear regression, this means that we can kernelize Multiclass Perceptron as well. Consider some kernel function $k(\cdot, \cdot)$ with a corresponding feature map $\boldsymbol{\phi}(\cdot)$. Fill in the missing details in the repeat-loop of the algorithm below that maintains and updates the coefficient $\alpha_{c,i}$ for all c and i for the the kernalized solution $\boldsymbol{w}_c = \sum_{i=1}^{n} \alpha_{c,i} \boldsymbol{\phi}(\boldsymbol{x}_i)$. To do this, try to see how the weights $\alpha_{c,i}$ should be updated such that $\boldsymbol{w}_c = \sum_{i=1}^{n} \alpha_{c,i} \boldsymbol{\phi}(\boldsymbol{x}_i)$ is always the same as what one would get by running Algorithm 1 with $\boldsymbol{x}_i$ replaced by $\boldsymbol{\phi}(\boldsymbol{x}_i)$ for all i.

Algorithm 2: Multiclass Perceptron with kernel function $k(\cdot, \cdot)$

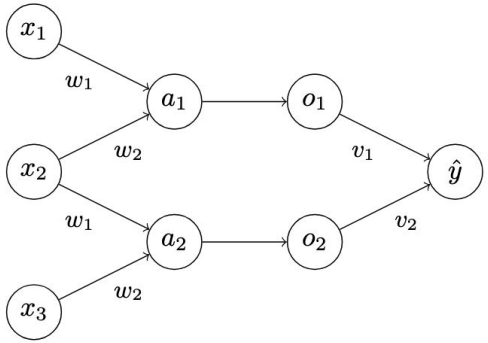
Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$

Initialize: $\alpha_{c,n} = 0$ for all $c \in [C]$ and $i \in [n]$

Repeat:

Problem 2: Backpropagation for CNN (18pts)

Consider the following mini convolutional neural net, where (x_1, x_2, x_3) is the input, followed by a convolution layer with a filter (w_1, w_2), a ReLU layer, and a fully connected layer with weight (v_1, v_2).



More concretely, the computation is specified by

$$
\begin{aligned}
a_1 &= x_1 w_1 + x_2 w_2 \\
a_2 &= x_2 w_1 + x_3 w_2 \\
o_1 &= \max\{0, a_1\} \\
o_2 &= \max\{0, a_2\} \\
\hat{y} &= o_1 v_1 + o_2 v_2
\end{aligned}
$$

For an example $(\boldsymbol{x}, y) \in \mathbb{R}^3 \times \{-1, +1\}$, the logistic loss of the CNN is

$$\ell = \ln(1 + \exp(-y\hat{y})),$$

which is a function of the parameters of the network: w_1, w_2, v_1, v_2.

2.1 (4pts) Write down $\frac{\partial \ell}{\partial v_1}$ and $\frac{\partial \ell}{\partial v_2}$ (show the intermediate steps that use chain rule). You can use the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ to simplify your notation.

2.2 (6pts) Write down $\frac{\partial \ell}{\partial w_1}$ and $\frac{\partial \ell}{\partial w_2}$ (show the intermediate steps that use chain rule). The derivative of the ReLU function is $H(a) = \mathbb{I}[a > 0]$, which you can use directly in your answer.

2.3 (8pts) Using the derivations above, fill in the missing details of the repeat-loop of the Backpropagation algorithm below that is used to train this mini CNN.

Algorithm 3: Backpropagation for the above mini CNN

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$, learning rate η

Initialize: set w_1, w_2, v_1, v_2 randomly

Repeat:

    randomly pick an example $(\boldsymbol{x}_i, y_i)$

    Forward propagation:

    Backward propagation:

Practice Problem 1

(a) Consider the following for loop block which attempts to do Backpropogation to train a 1 hidden layer neural network with a ReLU activation function on a regression task. Here `W1` are weights from the input to the hidden layer, `A1` is the input to the hidden layer, `O1` is the output of the hidden layer, `W2` weights from the hidden layer to the output, and `A2` is the output of the model. There are no bias terms in the architecture (assume that is not needed for this task).

```
# Training loop
for epoch in range(num_epochs):
    # Forward pass
    A1 = np.dot(X, W1)
    O1 = relu(A1)  # ReLU activation for hidden layer
    A2 = np.dot(O1, W2)   # Linear activation for output layer

    # Compute squared loss (regression problem)
    loss = np.mean((A2 - y) ** 2)  # Mean squared error

    # Backpropagation and update
    dA2 = 2 * (A2 - y) / X.shape[0]  # Derivative of squared loss
    dW2 = np.dot(O1.T, dA2)
    W2 -= learning_rate * dW2

    dO1 = np.dot(dA2, W2.T)
    dA1 = dO1 * relu_derivative(A1)  # ReLU derivative
    dW1 = np.dot(X.T, dA1)
    W1 -= learning_rate * dW1

return W1, W2
```

Figure 2: Code for training a 1 hidden layer neural network

The code (when accompanied with supporting functionality) will compile and run, but it has a conceptual mistake due to which the model will not train properly. Identify the mistake, and explain why it is a problem.

(b) Now assume that the error in the previous part has been fixed. Consider the following two initialization schemes in Fig. 3 for the weights `W1` and `W2`.

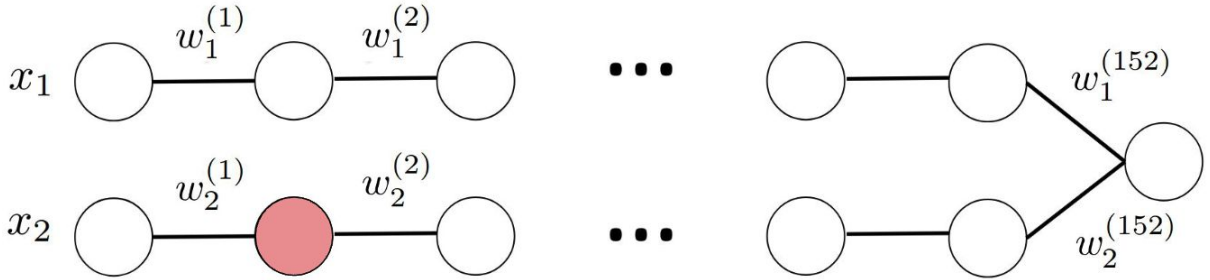
```
# Initialization scheme 1
W1 = np.random.randn(input_size, hidden_size) * 0.01
W2 = np.random.randn(hidden_size, output_size) * 0.01

# Initialization scheme 2
W1 = np.zeros((input_size, hidden_size))
W2 = np.zeros((hidden_size, output_size))
```

Figure 3: Two possible initialization schemes

Are both initialization schemes equally good? Explain.

Practice Problem 2

Consider the 152-layer neural network in the figure below. Throughout the question, we assume that the training example is $(\boldsymbol{x}, y)$ where $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$ and $y \in \mathbb{R}$ (all activations and gradients are with respect to this training example). The network has no bias terms, and it is fully-connected only for the final output node (i.e. the input x_1 is not connected to the red node in the next layer and so on, except for the last output node).



Notation: The notation for the weights is as described in the figure above. The notation for the inputs and outputs to the nodes is analogous. For e.g., the red node has input $a_2^{(1)} = w_2^{(1)} x_2$ and output $o_2^{(1)} = h(a_2^{(1)})$. Similarly, the final output node has input $a_1^{(152)} = w_1^{(152)} o_1^{(151)} + w_2^{(152)} o_2^{(151)}$ and output $o_1^{(152)} = h(a_1^{(152)})$.

Activation functions: The network uses some activation function $h(\cdot)$ for all nodes in all layers.

Loss function: The network is trained on the standard squared loss, so that $L(\boldsymbol{w}) = \frac{1}{2}(o_1^{(152)} - y)^2$ for the example $(\boldsymbol{x}, y)$, where $\boldsymbol{w}$ is the vector of all the weights in the network.

(a) Derive expressions for $\frac{\partial L(\boldsymbol{w})}{\partial o_1^{(152)}}$ and $\frac{\partial L(\boldsymbol{w})}{\partial w_1^{(152)}}$.

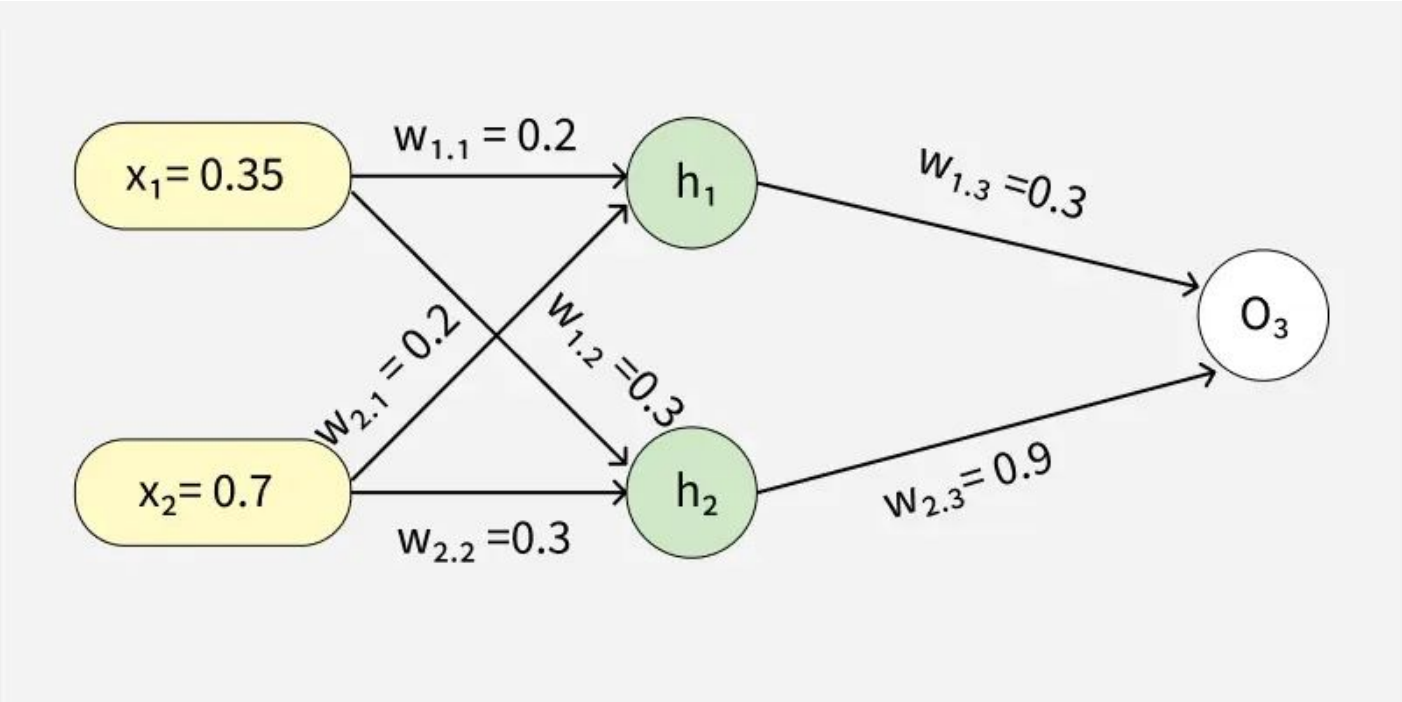
(b) Derive expressions for $\frac{\partial L(\boldsymbol{w})}{\partial o_1^{(l)}}$ and $\frac{\partial L(\boldsymbol{w})}{\partial w_1^{(l)}}$ for all hidden layers $l = 1, \ldots, 151$ in the network.

(Hint: Use the chain rule, for example, $\frac{\partial o_1^{(l+1)}}{\partial o_1^{(l)}} = \frac{\partial o_1^{(l+1)}}{\partial a_1^{(l+1)}} \frac{\partial a_1^{(l+1)}}{\partial o_1^{(l)}}$.)

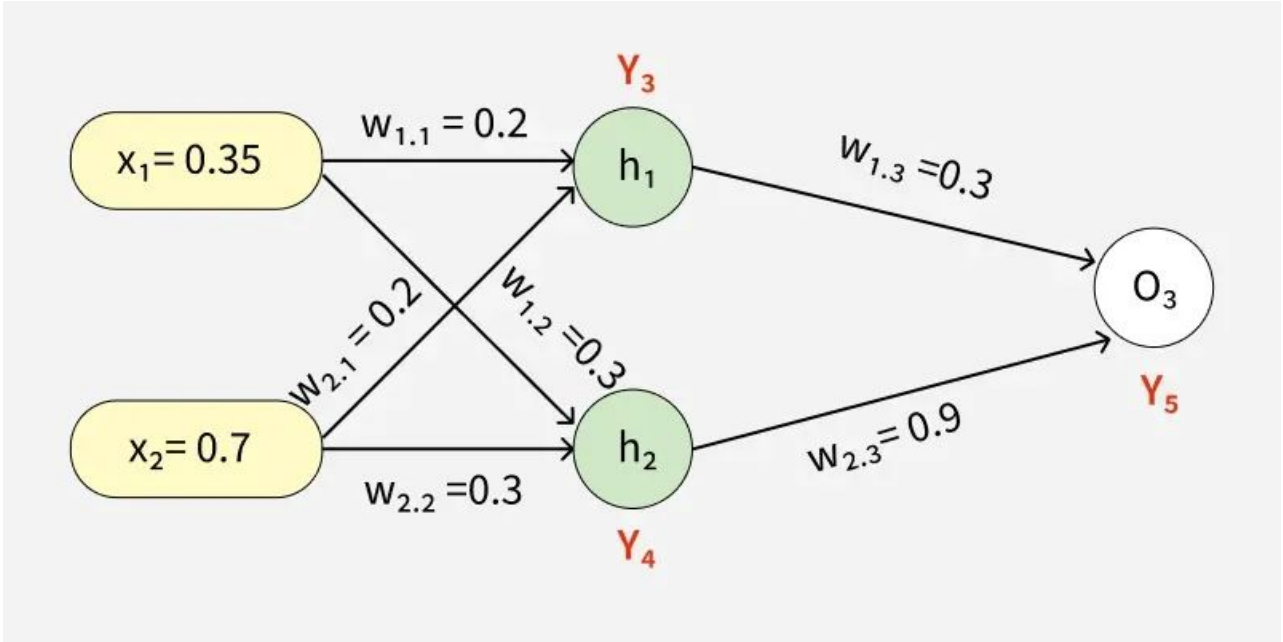
(c) Suppose we use the sigmoid activation $h(a) = \frac{1}{1+e^{-a}}$ for all nodes in this neural network. The gradient of $h(a)$ is $h'(a) = h(a)(1 - h(a))$. Using your expression from the previous part, describe what the gradients $\frac{\partial L(\boldsymbol{w})}{\partial w_1^{(l)}}$ for the weights in the earlier layers of the network would be like, and explain why (earlier layers refers to small l). (Hint: You don't need to actually use the expression for the sigmoid activation, just think about the range of values the gradient of the activation takes. The range of $h(a)$ is $[0, 1]$. What is the range of $h'(a)$?)

(d) Suppose we use ReLU as the activation function for all nodes, i.e. $h(a) = \max(0, a)$. When would the gradients $\frac{\partial L(\boldsymbol{w})}{\partial w_1^{(l)}}$ for the weights for some layer of the network be zero? If we continue with the ReLU activation function but modify the network to be fully connected for all layers (i.e. add all cross-connections by connecting x_1 and the red node and so on), when would the gradients of the weights be zero? Describing one case where the gradient is zero for each of these questions is enough.

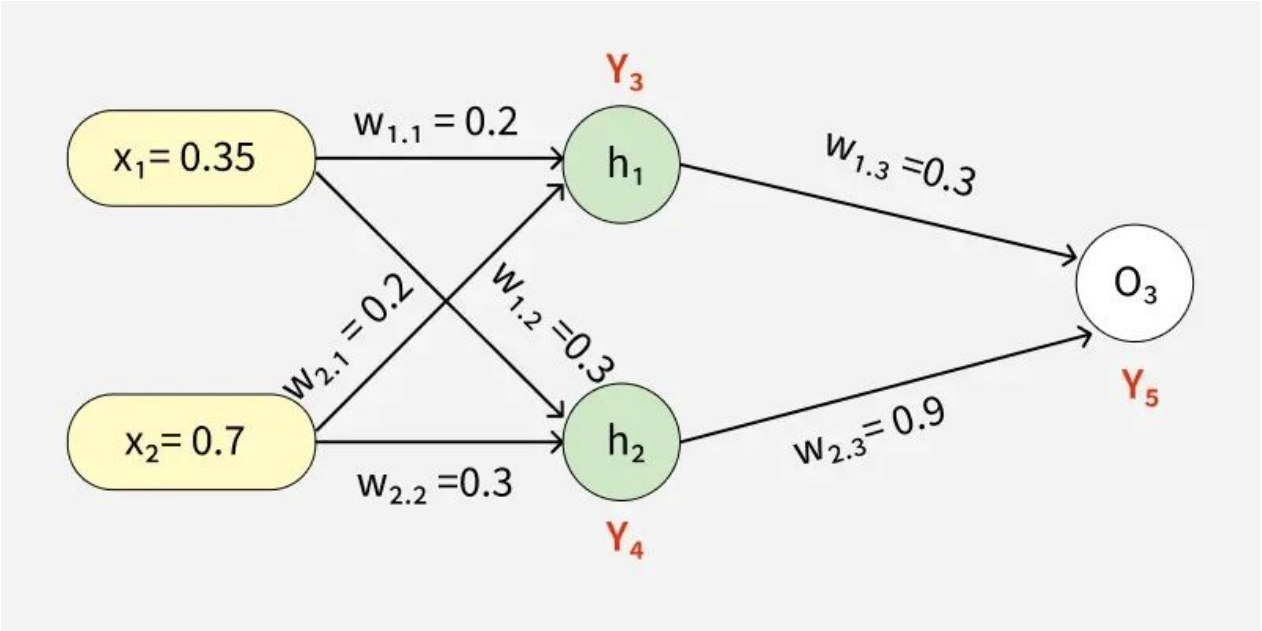
A Simple NN



Forward Pass



Backpropagation



Weight Update