

1 Conceptual Questions

1. The update rule for weights in a single-layer, multi-class neural network with cross-entropy loss is defined as follows:

Let w_{ij} be the weight associating the i th input feature x_i with the j th class. Let c be the index of the correct class for a given input (the *label*). The loss and its derivatives are then:

$$L = -\log(P_c), \quad \frac{\partial L}{\partial w_{ij}} = \begin{cases} (P_j - 1)x_i, & j = c \\ P_j x_i, & j \neq c \end{cases}$$

We use these partials with a learning rate α to descend along the gradient:

$$w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}$$

Derive the above rules from the original definition of cross-entropy loss.

Solution: We begin by expanding the definition of cross-entropy loss considering a situation where $j \neq c$.

Cross-entropy is defined as $L = -\sum_j y_j \log(P_j)$, where y_j is the true probability of class j and P_j is the predicted probability of class j .

In our case, y is a one-hot vector with $y_c = 1$ and $y_j = 0$ for $j \neq c$. Thus, the loss simplifies to $L = -\log(P_c)$.

Assuming we use softmax as our activation function, the logit for class j is $z_j = \mathbf{w}_j \cdot \mathbf{x}$. The probability of class j is then

$$P_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Now we can take the derivative with respect to the logit

$$\frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial P_c} \frac{\partial P_c}{\partial z_j}$$

$$\frac{\partial L}{\partial z_j} = -\frac{1}{P_c} \frac{\partial P_c}{\partial z_j}$$

$$\frac{\partial L}{\partial z_j} = -\frac{1}{P_c} \left(\frac{\partial}{\partial z_j} \left(\frac{e^{z_c}}{\sum_k e^{z_k}} \right) \right)$$

Which we now split into the cases where $j = c$ and $j \neq c$.

When $j = c$, we have

$$\frac{\partial P_c}{\partial z_c} = \frac{e^{z_c} \sum_k e^{z_k} - e^{z_c} e^{z_c}}{(\sum_k e^{z_k})^2} = P_c(1 - P_c)$$

and when $j \neq c$, we have

$$\frac{\partial P_c}{\partial z_j} = \frac{-e^{z_c} e^{z_j}}{(\sum_k e^{z_k})^2} = -P_c P_j$$

Giving us our original rules for the derivative of the loss with respect to the logit. By definition, we can use these to descend the gradient as stated.

2. In classification problems, we assign a likelihood probability to each class and use a loss function that outputs a loss based on this probability. Can you use MSE loss for classification tasks? Why or why not? Why is cross-entropy loss most commonly used for classification? (3-5 sentences)

Hint: Think about how each loss function is shaped, the inputs they take, and their range.

Solution: MSE loss is, fundamentally, a loss function designed for *continuous* outputs. It measures the squared difference between the predicted and true values. In classification tasks, we are dealing with *discrete* outputs, where the output is a probability distribution over classes. MSE loss is not well-suited for classification tasks because it does not capture the probabilistic nature of the problem. Cross-entropy loss, on the other hand, is designed to measure the difference between two probability distributions, making it a more natural choice for classification tasks.

3. Gradient Descent

- (a) What is a gradient? How is it different from a partial derivative? How do they relate? (2-4 sentences)

Solution: The gradient is a vector that describes the direction of steepest ascent of a function. It is composed of the partial derivatives of the function with respect to each of its variables. The partial derivative of a function is the derivative of the function with respect to one of its variables, holding all other variables constant. The gradient is a generalization of derivative as a vector-valued function of the partial derivatives.

- (b) Consider the formula for updating our weights:

$$\Delta w = -\alpha \frac{\partial L}{\partial w}$$

Why do we negate this quantity? What purpose does this serve? (2-4 sentences)

Solution: Our goal with gradient descent is to *minimize* the loss function. The gradient points in the direction of steepest ascent, so we negate it to move in the direction of steepest *descent*. This allows us to iteratively update our weights to minimize the loss function.

- (c) During gradient descent, we calculate the partial derivative of loss L with respect to each weight w_{ij} . Why must we do this for every weight? Why can't we do this for some weights? (1-3 sentences)

Solution: We reasonably *could* compute the partial derivative of the loss with respect to a subset of the weights, however each weight contributes to each output. If we want to minimize the loss function, we must consider the effect of each weight on the output. Thus, we must compute the partial derivative of the loss with respect to each weight.

- (d) In practice, most operations during gradient descent are vectorized. Why do we do this? Why might this make it beneficial to train the model on a GPU? (1-2 sentences)

Solution: Vectorizing operations allows us to parallelize otherwise linear computations, which is especially vital in neural networks that may have millions of weights. Training on a GPU is beneficial because GPUs are optimized for parallel computation, allowing us to perform these operations much faster than on a CPU, which has fewer cores and ALUs.

(e) Consider the following plot of a loss function for some neural network:

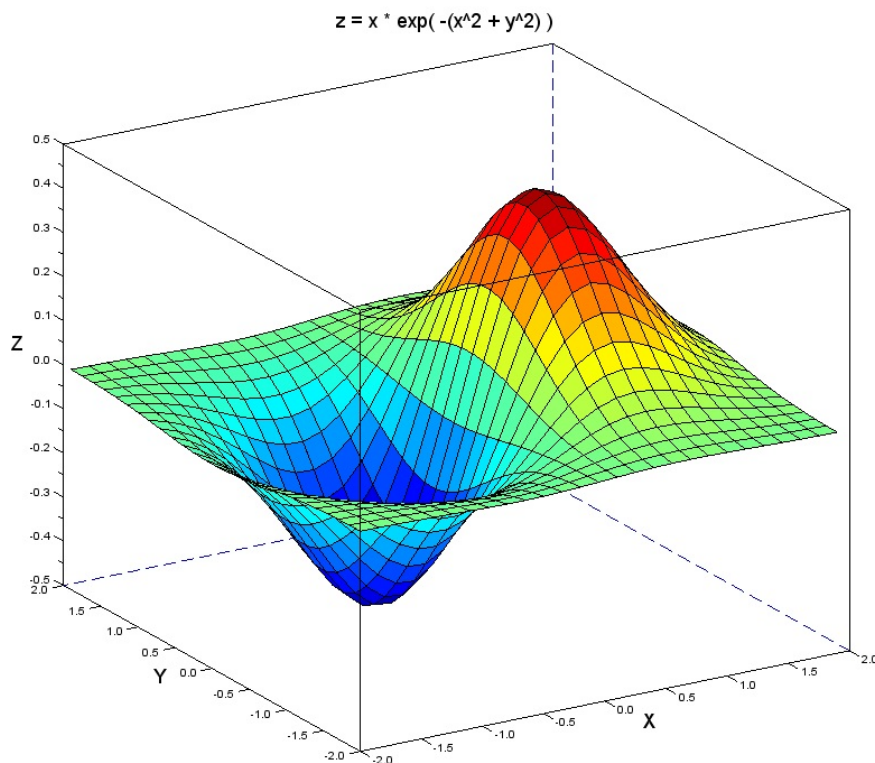


Figure 1: Loss Manifold Visualization

Where should gradient descent end up on the graph? How many weights does this model have? If our model starts training at $(-2, 2, 0)$, will the loss function ever reach the absolute minimum? Why? Assume the loss function at this point is *perfectly flat*. (3-5 sentences)

Solution:

- (a) Gradient descent should end up at the minimum of the graph, namely at the large hole around 0.7 on x and y.
- (b) Assuming the model has no hidden layers, the model has 2 weights corresponding to the two independent variables in the graph.
- (c) If the loss function is perfectly flat at $(-2, 2, 0)$, the model will not reach the absolute minimum. This is because the gradient at this point is zero, giving no information on how to update the weights. The model will be stuck at this point indefinitely. This problem with flat gradients is called the *vanishing gradient problem*.

4. We have previously worked on single-layer linear regression using one linear function:

$$\mathbf{x} \mapsto \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1$$

mapping from \mathbb{R}^s to \mathbb{R}^t . For many real-world scenarios, we actually need multiple layers to model more complex relationships.

- (a) Calculate the result after we stack another linear function

$$\mathbf{x} \mapsto \mathbf{W}_2 \mathbf{x} + \mathbf{b}_2$$

mapping from \mathbb{R}^t to \mathbb{R}^u right after the first one.

Solution: We can represent the composition of the two linear functions as

$$\mathbf{x} \mapsto \mathbf{W}_2(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Expanding this, we get

$$\mathbf{W}_2 \mathbf{W}_1 \mathbf{x} + \mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2$$

Thus, the result of stacking the two linear functions is

$$\mathbf{W}_2 \mathbf{W}_1 \mathbf{x} + \mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2$$

- (b) What is the shape of $\mathbf{W}_1, \mathbf{b}_1$ and $\mathbf{W}_2, \mathbf{b}_2$? Explain your reasoning.

Solution: The weights of an MLP layer are a matrix, while biases are column vectors. Following from this, if the first layer maps from \mathbb{R}^s to \mathbb{R}^t , then \mathbf{W}_1 is of size (t, s) and \mathbf{b}_1 is of size $(t, 1)$. Similarly, if the second layer maps from \mathbb{R}^t to \mathbb{R}^u , then \mathbf{W}_2 is of size (u, t) and \mathbf{b}_2 is of size $(u, 1)$.

- (c) Does the composition of the two linear functions offer an improvement over a single linear function? Explain your answer (2-4 sentences).

Solution: The composition of two linear functions does not offer an improvement over a single linear function, as the composition of linear functions in an ideal mathematical framework reduces to a single linear function. This is why we apply nonlinear activation functions to the output of each layer in a neural network in order to achieve higher expressiveness and describe complex systems. If we feel like being pedantic, we could say that the composition of two linear functions in a computationally imperfect system does offer more expressiveness due to IEEE error, but I prefer to leave that noise to the **PHDs**.