Neural Networks hw1

Dylan Satow

October 2025

1 Written Problems

1.1 Hard-Coding Neural Networks

The goal is to create a neural network that outputs 1 if the input sequence (x_1, x_2, x_3, x_4) is a superincreasing sequence, and 0 otherwise. A sequence is superincreasing if every element is greater than the sum of its predecessors. For a 4-element sequence, this translates to the following three conditions being true simultaneously:

1. $x_2 > x_1$
2. $x_3 > x_1 + x_2$
3. $x_4 > x_1 + x_2 + x_3$

We can design the network to check each of these conditions in one of the three hidden units, and then use the output unit to perform a logical AND on the results from the hidden units.

Activation Functions For both the hidden layer (ϕ_1) and the output layer (ϕ_2), we will use the hard threshold activation function, defined in the assignment as:

$$\phi(z) = I(z \geq 0) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Hidden Layer Each hidden unit h_k will compute $h_k = \phi_1(\mathbf{w}_k^{(1)} \cdot \mathbf{x} + b_k^{(1)})$. We want $h_k = 1$ if the k^{th} condition is met, and $h_k = 0$ otherwise.

- For h_1 (checks $x_2 > x_1 \iff x_2 - x_1 > 0$): We need the input to the activation, z_1, to be ≥ 0 if and only if this condition holds. We can set $z_1 = x_2 - x_1$. This gives the weights $\mathbf{w}_1^{(1)} = [-1, 1, 0, 0]$ and bias $b_1^{(1)} = 0$.
- For h_2 (checks $x_3 > x_1 + x_2 \iff x_3 - x_1 - x_2 > 0$): We set $z_2 = x_3 - x_1 - x_2$. This gives the weights $\mathbf{w}_2^{(1)} = [-1, -1, 1, 0]$ and bias $b_2^{(1)} = 0$.

- For h_3 (checks $x_4 > x_1 + x_2 + x_3 \iff x_4 - x_1 - x_2 - x_3 > 0$): We set $z_3 = x_4 - x_1 - x_2 - x_3$. This gives the weights $\mathbf{w}_3^{(1)} = [-1, -1, -1, 1]$ and bias $b_3^{(1)} = 0$.

Output Layer The output unit y computes $y = \phi_2(\mathbf{w}^{(2)} \cdot \mathbf{h} + b^{(2)})$. We need $y = 1$ if and only if $h_1 = 1$, $h_2 = 1$, and $h_3 = 1$. This is a logical AND operation. The input to the output activation is $z_{out} = w_1^{(2)} h_1 + w_2^{(2)} h_2 + w_3^{(2)} h_3 + b^{(2)}$. Let's set the weights $\mathbf{w}^{(2)} = [1, 1, 1]$. Then $z_{out} = h_1 + h_2 + h_3 + b^{(2)}$.

- If all hidden units are 1, their sum is 3. We need $z_{out} = 3 + b^{(2)} \geq 0$.
- If any hidden unit is 0, the maximum sum is 2. We need $z_{out} = 2 + b^{(2)} < 0$.

From these two conditions, we need $b^{(2)} \geq -3$ and $b^{(2)} < -2$. A value such as $b^{(2)} = -2.5$ would work. For an integer solution, if we choose $b^{(2)} = -3$, the first condition becomes $3 - 3 = 0 \geq 0$ (output 1) and the second becomes $2 - 3 = -1 < 0$ (output 0). This works perfectly.

Final Parameters

- **Activation Functions**: $\phi_1(z) = \phi_2(z) = I(z \geq 0)$.
- **Hidden Layer Weights**:

$$W^{(1)} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

- **Hidden Layer Biases**:

$$\mathbf{b}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- **Output Layer Weights**:

$$\mathbf{w}^{(2)} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

- **Output Layer Bias**:

$$b^{(2)} = -3$$

1.2 Backpropagation

Computation Graph

The computation graph describes the flow of data from inputs to the final loss. The nodes in the graph are the variables involved in the computation, and the directed edges represent the functions that transform one variable into another.

- **Inputs**: x (features), t (true labels), and all weight matrices and biases $(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, W^{(g)}, b^{(g)}, W^{(3)}, W^{(4)})$.
- **Graph Structure**:
  1. $x, W^{(1)}, b^{(1)} \to a_1 = W^{(1)}x + b^{(1)}$
  2. $x, W^{(2)}, b^{(2)} \to a_2 = W^{(2)}x + b^{(2)}$
  3. $a_1 \to c_1 = \sigma(a_1)$ (Sigmoid activation, where $\sigma(a) = 1/(1 + e^{-a})$)
  4. $a_1, a_2 \to u = a_1 + a_2$
  5. $u \to c_2 = \text{ReLU}(u)$
  6. $c_1, c_2, W^{(g)}, b^{(g)} \to g = W^{(g)}[c_1; c_2] + b^{(g)}$ (Concatenation)
  7. $g, x, W^{(3)}, W^{(4)} \to y = W^{(3)}g + W^{(4)}x$
  8. $y \to y' = \text{softmax}(y)$
  9. $y', t \to S = \sum_k I(t_k = 1) \log(y'_k)$
  10. $S \to J = -S$

The graph shows multiple paths from the input x to the loss J: one through a_1, one through a_2, and a direct one to y. The total gradient $\frac{\partial J}{\partial x}$ will be the sum of the gradients from these three paths.

Backward Pass

We derive the backpropagation equations by applying the chain rule, starting from the output J and moving backward through the graph. We compute the gradient of the cost J with respect to each variable. Let $\delta_v = \frac{\partial J}{\partial v}$ denote the gradient of the loss with respect to a variable v.

1. **Loss**: $J = -S$
$$\frac{\partial J}{\partial S} = -1$$
2. **Score**: $S = \sum_k t_k \log(y'_k)$ (assuming t is one-hot, so $I(t_k = 1)$ is t_k) The gradient of the cross-entropy loss with respect to the softmax input y is a standard result:
$$\delta_y = \frac{\partial J}{\partial y} = y' - t$$
3. **Output Logits**: $y = W^{(3)}g + W^{(4)}x$. From here, we can compute gradients for $g, x, W^{(3)}, W^{(4)}$.
$$\delta_g = \frac{\partial J}{\partial g} = (W^{(3)})^T \delta_y$$
$$\delta_x^{(\text{from y})} = (W^{(4)})^T \delta_y$$
The gradients for the weights are:
$$\delta_{W^{(3)}} = \delta_y g^T$$
$$\delta_{W^{(4)}} = \delta_y x^T$$

4. **Hidden Layer g**: $g = W^{(g)}[c_1; c_2] + b^{(g)}$. Let $c_{concat} = [c_1; c_2]$.

$$\delta_{c_{concat}} = \frac{\partial J}{\partial c_{concat}} = (W^{(g)})^T \delta_g$$

This gradient is then split for c_1 and c_2:

$$\delta_{c_1} = (\delta_{c_{concat}})_{1:d_1} \quad \text{and} \quad \delta_{c_2} = (\delta_{c_{concat}})_{d_1+1:end}$$

(where d_1 is the dimension of c_1). The gradients for the weights are:

$$\delta_{W^{(g)}} = \delta_g c_{concat}^T$$

$$\delta_{b^{(g)}} = \delta_g$$

5. **ReLU Activation**: $c_2 = \text{ReLU}(u)$

$$\delta_u = \frac{\partial J}{\partial u} = \delta_{c_2} \odot I(u > 0) \quad (\odot \text{ is element-wise product})$$

6. **Sum**: $u = a_1 + a_2$

$$\delta_{a_1}^{(\text{from u})} = \delta_u \quad \text{and} \quad \delta_{a_2} = \delta_u$$

7. **Sigmoid Activation**: $c_1 = \sigma(a_1)$

$$\delta_{a_1}^{(\text{from c1})} = \delta_{c_1} \odot (\sigma(a_1)(1 - \sigma(a_1))) = \delta_{c_1} \odot (c_1(1 - c_1))$$

8. **Total gradients for** a_1, a_2:

$$\delta_{a_1} = \delta_{a_1}^{(\text{from c1})} + \delta_{a_1}^{(\text{from u})} = (\delta_{c_1} \odot (c_1(1 - c_1))) + \delta_u$$

$$\delta_{a_2} = \delta_u$$

9. **Linear Layers** a_1, a_2: For $a_1 = W^{(1)}x + b^{(1)}$:

$$\delta_x^{(\text{from a1})} = (W^{(1)})^T \delta_{a_1}$$

$$\delta_{W^{(1)}} = \delta_{a_1} x^T$$

$$\delta_{b^{(1)}} = \delta_{a_1}$$

For $a_2 = W^{(2)}x + b^{(2)}$:

$$\delta_x^{(\text{from a2})} = (W^{(2)})^T \delta_{a_2}$$

$$\delta_{W^{(2)}} = \delta_{a_2} x^T$$

$$\delta_{b^{(2)}} = \delta_{a_2}$$

10. **Total Gradient for x**: The final gradient for x is the sum from all paths:

$$\delta_x = \frac{\partial J}{\partial x} = \delta_x^{(\text{from y})} + \delta_x^{(\text{from a1})} + \delta_x^{(\text{from a2})}$$

$$\frac{\partial J}{\partial x} = (W^{(4)})^T (y' - t) + (W^{(1)})^T \delta_{a_1} + (W^{(2)})^T \delta_{a_2}$$

1.3 Automatic Differentiation

Compute Hessian

The function is given by $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{v}\mathbf{v}^T\mathbf{x}$. Let's define a matrix $A = \mathbf{v}\mathbf{v}^T$. The function becomes a standard quadratic form: $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x}$.

The gradient of $\mathcal{L}$ with respect to $\mathbf{x}$ is given by $\mathbf{g} = \nabla_{\mathbf{x}}\mathcal{L} = \frac{1}{2}(A+A^T)\mathbf{x}$. The matrix $A = \mathbf{v}\mathbf{v}^T$ is symmetric, since $A^T = (\mathbf{v}\mathbf{v}^T)^T = (\mathbf{v}^T)^T\mathbf{v}^T = \mathbf{v}\mathbf{v}^T = A$. Therefore, the gradient simplifies to $\mathbf{g} = \frac{1}{2}(A + A)\mathbf{x} = A\mathbf{x}$.

The Hessian, H, is the Jacobian of the gradient $\mathbf{g}$ with respect to $\mathbf{x}$:

$$H = \frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \frac{\partial (A\mathbf{x})}{\partial \mathbf{x}} = A$$

So, the Hessian is simply $H = \mathbf{v}\mathbf{v}^T$. Note that the Hessian is constant and does not depend on the value of $\mathbf{x}$.

For $n = 3$ and $\mathbf{v} = [3, 1, 4]^T$, we compute the outer product:

$$H = \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix} \begin{bmatrix} 3 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 3 \cdot 3 & 3 \cdot 1 & 3 \cdot 4 \\ 1 \cdot 3 & 1 \cdot 1 & 1 \cdot 4 \\ 4 \cdot 3 & 4 \cdot 1 & 4 \cdot 4 \end{bmatrix}$$

The resulting Hessian matrix is:

$$H = \begin{bmatrix} 9 & 3 & 12 \\ 3 & 1 & 4 \\ 12 & 4 & 16 \end{bmatrix}$$

Computation Cost

Scalar Multiplications: To compute the Hessian $H = \mathbf{v}\mathbf{v}^T$, we need to compute each of its n^2 elements. Each element H_{ij} is the product of two scalars: $v_i \cdot v_j$. This requires one multiplication. Since there are n^2 elements in the $n \times n$ matrix, the total number of scalar multiplications is n^2. The computational cost is $O(n^2)$.

Memory Cost: The memory cost is determined by the size of the final matrix we need to store. The Hessian H is an $n \times n$ matrix. Storing this matrix requires space for n^2 scalar values. The memory cost is $O(n^2)$.

1.3.1 Vector-Hessian Products

Numerical Computation We are asked to compute $\mathbf{z} = H\mathbf{y}$ for $H = \mathbf{v}\mathbf{v}^T$, with $\mathbf{v} = [4, 1, 2]^T$ and $\mathbf{y} = [2, 2, 1]^T$.

Reverse-Mode: In this approach, we group operations as $\mathbf{z} = \mathbf{v}(\mathbf{v}^T\mathbf{y})$.

1. First, compute the scalar $M = \mathbf{v}^T\mathbf{y}$:

$$M = \begin{bmatrix}4 & 1 & 2\end{bmatrix}\begin{bmatrix}2\\2\\1\end{bmatrix} = (4)(2) + (1)(2) + (2)(1) = 8 + 2 + 2 = 12$$

2. Then, compute the final vector $\mathbf{z} = \mathbf{v}M$:

$$\mathbf{z} = \begin{bmatrix}4\\1\\2\end{bmatrix}(12) = \begin{bmatrix}48\\12\\24\end{bmatrix}$$

Forward-Mode: In this approach, we group operations as $\mathbf{z} = (\mathbf{v}\mathbf{v}^T)\mathbf{y}$.

1. First, compute the full Hessian matrix $H = \mathbf{v}\mathbf{v}^T$:

$$H = \begin{bmatrix}4\\1\\2\end{bmatrix}\begin{bmatrix}4 & 1 & 2\end{bmatrix} = \begin{bmatrix}16 & 4 & 8\\4 & 1 & 2\\8 & 2 & 4\end{bmatrix}$$

2. Then, compute the final vector $\mathbf{z} = H\mathbf{y}$:

$$\mathbf{z} = \begin{bmatrix}16 & 4 & 8\\4 & 1 & 2\\8 & 2 & 4\end{bmatrix}\begin{bmatrix}2\\2\\1\end{bmatrix} = \begin{bmatrix}16(2) + 4(2) + 8(1)\\4(2) + 1(2) + 2(1)\\8(2) + 2(2) + 4(1)\end{bmatrix} = \begin{bmatrix}32 + 8 + 8\\8 + 2 + 2\\16 + 4 + 4\end{bmatrix} = \begin{bmatrix}48\\12\\24\end{bmatrix}$$

Both methods yield the same result: $\mathbf{z}^T = [48, 12, 24]$.

Computational Cost Analysis We analyze the cost of computing $Z = H\mathbf{y}_1\mathbf{y}_2^T$, interpreted as $Z = (\mathbf{v}\mathbf{v}^T)(\mathbf{y}_1\mathbf{y}_2^T)$. The resulting matrix Z has dimensions $n \times m$. The relative efficiency of each mode depends on the shape of Z—that is, whether it is "tall" ($n > m$) or "wide" ($m > n$).

Forward-Mode Cost: This approach computes the full $n \times n$ Hessian matrix H first: $Z = (\mathbf{v}\mathbf{v}^T)(\mathbf{y}_1\mathbf{y}_2^T)$.

1. Compute $H = \mathbf{v}\mathbf{v}^T$: n^2 multiplications.
2. Compute $Y = \mathbf{y}_1\mathbf{y}_2^T$: nm multiplications.
3. Compute $Z = HY$: n^2m multiplications.

The total multiplications are $n^2 + nm + n^2m = n^2(1 + m) + nm$. This approach is burdened by the $O(n^2)$ cost of creating and using the explicit Hessian, which is expensive if n is large.

Reverse-Mode Cost: This approach avoids forming the full Hessian by using associativity: $Z = \mathbf{v}(\mathbf{v}^T(\mathbf{y}_1\mathbf{y}_2^T))$.

1. Compute $Y = \mathbf{y}_1\mathbf{y}_2^T$: nm multiplications.
2. Compute the row vector $\mathbf{r} = \mathbf{v}^T Y$: nm multiplications.
3. Compute the final matrix $Z = \mathbf{v}\mathbf{r}$: nm multiplications.

The total multiplications are $3nm$. This approach cleverly avoids the $O(n^2)$ costs.

Comparison: Forward-mode is preferable when $n^2(1+m) + nm < 3nm$, which simplifies to $n < m(2-n)$. This inequality reveals how the shape of the $n \times m$ output matrix Z dictates the best strategy:

- If $n \geq 2$, the term $(2-n)$ is zero or negative, making the inequality impossible to satisfy for positive n, m. This covers all cases where Z is square or "tall" ($n \geq m$). For these shapes, **reverse-mode** is always superior because it avoids the large $O(n^2)$ cost of building the Hessian.
- If $n = 1$, the inequality becomes $1 < m$. In this scenario, the Hessian is a trivial 1×1 scalar. **Forward-mode** becomes more efficient when $m > 1$. This corresponds to producing a very "wide" matrix Z (e.g., 1×100), where the negligible cost of the 1×1 Hessian makes the forward approach more economical.

2 Programming Problems

2.1 GLoVE Word Representations

2.1.1 GLoVE Parameter Count (2.1.1)

The problem states that for each word in a vocabulary of size V, the GLoVE model learns two embedding vectors and two scalar biases. For each word, the parameters are:

- A d-dimensional embedding vector $\mathbf{w}_i$, contributing d parameters.
- A second d-dimensional embedding vector $\tilde{\mathbf{w}}_i$, contributing d parameters.
- A scalar bias b_i, contributing 1 parameter.
- A second scalar bias $\tilde{b}_i$, contributing 1 parameter.

The total number of parameters for a single word is $d + d + 1 + 1 = 2d + 2$. Since there are V words in the vocabulary, the total number of parameters in the model is $V \times (2d + 2)$.

2.1.2 Expression for Gradient (2.1.2)

The simplified GLoVE loss function is given by:

$$L = \sum_{i=1}^{V} \sum_{j=1}^{V} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2$$

To find the gradients, we differentiate this loss function with respect to a specific parameter vector $\mathbf{w}_k$ and a specific bias b_k.

Gradient with respect to $\mathbf{w}_i$: The parameter vector $\mathbf{w}_i$ only appears in the loss function terms where the first summation index is equal to i. Therefore, we only need to sum over the second index, j.

$$\frac{\partial L}{\partial \mathbf{w}_i} = \frac{\partial}{\partial \mathbf{w}_i} \sum_{j=1}^{V} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2$$

Using the chain rule, where $\frac{d}{dx} u^2 = 2u \frac{du}{dx}$:

$$\frac{\partial L}{\partial \mathbf{w}_i} = \sum_{j=1}^{V} 2 \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right) \cdot \frac{\partial}{\partial \mathbf{w}_i} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)$$

The derivative of the inner term with respect to $\mathbf{w}_i$ is $\tilde{\mathbf{w}}_j$. This gives the final expression:

$$\frac{\partial L}{\partial \mathbf{w}_i} = 2 \sum_{j=1}^{V} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right) \tilde{\mathbf{w}}_j$$

Gradient with respect to b_i: Similarly, the bias b_i only appears in terms where the first index is i.

$$\frac{\partial L}{\partial b_i} = \frac{\partial}{\partial b_i} \sum_{j=1}^{V} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2$$

Applying the chain rule:

$$\frac{\partial L}{\partial b_i} = \sum_{j=1}^{V} 2 \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right) \cdot \frac{\partial}{\partial b_i} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)$$

The derivative of the inner term with respect to the scalar b_i is 1. This gives the final expression:

$$\frac{\partial L}{\partial b_i} = 2 \sum_{j=1}^{V} \left(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)$$