Team ML_ShibaInu

Group Members:

1. 2211649 - Trần Đình Đăng Khoa
2. 2252938 - Phan Chí Vỹ
3. 2252734 - Nguyễn Đức Tâm
4. 2252338 - Huỳnh Kiệt Khải
5. 2252289 - Hà Tuấn Khang

Homework 3: Perceptron, Artificial Neural Network, Backpropagation

---

Murphy's Book 1

Exercise 13.1

Consider the following classification Multi-Layer Perceptron (MLP) with one hidden layer:

$$
\begin{aligned}
x &= \text{input} \in \mathbb{R}^D \\
z &= Wx + b_1 \in \mathbb{R}^K \\
h &= \text{ReLU}(z) \in \mathbb{R}^K \\
a &= Uh + b_2 \in \mathbb{R}^C \\
\mathcal{L} &= \text{CrossEntropy}(y, \text{Softmax}(a)) \in \mathbb{R}
\end{aligned}
$$

where $x \in \mathbb{R}^D$, $b_1 \in \mathbb{R}^K$, $W \in \mathbb{R}^{K \times D}$, $b_2 \in \mathbb{R}^C$, $U \in \mathbb{R}^{C \times K}$, where D is the size of the input, K is the number of hidden units, and C is the number of classes.

We aim to compute the gradients of the loss function with respect to the parameters and the input, which are given by:

$$
\begin{aligned}
\nabla_U \mathcal{L} &= \left[\frac{\partial \mathcal{L}}{\partial \mathbf{V}}\right]_{1,:} = \delta_1 h^T \in \mathbb{R}^{C \times K} \\
\nabla_{b_2} \mathcal{L} &= \left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}_2}\right)^\top = \delta_1 \in \mathbb{R}^C \\
\nabla_W \mathcal{L} &= \left[\frac{\partial \mathcal{L}}{\partial \mathbf{W}}\right]_{1,:} = \delta_2 x^T \in \mathbb{R}^{K \times D} \\
\nabla_{b_1} \mathcal{L} &= \left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}_1}\right)^\top = \delta_2 \in \mathbb{R}^K \\
\nabla_x \mathcal{L} &= \left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}}\right)^\top = W^T \delta_2 \in \mathbb{R}^D
\end{aligned}
$$

where the intermediate gradients are given by:

$$
\begin{aligned}
\delta_1 = \nabla_a \mathcal{L} &= \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}}\right)^\top = (p - y)^T \\
\delta_2 = \nabla_z \mathcal{L} &= \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}}\right)^\top = (U^T \delta_1) \odot H(z) \in \mathbb{R}^K
\end{aligned}
$$

where $H(z)$ is the Heaviside step function.

Solution

To compute δ_1, we define the loss as:

$$
\mathcal{L} = \text{CrossEntropyWithLogits}(y, a)
$$

The gradient with respect to a is:

$$
\delta_1 = \frac{\partial \mathcal{L}}{\partial a} = (p - y)^T
$$

where $p = S(a)$ is the softmax output.

Next, to compute δ_2:

$$
\begin{aligned}
\delta_2 = \frac{\partial \mathcal{L}}{\partial z} &= \frac{\partial \mathcal{L}}{\partial a} \frac{\partial a}{\partial h} \frac{\partial h}{\partial z} \\
&= \delta_1 U \frac{\partial h}{\partial z} \text{ since } a = Uh + b_2 \\
&= \delta_1 U \odot \text{ReLU}'(z) \\
&= \delta_1 U \odot H(h)
\end{aligned}
$$

where we use the fact that $h = \text{ReLU}(z)$, whose derivative is the Heaviside step function.

Now, computing the gradients with respect to the parameters:

$$\frac{\partial \mathcal{L}}{\partial U} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial U} = \delta_1 \frac{\partial (Uh + b_2)}{\partial U} = \delta_1 \frac{\partial (Uh)}{\partial U} = \delta_1 h^T$$
$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial b_2} = \delta_1$$
$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial W} = \delta_2 x^T$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b_1} = \delta_2$$
$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial x} = \delta_2 W$$

where $a = Uh + b_2$ and $z = Wx + b_1$

These results show how the backpropagation algorithm efficiently computes gradients for a one-layer MLP, enabling parameter updates using gradient descent.

$$a = Uh + b_2$$
$$z = Wx + b_1$$

We will compute the partial derivatives step by step.

Step 1. $\frac{\partial a}{\partial U} = h^\top$

The variable a is computed as:

$$a = Uh + b_2$$

Taking the derivative with respect to U:

$$\frac{\partial a}{\partial U} = \frac{\partial (Uh + b_2)}{\partial U}$$

Since Uh is a matrix-vector multiplication, we differentiate it element-wise:

$$\frac{\partial (Uh)}{\partial U} = h^\top$$

As b_2 is independent of U, its derivative is zero. Thus,

$$\frac{\partial a}{\partial U} = h^\top$$

Step 2. $\frac{\partial a}{\partial b_2} = 1$

Since b_2 is simply added element-wise to Uh, its derivative is:

$$\frac{\partial a}{\partial b_2} = I$$

For element-wise differentiation, this simplifies to:

$$\frac{\partial a}{\partial b_2} = 1$$

Step 3. $\frac{\partial z}{\partial W} = x^\top$

The variable z is given by:

$$z = Wx + b_1$$

Taking the derivative with respect to W:

$$\frac{\partial z}{\partial W} = \frac{\partial (Wx + b_1)}{\partial W}$$

Since Wx is a matrix-vector multiplication, differentiating element-wise gives:

$$\frac{\partial (Wx)}{\partial W} = x^\top$$

Step 4. $\frac{\partial z}{\partial b_1} = 1$

Since b_1 is added element-wise to Wx, its derivative is:

$$\frac{\partial z}{\partial b_1} = 1$$

Step 5. $\frac{\partial z}{\partial x} = W$

Differentiating $z = Wx + b_1$ with respect to x:

$$\frac{\partial z}{\partial x} = W$$

Since Wx is a linear transformation, its derivative with respect to x is simply

$$W$$

Mitchell's ML book

Exercise 4.4

Implement the delta training rule for a two-input linear unit. Train it to fit the target concept $(-2 + x_1 + 2x_2 > 0)$. Plot the error E as a function of the number of training iterations.
Plot the decision surface after $5, 10, 50, 100, \ldots$, iterations.

a. Try this using various constant values for η and using a decaying learning rate of $\frac{\eta_0}{i}$ for the i-th iteration. Which works better?
b. Try incremental and batch learning. Which converges more quickly? Consider both number of weight updates and total execution time.

Problem

We have a two-input linear unit with inputs (x_1, x_2) and want to fit the target concept

$$-2 + x_1 + 2x_2 > 0$$

So, our target classification rule is:

$$\text{Class } 1 \quad \text{if } -2 + x_1 + 2x_2 \geq 0, \quad \text{Class } -1 \quad \text{otherwise.}$$

We will:

1. Implement the ***delta training rule*** (gradient descent) for a linear unit.

$$\hat{y}(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2$$

2. Plot the ***error*** E as a function of the number of training iterations.
3. Plot the ***decision surface*** after $5, 10, 50, 100, \ldots,$ iterations.
4. Use various constant values for the learning rate η and a decaying learning rate $\eta = \frac{\eta_0}{i}$, where i is the iteration index.
5. Compare ***incremental*** and ***batch*** learning, noting which converges more quickly.

Solution

1. Model Definition

Let $\mathbf{w} = (w_0, w_1, w_2)$ be the parameter vector. For each input $\mathbf{x} = (x_1, x_2)$, the linear unit produces the output

$$\hat{y}(\mathbf{x}) \;=\; w_0 + w_1 x_1 + w_2 x_2$$

We label each training point (x_1, x_2) by

$$t = \begin{cases} 1, & \text{if } -2 + x_1 + 2x_2 \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

We train by minimizing the ***Mean Squared Error*** (MSE):

$$E = \frac{1}{2N} \sum_{n=1}^{N} \left(t_n - \hat{y}_n\right)^2$$

where N is the number of training points.

2. Delta Rule Updates

The ***delta rule*** for weight update in gradient descent is:

$$w_i \leftarrow w_i + \eta \sum_{n=1}^{N} (t_n - \hat{y}_n) x_{n,i}$$

where $x_{n,0} = 1$ for the bias term. We can update either:

- **Batch mode:** Accumulate the gradient over all samples, then update once per epoch.
- **Incremental mode:** Update the weights after each individual sample.
  to satisfy part **b**.

We will also use:

- **Constant learning rate:** $\eta = \text{constant}$.
- **Decaying learning rate:** $\eta_i = \dfrac{\eta_0}{i}$, where i is the iteration or epoch index.
  to satisfy part **a**.

3. Generating Training Data

We create a synthetic dataset of N points (x_1, x_2), for instance sampling each coordinate uniformly in $[-2, 2]$. The label t_n is assigned by the target rule:

$$t_n = \text{sign}(-2 + x_{n,1} + 2x_{n,2}) \in \{+1, -1\}$$

4. Python Code

```
import numpy as np

# Generate random data
N = 100
X_data = np.random.uniform(-2, 2, size=(N, 2))

def target_label(x1, x2):
    return 1 if (-2 + x1 + 2*x2) >= 0 else -1

y_data = np.array([target_label(x[0], x[1]) for x in X_data])

def linear_output(w, x):
    """Compute w0 + w1*x1 + w2*x2."""
    return w[0] + w[1]*x[0] + w[2]*x[1]
```

```python
def mse_error(w, X, T):
    """Compute mean squared error on dataset."""
    N = len(X)
    errors = [(T[i] - linear_output(w, X[i]))**2 for i in range(N)]
    return 0.5 * np.mean(errors)

def train_delta_rule(X, T, eta=0.1, epochs=100, mode='batch', decay=False):
    """
    Train a linear unit using delta rule (gradient descent).
    mode = 'batch' or 'incremental'.
    decay: if True, eta_i = eta / epoch.
    """
    N = len(X)
    w = np.random.randn(3) * 0.01  # random init
    w_history = []
    E_history = []

    for epoch in range(1, epochs+1):
        if decay:
            eta_current = eta / epoch
        else:
            eta_current = eta

        if mode == 'batch':
            # Accumulate gradient over all samples
            grad = np.zeros(3)
            for i in range(N):
                y_hat = linear_output(w, X[i])
                error = T[i] - y_hat
                grad[0] += error * 1.0
                grad[1] += error * X[i, 0]
                grad[2] += error * X[i, 1]
            # Update once
            w += (eta_current / N) * grad

        elif mode == 'incremental':
            # Update after each sample
            for i in range(N):
                y_hat = linear_output(w, X[i])
                error = T[i] - y_hat
                w[0] += eta_current * error * 1.0
                w[1] += eta_current * error * X[i, 0]
                w[2] += eta_current * error * X[i, 1]

        # Record MSE after this epoch
```

```
            E_history.append(mse_error(w, X, T))
            w_history.append(w.copy())

    return w_history, E_history

# Example usage:
w_hist_batch_const, E_hist_batch_const = train_delta_rule(
    X_data, y_data, eta=0.1, epochs=100, mode='batch', decay=False
)
```

5. Plotting and Comparison

Error vs. Iterations: Plot E_{MSE} over epochs for:

1. Batch vs. Incremental
2. Constant vs. Decaying η

   Observe which setup converges faster and more stably.

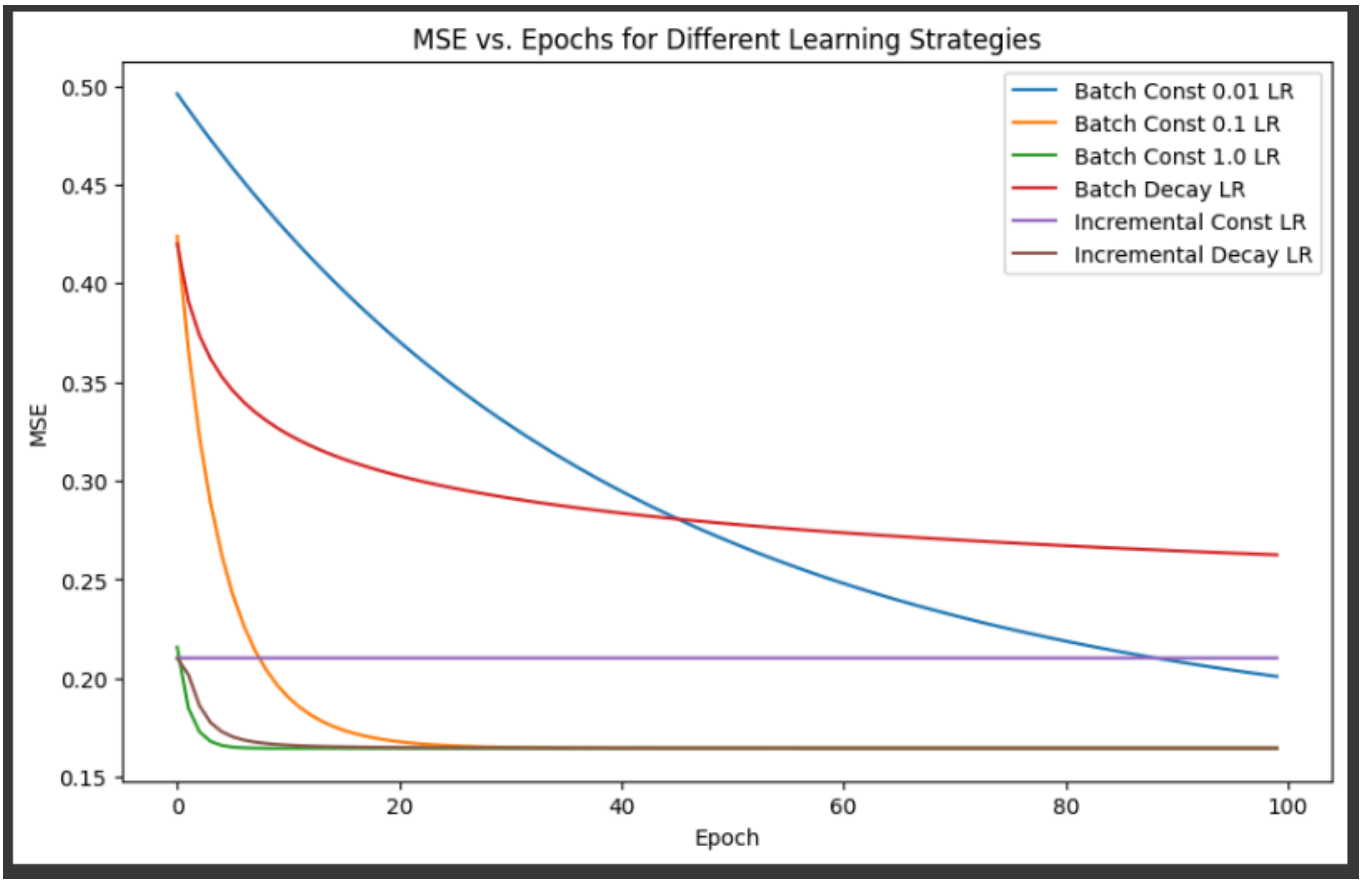


Figure 1: MSE vs. Epochs

Decision Boundary: At epoch k, the boundary is given by

$$w_0 + w_1 x_1 + w_2 x_2 = 0 \quad \Longrightarrow \quad x_2 = -\frac{w_0 + w_1 x_1}{w_2}$$

Plot this line at epochs 5, 10, 50, 100, etc. to see how quickly the classifier aligns with

$$2 + x_1 + 2x_2 = 0$$

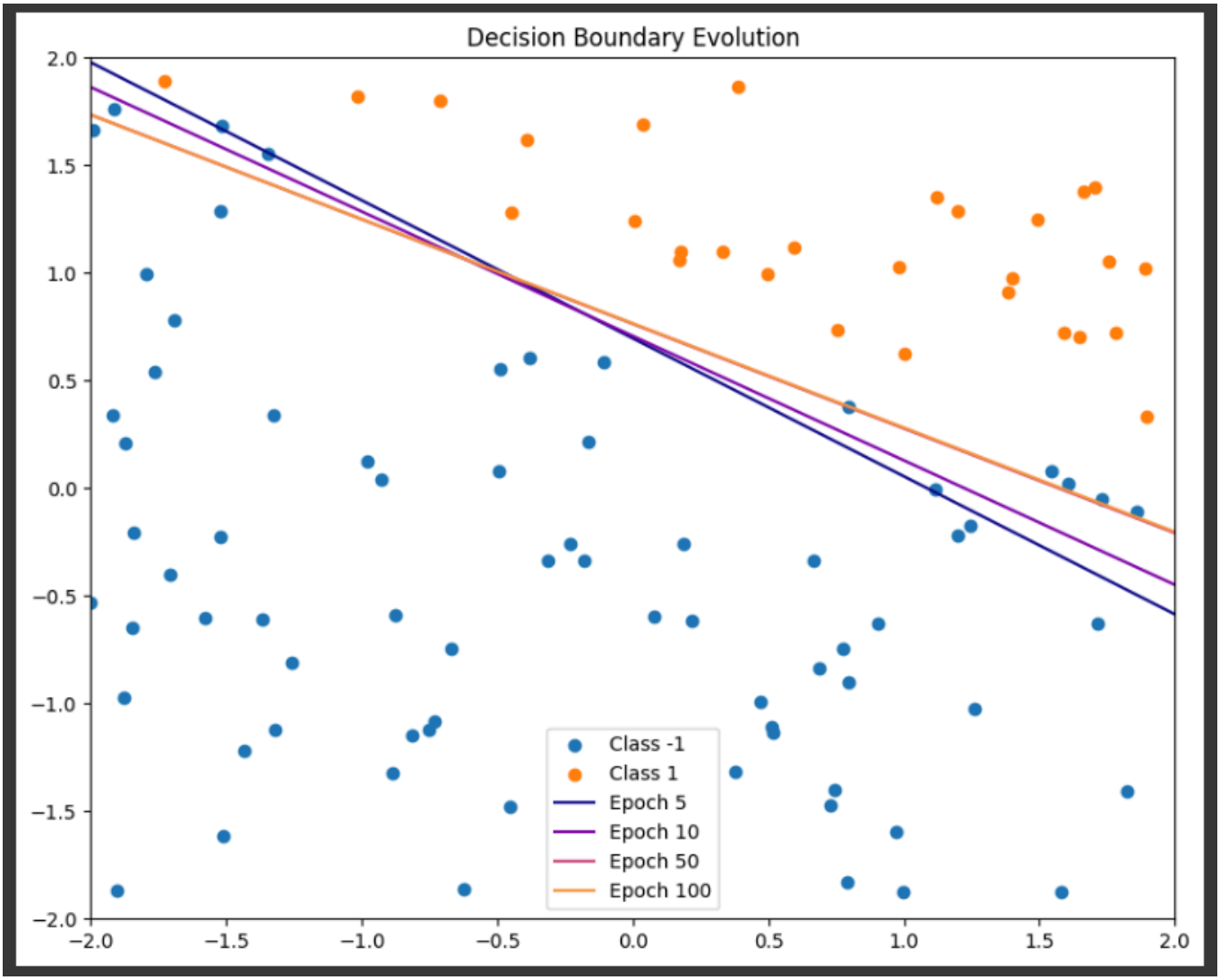


Figure 2: Decision Boundary (Batch Mode, $\eta = 0.1$)

6. Observations

- **Learning Rate:**
  - Here, when η is large, updates converge faster.
  - If η is too small, however, convergence is very slow.
  - A moderate value (e.g. $\eta = 0.1$) while converges not as quick as 1.0 but offer stability.
  - ***Decaying*** η can help stabilize learning over many epochs, but it can also make the learning process slower and harder to converge.
- **Batch vs. Incremental:**
  - **Incremental** often converges in fewer epochs, since it updates weights immediately after each sample.

- **Batch** is sometimes more stable (less noisy) and can be faster in practice if heavily vectorized.

- Decision Surface Plots: By epoch 50 or 100, the learned boundary closely matches

$$-2 + x_1 + 2x_2 = 0$$

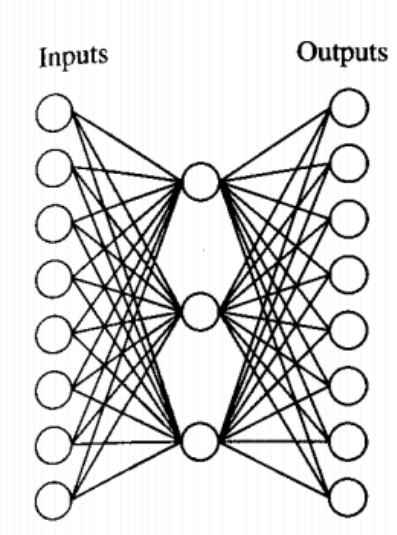
Exercise 4.9

Recall the $8 \times 3 \times 8$ network described in Figure 4.7. Consider trying to train an $8 \times 1 \times 8$ network for the same task; that is, a network with just one hidden unit. Notice the eight training examples in Figure~4.7 could be represented by eight distinct values for the single hidden unit (e.g., $0.1, 0.2, \ldots, 0.8$). Could a network with just one hidden unit therefore learn the identity function defined over these training examples?

Solution

Overview

In Figure 4.7, an $8 \times 3 \times 8$ network learns to map each of the eight distinct 8-bit input vectors to itself (the identity function). The question now is whether the same identity function can be learned by a smaller $8 \times 1 \times 8$ network. Intuitively, this means the single hidden unit would need to produce ***eight distinct*** activation values (e.g., $0.1, 0.2, \ldots, 0.8$) for the eight different inputs, and then the output layer would decode each of these hidden-unit values back into the correct 8-bit output.



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Figure 3: An $8 \times 3 \times 8$ *network*

ANSWER:

No, a network with just one hidden unit would not be able to learn the identity function defined over the eight training examples in Figure 4.7. This is because the eight distinct values for the single hidden unit cannot capture the *complexity* and *variability* of the input-output mapping. Additionally, there would not exist values for the output unit weights that could correctly decode this encoding of the input. Gradient descent is also unlikely to find such weights as it would not have enough *flexibility* to learn the desired mapping.

Exercise 4.10

Consider the alternative error function described in Section 4.8.1:

$$E(\mathbf{w}) \equiv \frac{1}{2} \sum_{d \in D_k} \sum_{\text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ij}^2$$

Derive the gradient descent update rule for this definition of E. Show that it can be implemented by multiplying each weight by some constant before performing the standard gradient descent update given in Table 4.2.

Solution

Let us denote E_1 as the first term of the error function, which corresponds to the sum of squared errors:

$$E_1 = \frac{1}{2} \sum_{d \in D_k} \sum_{\text{outputs}} (t_{kd} - o_{kd})^2$$

Derivative with Respect to Output Unit Weights

For the output unit weights, we compute the partial derivative:

$$\frac{\partial E_1}{\partial w_{ji}} = -(t_j - o_j) o_j (1 - o_j) x_{ji}$$

where:

- t_j is the target output,
- o_j is the actual output,
- x_{ji} is the input to the neuron.

Derivative with Respect to Hidden Unit Weights

For the hidden unit weights, we use the chain rule to obtain:

$$\frac{\partial E_1}{\partial w_{ji}} = -o_j(1-o_j)\left(\sum_{k \in \text{Downstream}(j)} \delta_k w_{kj}\right) x_{ji}$$

where δ_k represents the error term propagated from the downstream layers.

Regularization Term

Incorporating the regularization term $\gamma \sum_{i,j} w_{ij}^2$, we compute:

$$\frac{\partial}{\partial w_{ji}}\left(\gamma \sum_{i,j} w_{ij}^2\right) = 2\gamma w_{ji}$$

Thus, the total gradient of E with respect to w_{ji} is:

$$\nabla_{w_{ji}} E = \frac{\partial E_1}{\partial w_{ji}} + 2\gamma w_{ji}$$

Gradient Descent Update Rule

Using the standard gradient descent update rule:

$$w_{ji} \leftarrow w_{ji} - \eta \nabla_{w_{ji}} E$$

we substitute $\nabla_{w_{ji}} E$:

$$w_{ji} \leftarrow w_{ji} - \eta(\nabla_{w_{ji}} E_1 + 2\gamma w_{ji})$$

$$w_{ji} \leftarrow w_{ji} - \eta \nabla_{w_{ji}} E_1 - 2\eta\gamma w_{ji}$$

$$w_{ji} \leftarrow (1 - 2\eta\gamma) w_{ji} - \eta \nabla_{w_{ji}} E_1$$