VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MACHINE LEARNING (CO3117)

Homework 3:

Perceptron, ANN, Backpropagation

Team LHPD2

Semester 2, Academic Year 2024 - 2025

Teacher: Nguyen An Khuong
Students: Nguyen Quang Phu - 2252621 (***Leader***)
Nguyen Thanh Dat - 2252145 (Member)
Pham Huynh Bao Dai - 2252139 (Member)
Nguyen Tien Hung - 2252280 (Member)
Nguyen Thien Loc - 2252460 (Member)

HO CHI MINH CITY, FEBRUARY 2025

Contents

1 Introduction **2**

1.1 Purpose of This Document . . . 2

1.2 Workload . . . 2

2 Problem Description - Our Solution **3**

2.1 Exercise 13.1 [Backpropagation for a MLP] . . . 3

2.1.1 Description . . . 3

2.1.2 Solution . . . 5

2.1.2.1 Forward Pass . . . 5

2.1.2.2 Backward Pass (Gradient Computation) . . . 6

2.2 Question 4.4 . . . 8

2.2.1 Description . . . 8

2.2.2 Solution . . . 8

2.2.2.1 Model Setup . . . 8

2.2.2.2 Training Data . . . 9

2.2.2.3 Implementing the Delta Rule . . . 9

2.2.2.4 Plot Decision surface . . . 12

2.3 Question 4.9 . . . 13

2.3.1 Description . . . 13

2.3.2 Solution . . . 13

2.4 Question 4.10 . . . 15

2.4.1 Description . . . 15

2.4.2 Solution . . . 16

Chapter 1

Introduction

1.1 Purpose of This Document

This chapter serves as an introduction to our team, LHPD2, for the Machine Learning (CO3117) course in Semester 2, Academic Year 2024 - 2025. The purpose of this writing is to formally present our team members and confirm our participation in solving the given homework exercise, "Homework 3: Perceptron, ANN, Backpropagation".

1.2 Workload

Specifically, our contributions included:

- **Understanding the Problem:** Each member participated in analyzing and discussing the given exercise to ensure a shared understanding of the requirements.
- **Exploring Concepts:** We collectively researched and reviewed relevant concepts, theories, and methods necessary for solving the problems.
- **Reasoning for the Solution:** The team worked together to identify the rationale behind each approach, ensuring that all steps were logical and well-justified.
- **Implementing the Solution:** The solutions were implemented with equal collaboration, where every member contributed to coding, calculations, and documentation.
- **Final Review:** The team jointly reviewed the solutions to verify their correctness, clarity, and coherence before submission.

We believe that this exercise has enhanced our understanding of the concepts involved and strengthened our ability to work collaboratively as a team.

Chapter 2

Problem Description - Our Solution

2.1 Exercise 13.1 [Backpropagation for a MLP]

(Based on an exercise by Kevin Clark.)

2.1.1 Description

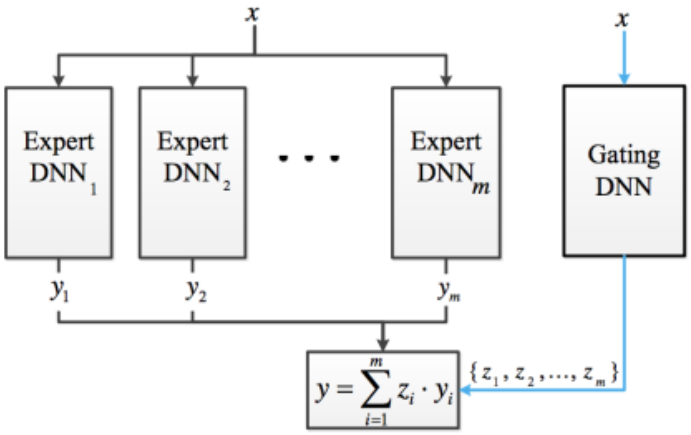


Figure 13.24: Deep MOE with m experts, represented as a neural network. From Figure 1 of [CGG17]. Used with kind permission of Jacob Goldberger.

Figure 2.1

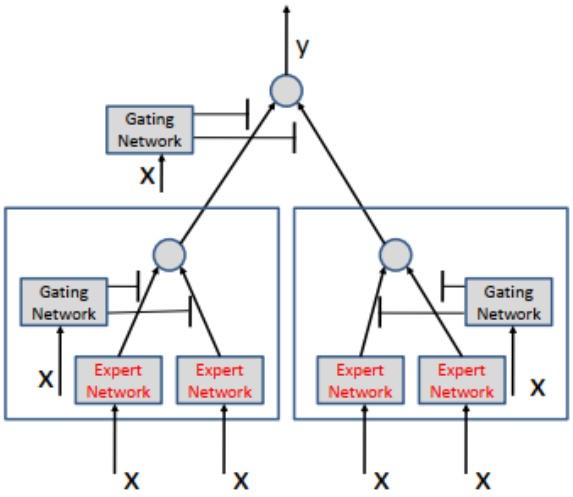


Figure 13.25: A 2-level hierarchical mixture of experts as a neural network. The top gating network chooses between the left and right expert, shown by the large boxes; the left and right experts themselves choose between their left and right sub-experts.

Figure 2.2

Consider the following classification MLP with one hidden layer:

$$\mathbf{x} = \text{input} \in \mathbb{R}^D \tag{13.111}$$
$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}_1 \in \mathbb{R}^K \tag{13.112}$$
$$\mathbf{h} = \text{ReLU}(\mathbf{z}) \in \mathbb{R}^K \tag{13.113}$$
$$\mathbf{a} = \mathbf{U}\mathbf{h} + \mathbf{b}_2 \in \mathbb{R}^C \tag{13.114}$$
$$\mathcal{L} = \text{CrossEntropy}(\mathbf{y}, \text{softmax}(\mathbf{a})) \in \mathbb{R} \tag{13.115}$$

where $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{b}_1 \in \mathbb{R}^K$, $\mathbf{W} \in \mathbb{R}^{K \times D}$, $\mathbf{b}_2 \in \mathbb{R}^C$, $\mathbf{U} \in \mathbb{R}^{C \times K}$, where D is the size of the input, K is the number of hidden units, and C is the number of classes. Show that the gradients for the parameters and input are as follows:

$$\nabla_{\mathbf{V}}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial \mathbf{V}}\right]_{1,:} = \delta_1 \mathbf{h}^T \in \mathbb{R}^{C\times K} \tag{13.116}$$

$$\nabla_{\mathbf{b}_2}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}_2}\right)^T = \delta_1 \in \mathbb{R}^C \tag{13.117}$$

$$\nabla_{\mathbf{W}}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial \mathbf{W}}\right]_{1,:} = \delta_2 \mathbf{x}^T \in \mathbb{R}^{K\times D} \tag{13.118}$$

$$\nabla_{\mathbf{b}_1}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}_1}\right)^T = \delta_2 \in \mathbb{R}^K \tag{13.119}$$

$$\nabla_{\mathbf{x}}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}}\right)^T = \mathbf{W}^T \delta_2 \in \mathbb{R}^D \tag{13.120}$$

where the gradients of the loss with respect to the two layers (logit and hidden) are given by the following:

$$\delta_1 = \nabla_{\mathbf{a}}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}}\right)^T = (\mathbf{p} - \mathbf{y}) \in \mathbb{R}^C \tag{13.121}$$

$$\delta_2 = \nabla_{\mathbf{z}}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}}\right)^T = (\mathbf{V}^T \delta_1) \odot H(\mathbf{z}) \in \mathbb{R}^K \tag{13.122}$$

where H is the Heaviside function. Note that, in our notation, the gradient (which has the same shape as the variable with respect to which we differentiate) is equal to the Jacobian's transpose when the variable is a vector and to the first slice of the Jacobian when the variable is a matrix.

2.1.2 Solution

2.1.2.1 Forward Pass

Consider a classification MLP with:

- Input: $\mathbf{x} \in \mathbb{R}^D$
- Hidden layer pre-activation: $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b_1} \in \mathbb{R}^K$
- Activation: $\mathbf{h} = \text{ReLU}(\mathbf{z})$
- Output layer pre-activation (logits): $\mathbf{a} = \mathbf{U}\mathbf{h} + \mathbf{b_2} \in \mathbb{R}^C$
- Softmax probabilities: $\mathbf{p} = \text{softmax}(\mathbf{a})$
- Loss function: $\mathcal{L} = \text{CrossEntropy}(\mathbf{y}, \mathbf{p})$

2.1.2.2 Backward Pass (Gradient Computation)

Using the chain rule, we derive the gradients:

Step 1: Compute δ_1

The gradient of the loss with respect to the logits $\mathbf{a}$ is derived from the Softmax + Cross-Entropy function:

$$\delta_1 = \frac{\partial \mathcal{L}}{\partial \mathbf{a}} = (\mathbf{p} - \mathbf{y})^T \tag{2.1}$$

This follows from the fact that the derivative of the softmax function coupled with cross-entropy simplifies to $\mathbf{p} - \mathbf{y}$, reducing computational complexity.

Step 2: Compute δ_2 (Hidden Layer)

The gradient with respect to the hidden layer pre-activation $\mathbf{z}$ is computed using the chain rule:

$$\delta_2 = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \delta_1 \mathbf{U} \odot H(\mathbf{h}) \tag{2.2}$$

where $H(\mathbf{h})$ is the Heaviside step function, representing the derivative of ReLU. Since ReLU is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2.3}$$

its derivative is:

$$H(\mathbf{h}) = \begin{cases} 1, & \text{if } \mathbf{z} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Thus, δ_2 propagates gradients only for activated neurons.

Step 3: Compute Gradients w.r.t. Parameters

Applying the chain rule again:

$$\nabla_{\mathbf{U}} \mathcal{L} = \delta_1 \mathbf{h}^T \tag{2.5}$$

This follows because $\mathbf{a} = \mathbf{U}\mathbf{h} + \mathbf{b_2}$, so $\frac{\partial \mathbf{a}}{\partial \mathbf{U}} = \mathbf{h}^T$.

$$\nabla_{\mathbf{b_2}} \mathcal{L} = \delta_1 \tag{2.6}$$

Since $\mathbf{b_2}$ is added directly to $\mathbf{a}$, its derivative is simply 1.

$$\nabla_{\mathbf{W}}\mathcal{L} = \delta_2 \mathbf{x}^T \tag{2.7}$$

Since $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b_1}$, we obtain $\frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \mathbf{x}^T$.

$$\nabla_{\mathbf{b_1}}\mathcal{L} = \delta_2 \tag{2.8}$$

Similar to $\mathbf{b_2}$, since $\mathbf{b_1}$ is added directly to $\mathbf{z}$, its derivative is 1.

$$\nabla_{\mathbf{x}}\mathcal{L} = \mathbf{W}^T \delta_2 \tag{2.9}$$

which propagates the gradients back to the input layer.

2.2 Question 4.4

2.2.1 Description

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, ..., iterations.

(a) Try this using various constant values for η and using a decaying learning rate of η_0/i for the ith iteration. Which works better?

(b) Try incremental and batch learning. Which converges more quickly? Consider both the number of weight updates and total execution time.

2.2.2 Solution

Our source code is here: Click here

2.2.2.1 Model Setup

We have a two-input linear unit with inputs x_1 and x_2, weights w_1, w_2, and a bias w_0. The output of the unit is:

$$y = w_0 + w_1 x_1 + w_2 x_2$$

The target concept is $-2 + x_1 + 2x_2 > 0$, which can be rewritten as:

$$x_1 + 2x_2 > 2$$

This is a linear decision boundary in the x_1-x_2 plane. We need to adjust the weights w_0, w_1, and w_2 to match this boundary using the delta rule.

The delta rule (or Widrow-Hoff rule) updates the weights as follows:

$$w_i(t+1) = w_i(t) + \eta(d - y)x_i$$

where:

- $w_i(t)$ is the weight at iteration t,
- η is the learning rate,
- d is the desired output (target),
- y is the actual output,
- x_i is the input feature.

For the bias w_0, we treat it as a weight with $x_0 = 1$:

$$w_0(t+1) = w_0(t) + \eta(d-y) \cdot 1$$

The error E for each iteration can be defined as the mean squared error over the training data:

$$E = \frac{1}{2}\sum(d-y)^2$$

2.2.2.2 Training Data

To train the model, we generate a set of training points (x_1, x_2) in a 2D space (e.g., $x_1, x_2 \in [-5, 5]$) and label them based on the target concept: - If $-2 + x_1 + 2x_2 > 0$, then $d = 1$, - Otherwise, $d = 0$.

We use a sufficiently large number of random points (e.g., 100–1000) to ensure good coverage of the input space.

2.2.2.3 Implementing the Delta Rule

We initialize the weights w_0, w_1, and w_2 to small random values (e.g., in the range $[-0.1, 0.1]$).

(a) Learning Rate Experiments

We experiment with two types of learning rates:

- Constant learning rate η: Try values like $\eta = 0.01, 0.1, 0.5, 1.0$.
- Decaying learning rate: Use $\eta_i = \eta_0/i$, where η_0 is an initial learning rate (e.g., $\eta_0 = 0.1, 0.5$) and i is the iteration number.

For each learning rate, we:

- Run the delta rule for a fixed number of iterations (e.g., 1000).
- Compute the error E after each iteration and plot E vs. the number of iterations.
- After 5, 10, 50, 100, ... iterations, plot the decision surface, which is the line $w_0 + w_1x_1 + w_2x_2 = 0$.

The decision surface is found by solving:

$$w_0 + w_1x_1 + w_2x_2 = 0 \implies x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2}x_1$$

Results:

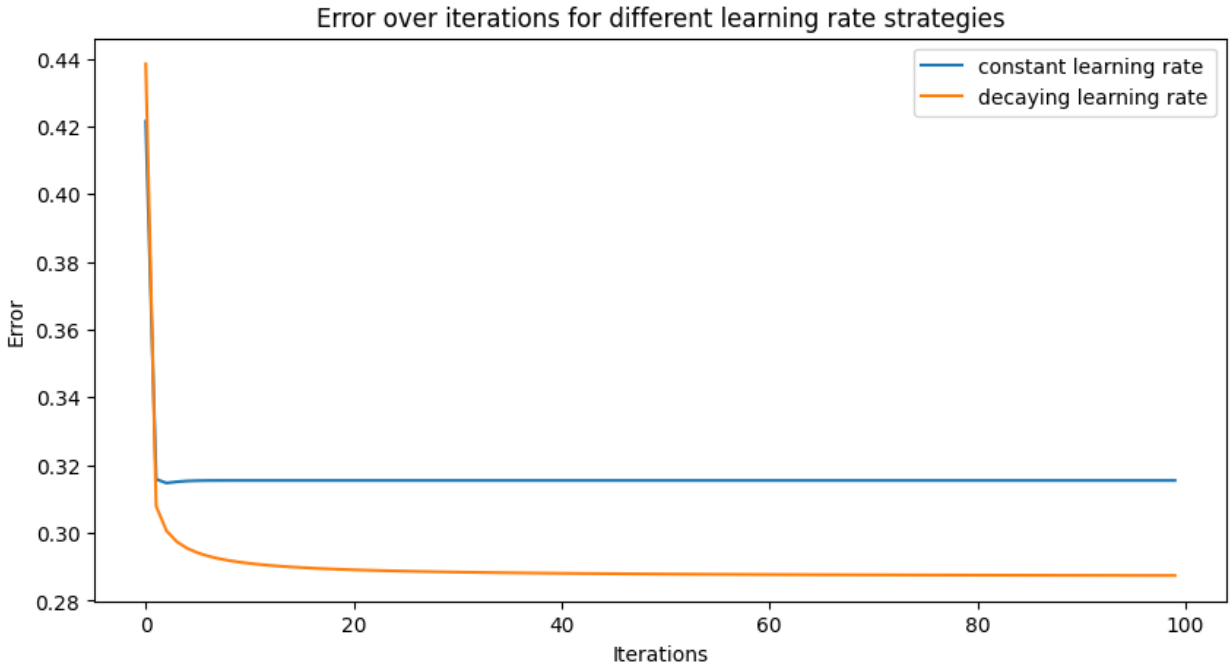


Figure 2.3: Error over iterations for constant learning rate $\eta = 0.01$

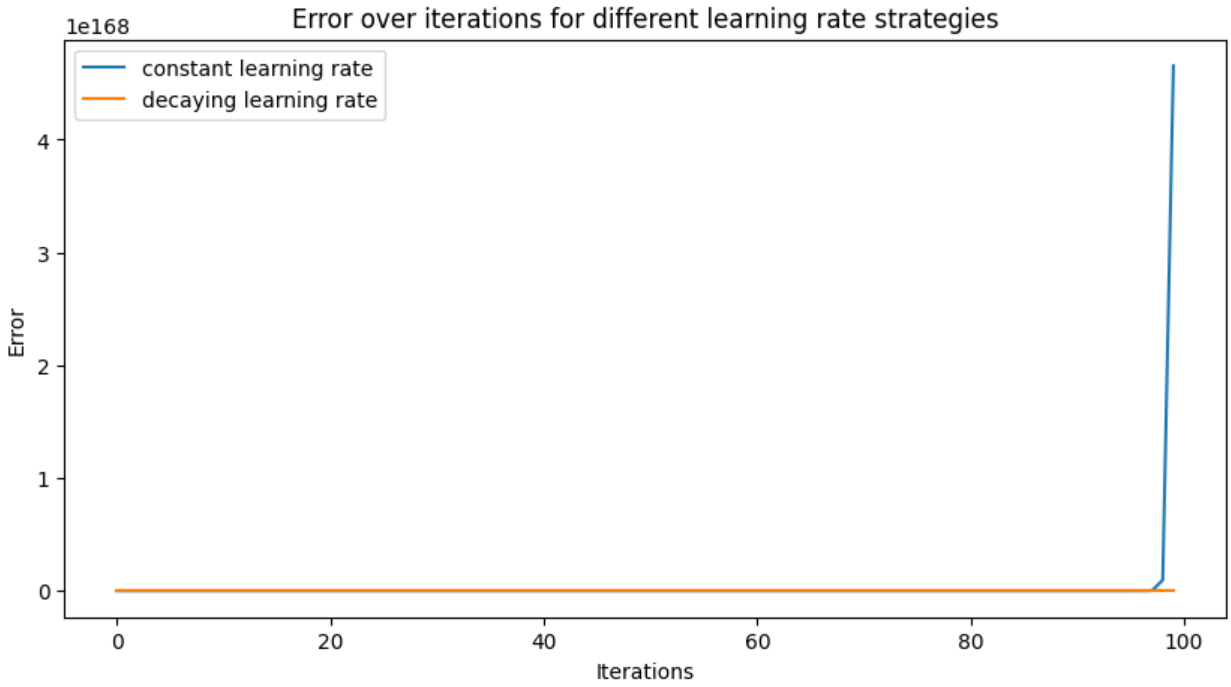


Figure 2.4: Error over iterations for constant learning rate $\eta = 0.1$

- A constant η that is too small (e.g., 0.01) may converge slowly, while a large η (e.g., 0.1) cause oscillate or diverge.
- A decaying learning rate η_0/i typically converges more stably and often faster in the long run because it starts with a larger step size and gradually reduces it, avoiding oscillations.

Conclusion: The decaying learning rate works better and is more stable.

(b) Incremental vs. Batch Learning

- **Incremental (Online) Learning**: Update weights after each training example:

$$w_i(t+1) = w_i(t) + \eta(d_k - y_k)x_{ki}$$

  where (x_{k1}, x_{k2}, d_k) is the kth training example.

- **Batch Learning**: Accumulate the gradient over all training examples and update weights once per epoch:

$$\Delta w_i = \eta \sum_k (d_k - y_k)x_{ki}$$

$$w_i(t+1) = w_i(t) + \Delta w_i$$

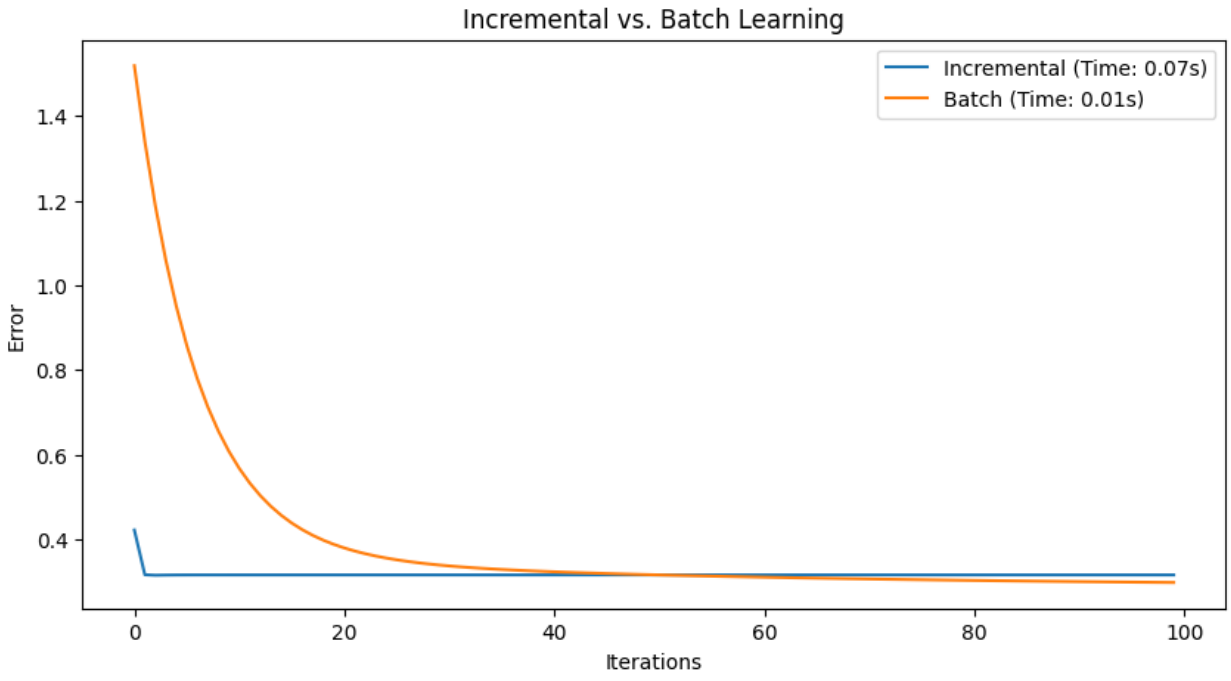


Figure 2.5: Incremental vs Batch Leaning $\eta = 0.01$

Conclusion

Batch learning converges more quickly than incremental learning.

- **In Terms of Weight Updates:** Batch learning processes the entire dataset per update, leading to larger, more stable updates and faster convergence in fewer iterations. Incremental learning updates weights more frequently but with smaller adjustments, requiring more iterations to reach stability.
- **In Terms of Execution Time:** Each batch update is computationally more efficient, reducing total training time. Incremental learning, while adaptable, tends to take longer due to frequent, smaller updates.

Overall, batch learning achieves convergence faster in both weight updates and total execution time.

2.2.2.4 Plot Decision surface

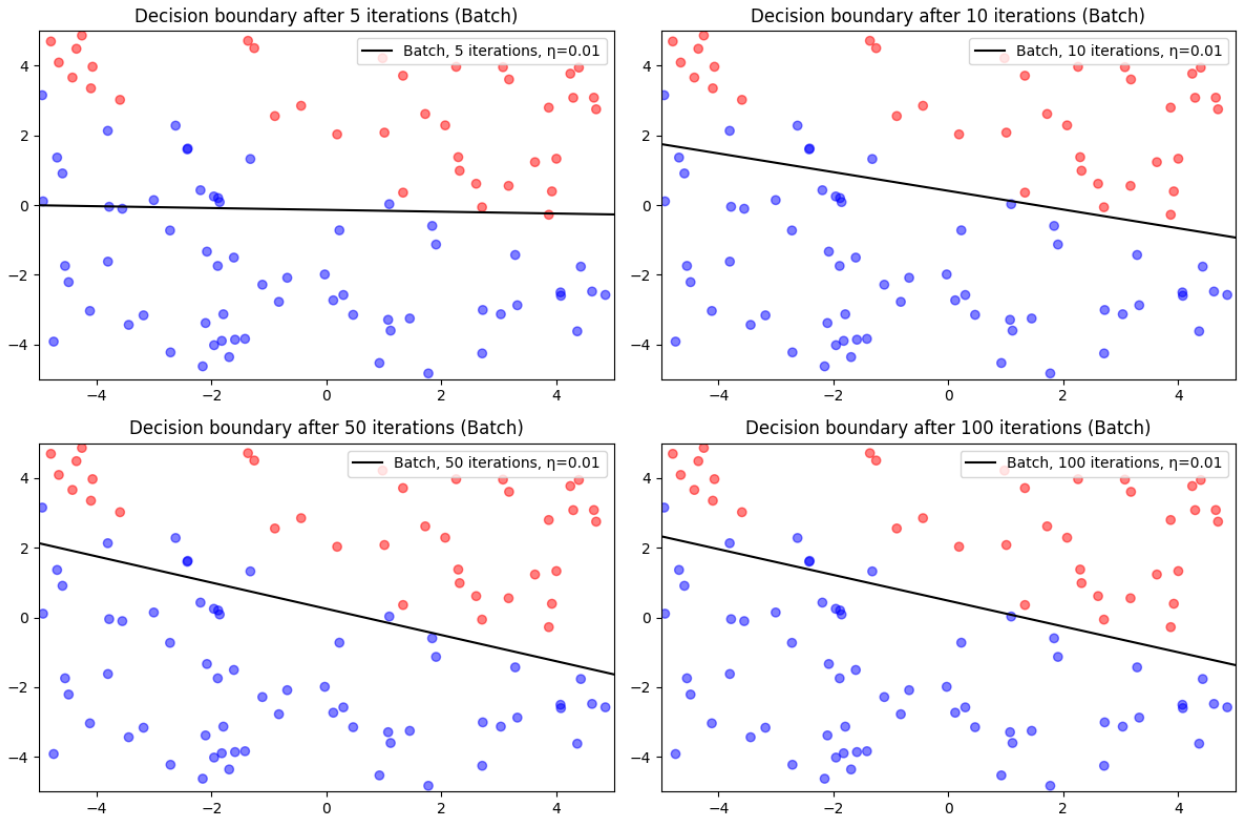


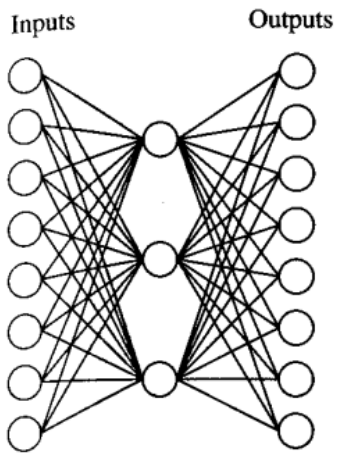
Figure 2.6: Decision surface for Batch Learning

2.3 Question 4.9

2.3.1 Description

Recall the $8 \times 3 \times 8$ network described in Figure 4.7. Consider trying to train a $8 \times 1 \times 8$ 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?

Hint: Consider questions such as "do there exist values for the hidden unit weights that can create the hidden unit encoding suggested above?" "do there exist values for the output unit weights that could correctly decode this encoding of the input?" and "is gradient descent likely to find such weights?"



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 4.7
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

Figure 2.7

2.3.2 Solution

I thought that a neural network with just one hidden unit (an $8 \times 1 \times 8$ network) can't learn the identity function for the eight training examples shown in the figure. Here are my detailed explanation:

1. **Do there exist values for the hidden unit weights that can create the hidden unit encoding suggested above?**

Yes, theoretically, there could exist values for the weights and biases of the hidden unit that could produce eight different output values, each represeting one of the eight 8-bit inputs(e.g., 0.1 for "1000000", 0.2 for "0100000",..., 0.8 for 0000001). However, this requires perfectly tuning the weights so that the hidden unit produces completely distinct and non-overlapping values for each input. With only one hidden unit, this is very difficult because it has to "compress" the information from eight different inputs into a single value, and its representational capacity is limited. Therefore, while it's possible in theory, in practice, it's very hard to achieve.

2. **Do there exist values for the output unit weights that could correctly decode this encoding of the input?**
   Yes, if the hidden unit truly produces eight distinct and distinguishable output values (as mentioned in question 1), then there could exist weights for the output layer (8 units) to decode those values and recreate the correct 8-bit output (e.g., from the hidden unit's value of 0.1, the output layer produces "10000000"). However, this depends on the hidden unit actually generating sufficiently distinct values. Because a single hidden unit has limited representational power, it's very difficult to ensure that its output values will always be clear and not overlap between different inputs. Thus, while it's possible in theory, in practice, it's challenging because the network lacks the capacity to encode the necessary information fully.

3. **Is gradient descent likely to find such weights?**
   No, gradient descent, a common algorithm for training neural networks, is very unlikely or unable to find such ideal weights in this case. The reasons are:

   - With only one hidden unit, the search space for the weights is very limited, and the network lacks sufficient dimensions to distinguish the eight different inputs effectively.
   - Gradient descent can get stuck in local minima or fail to converge to the correct solution, especially when the network is too simple (underdetermined) for a complex task like learning eight distinct patterns.
   - In the original network with three hidden units, gradient descent works better because there are more dimensions to adjust and find a solution, but with just one hidden unit, this ability is significantly reduced. Therefore, in practice, gradient descent is unlikely to find the necessary weights for the network to work correctly.

2.4 Question 4.10

2.4.1 Description

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, *where* $\vec{x}$ *is the vector of network input values, and* $\vec{t}$ *is the vector of target network output values.*

η *is the learning rate (e.g., .05).* n_{in} *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*

The input from unit i *into unit* j *is denoted* x_{ji}, *and the weight from unit* i *to unit* j *is denoted* w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

Figure 2.8

Consider the alternative error function described in Section 4.8.1:

$$E(\tilde{\mathbf{w}}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

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.

2.4.2 Solution

Given the alternative error function:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2 \tag{2.10}$$

where:

- The first term represents the standard squared error loss,
- The second term introduces L2 regularization (weight decay) with regularization coefficient γ.

Gradient of the Squared Error Term

The first term of $E(\mathbf{w})$ is:

$$E_1 = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 \tag{2.11}$$

Taking the derivative with respect to a weight w_{ji}, we apply the chain rule:

$$\frac{\partial E_1}{\partial w_{ji}} = \sum_{k \in \text{outputs}} (t_k - o_k) \frac{\partial}{\partial w_{ji}} (-o_k) \tag{2.12}$$

From the standard backpropagation rule, the derivative of the output activation function is:

$$\frac{\partial o_k}{\partial w_{ji}} = o_k(1 - o_k) x_i \tag{2.13}$$

Thus, we obtain:

$$\frac{\partial E_1}{\partial w_{ji}} = -(t_k - o_k) o_k (1 - o_k) x_i = -\delta_k x_i \tag{2.14}$$

where we define:

$$\delta_k = o_k(1 - o_k)(t_k - o_k) \tag{2.15}$$

This leads to the standard weight update rule in backpropagation:

$$\Delta w_{ji} = \eta \delta_j x_i \tag{2.16}$$

Gradient of the Regularization Term

The second term of $E(\mathbf{w})$ introduces weight decay:

$$E_2 = \gamma \sum_{i,j} w_{ji}^2 \tag{2.17}$$

Taking the derivative with respect to w_{ji}:

$$\frac{\partial E_2}{\partial w_{ji}} = 2\gamma w_{ji} \tag{2.18}$$

Final Weight Update Rule

Combining both derivatives:

$$\frac{\partial E}{\partial w_{ji}} = -\delta_j x_i + 2\gamma w_{ji} \tag{2.19}$$

The gradient descent update rule is:

$$w_{ji} \leftarrow w_{ji} - \eta \left(\delta_j x_i - 2\gamma w_{ji}\right) \tag{2.20}$$

which simplifies to:

$$w_{ji} \leftarrow (1 - 2\eta\gamma) w_{ji} + \eta \delta_j x_i \tag{2.21}$$

Interpretation

- The term $(1 - 2\eta\gamma)w_{ji}$ applies weight decay, preventing large weights and reducing overfitting.
- The term $\eta \delta_j x_i$ ensures proper error correction through standard backpropagation.

Thus, implementing regularization in backpropagation requires multiplying each weight by $(1 - 2\eta\gamma)$ before applying the usual update.