國立陽明交通大學

NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning

Lab1: back-propagation

Name: 許子駿

Student ID: 311551166

Institute of Computer Science and Engineering

2023 年 3 月 18 日

Table of Contents

Chapter 1

Introduction

This task is to build a deep neural network with 2 hidden layers to classify 2 datasets which each has 2 classes (See Figure 1.1).
One dataset can be approximately seperated by an $y = x$ linear classifier. The other dataset is like an X, the data points on the $y = x$ line belong to the same class, and data points on the $xy = 1$ line belong to the other class (See Figure 1.2).

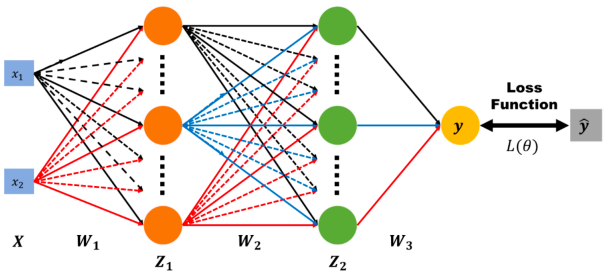

Figure 1.1: Illustration of deep neural network with 2 hidden layers.

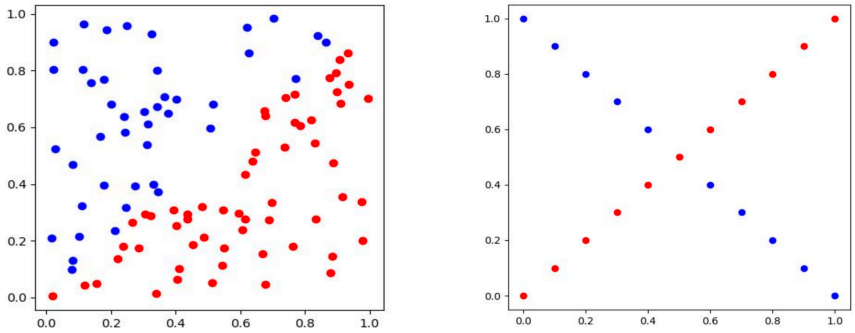

Figure 1.2: Illustration of two datasets.

Chapter 2

Experiment setup

Experiment setup is listed in this chapter including 5 parts, that is, initialization method, activation functions, optimizers, loss function, and the data structure of linear layer.

2.1 Initialization method

He normal initialization is used in this experiment (See Listing 2.1). This initialization method performs well empirically when neural networks include `ReLU` activation function by many researchers.

```
    dim_in = layers[i]
    dim_out = layers[i + 1]
    a = np.sqrt(6. / (dim_in + dim_out))
    std = np.sqrt(2. / ((1 + a ** 2) * dim_in))

    W = np.random.normal(loc=0., scale=std, size=[dim_out, dim_in])
    b = np.random.normal(loc=0., scale=std, size=[1, dim_out])
```

Listing 2.1: He normal initialization code.

2.2 Activation functions

There are many activation functions, e.g., `LeakyReLU`, `ReLU`, `Sigmoid`, and etc. In this experiment, 3 activation functions, `LeakyReLU`, `ReLU` and `Sigmoid` (See Figures 2.1, 2.2, and 2.3), are implemented.

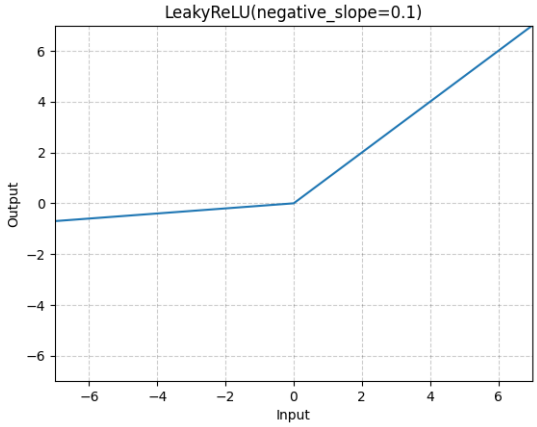

Figure 2.1: Illustration of LeakyReLU activation function.

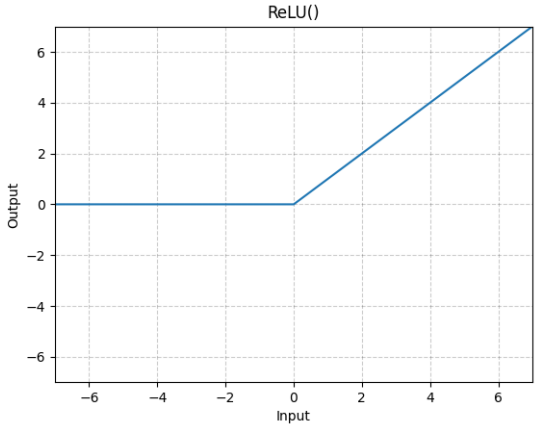

Figure 2.2: Illustration of ReLU activation function.

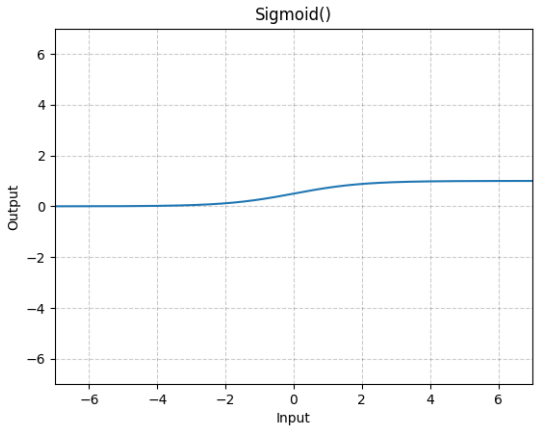

Figure 2.3: Illustration of Sigmoid activation function.

2.3 Optimizers

There are many optimizers, e.g., `SGD`, `Momentum`, `AdaGrad`, `Adam`, and etc. In this experiment, 2 optimizers, `SGD` (vanilla gradient descent in this case) and `Adam` (See Figure 2.4) optimizers, are implemented.
`Adam` optimizer is mostly used, and it has the great parts of `Momentum` and `AdaGrad`.

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

---

return θ_t

Figure 2.4: Pseudo code of Adam optimizer.

2.4 Loss function

Cross entropy (See Equation 2.1) is used in this experiment, and it is also mostly used for classification tasks.

$$L(y, \hat{y}) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \tag{2.1}$$

where y are ground truth labels and $\hat{y}$ are predicted labels.

2.5 Data structure of linear layer

Data structure of linear layer is implemented using a class with only two member variables. It is well-defined, since `W` (weight matrix) and `b` (bias vector) are almost used at the same situations.

```
class Linear(object):
    def __init__(self, W: np.ndarray, b: np.ndarray) -> None:
        self._W = W
        self._b = b

    @property
    def W(self) -> np.ndarray:
        return self._W

    @property
    def b(self) -> np.ndarray:
        return self._b

    @W.setter
    def W(self, W: np.ndarray) -> None:
        self._W = W

    @b.setter
    def b(self, b: np.ndarray) -> None:
        self._b = b

    def __str__(self) -> str:
        return f'<Linear W: {self._W.shape} b: {self._b.shape}>'
```

Listing 2.2: Date structure code of linear layer.

Chapter 3

Analysis and comments

Different optimizers, activation functions, learning rates, and dimension of hidden layers are tested in this chapter. Experiments of this chapter are done with common setup below:

- Seed: 42.
- Number of hidden layers: 2.

3.1 Different optimizers

Experiments of this section are done with common setup below:

- Activation function: `LeakyReLU`.
- Learning rate: 10^{-2}.
- Dimension of hidden layers: 4.

From Figures 3.1 and 3.2, we can see that using `Adam` optimizer makes the model converge faster in this case.

Figure 3.1: Results with Adam optimizer. In this case, the model converges in less than 100 and 200 epochs, respectively.

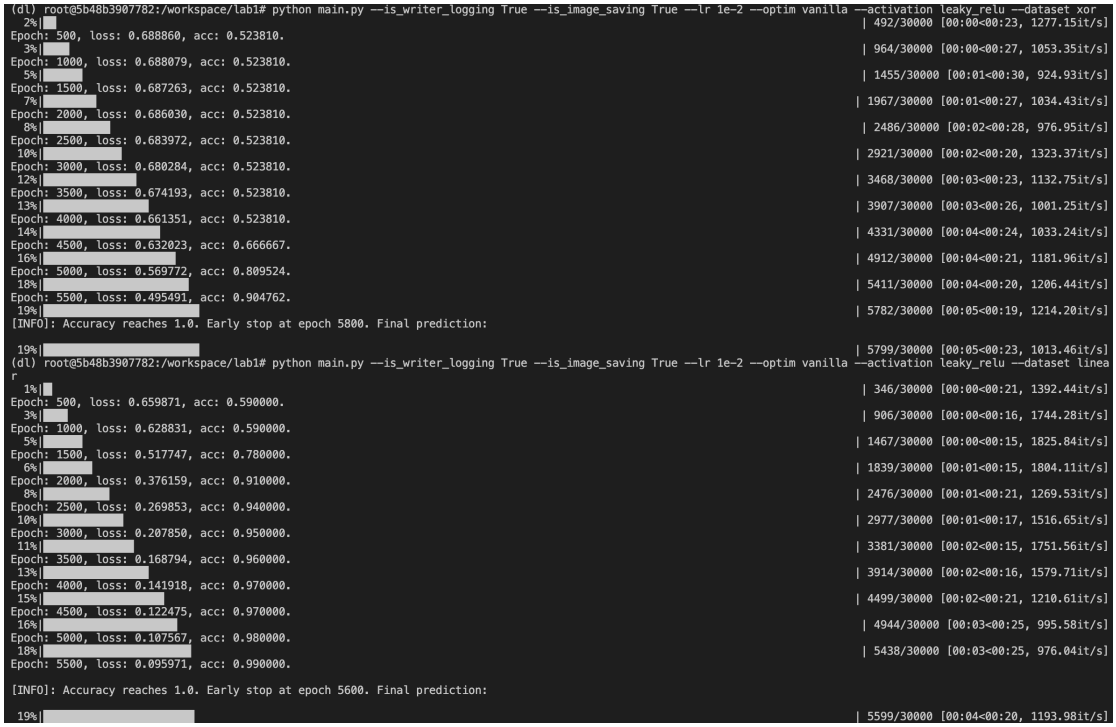

Figure 3.2: Results with SGD optimizer. In this case, the model converges in less than 5800 and 5600 epochs, respectively.

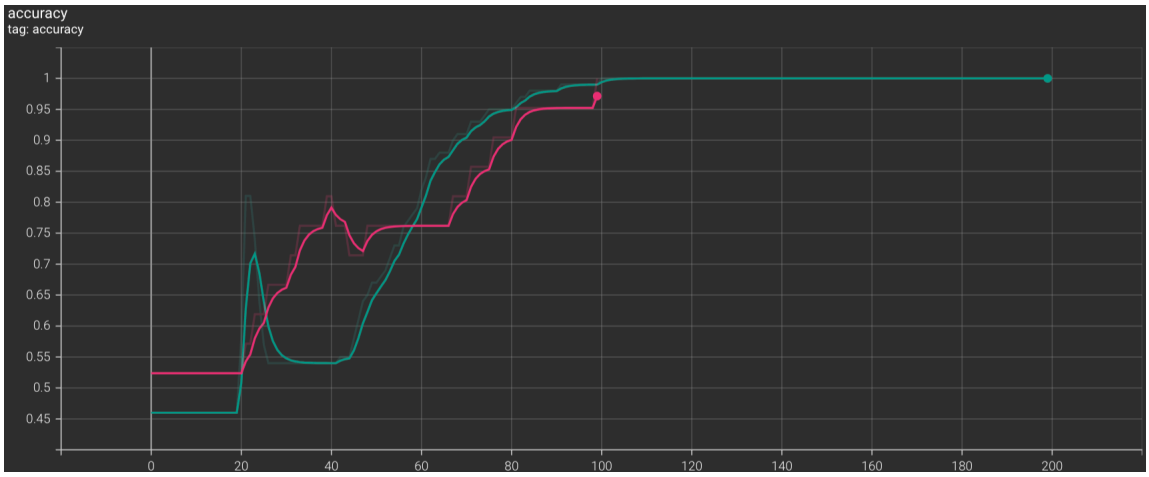

Figure 3.3: Accuracy with Adam optimizer.

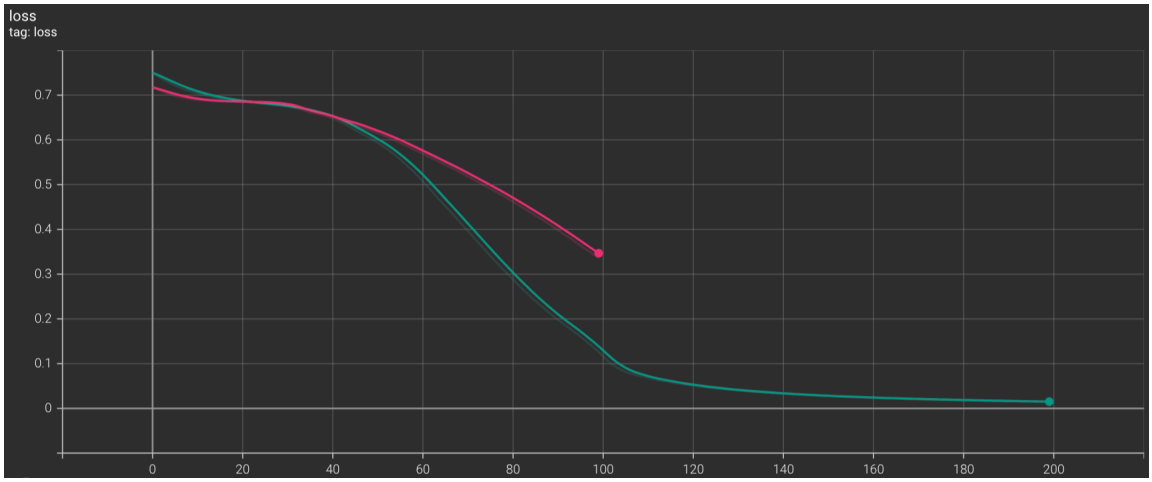

Figure 3.4: Loss with Adam optimizer.

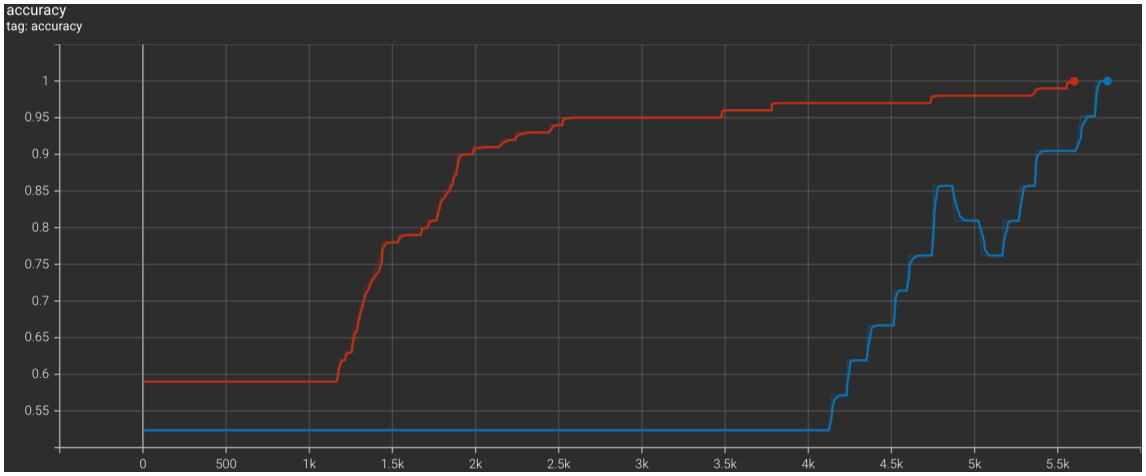

Figure 3.5: Accuracy with SGD optimizer.

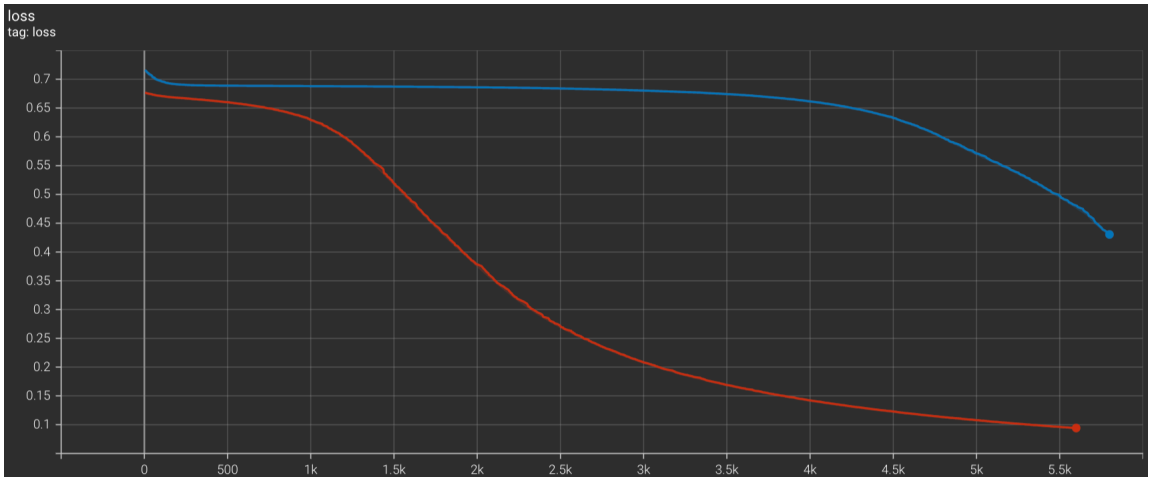

Figure 3.6: Loss with SGD optimizer.

3.2 Different activation functions

Experiments of this section are done with common setup below:

- Optimizer: `Adam`.
- Learning rate: 10^{-2}.
- Dimension of hidden layers: 4.

From Figures 3.7 and 3.8, we can see that using `LeakyReLU` optimizer makes the model converge faster in this case.

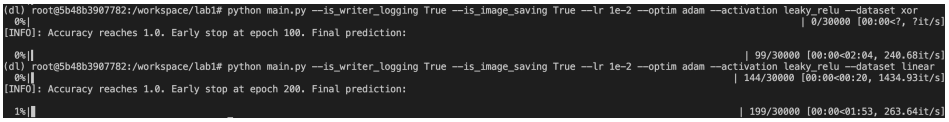

Figure 3.7: Results with LeakyReLU. In this case, the model converges in less than 100 and 200 epochs, respectively.

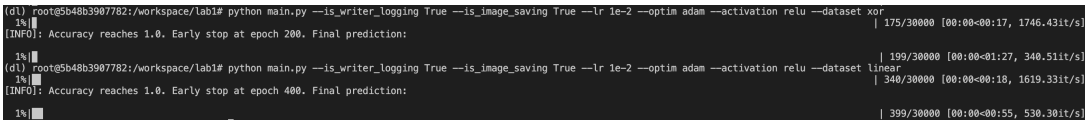

Figure 3.8: Results with ReLU. In this case, the model converges in less than 200 and 400 epochs, respectively.

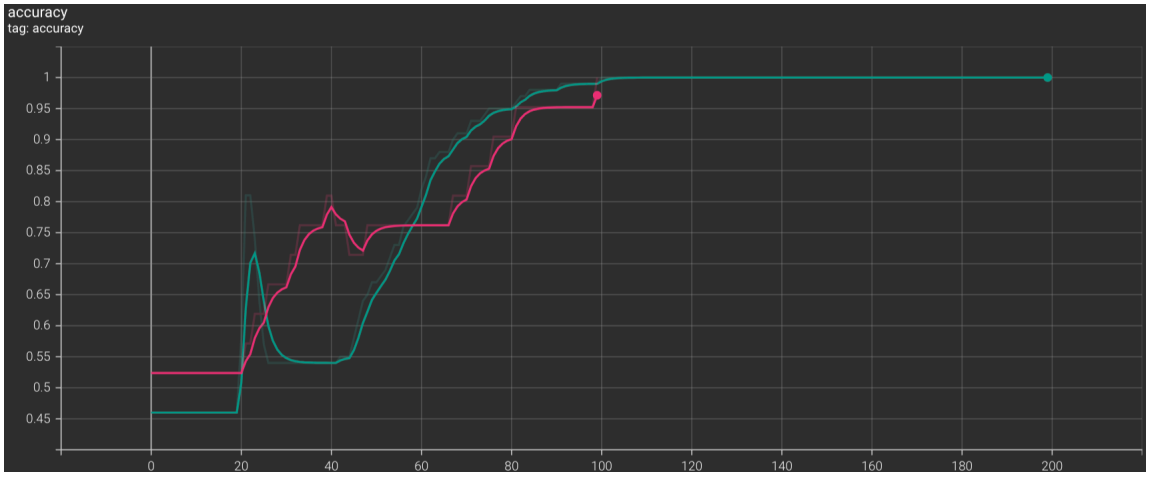

Figure 3.9: Accuracy with LeakyReLU.

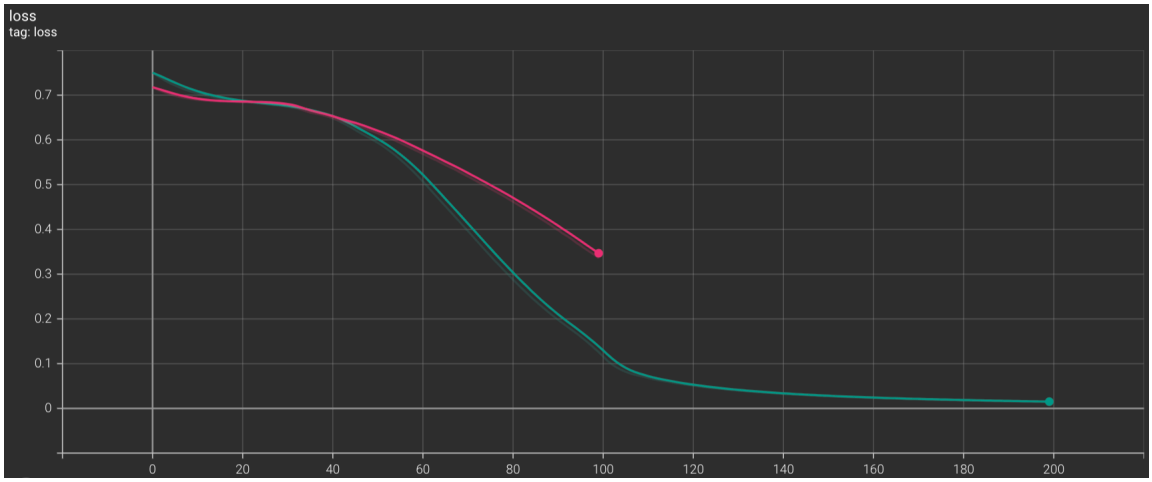

Figure 3.10: Loss with LeakyReLU.

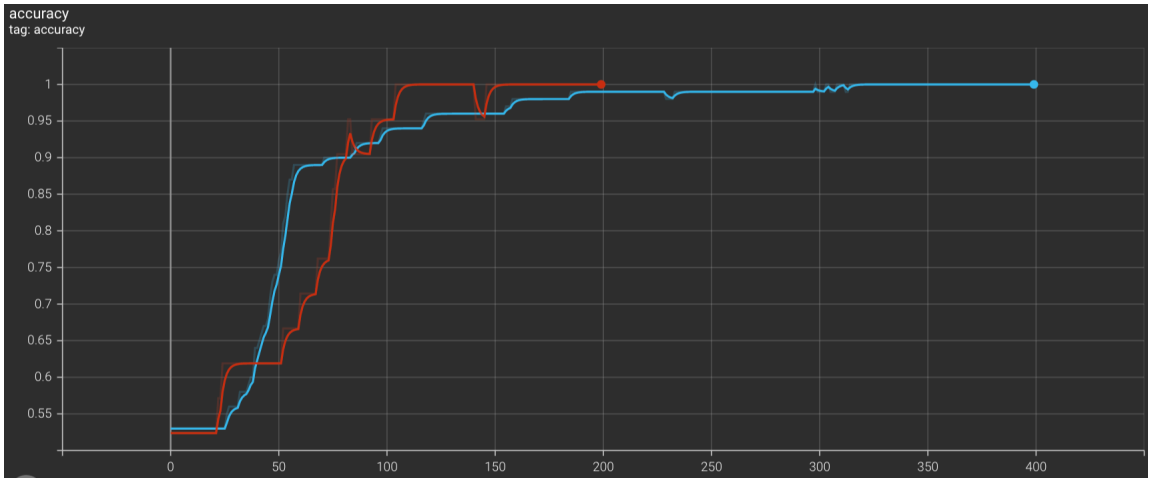

Figure 3.11: Accuracy with ReLU.

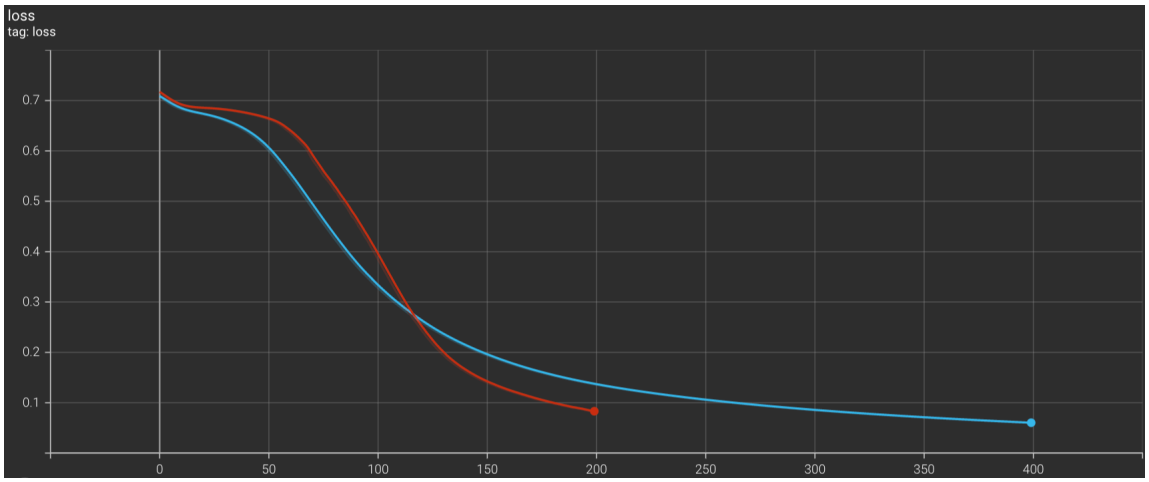

Figure 3.12: Loss with ReLU.

3.3 Different learning rates

Experiments of this section are done with common setup below:

- Optimizer: `Adam`.
- Activation function: `LeakyReLU`.
- Dimension of hidden layers: 4.

From Figures 3.13 and 3.14, we can see that using learning rate 10^{-2} makes the model converge faster in this case.

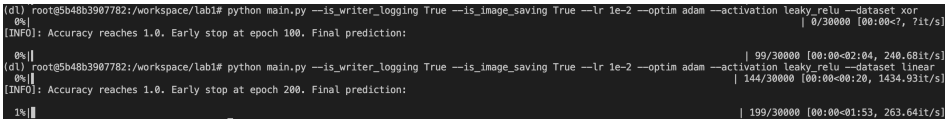

Figure 3.13: Results with learning rate 10^{-2}. In this case, the model converges in less than 100 and 200 epochs, respectively.

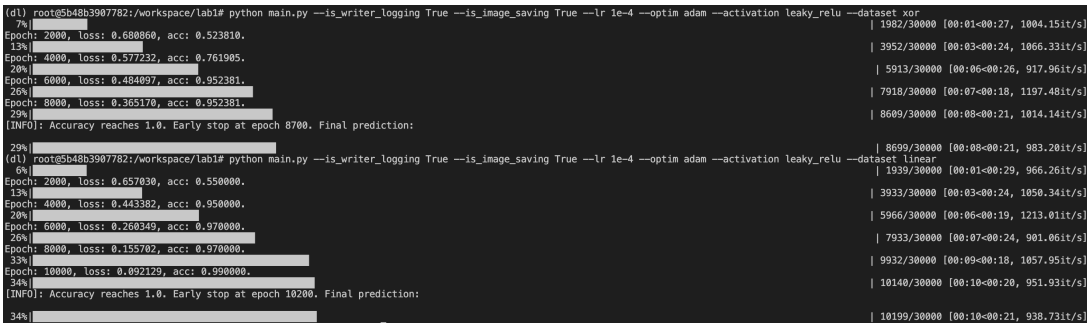

Figure 3.14: Results with learning rate 10^{-4}. In this case, the model converges in less than 8700 and 10200 epochs, respectively.

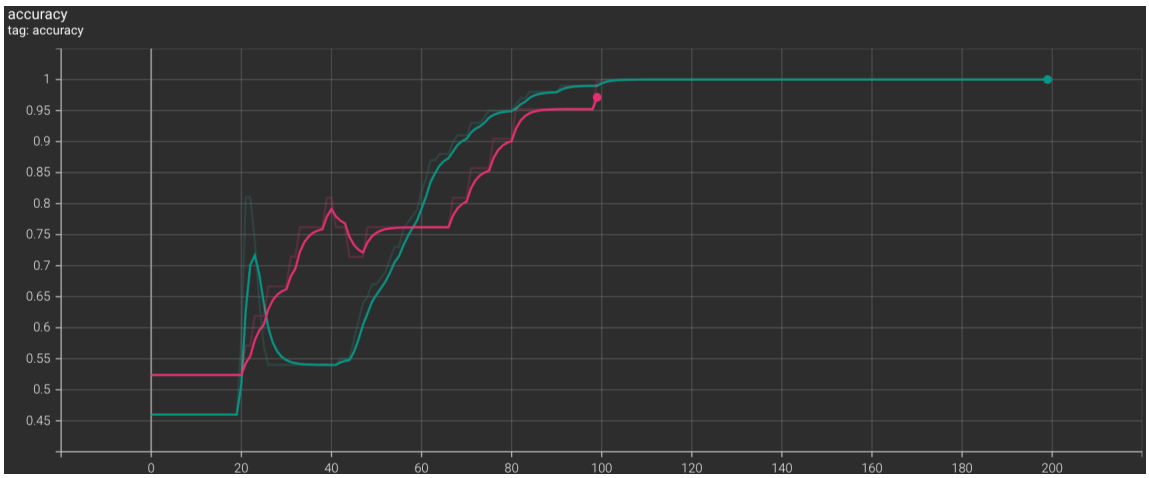

Figure 3.15: Accuracy with learning rate 10^{-2}.

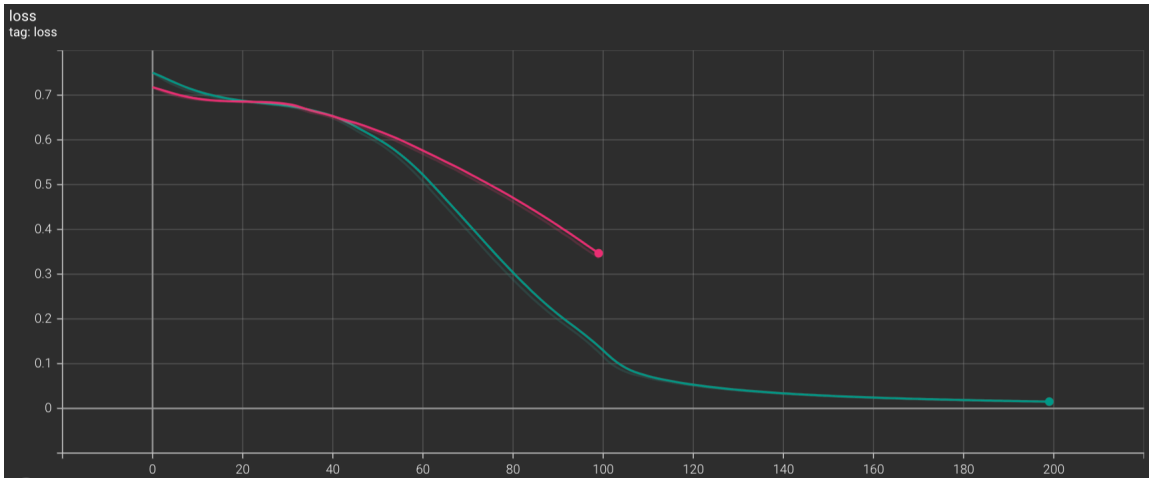

Figure 3.16: Loss with learning rate 10^{-2}.

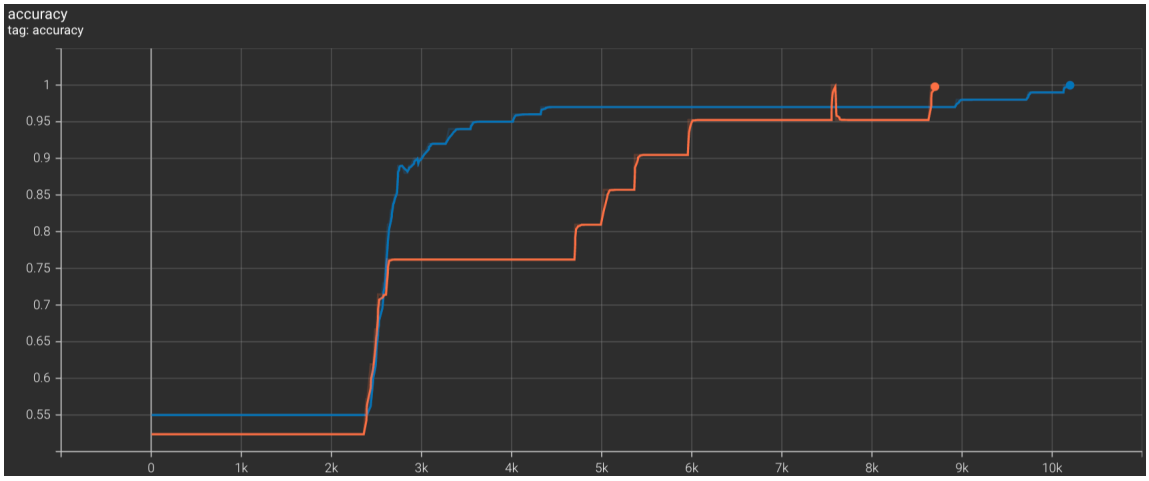

Figure 3.17: Accuracy with learning rate 10^{-4}.

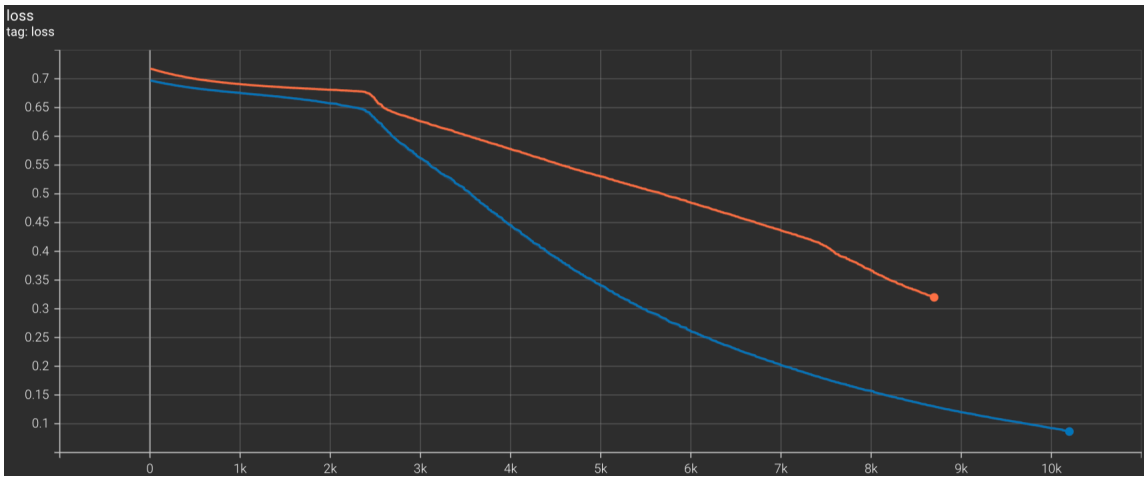

Figure 3.18: Loss with learning rate 10^{-4}.

3.4 Different dimension of hidden layers

Experiments of this section are done with common setup below:

- Optimizer: `Adam`.
- Activation function: `LeakyReLU`.
- Learning rate: 10^{-2}.

From Figures 3.19 and 3.20, we can see that using 16-dimension hidden layers makes the model converge faster in this case.

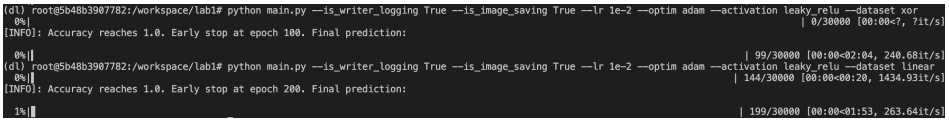

Figure 3.19: Results with 4-dimension hidden layers. In this case, the model converges in less than 100 and 200 epochs, respectively.

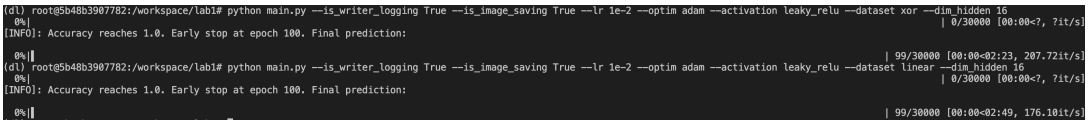

Figure 3.20: Results with 16-dimension hidden layers. In this case, the model converges in less than 100 and 100 epochs, respectively.

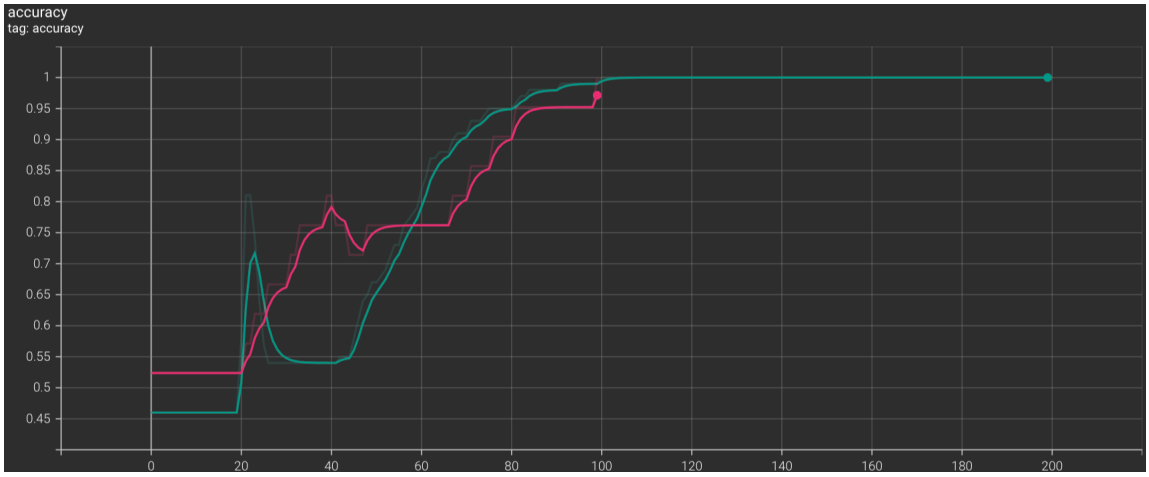

Figure 3.21: Accuracy with 4-dimension hidden layers.

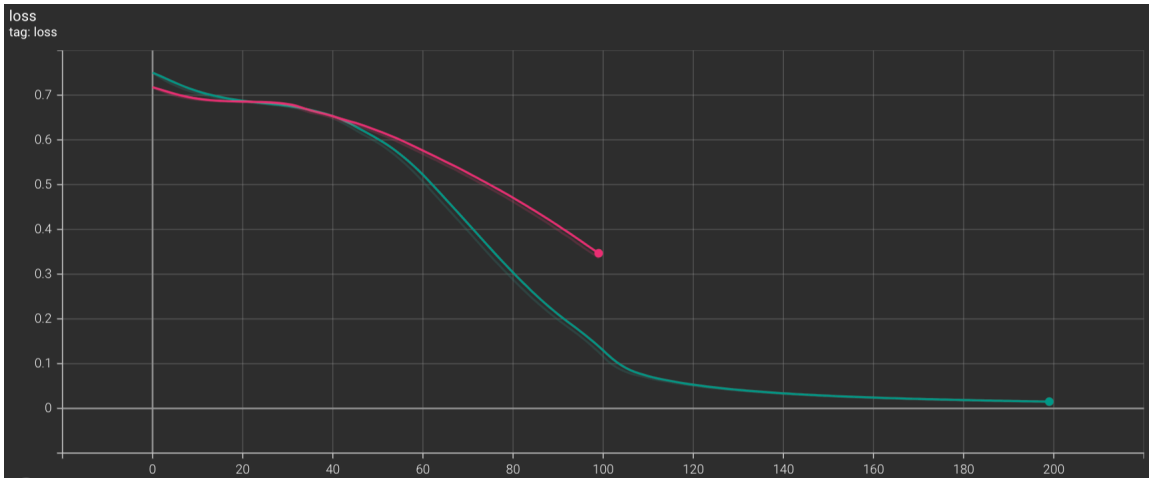

Figure 3.22: Loss with 4-dimension hidden layers.

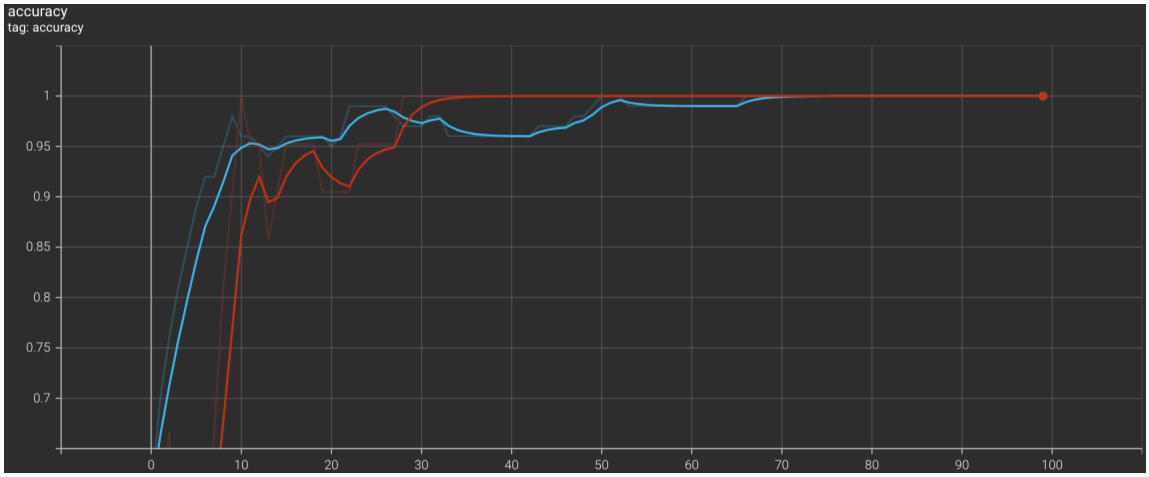

Figure 3.23: Accuracy with 16-dimension hidden layers.

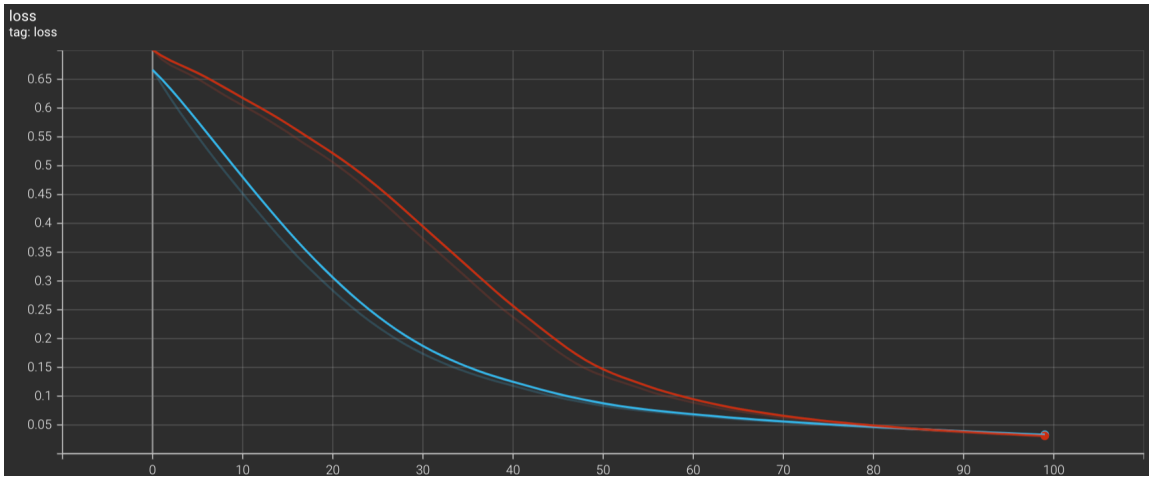

Figure 3.24: Loss with 16-dimension hidden layers.

Chapter 4

Conclusion

After trying so many experiments, I found some patterns for this task:

- `Adam` optimizer is **much better** than `SGD` optimizer.
- `LeakyReLU` is **a little better** than `ReLU`.
- Learning rate 10^{-2} is great, too large or too small is hard to converge.
- **Larger** dimension of hidden layers converge faster.