RDFIA TP1-ab
Introduction to Neural Networks

Jinyi WU

October 2024

1 Theoretical foundation

1.1 Supervised dataset

Question 1. What are the train, val and test sets used for?

In supervised learning, the data is typically split into three sets: train set, validation set, and test set.

- **Train Set**: The train set contains both features and target variables. It is used to train the model by learning the mapping between features and target variables. During training, the model parameters are updated by minimizing the loss function to get a well-trained model.
- **Validation Set**: The validation set also contains features and target variables, but it is not used to update the model's parameters. Instead, it is used during the training process to adjust hyper-parameters (such as learning rate) and select the best model. This helps to prevent over-fitting, where the model performs well on the training data but poorly on unseen data.
- **Test Set**: The test set contains only features. The trained model, after being tuned using the train and validation sets, is used to predict the target variables of the test set. The performance on the test set evaluates the model's generalization ability.

Question 2. What is the influence of the number of examples N?

In general, the dataset size N is typically split into train set, validation set, and test set in a certain proportion. When N is large, the model benefits from a larger training sample, which leads to better performance. Conversely, if N is small, the model may suffer from insufficient data, resulting in poorer performance.

1.2 Network architecture (forward)

Question 3. Why is it important to add activation functions between linear transformations?

First, activation functions can introduce non-linearity between features and target variables, leading to better learning outcomes.

Second, specific activation functions can control the output range of the model. For example, the output range of the tanh function is $[-1, 1]$, which makes the output more centralized. The SoftMax function has an output range of $[0, 1]$, and the sum of all outputs equals 1, which can effectively simulate a probability distribution.

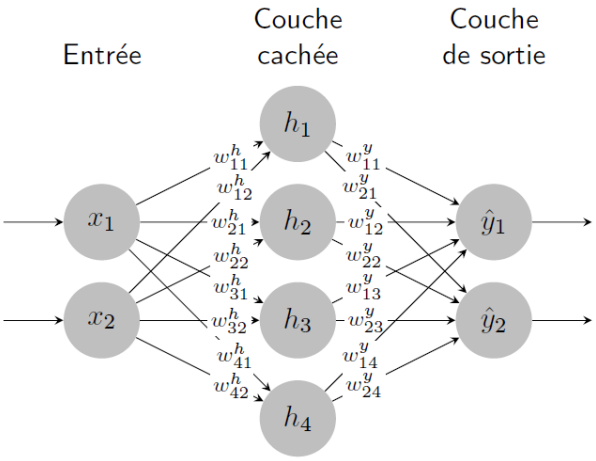


Figure 1: Neural network architecture with only one hidden layer.

Question 4. What are the sizes n_x, n_h, n_y in the figure 1? In practice, how are these sizes chosen?

In the figure 1, the sizes are

$$n_x = 2, \quad n_h = 4, \quad n_y = 2$$

In practice, n_x represents the dimension of the input layer, which is the number of features for each sample. n_h represents the dimension of the hidden layer, and n_y represents the dimension of the output layer, which is the number of targets(in classification tasks, it is always the number of classes).

Question 5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?

$\hat{y}$ represents the predicted value output by the model after inputting features, while y represents the given true value of target. In simple terms, the result of $\hat{y} - y$ is the error of the model.

Question 6. Why use a SoftMax function as the output activation function?

The SoftMax function is defined as follows:

$$SoftMax(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n_x} e^{x_j}}$$

In multi-class classification problems, the `SoftMax` activation function converts raw scores (logits) into probabilities, producing a probability distribution over the target classes. This allows the model's output to be interpreted as the probability of each class.

Question 7. Write the mathematical equations to perform the forward pass of the neural network, i.e., to successively produce $\tilde{h}$, h, $\tilde{y}$, and $\hat{y}$, starting from x.

The parameters are defined as follows:

$$W^h = \begin{bmatrix} w^h_{11} & w^h_{12} & \dots & w^h_{1n_x} \\ w^h_{21} & w^h_{22} & \dots & w^h_{2n_x} \\ \vdots & \vdots & \ddots & \vdots \\ w^h_{n_h1} & w^h_{n_h2} & \dots & w^h_{n_hn_x} \end{bmatrix}_{n_h \times n_x}, b^h = [b^h_1, b^h_2, \dots, b^h_{n_h}]_{1 \times n_h}.$$

$$W^y = \begin{bmatrix} w^y_{11} & w^y_{12} & \dots & w^y_{1n_h} \\ w^y_{21} & w^y_{22} & \dots & w^y_{2n_h} \\ \vdots & \vdots & \ddots & \vdots \\ w^y_{n_y1} & w^y_{n_y2} & \dots & w^y_{n_yn_x} \end{bmatrix}_{n_y \times n_h}, b^y = [b^y_1, b^y_2, \dots, b^y_{n_y}]_{1 \times n_y}.$$

- Firstly, we use a single sample $x = [x_1, \dots, x_{n_x}]_{1 \times n_x}$.

The forward pass equations are as follows:

$$\begin{cases} \tilde{h} = xW^{h^T} + b^h \\ h = \tanh(\tilde{h}) \\ \tilde{y} = hW^{y^T} + b^y \\ \hat{y} = \text{SoftMax}(\tilde{y}) \end{cases}$$

- Then, we use a batch of sample, so we have the input matrix

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n_x} \\ x_{21} & x_{22} & \dots & x_{2n_x} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{Nn_x} \end{bmatrix}_{N \times n_x}$$

The forward pass equations becomes:

$$\begin{cases} \tilde{H} = XW^{h^T} + \text{repmat}_{N\text{raw}}(b_h) \\ H = \tanh(\tilde{H}) \\ \tilde{Y} = HW^{y^T} + \text{repmat}_{N\text{raw}}(b_y) \\ \hat{Y} = \text{SoftMax}_{\text{line}}(\tilde{Y}) \end{cases}$$

1.3 Loss function

Question 8. During training, we try to minimize the loss function. For cross-entropy and squared error, how must $\hat{y}_i$ vary to decrease the global loss function L?

To decrease the loss function using the gradient descent method, in each iteration we update $\hat{y}^{(t+1)}$ as follows:

$$\hat{y}^{(t+1)} = \hat{y}^{(t)} - \alpha \cdot \nabla_{\hat{y}} l(\hat{y}, y)$$

where α is the learning rate, and $\nabla_{\hat{y}} l(\hat{y}, y)$ is the gradient of the loss function $l(\hat{y}, y)$ with respect to $\hat{y}$.

- For the **cross-entropy**

$$l(y, \hat{y}) = -\sum_i y_i log \hat{y}_i$$
$$\nabla_{\hat{y}_i} l(y, \hat{y}) = -\frac{y_i}{\hat{y}_i}$$

so we should update $\hat{y}_i$ by $\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} + \alpha \frac{y_i}{\hat{y}_i^{(t)}}$

- For the **MSE**

$$l(y, \hat{y}) = \sum_i (y_i - \hat{y}_i)^2$$
$$\nabla_{\hat{y}_i} l(y, \hat{y}) = -2(y_i - \hat{y}_i)$$

so we should update $\hat{y}_i$ by $\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} + 2\alpha(y_i^{(t)} - \hat{y}_i^{(t)})$.

Question 9. How are these functions better suited to classification or regression tasks?

- The cross-entropy works better on classification tasks Take a binary classification problem for example, the loss function is defined as $l(y, \hat{y}) = -(y_1 \log \hat{y}_1 + y_2 \log \hat{y}_2)$. Assuming $y = (0, 1)$, as $\hat{y}$ gets closer to y, the loss function decreases rapidly, and the gradient approaches -1. This effectively avoids the vanishing gradient problem;

- The MSE works better on regression tasks In regression problems, the target variable is continuous. The Mean Squared Error (MSE) calculates the average squared difference between the true values and the predicted values. Larger errors contribute more significantly to the loss function, which encourages the model to minimize these errors.

Question 10. What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic, and online stochastic versions? Which one seems the most reasonable to use in the general case?
Classic Gradient Descent:

- **Advantage:** In each iteration, the gradient of the loss function is calculated using the entire training set to update the parameters, which ensures a very stable direction for gradient updates.
- **Disadvantage:** Each update requires calculating the gradient for the entire training set, which results in high computational cost and memory usage, especially when there are many samples in the training set, leading to low efficiency.

Mini-Batch Stochastic Gradient Descent:

- **Advantage:** In each iteration, a mini-batch of samples is used to traverse the entire dataset. Each update only uses a mini-batch of samples, which makes it faster compared to classic gradient descent, while maintaining a certain level of stability.
- **Disadvantage:** The gradient for each update is based on a mini-batch of samples, which introduces some noise.

Online Stochastic Gradient Descent:

- **Advantage:** In each iteration, only one sample is used to update the parameters, which results in a high update frequency and low memory requirements. It can quickly adapt to changes in data, making it suitable for online learning and scenarios where data is generated in real-time.
- **Disadvantage:** Since each update is based on a single sample, the gradient has large fluctuations, which may lead to unstable convergence or slower convergence speed.

Summary: Mini-batch stochastic gradient descent achieves both fast updates and stable convergence, making it suitable for large datasets and allowing GPU acceleration. Therefore, it is the most commonly used in practice.

Question 11. What is the influence of the learning rate η on learning?
The learning rate η is a hyper-parameter that controls the speed of updates during optimization, following the rule:

$$w = w - \eta \frac{\partial L(X, Y)}{\partial w}.$$

A larger learning rate η results in larger update steps, which may lead to divergence and prevent convergence. Conversely, a smaller learning rate η results in smaller update steps, which may slow down the convergence process. Therefore, it is important to choose an appropriate learning rate, and a common value is $\eta = 0.01$.

Question 12. Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the *loss* with respect to the parameters, using the naive approach and the *backprop* algorithm.
Consider a neural network with L hidden layers (figure 2). We compare the computational complexity using the naive approach and the backpropagation algorithm.

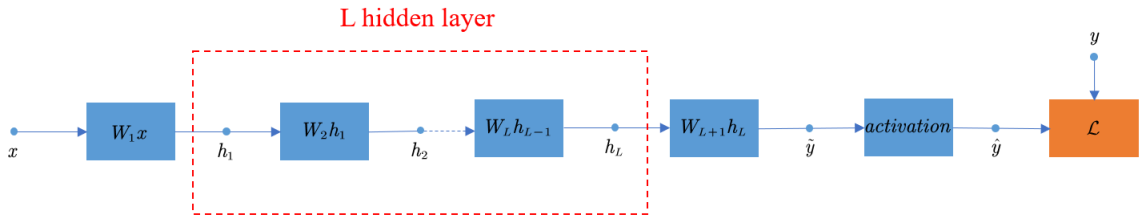


Figure 2: A neural network with L hidden layer

- **naive approach**
  Based on the chain rule for differentiation, we have:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{L+1}} &= \frac{\partial \mathcal{L}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial W_{L+1}} \\
\frac{\partial \mathcal{L}}{\partial W_L} &= \frac{\partial \mathcal{L}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial h_L} \cdot \frac{\partial h_L}{\partial W_L} \\
&\cdots \\
\frac{\partial \mathcal{L}}{\partial W_1} &= \frac{\partial \mathcal{L}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial h_L} \cdot \frac{\partial h_L}{\partial h_{L-1}} \cdots \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W_1}
\end{aligned}$$

  Since calculating the gradient for a given layer requires recomputing the derivatives from the output layer back to the current layer each time, the time complexity of the naive approach is $\mathcal{O}(n^L)$, where n is the number of neurons per layer, and L is the total number of layers.

- ***backprop* algorithm**

In this approach, we can avoid redundant computations by calculating gradients layer by layer and caching the intermediate results:

$$\begin{aligned}
\delta_{L+1} &= \frac{\partial \mathcal{L}}{\partial \tilde{y}}, & \frac{\partial \mathcal{L}}{\partial W_{L+1}} &= \delta_{L+1} \cdot \frac{\partial \tilde{y}}{\partial W_{L+1}} \\
\delta_L &= \delta_{L+1} \cdot \frac{\partial \tilde{y}}{\partial h_L}, & \frac{\partial \mathcal{L}}{\partial W_L} &= \delta_L \cdot \frac{\partial h_L}{\partial W_L} \\
&\cdots & &\cdots \\
\delta_1 &= \delta_2 \cdot \frac{\partial h_2}{\partial h_1}, & \frac{\partial \mathcal{L}}{\partial W_1} &= \delta_1 \cdot \frac{\partial h_1}{\partial W_1}
\end{aligned}$$

The gradient is computed by propagating the error backward through each layer. Each layer only requires the gradient and input from the previous layer, without needing to recompute the chain rule derivatives for all preceding layers. Therefore, the time complexity is $\mathcal{O}(nL)$, which significantly improves the computational efficiency compared to the naive approach, making gradient calculation feasible in deep learning.

Question 13 . What criteria must the network architecture meet to allow such an optimization procedure ?

All functions(including the loss function, the active function and the Linear layer) must be differentiable with respect to the model's output, allowing the error to be propagated backward.

Question 14. The function SoftMax and the *loss* of *cross-entropy* are often used together and their gradient is very simple. Show that the loss can be simplified by:

$$\ell = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right).$$

We can simplify the loss by:

$$\begin{aligned}
\ell &= -\sum_i y_i log(\hat{y}_i) \\
&= -\sum_i y_i log(\frac{e^{\tilde{y}_i}}{\sum_i e^{\tilde{y}_i}}) \\
&= -\sum_i y_i \tilde{y}_i + \sum_i y_i log(\sum_i e^{\tilde{y}_i}) \\
&= -\sum_i y_i \tilde{y}_i + log(\sum_i e^{\tilde{y}_i}), \quad since \sum_i y_i = 1
\end{aligned}$$

Question 15 . Write the gradient of the *loss (cross-entropy)* relative to the intermediate output $\tilde{y}$

$$\frac{\partial \ell}{\partial \tilde{y}_i} = \cdots \quad \Rightarrow \quad \nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \cdots$$

Based on the result from the previous question, we take the derivative with respect to $\tilde{y}_i$:

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{y}_i} &= -y_i + \frac{e^{\tilde{y}_i}}{\sum_i e^{\tilde{y}_i}} \\
&= -y_i + \hat{y}
\end{aligned}$$

Then we have:

$$\nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix} = \hat{y} - y$$

Question 16. Using the *backpropagation*, write the gradient of the loss with respect to the weights of the output layer$\nabla_{W_y} \ell$. Note that writing this gradient uses $\nabla_{\tilde{y}} \ell$. Do the same for $\nabla_{b_y} \ell$.

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \cdots \quad \Rightarrow \quad \nabla_{W_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1 n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y 1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_y n_h}} \end{bmatrix} = \cdots$$

First we calculate $\frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$:

$$\begin{aligned}
\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} &= \frac{\partial([h \cdot W_y^T + b_y]_k)}{\partial W_{y,ij}} \\
&= \frac{\partial(\sum_{p=1}^{n_h} h_p W_{y,kp} + b_k)}{\partial W_{y,ij}} \\
&= \begin{cases} h_j, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases}
\end{aligned}$$

Then we have:

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \frac{\partial \ell}{\partial \tilde{y}_j}\frac{\partial \tilde{y}_j}{\partial W_{y,ij}} = (\hat{y}_i - y_i)h_j$$

$$\begin{aligned}
\nabla_{W_y}\ell &= \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_yn_h}} \end{bmatrix} \\
&= \begin{bmatrix} (\hat{y}_1 - y_1)h_1 & \cdots & (\hat{y}_1 - y_1)h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y})h_1 & \cdots & (\hat{y}_{n_y} - y_{n_y})h_{n_h} \end{bmatrix} \\
&= (\hat{y} - y)^T h
\end{aligned}$$

Question 17. Compute other gradients:$\nabla_{\tilde{h}}\ell, \nabla_{W_h}\ell, \nabla_{b_h}\ell$.

- **Compute $\nabla_{\tilde{h}}\ell$**
  Given that $\tanh'(x) = 1 - \tanh^2(x)$, therefore

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{h}_i} &= \sum_{k=1}^{n_y} \frac{\partial \ell}{\partial \tilde{y}_k}\sum_{j=1}^{n_h}\frac{\partial \tilde{y}_k}{\partial h_j}\frac{\partial h_j}{\partial \tilde{h}_i} \\
&= \sum_{k=1}^{n_y}(\hat{y}_k - y_k)\frac{\partial \tilde{y}_k}{\partial h_i}\frac{\partial h_i}{\partial \tilde{h}_i} \\
&= \sum_{k=1}^{n_y}(\hat{y}_k - y_k)\frac{\partial(\sum_{p=1}^{n_h} h_p W_{y,kp} + b_k)}{\partial h_i}(1 - h_i^2) \\
&= (1 - h_i^2)\sum_{k=1}^{n_y}(\hat{y}_k - y_k)W_{y,ki}
\end{aligned}$$

$$\nabla_{\tilde{h}}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{h}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{h}_{n_h}} \end{bmatrix} = \begin{bmatrix} (1 - h_1^2)\sum_{k=1}^{n_y}(\hat{y}_k - y_k)W_{y,k1} \\ \vdots \\ (1 - h_{n_h}^2)\sum_{k=1}^{n_y}(\hat{y}_k - y_k)W_{y,kn_h} \end{bmatrix} = (1 - h^2) \odot (\nabla_{\tilde{y}}\mathbf{W}^y)$$

- **Compute $\nabla_{W_h}\ell$**

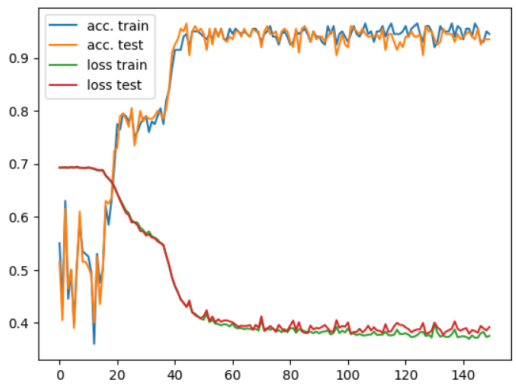
$$\begin{aligned}
\frac{\partial \ell}{\partial W_{h,ij}} &= \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial W_{h,ij}} \\
&= \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial(\sum_{p=1}^{n_x} h_p W_{h,kp} + b_k)}{\partial W_{h,ij}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_i}x_j \\
\nabla_{W_h}\ell &= \begin{bmatrix} \frac{\partial \ell}{\partial W_{h,11}} & \cdots & \frac{\partial \ell}{\partial W_{h,1n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{h,n_h1}} & \cdots & \frac{\partial \ell}{\partial W_{h,n_hn_x}} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{h}_1}x_1 & \cdots & \frac{\partial \ell}{\partial \tilde{h}_1}x_{n_x} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \tilde{h}_{n_h}}x_1 & \cdots & \frac{\partial \ell}{\partial \tilde{h}_{n_h}}x_{n_x} \end{bmatrix} \\
&= (\nabla_{\tilde{h}}\ell)^T x
\end{aligned}$$

- **Compute $\nabla_{b_h} \ell$**

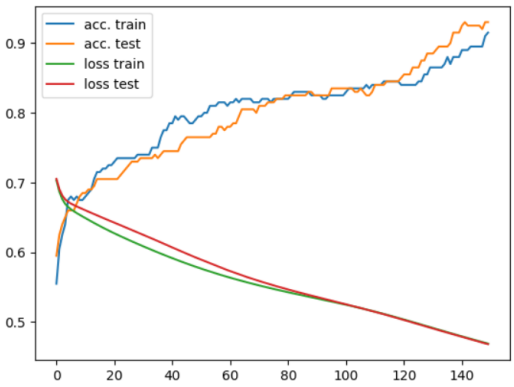
$$\begin{aligned}\frac{\partial \ell}{\partial b_{h,i}} &= \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} \\ &= \frac{\partial \ell}{\partial \tilde{h}_k} \\ \nabla_{b_h} \ell &= (\nabla_{\tilde{h}} \ell)^T\end{aligned}$$

2 Implementation

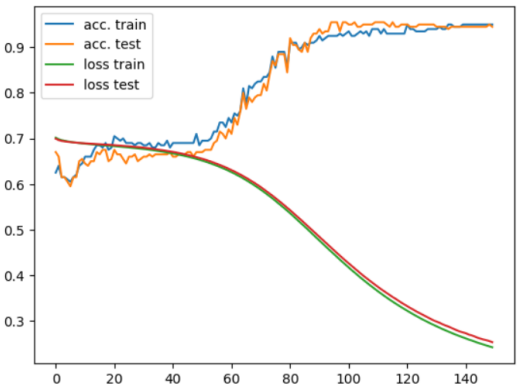
We apply different models to the CirclesData dataset and compare their performance.



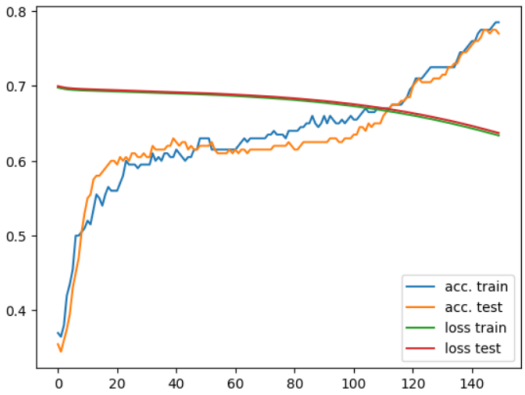
(a) Model 1 (Forward and backward manuals)



(b) Model 2 (backward pass with torch.autograd)



(c) Model 3 (forward pass with torch.nn layers)



(d) Model 4 (SGD with torch.optim)

Figure 3: Comparison of different models

As shown in the figure, the first model is the most stable, while the third model performs the best. The second and fourth models perform slightly worse, but the last three models are faster than the first one. We suspect that some internal optimizations in the torch library have caused these differences in results.
Finally, the fourth model was applied to the MNIST dataset, achieving the following results:

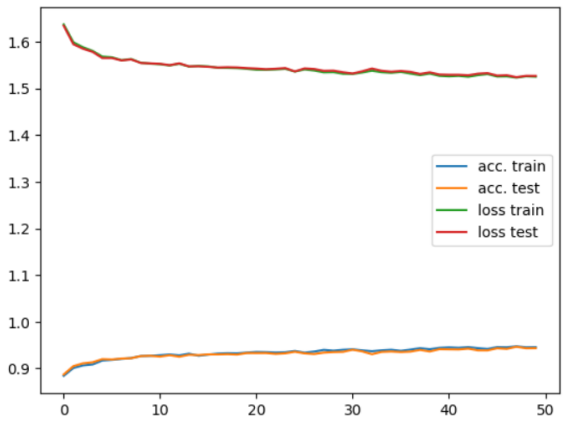


Figure 4: Model 4 (MINST dataset)

We also trained an SVM on the Circle dataset and observed that it performed well in this case, with faster execution than the neural network.

```
Accuracy : 94.00
```

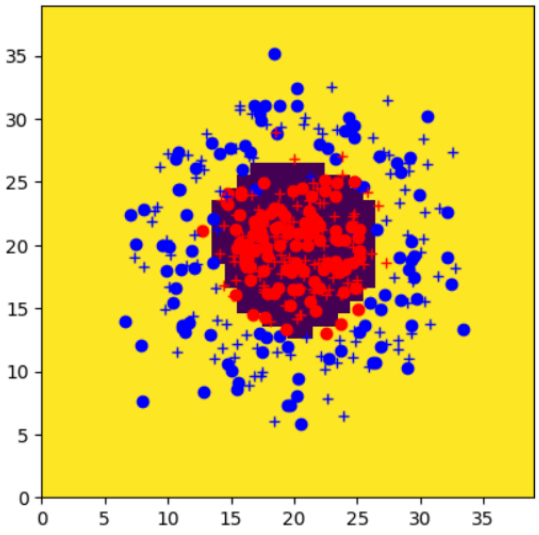


Figure 5: Enter Caption