TP 1-ab: Introduction to Neural Networks

CHAHINE Marilyn 21517286

MACQUET Jean 3532408

1 Theoretical Foundation

1.1 Supervised Dataset

1. The training set is used to train the model. The validation set is used to tune hyperparameters (such as the number of neurons in the hidden layer or the learning rate). The test set is used to evaluate the model on data that the model has never had access to during training, including their labels.

2. The number N corresponds to the number of available data points. The larger this number, the better the estimation of the parameters will be. Conversely, if it is small, the risk of overfitting increases.

1.2 Network Architecture

3. Activation functions introduce non-linearity into the model, enabling it to approximate non-linear relationships in the data. They also control the range of the outputs. For instance, tanh bounds the output between -1 and 1, while ReLU suppresses negative values. Without activation functions, all calculations in the network would be linear, so the final output would be linear.

4. In Figure 1, $n_x = 2$, $n_h = 4$, and $n_y = 2$. Here, n_x represents the number of input features, n_h the number of neurons in the hidden layer, and n_y the number of output units.

5. y represents the true label vector, and $\hat{y}$ represents the model's predicted output. For instance, in classification, y would be the class the input data should be classified into, and $\hat{y}$ is the probability of the input data belonging to each class. The difference between them (or similar formulas) represents the loss, which the network minimizes to learn.

6. The SoftMax function is used as a final activation layer to turn the final output or logits into a probability distribution, thus enabling proper interpretation of the output.

7. In order, the operations are as follows:

$$\tilde{h} = W_h x + b_h$$
$$h = \tanh(\tilde{h})$$
$$\tilde{y} = W_y h + b_y$$
$$\hat{y} = \text{softmax}(\tilde{y})$$

1.3 Loss Function

8. For the square loss, to decrease it, we must decrease $(y_i - \hat{y}_i)^2$ for all i, thus we must have $\hat{y}_i \to y_i$.

   For the cross-entropy loss, only the terms with $y_i = 1$ matter because the others are equal to zero. Thus, we aim to decrease $\log(1/\hat{y}_i)$, which means we must increase $\hat{y}_i$. But $\hat{y}_i$ has an upper bound of 1, thus $\hat{y}_i \to 1$ if $y_i = 1$, i.e., $\hat{y}_i \to y_i$ for all i because $(\hat{y}_i)_i$ is a probability vector.

9. Cross-entropy is better suited for classification because in this case, $y_i \in \{0, 1\}$, thus the sum in the cross-entropy loss is only made on non-zero y_i.

   Square loss is better suited for regression tasks because the square distance deals better with continuous values of the target.

1.4 Optimization Algorithm

10. Classic Gradient Descent:
    Advantages: Takes into account the whole dataset at the same time.
    Disadvantages: Memory and computation intensive, could get stuck in a local minimum because the large amount of data makes it rigid.

    Stochastic Gradient Descent by Minibatches:
    Advantages: Easy to compute (we can adapt the size of the batch to the available computing resources), can escape local minima due to randomness, generalizes better.
    Disadvantages: More iterations needed to converge.

    Online Stochastic Gradient Descent:
    Disadvantages: Hard to converge.
    Advantages: Easy to compute.

11. The learning rate has several influences. A large learning rate can cause large steps, which can lead to divergence or unstable descent. A small learning rate can cause slow learning and increase the chance of getting stuck in a local minimum.

12. Using the naive approach, we would have to compute the gradient along each direction of the parameters, and we would have to do it for each forward pass. If we do N forward passes with M parameters, we would have to do it $N \times M \times \lambda$ times, where M is the number of parameters, N the number of forward passes, and λ the cost of a forward pass. Using backpropagation, we only need to do the forward pass once and then the backward pass, which has the same cost as the forward pass. Thus, the total cost is $N \times 2 \times \lambda$.

13. To allow backpropagation, a network must only be made of differentiable operations, and the architecture must take into account the evolution of the gradient across the layers to avoid exploding or vanishing gradients.

14. $\hat{y} = \text{softmax}(\tilde{y})$ and $\hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}$. By substituting this expression into the cross-entropy formula, we get:

$$\begin{aligned}
\ell &= -\sum_i y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}\right) \\
&= -\sum_i y_i \tilde{y}_i + \left(\sum_i y_i\right) \log\left(\sum_j e^{\tilde{y}_j}\right) \\
&= -\sum_i y_i \tilde{y}_i + \log\left(\sum_j e^{\tilde{y}_j}\right)
\end{aligned}$$

where we use the fact that $(y_i)_i$ is a probability vector in the last row.

15. Computing the partial derivative of ℓ with respect to $\tilde{y}_k$ leads to:

$$\frac{\partial \ell}{\partial \tilde{y}_k} = -y_k + \frac{e^{\tilde{y}_k}}{\sum_j e^{\tilde{y}_j}} = \hat{y}_k - y_k$$

Thus,

$$\nabla_{\tilde{y}} \ell = \hat{y} - y$$

16. Thanks to question 15, we have:

$$\nabla_{\tilde{y}} \ell = \hat{y} - y$$

and since $\tilde{y}_k = \sum_r W_{y,kr} h_r + b_{y,k}$, we have:

$$\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = h_j \text{ if } k = i, \text{ else } 0.$$

Thus,

$$\begin{aligned}\frac{\partial \ell}{\partial W_{y,ij}} &= \sum_k \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} \\ &= \sum_k (\nabla_{\tilde{y}} \ell)_k \cdot \mathbb{1}_{i=k} h_j \\ &= (\nabla_{\tilde{y}} \ell)_i \cdot h_j\end{aligned}$$

Thus, $\nabla_{W_y} \ell$ is a matrix whose components are the product of the components of $\nabla_{\tilde{y}} \ell$ and h. Therefore,

$$\nabla_{W_y} \ell = \nabla_{\tilde{y}} \ell \, h^\top = (\hat{y} - y) \, h^\top$$

17. tanh is computed for each component independently, thus

$$\begin{aligned}\frac{\partial \ell}{\partial \tilde{h}_k} &= \frac{\partial \ell}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_k} \\ &= \left(\sum_i \frac{\partial \ell}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial h_k} \right) (1 - h_k^2) \\ &= \left(\sum_i (\hat{y} - y)_i W_{y,ik} \right) (1 - h_k^2) \\ &= (W_y^\top (\hat{y} - y))_k \, (1 - h_k^2)\end{aligned}$$

Using $*$ to represent the component-wise product of two vectors of the same size, we get:

$$\nabla_{\tilde{h}} \ell = W_y^\top (\hat{y} - y) * (1 - h * h)$$

We can now compute the remaining gradients with respect to W_h and b_h. Since $\tilde{h} = W_h x + b_h$, we have, for each component:

$$\frac{\partial \tilde{h}_i}{\partial (W_h)_{ij}} = x_j \quad \text{and} \quad \frac{\partial \tilde{h}_i}{\partial (b_h)_i} = 1.$$

Therefore:

$$\begin{aligned}\nabla_{W_h} \ell = \frac{\partial \ell}{\partial W_h} &= \frac{\partial \ell}{\partial \tilde{h}} \frac{\partial \tilde{h}}{\partial W_h} = (\nabla_{\tilde{h}} \ell) \, x^\top, \\ \nabla_{b_h} \ell = \frac{\partial \ell}{\partial b_h} &= \nabla_{\tilde{h}} \ell.\end{aligned}$$

For a batch of size B, if we denote by $\nabla_{\tilde{H}} \ell \in \mathbb{R}^{B \times n_h}$ the matrix of gradients for each example and $X \in \mathbb{R}^{B \times n_x}$ the batch of inputs, the corresponding averaged gradients are:

$$\nabla_{W_h} \ell = \frac{1}{B} (\nabla_{\tilde{H}} \ell)^\top X, \quad \nabla_{b_h} \ell = \frac{1}{B} \sum_{t=1}^{B} \nabla_{\tilde{h}}^{(t)}.$$

2 Implementation

We can compare the four loss and accuracy curves resulting from the four methods implemented: `manual`, with `autograd`, with `torch.nn`, and with `torch.optim`. The graphs are shown in Figure 1

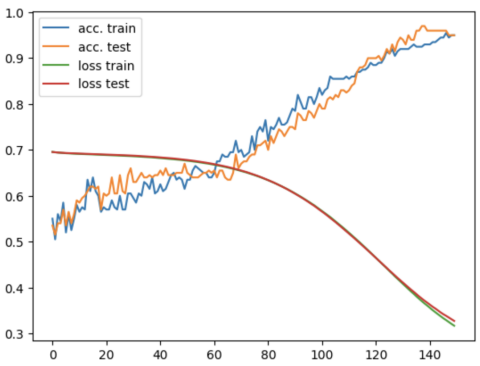
(a) `manual`: Everything is done with matrix calculations. The training and test accuracy are correct, but slow. This is due to the accumulated error in the calculation of the gradient.

(b) `autograd`: We replace the manual backward pass with PyTorch's autograd. We observe fast convergence: gradients are computed automatically and precisely.

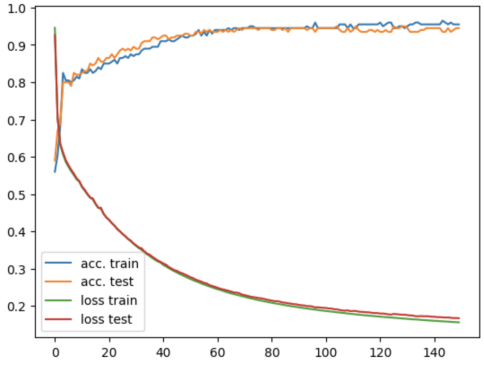
(c) `torch.nn`: The results are the same as before but even better, although the code is simpler. We can note that in this figure, the loss decreases slower than before because we did not initialize the weights by multiplying by 0.3.

(d) `torch.optim`: The convergence is even faster and smoother than before: the stochastic gradient descent is standardized and optimized.

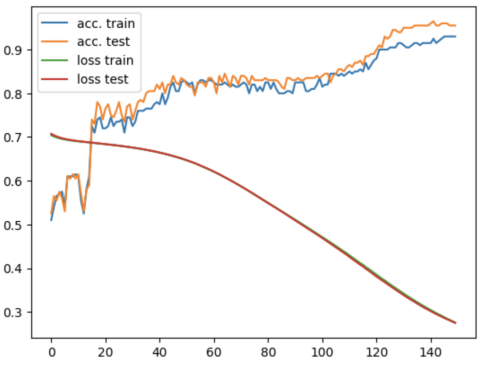
Figure 1: **Accuracy and Loss of the Different Implementations**



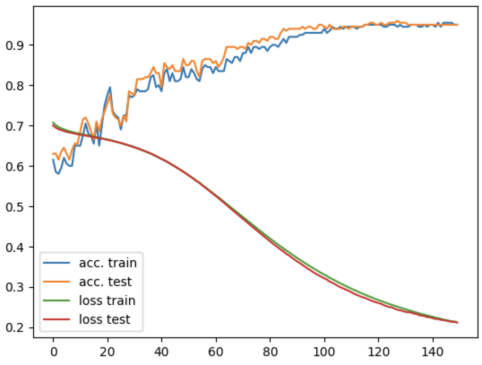
(a) Part 1: Manual



(b) Part 2: Torch Autograd



(c) Part 3: Torch NN



(d) Part 4: Torch Optim