CS587 - Neural Networks & Learning of Hierarchical Representation

Spring Semester 2024
Assignment 2

Papageridis Vasileios - 4710
csd4710@csd.uoc.gr

April 14, 2024

Part 1: Linear Regression

Plot 1 shows the loss of the model over 100 iterations. It starts at a relatively high value and quickly decreases, leveling off as the number of iterations increases. The steep drop at the beginning suggests that the model is rapidly learning from the training data. The flattening of the curve indicates that the model is approaching a minimum loss, where further iterations do not significantly change the value of the loss function. This is a sign that the model might have reached convergence or is very close to it.

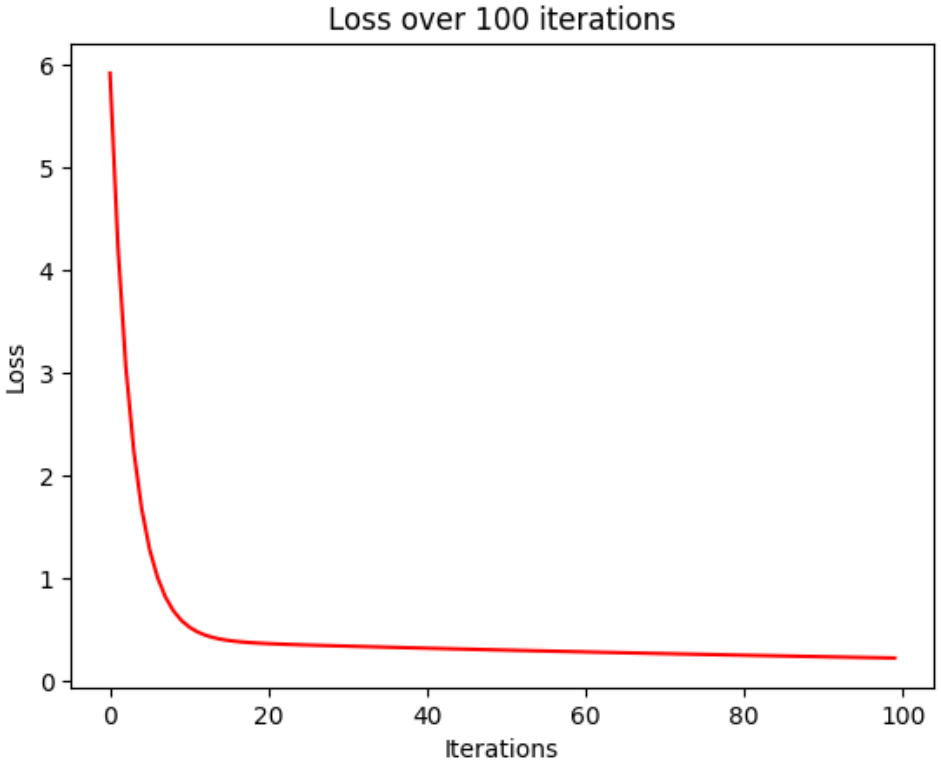

Figure 1: Loss over 100 iterations.

Plot 2 displays the model visualization with the actual data points and the fitted line resulting from training. The data points are in blue, and they lie along a declining trend. The red line represents the model's prediction. Upon closer examination of the plot, it is evident that the

fitted line does not perfectly capture the positions of the actual data points. This suggests that while the model has learned a linear relationship between x and y, there is still a noticeable error between the predictions and the actual values.

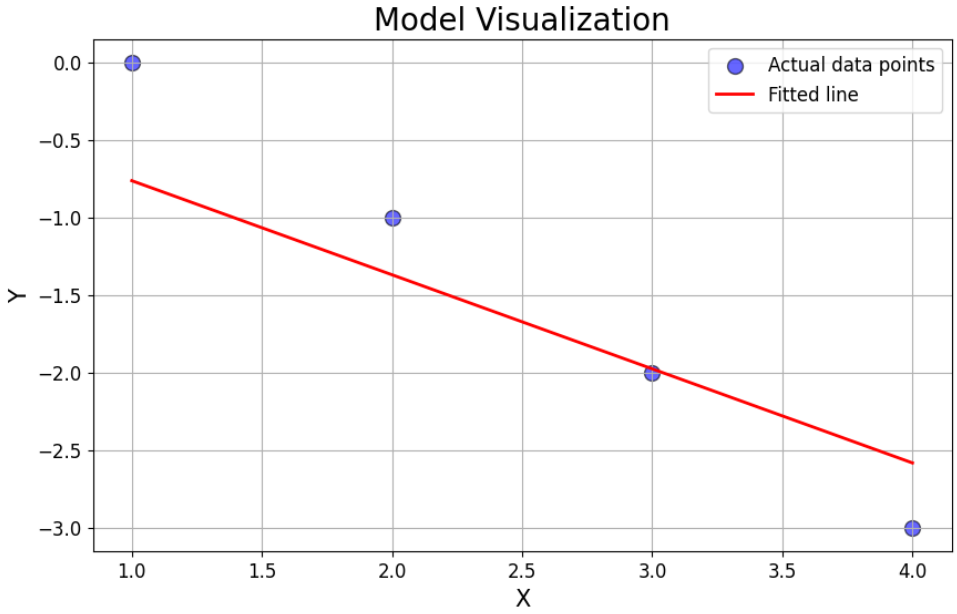

Figure 2: Model Visualization with the fitted line and actual data points, indicating a discrepancy between the model's predictions and the actual values.

Although the loss in Plot 1 has decreased significantly over the 100 iterations, the fact that the fitted line does not pass through all the actual data points in Plot 2 indicates that the model could be improved. The discrepancy suggests that either the model has not fully converged, the linear model is too simple, or the training process could be adjusted to achieve a better fit. This could include more iterations, a different learning rate, or a more complex model.
The initialization of the model with weights of 0.3 and bias of -0.3 and the training with a learning rate of 0.01 for 100 iterations seems to have been insufficient for this task. It might be beneficial to continue the training for more iterations, consider a smaller learning rate to allow finer adjustments to the weights, or the creation of models that could capture the data trend better.

Questions

Q1: In PyTorch, the computational graph is dynamic. Can you explain what this means in terms of how the graph is constructed and utilized?

In PyTorch, a dynamic computational graph means that the graph of operations (where each operation is a node and data flows between nodes along edges) is built on-the-fly during runtime. As we execute operations in the code, PyTorch dynamically creates and adds nodes to the graph, and this graph is different for every run of the program. This is particularly useful for models where the operations can change dynamically with each input (like varying the number of iterations in a loop based on input).

Q2:How does a dynamic computational graph compare to static graphs? What are the advantages and disadvantages of each approach?

Dynamic Graphs:

- *Advantages:* They are flexible, allowing for easy modification of the graph during runtime. This is useful for models that have conditional computations or loops that depend on input data.
- *Disadvantages:* Can be less efficient compared to static graphs as the graph needs to be built from scratch at each run, which might add overhead.

Static Graphs:

- *Advantages:* They can be optimized upfront since the graph is defined and compiled before execution, potentially leading to better performance and efficiency.
- *Disadvantages:* Less flexible in handling dynamic changes within the graph; modifications require redefining and recompiling the graph.

Q3:Why did we use w.grad.zero () and b.grad.zero () in our code? Are they necessary for proper program execution?

In training neural networks, gradients accumulate by default with each backpropagation step in PyTorch. This means that if `w.grad.zero_()` and `b.grad.zero_()` were not called, the gradient computed from the loss function in each training step would be added to the gradients accumulated from the previous steps, leading to incorrect gradient values. Therefore, clearing the gradients after each update (using `zero_()` methods) is necessary to ensure that each training step computes gradients from a clean state.

Q4:Why did we use torch.no grad in our code?

`torch.no_grad()` is used to block the tracking of gradients in the autograd system. This is useful in scenarios where we are sure that we will not call `Tensor.backward()`. It is particularly important during inference or evaluation of the model when we are not updating weights and biases. This context manager reduces memory usage and speeds up computations since gradients are not computed or stored.

1 Part 2: Computational graphs and custom gradients

A) Theory

Backpropagation

Backpropagation is an algorithm used for optimizing the parameters of a neural network model. Fundamentally, backpropagation is used to compute the gradient of a loss function with respect to each weight in the network by applying the so called chain rule in a multi-layer network structure. This process allows for the efficient calculation of gradients layer by layer from the output end (where the loss is computed) to the input end of the network. This systematic propagation of error information helps to update the weights in such a way that the overall network error is minimized over time through iterative adjustments.
Mathematically, if L is the loss function, and w represents the weights of the network, backpropagation computes $\frac{\partial L}{\partial w}$, which is essential for performing gradient descent or any of its variants.

Local Gradients

Local gradients in neural networks refer to the derivatives of the neuron activations prior to the application of the activation function with respect to their net input. These gradients represent the immediate rate of change of the loss function with respect to the pre-activation outputs at each neuron, and they are used locally at each neuron during the backpropagation process.
For a neuron with activation function σ and pre-activation z, if the post-activation output is $a = \sigma(z)$, the local gradient would be $\frac{\partial \sigma}{\partial z}$. This derivative forms part of the product chain used to calculate the gradient of the loss with respect to the weights and biases directly associated with each neuron.

Upstream Gradient

Upstream gradient refers to the gradient of the loss function propagated back from the output towards the input of the network during backpropagation. It quantifies the contribution of the neurons downstream (i.e., closer to the output layer) to the derivative of the loss function with respect to the neurons in the current layer or the input features. This gradient is termed ”upstream” because it flows in the opposite direction of the activation propagation during the forward pass.
The upstream gradient for a given layer l in a network is computed as a product of the local gradients at layers subsequent to l and the derivatives of the activations at layer l. This propagated gradient is a critical component in updating the parameters of layer l, as it essentially communicates how much a change in the output of layer l would impact the overall loss, allowing for targeted updates that more effectively reduce the loss.

B) Custom Functions

One-variable function

Let

$$y(x) = \frac{\sin(x)}{x} \tag{1}$$

The computational graph of the function is given below.

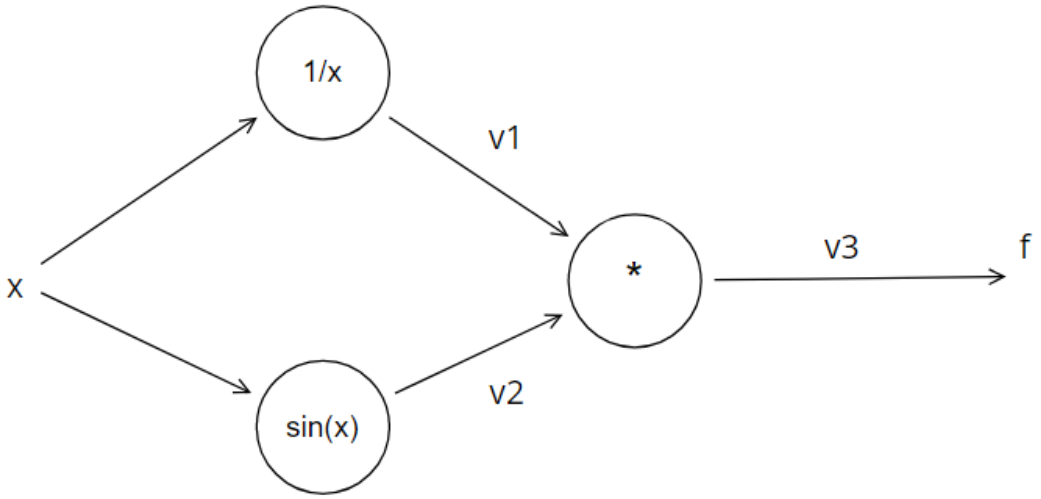

Figure 3: Computational graph of the function $y(x)$.

During the forward pass:

$$v_1 = \frac{1}{x}$$
$$v_2 = \sin(x)$$
$$v_3 = v_1 \cdot v_2 = \frac{\sin(x)}{x}$$

Calculating the backward pass:
We define:

$$\overline{v} = \frac{\partial f}{\partial v}$$

$$\overline{v_3} = \frac{\partial v_3}{\partial v_3} = 1$$
$$\overline{v_1} = \frac{\partial v_3}{\partial v_1} \cdot \overline{v_3} = v_2 \cdot \overline{v_3} = sinx \cdot 1 = sinx$$
$$\overline{v_2} = \frac{\partial v_3}{\partial v_2} \cdot \overline{v_3} = v_1 \cdot \overline{v_3} = \frac{1}{x} \cdot 1 = \frac{1}{x}$$
$$\overline{x} = \frac{\partial v_1}{\partial x} \cdot \overline{v_1} + \frac{\partial v_2}{\partial x} \cdot \overline{v_2} = -\frac{1}{x^2} \cdot \overline{v_1} + cosx \cdot \overline{v_2} = -\frac{1}{x^2} \cdot sinx + cosx \cdot \frac{1}{x} = \frac{xcosx - sinx}{x^2}$$

Two-variable function

Let

$$f(x, y) = ax^2 + bxy + cy^2 \tag{2}$$

The computational graph of the function is given below.

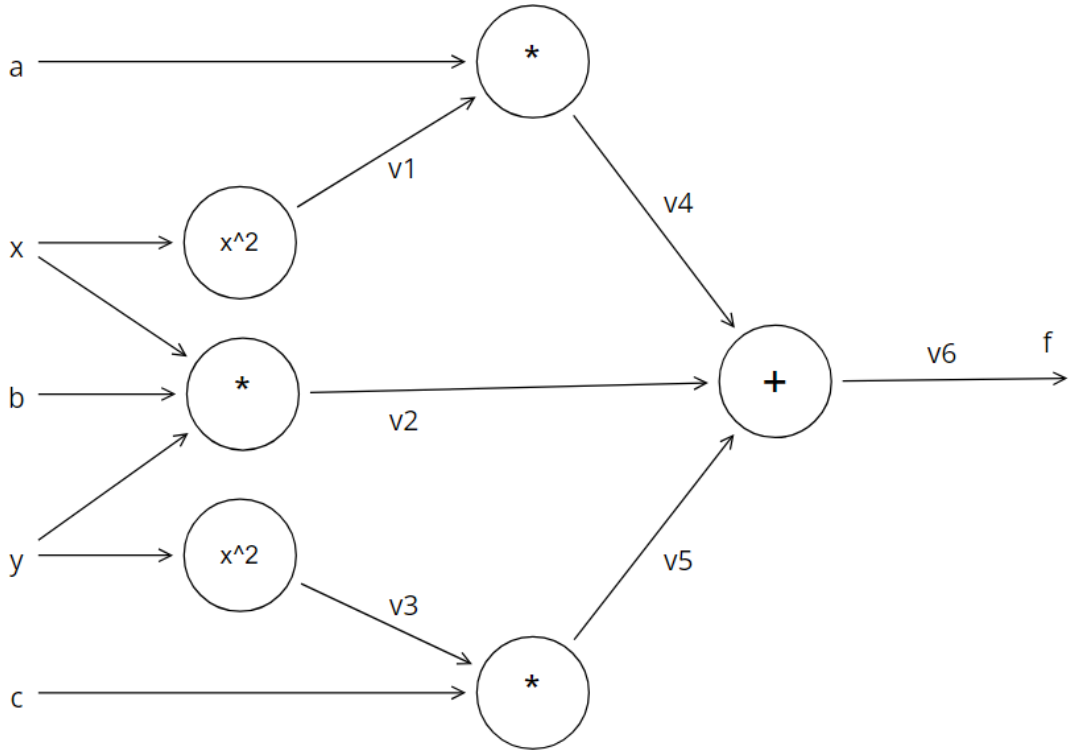

Figure 4: Computational graph of the function $f(x, y)$.

During the forward pass:

$$\begin{aligned}
v_1 &= x^2 \\
v_2 &= bxy \\
v_3 &= y^2 \\
v_4 &= av_1 = ax^2 \\
v_5 &= cv_3 = cy^2 \\
v_6 &= v_4 + v_2 + v_5 = ax^2 + bxy + cy^2
\end{aligned}$$

Calculating the backward pass:
We define:

$$\overline{v} = \frac{\partial f}{\partial v}$$

$$\overline{v_6} = \frac{\partial v_6}{\partial v_6} = 1$$
$$\overline{v_4} = \frac{\partial v_6}{\partial v_4} \cdot \overline{v_6} = 1 \cdot 1 = 1$$
$$\overline{v_5} = \frac{\partial v_6}{\partial v_5} \cdot \overline{v_6} = 1 \cdot 1 = 1$$
$$\overline{v_2} = \frac{\partial v_6}{\partial v_2} \cdot \overline{v_6} = 1 \cdot 1 = 1$$
$$\overline{v_3} = \frac{\partial v_5}{\partial v_3} \cdot \overline{v_5} = c \cdot 1 = c$$
$$\overline{v_1} = \frac{\partial v_4}{\partial v_1} \cdot \overline{v_4} = a \cdot 1 = a$$
$$\overline{x} = \frac{\partial v_1}{\partial x} \cdot \overline{v_1} + \frac{\partial v_2}{\partial x} \cdot \overline{v_2} = 2ax + by \cdot 1 = 2ax + by$$
$$\overline{y} = \frac{\partial v_3}{\partial y} \cdot \overline{v_3} + \frac{\partial v_2}{\partial y} \overline{v_2} = 2y \cdot c + bx \cdot 1 = bx + 2cy$$

Second-order derivative

Let

$$f(x) = ln(1 + e^x) \tag{3}$$

The computational graph of the function is given below.

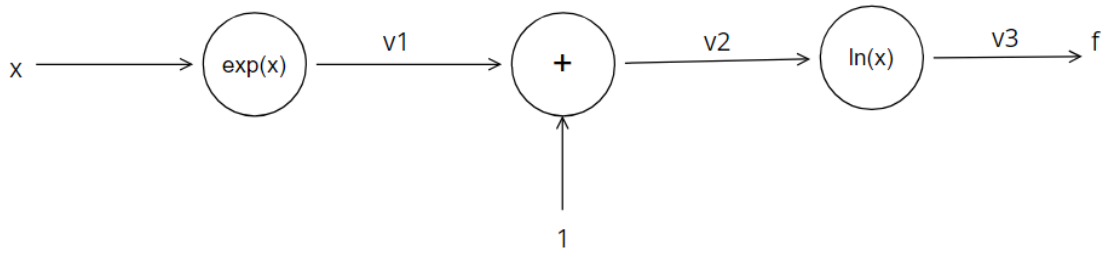

Figure 5: Computational graph of the function $f(x)$.

Computation of the first-order derivative

During the forward pass:

$$v_1 = e^x$$
$$v_2 = v1 + 1 = e^x + 1$$
$$v_3 = ln(v_2) = ln(e^x + 1)$$

Calculating the backward pass:

We define:

$$\overline{v} = \frac{\partial f}{\partial v}$$

$$\overline{v_3} = \frac{\partial v_3}{\partial v_3} = 1$$
$$\overline{v_2} = \frac{\partial v_3}{\partial v_2} \cdot \overline{v_3} = \frac{1}{v_2} \cdot 1 = \frac{1}{v_2}$$
$$\overline{v_1} = \frac{\partial v_2}{\partial v_1} \cdot \overline{v_2} = 1 \cdot \frac{1}{v_2} = \frac{1}{v_2}$$
$$\overline{x} = \frac{\partial v_1}{\partial x} \cdot \overline{v_1} = e^x \cdot \frac{1}{v_2} = \frac{e^x}{1+e^x}$$

Computation of the second-order derivative In order to compute the second order derivative of the function, we have to construct the computational graph of the first-order derivative. After this step we will use again backpropagation and we will define the forward and backward pass, in order to get the second-order derivative as the final result.
The first-order derivative is given as:

$$\frac{e^x}{1+e^x}$$

and the corresponding computational graph is the given below:

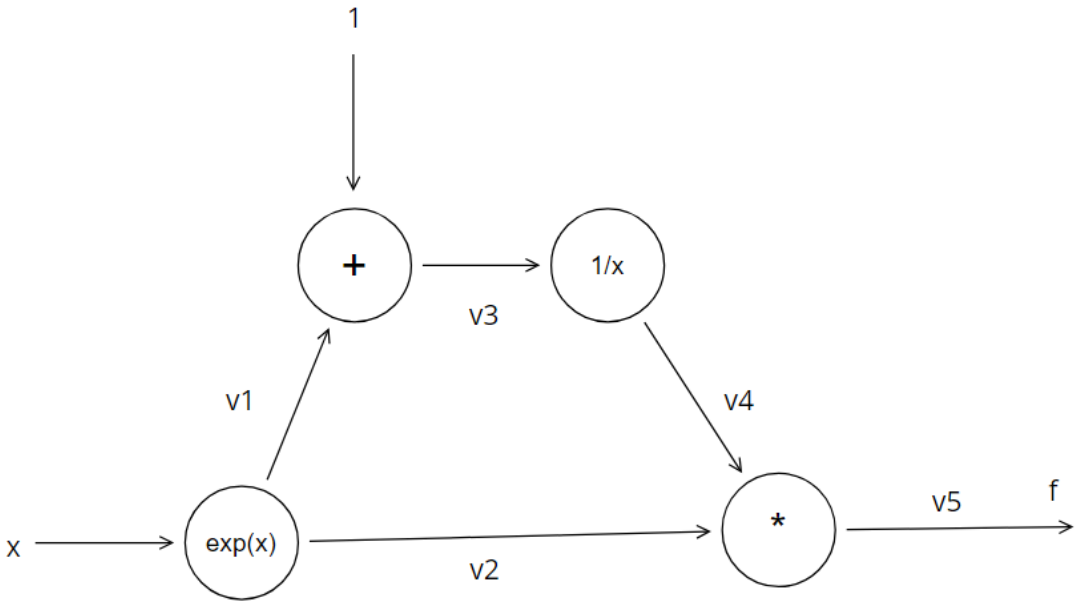

Figure 6: Computational graph of the first-order derivative of the function $f(x)$.

Computation of the first-order derivative

During the forward pass:

$$
\begin{aligned}
v_1 &= e^x \\
v_2 &= e^x \\
v_3 &= v_1 + 1 = e^x + 1 \\
v_4 &= \frac{1}{v_3} = \frac{1}{e^x + 1} \\
v_5 &= v_4 \cdot v_2 = \frac{e^x}{e^x + 1}
\end{aligned}
$$

Calculating the backward pass:
We define:

$$\overline{v} = \frac{\partial f}{\partial v}$$

$$
\begin{aligned}
\overline{v_5} &= \frac{\partial v_5}{\partial v_5} = 1 \\
\overline{v_4} &= \frac{\partial v_5}{\partial v_4} \cdot \overline{v_5} = v_2 \cdot 1 = v2 = e^x \\
\overline{v_2} &= \frac{\partial v_5}{\partial v_2} \cdot \overline{v_5} = v_4 \cdot 1 = \frac{1}{e^x + 1} \\
\overline{v_3} &= \frac{\partial v_4}{\partial v_3} \cdot \overline{v_4} = -\frac{1}{v_3^2} \cdot e^x = -\frac{1}{(e^x + 1)^2} \cdot e^x \\
\overline{v_1} &= \frac{\partial v_3}{\partial v_1} \cdot \overline{v_3} = 1 \cdot (-\frac{1}{v_3^2}) \cdot e^x = -\frac{1}{v_3^2} \cdot e^x = -\frac{1}{(e^x + 1)^2} e^x \\
\overline{x} &= \frac{\partial v_1}{\partial x} \cdot \overline{v_1} + \frac{\partial v_2}{\partial x} \cdot \overline{v_2} = e^x \cdot (-\frac{e^x}{(e^x + 1)^2}) + e^x \cdot \frac{1}{e^x + 1} = e^x (\frac{1}{e^x + 1} - \frac{e^x}{(e^x + 1)^2}) \Rightarrow \\
\overline{x} &= e^x \frac{1}{(e^x + 1)^2} = \frac{e^x}{(e^x + 1)^2}
\end{aligned}
$$

References

[1] Simon J.D. Prince, *Understanding Deep Learning*, MIT Press, 2023, https://udlbook.github.io/udlbook/.

[2] Sebastian Nowozin and Christoph H. Lampert, *Structured Learning and Prediction in Computer Vision*, 2012, https://www.nowozin.net/sebastian/cvpr2012tutorial/.

[3] PyTorch, *Extending PyTorch with custom C++ extensions*, https://pytorch.org/tutorials/intermediate/custom_function_double_backward_tutorial.html.