RDFIA Reports: tp 1, 2 and 3

Titouan Guerin & Yacine Chettab

October 13, 2025

1 Theoretical foundation

1.1 Supervised Dataset

Questions

1. ⋆ **What are the train, val and test sets used for?**
   The train set is used to train the model, the validation set is used to tune the hyperparameters during the training of the model, and the test set is used to evaluate the model's performance (data that is not in the train set, so not seen by the model previously).

2. **What is the influence of the number of examples N?**
   The number of examples N influences the model's ability to generalize, which means that by seeing enough examples, the model can learn to make predictions on unseen data. On the other hand, if N is too little, then the model will do more of a memorization of the training data, so it does not actually learn to generalize the problem. However, having too many samples N can be a problem in terms of training time and computational resources.

1.2 Network Architecture (forward)

Questions

3. **Why is it important to add activation functions between linear transformations?**
   Having no activation functions between the linear layers would make the whole network linear (which is not what we are looking for). Chaining linear transformations is the same as having one linear layer but with just more weights. So the activation functions allow the network to learn non-linear patterns in the data.

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

   - n_x is the size of the input layer, the number of features for one data point,
   - n_h is the size of the hidden layer, a hyperparameter that is determined before training,
   - n_y is the size of the output layer, the number of classes for classification problems.

5. **What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?**
   The vector $\hat{y}$ represents the predicted output of the model, while y represents the true labels.

6. **Why use a SoftMax function as the output activation function?**
The SoftMax function maps the output of the neural network to a probability distribution. This means that all the values outputted by the model are mapped between 0 and 1, and the sum of all the values are equal to 1: $\sum_{i=1}^{n} \hat{y}_i = 1$.

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

   - $\tilde{h} = W_h x + b_h$
   - $h = \text{ReLU}(\tilde{h})$
   - $\tilde{y} = W_y h + b_y$
   - $\hat{y} = \text{SoftMax}(\tilde{y})$

1.3 loss function

Questions

8. **During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function $\mathcal{L}$?**
For the MSE, the prediction $\hat{y}$ is compared directly to y, so to minimize the loss, we want the difference between those two values to be as small as possible. In this case we work with real numerical values. This loss works for both regression and classification problems.
On the other hand, for the cross-entropy, we are working exclusively with probabilities, so to minimize the loss, we want the predicted probability $\hat{y}_i$ of the true class to be as close to 1 as possible, (and the predicted probabilities of the other classes to be as close to 0 as possible). Here y is just a probability vector, with just zeros except for the true class which is 1.

9. **How are these functions better suited to classification or regression tasks?**
MSE is naturally suited to regression, since its gradient is linear in the prediction error, providing a good measure of how far the prediction is from the target. However, in classification this linear behavior is less effective, because it does not properly penalize cases where the model is very confident but wrong (not enough at least).

   Cross-entropy, on the other hand, has a gradient that grows rapidly when the predicted probability for the correct class is small. Basically you want to penalize a confident but wrong prediction because it impacts the other predicted probabilities. Since the penalty (or cost) of CE is logarithmic, and hence more harsh on wrong predictions, the gradient of the loss is larger and allows the model to learn faster.

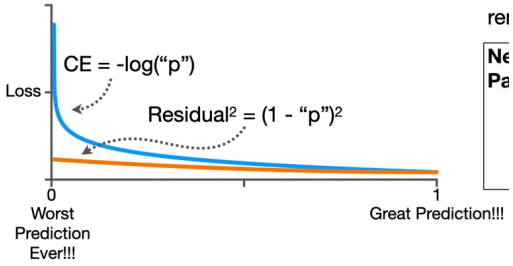

Figure 1: Cross-Entropy vs MSE derivative comparison.

1.4 Optimization algorithm

Questions

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?**

Method	Advantages	Disadvantages
Batch GD	Stable convergence, accurate gradients	High memory usage, slow & expensive for large datasets, possible local minimum
Mini-batch SGD	faster convergence, no local minima, better 'exploration' of the dataset	batch size tuning, noise
Online SGD	Fast updates, low memory usage	Noisy gradients, less stable convergence

Overall it is pretty safe to conclude that Mini-batch SGD is the most reasonable method to use in the general case, because it essentially is a mix of the two other methods: faster convergence and lower computational cost.

11. ⋆ **What is the influence of the learning rate η on learning?**
Recap: the learning rate η is a hyperparameter that controls how strongly the weights are updated during training.
A high learning rate means that the weights are updated strongly at each training step, which can lead to fast convergence but also to the overshooting of the optimal solution. Overshooting means that the model will diverge, or jump around the optimal solution without actually reaching it.
On the contrary, a low learning rate means that each update step changes the weights only slightly, which can lead to slow convergence. This can be helpful to explore the local minima, but can also lead to getting stuck there.
To address these drawbacks, one good technique is to change η over time. By starting with a high learning rate, and then decreasing it as we get closer to the optimal solution (over time), we allow the model to make bigger steps at the beginning of the training, and to then make smaller careful steps when we are close to the optimal solution.

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**
For the naive approach, the gradient of the loss is calculated for each parameter independently. For example, for the weights of the second layer, we would calculate the gradient of the loss with respect to each weight w_{ij} in the weight matrix W_2. And for the gradients of the first layer, we would re-calculate the gradients of the second layer, and then compute the gradients with respect to the first layer. Even for a two-layer network, we can see that this approach is very computationally expensive, since we keep calculating the same gradients over and over.
With the backpropagation algorithm, each gradient is calculated only once. If we take our previous example with only two layers, after calculating the gradients of the second layer, we keep those gradients in memory and use them to calculate the gradients of the first layer. In essence, this means that all of the gradients are calculated in a single backward pass, without having to recompute the gradients of the loss like in the naive method.
This backpropagation algorithm is much more efficient and becomes mandatory for deep

networks with many layers.
Remark on the $\mathcal{O}$ complexity:
For a network with L layers, let n_l be the number of parameters in layer l.
Let t_{avg} = the time to calculate the gradient for one parameter,

- computing $\frac{\partial \mathcal{L}}{\partial W_l}$ for each layer separately (naive):

$$T_1 = t_{avg} \cdot \Big(\sum_{l=1}^{L} \sum_{k=l}^{L} n_k \Big)$$
$$T_1 = t_{avg} \cdot n \Big(\sum_{l=1}^{L} (L - l + 1) \Big)$$
$$T_1 = t_{avg} \cdot n(1 + 2 + \ldots + L) = n \frac{L(L+1)}{2}$$
$$T_1 = L^2 \cdot t_{avg} \cdot n_{avg} \sim \mathcal{O}(L^2)$$

  where n_{avg} is the average number of parameters per layer.

- For the backpropagation algorithm:

$$T_2 = t_{avg} \cdot \Big(\sum_{l=1}^{L} n_l \Big)$$
$$T_2 = t_{avg} \cdot (n + n + \ldots + n)$$
$$T_2 = t_{avg} \cdot L \cdot n_{avg} \sim \mathcal{O}(L)$$

13. **What criteria must the network architecture meet to allow such an optimization procedure ?**
Each layer (linear, activation, cost ...) must be differentiable in order for the backpropagation to be possible. There also needs to be some form of memory to save every input and output of each layer during the forward pass as they are needed for the backward pass. Finally, the gradients produced must not be either extremely small or large, as they would stop the backpropagation (a.k.a the vanishing gradient problem) or the algorithm diverges (a.k.a the exploding gradient problem).

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:**

$$l = -\sum_i y_i \tilde{y}_i + \log(\sum_i e^{\bar{y}_i})$$

Recap: we write the cross-entropy loss as:

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

Here $\hat{y}_i$ is the output of the SoftMax function, so we can replace it in the loss:

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

Since y is the label vector, composed of zeros except for the true class which is 1, then we know that the sum of all the y_i is equal to 1, so:

$$
l = -\sum_i y_i \tilde{y}_i + \log(\sum_j e^{\tilde{y}_j})
$$

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

$$
\frac{\partial l}{\partial \tilde{y}_i} = \cdots \implies \nabla_{\bar{y}} l = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_n} \end{bmatrix} = \ldots
$$

We are trying to find:

$$
\frac{\partial l}{\partial \tilde{y}_i} = \sum_k \frac{\partial l}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial \tilde{y}_i}
$$

We need to compute $\frac{\partial l}{\partial \tilde{y}_k}$:

$$
\begin{aligned}
\frac{\partial l}{\partial \hat{y}_k} &= \frac{\partial}{\partial \hat{y}_k}\Big(-\sum_k y_k \log(\hat{y}_k)\Big) \\
&= -y_k \frac{\partial}{\partial \hat{y}_k}(\log(\hat{y}_k)) \\
&= -y_k \frac{1}{\hat{y}_k} \\
&= -\frac{y_k}{\hat{y}_k}
\end{aligned}
$$

Next we compute $\frac{\partial \hat{y}_k}{\partial \tilde{y}_i}$, which is the derivative of the SoftMax function.
The derivative of the softmax has two forms:
$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}}{\partial a_j} = \frac{e^{a_i} \sum - e^{a_j} e^{a_i}}{\sum^2} = \frac{e^{a_i}}{\sum} \frac{\sum - e^{a_j}}{\sum} = S_i(1 - S_j)$ if i = j and $\sum = \sum_{k=1}^N e^{a_k}$
$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}}{\partial a_j} = \frac{0 - e^{a_j} e^{a_i}}{\sum^2} = -\frac{e^{a_j}}{\sum} \frac{e^{a_i}}{\sum} = -S_j S_i$ if i $\neq$ j

So in our case, the derivative is:

$$
\frac{\partial \hat{y}_i}{\partial \tilde{y}_i} = \hat{y}_i(1 - \hat{y}_i) \quad \text{and} \quad \frac{\partial \hat{y}_k}{\partial \tilde{y}_i} = -\hat{y}_i \hat{y}_k
$$

So if we combine all of this:

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{y}_i} &= \sum_{k=i} \Big(-\frac{y_k}{\hat{y}_k} \cdot \hat{y}_i(1-\hat{y}_k) \Big) + \sum_{k\neq i} \Big(-\frac{y_k}{\hat{y}_k} \cdot \hat{y}_i(0-\hat{y}_k) \Big) \\
&= \Big(-y_i \frac{\hat{y}_i(1-\hat{y}_i)}{\hat{y}_i} \Big) + \sum_{k\neq i} \Big(-y_k \frac{\hat{y}_i(-\hat{y}_k)}{\hat{y}_k} \Big) \qquad (1) \\
&= \Big(-y_i(1-\hat{y}_i) \Big) + \sum_{k\neq i} \Big(y_k \hat{y}_i \Big) \\
&= -y_i(1-\hat{y}_i) + \sum_{k\neq i} y_k \hat{y}_i \\
&= -y_i + y_i\hat{y}_i + \hat{y}_i \sum_{k\neq i} y_k \\
&= -y_i + y_i\hat{y}_i + \hat{y}_i(1-y_i) \\
&= \hat{y}_i - y_i
\end{aligned}$$

In terms of matrices, the gradient of the loss than simply be expressed as:

$$\nabla_{\bar{y}} l = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_n} \end{bmatrix} = \hat{y} - y$$

Note: at the equation (1), we change the index from k to i for the term where k=i. This is because the sum only includes the case k=i, so k is effectively fixed to i. All other terms do not appear in this sum. This simplification allows us to replace k with i and remove the index from the expression.

16. **Using backpropagation, write the gradient of the loss with respect to the weights of the output layer $\mathbf{W}_y$. Note that computing this gradient uses $\nabla_{\tilde{\mathbf{y}}} \ell$. Do the same for $\mathbf{b}_y$.**

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \dots \quad \Longrightarrow \quad \nabla_{\mathbf{W}_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_yn_h}} \end{bmatrix} = \dots$$

So first of all,

$$\tilde{y} = W_y h + b_y \quad \text{with} \quad h \in \mathbb{R}^{n_h}$$

then by defining,

$$\tilde{y}_k = \sum_{j=1}^{n_h} W_{y,ij}\, h_j + b_{y,k},$$

We will now be able to simplify $\frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$:

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_{k=1}^{n_y} \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \frac{\partial \ell}{\partial \tilde{y}_i} h_j$$

In matrix form:

$$\nabla_{\mathbf{W}_y}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1}h_1 & \cdots & \frac{\partial \ell}{\partial \tilde{y}_1}h_{n_h} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}}h_1 & \cdots & \frac{\partial \ell}{\partial \tilde{y}_{n_y}}h_{n_h} \end{bmatrix} = \left(\nabla_{\tilde{\mathbf{y}}}\ell\right)\mathbf{h}^\top$$

This means that the gradient of the weights is obtained by multiplying the loss gradient with the output of the hidden layer.

For the bias b_y:

$$\frac{\partial \ell}{\partial b_{y,i}} = \sum_{k=1}^{n_y} \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \tilde{y}_i}$$

so in vector form

$$\nabla_{\mathbf{b}_y}\ell = \nabla_{\tilde{\mathbf{y}}}\ell.$$

17. ⋆ **Compute the gradients:** $\nabla_{\tilde{h}}\ell, \nabla_{W_h}\ell, \nabla_{b_h}\ell$

(a) $\nabla_{\tilde{h}_i}\ell$
First new define the deltas:

$$\delta_i^{(4)} = \frac{\partial \ell}{\partial \tilde{y}_i} \implies \delta_i^{(3)} = \frac{\partial \ell}{\partial W_y} = \sum_j \delta_j^{(4)}\frac{\partial \tilde{y}_j}{\partial W_y} = \sum_j \delta_j^{(4)} h_{ij}$$

and

$$\delta_i^{(2)} = \frac{\partial \ell}{\partial \tilde{h}_i} = \sum_j \delta_j^{(3)}\frac{\partial h_j}{\partial \tilde{h}_i} = \delta_i^{(3)}(1-\tanh^2(\tilde{h}_i))$$

so $\delta_i^{(3)}(1-\tanh^2(\tilde{h}_i)) = \nabla_{\tilde{h}_i}\ell$

(b) $\nabla_{W_h}\ell$
We define:

$$\delta_i^{(1)} = \frac{\partial \ell}{\partial W_h} = \sum_j \delta_j^{(2)}\frac{\partial \tilde{h}_j}{\partial W_h} = \sum_j \delta_j^{(2)} x_{ij}$$

So $\nabla_{W_h}\ell = \sum_j \delta_j^{(2)} x_{ij}$

(c) $\nabla_{b_h}\ell$
And finally,

$$\frac{\partial \ell}{\partial b_h} = \sum_j \delta_j^{(2)}\frac{\partial \tilde{h}_j}{\partial b_h} = \sum_j \delta_j^{(2)}$$

So $\nabla_{b_h}\ell = \sum_j \delta_j^{(2)}$