Autonomous Vision Group
Prof. Dr.-Ing. Andreas Geiger

University of Tübingen
MPI for Intelligent Systems
Autonomous Vision Group

Deep Learning

Exercise 2 – Image Classfication

Release date: October 30, 2024

2.1 Pen & Paper Exercise (1 + 1 + 1 + 2 + 1 + 1 + 2 + 1 + 2 = 12 Points)

a) Analytic Derivative Calculation

i) $\text{ReLU}_c(\mathbf{x}) = \max(0, \mathbf{x} - c)$, $\mathbf{x} \in \mathbb{R}$ and c is a scalar constant.

$$\frac{\partial \text{ReLU}_c(\mathbf{x})}{\partial \mathbf{x}} = \begin{cases} 1 & \mathbf{x} > c \\ 0 & \mathbf{x} \le c \end{cases}$$

ii) $\text{ELU}(\mathbf{x}) = \begin{cases} \mathbf{x} & \mathbf{x} \ge 0 \\ c(e^{\mathbf{x}} - 1) & \mathbf{x} < 0 \end{cases}$, where $\mathbf{x} \in \mathbb{R}$ and c is a scalar constant.

$$\frac{\partial \text{ELU}(\mathbf{x})}{\partial \mathbf{x}} = \begin{cases} 1 & \mathbf{x} \ge 0 \\ ce^{\mathbf{x}} & \mathbf{x} < 0 \end{cases}$$

iii) $\text{Tanh}(\mathbf{x}) = \frac{2}{1+e^{-2\mathbf{x}}} - 1$, $\mathbf{x} \in \mathbb{R}$

$\text{Tanh}(\mathbf{x}) = \frac{2}{1+e^{-2\mathbf{x}}} - 1 = \frac{e^{\mathbf{x}}-e^{-\mathbf{x}}}{e^{\mathbf{x}}+e^{-\mathbf{x}}}$. So,

$$\begin{aligned}\frac{\partial \text{Tanh}(\mathbf{x})}{\partial \mathbf{x}} &= \frac{(e^{\mathbf{x}}+e^{-\mathbf{x}})(e^{\mathbf{x}}+e^{-\mathbf{x}}) - (e^{\mathbf{x}}-e^{-\mathbf{x}})(e^{\mathbf{x}}-e^{-\mathbf{x}})}{(e^{\mathbf{x}}+e^{-\mathbf{x}})^2} \\ &= \frac{(e^{\mathbf{x}}+e^{-\mathbf{x}})^2 - (e^{\mathbf{x}}-e^{-\mathbf{x}})^2}{(e^{\mathbf{x}}+e^{-\mathbf{x}})^2} \\ &= 1 - \frac{(e^{\mathbf{x}}-e^{-\mathbf{x}})^2}{(e^{\mathbf{x}}+e^{-\mathbf{x}})^2} \\ &= 1 - \text{Tanh}^2(\mathbf{x})\end{aligned}$$

iv) $\text{Softmax}(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{n} e^{x_i}}$, $\mathbf{x} \in \mathbb{R}^n$

For $\mathbf{x} = [x_1, x_2, \cdots, x_n]^\top$, the output can be defined as $S_j = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}$. If we set $q(x_i) = e^{x_i}$ and $h(\mathbf{x}) = \sum_{i=1}^{n} e^{x_i}$, then the derivative of the output S_j with respect to any input element x_k can be then written as:

$$\frac{\partial S_j}{\partial x_k} = \frac{\partial \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}}{\partial x_k} = \frac{\partial \frac{q(x_j)}{h(\mathbf{x})}}{\partial x_k} = \frac{\frac{\partial q(x_j)}{\partial x_k} h(\mathbf{x}) - q(x_j) \frac{\partial h(\mathbf{x})}{\partial x_k}}{h(\mathbf{x})^2}$$

We have: $\frac{\partial q(x_j)}{\partial x_k} = e^{x_j}$ when $j = k$ and $\frac{\partial q(x_j)}{\partial x_k} = 0$ when $j \neq k$. Also, $\frac{\partial h(\mathbf{x})}{\partial x_k} = \frac{\partial e^{x_1+e^{x_2}+\cdots+e^{x_n}}}{\partial x_k} = e^{x_k}$. Therefore, we now can acquire the derivatives for two different cases.

First, when $j = k$:

$$\frac{\partial S_j}{\partial x_k} = \frac{e^{x_j}\sum_{i=1}^{n} e^{x_i} - e^{x_j}e^{x_j}}{\sum_{i=1}^{n} e^{x_i}\cdot\sum_{i=1}^{n} e^{x_i}} = \frac{e^{x_j}(\sum_{i=1}^{n} e^{x_i} - e^{x_j})}{\sum_{i=1}^{n} e^{x_i}\cdot\sum_{i=1}^{n} e^{x_i}} = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}(1 - \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}) = S_j(1 - S_j)$$

When $j \neq k$:

$$\frac{\partial S_j}{\partial x_k} = \frac{0 - e^{x_k}e^{x_j}}{\sum_{i=1}^{n} e^{x_i}\cdot\sum_{i=1}^{n} e^{x_i}} = -\frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}\frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} = -S_kS_j$$

Note that in `edf.py`, we have a vectorized version of this derivative.

v) Given the predicted probability $\mathbf{p} \in \mathbb{R}^n$ and ground-truth label $\mathbf{y} \in \mathbb{R}^n$, calculate the gradient of cross entropy loss $H(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{n} y_i \log p_i$ wrt. every element of $\mathbf{p}$, i.e. $\frac{\partial H}{\partial p_i}(\mathbf{y}, \mathbf{p})$.

$$\frac{\partial H}{\partial p_i}(\mathbf{y}, \mathbf{p}) = \frac{-\partial \sum_{i=1}^{n} y_i \log p_i}{\partial p_i} = -\frac{y_i}{p_i}$$

b) Prove the equation: Softmax($\mathbf{x}$) = Softmax($\mathbf{x} + c$), where c is a constant.

$$\text{Softmax}(\mathbf{x} + c) = \frac{e^{\mathbf{x}+c}}{\sum_{i=1}^{n} e^{x_i+c}} = \frac{e^{\mathbf{x}}\cdot e^{c}}{e^{c}\cdot\sum_{i=1}^{n} e^{x_i}} = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{n} e^{x_i}} = \text{Softmax}(\mathbf{x})$$

c) Consider an 1-layer neural network $\mathbf{y} = g(\mathbf{AX})$ with input $\mathbf{X}$, output $\mathbf{y}$, network weight $\mathbf{A}$ and output function g. Let's first assume the input and the network weights are

$$\mathbf{A} = \begin{pmatrix} 3.0 & 2.0 \end{pmatrix} \in \mathbb{R}^{1\times 2} \qquad \mathbf{X} = \begin{pmatrix} 2.0 & 1.0 & -1.0 \\ 4.0 & -2.0 & 0.0 \end{pmatrix} \in \mathbb{R}^{2\times 3}$$

where $\mathbf{x}$ are 2D points with a **minibatch** size of 3.

i) Assume the function g is linear, i.e. $g(\mathbf{AX}) = \mathbf{AX}$. Given the target output $\mathbf{t}$ and the loss function $\mathcal{L}$, perform weight update. Assume a learning rate of 1. Please provide the loss *before* and *after* the weight update.

$$\mathbf{t} = \begin{pmatrix} 15 & 3 & 1 \end{pmatrix} \in \mathbb{R}^{1\times 3} \qquad \mathcal{L} = \frac{1}{2}\sum_j (y_j - t_j)^2$$

Hint: Use the algorithm presented in slide 35 in lecture 3. First calculate the forward pass and obtain the loss. Next, you can derive the gradients of loss wrt. to every element of the network weight $\frac{\partial \mathcal{L}}{\partial a_i}$. Make sure to include all steps of your derivation. Once the network weight is updated, calculate the forward pass again to acquire the loss.

Set $\mathbf{A} = \begin{pmatrix} a_1 & a_2 \end{pmatrix}$, $\mathbf{y} = \mathbf{AX} = \begin{pmatrix} 2a_1 + 4a_2 & a_1 - 2a_2 & -a_1 \end{pmatrix}$. Therefore, the loss is:

$$\mathcal{L} = \frac{1}{2}(2a_1 + 4a_2 - 15)^2 + \frac{1}{2}(a_1 - 2a_2 - 3)^2 + \frac{1}{2}(-a_1 - 1)^2$$

At time step t, the loss is:

$$\mathcal{L}^t = \frac{1}{2}(1 + 16 + 16) = 16.5$$

The derivative w.r.t. **A**:

$$\frac{\partial \mathcal{L}^t}{\partial a_1} = 2 \cdot (2a_1 + 4a_2 - 15) + (a_1 - 2a_2 - 3) - (-a_1 - 1) = -2$$
$$\frac{\partial \mathcal{L}^t}{\partial a_2} = 4 \cdot (2a_1 + 4a_2 - 15) - 2 \cdot (a_1 - 2a_2 - 3) = 4$$

Using the algorithm in slide 35 lecture 3, we can obtain:

$$a_1^{t+1} = a_1^t - \eta \cdot \frac{1}{3}\frac{\partial \mathcal{L}^t}{\partial a_1} = 3 + \frac{2}{3} = 3.667$$
$$a_2^{t+1} = a_2^t - \eta \cdot \frac{1}{3}\frac{\partial \mathcal{L}^t}{\partial a_2} = 2 - \frac{4}{3} = 0.667$$

The update $\mathbf{A}^{t+1} = \begin{pmatrix} 3.667 & 0.667 \end{pmatrix}$ and new $\mathcal{L}^{t+1}$ becomes:

$$\mathcal{L}^{t+1} = 23.611$$

Alternatively, you can also consider the chain rule to calculate the gradient.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{A}}$$

where

$$\frac{\partial \mathcal{L}}{\partial \mathbf{y}} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial y_1} & \frac{\partial \mathcal{L}}{\partial y_2} & \frac{\partial \mathcal{L}}{\partial y_3} \end{pmatrix} = \begin{pmatrix} y_1 - t_1 & y_2 - t_2 & y_3 - t_3 \end{pmatrix} = \begin{pmatrix} -1 & -4 & -4 \end{pmatrix}$$
$$\frac{\partial \mathbf{y}}{\partial \mathbf{A}} = \begin{pmatrix} \frac{\partial y_1}{a_1} & \frac{\partial y_1}{a_2} \\ \frac{\partial y_2}{a_1} & \frac{\partial y_2}{a_2} \\ \frac{\partial y_3}{a_1} & \frac{\partial y_3}{a_2} \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 1 & -2 \\ -1 & 0 \end{pmatrix}$$

so

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{A}} = \begin{pmatrix} -1 & -4 & -4 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ 1 & -2 \\ -1 & 0 \end{pmatrix} = \begin{pmatrix} -2 & 4 \end{pmatrix}$$

The rest is the same as before.

ii) For the loss that you calculated before and after the weight update in **i)**, does the loss decrease, or in other word, is the new prediction closer to the target **t**? If not, what could be the reason?

The loss increases because the learning rate is too high.

iii) Now we set the function g to be ReLU. For $\mathbf{AX} \in \mathbb{R}^{1\times 3}$, ReLU is applied to every element. Now please calculate the gradients wrt. the network weight $\frac{\partial \mathcal{L}}{\partial a_i}$. You can use the chain rule $\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial a_i}$. What do you notice in the calculated gradient?
Hint: Consider what you have learnt in lecture 4.

The derivative w.r.t. **A**:
You can consider the chain rule to calculate the gradient. Let's first denote $\mathbf{X} = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 \end{pmatrix}, \mathbf{x}_j \in \mathbb{R}^{2\times 1}$. According to the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{A}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{y}} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial y_1} & \frac{\partial \mathcal{L}}{\partial y_2} & \frac{\partial \mathcal{L}}{\partial y_3} \end{pmatrix} = \begin{pmatrix} y_1 - t_1 & y_2 - t_2 & y_3 - t_3 \end{pmatrix} = \begin{pmatrix} -1 & -3 & -1 \end{pmatrix}$$

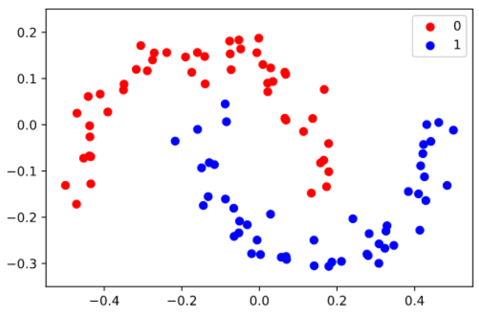
$$\frac{\partial \mathbf{y}}{\partial \mathbf{A}} = \begin{pmatrix} \frac{\partial y_1}{a_1} & \frac{\partial y_1}{a_2} \\ \frac{\partial y_2}{a_1} & \frac{\partial y_2}{a_2} \\ \frac{\partial y_3}{a_1} & \frac{\partial y_3}{a_2} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial a_1}\max(0, \mathbf{A}\mathbf{x}_1) & \frac{\partial}{\partial a_2}\max(0, \mathbf{A}\mathbf{x}_1) \\ \frac{\partial}{\partial a_1}\max(0, \mathbf{A}\mathbf{x}_2) & \frac{\partial}{\partial a_2}\max(0, \mathbf{A}\mathbf{x}_2) \\ \frac{\partial}{\partial a_1}\max(0, \mathbf{A}\mathbf{x}_3) & \frac{\partial}{\partial a_2}\max(0, \mathbf{A}\mathbf{x}_3) \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$$

so

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{A}} = \begin{pmatrix} -1 & -3 & -1 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} -5 & -4 \end{pmatrix}$$

You can notice that the gradients for the x_3 did not contribute at all to the final gradients. We have learnt this behaviour in the lecture and it is called **Dead ReLU** or **Dying ReLU**.

2.2 Binary Classification on 2D Point Cloud (2 + 1 + 4 + 1 = 8 Points)



First of all, you can directly use your conda environment installed for exercise 1. You can simply run `conda activate deep_learning_ex_1` in your terminal. If you don't have the environment, please check `install_guide.txt` in exercise 1.
Now, in the jupyter notebook `pcl_binary_classification.ipynb`, you can find the code for loading a 2D point cloud dataset. The training code is also provided. Here, the input `X` contain the 100 2D points, and `y` are their corresponding labels (0 or 1). The goal is to train a model that can classify every point to its correct label.

a) As the first task, you are required to train a logistic regression model with help of the Educational Framework (EDF). The EDF's `CrossEntropyLoss` should be used. You can also reuse use your code from exercise 1.

b) Next, visualize your trained model. What we do here is, within the range of [-0.5, 0.5] in both x and y axis, we sample points with the space of 0.002. In this way, there are $500 \times 500 = 250K$ 2D points in total, denoted as `X_plot`. We want to predict their corresponding label with your trained logistic regression model. To this end, we have built the input as `input = edf.Parameter(X_plot)`. All you need to do is to fill `output` with your trained model. If everything is correct, you should see a decision boundary in black.

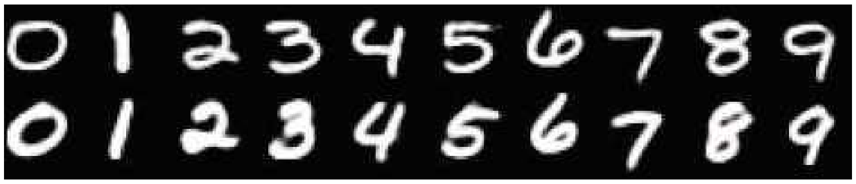
Hint: you can basically use your code in assembling the computational graph, but remove `SingleProbToProbVector`.

c) After visualizing the classification result, you should notice that your logistic regression model is not able to classify correctly. You can find the discussion in Lecture 3.2. Therefore, now your task is to implement your first Multi-Layer Perceptron (MLP) for the same binary classification task. You should implement an MLP with two layers: the first one takes the input point coordinates and the other one outputs the classification predictions. The hidden dimension is set to `nHiddens`=16. For example, given 100 points as input, the first affine node should output a matrix of 100×16. `Sigmoid` is used as the activation function in the hidden layer as well as the final output function.

d) Similar to **b)**, visualize and verify your trained MLP model. Take the `input`, you should fill `output` with your model. You should be able to see that MLP is able to classify the 2D point clouds properly with a bending decision boundary.

2.3 Image Classification on MNIST (2 + 1 + 2 + 6 + 2 + 2 + 2 + 2 + 1 = 20 Points)

We can now move to `image_classification.ipynb`. Unlike in exercise 1 where we only pick two digits to perform binary classification, now we move to a slightly more difficult task: classification on all 10 digits.



There are $60K$ `train_images` and $10K$ `test_images`, where every image is a 784-dim vector reshaped from the original grayscale image with the resolution of 28×28. Correspondingly, `train_labels` and `test_labels` contain their real labels between 0 and 9. The goal of this homework is to train a model that can make accurate classification given some hand-written digit images.

a) Again, the first task is to implement our old friend, the logistic regression model. You should be able to reuse your code from before. Note that the output dimension should be `nOutputs`=10 now, which represents the predicted probability on all 10 classes. Also, instead of using `Sigmoid` we replace the output function now with `Softmax`. You can directly use the implemented `Softmax` in `edf.py`. Moreover, we don't need `SingleProbToProbVector` any more. Finally, you should use `CrossEntropyLoss` as your loss function. If the model is correctly implemented, the test error should be around 8% after 10 epochs.

b) *Open Question*: If your logistic regression model weight is randomly initalized, and no training is performed, what test error do you think the model will get? Please answer inside the notebook.

c) Now you should implement an MLP with two linear layers and `Sigmoid` as the activation function in the hidden layer. Note that you should also use `Softmax` as your function for the final output. A large part of your code in 2.2 can be reused. If it is correctly implemented, the test error now becomes around 2%.

d) As taught in lecture 4, there are many other activation functions other than `Sigmoid`. We want to explore the different performance when using different activation functions. Your task now is to implement the `forward` and `backward` pass for the new activation functions. The basic structure has been provided in the notebook. Note that you can still base your implementation on `Sigmoid` in `edf.py`. Moreover, the analytical gradient that you derived in exercise 2.1 is also useful. For each activation function, we also provide the code to plot output and analytical gradients computed by your implementation as a sanity check.

- **i)** Implement `Tanh`.
- **ii)** Implement `ReLU`.
- **iii)** `LeakyReLU` with $c = 0.01$.

After you implement every new activation function, you simply need to define a computation graph and train your MLP with this function. Follow the instruction inside notebook. Test errors will be less than or around 2% for all of them if correctly implemented.

e) After implementing all the activation functions, now we want to explore Numerical Differentiation discussed in the lecture 4.2. Given an activation function f and an input vector $\mathbf{x}$, please implement the numeric gradient of f using symmetric difference quotient, i.e. $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \approx \frac{f(\mathbf{x}+h)-f(\mathbf{x}-h)}{2h}$, where

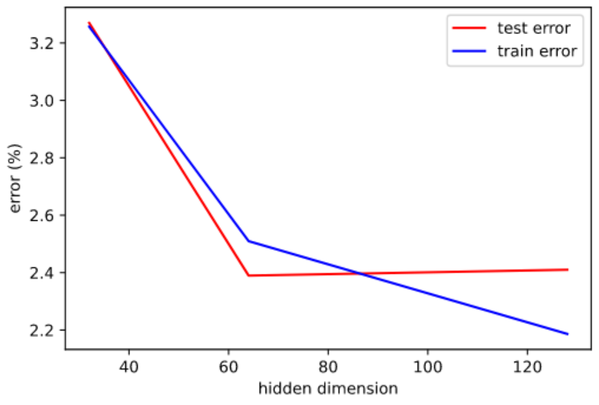
$h = 10^{-4}$. You can consider $\mathbf{x}+h$ and $\mathbf{x}-h$ as objects of `edf.Input()`. Then, you perform a forward pass and calculate `numeric_grad`. Once you are done, you can use the provided plot code to verify. The curve for analytical gradient and numeric gradient should overlap each other.

f) Let's check how changing the learning rate affects the model's train and test error. We provide the for-loop over the provided list `learning_rates` = $[0.1, 0.5, 1.0]$ and use a new learning rate every time `edf.learning_rate = learning_rates[i]`. You should implement the training process in a way to collect the network's final train/test errors for each of the learning rates, and keep them in the lists `train_err_per_lr` and `test_err_per_lr`. Try to find the best learning rate for both `ReLU` and `Sigmoid`. Note that you can take the training code in **c)** or **d)** as reference.

g) We will check how the number of hidden layers affects the model's performance. Similar to **d)**, you will see how the depth of the network `nLayers` affects its performance. We provide a list `num_layers` = $[2, 4, 6]$. Therefore, you can use a for-loop over the list `num_layers`, and you update your intermediate output within the loop. First try with the Sigmoid MLP and then a ReLU MLP. Again, populate `train_err_per_depth` and `test_err_per_depth` accordingly. Your code should support multiple hidden layers (note the new nLayers argument). Make sure that each hidden layer in your MLP should have the same `nHiddens` number of neurons.

h) Next, let's check how changing hidden dimensions in your MLP affect the model's performance. Need to change `nHiddens` within a for-loop. Again, populate `train_err_per_hidden` and `test_err_per_hidden` accordingly. You can reuse most of your code from above.

i) *Open Question*: Assume a model's train error keeps decreasing, but the test error increases when hidden dimension is too large (as shown below), what might be the reason? Please write down the reason inside the provided blank cell inside the notebook.


test error
train error
error (%)
hidden dimension