Problem set 1
Deep Learning E1394

Danial Riaz, 201678
Carlo Greß, 216319

Out on Sept 18
Due on Oct 2, 23:59

Submit your written answers as a pdf typed in LaTeXtogether with your code. Submit one answer per group (as assigned on Moodle) and include names of all group members in the document. Round answers to two decimal places as needed. Include references to any external sources you have consulted (points are deducted if those were used but not cited). See "Submission" at the bottom of the problem set for more details on how to submit using Github classroom.

1 Theoretical part

These are some recap and refresher problems to get you up to speed with the mathematics and statistical learning that is behind deep learning, as well as problems that help you work out how neural networks are trained.

1.1 Optimization

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be defined by $f(x, y) = x^2y + xy^2 - 6xy$. Identify and classify all critical points of f.

For identifying and classifying the critical points of $f(x, y) = x^2y+xy^2-6xy$, we'll need both the gradient and the Hessian of the function. The gradient is defined as the vector of partial derivatives of the function:

$$\nabla f(x, y) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

Hence, we need to derive $f(x, y)$, once with respect to x and once with respect to y. Deriving with respect to x gives us:

$$\frac{\partial f}{\partial x} = 2xy + y^2 - 6y$$

Deriving with respect to y gives:

$\frac{\partial f}{\partial y} = x^2 + 2xy - 6x$

Subsequently, the gradient is: $\nabla f(x,y) = \begin{pmatrix} 2xy + y^2 - 6y \\ \\ x^2 + 2xy - 6x \end{pmatrix}$

For classification of the points, we'll need the Hessian matrix, which is defined as:

$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$

Therefore, we need three more derivatives: The first partial derivative with regard to x derived once for x and once for y again, and the partial derivative with regard to y again with regard to y.

$\frac{\partial^2 f}{\partial x^2} = 2y$
$\frac{\partial^2 f}{\partial x \partial y} = 2x + 2y - 6y$
$\frac{\partial^2 f}{\partial y^2} = 2x$

Subsequently:

$H(f) = \begin{bmatrix} 2y & 2x + 2y - 6y \\ 2x + 2y - 6y & 2x \end{bmatrix}$

For identifying the critical points, we need to set the gradient equal to 0. Subsequently, we solve for both x and y.

$2xy + y^2 - 6y = 0$
$x^2 + 2xy - 6x = 0$

Factoring out x from the second equation gives: $x(x+2y-6) = 0$. The first solution for this is $x = 0$. Plugging in $x = 0$ into the first equation results in: $y^2 - 6y = 0$, which is solved by $y = 0$ and $y = 6$. Hence, our first two critical points are (0/0) and (0/6).

The second solution of $x(x + 2y - 6) = 0$ is $x = -2y + 6$. Plugging this into the first equation of the gradient gives $2y(-2y + 6) + y^2 - 6y = 0$, which results in $-3x^2 + 6x = 0$. Solving for y gives 0 and 2 as solutions. Plugging these values back in the first equation results in the x-values 0 and 2. Hence our third critical point is (2/2).

Ultimately, we are factoring out y from the first equation, which gives $y(2x + y - 6) = 0$. Again, $y = 0$ is one solution, and plugging in $y = 0$ in the second equation results in the corresponding x-values 0 and 6. Hence, our

fourth and final critical point is (6/0). We could again solve for the second solution of $y(2x + y - 6) = 0$, but this only leads to the critical points that we already identified.

For classifying the four critical points, we calculate the determinant of the Hessian at each of them. Starting with (6/0), the Hessian is:

$\det(H) = \begin{vmatrix} 0 & -6 \\ -6 & 12 \end{vmatrix} = -36$. Since $-36 < 0$, (6/0) is a saddle point.

For (0/6):

$\det(H) = \begin{vmatrix} 12 & 6 \\ 6 & 0 \end{vmatrix} = -36$. Since $-36 < 0$, (0/6) is a saddle point.

For (0/0):

$\det(H) = \begin{vmatrix} 0 & -6 \\ -6 & 0 \end{vmatrix} = -36$. Since $-36 < 0$, (0/0) is a saddle point.

For (2/2):

$\det(H) = \begin{vmatrix} 4 & 2 \\ 2 & 4 \end{vmatrix} = 12$. Since $12 > 0$, (2/2) is a local minimum or local maximum. To check that, we look at the value of the partial derivative which has been derived with respect to x twice. This value is $4 > 0$, so (2/2) is a local minimum.

1.2 Activation functions

Compute the gradient of the function

$$f(b, w) = \text{ReLU}(b + xw) = \begin{cases} 0 & \text{if } b + xw < 0, \\ b + xw & \text{if } b + xw \geq 0. \end{cases} \tag{1}$$

with respect to the parameters b and w, where $x, b, w \in \mathbb{R}$.

For calculating the gradient, we need to differentiate once with respect to b and once with respect to w. Differentiating with respect to b gives:

$$\frac{\partial f}{\partial b} = \begin{cases} 0 & \text{if } b + xw < 0, \\ 1 & \text{if } b + xw > 0. \end{cases}$$

Differentiating with respect to w gives:

$$\frac{\partial f}{\partial w} = \begin{cases} 0 & \text{if } b + xw < 0, \\ x & \text{if } b + xw > 0. \end{cases}$$

Thus, the gradient is:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial b} \\ \frac{\partial f}{\partial w} \end{pmatrix} = \begin{pmatrix} \begin{cases} 0 & \text{if } b + xw < 0, \\ 1 & \text{if } b + xw > 0. \end{cases} \\ \begin{cases} 0 & \text{if } b + xw < 0, \\ x & \text{if } b + xw > 0. \end{cases} \end{pmatrix}$$

If $b + wx = 0$, the derivative is undefined.

1.3 Overfitting

This question will test your general understanding of overfitting as it relates to model complexity and training set size. Consider a continuous domain and a smooth joint distribution over inputs and outputs, so that no test or training case is ever duplicated exactly. *The graphs in your solutions should be drawn on paper and then included as a picture. In your answers describe how how your graphs should be read.*

1.3.1 Error rate versus dataset size

Sketch a graph of the typical behavior of training error rate (y-axis) as a function of training set size (x-axis). Add to this graph a curve showing the typical behavior of test error rate versus training set size, on the same axes. (Assume that we have an infinite test set drawn independently from the same joint distribution as the training set). Indicate on your y-axis where zero error is and draw your graphs with increasing error upwards and increasing training set size rightwards.

1.3.1 Solution

When our training set size is small, the model is not sufficiently trainned and there is a big gap between the Training and Test error, we are overfitting here. As the training set size increases, there are less extreme variations between the Training and Test errors and the test error decreases sharply, coming close to the training error. The training error, beyond a certain point, only marginally improves and there are diminishing returns to increasing the training set size. The Test error eventually comes to an irreducible error which is due to the noise in the data generative process.

1.3.2 Error rate versus model complexity

For a fixed training set size, sketch a graph of the typical behavior of training error rate (y-axis) versus model complexity (x-axis). Add to this graph a curve showing the typical behavior of the corresponding test error rate versus model complexity, on the same axes (again on an IID infinite test set). Show where on

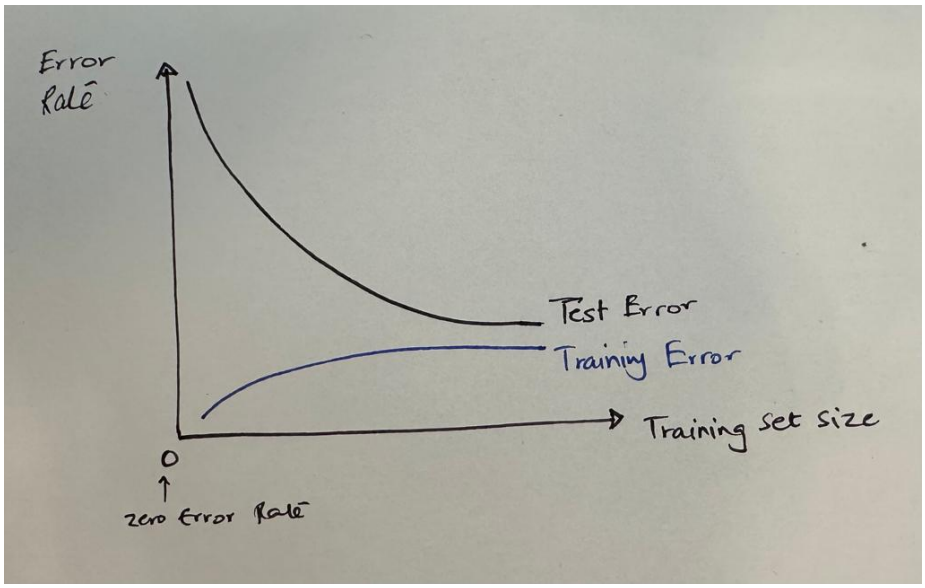

Figure 1: Error Rate vs Training Set Size

the x-axis you think is the most complex model that your data supports (mark this with a vertical line). Choose the x-range so that this line is neither on the extreme left nor on the extreme right. Indicate on your vertical axis where zero error is and draw your graphs with increasing error upwards and increasing complexity rightwards.

1.3.2 Solution

In this figure we see that as the complexity of the model increases, the training error decreases, but the testing error decreases up to a certain point and then increases. This means that the model is more flexible and fits the training data better, but it can also end up over-fitting to random effects that are present only in the specific data-set used for training.

A high error-rate on both training and validation data indicates that the model may be too simple to faithfully capture any relationships present in the data. In this case, the model is said to have high bias. If the error-rate is low on training data but high on validation data, then the model may be too complex - it suffers from high variance.

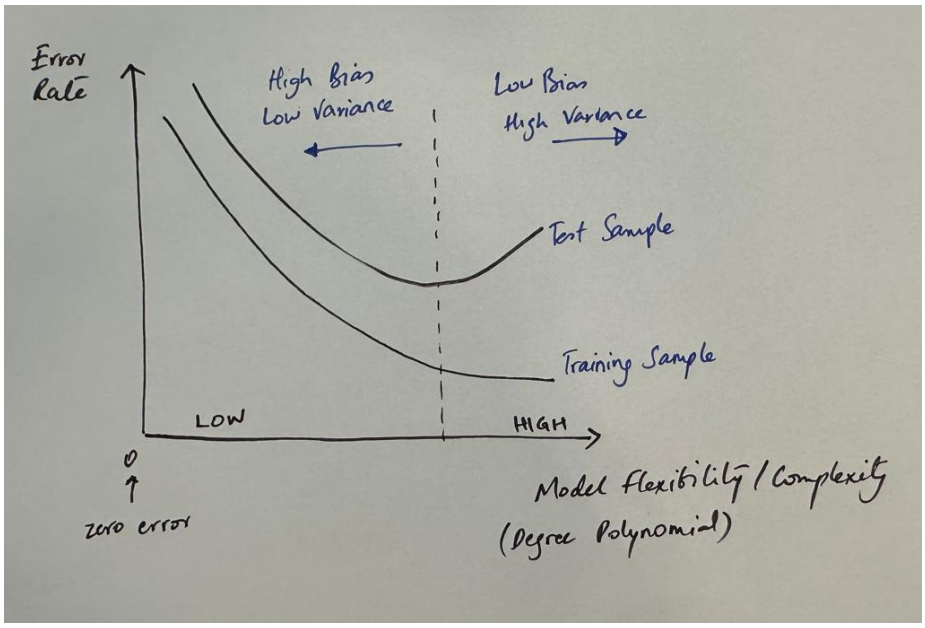

Figure 2: Error Rate vs Model Complexity

1.3.3 Training epochs

Use a similar graph to illustrate the error rate as a function of training epochs in neural networks. One of the commonly used regularization methods in neural networks is early stopping. Describe (also using the graph) how early stopping is applied, and argue qualitatively why (or why not) early stopping is a reasonable regularization metric.

1.3.3 Solution

Though our training loss is decreasing with every attrition, 'Validation loss' highlights the performance of our model on unseen data, which only decreases till the 5th epoch in our diagram. After this point our validation loss starts increasing. That implies our model has started to overfit. We stop the training at the 5th epoch to prevent overfitting from happening. This is called Early Stopping.

The benefit of this is that we minimize our validation loss. The challenge is that this approach doesn't 'cure for' overfitting, we are just stopping training before this happens. We want to reduce the error gap between training and validation/reduce validation loss even more.

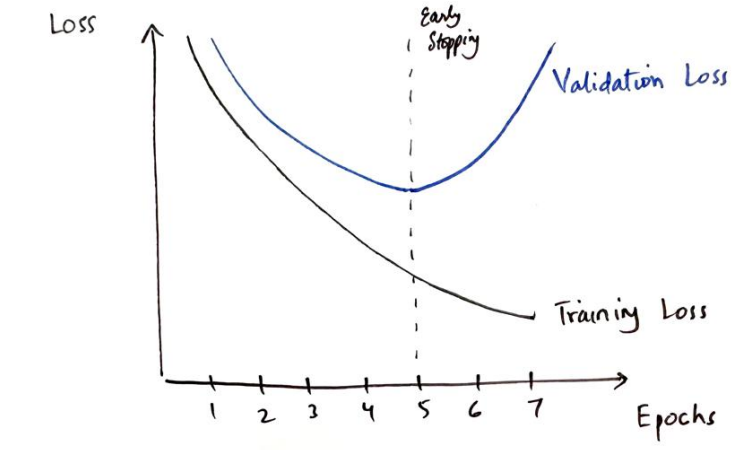

Figure 3: Loss vs Epochs

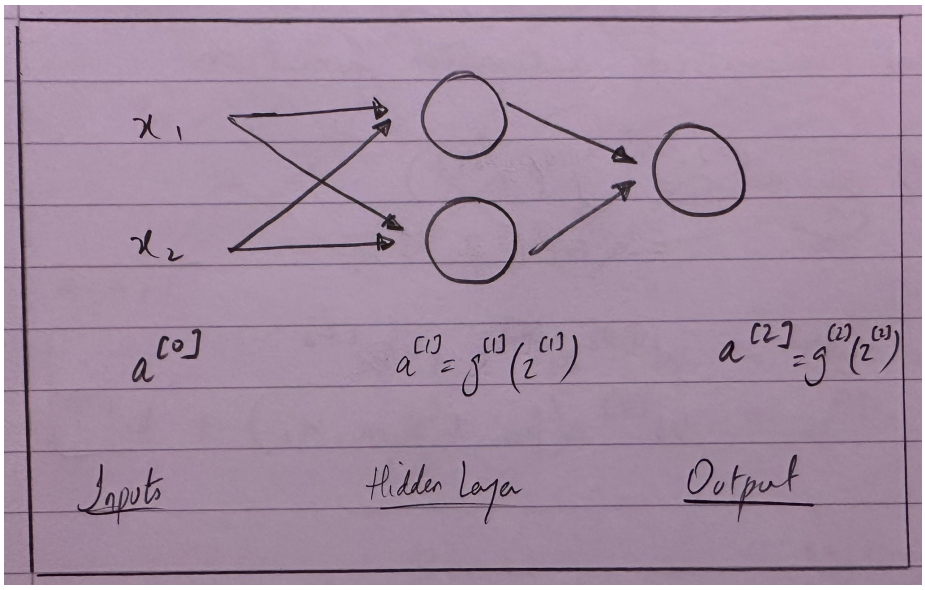

Figure 4: Neural Network sketch for 1.4

1.4 Capacity of neural network

We have a single hidden-layer neural network with two hidden units, and data with 2 features $X = (x_1, x_2)$. Design an activation function $g(\cdot)$ applied in the hidden layer, and an output activation function $o(\cdot)$ applied in the output layer, as well as a set of values of the weights that will create a logistic regression model where the input to the logistic function is of the form: $\beta_0 + \beta_1 x_1 x_2$.

1.4 Solution

We know that our input to the logistic function ie. our $a^{[1]} = \beta_0 + \beta_1 x_1 x_2$

$$= g^{[1]}\left(z^{[1]}\right) = \beta_0 + \beta_1 x_1 x_2$$

$$g^{[1]}\left(W^{[1]}X + b^{[1]}\right) = \beta_0 + \beta_1 x_1 x_2$$

* We can take $g^{[1]}$ to be a linear activation function applied to $z^{[1]}$ where $z^{[1]}$ is the waited sum of our input features x_1 and x_2 $\therefore$ our hidden layer activation function

$$g^{[1]}\left(z^{[1]}\right) = z^{[1]}$$

* Since the output activation function is stated to be a logistic function, we can

make use of the sigmoid activation function.

$$\begin{aligned}\therefore \quad o\left(z^{[2]}\right) = a^{[2]} &= g^{[2]}\left(z^{[2]}\right) \\ &= \sigma\left(z^{[2]}\right) = \frac{1}{1+e^{-z}} \\ &\Big[\text{ where } z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ &= W^{[2]}\left(\beta_0 + \beta_1 x_1 x_2\right) + b^{[2]}\Big]\end{aligned}$$

* Finally, the values of the weight could be set as :

$$\beta_0 = 0$$

assuming no bias in our linear activation function and

$$\beta_1 = 1$$

to control the slope of the logistic function influencing how fast the probabilitic change with respect to $x_1 x_2$ is.

1.5 Neural network theory

Derive the weight updates in gradient descent for a neural network with 2 hidden layers (superscripts [1] and [2]) that each have $H^{[1]}$ and $H^{[2]}$ hidden units respectively. The output layer has superscript [3], and we want to classify the data into K classes.

The output of the network for classes $k = 1, 2, ..., K$ and one data point $x \in \mathbb{R}^d$ can be written as such:

$$f_k(\theta; X) = o(a_k^{[3]}) = o(b_k^{[3]} + \sum_{m=1}^{H^{[2]}} w_{km}^{[3]} h_m^{[2]}) \tag{2}$$

$$h_m^{[2]} = \sigma\left(a_m^{[2]}\right) = \sigma\left(b_m^{[2]} + \sum_{i=1}^{H^{[1]}} w_{mi}^{[2]} h_i^{[1]}\right) \tag{3}$$

$$h_i^{[1]} = \sigma\left(a_i^{[1]}\right) = \sigma\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right), \tag{4}$$

where θ indicates the vector of all weights and biases.

While this is typically not recommended, in this exercise we use a a quadratic loss function for classification. We use a softmax activation function at the output layer and a ReLU activation function at the hidden layers. Derive the the weight update for the weights of the first hidden layer $w_{ij}^{[1]}$. Start by stating

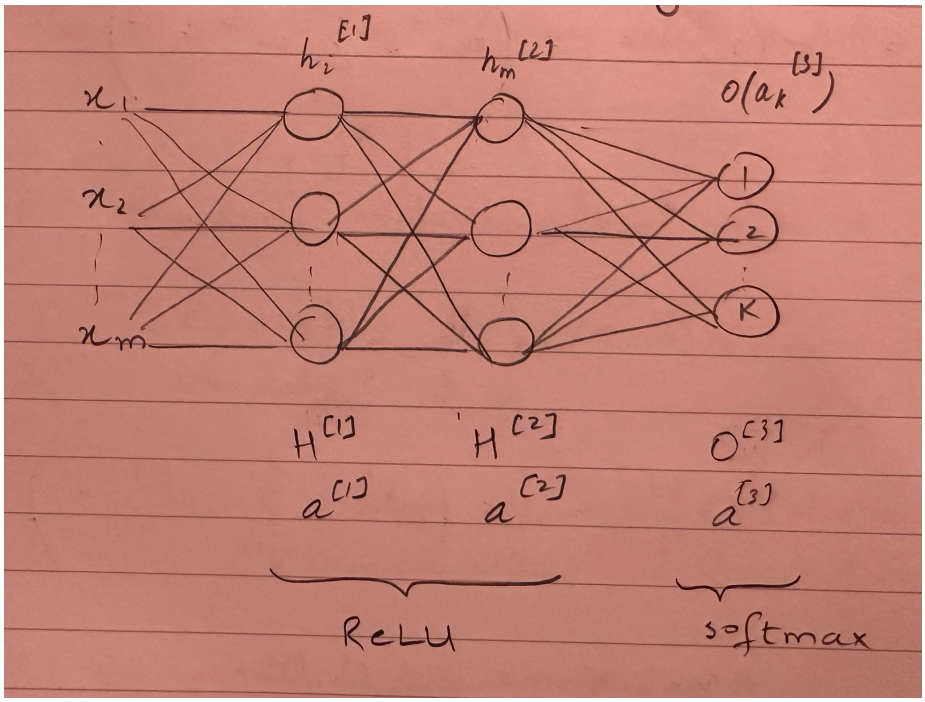

Figure 5: Neural Network Sketch for 1.5

the gradient descent weight update for the $(r+1)^{th}$ iteration as a function of the $(r)^{th}$ iteration, and then compute the partial derivative needed in the update.

Show all the steps in your derivation. You may use the derivatives for activation functions introduced in class. There is no need to show the derivations of those.

1.5 Solution

We need to determine $\frac{\partial \mathcal{L}}{\partial w_{ij}}$ We know that $\mathcal{L}$ is a quadratic loss function

$$L = \frac{1}{2}(y - \hat{y})^2$$

$$\text{where } \hat{y} = f_k = o\left(a_k^{[3]}\right) = o\left(b_k^{[3]} + \sum_{m=1}^{H^{[2]}} w_{km}^{[3]} h_m^{[2]}\right)$$

and $o(.)$ is a softmax function given by:

$$o(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

$$h_m^{[2]} = \sigma\left(a_m^{[2]}\right) = \sigma\left(b_m^{[2]} + \sum_{i=1}^{H[1]} w_{mi}^{[2]} h_i^{[1]}\right)$$

Where sigma is a Relu activation function

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{ofhewire} \end{cases}$$

$$h_i^{[1]} = \sigma\left(a_1^{[1]}\right) = \sigma\left(b_i^{[1]} + \sum_{j=1}^{d} w_i j^{[1]} x_j\right)$$

Again, sigma is a Relu activation function

$\Rightarrow$ To find the weight update for the $(r+1)^{\text{th}}$ iteration we need the derivative of the loss function with respect to w_{ij} at the r^{th} iteration.

$$\begin{aligned} w_{ij}^{(r+1)} &= w_{ij}^{(r)} + \eta \Delta w_{ij}^{(r)} \\ &= w_{ij}^{(r)} - \eta \left(\frac{\partial L(\hat{y}, y)}{\partial w_{ij}}\right)^{(r)} \end{aligned}$$

where η is the learning rate $\Rightarrow$ We an use the chain rule to compute $\frac{\partial L}{\partial w_{ij}}$

$$\frac{\partial \mathcal{L}}{\partial \omega_{ij}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_k^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial h_m^{[2]}} \cdot \frac{\partial h_m^{[2]}}{\partial a_m^{[2]}} \cdot \frac{\partial a_m^{[2]}}{\partial h_i{}^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i{}^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial \omega_{ij}{}^{[1]}}$$

Now we compute each of these terms in turn.

$$\cdot \frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial}{\partial y}\left(\frac{1}{2}(y - \hat{y})^2\right) = 2 \times \frac{1}{2} \times (-1)(y - \hat{y}) = \hat{y} - y$$

$$\cdot \frac{\partial \hat{y}}{\partial a_k^{[3]}} = \frac{\partial O\left(a_k^{[3]}\right)}{\partial a_k^{[3]}} = O\left(a_k^{[3]}\right)\left(\delta_{kL} - O\left(a_\iota{}^{[3]}\right)\right)$$

$$\text{where } \delta_{kl} = \begin{cases} l & k = l \\ 0 & k \neq l \end{cases}$$

Assuming $k = l$, this simplifies to

$$= o\left(a_k{}^{[3]}\right)\left(1 - o\left(a_k^{(3)}\right)\right)$$
$$= \hat{y}(1-\hat{y})$$

- $\frac{\partial a^{[3]}}{\partial h_m[2]} = \frac{\partial}{\partial h_m}\left(b_k^{(3)} + \sum w_{km}^{(3)} h_m^{(2)}\right)$

$$= \omega_{k_m}^{[3]}$$

- $\frac{\partial h_m^{[2]}}{\partial a_m[2]} = \begin{cases} 0 & \text{if } a_m^{(2)} < 0 \\ 1 & \text{if } a_m^{(2)} > 0 \end{cases}$ and undefined at $a_m^{(2)} = 0$

- $\frac{\partial a_m{}^{(2)}}{\partial h_i{}^{(1)}(1)} = \frac{\partial}{\partial h_i{}^{(1)}}\left(b_m{}^{(2)} + \sum_{i=1}^{H[1]} w_{mi}^{[2]} h_i^{[1]}\right)$

$$= \omega_{mi}^{[2]}$$

- $\frac{\partial h_i^{[1]}}{\partial a_i[1]} = \begin{cases} 0 & \text{if } a_i^{[1]} < 0 \\ 1 & \text{if } a_i^{[1]} > 0 \end{cases}$ and undefined at $a_i^{[1]} = 0$

$$\cdot \qquad \frac{\partial a_i{}^{[1]}}{\partial w_i j[1]} = \frac{\partial}{\partial w_j}\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)$$
$$= x_j$$

Now puting it all together, this becomes:

$$\frac{\partial \mathcal{L}}{\partial \omega_{ij}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_k^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial h_m^{[2]}} \cdot \frac{\partial h_m^{[2]}}{\partial a_m^{[2]}} \cdot \frac{\partial a_m^{[2]}}{\partial h_i{}^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i{}^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial \omega_{ij}{}^{[1]}}$$
$$= (\hat{y} - y) \cdot \hat{y}(1-\hat{y}) \cdot \omega_{km}^{[3]} \cdot 1 \cdot \omega_{mi}^{[2]} \cdot 1 \cdot x_j$$
$$\text{assuming } a_m{}^{[2]} \text{ and } a_i^{[1]} > 0$$

$\Rightarrow$ Our weight w at the $(r+1)^{\text{th}}$ iteration can be given as:

$$\omega_j^{(r+1)} = \omega_j^{(r)} + \eta\left[\frac{\partial L}{\partial \omega_{ij}}\right]^{(r)}$$

$$\omega_{ij}{}^{(r+1)} = \omega_{ij}{}^{(r)} + \eta\left[(\hat{y} - y) \cdot \hat{y}(1-\hat{y}) \cdot \omega_{km}^{[3]} \cdot \omega_{mi}{}^{[2]} \cdot x_j\right]^{(r)}$$

for $a_m^{(2)} > 0$ and $a_i^{(1)} > 0$ and

$$\omega_j{}^{(n+1)} = \omega_j^{(n)}$$

for cither $a_m^{[2]} < 0$ or $a_i^{[2]} < 0$

2 Neural network implementation

You will implement different classes representing a fully connected neural network for image classification problems. There are two classes of neural networks: one using only basic packages and one using PyTorch. In addition to that, a third class of neural network has been implemented using Tensorflow and requires some fixing in order to function correctly.
As you work through this problem, you will see how those machine learning libraries abstract away implementation details allowing for fast and simple construction of deep neural networks.

For each approach, a Python template is supplied that you will need to complete with the missing methods. Feel free to play around with the rest, but please do not change anything else for the submission. All approaches will solve the same classification task, which will help you validate that your code is working and that the network is training properly.

For this problem, you will work with the MNIST dataset of handwritten digits, which has been widely used for training image classification models. You will build models for a multiclass classification task, where the goal is to predict what digit is written in an image (to be precise, this is a k-class classification task where in this case $k = 10$). The MNIST dataset consists of black and white images of digits from 0 to 9 with a pixel resolution of 28x28. Therefore, in a tensor representation the images have the shape 28x28x1. The goal is to classify what digit is drawn on a picture using a neural network with the following characteristics:

- an arbitrary amount of hidden layers, each with arbitrary amount of neurons
- sigmoid activation function for all hidden layers
- softmax activation function for the output layer
- cross entropy loss function. *

** For the implementation from scratch (Part (a)) we use a mean squared error (MSE) loss function. This is not recommended for a classification task, but we use it to simplify the implementation. In the future please consider using cross entropy / log loss instead.*

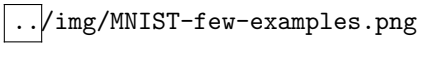

Figure 6: Example images from MNIST dataset with 28x28 pixel resolution.

The training set with n data points is of the form

$$(x^{(i)}, y^{(i)}),\ i = 1, ..., n, \tag{5}$$

with $y^{(i)} \in \{0,1\}^k$, where $y_j^{(i)} = 1$ when j is the target class, and 0 otherwise (i.e., the output is one-hot encoded). The softmax output function $\hat{y} = h_\theta(x)$ outputs vectors in $\mathbb{R}^k$, where the relative size of the $\hat{y}_j$ corresponds roughly to how likely we believe that the output is really in class j. If we want the model to predict a single class label for the output, we simply predict class j for which $\hat{y}_j$ takes on the largest value.

Tasks

1. Complete the implementation of the neural network classes marked by TODO comments:

   (a) From scratch (`network_scratch.py`)

   (b) Using PyTorch (`network_pytorch.py`)

   A repository with the template files will automatically be created for your team when registering for the GitHub classroom assignment at `https://classroom.github.com/a/XgELWE_C`.

2. In `network_tensorflow.py` a Neural Network class has been implemented using Tensorflow and Keras.

   (a) There are three mistakes in the code preventing the network from working correctly. Comment the mistakes out and repair the class.

   (b) Implement a time-based learning rate class `TimeBasedLearningRate`: it should be initialized with a positive integer as initial learning rate and have the learning rate reduced by 1 at each step, until a learning rate of 1 is reached.

3. In addition to the implementation, show how your networks perform on the MNIST classification task in `MNIST_classification.ipynb`.

   (a) Plot the accuracy of each neural network on the training and on the validation set as a function of the epochs

   (b) Please make sure that the uploaded Notebook shows printed training and validation progress.

   (c) Please add team number, team member names and matriculation numbers as a comment at the top of the notebook.

Please note that the classifier does not need to get anywhere close to 100% accuracy. Especially, with the small amount of training data this is difficult. Instead, the training output should indicate that the model learns something, i.e., the accuracy increases over the epochs on the training data and is significantly better than random guessing.

Submission

The submission of the whole problem set is done via GitHub classroom. You have to register for the assignment with your GitHub account at `https://classroom.github.com/a/XgELWE_C`. Please make sure that your GitHub account profile includes your real name. When starting the assignment, please create or join a team according to the teams assigned in Moodle (use the exact team name from Moodle).

Please upload / push all solutions to the GitHub repository which was created for your team. Please upload your solutions for the theoretical part (1) as a PDF to GitHub. If you upload multiple PDF files, please indicate in the file name to which subtask they are corresponding (we much prefer one single pdf). For the practical part (2), just push your code changes to the existing files.

Anybody in your team can push as often as they want. Once the deadline has passed, you are not able to push to your repository anymore and all changes on your **main** branch will be considered for grading. See the README.md in the repository for more details on how to push your changes to GitHub.

Sources

Aflak, Omar. Neural Network from scratch in Python. https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

PyTorch. Neural Networks. https://pytorch.org/tutorials/beginner/blitz/neural$_{n}etworks_{t}utorial.html$

TensorFlow 2. Quickstart for beginner. https://www.tensorflow.org/tutorials/quickstart/beginner