Université d'Orléans
Graduate School Orléans Numérique (GSON)

Introduction to Deep Learning
Problem Sheet 2

Work in a Colab notebook, with NumPy and Matplotlib only: no TensorFlow in this practical.

1) Gradient descent on a simple function

Visualize how gradient descent minimizes a scalar function. Consider $f(x) = x^2 + 2x + 1$.

1. Derive the analytical gradient $f'(x)$. Where is the minimum?
2. Starting from $x_0 = 5$, perform 20 steps of gradient descent with learning rate $\eta = 0.2$.
3. Plot $f(x)$ and the successive points $(x_k, f(x_k))$ on the curve.
4. Repeat with $\eta = 0.01$, $\eta = 1.0$ and $\eta = 1.1$. Discuss the effect of the learning rate on convergence.
5. Show that $x_{k+1} + 1 = (1 - 2\eta)(x_k + 1)$. For which learning rates does gradient descent converge? Compare with your observations.

2) Linear regression by gradient descent

Implement gradient descent manually for a simple regression model.

1. Generate 50 noisy data points around the line $y = 5x + 3$:

```
x = np.linspace(0, 10, 50) + np.random.uniform(-1, 1, 50)
y = 5 * x + 3 + np.random.uniform(-5, 5, 50)
```

2. The model predicts $\hat{y}_i = wx_i + b$. Define the mean squared error loss

$$E(w, b) = \frac{1}{n} \sum_i (wx_i + b - y_i)^2.$$

3. Derive the gradient and check that you obtain

$$\frac{\partial E}{\partial w} = \frac{2}{n} \sum_i x_i(wx_i + b - y_i), \quad \frac{\partial E}{\partial b} = \frac{2}{n} \sum_i (wx_i + b - y_i).$$

4. Implement the training in Python using NumPy only. Initialize $w = b = 0$, use the learning rate $\eta = 10^{-3}$, and train for 500 epochs. Plot the data with the fitted line, and the loss as a function of the epoch.
5. Compare (w, b) with the exact least-squares solution, `np.polyfit(x, y, 1)`. Which parameter has not converged? Increase η: what is the largest learning rate for which gradient descent still converges? Relate your observations to the shape of the loss surface (lecture 2).

3) Mini neural network: manual forward and backward pass

Implement a small fully connected neural network that performs regression on the Diabetes dataset, using the `dense` and `relu` classes of lecture 2 (appendix, and the notebook `neural_network_classes.ipynb` linked on the course website).

1. The file `diabetes_dataset.csv` is shared on Google Drive. In Colab, copy it into your session and load it (the first line of the file holds the column names):

```
!gdown 1XUGmFZk-unHBo31zA29aVDuuP53u4ZOH
data = np.loadtxt("diabetes_dataset.csv", delimiter=",", skiprows=1)
```

The dataset describes 442 patients: 10 input features (age, sex, body-mass index, blood pressure and six blood-serum measurements, already normalized) and, in the last column, a measure of disease progression one year later.

2. Shuffle the rows (fix the random seed), then split the dataset into a training (80%), a validation (10%) and a test set (10%).

3. Standardize the target with the mean $\bar{y}$ and the standard deviation σ of the *training* set: the network learns $y_s = (y - \bar{y})/\sigma$. An MSE computed on y_s, multiplied by σ^2, is the MSE in the original units.

4. Build the network

$$\text{Dense}(10,\ 16) \to \text{ReLU} \to \text{Dense}(16,\ 1)$$

as a list of layers: `network = [dense(10, 16), relu(), dense(16, 1)]`.

5. Implement the **forward pass**: propagate the input data through each layer and compute the predictions. Then compute the mean squared error (MSE) between predictions and targets:

$$E = \frac{1}{n}\sum_i (\hat{y}_i - y_i)^2.$$

6. Implement the **backward pass**:

   (a) Compute the gradient of the MSE with respect to the predictions:

$$\frac{\partial E}{\partial \hat{y}} = \frac{2}{n}(\hat{y} - y).$$

   (b) Pass this error through the layers in reverse order (`reversed(network)`), calling for each layer

```
error = layer.backprop(error, learning_rate)
```

   The layers sum their gradients over the examples: the factor $1/n$ appears only here, in the first error.

7. Train the network on the whole training set at each step, with $\eta = 0.01$, for 20 000 epochs (a few seconds). Every 100 epochs, record the training and the validation loss; plot both curves as a function of the epoch, in the original units. Compare with two baselines: always predicting the training mean, and linear regression (`np.linalg.lstsq`).

8. Evaluate the MSE on the test set, once, at the very end.

9. What happens if you skip the standardization of question 3.3, with the same learning rate? After a few epochs, how many hidden ReLUs are still active for at least one example?

4) Effect of initialization

Observe the impact of the weight initialization on convergence.

1. Repeat the training of your network from exercise 3 with different initialization schemes (biases at zero):

   - all weights set to zero;
   - random values from $\mathcal{N}(0, 1)$;
   - Xavier initialization ($\text{Var}(w) = \frac{2}{n_{in}+n_{out}}$);
   - He initialization ($\text{Var}(w) = \frac{2}{n_{in}}$).

   Plot the training loss as a function of the epoch for each case.

2. Why does the network with zero weights not learn?

3. Repeat with a deeper network: six hidden layers of 32 neurons with ReLU, 5000 epochs. Which initializations still train? Relate your observations to vanishing and exploding gradients. Compare the training and validation losses of the deep network.

Appendix: the `dense` and `relu` classes (lecture 2)

```python
import numpy as np

class dense():
    def __init__(self, n_in, n_out):
        std = np.sqrt(2 / n_in)          # He initialisation
        self.weights = std * np.random.randn(n_in, n_out)
        self.biases = np.zeros((1, n_out))

    def forward(self, data_in):
        self.input = data_in             # kept for backprop
        return data_in @ self.weights + self.biases

    def backprop(self, error_out, learning_rate):
        error_in = error_out @ self.weights.T    # dE/dX
        grad_weights = self.input.T @ error_out  # dE/dW
        grad_biases = error_out.sum(axis=0)      # dE/db
        self.weights -= learning_rate * grad_weights
        self.biases -= learning_rate * grad_biases
        return error_in

class relu():
    def forward(self, data_in):
        self.input = data_in
        return np.maximum(data_in, 0)

    def backprop(self, error_out, learning_rate):
        return error_out * (self.input > 0)
```