Technische Universität Hamburg
Institut für Mathematik
Lehrstuhl Numerische Mathematik

Jens-Peter M. Zemke
Jonas Grams

Mathematics of Neural Networks
winter semester 2021/2022
exercise sheet 2

Exercise 1: (2 points) Check $\mathsf{SoftAbs}_k(x)$ for monotonicity, symmetry, and asymptotic behavior and calculate the derivative.

Exercise 2: (3 points) Prove that any continous function

$$f : [a, b] \to [0, \infty)$$

can be approximated arbitrarily well by a neural network $N : \mathbb{R} \to \mathbb{R}$ with a single hidden layer comprising n neurons for some $n \in \mathbb{N}$ based on ReLU activations.

Hint: Can you express the linear interpolant through the n equidistant points

$$a = x_1, x_2, \dots, x_n = b \quad \text{and function values} \quad f_i = f(x_i), \quad i = 1, \dots, n$$

using linear combinations of ReLU functions $\max(0, x + b_i)$ with biases $b_1 = -(a - h), b_2 = -a, b_3 = -(a + h), b_4 = -(a + 2h), \dots, b_n = -(b - h)$, where $h = x_2 - x_1$?

Exercise 3: (3+3 points)

a) Add backpropagation to our feedforward neural network. You can use the given templates.
   - Add an attribute `data` to our `ReLU` class that stores data for the backpropagation from the last evaluation.
   - Add a method `backprop(self, delta)` to our `ReLU` class that performs the backpropagation $a'(\mathbf{z}) \circ \boldsymbol{\delta}$, where $\mathbf{z}$ can be recovered from `self.data`.
   - Add attributes `__a` and `__z`, both initialised with `None`, to our `DenseLayer` class which store the input and the affine linear combination $\mathbf{Wa} + \mathbf{b}$ from the last evaluation.
   - Add attributes `dW` and `db`, initialised with zeros, to our `DenseLayer` class that store the updates for the weights and biases.
   - Add the method `backprop(self, delta)` to our `DenseLayer` class that computes the $\boldsymbol{\delta}$ for this layer, the derivatives of the cost function with respect to the weights and the biases of this layer, and returns $\mathbf{W}^{\mathsf{T}}\boldsymbol{\delta}$ for the previous layer.
   - Implement the method `backprop(self, x, y)` in our `SequentialNet` class that computes and returns the derivatives of the cost function with respect to the weights and the biases for the training data `x` and `y` corresponding to one minibatch.

b) Implement `train(self, x, y, batch_size=16, epochs=10)` that uses repeated calls to `backprop()` to carry out `epochs` epochs of SGD with batch size `batch_size` for the training data `x` and `y` corresponding to the whole training set. For this
   - implement a class `SGD` which is initialised with a learning rate `eta` and has a method `update(self, data, ddata)` that performs a SGD update step on the data in the list `data` with update data in the list `ddata`,

- add an attribute `optim` to your `DenseLayer` class, which is is given on initialization and an instance of SGD by default,
- add a method `update(self)` to your `DenseLayer` class that updates the weights and biases for a layer using the `update` method from the optimizer given in `optim`.

Exercise 4: (2+2 points) In multiclass classification the function $\mathsf{SoftMax} : \mathbb{R}^n \to (0,1)^n$,

$$\mathbf{p} := \mathsf{SoftMax}(\mathbf{x}) := \frac{e^{\mathbf{x}}}{\mathbf{e}^{\mathsf{T}} e^{\mathbf{x}}} = \begin{pmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{pmatrix} / \Big(\sum_{j=1}^{n} e^{x_j} \Big),$$

which returns a probability distribution, is used as last layer together with the Cross Entropy Loss function

$$H(\mathbf{y}, \mathbf{p}) := -\sum_{j=1}^{n} y_j \ln(p_j)$$

as cost function. In training, the vector $\mathbf{y}$ will be a unit vector, $\mathbf{p}$ will be the outcome of the $\mathsf{SoftMax}$-layer. This is known as categorical cross entropy loss. Compute the derivative of the

a) $\mathsf{SoftMax}$ function $\mathbf{p} = \mathsf{SoftMax}(\mathbf{x})$ with respect to $\mathbf{x}$,

b) Cross Entropy Loss function combined with the $\mathsf{SoftMax}$ layer, $H(\mathbf{y}, \mathsf{SoftMax}(\mathbf{x}))$, with respect to $\mathbf{x}$.

How do you initialize backpropagation in this case?

Exercise 5: (1+1+2 points) Test your class `SequentialNet` or an implementation using TensorFlow/Keras with

a) the data from exercise 1b) of the first exercise sheet with one hidden layer comprising 3, 4, 10, or 100 neurons and ReLU activation functions,

b) the data given by

```
x = np.expand_dims(np.linspace(0, np.pi/2, 2000),0)
y = np.concatenate((np.sin(x), np.cos(x)))/2+.5
x_train, y_train = x[:,::2], y[:,::2]
x_test, y_test = x[:,1::2], y[:,1::2]
```

with one hidden layer of 100 neurons and ReLU activation functions,

c) MNIST with input layer of size 784, a hidden layer of size 100, and an output layer of size 10 with ReLU activation functions, either with the squared error cost function or a $\mathsf{SoftMax}$ layer followed by cross entropy loss. This data set is provided to you in the file `mnist.npz`. Further information can be found here:

   http://yann.lecun.com/exdb/mnist/

   Import the data set using `DATA = np.load('mnist.npz')` function followed by `x_train, y_train = DATA['x_train'], DATA['y_train']` and `x_test, y_test = DATA['x_test'], DATA['y_test']`.
   Note that you need to reshape the data before you can train your neural net.

If you want to use TF/Keras you can find a description on

https://keras.io/getting-started/sequential-model-guide/

and a code skeleton in `TF_skeleton.py`.

Exercise 6: (2+2 points) In this exercise we consider part of the proof of the backpropagation. Let

$$\mathbf{z} = \mathbf{W}\mathbf{a} + \mathbf{b}, \quad \mathbf{z}, \mathbf{b} \in \mathbb{R}^m, \quad \mathbf{a} \in \mathbb{R}^n, \quad \mathbf{W} \in \mathbb{R}^{m \times n}$$

denote the affine mapping in one of the layers. Let C be a cost function and let

$$\boldsymbol{\delta} = \frac{\partial C}{\partial \mathbf{z}} = \nabla_{\mathbf{z}} C \in \mathbb{R}^m$$

be given. Let the 3-tensor[1] $\mathbf{T} \in \mathbb{R}^{m \times m \times n}$ be defined by elements

$$t_{i,j}^{(\ell)} := \frac{\partial z_\ell}{\partial w_{i,j}}$$

for $\ell = 1, \dots, m$, $i = 1, \dots, m$, $j = 1, \dots, n$.

a) Compute for $\ell = 1, \dots, m$ the 2-tensor slices

$$\mathbf{T}^{(\ell)} := \begin{pmatrix} t_{1,1}^{(\ell)} & \cdots & t_{1,n}^{(\ell)} \\ \vdots & \ddots & \vdots \\ t_{m,1}^{(\ell)} & \cdots & t_{m,n}^{(\ell)} \end{pmatrix} \in \mathbb{R}^{m \times n}.$$

b) Show that the application of the 3-tensor $\mathbf{T}$ to the 1-tensor $\boldsymbol{\delta}$ gives the same 2-tensor as obtained using the elementwise chain rule,

$$\mathbf{T}\boldsymbol{\delta} := \sum_{\ell=1}^{m} \mathbf{T}^{(\ell)} \mathbf{e}_\ell^\top \boldsymbol{\delta} = \frac{\partial C}{\partial \mathbf{W}} \in \mathbb{R}^{m \times n}$$

[1] A 3-tensor is a natural generalization of a scalar (0-tensor), column vector (1-tensor), and matrix (2-tensor) to an additional dimension. Think of it as a cube of numbers, or as matrices stacked on top of each other towards the reader. Here, it makes sense to think of it as a row vector of matrices.