Assignment 3: Optimization and Regularization
MA-INF 2313: Deep Learning for Visual Recognition

Theoretical Task Due Date:	7.12.2020
Practical Task Due Date:	7.12.2020
Assistants:	Jan Müller, Leif Van Holland

1 Theoretical Task (15 pts)

a) Fundamentals of Unconstrained Optimization (*8 points*)
The Rosenbrock function is a non-convex function, which is commonly used to demonstrate the behaviour of an optimizer or used to evaluate its performance. For a point $x \in \mathbb{R}^2$ the Rosenbrock function is defined as

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{1}$$

1. Compute the gradient $\nabla f(x)$ and Hessian $\nabla^2 f(x)$ of the Rosenbrock function.
2. Show that $x^* = (1, 1)^T$ is the only local minimizer of this function, and that the Hessian matrix at that point is positive definite.
3. Show that the function $f(x) = 8x_1 + 12x_2 + x_1^2 - 2x_2^2$ has only one stationary point, and that it is neither a maximum or minimum, but a saddle point.

b) Case Study *(7 points)*
For this question, we consider a MLP with one hidden layer: two input units, three hidden units, one output unit. The non-linearity being used is sigmoid, the error function is given by the square of the euclidian distance. A gradient descent is performed with 'online' learning, meaning that the error is backpropagated after each sample iteration. Below is the learning curve of the error on the training set.

(a) For each training sample, the error function can be seen as a function of the parameters of the network. If the network has n parameters, the graph of this function gives a hypersurface in $\mathbf{R}^{n+1}$, also called the error surface. How would you describe what this error surface looks like around the parameters obtained between the times corresponding to iteration 100^{th} and iteration $10,000^{th}$? How is such an area of the error surface usually called ?

(b) If you could choose a property that the Hessian matrix of the error function would satisfy at, say, iteration 100, what would you choose?

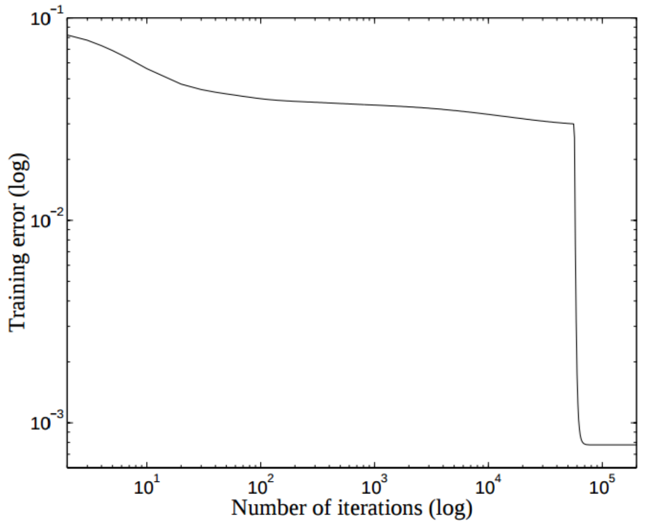

Figure 1: K.Fukumizu, S.Amari. "Local minima and plateaus in hierarchical structures of multilayer perceptrons."

2 Programming Exercises (15 pts)

In the last lecture you learned about (stochastic) gradient descent as the fundamental optimization principle used in deep learning frameworks. In this programming exercise we will expand gradient descent with a technique called momentum. The goal of a momentum-based optimization algorithm is to increase its convergence speed and to reduce the likelihood to prematurely converge to a local minimum. These two properties are achieved by accumulating the gradient across multiple time steps with an exponential decay. Let L be a differentiable loss function, $\{(x_i, y_i)\}_{i\in[1:m]}$ be the dataset, and $f(\cdot;\theta)$ be our differentiable model with parameter θ. To perform gradient descent with momentum, we have to introduce a velocity $v^{(t)}$ (with $v^{(0)} = 0$), which is influenced by previous and currents gradients. The exponential decay of previous gradients is controlled by a parameter $\alpha \in [0, 1]$ whereas the effect of the current gradient is controlled by the learning rate $\eta \in [0, 1]$. The update step in a momentum-based gradient descent is

$$\begin{aligned} v^{(t+1)} &:= \alpha v^{(t)} - \eta \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)}) \right) \\ \theta &\leftarrow \theta + v^{(t+1)}. \end{aligned}$$

Another very similar approach is the "Nesterov Accelerated Gradient" (often shortened to NAG). Intuitively, the NAG computation "trusts" the current velocity $v^{(t)}$ and uses it to compute the function value with parameter $\theta + \alpha v^{(t)}$. The resulting gradient is then used to update the velocity and the parameter by applying the following update rule:

$$v^{(t+1)} := \alpha v^{(t)} - \eta \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta + \alpha v^{(t)}), y^{(i)}) \right)$$
$$\theta \leftarrow \theta + v^{(t+1)}.$$

We can use both momentum as well as the NAG in "PyTorch" without much effort. Consider PyTorch's stochastic gradient descent class:

```
torch.optim.SGD(params, lr=<required parameter>, momentum=0, nesterov=False),
```

To enable momentum during training you only need to set the argument `momentum` to a value greater than 0, and to enable the use of the NAG procedure you additionally pass `nesterov=True` as an argument.

a) Visualization of an unconstrained optimization *(5 points)*
You first task is to visualize the different paths a gradient descent optimizer takes to find a (local) minimum of the Rosenbrock function.

1. Visualize the Rosenbrock function (see Equation 1) as a surface plot in the interval $[-2, 2]^2$.
2. Implement a function `optimize` which takes an parameter $x \in \mathbb{R}^2$, a "PyTorch" optimizer, and a lower bound l as its arguments. Your function has to use the optimizer to update the parameter x to find a local minimum of the Rosenbrock function but you can stop the interation if $f(x) \leq l$. In addition, your function has to return a list of positions $(x^{(t)}, f(x^{(t)}))$ where $x^{(t)}$ is the value of the parameter after t-th update step.
3. Visualize the path the parameter took to reach a local minimum when optimized with a SGD optimizer without momentum, with momentum and with NAG momentum (visualize the behavior for $\alpha \in \{0.1, 0.5, 0.9\}$ in both cases). Set the lower bound to $l = 0.01$, the learning rate to $\eta = 0.0001$, and start every optimization run at the point $x = (-1.5\ 1.5)^T$.

b) Constrained optimization *(5 points)* In the second task you will implement your own PyTorch optimizer class. Your optimizer has to use the gradient direction to update the the parameter while ensuring that $\|x\|_2 = 1$ after every step. We will then use this optimizer to find a solution to the constrained optimization problem

$$x^* = \arg\min_{x; \|x\|=1} f(x) \text{ where } f \text{ is defined in Equation 1.} \tag{2}$$

In order to implement you own optimizer we will extend the class `torch.optim.Optimizer` which handles most of the PyTorch specific details for us.

1. Implement a class `ConstraintOptimizer` which inherits the class

   `torch.optim.Optimizer`.

   The constructor of your class has to take a list of parameters and a learning rate as its attributes. Create a dictionary which contains a key `lr` with the learning rates as its value. Call the constructor of the parent class and pass the parameter list and the dictionary as arguments.

2. Implement a member function `step(self, closure=None)`, which uses the decorator `@torch.no_grad()`. This function has to update each parameter in the parameter list. For more details, check out the reference implementation of the SGD class. Instead of the standard additive update rule for gradient descent, you have to implement a custom update rule:

   - Use the gradient direction to identify a parametric curve on the unit sphere $\mathcal{B}_{\mathbb{R}^2}$. Hint: Recall the parameterization of a sphere.
   - Take a step along the curve in gradient direction and set the new weight as the resulting position on the unit sphere. The step size should be proportional to the learning rate. The value of a tensor `p` can be replaced by calling `p.set_(newP)`.

3. Use your optimizer class to find a solution to Equation 2 and visualize both the function $f(x)$ on $x \in \{x \in [-1,1]^2 \mid \|x\| = 1\}$ and the parameter in every iteration. Start the optimization at the point $x = \frac{1}{\sqrt{2}}[-1,1]$, and set the learning rate to $\eta = 0.05$.

c) Improve your classifier *(5 points)*
For this task, you can build on the implementation of your MLP, which you used to classify the CIFAR-10 dataset. Make sure that your MLP has at least **2 hidden layers** with **512 hidden units** in each layer with **ReLU** as the activation function. The second hidden layer is followed by a **softmax layer** with 10 classes. Use cross-entropy loss as the loss function; it is often the most appropriate when working with logistic or softmax output layers. Using this MLP as a starting point, create the following variants by modifying the training-loss function and parameter update routines as shown in the table.

Type	REG	OPT
MLP I	-	SGD
MLP II	-	NESTEROV
MLP III	L1	NESTEROV
MLP IV	L2	NESTEROV

Here we have two different types of parameter update routines: 'SGD' which refers to stochastic gradient descent and 'Nesterov' which refers to SGD with Nesterov momentum. To safeguard against over-fitting, we also use the l_1 and l_2 regularization. Keep the number of epochs fixed at 25. In order to compute the l_1 or l_2 regularization you can apply torch.linalg.norm to the parameters of your model. Recall that you can access the parameters of any class which inherits torch.nn.Module by calling `module.parameters()`.

1. Plot the validation loss vs epochs for each of the variants (MLP I to IV) on the same plot. Which variant converges the fastest in training?
2. Report the classification accuracy for each of the variants on the CIFAR-10 dataset.

Also, feel free to play with the hyper-parameters and report the network with the highest classification accuracy on the test set.