Optimization for Learning - FRTN50
Assignment 3

Written by: Martin Morin, Mattias Fält

Latest update: October 18, 2019

Introduction

The topic of this hand-in is practical implementations. It will cover some of the basic ideas for solving huge problems and the concept of automatic differentiation/back-propagation. The SVM from the previous hand-in will be revisited together with new neural network based problems.

When problem sizes grow the cost of evaluating gradients grows with it, making gradient descent more and more expensive. Current research is therefore directed towards making approximate gradient evaluations in order to make each iterations cheaper. This is the idea behind coordinate gradient, stochastic gradient, and their many variants. In this hand-in you will implement variants of both.

Back-propagation can in part be contributed to the resurgence of neural networks and deep learning in recent years. However, the basic concept is not new since back-propagation is nothing more than an automated way of calculating gradients for neural networks. Calculating gradients by hand and implementing specific gradients for every new neural network structure is time consuming and error prone. Automatic differentiation (AD) solves this problem and efficient AD implementations are an integral part of machine learning libraries like `TensorFlow`, `PyTorch` and `Flux.jl`. To get a feel for what these libraries do, you will implement a basic back-propagation algorithm for dense neural networks.

Acceleration

We will start with (Nesterov) acceleration. Acceleration is not necessarily related to large-scale problems or neural networks but is a useful trick that is worth knowing since it can speed up convergence.

The idea is simple. Ordinary proximal gradient computes the next iterate based only on the current iterate, completely forgetting the trajectory. However, the trajectory can contain useful information about the function. Accelerated methods (also called momentum methods) stores some information regarding the previous iterates and extrapolate based on this information. There are a couple different ways of doing this but here we choose the following

$$\begin{aligned} x^{k+\frac{1}{2}} &= x^k + \beta^k (x^k - x^{k-1}) \\ x^{k+1} &= \mathrm{prox}_{\gamma g}(x^{k+\frac{1}{2}} - \gamma \nabla f(x^{k+\frac{1}{2}})) \end{aligned}$$

with $x^{-1} = x^0$. The sequence β^k is here a design choice and several different choices have been presented in literature.[1] A simple choice is

$$\beta^k = \frac{k-2}{k+1} \tag{1}$$

while another common choice is

$$\beta^k = \frac{t^k - 1}{t^{k+1}}, \quad t^0 = 1, \quad t^{k+1} = \frac{1+\sqrt{1+4(t^k)^2}}{2}. \tag{2}$$

These two choices mainly differ in the first tens of iterations. If f is μ-strongly convex the following choice can be made

$$\beta^k = \frac{1-\sqrt{\mu\gamma}}{1+\sqrt{\mu\gamma}}. \tag{3}$$

Task 1 For the SVM problem from the previous hand in, implement the accelerated proximal gradient method above. Use the hyper-parameters that gave the best performance in the previous hand-in.

Compare ordinary proximal gradient with (3) and one of (1) and (2). Use the same initial point x^0 and step size $\gamma = \frac{1}{L}$ for all methods, where L is the smoothness-constant of f. For μ in (3) you could either calculate the strong-convexity parameter directly or tune it until you get convergence.

Plot $\|x^k - x^\star\|$ where $x^\star$ is the solution. Find $x^\star$ by simply solving the problem to extra high precision before generating your plot. Is the convergence rate improved over the non-accelerated method? Comment on the behavior seen in the plot.

Coordinate Gradient Descent

If the proximable-term, g, in the proximal gradient method is separable $g(x) = \sum_{i=1}^n g_i((x)_i)$ it is possible to perform coordinate-wise updates,

$$\begin{aligned}
&\text{Sample } i \text{ uniformly from } \{1, ..., n\} \\
&(x^{k+1})_i = \text{prox}_{\gamma_i g_i}((x^k)_i - \gamma_i(\nabla f(x^k))_i) \\
&(x^{k+1})_j = (x^k)_j, \quad \forall j \neq i
\end{aligned}$$

where $x^k \in \mathbf{R}^n$. Index i could be selected in a number of different ways but here we will simply sample it uniformly and independently in each iteration.

In general, coordinate gradient requires more iterations to converge compared to standard proximal gradient. However, if one coordinate of ∇f can be computed a factor m cheaper compared to the full gradient, and the number of iterations required are less than a factor m more, there is still a potential for a real world speed up.

One such case where coordinate computation is cheap is when f is a quadratic, $f(x) = \frac{1}{2}x^T Q x + q^T x$. One coordinate can be evaluated up to n times cheaper than a full gradient. This can be seen from the expression of the gradient,

$$\nabla f(x) = Qx + q = (Q_1 x + (q)_1, ..., Q_n x + (q)_n)$$

[1]There are in some sense optimal choices but it is still useful to see this as a parameter to tune for your specific problem.

where Q_i is the i:th row of Q. More general assumptions for cheap coordinate evaluations exist but for the purposes of this hand-in the quadratic is sufficient.

It is possible to use coordinate-wise step-sizes, γ_i, for coordinate gradient. This allows the step-size to be adapted to the curvature/smoothness in each direction instead of defining a step-size based on the global smoothness constant. This can greatly reduce the number of iterations needed. For more information about coordinate gradient we recommend studying exercises 5.14, 5.16, and 5.17.

Task 2 For the SVM problem from the previous hand in, implement the coordinate proximal gradient method. Use the hyper-parameters that gave the best performance in the previous hand-in.

Compare the coordinate method to the ordinary and accelerated proximal gradient methods from Task 1. Use the same initial point x^0 for all methods and use the same algorithm-parameters for the ordinary and accelerated proximal gradient as in Task 1. For coordinate gradient, compare both the coordinate-wise step-size of $\gamma_i = \frac{1}{Q_{ii}}$ and the uniform choice $\gamma_i = \frac{1}{L}$ from ordinary gradient descent. Q_{ii} is here the i:th diagonal element of the Q matrix of the function f.

Plot $\|x^k - x^\star\|$ but for coordinate gradient scale the x-axis (iteration axis) with $\frac{1}{n}$ to normalize the computational cost. Each iteration of coordinate gradient should be $\frac{1}{n}$:th as expensive proximal gradient, so n iterations of coordinate gradient should correspond to the actual computational effort compared to one iteration of ordinary/accelerated proximal gradient.

Which method required the least amount of computational effort? Which method required the least amount of iterations? Which was fastest in real time? Can you comment on the similarities/differences between real time performance and number of iterations needed? How fair is it to compare real time performance? Can it be easily affected?

A few implementation notes.

- `ProximalOperators.jl` expects array inputs and returns array for most functions. Even if the input/output is just a scalar it needs to be wrapped in a one element vector.
- Be mindful of the code inside your update loop. Any operation involving the full iterate, x^k, is likely to add a considerate amount of computational time to each iteration. This includes any type of logging, printing and convergence checking. It is therefore advisable to only do this every 1000-10000 iteration.
- The macro `@time` can be used to time a function call in `Julia`. For example `@time x = rand()` will print the amount of time it took to generate a random number and store it in `x`.

Neural Networks

This part of the hand-in is designed to give an understanding of how the training of neural networks works. To do this, you should implement a neural network and the training more or less from scratch.

When working with datasets that are very large, it is not practical to compute the full gradient over all the data. It is therefore common to do training over *batches*. This means that a gradient is computed over a random subset of the data, and a stochastic gradient step is taken. To make the implementation in this exercise easier, we will work with batches of size 1, i.e. we compute the gradient with respect to one input data point and one output label, and update the parameters in the network accordingly. This makes the variance of the stochastic gradients larger, but greatly simplifies implementation.

You will be implementing back-propagation and the ADAM algorithm. The ADAM algorithm is a version of stochastic gradient, where each parameter gets individual step-lengths and extra care is taken to handle the variance of the stochastic gradients.

The file `backprop.jl` will include an outline for most of the network and training, but you will fill in some of the functions. Places where you are expected to add code are marked with `#+++`.

Make sure to test that each part that you write works as expected. It is much easier to detect errors this way than trying to debug the final result.

Remember to read the introduction to Julia document if you are unsure about some of the concepts. In particular, it is important to understand the section on *References and in-place operations*.

Model

We will implement a simple neural network consisting only of `Dense` layers of the form

$$l_i(x) = \sigma_i(W_i x + b_i)$$

where $W_i \in \mathbb{R}^{m_i \times n_i}, b_i \in \mathbb{R}^{m_i}$ and σ_i is some activation function that is applied element-wise. The code is written so that it would be possible to implement more types of layers, but for simplicity, these are not included in the hand-in.

For some input $x \in \mathbb{R}^{n_1}$, the network n is defined as applying each layer to the output of the previous layer

$$n(x) = l_k(l_{k-1}(...l_2(l_1(x))...))$$

where k is the number of layers.

To measure how well the model approximates some output y, we define a cost function $L(n(x), y)$ that penalize the error between the model output $n(x)$ and the true output y in some way. The training problem can the be formalized as minimizing the total cost for all known data

$$\min_{(W_i,b_i)\forall i} \sum\nolimits_{(x,y)\in D} L(n(x), y)$$

where D is the set of all known data. As stated before, when D is large, computing the gradient w.r.t. W_i and b_i of the full sum $\sum_{(x,y)\in D} L(n(x), y)$ is expensive. Instead a stochastic approach is used, one pair (x, y) is drawn at random from D and backpropagation/gradient calculation is performed on $L(n(x), y)$.

Activation Functions

Start by defining a few activation functions. The definition of the sigmoid and its derivative is already implemented.

```julia
sigmoid(x::Float64) = exp(x)/(1 + exp(x))

derivative(f::typeof(sigmoid), x::Float64) = sigmoid(x)*(1-sigmoid(x))
```

Note that the functions are defined for `Float64`, so to apply them to a vector you need to use broadcasting

```julia
sigmoid(1.0) #0.7310...
sigmoid.([0.0, 1.0]) # [0.5, 0.7310]
```

The function `derivative` is defined so that the first argument is the function we want to calculate the derivative of

```julia
# get derivative of sigmoid function at the point 0.0
derivative(sigmoid, 0.0) # 0.25
derivative(relu, 2.0) # 1.0
```

Implement the functions

$$\text{relu}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}, \quad \text{leakyrelu}(x) = \begin{cases} 0.2x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

and functions to calculate the derivatives. Make sure to test that they work as expected.

Dense Layer

The type `Dense` and a constructor has already been defined for you. The type includes the matrix W, the vector b, the activation function σ as well as fields for storing some intermediate values and the output of the layer. The latter two will be needed later when we do back-propagation.

The constructor takes arguments for: the size of the output, size of input, activation function, as well as some arguments that can be used to tune the mean and variance of the initial random weights.

A function to evaluate a Dense layer `(l::Dense)(z)` has been defined, but you will fill in the code. The definition is written so that it should be possible to run

```julia
z = [1.0, 2.0]
l1 = Dense(3, 2, relu)
# Evaluate layer
l1(z) # vector of length 3
```

Make sure that your implementation stores the intermediary value $Wz + b$ in the field `l.x` and the final output in `l.out`.

Network

A network is defined as nothing more than a list of layers.

```
l1 = Dense(3, 2, relu) # 3 outputs, 2 inputs
l2 = Dense(1, 3, relu) # 1 output, 3 inputs
n = Network([l1, l2])  # 1 output, 2 inputs
```

Fill in the function `(n::Network)(z)` so that it computes the result of applying each layer to the previous output.

```
z = [1.0, 2.0]
out = n(z) # Should return a vector of length 1 for the example above
```

Make sure to verify that each `Dense` layer saved their intermediary values `x` and `out` and that they are correct.

Back-Propagation of a Layer

You are now ready to start implementing the back-propagation.

To optimize over the variables W_i and b_i we need to be able to compute the derivative of the cost with respect them. W_i and b_i will change at every iteration in the training, but to simplify notation, we will not explicitly write iteration indices and we do not explicitly write that $L(\hat{y}, y)$ and $\hat{y} := n(x)$ are functions of these variables.

The derivatives of the cost L with respect to the elements in W_i and b_i are given by the Jacobians $\frac{\partial L}{\partial W_i}$ and the gradients $\nabla_{b_i} L$. The Jacobian is defined for each element j, k as $(\frac{\partial L}{\partial W_i})_{j,k} := \frac{\partial L}{\partial (W_i)_{j,k}}$, and the gradient is a column vector defined as $(\nabla_{b_i} L)_j := \frac{\partial J}{\partial (b_i)_j}$. [2]

If we are able to compute these derivatives, it is possible to optimize over them, for example with gradient steps:

$$
\begin{aligned}
&W_i - \gamma \frac{\partial L}{\partial W_i} \\
&b_i - \gamma \nabla_{b_i} L
\end{aligned}
$$

The calculation of these derivatives, known as back-propagation, can be summarized as follows:

Define $\delta_{k+1} := \frac{\partial L}{\partial \hat{y}}$ and

$$
\begin{aligned}
z_{i+1} &:= \sigma_i(W_i z_i + b_i) \\
\bar{z}_i &:= W_i z_i + b_i \\
\delta_i &:= W_i^T(\delta_{i+1} \odot \sigma_i^{'}(\bar{z}_i)),
\end{aligned}
$$

[2]The Jacobian and the gradient are the same thing in the sense that they both are collections of all first partial derivatives. Different conventions exist for both and the choice between them is based on notational convenience.

for $i = 1, ..., k$, where $\odot$ is element-wise multiplication.

Using the chain rule we get the following property

$$\frac{\partial L}{\partial z_i} = \delta_i^T,$$

and

$$\begin{aligned}\frac{\partial L}{\partial W_i} &= \delta_{i+1}^T \frac{\partial z_{i+1}}{\partial W_i} = \left(\delta_{i+1} \odot \sigma_i^{'}(\bar{z}_i)\right) z_i^T \\ \frac{\partial L}{\partial b_i} &= \left(\delta_{i+1} \odot \sigma_i^{'}(\bar{z}_i)\right)^T\end{aligned}$$

i.e.

$$\begin{aligned}\nabla_{b_i} L &= \left(\delta_{i+1} \odot \sigma_i^{'}(\bar{z}_i)\right) \\ \frac{\partial L}{\partial W_i} &= (\nabla_{b_i} L) z_i^T \\ \delta_i &= (W_i^T \delta_{i+1}) \odot \sigma_i^{'}(\bar{z}_i) = W_i^T (\nabla_{b_i} L)\end{aligned}$$

We now see that it is possible to calculate the gradient and Jacobian as well as the new δ_i if we have access to δ_{i+1} from the next layer, the derivative of the activation function σ_i, the intermediary value $\bar{z}_i$, as well as W_i and b_i.

Now implement the function `backprop!(l::Dense, δnext, zin)` so that it computes and stores the values ∇b, ∂W and δ (corresponding to $\nabla_{b_i} L$, $\frac{\partial L}{\partial W_i}$, δ_i) in the layer, and returns δ.

It is again important that you test that your implementation is correct. We can do this by making small adjustments to the variables and see if the derivatives seem to be correct. For example, you can test that δ is computed correctly using:

```julia
l = Dense(1, 3, sigmoid) # 1 output, 3 inputs
z1 = [1.0, 2.0, 3.0]
out1 = copy(l(z1)) # Vector with 1 number, make a copy so next call won't
↪  overwrite the old result

# Derivative of output with respect to itself
δnext = [1.0]
# Calculate gradients
δ = backprop!(l, δnext, z1)

# Try with a slightly different input in second input
z2 = [1.0, 2.0 + 0.0001, 3.0]
out2 = copy(l(z2))

(out2-out1)./0.0001 # Should be roughly the same as δ[2]
```

And for the stored gradients:

```julia
l = Dense(2, 3, sigmoid) # 2 output, 3 inputs
z1 = [1.0, 2.0, 3.0]
```

```julia
out1 = copy(l(z1))

# Derivative of second output with respect to itself
δnext = [0.0, 1.0]
backprop!(l, δnext, z1)

δW = copy(l.∂W) # The first row should be zero, it doesn't affect second output

# Make a small change to l.W
l.W[2,3] += 0.0001
# Run with the same input
out2 = copy(l(z1))

# Second element here
(out2-out1)./0.0001
# should be roughly the same as
δW[2,3]
```

Back-Propagation of the Network

We now want to do the back-propagation through the whole network using the function we already defined for a layer. Fill in the missing lines in `backprop!(n::Network, input, ∂J∂y)`. Note that the input to each layer is saved as the output in the previous layer.

Make sure to test the code again. The easiest way is to define a `Network` with one output and setting `∂J∂y=[1.0]`. Each layer can be accessed using `n.layers[i]`. Make a small adjustment to one of the `W` or `b` and see that the output changes according to `∂W` or `∇b` when called with the same input.

Loss Function

You have now implemented code to evaluate a network and calculate gradients of the output with respect to its parameters. A function for retrieving all the parameters and gradients is already implemented for you. To be able to train the network we need a loss function that evaluates how well the network is doing in approximating some wanted output y. A simple least squares penalty has been implemented for you

```julia
sumsquares(yhat,y) =  norm(yhat-y)^2
derivative(::typeof(sumsquares), yhat, y) =  yhat - y
```

which also includes the code for calculating the derivative of the loss with respect to the output of the network $\hat{y}$: $\partial L(\hat{y}, y)/\partial\hat{y}$

Training - Stochastic Gradient

All the pieces for training a network is now in place. A simple stochastic gradient step could now be implemented as

```julia
function gradientstep!(n, lossfunc, x, y)
    out = n(x)
    # Calculate (∂L/∂out)ᵀ
    ∇L = derivative(lossfunc, out, y)
    # Backward pass over network
    backprop!(n, x, ∇L)
    # Get list of all parameters and gradients
    parameters, gradients = getparams(n)
    # For each parameter, take gradient step
    for i = 1:length(parameters)
        p = parameters[i]
        g = gradients[i]
        # Update this parameter with a small step in negative gradient
        ↪  direction
        p .= p .- 0.001.*g
        # The parameter p is either a W, or b so we broadcast to update all the
        ↪  elements
    end
end
```

where x is some input vector and y is the desired output vector. Try this function with some random network to see if it improves it in the right direction

```julia
# Define a network with 1 input and 1 output
n = Network([Dense(3, 1, sigmoid), Dense(1, 3, sigmoid)])
x = randn(1)
y = [1.0] # We want the output to be 1

n(x) # This is probably not close to 1

gradientstep!(n, sumsquares, x, y)

n(x) # Now it is hopefully be slightly closer to 1
```

However, standard stochastic gradient descent can be quite slow, so instead you will use the ADAM method.

ADAM

The ADAM algorithm is a common choice in deep-learning. It is similar to the stochastic gradient method, but instead of taking a step in the direction of the stochastic gradient, it takes a step in the direction of a weighted average of the previous gradients. It also weights each step based on the weighted variance of previous gradients.

For each parameter p and time step t in the network, the ADAM algorithm does the following update

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla p_{t-1} \\ \hat{m}_t &= \frac{m_t}{1-(\beta_1)^t} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla p_{t-1}\right)^2 \\ \hat{v}_t &= \frac{v_t}{1-(\beta_2)^t} \\ p_t &= p_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} \end{aligned}$$

where $\gamma > 0$, $\beta_1 \in [0,1)$, $\beta_2 \in [0,1), \epsilon > 0$ are parameters and ∇p_{t-1} is the computed gradient of parameter p at time $t-1$ (i.e. when the parameters were p_{t-1}). The variables m and v are estimations of mean and variance of the gradients for each parameter, and $\hat{m}, \hat{v}$ are bias corrected versions of those.

A structure called `ADAMTrainer` has been created for you that stores all of these values for each parameter. It also keeps track of all the parameters and gradients in the network. Note that `p = params[i]` in the code is a reference to one of the matrices/vectors W or b, and not a single element in these matrices. The ADAM algorithm above therefore needs to be applied element-wise to `p`.

For example, the field `ms` is a vector, so that `ms[i]` can contain the matrix/vector m_t that corresponds to `params[i]`. Similarly, `mhs[i]` should contain the $\hat{m}_t$ corresponding to `params[i]`.

The constructor for the `ADAMTrainer` is already written. It creates all the necessary vectors and matrices to keep track of the variables. You should now complete the implementation of `update!(At::ADAMTrainer)` so that it updates the parameters according to the equations above.

Training - ADAM

The last part of the implementation is the function that will do the training. Finish the implementation of the function `train!(n, alg, xs, ys, lossfunc)` that accepts a network `n`, an algorithm (`ADAMTrainer`), a list of input data points `xs`, a list of corresponding true outputs `ys` and a loss-function `lossfunc`. For each input/output it should run the network, calculate the gradients and call the algorithm to update the parameters.

Remember that you are now doing non-convex optimization, and you may therefore get stuck in a local minimum. If the network performance seems to be too bad, try to redefine the network (which randomizes the initial weights), and re-run the training.

Task 3 We now want to train the network to approximate some function f. To make sure that everything works as expected, run the `### Task 3` code in `backprop.jl` to make sure that the pre-defined network can approximate the function $f(x) = \min(\|x\|_2^2, 3)$. Training over all the data 100 times should result in an average loss of less than 0.001. Plot the result and make sure it looks reasonable. Compare average loss on training data and the test data, how do they compare? If you continue training, how do these values change?

With 2000 data-points and a decently optimized network, one pass over the data-set takes roughly 0.2s on a 3.6GHz processor and allocates less than 20MiB of data, as reported by `@time`. If your code is many times slower, you might consider optimizing the code, for example using in-place operations.

Task 4 Add noise to the training data using the following line

```
ys = [fsol(xi).+ 0.1.*randn(1) for xi in xs]
```

and retrain the network.

Remember to redefine the layers, network and ADAMTrainer so that you start from scratch. What error do you get on the training vs test data now? Explain the differences and what they mean.

Task 5 Decrease the number of points in the training data from 2000 to 30, and redo Task 4. Since we have less data you have to increase the number of times you train over the data significantly. How do the results change compared to Task 4?

Task 6 Change back to 2000 data-points in the training set. We now want to approximate $f(x) = \min(0.5 \sin(\|x\|_2^2), 0.5)$ with $x \in \mathbb{R}^2$. Modify the network so that the first layer has 2 inputs and update the data generation accordingly. We again want to train the network with 2000 data-points uniformly in $[-4, 4] \times [-4, 4]$, without noise. What is the lowest error you are able to get without changing the network structure? Report the number of iterations you used. What happens if you resume the training with a lower learning rate $\gamma = 10^{-5}$ instead of 10^{-4}? Why?

Task 7 Reset the learning rate to 10^{-4}. Make a quick try to redo the training with `relu` instead of `leakyrelu` as activation functions. What happens? Give some plausible explanation.

Submission

See the latest version of the course program for instruction on how to submit the assignment. Your submission should contain the following.

- Your code with your implementation from Task 1 and 2.
- The `backprop.jl` file with full implementation of the network and training.
- A single pdf containing the following:
  - A couple of paragraphs describing/commenting on your findings for each of the Tasks 1-7.

Use plots, figures and tables to motivate your answers when possible.