DEEP LEARNING – WORKSHEET 5

Q1 to Q8 are MCQs with only one correct answer. Choose the correct option.

1. Which of the following are advantages of batch normalization?
   A) Reduces internal covariant shift.
   B) Regularizes the model and reduces the need for dropout, photometric distortions, local response normalization and other regularization techniques.
   C) allows use of saturating nonlinearities and higher learning rates.
   D) All of the above
   ANS D
2. Which of the following is not a problem with sigmoid activation function?
   A) Sigmoids do not saturate and hence have faster convergence
   B) Sigmoids have slow convergence.
   C) Sigmoids saturate and kill gradients.
   D) Sigmoids are not zero centered; gradient updates go too far in different directions, making optimization more difficult.
   ANS
3. Which of the following is not an activation function?
   A) Swish B) Maxout
   C) SoftPlus D) None of the above
   ANS B
4. The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer. True/False?
   A) True B) False
   ANS A
5. In which of the weights initialisation techniques, does the variance remains same with each passing layer?
   A) Bias initialisation B) Xavier Initialisation
   C) He Normal Initialisation D) None of these
   ANS B
6. Which of the following is main weakness of AdaGrad?
   A) learning rate shrinks and becomes infinitesimally small
   B) learning rate doesn't shrink beyond a point
   C) change in learning rate is not adaptive
   D) AdaGrad adapts updates to each individual parameter
   ANS A
7. In order to achieve right convergence faster, which of the following criteria is most suitable?
   A) momentum and learning rate both must be high
   B) momentum must be high and learning rate must be low
   C) momentum and learning rate both must be low
   D) momentum must be low and learning rate must be high
   ANS C
8. When is an error landscape is said to be poor(ill) conditioned?
   A) when it has many local minima
   B) when it has many local maxima
   C) when it has many saddle points and flat areas
   D) None of these
   ANS C

Q9 and Q10 are MCQs with one or more correct answers. Choose all the correct options.

9. Which of the following Gradient Descent algorithms are adaptive?
   A) ADAM B) SGD
   C) NADAM D) RMS Prop.

ANS A

10. When should an optimization function (gradient descent algorithm) stop training:

A) when it reaches local minimum B) when it reaches saddle point

C) when it reaches global minimum

D) when it reaches a local minima which is similar to global minima (i.e. which has very less error distance with global minima)

ANS C

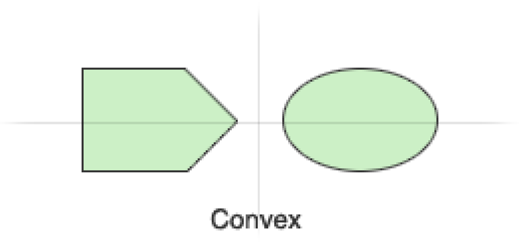
Q11 to Q15 are subjective answer type question. Answer them briefly.

11. What are convex, non-convex optimization?

Convex optimization problems are far more general than linear programming problems, but they share the desirable properties of LP problems: They can be solved quickly and reliably up to very large size -- hundreds of thousands of variables and constraints. The issue has been that, unless your objective and constraints were linear, it was difficult to determine whether or not they were convex. But Frontline System's Premium Solver Platform products includes an automated test for convexity of your problem functions.

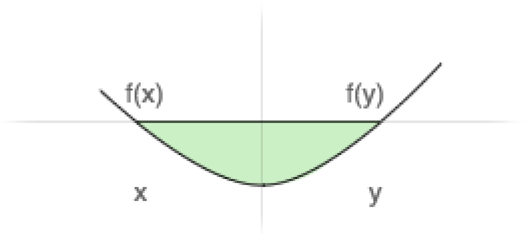
Convex Optimization Problems

A convex optimization problem is a problem where all of the constraints are convex functions, and the objective is a convex function if minimizing, or a concave function if maximizing. Linear functions are convex, so linear programming problems are convex problems. Conic optimization problems -- the natural extension of linear programming problems -- are also convex problems. In a convex optimization problem, the feasible region -- the intersection of convex constraint functions -- is a convex region, as pictured below.



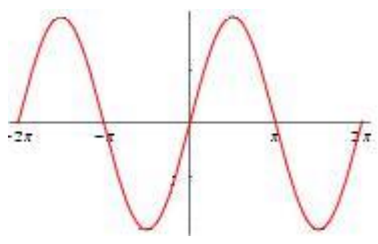
Convex Functions

Geometrically, a function is convex if a line segment drawn from any point (x, f(x)) to another point (y, f(y)) -- called the chord from x to y -- lies on or above the graph of f, as in the picture below:



Algebraically, f is convex if, for any x and y, and any t between 0 and 1, f(tx + (1-t)y) <= t f(x) + (1-t) f(y). A function is concave if -f is convex -- i.e. if the chord from x to y lies on or below the graph of f. It is easy to see that every linear function -- whose graph is a straight line -- is both convex and concave.

A non-convex function "curves up and down" -- it is neither convex nor concave. A familiar example is the sine function:

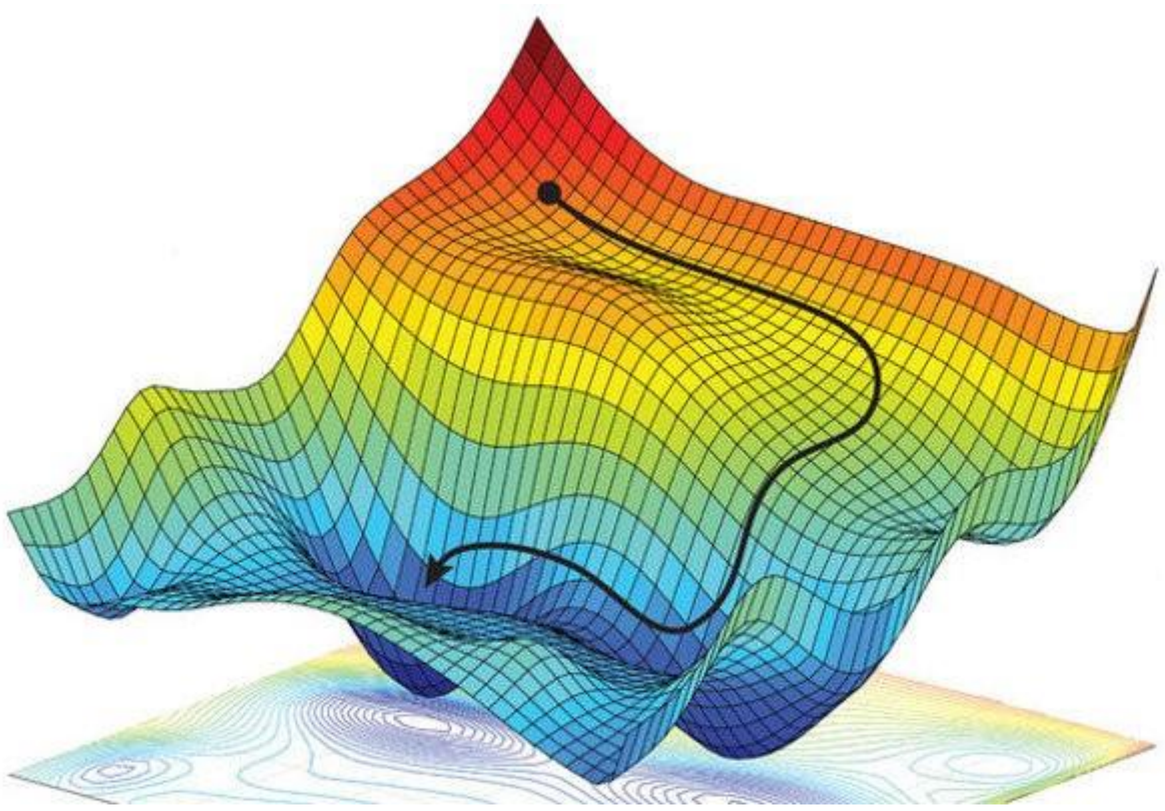


but note that this function is convex from -pi to 0, and concave from 0 to +pi. If the bounds on the variables restrict the domain of the objective and constraints to a region where the functions are convex, then the overall problem is convex.

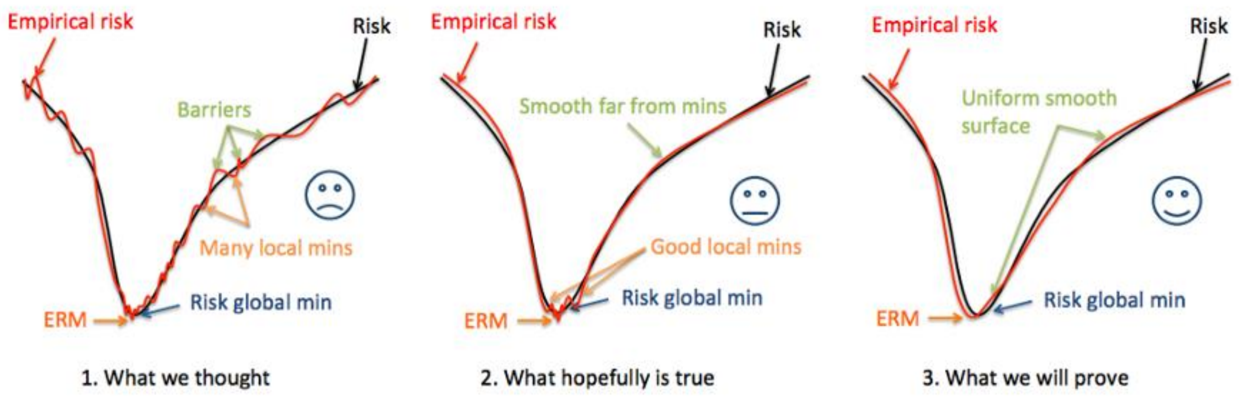
Solving Convex Optimization Problems
Because of their desirable properties, convex optimization problems can be solved with a variety of methods. But Interior Point or Barrier methods are especially appropriate for convex problems, because they treat linear, quadratic, conic, and smooth nonlinear functions in essentially the same way -- they create and use a smooth convex nonlinear barrier function for the constraints, even for LP problems.

These methods make it practical to solve convex problems up to very large size, and they are especially effective on second order (quadratic and SOCP) problems, where the Hessians of the problem functions are constant. Both theoretical results and practical experience show that Interior Point methods require a relatively small number of iterations (typically less than 50) t o reach an optimal solution, independent of the number of variables and constraints (though the computational effort per iteration rises with the number of variables and constraints). Interior Point methods have also benefited, more than other methods, from hardware advances -- instruction caching, pipelining, and other changes in processor architecture.

Non-Convex Optimization
A NCO is any problem where the objective or any of the constraints are non-convex. Even simple looking problems with as few as ten variables can be extremely challenging, while problems with a few hundreds of variables can be intractable. Here, generally, we face the obligation to compromise, i.e. give up seeking the optimal solution, which minimizes the objective over all feasible points. This compromise opens a door for local optimization where intensive research has been conducted and many developments were achieved. As a result, local optimization methods are widely used in applications where there is value in finding a good point, if not the very best. In an engineering design application, for example, local optimization can be used to improve the performance of a design originally obtained by manual, or other, design methods.

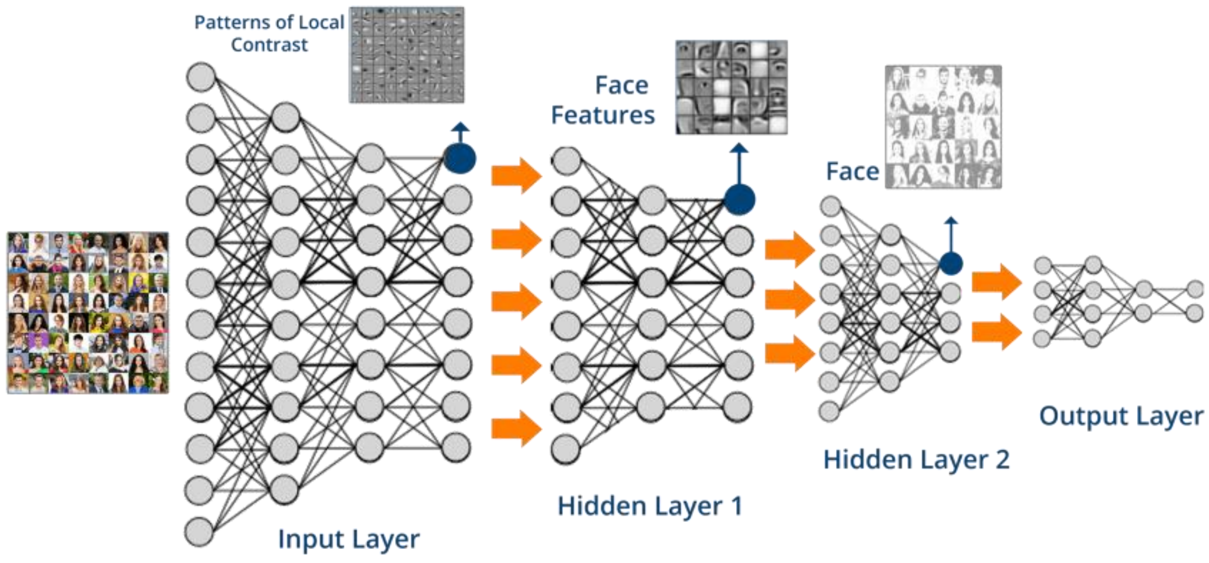


What makes non-convex optimization hard?
Such optimization problems may have multiple feasible and very flat regions, a widely varying curvature, several saddle points, and multiple local minima within each region. It can then take time exponential in the number of variables and constraints to determine that a non-convex problem is infeasible, that the objective function is unbounded, or that an optimal solution is the global optimum across all feasible regions. The optimization may also require an initial guess, which is critical and can greatly affect the objective value of the local solution obtained. Furthermore, little information is provided about how far from the global optimum the local solution is. Beyond this, local optimization methods are often sensitive to algorithm parameter values, which may need to be adjusted for a particular problem, or family of problems. To sum it up, roughly speaking, local optimization methods, used in non-convex optimization, are more art than technology. In contrast, except some cases on the boundary of what is currently possible, there is little art involved in solving a least-squares problem or a linear program. Consequently, there cannot be a general algorithm to solve efficiently NCO problems in all cases and scenarios. NCP is then at least NP-hard. Weak theoretical guarantees add to the downsides of this type of optimization. However, it has the upside of an endless set of problems that we can try to solve better while learning more theoretical insights and improve the practical implementations.



Why is it needed then?

NCO existed since the early days of operations research. Still, the topic gained more interest and focus with the emergence of DL in the recent years. In fact, neural networks (NN)are universal function approximators. With enough neurons, they have the ability to approximate any function well. This the beauty of such networks. Among the functions to be approximated, non-convex functions are gaining more focus. To approximate them, convex functions cannot be good enough. Hence, the importance of using NCO. The freedom to express the learning problem as a non-convex optimization problem gives immense modeling power to the algorithm designer. Since then, the issue of convex optimization became very important to the ML community and consequently some scientific meetings have been devoted solely to this issue, such as the NIPS 2015 Workshop on Non-convex Optimization for Machine Learning: Theory and Practice. Despite the guarantees achieved in individual instances, it is still complex to unify a framework of what makes NCO easy to control. On the practical side, conversations between theorists and practitioners can support the identification of what kind of conditions are rational for specific applications, and thus lead to the design of practically motivated algorithms for non-convex optimization with rigorous guarantees.



12. What do you mean by saddle point? Answer briefly.

Let us consider the optimization problem in low dimensions vs high dimensions. In low dimensions, it is true that there exists lots of local minima. However in high dimensions, local minima are not really the critical points that are the most prevalent in points of interest. When we optimize neural networks or any high dimensional function, for most of the trajectory we optimize, the critical points(the points where the derivative is zero or close to zero) are saddle points. Saddle points, unlike local minima, are easily escapable.

The intuition with the saddle point, is that, for a minima located close to the global minima, all directions should be climbing upward; going further downward is not possible. Local minima exist, but are very close to global minima in terms of objective functions, and theoretical results suggest that some large functions have their probability concentrated between the index (the critical points) and the objective function. The index is the fraction of directions moving downward; for all values of index not 0 or 1 (local minima and maxima, respectively), then it is a saddle point.

Boney goes on to say that there has been empirical validation corroborating this relationship between index and objective function, and that, while there is no proof the results apply to neural network optimization, some evidence suggests that the observed behavior may well correspond to the theoretical results. Stochastic gradient descent, in practice, almost always escapes from surfaces that are not local minima.

This all suggests that local minima may not, in fact, be an issue because of saddle points.

Boney follows his saddle points discussion up by pointing out a few other priors that work with deep distributed representations; human learning, semi-supervised learning, and multi-task learning. He then lists a few related papers on saddle points.

13 What is the main difference between classical momentum and Nesterov momentum? Explain briefly.

The Nesterov Accelerated Gradient method consists of a gradient descent step, followed by something that looks a lot like a momentum term, but isn't exactly the same as that found in classical momentum. I'll call it a "momentum stage" here. It's important to note that the parameters being minimized by NAG are given the symbol y by Sutskever, not θ. You'll see that θ is the symbol for the parameters after they've been updated by the gradient descent stage, but before the momentum stage.
Here's the gradient descent stage:
$\theta_t=y_t-\varepsilon_t\nabla f(y_t)$
And here's the momentum stage:
$y_{t+1}=\theta_t+\mu_t(\theta_t-\theta_{t-1})$
That concludes one iteration of NAG. The hard part is actually finding the correct learning rate and momentum value in order to get the convergence guarantees that make the method attractive, but that needn't concern us. Sutskever in his thesis suggests manual tuning to get an optimal result (at least for deep learning applications) so we are in heuristic territory here anyway.

The classical momentum velocity vector is defined by Bengio as:

$v_t=\mu_{t-1}v_{t-1}-\varepsilon_{t-1}\nabla f(\theta_{t-1})$

Sutskever gives the same definition but without any t subscript on the velocity vector or the momentum coefficient.

We can write out the full update for classical momentum as:

$\theta_{t+1}=\theta_t+\mu_t v_t-\varepsilon_t\nabla f(\theta_t)$
with velocity:

$v_{t+1}=\mu_t v_t-\varepsilon_t\nabla f(\theta_t)$

14 What is Pre initialisation of weights? Explain briefly.?

I'd like to invite you to join me on an exploration through different approaches to initializing layer weights in neural networks. Step-by-step, through various short experiments and thought exercises, we'll discover why adequate weight initialization is so important in training deep neural nets. Along the way we'll cover various approaches that researchers have proposed over the years, and finally drill down on what works best for the contemporary network architectures that you're most likely to be working with.
The examples to follow come from my own re-implementation of a set of notebooks that Jeremy Howard covered in the latest version of fast.ai's Deep Learning Part II course, currently being held this spring, 2019, at USF's Data Institute.
Why Initialize Weights.

The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a

forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.
Matrix multiplication is the essential math operation of a neural network. In deep neural nets with several layers, one forward pass simply entails performing consecutive matrix multiplications at each layer, between that layer's inputs and weight matrix. The product of this multiplication at one layer becomes the inputs of the subsequent layer, and so on and so forth.
For a quick-and-dirty example that illustrates this, let's pretend that we have a vector x that contains some network inputs. It's standard practice when training neural networks to ensure that our inputs' values are scaled such that they fall inside such a normal distribution with a mean of 0 and a standard deviation of 1.
Image for post

How can we find the sweet spot?

Remember that as mentioned above, the math required to complete a forward pass through a neural network entails nothing more than a succession of matrix multiplications. If we have an output y that is the product of a matrix multiplication between our input vector x and weight matrix a, each element i in y is defined as
Image for post
where i is a given row-index of weight matrix a, k is both a given column-index in weight matrix a and element-index in input vector x, and n is the range or total number of elements in x. This can also be defined in Python as:
y[i] = sum([c*d for c,d in zip(a[i], x)])
We can demonstrate that at a given layer, the matrix product of our inputs x and weight matrix a that we initialized from a standard normal distribution will, on average, have a standard deviation very close to the square root of the number of input connections, which in our example is √512.
Image for post

15 What is internal covariance shift in Neural Networks?

We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training.
In neural networks, the output of the first layer feeds into the second layer, the output of the second layer feeds into the third, and so on. When the parameters of a layer change, so does the distribution of inputs to subsequent layers.

These shifts in input distributions can be problematic for neural networks, especially deep neural networks that could have a large number of layers.
a neural network composed of several layers, the forward propagation can be abstracted as function composition (function composition): each layer can indeed be viewed as a function that takes as input a vector from $latex \mathbb{R}^n$ and outputs a vector from $latex \mathbb{R}^m$ (for a layer of $latex m$ neurons preceded by a layer of $latex n$ neurons). For each function, there is a linear mapping from $latex \mathbb{R}^n$ to $latex \mathbb{R}^m$, given a matrix $latex w$ of weights of dimension $latex n, m$, followed by the addition of the biases $latex b$ ($latex b \in\mathbb{R}^m$) and finally a non-linear activation (*e.g* the *sigmoid* function).

$latex \sigma (x) = \frac{1}{1 + e^{-x}}$$latex \sigma^{\prime}(x) = (1 – \sigma(x))\sigma (x)$