CS 287, Fall 2015 Problem Set Backpropagation, RMSProp, and Guided Policy Search

Deliverable: pdf write-up by Wednesday October 7th, 11:59pm, submitted through Gradescope. **Your pdf write-up should be typeset as follows: answers to Question 1(a), 1(b), 1(c), and 1(d) an page 1; answer to Question 2 on pages 2 and 3; answer to Question 3 on page 4; answer to Question 4 on page 5. Make sure to follow the requested typesetting, and insert blank regions (or blank pages) as necessary. Thanks!**

Refer to the class webpage for the homework policy.

Various starter files are provided on the course website.

1. Backpropagation

In this question we will explore the backpropagation algorithm. We begin with a short tutorial on the algorithm and its relation to neural networks. For more detail there are a host of online resources available by searching for neural networks.

A neural network is a mathematical function that is constructed by composing simple mathematical functions such as matrix multiplication, addition, and simple nonlinearities. Typically, this function inputs a vector of some dimension and outputs a vector of some other (possibly the same) dimension and is parameterized by variables in the function, such as the matrices to multiply by. This is a general and flexible class of functions when there are many parameters and several layers of composition. The collection of operations in a neural network forms a computation graph where each node is an input, intermediate, or output variable and the edges into a node are the inputs to that particular computation.

In order to utilize neural networks, we wish to optimize the parameters of some fixed neural network structure in order to minimize some scalar loss function on the output of the neural network that measures performance of the neural network on some dataset on some task, such as classification or regression. Assuming the loss function is differentiable, the standard approaches to optimization will utilize the gradient. A typical neural network could contain anywhere from hundreds to billions of parameters and so calculating the gradient through finite differences, where the number of function evaluations required to compute a single gradient evaluation scales linearly with the number of parameters, quickly becomes intractable.

Fortunately, since neural networks are a composition of simple functions, and our loss function outputs a scalar, we can utilize the chain rule of calculus in order to calculate the gradient of the loss efficiently. We wish to compute $\frac{\partial L}{\partial \theta_i}$ for each θ_i parameter in the network and L the loss function. We calculate $\frac{\partial L}{\partial v_i}$ analytically for each intermediate variable v_i in the computation graph in reverse topological order using dynamic programming. That is, we calculate that quantity for the each variable that depends on a given variable before that variable itself.

To begin with, $\frac{\partial L}{\partial L} = 1$. Assume that we want to calculate $\frac{\partial L}{\partial v_i}$ for a variable v_i and have already calculated $\frac{\partial L}{\partial v_j}$ for each v_j that has an edge from v_i to v_j in the computation graph that represents the neural network (v_i's children). Then we can utilize the chain rule of calculus to calculate:

$$\frac{\partial L}{\partial v_i} = \sum_{v_j} \frac{\partial L}{\partial v_j}\frac{\partial v_j}{\partial v_i}$$

where the sum is over the children v_j of v_i. The first quantity we have already computed, and so we only need to calculate $\frac{\partial v_j}{\partial v_i}$. That quantity is simply the derivative of one intermediate calculation in the network with respect to that calculation's inputs. We restrict ourselves to the composition of simple functions, where we know the derivatives in closed form, and so we can calculate that efficiently. This process proceeds until we have calculated $\frac{\partial L}{\partial \theta_i}$ for each parameter θ_i and then we have the gradient we desire.

Asymptotically, this allows us to compute the gradient with respect to every parameter (even every variable in the graph) with the same computational complexity as one function evaluation. This is incredible since the network could have a billion of parameters and evaluating the approximate gradient would cost two billion function evaluations using finite differences but here we get the exact gradient for no more than the cost of 5 function evaluations (the constant varies slightly based off of which functions are used but most are less than 5). The tradeoff is that the calculation of $\frac{\partial v_j}{\partial v_i}$ often requires the value of v_i and v_j's other inputs, so we have to record the intermediate variables in memory where we could otherwise discard them after using them.

Question For this question you will familiarize yourself with the backpropagation algorithm by performing it on a simple fixed neural network for regression. The inputs are vectors x and y, and parameters are the matrix A and vector b. There are intermediate variables $v = Ax + b$, v_p =rect(v) where the rect function acts elementwise on the input: $\text{rect}(v) = \max(v, 0)$ and keeps the positive part of v. The output of the network would be v_p and we want to optimize the network such that v_p when the network is applied to x is close to the input y (a regression task). Thus we calculate the error= $v_p - y$ and want to minimize the loss: sqerror= $\frac{1}{2}||error||_2^2$. For this problem you will look in `q1_starter.m` for the main file and will calculate the derivatives for the fixed neural network in `simple_backprop_net.m`. Note that the code uses the variable naming convention: $\frac{\partial L}{\partial variable} = variable\_bar$. You should implement backpropagation to calculate $\frac{\partial L}{\partial v_i}$ for each variable v_i in the network. The correct values for a given input are provided to debug and you can utillize finite differences to check your work (but that approach wouldn't scale as the network got larger). Report the values calculated for the `variable_bar_test` variables.

2. RMSProp

In this question you get to implement a variant of gradient descent called RMSProp. This algorithm is a descent update rule that takes a step towards a rescaled gradient. The scaling is done separately for each dimension of the gradient by keeping a running average of recent gradient components squared and dividing each component of the gradient by the square root of those averages, hence Root Mean Square (RMSProp). The RMSprop update rule is as follows. Let t be the iteration and $\nabla f^t(x^t) \in \mathbb{R}^n$ be the gradient of some function f at the current point x^t and $\Delta^t \in \mathbb{R}^n$ be the rmsprop descent vector. Let $v_{ms}^t \in \mathbb{R}^n$ be the mean square vector and $\epsilon \in (0,1)$. Then:

$$v_{ms}^t = (1-\epsilon)(\nabla f^t)^2 + \epsilon v_{ms}^{t-1}$$

for the elementwise square of ∇f^t. Then, for some small $\tau \sim 1e{-8}$ to prevent numerical overflow,

$$\Delta^t = \frac{\nabla f^t}{\sqrt{v_{ms}^t} + \tau}$$

where the square root is elementwise and the division is elementwise between ∇f^t and $\sqrt{v^t_{ms}}$. Then the RMSProp update rule is: $x^{t+1} = x^t - \alpha\Delta^t$ for some global learning rate $\alpha \in \mathbb{R}_+$.

You will be using a limited but simple neural net library for this and question 3. The neural network library is limited to feedforward networks, those that have one vector input at the bottom and all functions are applied on to the previous function's output. Implement the RMSProp update rule in `rmsprop_update.m` and use `q2_starter.m` to verify its correctness. Report the value of the RMSProp update for `x_1_test`.

3. Guided Policy Search (lite)

In this question you get to train a neural network policy for helicopters. We have provided a dataset of 3000 simulated helicopter trajectories that were generated from the following process:

The helicopter starts at some initial position. Generate a nearby goal position and orientation relative to the current position and orientation. A tentative trajectory is generated that interpolates between the current and goal positions with extra zero velocity padding around the ends that has a proportional duration to the magnitude of the distance between the start and goal positions. The SCP-based trajectory optimization from problem set 3 is used to attempt to stay close to the tentative trajectory while satisfying dynamics. Then the generated trajectory is followed by utilizing the LQR based trajectory following. Then the position origin is recentered around the ending position and the end state is used as the beginning state for the next trajectory. A new goal is generated and the above process is repeated. If the trajectory optimization fails to find a path to the goal, a new goal is generated. If 3 failures happen from the same state, the state is reset to the initial state.

This process was chosen to generate a wide range of (current state, current goal, control action) triplets. You will utilize the neural network library and the above dataset as supervision in order to train a neural network that inputs current state and current goal and outputs control action to try to mimic what the trajectory optimizer would do in that situation. The result is a neural network control policy that can be used to guide the helicopter to nearby positions for very little computational cost.

Normally, in guided policy search, there is an alternation between trajectory optimization and supervised neural net training that repeats for several iterations. We omit the alternation here for simplicity. Usually we need this alternation since the neural network fails to reduce the error between its control output and the trajectory optimization's control output to zero. This error means that during execution, the neural network's trajectory will diverge from what the trajectory optimization's trajectory would be and the neural network wouldn't necessarily be able to recover unless there was extra training data for how to recover from that state that the neural net has driven the system to. We generate a large diversity of training data from many initial states to many goals, however, so that hopefully the neural net stays within the regime of states that we have training data for and errors don't compound. Note also that the formulation presented here where the neural network is trying to minimize mean squared distance from its control output to the trajectory optimization's control output is different from the guided policy search presentation in class since the form of trajectory optimization utilized in problem set 3 is deterministic instead of probabilistic.

Experiment with different neural network architectures using the provided neural network library and different values of stepsizes and minibatch sizes and epsilons for rmsprop to train the neural network. You can visualize the resulting trajectories. Report the plot showing learning performance over iteration count and the two histograms showing position error and orientation error magnitudes after executing many trajectories.

See `test_traj_opt.m` and look for `TODO` and `YOUR_CODE_HERE` for the parts you need to write. Note this questions relies on your solution to Question 2, and you'll need to make sure that your code for Question 2 is in the matlab path. Four sets of obstacles are provided: `obstacles{0,1,2,3,4}.mat`. The optimizer should successfully find solutions for 0, 1, and 2, and it will fail on 3 and 4 by getting stuck in a bad local optimum. As your solution write-up, just provide the paths found for each test case by saving the last figure from running `test_traj_opt.m`.