
Assignment 2: Automatic differentiation

Goal: Work out how backpropagation and local gradients work for computations involving vectors and matrices. Understand and extend a mini library that implements automatic differentiation.

Submission: The assignment consists of two parts: implementation and an analysis. You are supposed to present results for both parts in the following manner:

1. Prepare a report with an analysis of results obtained at home.
2. Upload your code.

The code and the report must be uploaded before the deadline to Canvas.
UPLOAD A **SINGLE FILE** (a zip file) containing your code and the report. Name your file as follows: [vunetid]_[assignment number].zip

---

Changelog
11 Nov. Fixed a bug in the backward traversal of vugrad, and used logsoftmax (instead of plain softmax) to stabilize learning with the ReLU. Previously, the probabilities would get too close to zero, which caused NaN in the loss computation.
17 Nov. Clarified questions 3 and 10.
18 Nov. Clarified opening to part 1.
19 Nov. Changed initialization of Linear in vugrad to proper Glorot init. Both versions should work fine with synth and MNIST, but if you want the real Glorot initialization, you should pull the most recent code.

Introduction

In the last assignment, we created a fully connected network, with a logistic regression output to do basic classification. We derived backpropagation for this network on paper, and in terms of the individual scalar units.

Most of this assignment deals with answering direct questions. **Question 9** asks for a quick investigation, which you can use to structure your report, but we don't expect this to take up more than 1 extra page beyond the answers to the questions. If you do some bonus work, your report may turn out longer.

Preliminaries:

- Make sure you have a reasonable understanding of python concepts like magic functions, operator overloading, classes, class functions and static functions. If any of these are new to you, take a minute to look into them before proceeding with the exercise.
- Make sure you have a working knowledge of NumPy. In particular the concepts of broadcasting and indexing. Follow this notebook to brush up.

Part 1: Working out backward functions

One of the most important elements of building a deep learning autodiff system is working out what the backward function is for a particular forward.

The slides define this in detail and give a basic recipe for working out a backward function. Here are the important parts:

- The backward does *not* compute the local derivative (the gradient of the input with respect to the output), it computes the gradient of the loss with respect to the inputs, given the gradient of the loss with respect to the output.
    - That is, if f is a function $\mathrm{f} : \mathbf{A} \mapsto \mathbf{B}$ then its backward is the function $\mathrm{f} : \mathbf{B}^{\nabla} \mapsto \mathbf{A}^{\nabla}$ with $\mathbf{B}^{\nabla} = \nabla_{\mathbf{B}} \mathrm{loss}$ and likewise for $\mathbf{A}^{\nabla}$. Here, loss represents the single scalar output of the computation graph, downstream of f. This is always the value for which we compute the gradients.
- Draw a computation graph that contains the inputs of the function, the outputs of the function and the rest of the computation graph (from the outputs to the loss)
- Start with a scalar derivative of the loss over one element of the input.
- Apply the multivariate chain rule and sum over all elements of the output. Work out the derivative in terms of the inputs and the gradient of the output.
- Work out what tensor operations are needed to compute the whole gradient at once.

If you get stuck on any of these, try moving on to the second part. The code contains answers to many of these (but not the derivation), so you can use those as a hint.

Question 1: Let f(X, Y) = X / Y for two matrices X and Y (where the division is element-wise). Derive the backward for X and for Y. Show the derivation.

Question 2: Let f be a scalar-to-scalar function $\mathrm{f} : \mathbb{R} \to \mathbb{R}$. Let F(X) be a tensor-to-tensor function that applies f element-wise (For a concrete example think of the sigmoid function from the lectures). Show that whatever f is, the backward of F is the element-wise application of f' applied to the elements of X, multiplied (element-wise) by the gradient of the loss with respect to the outputs.

Question 3: Let matrix **W** be the weights of an MLP layer with f input nodes and m output nodes, with no bias and no nonlinearity, and let **X** be an n-by-f batch of n inputs with f

features each. Which matrix operation computes the layer outputs? Work out the backward for this operation, providing gradients for both **W** and **X**.

Question 4: Let f(**x**) = **Y** be a function that takes a vector **x**, and returns the matrix **Y** consisting of 16 columns that are all equal to **x**. Work out the backward of f. (This may seem like a contrived example, but it's actually an instance of broadcasting).

Part 2: Backpropagation

Clone the following repository

https://github.com/dlvu/vugrad

and run it from the command line with:

```
python experiments/train_mlp.py
```

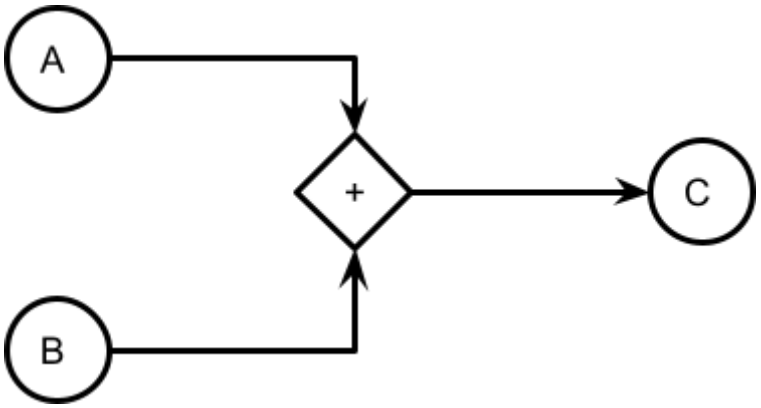
It trains a two-layer MLP on a simple synthetic dataset. Run it as

```
python experiments/train_mlp.py -D mnist -l 0.0001
```

to train it on the digit data from the last assignment (this takes a bit longer).

If the commands above don't work, or your IDE doesn't understand references to the vugrad package, go to the root of the repository (where the `setup.py` *file is located) and run "*`pip install -e .`*". Then try again.*

Most of this assignment is in understanding the code, and how it builds up a computation graph. Remember that a computation graph is a bipartite graph consisting of tensor nodes and operation nodes.



This is the computation graph for the operation C <- A + B, with the tensor nodes as circles and the operation node as a diamond.

Question 5: Open an ipython session or a jupyter notebook in the same directory as the README.md file, and import the library with `import vugrad as vg`. Also do `import numpy as np`.

Create a TensorNode with

```
x = vg.TensorNode(np.random.randn(2, 2))
```

Recreate the computation graph above: create two tensor nodes `a` and `b` containing numpy arrays of the same size, and sum them (using the + operator) to create a TensorNode `c`. Answer the following questions (in words, tell us what these class members mean, don't just copy/paste their values).
1) What does `c.value` contain?
2) What does `c.source` refer to?
3) What does `c.source.inputs[0]`.value refer to?
4) What does `a.grad` refer to? What is its current value?

If you can't answer these questions, dig into the codebase, and come back to them later. If you watch video 4 of lecture 2 attentively, you should be able to make an educated guess, which you can then confirm by looking into the code.

Question 6: You will find the implementation of `TensorNode` and `OpNode` in the file `vugrad/core.py`. Read the code and answer the following questions
1) An `OpNode` is defined by its inputs, its outputs and the specific *operation* it represents (i.e. summation, multiplication). What kind of object defines this operation?
2) In the computation graph of question 5, we ultimately added one numpy array to another (albeit wrapped in a lot of other code). In which line of code is the actual addition performed?
3) When an `OpNode` is created, its inputs are immediately set, together with a reference to the op that is being computed. The pointer to the output node(s) is left None at first. Why is this? In which line is the `OpNode` connected to the output nodes?

Question 7: When we have a complete computation graph, resulting in a `TensorNode` called loss, containing a single scalar value, we start backpropagation by calling

```
loss.backward()
```

Ultimately, this leads to the `backward()` functions of the relevant `Ops` being called, which do the actual computation. In which line of the code does this happen?

NOTE: Don't just give the line number (there may be small changes to the code), but show the line in context in your report.

Question 8: `core.py` contains the three main `Ops`, with some more provided in ops.py. Choose one of the ops Normalize, Expand, Select, Squeeze or Unsqueeze, and show that the implementation is correct. That is, for the given forward, derive the backward, and show that it matches what is implemented.

Question 9: The current network uses a Sigmoid nonlinearity on the hidden layer. Create an Op for a ReLU nonlinearity (details in the last part of the lecture). Retrain the network. Compare the validation accuracy of the Sigmoid and the ReLU versions.

NOTE: you don't have to change any of the code in the package. You can create your own experiment script, or adapt `train_mlp.py` and create any `Ops` and `Modules` you need in that file.

Question 10*: Change the network architecture (and other aspects of the model) and show how the training behavior changes. Here are some ideas (but feel free to try something else).

- Try adding more layers to the MLP, or widening the network (more nodes in the hidden layer).
- Add a momentum term to the gradient descent. This is discussed in lecture 4.
- Try adding a residual connection between layers. These are also discussed in lecture 4.
- It is often said that good initialization is the key to neural network performance. What happens if you replace the Glorot initialization (used in the `Linear` module) by something else? If you initialize to all 0s or sample from a standard normal distribution, do you see a drop in performance?

Part 3: Pytorch

For the final part of this exercise, we want you to get acquainted with the basics of pytorch. Start by following the installation instructions and the 60-minute blitz tutorial.

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

The things you've learned in the first part of this exercise will hopefully make sense of what you encounter in this tutorial. To map what we've built to pytorch, note the following:

- Pytorch doesn't build on numpy. They have their own tensor object, which works in a very similar way. In fact the tensor data is guaranteed to be stored in memory in the same way, so that you can create a pytorch tensor from a numpy tensor backed by the same data in memory (whether this is a good idea is another question).
- Pytorch used to have a class that does what our `TensorNode` does, called `Variable`. This is now deprecated. Instead the Tensor class is used for everything. During backpropagation, a grad and source field are simply added to the Tensor directly.
- We've computed and remembered gradients for every node. This is a bit wasteful, since many gradients are never needed at all, or if they are, can be forgotten when the upstream gradients have been computed. In pytorch, the default is to set gradients to "not required" which means that pytorch computes them only if they are necessary for other nodes, and forgets them when they are no longer needed.
- Ops are called Functions in pytorch. They are implemented pretty much as they are in our example. One difference is that the context variable is a tuple in pytorch, and not a dict. In most situations, you won't need to dig this deep into the pytorch API.
- In pytorch, you almost never call a Function directly. The Functions are usually wrapped in Modules or in a utility function in torch.nn.functional.

- In pytorch, all familiar array operations are implemented and are differentiable. This includes things like broadcasting and slicing. For instance if you have some matrix X (with gradients required) and you do Y = X[:, None, 3:5], you are creating nodes in the computation graph, and you will get gradients over the selected parts of X.

Question 11:
Install pytorch using the installation instructions on its main page. Follow the Pytorch 60-minute blitz. When you've built a classifier, play around with the hyperparameters (learning rate, batch size, nr of epochs, etc) and see what the best accuracies are that you can achieve. **Report your hyperparameters and your results.**

Question 12*:
Change some other aspects of the training and report the results. For instance, the package `torch.optim` contains other optimizers than SGD which you could try. There are also other loss functions. You could even look into some tricks for improving the network architecture, like batch normalization or residual connections. We haven't discussed these in the lectures yet, but there are plenty of resources available online. Don't worry too much about getting a positive result, just report what you tried and what you found.

Grading (10pt)

- Question 1: 1pt
- Question 2: 1pt
- Question 3: 1pt
- Question 4: 1pt
- Question 5: 1pt
- Question 6: 1pt
- Question 7: 0.5pt
- Question 8: 0.5pt
- Question 9: 1pt
- Question 10: 1pt
- Question 11: 0.5pt
- Question 12: 0.5pt