HW2: Nonlinear Classifiers

CSCI 662: Fall 2023

out: Sep 25, 2023
due: Oct 13, 2023

This assignment is about using the infrastructure you built in HW1, now applying a non-linear classifier (specifically, a one-layer feed-forward network) to the problem. The key is that we want you to implement the model by hand, without using neural network or machine learning frameworks. Here is a nice post on why such exercise of implementing your own forward and backward passes is useful. After you do this, you should compare your implementation to a PyTorch implementation (that you should also write) and to the results you got in HW1.

WARNING: While some of this writeup will look similar to the HW1 write up, there are important differences throughout so it is a good idea to *read thoroughly*.

WARNING: Expect to spend way more time on this assignment than you did on assignment 1. It has a lot more room for mistakes and bugs. This is, of course, not to scare you; it's a fun assignment, and we are here to help. It's so you plan accordingly.

Code To Write

1. Write a trainer for a feed-forward neural network, using python 3. The trainer should be called `train.py`. It should display its invocation and brief help when invoked as `python3 train.py -h`. You should use the argparse package for ease here. The program should take the following options:

   - `-u <integer>` to specify the number of hidden units.
   - `-l <float>` to specify the learning rate.
   - `-f <integer>` to specify the number of words to read per data item, i.e., the max sequence length of each input.
   - `-b <integer>` to specify the minibatch size.
   - `-e <integer>` to specify the number of epochs to train for.
   - `-E <embeddingfile>` to specify the word embedding file to be read.
   - `-i <inputfile>` to specify a training file to be read. Training files will be in the form `<text>TAB<label>`, i.e., a line of text (that does not contain a tab), a tab, and a label.
   - `-o <modelfile>` to specify a model file to be written.

   It can take other options too such as specifying additional hyperparameters, a dev set so that held-out loss can be displayed, etc.

2. Write a classifier that takes in the feed-forward model and unlabeled data set and labels it. The classifier should be called `classify.py`. It should display its invocation and brief help when invoked as `python3 classify.py -h`. The program should take the following options:
   - `-m <modelfile>` to specify the trained model file to read, i.e., the output of `train.py`.
   - `-i <inputfile>` to specify a test file to be read. Test files will be in the form `<text>`, i.e., a line of text that does not contain a tab. For each line, a label should be predicted.
   - `-o <outfile>` to specify an output file to be written. The output file should contain one label for each line in the input file.
3. After you've written your own backprop/feed forward, you should write your own PyTorch implementation of the same network. You should not use any other previously written trainer/classifier code except code that you wrote for HW1. Name these `train-torch.py` and `classify-torch.py`. The options should be at least the same as those used in `train.py` and `classify.py`.

More details

To make things simple, here is the specific architecture you should implement:

- Retrieve 50-dimensional GloVe English vectors, 300-dimensional fastText Odiya (spelled 'oriya' at fasttext – same thing!) vectors or 100-dimensional 'ufvytar' vectors of uncertain origin for each of the maximum f words (as specified by option `-f` above) in the input you will consider to make a $50f$-dim (or $300f$-dim in the case of Odiya) input vector x: find the file `glove.6B.50d.txt`, `fasttext.wiki.300d.vec`, and `ufvytar.100d.txt` available in your workspace. If you need padding words (i.e., your input is too short), use a vector of zeroes. If your word is not found, you can use a mean of all the vectors in the embedding file; as a convenience, these have been added to vocareum as `unk-eng.vec`, `unk-odiya.vec`, and `unk-ufvytar.vec`; append it to the embedding file. The index for this word is "UNK" (all caps). **Note that all 'regular' words in this file are lowercase, so adjust your input data accordingly.** You won't adjust these embeddings while learning.
  This step is essentially an implementation of the `getFeatures` function of the `Features` base class from HW1; this is the featurization we use to feed the input in the network.
- Let w_A and b_A be the input-to-hidden layer ($50f \times u$)-dim (or $300f \times u$ in the case of odiya) weight matrix and u-dim bias vector, respectively. The hidden vector, h, is $relu(x \cdot w_A + b_A)$ where $relu$ is the rectified linear unit, i.e., $relu(x) = x$ when $x > 0$ and $relu(x) = 0$ otherwise.
- Assuming there are d output classes, let w_B and b_B be the hidden-to-output layer ($u \times d$)-dim weight matrix and d-dim bias vector, respectively.
- The loss, L, is $-\sum_{c=1}^{d} \mathbb{1}[y = c] \log(softmax(h \cdot w_B + b_B)_c)$, i.e., the cross entropy of the softmax of the output logits relative to the ground truth class label. Another way to write this is to define the logits, ℓ, as $\ell = h \cdot w_B + b_B$; then $L = -\sum_{c=1}^{d} \mathbb{1}[y = c] \log(softmax(\ell)_c)$
- You don't need to implement regularizers (though you can if you want to; it would be in the extra mile category).
- You've probably noticed that a fixed-length feed-forward approach is not a great way to classify some of this data (e.g. `products`). That's ok, we just want a comparison of your implementation and a pytorch implementation. But if you wanted to explore ways to use a feed forward model, you could try variants of input. For example, the f words you input could be just verbs. Or they could be the most frequently occurring words in the input. Or the terms with the highest $tf * idf$. Or other ideas you come up with.
- Some useful notes on calculating gradients are here (they require concentration to read through): http://www.cs.columbia.edu/~mcollins/cs4705-spring2019/notes/ff2.pdf.

Coding Requirements

- The classifier should be able to handle words that haven't been seen before.
- The trainer/classifier should be multi-class, not binary, and should generate actual class labels. Assume all classes are seen in training and save the label info in the model file.
- In your writeup, discuss different values of hidden layer size, number of words allowed per item, learning rate, and number of epochs you tried and the different scores you got on your own internal test set and on the blind set (on Vocareum). Compare your results to the results you got in HW1. **Note that due to being feed forward, you will probably only use a small subset of the input data in many of the data sets; you can increase the number of words you read in but eventually may run into memory issues.** A different neural architecture (e.g. CNN, RNN) will probably be better (this is outside of the scope of this assignment but reasonable for extra mile, as are other creative approaches to handling input).
- Submit `train.py`, `classify.py`, `train-torch.py`, `torch-classify.py`, any other code needed, and `[questions,torch.questions,products,torch.products,4dim,torch.4dim,odiya,torch.odiya].model` as discussed below.
- Make sure the Vocareum auto-scoring script runs and gives reasonable results. You may also try running alternate models interactively on Vocareum. Some larger models may not run but baselines definitely should! We'd like to try these out so please provide instructions for trying the models you most want to demonstrate by creating a usable `README` file.
- Once you are done with your primary models, compare your implementations to 'off the shelf' implementations. Are there any differences in speed and/or performance? Can you determine what accounts for these differences?

Coding Recommendations

- When writing your code you should seed the random number generator so your results do not change between executions. When you are ready to run for real, don't seed the generator, so the results will be different every time (you can try running your model several times to get an idea of the variance).
- One design you might consider and choose is programming parameters and the nonlinearities of your feed-forward neural network as concrete implementations of a base class with `forward` and `backward` abstract methods. In fact, this is what happens way below under the hood of PyTorch.
- Implementing a neural network by hand is tricky! You should run tests to make sure your model is correct. One test you can run is to only allow one kind of gradient update at a time (e.g. just the bias on the hidden layer). If things are implemented correctly your loss on training should still improve on every epoch if your gradients are correct (it should just do so less well than with the entire, correctly implemented network). Another test you can run is to do gradient checking; this is done by calculating, for one of the functions f you want to check the gradient of, $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$ for several random values of x and a specified small number ϵ. Compare this number to the gradient calculated for those files by your gradient equations. Per calculus theory, your manual gradient should vary from your calculated gradient by no more than ϵ^2.
- For general approaches to training a classifier for multiple tasks and evaluating a classifier see the tips from homework 1.
- If you're not familiar with PyTorch, here is a good place to start and look for pointers: Learning PyTorch with Examples.

Data

We have provided several different classification tasks (this is the same as in HW1):

- `products`: Very variable length lines of various kinds of English product reviews that are either positive (pos) or negative (neg). 32,592 reviews.
- `4dim`: English reviews of variable length that are positive or negative and truthful or deceptive (pos.tru, pos.dec, neg.tru, neg.dec). 1,560 reviews.
- `odiya`: News headlines written in odiya (or Oriya, or Odia) which is a low-resourced language spoken in India. They are classified into business, sports and entertainment. 15,200 headlines.
- `questions`: Questions to be classified into 6 categories: abbreviation, entity, description, human, location, and numeric value. There are 5,452 questions provided. The language of the questions is uncertain; before undertaking the arduous hand-transcription of this data, we noticed a faded label that said "ufvytar" on the library archive box. But there's something mysterious about this data...

Data is on Vocareum; when you submit, if you provide trained model files called `[questions, torch.questions, products, torch.products, 4dim, torch.4dim, odiya, torch.odiya].model`, we will test your code on hidden files and return a score. If this score differs greatly from your expectation you may a) be overfit, or b) have some design flaw in your code structure (e.g. hardcoded assumptions).

Your Report

Your report should at a *minimum*:

- Show the specific equations for the gradient updates you implemented.
- Discuss differences between your from-scratch implementation and the PyTorch implementation.
- Discuss differences in your feed-forward classifier(s) on the different data sets and how the classifiers built in this homework compare to the performance of the classifiers from the last homework on the same data sets.
- Where relevant, show learning rates, discuss overfitting and loss convergence. Use graphs and tables *appropriately*, not superfluously. This means the graphs/tables should emphasize the message you are delivering, not simply be in place without thinking about why you are using that particular medium to convey an idea.

Use the ACL style files : https://github.com/acl-org/acl-style-files

Your report should be at least two pages long, including references, and not more than four pages long, not including references (i.e. you can have up to four pages of text if you need to). Just like a conference paper or journal article it should contain an abstract, introduction, experimental results, and conclusion sections (as well as other sections as deemed necessary). Unlike a conference paper/journal article, a complete related works section is not obligatory (but you may include it if it is relevant to what you do).

Grading

Grading will be roughly broken down as follows:

- about 50% – did you clearly communicate your description of what you implemented, how you implemented it, what your experiments were, and what conclusions you drew from them? This includes appropriate use of graphics and tables where warranted that clearly explain your point. This also includes well written explanations that tell a compelling story. Grammar and syntax are a small part of this (maybe 5% of the grade, so 10% of this section) but much more important is the narrative you tell. Also a part of this is that you clearly acknowledged your sources and influences with appropriate bibliography and, where relevant, cited influencing prior work.

- about 20% – is your code correct? Did you implement what was asked for, and did you do it correctly?
- about 20% – is your code well-written, documented, and robust? Will it run from a different directory than the one you ran it in? Does it rely on hard-codes? Is it commented and structured such that we can read it and understand what you are doing?
- about 10% – did you go the extra mile? Did you push beyond what was asked for in the assignment, trying (well-justified) new models, features, or approaches? Did you use motivation (and document appropriately) from another researcher trying the same problem or from an unrelated but transferrable other paper?

'Extra Mile' ideas

This is not meant to be comprehensive and you do not have to do any of the things here (nor should you do all of them). But an 'extra mile' component is 10% of your grade.

- Add multiple hidden layers to your from-scratch implementation to create a deeper neural network (you can do this for your PyTorch version too but you'll get more credit for doing it on the from-scratch implementation).
- Try different batch sizes to speed up training.
- Consider alternate ways to deal with the varying input lengths—different model architecture, creative and reasonable input approaches, etc. Justify your choices!
- Try different activation functions for the hidden layer and see how they change the performance.
- Implement a dropout layer (again, more credit if done in from-scratch than in PyTorch) and see if it helps reduce overfitting.
- Experiment with different weight initialization strategies and investigate if they help the model during the early phase of training.
- Implement adaptive learning rates and momentum to make the model converge faster.
- Dig into the ACL archives and find ideas for other architectures or approaches; try them out, analyze performance.

+ batchNorm

Rules

- This is an individual assignment. You may not work in teams or collaborate with other students. You must be the sole author of 100% of the code you turn in.
- Depending on need and class interest, we may collaborate *in class* or *publicly on Piazza* if you get stuck; this kind of collaboration is okay.
- You may not look for coded solutions on the web, or use code you find online or anywhere else. You can and are encouraged to read material beyond what you have been given in class (see above) but should not copy code.
- You may not download the data from any source other than the files provided on Vocareum, and you may not attempt to locate the test data on the web or anywhere else.
- For this assignment you may **not** use other *external* data that is not the training/test data provided (you may of course use the vectors we have provided). THIS IS A DIFFERENCE FROM THE PREVIOUS ASSIGNMENT.

- You may use packages in the Python Standard Library (including numpy) for your original implementation. You may not use any other packages (e.g. scikit-learn, keras, tensorflow, or other machine learning libraries). For your comparison implementation you should use PyTorch.
- You may use external resources to learn basic functions of Python (such as reading and writing files, handling text strings, and basic math), but the extraction and computation of model parameters, as well as the use of these parameters for classification, must be your own work.
- Generative language, code, and vision models (e.g. ChatGPT, Llama 2, Midjourney, Github Copilot, etc.; if you are unsure, ask and don't assume!!) can be used (to aid in report writing/coding, not to actually do the classification tasks) with the following caveats:
  - You must declare your use of the tools in your submitted artifact. If you don't declare the tool usage but you did use these tools, we will consider that plagiarism
  - For code and image generation, you must indicate the prompt used and output generated
  - For text generation you must provide either a link to the chat session you used to help write the content or an equivalent readout of the inputs you provided and outputs received from the system. You will lose credit if "the AI" is doing the work rather than you.
- Failure to follow the above rules is considered a violation of academic integrity, and is grounds for failure of the assignment, or in serious cases failure of the course.
- We use plagiarism detection software to identify similarities between student assignments, and between student assignments and known solutions on the web. Any attempt to fool plagiarism detection, for example the modification of code to reduce its similarity to the source, will result in an automatic failing grade for the course.
- If you have questions about what is and isn't allowed, post them to Piazza!