Lab 3: Backprop with Pytorch

Lab 3: Backprop with Pytorch 95/95 Points

Due: Tue Mar 28, 2023 8:30am3/28/2023 95/95 Points

Attempt

Attempt 1

REVIEW FEEDBACK
Submitted on Mar 24, 2023 10:26am3/24/2023

Attempt 1 **95/**
Score:

Add Comment

Unlimited Attempts Allowed

Details

Subtitle: From-scratch deep learning except for backpropagation, which is with Pytorch

Overview

This is an individual lab. In this one-week lab, you will define a multi-layer network by applying PyTorch operations and use PyTorch's autograd to perform stochastic gradient descent. You will train your network on a simple benchmark dataset.

In this lab, you will develop a single-script neural network that uses Pytorch to train on data. This is a parallel development to the classes you started writing last week. **You will NOT continue to develop these classes until a couple weeks from now**. For example, you don't need to use your Layer class in this week's lab.

At the same time, you **should NOT** use PyTorch's built-in layer classes for this lab. For example, do not use anything from the torch.nn subpackage. Instead, develop your own pytorch code for each activiation function and layer you use.

While it can be effective to use helper methods in this lab, I strongly recommend using a Jupyter notebook and evaluating each piece of code outside of a helper method before wrapping it in a helper method for later use. This gives you both the iterative benefits of a Jupyter Notebook and modularity.

Your final training loop should NOT be in a helper method. Then, if anything "crashes", the variables of this loop should still be available for you to inspect.

This form of interactive code writing can be highly effective!

You will implement all of the core parts of a deep learning algorithm using PyTorch:

- Forward propagation
- Backward propagation
- Loss and regularization
- Stochastic gradient descent

Your network will have the following structure:

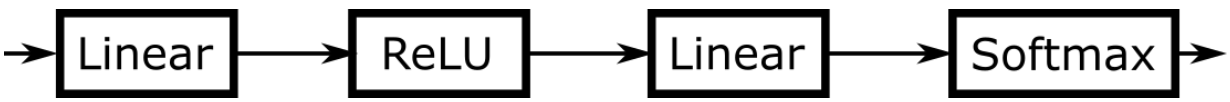


Figure: Simple Network

(Though this week's implementation will not include a Softmax layer.)

Your implementation will include regularization (weight decay) and a cross-entropy loss function during training. Including all of the trainable parameters as additional inputs, the structure of your network will be:

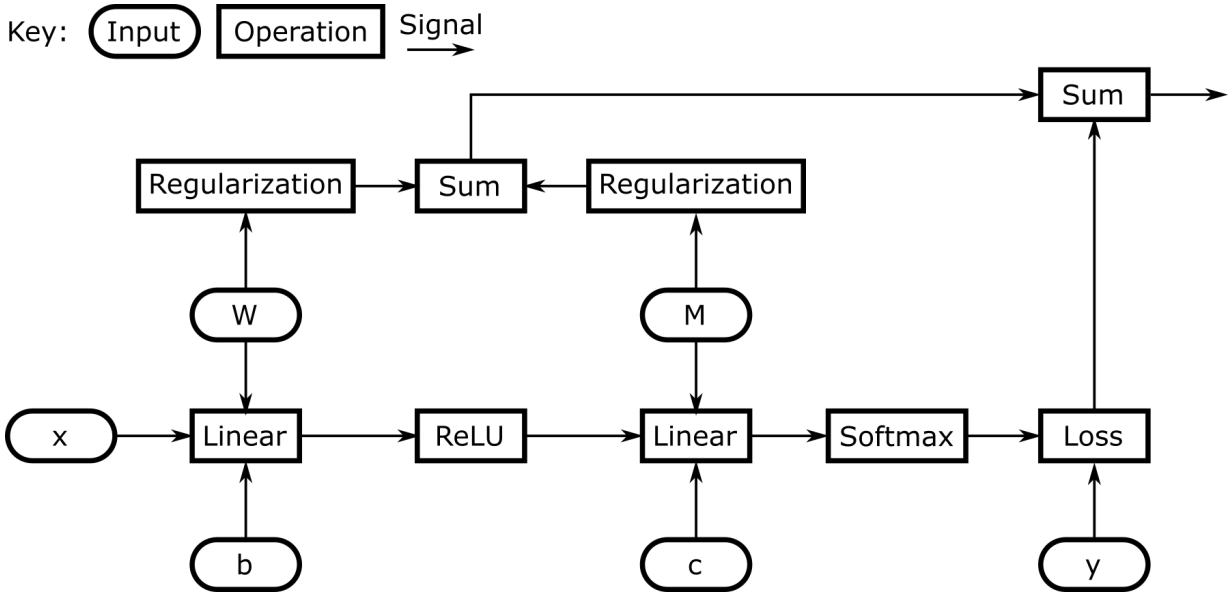


Figure: Same network showing all operations and input variables

In the diagram above, the ovals represent numbers, vectors, and matrices (tensors) fed into the network. Other than the true input x and the true output y, they are all trainable parameters – tensors whose values should be changed during training to optimize the objective function J.

The rectangles represent operations:

- Linear – this is a fully-connected layer without any activation function
- ReLU – this is the ReLU activation function, applied element-wise to each element of the input
- Softmax – this is the softmax activation function, normalizing all the elements within the layer to sum to 1
- Loss – this is cross-entropy or MSE loss. Each loss should have its own class. In this week's lab, you will use only the MSE loss.
- Regularization – this is the squared Frobenius norm of its matrix input. It is identical to the sqaured L2 norm for a vector: The sum of the squares of all the elements
- Sum – this is simply the sum of all the elements in the vector or matrix input

We will use PyTorch autodiff functionality to perform back propagation:

- Mark all trainable parameters as requiring gradients.
- Simply compute the forward operations as in numpy, but using the corresponding PyTorch operations. As they are executed in the code, PyTorch will automatically construct the backward graph.
- Call the backward() function to propagate the gradients throughout the network.

Because you are using PyTorch, you will be able to take advantage of some of the GPU parallelization techniques built into PyTorch.

Assignment

You will implement a network that supports forward-propagation and derive the equations for forward propagation.

Write the forward-propagation equations

Write your derivation report **by hand** on blank, lined, grid, or "digital" paper. In the top-right corner, write your name, the date, the lab assignment number and name, and the page number.

Write the equations for the network, using the variable names defined in the input boxes (rounded rectangles) and on the signal lines in this figure:

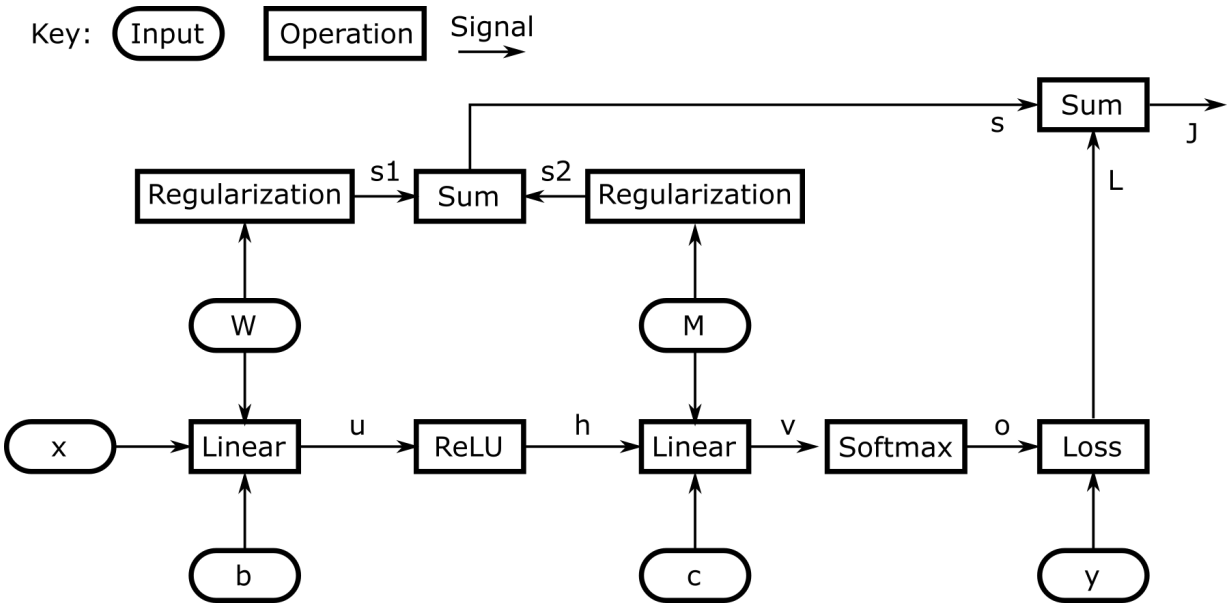


Figure: Computation graph with variables on signal paths

See our textbook, **Section 4.7 – Forward Propagation, Backward Propagation, and Computation Graphs (https://www.d2l.ai/chapter_multilayer-perceptrons/backprop.html)** for an example done on a simpler network. As in the textbook, use vector and matrix variables rather than formulas that use weight or neuron indices to keep your formulas simple. You are welcome to use indices or other sketches as you work out what the matrix math should be, but box your final simplified equations to make your final results clear. It is OK to include crossed-out work on your page. Please neatly cross it out by lining it through at most twice or drawing a large X through a portion of the text you don't want to keep. Please do **not** rewrite your work to make it cleaner unless you also include all of your sketch work as well.

Indicate the shape of all variables. One way to do this is by writing out equations in element form. Another is by simply specifying the dimensions.

In defining these layers, you don't need to match your equations exactly to what your instructors may have done. You may have different constant multipliers, or a different data orientation, for example.

These equations should NOT be in a polished lab format, and you MUST work them by hand.

Selecting Initial Metaparameters

Metaparameters (also known as hyperparameters) are the numbers that describe the structure of a network. These don't change during backpropagation.

You will have to select several metaparameters while implementing your network:

- How many hidden layers will there be? Starting with the one hidden layer shown in the diagrams above is a good choice. But once this is working, you may wish to add more layers to see if this can improve performance.
- How many nodes are in each layer? Remember that the total number of nodes in each layer should, in general, gradually decrease as you work your way through the network.
- What learning rate will you use? (Values of 0.1, 0.01, or 0.001 are often used for networks of the size we are training in this lab.)
- How much regularization will you use? Start with 0 regularization, then once the network is training pretty well, add a little in – perhaps 0.01 or 0.001 or even 1e-5 might be useful values. Regularization sometimes improves performance a few percent, but it won't make a broken network start training, so stick with 0 until the network is achieving a reasonable starting accuracy.

Choose an initial value for each metaparameter. Feel free to adjust these as you debug your basic code trying to get it to train at least partially. Discuss your intuition for choosing metaparameter with your peers, but try not to just give them your full metaparameter set. Exploring the metaparameter space is a key part of this lab sequence and you don't want to eliminate your peers' opportunity for learning here!

This week's dataset

This week, you will train your network on one of the simpler datasets. For this dataset, you will need to replace the softmax and cross-entropy loss layers with an L2 loss. For these networks, you will only need a few hidden nodes – for example, `create_linear_data` works well with three hidden nodes.

Test cases

Create a few very-low dimensional test cases for the simplified architecture used in this week's lab. Include the regularization term, and use MSE loss in the place of softmax and cross-entropy loss. In Dr. Yoder's Spring 2022 offering, variations on the network used on Quiz 1 are a good starting point.

A network with two inputs, three nodes in the hidden layer, and two outputs is likely sufficient for all your tests. Create two to five tests, using different inputs or weights to test the network under a variety of conditions. Manually compute the output of each network to test your feedforward implementation against.

Work these examples, however sketchy, by hand on scratch or engineering paper. **These predictions should NOT be in a polished lab format, and you MUST work them by hand.**

Computing Forward

Copy the starting code to a directory of your choice:

```
cp /data/cs3450/mlp_from_scratch/week1/client.py .
```

We strongly encourage you to save your code to a private repository and to make frequent backups through git commits.

Create a Python implementation to train and test your multi-layer perceptron (MLP) as follows:

1, Launch the OOD app *Jupyter Notebook - Containerized* using the container *CS3450 Spring 2022*. Open the notebook mlp_with_autograd.ipynb.

2. Translate your forward-propagation equations into PyTorch code. You may implement these to support just one sample at this point. Set the inputs, W's, and b's to the simple values you prepare in the previous section. Check that the output matches your predictions.

Include your notes on these checks, however sketchy, in your lab final submission. These should **not** be polished into a report format.

For the in-lab checkoff in Dr. Yoder's section, your in-lab predictions must match what your network produces for at least one network.

Backpropagation with Autograd

Train your network on one of the simpler datasets. For these datasets, you will need to replace the softmax and cross-entropy loss layers with an MSE loss. For these networks, you will only need a few hidden nodes – for example, `create_linear_data` works well with three hidden nodes.

1. Initialize the W's to random values. Either normally-distributed or uniformly distributed values are fine. Multiplying these values by 0.1 is often important, especially when there is more than one hidden layer. Initialize the b values to zero.
2. Wrap your code in a training loop:
    1. Provide a sample on each time through the loop, computing the loss from the network's output compared against the the output sample and from the regularization term (which should be set to zero in your early tests).
    2. Perform backward propagation using `.backward()` on the objective function.
    3. Update the learnable parameters from the weights and the learning rate. Because the learnable parameters have `requires_grad=True`, you have to set their internal data, e.g. `param.data -= alpha * param.grad`
    4. Zero the gradients: `param.grad.zero_()` where param is the learnable parameter whose gradients you are zeroing. Repeat for all learnable parameters.
3. Run your network for a few steps. At the end of each step, the loss on the current sample should decrease. If it does not, check your equations!
4. Run your network for a few epochs. Your loss should become very close to zero. When training with the data from `create_linear_training_data()`, `W*M` should be the same matrix as $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$, if you use the column-ordered data as we use in the

inclass derivations. In this case, the network should learn to keep the ReLUs always active because their are no nonlinearities and the network SHOULD reduce to a single linear operation!

5. Once your network is learning OK with one sample, update all your operations to support batches of samples. This will be important to support faster training.

Tuning

Continue tuning your network on the linear dataset until `W*M` matches the matrix $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$, up to rounding error. If you use a different matrix notation than we use in class, describe that notation next to your `W*M` (or equivalent) and describe why you believe this is the correct value for these layers to learn.

Deliverables

A complete submission consists of:

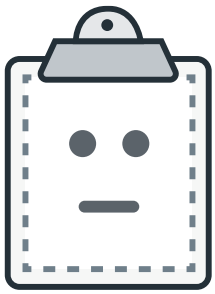
- Your pages of forward propagation derivations, including dimensions for each variable.
- Your ipynb notebook, with the main training loop training on the linear dataset.
- The ipynb notebook should end with the value of `W*M` for your trained network and a discussion of how closely the network learned these values.

You will submit your ipynb notebook through esubmit. Submit your derivations in hard copy in class on the due-date.

Just for fun

You may wish to clone this dataset and add a littl numeric noise.

Then, does regularization improve the training?



Preview Unavailable

pytorch_autograd.ipynb

Download

(https://msoe.instructure.com/files/1993192/download?download_frd=1&verifier=aeTOuAHQb3j3EuZEh0MlXHM5sEpFUi2ikNeAmTC3)

Previous Module Previous

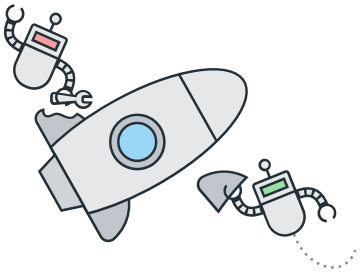
Try Again

Next ModuleNext

(https://msoe.instructure.com/courses/13814/modules/items/547984)

(https://msoe.instructure.com/courses/13814/modules/items/547973)

Sorry, Something Broke



Help us improve by telling us what happened

Report Issue