Homework 1 Bonus

ADAM, Dropout, and BatchNorm

11-785: Introduction to Deep Learning (Fall 2020)

Out: **Sept 28th 2020 12:00:00am EST**
Due: **TBA**

(Last updated: 9/28/20 12:00AM EST)

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.

- **Overview:**
  - **Adam** (5 points): Implementing the Adam optimizer
  - **Dropout** (10 points): Implementing dropout regularization
  - **BatchNorm1d** (10 points): Implementing BatchNorm (this time for real!)

- **Directions:**
  - Implement the above. After finishing hw1p1, we're hoping this will feel relatively straightforward and not too difficult.
  - **Come to office hours or post on Piazza if you get stuck. We'll consider making a hw1p1 bonus FAQ if it becomes popular enough.**
  - You are required to do this assignment using Python3. Other than for testing, do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
  - Note that Autolab uses numpy v1.18.1

Introduction

In this assignment, you'll be implementing **Adam** and **Dropout**. These will be a little simplified from the actual torch implementation, but the spirit will be the same.

IMPORTANT: First, copy the highlighted files/folders from the HW1 Bonus handout over to the corresponding folders that you used in hw1.

Yes, the batchnorm file should be identical to the previous one; if you already have it you don't need to copy it over again.

NOTE: We recommend you make a backup of your hw1 files before copying everything over, just in case you break code or want to revert back to an earlier version.

Bonus Structure

(Displaying only relevant files for this homework)

```
handout
├── autograder
│   └── hw1_bonus_autograder ..... Copy over this entire folder
│       ├── runner.py
│       ├── test_adam.py
│       ├── test_batchnorm.py
│       └── test_dropout.py
├── hw1_bonus ..... Copy over this entire folder
│   └── stubs.py ..... Contains stub for Dropout(Function) that you must copy into functional.py
├── mytorch
│   ├── nn
│   │   ├── batchnorm.py ..... Should be identical to the one in HW1P1
│   │   ├── dropout.py ..... Dropout module (to be called by user)
│   │   └── functional.py ..... Dropout will need a Function here
│   └── optim
│       └── adam.py ..... Adam
├── sandbox.py ..... To test your function, like in hw1p1
├── create_tarball.sh
└── grade.sh
```

Next, copy and paste the Dropout(Function) stub from hw1_bonus/stubs.py into the nn/functional.py.

0.1 Running/Submitting Code

This section covers how to test code locally and how to create the final submission.

0.1.1 Running Local Autograder

Run the command below to calculate scores for both problems.

```
./grade.sh 1b
```

If this doesn't work, converting line-endings may help:

```
sudo apt install dos2unix
dos2unix grade.sh
./grade.sh 1b
```

If all else fails, you can run the autograder manually with this:

```
python3 ./autograder/hw1_bonus_autograder/runner.py
```

0.1.2 Running the Sandbox

We've provided `sandbox.py`: a script to test and easily debug basic operations and autograd. When you add your own new operators, write your own tests for these operations in the sandbox.

Note: We will not provide new sandbox methods for this homework. You are required to write your own from now onwards.

```
python3 sandbox.py
```

0.1.3 Submitting to Autolab

Note: You can submit to Autolab even if you're not finished yet. You should do this early and often, as it guarantees you a minimum grade and helps avoid last-minute problems with Autolab.

Run this script to gather the needed files into a `handin.tar` file:

```
./create_tarball.sh
```

If this doesn't work, converting line-endings may help:

```
sudo apt install dos2unix
dos2unix create_tarball.sh
./create_tarball.sh
```

You can now upload `handin.tar` to Autolab.

1 Adam [Total: 5 points]

Adam (Kingma and Ba 2015) is a per-parameter adaptive optimizer that considers both the first and second moment of the current gradient. In practice, Adam is highly popular in the field, and is used for a variety of network architectures. It's considered to generate good training results relatively quicker than other optimizers.

Torch implements its own re-ordered version of Adam[1]. This implementation is a little less intuitive but it runs faster than the traditional one. Either way, translating the math below will be enough to score full points.

Remember: none of this should add to the computational graph.

Like all optimizers in Torch, Adam receives a `params` variable that is given to it during initialization. This variable contains the output of running `.parameters()` on any subclass of `Module`.

For each parameter tensor in `params`, Adam maintains two tensors that track the **running estimate of the mean gradient for each parameter** m_t (est. first moment) and the **running estimate of the mean squared gradient for each parameter** v_t (est. second moment).

For each time step t, Adam iterates through its list of parameters. For each parameter p, Adam retrieves the corresponding state for this parameter and updates it using the formulas below.

First, because the estimate tensors were initialized to 0's, Adam calculates one bias correction term for each of them:

$$B_1 = (1 - \beta_1^t)$$
$$B_2 = (1 - \beta_2^t)$$

Next, it updates the moment estimation tensors using the gradients in the module weights:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \nabla_t^2$$

It then calculates the 'step size' by adjusting the learning rate:

$$s_t = \frac{\eta}{B_1}$$

And finally updates the parameters:

$$D = \frac{\sqrt{v_t}}{\sqrt{B_2}} + \epsilon$$

$$\theta_t = \theta_{t-1} - \frac{m_t}{D} \cdot s_t$$

In `mytorch/optim/adam.py`, implement `Adam.step()` using the above description.

[1] This alternate formulation that we are implementing is unfortunately mentioned only briefly in the Adam paper in the last paragraph of section 2 ('Algorithm').

2 Dropout [Total: 10 points]

Dropout (Srivastava et al. 2014) is a regularization method where neurons are randomly "turned off" during training to improve network robustness. In theory, this approximates ensemble learning of multiple networks. One way to conceptualize this is that it encourages neurons to avoid becoming overly reliant on certain inputs.

In practice, this is generally done by applying a `Dropout` module after some layer. This module then selects neuron outputs to set to 0 based on a random parameter p. For example, if $p = 0.3$, each neuron output has a 30% chance of being zeroed[2].

Note that the randomly outputs change every forward pass.

After zeroing out the values, the other values are all **rescaled** by a factor of $\frac{1}{1-p}$ to dampen the impact of sharply reducing some values to 0.

As a final note, this only occurs if a model is in 'training mode'. If the model is in evaluation OR if the probability of dropout is 0, dropout does nothing and simply returns the input as it was given. This will also affect backprop correspondingly (we won't tell you how, think about it).

We encourage you to read the original paper if anything is unclear. It is very clearly and simply written. Keep in mind, p in their notation is $(1 - p)$ in our notation.

We've already completed the user-facing module for you in `nn/dropout.py`. Now, in `nn/functional.py`, you will need to create and complete a `Dropout(Function)` class. Remember that this would link dropout to the comp graph.

2.1 `Dropout(Function).forward()` [5 points]

In `nn/functional.py`, complete the `Dropout(Function).forward()` method. Note that you should have already copied this over, as per the instructions on the Introduction page.

Everything you need to implement this was described above.

2.2 `Dropout(Function).backward()` [5 points]

In `nn/functional.py`, complete the `Dropout(Function).forward()` method.

Try to understand what backprop for this would look like, given the forward. We won't tell you how; this is up to you (this is a bonus assignment, after all).

Implementation Tips:

1. The code is around 10 lines of straightforward code. Don't overthink it; dropout does exactly what it sounds like it does.
2. Use `np.random.binomial()` to generate a binary 'mask' tensor that zeros out values when multiplied to another tensor.
3. In this implementation, you should make a new tensor for the output.
4. Think about what you'd need to save in `ctx`.

[2]Technically, in the original notation it's the opposite; if $p = 0.3$, there is a 70% the output is dropped. In other words, the p we are using is technically $1 - p$ in the original paper

3 Batch Normalization (BatchNorm) [Total: 10 points]

Note that this problem was on HW1P1, but has now been converted to a bonus. Read the Piazza announcement on this change first before posting new questions about our grading policy for this.

In `nn/batchnorm.py`, complete the `BatchNorm1d` class.

BatchNorm (Ioffe and Szegedy 2015) uses the following equations for its forward function:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i \tag{1}$$

$$s_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (\mathbf{x}_i - \mu_{\mathcal{B}})^2 \tag{2}$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{s_{\mathcal{B}}^2 + \epsilon}} \tag{3}$$

$$\mathbf{y}_i = \gamma_i \hat{\mathbf{x}}_i + \beta_i \tag{4}$$

Note that you will want to convert these equations to matrix operations.
x_i is the input to the BatchNorm layer. Within the forward method, compute the sample mean ($\mu_{\mathcal{B}}$), sample variance ($s_{\mathcal{B}}^2$), and norm ($\hat{\mathbf{x}}_i$). Epsilon (ϵ) is used when calculating the norm to avoid dividing by zero. Lastly, return the final output (y_i).

You may need to implement operation(s) like `Sum` in `nn/functional.py` and `tensor.py`. Remember to use matrix operations instead of `for` loops; loops are too slow.

Also, you'll need to calculate the **running mean/variance too**:

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m-1}\sum_{i=1}^{m} (\mathbf{x}_i - \mu_{\mathcal{B}})^2 \tag{5}$$

$$E[x] = (1-\alpha) * E[x] + \alpha * \mu_{\mathcal{B}} \tag{6}$$

$$Var[x] = (1-\alpha) * Var[x] + \alpha * \sigma_{\mathcal{B}}^2 \tag{7}$$

Note: The notation above is consistent with PyTorch's implementation of Batchnorm. The α above is actually $1-\alpha$ in the original paper.

BatchNorm operates differently during training and eval. During training (`BatchNorm1d.is_train==True`), your forward method should calculate an unbiased estimate of the variance ($\sigma_{\mathcal{B}}^2$), and maintain a running average of the mean and variance. These running averages should be used during inference (`BatchNorm1d.is_train==False`) in place of $\mu_{\mathcal{B}}$ and $s_{\mathcal{B}}^2$.

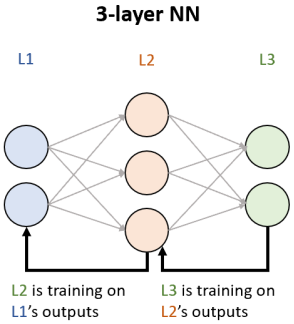
Appendix

A BatchNorm

Batch Normalization ("BatchNorm") is a wildly successful technique for improving the speed and quality of learning in NNs. It does this by attempting to address an issue called **internal covariate shift**. We encourage you to read the original paper ; it's written very clearly, and walks you through the math and reasoning behind each decision.

A.1 Motivation: Internal Covariate Shift

Internal covariate shift happens while training an NN.



In an MLP, each layer is training based on the activations of previous layer(s). But remember - **previous layers are ALSO training, changing their weights and outputs all the time**. This means that the later layers are working with frequently shifting information, leading to less stable and significantly slower training. This is especially true at the beginning of training, when parameters are changing a lot. That's internal covariate shift. It's like if your boss AND your boss's boss joined the company on the same day that you did. And they also have the same level of experience that you do. Now you'll never get into FAANG...

A.2 Intro to BatchNorm

BatchNorm essentially introduces normalization/whitening *between* layers to help mitigate this problem. Specifically, a BN layer aims to linearly transform the output of the previous layer s.t. across the entire dataset, each neuron's output has `mean=0` and `variance=1` AND is linearly decorrelated with the other neurons' outputs.

By 'linearly decorrelated', we mean that for a layer l with m units, individual unit activities $\mathbf{x} = \{\mathbf{x}^{(k)}, \ldots, \mathbf{x}^{(d)}\}$ are independent of each other – $\{\mathbf{x}^{(1)} \perp \ldots \mathbf{x}^{(k)} \ldots \perp \mathbf{x}^{(d)}\}$. Note that we consider the unit activities to be random variables.

In short, we want to make sure that normalization/whitening for a single neuron's output is happening consistently across the entire dataset. In truth, this is not computationally feasible (you'd have to feed in the entire dataset at once), nor is it always fully differentiable. So instead, we maintain "running estimates" of the dataset's mean/variance and update them as we see more observations.

How do we do this? Remember that we're training on batches[3] - small groups of observations usually sized 16, 32, 64, etc. **Since each batch contains a random subsample of the dataset, we assume that each batch is somewhat representative of the entire dataset**. Based on this assumption, we can use their means and variances to update our running estimates.

[3]Technically, *mini-batches*. "Batch" actually refers to the entire dataset. But colloquially and even in many papers, "batch" means "mini-batch".

A.3 Implementing BatchNorm

This section gives more detail/explanation about the implementation of BatchNorm. Read it if you need any clarifications.

Given this setup, consider $\mu_{\mathcal{B}}$ to be the mean and $\sigma^2_{\mathcal{B}}$ the variance of a unit's activity over the batch $\mathcal{B}$[4]. For a training set $\mathcal{X}$ with n examples, we partition it into n/m batches $\mathcal{B}$ of size m. For an arbitrary unit k, we compute the batch statistics $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ and normalize as follows:

$$u^{(k)}_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}^{(k)}_i \tag{8}$$

$$(\sigma^2_{\mathcal{B}})^{(k)} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \left(\mathbf{x}^{(k)}_i - \mu^{(k)}_{\mathcal{B}}\right)^2 \tag{9}$$

$$\hat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{10}$$

Note that we add $\epsilon = 1e-5$ to $\sigma^2_{\mathcal{B}}$ in order to avoid dividing by zero.

A significant issue posed by simply normalizing individual unit activity across batches is that it limits the set of possible network representations. In order to avoid this, we introduce a set of trainable parameters ($\gamma^{(k)}$ and $\beta^{(k)}$) that learn to make the BatchNorm transformation into an identity transformation.

To do this, these per-unit learnable parameters $\gamma^{(k)}$ and $\beta^{(k)}$ rescale and reshift the normalized unit activity. Thus the output of the BatchNorm transformation for a data example, $\mathbf{y}_i$ is:

$$\mathbf{y}_i \leftarrow \gamma\hat{\mathbf{x}}_i + \beta$$

Training Statistics

$$E[x] = (1-\alpha) * E[x] + \alpha * \mu_{\mathcal{B}} \tag{11}$$

$$Var[x] = (1-\alpha) * Var[x] + \alpha * \sigma^2_{\mathcal{B}} \tag{12}$$

Note: The notation above is consistent with PyTorch's implementation of Batchnorm. The α above is actually $1-\alpha$ in the original paper.

This is the running mean $E[x]$ and running variance $Var[x]$ we talked about. We need to calculate them during training time when we have access to training data, so that we can use them to estimate the true mean and variance across the entire dataset.

If you didn't do this and recalculated running means/variances during test time, you'd end up wiping the data out (mean will be itself, var will be inf) because you're typically only shown one example at a time during test. This is why we use the running mean and variance.

[4] Again, technically 'mini-batch'

References

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167. arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.