Homework 1 Bonus

Adam, AdamW, LayerNorm, and Dropout

11-785: Introduction to Deep Learning (Spring 2026)
Carnegie Mellon University

Release Date: **January 14, 2026, 06:00 P.M., E.S.T.**
Final Due Date: **April 24, 2026, 11:59 P.M., E.S.T.**

1 Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to help your friends debug
  - You are allowed to look at your friends code
  - You are allowed to copy math equations from any source that are not in code form
  - You are not allowed to type code for your friend
  - You are not allowed to look at your friends code while typing your solution
  - You are not allowed to copy and paste solutions off the internet
  - You are not allowed to import pre-built or pre-trained models
  - 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.

  We encourage you to meet regularly with your study group to discuss and work on the homework. You will not only learn more, you will also be more efficient that way. However, as noted above, the actual code used to obtain the final submission must be entirely your own.

- **Directions:**
  - You are required to do this assignment in the Python (version 3) programming language. 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.
  - We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.
- **Overview:**
  - Adam
  - AdamW
  - Layer Normalization
  - Dropout

Homework objective

After this homework, you would ideally have learned:

- How to write code to implement different optimizers from scratch
  - How to implement Adam
  - How to implement AdamW
- How to write code to implement layer normalization from scratch
  - How to implement Forward Propagation with Layer Normalization
  - How to implement Back Propagation with Layer Normalization
- How to write code to implement dropout from scratch
  - How to implement Forward Propagation with Dropout
  - How to implement Back Propagation with Dropout

Contents

2 MyTorch

The culmination of all of the Homework Part 1's will be your own custom deep learning library, which we are calling MyTorch. It will act similar to other deep learning libraries like PyTorch or Tensorflow.

The files in your homework are structured in such a way that you can easily import and reuse modules of code for your subsequent homeworks.

3 Setup and Submission

- **Extract** the handout *hw1p1_bonus.tar* by running the following command in the same directory[1]

  ```
  tar -xvf hw1p1_bonus_handout.tar
  ```

  This will create a directory called `HW1P1_bonus` with the following file structure: [2]

```
HW1P1_bonus
├── mytorch
│   ├── models
│   │   └── mlp.py (Copy your file from HW1P1)
│   ├── nn
│   │   ├── activation.py (Copy your file from HW1P1)
│   │   ├── batchnorm.py (Copy your file from HW1P1)
│   │   ├── linear.py (Copy your file from HW1P1)
│   │   ├── loss.py (Copy your file from HW1P1)
│   │   ├── dropout.py
│   │   └── layernorm.py
│   └── optim
│       ├── adam.py
│       ├── adamW.py
│       └── sgd.py (Copy your file from HW1P1)
└── hw1p1_bonus_autograder.py
```

- **Install** Activate the same conda environment created for HW1P1:

  ```
  conda activate idls26
  ```

- **Autograde** your code by executing the following in terminal from top level directory:

  ```
  python3 hw1p1_bonus_autograder.py
  ```

- **Hand-in** your code by running the following command from the top level directory, then **SUBMIT** the created *handin.tar* file to autolab:

  ```
  tar -cvf handin.tar mytorch
  ```

  The handout might have an extension like **handout.tar.112**. In such a case, you will have to first rename downloaded file as **handout.tar** by removing the **.112** extension and then untar the file.

[1] The handout might have an extension like handout.tar.112. In such a case, you will have to first rename downloaded file as handout.tar by removing the .112 extension and then untar the file.

[2] For using code from Homework 1, ensure that you received all 100 autograded points.

- **DO**
  - We strongly recommend that you review the ADAM, Layer Normalization, and Dropout papers.
- **DO NOT**
  - Import any other external libraries other than `numpy` in your submission, as extra packages that do not exist in autolab will cause submission failures.
  - Move or remove any files or change any file names.

4 ADAM [5 points]

Adam is a per-parameter adaptive optimizer that considers both the first and second moment of the current gradient. Implement the `adam` class in `mytorch/optim/adam.py`.

At any time step t, Adam keeps a running estimate of the mean deriviative for each parameter m_t and the mean squared derivative for each parameter v_t. t is initialized as 0 and m_t, v_t are initialized as 0 tensors. They are updated via:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

where g_t are the current gradients for the parameters. β_1 and β are hyper-parameters which control the exponential decay rates of these moving averages, their default values are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

We call m_t and v_t as **biased** moments, because they are initialized as 0 tensors. As such, they are biased towards 0, especially in earlier steps. As such, Adam corrects this with:

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$

$$\hat{v}_t = v_t / (1 - \beta_2^t)$$

Lastly, the parameters are updated via

$$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t + \epsilon})$$

where α is the learning rate. Recall that we intuitively divide $\hat{m}_t$ by $\sqrt{\hat{v}_t}$ in the last step to normalize out the magnitude of the running average gradient and to incorporate the second moment estimate.

For more detailed explanations, we recommend that you reference the original paper.

You need to keep a running estimate for some parameters related to W and b of the linear layer. We have defined instance variables inside our implementation of the Linear class that you can use.

5 AdamW [5 points]

AdamW is an optimizer which uses weight decay regularization with Adam. Implement the `adamW` class in `mytorch/optim/adamW.py`.

If you implemented Adam, then the only additional parameter in AdamW is the weight decay. In this, we reduce the network parameter a portion of the model parameter at each time step.

$$W_t = W_t - W_{t-1} * \alpha * \lambda$$
$$b_t = b_t - b_{t-1} * \alpha * \lambda$$

Where W_t and b_t are your parameters after the standard Adam update in iteration t, λ is weight decay and α is the learning rate. Alternatively, you can *first* perform the following two weight decay updates:

$$W_t = W_{t-1} - W_{t-1} * \alpha * \lambda$$
$$b_t = b_{t-1} - b_{t-1} * \alpha * \lambda$$

and then, add the standard Adam update to W_t and b_t obtained above using W_{t-1} and b_{t-1}.
(**NOTE:** Observe the subscripts indicating the iteration number in the two presented ways).

For more detailed explanations, we recommend that you reference the original paper

6 Layer Normalization [mytorch.nn.layernorm] [10 points]

Layer Normalization is a method commonly used in Recurrent Neural Networks (RNNs) and Transformers. Unlike Batch Normalization, which normalizes across the batch dimension, Layer Normalization normalizes across the feature dimension for each independent sample. It comes from the paper Layer Normalization

In this section, you will implement the "LayerNorm" class in "mytorch/nn/layernorm.py".

Table 1: Layer Normalization Components

Code Name	**Math**	**Shape**	**Meaning**
N	N	scalar	Batch size
num_features	C	scalar	Number of features
Z	Z	$N \times C$	Data input to the LN layer
M	μ	$N \times 1$	Per-sample mean (note the shape difference vs. BN)
V	σ^2	$N \times 1$	Per-sample variance
NZ	$\hat{Z}$	$N \times C$	Normalized input data
gamma	γ	$1 \times C$	Scaling parameters
beta	β	$1 \times C$	Shifting parameters

6.1 Layer Normalization Forward Equations

Unlike Batch Normalization, which calculates statistics over the batch dimension N, Layer Normalization calculates statistics over the feature dimension C for every single sample in the batch independently.

First, calculate the mean μ and variance σ^2 for each sample i in the batch. Note that the summation is performed over the features $j = 1...C$.

$$\mu_i = \frac{1}{C}\sum_{j=1}^{C} Z_{ij} \quad \in \mathbb{R}^{N\times 1} \tag{1}$$

$$\sigma_i^2 = \frac{1}{C}\sum_{j=1}^{C}(Z_{ij} - \mu_i)^2 \quad \in \mathbb{R}^{N\times 1} \tag{2}$$

Implementation Hint: In `numpy`, taking the mean over an axis reduces that dimension. You will need to ensure μ and σ^2 retain the shape $(N, 1)$ to broadcast correctly against Z $(N \times C)$.

Then, normalize the input Z using the calculated statistics:

$$\hat{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \quad \in \mathbb{R}^{N\times C} \tag{3}$$

Finally, scale and shift the result using the learnable parameters γ and β. Note that while normalization is performed per-sample, the parameters are applied per-feature (shared across the batch).

$$\tilde{Z}_{ij} = \gamma_j \odot \hat{Z}_{ij} + \beta_j \quad \in \mathbb{R}^{N\times C} \tag{4}$$

6.2 Layer Normalization Backward Equations

You are expected to derive or research the backward pass for Layer Normalization. The gradients for the learnable parameters γ and β are similar to Batch Normalization, but simpler, as they accumulate gradients over the batch dimension:

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{N} \frac{\partial L}{\partial \tilde{Z}_i} \odot \hat{Z}_i \quad \in \mathbb{R}^{1\times C} \tag{5}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{N} \frac{\partial L}{\partial \tilde{Z}_i} \quad \in \mathbb{R}^{1\times C} \tag{6}$$

For the gradient with respect to the input $\frac{\partial L}{\partial Z}$, the derivation is distinct because the mean and variance depend on the sample i itself. Refer to the original paper (equation 4) or standard derivations for the full expansion. The final gradient form is:

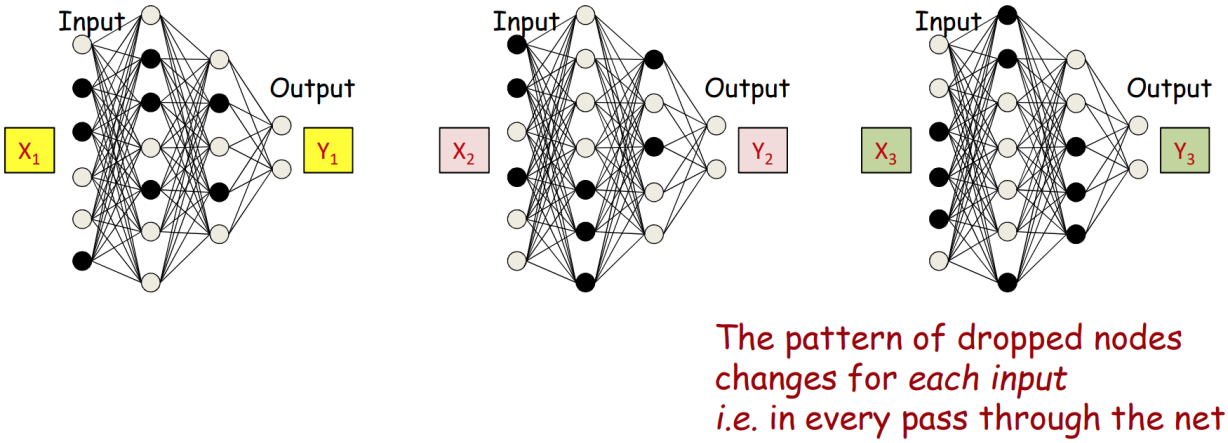
$$\frac{\partial L}{\partial Z_{ij}} = \frac{1}{C\sqrt{\sigma_i^2+\epsilon}} \left(C\frac{\partial L}{\partial \hat{Z}_{ij}} - \sum_{k=1}^{C} \frac{\partial L}{\partial \hat{Z}_{ik}} - \hat{Z}_{ij} \sum_{k=1}^{C} \left(\frac{\partial L}{\partial \hat{Z}_{ik}} \odot \hat{Z}_{ik} \right) \right) \tag{7}$$

Implementation Hint: Pay close attention to the axes of summation in the equation above. The terms $\sum \frac{\partial L}{\partial \hat{Z}}$ and $\sum(\frac{\partial L}{\partial \hat{Z}} \odot \hat{Z})$ are summations over the feature dimension C, resulting in column vectors of shape $(N, 1)$.

7 Dropout [10 points]

Dropout is a regularization method that approximates ensemble learning of networks by randomly "turning off" neurons in a network during training. Implement the `Dropout` class in `mytorch/nn/dropout.py`.

For every input, the neurons that are "turned off" are randomly chosen via the dropout rate p. The probability of zero-ing out a neuron output is then $1 - p$.



We can implement this by generating and applying a binary mask to the output tensor of a layer. As dropout zeros out a portion of the tensor, we need to re-scale the remaining numbers so the total expected "intensity" of the output is same as in testing, where we don't apply dropout.

For more detailed explanations, we recommend that you reference the paper.

Implementation Notes:

- You should use `np.random.binomial`
- You should scale during training and not during testing