EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade Homework 6

This homework is due on Oct 17, at 10:59PM.

1. Memory considerations when using GPUs (Coding Question)

In this homework, you will run GPUMemory.ipynb to train a ResNet model on CIFAR-10 using PyTorch and explore its implications on GPU memory.

We will explore various systems considerations, such as the effect of batch size on memory usage and how different optimizers (SGD, SGD with momentum, Adam) vary in their memory requirements.

It is strongly recommended that you start early on this question, since colab daily GPU limits may require you to complete this question over a few days with breaks in between.

(a) Managing GPU memory for training neural networks (Notebook Section 1).

(i) **How many trainable parameters does ResNet-152 have**? What is the **estimated size of the model in MB**?

Solution: ResNet-152 has 58164298 parameters, and on a colab T4 GPU, it is estimated to be 232MB in size.

(ii) Which GPU are you using? **How much total memory does it have**?

Solution: Colab typically has Nvidia T4 with 15360 MB memory.

(iii) After you load the model into memory, **what is the memory overhead (MB) of the CUDA context loaded with the model**?

Solution: It can be anywhere between 500-1000 MB. On a colab T4 GPU, it is about 582 MB.

(b) Optimizer memory usage (Notebook Section 2).

(i) **What is the total memory utilization during training with SGD, SGD with momentum and Adam optimizers**? Report in MB individually for each optimizer.

Solution: Generally SGD would have lowest memory utilization, followed by SGD with momentum. Adam would have maximum memory usage. Example values from a Colab T4 GPU: SGD: 3192.0 MB, SGD with momentum: 3274.0 MB, ADAM: 3358.0 MB.

(ii) **Which optimizer consumes the most memory**? Why?

Solution: ADAM consumes more memory than SGD or SGD with momentum because it stores two moving averages of the gradients and their squares, which are used to estimate the first and second moments of the gradient. Because these moving averages are updated in each step, and they need to be stored in memory.

(c) Batch size, learning rates and memory utilization (Notebook Section 3)

(i) **What is the memory utilization for different batch sizes (4, 16, 64, 256)? What is the largest batch size you were able to train?**

Solution: Exact answers depend on the GPU used by the student. Example memory utilizations (trends are important, not absolute values): 4: 1406.0 MB, 16: 1706.0 MB, 64: 2738.0 MB, 256: 8744.0 MB. Batch sizes >= 1024 result in an out of memory error on a T4 GPU.

(ii) **Which batch size gave you the highest accuracy at the end of 10 epochs?**

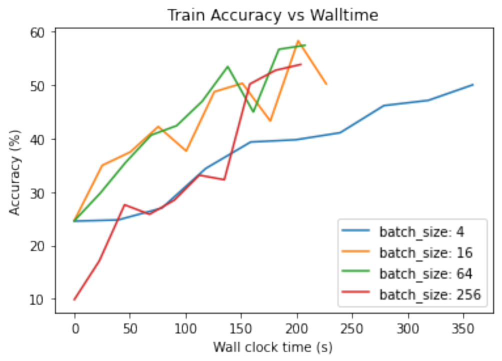
Solution: Answers may vary, but typically would be between 16 and 256.

(iii) **Which batch size completed 10 epochs the fastest (least wall clock time)? Why?**

Solution: The largest batch size run (256 or 512) would have completed 10 epochs fastest because they can take advantage of hardware acceleration and parallelism, and they can use memory more efficiently. Specifically, GPUs are based on the SIMD (Single Instruction, Multiple Data) paradigm. This means that if we have a large batch of data, we can perform the same operation on each piece of data in parallel, increasing the throughput we see.

(iv) Attach your **training accuracy vs wall time plots** with your written submission.

Solution: Exact plots may vary. Example accuracy vs wall time plot:



2. Graph Dynamics and GNN Concepts

In this problem, we will explore connections between graph dynamics and graph neural networks (GNNs).

Diagrams in this question were taken from https://distill.pub/2021/gnn-intro. This blog post is an excellent resource for understanding GNNs and contains interactive diagrams. Some graph neural network methods operate on the full adjacency matrix. Others, such as those discussed in https://distill.pub/2021/gnn-intro/, at each layer apply the same local operation to each node based on inputs from its neighbors.

This problem is designed to:

- Show connections between these methods.
- Show that for a positive integer k, the matrix A^k has an interesting interpretation. That is, the entry in row i and column j gives the number of walks of length k (i.e., a collection of k edges) leading from vertex i to vertex j.

To do this, let's consider a very simple deep linear network, defined as follows:

- Its underlying graph has n vertices, with adjacency matrix A. That is, $A_{i,j} = 1$ if vertices i and j are connected in the graph and 0 otherwise.
- It has n vertices in each layer, corresponding to the n vertices of the underlying graph.
- Each vertex has n channels.
- The input to each node in the 0-th layer is a one-hot encoding of own identity. That is, the node i in the graph has input $(0, \cdots, 0, \underbrace{1}_{i\text{-th entry}}, 0, \cdots, 0)$.

- The weight connecting node i in layer k to node j in layer $k+1$ is $A_{i,j}$.
- At each layer, the operation at each node is simply to sum up the weighted sum of its inputs and to output the resulting n-dim vector to the next layer. You can think of these as being depth-wise operations if you'd like.

(a) **Write the output of the j-th node at layer k in this network in terms of the matrix A.**

(Hint: This output is an n-dimensional vector since there are n output channels at each layer.)

Solution: A_j^k

Explanation: Think of the initial graph as a matrix of length n, where every node is a row. The initial graph is just the identity.

At each timestep, we can compute the next timestep of the graph by taking $G^{t+1} = AG_t$. This works b/c every row j in G^{t+1} is produced by summing the rows of G_t specified by the 1s in A_j - i.e. the rows corresponding to the neighbors of node j.

If we do this repeatedly, we get that the value at node j is $(A^k G_0)_j = A_j^k$.

(b) Recall that a path from i to j in a graph is a sequence of vertices that starts with i, ends with j, and every successive vertex in the sequence is connected by an edge in the graph. The length of a path is the number of edges in it.

Here is some helpful notation:

- $V(i)$ is the set of vertices that are connected to vertex i in the graph.
- $L_k(i,j)$ is the number of distinct paths that go from vertex i to vertex j in the graph where the number of edges traversed in the path is exactly k.
- By convention, there is exactly 1 path of length 0 that starts at each node and ends up at itself. That is, $L_0(i,j) = 1_{i=j}$.

Prove that the i-th output of node j at layer k in the network above is the count of how many paths there are from i to j of length k.

(Hint: Induct on k.)

Solution:

We can prove the result by induction on k. For k = 1, the result follows from the very definition of A. Let $L_k(i,j)$ denote the number of paths of length k between nodes i and j, and assume that the result we wish to prove is true for some given $h \geq 1$, so that $L_h(i,j) = [A_h]_{i,j}$.

We next prove that it must also hold that $L_{h+1}(i,j) = [A_{k+1}]_{i,j}$, thus proving by inductive argument that $L_k(i,j) = [A_k]_{i,j}$ for all $k \geq 1$.

Indeed, to go from a node i to a node j with a walk of length h + 1, one needs first reach, with a walk of length h, a node l linked to j by an edge. Thus:

$$L_{h+1}(i,j) = \Sigma_{l \in V(j)} L_h(i,l) \tag{1}$$

where V(j) is the neighbor set of j, which is the set of nodes connected to the j-th node, that is, nodes l such that $A_{l,j} \neq 0$. Thus:

$$L_{h+1}(i,j) = \Sigma_{l=1}^{n} L_h(i,l) A_{l,j} \tag{2}$$

But we assumed that $L_h(i,j) = [A_h]_{i,j}$, hence the previous equation can be written as:

$$L_{h+1}(i,j) = \Sigma_{l=1}^{n}[A_h]_{i,l}A_{l,j} \tag{3}$$

In the above we recognize the (i,j)-th element of the product $A^h A = A_{h+1}$, which proves that $L_{h+1}(i,j) = [A_{h+1}]_{i,j}$, and hence concludes the inductive proof.

(c) The GNN we have worked on so far is essentially linear, since the operations performed at each layer are permutation-invariant locally at each node, and can be viewed as essentially doing the exact same thing at each vertex in the graph based on inputs coming from its neighbors. This is called "aggregation" in the language of graph neural nets.

If we represent the graph as a matrix, with the activatios of the i-th node in the i-th row, **what is the update function?**

In the case of the computations in previous parts, **what is the update function that takes the aggregated inputs from neighbors and results in the output for this node?**

Solution: The update function is to multiply by A on the left.[1]

If we represent the graph as a set of nodes, the update function is

$$v_j = \sum_{i \in V(j)} v_i.$$

(d) The simple GNN described in the previous parts counts paths in the graph. If we were to replace `sum` aggregation with `max` aggregation, **what is the interpretation of the outputs of node j at layer k?**

Solution: It is 1 if there is a path from i to j of length precisely k (not more or less), and 0 otherwise.

(e) You are studying how organic molecules break down when heated. For each molecule, you know the element for each atom, which other atoms it is connected to, the length of the bonds, and the type of molecule it is (carbohydrate, protein, etc.) You are trying to predict which bond, if any, will break first if the molecule is heated.

(i) **How would you represent this as a graph?** (What are the nodes, edges, and global state representations? Is it directed or undirected?) **Solution:** Each atom is a node with the element and weight as its value. There is an undirected edge between each pair of bonded atoms, with the bond length as the value. The global representation includes the molecule type.

(ii) **How would you use the outputs of the last GNN layer to make the prediction?** **Solution:** This is a classification problem across edges. One reasonable approach would be to softmax across logits produced by each edge as well as one logit produced by the global state which represents the option that no edges break.

(iii) **How would you encode the node representation for the input to the GNN?** **Solution:** Use a learned embedding for the element IDs. Have special learned embeddings at the global node that encode the type of the molecule.

[1]For numerical stability, in practice we would use the symmetrically normalized adjacency matrix $A^{SymNorm}$ instead.

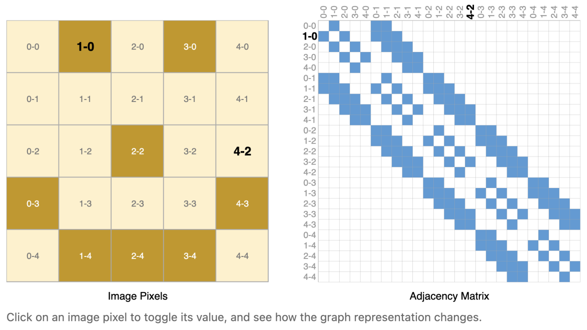


Image Pixels
Adjacency Matrix
Graph
Click on an image pixel to toggle its value, and see how the graph representation changes.

Figure 1: Images as Graphs

(f) There are analogs of many ConvNet operations which can be done with GNNs. As Figure 1 illustrates, we can think of pixels as nodes and pixel adjacencies as similar to edges. Graph-level classification tasks, for instance, are analogous to image classification, since both produce a single, global prediction. **Fill out the rest of the table.** (Not all rows have a perfect answer. The goal is to think about the role an operation serves in one architecture and whether you could use a technique which serves a similar role in the other architecture.)

CNN	GNN
Image classification	Graph-level prediction problem
Solution: Semantic segmentation (classifying what object is present at each pixel)	Node-level prediction problem
Color jitter data augmentation (adjusting the color or brightness of an image)	**Solution:** Jittering node values
Image flip data augmentation	**Solution:** A flip preserves graph values and connectivity, but changes its spatial orientation. Graphs don't have an orientation, so they don't need an analog. More generally, image flips augment the data by exploiting invariances in the task (i.e. an image's class shouldn't change if you flip it), and there might be similar invariances in graph problems.
Channel dropout	**Solution:** Set some channels for all nodes (or edges) to zero at a given layer of the network
Zero padding edges	**Solution:** Zero padding addresses the fact that CNNs require that every pixel has the same number of neighbors. Graphs don't need an equivalent since they already handle variable numbers of neighbors
ResNet skip connections	**Solution:** Add a skip connection to the update - i.e. $v_i^{t+1} = v_i^t + UPDATE(v_i^t)$
Blurring an image	**Solution:** Averaging node values with neighbors
Solution: Image inpainting (filling in a missing section of an image)	Predicting missing values of nodes

(g) **If you're doing a graph-level classification problem, but node values are missing for some of your graph nodes, how would you use this graph for prediction?** **Solution:** There are multiple strategies (including all of the strategies used in other ML problems with missing values), including creating a special 'missing' token that is learnable or filling the node with the mean value, and — very importantly — training with augmentations on the input that randomly remove nodes from training graphs. In graphs where individual nodes don't matter much (e.g. point clouds), it may be appropriate to remove the node and associated edges entirely instead of masking the value inside.

(h) Consider the graph neural net architecture shown in Figure 2. It includes representations of nodes (V_n), edges (E_n), and global state (U_n). At each timestep, each node and edge is updated by aggregating neighboring nodes/edges, as well as global state. The global state is the updated by getting information from all nodes and edges. For more details on the architecture setup, see `https://distill.pub/2021/gnn-intro/#passing-messages-between-parts-of-the-graph`.

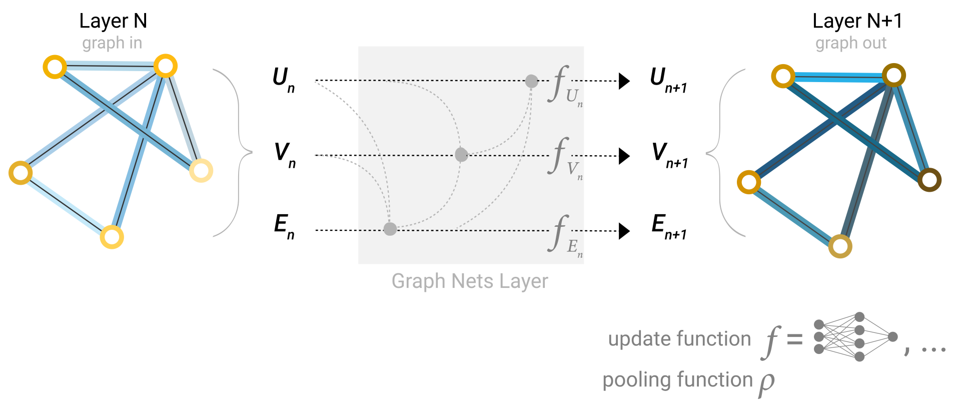


Figure 2: GNN architecture

(i) **If we double the number of nodes in a graph which only has node representations, how does this change the number of learned weights in the graph? How does it change the amount of computation used for this graph if the average node degree remains the same? What if the graph if fully connected?** (Assume you are not using a global state representation).
Solution: The number of learned weights remains the same. If the node degree remains the same, computation will also double. If the graph is fully connected, computation scales by a factor of 4 (2x nodes * 2x neighbors to aggregate per timestep). (Here, we're thinking about how computational complexity scales with # nodes and edges. There are some fixed costs which don't scale, so it won't be quite 2x or 4x in practice.)

(ii) **Where in this network are learned weights/parameters?** **Solution:** The update functions include learned weights.

(iii) The diagram provided shows undirected edges. **How would you incorporate directed edges?**
Solution: There are multiple strategies here, but one reasonable option is that within a node's update function, use different weights to incorporate incoming and outgoing edges and nodes. (Incoming edges are edges pointing toward the node, and outgoing edges have the arrows away from the node.) In some cases, it might make sense to use only incoming or outgoing nodes/edges for the update.

3. Graph Neural Networks

For an undirected graph with no labels on edges, the function that we compute at each layer of a Graph Neural Network must respect certain properties so that the same function (with weight-sharing) can be used at different nodes in the graph. Let's focus on a single particular "layer" ℓ. For a given node i in the graph, let $\mathbf{s}_i^{\ell-1}$ be the self-message (i.e. the state computed at the previous layer for this node) for this node from the preceeding layer, while the preceeding layer messages from the n_i neighbors of node i are denoted by $\mathbf{m}_{i,j}^{\ell-1}$ where j ranges from 1 to n_i. We will use w with subscripts and superscripts to denote learnable scalar weights. If there's no superscript, the weights are shared across layers. Assume that all dimensions work out.

(a) **Tell which of these are valid functions for this node's computation of the next self-message $\mathbf{s}_i^{\ell}$. For any choices that are not valid, briefly point out why.**
Note: we are *not* asking you to judge whether these are useful or will have well behaved gradients. Validity means that they respect the invariances and equivariances that we need to be able to deploy as a GNN on an undirected graph.

(i) $\mathbf{s}_i^\ell = w_1 \mathbf{s}_i^{\ell-1} + w_2 \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{m}_{i,j}^{\ell-1}$

Solution: This is valid because it is permutation invariant to the ordering of neighbors. This is the classic averaging form. Notice that a dependence on the number of neighbors is fine.

(ii) $\mathbf{s}_i^\ell = \max(w_1^\ell \mathbf{s}_i^{\ell-1}, w_2 \mathbf{m}_{i,1}^{\ell-1}, w_3 \mathbf{m}_{i,2}^{\ell-1}, \ldots, w_{n_i+1} \mathbf{m}_{i,n_i}^{\ell-1})$ where the max acts component-wise on the vectors.

Solution: This is invalid. Since different scalar weights are applied to different $\mathbf{m}$, it is not permutation invariant to the ordering of neighbors.

(iii) $\mathbf{s}_i^\ell = \max(w_1^\ell \mathbf{s}_i^{\ell-1}, w_2 \mathbf{m}_{i,1}^{\ell-1}, w_2 \mathbf{m}_{i,2}^{\ell-1}, \ldots, w_2 \mathbf{m}_{i,n_i}^{\ell-1})$ where the max acts component-wise on the vectors.

Solution: This is valid. Since the same weight w_2 is applied to all $\mathbf{m}$, it is permutation invariant to the ordering of neighbors. The max is another classic permutation-invariant operation.

(b) We are given the following simple graph on which we want to train a GNN. The goal is binary node classification (i.e. classifying the nodes as belonging to type 1 or 0) and we want to hold back nodes 1 and 4 to evaluate performance at the end while using the rest for training. We decide that the surrogate loss to be used for training is the average binary cross-entropy loss.

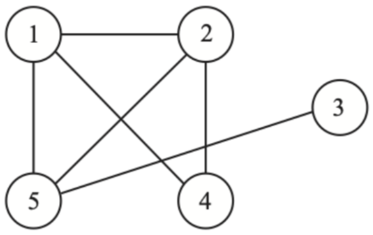


Figure 3: Simple Undirected Graph

nodes	1	2	3	4	5
y_i	0	1	1	1	0
$\hat{y}_i$	a	b	c	d	e

Table 1: y_i is the ground truth label, while $\hat{y}_i$ is the predicted probability of node i belonging to class 1 after training.

Table 1 gives you relevant information about the situation.

Compute the training loss at the end of training.

Remember that with n training points, the formula for average binary cross-entropy loss is

$$\frac{1}{n}\sum_x \left(y(x) \log \frac{1}{\hat{y}(x)} + (1 - y(x)) \log\left(\frac{1}{1-\hat{y}(x)}\right)\right)$$

where the x in the sum ranges over the training points and $\hat{y}(x)$ is the network's predicted probability that the label for point x is 1.

Solution:

Since our testing nodes are nodes $\{1, 4\}$, a mask will be used to remove their contribution to the loss. We therefore get:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{3}[y_2 \times \log(\hat{y}_2) + y_3 \times \log(\hat{y}_3) + y_5 \times \log(\hat{y}_5) + (1 - y_5) \times \log(1 - \hat{y}_5)] \qquad (4)$$

$$= -\frac{1}{3}[(1. \times \log(b)) + (1. \times \log(c)) + (0 \times \log(e)) + \log(1-e)] \tag{5}$$

$$= -\frac{1}{3}[\log(b) + \log(c) + \log(1-e)] \tag{6}$$

$$= -\frac{1}{3}[\log\big(bc(1-e)\big)] \tag{7}$$

(c) Suppose we decide to use the following update rule for the internal state of the nodes at layer ℓ.

$$\mathbf{s}_i^{\ell} = \mathbf{s}_i^{\ell-1} + W_1 \frac{\sum_{j=1}^{n_i} \tanh\Big(W_2 \mathbf{m}_{i,j}^{\ell-1}\Big)}{n_i} \tag{8}$$

where the $\tanh$ nonlinearity acts element-wise.

For a given node i in the graph, let $\mathbf{s}_i^{\ell-1}$ be the self-message for this node from the preceding layer, while the preceding layer messages from the n_i neighbors of node i are denoted by $\mathbf{m}_{i,j}^{\ell-1}$ where j ranges from 1 to n_i. We will use W with subscripts and superscripts to denote learnable weights in matrix form. If there's no superscript, the weights are shared across layers.

(i) **Which of the following design patterns does this update rule have?**

☐ Residual connection

☐ Batch normalization

Solution: This rule shows the residual connection pattern since the previous layer's state gets added in. This kind of residual connection enables gradients to flow back to earlier layers more easily. None of the other patterns are present.

(ii) **If the dimension of the state s is d-dimensional and W_2 has k rows, what are the dimensions of the matrix W_1?**

Solution: W_1 needs to return something that can add to the state. This means that it needs d rows. It also has to be able to act on vectors that are k dimensional since W_2 has k rows and we say that the $\tanh$ operation acts element-wise. This means that W_1 needs k columns. Putting this together, the dimensions of W_1 are $d \times k$.

(iii) If we choose to use the state $\mathbf{s}_i^{\ell-1}$ itself as the message $\mathbf{m}^{\ell-1}$ going to all of node i's neighbors, **please write out the update rules corresponding to** (8) **giving $\mathbf{s}_i^{\ell}$ for the graph in Figure 3 for nodes $i = 2$ and $i = 3$ in terms of information from earlier layers.** Expand out all sums.

Solution: We can write the update rule for each node by considering its neighborhood N_i, which is the set of nodes connected to node i. Particularly, we will use (8) with $\mathbf{m}_{i,j}^{\ell-1} = \mathbf{s}_j^{\ell-1}$.

For the graph in Figure 3, we can identify the neighborhoods: - $N_1 = \{2, 4, 5\}$ - $N_2 = \{1, 4, 5\}$ - $N_3 = \{5\}$ - $N_4 = \{1, 2\}$ - $N_5 = \{1, 2, 3\}$

Therefore:

$i = 2$:

$$s_2^l = s_2^{l-1} + \frac{W_1}{3}(\tanh\Big(W_2 s_1^{l-1}\Big) + \tanh\Big(W_2 s_4^{l-1}\Big) + \tanh\Big(W_2 s_5^{l-1}\Big))$$

$i = 3$:

$$s_3^l = s_3^{l-1} + \frac{W_1}{1}(\tanh\Big(W_2 s_5^{l-1}\Big))$$

The above equations can be found by reducing the sum in Equation 8 and applying the accurate neighbors from Figure 3.

4. Exploring Deep Learning Tooling

Deep learning in practice often requires the use of various tooling in order make the learning workflow efficient. Among these include tools for creating graphs and organizing experiments. These kinds of tools can aid in careful ablation studies, and can accelerate research. We will explore the use of two different tools in this question: tensorboard and wandbai.

(a) Complete the first notebook: tensorboard.ipynb. Provide your graphs here. What is easy about tensorboard? What do you dislike? Would it still be easy to use when we need to run massive amounts of experiments? How organized is it?

Solution: Students should include some training graphs here. Tensorboard allows students to log runs without much thought, as well as visualize them. Students should note that it can be quite disorganized and harder to use because they can only sort with regex. It would not be good to run massive amounts of experiments due to the inabilty to sort by configurations.

(b) Complete the second notebook: wandb.ipynb. No need to provide your graphs. What does wandb have that tensorboard does not?

Solution: Students should include the best hyperparameters. In addition they should note that wandb is useful for visualizations and having many runs. Since wandb allows for sorting based on configuration, this could be very useful.

5. Zachary's Karate Club (Coding)

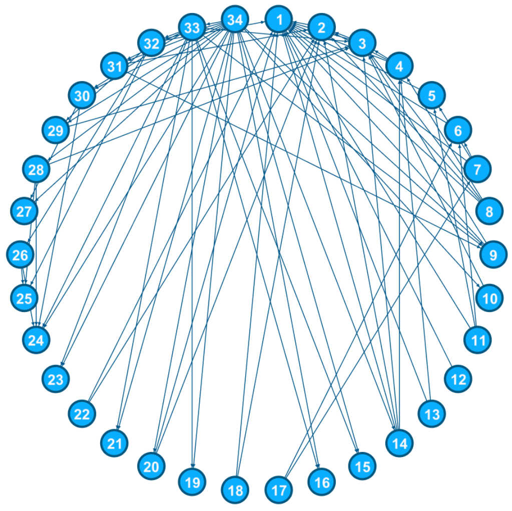


Figure 4: Zachary's Karate Club Graph

Zachary's Karate Club (ZKC) is a social network of a university karate club, described in the paper "An Information Flow Model for Conflict and Fission in Small Groups" by Wayne W. Zachary.

A social network captures 34 members of a karate club, documenting links between pairs of members who interacted outside the club.

During the study a conflict arose between the officer/ administrator ("John A") and the instructor "Mr. Hi", which led to the split of the club into two.

Half of the members formed a new club around Mr. Hi; members from the other part found a new instructor or gave up karate.

Based on collected data Zachary correctly assigned all but one member of the club to the groups they actually joined after the split. You could read more about it here https://www.jstor.org/stable/3629752, and here https://commons.wikimedia.org/wiki/File:Social_Network_Model_of_Relationships_in_the_Karate_Club.png

We will train a GNN to cluster people in the karate club in such that people who are more likely to associate with either the officer or Mr. Hi will be close together, while the distance beween the 2 classes will be far.

In the original paper titled "Semi-Supervised Classification with Graph Convolutional Networks" that can be found here https://arxiv.org/pdf/1609.02907.pdf, the authors framed this as a node-level classification problem on a graph. We will pretend that we only know the affiliation labels for some of the nodes (which we'll call our training set) and we'll predict the affiliation labels for the rest of the nodes (our test set).

Implement all the TODOs in `zkc.ipynb` and include your notebook with your submission.

(a) Go through `q_zkc.ipynb`. We want our network to be aware of information about the nodes themselves instead of only the neighborhood, so we add self loops our adjacency matrix. The paper called this $\tilde{A}$. **Compute $\tilde{A}$ to add self loops to your adjacency matrix.**

Solution: See solution notebook.

(b) **Write a function that takes in $\tilde{A}$ as argument and returns the $\tilde{A}^{SymNorm}$ adjacency matrix**.

Solution: See solution notebook.

(c) The other input to our GNN is the graph node matrix X which contains node features. For simplicity, we set X to be the identity matrix because we don't have any node features in this example. **Generate the feature input matrix X.**

Solution: See solution notebook.

(d) We will now implement a single layer GNN. **Implement the forward and backward pass functions for `GNN_Layer` class.** Details can be found in the notebook.

Solution: See solution notebook.

(e) **Run the forward and backward passes and ensure the checks pass.**

Solution: See solution notebook.

(f) We are now ready to setup our classification network! **Use the GNN and Softmax layers to setup the network.**

Solution: See solution notebook.

(g) **Instantiate the GNN model with the correct input and output dimensions.**

Solution: See solution notebook.

(h) With the model, data and optimizer ready, **fill in the todos in the training loop function and train your model. Plot the clustered data.**

(i) **Explain why we obtain 100% on accuracy on our test set, yet we see in the plot that 2 samples seem to be misclassified.**

Solution: Those samples were part of the training set and were never used to evaluate the model at test time. All nodes used in the test set were correctly classified hence the 100% accuracy.

6. Implementing Muon (Coding Question)

In this assignment, you will implement key components of a Muon optimizer in PyTorch. For this assignment, we recommend using Google Colab as some PCs or laptops may not support GPU-based PyTorch. You will be implementing the code at q_coding_muon.ipynb, then answering the questions in the notebook in your submission. The notebook itself does not need to be submitted.

(a) Notice that in this Newton-Schulz implementation, we initially scale the Frobenius norm to be at most $\sqrt{3}$. **Can you explain why we choose this particular scaling?** Comment on why this is better than using 1.

(Hint: Inspect the roots of the cubic polynomial. What is the connection between the roots and the convergence properties of the singular values? You can refer to Discussion 4 for the answer)

Solution: We choose $\sqrt{3}$ as the positive root of the cubic polynomial. This is because for singular values $> \sqrt{3}$, Newton-Schulz (NS) using our polynomial will not converge to 1.

Note that the Frobenius norm is a conservative overestimate of the the Spectral norm. When we have a mini-batch size larger than 1 and even without that in the case of a conv-net where we have weight-sharing for conv-filters across the image, we generally don't expect the gradients to be rank-1. They might be effectively low rank, but 1 is a bit extreme. Consequently, if we scale the Frobenius norm to 1, while we are being safe, we also are probably scaling many of the interesting singular values to be close to 0. Iterating the polynomial for singular values close to zero is like multiplying by $\frac{3}{2}$, the slope at the origin. This means that we expect exponential growth in those singular values as $(\frac{3}{2})^k$. Notice that $\sqrt{3} > \frac{3}{2}$ and so adjusting this scaling factor is like getting a free iteration of NS. Since these matrix multiplies in NS iterations are nontrivially expensive, this is a computational savings worth taking.

(b) Note that Muon requires that parameters are 2D matrices of shape $d_{out} \times d_{in}$. However, we know that parameters that are convolutional kernels have shape $c_{out} \times c_{in} \times k \times k$, where c denotes number of channels and k is kernel size.

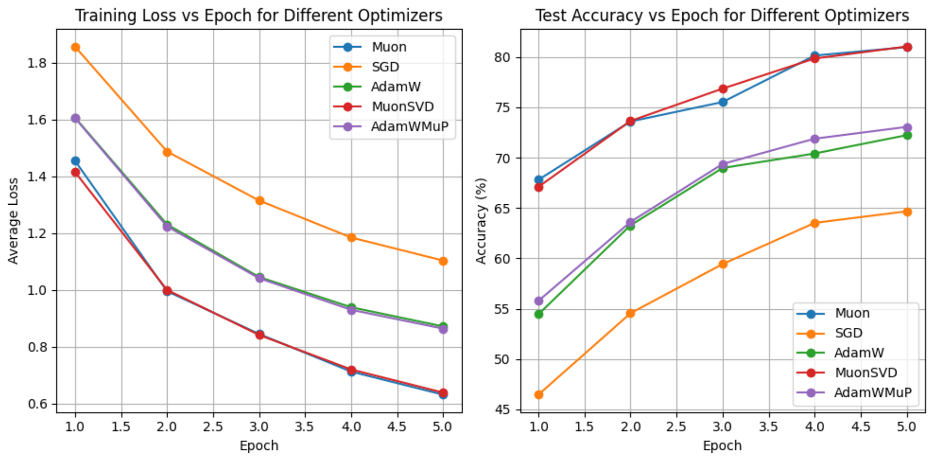
Modern implementations of convolutional layers will (implicitly) transform an input image $\mathbf{x}$ of shape $c_{in} \times h \times w$ to $\mathbf{x}'$ such that each column of $\mathbf{x}'$ has size $c_{in} \cdot k \cdot k$ and corresponds to one flattened "receptive field" of the image (or one patch of the image that a filter passes over to compute one output value). This allows the GPU to use heavily optimized matrix-multiplication capabilities directly. You may look up im2col for more details and take a look at `https://arxiv.org/abs/2110.03901` if you are interested.

Given this fact, **how do we modify the convolutional kernel into a $d_{out} \times d_{in}$ matrix C such that the output of the convolutional layer can be expressed as $C\mathbf{x}'$?**

Solution: Before orthogonalization, we reshape the convolutional kernel by flattening all dimensions except for c_{out}, resulting in a 2D matrix of dimensions $c_{out} \times (c_{in} \cdot k \cdot k)$. The reason why is because such flattened matrix C is the one such that $C\mathbf{x}'$ is exactly the output of the convolutional layer applied to input $\mathbf{x}$.

(c) **Which optimizer performed best between Muon, SGD, and AdamW?** Also **copy the resulting plots** into the submission as well.

Solution: Here are the output plots evaluating Muon against baseline optimizers including SGD and AdamW:



It is clear from the curves that Muon performs best, achieving almost 10% higher accuracy on the test set than AdamW, and 15% than SGD. This trend is consistent across all epochs.

Even more surprising is the seeming evidence of a better inductive bias for Muon relative to SGD. Look at the training loss for Epoch 1 of Muon. It's between that of Epochs 2 and 3 of SGD. So far, this is just evidence of faster convergence in training loss. But look at the test accuracy. The test loss at Epoch 1 of Muon is better than SGD at the very end of training. This means that, at least in this phase of training (remember what you saw in the homework comparing Adam to SGD. Advantages can shift.), the mapping from training loss to test accuracy is favorable for Muon over SGD. Notice that relative to Adam, a similar tendency holds — Epoch 2 of Adam is better training loss than Epoch 1 of Muon. But the test accuracy for Muon at the end of Epoch 1 is better than Adam at the end of Epoch 2.

(d) Compare the loss curves between Muon and MuonSVD. **Are the results expected? Explain why.**

Solution: MuonSVD performs the same as Muon. This is expected because Newton-Schulz aims to be a tractable approximation to orthogonalization, which SVD does exactly but less efficiently.

(e) (Optional) Our implementation of Newton-Schulz is suboptimal in the polynomial used for convergence. The community has developed quintic polynomials that converge faster while still being efficient. **Implement an improved Newton-Schulz and compare. Comment on the speed advantage of the improved Muon relative to the MuonSVD.**

(Hint: You can modify the number of iterations by setting the `ns_iters` *parameter in the Muon optimizer)*

Solution: If we replace our initial cubic polynomial with a quintic one, we are able to converge in fewer iterations (from $5 \to 3$). This makes the Muon run 25% faster than MuonSVD.

(f) (Optional) **What were the best choices for hyperparameters for Muon? What about for AdamW?**

Solution: For Muon, the best choice for learning rate was 0.01 and for weight decay was 0.001. For AdamW, the best choice for learning rate was 0.001 and for weight decay was 0.001.

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Romil Bhardwaj.
- Linyuan Gong.
- Jerome Quenum.
- Olivia Watkins.
- Anant Sahai.
- Anrui Gu.
- Matthew Lacayo.
- Past EECS 282 and 227 Staff.
- Naman Jain.
- Liam Tan.
- Joey Hong.
- Gireeja Ranade.