EECS 182 Deep Neural Networks
Fall 2023 Anant Sahai

Homework 6

This homework is due on October 7, at 10:59PM.

1. Debugging DNNs

(a) Your friends want to train a classifier for a new app they're designing. They implement a deep convolutional network and train models with two configurations: a 20 layer model and a 56 layer model. However, they observe the following training curves and are surprised that the 20-layer network has better training as well as test error.

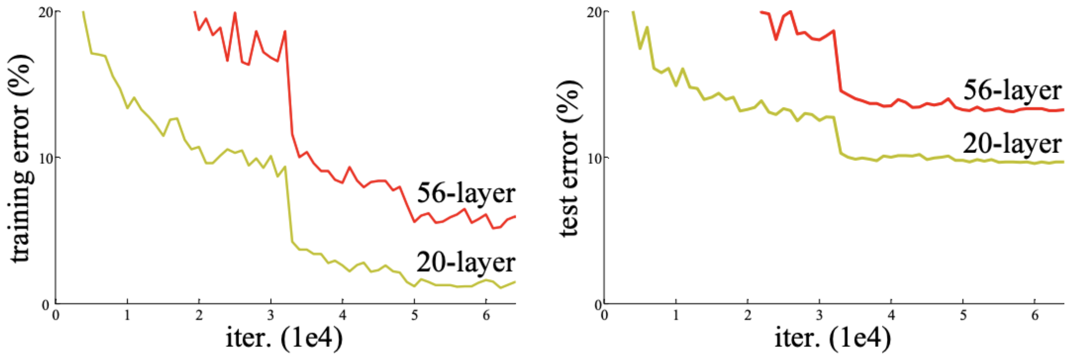

Figure 1: Training deep networks on CIFAR10

What are the potential reasons for this observation? Are there changes to the architecture design that could help mitigate the problem?

(b) You and your teammate want to compare batch normalization and layer normalization for the ImageNet classification problem. You use ResNet-152 as a neural network architecture. The images have input dimension $3 \times 224 \times 224$ (channels, height, width). You want to use a batch size of 1024; however, the GPU memory is so small that you cannot load the model and all 1024 samples at once — you can only fit 32. Your teammate proposes using a gradient-accumulation algorithm:

> Gradient accumulation refers to running the forward and backward pass of a model a fixed number of steps (`accumulation_steps`) without updating the model parameters, while aggregating the gradients. Instead, the model parameters are updated every (`accumulation_steps`). This allows us to increase the effective batch size by a factor of `accumulation_steps`.

You implement the algorithm in PyTorch as:

```
model.train()
optimizer.zero_grad()
for i, (inputs, labels) in enumerate(training_set):
    predictions = model(inputs)
    loss = loss_function(predictions, labels)
    loss = loss / accumulation_steps
    loss.backward()
    if (i+1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

Note that the `.backward()` operator in PyTorch implicitly keeps accumulating the gradient for all the parameters, unless zero'd out with an `optimizer.zero_grad()` call.

Before running actual experiments, your friend suggests that you should test whether the gradient accumulation algorithm is implemented correctly. To do so, you collect the output logits (i.e. the outputs of the last layer) from two models — ResNet-152, one with batchnorm and the other with layernorm — using different combinations of batch sizes and the number of accumulation steps that keep the effective batch size to 32.

[*Note that the effective batch size is product of the batch size and* `accumulation_steps`*. In other words, the possible combinations for effective batch-size 32 are:*
(`batch_size`*,* `accumulation_steps`*) = (1, 32), (2, 16), (4, 8), (8, 4), (16, 2), (32, 1).*]

Here, the (32,1) combination is the baseline approach without the "gradient accumulation" trick, and we want to see whether the others output exactly the same logits (up to floating point inaccuracies) as the baseline.

On running these tests, you observe that one of models (either with batchnorm or with layernorm), doesn't pass the test. **Which one is it, and why?**

(c) You are training a CIFAR model and observe that the model is diverging (instead of the training loss decreasing over iterations). **Debug the pseudocode and give a correction that you believe would actually result in reasonable convergence during training.**

(Note: You can assume that the datasets are loaded correctly, model is trained with SGD optimizer with learning rate= 0.001*, batchsize=* 100*)*

(HINT: Ideas from the previous part of this question might be relevant.)

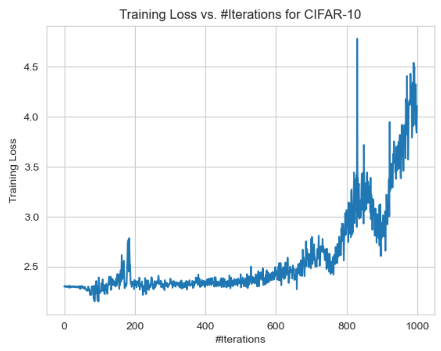

Figure 2: Training loss for CIFAR10

```
model.train()
optimizer.zero_grad()
for (inputs, labels) in training_set:
    predictions = model(inputs)
    loss = loss_fn(predictions, labels)
    loss.backward()
    optimizer.step()
```

2. Tensor Rematerialization

You want to train a neural network on a new chip designed at UC Berkeley. Your model is a 10 layer network, where each layer has the same fixed input and output size of s. The chip your model will be trained on is heavily specialized for model evaluation. It can run forward passes through a layer very fast. However, it is severely memory constrained, and can only fit in memory the following items (slightly more than twice of the data necessary for performing a forward pass):

(a) the inputs;

(b) $2s$ activations in memory;

(c) optimizer states necessary for performing the forward pass through the current layer.

To train despite this memory limitation, your friend suggests using a training method called **tensor rematerialization**. She proposes using SGD with a batch size of 1, and only storing the activations of every 5th layer during an initial forward pass to evaluate the model. During backpropagation, she suggests recomputing activations on-the-fly for each layer by loading the relevant last stored activation from memory, and rerunning forward through layers up till the current layer.

Figure 3 illustrates this approach. Activations for Layer 5 and Layer 10 are stored in memory from an initial forward pass through all the layers. Consider when weights in layer 7 are to be updated during backpropagation. To get the activations for layer 7, we would load the activations of layer 5 from memory, and then run them through layer 6 and layer 7 to get the activations for layer 7. These activations can then be used (together with the gradients from upstream) to compute the gradients to update the parameters of Layer 7, as well as get ready to next deal with layer 6.

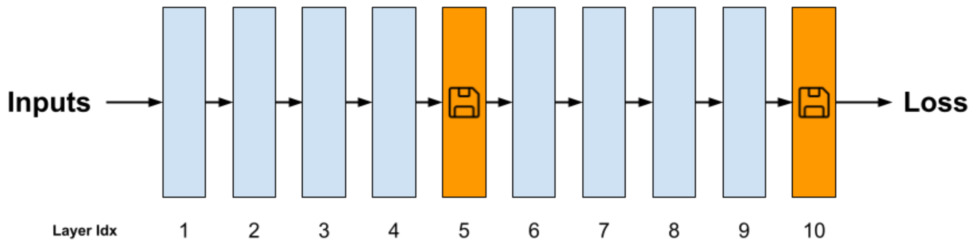

Figure 3: Tensor rematerialization in action - Layer 5 and Layer 10 activations are stored in memory along with the inputs. Activations for other layers are recomputed on-demand from stored activations and inputs.

(a) Assume a forward pass of a single layer is called a `fwd` operation. **How many `fwd` operations are invoked when running a single backward pass through the entire network?** Do not count the initial forward passes required to compute the loss, and don't worry about any extra computation beyond activations to actually backprop gradients.

(b) Assume that each memory access to fetch activations or inputs is called a `loadmem` operation. **How many `loadmem` operations are invoked when running a single backward pass?**

(c) Say you have access to a local disk which offers practically infinite storage for activations and a `loaddisk` operation for loading activations. You decide to not use tensor rematerialization and instead store all activations on this disk, loading each activation when required. Assuming each `fwd` operation takes 20ns and each `loadmem` operation (which loads from memory, not local disk) takes 10ns for tensor rematerialization, **how fast (in ns) should each `loaddisk` operation be to take the same time for one backward pass as tensor rematerialization?** Assume activations are directly loaded to the processor registers from disk (i.e., they do not have to go to memory first), only one operation can be run at a time, ignore any caching and assume latency of any other related operations is negligible.

3. Graph Dynamics

Some graph neural network methods operate on the full adjacency matrix. Others, such as those discussed here 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.)

(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.)

(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?**

(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?**

4. Graph Neural Networks (Optional)

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

(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.

(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.

(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.

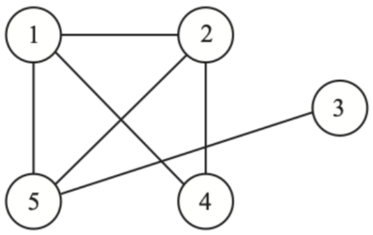

Figure 4: Simple Undirected Graph

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.

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

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

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 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 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

(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?**

(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 (1) giving $\mathbf{s}_i^{\ell}$ for the graph in Figure 4 for nodes $i=2$ and $i=3$ in terms of information from earlier layers.** Expand out all sums.

5. Learning from Point Clouds (Optional)

A point cloud is a discrete *set* of data points in space. Because of this *set*-valued nature of point clouds, concepts from graph neural nets are often relevant in their processing.

In Fig.5, we consider a simple network to process a 2d point cloud $X=\{x_i\}_{i=1}^{n}\in\mathbb{R}^{n\times 2}$, where n is the number of points. The original features of each point are its horizontal and vertical coordinates.

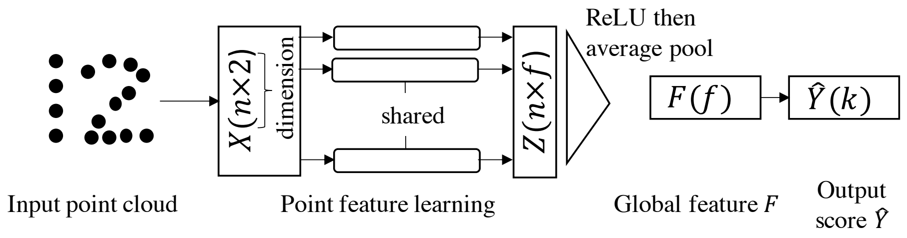

Figure 5: 2d point cloud processing network.

For example, an input point cloud with ground-truth of digit 1 could be represented as $\begin{bmatrix}0&4\\0&3\\0&2\\0&1\end{bmatrix}$ where each row is a different point in the point cloud.

(a) The *point feature learning* module in Figure 5 learns f-dimensional features for each point separately. Specifically, it learns (shared) weights $W_1 \in \mathbb{R}^{2\times f}$ to get hidden layer outputs $Z = XW_1$. We then apply the nonlinear activation function element-wise and then use average pooling to yield the f-dimensional global feature vector $F \in \mathbb{R}^f$.

Suppose we swap the first two points of the input point cloud X, i.e. $\{x_1, x_2, ..., x_n\}$ to $\{x_2, x_1, ..., x_n\}$. **Show that the global feature F will not change.**

Note: In reality, the network here is *permutation invariant*, as changing the ordering of the n input points in X will not affect the global feature F.

(b) One drawback of the point feature learning discussed in part (a) is that the spatial inter-relationships of different points is not considered.

One idea is to use the Euclidean distance as a similarity measure to group points together into local neighborhoods, and to then process each point together with contextual information about that neighborhood. Your friend proposed several different point grouping methods listed below.

Select all methods that guarantee permutation-invariance, i.e. the global feature vector F will not change with different orderings of the points. Please use ■ for your selections.

- ☐ For each point x_i of the point cloud X, we find the top-m nearest neighbor points. We then augment each point coordinates with its nearest neighbors' coordinates to make $\widetilde{X} \in \mathbb{R}^{n\times 2(m+1)}$. The order of a concatenated group of points would be: the center point, 1st closest, 2nd closest point, ..., m^{th} closest point. Now $Z = \widetilde{X}\widetilde{W_1}$ where $\widetilde{W_1} \in \mathbb{R}^{2(m+1)\times f}$ are the shared learnable weights.
- ☐ For each point x_i of the point cloud X, we find the top-m nearest neighbor points. As in the previous choice, we then augment each point coordinates with its nearest neighbors' coordinates and get $\widetilde{X} \in \mathbb{R}^{n\times 2(m+1)}$. *The difference from the previous choice is that the order of a concatenated group of nearby points simply follows their order in the original X.* The $\widetilde{W_1}$ is the same as the previous choice.
- ☐ For each point x_i of the point cloud X, we *instead find all neighboring points with the distance to x_i smaller than a predefined radius r. We then augment each point coordinates with its neighbors' coordinates with the order being the center point, the 1st closet point within r, the 2nd closet point within r, ..., the furthest point within r.* Because different points might have a different number of neighbors within radius r, the shared learnable weights $\widetilde{W_1} \in \mathbb{R}^{2(n+1)\times f}$ are applied by using the relevant-size truncation of $\widetilde{W_1}$ for every point.
- ☐ For each point x_i of the point cloud X, as in the previous option, we find all neighboring points with the distance to x_i smaller than a predefined radius r. But instead of concatenating the representation of the point with its neighboring points, we instead extend the point's representation with just the distance d from x_i to the furthest point within the radius r resulting in an $\widetilde{X} \in \mathbb{R}^{n\times 3}$. The $\widetilde{W_1} \in \mathbb{R}^{3\times f}$.

(c) *Point downsampling (reducing the number of points for deeper layers to process).* Let's consider a deeper network, as shown in Fig.6. We are adding a pooling layer (highlighted in bold text) after the first point feature learning layer to downsample half of the points in the cloud ($Z \to Z_1$), followed by another point feature learning layer to increase the dimension of the point features from f to $2f$ ($Z_1 \to Z_2$). (Note that this mimics pooling procedures in CNNs: the spatial size shrinks, followed by an increase of the feature dimensionality.)

Consider two candidate algorithms. Both start with the point cloud comprising n points and both iteratively select points until you have at least $\frac{n}{2}$ samples. For both, we construct two sets: **sampled**

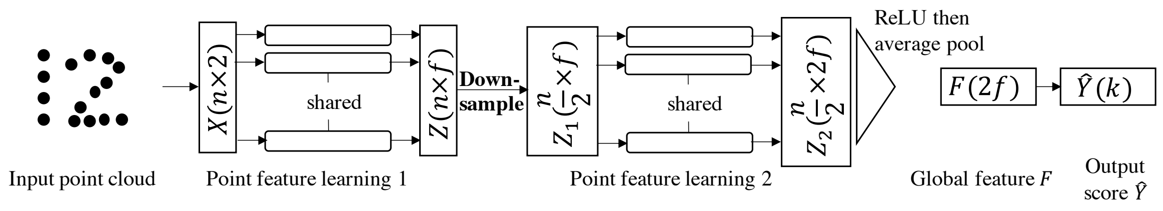

Figure 6: A deeper point cloud processing network.

and **remaining** to denote the set of sampled and remaining points. For both, we first pick a random point and use it to initialize **sampled** and we initialize **remaining** with all the other points. Then, the iterative processes are different for the two algorithms as described below.

Which algorithm is more similar in spirit to using standard max pooling for downsampling in CNNs? Please use ■ for your selections.

☐ **Algorithm 1:**

i. For each point in **remaining** find its nearest neighbor in **sampled**, saving the distance.
ii. Select the point in **remaining** whose nearest neighbor distance is the *largest* and move it from **remaining** to **sampled**. (*i.e.* We keep points far from those we already have.)

☐ **Algorithm 2:**

i. For each point in **remaining** find its nearest neighbor in **sampled**, saving the distance.
ii. Select the point in **remaining** whose nearest neighbor distance is the *smallest* and move it from **remaining** to **sampled**. (*i.e.* We keep points close to those we already have.)

At the end, only the points in **sampled** get sent on to the next layer.

6. The power of the graph perspective in clustering (Coding)

Implement all the TODOs in the q_graph_clustering.ipynb. **Answer the written questions below.**

(a) We used the KMeans algorithm implementation of sklearn, and showed our attempt to cluster this dataset into 3 classes. **Comment on the output the KMeans algorithm. Did it work? If so, explain why, if not, explain why not.**

(b) `adjacency_matrix = ?`

(c) As given, the data points in our dataset are represented simply with their 2D Cartesian coordinates. Let's now interpret every single point as a node in a graph. Our goal is to find a way to relate every node in the graph in such way that the points that are closer together and points that are far apart maintain that relationship explicitly.

That is, we will choose to look at every point in the dataset as a vertex in a graph where the edge connection between two vertexes is determined by the weighted distances between them.

In the notebook, implement a function that takes in the input dataset and some coefficient gamma and returns the adjacency matrix A:

$$A_{i,j} = e^{-\gamma||x_i - x_j||^2} \tag{2}$$

where x_i and x_j represent each point in the provided dataset, γ is positive. You may find the `distance` module from `scipy.spatial` useful.

Is this a directed or an undirected graph?

(d) The degree matrix of an undirected graph is a diagonal matrix which contains information about the degree of each vertex. In other words, it contains the number of edges attached to each vertex and it is given by:

$$D_{i,j} = \begin{cases} deg(v_i) & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

where the degree $\deg(v_i)$ of a vertex counts the number of times an edge terminates at that vertex. Note that in the traditional definition of the adjacency matrix, this boils down to the diagonal matrix in which element along the diagonals are column-wise sum of the adjacency matrix.

Using the same idea, **write a function that takes in the adjacency matrix as an argument and returns the degree matrix.**

(e) Using γ = 7.5, **compute the adjacency matrix A, degree matrix D and the symmetrically normalized adjacency matrix matrix** M,

$$M = A^{SymNorm} = D^{-1/2} A D^{-1/2} \tag{3}$$

Note that another interpretation of the matrix M is that it shows the probability of moving/jumping from one node to another.

(f) Applying SVD decomposition on M, **write a function that selects the top 3 vectors (corresponding to the highest singular values) in the matrix U and performs the same KMeans clustering used above on them. Show the plots. What do you observe? Did it work? If so, explain why, if not, explain why not.**

Intuition: By selecting the top 3 vectors of the U matrix, we are selecting a new representation of the data points which could be seen as a construction of a low dimension embedding of the data points as mentioned in problem 3.

(g) Now let's think of the symmetrically normalized adjacency matrix obtained above as the transition matrix in of a Markov Chain. That is, it represents the probability of jumping from one node to another. In order to fully interpret M in such way, it needs to be a proper stochastic matrix which means that the sum of the elements in each column must add up to 1. **Write a function that takes in the matrix M and returns M_{stoch}, the stochastic version of M; compute the stochastic matrix.**

Using SVD decomposition on the newly obtained stochastic matrix M_{stoch}, **use your function in part (e) to select the top 3 vectors of the matrix U_{stoch} and perform the same KMeans clustering used above on them and show the plots. What do you observe? Did it work?**

(h) Now, let's investigate how we could have made the matrix M work directly in our original interpretation. To do this, normalize those 3 vectors before performing the clustering.

Show the plots. What do you observe? Did it work? If so, explain why normalizing the vectors gives what is expected.

Hint: you may use

```
from sklearn.preprocessing import normalize
```

7. Zachary's Karate Club (Coding)

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.

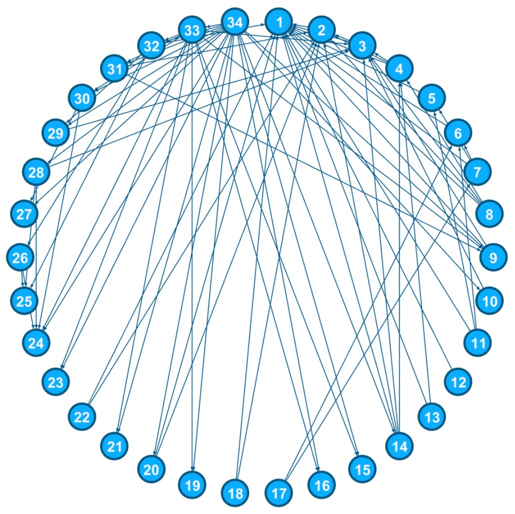

Figure 7: Zachary's Karate Club Graph

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.**

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

(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.**

(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.

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

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

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

(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.**

8. Stochastic Gradient Descent (when it is possible to interpolate)

This is a problem about the convergence of SGD for least-squares problems when there is actually a solution that achieves zero loss.

For simplicity, suppose that the problem we are given is

$$X\mathbf{w} = \mathbf{y} \tag{4}$$

where $X = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$ is a wide matrix with $\mathbf{x}_i$ being d-dimensional vectors and $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ is an n-dimensional vector. Here, we assume that X has full row-rank (i.e. the $\mathbf{x}_i$ are linearly independent) and so (4) indeed has solutions. (Note that as $d > n$, there there would be infinitely many solutions.)

While we already know lots of ways of solving this problem, it is an illustrative toy example to make sure we understand why SGD works in such settings. (This material was covered in lecture but you really do need to understand it yourself so you can deal with variations you might encounter.) This problem has a jupyter demo (**demo link**) attached to it at the end to help you play around with things to get an even deeper set of intuitions.

In this problem, we will just initialize $\mathbf{w}_0 = \mathbf{0}$ for simplicity.

(a) Let's do some preliminaries. First, we want to change coordinates to notationally simplify our analysis of SGD.

Let $\mathbf{w}^*$ be the min-norm solution to (4).

Write out what $\mathbf{w}^*$ is explicitly with respect to X and $\mathbf{y}$ and then, change coordinates to $\mathbf{w}' = \mathbf{w} - \mathbf{w}^*$ to write the new equations as:

$$X\mathbf{w}' = \mathbf{0} \tag{5}$$

What is the new initial condition for $\mathbf{w}'_0$?

(b) Next, let's leverage SVD coordinates to further simplify the problem. **Show that there exists an orthonormal transformation V of variables $\mathbf{w}'' = V\mathbf{w}'$ so that (5) looks like**

$$[\widetilde{X} \ \ \mathbf{0}_{n\times(d-n)}]\mathbf{w}'' = \mathbf{0} \tag{6}$$

and furthermore, **show that the initial condition for $\mathbf{w}'_0$ you computed in the previous part, when viewed as $\mathbf{w}''_0$ has all zeros in the final $(d-n)$ positions.**

(c) **Argue why what you have seen in the previous parts allows us to now focus on a square system of equations:**

$$\widetilde{X}\widetilde{w} = \mathbf{0} \tag{7}$$

and furthermore show that each of the n constituent equations (corresponding to rows) of (7) can be obtained by means of coordinate changes from the same indexed equation in (4).

(d) Let's now engage with SGD itself. Here, we will just use a minibatch length of 1 and batch sampling with replacement. This means that at every iteration of SGD, we roll a fair n-sided die and choose the single equation in (4) that corresponds to the row that came up on the die. Let I_t be the iid uniform random variable on $\{1, \dots, n\}$ that we roll after iteration t.

At the $t+1$-th iteration, we compute

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla \mathcal{L}_{I_t}(\mathbf{w}_t) \tag{8}$$

where $\mathcal{L}_i(\mathbf{w}) = (y[i] - \mathbf{x}_i^\top \mathbf{w})^2$ is the squared loss on the i-th equation and η is the step-size (learning rate).

Show that an SGD step taken in (8) for the original optimization problem matches exactly to an SGD step taken for $\widetilde{w}$ for solving (7), and that in particular these steps look like:

$$\widetilde{w}_{t+1} = \widetilde{w}_t - 2\eta \widetilde{x}_{I_t} \widetilde{x}_{I_t}^\top \widetilde{w}_t \tag{9}$$

(e) At this point, we can focus entirely on the simplified square system (7) and the stochastic evolution of the iterations described by (9).

To show convergence to zero, we need to pick a suitable stochastic Lyapunov function $\mathcal{L}(\widetilde{w})$ that is bounded below by zero and will decrease in expectation at every time step. In particular, we want to establish

$$E[\mathcal{L}(\widetilde{w}_{t+1}) | \widetilde{w}_t] < (1-\rho)\mathcal{L}(\widetilde{w}_t) \tag{10}$$

with a $1 > \rho > 0$ so that this Lyapunov function tends to decrease exponentially to zero. We will have to have a suitably small step-size/learning-rate η for this to happen, of course.

Show that (10) indeed implies that for every $\epsilon > 0$ and $\delta > 0$, there exists a $T > 0$ for which

$$P(\mathcal{L}(\widetilde{w}_T) < \epsilon) \geq 1 - \delta. \tag{11}$$

(f) One natural guess for a stochastic Lyapunov function is

$$\mathcal{L}(\widetilde{w}) = \widetilde{w}^\top \widetilde{X}^\top \widetilde{X} \widetilde{w}. \tag{12}$$

Argue why the candidate Lyapunov function $\mathcal{L}(\widetilde{w})$ in (12) is non-negative and is only equal to zero at $\widetilde{w} = \mathbf{0}$.

(g) Now, with a guessed stochastic Lyapunov function in hand, we can try to show (10). The first step will be to decompose the evolution of $\mathcal{L}(\widetilde{w})$ into three parts:

$$\mathcal{L}(\widetilde{w}_{t+1}) = \mathcal{L}(\widetilde{w}_t) + A + B \tag{13}$$

where the term A is linear in the actual stochastic update $(\widetilde{w}_{t+1} - \widetilde{w}_t)$ and the term B is quadratic in that update.

Expand out $\mathcal{L}(\widetilde{w}_t + (\widetilde{w}_{t+1} - \widetilde{w}_t))$ to give explicit forms for A and B.

(h) We are counting on the term A in (13) to give us actual contraction in expectation since this looks like a gradient-descent step. **Show:**

$$E[A|\widetilde{w}_t] \leq -c_1 \eta \mathcal{L}(\widetilde{w}_t) \tag{14}$$

where the $c_1 > 0$ is a positive constant that depends on the problem.

(Hint: you are going to want to leverage the actual updates in (9) *as well as the singular value structure for $\widetilde{X}$. Can the smallest singular value be zero?)*

(i) We need to make sure that the "quadratic" term B in (13) cannot undo the progress made by A in expectation. **Show:**

$$E[B|\widetilde{w}_t] \leq c_2 \eta^2 \mathcal{L}(\widetilde{w}_t) \tag{15}$$

where $c_2 > 0$ is another positive constant that depends on the problem.

(Hint: you are going to want to leverage the actual updates in (9)*, the singular value structure for $\widetilde{X}$, and the fact that the rows of $\widetilde{X}$ can only be so big. You will want to leverage the largest singular value of $\widetilde{X}$ for one bounding step and then let β be the largest norm of the rows of $\widetilde{X}$ to do another bounding step.)*

(j) Finally, we can put the pieces together to see that

$$E[\mathcal{L}(\widetilde{w}_{t+1})|\widetilde{w}_t] \leq (1 - c_1\eta + c_2\eta^2)\mathcal{L}(\widetilde{w}_t) \tag{16}$$

where $c_1 > 0$ and $c_2 > 0$ as well. **Show that this means that there exists a small enough η so that $1 - c_1\eta + c_2\eta^2 < 1$.**

(k) In earlier problem set, you saw how you could reinterpret ridge-regression using feature-aumentation. The earlier parts of this problem have now established that leveraging that trick, you can get SGD to converge exponentially for ridge regression. Check out Jupyter notebook in this **demo link**, and **report what you observed in terms of the convergence rate.**

One of the lessons that you will observe from the code is that the implementation details matter. If you do ridge regression and just treat it as an optimization problem, you won't just be able to use SGD and get exponential convergence with a constant step size. (You would have to adjust the step sizes to make them smaller, but this would slow down your convergence considerably.) But if you intelligently use the feature-aumentation perspective on ridge regression, you'll get exponential convergence.

This is why it is vital for people in EECS to really understand machine learning at the level of detail that we are teaching you. Because in the real world, even if you are a practicing machine learning engineer, if you are working on cutting-edge systems, you need to understand how to implement what you want to do so that it works fast. Equivalent formulations mathematically need not be equivalent from the point of view of implementation – this is one dramatic example of a case when they are not. Take EE227C and beyond if you want to understand these things more deeply.

9. 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:

- Saagar Sanghavi.
- Suhong Moon.
- Kumar Krishna Agrawal.
- Romil Bhardwaj.
- Jerome Quenum.
- Olivia Watkins.
- Anant Sahai.
- Anrui Gu.
- Matthew Lacayo.
- Past EECS 282 and 227 Staff.
- Peter Wang.
- Long He.
- Yaodong Yu.