Randomization/Approximation in ML Optimization and Randomized Pruning

Warren Shepard Sri Korandla Tushar Aggarwal

18 August 2024

Introduction & Motivation

It is no secret that many machine learning algorithms, particularly neural networks and deep learning, are very computationally expensive (in both time and space). Additionally, many popular machine learning packages including scikitlearn and tensorflow do not implement the most optimized versions of many machine learning algorithms. Their implementation may seem fast for most purposes because they are compiled in C rather than python (which is much faster), however, for larger scale datasets it would be beneficial to use more efficient algorithms. Additionally, we notice a flaw in the current approach to machine learning. Machine learning itself is an approximation problem. However, we are often obsessed with *perfect* prediction algorithms which never make computational errors. But, what if, like in the real world, we allow some room for some error? In some cases, this may lead to better (and more realistic) predictions and less over-fitting of models. To summarize, our motivations for using randomized/approximation algorithms in ML are as follows:

- Optimize runtime of selected ML algorithms
- Optimize space of selected ML algorithms
- Improve prediction accuracy of selected ML algorithms.

Our project follows the same outline as this course. We start with very basic ML models in Part I and work our way up to neural networks in Part III.

Part I: k Nearest Neighbors (kNN) vs. k Approximate Nearest Neighbors (kANN)

Problem Background

k Nearest Neighbors is one of the simplest machine learning techniques. The problem is very simple:

kNN Problem: Given a set of vectors S in a data space and a similarity measure DIST (for example euclidean distance) find the k nearest ("most similar") neighbors to some input x.

Assuming the neighbors are labeled, the prediction of x's label is simply the label which the majority of it's k nearest neighbors have. The most simple algorithm for computing kNN is as follows:

```
1: procedure BRUTE-KNN(x, S, DIST, k = 1)
2:     # Compare x to every other point
3:     Compute DIST(x, s) ∀ s ∈ S
4:     return k points (s ∈ S) with smallest DIST values
```

Oftentimes, we are not just querying one point but rather many. For example, suppose we would like to query the k nearest neighbors of m points. Then the runtime is $O(nm)$ where $n = |S|$. In this problem, there exist more efficient ways to compute the kNN, however, they all suffer from the curse of dimensionality. For higher dimensions, the brute force algorithm is the most efficient solution.

However, faster algorithms are achievable if we allow some error: specifically, if we allow the k nearest neighbors returned by the algorithm to *approximate* [3] [1]. That is, they are usually the correct k nearest neighbors to the input, but there is a chance of error. In the case of error, the point/s returned by the algorithm are still "nearby" the input. [3][1].

One of the early solutions for approximate k Nearest Neighbors was via Locality Sensitive Hashing (LSH) [3]. Recall that a hash function $h : U \to [n]$ maps input from a universe U of cardinality N to a "hash table" of size n where n is (usually) a prime number. Hashing is considered a randomized algorithm, but not because the hash function itself is randomized. Instead, the hash function is chosen at random from a family (a collection) of similar hash functions. In "standard" hash function, any two non-equal inputs will have a *very* small chance of colliding (mapping to the same spot in the hash table), even if there is a very small difference between the inputs. In locality sensitive hash functions, however, inputs which are very "similar" to each other will have a high probability of colliding. In other words, close points will have a high probability of colliding, which can help approximate k nearest neighbors. This is because we would only need to process the dataset once to create several hash tables (several are computed to boost accuracy since the process is randomized) to store the "hashes" (result of $h(s)$ where s is the point) of each point. Then, for a given query s, we can compute its hash and look up its nearby neighbors in our hash tables. A visualization of this is shown below with a single hash table.

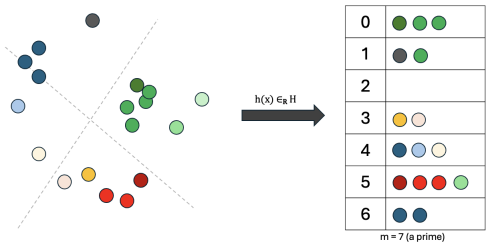

Figure 1: LSH Processing

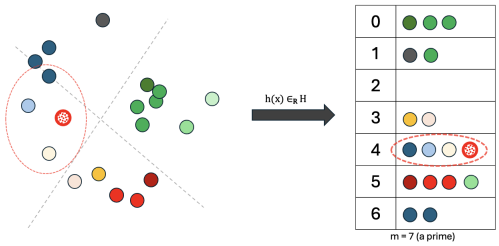

Figure 2: LSH Query

Methodology

We compared kNN to kANN in python. For the kNN, we implemented BRUTE-KNN in python rather than using a package such as scikitlearn. The reason for that is becuase kNN algorithms in popular packages like scikitlearn are compiled in C, and we would like to have everything in the same language for accurate and fair comparison. For our the kANN algorithm, we use the pynndescent package [6] (which is written in python) and whose implementation follows *Dong et al.*[1].

Results

Since kNN/kANN are primarily subroutines for the methods described in Part II, our testing was inextensive but enough to prove they both work and that the approximation algorithm for kANN is correct. Hence, we just used the following synthetic dataset:

```
# Generate a synthetic dataset with 10,000 samples and 1,000 features
X, y = make_classification(n_samples=10000, n_features=200, n_informative=50,
    n_classes=3, random_state=42)
```

Indeed, we see that the kANN algorithm is at least 3 times faster than the naive with comparable prediction accuracy. Note that $\sim 90\%$ of it's approximate k nearest neighbors are indeed the correct nearest neighbors. The results are shown below.

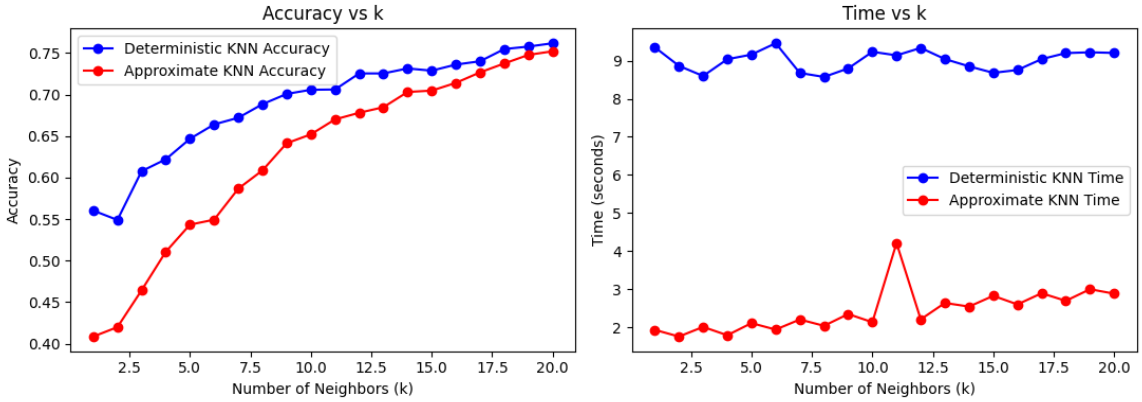

Figure 3: kNN vs kANN accuracy and time comparison

Part II: Graph Based Semi-Supervised Learning

Semi-Supervised Learning

We know in supervised learning we train our model on *only* labeled data whereas in unsupervised learning models are trained on *only* unlabeled data. As one would assume, semi-supervised learning is in between, were models are trained on a combination of labeled or unlabeled data. For example, one could use the labeled points to label the unlabeled points, then train the model on the entire set of (now 100% labeled) data. In this section, we will focus on the first step in that example: using a small set of unlabeled data to label a much larger set of unlabeled data.

Label Propagation Algorithm (LPA) on Nearest Neighbor Graphs

Again, suppose that you have a set of data in some feature-space where only a small subset ($\sim 10\%$) is labeled. The rest is unlabeled. One way of labeling the unlabeled points is to use a Label Propagation Algorithm (LPA) on a Nearest Neighbor (NN) Graph. First, one constructs an NN-graph where the vertices are defined by the union of the labeled and unlabeled nodes. Edges connect each vertex to its k nearest neighbors. This can easily be constructed using the algorithms in Part I as a subroutine. Then, the problem becomes "propagating" the labels such that every vertex is labeled. This can be visualized as similar to a flood fill algorithm where labeled vertices "spread" their labels to their neighbors, who then spread them to their neighbors in the next iteration, and so on until the entire graph is labeled. A nice schematic is shown below.

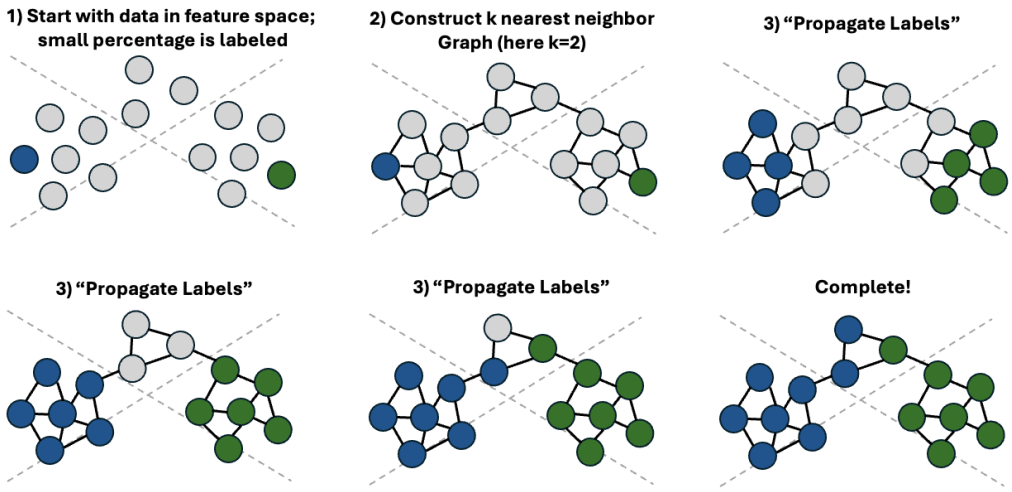

Figure 4: Label Propagation Algorithm Schematic

Of course, this process must be mathematically formalized in order to actually be implemented. To help, we'll define the following variables:

- $y[1:n]$ is label vector for n data points; -1 represents unlabeled data
- W is the adjacency matrix of the kNN/kANN graph
- t is the "current iteration"
- $t_{\max}$ is the total number of iterations
- α is called the "clamping factor" indicating influence of initial labels **per iteration**.

Let $F^{(t)}$ be the "confidence matrix" where F_{ic} is the confidence (a higher number means more confidence) of vertex i having label c. We initialize $F_{ic}^{(0)}$ as $F_{ic}^{(0)} \leftarrow \begin{cases} 1 & \text{if } y_i = c \\ 0 & \text{otherwise} \end{cases}$. Then, we update using the following equaiton:

$$F_{ic}^{(t+1)} \leftarrow \underbrace{\alpha W F^{(t)}}_{\text{propagate from neighbors}} + \underbrace{(1-\alpha)F^{(0)}}_{\text{account for initial labels}} . \tag{1}$$

Once we reach $t_{\max}$, the final label vector is given by $y_i^* \leftarrow \arg\max_c F_{ic}^{t_{\max}}$. The above process can be organized into the short algorithm below, which can easily be transferred to python.

procedure PROPAGATE-LABELS($y, G, t_{\max}, \alpha$)
    # $y[1:n]$ is label vector for n data points; -1 represents unlabeled data
    # $G = (V, E)$ is graph derived by kNN/kANN algorithm
    # $t_{\max}$ is the number of iterations
    # $\alpha \in [0, 1]$ is the clamping factor
    $W \leftarrow$ adjacency matrix representation of G
    Initialize label confidence matrix F # F_{ic} is confidence point i is labeled as c
    Initialize $F_{ic}^{(0)} \leftarrow \begin{cases} 1 & \text{if } y_i = c \\ 0 & \text{otherwise} \end{cases}$
    for $0 \leq t < t_{\max}$ **do**
        $$F_{ic}^{(t+1)} \leftarrow \underbrace{\alpha W F^{(t)}}_{\text{propagate from neighbors}} + \underbrace{(1-\alpha) F^{(0)}}_{\text{account for initial labels}}$$
    $y_i^* \leftarrow \arg\max_c F_{ic}^{t_{\max}}$ # compute final labels
    return y^*

Methodology

Many packages implement the NN-graph construction Label Propagation Algorithm as one function. Since we wanted to test two different NN-graph construction methods (using BRUTE-KNN and the approximated kNN from pynndescent), we implemented PROPAGATE-LABELS from scratch.

Results

First, we work with a toy dataset to demonstrate the effectiveness of the learning pipeline described above. The results of our pipeline are shown in Figures 5, 6, and 7.

```
# Generate a synthetic dataset with n_samples=100 samples and noise=0.075
X, y = make_moons(n_samples=100, noise=0.075)
# we also mask the data such that only 15% is labeled
```

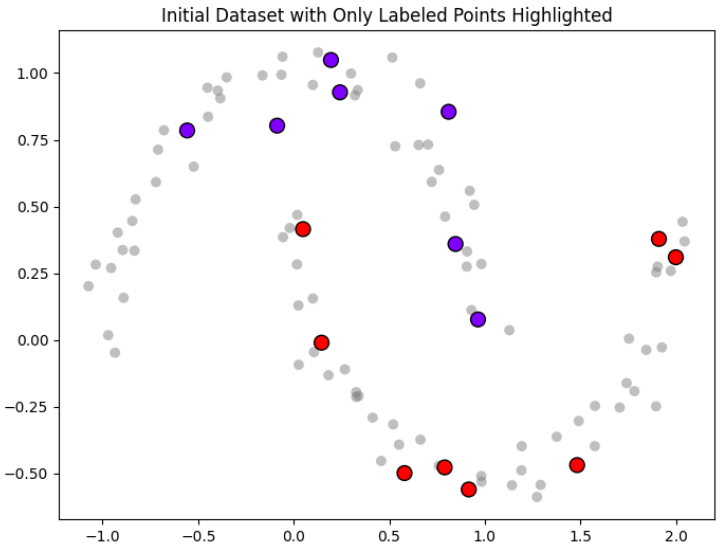

Figure 5: Initial Dataset with n=100 samples and 15% labeled

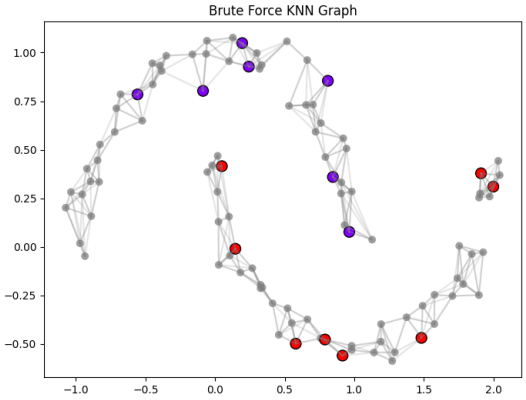

Figure 6: brute NN-graph; k=5

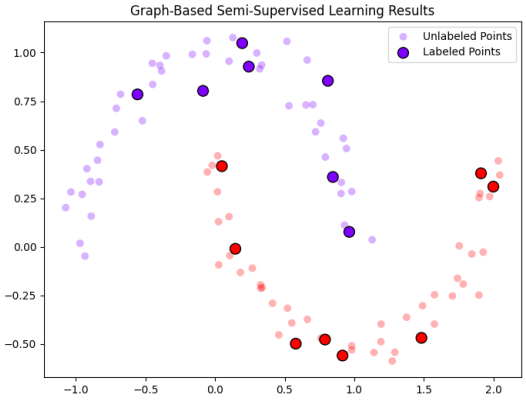

Figure 7: labels post LPA

Using a slightly larger, but still synthetic dataset, we get the following results:

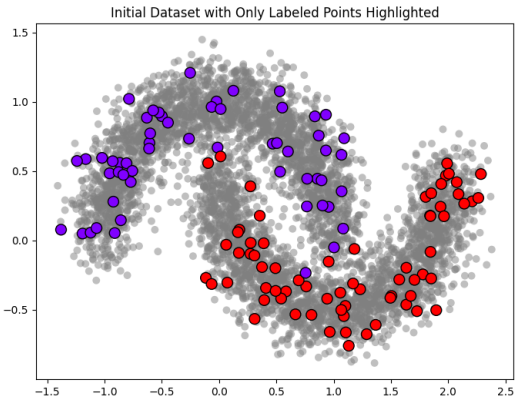

Figure 8: Initial Dataset with n=5000 samples and 2.5% labeled

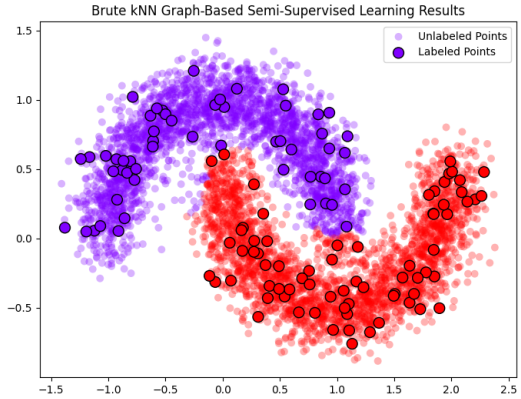

Figure 9: Deterministic NN-graph; k=5

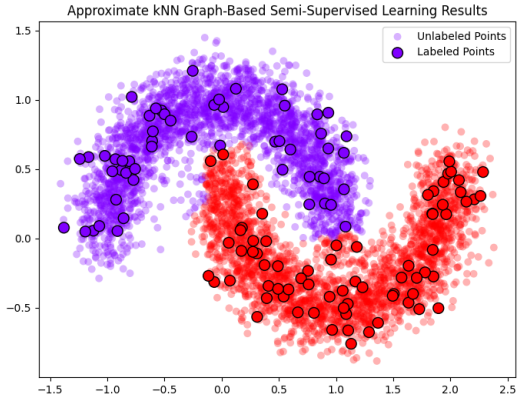

Figure 10: Approximate NN-graph; k=5

Method	Brute Force	k Approximate
Time	4.20 seconds	2.89 seconds
Accuracy	0.976	0.977

Table 1: Comparison of Brute Force kNN and k Approximate NN.

For larger scale testing, we used the MNIST dataset, which includes 60,000 data points with dimensionality 28^2. Of course, this is a fully labeled dataset. To simulate semi-supervised learning, we "masked" 90% of the data such that their labels were not visible to the classification pipeline described above. The original labels were saved for accuracy comparison. Note that the data was pre-processed by selecting the first 50 principle components. An example of the (unprocessed) data and the first two principle components are shown below.

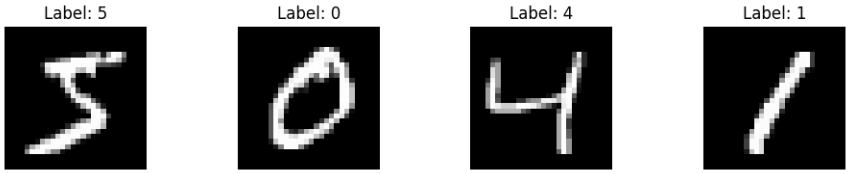

Figure 11: First 4 items in the MNIST training set

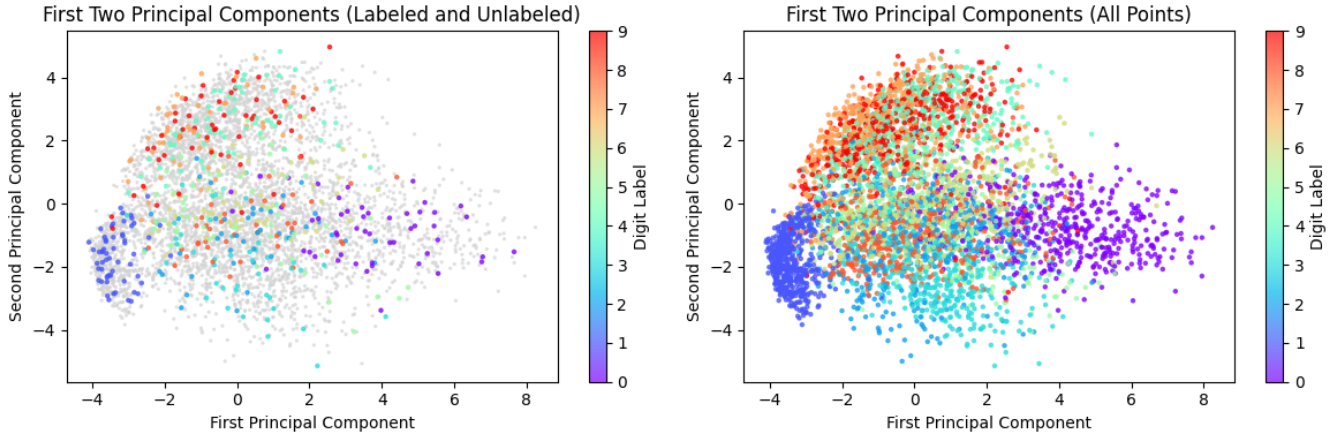

Figure 12: First two principle components; masked (left) and unmasked (right)

The results of label propagation using both methods are shown in Table 2.

Method	Accuracy	Time
Brute Force kNN	0.933	18.81 minutes Time to construct kNN-graph: 17.14 minutes
k Approximate NN	0.955	1.79 minutes Time to construct kANN-graph: 6.2 seconds

Table 2: Comparison of Brute Force kNN and k Approximate NN methods.

Clearly, the approximation algorithm for LSH is **much** faster (it even beats the C implementation of the naive algorithm), and as an added bonus, more accurate! Naturally, we notice that the Label Propagation Algorithm is dominating the runtime, and if we wanted to further optimize this problem, that would be our next area of focus. Indeed, we do that in the next subsection.

The Problem With our LPA Implementation

Our original LPA implementation definitely has it's inefficiencies. They can be summarized in the below list:

- In theory, LPA can be parallelized
  - instead of executing **sequentially**, tasks are executed **simultaneously** on multiple units (i.e. CPU/GPU)
- No early stopping
  - the number of iterations is a parameter
  - stop early if there is no/little change between two consecutive iterations
- ~~Not leveraging sparse matrices~~
  - MNIST PCA is not sparse!

In practice, parallelization was not helpful. However, using early stopping, we were able to get the total time down to 16 seconds, which is a 10 times increase in LPA speed. As an added bonus, this increased the accuracy to 96.6%, implying the original model in Table 1 may have been over fit.

Part III: Pruning in Neural Networks & Randomized Pruning Masks

Pruning Background

Pruning is defined as reducing the size/complexity of neural networks by removing less important neurons, weights, layers, or connections. It is well known that large and sparse models are more accurate than small and dense models. [4]. Hence, there is motivation to train large and dense models then prune them to get a large but sparse model. Additionally, pruned models require less memory than unpruned models, which make them much more desirable. By eliminating unnecessary components of the model, pruning also helps reduce over-fitting.

Pruning is inspired by the brain, where pruning neural connections occurs in every human being [2] (see Figure 13). If our goal in making neural networks is to "mimic the brain", it then makes sense to prune neural networks as well. Issues with pruning (too much/too little/pruning the wrong connections), have been linked to conditions including autism and schizophrenia [2], so it is important to carefully analyze the effectiveness of different pruning methods.

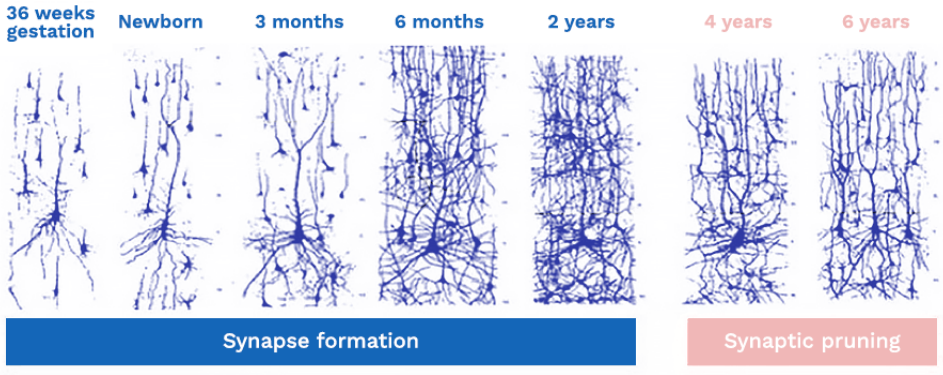

Figure 13: Synaptic pruning in the brain [2]

The most basic method of pruning would be to set a threshold for the weight of a connection and prune any connection below that threshold. There also exist more complex ways of deterministic pruning [5] for larger

models, and recent research has suggested that randomized pruning methods (also used in large models) could be even better than their deterministic counterparts [4].

Methodology

Given our available resources, it would be very hard to train larger neural networks. Hence, for the purposes of this project, we stick to smaller neural networks. For our dataset, we will use the MNIST Fashion dataset, which is similar to the standard MNIST dataset except that it contains images of clothing instead of handwritten digits. An example is shown below:

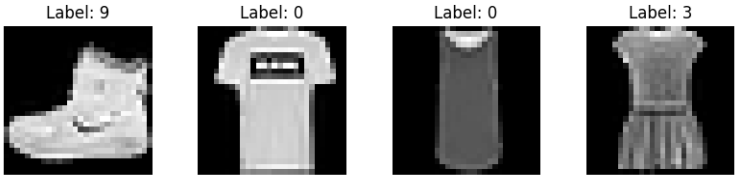

Figure 14: First 4 items in the MNIST Fashion training set

For motivation, we analyzed the code provided in the tensorflow pruning guide [7]. Then, we created five neural networks with identical structures and different pruning strategies. Each network was also trained in the same way. These networks are summarized in the Table 3 and the model architecture is outlined in Table 4. Note that we chose to go with an incredibly simple model architecture because the focus of this section is **evaluate different pruning methods**, particularly randomized pruning methods.

Network	**Pruning Strategy**	**Accuracy**
No Pruning	No pruning is applied. All connections are retained.	87.68%
Threshold Fraction Pruning	Prune a specified fraction of connections with the smallest weights.	86.63%
Random Pruning	Prune a random subset of connections, regardless of their weights or importance.	64.56%
Importance-Weighted Random Pruning	Prune connections probabilistically based on their importance, where more important connections are less likely to be pruned.	78.06%
Fractional Random Threshold Pruning	First, select a fraction of connections with the smallest weights. Then, randomly prune a smaller fraction of these selected connections.	87.47%

Table 3: Summary of Pruning Strategies for Five Neural Networks

Layer Type	**Output Shape**	**Parameters**
Input Layer	(28, 28)	0
Reshape Layer	(28, 28, 1)	0
Layer from Table 3	(26, 26, 12)	Conv2D: 120
MaxPooling2D	(13, 13, 12)	0
Flatten	(2028)	0
Dense	(10)	20,290

Table 4: Neural Network Summary Table

Learning Outcomes

Warren: Through this project I learned a lot more about how approximate kNN algorithms are implemented in practice and how they can be optimized. I also learned more about semi-supervised learning and some algorithms for graph label propagation as well as how to optimize them. Additionally, I got more practice working with neural networks and playing around with their parameters and architecture. This is especially helpful for me because I will defiantly be taking more machine learning classes in the future and neural networks are fundamental to know for those types of classes, especially in python. Believe it or not, I did a lot of the coding for this project and also probably became a marginally better programmer.

Sri: This project was an opportunity to get better experience with writing neural networks, and exploring optimization problems in computer science. The implementation of approximate KNN using the pynndescent package was an opportunity to reinforce understanding and gain a practical appreciation for the trade-offs between computational efficiency and accuracy in large-scale, high-dimensional datasets. Graph labeling was interesting to explore, as it was something added late to the project and was not something covered during lectures. Exploration of graph-based machine learning approaches provided valuable insights into the practical challenges of working with graph algorithms and their application in real-world data scenarios.

Tushar: This project helped me to better understand different approaches to optimization problems in computer science. Comparing the performance of the simpler KNN algorithm with the approximate KNN algorithm helped me to better understand the relationship between test accuracy and overall efficiency, which became more apparent in larger datasets with higher dimensional feature spaces. By expanding the concept of approximation to a semi-supervised learning via graph labeling, I was able to better appreciate the value and improvement gained in efficiency from approximation. Understanding these two parts ultimately improved my intuition to understand how pruning impacts both the accuracy and efficiency of a neural network.

Individual Contributions

- **Warren** wrote most of the code, particularly the algorithms. He also made the presentation slides and did part of the write up.
- **Sri** wrote code pertaining to neural networks, code review + documentation, as well as this beautiful write up!
- **Tushar** created synthetic and "real" datasets and tested the functions that Warren wrote using them

References

[1] Wei Dong, Moses Charikar, and Kai Li. "Efficient k-nearest neighbor graph construction for generic similarity measures". In: Proceedings of the 20th international conference on World wide web. ACM. 2011, pp. 577–586.

[2] Joanna Embrace. "Synaptic Growth, Synesthesia, and Savant Abilities". In: Embrace Autism (2024). Accessed: 2024-08-12. URL: https://embrace-autism.com/synaptic-growth-synesthesia-and-savant-abilities/#Synaptic_pruning.

[3] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. "Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality". In: Theory of Computing 8 (2012). Special issue in honor of Rajeev Motwani, pp. 321–350. DOI: 10.4086/toc.2012.v008a014. URL: https://www.theoryofcomputing.org/articles/v008a014.

[4] Jianwei Li et al. "Breaking through Deterministic Barriers: Randomized Pruning Mask Generation and Selection". In: arXiv preprint arXiv:2310.13183 (2023). Accessed: 2024-08-14. URL: https://arxiv.org/abs/2310.13183.

[5] Jianwei Li et al. "Towards Robust Pruning: An Adaptive Knowledge-Retention Pruning Strategy for Language Models". In: arXiv preprint arXiv:2310.13191 (2023). Accessed: 2024-08-14. URL: https://arxiv.org/abs/2310.13191.

[6] PyNNDescent Developers. PyNNDescent Documentation. Accessed: 2024-08-14. 2024. URL: https://pynndescent.readthedocs.io/en/latest/how_to_use_pynndescent.html#Query-parameters.

[7] TensorFlow Model Optimization Team. Pruning with Keras. https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras. Accessed: 2024-08-18. 2023.