CS 484 Final Project Description

Alexander Wei, a6wei@uwaterloo.ca, 20836214 Kaiz Nanji, k4nanji@uwaterloo.ca 20873846

Project #5

Semi-supervised image classification: assuming that only M out of N images in the traingin data have ground truth labels, design and implement a weakly supervsied training of classification network that can benefit from unlabeled examples in the training dataset(e.g. MNIST or CIFAR-10, but you need to ignore labels on a subset of training examples).You should demonstrate how the performance changes as M gets progressively smaller. While you can use any well-motivated ideas, one basic approach could be to combine cross-entropy on labeled points with (unsupervised) K-means clustering loss over deep features (e.g. in the last layer before the linear classifier). It is also advisable to use augmentation (a loss enforcing consistent labeling of augmented training examples). You can also explore Mutual Information loss function formulated in Bridle & MacKay "Unsupervised Classifiers, Mutual Information and Phantom Targets", NIPS 1991.

Abstract

This project tackles the challenge of semi-supervised image classification, aiming to leverage the power of unlabeled data alongside a limited set of labeled examples. We focus on the MNIST handwritten digit classification task, where only a portion of the training images possess ground truth labels. To achieve this, we propose a methodology that combines K-means clustering with an encoder model.

The auto-encoder neural network is an unsupervised learning technique that reduces the dimensionality of the data while attempting to extract relevant features from the data. The initial dimensionality of the MNIST dataset is 1*28*28 = 784, and the auto-encoder outputs a dimensionality of 64*2*2 = 256. The model is optimized against pixel-wise Mean Squared Error (MSE) loss. For this project, varying amounts of the total training data is used to train the auto-encoder. After training the auto-encoder, both the labeled and unlabeled dataset images are encoded.

The encodings of the labeled training dataset are used to determine the initial centroids of the K-Means classifier. The hope is that a well-trained auto-encoder is able to successfully separate the encoded dataset into distinct, non-overlapping clusters. Ideally, the Euclidean distance between the encodings of two different digits is large, and the Euclidean distance between the encoding of the same digit is small. The K-Means classifier is then fitted with all labeled and unlabeled encoded data points.

By effectively combining these supervised and unsupervised learning components, our approach aims to improve classification performance even when labeled data is scarce.

Contributions

Alexander Wei (a6wei@uwaterloo.ca):

Built the MNIST downloader wrapper. Built the autoencoder neural network, classifier neural network, encoder_classifier neural network, K-Means model wrapper. Built the train/validate script `run.py`.

Kaiz Nanji (k4nanji@uwaterloo.ca):

Built the MNIST data loader to effectively seperate labelled and unlabelled data. Built the ClassificationNetwork class to experiment K-means model with the encoder. Wrote the abstract and conclusion for the project.

Code Libraries

This project uses many open-source code packages and are essential to the success of the project. The use cases for the most critical packages will be briefly acknowledged below. Please see `requirements.txt` and `requirements_cpu.txt` for a full list of required Python packages.

numpy: This package is used for efficient vector and matrix mathematical operations.

matplotlib: This package provides an interface for plotting graphs. We used this to plot the loss rate of the model over training epochs.

scikit-learn: This package provides a fully implemented K-Means classification model, which we use for the final output of the project's network. It also provides a plotting interface for the confusion matrix.

pytorch: This package provides an interface for downloading the MNIST dataset, the dataset we used to train and validate our models. It also provides all of the needed neural network infrastructure, including but not limited to convolution layers, backpropagation algorithms, optimizers, and loss functions.

In [ ]:

```
from src.encoder import *
from src.k_means import *
from src.loader import *
from src.utils import *

import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import torch
from torch.utils.data import DataLoader
```

In [ ]:
```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print(device)
```

cuda

In [ ]:
```
set_seed()

transform = get_transform()

class ClassificationNetwork:
    def __init__(self, labeled_percent):
        self.labeled_percent = labeled_percent
        self.labeled_train_loader, self.unlabeled_train_loader, self.val_loader = self.get_loaders()

    def get_loaders(self):
        # Increase TRAIN_BATCH_SIZE if you are using GPU to speed up training.
        # When batch size changes, the learning rate may also need to be adjusted.
        # Note that batch size maybe limited by your GPU memory, so adjust if you get "run out of GPU memory" error.
        TRAIN_BATCH_SIZE = 100

        # If you are NOT using Windows, set NUM_WORKERS to anything you want, e.g. NUM_WORKERS = 4,
        # but Windows has issues with multi-process dataloaders, so NUM_WORKERS must be 0 for Windows.
        NUM_WORKERS = 0

        dataset_handler = LabeledUnlabeledMNIST(self.labeled_percent)
        labeled_train = dataset_handler.labeled_dataset
        unlabeled_train = dataset_handler.unlabeled_dataset
        mnist_test = dataset_handler.mnist_test

        self.VALIDATION_SIZE = len(mnist_test)

        labeled_train_loader = DataLoader(labeled_train, batch_size=TRAIN_BATCH_SIZE, num_workers=NUM_WORKERS, shuffle=False)
        unlabeled_train_loader = DataLoader(unlabeled_train, batch_size=TRAIN_BATCH_SIZE, num_workers=NUM_WORKERS, shuffle=False)
        val_loader = DataLoader(mnist_test, batch_size=len(mnist_test), num_workers=NUM_WORKERS, shuffle=False)

        return labeled_train_loader, unlabeled_train_loader, val_loader

    def train(self, epochs = 1, train_with_all_labeled = False):
        if train_with_all_labeled:
            %run -i "run.py" encoder 1 -t -e "{epochs}"
        else:
            %run -i "run.py" encoder "{self.labeled_percent}" -t -e "{epochs}"

    def get_encodes_targets(self, loader, encoder_model):
        encodes = None
        targets = None
```

```
        with torch.no_grad():
            for batch_id, (data, target) in enumerate(loader):
                data = data.to(device)
                encoded = encoder_model.encode(data)

                if batch_id == 0:
                    encodes = encoded
                    targets = target
                else:
                    encodes = torch.cat((encodes, encoded))
                    targets = torch.cat((targets, target))

        return encodes, targets

    def run_kmeans(self):
        k_means = KMeansModel(tensor_dims=256)

        encoder_model = EncoderModel(device, None)
        encoder_model.load_state_dict(torch.load("./saves/encoder_model_{}.pth".format(str(self.labeled_percent)[2:])))
        encoder_model = encoder_model.to(device)

        encoder_model.train(False)

        labeled_encodes, labeled_targets = self.get_encodes_targets(self.labeled_train_loader, encoder_model)
        unlabeled_encodes, unlabeled_targets = self.get_encodes_targets(self.unlabeled_train_loader, encoder_model)

        k_means.fit(labeled_encodes, labeled_targets, unlabeled_encodes)

        val_data, val_labels = next(iter(self.val_loader))
        encoded_val_data = encoder_model.encode(val_data.to(device))

        predictions = k_means.predict(encoded_val_data)

        confusion_matrix = np.zeros((10, 10))
        for label, pred in zip(val_labels, predictions):
            confusion_matrix[label, pred] += 1

        accuracy = sum([confusion_matrix[i][i] for i in range(10)]) / self.VALIDATION_SIZE
        print(f"Accuracy: {accuracy * 100}%")

        disp = ConfusionMatrixDisplay(confusion_matrix=confusion_matrix, display_labels=[i for i in range(10)])
        disp.plot()
        plt.title("K Means Classifier Confusion Matrix")
        plt.show()
```

Important Notes For Readers

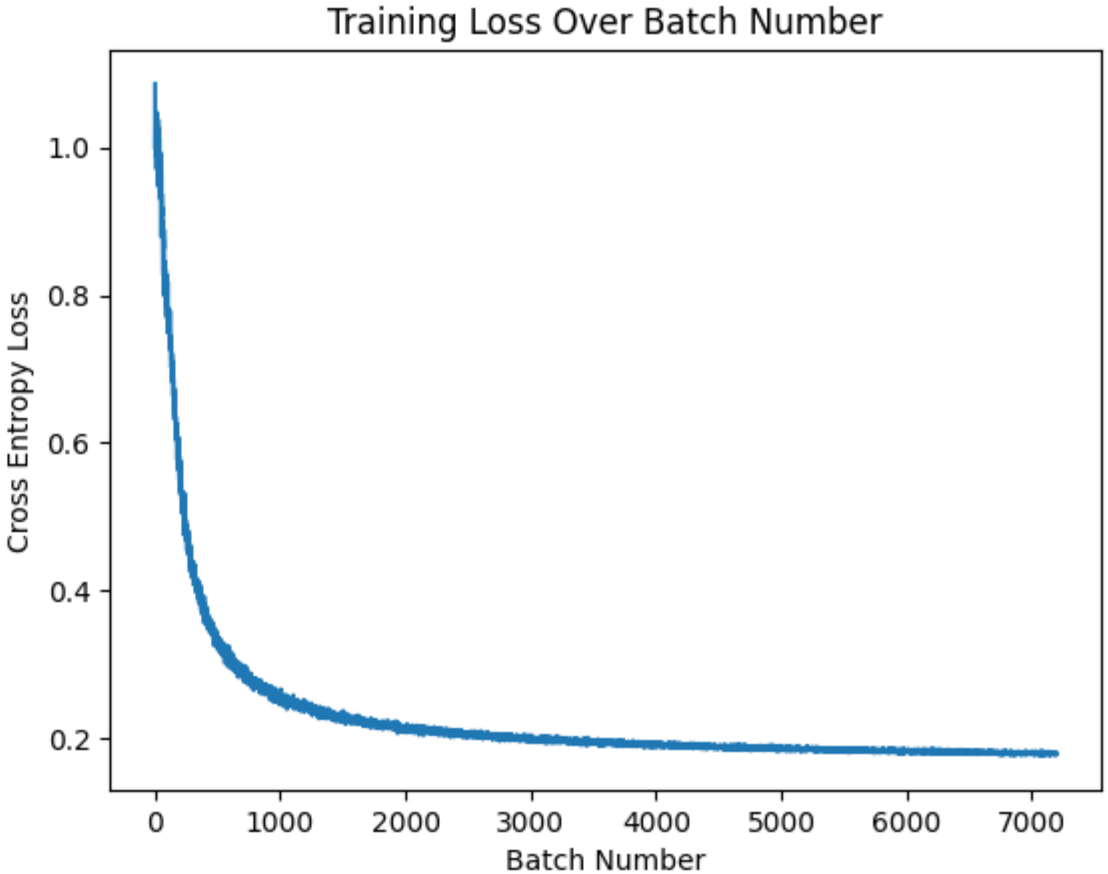
Occcasionally training the autoencoder fails because the loss diverges. This doesn't happen often and just try running it again a few times.

If you get a "no directory named saves found" error, please create that directory as a child to the root directory of this repo.

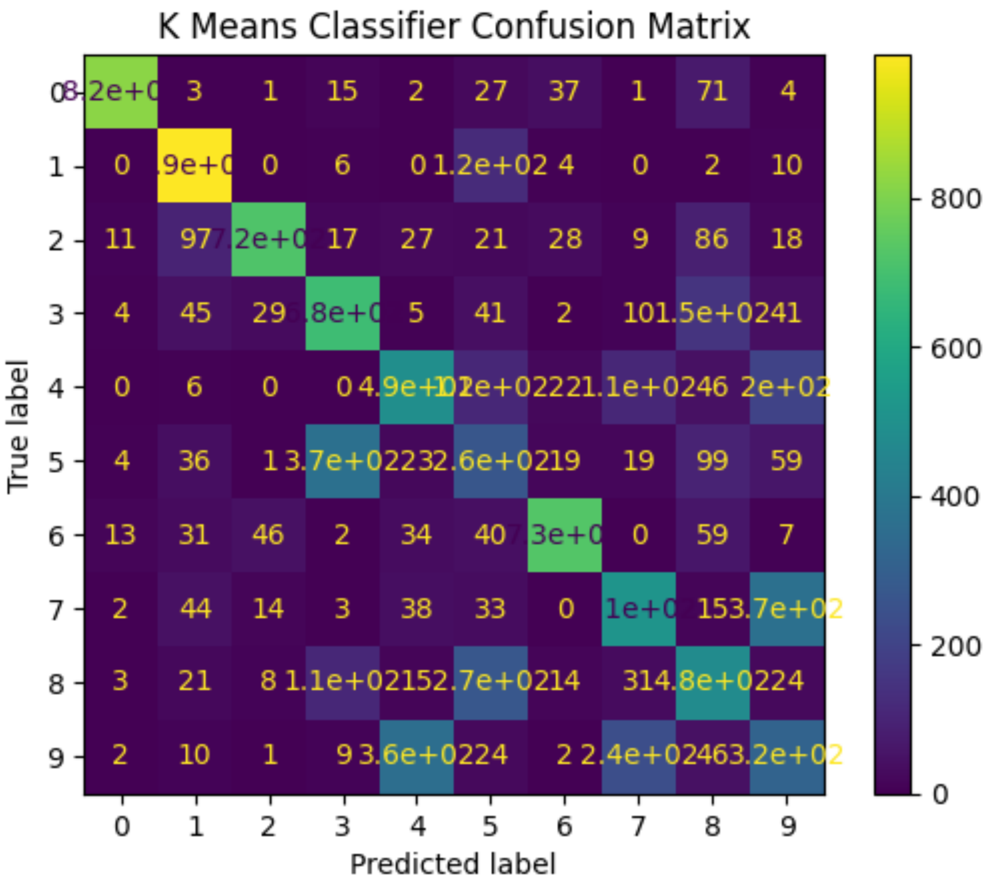
Calling `network.train()` calls the underlying `run.py` script. You may choose to directly execute `run.py` with your preferred arguments and run `network.train()` without saving the model. Additionally, the script will prompt the user for whether to save the model. In Visual Studio Code, a text box will appear at the top of the screen instead. If you do not see the text box, try invoking the script directly. When executing `run.py`, provide the input through the terminal window.

In [ ]:
```
network = ClassificationNetwork(0.8)
network.train(epochs=15)
network.run_kmeans()
```

```
cuda
Loss at epoch 0: 0.33091145753860474
Loss at epoch 1: 0.2560620903968811
Loss at epoch 2: 0.22782114148139954
Loss at epoch 3: 0.2162703275680542
Loss at epoch 4: 0.20473934710025787
Loss at epoch 5: 0.20091059803962708
Loss at epoch 6: 0.19371020793914795
Loss at epoch 7: 0.19449259340763092
Loss at epoch 8: 0.18702203035354614
Loss at epoch 9: 0.18526966869831085
Loss at epoch 10: 0.18813349306583405
Loss at epoch 11: 0.18245390057563782
Loss at epoch 12: 0.18067218363285065
Loss at epoch 13: 0.1819620132446289
Loss at epoch 14: 0.17978964745998383
Completed training! Final loss: 0.17978964745998383
Running validation...
```



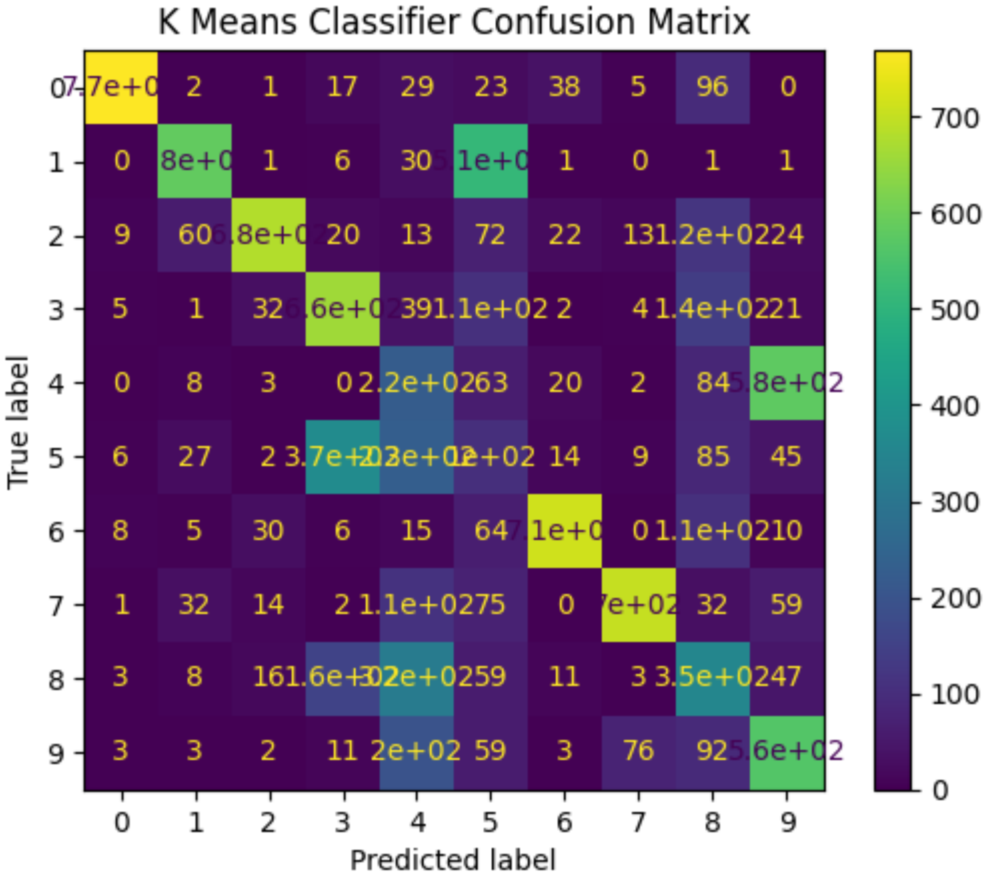

```
Saving model...
Accuracy: 59.98%
```



In [ ]:
```
network = ClassificationNetwork(0.5)
network.train(epochs=15)
network.run_kmeans()
```

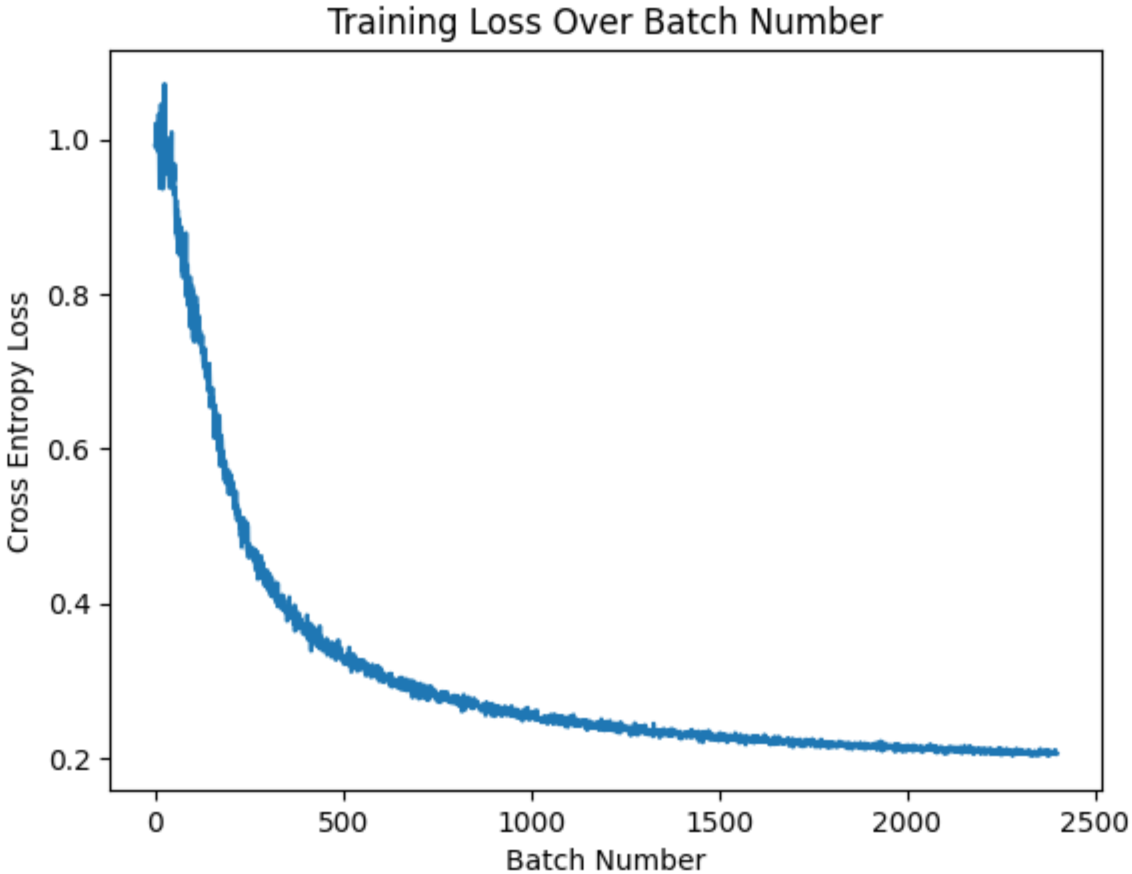
```
cuda
Loss at epoch 0: 0.42159831523895264
Loss at epoch 1: 0.30650103092193604
Loss at epoch 2: 0.26930445432662964
Loss at epoch 3: 0.24852947890758514
Loss at epoch 4: 0.22691136598587036
Loss at epoch 5: 0.22043336927890778
Loss at epoch 6: 0.21035896241664886
Loss at epoch 7: 0.20402202010154724
Loss at epoch 8: 0.2021605521440506
Loss at epoch 9: 0.20060840249061584
Loss at epoch 10: 0.1997266411781311
Loss at epoch 11: 0.19533437490463257
Loss at epoch 12: 0.19170932471752167
Loss at epoch 13: 0.18996092677116394
Loss at epoch 14: 0.19289632141590118
Completed training! Final loss: 0.19289632141590118
Running validation...
```


```
Saving model...
Accuracy: 53.5%
```

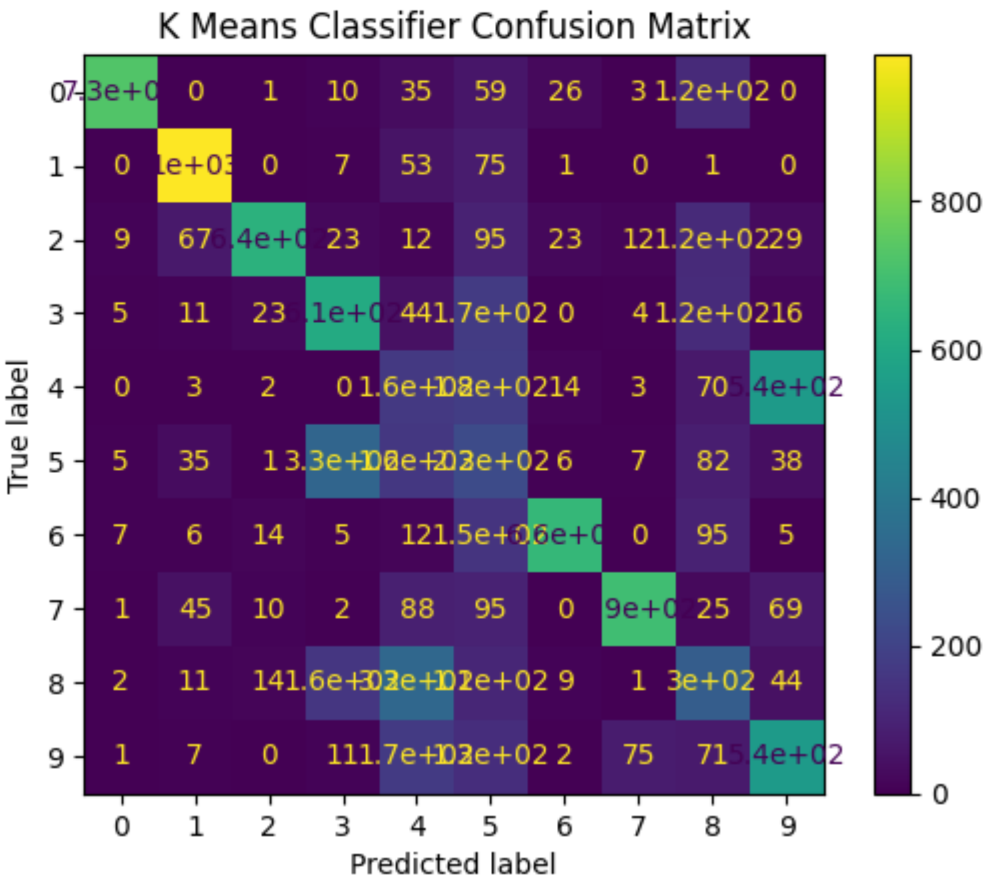



```
network = ClassificationNetwork(0.2)
network.train(epochs=20)
network.run_kmeans()
```

```
cuda
Loss at epoch 0: 0.7530671954154968
Loss at epoch 1: 0.48417922854423523
Loss at epoch 2: 0.3883611857891083
Loss at epoch 3: 0.3477868139743805
Loss at epoch 4: 0.3000766336917877
Loss at epoch 5: 0.28933629393577576
Loss at epoch 6: 0.2667415738105774
Loss at epoch 7: 0.256594181060791
Loss at epoch 8: 0.2509666085243225
Loss at epoch 9: 0.24210691452026367
Loss at epoch 10: 0.23342913389205933
Loss at epoch 11: 0.23142896592617035
Loss at epoch 12: 0.225728839635849
Loss at epoch 13: 0.22331027686595917
Loss at epoch 14: 0.21983568370342255
Loss at epoch 15: 0.2192114293575287
Loss at epoch 16: 0.21340824663639069
Loss at epoch 17: 0.20907363295555115
Loss at epoch 18: 0.20446403324604034
Loss at epoch 19: 0.20521394908428192
Completed training! Final loss: 0.20521394908428192
Running validation...
```




```
Saving model...
Accuracy: 55.67999999999999%
```

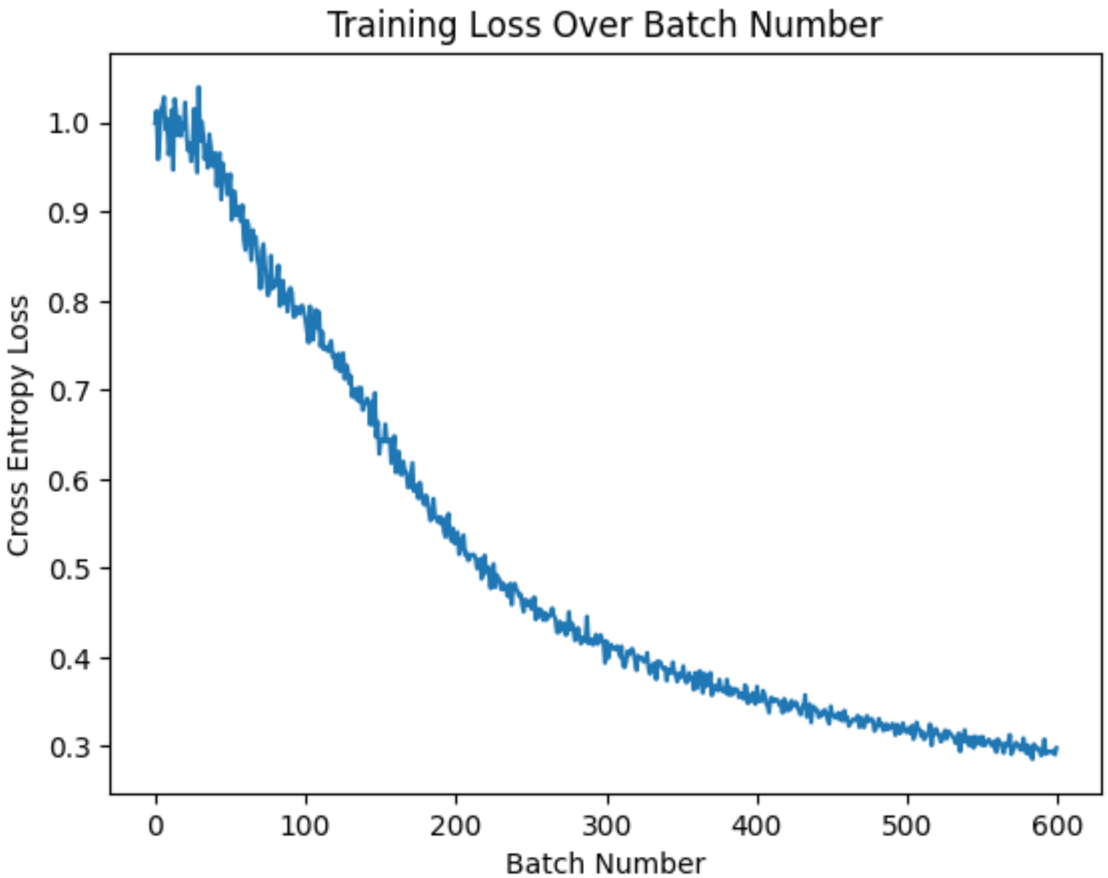



```
In [ ]: network = ClassificationNetwork(0.01)
        network.train(epochs=100)
        network.run_kmeans()
```

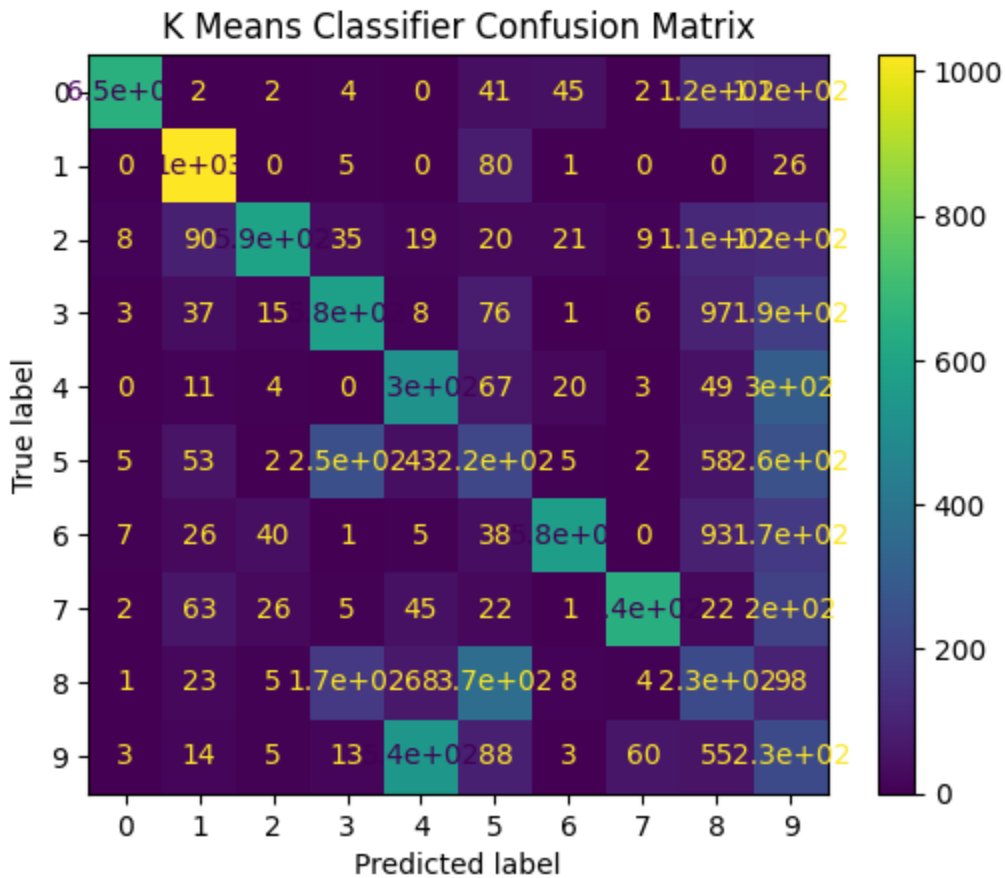
```
cuda
Loss at epoch 0: 1.0189646482467651
Loss at epoch 1: 1.0140256881713867
Loss at epoch 2: 0.9849298596382141
Loss at epoch 3: 0.9762744307518005
Loss at epoch 4: 1.0400704145431519
Loss at epoch 5: 0.9492043852806091
Loss at epoch 6: 0.9286838173866272
Loss at epoch 7: 0.9417235255241394
Loss at epoch 8: 0.9226093888282776
Loss at epoch 9: 0.8680927157402039
Loss at epoch 10: 0.8793146014213562
Loss at epoch 11: 0.8565666079521179
Loss at epoch 12: 0.8507202863693237
Loss at epoch 13: 0.7940434813499451
Loss at epoch 14: 0.8112183213233948
Loss at epoch 15: 0.7897891402244568
Loss at epoch 16: 0.7660057544708252
Loss at epoch 17: 0.7899115085601807
Loss at epoch 18: 0.7458482384681702
Loss at epoch 19: 0.7397868037223816
Loss at epoch 20: 0.7412845492362976
Loss at epoch 21: 0.6923472285270691
Loss at epoch 22: 0.7024216651916504
Loss at epoch 23: 0.660981297492981
Loss at epoch 24: 0.6281543970108032
Loss at epoch 25: 0.6440247297286987
Loss at epoch 26: 0.626072347164154
Loss at epoch 27: 0.6074645519256592
Loss at epoch 28: 0.5928282737731934
Loss at epoch 29: 0.5710073709487915
Loss at epoch 30: 0.5778322219848633
Loss at epoch 31: 0.5562012791633606
Loss at epoch 32: 0.5305103063583374
Loss at epoch 33: 0.5207403898239136
Loss at epoch 34: 0.5102388858795166
Loss at epoch 35: 0.5062177181243896
Loss at epoch 36: 0.5013231039047241
Loss at epoch 37: 0.49236008524894714
Loss at epoch 38: 0.4803142845630646
Loss at epoch 39: 0.4825390577316284
Loss at epoch 40: 0.45086878538131714
Loss at epoch 41: 0.45449772477149963
Loss at epoch 42: 0.45019376277923584
Loss at epoch 43: 0.45081084966659546
Loss at epoch 44: 0.4399627447128296
Loss at epoch 45: 0.45078274607658386
Loss at epoch 46: 0.4329163432121277
```

```
Loss at epoch 47: 0.44556909799575806
Loss at epoch 48: 0.42542973160743713
Loss at epoch 49: 0.3934899866580963
Loss at epoch 50: 0.4089652895927429
Loss at epoch 51: 0.39100250601768494
Loss at epoch 52: 0.40966105461120605
Loss at epoch 53: 0.3998723328113556
Loss at epoch 54: 0.38107505440711975
Loss at epoch 55: 0.39338618516921997
Loss at epoch 56: 0.3863826394081116
Loss at epoch 57: 0.37240099906921387
Loss at epoch 58: 0.3711874186992645
Loss at epoch 59: 0.3824707865715027
Loss at epoch 60: 0.3816240429878235
Loss at epoch 61: 0.36664748191833496
Loss at epoch 62: 0.36200395226478577
Loss at epoch 63: 0.35767862200737
Loss at epoch 64: 0.35948094725608826
Loss at epoch 65: 0.35648584365844727
Loss at epoch 66: 0.35030391812324524
Loss at epoch 67: 0.34534233808517456
Loss at epoch 68: 0.3526282012462616
Loss at epoch 69: 0.34926021099090576
Loss at epoch 70: 0.345447301864624
Loss at epoch 71: 0.34439024329185486
Loss at epoch 72: 0.3439176082611084
Loss at epoch 73: 0.3339451551437378
Loss at epoch 74: 0.34478893876075745
Loss at epoch 75: 0.3296072781085968
Loss at epoch 76: 0.3226950764656067
Loss at epoch 77: 0.33449989557266235
Loss at epoch 78: 0.3342174291610718
Loss at epoch 79: 0.3191961646080017
Loss at epoch 80: 0.3198850154876709
Loss at epoch 81: 0.3200540840625763
Loss at epoch 82: 0.3259633481502533
Loss at epoch 83: 0.3230980336666107
Loss at epoch 84: 0.3116162419319153
Loss at epoch 85: 0.3247646689414978
Loss at epoch 86: 0.3132062554359436
Loss at epoch 87: 0.3170585036277771
Loss at epoch 88: 0.3080790340900421
Loss at epoch 89: 0.30432382225990295
Loss at epoch 90: 0.3111153841018677
Loss at epoch 91: 0.3007469177246094
Loss at epoch 92: 0.3000059425830841
Loss at epoch 93: 0.2973717749118805
Loss at epoch 94: 0.29093635082244873
```

```
Loss at epoch 95: 0.295704185962677
Loss at epoch 96: 0.29062050580978394
Loss at epoch 97: 0.2980354130268097
Loss at epoch 98: 0.29382726550102234
Loss at epoch 99: 0.2981114387512207
Completed training! Final loss: 0.2981114387512207
Running validation...
```

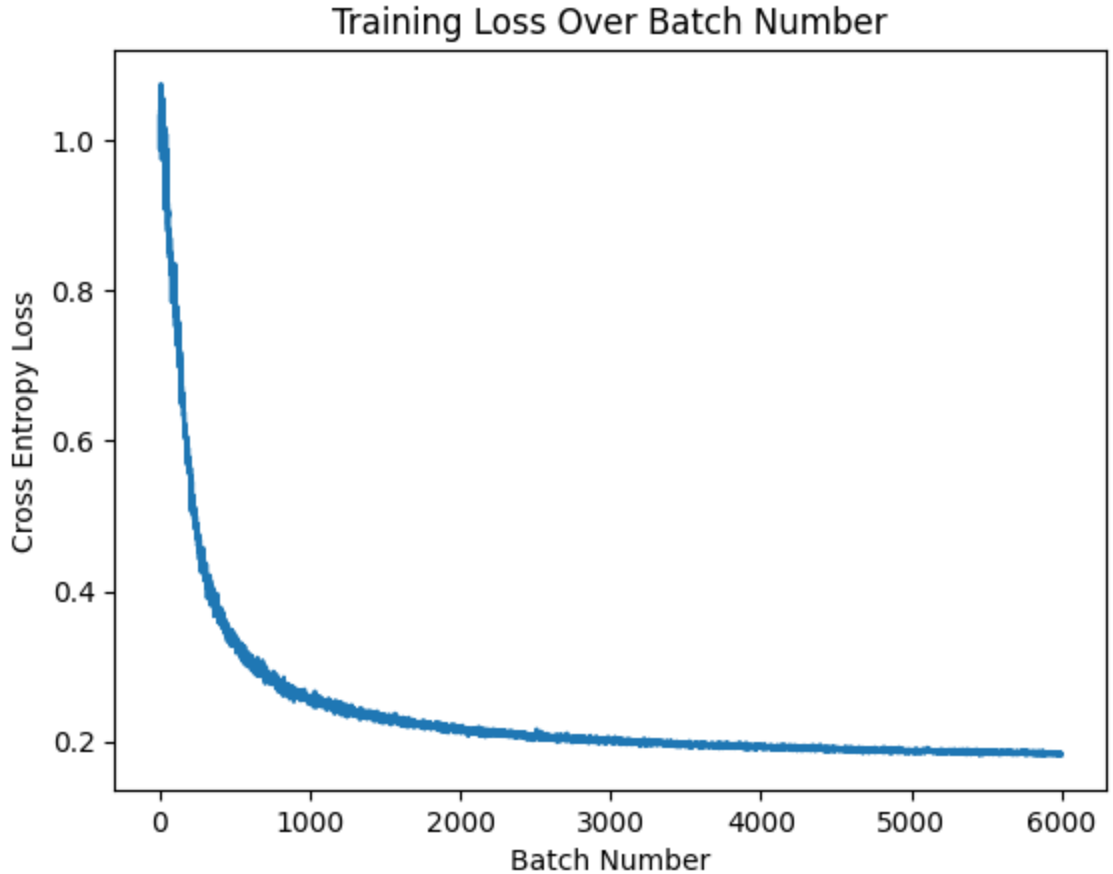



```
Saving model...
Accuracy: 52.61%
```

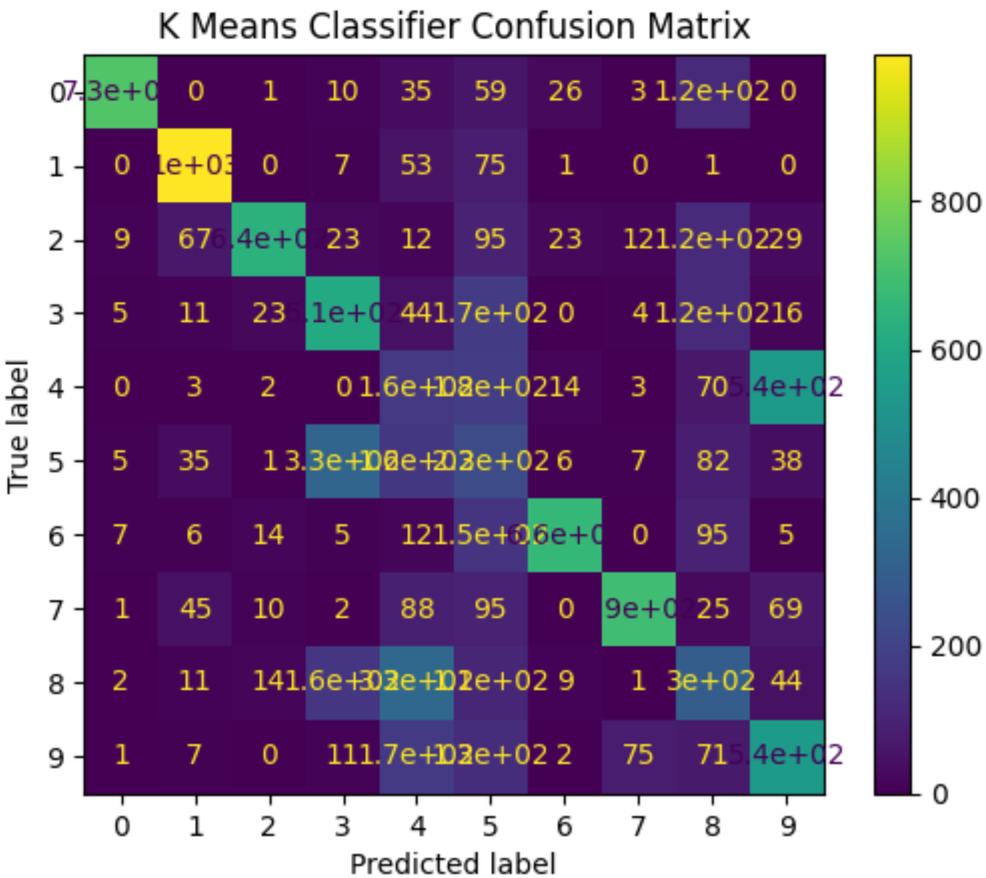


In [ ]:
```
# As an experiment, training the auto-encoder with the full training dataset
network = ClassificationNetwork(0.2)
network.train(epochs=10, train_with_all_labeled=True)
network.run_kmeans()
```

```
cuda
Loss at epoch 0: 0.3092639446258545
Loss at epoch 1: 0.24355114996433258
Loss at epoch 2: 0.2253483533859253
Loss at epoch 3: 0.21067193150520325
Loss at epoch 4: 0.2013758420944214
Loss at epoch 5: 0.19519703090190887
Loss at epoch 6: 0.19179058074951172
Loss at epoch 7: 0.18729203939437866
Loss at epoch 8: 0.18548555672168732
Loss at epoch 9: 0.1816447377204895
Completed training! Final loss: 0.1816447377204895
Running validation...
```




```
Saving model...
Accuracy: 55.67999999999999%
```



Conclusion

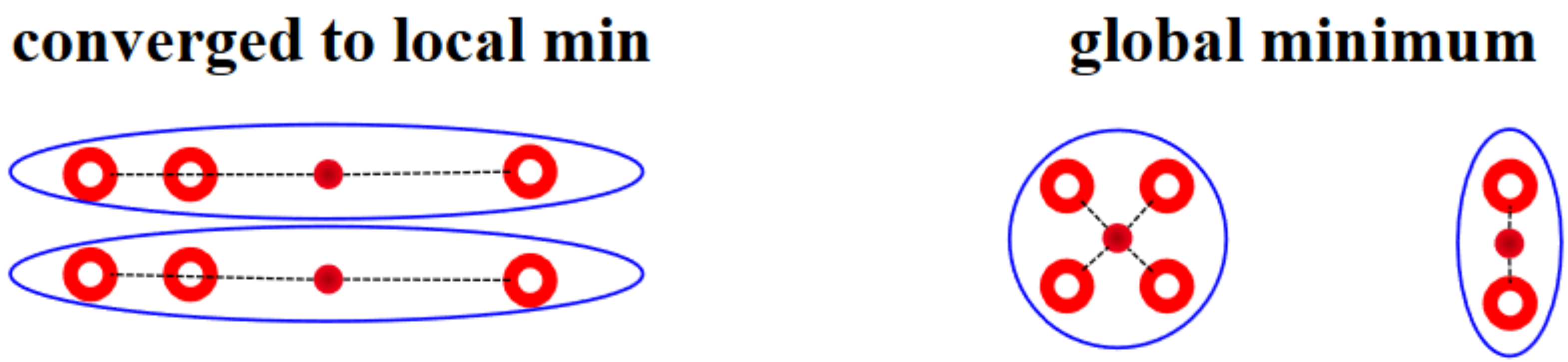
In this project, we investigated the potential of using semi-supervised learning for image classification on the MNIST handwritten digit dataset. We proposed a methodology that combined an auto-encoder for dimensionality reduction with a K-means clustering algorithm for classification. While the approach achieved some success in leveraging unlabeled data, limitations were identified.

Currently, our method of adding unlabeled data to the total dataset was a suboptimal approach. Instead, maintaining the full MNIST training dataset as labeled data and using data augmentation to generate new data would have made much more sense compared to removing data from the training set and marking them as unlabeled data could have provided the auto-encoder with a stronger foundation of data. Our initial approach of using the auto-encoder to directly reduce dimensionality without data augmentation might have contributed to the overfitting issue. Employing data augmentation techniques like Gaussian blur, contrast stretching, or impulse noise could have helped the auto-encoder extract more robust features from the data. This, in turn, could have improved the separation of encoded digits and potentially mitigated the overfitting observed in K-means clustering. While adding unlabeled data can be beneficial, it's crucial to ensure the labeled data is sufficient to guide the auto-encoder towards more robust feature extraction.

Furthermore, the auto-encoder's ability to separate similar digits significantly impacted K-means clustering performance. Looking at the confusion matrix, it appeared that the K-Means classifier struggled to differentiate visually similar digits. For example, the numbers "4" and "9" have much in common. There are two explanations we can offer that may contribute to this problem.

The first issue is that the labeled training set of the auto-encoder was too small, and may have introduced overfitting to the auto-encoder because not enough "unique-looking" digits were used for training. Our approach to labeled and unlabeled data is further described below and is a possible explanation to the failures of the auto-encoder. The hope for the auto-encoder was that the Euclidean distance between the encodings of two different digits is large, and the Euclidean distance between the encoding of the same digit is small, so that K-Means classification can converge to distinct clusters. However, this was not achieved as shown by digits such as "1" performing very well, but digits like "5" and "9" performed poorly.

The second issue is that K-Means classification is generally non-robust and sensitive to outliers. For example, suppose the cluster of "4" and "9" points were very close. Then the centroids of "4" and "9" could possibly merge, leading to a local minima.



(Image taken from CS 484 Topic 9A Slides)

Overall, this project highlights the potential of semi-supervised learning for image classification tasks. However, it also emphasizes the importance of carefully considering data augmentation techniques and the balance between labeled and unlabeled data for optimal performance. Future work could explore these avenues to further enhance the effectiveness of the proposed methodology.