Copy of MNIST

May 4, 2024

1 SBU CSE 352 - HW 4 - Machine Learning From Scratch

All student names in group: Joseph Hess , Brian Lopez Calle

I understand that my submission needs to be my own group's work: J.M.H. , B.LC

I understand that ChatGPT / Copilot / other AI tools are not allowed: J.M.H. , B.LC

1.1 Instructions

Total Points: 100

1. Complete this notebook. Use the provided notebook cells and insert additional code and markdown cells as needed. Only use standard packages (numpy and built-in packages like random). Submit the completely rendered notebook as a HTML file.

Important: Do not use scikit-learn or other packages with ML built in. The point of this is to be a learning exercise. Using linear algebra from numpy is okay (things like matrix operations or pseudoinverse, for example, but not lstsq).

2. Your notebook needs to be formatted professionally.
    - Add additional markdown blocks for your description, comments in the code, add tables and use matplotlib to produce charts where appropriate
    - Do not show debugging output or include an excessive amount of output.
    - Check that your PDF file is readable. For example, long lines are cut off in the PDF file. You don't have control over page breaks, so do not worry about these.
3. Document your code. Add a short discussion of how your implementation works and your design choices.

1.2 Introduction

You will implement several machine learning algorithms and evaluate their accuracy. This will be done for a downscaled version of the MNIST digit recognition dataset.

Like in real life, some of the tasks you will be asked to do may not be possible, at least directly. In these cases, your job is to figure out why it won't work and either propose a fix (best), or provide a clear explanation why it won't work.

For example, if the problem says to do k-nearest neighbors with a dataset of a billion points, this could require too much time to do each classification so it's infeasible to evaluate its test accuracy. In this case, you could suggest randomly downsample the data to a more manageable size, which

will speed things up by may lose some accuracy. In your answer, then, you should describe the problem and how you solved it and the trade-offs.

2 Data

First the code below ensures you have access to the training data (a subset of the MNIST images), consisting of 100 handwritten images of each digit.

[1]:
```
# First download the repo and change the directory to be the one where the␣
 ↪dependencies are.
# You should only need to do this once per session. If you want to reset, do␣
 ↪Runtime -> Disconnect and Delete Runtime
# You can always do !pwd to see the current working directory and !ls to list␣
 ↪current files.
!git clone https://github.com/stanleybak/CS7320-AI.git
%cd CS7320-AI/ML
!ls
```

```
Cloning into 'CS7320-AI'…
remote: Enumerating objects: 2738, done.
remote: Counting objects: 100% (855/855), done.
remote: Compressing objects: 100% (372/372), done.
remote: Total 2738 (delta 522), reused 796 (delta 478), pack-reused 1883
Receiving objects: 100% (2738/2738), 285.30 MiB | 12.50 MiB/s, done.
Resolving deltas: 100% (1690/1690), done.
Updating files: 100% (135/135), done.
/content/CS7320-AI/ML
line_fitting.ipynb      ML_example.ipynb        ML_for_tictactoe_self_play.ipynb
README.md
mini-mnist-1000.pickle  ML_for_tictactoe.ipynb  MNIST.ipynb
```

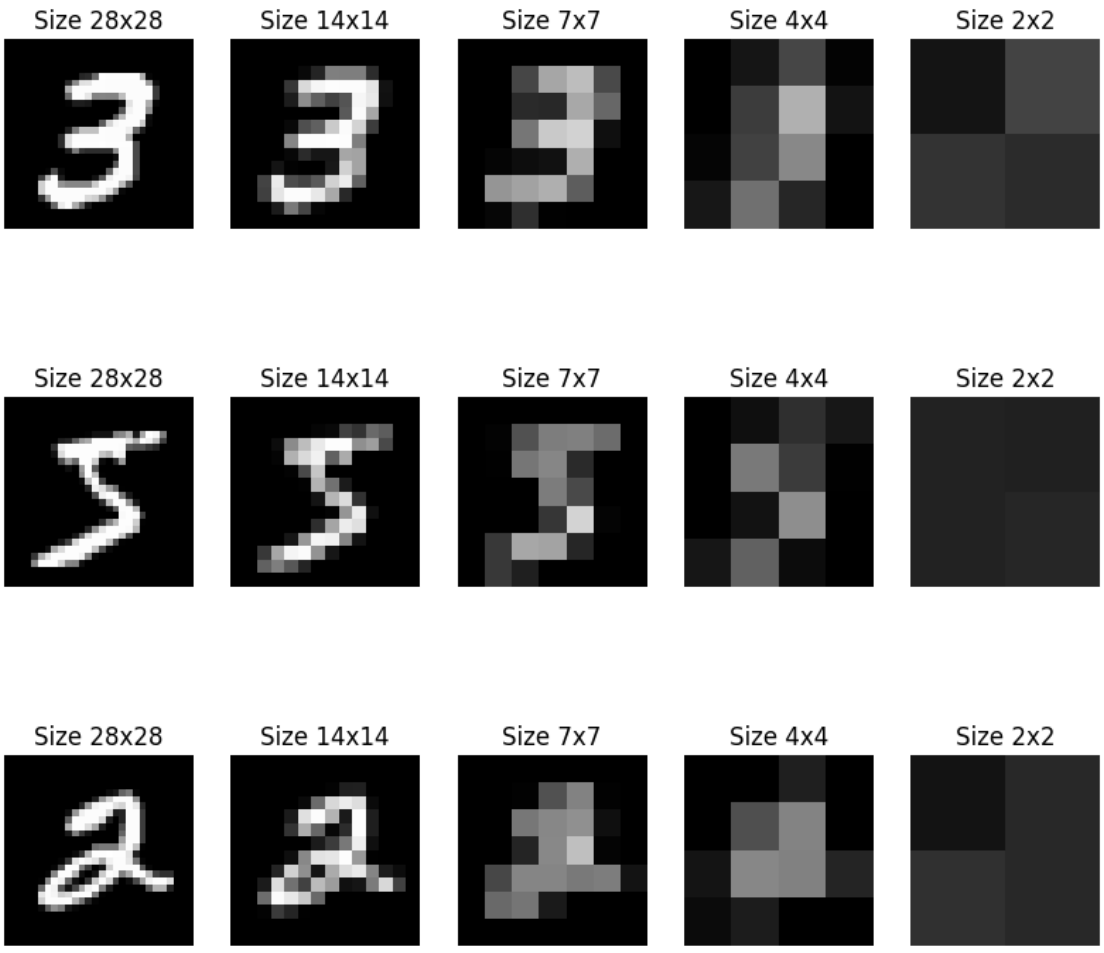
[2]:
```
import matplotlib.pyplot as plt
import pickle

# if the below fails to open, then the data file is not in the current working␣
 ↪directory (see above code block)
with open('mini-mnist-1000.pickle', 'rb') as f:
  data = pickle.load(f)

im3 = data['images'][300] # 100 images of each digit
plt.figure(figsize=(2, 2))  # Adjust size as needed
plt.imshow(im3, cmap='gray')
plt.axis('off')
plt.show()
```

3 Downscaling Images

MNIST images are originally 28x28. We will train our models not just on the original images, but also on downscaled images with the following sizes: 14x14, 7x7, 4x4, 2x2. The next code block shows one way to do downscaling. As you can tell from the output, we cannot expect our model's accuracy will be too high on lower resolution versions, although it's unclear how much better you can do than random chance, which should have a 10% accuracy.

[3]:
```
import numpy as np
import pickle
import matplotlib.pyplot as plt

# Function to downscale an image to different sizes
def downscale_image(image, downscaled_size):
    block_size = 28 // downscaled_size
    downscaled = np.zeros((downscaled_size, downscaled_size))
    for i in range(downscaled_size):
        for j in range(downscaled_size):
            # Calculate the average for each block
            block = image[i*block_size:(i+1)*block_size, j*block_size:
 ↪(j+1)*block_size]
            downscaled[i, j] = np.mean(block)
    return downscaled

# Load the dataset (assuming this file is in your working directory)
with open('mini-mnist-1000.pickle', 'rb') as f:
    data = pickle.load(f)

images = data['images']  # a list of 1000 numpy image matrices
labels = data['labels']  # a list of 1000 integer labels

# Select 3 "random" indices from the dataset
random_indices = [300, 500, 200]
```

```python
# Downscale the images to multiple sizes and display them
sizes = [28, 14, 7, 4, 2]
for index in random_indices:
    fig, axs = plt.subplots(1, len(sizes), figsize=(10, 2))
    for ax, size in zip(axs, sizes):
        downscaled_image = downscale_image(images[index], size)
        ax.imshow(downscaled_image, cmap='gray', vmin=0, vmax=255)
        ax.set_title(f'Size {size}x{size}')
        ax.axis('off')
    plt.show()
```



3.1 Task 1: Linear Classifier [20 points]

First, implement a linear classifier. The simplest way to do this is to adapt linear regression approaches that we learned about in class, where the output is a real number. For classification, we can let one category be an output of 1.0 and the other -1.0. Then, after the classifier is trained we can use the sign of the output to determine the predicted class.

However, since in MNIST there are multiple classes (10 digits, not just 2), we need to adapt the approach further. We will try both of the following two popular strategies: One-vs-Rest (OvR) and One-vs-One (OvO).

One-vs-Rest (OvR) is a strategy for using binary classification algorithms for multiclass problems. In this approach, a separate binary classifier is trained for each class, which predicts whether an instance belongs to that class or not, making it the 'one' against all other classes (the 'rest'). For a new input instance, compute the output of all classifiers. The predicted class is the one whose corresponding classifier gives the highest output value.

One-vs-One (OvO) is another strategy where a binary classifier is trained for every pair of classes. If there are N classes, you will train N(N−1)/2 classifiers. For a new input, evaluate it using all N(N−1)/2 classifiers. Count the number of times each class is predicted over all binary classifications. The class with the highest count is selected as the final prediction.

3.1.1 Report Results

Report the test accuracy for OvR and OvO, for each of the input image sizes, 28x28, 14x14, 7x7, 4x4, 2x2. A table may be helpful. Also report any interesting observations.

3.2 Preparing the Data

Before training the OvR and OvO linear classifiers, we split the data into training, evaluation, and test datasets.

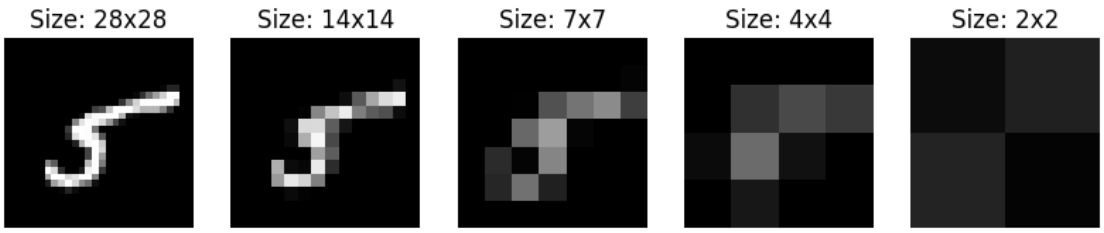
[4]:
```
train_pct = 80
evaluation_pct = 10
test_pct = 10

# Split the data by image category
images_by_category = np.split(images, [100, 200, 300, 400, 500, 600, 700, 800,␣
 ↪900])

# For each image class (e.g. 0, 1, 2, ..., 9), split the images in that
# class into training, evaluation, and testing data
# Note that since the classes start off in order we don't need to also keep
# track of the label, we get that for free
def get_split_by_label(images_for_label):
    # Shuffle indices within this class
    shuffled_indices = np.random.permutation(100)
    train_data = images_for_label[shuffled_indices[0:train_pct]]
    evaluation_data = images_for_label[shuffled_indices[train_pct:train_pct +␣
 ↪evaluation_pct]]
    test_data = images_for_label[train_pct + evaluation_pct:]

    return train_data, evaluation_data, test_data

# Get a list of tuples of training, eval, test data
train_eval_test = [get_split_by_label(images_by_category[i]) for i in range(10)]
```

```
# Now create ndarrays of all of the training, eval, test images
# Each of these arrays is 10*pct_this_class*28*28
# So e.g. training_data[3][61] would be a training image of a 3,
# though probably not the 61st image of a three in the original full
# set.
training_images = np.array([split[0] for split in train_eval_test])
evaluation_images = np.array([split[1] for split in train_eval_test])
test_images = np.array([split[2] for split in train_eval_test])
```

[5]:
```
# Same as above, copied here for clarity
sizes = [28, 14, 7, 4, 2]

# Now we generate the resized images for each of training, evaluation, and
# test data, then flatten each image from a shape of (size, size) to a shape
# of (size * size, 1), to make training more efficient
# The resulting new_data list is a list (NOT a numpy.ndarray, since the
# elements are of different shapes) of length 5 (since there are 5
# different sizes we are examining). Each element of the list is a numpy
# ndarray of shape (10, number_images_this_set, size*size)
def create_flattened_data(image_set: np.ndarray) -> list[np.ndarray]:
    new_data = []

    # For each size class
    for s, size in enumerate(sizes):
        new_data.append([])
        # For each digit
        for i in range(image_set.shape[0]):
            new_data[s].append([])
            # For each image
            for j in range(image_set.shape[1]):
                # Downscale the image
                new_data[s][i].append(downscale_image(image_set[i][j], size))
                # Then flatten the downscaled image
                new_data[s][i][j] = new_data[s][i][j].flatten()

    new_data = [np.array(new_data[i]) for i in range(len(new_data))]

    return new_data
```

[10]:
```
# Now convert each of the dataset into the flattened format

training_data = create_flattened_data(training_images)
evaluation_data = create_flattened_data(evaluation_images)
test_data = create_flattened_data(test_images)
```

[11]:
```
# Code taken from above and wrapped in a function
# Show the image at its various downscalings
def show_image(image: np.ndarray):
    sizes = [28, 14, 7, 4, 2]
    fig, axs = plt.subplots(1, len(sizes), figsize=(10, 2))
    for ax, size in zip(axs, sizes):
        downscaled_image = downscale_image(image, size)
        ax.imshow(downscaled_image, cmap='gray', vmin=0, vmax=255)
        ax.set_title(f'Size: {size}x{size}')
        ax.axis('off')

    plt.show()

# While the flattened format is easier to train with, it is useful to still be␣
 ↪able
# to use the visualization function given above, so we create a helper function
# to make that convenient.
def show_flattened_image(image_data: np.ndarray, size):
    show_image(image_data.reshape(size, size))

show_flattened_image(training_data[0][5][1], 28)
```



3.3 Defining the OvR Classifier

Here, we define the OvR linear classifier. We create a class to store the weights, and define methods on that class to train on a single image or on an entire dataset, as well as a method to evaluate the accuracy of the classifier over some other dataset.

[12]:
```
class OVRDigitClassifier:
    def __init__(self, image_size):
        self.image_size = image_size
        self.weights = np.zeros((10, image_size * image_size))

    # Predict the class of a given input
    # input_embedding should be of shape (size*size,1), not (size, size)
    # That is, it should already be flattened)
    def predict(self, input_embedding):
```

```
        dot_prod_vector = np.dot(self.weights, input_embedding)
        # Note that each index corresponds directly to the digit
        return np.argmax(dot_prod_vector)

    # Perform training against one image
    # input_embedding should be of shape (size*size,1), not (size, size)
    def train(self, input_data_raw, true_class):
        input_data = input_data_raw.reshape(self.image_size * self.image_size,␣
↪1)
        dot_prod_vector = np.dot(self.weights, input_data)
        prediction_vector = np.where(dot_prod_vector < 0, -1, 1)
        true_value_vector = np.where(np.arange(10) == true_class, 1, -1).
↪reshape(10, 1)
        update_sign_vector = np.where(prediction_vector == true_value_vector,␣
↪0, true_value_vector)
        update_value_vector = update_sign_vector * input_data.T
        self.weights = self.weights + update_value_vector

    # Train on an entire flattened dataset
    # Dataset should be an ndarray of shape
    # (10, num_images, size*size)
    def train_dataset(self, input_dataset):
        for digit in range(input_dataset.shape[0]):
            for example in range(input_dataset.shape[1]):
                self.train(input_dataset[digit][example], digit)

    def evaluate(self, evaluation_dataset):
        correct = 0.0
        total = 0.0
        for digit in range(evaluation_dataset.shape[0]):
            for example in range(evaluation_dataset.shape[1]):
                prediction = self.predict(evaluation_dataset[digit][example])

                if prediction == digit:
                    correct += 1
                total += 1

        return correct / total
```

3.4 Training OvR Classifiers

Here, we train linear OvR classifiers for each of the given images sizes.

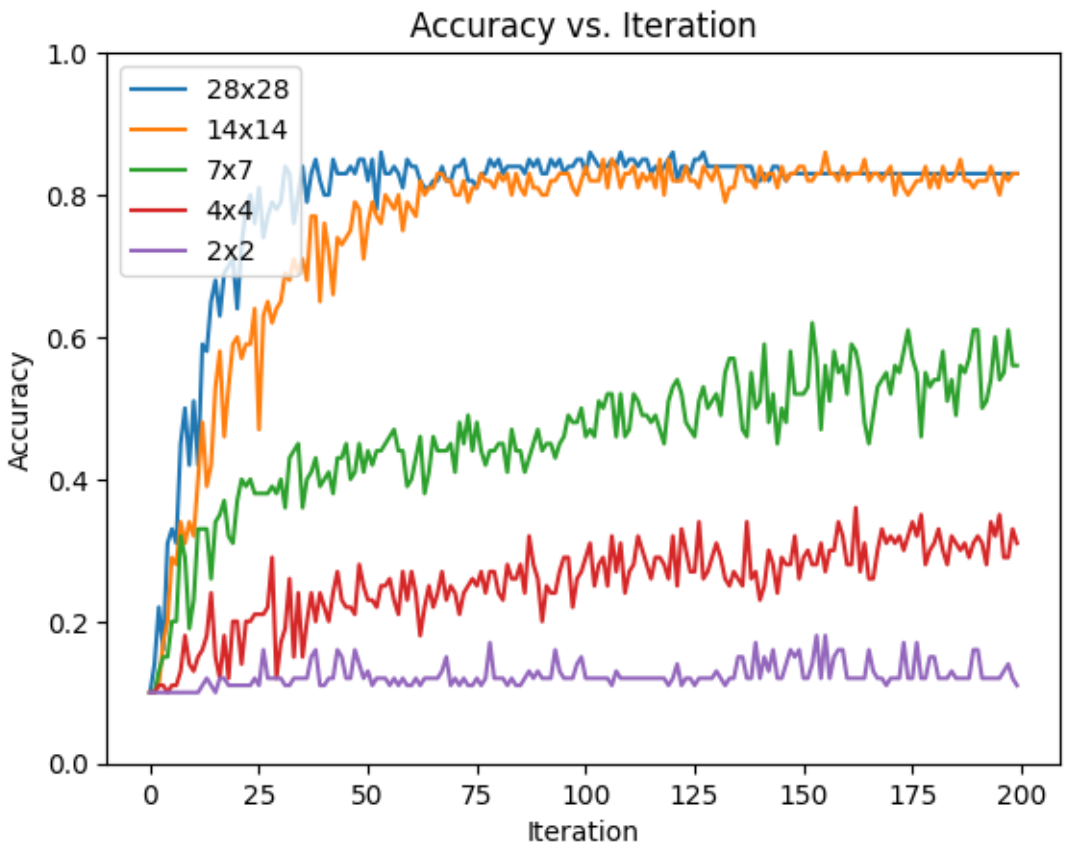
After some experimentation, we determined that after about 200 iterations, all classifiers had essentially stabilized.

[ ]:
```
from tqdm import tqdm

iterations = 200

ovr_classifiers = [OVRDigitClassifier(size) for size in sizes]
ovr_accuracy_over_time = [[] for _ in sizes]
for i in tqdm(range(iterations)):
    for index, classifier in enumerate(ovr_classifiers):
        classifier.train_dataset(training_data[index])
        ovr_accuracy_over_time[index].append(classifier.
 ↪evaluate(evaluation_data[index]))
```

```
100%|      | 200/200 [00:15<00:00, 12.74it/s]
```

[ ]:
```
x = np.arange(iterations)

for index, classifier_accuracy in enumerate(ovr_accuracy_over_time):
    plt.plot(x, classifier_accuracy, label=f'{sizes[index]}x{sizes[index]}')

plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Iteration')
plt.ylim(0, 1.0)
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7fb9d9cdb910>
```



[ ]:
```
for index, classifier in enumerate(ovr_classifiers):
    print(f'{sizes[index]}x{sizes[index]} classifier achieved test accuracy of␣
 ↪{classifier.evaluate(test_data[index])}')
```

```
28x28 classifier achieved test accuracy of 0.99
14x14 classifier achieved test accuracy of 0.9
7x7 classifier achieved test accuracy of 0.49
4x4 classifier achieved test accuracy of 0.28
2x2 classifier achieved test accuracy of 0.11
```

3.5 Results

We see, as expected, that the 28x28 classifier performs the best out of all. Interestingly, the downscaled 14x14 classifier achieves similar accuracy, albeit after a longer training period. The 7x7 and 4x4 classifiers achieve unremarkeable performance, and unsurprisingly, the 2x2 classfier barely does better than random

3.6 Defining the OvO digit classifier

Here, we define a class similar to the one above, but for OvO digit classifiers instead of OvR. Whereas an OvR classifier of a given size contains 10 different subclassifiers - one for each digit

- the OvO classifier contains 90 different subclassifiers - one for each pair of digits (note that the nonlinear update step means that an a-against-b classifier will not necessarily be identical to a b-against-a classifier).

[7]:
```
# Here we define a "one-vs-one" classifier for base 10 digits
# Each classifier has a "base digit" that corresponds to 1
# and a "competitor digit" that corresponds to -1; this
# was chosen arbitrarily but seems to match the intuition
# gleaned from the one-vs-rest case
# Note that since the training updates only trigger when
# the dot product is negative training is nonlinear so a
# (0 vs 1) classifier is not the same as a (1 vs 0)
class OVODigitClassifier:
    def __init__(self, image_size: int):
        self.image_size = image_size
        # Create a grid of classifiers matched by size
        # I.e. weights[0] is (0 vs 1), weights[8] is (0 vs 9),
        # weights[9] is (1 vs 0), etc
        self.weights = np.zeros((10 * (10 - 1), image_size * image_size))
        # For comparisons during training
        self.base_digit = np.array([np.full(9, i) for i in range(10)]).flatten()
        # Each digit should match up against everything that it isn't
        self.competitor_digit = np.array([np.delete(np.array([0, 1, 2, 3, 4, 5,␣
↪6, 7, 8, 9]), i) for i in range(10)]).flatten()

    # Train on a single image
    # We only update the classifiers where the given image
    # is one of the base digit or competitor digit
    def train(self, input_data_raw, actual_value):
        # Again, make sure it's a matrix
        input_data = np.reshape(input_data_raw, (self.image_size * self.
↪image_size, 1))
        # Get the indices where the given image is relevant
        relevant_indices = np.where((self.base_digit == actual_value) | (self.
↪competitor_digit == actual_value))
        # Retrieve only the classifiers that are relevant
        relevant_rows = self.weights[relevant_indices]
        dot_prod_vector = np.dot(relevant_rows, input_data)
        prediction_vector = np.where(dot_prod_vector < 0, -1, 1)
        true_value_vector = np.where(self.base_digit[relevant_indices] ==␣
↪actual_value, 1, -1).reshape(relevant_rows.shape[0], 1)
        update_sign_vector = np.where(prediction_vector == true_value_vector,␣
↪0, true_value_vector)
        update_value_vector = update_sign_vector * input_data.T
        self.weights[relevant_indices] = self.weights[relevant_indices] +␣
↪update_value_vector
```

```
    # Train on an entire flattened dataset
    # Dataset should be an ndarray of shape
    # (10, num_images, size*size)
    def train_dataset(self, input_dataset):
        for digit in range(input_dataset.shape[0]):
            for example in range(input_dataset.shape[1]):
                self.train(input_dataset[digit][example], digit)

    def predict(self, input_embedding):
        # Get the predictions
        dot_prod_vector = np.dot(self.weights, input_embedding)
        # -1 means the competitor/other wins, 1 means
        results_each_ovo = np.where(dot_prod_vector < 0, self.competitor_digit,␣
↪self.base_digit)
        vals, counts = np.unique(results_each_ovo, return_counts=True)
        # No reason to assume each value will even appear in np.unique, let␣
↪alone in order
        mode_index = np.argmax(counts)
        return vals[mode_index]

    def evaluate(self, evaluation_dataset):
        correct = 0.0
        total = 0.0
        for digit in range(evaluation_dataset.shape[0]):
            for example in range(evaluation_dataset.shape[1]):
                prediction = self.predict(evaluation_dataset[digit][example])

                if prediction == digit:
                    correct += 1
                total += 1

        return correct / total
```

3.7 Training the OvO Classifiers

Here, we train an OvO classifier for each given image size. As before, we experimentally determined 200 iterations to be a reasonable stopping point for training most classifiers.

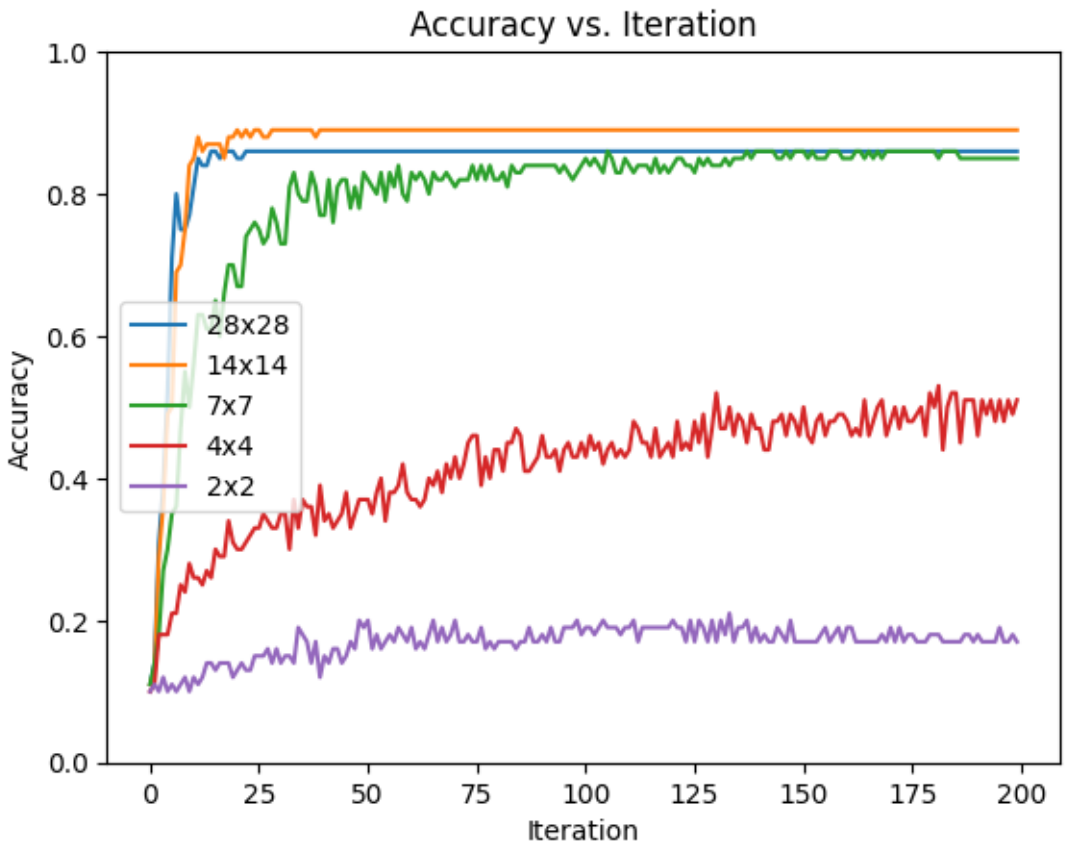
[ ]:
```
from tqdm import tqdm

iterations = 200

ovo_classifiers = [OVODigitClassifier(size) for size in sizes]
ovo_accuracy_over_time = [[] for _ in sizes]
for i in tqdm(range(iterations)):
    for index, classifier in enumerate(ovo_classifiers):
        classifier.train_dataset(training_data[index])
```

```
        ovo_accuracy_over_time[index].append(classifier.
↪evaluate(evaluation_data[index]))
```

```
100%|      | 200/200 [00:51<00:00,  3.90it/s]
```

[ ]:
```
x = np.arange(iterations)

for index, classifier_accuracy in enumerate(ovo_accuracy_over_time):
    plt.plot(x, classifier_accuracy, label=f'{sizes[index]}x{sizes[index]}')

plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Iteration')
plt.ylim(0, 1.0)
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7fb9d97054e0>
```



[ ]:
```
for index, classifier in enumerate(ovo_classifiers):
    print(
```

```
        f'{sizes[index]}x{sizes[index]} classifier achieved test accuracy of␣
↪{classifier.evaluate(test_data[index])}')
```

```
28x28 classifier achieved test accuracy of 0.98
14x14 classifier achieved test accuracy of 0.97
7x7 classifier achieved test accuracy of 0.92
4x4 classifier achieved test accuracy of 0.41
2x2 classifier achieved test accuracy of 0.16
```

3.8 Results

These results are much more interesting than those for the OVR classifiers. Here, the 28x28, 14x14, and 7x7 classifiers all achieve similar evaluation accuracy, and all of them beat their OVR counterparts. In particular, the 7x7 classifier did not even break 50% in the previous experiment. The 4x4 classifier is improved here, but still unremarkeable, and the 2x2 continues to be a disappointment.

3.9 Task 2: Data Augmentation [20 points]

Your boss was unhappy with the test accuracy, especially of your 2x2 image classifier, and has made some suggestions. The problem, according to your boss, is that there is not enough data in each input x. You are told to augment the data with derived features in order to help the classifier.

Specifically, given an input x, create additional attributes by computing all of the data up to powers of two. For example, in the 2x2 case your example x consists of four pixel values x_0, x_1, x_2, and x_3. Your new input data would have:

- all power of zero: 1 (constant)
- all powers of one: x_0, x_1, x_2, x_3
- all powers of two:

  x_0^2, x_0x_1, x_0x_2, x_0x_3,

  x_1^2, x_1x_2, $ x_1 x_3$,

  x_2^2, x_2x_3,

  x_3^2

The data would have 15 values, which has the potential to learn nonlinear relationships between the original inputs, which was not possible before.

3.9.1 Report Results

Report the test accuracy for OvR only, with the data augmentation approach, for each of the input image sizes, 28x28, 14x14, 7x7, 4x4, 2x2 (again, perhaps incorporating a table). Report any interesting results or observations.

Also, explain to your boss what the danger is of looking at a model's final test accuracy and then suggesting changes to improve it. What should be done instead, if you know you will consider different types of models or hyperparameters in the same model class?

3.10 Explanation of the Problem

Once you use test data to change something about your model, that test data ceases to be test data. The whole point of test data is that whatever model you build runs the risk of overfitting on the training data and failing to generalize. Holding on to the test data until you are ready to deploy the solution helps provide a reasonable estimate of the real world accuracy of the model, and revising the model destroys the ability to do that. Instead, the validation data should be used for things like hyperparameter tuning or model selection; if this was a possibility to begin with, the models should have been created before breaking in to the test data.

3.11 Defining the Augmented OvR Classifier

The augmented OvR classifier is very similar to the regular classifier; we only really modify how we use the input_size hyperparameter to match the size of the augmented feature set.

We also define an augment_data function to construct the augmented datasets, as per the instructions.

[32]:
```
class AugmentedOVRDigitClassifier:
    def __init__(self, input_size):
        # Size will be different since we are augmenting.
        # We still input the actual image size here, but internally, we
        # need the augmented data size.
        # Note that for any natural number n, n*(n+1) is even,
        # so floor division is fine here
        full_image_size = input_size * input_size
        self.input_size = 1 + full_image_size + (full_image_size *␣
↪(full_image_size + 1)) // 2
        self.weights = np.zeros((10, self.input_size))

    def predict(self, input_embedding):
        dot_prod_vector = np.dot(self.weights, input_embedding)
        return np.argmax(dot_prod_vector)

    def train(self, input_data_raw, true_class):
        input_data = input_data_raw.reshape(self.input_size, 1)
        dot_prod_vector = np.dot(self.weights, input_data)
        prediction_vector = np.where(dot_prod_vector < 0, -1, 1)
        true_value_vector = np.where(np.arange(10) == true_class, 1, -1).
↪reshape(10, 1)
        update_sign_vector = np.where(prediction_vector == true_value_vector,␣
↪0, true_value_vector)
        update_value_vector = update_sign_vector * input_data.T
        self.weights = self.weights + update_value_vector


    def train_dataset(self, input_dataset):
        for digit in range(input_dataset.shape[0]):
            for example in range(input_dataset.shape[1]):
```

```
                self.train(input_dataset[digit][example], digit)

    def evaluate(self, evaluation_dataset):
        correct = 0.0
        total = 0.0
        for digit in range(evaluation_dataset.shape[0]):
            for example in range(evaluation_dataset.shape[1]):
                prediction = self.predict(evaluation_dataset[digit][example])
                if prediction == digit:
                    correct += 1
                total += 1
        return correct / total


def augment_data(input_data):
    power_zero = [1]
    power_one = input_data.tolist()
    power_two = []
    for i in range(len(input_data)):
        for j in range(i, len(input_data)):
            power_two.append(input_data[i]*input_data[j])

    augmented_data = power_zero + power_one + power_two

    return np.array(augmented_data)
```

3.12 Performing the data augmentation

Here, we use the function defined above to actually augment the data.

[33]:
```
training_data_augmented = [np.array([[augment_data(data) for data in␣
 ↪digit_data] for digit_data in size_data]) for size_data in training_data]
evaluation_data_augmented = [np.array([[augment_data(data) for data in␣
 ↪digit_data] for digit_data in size_data]) for size_data in evaluation_data]
test_data_augmented = [np.array([[augment_data(data) for data in digit_data]␣
 ↪for digit_data in size_data]) for size_data in test_data]
```

3.13 Training the augmented classifiers

Here, we train the augmented classifiers, as in the previous example.

Since the datasets and classifiers are much larger here, they take much longer to train, so we only run for 50 iterations instead of 200.

[34]:
```
from tqdm import tqdm

aug_ovr_iterations = 50
```

```
aug_ovr_classifiers = [AugmentedOVRDigitClassifier(size) for size in sizes]
aug_ovr_accuracy_over_time = [[] for _ in sizes]
for i in tqdm(range(aug_ovr_iterations)):
    for index, classifier in enumerate(aug_ovr_classifiers):
        classifier.train_dataset(training_data_augmented[index])
        aug_ovr_accuracy_over_time[index].append(classifier.
 ↪evaluate(evaluation_data_augmented[index]))
```

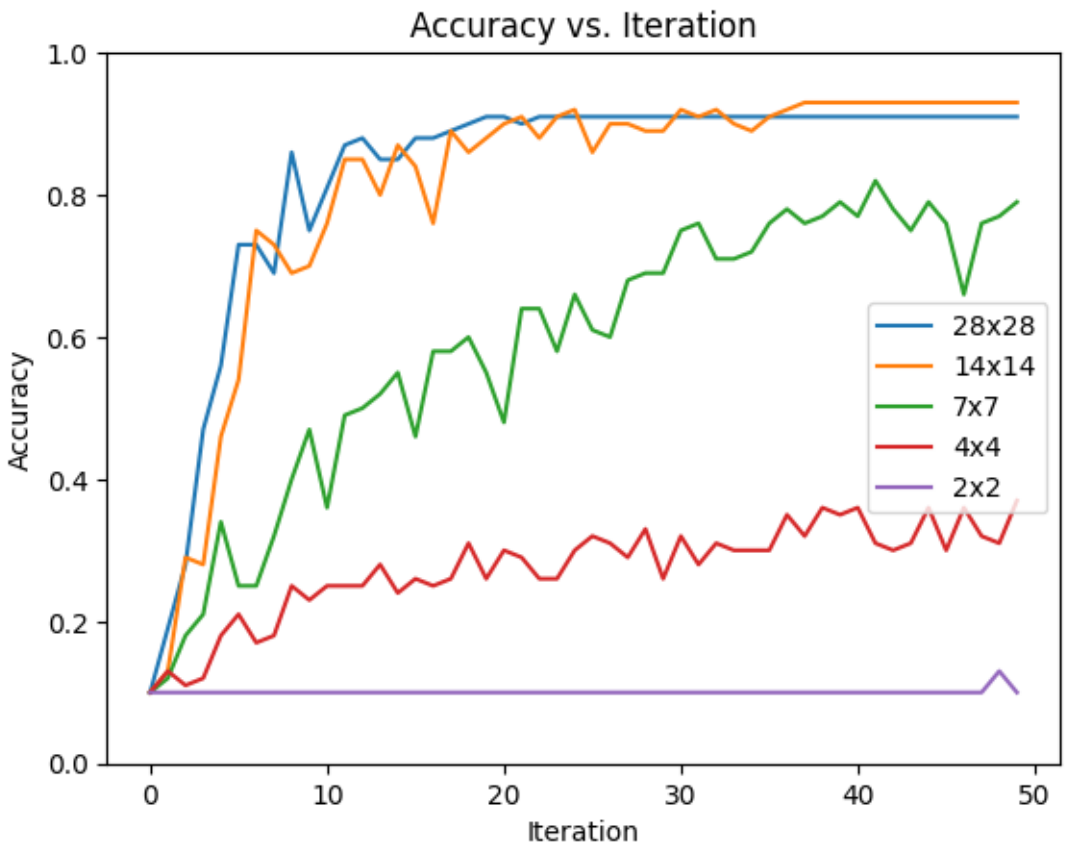
```
100%|      | 50/50 [14:35<00:00, 17.50s/it]
```

[ ]:
```
aug_ovr_accuracy_over_time
```

[ ]:
```
[[0.17, 0.21, 0.4, 0.49, 0.67],
 [0.17, 0.28, 0.39, 0.35, 0.49],
 [0.11, 0.16, 0.2, 0.34, 0.32],
 [0.11, 0.13, 0.1, 0.19, 0.12],
 [0.1, 0.1, 0.11, 0.1, 0.1]]
```

[35]:
```
x = np.arange(aug_ovr_iterations)

for index, classifier_accuracy in enumerate(aug_ovr_accuracy_over_time):
    plt.plot(x, classifier_accuracy, label=f'{sizes[index]}x{sizes[index]}')

plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Iteration')
plt.ylim(0, 1.0)
plt.legend()
```

[35]:
```
<matplotlib.legend.Legend at 0x7880de4fab00>
```



[36]:
```
for index, classifier in enumerate(aug_ovr_classifiers):
    print(f'{sizes[index]}x{sizes[index]} classifier achieved test accuracy of␣
 ↪{classifier.evaluate(test_data_augmented[index])}')
```

```
28x28 classifier achieved test accuracy of 0.96
14x14 classifier achieved test accuracy of 0.96
7x7 classifier achieved test accuracy of 0.76
4x4 classifier achieved test accuracy of 0.37
2x2 classifier achieved test accuracy of 0.1
```

3.14 Results

After a significantly longer training time, we observe little increase in the augmented dataset compared to the original dataset. Overall, the increased training and inference costs make this approach ill advised in the context of the given problem.

3.15 Task 3: k-Nearest Neighbors Classifier [20 points]

Your boss is still unhappy with the results (and still ignoring your advice about not using test data accuracy for model decisions).

Next, you are to use the k-nearest neighbors approach to build a classifier for our data. Since we have multiple classes, the one that gets selected can be based on a plurality vote of the k closest samples (whichever category is most frequent). If there are ties, select the class based on the sum of the distances from the test point. For example, if $k = 5$, and the closest 5 samples have two pictures that are from category "1" and two pictures that are from category "7", then you choose the output by computing the sum of the distance from the test point and the two "5" samples, as well as the sum of distances from the test point to the two "7" samples, and then outputting the class with the smaller total distance.

3.15.1 Report Results

For each image size, exhaustively explore different values of k up to 50. Report the best test accuracy. Report the average time taken to do a lookup with the model.

[ ]:
```
from scipy.spatial import distance
import time


class KNNClassifier:
    def __init__(self, k, training_data, training_labels):
        self.k = k
        self.training_data = training_data
        self.training_labels = training_labels

    def predict(self, test_image):
    # calculate euclidean dist from test image to training images
      distances = [distance.euclidean(test_image, train_image) for train_image␣
 ↪in self.training_data]
    # Get the indices of the k closest training images
      k_nearest_indices = np.argsort(distances)[:self.k]
    # Get the labels of the k nearest neighbors
      k_nearest_labels = [self.training_labels[i] for i in k_nearest_indices.
 ↪tolist()]
    # Identify the most common label
      predicted_label = np.argmax(np.bincount(k_nearest_labels))
    # If there's a tie, choose the label with the smallest total distance
      if list(k_nearest_labels).count(predicted_label) < self.k / 2:
        label_distances = [distances[i] for i in k_nearest_indices if self.
 ↪training_labels[i] == predicted_label]
        predicted_label = min(set(k_nearest_labels), key=label_distances.count)

      return predicted_label
```

```
training_data_flattened = [image.flatten() for size in training_data for digit␣
 ↪in size for image in digit]
training_labels_flattened = [label for label in range(10) for _ in range(100)] ␣
 ↪#100 training images per digit


best_accuracy = 0
best_k = 0
best_size = 0
average_time = 0


for size_index, size in enumerate(sizes):
    training_data_flattened = [image.flatten() for digit in␣
 ↪training_data[size_index] for image in digit]
    training_labels_flattened = [label for label in range(100) for _ in␣
 ↪range(len(training_data[size_index][0]))]

    # For each value of k
    for k in range(1, 51):
        start_time = time.time()
        correct_predictions = 0
        total_predictions = 0

        knn = KNNClassifier(k, training_data_flattened,␣
 ↪training_labels_flattened)

        for label in range(10):
            for test_image in test_data[size_index][label]:
                test_image_flattened = test_image.flatten()
                predicted_label = knn.predict(test_image_flattened)

                if predicted_label == label:
                    correct_predictions += 1

                total_predictions += 1

        accuracy = correct_predictions / total_predictions

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_k = k
            best_size = size

        average_time += (time.time() - start_time) / total_predictions

print(f"Best Accuracy: {best_accuracy}")
```

```
print(f"Best k: {best_k}")
print(f"Best Size: {best_size}")
print(f"Average Time: {average_time}")
```

```
Best Accuracy: 0.99
Best k: 1
Best Size: 7
Average Time: 1.090654752254486
```

3.16 Task 4: Neural Networks [40 Points]

Next, your boss wants you to try neural networks. Rather than using a library for everything, you will **only** use `pytorch` to perform backpropagation and compute gradients. You can write your own neural network class if desired, don't use anything from `pytorch` for that.

An example network and how to compute gradients with pytorch is shown below.

[13]:
```
# Example of using pytorch to compute gradients and updates weights and biases
#
# The network consists of:
# 1. An input layer with 3 features.
# 2. A first hidden layer with 3 neurons. Each neuron in this layer performs a linear transformation
#    on the input data using a weight matrix (W1) and a bias vector (b1). This is followed by a sigmoid
#    activation function.
# 3. A second hidden layer, also with 3 neurons, which processes the output of the first layer. Similar
#    to the first layer, it uses a weight matrix (W2) and a bias vector (b2) for linear transformation,
#    followed by a softmax activation function. The softmax activation is used here to normalize the
#    output of the second layer into a probability distribution over the three classes. This is particularly
#    useful for multi-class classification problems.
# 4. The network uses cross-entropy as the loss function, which is a common choice for classification tasks
#    involving softmax outputs. This loss function compares the predicted probability distribution with the
#    true distribution (one-hot encoded) and penalizes the predictions that diverge from the actual labels.
#


import torch
```

```python
# Initialize input, weights, and biases
x = torch.tensor([1.0, 2.0, 3.0])
W1 = torch.tensor([[0.1, 0.2, 0.5],
                   [-0.1, -0.5, -1.1],
                   [0, 7.5, -1.1]], requires_grad=True)
b1 = torch.tensor([0.0, 0.0, 0.0], requires_grad=True)

W2 = torch.tensor([[0.1, -0.3, 0.4],
                   [0.2, 0.4, -0.6],
                   [-0.1, 0.5, -0.2]], requires_grad=True)
b2 = torch.tensor([0.0, 0.0, 0.0], requires_grad=True)

# Target output
y_true = torch.tensor([1.0, 0.0, 0.0])

# Forward pass through first layer
z1 = torch.matmul(W1, x) + b1
a1 = torch.sigmoid(z1)  # Sigmoid activation

# Forward pass through second layer
z2 = torch.matmul(W2, a1) + b2
a2 = torch.softmax(z2, dim=0)  # Softmax activation

print("Initial Output:", a2)
print("Desired Output:", y_true)

# Compute loss (Cross-entropy): https://en.wikipedia.org/wiki/Cross-entropy
loss = -torch.sum(y_true * torch.log(a2))
print("Initial loss:", loss.item())

# Backpropagation
loss.backward()

# you can print out gradient for each element now
print("Gradient for weights matrix W1:", W1.grad)

# Update weights and biases based on gradient (should reduce loss)
learning_rate = 0.02

# the no_grad() environment is needed to indicate that the computation should␣
 ↪not
# be part of the gradient computation
with torch.no_grad():
    W1 -= learning_rate * W1.grad
    b1 -= learning_rate * b1.grad
    W2 -= learning_rate * W2.grad
    b2 -= learning_rate * b2.grad
```

```
# After the update, clear the gradients (in case we want to compute them again␣
 ↪later)
W1.grad.zero_()
b1.grad.zero_()
W2.grad.zero_()
b2.grad.zero_()

# Forward pass with updated weights and biases
z1 = torch.matmul(W1, x) + b1
a1 = torch.sigmoid(z1)  # Sigmoid activation
z2 = torch.matmul(W2, a1) + b2
a2 = torch.softmax(z2, dim=0)  # Softmax activation

# Compute new loss
new_loss = -torch.sum(y_true * torch.log(a2))
print("New loss after updating weights and biases:", new_loss.item())
```

```
Initial Output: tensor([0.5348, 0.2167, 0.2485], grad_fn=<SoftmaxBackward0>)
Desired Output: tensor([1., 0., 0.])
Initial loss: 0.625852644443512
Gradient for weights matrix W1: tensor([[-2.9431e-03, -5.8862e-03, -8.8293e-03],
        [ 4.1993e-03,  8.3986e-03,  1.2598e-02],
        [-3.0524e-06, -6.1048e-06, -9.1572e-06]])
New loss after updating weights and biases: 0.6079817414283752
```

The code above updates the parameters based on a single piece of data, but often multiple inputs are used and their gradient is averaged when updating a model.

Your task is to write the training code for the different neural network architectures proposed and report accuracy. Start with all random parameters between -1 and 1. Training should stop when the accuracy, as measured on the validation data, no longer appears to be improving. You can plot the validation data accuracy over time to ensure this looks correct. If this takes too long but it appears the model is still improving in accuracy, consider increasing the learning rate (start with 0.02 as in the example).

For the gradient, you are to compute the gradient over the full set of training data, and then average them together before you update. Then, repeat with mini-batches of size 100, with 10 random samples from each class. This should update the model weights faster, but may require more updates to get the accuracy down.

3.16.1 Report Results

Provide at least one plot of your validation data accuracy going down over time as training progresses. What was the condition you decided to use to detect if training should stop? How many updates were needed in the case of your plot?

Create a table where each row corresponds to one model and training method (mini-batch or full). Use the 7x7 version of the data (49-dimensional inputs). You are to explore the following models: the number of hidden layers can be varied between 2 and 4. Each layer's size can be

16, 32, or 64 neurons (all hidden layers have the same number of neurons). Explore three different activation functions for the network, ReLU (`torch.relu`), arctan (`torch.atan`), and sigmoid (`torch.sigmoid`). After the final layer, use a softmax rather than the normal network activation function, to ensure all outputs are between 0 and 1. There should be 10 outputs, one for each class in the MNIST data.

In the table, report the architecture, training time, number of model updates and test accuracy. What is the best architecture? Did mini-batches help with anything? Report any other interesting observations.

4 Converting Data to Tensors

PyTorch uses its own Tensor datastructures internally, so we convert our training data to this format ahead of time to speed up training.

[14]:
```
# We only need the 7x7 data, per the instructions#
perceptron_training_data = torch.tensor(training_data[2]).float()
perceptron_evaluation_data = torch.tensor(evaluation_data[2]).float()
perceptron_test_data = torch.tensor(test_data[2]).float()

# Precalculate the true vectors to make training more efficient
# All the true vectors for each digit will be the same, but
# it is easier to index the tensor if its index matches the original
# data, and its not that much extra space in any case
perceptron_training_true = torch.zeros(perceptron_training_data.shape[0],␣
 ↪perceptron_training_data.shape[1], 10)
perceptron_evaluation_true = torch.zeros(perceptron_evaluation_data.shape[0],␣
 ↪perceptron_evaluation_data.shape[1], 10)
perceptron_test_true = torch.zeros(perceptron_test_data.shape[0],␣
 ↪perceptron_test_data[1].shape[1], 10)

# For each digit
for digit in range(perceptron_training_true.shape[0]):
    true_tensor = torch.zeros(10)
    true_tensor[digit] += 1.0
    # For each example
    for j in range(perceptron_training_true.shape[1]):
        perceptron_training_true[digit][j] += true_tensor

for digit in range(perceptron_evaluation_true.shape[0]):
    true_tensor = torch.zeros(10)
    true_tensor[digit] += 1.0
    # For each example
    for j in range(perceptron_evaluation_true.shape[1]):
        perceptron_evaluation_true[digit][j] += true_tensor

for digit in range(perceptron_test_true.shape[0]):
```

```
    true_tensor = torch.zeros(10)
    true_tensor[digit] += 1.0
    # For each example
    for j in range(perceptron_test_true.shape[1]):
        perceptron_test_true[digit][j] += true_tensor
```

5 Building a Perceptron

Here, we define a class that can build a perceptron to aid in building all 54 perceptrons needed for the experiment. We define training and evaluation methods on this class itself, rather than on some external training class.

Our model accepts the following arguments:

input_dim: The dimension of the input data. For the 7x7 image class we will be training on, this will be 49.

num_hidden_layers: The number of hidden layers, including the layer directly after the inputs and the final classifier layer.

hl_size: The number of neurons in a hidden layer.

activation: The nonlinear activation function applied to each hidden layer (except for the last). The function itself should be passed here, not a string, enum, or anything of that sort.

learning_rate: The learning rate of the perceptron, applied during training.

min_init: The lower bound of the probability distribution that dictates the initial values in the weight tensors. When the weight tensors are first randomly initialized, they will be set by sampling from a uniform distribution ranging from min_init to max_init. The instructions require us to set this to -1.0; we will use it later to explore the behavior of ReLU.

max_init: The upper bound of the probability distribution that dictates the initial values in the weight tensors. This is defaulted to 1.0, per the instructions.

[15]:
```
import sys

class Perceptron:
    def __init__(self, input_dim, num_hidden_layers, hl_size, activation,␣
 ↪learning_rate = 0.02, min_init=-1.0, max_init=1.0):
        self.input_dim = input_dim
        self.num_hidden_layers = num_hidden_layers
        self.hl_size = hl_size
        self.activation = activation
        self.learning_rate = learning_rate

        spread = max_init - min_init

        self.hidden_layers = []
        self.biases = []
```

```
        # Add the first layer separately
        self.hidden_layers.append(spread * torch.rand(self.hl_size, input_dim)␣
↪+ min_init)
        self.biases.append(spread * torch.rand(self.hl_size) + min_init)
        # Require gradients after the layers are created so that
        # shifting them from (0,1) to (-1,1) isn't included
        # in the gradient
        self.hidden_layers[0].requires_grad_()
        self.biases[0].requires_grad_()
        # The first hidden layer has a different shape
        # And the last layer has a different activation function (and shape)
        for i in range(num_hidden_layers - 2):
            self.hidden_layers.append(spread * torch.rand(hl_size, hl_size) +␣
↪min_init)
            self.biases.append(spread * torch.rand(hl_size) + min_init)
            # +1 because the first layer is handle separately
            self.hidden_layers[i + 1].requires_grad_()
            self.biases[i + 1].requires_grad_()

        self.output_layer = spread * torch.rand(10, self.hl_size) + min_init
        self.output_bias = spread * torch.rand(10) + min_init
        self.output_layer.requires_grad_()
        self.output_bias.requires_grad_()

    def forward_pass(self, input_vector):
        current_vec = input_vector

        for i in range(len(self.hidden_layers)):
            current_vec = torch.matmul(self.hidden_layers[i], current_vec) +␣
↪self.biases[i]
            current_vec = self.activation(current_vec)

        current_vec = torch.matmul(self.output_layer, current_vec) + self.
↪output_bias
        current_vec = torch.softmax(current_vec, dim=0)

        return current_vec

    def update_weights(self):
        with torch.no_grad():
            for i in range(len(self.hidden_layers)):
                self.hidden_layers[i] -= self.hidden_layers[i].grad * self.
↪learning_rate
                self.biases[i] -= self.biases[i].grad * self.learning_rate

            self.output_layer -= self.output_layer.grad * self.learning_rate
            self.output_bias -= self.output_bias * self.learning_rate
```

```
        # Zero out the gradients
        for i in range(len(self.hidden_layers)):
            self.hidden_layers[i].grad.zero_()
            self.biases[i].grad.zero_()

        self.output_layer.grad.zero_()
        self.output_bias.grad.zero_()

    # Train on a single example
    def train(self, input_vector, true_vector, epsilon=1e-8):
        output = self.forward_pass(input_vector)
        # Cross Entropy loss
        # If the softmax function ever returns a zero in any category
        # that is not the correct feature, then we end up getting
        # -inf from torch.log, which gets multiplied by 0 in the
        # true_vector, resulting in a nan which ruins the training.
        # This is especially impactful for ReLu, which gives a NaN
        # after the very first training pass if epsilon is not added.
        # Adding a small epsilon here prevents this.
        loss = -torch.sum(true_vector * torch.log(output+epsilon))
        # Do the loss here instead of in backward so that
        # we can reuse backward_pass for training in
        # dataset/minibatches
        loss.backward()
        self.update_weights()

        return loss

    # Train over the training corpus, once
    # input_dataset should be a tensor of shape
    # (10, num_examples, size_single_example), so
    # in this case (10, 80, 49)
    def train_dataset(self, input_dataset, input_true, epsilon=1e-8):
        total_loss = 0.0
        # First, do all the forward passes
        for digit in range(input_dataset.shape[0]):
            for example in range(input_dataset.shape[1]):
                output = self.forward_pass(input_dataset[digit][example])
                # See comment in self.train() for explanation of why we add␣
↪epsilon
                loss = -torch.sum(input_true[digit][example] * torch.log(output␣
↪+ epsilon))
                loss.backward()
                total_loss += loss.item()

        # Next, average each gradient over the number of training passes
```

```
        for i in range(len(self.hidden_layers)):
            self.hidden_layers[i].grad /= input_dataset.shape[1]
            self.biases[i].grad /= input_dataset.shape[1]

        self.output_layer.grad /= input_dataset.shape[1]
        self.output_bias.grad /= input_dataset.shape[1]

        # Now update the weights
        self.update_weights()

        return total_loss

    # Keep training until accuracy fails to improve for max_tries steps
    def train_until_cutoff(self, input_dataset, input_true, evaluation_dataset,␣
↪max_tries=20, debug_output=False, minibatch = False):
        losses = []
        accuracies = []
        step = 0

        best_accuracy = 0.0
        current_tries = 0
        while current_tries < max_tries:
            if minibatch:
                new_loss = self.train_minibatch(input_dataset, input_true)
            else:
                new_loss = self.train_dataset(input_dataset, input_true)
            losses.append(new_loss)
            new_accuracy = self.evaluate(evaluation_dataset)
            accuracies.append(new_accuracy)
            if new_accuracy > best_accuracy:
                current_tries = 0
                best_accuracy = new_accuracy

            # Can't use tqdm because I don't know the number of iterations,
            # SO says this should work
            if debug_output:
                sys.stdout.write(f'\rStep: {step} | Current Tries:␣
↪{current_tries} | Best: {best_accuracy} | Current: {new_accuracy} | Loss:␣
↪{new_loss}')
                sys.stdout.flush()

            step += 1
            current_tries += 1

        sys.stdout.write('\n')

        return losses, accuracies
```

```
    def train_minibatch(self, input_dataset, input_true):
        new_training_set = torch.zeros((10, 10, self.input_dim))
        new_true_set = torch.zeros((10, 10, 10))
        # For each digit
        for digit in range(input_dataset.shape[0]):
            # Randomly select 10 images
            selected_rows = list(np.random.permutation(input_dataset.
↪shape[1])[0:10])
            # and put them in the minibatch
            new_training_set[digit] = input_dataset[digit][selected_rows]
            # Don't forget the true vectors!
            new_true_set[digit] = input_true[digit][selected_rows]

        return self.train_dataset(new_training_set, new_true_set)

    # Predict a single value
    def predict(self, input_vector):
        with torch.no_grad():
            return torch.argmax(self.forward_pass(input_vector)).item()


    # Evaluate accuracy over an evaluation dataset
    # Shape should be (digits, num_examples, example_size),
    # which for these experiments is (10, 10, 49)
    def evaluate(self, evaluation_dataset):
        total = 0.0
        total_correct = 0.0
        for digit in range(evaluation_dataset.shape[0]):
            for example in range(evaluation_dataset.shape[1]):
                total += 1.0
                prediction = self.predict(evaluation_dataset[digit][example])
                if prediction == digit:
                    total_correct += 1.0

        return total_correct / total
```

6 Testing the model

Here, we test the model and observe its training accuracy over time. In order to investigate the dip in evaluation accuracy after a given peak, we first run the model for 2000 steps, well past the point at which it stops improving.

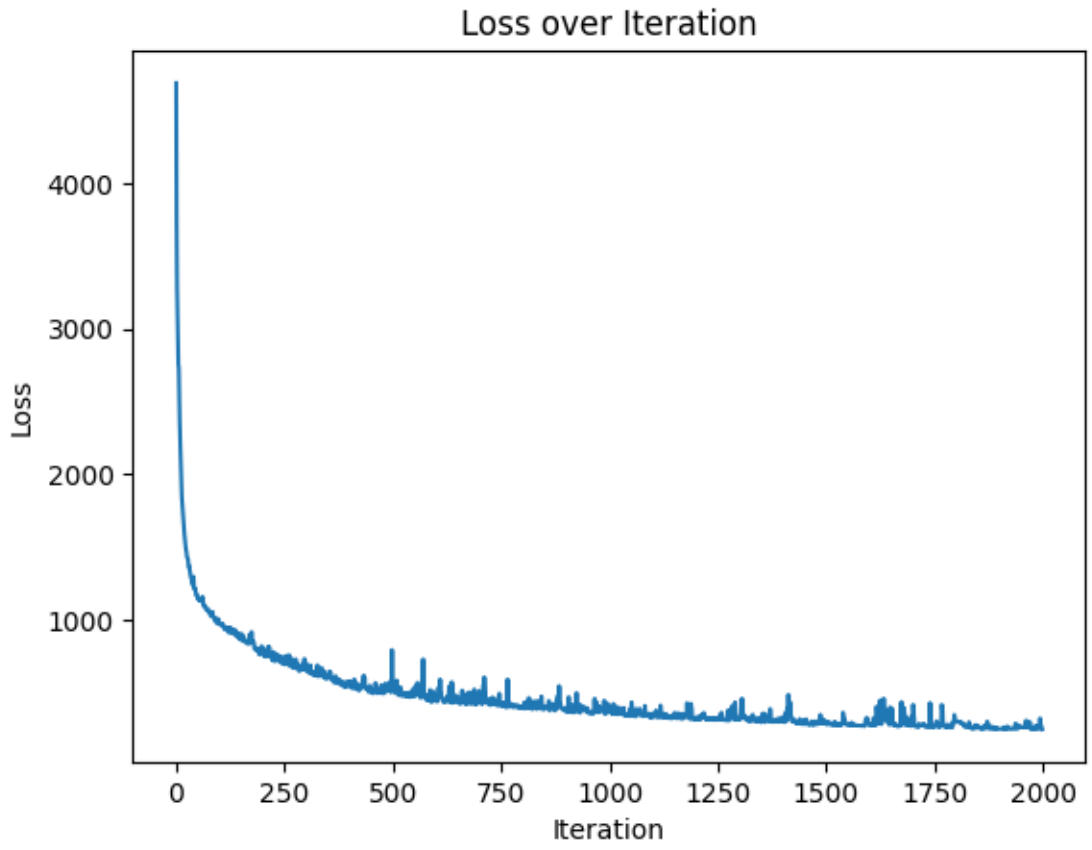
[16]:
```
from tqdm import tqdm

losses = []
```

```
perceptron_accuracies = []

steps = 2000

digit_perceptron = Perceptron(49, 2, 16, torch.atan, learning_rate = 0.02)

for _ in tqdm(range(steps)):
    losses.append(digit_perceptron.train_dataset(perceptron_training_data,␣
 ↪perceptron_training_true))
    perceptron_accuracies.append(digit_perceptron.
 ↪evaluate(perceptron_evaluation_data))
```

```
100%|      | 2000/2000 [09:20<00:00,  3.57it/s]
```

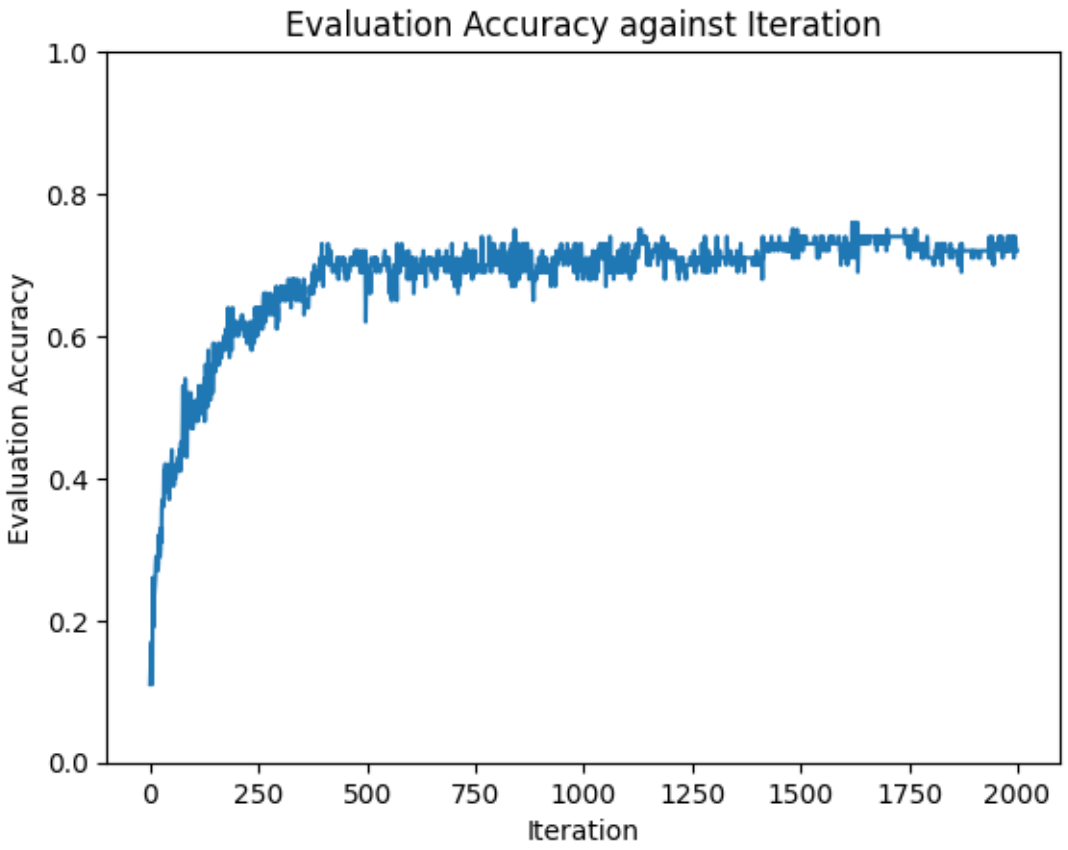
[17]:
```
plt.plot(range(len(losses)), losses)
plt.ylabel('Loss')
plt.xlabel('Iteration')
plt.title('Loss over Iteration')
```

[17]:
```
Text(0.5, 1.0, 'Loss over Iteration')
```



[18]:
```
plt.plot(range(len(perceptron_accuracies)), perceptron_accuracies)
plt.ylim(0.0, 1.0)
plt.ylabel('Evaluation Accuracy')
plt.xlabel('Iteration')
plt.title('Evaluation Accuracy against Iteration')
```

[18]:
```
Text(0.5, 1.0, 'Evaluation Accuracy against Iteration')
```



6.1 Observations

Here, we see evaluation accuracy dip slightly after reaching a peak at around iteration ~750 before oscillating around that value. We note that while the accuracy did decrease somewhat, it did not plummet; we use this observation to inform our cutoff criterion. It seems unlikely that we would achieve better accuracy by pushing past the 2000 mark.

6.2 Cutoff criterion

In order to cut off our training, we pass a hyperparameter max_tries to our perceptron's training method. At the beginning of training, we set the current_tries counter variable to zero. After each training iteration, we evaluate the model's evaluation accuracy against the previously recorded peak

accuracy. If the model has beaten its own record, we reset current_tries to zero. If not, we increment current_tries by 1. We cutoff training when the model fails to beat its previously recorded best accuracy for max_tries iterations. Based on the observations from the run above, as well as other training runs not shown, we believe that an intelligently selected max_tries hyperparameter will allow us to cut off training when the model is still near its peak accuracy without undershooting.

The one complication with this criterion is that minibatch-trained models tend to exhibit higher variance in their evaluation accuracy from run to run than full training set models, so we need to increase max_tries in order to compensate for this. This complicates the comparison between minibatch models and full models, since it is difficult to determine the effects that our new hyperparameter has had on the experiment; we aim to keep our minibatch models' training time lower than the training time for full models to ensure we are not simply giving them more slack.

7 Running the experiment

Here, we define the hyperparameters we will be using throughout the experiment, build and train the perceptrons, and record the results.

[29]:
```
import pandas as pd
from datetime import datetime

training_type_names = ['Full', 'Mini-batch']
hidden_layer_amounts = [2, 3, 4]
hidden_layer_sizes = [16, 32, 64]
activation_types = [torch.relu, torch.atan, torch.sigmoid]
activation_names = ['ReLu', 'Atan', 'Sigmoid']

#results = pd.DataFrame
#results.add
types = []
layer_num_labels = []
layer_size_labels = []
activations = []
execution_times = []
total_iterations = []
test_accuracy = []

# First, full batches
for layer_count in tqdm(range(len(hidden_layer_amounts))):
    for layer_size in range(len(hidden_layer_sizes)):
        for activation_type in range(len(activation_types)):
            next_perceptron = Perceptron(49, hidden_layer_amounts[layer_count],␣
 ↪hidden_layer_sizes[layer_size], activation_types[activation_type])
            # We do not need extreme accuracy here, datetime will do
            start_time = datetime.now()
```

```
            new_losses, new_accuracies = next_perceptron.
 ↪train_until_cutoff(perceptron_training_data, perceptron_training_true,␣
 ↪perceptron_evaluation_data, max_tries=20, debug_output=False,␣
 ↪minibatch=False)
            end_time = datetime.now()
            types.append(training_type_names[0])
            layer_num_labels.append(hidden_layer_amounts[layer_count])
            layer_size_labels.append(hidden_layer_sizes[layer_size])
            activations.append(activation_names[activation_type])
            execution_times.append((end_time - start_time).total_seconds())
            total_iterations.append(len(new_losses))
            new_accuracy = next_perceptron.evaluate(perceptron_test_data)
            test_accuracy.append(new_accuracy)

# Now minibatches
for layer_count in tqdm(range(len(hidden_layer_amounts))):
    for layer_size in range(len(hidden_layer_sizes)):
        for activation_type in range(len(activation_types)):
            next_perceptron = Perceptron(49, hidden_layer_amounts[layer_count],␣
 ↪hidden_layer_sizes[layer_size], activation_types[activation_type])
            start_time = datetime.now()
            new_losses, new_accuracies = next_perceptron.
 ↪train_until_cutoff(perceptron_training_data, perceptron_training_true,␣
 ↪perceptron_evaluation_data, max_tries=100, debug_output=False,␣
 ↪minibatch=True)
            end_time = datetime.now()
            types.append(training_type_names[1])
            layer_num_labels.append(hidden_layer_amounts[layer_count])
            layer_size_labels.append(hidden_layer_sizes[layer_size])
            activations.append(activation_names[activation_type])
            execution_times.append((end_time - start_time).total_seconds())
            total_iterations.append(len(new_losses))
            new_accuracy = next_perceptron.evaluate(perceptron_test_data)
            test_accuracy.append(new_accuracy)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

```
 33%|          | 1/3 [03:26<06:53, 206.53s/it]
```

```
 67%|       | 2/3 [06:16<03:05, 185.04s/it]
```

```
100%|      | 3/3 [09:45<00:00, 195.25s/it]
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

```
 33%|        | 1/3 [01:36<03:13, 96.97s/it]
```

```
 67%|       | 2/3 [03:48<01:57, 117.10s/it]
```

```
100%|      | 3/3 [05:52<00:00, 117.55s/it]
```

7.1 Results

Here, we build a Pandas dataframe to collect and analyze the results.

[30]:
```
data = {'Type': types, 'Layers': layer_num_labels, 'Layer Size':␣
 ↪layer_size_labels, 'Activation': activations, 'Time': execution_times,␣
 ↪'Iterations': total_iterations, 'Accuracy': test_accuracy}
df = pd.DataFrame(data=data)
df
```

[30]:

	Type	Layers	Layer Size	Activation	Time	Iterations	Accuracy
0	Full	2	16	ReLu	6.000886	22	0.18
1	Full	2	16	Atan	29.080719	95	0.59
2	Full	2	16	Sigmoid	40.940173	153	0.60
3	Full	2	32	ReLu	16.049314	52	0.27
4	Full	2	32	Atan	27.997217	100	0.74
5	Full	2	32	Sigmoid	32.894463	126	0.65
6	Full	2	64	ReLu	4.899778	20	0.10
7	Full	2	64	Atan	14.788104	54	0.71
8	Full	2	64	Sigmoid	33.815546	125	0.71
9	Full	3	16	ReLu	5.900219	20	0.15
10	Full	3	16	Atan	31.677809	93	0.55
11	Full	3	16	Sigmoid	5.803709	20	0.03
12	Full	3	32	ReLu	6.794970	20	0.15
13	Full	3	32	Atan	23.886321	70	0.75
14	Full	3	32	Sigmoid	36.042895	113	0.55
15	Full	3	64	ReLu	6.665277	20	0.15
16	Full	3	64	Atan	18.361584	53	0.74
17	Full	3	64	Sigmoid	34.802931	106	0.71
18	Full	4	16	ReLu	7.700433	20	0.10
19	Full	4	16	Atan	45.067983	113	0.67
20	Full	4	16	Sigmoid	15.420143	39	0.27
21	Full	4	32	ReLu	7.756204	20	0.10
22	Full	4	32	Atan	36.149270	89	0.71
23	Full	4	32	Sigmoid	10.885658	29	0.21
24	Full	4	64	ReLu	7.218679	20	0.14

25	Full	4	64	Atan	36.915671	87	0.82
26	Full	4	64	Sigmoid	42.005840	104	0.73
27	Mini-batch	2	16	ReLu	3.969850	100	0.10
28	Mini-batch	2	16	Atan	14.212722	364	0.61
29	Mini-batch	2	16	Sigmoid	11.297341	295	0.61
30	Mini-batch	2	32	ReLu	3.813542	101	0.10
31	Mini-batch	2	32	Atan	17.564721	437	0.65
32	Mini-batch	2	32	Sigmoid	17.841669	460	0.81
33	Mini-batch	2	64	ReLu	4.692308	132	0.10
34	Mini-batch	2	64	Atan	11.648956	289	0.81
35	Mini-batch	2	64	Sigmoid	11.876622	303	0.84
36	Mini-batch	3	16	ReLu	4.649059	100	0.10
37	Mini-batch	3	16	Atan	16.671530	336	0.55
38	Mini-batch	3	16	Sigmoid	27.473595	578	0.71
39	Mini-batch	3	32	ReLu	4.326637	100	0.10
40	Mini-batch	3	32	Atan	14.220106	291	0.66
41	Mini-batch	3	32	Sigmoid	21.385022	457	0.82
42	Mini-batch	3	64	ReLu	4.540968	100	0.10
43	Mini-batch	3	64	Atan	19.257941	374	0.59
44	Mini-batch	3	64	Sigmoid	18.600242	401	0.81
45	Mini-batch	4	16	ReLu	5.825655	100	0.10
46	Mini-batch	4	16	Atan	6.737685	123	0.49
47	Mini-batch	4	16	Sigmoid	17.431572	293	0.38
48	Mini-batch	4	32	ReLu	5.021848	100	0.22
49	Mini-batch	4	32	Atan	14.382130	250	0.49
50	Mini-batch	4	32	Sigmoid	31.544056	565	0.76
51	Mini-batch	4	64	ReLu	5.198819	100	0.12
52	Mini-batch	4	64	Atan	21.804624	357	0.61
53	Mini-batch	4	64	Sigmoid	16.443465	296	0.83

7.2 Results

The most striking result is that ReLu based models failed to move away from baseline accuracy in almost every case. We examine the cause of this later on in the results.

Both the sigmoid and atan activation functions performed decently depending upon other hyperparameters. Sigmoid generally performed somewhat better than Atan, but only slightly.

Perceptrons with 64 neurons per layer also performed well; most of the highest test accuracy classifiers had this hyperparameter set. Notably, the number of hidden layers did not significantly impact the accuracy.

Training the minibatch based classifiers took significantly less time than full batch classifiers, but despite this, they achieved similar accuracy numbers.

It is somewhat surprising that wider perceptrons outperformed deeper ones, but the training data is quite limited, so this is perhaps not incredibly surprising.

Next, we examine the failure of the ReLU based models more closely.

7.3 Examining ReLU

First, we define a ReLU classifier to examine what may be going wrong. We use a 2-hidden-layer, 16 neuron per layer model to make the output easier to analyse.

[19]:
```
relu_perceptron = Perceptron(49, 2, 16, torch.relu, learning_rate=0.02,␣
↪min_init=-1.0, max_init=1.0)
```

Next, we manually run the untrained classifier's layers against a sample input and observe the results. Note that we stop before applying softmax.

[20]:
```
current_vec = perceptron_training_data[0][0]
for i in range(len(relu_perceptron.hidden_layers)):
  current_vec = torch.matmul(relu_perceptron.hidden_layers[i], current_vec) +␣
↪relu_perceptron.biases[i]

current_vec = torch.matmul(relu_perceptron.output_layer, current_vec) +␣
↪relu_perceptron.output_bias

current_vec
```

[20]:
```
tensor([ -912.3993,  -288.3857,   -29.6673, -1581.9635,  1761.6512,  1343.2122,
          138.8708,   543.1945,  -773.9258,   115.6192],
       grad_fn=<AddBackward0>)
```

We note that these values are extremely high. When we apply softmax we see:

[21]:
```
current_vec = torch.softmax(current_vec, dim=0)
current_vec
```

[21]:
```
tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.], grad_fn=<SoftmaxBackward0>)
```

The perceptron is extremely confident in its prediction of the image, despite only just being initialized. If we try to find the loss of this vector, we will take the logarithm of 0, which evaluates to -inf; multiplying this by the 0 in the true vector results in NaN, so we add a small delta before doing so; we then backpropagate and examine the gradient.

[22]:
```
current_vec = current_vec + 1e-8
loss = -torch.sum(perceptron_training_true[0][0] * torch.log(current_vec))
loss.backward()
relu_perceptron.output_layer.grad
```

[22]:
```
tensor([[-0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0.,
-0.],
        [0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
        [0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
        [0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
        [0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
        [0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
```

```
[0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
[0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
[0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.],
[0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., -0., -0., 0., 0., 0.]])
```

We see that the gradient here is uniformly zero.

The network that we have defined is quite small and shallow, so we do not believe this is a case of the vanishing gradient problem. Rather, we believe that the zero-gradient is a result of the extremity of the pre-softmax outputs of the hidden layers, since unlike atan or sigmoid, the values of each hidden layer are not squished to the range [-1.0,1.0]. This essentially implies that the network has already been shoved so far off into a plateua, there is no gradient that meaningfully reduces the loss function. To test this, we use the min_init and max_init hyperparameters we introduced in the definition of the Perceptron class to test our hypothesis. We also decrease the learning rate, since the values we will be adjusting are already much smaller.

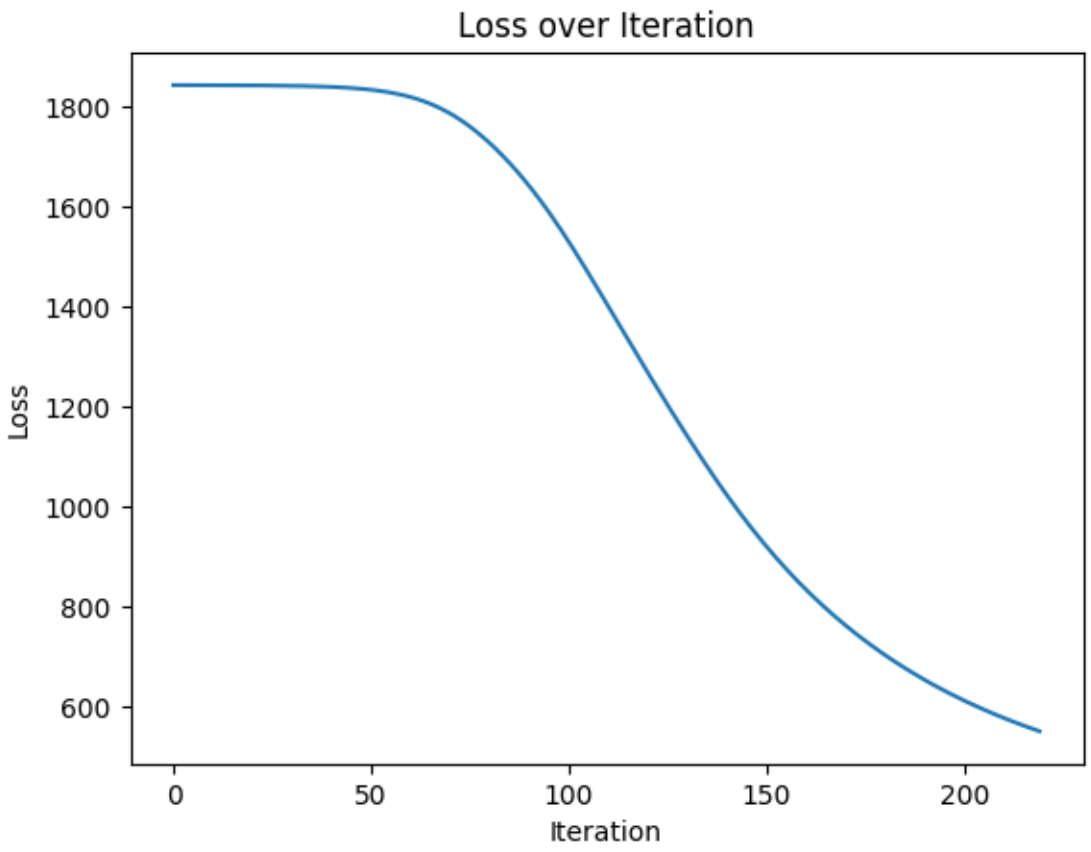
[26]:
```
relu_squished_test = Perceptron(49, 2, 16, torch.relu, learning_rate = 0.0002,␣
↪min_init=-.001, max_init=0.001)

relu_losses, relu_accuracies = relu_squished_test.
↪train_until_cutoff(perceptron_training_data, perceptron_training_true,␣
↪perceptron_evaluation_data, max_tries=50, debug_output=True, minibatch=False)
```

```
Step: 219 | Current Tries: 49 | Best: 0.75 | Current: 0.75 | Loss:
551.7081687089521
```

7.4 Results

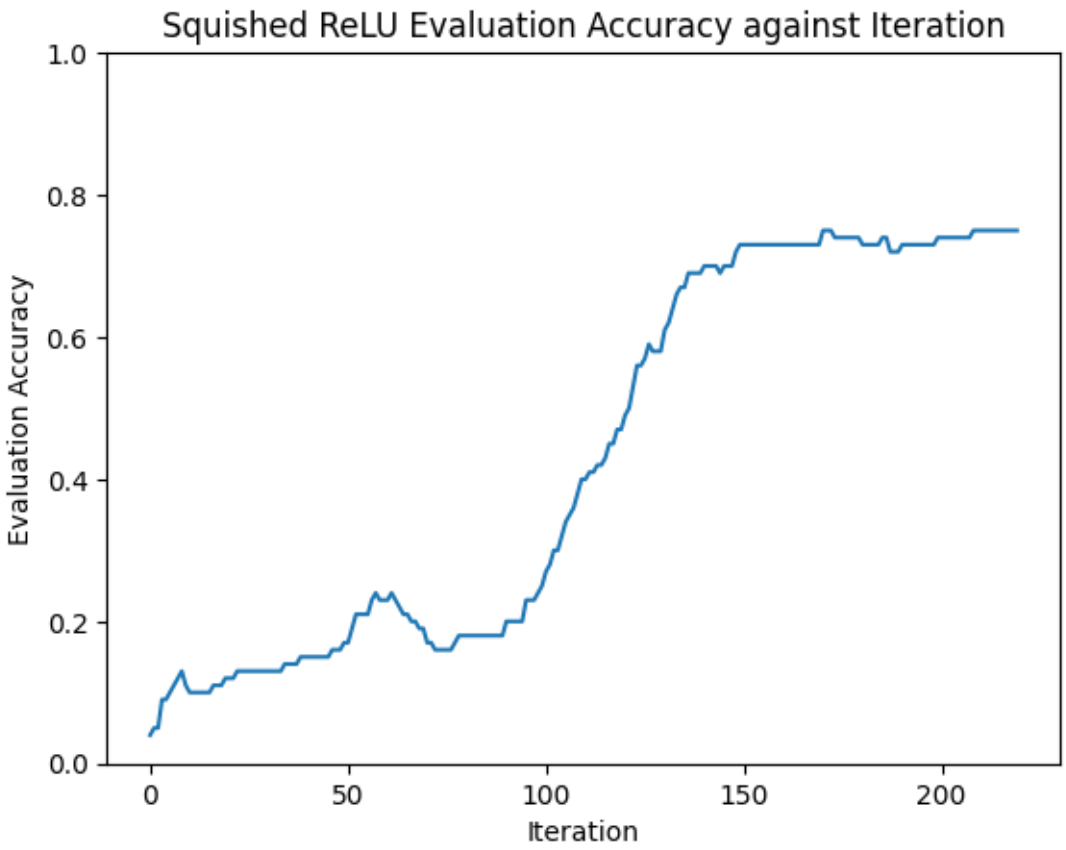
[27]:
```
plt.plot(range(len(relu_losses)), relu_losses)
plt.ylabel('Loss')
plt.xlabel('Iteration')
plt.title('Loss over Iteration')
```

[27]:
```
Text(0.5, 1.0, 'Loss over Iteration')
```



[37]:
```
plt.plot(range(len(relu_accuracies)), relu_accuracies)
plt.ylim(0.0, 1.0)
plt.ylabel('Evaluation Accuracy')
plt.xlabel('Iteration')
plt.title('Squished ReLU Evaluation Accuracy against Iteration')
```

[37]:
```
Text(0.5, 1.0, 'Squished ReLU Evaluation Accuracy against Iteration')
```



After changing the min and max values of the random initialized values in neuron tensors, we are able to train ReLU activated perceptrons that are at least capable of achieving some kind of reasonable accuracy, although they seem relatively unstable and do not consistently converge like this. In particular, the dip in evaluation accuracy after the initial climb happens consistently, and the classifiers do not always recover as this particular example did.

Another method of solving this issue might be to somehow normalize the output of each hidden layer after applying ReLU to the range [-1.0,1.0]. However, this would have to been done carefully, as one of the primary advantages of ReLU in the first place is its computational efficiency. It could also be worth investing what effect the small delta we add to the logits to prevent NaNs has on the stability of the classifiers. These considerations, however, are left for potential follow on experiments.