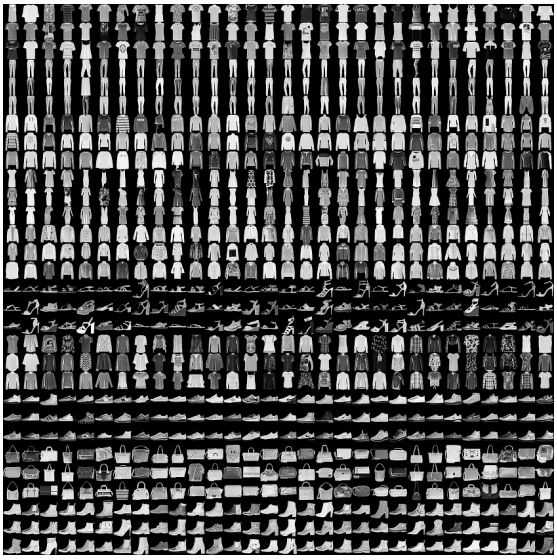
CV Novelty Detection Project

UC Berkeley, Machine Learning Decal
May 10, 2019
Jared Zhao

Understanding the Problem

Given a set of training images, our goal is to be able to detect (amongst another set of images), if we have seen any of these images in our training set. This is a problem called 'Novelty Detection'. However, I had questions about how we should decide if 2 items are the same. For an example, we know that 2 shoes are both classified as shoes. After seeing one of the shoes, when we are presented with the other, do we say that we have seen shoes before, or that we haven't seen that specific type of shoe yet? This should be easy enough to figure out through data exploration.

Our data will come from the MNIST Fashion dataset:



Data Exploration

I began my project by first exploring what type of images are contained within the dataset. I felt that, if I could visualize the dataset better, then I would also be able to build a model better. It would aid in my intuition as to how I would tackled the problem, since I had no idea how to start. So, I started by writing a script that could render the vectors that were outputted through the dataloader that was provided in our starter script.

In [2]:
```
def display_array(array, scale, interpolation, should_save, directory, f
ile_name):
    if (should_save):
        if not os.path.exists(directory):
            os.makedirs(directory)

    array = cm.binary(array)
    image = Image.fromarray(np.uint8(array * 255))
    image = image.convert('RGB')
    if (should_save):
        image.save(directory + file_name + '.png')
    if (interpolation != None):
        image = image.resize((image.width * scale, image.height * scale
), interpolation)
    else:
        image = image.resize((image.width * scale, image.height * scale
))
    display(image)

def generate_stitched_image(directory, output_path, start='start', end=
'end'):
    dataframe = pd.DataFrame()
    filenames = []
    iteration = []

    for filename in os.listdir(directory):
        filenames.append(filename)
        as_arr = filename.split('_')
        iteration.append(int(as_arr[0]))

    dataframe['filenames'] = filenames
    dataframe['iteration'] = iteration

    dataframe = dataframe.sort_values('iteration')
    dataframe = dataframe.reset_index(drop=True)
    image_sizes = Image.open(directory + dataframe['filenames'][0]).size

    if (end != 'end'):
        dataframe = dataframe.drop([i for i in range(end, len(dataframe[
'filenames']))])

    if (start != 'start'):
        dataframe = dataframe.drop([i for i in range(0, start)])

    result = Image.new("RGB", (len(dataframe['filenames']) * image_sizes
[0], image_sizes[1]))

    for i in tqdm(range(len(dataframe))):
        filename = dataframe.iloc[i]['filenames']

        input_image = Image.open(directory + filename)

        result.paste(input_image, (i * image_sizes[0], 0, (i + 1) * imag
e_sizes[0], image_sizes[1]))

    result.save(output_path)
```

I chose to render the transformed data from the dataloader, because this would most closely represent what data I could input into my model.

It appears that the task of the project is to be able to tell apart different articles of clothing, not just different types of clothing. In other words, we must be able to 'see' the difference between shoes, and not just be able to classify shoes as shoes.

This means that the labels (T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot) useless!

This, however, also gave me another idea. In class, we were warned about overfitting our data: detection of shoes only works with the training data, but doesn't work with other shoes that the model hasn't seen yet before. This is exactly the behavior that I want! What if I purposely overfitted my model to my data?

Model Choice

Rationale

So, continuing with my idea of purposely overfitting the training data, so that my model only works with the training set, and not with any other set, I decided to start by experimenting with manually setting the weights of a model. I decided to start with a convolutional layer, where there are 60,000 channels (the size of the training set), and where the kernel size was 28x28 (the image sizes).

Architecture & Implementation Details

I manually set the weights of each kernel to the inverse weights of each training image, and the bias to -784. What this should do is, when an image is passed through the convolutional layer, it is compared against 60,000 kernels, each containing the weights of one of the images in the training set (an overfit kernel, if you will). Since the equation for evaluating each kernel is:

$$z = b + \sum_{i=1}^{N} w_i a_i$$

the expected value of the summation is 784 for an exactly matching image and kernel, so if I divide each image by 784, and also set the bias to -1, I should get a z value of 0. So I designed an experiment around this concept, and tested some hyperparameters.

In [ ]:
```
class Net(nn.Module):
    def __init__(self, threshold):
        super(Net, self).__init__()
        self.threshold = threshold
        self.conv = nn.Conv2d(1, 60000, 28)

    def forward(self, x):
        x = torch.abs(self.conv(x))
        x[x > self.threshold] = 0
        x = x.nonzero()
        if (x.shape[0] == 1):
            return torch.tensor(0)
        return torch.tensor(1)

def train(net, bias):
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data

        net.conv.weight[i] = (i + 1) * bias / inputs / 784
        net.conv.bias[i] = -(i + 1) * bias
```

I discovered that, although the summation portion is expected to be 1 after it is divided by 784, so the bias portion could be -1, this actually gave poor results. The resultant values were slightly off from 0. I'm guessing that this comes down to the inaccuracies of single accuracy floating point operations that PyTorch relies on.

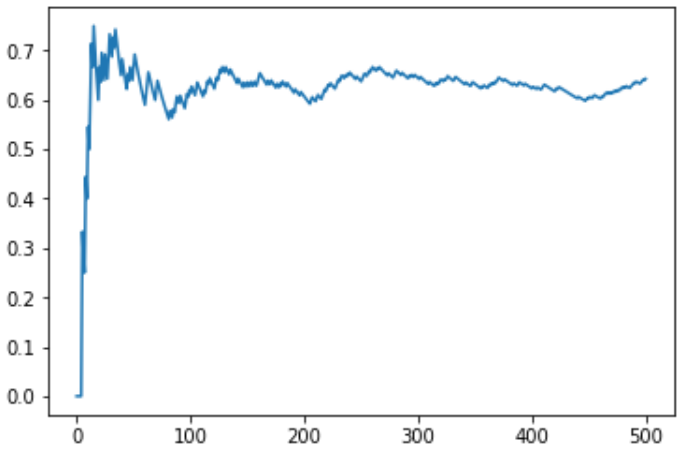
Using this method that I've outlined, I was only getting around 25% accuracy in the validation set. This means that I could have gotten better results if I had just randomly returned either 0 or 1! This is a very bad solution! Seeing that Facebook had just announced and released BoTorch and AX Platform for Hyperparameter tuning though, I couldn't pass up this opportunity to experiment with it.

So, I set up the AX testing framework, in the hopes of tuning my hyperparameters (threshold & bias) for better results.

In [ ]:
```
best_parameters, best_values, experiment, model = optimize(
        parameters=[
          {
            "name": "threshold",
            "type": "range",
            "bounds": [0.0000001, 1.0],
          },
          {
            "name": "bias",
            "type": "range",
            "bounds": [0.0001, 10.0],
          },
        ],
        # Booth function
        evaluation_function=lambda p: train_and_evaluate(p["threshold"],
p["bias"]),
        minimize=True,
    )
```

My tuned hyperparameter output was: {'threshold': 0.1058580279439695, 'bias': 3.379729986190796}, and gave a validation score of 76.529%. This score is low, but this model is extremely good at rejecting images that it hasn't seen. In essense, it has prioritized the 'novelty' part of 'novelty detection'.

Accuracy Graph:



K-Fold Validation

For K-Fold Validation, I decided to use a k of 10, since this would mean 7000 items are used for validation each time, giving a good averaged result that isn't too heavily biased due to low sample count.

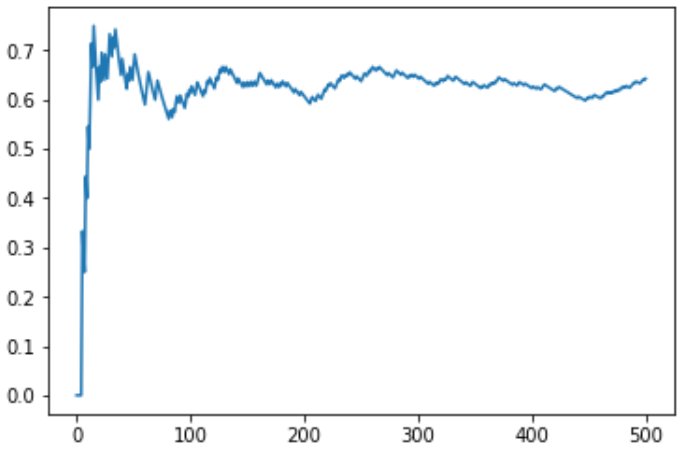
In [7]:
```
def k_fold_validation(k):
    net = Net()
    for fold in range(k):
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data

            if (i % fold != 0):
                train_model(net)

        for i, data in enumerate(train_loader, 0):
            inputs, labels = data

            if (i % fold != 0):
                train_model(net)

        for i, data in enumerate(train_loader, 0):
            inputs, labels = data

            if (i % fold == 0):
                validate_model(net)

        for i, data in enumerate(train_loader, 0):
            inputs, labels = data

            if (i % fold == 0):
                validate_model(net)
```

The result of my K-Fold Validation was still roughly the same, at a 74.829% accuracy.

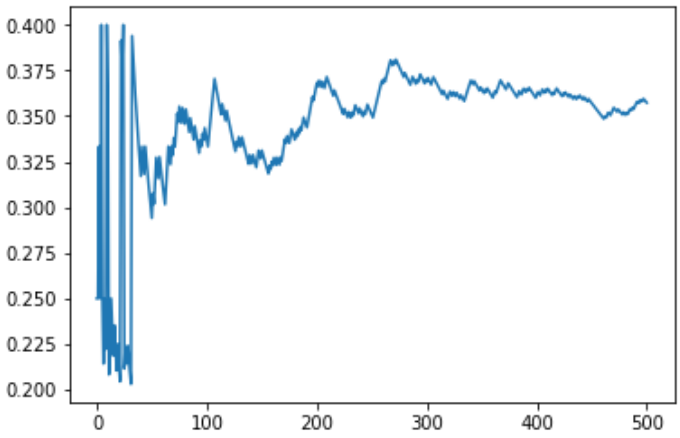
Hyper-parameter Tuning

Since I have used Facebook's new BoTorch and AX to tune my hyper parameter, I am fairly confident that they are near the global optimum. I will use 3 examples that the Adaptive Experimentation Platform used while optimizing my hyper parameters that I believe are representative of my model, and how the hyperparameters affect it.

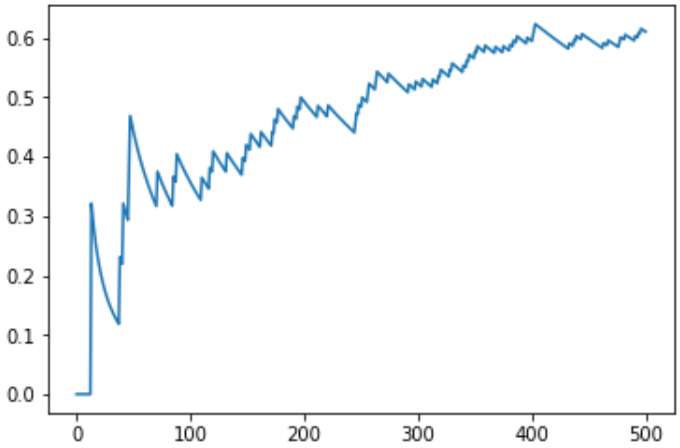
1. {'threshold': 0.1058580279439695, 'bias': 3.379729986190796}



1. {'threshold': 0.0112039852508437, 'bias': 0.634095729823543}



1. {'threshold': 0.6432805702398502, 'bias': 6.203472093852355}



Final Results

The best results I got, after experimenting with many different models and hyper-parameters was about a 76% accuracy in the provided validation data. I know that this validation data isn't representative of the final data that we will be tested on though, since this set only included data that was also contained within the training data set. I would be curious to find out how well I do compared to the dataset that we will be tested on.