Neural Network Combined with K Nearest Neighbors Applied to Computer Vision

Department of Scientific Computing Practicum Project

Scott Williams

Advised by Gordon Erlebacher and Brian Bartoldson

1 Introduction

1.1 Goal of Project

Generalization error is a measure of how accurately an algorithm is able to predict outcome values for previously unseen data.[5] In this project, the generalization problem is being approached in a similar manner to Khandelwal et al. in the paper Generalization through Memorization: Nearest Neighbor Language Models. However, here it will be applied to computer vision. In the paper by Khandelwal et al., including k-nearest neighbors in addition to the neural network yielded better results with less training. The k-nearest neighbors language model trained on 100M tokens outperformed the language model lacking k-nearest neighbors trained on 3B tokens.[10] The implication is that models can be trained using smaller data sets to learn representations; furthermore, applying the trained models in tandem with k-nearest neighbors can improve accuracy on unseen data.[10] We wanted to see if these results stayed consistent in the realm of image classification. A neural network was combined with k-nearest neighbors and tested on previously unseen data. We found that this method resulted in a 2.190% increase in accuracy.

1.2 Generalization Problem

A phenomenon that can hinder the success of generalization is overfitting. Overfitting occurs when a neural network has been trained excessively to the point where it has learned the training data too well.[6] At first, it may seem like this level of learning would be desirable. However, while

neural networks that have learned training data too well do not always perform poorly, they can sometimes turn out to be non-functioning on testing data. Generalization is a major problem in machine learning. If models for computer vision, natural language processing, and more were able to be trained on one data set and then applied to any problem in the relevant field it would be revolutionary. There are two types of generalization; in-distribution generalization and out-of-distribution generalization. In-distribution generalization occurs when a single data set is split into training and testing data. While the testing data is unseen by the network during training, it is still part of the same data set or distribution. Out-of-distribution generalization occurs when training data and testing data come from different distributions.[17] Out-of-distribution generalization is a very difficult problem so here we focused on in-distribution generalization.

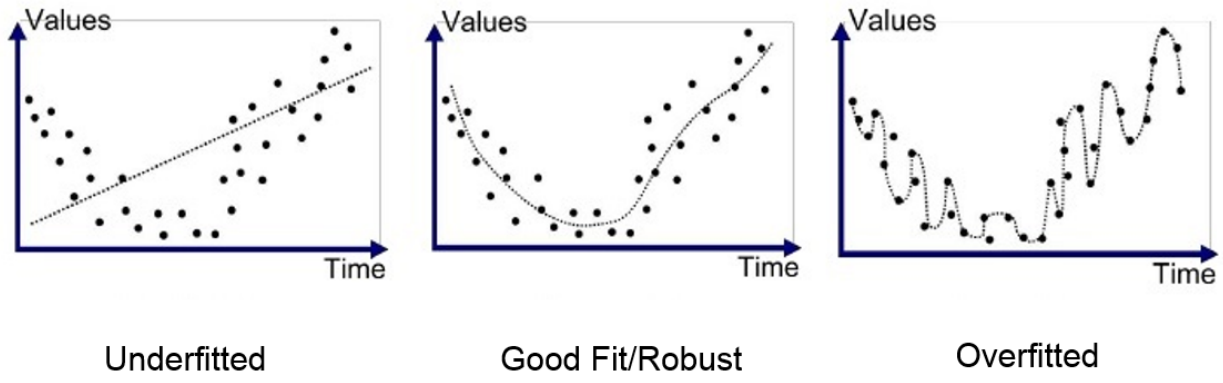


Figure 1: Example of overfitting.[27]

2 Background

2.1 Why is Machine Learning Important?

Machine Learning has become a popular solution to problems that typically required an abundance of human effort. With the explosion in popularity came a push in many different disciplines to become "AI Enabled." The financial industry uses machine learning to analyze and predict market behavior. Medical research has started to entertain the idea of using machine learning as a supplement for cancer detection. Auto companies have been looking into self driving cars as the

next advancement in automation. Machine learning continues to innovate and perform important tasks in many different fields. One might think that these applications of machine learning are wonderful; but why we should care about machine learning? The benefit of this technology is that organizations can build or make use of pre-trained machine learning models that will allow them to yield faster results due to eliminating the need for potentially slow human work.[1]

Now that the importance of machine learning has been established, what exactly goes into it? Perhaps even more important than the ability to program a machine learning algorithm is the acquisition of data. Data is what allows machine learning to happen. After gathering data relevant to the problem, one splits the data into a training set and a testing set. The process of machine learning takes place as the computer begins to learn based on the training data provided. The model will then be applied to the testing data to see how well it has been trained. "At its most basic level, machine learning involves computers processing a large amount of data to predict outcomes."[2] There are many different methods to utilize machine learning. These include supervised learning, unsupervised learning, semisupervised learning, and reinforcement learning.[1] Supervised learning involves training on labeled data. Labeled data is data in which the desired output is known.[1] For example, a picture of a cat would be labeled as cat and a picture of a dog would be labeled as dog. For this project we will be using supervised learning.

2.2 Neural Networks

A neural network is a computing system that identifies patterns in data sets and can be used to cluster, classify, and continuously learn and improve.[3] Neural networks, as referenced in their name, are meant to mimic the brain. The structure of a neural network involves a system of layers. First, there is the input layer into which the data is fed. Following the input layer, the information goes through processing within the hidden layer; this is where the pattern recognition takes place.[3] Finally, the output layer is where the prediction/classification is given. The loss is then calculated between the neural network's prediction and the true label of the input. Loss is a measure of error that tells you how far from correct the network's prediction is.[24] The error is then fed back through the neural network to adjust weights thus helping it get closer to an accurate prediction.[2]6 Neural networks are significant in the field of machine learning because they are powerful and can produce accurate results due to their learning capabilities.[4]

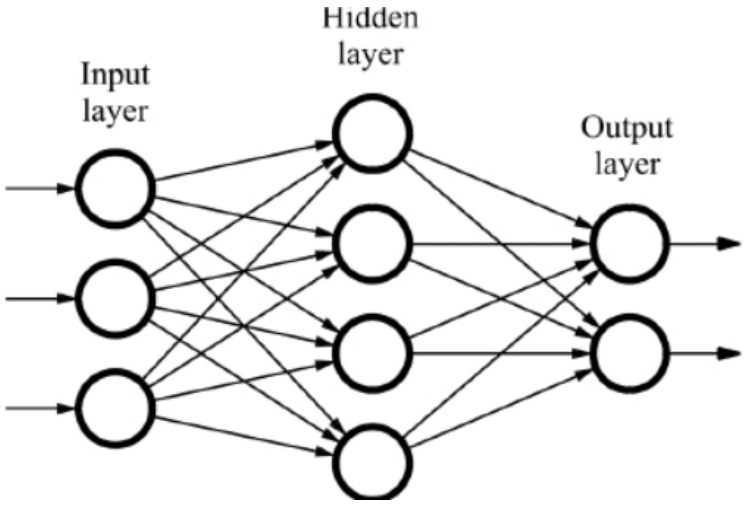


Figure 2: Example of a neural network.[23]

2.3 Computer Vision

Computer vision is a field within machine learning that uses deep learning models to train computers using digital images and videos to accurately identify and classify objects.[7] There are a variety of applications for computer vision. Areas such as biology, the automotive industry, and surveillance have started to apply computer vision techniques. Species identification, autonomous vehicles, and event detection are all applications of computer vision.[8] For the purpose of this project, computer vision will be used for image classification.

2.4 K-Nearest Neighbors

K-nearest neighbors will be implemented to aid in classification on unseen data. K-nearest neighbors is a classification method in which the input is classified based on the label of the data points(the k-nearest neighbors) closest to it. K is a parameter chosen by the user that represents the number of neighbors examined.[9] The nearest neighbors are found using different distance metrics such as the Euclidean distance for continuous data or the overlap metric for discrete data.[9]

2.5 Representation Learning

Representation learning allows a system to automatically discover the representations needed for feature detection or classification from raw data.[11] While there is no exact definition of a feature,

they can be summarized, at a high level, as "interesting" or significant parts of an image.[12] In other words, representation learning allows computers to learn the representations of data that make it easiest to extract useful information.[13] This technique has been applied to problems such as speech recognition, object recognition, and natural language processing.[13] For this project, the neural network built will make use of representation learning for image classification. Within the hidden layer, the neural network will learn a representation of the input and use this representation for classification.[11] According to Bengio et al., representation learning has led to remarkable successes in both industry and academia.[13] Given these results, image classification should not be a task that representation learning cannot handle.

2.6 Ensembles

The combination of k-nearest neighbors with a neural network can be classified as an ensemble. In machine learning, ensemble learning uses multiple learning algorithms to obtain better predictive results.[14] Ensembles make use of a group decision when it comes to classification. Instead of only having one "vote" as to what the output label should be, ensembles have multiple "votes." Just like in real life, having multiple "people" attempt a problem together will usually lead to better results than having someone work alone. "Research conducted by Hansen and Salamon found that predictions made by a set of classifiers is often more accurate than predictions made by the best single classifier."[15] Ensemble learning will work especially well for this project since the generalization ability of an ensemble is typically stronger than a single learner.[15] According to Dietterich, there are three reasons why ensembles generalize more effectively. Machine learning can be described as searching a hypothesis space for the most accurate hypothesis.[15] With this in mind, Dietterich claims ensembles generalize better than single learners because

- A lack of training data can lead to an algorithm finding multiple answers within the hypothesis space. Ensembles will "vote" on these answers and find the most likely label. This reduces the classification error.[16]

- It may prove difficult for a single learning algorithm to find the correct label in a broad hypothesis space. Making use of an ensemble allows for searching the hypothesis space from multiple starting points. An approximation to the correct label can then be made.[16]

- Ensembles can expand the space of representable functions by taking a weighted sum of the hypotheses which enables classification on more data.[16]

2.7 Data

For this project we will be using the CIFAR-10 data set.[20] CIFAR-10 consists of 60,000 images of 10 different classes. The 10 classes being airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. We split this data set into 50,000 images for training and 10,000 images for testing.

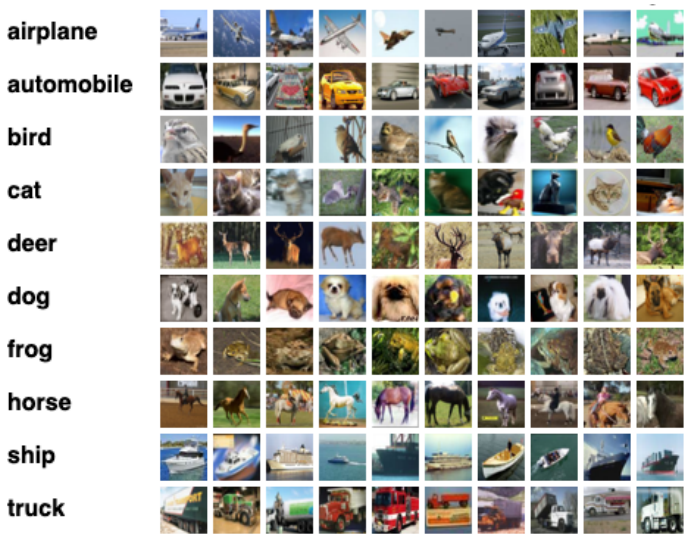


Figure 3: Examples of CIFAR-10 images.[21]

3 Approach

3.1 High Level Description of the Approach

A convolutional neural network will be used for image classification. We will be using the CIFAR-10 data set to train the neural network. After the process of deep learning, the accuracy of the neural network will be measured. The trained neural network will then be applied to the testing data set to see how the base network generalizes.

K-nearest neighbors will be implemented as an additional classifier for building the ensemble. Here we are using k = 7 neighbors as it led to the best classification accuracy using k-nearest

neighbors alone. For the distance metric, we have chosen Euclidean Distance.

The convolutional neural network's probability distribution for each class will then be combined with k-nearest neighbor's probability distribution for each class in a weighted average. The argmax of this new probability distribution will be the ensemble's prediction for the image's class.

3.2 Programming Environment

To carry out the tasks detailed above, Google Colab was used. Google Colab is an in-browser Python notebook interface. This was an advantageous choice as it enabled storage of large data sets in the cloud and computational power through a cuda enabled GPU. For twelve hours at a time, users have access to a free NVIDIA Tesla K80 through Google Colab. GPU computing is especially useful in the domain of machine learning. GPUs excel at matrix operations due to their multi core structure. Different cores can simultaneous handle different parts of a calculation to speed up the time of computation. CPUs can only do these calculations step by step in a linear fashion which leads to major slow down on large calculations.

3.3 Convolutional Neural Network

A convolutional neural network was constructed to classify images belonging to the CIFAR-10 dataset. Convolutional neural networks make use of an operation called convolution in the hidden layer instead of regular matrix multiplication.[18] However, the term convolution in this context actually refers to something called cross correlation.[18] Cross correlation measures the similarity between two functions.[19] Convolutional neural networks simultaneously decrease dimensionality while keeping the most important features of the images.[22]

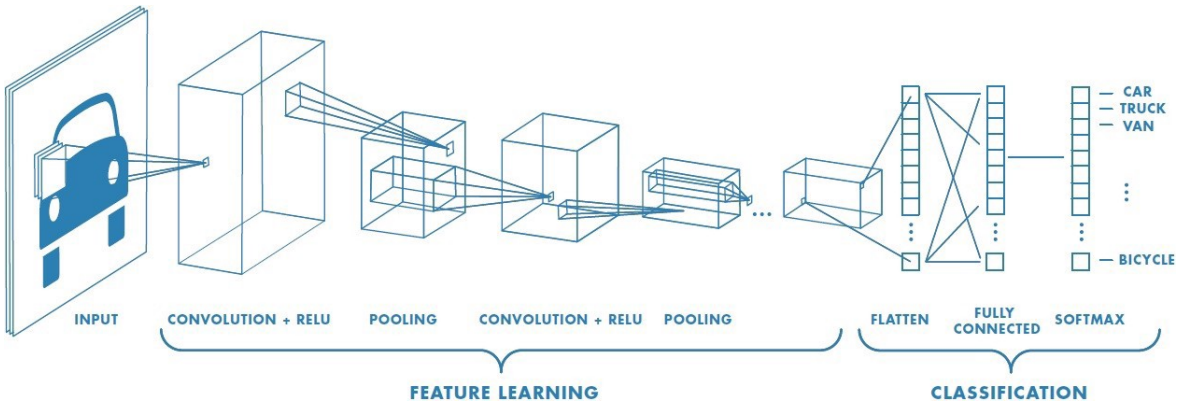


Figure 4: Example of a convolutional neural network

3.4 Convolutional Neural Network Code

The image classifier was implemented using PyTorch. The base structure came from the PyTorch tutorial Deep Learning with PyTorch: A 60 Minute Blitz. First, the necessary packages for classification and visualization needed to be imported.

The packages function as follows:

- matplotlib.pyplot was imported to visualize the distance between different classes in different layers of the convolutional neural network.
- numpy was imported so that numpy functions could be used after transforming PyTorch tensors into NumPy arrays.
- linalg was imported from numpy so that the norm function could be used to normalize vectors when calculating distance.
- distance was imported from scipy.spatial so that the Euclidean and cosine functions could be used to calculate the Euclidean distance and cosine similarity between vectors.
- KNeighborsClassifier was imported from sklearn.neighbors so that k-nearest neighbors could be used.
- torch was imported so that code could be run on a GPU for efficiency. Torch was also used to load the CIFAR-10 training and testing data sets into variables that could be used with the network.

- torchvision was imported to download the CIFAR-10 training and testing data sets. Additionally it was used to make a grid to visualize a batch of images from the training data set.
- torchvision.transforms was imported to convert the data to tensors and to normalize the data.
- torch.nn and torch.nn.functional were imported so that the convolutional neural network could be built. Torch.nn provided access to convolutional layers, pooling, and linear layers. Torch.nn also provided the .backward() command which automatically performs backpropagation. Additionally, torch.nn provided built in loss functions. Torch.nn.functional provided built in activation functions.
- torch.optim was imported so that built in methods for optimization could be used. Torch.optim also allowed for automatic updating of the weights.

```
[ ]  %matplotlib inline
     import matplotlib.pyplot as plt
     import numpy as np
     from numpy import linalg as LA
     from scipy.spatial import distance as dist
     from sklearn.neighbors import KNeighborsClassifier
     import torch
     import torchvision
     import torchvision.transforms as transforms
     import torch.nn as nn
     import torch.nn.functional as F
     import torch.optim as optim
```

Figure 5: Importing the necessary packages

Due to the computationally intense task we wanted to perform, we wanted to make use of GPU computing. Device is defined as whatever cuda enabled GPU is available. As one can see, a cuda enabled GPU was found. Cuda:0 is an Nvidia Tesla K80 offered through Google Colab.

```
[ ]  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

     print(device)

     cuda:0
```

Figure 6: Verifying that the GPU has been recognized

To download the CIFAR-10 training and testing datasets, torchvision was utilized. Here a transform was composed that converts the PIL images in the CIFAR-10 dataset into tensors. Furthermore, the data was normalized with a mean and standard deviation of 0.5 for each channel. The CIFAR-10 training data was then downloaded. The training data with a batch size of 4 was stored in the variable trainloader. The CIFAR-10 testing data was then downloaded. The testing data with a batch size of 4 was stored in the variable testloader. A batch is a group of images that is passed into the convolutional neural network all at once. Since we are taking advantage of a GPU, this will speed up the execution of the code. A batch of multiple images takes advantage of the GPU's strength in parallel computing as explained in the Programming Environment section. Trainloader and testloader were used to train and validate the performance of the convolutional neural network respectively. The classes tuple stores every predictable class that an image could be within CIFAR-10.

```
[ ] transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    # Loading in the CIFAR10 dataset and making the training dataset
    trainset = torchvision.datasets.CIFAR10(root='./drive/My Drive/Colab Notebooks/Practicum/data', train=True, download=True, transform=transform)

    trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)

    # Loading in the CIFAR10 dataset and making the testing dataset
    testset = torchvision.datasets.CIFAR10(root='./drive/My Drive/Colab Notebooks/Practicum/data', train=False, download=True, transform=transform)

    testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)


    # Defining the classes that could be predicted
    classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Figure 7: Loading the data

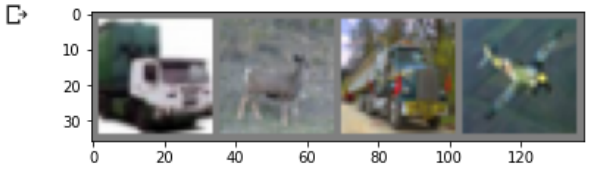
To visualize a batch of the data, random images from trainloader were plotted on a grid using matplotlib, numpy, and torchvision. The classes of each image were printed as well to show what kinds of images belong to each class.

```
[ ]  # Function to unnormalize and show images from the dataset
     def imshow(img):
         img = img/2 + 0.5
         npimg = img.numpy()
         plt.imshow(np.transpose(npimg, (1, 2, 0)))
         plt.show()

     # Grab random images and labels from the training dataset
     dataiter = iter(trainloader)
     images, labels = dataiter.next()

     # Show the random images
     imshow(torchvision.utils.make_grid(images))

     # Print the labels of the images
     print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```




```
truck  deer truck plane
```

Figure 8: A batch of the data

Before getting into the code for the convolutional neural network, here are explanations of concepts that you will see in the description of the code.

- Kernel – The filter that is applied to the image. Performs something similar to matrix multiplication and reduces dimension of image when no padding is present.[22]

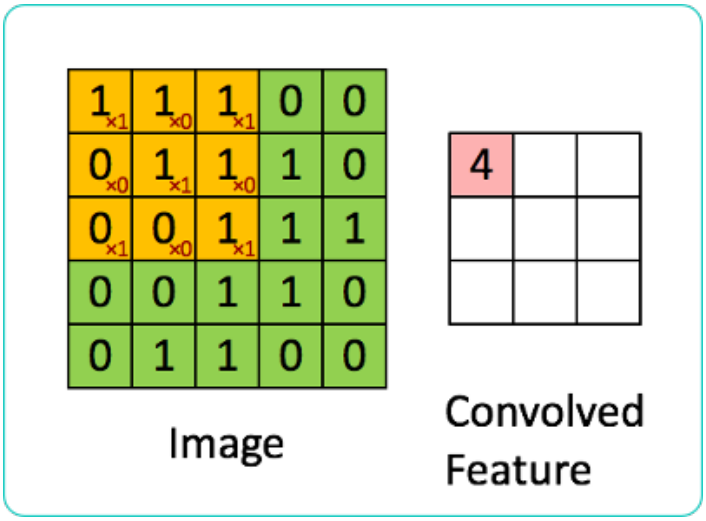


Figure 9: Example of a kernel applied to an image.[22]

- Stride - The amount of pixels the kernel moves to the right when filtering through the image.[22]
- Padding – Adding zeros around the matrix. This prevents loss of size.[22]

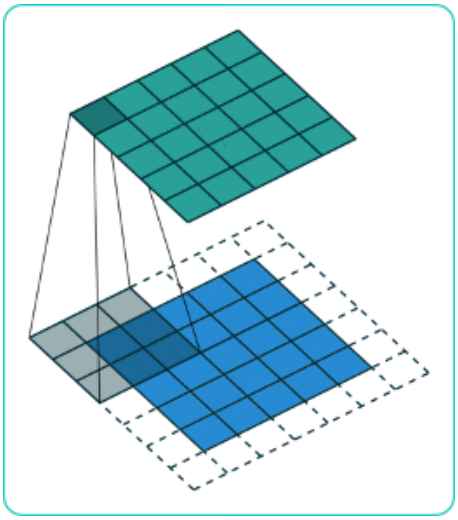


Figure 10: Example of an image with padding.[22]

- Pooling – A technique used to even further reduce the dimension of the image.[22] In this project, max pooling was used. After the image has been passed through a convolution layer, a filter is applied to the resulting matrix. A new matrix is constructed where each element of the new matrix is the max element of the region that the filter was applied to.[22]

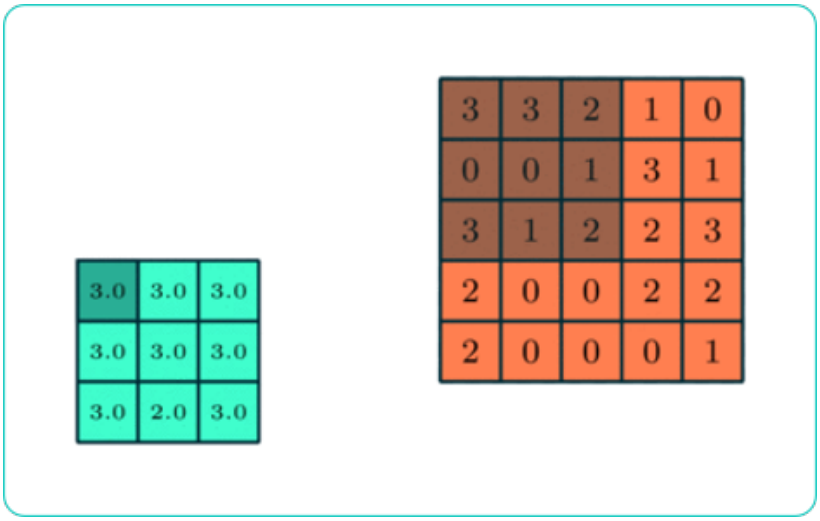


Figure 11: An example of max pooling.[22]

The convolutional neural network was implemented inside the class Net. The images pass through a first convolutional layer with 3 in channels, 32 out channels, and a 5x5 kernel. This makes the image 32x28x28. The image is then passed through a pooling layer which alters the image into a 32x14x14. After passing through a second convolutional layer that has 32 in channels, 64 out channels, and a 5x5 kernel, the image is reshaped into a 64x10x10. Pooling applied again which makes the shape of the image 64x5x5. The image is passed through a third convolutional layer with 64 in channels, 128 out channels, and a 5x5 kernel. This reshapes the image into 128x1x1. Finally, the image is passed into a linear layer that has 128 in features and 10 out features. This linear layer combined with the ReLu activation function will output the probability distribution of the image belonging to a class.

```
[ ] class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.conv1 = nn.Conv2d(3, 32, 5)
            self.pool = nn.MaxPool2d(2, 2)
            self.conv2 = nn.Conv2d(32, 64, 5)
            self.conv3 = nn.Conv2d(64,128,5)
            self.fc1 = nn.Linear(128 * 1 * 1, 10)

        def forward(self, x):
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = F.relu(self.conv3(x))
            x = x.view(-1, 128 * 1 * 1)
            x = F.relu(self.fc1(x))

            return x

        def get_1st_layer(self,x):
            x = self.pool(F.relu(self.conv1(x)))

            return x

        def get_2nd_layer(self,x):
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))

            return x
```

```
[ ]     def get_3rd_layer(self,x):
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = F.relu(self.conv3(x))

            return x

        def get_4th_layer(self,x):
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = F.relu(self.conv3(x))
            x = x.view(-1, 128 * 1 * 1)
            x = F.relu(self.fc1(x))

            return x



    net = Net()

    net.to(device)
```

Figure 12: Defining the CNN using PyTorch

```
[ ] criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Figure 13: Defining the loss and optimizer

3.5 CNN Training and Testing

At this point, the training of the convolutional neural network began. The variables correct and total were initialized to zero so that training accuracy could be calculated within the training loop. The convolutional neural network was trained for 5 epochs. An epoch is a complete run through of the training data set. Images were passed into the convolutional neural network and the loss was calculated between the network's prediction and the true label. Backpropagation was performed and the weights were updated accordingly. The training accuracy was calculated by finding out how many images the network correctly classified divided by the total amount of images all multiplied by 100. The convolutional neural network achieved 77% training accuracy.

```
[ ] for epoch in range(5):

        delta_1, c_delta_1 = representation_dist_1(train_3, test_4, test_3)
        delta_2, c_delta_2 = representation_dist_2(train_3, test_4, test_3)
        delta_3, c_delta_3 = representation_dist_3(train_3, test_4, test_3)
        delta_4, c_delta_4 = representation_dist_4(train_3, test_4, test_3)

        delta_vals_1.append(delta_1)
        delta_vals_2.append(delta_2)
        delta_vals_3.append(delta_3)
        delta_vals_4.append(delta_4)

        c_delta_vals_1.append(c_delta_1)
        c_delta_vals_2.append(c_delta_2)
        c_delta_vals_3.append(c_delta_3)
        c_delta_vals_4.append(c_delta_4)

        running_loss = 0.0

        correct = 0
        total = 0

        for i, data in enumerate(trainloader, 0):
            # Get the inputs for the CNN. Data is a list of [inputs, labels]
            inputs, labels = data[0].to(device), data[1].to(device)

            # zero the parameter gradients to prevent gradients from being accumulated to existing gradients
            optimizer.zero_grad()
```

```
[ ]         # Predict based on inputs
            outputs = net(inputs)

            # Calculate the loss
            loss = criterion(outputs, labels)

            # Perform backprop
            loss.backward()

            # Update the weights
            optimizer.step()

            running_loss += loss.item()

            # Calculating the training loss

            # Getting the index of the highest energy. The higher the energy the more confident in the label
            _, predicted = torch.max(outputs.data, 1)


            # Calculating total amount of labels
            total += labels.size(0)

            # Calculating the total amount of images where the predicted label matched the true label
            correct += (predicted == labels).sum().item()
```

```
        if i%2000 == 1999: # Print every epoch
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss/2000))
            running_loss = 0.0

    print(correct/total * 100)
```

```
[ ] delta_1, c_delta_1 = representation_dist_1(train_3, test_4, test_3)
    delta_2, c_delta_2 = representation_dist_2(train_3, test_4, test_3)
    delta_3, c_delta_3 = representation_dist_3(train_3, test_4, test_3)
    delta_4, c_delta_4 = representation_dist_4(train_3, test_4, test_3)

    delta_vals_1.append(delta_1)
    delta_vals_2.append(delta_2)
    delta_vals_3.append(delta_3)
    delta_vals_4.append(delta_4)

    c_delta_vals_1.append(c_delta_1)
    c_delta_vals_2.append(c_delta_2)
    c_delta_vals_3.append(c_delta_3)
    c_delta_vals_4.append(c_delta_4)

    print('Finished Training')
```

Figure 14: Training the convolutional neural network

```
[ ] print('Training accuracy of the network: %d %%' % (100 * correct/total))
```

```
Training accuracy of the network: 77 %
```

Figure 15: CNN's training accuracy

To test the convolutional neural network's performance, the testing dataset was used as input. The testing accuracy was calculated in the same way as training accuracy. The convolutional neural network achieved 70.320% testing accuracy.

```
[ ]  correct = 0
     correct_ens = 0
     total = 0

     with torch.no_grad():
         for data in testloader:
             # Saving images and labels from data
             images, labels = data[0].to(device), data[1].to(device)

             # Predicting the labels
             outputs = net(images)

             # Getting the index of the highest energy. The higher the energy the more confident in the label
             _, predicted = torch.max(outputs.data, 1)

             # Probability Distribution from CNN
             CNN_probs = outputs


             # KNN prediction probability distribution
             # with torch.no_grad():
             KNN_probs = neigh.predict_proba( [ x.flatten() for x in net.get_3rd_layer(images).cpu().numpy() ] )
```

```
[ ]
             # Ensemble Prediction
             lam = .75

             ensemble_prob = (1-lam)*CNN_probs.detach().cpu().numpy() + (lam)*KNN_probs

             ensemble_labels = np.argmax(ensemble_prob,1)

             # Calculating total amount of labels
             total += labels.size(0)

             # Calculating the total amount of images where the predicted label matched the true label
             correct += (predicted == labels).sum().item()
             correct_ens += (ensemble_labels == labels.cpu().numpy()).sum()

     print('Accuracy of the network on the 10,000 test images: %.3f %%' % (100 * correct/total))
     print('Accuracy of the ensemble on the 10,000 test images: %.3f %%' % (100 * correct_ens/total))
```

Figure 16: Testing the convolutional neural network

```
Accuracy of the network on the 10,000 test images: 70.320 %
```

Figure 17: Convolutional neural network accuracy

3.6 K-Nearest Neighbors Code

K-nearest neighbors was implemented using the built in Scikit-learn function.

```
[ ]  neigh = KNeighborsClassifier(n_neighbors=7, metric='euclidean')
     neigh.fit(train_image, train_label)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=7, p=2,
                     weights='uniform')
```

Figure 18: Creating the k-nearest neighbors classifier

4 Experiments

4.1 K-Nearest Neighbors

Decisions needed to be made in regards to k-nearest neighbors before attempting to construct the ensemble. Specifically, should we use k-nearest neighbors with the raw images or the representations? Therefore, we pose the question; do we expect images to be close together just because they are of the same class in pixel space? Our intuition led us to believe the answer would be no. In pixel space, the entries in the image matrix represent color values and have no significance as to what the subject of the image actually is. However, in representation space, the convolutional neural network has learned what features map to a specific class.

To prove this to ourselves, we decided to run some trials. First, k-nearest neighbors was used with the base images in the CIFAR-10 data set. As expected, k-nearest neighbors performed poorly and only achieved 9% accuracy or approximately the accuracy of taking a random guess(10 %) on the data. This test was run using only 500 of the test images due to programming environment restrictions. Due to the size of the raw images, k-nearest neighbors ran incredibly slow and Google Colab often disconnected during the calculation.

However, one question remained. Which layer should be used with k-nearest neighbors to obtain an increase in accuracy when constructing the ensemble? To gain some insight before making this decision, we constructed an experiment.

4.1.1 Distance in Representation Space Experiment

Perhaps one noticed in figure 13, at the start and end of the for loop there are distance arrays. These arrays are filled with distances in representation space. The following steps are how the experiment was performed.

- Grab 3 images. One image of class 3 from the training data, 1 image of class 3 from the testing data, and 1 image from class 4 of the testing data.
- Calculate the distance between the images of class 3 from the training and testing data. Call this distance_3.

- Calculate the distance between the image of class 3 from the training data and the image of class 4 from the testing data. Call this distance_4.
- Calculate the distance between distance_3 and distance_4

```
[ ] def representation_dist_3(train_3, test_4, test_3):
      with torch.no_grad():

        rep_train_3 = net.get_3rd_layer(train_3.cuda())
        rep_test_4 = net.get_3rd_layer(test_4.cuda())
        rep_test_3 = net.get_3rd_layer(test_3.cuda())

        rep_train_3 = rep_train_3.flatten()
        rep_test_4 = rep_test_4.flatten()
        rep_test_3 = rep_test_3.flatten()

        rep_train_3 = rep_train_3.cpu()
        rep_test_4 = rep_test_4.cpu()
        rep_test_3 = rep_test_3.cpu()

        dist_3 = dist.euclidean(rep_train_3/LA.norm(rep_train_3), rep_test_3/LA.norm(rep_test_3))
        dist_3_4 = dist.euclidean(rep_train_3/LA.norm(rep_train_3), rep_test_4/LA.norm(rep_test_4))

        cdist_3 = dist.cosine(rep_train_3, rep_test_3)
        cdist_3_4 = dist.cosine(rep_train_3, rep_test_4)

        delta = LA.norm(dist_3 - dist_3_4)

        c_delta = LA.norm(cdist_3 - cdist_3_4)

        return delta, c_delta
```

Figure 19: Calculating the distances in the third layer. This same process was replicated for each layer in the convolutional neural network

We expected the distances to start out close as the convolutional neural network had not learned meaningful features to distinguish different classes. However, as training accuracy rises, we expect the distances to grow as the convolutional neural network will have learned more meaningful representations.

We used two distance metrics to verify our hypothesis. Euclidean distance with normalized vectors (since we do not care about magnitude but only position in space) and cosine similarity. At the end of training, we plotted epochs vs distances. We do not believe this is a perfect experiment for deciding which layer to use in k-nearest neighbors. There are some oddities that we are not sure how to explain. Why does the third layer spike and then decrease to similar behavior as the first layer? Why do the different distance metrics portray the same general behavior but have slight differences at peak values? These are questions that we did not have time to look into and

would require more experimentation. However, the graphs do exhibit the aforementioned behavior of distance growing as the network learned representations. We did not want to use the final layer of the convolutional neural network as we felt that would be creating two classifiers out of the same representations and would not lead to any significant insight as to what layer the convolutional neural network starts to identify significant features.

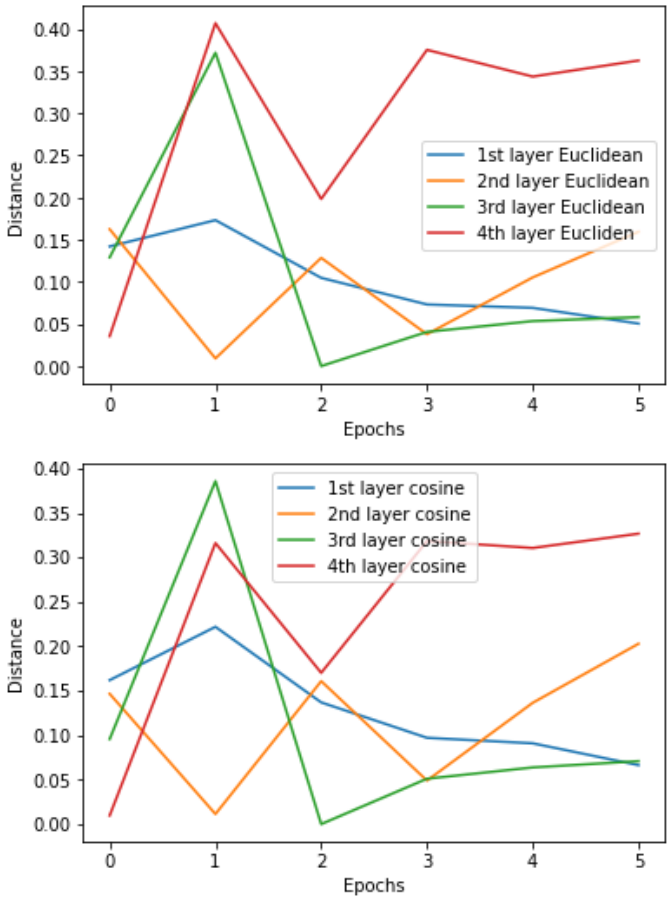


Figure 20: Plots of distances in different layers of the convolutional neural network

4.2 K-Nearest Neighbors in Representation Space

The experiment led us to choose the third layer due to the large spike in distance that it portrayed during training. To test if the third layer would work, k-nearest neighbors was performed using these representations of the data. As expected, representations of like classes were close in representation space and the accuracy improved to 70.66%.

```
[ ]  train_image = []
     train_label = []
     test_image = []
     test_label = []

     for i, data in enumerate(trainloader, 0):
       with torch.no_grad():
         # Train set images and labels
         inputs, labels = data[0].to(device), data[1].to(device)

         train_image += [x.flatten() for x in net.get_3rd_layer(inputs).cpu().numpy()]
         train_label += [x for x in labels.cpu()]




     for i, data in enumerate(testloader, 0):
       with torch.no_grad():
         # Test set images and labels
         inputs, labels = data[0].to(device), data[1].to(device)

         test_image += [x.flatten() for x in net.get_3rd_layer(inputs).cpu().numpy()]
         test_label += [x for x in labels.cpu()]



     train_image = np.array(train_image)
     train_label = np.array(train_label)
     test_image = np.array(test_image)
     test_label = np.array(test_label)
```

Figure 21: Creating the data set of third layer representations to use with k-nearest neighbors

```
[ ]  correct = 0
     total = 0

     for i in range(0, 10000):

       predicted = neigh.predict([test_image[i]])
       # print(neigh.predict_proba([test_image[i]]))
       total = total + 1

       if predicted == test_label[i]:
         correct = correct + 1

     knn_accuracy = correct/total * 100
```

```
[ ]  print(knn_accuracy)

     70.66
```

Figure 22: Evaluating k-nearest neighbor's performance on the testing data

4.3 Constructing the Ensemble

To construct the ensemble, a method similar to Khandelwal et al. was used. Although they applied the method to natural language processing, we believed the method applied to computer vision would produce similar results. At its core, the concept of using representations to increase accuracy with less training is the same. However, there was no guarantee that their method would generalize across different fields of machine learning. The probability distributions from the convolutional neural network and k-nearest neighbors were combined in a weighted average.

$$\text{Ensemble Probability Distribution} = (1 - \lambda) * CNN\_probabilities + \lambda * KNN\_probabilities \quad (1)$$

$$0 \leq \lambda \leq 1 \quad (2)$$

To extract the predicted class from the ensemble, the argmax of the probability distribution was taken.

$$Prediction = \arg\max(\text{Ensemble Probability Distribution}) \quad (3)$$

Through trial and error, the optimal value of λ was determined to be 0.75.

5 Conclusions

5.1 Results

After finding the optimal value of $\lambda = 0.75$, we found this resulted in a 2.190% increase in testing accuracy. As stated before, the testing accuracy of the convolutional neural network alone was 70.320%. The accuracy of k-nearest neighbors alone was 70.66%. The testing accuracy of the ensemble was 72.510%. A 2.190% increase in accuracy could seem mundane in the context of a base accuracy of 70%. However, if this method was applied to a more robust network it could potentially push the accuracy into state of the art range.

```
[ ] correct = 0
    correct_ens = 0
    total = 0

    with torch.no_grad():
        for data in testloader:
            # Saving images and labels from data
            images, labels = data[0].to(device), data[1].to(device)

            # Predicting the labels
            outputs = net(images)

            # Getting the index of the highest energy. The higher the energy the more confident in the label
            _, predicted = torch.max(outputs.data, 1)

            # Probability Distribution from CNN
            CNN_probs = outputs


            # KNN prediction probability distribution
            # with torch.no_grad():
            KNN_probs = neigh.predict_proba( [ x.flatten() for x in net.get_3rd_layer(images).cpu().numpy() ] )


            # Ensemble Prediction
            lam = .75

            ensemble_prob = (1-lam)*CNN_probs.detach().cpu().numpy() + (lam)*KNN_probs
```

```
[ ]         ensemble_labels = np.argmax(ensemble_prob,1)

            # Calculating total amount of labels
            total += labels.size(0)

            # Calculating the total amount of images where the predicted label matched the true label
            correct += (predicted == labels).sum().item()
            correct_ens += (ensemble_labels == labels.cpu().numpy()).sum()

    print('Accuracy of the network on the 10,000 test images: %.3f %%' % (100 * correct/total))
    print('Accuracy of the ensemble on the 10,000 test images: %.3f %%' % (100 * correct_ens/total))
```

```
Accuracy of the network on the 10,000 test images: 70.320 %
Accuracy of the ensemble on the 10,000 test images: 72.510 %
```

Figure 23: Testing the base convolutional neural network and the ensemble

5.2 Future Work

There are some areas that need work in this project. Scikit-learn does not have any GPU support. This was somewhat of a hindrance as some calculations had to be sent to CPU. It would be beneficial to construct our own GPU enabled k-nearest neighbors method using PyTorch.

Due to time constraints, the final convolutional neural network used was still fairly small. As previously stated, it would be interesting to see how this method performs on a more powerful neural network. At one point, the convolutional neural network was made deeper but performed worse. This behavior was unexpected and would need more experimentation/education to figure out why the accuracy was lower. An area of study that was suggested by my advisor to potentially remedy this was batch normalization. The larger a neural network gets, the more difficult it is to train. Therefore, batch normalization, a technique used to improve performance and stability of a network, appears to be a potential solution to the problem.

References

1. Machine Learning: What it is and why it matters. (n.d.). Retrieved from https://www.sas.com/en_us/insights/analytics/machine-learning.html

2. 1.5 Machine learning, statistics, data science, robotics, and AI. (2017). In *Machine learning: The power and promise of computers that learn by example* (p. 24). The Royal Society.

3. Neural Networks - What are they and why do they matter? (n.d.). Retrieved from https://www.sas.com/en_us/insights/analytics/neural-networks.html

4. *Machine learning: The power and promise of computers that learn by example* (p. 94). (2017). The Royal Society.

5. Generalization error. (2019, September 03). Retrieved from https://en.wikipedia.org/wiki/Generalization_error

6. Generalization and Overfitting. (2017, January 24). Retrieved from https://wp.wwu.edu/machinelearning/2017/01/22/generalization-and-overfitting/

7. Computer vision: What it is and why it matters. (n.d.). Retrieved January 28, 2020, from https://www.sas.com/en_us/insights/analytics/computer-vision.html

8. Computer vision. (2020, January 27). Retrieved January 28, 2020, from https://en.wikipedia.org/wiki/Computer_vision

9. K-nearest neighbors algorithm. (2020, January 04). Retrieved January 28, 2020, from https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

10. Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., Lewis, M. (2019, November 01). Generalization through memorization: Nearest neighbor language models. Retrieved January 28, 2020, from https://arxiv.org/abs/1911.00172

11. Feature learning. (2019, December 13). Retrieved January 28, 2020, from https://en.wikipedia.org/wiki/Feature_learning

12. Feature detection (computer vision). (2019, December 18). Retrieved January 28, 2020, from https://en.wikipedia.org/wiki/Feature_detection_(computer_vision)

13. Bengio, Y., Courville, A., Vincent, P. (2014, April 23). Representation learning: A review and new perspectives. Retrieved January 28, 2020, from https://arxiv.org/abs/1206.5538

14. Ensemble learning. (2020, January 05). Retrieved January 28, 2020, from https://en.wikipedia.org/wiki /Ensemble_learning

15. Zhou, Z. (2009). Ensemble Learning. Retrieved January 27, 2020, from https://cs.nju.edu.cn/zhouzh /zhouzh.files/publication/springerEBR09.pdf

16. Dietterich, T. G. (2000). Ensemble Methods in Machine Learning. Retrieved January 27, 2020, from https://web.engr.oregonstate.edu/ tgd/publications/mcs-ensembles.pdf

17. Sun, Y., Wang, X., Liu, Z., Miller, J., Efros, A. A., Hardt, M. (2019, October 25). Test-time training for out-of-distribution generalization. Retrieved January 28, 2020, from https://arxiv.org/abs /1909.13231

18. Convolutional neural network. (2020, February 21). Retrieved from https://en.wikipedia.org/wiki/ Convolutional_neural_networkDesign

19. Part 2: Convolution and Cross-Correlation - G. Jensen [Video file]. (2015, February 5). Retrieved February 23, 2020, from https://www.youtube.com/watch?v=MQm6ZP1F6ms

20. Krizhevsky, A. (2009, April 8). Learning Multiple Layers of Features from Tiny Images (Tech.). Retrieved April 27, 2020, from https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf

21. (n.d.). Retrieved April 27, 2020, from https://www.cs.toronto.edu/ kriz/cifar.html

22. Saha, S. (2018, December 17). A comprehensive guide to convolutional neural networks - the eli5 way. Retrieved April 23, 2020, from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

23. What is a neural network? (n.d.). Retrieved April 28, 2020, from https://databricks.com/glossary/neural-network

24. Neural networks. (n.d.). Retrieved April 29, 2020, from https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

25. Backpropagation. (2020, April 27). Retrieved April 29, 2020, from https://en.wikipedia.org/wiki/Backpropagation

26. A beginner's guide to backpropagation in neural networks. (n.d.). Retrieved April 29, 2020, from https://pathmind.com/wiki/backpropagation

27. Bhande, A. (2018, March 18). What is underfitting and overfitting in machine learning and how to deal with it. Retrieved April 29, 2020, from https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76