Testing the Performance Impact of Noisy Images on a LeNet-based CNN Architecture

By:

Timmy Tushar Rajan Susai Rajan

Abstract

In this project, I have developed a convolutional neural network (CNN) for multi-class classification. I am training and testing the performance of my CNN using the CIFAR-10 dataset [1]. The CNN architecture that I implemented is a modified version of the LeNet architecture [2]. I have established two goals. The first is to evaluate how noisy images impact the performance of my CNN. This is accomplished by adding noise to the test images and evaluating the accuracy of the CNN's prediction using several metrics, such as precision, recall, and sparse categorical accuracy. The second goal is to evaluate the performance of my CNN when an autoencoder is used to denoise/reconstruct the original image. My autoencoder is a modified version of the one that can be found in the TensorFlow documentation for autoencoders [3]. I am using four different types of noise in the images, which I refer to as uniform noise, concentrated noise, removing pixels, and random noise. Through testing, I discovered that my CNN performs worse with noisy test images and that an autoencoder's ability to denoise images is a function of the amount and type of noise applied to the images.

Introduction

The main goal of my project is to determine how noisy images affect the performance of my CNN. I anticipate that my CNN will have a significantly worse performance when classifying noisy images than clean images. I also anticipate that I can use an autoencoder that can be trained to infer features from the images and denoise/reconstruct them to improve the performance of my CNN. It is very likely that when I am conducting image classification, the dataset of images that I am dealing with consists of noisy images, so I wanted to develop an architecture that can help denoise these images to improve the performance of classification. Therefore, I am first seeing the performance drop-off of my CNN when trying to predict the classes of noisy test images and then using an autoencoder to denoise the test images to improve the performance of my CNN. I used the TensorFlow package in Python to develop my neural networks. TensorFlow is a Python library that allows me to develop, train, and test the architecture for my neural networks. Some of the other packages I used are Matplotlib to plot some of the graphs in the results section, NumPy to work with the image arrays, Seaborn to generate a confusion matrix, and Numba to speed up the process of adding noise to the test dataset. I also leveraged different sources to develop the architecture for my CNN and autoencoder. My CNN is based on a LeNet architecture [2] because it gave me the best performance out of all the ones I researched and experimented with. My autoencoder is based on an architecture I found in the TensorFlow documentation [3]. I made modifications to both architectures to better suit my dataset.

Initial Experimentation

I was initially planning on implementing a hetero-associative network instead of an autoencoder to denoise images. However, in my development of the network, I realized that the network had to be retrained for each image that had to be denoised. If the network was trained with even just two images, it was not able to recall either of them. The weight matrix was also large (3072 x 3072) which made the weight training process long. The inefficiency and the abysmal results pushed me to look for alternative denoising techniques, and I ultimately decided to implement a

convolutional autoencoder. I chose an autoencoder because it can denoise images that it has never seen before more accurately than a hetero-associative network can.

```
@njit
def noise_filter(images):
  filtered_images = []
  for i in range(len(test_images)):
    w = np.outer(test_images[i].flatten(), test_images[i].flatten())
    np.fill_diagonal(w, 0)
    image = images[i].flatten().reshape((-1, 1))
    image = np.dot(w, image)
    image = image / image.max()
    filtered_images.append(image.reshape(test_images[i].shape))
  return filtered_images
denoised_images = noise_filter(noisy_test_images)
```

Figure 1: Heteroassociative Network

Methodology Overview

First, I load the dataset and split it into training and testing data. I then normalize the images to a value between 0 and 1. For the baseline measure of my CNN's performance, I will first evaluate the CNN by using the original training and test images as inputs. The performance metric for my CNN is sparse categorical accuracy. I will then evaluate my CNN with noisy images. I am adding noise to only the test images to avoid too many variables in my experimentation. For each type of noise, I will first run the noisy test images directly into my CNN and evaluate its performance. Then I will run the noisy test images through the autoencoder to denoise the images and use its output as input to the CNN and again evaluate its performance. To train the autoencoder, I am adding noise to both the training images and the test images. I then pass into my autoencoder network the noisy training images as the training data for the autoencoder and the original training images as the target for the network in the training phase. My validation data for the autoencoder is the noisy test images and the target is the original test images. The metric I use for validation is the mean squared error (MSE) between the training images and the test images. MSE measures the distance between the pixels of the original image and the image reconstructed by the autoencoder.

Implementation

Dataset: I used the CIFAR-10 dataset to train both the autoencoder and CNN. The dataset contains 60,000 32x32 color images, where each image belongs to one of 10 classes. The ten classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. I split this dataset into 50,000 images for training and 10,000 images for testing. The dataset is available here [1], however, I used a TensorFlow library call to download the data. I normalized the images by dividing the pixels by 255, so the pixels carry a value between 0 and 1. Normalization is to ensure better performance for my CNN.

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

Figure 2: TensorFlow library call to load CIFAR-10 dataset

Adding Noise: I experimented with the effects of adding four different types of noise to the test images. The four different types of noise are uniform noise, concentrated noise, removing pixels, and random noise. I used linear signal-to-noise ratio (SNR) as a metric to ensure that every image in the dataset is subject to the same proportional amount of noise to its signal strength. The SNR is kept constant at 10 for each image. For uniform noise, the same amount of noise is added or subtracted from each pixel. For concentrated noise, I first calculate the total mean squared noise using the fixed SNR and the image. Then, I add as much noise as I can to each pixel starting from the middle row of the image until the SNR reaches 10. For removing pixels, I reset all pixels to zero starting from the middle row of the image until the SNR reaches 10. Finally, for random noise, I randomly pick pixels and reset them to zero until the SNR reaches 10.

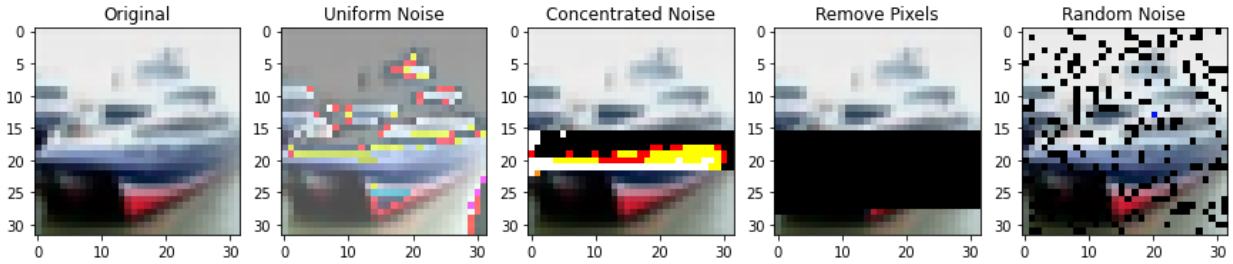

Figure 3: Different Types of Noise

Autoencoder Architecture: My convolutional autoencoder is based on an example from the TensorFlow documentation for autoencoders [3]. I made a few changes to the architecture to get a lower loss for my dataset, including an upsampling layer and a few batch normalization layers. The first layer of the encoder is an upsampling layer that takes in a 32x32 image and doubles the size of its dimensions to 64x64. Then, I have three convolution blocks, each block being a 2D convolutional layer followed by a batch normalization layer. The convolutional layers use the ReLU activation function, ‘same’ padding, and a stride of 2. The decoder consists of two 2D transposed convolutional layers followed by a 2D convolutional layer. The transposed convolutional layers use the ReLU activation function, ‘same’ padding, and a stride of 2, whereas the convolutional layer uses the ReLU activation function, ‘same’ padding, and a stride of 1.

Autoencoder Training Parameters: I used the Adam optimizer and MSE loss function. The Adam optimizer is an extended version of stochastic gradient descent, which minimizes the objective function. The batch size is 32 and the initial learning rate is 1e-3. The learning rate is scheduled to decay exponentially with a decay rate of 0.8 every 4000 steps. The model is run for 10 epochs.

CNN Architecture: My CNN architecture is based on the LeNet model [2]. I tried several different CNN architectures including VGG-16, AlexNet, and many versions of LeNet to

determine the best architecture based on which one network gave me the highest accuracy. The architecture I selected can be found here [2]. I used this architecture as a starting point and made several modifications. The modifications I made to the architecture are adding a couple of image preprocessing layers and I also added a couple of additional layers as well. The first preprocessing layer is an upsampling layer which increases the dimension of the input images. I am upsampling by a factor of 2, which means it'll take a 32x32 image and convert it into a 64x64. The next preprocessing layer is a RandomZoom layer, which zooms out the images by a randomly picked factor within the range [0.4, 0.6]. I then have 5 blocks of convolutional layers, each structured in the following way: 2 convolutional layers, followed by a max-pooling layer, and finally a batch normalization layer. The convolutional layers all have a stride of 1, a padding of "same", a ReLU activation function, and a kernel size of 3x3. The first block uses 32 filters for its first convolutional layer and 64 for its second. The second block uses 64 filters for both of its convolutional layers, and I double the filter bank size for every successive block. The maxpooling layer has a pool size of 2x2 and "same" padding, so after each maxpooling layer, the image dimensions are reduced by a factor of 2. The batch normalization layer normalizes the output from the pooling layer based on the mean and the standard deviation of the current batch of inputs. After the convolutional layers, I have three fully connected layers. The first fully connected layer has a total of 1024 nodes and the second one has 512 nodes, both with a ReLU activation function. The third fully connected layer is also my output layer, which has 10 nodes, each of which represents one of the classes from my data. The softmax activation function is used for the output layer to produce the probabilities with which the model classifies the images.

CNN Training Parameters: I added my optimizations to the training parameters. I used the Adam optimizer, the sparse categorical accuracy metric, and the sparse categorical cross-entropy loss function. Sparse categorical accuracy is a metric that determines how often a prediction matches its label. The sparse categorical cross-entropy loss function measures the cross-entropy loss between predictions and labels. The batch size is 25 and the initial learning rate is 1e-3. The learning rate is scheduled to decay exponentially with a decay rate of 0.7 every 4000 steps or 2 epochs. I also implemented early stopping to avoid overfitting the training data. The training phase will end if the validation loss increases in the two most recent epochs. The number of epochs is capped at 20.

```
# Exponentially decaying learning rate
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=1e-3, decay_steps=4000, decay_rate=0.7)
```

Figure 4: Exponentially decaying learning rate

```
# Stops early if validation loss increased in the 2 most recent epochs
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2, restore_best_weights=True)
```

Figure 5: Early stopping criteria for the training phase for CNN

CNN Performance Evaluation

For every evaluation of my CNN model, the input remains the same, which is the training images and the training labels that were first established. The only moving part for the input is the validation images. I first evaluate my CNN using the original test images and test labels. The

results I get from this evaluation will be used as my baseline comparison. Next, for each type of noise, I will do two different runs with two different inputs for the validation images into the CNN model. In the first run, I will directly input the noisy validation images into the CNN model. In the second run, I will pass the noisy validation images through the autoencoder and input the denoised test images from the autoencoder into the CNN model. I will then compare the CNN's performance with the noisy and denoised validation images to its performance with the original, clean validation images.

Results

Summary of Results

I found that my CNN performs significantly worse when concentrated noise, remove pixels, and random noise are added to the test images than with the noise-free images. However, when uniform noise is applied to the test images, while my CNN performs worse than when no noise is added, it still performs better than when the images are denoised by the autoencoder. For the three types of noise that my CNN performs significantly worse for, when I run the test images through the autoencoder first before inputting them into the CNN, the performance is significantly higher than when the noisy images are inputted directly. The performance is still not as good as when the images with no noise are inputted into the CNN, but it is still a major improvement compared to when noise was added. From this, I have concluded that uniform noise affects the CNN's performance the least and it affects the autoencoder's performance the most. I also concluded that random and concentrated noise affects the CNN's performance the most. The autoencoder gave the best performance boost to the CNN for test images with concentrated noise. Another observation that I made is that when concentrated noise, remove pixels, and random noise are added, there are one or two classes that are predicted more frequently than others. I also noticed that the validation loss changed dramatically between epochs when tested with noisy images as shown in Figure 6. A batch size of 25 was chosen because it gave the best results when tested with noise-free images. However since a smaller batch size means more weight updates per epoch, we see these dramatic rises and falls in the validation loss when the CNN is tested with noisy images. This can be avoided by optimizing the batch size for all types of noise types being tested. Also, I realized that better denoising results can be obtained by training ten different instances of the autoencoder, each with images from only one of the classes. For a more detailed evaluation of my results, please refer to the following figures.

```
val_loss: 1.6665 - val_sparse_categorical_accuracy: 0.4399

val_loss: 3.6339 - val_sparse_categorical_accuracy: 0.2085

val_loss: 2.0544 - val_sparse_categorical_accuracy: 0.3786
```

Figure 6: An example of validation loss changing dramatically between epochs

Summary

One of my goals in this project was to experiment with how different types of noise impact the performance of a CNN. My second goal was to determine the performance of a CNN if an autoencoder was implemented to denoise the images. The autoencoder and CNN were developed and trained using the CIFAR-10 dataset. I developed four different methods to add noise to images, which are uniform noise, concentrated noise, remove pixels, and random noise. For each type of noise, I tested the CNN's performance by directly inputting the noisy images into the CNN as well as by inputting the noisy images into the autoencoder and then using the output of the autoencoder as input into the CNN. I compared the performance for each scenario to a baseline, which was images with no noise as input to my CNN. My findings show that not all types of noise affect a CNN's performance the same even when the linear SNR is kept constant. Some improvements that I propose are to optimize the batch size for evaluations with noisy images and to separately train the autoencoder for each class of images.

References

[1] CIFAR-10 and CIFAR-100 datasets (toronto.edu)

[2] Convolutional Neural Network - PyTorch implementation on CIFAR-10 Dataset (analyticsvidhya.com)

[3] Intro to Autoencoders | TensorFlow Core