Image Denoising with the UNet and Autoencoders

Alan Casey and Alex Racapé

November 2023

1 Introduction

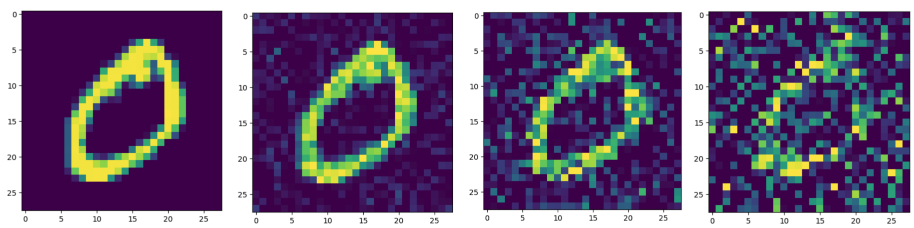
In this study, we tested the U-Net and Autoencoder architecutres to solve the problem of image denoising. This goal of this problem is to remove noise from an image to yield a cleaner result. U-Nets and Autoencoders make for an interesting comparison since they share some similarities in their architecture. U-Nets are commonly used for image segmentation and it centeres a round a sort of U shape. Convolutional and max-pooling layers reduce the dimensions of each channel while building up a greater depth of channels. Then at the bottom of the "U", tanspose convolutional layers reduce the amount of channels while increasing the dimensions of each channel. The idea is to reduce an image to some core features then construct a new image from these features. Autoencoders are similar. Convolutional layers can be used for dimensionality reduction until a vector contains some core features in a lower dimensional latent space. A decoder can then reconstruct an image from the features of this latent vector. These are essentially the same process except the U-Net adds some skip connections to the networks architecture. This study contrasts the results from these two networks, and isolates the impact of these added skip connections. We constructed our own simplified networks and used the MNIST dataset to evaluate the two models ability to denoise images both qualitatively and quantitatively.

2 Methodology

Our studies aimed to investigate the relative performance of the U-Net and Autoencoder models on the MNIST dataset. For our denoising task, we created a custom dataset object that could be used to dynamically set the amount of noise used while training and testing a network (Line 37). This dataset has an attribute *mu* which corresponds to the amount of noise in the image. We used the following equation to add noise to our image where I is the original image and N is tensor of random noise.

$$Y = (1 - mu) * I + mu * N$$

Mu can then be interpreted as the percentage of the image that is random noise. For our experiment we defined three levels of noise. Mild noise corresponds with a mu of .1, moderate noise corresponds with a mu or .25, and severe noise corresponds with a mu or .4. We set these thresholds qualitatively based on some test plots. Figure 1 shows what these levels of noise look like with our data.



(a) No Noise (b) Mild (c) Moderate (d) Severe

Figure 1: MNIST Data with Varying levels of Noise

Our two models (UNet and Autoencoder) were simplified versions of the U-Net architecture with and without the skip connections. They can be found on lines 69 and 145, and the model summaries show that they are essentially the same. In our implementation, the Autoencoder has slightly more weights to learn in the transpose convolutional layers since we wanted the number of channels to match up with U-Net. U-Net concatenates previous channels, so the Autoencoder needs to learn more filters in order to match this depth. However, we made sure the total number of parameters for both were as close as possible for our specified architecture: 88,641 for U-Net, and 104,521 for Autoencoder.

Since the input images from MNIST are only 28x28, we did not want to fully replicate the original U-Net. Since we were simplifying the problem by using smaller images with less channels, a reduction in channel dimension as well as a reduction in total layers makes sense. Where the classic U-Net has four main tiers with skip connections, ours has two skip connection layers. The image gets reduced from a size of 28x28 to 14x14 to 7x7, and the number of channels gets increased from 1 to 20 to 40 to 80. Then the size scales back up while the channel size scales back down resulting in a single new output image.

To test our models, we trained three versions of each network that were trained on varying levels of noise. The noisy image was the input to the network, and the model trained to output a clean image that matched the original. We trained our models for ten epochs. Then we ran several tests on the trained models. First, we plotted 64 images with the model's predictions. Then we calculated the average MSE across 1000 test images. While training and testing, we also used timers to track the training and inference times. All training and testing was done using a T4 GPU.

((a)) Input ((b)) U-Net Predictions ((c)) Autoencoder Predictions

Figure 2: U-Net and Autoencoder Results with Mild Noise

((a)) Input ((b)) U-Net Predictions ((c)) Autoencoder Predictions

Figure 3: U-Net and Autoencoder Results with Moderate Noise

3 Results and Discussion

3.1 Qualitative Observations

For our qualitative analysis, we used 64 data points and plotted results from our models using varying levels of noise. In general, both models performed really well with results that closely resembled the clean original images. While the results were very similar, the results from the Autoencoder seemed slightly more blurry, especially for the lower noise levels. Both models struggled more with the severe level of noise, yet the results look very similar. The only noticable difference is the six in the bottom right hand corner, where Autoencoder's result looks more like a pretzel. With this difference U-Net may have done slightly better, but it is hard to say conclusively through a purely qualitative analysis.

((a)) Input ((b)) U-Net Predictions ((c)) Autoencoder Predictions

Figure 4: U-Net and Autoencoder Results with Severe Noise

Table 1: Model Performance Comparison

Model	Noise	Average MSE	Training Time	Inference Time
U-Net	Mild	0.00095	1m 4s	1.3ms
	Moderate	0.0047	1m 8s	1.3ms
	Severe	0.014	1m 4s	1.4ms
Autoencoder	Mild	0.0020	1m 7s	2.6ms
	Moderate	0.0055	1m 11s	1.4ms
	Severe	0.015	1m 6s	2.0ms

3.2 Quantitative Observations

In order to get more accurate measures that could help us in our analysis, we decided to measure differences from a quantitative perspective. We decided to measure both our UNet and Autoencoder networks on their resulting MSEs (Mean Squared Error Loss) after ten epochs, their total training time, and their inference time.

The Mean Squared Error Loss Function creates a criterion that measures the mean squared error between each pixel from the original image to the output image from the neural network. This measures how far off the de-noised image was from the original image.

As shown in Table 1, our UNet model's average MSE outperformed the Autoencoder's average MSE in all three noise categories, with the most significant out-performance being in the Mild-noise category. One of the reasons for this could be that, even though the Autoencoder has more weights it can learn, the UNet employs skip-connections. This helps preserve information, and these skip connections allow for a direct flow of information from the early layers (which extract low-level features) to the later layers (which contain higher-level features) of the UNet network. These added channels contribute some information that is lost for the Autoencoder.

The total training time was also faster for UNet—this is most likely the result of having fewer parameters. Skip connections often result in fewer parameters compared to fully connected or densely connected networks. This is because skip connections allow the re-utilization of features from lower layers instead of creating entirely new sets of parameters. This is another potential reason as to why the UNet outperformed the Autoencoder in its average MSE: fewer parameters can help prevent the overfitting of data while reducing the computational cost (which also reduces the total inference time for each image).

4 Conclusion

Our study aimed to compare the performance of both the UNet and Autoencoder architectures for the task of image denoising. We did this using the MNIST dataset of handwritten numbers. Both models showed promising results in reducing noise and restoring images to their cleaner versions. However, there were notable differences between the two models, with our UNet architecture demonstrating a clear advantage.

Qualitatively, when analyzing the output images generated by both models, we observed that both U-Net and Autoencoder produced impressive denoised images which closely resembled the original images. However, the Autoencoder's results were slightly more blurry, especially for images with lower levels of noise. However, it was still challenging to make a definitive conclusion based solely on a qualitative analysis.

To gain a more quantitative perspective, we measured the Mean Squared Error (MSE) loss for both models after training on images with varying levels of noise. The results revealed that UNet consistently outperformed the Autoencoder in terms of average MSE, with the most significant difference observed in the mild noise category. This superior performance could be attributed to U-Net's use of skip connections, which facilitate the preservation of fine-grained details and transfer information between different network layers. This allows U-Net to handle the denoising task more effectively.

Furthermore, our UNet exhibited shorter training times and faster inference times compared to the Autoencoder, likely due to its fewer parameters. Skip connections in our UNet can lead to more efficient training and reduced computational cost, which can be advantageous in practical applications.

In summary, our study highlights the effectiveness of both the U-Net and Autoencoder architectures for image denoising. However, U-Net, with its skip connections and fewer parameters, outperformed the Autoencoder in terms of denoising accuracy and computational efficiency. These findings provide valuable insights for researchers and practitioners working on image denoising tasks and related applications.

5 Appendix

```
# -*- coding: utf-8 -*-
"""lab6

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1V8xuNuqp8dkIQoRJZR2Hc_2fWvWMm2rd
"""

# Project 6: U-Net and Autoencoders

# Imports
import torch
from torchvision.datasets import MNIST
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torch.nn as nn
from torchsummary import summary
from torch.optim import Adam
import numpy as np
import matplotlib.pyplot as plt
import time

device = "cuda" if torch.cuda.is_available() else "cpu"
print(device)

# Global noise values
MILD = .1
MEDIUM = .25
SEVERE = .4

# Load MNIST data
mnist_train = MNIST('~/data', train=True, download=True)
mnist_test = MNIST('~/data', train=False, download=True)


class MNISTDataset(Dataset):
    """Noisy wrapper for MNIST Dataset

    We can dynamically set the noise using the mu attribute
    X's are noisy images, and Y's are the clean original images
    Images are (1, 28, 28)
    """

    def __init__(self, x, y, mu):

        # Reformat input image
```

```python
        x = x.view(-1, 1, 28, 28)
        normal_image = x.float()/255

        # Add the noise to the image, but keep vals in [0, 1]
        noise = torch.randn_like(normal_image, dtype=torch.float)
        noisy_image = (1 - mu) * normal_image + (mu) * noise
        noisy_image = torch.clamp(noisy_image, 0, 1)

        # Save as input and ground-truth
        self.x, self.y = noisy_image, normal_image

    def __getitem__(self, ix):
        return self.x[ix].to(device), self.y[ix].to(device)

    def get_group(self, size):
        """Get subset with `size` images"""
        return self.x[0:size].to(device), self.y[0:size].to(device)

    def __len__(self):
        return len(self.x)

class UNet(nn.Module):

    """
        Our UNet is a neural network that reduces an images to core features
        (while reducing the size of the image and increasing the channels),
        and then increases it back to its original size.

        Since the input images from MNIST are size 28x28, we did not want to
        fully replicate the original UNet. Because we are simplifying the problem,
        a reduction in channel dimension makes sense here, as well as a reduction
        in total layers.

        We decided our architecture would look like this:
          - 1 CL (Convolutional Layer) followed by a Max Pool Layer 2 times, then
          - A bottom CL followed by a TCL (Transpose CL) 2 times, followed by
          - Two CL (the last one reducing the dimension size back to 1)

        The image gets reduced from a size of 28x28 -> 14x14 -> 7x7, the channel
        size gets increased from 1 -> 20 -> 40 -> 80, and then the size scales
        back up and the channel size scales back down.

        One key aspect of the UNet is that before the MaxPool layers, a copy of
        the outputs get saved so it can be concatenated later in the re-scaling side.
    """

    def __init__(self, channels=1, output_classes=10):
        super().__init__()
        self.loss_func = nn.MSELoss()

```

```python
        # Convolution Layers
        self.ConvLayer1 = self.ConvLayer(1, 20, 3)
        self.MaxPool = nn.MaxPool2d(2)
        self.ConvLayer2 = self.ConvLayer(20, 40, 3)

        # Bottom Layer
        self.ConvLayer3 = self.ConvLayer(40, 80, 3)

        # Transpose Convolution Layers
        self.TransposeLayer1 = self.TransposeConvLayer(80, 40, 2)
        self.ConvLayer4 = self.ConvLayer(80, 40, 3) # Concatenate ConvLayer2 result
        self.TransposeLayer2 = self.TransposeConvLayer(40, 20, 2)
        self.ConvLayer5 = self.ConvLayer(40, 20, 3) # Concatenate ConvLayer1 result

        # Flatten to final image
        self.ConvOneByOne = nn.Conv2d(20, 1, kernel_size=1)

    def ConvLayer(self, in_ch, out_ch, kernel_size=3):
        sequence = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size, 1, 1),
            nn.ReLU())
        return sequence

    def TransposeConvLayer(self, in_ch, out_ch, kernel_size=2):
        sequence = nn.Sequential(
            nn.ConvTranspose2d(in_ch, out_ch, kernel_size, 2),
            nn.ReLU())
        return sequence

    def forward(self, x):
        a = self.ConvLayer1(x)
        x = a
        b = self.ConvLayer2(self.MaxPool(x))
        x = b
        x = self.TransposeLayer1(self.ConvLayer3(self.MaxPool(x)))
        x = self.TransposeLayer2(self.ConvLayer4(torch.cat([b,x], dim=1)))
        return self.ConvOneByOne(self.ConvLayer5(torch.cat([a,x], dim=1)))

    def num_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad)


# Create a UNet instance and visualize parameter grid
unet = UNet().to(device)
print(f"Total parameters for UNet: {unet.num_parameters()}")
summary(unet, (1, 28, 28))

class Autoencoder(nn.Module):

    """
```

```
        Our Autoencoder network is essentially the same thing as our UNet.
        However, we do not employ skip connections (we don't save the output
        of the Convolutional Layers in order to do concatenations later).

        The autoencoder employs the same architecture and size reduction/
        scaling sizes as our previous UNet architecture.
    """

    def __init__(self, channels=1, output_classes=10):
        super().__init__()
        self.loss_func = nn.MSELoss()

        self.layers = nn.Sequential(self.ConvLayer(1, 20, 3),
                                    nn.MaxPool2d(2),
                                    self.ConvLayer(20, 40, 3),
                                    nn.MaxPool2d(2),
                                    self.ConvLayer(40, 80, 3),
                                    self.TransposeConvLayer(80, 40, 2),
                                    self.TransposeConvLayer(40, 20, 2),
                                    nn.Conv2d(20, 1, kernel_size=1))

    def ConvLayer(self, in_ch, out_ch, kernel_size=3):
        return nn.Sequential(nn.Conv2d(in_ch, out_ch, kernel_size, 1, 1),
                             nn.ReLU())

    def TransposeConvLayer(self, in_ch, out_ch, kernel_size=2):
        return nn.Sequential(nn.ConvTranspose2d(in_ch, in_ch, kernel_size, 2),
                             nn.ReLU(),
                             nn.Conv2d(in_ch, out_ch, 3, 1, 1),
                             nn.ReLU())

    def forward(self, x):
        return self.layers(x)

    def num_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad)


# Create an autoencoder instance and visualize parameter grid
autoencoder = Autoencoder().to(device)
print(f"Total parameters for Autoencoder: {autoencoder.num_parameters()}")
summary(autoencoder, (1, 28, 28))

def train_batch(model, X_train, Y_train, opt):
    opt.zero_grad()                                 # Flush memory
    pred = model(X_train)                           # Get predictions
    batch_loss = model.loss_func(pred, Y_train)   # Compute loss
    batch_loss.backward()                           # Compute gradients
    opt.step()                                      # Make a GD step
    return batch_loss.detach().cpu().numpy()
```

```python


def train(model, train_dl, epochs, optimizer):

    loss_history = []
    start = time.time()
    for epoch in range(epochs):

        print(f"Running Epoch {epoch + 1} of {epochs}")
        epoch_losses = []
        for i, batch in enumerate(train_dl):
            x, y = batch
            x, y = x.to(device), y.to(device)
            batch_loss = train_batch(model, x, y, optimizer)
            epoch_losses.append(batch_loss)

        epoch_loss = np.mean(epoch_losses)
        loss_history.append(epoch_loss)

    end = time.time()
    training_time = end - start
    return loss_history, training_time


# Train the models
unet_models = []
unet_losses = []
unet_times = []
autoencoder_models = []
autoencoder_losses = []
autoencoder_times = []
noise_levels = [MILD, MEDIUM, SEVERE]
for noise in noise_levels:

    # Create dataset and loader for training and testing
    train_dataset = MNISTDataset(mnist_train.data, mnist_train.targets, noise)
    train_dl = DataLoader(train_dataset,batch_size=128, shuffle=True)

    # Plot a sample of the noisiness
    normal, noisy = train_dataset[1]
    plt.imshow(normal.cpu()[0, :, :])
    plt.show()
    plt.imshow(noisy.cpu()[0, :, :])
    plt.show()

    # Create Models
    unet = UNet().to(device)
    autoencoder = Autoencoder().to(device)

    # Train competing models
```

```
    num_epochs = 5
    lr = .001
    u_opt = Adam(unet.parameters(), lr=lr)
    a_opt = Adam(autoencoder.parameters(), lr=lr)
    u_loss_history, u_train_time = train(unet, train_dl, num_epochs, u_opt)
    a_loss_history, a_train_time = train(autoencoder, train_dl, num_epochs, a_opt)

    # Save Models
    unet_models.append(unet)
    unet_losses.append(u_loss_history)
    unet_times.append(u_train_time)
    autoencoder_models.append(autoencoder)
    autoencoder_losses.append(a_loss_history)
    autoencoder_times.append(a_train_time)

# Test the models
!pip install -q torch_snippets

from torchvision.utils import make_grid
from torch_snippets import show


def plot_loss(loss_history):
    plt.plot(loss_history)
    plt.title("Loss Over Time")
    plt.xlabel("Epochs")
    plt.ylabel("Loss Value")
    plt.show()


def plot_comparison(model, x, y):

    # Plot noisy input
    true_images = x
    true_grid = make_grid(true_images, nrow=8, normalize=True)
    show(true_grid.cpu().detach().permute(1,2,0), sz=5)

    # Plot a grid of our predictions
    predicted_images = model(x).data.cpu().view(64, 1, 28, 28)
    prediction_grid = make_grid(predicted_images, nrow=8, normalize=True)
    show(prediction_grid.cpu().detach().permute(1,2,0), sz=5)

    # Plot a grid of real images
    true_images = y
    true_grid = make_grid(true_images, nrow=8, normalize=True)
    show(true_grid.cpu().detach().permute(1,2,0), sz=5)


def calculate_mse(model, x, y):
```

```python
    # Calculate MSE per pixel across all images pairwise
    loss = nn.MSELoss()
    start = time.time()
    pred = model(x)
    end = time.time()
    output = loss(pred, y)

    return output, end - start


def run_tests(model, dataset):

    # Calculate quantitative stats
    x, y =dataset.get_group(len(dataset))
    mse, test_time = calculate_mse(model, x, y)
    print(f"MSE: {mse}")
    print(f"Inference time: {test_time}")

    # Plot for qualitative differences
    x, y = dataset.get_group(64)
    plot_comparison(model, x, y)


# Run through our tests on the saved models
for i, noise in enumerate(noise_levels):

    test_dataset = MNISTDataset(mnist_test.data, mnist_test.targets, noise)

    print(f"-- Starting U-Net tests - Noise: {noise}")
    print(f"Training time: {unet_times[i]}")
    run_tests(unet_models[i], test_dataset)
    print(f"-- Starting Autoencoder tests - Noise: {noise}")
    print(f"Trainint time: {autoencoder_times[i]}")
    run_tests(autoencoder_models[i], test_dataset)
```