Introduction to Machine Learning:
Using TensorFlow to Reduce Noise within Images through Autoencoders

Team: Ryan Ho (rh564), Andrew Chen (ac2463), Kevin Ram (kmr232), Junkai Zheng (jz765), Matthew Liang (ml2483), Katherine Chang-wu (kc842)

Goal:

The focus of this project is to develop a noise reduction algorithm based on autoencoders and machine learning that can be used to improve the quality of images. This project is motivated by the fact that noise is a common problem in image processing, and that current methods for noise reduction are often ineffective. The goal is to develop a noise reduction algorithm that is both effective and efficient.

To accomplish this goal, we will first train an autoencoder to learn a representation of the data that is robust to noise. We will then use this autoencoder to train a denoising autoencoder, which will learn to map noisy inputs to the corresponding clean outputs. Finally, we will use the denoising autoencoder to denoise images.

This project is based on the following paper:

Autoencoding Variational Bayes by Kingma and Welling (https://arxiv.org/abs/1312.6114)

The paper describes a method for training an autoencoder by minimizing the variational lower bound on the data. This method is closely related to the method we will use to train our denoising autoencoder. In particular, we will use the same objective function, which is based on the principle of maximum likelihood.

We will train our autoencoders using the TensorFlow library. TensorFlow is a powerful tool for doing machine learning, and it has excellent support for autoencoders. In particular, TensorFlow includes a library for training variational autoencoders. We will use this library to train our autoencoders.

The goal of this project is to develop a noise reduction algorithm that is both effective and efficient. To accomplish this goal, we will use autoencoders and machine learning.

Agenda (initial):

1. Teaching and research autoencoders and how they can be used to manipulate image data.
2. Create machine learning models that utilize TensorFlow
3. Setup and optimize workflow runtimes for Google Collaboratory/Jupyter Notebook
4. Determine what training samples should be used to train ML model

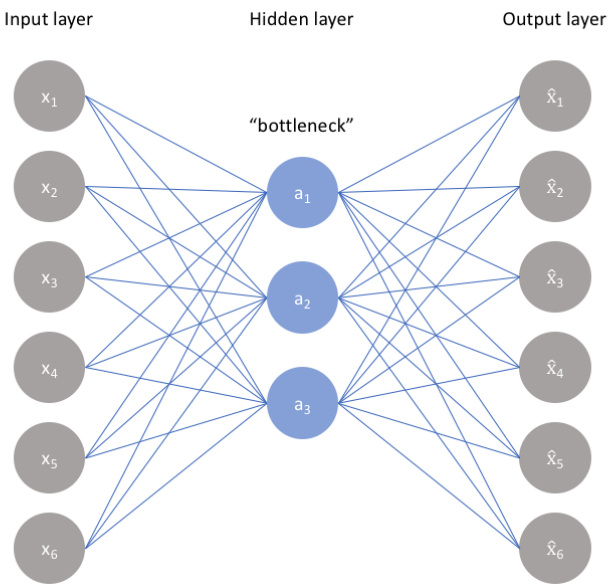
Agenda (Weekly):

1. Review and present updates of what each member has been doing
2. Discuss any new information related to the project
3. Continue development of ML model and training as well as front-end React.js implementation
4. Discuss further about future projects and prospects for ML model and actual project uses

Research:

What are Autoencoders:

Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we'll design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input. If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data (ie. correlations between input features), this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck.



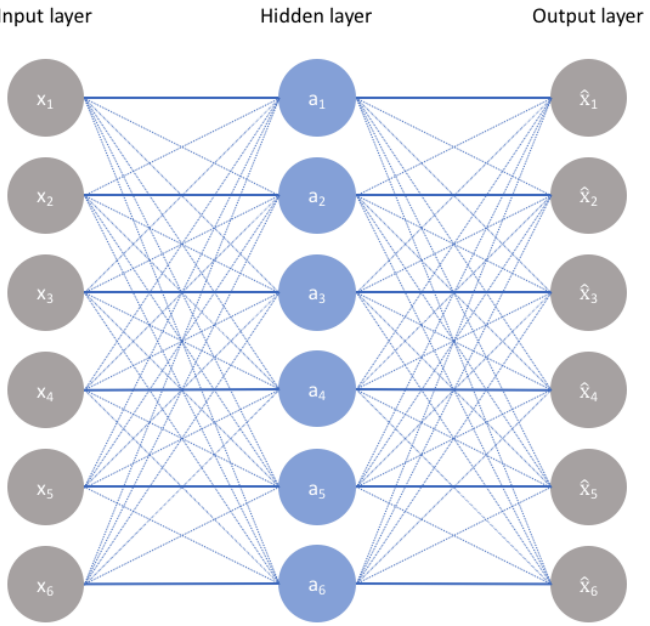
As visualized above, we can take an unlabeled dataset and frame it as a supervised learning problem tasked with outputting x, a reconstruction of the original input x. This network can be trained by minimizing the reconstruction error, $(x,\hat{x})$, which measures the differences between our original input and the consequent reconstruction. The bottleneck is a key attribute of our network design; without the presence of an information bottleneck, our network could easily learn to simply memorize the input values by passing these values along through the network (visualized below).

A bottleneck constrains the amount of information that can traverse the full network, forcing a learned compression of the input data.

The ideal autoencoder model balances the following:

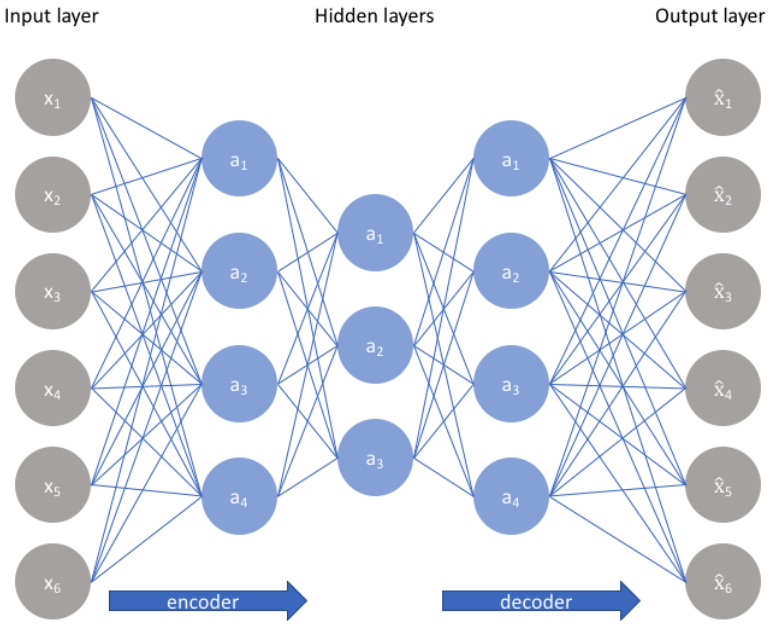
- Sensitive to the inputs enough to accurately build a reconstruction.
- Insensitive enough to the inputs that the model doesn't simply memorize or overfit the training data.
- This trade-off forces the model to maintain only the variations in the data required to reconstruct the input without holding on to redundancies within the input. For most cases, this involves constructing a loss function where one term encourages our model to be sensitive to the inputs (ie. reconstruction loss $(x,\hat{x})$) and a second term discourages memorization/overfitting (ie. an added regularizer).

"$(x,\hat{x})$ + regularizer"

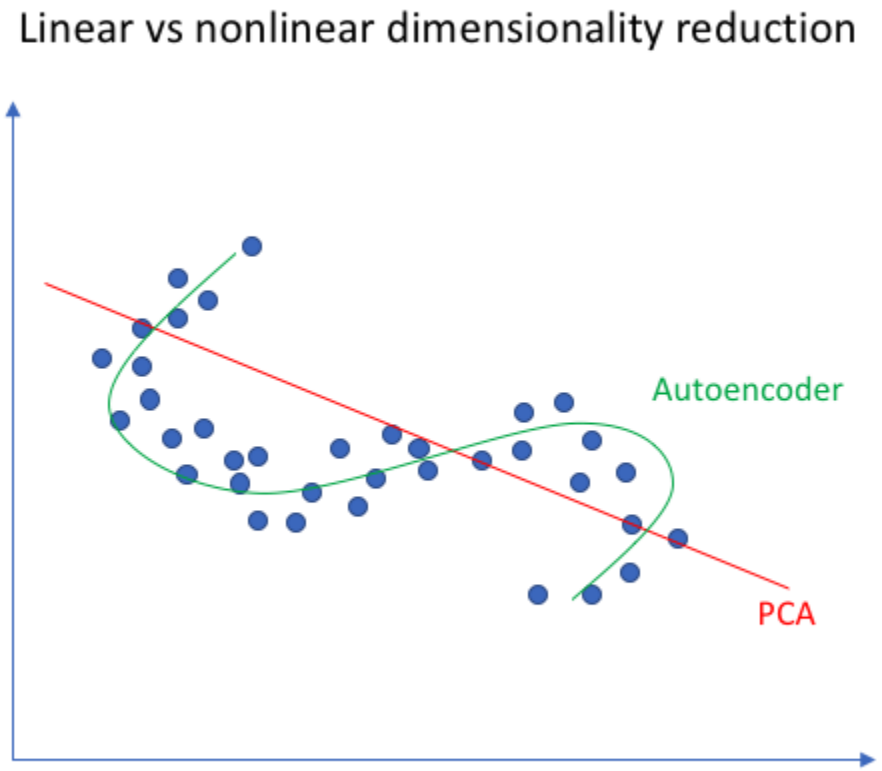


Undercomplete autoencoder

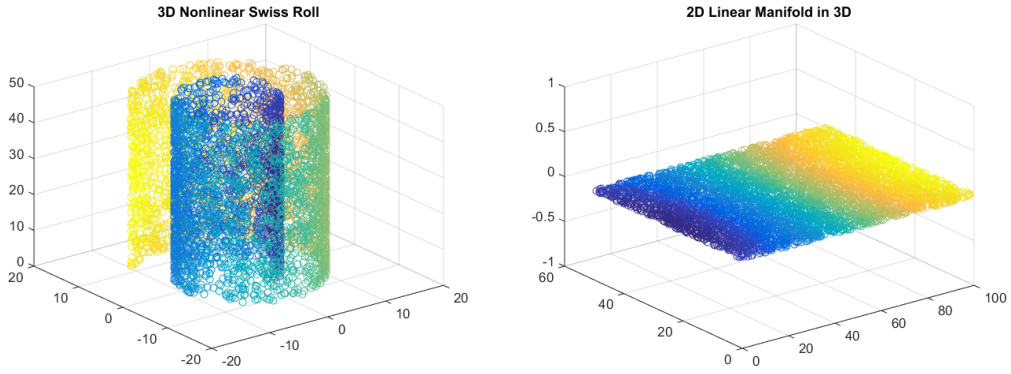
The simplest architecture for constructing an autoencoder is to constrain the number of nodes present in the hidden layer(s) of the network, limiting the amount of information that can flow through the network. By penalizing the network according to the reconstruction error, our model can learn the most important attributes of the input data and how to best reconstruct the original input from an "encoded" state. Ideally, this encoding will learn and describe latent attributes of the input data.



Because neural networks are capable of learning nonlinear relationships, this can be thought of as a more powerful (nonlinear) generalization of PCA. Whereas PCA attempts to discover a lower dimensional hyperplane which describes the original data, autoencoders are capable of learning nonlinear manifolds (a manifold is defined in simple terms as a continuous, non-intersecting surface). The difference between these two approaches is visualized below.



For higher dimensional data, autoencoders are capable of learning a complex representation of the data (manifold) which can be used to describe observations in a lower dimensionality and correspondingly decoded into the original input space.



An undercomplete autoencoder has no explicit regularization term - we simply train our model according to the reconstruction loss. Thus, our only way to ensure that the model isn't memorizing the input data is to ensure that we've sufficiently restricted the number of nodes in the hidden layer(s).

For deep autoencoders, we must also be aware of the capacity of our encoder and decoder models. Even if the "bottleneck layer" is only one hidden node, it's still possible for our model to memorize the training data provided that the encoder and decoder models have sufficient capability to learn some arbitrary function which can map the data to an index.

Given the fact that we'd like our model to discover latent attributes within our data, it's important to ensure that the autoencoder model is not simply learning an efficient way to memorize the training data. Similar to supervised learning problems, we can employ various forms of regularization to the network in order to encourage good generalization properties; these techniques are discussed below.

Sparse autoencoders

Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Rather, we'll construct our loss function such that we penalize activations within a layer. For any given observation, we'll encourage our network to learn encoding and decoding which only relies on activating a small number of neurons. It's worth noting that this is a different approach towards regularization, as we normally regularize the weights of a network, not the activations.

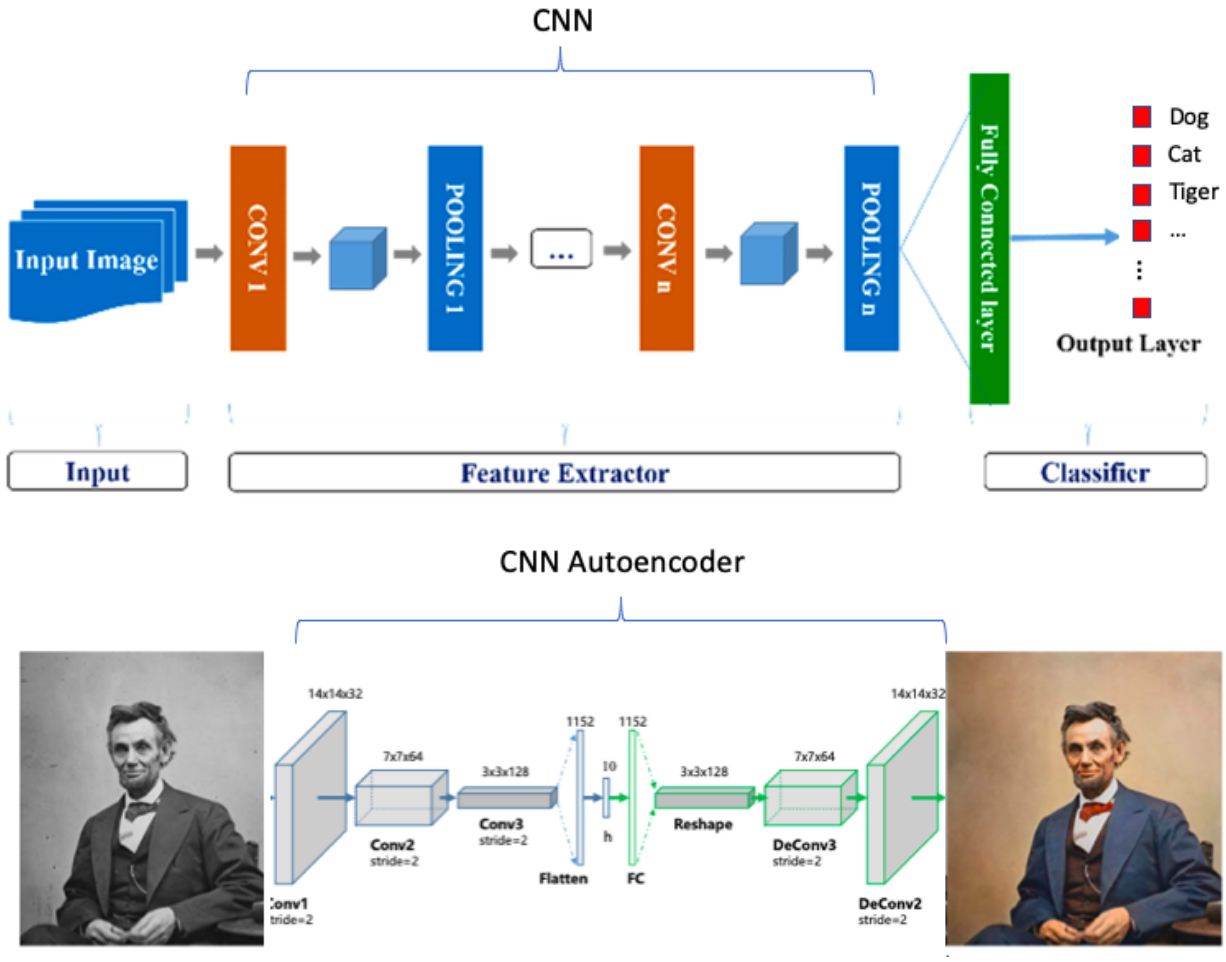
A generic sparse autoencoder is visualized below where the opacity of a node corresponds with the level of activation. It's important to note that the individual nodes of a trained model which activate are data-dependent, different inputs will result in activations of different nodes through the network.

Noise Reduction with TensorFlow:

Modeling image data requires a special approach in the neural network world. The best-known neural network for modeling image data is the Convolutional Neural Network (CNN, or ConvNet). It can better retain the connected information between the pixels of an image. The particular design of the layers in a CNN makes it a better choice to process image data.

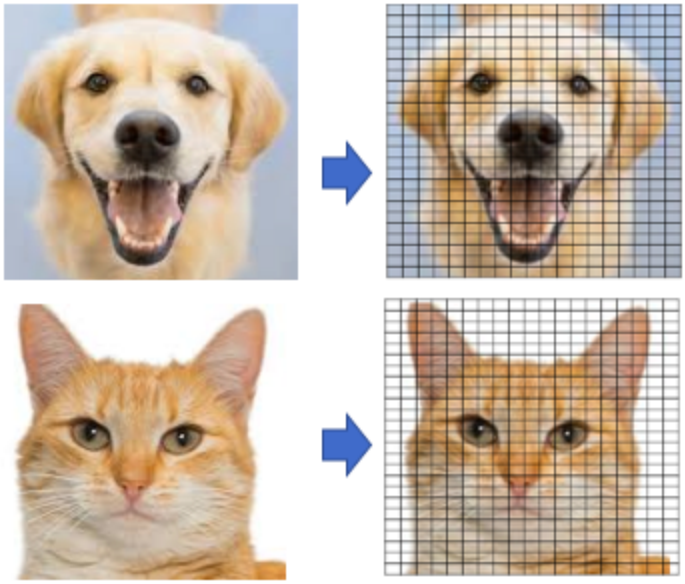
The CNN design can be used for image recognition/classification as shown in Figure (1), or be used for image noise reduction or coloring as shown in Figure (2). In Figure (1), we train the CNN model by taking many image samples as the inputs and labels as the outputs. We then use this trained CNN model to create a new image to recognize if it is a “dog”, or “cat”, etc. CNN also can be used as an autoencoder for image noise reduction or coloring.

When CNN is used for image noise reduction or coloring, it is applied in an Autoencoder framework, i.e, the CNN is used in the encoding and decoding parts of an autoencoder. Figure (2) shows a CNN autoencoder. Each of the input image samples is an image with noises, and each of the output image samples is the corresponding image without noises. We can apply the trained model to a noisy image and then output a clear image. Likewise, it can be used to train a model for image coloring. Figure (2) is an example that uses CNN Autoencoder for image coloring.



Understand Image Data

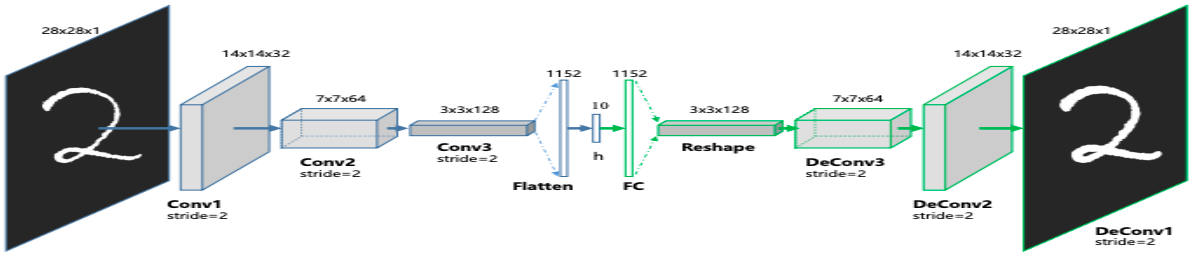
An image is made of “pixels” as shown in Figure (A). In a black-and-white image, each pixel is represented by a number ranging from 0 to 255. Most images today use 24-bit color or higher. An RGB color image means the color in a pixel is a combination of Red, Green, and Blue, each of the colors ranging from 0 to 255. The RGB color system constructs all the colors from the combination of Red, Green, and Blue colors as shown in this RGB color generator. So a pixel contains a set of three values RGB(102, 255, 102) refers to color #66ff66.



An image 800 pixel wide, 600 pixels high has 800 x 600 = 480,000 pixels = 0.48 megapixels (“megapixel” is 1 million pixels). An image with a resolution of 1024×768 is a grid with 1,024 columns and 768 rows, which therefore contains 1,024 × 768 = 0.78 megapixels.

Why Are the Convolutional Autoencoders Suitable for Image Data?

We see a huge loss of information when slicing and stacking the data. Instead of stacking the data, the Convolution Autoencoders keep the spatial information of the input image data as they are, and extract information gently in what is called the Convolution layer. Figure (D) demonstrates that a flat 2D image is extracted to a thick square (Conv1), then continues to become a long cubic (Conv2) and another longer cubic (Conv3). This process is designed to retain the spatial relationships in the data. This is the encoding process in an Autoencoder. In the middle, there is a fully connected autoencoder whose hidden layer is composed of only 10 neurons. After that comes with the decoding process that flattens the cubics, then to a 2D flat image. The encoder and the decoder are symmetric in Figure (D). They do not need to be symmetric, but most practitioners just adopt this rule as explained in “Anomaly Detection with Autoencoders made easy”.



Sources/References:

- https://towardsdatascience.com/how-to-easily-deploy-machine-learning-models-using-flask-b95af8fe34d4
- https://www.jeremyjordan.me/autoencoders/
- https://www.askpython.com/python-modules/flask/deploy-ml-models-using-flask
- https://flask.palletsprojects.com/en/2.2.x/debugging/#:~:text=The%20debugger%20is%20enabled%20by,is%20run%20in%20debug%20mode.&text=When%20running%20from%20Python%20code,mode%2C%20which%20is%20mostly%20equivalent.&text=Development%20Server%20and%20Command%20Line,the%20debugger%20and%20debug%20mode.
- https://www.youtube.com/watch?v=AgjKMgPdUFc
- https://blog.devgenius.io/deploy-machine-learning-model-as-a-rest-api-in-a-web-application-e802b9785db6
- https://www.freecodecamp.org/news/machine-learning-web-app-with-flask/