HOML Chapter 17 - Representation Learning and Generative Learning Using Autoencoders and GANs

I. Efficient Data Representations

- Autoencoders are composed of two parts - 1. Encoder (recognition network) - converts inputs to a latent representation, 2. Decoder (generative network) - converts internal representation to the outputs.
- Outputs are referred to as reconstructions because they reconstruct inputs and the cost function contains a reconstruction loss, penalizing the model for differences between the reconstructions and inputs.
- An undercomplete autoencoder has a lower dimensionality than the input data (2D vs. 3D). To copy its inputs to the codings, it needs to learn the most important features of the input data, dropping the unimportant ones.

II, Performing PCA with an Undercomplete Linear Autoencoder

- Autoencoders perform PCA (Principal Component Analysis) if it only uses linear activations and the cost function is MSE (Mean Squared Error).

III. Stacked Autoencoders

- Stacked (Deep) Autoencoders - autoencoders with multiple hidden layers. Symmetrical architecture - input layer has the same units as output layer; hidden layers are also symmetrical.

i. Implementing a Stacked Autoencoder Using Keras

- Can be implemented much like a deep MLP (Multi Layer Perceptron). See page 573 for code.

ii. Visualizing the Reconstructions

- Plot the output images from the validation set against the input to ensure the

model is training correctly. Too much loss may require longer training, deeper layers, or larger codings.

iii. Visualizing the Fashion MNIST Dataset

- Autoencoders can handle large datasets, so a dimensionality reduction algorithm for visualization can be used.
- Use Scikit_Learn's t-SNE for dimensionality reduction for visualizations. Code and visualization on page 575.

iv. Unsupervised Pretraining Using Stacked Autoencoders

- For a large mostly unlabeled dataset, one approach is to train on all of the data, reuse the lower layers to create a neural and then train it using the labeled data. Freezing the lower pretrained layers may be a good approach if the amount of labeled training data is low.

v. Tying Weights

- With a symmetrical autoencoder, one approach to speed up training and limit overfitting is to tie the weights of the decoder layers to the weights of the encoder layers.
- A custom layer can be created that acts as a dense layer, using another dense layer's weights transposed. But it uses its own bias vector. Code on page 577

vi. Training One Autoencoder at a Time

- Also known as greedy layerwise training, training one layer at a time involves the first autoencoder learning to construct the inputs. The whole training set is then encoded using the first autoencoder, resulting in a new compressed training set. The second autoencoder is then trained on this new dataset. This process continues until a final deep stacked autoencoder is constructed.

IV.Convolutional Autoencoders

- Convolutional autoencoders are far better for working with images than dense networks. The encoder is a regular CNN composed of convolutional and pooling layers, typically reducing the spatial dimensionality of the inputs and increasing the number of feature maps.
- The decoder consists of transposed convolutional layers. Code of such an implementation for the MNIST dataset is on page 579.

V. Recurrent Autoencoders

- Recurrent autoencoders are the best option for time series or text sequences. The encore is typically a sequence-to-text vector RNN which compresses the input sequence to a single vector. The decoder is a vector-to-sequence RNN, reversing the process.

VI. Denoising Autoencoders

- Adding noise, either Gaussian or randomly switched-off inputs, can be a technique for forcing an autoencoder to learn useful features.
- Implementation can involve adding a stacked autoencoder with a dropout (or Gaussian) layer applied to the encoder's inputs. Code on page 581.

VII. Sparse Autoencoders

- Sparsity - adding an appropriate term to the cost functions leads the autoencoder to reduce the number of active neurons in the coding layer. Each input is, thus, represented as a combination of small activations.
- A simple approach can include adding a sigmoid activation function in the coding layer, a large coding layer, and L1 regularization applied to the coding layer's activations. Code on page 583.
- A better approach involves measuring the sparsity of the coding layer at each training iteration and penalizing the model when there is a difference between the measured and target sparsity values. After attaining the mean activation per neuron, the under and over performing neurons are penalized by adding a sparsity loss to the cost function.

VIII. Variational Autoencoders

- They possess two main characteristics: 1) Probabilistic Autoencoders - outputs are partly determined by chance, even after training (for denoising autoencoders, randomness only occurs during training). 2) Generative Autoencoders - can generate new instances that appear as though they were sampled from the training set.
- Similar to Restricted Boltzmann Machines (RBM) but easier and faster to train.
- Perform variational Baysian inference.
- Instead of directly producing a coding for a given input, the encoder produces a mean coding and standard deviation. Actual coding is sampled from a Gaussian distribution with a mean and standard distribution. The decoder decodes the sampled coding normally.
- Variational autoencoders produce codings that resemble those sampled from Gaussian distributions but are more general and not limited to Gaussian distributions.
- -The cost function is composed of two parts. 1) Reconstruction loss - pushes the autoencoder to reproduce its inputs. 2) Latent loss - pushes the autoencoder to have codings resembling those sampled from a Gaussian distribution.

i. Generating Fashion MNIST Images

- Semantic Interpolation - interpolate images at the coding level rather than the pixel level. Run images through the encoder, interpolate the codings, and decode the interpolated codings to get the final images.

IX. Generative Adversarial Networks

- Composed of two competing neural networks: 1) Generator - takes a random distribution (usually images) as inputs and outputs data (typically an image). Same functionality as the decoder in a variational autoencoder, but trained much differently. Tries to produce images to fool the discriminator. 2) Discriminator - takes either a fake or real image from the training set as input and guesses whether the input image is fake or real.
- Each training iteration is divided into two phases: 1) Train the discriminator - Batch of real images sampled from the training set is completed with an equal number of fake images from the

generator. The discriminator is trained on images labeled fake or real for one step using binary cross-entropy loss. Backpropagation only affects the weights of the discriminator during this phase. 2) Train the generator - Use it to first produce a batch of fake images, and again after the discriminator is used to determine real vs. fake images. During this next run, real images are not added to the batch and all images are labeled real - the purpose is to make the generator produce images that the discriminator will wrongly believe are fake. The generator never actually sees any real images. Weights of the discriminator are frozen, so backpropagation only affects the weights of the generator.

i. The Difficulties of Training GANs

- A GAN can only reach a single Nash equilibrium: when the generator produces perfectly realistic images and the discriminator is forced to guess. Nothing guarantees the equilibrium can be reached.
- Mode collapse - generator's output gradually becomes less diverse. Generator becomes good at fooling the discriminator with a particular class and eventually only focuses on one class, neglecting improvement on the others. Once the discriminator is no longer fooled, the generator moves onto another class and the cycle repeats, with the generator never becoming good at any class.
- GANs are very sensitive to hyperparameters and a great deal of time can be spent tuning them.
- Experience replay - technique involving storing images in a replay buffer (gaining new and dropping old) and training the discriminator on real and fake images from the buffer. Reduces discriminator overfitting generator's outputs.
- Mini-batch discrimination - measures similarity of images across a batch and sends static to the discriminator to allow it to reject a batch of fake images that lack diversity. Reduces the chances of mode collapse.

ii. Deep Convolutional GANs (DCGANs)

- replace pooling layers with strided convolutions in discriminator and transposed convolutions in the generator.
- use Batch Normalization in both generator and discriminator, expect in generator's output

and discriminator's output.

- remove fully connected hidden layers for deeper architectures.
- use ReLU activation in the generator for all layers except output, which should use tanh. Use leaky ReLU activation in the discriminator for all layers.
- still may need to tune hyperparameters. Implementation on Fashion MNIST on page 599.
- Conditional GAN (CGAN) - adding each image's class as an extra input to both generator and discriminator will allow control of each class of image produced by the generator.

iii. Progressive Growing of GANs

- Nvidia researchers proposed generating small images at the start of training and gradually adding convolutional layers to both the generator (at the end of it) and discriminator (at the beginning of it) to produce larger images.
- Several more techniques were also introduced to increase the diversity of the outputs and to make training more stable.
- Minibatch standard deviation layer - added near the end of the discriminator. Standard deviation is computed across all channels and all instances in the batch for each position in the inputs. The average of the standard deviations results in a single value. A single feature map is added to each instance in the batch. This makes it less likely for the discriminator to be fooled by less diverse generator outputs.
- Equalized learning rate - All weights are initialized with a Gaussian distribution where mean is 0 and standard deviation is 1. The weights are scaled down at runtime by a factor $\text{sqrt}(2/n_{\text{inputs}})$, where n is the number of inputs to the layer. When paired with adaptive gradient optimizers, this technique ensures that parameters have a larger dynamic range (ratio between highest and lowest values). It speeds up and stabilizing training.
- Pixelwise normalization layer - Added after each convolutional layer in the generator. It normalizes each activation based on all the activations in the same image and the same location, but across all channels. This technique avoids explosions in the activations from the generator and discriminator competing.

iv. StyleGANs

- Style transfer techniques are used to ensure that generated images have the same local structure as the training images at every scale, leading to better quality images. Only the generator is modified; the discriminator and loss function are left unchanged. There are two networks:
- Mapping network - maps codings to multiple style vectors. It starts with an eight layer MLP, which maps the latent representations to a vector. THe vector is sent through multiple affine transformations (dense layers with no activation functions), to produce multiple vectors. These vectors control the style of the generated image at different levels, from texture to features.
- Synthesis network - responsible for image generation. It contains a constant (tduring training) learned input, which is processed through multiple convolutional and upsampling layers. However, some noise is added to the input and all outputs of the convolutional layers (before activation) Each noise layer is followed by an Adaptive Instance Normalization (AdaIN), which standardizes each feature map independently by subtracting the feature's mean and dividing it by its standard deviation. The style vector is then used to determine the scale and offset of each feature map.
- Adding noise independently from the codings is crucial. It avoids issues that earlier GANs experienced from adding noise, including resource waste, slowed down training time, and the appearance of visual artifacts. The added noise is different for each level.
- Mixing regularization (style mixing) - percentage of the generated images are produced using two different codings, which leads to the creation of two different style vectors. Images are generated using one vector for the first several levels and the second vector for the remaining levels. The level cutoff is random. This technique prevents the network from assuming staples at adjacent levels are correlated and encourage locality of the GAN. As a result, each style vector only influences a limited number of traits in the generated images.