Deep Learning

MInDS @ Mines

We've covered a variety of feature learning methods and we've covered the simplest type of neural network, the feed forward neural network. In this lecture we cover how to use neural networks for dimensionality reduction and feature learning using autoencoders. We also cover three variations on simple autoencoders; stacked autoencoders, sparse autoencoders, and variational autoencoders.

We've already covered the importance of dimensionality reduction, its uses, and various methods used to reduce the dimensionality of data. In particular we examined principal component analysis, PCA, as a reliable method for linear dimensionality reduction. PCA has the objective of learning a projection of the data in orthogonal dimensions while minimizing the reconstruction error,

$$\begin{aligned} &\min_{\mathsf{W}} ||\mathsf{X} - \mathsf{W}\mathsf{W}^T\mathsf{X}||_F^2, \\ &s.t.\ \mathsf{W}^T\mathsf{W} = \mathsf{I}. \end{aligned} \tag{1}$$

Here, W is the projection matrix transforming the data from the original space to the learned embedding space with orthogonal dimensions maximizing the variance of the data. The resulting embedding, $\mathsf{W}^T\mathsf{X}$ is in r dimensions whereas the original data was in d dimensions, effectively reducing the dimensionality of the data. PCA can be an effective method when the best transformation for the data is a linear one, however it fails to achieve a useful embedding when the data exhibits non-linear patterns. For this, we introduced kernel PCA which applies a kernel to the data, allowing PCA to apply non-linear transformations. When we solve PCA by getting the eigenvalues of the covariance matrix, instead of using the original data, we transform the data using a specified kernel. The approach of kernel PCA can be effective on its own, however with neural networks, we can achieve a more complex learned embedding.

Neural Network Generalization

We looked at feed forward neural networks (FFNN) as the simplest form of neural networks where data is passed forward in only one direction. We can set up neural networks with many neurons per hidden layer and many hidden layers as well to create complex models. Let's instead examine a simpler case of a FFNN with only 2 layers where each layer has only one neuron. We know that the output from neuron i, a, which is neuron a in layer i, that takes an input x is,

$$g_{i,a}(\mathsf{x}) = f(\mathsf{w}_{i,a}^T\mathsf{x}), \tag{2}$$

where $\mathsf{w}_{i,a}$ is the vector of weights learned by neuron i, a, and f is the activation function specified by the neuron. If we are interested in a fully connected FFNN where we use the identity function as the activation function, where $f(\mathsf{X}) = \mathsf{X}$, for simplicity, we can extrapolate to say that the output from layer i can be calculated as,

$$g_i(\mathsf{X}) = \mathsf{W}_i^T \mathsf{X}, \tag{3}$$

where W_i is the weights of all the neurons in layer i. The output from layer j that takes its input from layer i would be,

$$g_j(\mathsf{X}) = \mathsf{W}_j^T \mathsf{W}_i^T \mathsf{X}, \tag{4}$$

With a neural network used for supervised learning we have an objective to minimize the distance between the prediction and the original value such as,

$$\min_{\mathsf{W}} ||\mathsf{y} - \hat{\mathsf{y}}||_p^p. \tag{5}$$

What if instead of predicting some target class, we predict the original input value? This may seem counter intuitive at first but let's look at what happens to our objective. We replace the prediction $\hat{\mathsf{y}}$ with $\mathsf{W}_j^T \mathsf{W}_i^T \mathsf{X}$ and our true value, y, with the original input X to get,

$$\min_{\mathsf{W}} ||\mathsf{X} - \mathsf{W}_j^T \mathsf{W}_i^T \mathsf{X}||_F^2. \tag{6}$$

This objective looks very similar to PCA's objective. It is in fact solving the same objective of minimizing reconstruction error. The main difference is that we do not enforce an orthogonality constraint on our projection matrix. If the number of neurons in the two layers is r, the resulting effect of the neural network is to reduce the input features to r dimensions and then reconstruct the original input back to d dimensions. What is likely to occur is that $\mathsf{W}_j = \mathsf{W}_i^T$ resulting in the same objective as PCA without the orthogonality constraint. This network is an autoencoder.

Autoencoders

An autoencoder is a neural network that learns to encode its input and then output a value that is equivalent to the original input. It is a feed forward network that has two sets of hidden layers, the encoding layer(s) and decoding layer(s). The network starts with high dimensional input and a large number of neurons in the input layer followed by a lower number of neurons per layer until it reaches a desired low number of neurons. It then adds more hidden layers with a larger and larger number of neurons per layer until reaching the intitial input's dimensions. The hidden layers that start with a high neuron count per layer and end with a low neuron count are called the encoding layers since their goal is to encode the input into a lower dimension. The hidden layers that start with a low neuron count and end with the original input's dimension for the neuron count are called the decoding layers since their

goal is to take the encoded data and decode it to produce the input originally passed to the network. With the network we discussed earlier with layers i and j, layer i is the encoding layer and layer j the decoding layer.

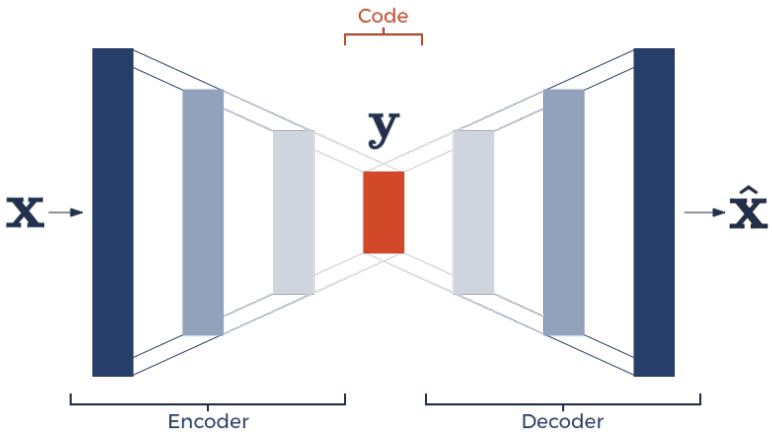

Figure 1: A visualization of autoencoders and how they work.

The encoding layer forces the input to be transformed into a lower dimensional space and the network learns the decoding layer such that it can take the lower dimensional embedding and reconstruct it to the original space. The output from the encoding layer is therefore the learned feature embedding. When the transformations performed by each neuron are linear, the result can be similar to PCA. We can add more complexity to our model by modifying the activation functions to incorporate non-linearity and this can produce much more effective results.

Figure 2: Original images and reconstructions using autoencoders and PCA from "Reducing the Dimensionality of Data with Neural Networks". We can see the autoencoders' ability to generalize better than PCA.

Sparse Autoencoders

Depending on the number of neurons per layer, the model could end up overfitting to the data. One approach to target overfitting is to use sparse autoencoders. Sparse autoencoders add a hyperparameter called the sparsity hyperparameter, ρ. The sparsity hyperparameter controls the average output value per neuron,

$$\hat{\rho}_{ia} = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{w}_{i,a}^{T}\mathbf{x}_{i-1}), \tag{7}$$

where n is the number of datapoints, and $\mathbf{x}_{i-1}$ is the output from the previous layer to i. This approach results in some of the neurons producing an output of 0 and forcing the model to generalize. Note that this approach does not zero out the neurons in all cases. The sparsity parameter forces the model to learn parameters that track patterns in the data based on each sample of data.

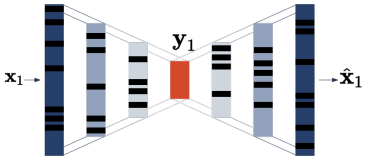

Figure 3: An example of a sparse encoder for one sample of data showing the dead neurons.

Denoising Autoencoders

One issue that may rise with autoencoders is that they are not learning a useful enough feature by ensuring that the encoded embedding doesn't deviate too far from the original.

One way to mitigate this issue is to use denoising autoencoders. Denoising autoencoders add noise to the input data that is passed to that autoencoder but when it comes to calculating the loss function, they calculate it with respect to the clean original value.

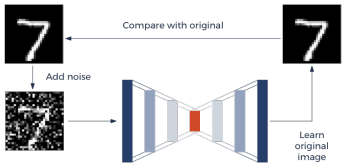

Figure 4: An illustration of denoising autoencoders.

Stacked Autoencoders

Stacked autoencoders are a way of improving on the performance of autoencoders and work very well when combined with a supervised learning problem. The stacked autoencoder can learn a more powerful feature without supervision then utilize that to perform the supervised learning task.

Stacked autoencoders work by learning multiple encoded features. If we consider an autoencoder with the encoding portion consisting of three hidden layers, the stacked autoencoder will train those as three autoencoders. First, we start with the input and hidden layer 1, determining a new encoding with just one layer. Afterwards, we start with the embedding from hidden layer 1 and pass it to hidden layer 2, training the autoencoder to predict hidden layer 1. We repeat this until we've completed all the desired hidden layers.

Once they are each trained separately, we can connect their output to another network with the goal of predicting a target value.

Figure 5: Separate training of each layer before stacking.

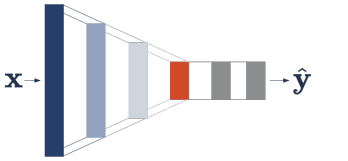

Figure 6: The stacked layers connected to a supervised learning network.