Representation learning: Using deep learning, Bayesian inference and Generative Adversarial Networks

Hamaad Shah

10/04/2018

Disclaimer

This work is not necessarily a representation of any past or current employer of mine.

Introduction

- We will explore the use of deep learning, Bayesian inference and Generative Adversarial Networks (GANs) for automatic feature engineering or representation learning.
- The idea is to automatically learn a set of features from, potentially noisy, raw data that can be useful in supervised learning tasks such as in computer vision and insurance.
- In this manner we avoid the manual process of handcrafted feature engineering by learning a set of features automatically, i.e., representation learning, that can help in solving certain tasks such as image recognition and insurance loss risk prediction.

Computer Vision Task

- We will use the MNIST dataset for this task where the raw data is a 2 dimensional tensor of pixel intensities per image.
- The image is our unit of analysis: We will predict the probability of each class for each image.
- This is a multiclass classification task and we will use the accuracy score to assess model performance on the test fold.

Computer Vision Task

$$\begin{array}{cc} & \begin{array}{ccc} \text{Column}_1 & \dots & \text{Column}_{28} \end{array} \\ \begin{array}{c} \text{Row}_1 \\ \dots \\ \text{Row}_{28} \end{array} & \left(\begin{array}{ccc} 1 & \dots & 2 \\ \dots & \dots & \dots \\ 3 & \dots & 4 \end{array} \right) \end{array} \in \mathbb{R}^{28 \times 28}$$

Figure 1: 2 dimensional tensor of pixel intensities per image.

Insurance Task

- We will use a synthetic dataset where the raw data is a 2 dimensional tensor of historical policy level information per policy-period combination: Per unit this will be $\mathbb{R}^{4\times 3}$, i.e., 4 historical time periods and 3 transactions types.
- The policy-period combination is our unit of analysis: We will predict the probability of loss for time period 5 in the future - think of this as a potential renewal of the policy for which we need to predict whether it would make a loss for us or not hence affecting whether we decided to renew the policy and / or adjust the renewal premium to take into account the additional risk.
- This is a binary class classification task and we will use the AUROC score to assess model performance.

Insurance Task

$$\begin{array}{c c} & \begin{array}{ccc} \text{Paid} & \text{Reserves} & \text{Recoveries} \end{array} \\ \begin{array}{c} \text{Period}_1 \\ \text{Period}_2 \\ \text{Period}_3 \\ \text{Period}_4 \end{array} & \begin{pmatrix} \$0 & \$100 & \$0 \\ \$10 & \$50 & \$0 \\ \$10 & \$15 & \$0 \\ \$100 & \$0 & \$0 \end{pmatrix} \end{array} \in \mathbb{R}^{4\times 3}$$

Figure 2: 2 dimensional tensor of transaction values per policy-period combination.

Supervised Learning

- Please note the similarities between the raw data for the computer vision task and the raw data for the insurance task.
- Our main goal here is to learn a good representation of this raw data using automatic feature engineering via deep learning, Bayesian inference and GANs.

Scikit-learn, Keras and TensorFlow

- We will use the Python machine learning library scikit-learn for data transformation and the classification task.
- We will code the representation learners as scikit-learn transformers such that they can be readily used by scikit-learn pipelines.
- The representation learners will be coded using Keras with the TensorFlow backend.
- We use an external GPU, i.e., GTX 1070, on a MacBook Pro.

Scikit-learn, Keras and TensorFlow

```python
dcgan = DeepConvGenAdvNet(batch_size=100,
                          iterations=10000,
                          z_size=2)

pipe_dcgan = Pipeline(steps=[("DCGAN", dcgan),
                             ("scaler_classifier", scaler_classifier),
                             ("classifier", logistic)])

pipe_dcgan = pipe_dcgan.fit(x_train, y_train)

acc_dcgan = pipe_dcgan.score(x_test, y_test)
```

Vanilla Autoencoders

- An autoencoder is an unsupervised learning technique where the objective is to learn a set of features that can be used to reconstruct the input data.
- Our input data, for instance for the computer vision task, is $X \in \mathbb{R}^{N \times 784}$.
- An encoder function E maps this to a set of K features such that $E : \mathbb{R}^{N \times 784} \to \mathbb{R}^{N \times K}$.
- A decoder function D uses the set of K features to reconstruct the input data such that $D : \mathbb{R}^{N \times K} \to \mathbb{R}^{N \times 784}$.

Vanilla Autoencoders

- Lets denote the reconstructed data as $\tilde{X} = D(E(X))$.
- The goal is to learn the encoding and decoding functions such that we minimize the difference between the input data and the reconstructed data.
- An example for an objective function for this task can be the Mean Squared Error (MSE) such that $\frac{1}{N}||\tilde{X} - X||_2^2$.
- We learn the encoding and decoding functions by minimizing the MSE using the parameters that define the encoding and decoding functions.
- The gradient of the MSE with respect to the parameters are calculated using the chain rule, i.e., backpropagation, and used to update the parameters via an optimization algorithm such as Stochastic Gradient Descent (SGD).

Vanilla Autoencoders

- Lets assume we have a single layer autoencoder using the Exponential Linear Unit (ELU) activation function, batch normalization, dropout and the Adaptive Moment (Adam) optimization algorithm.
- B is the batch size, K is the number of features.

Vanilla Autoencoders

- Exponential Linear Unit:
  - The activation function is smooth everywhere and avoids the vanishing gradient problem as the output takes on negative values when the input is negative.
  - α is taken to be 1.0.

$$H_\alpha(z) = \begin{cases} \alpha\left(\exp(z) - 1\right) \quad \text{if} \quad z < 0 \\ z \quad \text{if} \quad z \geq 0 \end{cases}$$

$$\frac{dH_\alpha(z)}{dz} = \begin{cases} \alpha\left(\exp(z)\right) \quad \text{if} \quad z < 0 \\ 1 \quad \text{if} \quad z \geq 0 \end{cases}$$

Vanilla Autoencoders

- Batch Normalization:
  - The idea is to transform the inputs into a hidden layer's activation functions.
  - We standardize or normalize first using the mean and variance parameters on a per feature basis and then learn a set of scaling and shifting parameters on a per feature basis that transforms the data.

Vanilla Autoencoders

- Batch Normalization:
  - The following equations describe this layer succintly: The parameters we learn in this layer are $\left(\mu_j, \sigma_j^2, \beta_j, \gamma_j\right) \quad \forall j \in \{1, \ldots, K\}$.

$$\mu_j = \frac{1}{B} \sum_{i=1}^{B} X_{i,j} \qquad \forall j \in \{1, \ldots, K\}$$

$$\sigma_j^2 = \frac{1}{B} \sum_{i=1}^{B} (X_{i,j} - \mu_j)^2 \qquad \forall j \in \{1, \ldots, K\}$$

$$\hat{X}_{:,j} = \frac{X_{:,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \qquad \forall j \in \{1, \ldots, K\}$$

$$Z_{:,j} = \gamma_j \hat{X}_{:,j} + \beta_j \qquad \forall j \in \{1, \ldots, K\}$$

Vanilla Autoencoders

- Dropout:
  - This regularization technique simply drops the outputs from input and hidden units with a certain probability say 50%.
  - Dropout can be used for approximate Bayesian inference for deep learners. On this topic there has been great work done by Zoubin Ghahramani's student Yarin Gal.
  - I recommend reading their paper(s): "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning".
- Adam Optimization Algorithm:
  - This adaptive algorithm combines ideas from the Momentum and RMSProp optimization algorithms.
  - The goal is to have some memory of past gradients which can guide future parameters updates.

Vanilla Autoencoders

- Adam Optimization Algorithm:
  - The following equations for the algorithm succintly describe this method assuming θ is our set of parameters to be learnt and η is the learning rate.

$$\begin{aligned} m &\leftarrow \beta_1 m + [(1-\beta_1)(\nabla_\theta \mathrm{MSE})] \\ s &\leftarrow \beta_2 s + [(1-\beta_2)(\nabla_\theta \mathrm{MSE} \otimes \nabla_\theta \mathrm{MSE})] \\ \theta &\leftarrow \theta - \eta m \oslash \sqrt{s+\epsilon} \end{aligned}$$

Denoising Autoencoders

- The idea here is to add some noise to the data and try to learn a set of robust features that can reconstruct the non-noisy data from the noisy data.
- The MSE objective functions is as follows, $\frac{1}{N}||D(E(X+\epsilon)) - X||_2^2$, where ϵ is some noise term.

1 Dimensional Convolutional Autoencoders

- So far we have used flattened or reshaped raw data.
- Such a 1 dimensional tensor of pixel intensities per image, $\mathbb{R}^{784}$, might not take into account useful spatial features that the 2 dimensional tensor, $\mathbb{R}^{28\times 28}$, might contain.
- To overcome this problem, we introduce the concept of convolution filters, considering first their 1 dimensional version and then their 2 dimensional version.

$$X \in \mathbb{R}^{N\times 28\times 28}$$
$$E : \mathbb{R}^{N\times 28\times 28} \to \mathbb{R}^{N\times K}$$
$$D : \mathbb{R}^{N\times K} \to \mathbb{R}^{N\times 28\times 28}$$

1 Dimensional Convolutional Autoencoders

- The ideas behind convolution filters are closely related to handcrafted feature engineering: One can view the handcrafted features as simply the result of a predefined convolution filter, i.e., a convolution filter that has not been learnt based on the raw data at hand.
- Suppose we have raw transactions data per some unit of analysis, i.e., mortgages, that will potentially help us in classifying a unit as either defaulted or not defaulted.
- We will keep this example simple by only allowing the transaction values to be either \$100 or \$0.
- The raw data per unit spans 5 time periods while the defaulted label is for the next period, i.e., period 6.

1 Dimensional Convolutional Autoencoders

- Here is an example of a raw data for a particular unit:

$$x = \begin{array}{c} \text{Period 1} \\ \text{Period 2} \\ \text{Period 3} \\ \text{Period 4} \\ \text{Period 5} \end{array} \left[\begin{array}{c} \$0 \\ \$0 \\ \$100 \\ \$0 \\ \$0 \end{array} \right]$$

1 Dimensional Convolutional Autoencoders

- Suppose further that if the average transaction value is \$20 then we will see a default in period 6 for this particular mortgage unit. Otherwise we do not see a default in period 6.
- The average transaction value is an example of a handcrafted feature: A predefined handcrafted feature that has not been learnt in any manner.
- It has been arrived at via domain knowledge of credit risk. Denote this as $\mathbf{H}(x)$.

1 Dimensional Convolutional Autoencoders

- The idea of learning such a feature is an example of a 1 dimensional convolution filter. As follows:

$$\mathbf{C}(x|\alpha) = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4 + \alpha_5 x_5$$

- Assuming that $\mathbf{H}(x)$ is the correct representation of the raw data for this supervised learning task then the optimal set of parameters learnt via supervised learning, or perhaps unsupervised learning and then transferred to the supervised learning task, i.e., transfer learning, for $\mathbf{C}(x|\alpha)$ is as follows where α is $[0.2, 0.2, 0.2, 0.2, 0.2]$:

$$\mathbf{C}(x|\alpha) = 0.2x_1 + 0.2x_2 + 0.2x_3 + 0.2x_4 + 0.2x_5$$

1 Dimensional Convolutional Autoencoders

- This is a simple example however this clearly illusrates the principle behind using deep learning for automatic feature engineering or representation learning.
- One of the main benefits of learning such a representation in an unsupervised manner is that the same representation can then be used for multiple supervised learning tasks: Transfer learning. This is a principled manner of learning a representation from raw data.

1 Dimensional Convolutional Autoencoders

- To summarize the 1 dimensional convolution filter for our simple example is defined as:

$$\begin{aligned}\mathbf{C}(x|\alpha) &= x * \alpha \\ &= \sum_{t=1}^{5} x_t \alpha_t\end{aligned}$$

- x is the input.
- α is the kernel.
- The output $x * \alpha$ is called a feature map and $*$ is the convolution operator or filter. This is the main difference between a vanilla neural network and a convolution neural network: We replace the matrix multiplication operator by the convolution operator.

1 Dimensional Convolutional Autoencoders

- Depending on the task at hand we can have different types of convolution filters.
- Kernel size can be altered. In our example the kernel size is 5.
- Stride size can be altered. In our example we had no stride size however suppose that stride size was 1 and kernel size was 2, i.e., $\alpha = [\alpha_1, \alpha_2]$, then we would apply the kernel α at the start of the input, i.e., $[x_1, x_2] * [\alpha_1, \alpha_2]$, and move the kernel over the next area of the input, i.e., $[x_2, x_3] * [\alpha_1, \alpha_2]$, and so on and so forth until we arrive at a feature map that consists of 4 real values. This is called a valid convolution while a padded, i.e., say padded with zero values, convolution would give us a feature map that is the same size as the input, i.e., 5 real values in our example.

1 Dimensional Convolutional Autoencoders

- We can apply an activation function to the feature maps such as ELU mentioned earlier.
- Finally we can summarize the information contained in feature maps by taking a maximum or average value over a defined portion of the feature map. For instance, if after using a valid convolution we arrive at a feature map of size 4 and then apply a max pooling operation with size 4 then we will be taking the maximum value of this feature map. The result is another feature map.
- This automates feature engineering however introduces architecture engineering where different architectures consisting of various convolution filters, activation functions, batch normalization layers, dropout layers and pooling operators can be stacked together in a pipeline in order to learn a good representation of the raw data. One usually creates an ensemble of such architectures.

1 Dimensional Convolutional Autoencoders

- The goal behind convolutional autoencoders is to use convolution filters, activation functions, batch normalization layers, dropout layers and pooling operators to create an encoder function which will learn a good representation of our raw data.
- The decoder will also use a similar set of layers as the encoder to reconstruct the raw data with one exception: Instead of using a pooling operator it will use an upsampling operator.
- The basic idea behind the upsampling operator is to repeat an element a certain number of times say size 4: One can view this as the inverse operator to the pooling operator.
- The pooling operator is essentially a downsampling operator and the upsampling operator is simply the inverse of that in some sense.

2 Dimensional Convolutional Autoencoders

- For 2 dimensional convolution filters the idea is similar as for the 1 dimensional convolution filters.
- We will stick to our previously mentioned banking example to illustrate this point.

$$x = \begin{array}{l} \text{Period 1} \\ \text{Period 2} \\ \text{Period 3} \\ \text{Period 4} \\ \text{Period 5} \end{array} \left[\begin{array}{ccc} \$0 & \$0 & \$0 \\ \$0 & \$200 & \$0 \\ \$100 & \$0 & \$0 \\ \$0 & \$0 & \$300 \\ \$0 & \$0 & \$0 \end{array} \right]$$

2 Dimensional Convolutional Autoencoders

- In the 2 dimensional tensor of raw transactions data now we have 5 historical time periods, i.e., the rows, and 3 different transaction types, i.e., the columns.
- We will use a kernel, $\alpha \in \mathbb{R}^{2\times 3}$, to extract useful features from the raw data.
- The choice of such a kernel means that we are interested in finding a feature map across all 3 transaction types and 2 historical time periods.
- We will use a stride length of 1 and a valid convolution to extract features over different patches of the raw data.

2 Dimensional Convolutional Autoencoders

- The following will illustrate this point where $x_{\text{patch}} \subset x$:

$$\begin{aligned}
\alpha &= \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} \end{bmatrix} \\
x_{\text{patch}} &= \begin{bmatrix} \$0 & \$0 & \$0 \\ \$0 & \$200 & \$0 \end{bmatrix} \\
\mathbf{C}(x = x_{\text{patch}} | \alpha) &= x * \alpha \\
&= \sum_{t=1}^{2} \sum_{k=1}^{3} x_{t,k} \alpha_{t,k}
\end{aligned}$$

2 Dimensional Convolutional Autoencoders

The principles and ideas apply to 2 dimensional convolution filters as they do for their 1 dimensional counterparts there we will not repeat them here.

$$X \in \mathbb{R}^{N \times 28 \times 28}$$
$$E : \mathbb{R}^{N \times 28 \times 28} \rightarrow \mathbb{R}^{N \times K}$$
$$D : \mathbb{R}^{N \times K} \rightarrow \mathbb{R}^{N \times 28 \times 28}$$

Sequence to Sequence Autoencoders

- Given our mortgage default example a potentially more useful deep learning architecture might be the Recurrent Neural Network (RNN), specifically their state of the art variant the Long Short Term Memory (LSTM) network.
- The goal is to explicitly take into account the sequential nature of the raw data.

$$X \in \mathbb{R}^{N \times 28 \times 28}$$
$$E : \mathbb{R}^{N \times 28 \times 28} \to \mathbb{R}^{N \times K}$$
$$D : \mathbb{R}^{N \times K} \to \mathbb{R}^{N \times 28 \times 28}$$

Sequence to Sequence Autoencoders

- The gradients in a RNN depend on the parameter matrices defined for the model.
- Simply put these parameter matrices can end up being multiplied many times over and hence cause two major problems for learning: Exploding and vanishing gradients.
- If the spectral radius of the parameter matrices, i.e., the maximum absolute value of the eigenvalues of a matrix, is more than 1 then gradients can become large enough, i.e., explode in value, such that learning diverges and similarly if the spectral radius is less than 1 then gradients can become small, i.e., vanish in value, such that the next best transition for the parameters cannot be reliably calculated.
- Appropriate calculation of the gradient is important for estimating the optimal set of parameters that define a machine learning method and the LSTM network overcomes these problems in a vanilla RNN. We now define the LSTM network for 1 time step, i.e., 1 memory cell.

Sequence to Sequence Autoencoders

- We calculate the value of the input gate, the value of the memory cell state at time period t where $f(x)$ is some activation function and the value of the forget gate:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
\tilde{c}_t &= f(W_c x_t + U_c h_{t-1} + b_c) \\
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f)
\end{aligned}
$$

Sequence to Sequence Autoencoders

- The forget gate controls the amount the LSTM remembers, i.e., the value of the memory cell state at time period $t-1$ where $\otimes$ is the hadamard product:

$$c_t = i_t \otimes \tilde{c}_t + f_t \otimes c_{t-1}$$

- With the updated state of the memory cell we calculate the value of the outputs gate and finally the output value itself:

$$\begin{aligned} o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ h_t &= o_t \otimes f(c_t) \end{aligned}$$

Sequence to Sequence Autoencoders

- We can have a wide variety of LSTM architectures such as the convolutional LSTM where note that we replace the matrix multiplication operators in the input gate, the initial estimate $\tilde{c}_t$ of the memory cell state, the forget gate and the output gate by the convolution operator $*$:

$$
\begin{aligned}
i_t &= \sigma(W_i * x_t + U_i * h_{t-1} + b_i) \\
\tilde{c}_t &= f(W_c * x_t + U_c * h_{t-1} + b_c) \\
f_t &= \sigma(W_f * x_t + U_f * h_{t-1} + b_f) \\
c_t &= i_t \otimes \tilde{c}_t + f_t \otimes c_{t-1} \\
o_t &= \sigma(W_o * x_t + U_o * h_{t-1} + b_o) \\
h_t &= o_t \otimes f(c_t)
\end{aligned}
$$

Sequence to Sequence Autoencoders

- Another popular variant is the peephole LSTM where the gates are allowed to peep at the memory cell state:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + U_i h_{t-1} + V_i c_{t-1} + b_i) \\
\tilde{c}_t &= f(W_c x_t + U_c h_{t-1} + V_c c_{t-1} + b_c) \\
f_t &= \sigma(W_f x_t + U_f h_{t-1} + V_f c_{t-1} + b_f) \\
c_t &= i_t \otimes \tilde{c}_t + f_t \otimes c_{t-1} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + V_o c_t + b_o) \\
h_t &= o_t \otimes f(c_t)
\end{aligned}
$$

Sequence to Sequence Autoencoders

- The goal for the sequence to sequence autoencoder is to create a representation of the raw data using a LSTM as an encoder.
- This representation will be a sequence of vectors say, $h_1, \ldots, h_T$, learnt from a sequence of raw data vectors say, $x_1, \ldots, x_T$. The final vector of the representation, h_T, is our encoded representation, also called a context vector.
- This context vector is repeated as many times as the length of the sequence such that it can be used as an input to a decoder which is yet another LSTM.
- The decoder LSTM will use this context vector to recontruct the sequence of raw data vectors, $\tilde{x_1}, \ldots, \tilde{x_T}$.
- If the context vector is useful in the recontruction task then it can be further used for other tasks such as predicting default risk as given in our example.

Variational Autoencoders

- We now combine Bayesian inference with deep learning by using variational inference to train a vanilla autoencoder.
- This moves us towards generative modelling which can have further use cases in semi-supervised learning.
- The other benefit of training using Bayesian inference is that we can be more robust to higher capacity deep learners, i.e., avoid overfitting.

Variational Autoencoders

- Assume X is our raw data while Z is our learnt representation.
- We have a prior belief on our learnt representation:

$$p(Z)$$

- The posterior distribution for our learnt representation is:

$$p(Z|X) = \frac{p(X|Z)p(Z)}{p(X)}$$

Variational Autoencoders

- The marginal likelihood, $p(X)$, is often intractable causing the posterior distribution, $p(Z|X)$, to be intractable:

$$p(X) = \int_Z p(X|Z)p(Z)dZ$$

- We therefore need an approximate posterior distribution via variational inference that can deal with the intractability.
- This additionally also provides the benefit of dealing with large scale datasets as generally Markov Chain Monte Carlo (MCMC) methods are not well suited for large scale datasets.

Variational Autoencoders

- One might also consider Laplace approximation for the approximate posterior distribution however we will stick with variational inference as it allows a richer set of approximations compared to Laplace approximation.
- Laplace approximation simply amounts to finding the Maximum A Posteriori (MAP) estimate to an augmented likelihood optimization, taking the negative of the inverse of the Hessian at the MAP estimate to estimate the variance-covariance matrix and finally use the variance-covariance matrix with a multivariate Gaussian distribution or some other appropriate multivariate distribution.

Variational Autoencoders

- Assume that our approximate posterior distribution, which is also our probabilistic encoder, is given as:

$$q(Z|X)$$

- Our probabilistic decoder is given by:

$$p(X|Z)$$

Variational Autoencoders

- Given our setup above with regards to an encoder and a decoder let us now write down the optimization problem where θ are the generative model parameters while ϕ are the variational parameters:

$$\log p(X) = \underbrace{D_{KL}(q(Z|X)||p(Z|X))}_{\text{Intractable as p(Z|X) is intractable}} + \underbrace{\mathcal{L}(\theta, \phi|X)}_{\text{Evidence Lower Bound or ELBO}}$$

- Note that $D_{KL}(q(Z|X)||p(Z|X))$ is non-negative therefore that makes the ELBO a lower bound on $\log p(X)$:

$$\log p(X) \geq \mathcal{L}(\theta, \phi|X) \quad \text{as} \quad D_{KL}(q(Z|X)||p(Z|X)) \geq 0$$

Variational Autoencoders

- Therefore we can alter our optimization problem to look only at the ELBO:

$$
\begin{aligned}
\mathcal{L}(\theta, \phi | X) &= \mathbb{E}_{q(Z|X)}\left[\log p(X, Z) - \log q(Z|X)\right] \\
&= \mathbb{E}_{q(Z|X)}\left[\underbrace{\log p(X|Z)}_{\text{Reconstruction error}} + \log p(Z) - \log q(Z|X)\right] \\
&= \mathbb{E}_{q(Z|X)}\left[\underbrace{\log p(X|Z)}_{\text{Reconstruction error}} - \underbrace{D_{KL}(q(Z|X)||p(Z))}_{\text{Regularization}}\right] \\
&= \int_Z \left[\log p(X|Z) - D_{KL}(q(Z|X)||p(Z))\right] q(Z|X) dZ
\end{aligned}
$$

Variational Autoencoders

- The above integration problem can be solved via Monte Carlo integration as $D_{KL}(q(Z|X)||p(Z))$ is not intractable.
- Assuming that the probabilistic encoder $q(Z|X)$ is a multivariate Gaussian with a diagonal variance-covariance matrix we use the reparameterization trick to sample from this distribution say M times in order to calculate the expectation term in the ELBO optimization problem.
- The reparameterization trick in this particular case amounts to sampling M times from the standard Gaussian distribution, multiplying the samples by σ and adding μ to the samples.

Variational Autoencoders

- μ is our learnt representation used for the reconstruction of the raw data. If the learnt representation is useful it can then be used for other tasks as well.
- This is a powerful manner of combining Bayesian inference with deep learning. Variational inference used in this manner can be applied to various deep learning architectures and has further links with the Generative Adversarial Network (GAN).

MNIST Results: Accuracy Scores

- As expected, the best score achieved here is by a 2 dimensional convolutional autoencoder.
- No autoencoders: 92.000000%.
- PCA: 91.430000%.
- Vanilla autoencoder: 96.940000%.
- Denoising autoencoder: 96.930000%.
- 1 dimensional convolutional autoencoder: 97.570000%.
- 2 dimensional convolutional autoencoder: 98.860000%.
- Sequence to sequence autoencoder: 97.600000%.
- Variational autoencoder: 96.520000%.

Insurance Results: AUROC Scores

- The best score achieved on this task is by a vanilla autoencoder.
- This highlights the automation of feature engineering via deep learning: I believe it was Ian Goodfellow who said that a learnt representation is better than a handcrafted representation.
- Note that the sequence to sequence and convolutional autoencoders did not do well on this task simply because of the manner in which I generated the synthetic transactions data: Should the data have been from a process more amenable to sequence to sequence or convolutional autoencoders it is highly likely that these architectures would've performed better.

Insurance Results: AUROC Scores

- Without autoencoders: 92.206261%.
- PCA: 91.128859%.
- Handcrafted features: 93.610635%.
- Handcrafted features and PCA: 93.160377%.
- Vanilla autoencoder: 93.932247%.
- Denoising autoencoder: 93.712479%.
- 1 dimensional convolutional autoencoder: 91.509434%.
- 2 dimensional convolutional autoencoder: 92.645798%.
- Sequence to sequence autoencoder: 91.418310%.
- Variational autoencoder: 90.871569%.

Generative Adversarial Networks

- The purpose of deep learning is to learn a representation of high dimensional and noisy data using a sequence of differentiable functions, i.e., geometric transformations, that can perhaps be used for supervised learning tasks among others.
- It has had great success in discriminative models while generative models have fared worse due to the limitations of explicit maximum likelihood estimation (MLE).
- Adversarial learning as presented in the Generative Adversarial Network (GAN) aims to overcome these problems by using implicit MLE.

Generative Adversarial Networks

- We will use the MNIST computer vision dataset and a synthetic financial transactions dataset for an insurance task for these experiments.
- GAN is a remarkably different method of learning compared to explicit MLE. Our purpose will be to show that the representation learnt by a GAN can be used for supervised learning tasks such as image recognition and insurance loss risk prediction.
- In this manner we avoid the manual process of handcrafted feature engineering by learning a set of features automatically, i.e., representation learning.

Generative Adversarial Networks

- There are 2 main components to a GAN, the generator and the discriminator, that play an adversarial game against each other.
- In doing so the generator learns how to create realistic synthetic samples from noise, i.e., the latent space z, while the discriminator learns how to distinguish between a real sample and a synthetic sample.
- The representation learnt by the discriminator can later on be used for other supervised learning tasks, i.e., automatic feature engineering or representation learning.

Generative Adversarial Networks

Generator

- Assume that we have a prior belief on where the latent space z lies: $p(z)$.
- Given a draw from this latent space the generator G, a deep learner parameterized by θ_G, outputs a synthetic sample.
  - $G(z|\theta_G) : z \rightarrow x_{synthetic}$

Generative Adversarial Networks

Discriminator

- The discriminator D is another deep learner parameterized by θ_D and it aims to classify if a sample is real or synthetic, i.e., if a sample is from the real data distribution: $p_{\text{data}}(x)$
- Or the synthetic data distribution: $p_G(x)$
- Let us denote the discriminator D as follows.
  - $D(x|\theta_D) : x \to [0, 1]$
- Here we assume that the positive examples are from the real data distribution while the negative examples are from the synthetic data distribution.

Generative Adversarial Networks

Game

- A GAN simultaneously trains the discriminator to correctly classify real and synthetic examples while training the generator to create synthetic examples such that the discriminator incorrectly classifies real and synthetic examples.
- This 2 player minimax game has the following objective function.

$$\min_{G(z|\theta_G)} \max_{D(x|\theta_D)} V(D(x|\theta_D), G(z|\theta_G)) = \mathbb{E}_{x \sim p_{\text{data}}(x)} \log D(x|\theta_D) + \mathbb{E}_{z \sim p(z)} \log\left(1 - D(G(z|\theta_G)|\theta_D)\right)$$

Generative Adversarial Networks

Game

- Please note that the above expression is basically the objective function of the discriminator.

$$\mathbb{E}_{x\sim p_{\text{data}}(x)} \log D(x|\theta_D) + \mathbb{E}_{x\sim p_G(x)} \log\left(1 - D(x|\theta_D)\right)$$

- It is clear from how the game has been set up that we are trying to obtain a solution θ_D for D such that it maximizes $V(D, G)$ while simultaneously we are trying to obtain a solution θ_G for G such that it minimizes $V(D, G)$.

Generative Adversarial Networks

Game

- We do not simultaneously train D and G. We train them alternately: Train D and then train G while freezing D. We repeat this for a fixed number of steps.
- If the synthetic samples taken from the generator G are realistic then implicitly we have learnt the distribution $p_G(x)$. In other words, $p_G(x)$ can be seen as a good estimation of $p_{\text{data}}(x)$.
- The optimal solution will be as follows.

$$p_G(x) = p_{\text{data}}(x)$$

Generative Adversarial Networks

Game

- To show this let us find the optimal discriminator D^* given a generator G and sample x.

$$\begin{aligned}
V(D, G) &= \mathbb{E}_{x\sim p_{\text{data}}(x)} \log D(x|\theta_D) + \mathbb{E}_{x\sim p_G(x)} \log(1 - D(x|\theta_D)) \\
&= \int_x p_{\text{data}}(x) \log D(x|\theta_D) dx + \int_x p_G(x) \log(1 - D(x|\theta_D)) dx \\
&= \int_x \underbrace{p_{\text{data}}(x) \log D(x|\theta_D) + p_G(x) \log(1 - D(x|\theta_D))}_{J(D(x|\theta_D))} dx
\end{aligned}$$

Generative Adversarial Networks

Game

- Let us take a closer look at the discriminator's objective function for a sample x.

$$
\begin{aligned}
J(D(x|\theta_D)) &= p_{\text{data}}(x) \log D(x|\theta_D) + p_G(x) \log (1 - D(x|\theta_D)) \\
\frac{\partial J(D(x|\theta_D))}{\partial D(x|\theta_D)} &= \frac{p_{\text{data}}(x)}{D(x|\theta_D)} - \frac{p_G(x)}{(1 - D(x|\theta_D))} \\
0 &= \frac{p_{\text{data}}(x)}{D^*(x|\theta_{D*})} - \frac{p_G(x)}{(1 - D^*(x|\theta_{D*}))} \\
p_{\text{data}}(x)(1 - D^*(x|\theta_{D*})) &= p_G(x)D^*(x|\theta_{D*}) \\
p_{\text{data}}(x) - p_{\text{data}}(x)D^*(x|\theta_{D*})) &= p_G(x)D^*(x|\theta_{D*}) \\
p_G(x)D^*(x|\theta_{D*}) + p_{\text{data}}(x)D^*(x|\theta_{D*})) &= p_{\text{data}}(x) \\
D^*(x|\theta_{D*}) &= \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}
\end{aligned}
$$

Generative Adversarial Networks

Game

- We have found the optimal discriminator given a generator. Let us focus now on the generator's objective function which is essentially to minimize the discriminator's objective function.

$$\begin{aligned}
J(G(x|\theta_G)) &= \mathbb{E}_{x\sim p_{\text{data}}(x)} \log D^*(x|\theta_{D^*}) + \mathbb{E}_{x\sim p_G(x)} \log\left(1 - D^*(x|\theta_{D^*})\right) \\
&= \mathbb{E}_{x\sim p_{\text{data}}(x)} \log\left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right) + \mathbb{E}_{x\sim p_G(x)} \log\left(1 - \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right) \\
&= \mathbb{E}_{x\sim p_{\text{data}}(x)} \log\left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right) + \mathbb{E}_{x\sim p_G(x)} \log\left(\frac{p_G(x)}{p_{\text{data}}(x)+p_G(x)}\right) \\
&= \int_x p_{\text{data}}(x) \log\left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right) dx + \int_x p_G(x) \log\left(\frac{p_G(x)}{p_{\text{data}}(x)+p_G(x)}\right) dx
\end{aligned}$$

- We will note the Kullback–Leibler (KL) divergences in the above objective function for the generator:

$$D_{KL}(P||Q) = \int_x p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

Generative Adversarial Networks

Game

- Recall the definition of a λ divergence.

$$D_\lambda(P||Q) = \lambda D_{KL}(P||\lambda P + (1-\lambda)Q) + (1-\lambda)D_{KL}(Q||\lambda P + (1-\lambda)Q)$$

- If λ takes the value of 0.5 this is then called the Jensen-Shannon (JS) divergence. This divergence is symmetric and non-negative.

$$D_{JS}(P||Q) = 0.5D_{KL}\left(P\middle|\middle|\frac{P+Q}{2}\right) + 0.5D_{KL}\left(Q\middle|\middle|\frac{P+Q}{2}\right)$$

Generative Adversarial Networks

Game

- Keeping this in mind let us take a look again at the objective function of the generator.

$$
\begin{aligned}
J(G(x|\theta_G)) &= \int_x p_{\text{data}}(x)\log\left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right)dx + \int_x p_G(x)\log\left(\frac{p_G(x)}{p_{\text{data}}(x)+p_G(x)}\right)dx \\
&= \int_x p_{\text{data}}(x)\log\left(\frac{2}{2}\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right)dx + \int_x p_G(x)\log\left(\frac{2}{2}\frac{p_G(x)}{p_{\text{data}}(x)+p_G(x)}\right)dx \\
&= \int_x p_{\text{data}}(x)\log\left(\frac{1}{2}\frac{1}{0.5}\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_G(x)}\right)dx + \int_x p_G(x)\log\left(\frac{1}{2}\frac{1}{0.5}\frac{p_G(x)}{p_{\text{data}}(x)+p_G(x)}\right)dx \\
&= \int_x p_{\text{data}}(x)\left[\log(0.5)+\log\left(\frac{p_{\text{data}}(x)}{0.5(p_{\text{data}}(x)+p_G(x))}\right)\right]dx \\
&+ \int_x p_G(x)\left[\log(0.5)+\log\left(\frac{p_G(x)}{0.5(p_{\text{data}}(x)+p_G(x))}\right)\right]dx
\end{aligned}
$$

Generative Adversarial Networks

Game

$$
\begin{aligned}
&= \log\left(\frac{1}{4}\right) + \int_x p_{\text{data}}(x)\left[\log\left(\frac{p_{\text{data}}(x)}{0.5(p_{\text{data}}(x)+p_G(x))}\right)\right]dx \\
&+ \int_x p_G(x)\left[\log\left(\frac{p_G(x)}{0.5(p_{\text{data}}(x)+p_G(x))}\right)\right]dx \\
&= -\log(4) + D_{KL}\left(p_{\text{data}}(x)\middle|\middle|\frac{p_{\text{data}}(x)+p_G(x)}{2}\right) + D_{KL}\left(p_G(x)\middle|\middle|\frac{p_{\text{data}}(x)+p_G(x)}{2}\right) \\
&= -\log(4) + 2\left(0.5D_{KL}\left(p_{\text{data}}(x)\middle|\middle|\frac{p_{\text{data}}(x)+p_G(x)}{2}\right) + 0.5D_{KL}\left(p_G(x)\middle|\middle|\frac{p_{\text{data}}(x)+p_G(x)}{2}\right)\right) \\
&= -\log(4) + 2D_{JS}(p_{\text{data}}(x)||p_G(x))
\end{aligned}
$$

Generative Adversarial Networks

Game

- It is clear from the objective function of the generator above that the global minimum value attained is $-\log(4)$ which occurs when the following holds.

$$p_G(x) = p_{\text{data}}(x)$$

- When the above holds the Jensen-Shannon divergence, i.e., $D_{JS}(p_{\text{data}}(x)||p_G(x))$, will be zero. Hence we have shown that the optimal solution is as follows.

$$p_G(x) = p_{\text{data}}(x)$$

Generative Adversarial Networks

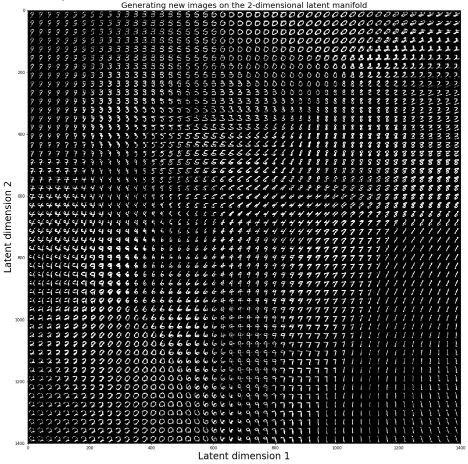
Results

- In these experiments we show the ability of the generator to create realistic synthetic examples for the MNIST dataset and the insurance dataset. We use a 2-dimensional latent manifold.
- Finally we show that using the representation learnt by the discriminator we can attain competitive results to using other representation learning methods for the MNIST dataset and the insurance dataset such as a wide variety of autoencoders.

Generative Adversarial Networks

Results

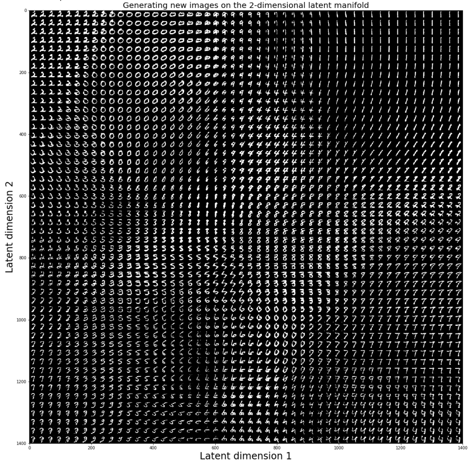
Deep Convolutional Generative Adversarial Network (DCGAN) with a 2-dimensional latent manifold
Generating new images on the 2-dimensional latent manifold



Generative Adversarial Networks

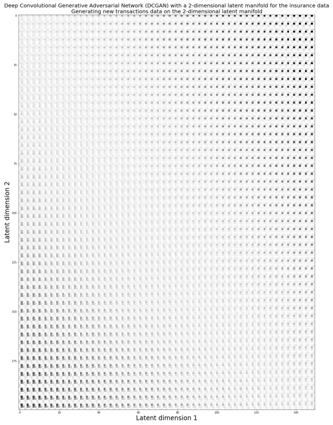
Results

Deeper Convolutional Generative Adversarial Network (DCGAN) with a 2-dimensional latent manifold
Generating new images on the 2-dimensional latent manifold



Generative Adversarial Networks

Results



Generative Adversarial Networks

Results

- The accuracy score for the MNIST classification task with DCGAN: 98.350000%.
- The accuracy score for the MNIST classification task with Deeper CGAN: 99.090000%.
- The AUROC score for the insurance classification task with DCGAN: 92.795883%.

Generative Adversarial Networks

The insurance data: A closer look

- With image data we can perhaps judge qualitatively whether the generated data makes sense. For financial transactions data this is not possible.
- However let's have a look at an example of a generated transactions lattice.
- Please note that all financial transactions data has been transformed to lie between 0 and 1.

Generative Adversarial Networks

The insurance data: A closer look

$$\begin{array}{c|ccc} & \text{Paid} & \text{Reserves} & \text{Recoveries} \\ \hline \text{Period}_1 & 0.995 & 0.519 & 0.013 \\ \text{Period}_2 & 0.001 & 0.040 & 0.584 \\ \text{Period}_3 & 0.128 & 0.422 & 0.560 \\ \text{Period}_4 & 0.006 & 0.028 & 0.006 \end{array} \in \mathbb{R}^{4\times 3}$$

Figure 6:

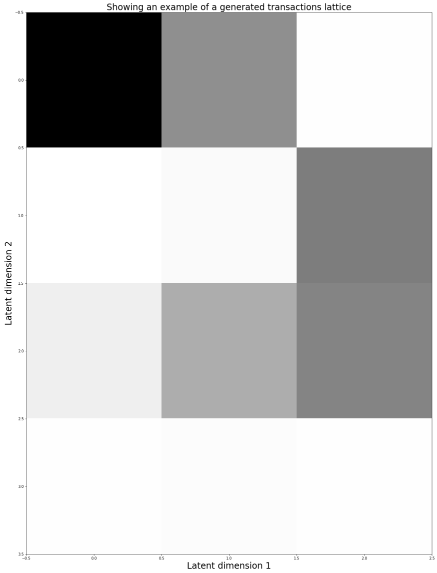
Generative Adversarial Networks

The insurance data: A closer look

- If we use the same matplotlib code as applied to the image data to plot the above generated transactions lattice we get the following image.
- We can see that where we have the maximum value possible for a transaction, i.e., 1, that is colored as black, while where we have the minimum value possible for a transaction, i.e., 0, that is colored as white.
- Transactions values in between have some gray color.

Generative Adversarial Networks

The insurance data: A closer look



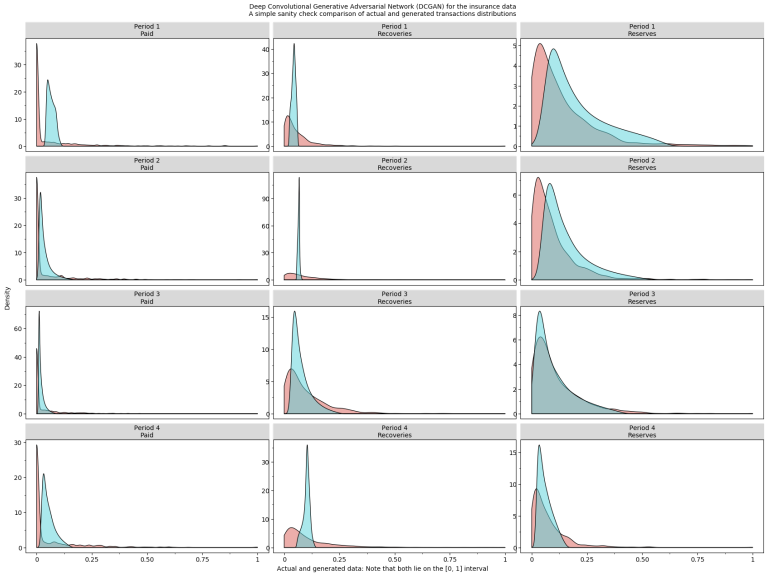
Generative Adversarial Networks

The insurance data: A closer look

- Finally let us compare the distributions of actual and generated transactions lattices to see whether generated values are similar to actual values. This is a simple sanity check and it seems appears that the distributions are fairly similar.
- Another way perhaps is to check if the features learnt by the discriminator are useful for a supervised learning task. This seems to be the case in our insurance data example.

Generative Adversarial Networks

The insurance data: A closer look



Conclusion

- We have shown how to use deep learning, Bayesian inference and Generative Adversarial Networks to learn a good representation of raw data, i.e., 1 or 2 dimensional tensors per unit of analysis, that can then perhaps be used for supervised learning tasks in the domain of computer vision and insurance.
- This moves us away from manual handcrafted feature engineering towards automatic feature engineering, i.e., representation learning.
- GANs can perhaps be also used for semi-supervised learning which will be the topic of another paper.

References

1. Goodfellow, I., Bengio, Y. and Courville A. (2016). Deep Learning (MIT Press).
2. Geron, A. (2017). Hands-On Machine Learning with Scikit-Learn & Tensorflow (O'Reilly).
3. Radford, A., Luke, M. and Chintala, S. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (https://arxiv.org/abs/1511.06434).
4. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2014). Generative Adversarial Networks (https://arxiv.org/abs/1406.2661).

References

5. http://scikit-learn.org/stable/#
6. https://towardsdatascience.com/
   learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-l
7. https://stackoverflow.com/questions/42177658/
   how-to-switch-backend-with-keras-from-tensorflow-to-theano
8. https://blog.keras.io/building-autoencoders-in-keras.html
9. https://keras.io
10. https://github.com/fchollet/keras/blob/master/examples/
    mnist_acgan.py#L24
11. https://en.wikipedia.org/wiki/Kullback\T1\
    textendashLeibler_divergence

Blog on Medium and Code on GitHub

- On GitHub:
  - https://github.com/hamaadshah/autoencoders_public
  - https://github.com/hamaadshah/gan_public
- On Medium:
  - Automatic feature engineering using deep learning and Bayesian inference.
  - Automatic feature engineering using Generative Adversarial Networks.