Institut de Mathématiques de Toulouse, INSA Toulouse

Autoencoders and Generative Adversarial Netwoks

High Dimensional and Deep Learning
INSA- Applied Mathematics

Béatrice Laurent

Outline

- We have seen how to learn deep neural networks for regression or classification.
- In this course, we will see how to use the deep neural network to learn sparse representation of the data, and to generate new samples.
- The outline of the course is the following :
  - Autoencoders
  - Variational Autoencoders (VAE)
  - Generative Adversarial Networks (GAN)

Autoencoders

- An **autoencoder** is a neural network trained to learn an efficient data coding in an unsupervised way.
- The aim is to learn a sparse representation of the data, typically for the purpose of dimension reduction.
- Find underlying structures in the data, for clustering, visualization, more robust supervised algorithms.
- For complex data such as signals, images, text, there are various hidden latent structures in the data that we try to capture with the autoencoders.

Autoencoders

- An autoencoder learns to compress the data into a short code (**encoder**) and then uncompress that code to reconstruct something which is close to the original data (**decoder**).
- This forces the autoencoder to reduce the dimension, for example by learning how to ignore the noise, and by learning features in the original data.
- The autoencoder has a hidden layer, **h** that describes a **code** used to represent the input, and that is forced to provide a smart compression of the data. The autoencoder is hence a feedforward network consisting in two parts :
  - **An encoder function :** $\mathbf{h} = \mathbf{f}(\mathbf{x})$
  - **A decoder function** that provides a reconstruction : $\mathbf{r} = \mathbf{g}(\mathbf{h})$.

Autoencoders

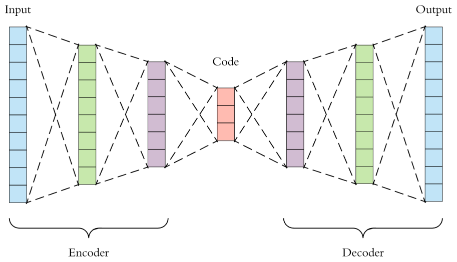


FIGURE – The structure of an autoencoder : mapping the input data to a reconstruction via a sparse internal representation code. Source : *towardsdatascience.com*

Hence, an autoencoder is a neural network trained to "copy" its input into its output, hence to learn the Identity function.
Of course, this would be useless without additional constraints, forcing the **code** to be sparse.

Autoencoders

- Encoders and decoders have arbitrary architectures such as Convolutional Neural Networs, Recurrent Neural Networks ..
- In order to obtain useful features from the autoencoder, the code $\mathbf{h}$ is forced to have a smaller dimension than the input data $\mathbf{x}$.
- As usual, the learning results from the minimization of a loss function $L(\mathbf{x}, g(f(\mathbf{x})))$ (for example the quadratic loss), with a regularization (or additional constraints) to enforce the sparsity of the code.
- When the encoder and decoder are linear and L is the quadratic loss, the autoencoder is equivalent to a PCA ; with nonlinear encoder and decoder functions, such as neural networks, we get nonlinear generalizations of the PCA, which might be more efficient.
- The regularization will lead the autoencoder to provide a sparse representation, with small derivatives, and to be robust to the noise and to missing data.

Regularization for autoencoders

- The first kind of regularization is to induce sparsity.
- For sparse autoencoders, we add to the quadratic loss a sparsity penalty $\Omega(\mathbf{h})$ on the code $\mathbf{h}$, this leads to minimize

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}).$$

- This kind of autoencoders are typically used to learn features for classification purposes.
- This criterions can also be reinterpreted as the likelihood of a generative model with latent variables. Assume that the model has observable variables $\mathbf{x}$ and latent variables $\mathbf{h}$ with joint distribution $p(\mathbf{x}, \mathbf{h}) = p(\mathbf{h})p(\mathbf{x}/\mathbf{h})$. We can consider $p(\mathbf{h})$ as the model's distribution over the latent variables $\mathbf{h}$.

Regularization for autoencoders

- We have
$$\log p(\mathbf{x}, \mathbf{h}) = \log p(\mathbf{h}) + \log p(\mathbf{x}/\mathbf{h}).$$
- The term $\log p(\mathbf{h})$ can induce sparsity. This is the case for example for a Laplace prior :
$$p(\mathbf{h_i}) = \frac{\lambda}{2} e^{-\lambda |h_i|},$$
which leads to
$$-\log p(\mathbf{h}) = \sum_i \left(\lambda |h_i| - \log(\frac{\lambda}{2}) \right) = \Omega(\mathbf{h}) + C,$$
where C is constant (independent of $\mathbf{h}$). In this interpretation, the sparsity can be viewed as a consequence of the model's distribution on the latent variables.

Regularization for autoencoders

- A second kind of regularization is achieved by **denoising autoencoders**.
- Instead of minimizing the loss function $L(\mathbf{x}, g(f(\mathbf{x})))$, a denoising autoencoder minimizes $L(\mathbf{x}, g(f(\tilde{\mathbf{x}})))$, where $\tilde{\mathbf{x}}$ is a copy of $\mathbf{x}$ where some noise has been added.
- Hence the denoising autoencoder has to remove this noise to ressemble to the original object $\mathbf{x}$.
- The procedure can be summarized as follows :
  - Sample a training example x from the training data
  - Sample a noised version $\tilde{x}$ from a given corruption process $C(\tilde{\mathbf{x}}/\mathbf{x} = x)$.
  - Use $(x, \tilde{x})$ as a training example to estimate the encoder reconstruction distribution $p(\mathbf{x}/\tilde{\mathbf{x}})$, by minimizing the loss $L = -\log p(\mathbf{x}/\mathbf{h} = f(\tilde{\mathbf{x}}))$.

Regularization for autoencoders

- A third kind of regularization method consists in a penalization of the derivatives by minimizing the criterion

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}, \mathbf{x}),$$

  where

$$\Omega(\mathbf{h}, \mathbf{x}) = \lambda \sum_i \|\nabla_x h_i\|^2.$$

- By this way, we force the encoder **h** to be robust to small variations in **x**.
- Other kinds of autoencoders : Variational AutoEncoders (VAE) have been introduced by Kingma (2013), Rezende et al (2014).

Variational autoencoders

- We have seen that the encoders convert the input data into an encoding vector, corresponding to a sparse representation of the data.
- Each component of the encoding vector corresponds to some learned feature of the input data.
- An important fact is that, for each encoding dimension, a single value is given by the encoder.
- The decoder then takes these values as input to reconstruct the original data.
- **The variational autoencoder (VAE)** does not more return a single value for each component of the encoding vector, but a probability distribution.

Variational autoencoders

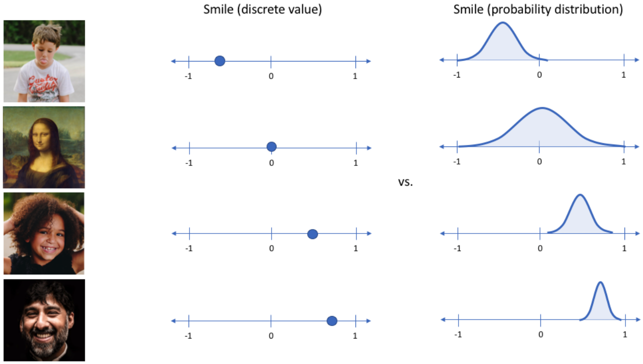


FIGURE – Source : https ://www.jeremyjordan.me

Variational autoencoders

- In VAEs, each latent variable is represented by a probability distribution, and to provide the inputs of the decoder, we sample from this probability distribution for each component of the encoding vector.
- The decoder should accurately reconstruct the input whatever the sampling for the encoding vector.
- By this way, we enforce close values in the latent space to provide quite similar reconstructions. Hence the reconstruction is smooth with respect to the latent space representation.

Variational autoencoders

- Let us assume that a hidden variable z with know prior distribution $p(z)$ (generally standard Gaussian variables) generates the observation x. We only observe x, we want to infer the distribution of z given x.
- We have $p(z/x) = \frac{p(x/z)p(z)}{p(x)}$, but $p(x)$ is intractable to compute in general.
$$p(x) = \int p(x/z)p(z)dz$$
which is generally in integral in high dimension.
- The **variational approach** consists in approximating $p(z/x)$ by a simple distribution $q(z/x)$ (for example a normal distribution). The parameters of $q(z/x)$ (mean and standard deviation for a normal distribution) will be estimated in such a way that $q(z/x)$ is close to $p(z/x)$.
- We aim at minimizing $KL(q(z/x), p(z/x))$ (where KL denotes the Kullback-Leibler divergence).

Variational autoencoders

$$KL(q(z/x), p(z/x)) = -\int q(z/x) \log\left(\frac{p(z/x)}{q(z/x)}\right) dz$$

- $p(z/x) = p(x,z)/p(x)$ and $\int q(z/x)dz = 1$, hence we have

$$KL(q(z/x), p(z/x)) = -\int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) + \log(p(x)).$$

- Equivalently,

$$\log(p(x)) = KL(q(z/x), p(z/x)) + \int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) dz.$$

- $\log(p(x))$ is a fixed quantity (with respect to $q(z/x)$).
- Hence, minimizing $KL(q(z/x), p(z/x))$ is equivalent to maximize $\int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) dz$.

Variational autoencoders

- We want to maximize $\int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) dz$. This is equal to

$$\begin{aligned} & \int q(z/x) \log\left(\frac{p(x/z)p(z)}{q(z/x)}\right) dz \\ = & \int q(z/x) \log\left(p(x/z)\right) dz + \int q(z/x) \log\left(\frac{p(z)}{q(z/x)}\right) dz \\ = & \mathbb{E}_{z\sim q(z/x)} \log(p(x/z)) - KL(q(z/x), p(z)). \end{aligned}$$

- We recall that we choose a very simple prior on z, $p(z)$: generally independent standard normal distributions, we also choose a simple distribution for $q(z/x)$: generally independent normal distributions.
- The means and standard deviation of the $q(z/x)$ are learned to maximize the above quantity, as well as the parameters of the neural networks corresponding to the encoder and decoder.

Variational autoencoders

We maximize $\mathbb{E}_{z \sim q(z/x)} \log(p(x/z)) - KL(q(z/x), p(z))$.

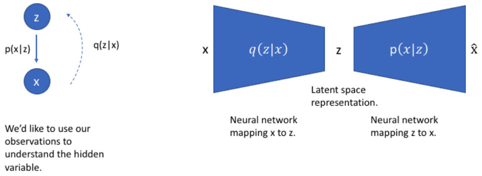

z
p(x|z)
x
q(z|x)
We'd like to use our observations to understand the hidden variable.
x
$q(z|x)$
z
$p(x|z)$
$\hat{x}$
Latent space representation.
Neural network mapping x to z.
Neural network mapping z to x.

FIGURE – Source : https ://www.jeremyjordan.me

If we consider the loss function associated to the negative log-likelihood, we have

$$\ell(x, \hat{x}) = -\mathbb{E}_{z \sim q(z/x)} \log(p(x/z))$$

The VAE is trained by minimizing the penalized loss function :

$$\ell(x, \hat{x}) + KL(q(z/x), p(z)).$$

Variational autoencoders

- For the sake of simplification, we assume here that $q(z/x)$ is Gaussian with independent components. Instead of computing the latent variables as classical autoencoders, the VAE gives as output of the encoder the estimated mean and standard deviation for each latent variable z_j. The decoder will then sample from this distribution to compute $\hat{x}$.

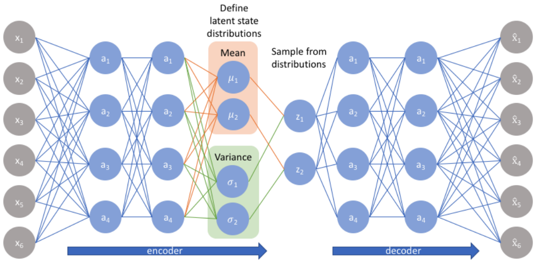
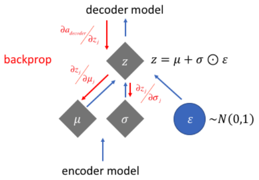


FIGURE – Source : https ://www.jeremyjordan.me

Variational autoencoders

- All the parameters are estimated by using the backpropagation equations.
- The problem is that here we have random variables (the variables z_j) in the network.
- In order to overcome this problem, we use a reparametrization trick, schematized below :



decoder model
$\partial a_{decoder} / \partial z_j$
backprop
z
$z = \mu + \sigma \odot \varepsilon$
$\partial z_j / \partial \mu_j$
$\partial z_j / \partial \sigma_j$
μ
σ
ε
$\sim N(0,1)$
encoder model

FIGURE – Source : https ://www.jeremyjordan.me

Variational autoencoders

- It is interesting to visualize the latent variables to interpret these features learned from the data.
- Variational autoencoder can be use to generate new data : we generate z from the learned distribution $q(z/x)$ and we the compute the corresponding $\hat{x}$ from the decoder.
- This provides smooth transformations of the original data, which can be interesting when we want to interpolate between observations (for example images or sound).

- Autoencoders have many applications : they can be used for classification considering the latent code as input to a classifier instead of $\mathbf{x}$.
- The may also be used for semi-supervised learning for simultaneous learning of the latent code on a large unlabeled dataset and then the classifier on a smaller labeled dataset.
- They are also widely used for Generative models.
- They nevertheless have some limitations : autoencoders fail to capture a good representation of the data for complex objects such as images ; the generative models are sometimes of poor quality (blurry images for example).
- Another way to generate new samples is to use Generative Adversarial Networks (GAN) presented in the next section.

Generative Adversarial Networks

- **Generative Adversarial Networks (GANs)** are generative algorithms that are known to produce state-of-the art samples, especially for image creation inpainting, speech and 3D modeling.
- They were introduced by Goodfellow et al (2014) The more recent advances are presented in Goodfellow (2016).
- The aim of GANS is to produce fake observations of a given distribution p^* from which we only have a true sample (for example faces images).
- This data are generally complex and it is impossible to consider a parametric model to capture the complexity of p^* nor to consider nonparametric estimators.
- In order to generate new data from the distribution p^*, GANs trains simultaneously two models :
  - a generative model G that captures the data distribution
  - a discriminative model D that estimates the probability that a sample comes from the training data rather than from the generator G.

Generative Adversarial Networks

- The generator admits as input a random noise (such as Gaussian or Uniform), and eventually latent variables with small dimension, and tries to transform them into fake data that match the distribution p^* of the real data.
- The generator and the discriminator have conflicting goals : simultaneously, the discriminator tries to distinguish the true observations from the fake ones produced by the generator and the generator tries to generated sample that are as undistinguishable as possible from the original data.
- In practice, G and D are defined by multilayers perceptrons learned from the original data via the optimization of a suitable objective function.
- Figure 24 gives a schematic representation of GANs.

Generative Adversarial Networks

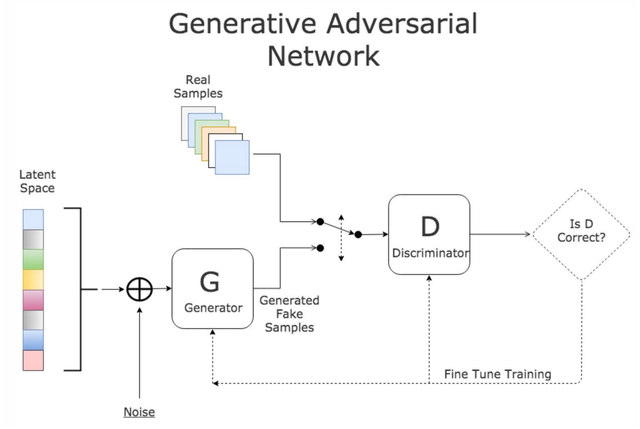


FIGURE – Source : blog.statsbot.co

Generative Adversarial Networks

- In the original paper (Goodfellow, 2014), the input of the generator is only a random noise z from distribution p_z.
- The **generator** is a multilayer perceptron with parameters θ_g and denoted $G(z, \theta_g)$**.**
- We also define another multilayer perceptron : **the discriminator** $D(x, \theta_d)$ with values in $[0, 1]$, that represents the probability that x comes from the real data rather than from the generator.
- D is trained to maximize the probability to assign the correct labels to the real (training) data as well as to the fake ones.
- Hence, $D(x)$ should be close to 1 when x is a real data, and close to 0 otherwise, namely $D(G(z))$ should be close to 0.

Generative Adversarial Networks

- At the same time, G is trained to maximize $D(G(z))$, or equivalently to minimize $\log(1 - D(G(z)))$; indeed $D(G(z))$ should be close to 1 so that $G(z)$ is undistinguishable from the real data.
- In other words, D and G play a two-players minimax game :

$$\min_G \max_D V(D, G)$$

where

$$V(D, G) = \mathbb{E}_{x \sim p^*}(\log(D(x))) + \mathbb{E}_{z \sim p_z}(\log(1 - D(G(z))).$$

- For a given generator, the optimal solution for the discriminator that realizes $\max_D V(D, G)$ is

$$D^*(x) = \frac{p^*(x)}{p^*(x) + p_g(x)},$$

where p_g is the distribution of $G(z)$.

Generative Adversarial Networks

- Concerning the minimax game, in the space of arbitrary functions G and D, it can be shown that the optimal solution is that $G(z)$ recovers exactly the training data distribution ($p_g = p^*$) and D equals $1/2$ everywhere.
- In practice, D and G are trained from the original data X_i and the simulated random noise Z_i to realize

$$\min_G \max_D \left[\sum_{i=1}^{n} \log(D(X_i)) + \sum_{i=1}^{n} \log(1 - D(G(Z_i))) \right].$$

Goodfellow et al (2014) propose the following algorithm :

Algorithm : Minibatch SGD training of GANs.

```
for $I$ training iterations do
    for $k$ steps do
        • Sample a minibatch of $m$ noise samples $z^{(1)}, \dots, z^{(m)}$ from noise prior $p_z$
        • Sample a minibatch of $m$ examples $x^{(1)}, \dots, x^{(m)}$ from the original data with distribution $p^*$
        • Update the discriminator by ascending its stochastic gradient :
          $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log(D(x^{(i)})) + \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))) \right].$$
    end for
    • Sample a minibatch of $m$ noise samples $z^{(1)}, \dots, z^{(m)}$ from noise prior $p_z$
    • Update the generator by descending its stochastic gradient :
      $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))).$$
end for
```

Generative Adversarial Networks

The generator never sees the training data, it is just updated via the gradients coming from the discriminator.
The next Figure presents a one-dimensional example to understand the successive steps in GANs training .

Generative Adversarial Networks

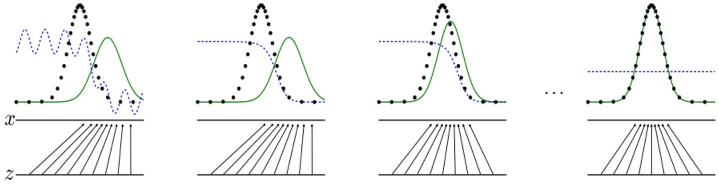


FIGURE – Successive steps in GANs training- In black, dotted line : the data generating distribution ; in green solid line : the generator distribution ; in blue, dashed line : the discriminator - z is sampled uniformly over the domain described by the lower lines- Source : cs.stanford.edu

- on the left, the initial values of the data distribution, the generator distribution and the discriminator.
- Then the discriminator has been trained , converging to $D^*(x)$.
- In the third picture, the generator has been updated, the gradient of D has guided $G(z)$ to high probability regions for the original data.
- On the right : after several training steps, p_g is very close to p^* and the discriminator does no more distinguish the true observations and those coming from the generator.

Generative Adversarial Networks

- Several extension of the original GANs have been proposed such as InfoGANs (Chen, 2016).
- The idea is to take as inputs of the generator not only a random noise but also interpretable latent variables learned from the data.
- This is done by maximizing the mutual information between a small subset of the GANs noise variables and the observations.
- It was shown that the procedure automatically discovers meaningful hidden representations in several image datasets.
- In order to discover the latent factors in an unsupervised way, the algorithm imposes a high mutual information between the generator $G(z, c)$ and the latent variables c.

Generative Adversarial Networks

- Let us recall that, in information theory, the mutual information between two random variables X and Y is a measure of the mutual dependence between the two variables.
- It represents the amount of information obtained about one variable through the other one. The mathematical formulation is

$$I(X, Y) = H(X) - H(X/Y) = H(Y) - H(Y/X)$$

where $H(X) = \int p_X \log(p_X)$ and $H(X/Y) = \int p_{X,Y} \log(p_{X,Y}/p_Y)$. If X and Y are independent, $I(X, Y) = 0$. In InfoGans, the original minimax game is replaced by

$$\min_G \max_D V_I(D, G)$$

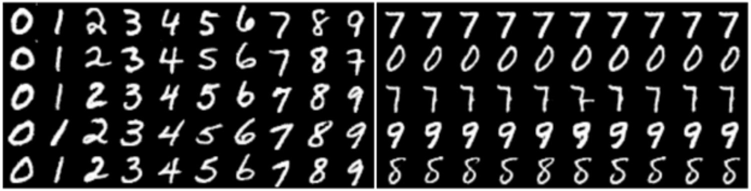
where

$$V_I(D, G) = V(D, G) - \lambda I(c, G(z, c)).$$

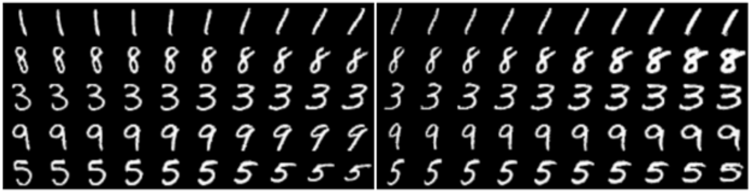
Generative Adversarial Networks

- Chen et al (2016) present an application to the MNIST data where they introduce a tree dimensional latent variable c :
  - the first component c_1 is a discrete random variable with 10 categories with equiprobability,
  - c_2, c_3 are two independent variables with uniform distribution on $[-1, 1]$.
- It appears that the latent variables are easily interpretable : the variable c_1 captures mostly the digit code, c_2 models rotation on digits and c_3 captures the width as shown in the next Figure.

Generative Adversarial Networks



(a) Varying c_1 on InfoGAN (Digit type) (b) Varying c_1 on regular GAN (No clear meaning)



(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation) (d) Varying c_3 from -2 to 2 on InfoGAN (Width)

FIGURE – InfoGAN- The different rows correspond to different random generated samples $G(z, c)$ with fixed latent variables (except one) and fixed random variables z. Ref. Chen et al. (2016)

Generative Adversarial Networks

The latent variables obtained from an encoder can also be considered as latent input variables for GANs.

The literature on GANs, and more generally on deep learning is very abundant, the methods evolve very quickly and it crucial to train yourself permanently on these topics.

Some references

- Ian Goodfellow, Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press
- Jeremy Jordan : Variational autoencoders, https ://www.jeremyjordan.me/
- Ali Ghodsi, Lec : Deep Learning, Variational Autoencoder, Oct 12 2017 [Lect 6.2]
- Deep learning course, Charles Ollion et Olivier Grisel https ://github.com/m2dsupsdlclass/lectures-labs