Deep Probabilistic Generative Models - Variational Auto-Encoders

M2 ANO BEHIDJ Ramzi
M2 AI JEMLI Nour

October 2021

Table des matières

1 Introduction

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional code and then reconstruct the output from this representation. The code is a compact "summary" or "compression" of the input, also called the latent-space representation.
A variational autoencoder (VAE) provides a probabilistic manner for describing an observation in latent space. Thus, rather than building an encoder which outputs a single value to describe each latent state attribute, we'll formulate our encoder to describe a probability distribution for each latent attribute.

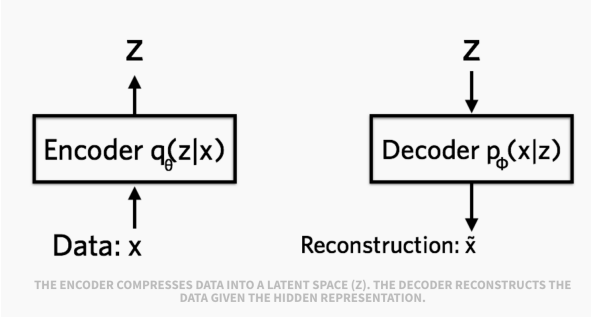


FIGURE 1 – VAE architecture

The latent variables, z, are drawn from a probability distribution depending on the input, X, and the reconstruction is chosen probabilistically from z.
The lab is divided in 3 parts, at firsl we will train a VAE with continious latent space and binary observed space then a VAE with binary latent space. The final part will consist to turn a deterministic Auto-Encoder into a generative model.

2 Variational AutoEncoders

2.1 Main idea

The probabilistic spin on autoencoders will let us sample from the model to generate data. So we might think about sampling from the true prior $p(z)$ first, then sampling from the conditional distribution $p(x|zi;\theta)$. In order to represent this distribution, it requires us to make assumptions about the probability distribution familly of the prior p(z). Usually, we choose p(z) to be Gaussian or Bernoulli. If we assume that z is a Gaussian vector, then the output of the encoder network would be a mean vector $\mu_{z|x}$ and a covariance matrix $\sum_{z|x}$. However, if we assume that z follows a Bernoulli distribution then the output of the encoder would only be the mean $\mu_{z|x}$.
To train this decoder network, we have to learn parameters $/theta$ to maximize the log-likelihood of training data as :

$$\begin{aligned}\widehat{\theta} &= \underset{x}{\operatorname{argmax}} \sum_d \log p(x^d;\theta) \\ &= \underset{x}{\operatorname{argmax}} \sum_d \log \int p(x^d;\theta)\,\mathrm{d}z \\ &= \underset{x}{\operatorname{argmax}} \sum_d \log \int p(x^d|z;\theta)p(z)\,\mathrm{d}z\end{aligned} \tag{1}$$

From the equation above, we can see that in order to compute the marginal distribution we have to compute the integral for z, for which we have to compute $p(x|z)$ for each z. However, in practice in order to express more information, we usually assume z to be a continuous variable (eg. Gaussian). But it is intractible for a computer to compute an infinite sum! Moreover, we do not have any information about the form of $p(x|z)$ since it is just a distribution parameterized by neural network.

2.2 Monte Carlo Sampling Method

A way we saw in class to solve this intracability is the Monte Carlo Sampling Method.It helps to replace the intergral term by the expecta- tion of latent variable z :

$$\int p(x^d|z;\theta)p(z)\,\mathrm{d}z = \mathbb{E}_{z\sim p(z)}[p(x^d|z;\theta)] \tag{2}$$

We can do this because according to the MC sampling method, expectation of a function f(z) can be considered as an average of multiple sampling for $z \sim p(z)$ i.e. $\mathbb{E}_{z\sim p(z)}[f(z)] = \frac{1}{k}\sum_{i=1}^{k} f(z_i)$ By applying this we have :

$$\int p(x^d|z;\theta)p(z)\,\mathrm{d}z = \frac{1}{k}\sum_{i=1}^{k} p(x^d|z_i) \tag{3}$$

2.3 ELBO

Now we can think about inference in this model. The goal is to infer good values of the latent variables given observed data, or to calculate the posterior $p(z|x;\theta)$. Using Bayes formula we have :

$$p(z|x;\theta) = \frac{p(x|z;\theta)p(z)}{p(x;\theta)} \tag{4}$$

We can calculate the denominator by marginalazing the latent variable.Unfortunately, this integral requires exponential time to compute as it needs to be evaluated over all configurations of latent variables. We therefore need to approximate this posterior distribution. To do this aproximation we will introduce a new distribution $q(z|x;\phi)$. We can use the Kullback-Leibler divergence, which measures the information lost when using q to approximate p.

$$\begin{aligned}KL[p(z|x;\theta)||q(z|x;\phi)] &= \mathbb{E}_{z\sim q(z|x;\phi)}[\log q(z|x;\phi) - \log p(x|z;\theta) - \log p(z)] + \log p(x;\theta) \\ KL[p(z|x;\theta)||q(z|x;\phi)] - \log p(x;\theta) &= \mathbb{E}_{z\sim q(z|x;\phi)}[\log q(z|x;\phi) - \log p(x|z;\theta) - \log p(z)]\end{aligned} \tag{5}$$

Our goal here is to minimize the KL-divergence which give us our objective function, we can rewrite it as follow :

$$\begin{aligned}\log p(x;\theta) - KL[p(z|x;\theta)||q(z|x;\phi)] &= \mathbb{E}_{z\sim q(z|x;\phi)}[\log p(x|z;\theta - (\log q(z|x;\phi - \log p(z))] \\ &= \mathbb{E}[\log p(x|z;\theta] - \mathbb{E}[\log q(z|x;\phi) - \log p(z)] \\ &= \mathbb{E}[\log p(x|z;\theta] - KL[q(z|x;\phi)||p(z)]\end{aligned} \tag{6}$$

Consequently, we can define the expression above as a variational lower bound of log-likelihood (ELBO), and the problem of maximizing log-likelihood can then be converted to maximizing the ELBO.

$$\varepsilon(x,\theta,\phi) = \mathbb{E}_{z\sim q(z|x;\phi)}[\log p(x|z;\theta] - KL[q(z|x;\phi)||p(z)] \tag{7}$$

At this point we have $q(z|x;\phi)$ that project the data X into the latent space, z the latent variable and $p(x|z;\theta)$ that generate the data given latent variable. In other terms we finnaly have the structure of our variationnal auto encoder, $q(z|x;\phi)$ is the encoder, z the encoded representation and $p(x|z;\theta)$ the decoder.
We define our maximization problem as follow :

$$\hat{\theta},\hat{\phi} = \underset{\theta,\phi}{\operatorname{argmax}} \sum_d \varepsilon(x^d,\theta,\phi) \tag{8}$$

2.4 Sigmoid bielief network

Sigmoid belief networks are described in full generality by Neal (1990), it's implementing a sigmoid function for each node. The main advantage of this kind of process is to remove the conditional dependencies of the underlying probability model. In SBN, the conditional distributions atach to each node are based on log linear models.
In particular we have according to [3] the probability that ith node is activated is given by

$$P\left(S_i = 1 \mid \operatorname{pa}\left(S_i\right)\right) = \sigma\left(\sum_j J_{ij}S_j + h_i\right) \tag{9}$$

where

$$P\left(S_i \mid S_1, S_2, \ldots, S_{i-1}\right) = P\left(S_i \mid \operatorname{pa}\left(S_i\right)\right) \tag{10}$$

Let $x, y \in \{0,1\}^n$
We have $y \sim p(y;\mu)$ where $\mu = \sigma(a) = \frac{\exp(a)}{1+\exp(a)}$ and $x \sim p(x|y;\lambda)$ where $\lambda = \sigma(\beta(y) + c)$

$$\log p(x) = \log \sum_y p(y)p(x|y) \tag{11}$$

Unlike VAE, here we have a sum of a finite number of PMFs since they are all Bernoulli variables. Nevertheless, as the size of latent variables gets larger the gradient ascent would be intractable (sum over 2^n values). The main difference between VAE and SBN concern the probability familly of the variables, VAE assumes that all the variables follows the same distribution which is not the case for the SBN where its gives a distribution for each given input variable.

3 VAE with continuous latent space and binary observed space

In this part, we assume that the latent random variable Z takes value in $\mathbb{R}^n$. The prior distribution p(z) is a multivariate Gaussian where each coordinate is independent. We x the mean and variance of each coordinate to 0 and 1, respectively. The conditional distribution $p(x|z;\theta)$ is parameterized by a neural network. The random variables X are m independent Bernoulli random variables.

3.1 KL divergence

In order to sample the prior p(z) later, we want to make the posterior $q(z|x;\phi)$ to be close to the gaussian distribution $\mathcal{N}(0, I)$. We also make the assumption that the prior is gaussian. Let μ and σ be the variational mean and s.d given data points X. Lets compute the KL-divergence term of the ELBO : From Apendix B of [2]

$$\begin{aligned} -KL[q(z|x;\phi)||p(z)] &= \int q(z|x;\theta)(\log(p(x|z;\theta) - q(z|x;\theta))\,\mathrm{d}z \\ &= \frac{1}{2}\sum_{j=1}^{j}(1 + \log((\sigma_j)^2) - (\mu_j)^2 - (\sigma_j)^2) \end{aligned} \tag{12}$$

3.2 Architecture and training loop

3.2.1 GaussianEncoder

Our neural networks are constructed in the following way. From input x to the hidden layer, we have a full linear connection and a non-linear relu connection :

```
def __init__(self, dim_input, dim_latent):
        super().__init__()

        self.hidden = nn.Linear(input_dim, hidden_dim)
        self.output_mean = nn.Linear(hidden_dim, output_dim)
        self.output_var = nn.Linear(hidden_dim, output_dim)

def forward(self, batch):
        z = F.relu(self.hidden(batch))
        mu = self.output_mean(z)
        log_sigma_squared = self.output_var(z)

        return mu, log_sigma_squared
```

3.2.2 GaussianPriorDecoder

We have a full linear connection with a relu to the hidden later, then a full linear connection to the output :

```
def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.input_dim = input_dim
        self.hidden = nn.Linear(input_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, output_dim)

def forward(self, batch):
        z = F.relu(self.hidden(batch))
        return self.output(z)
```

3.2.3 Training loop

The encoder outputs 2 arrays of shape (batch_size, dim_latent), one for means and for variances, and the decoder takes as input a single array of shape (batch_size, dim_latent), computed from a « reparameterization trick ». Indeed, reparamtrization trick is a way to rewrite the expectation so that the distribution with respect to which we take the gradient is independent of parameter θ. To do this, we have to make the stochastic element in q independent of θ. we sample from another random variable $\epsilon \sim \mathcal{N}(0, I)$ each time. Finally, we can combine the output of the encoder with sampling ϵ to represent z as :

$$z = \epsilon \times \sigma_{z|x} + \mu_{z|x}$$

4 VAE with binary latent space and binary observed space

In this part, we assume that the prior distribution p(z) is a multivariate Bernoulli where each coordinate is independent, the mean of each coordinate is fixed to 0.5, so we have $z_i \sim \mathcal{B}(0.5)$ The latent random variable Z takes value in $[0\,;\,1]$ which is sampled from the corresponding Bernoulli coordinate. The conditional distribution $p(x|z;\theta)$ is parameterized by a neural network. The random variables X are m independent Bernoulli random variables.

4.1 KL divergence

According to the definition of a bernoulli distribution and the assumption of the independenty of the outcomes we have :

$$p(z) = \prod_{i=1}^{k} p_i(z_i)$$

Since we already have the definition of the KL-divergence between two distributions we can compute the following KL-divergence betwen two Bernoulli distributions from [1] :

$$\begin{aligned} D_{KL}(p||q) = \sum_z \prod_{i=1}^{k} p_i(z_i) \log \prod_{j=1}^{k} \frac{p_j(z_j)}{q_j(z_j)} &= \sum_z \prod_{i=1}^{k} p_i(z_i) \sum_{j=1}^{k} \log \frac{p_j(z_j)}{q_j(z_j)} \\ &= \sum_{j=1}^{k} \sum_z \prod_{i=1}^{k} p_i(z_i) \log \frac{p_j(z_j)}{q_j(z_j)} \end{aligned} \tag{13}$$

We can write the KL-divergence as :

$$\begin{aligned} D_{KL}(p||q) &= \sum_{j=1}^{k} \sum_{z_j} p_j(z_j) \log \frac{p_j(z_j)}{q_j(z_j)} \\ D_{KL}(p||q) &= \sum_{j=1}^{k} p_j \log \frac{p_j}{q_j} + (1-p_j) \log \frac{1-p_j}{1-q_j} \end{aligned} \tag{14}$$

By using this formula we can deduce the KL-divergence that we will implement in our Bernoulli VAE as :

$$\begin{aligned} D_{KL}(q_\phi(z)||p_\theta(z)) &= \sum_j \mu_j \log \frac{\mu_j}{0.5} + (1-\mu_j) \log \frac{1-\mu_j}{1-0.5} \\ &= \sum_j [\mu_j \log \mu_j + (1-\mu_j) \log(1-\mu_j) + \log 2] \end{aligned} \tag{15}$$

4.2 Architecture and training loop

4.2.1 BernoulliEncoder and Decoder

The neural architecture of the barnoulli VAE is very close to the gaussian one, we have a full linear conection and a non linear conection and one output from the hidden layer. The difference with the gaussian VAE is the number of outputs, here we have only one.
The decoder in the other hand is exactly same, we have a full linear connection with a relu to the hidden later, then a full linear connection to the output.

4.2.2 Training loop

The main architecture of the training loop of the bernoulli encoder is close to the gaussian VAE which was already implemented in the notebook. The main difference is that we can no longer apply the reparametrization trick. So in order to have differentiable sampling, we will use another tool called the Score Function Estimator. It works when doing back-propagation for the encoder network :

$$\nabla_\phi \varepsilon(x, \theta, \Phi) = \nabla_\Phi \, \mathbb{E}_{z \sim q(z|x;\Phi)}[\log p(x|z;\theta)] - \nabla_\Phi D_{KL}(q(z|x;\Phi)||p(z)) \tag{16}$$

If we developp the first term we have :

$$\begin{aligned}
\nabla_\Phi \mathbb{E}_{q(z|x;\Phi)}[\log p(x \mid z;\theta)] &= \nabla_\Phi q(z \mid x;\Phi) \log p(x \mid z;\theta) \\
&= \log p(x \mid z;\theta) \nabla_\Phi q(z \mid x;\Phi) \\
&= \log p(x \mid z;\theta) \frac{q(z \mid x;\Phi)}{q(z \mid x;\Phi)} \nabla_\Phi q(z \mid x;\Phi) \\
&= \mathbb{E}_{q(z|x;\Phi)} \left[\log p(x \mid z;\theta) \frac{1}{q(z \mid x;\Phi)} \nabla_\Phi q(z \mid x;\Phi) \right] \\
&= \mathbb{E}_{q(z|x;\Phi)} \left[\log p(x \mid z;\theta) \nabla_\Phi \log q(z \mid x;\Phi) \right]
\end{aligned} \tag{17}$$

We can use Monte Carlo sampling to approximate the expectation term :

$$\begin{aligned}
&\mathbb{E}_{q(z|x;\Phi)} \left[\log p(x \mid z;\theta) \nabla_\Phi \log q(z \mid x;\Phi) \right] \\
&\simeq \frac{1}{K} \sum_{i=1}^{K} \log p\left(x \mid z^{(i)};\theta\right) \cdot \nabla_\Phi \log q(z \mid x;\Phi)
\end{aligned} \tag{18}$$

We can see that the score function is the derivative of the log of probabaility distribution $q(z|x;\Phi)$. In the implemnation process we used the pytorch function .detach() to say no for the backpropagation. The above SFE gives us an unbiased but high variance, we are gonna use a baseline to stabilize the variance of the estimator. Here, we will rely on the average reconstruction value from all previous update and that's what we call the running average trick.

5 Experiments

In the following section we will present our results about our data generation using variational autoencoders by comparing differents results of the Gaussian and Bernoulli VAE to see which one is more effecient for data generation.
Using the approximate posterior, we can visualize the latent space, by assigning one color per digit. The mean value is displayes.For a well trained model,we should see all points close to 0 and each class should be well delimited, i.e. there must be clusters of the same color.

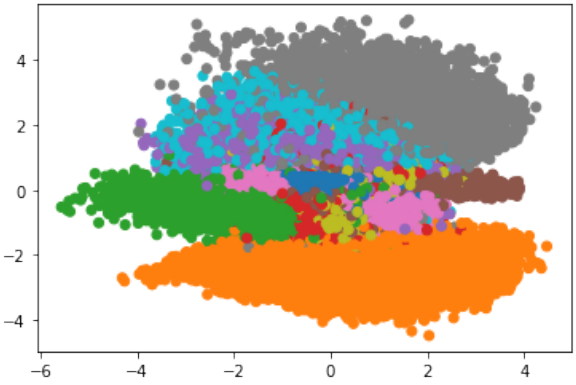
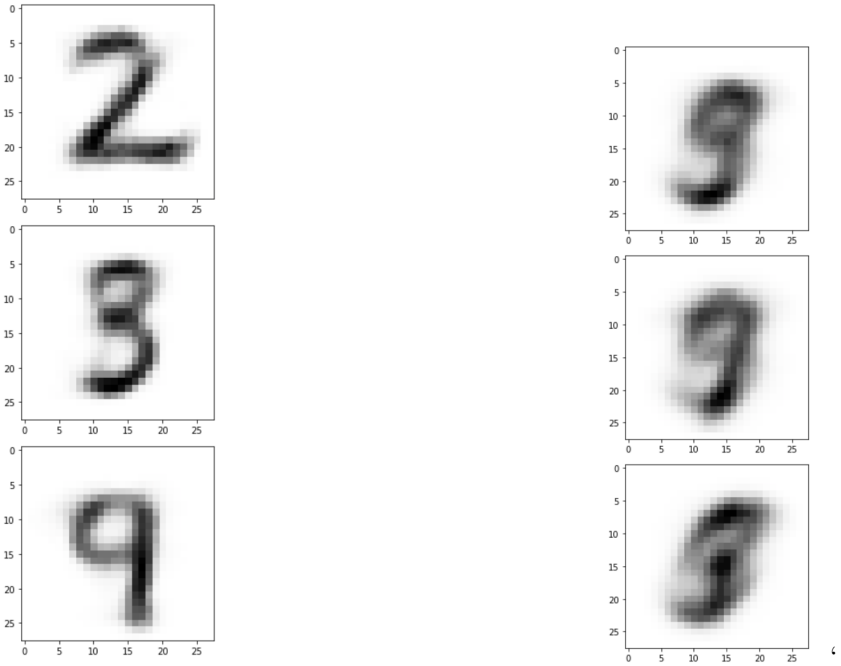


FIGURE 2 – Latent Space visiualisation

From the result we can see that almost all the points are close to 0 and there are several clusters of the same color, which means some of the classes are well delimited, but there also exist some classes with mean values really close to 0 are mixed together.

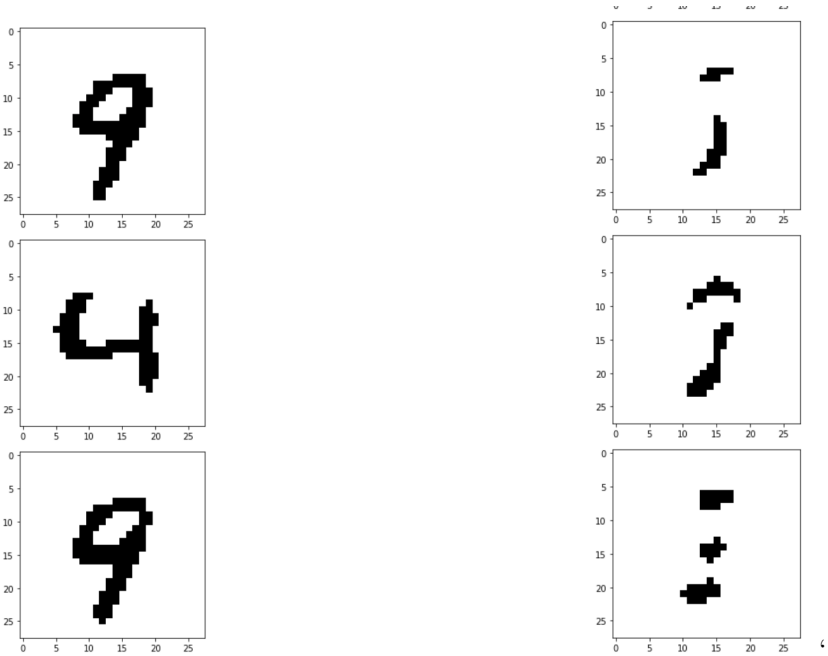
The second experiment will compare the two VAEs for their distribution generation and then the image generated with the most probable value. We generate some new distributions of random variable X by applying the trained decoders on some new samples from the latent space. The new images are then generated by sampling from the output distribution or directly using the most probable value for each random variable. In this work, we display the output image with the most probable value.



(a) Gaussian VAE (b) Bernoulli VAE

FIGURE 3 – Distribution generated by the VAEs

We notice that the Gaussian VAE generate more precise distribution, we can destinguish more easily the digits. On the other hand, the bernoulli model generate a pretty bad distributions samples.



(a) Gaussian VAE (b) Bernoulli VAE

FIGURE 4 – Images generated by the VAEs

After taking the most probable value of the distribution, we can see the gaussian VAE is more efficient, we clearly see that's more easy to identify the MISNT data generated by the gaussian VAE then the Bernoulli one where the distribution are very blurred and obviously non visible image digits generated.

6 Turning a Deterministic Auto-Encoder into a generative model

With our deterministic Auto-Encoder, an image is projected in a 2 dimension latent space. In other words, each image with 28×28 pixels is represented by a point in the $\mathbb{R}^2$ space,it's possible to reduce the information contained on the pixels of this image to two parameters, for example. In this case, the latent distribution is of order 2. Thus, after training your hidden layers to encode and decode very well for a given dataset one may say that the for every pair of this R space of latent variables if you forward it through the decoder the output will be an image, in the case of the mnist dataset, a new and possibly never written handwritten digit.
To turn our model into a deterministic auto-encoder generative model, the KL divergence is removed from the previous form of the algorithm. After this, a Gaussian Mixture Model (GMM) is used to clusterize the output of the decoder and thus each of the Gaussians of this model shall fit to one region of points and possibly to a single digit on the reconstruction dimension.

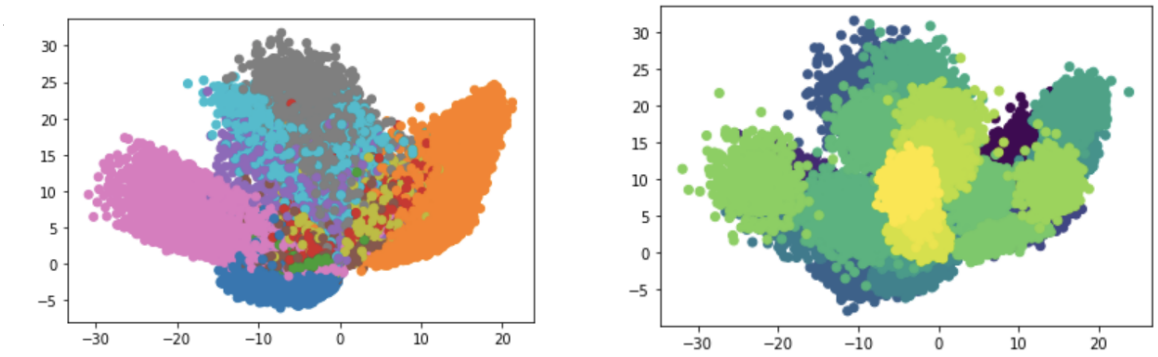


FIGURE 5 – 2d points in latent space / GMM model

As we can see from the plot of latent space, the 2d points also form several clusters. And the resulting points cloud of GMM looks very similar to the 2d points cloud, which means our GMM model can well represent the latent space. This GMM model can then be used to generate new points.

Références

[1] Diego GOMEZ. *KL divergence between two multivariate Bernoulli distribution*. 2019. URL : `https://math.stackexchange.com/questions/2604566/kl-divergence-between-two-multivariate-bernoulli-distribution` (visité le 06/10/2021).

[2] Diederik P KINGMA et Max WELLING. "Auto-encoding variational bayes". In : *arXiv preprint arXiv :1312.6114* (2013).

[3] Lawrence K SAUL, Tommi JAAKKOLA et Michael I JORDAN. "Mean field theory for sigmoid belief networks". In : *Journal of artificial intelligence research* 4 (1996), p. 61-76.