Auto Encodeur variationnel pour la génération d'électrocardiogrammes

April 8, 2023

Antonin Lefevre
antoninlefevre45@icloud.com

Auto-Encodeur

Les auto-encodeurs (AE) sont des réseaux de neurones, entraînés de manière non supervisée, dont le but est de reconstruire les données d'entrées avec un maximum de précision en ayant codé ces dernières. Ils sont composés de deux parties, un encodeur et un décodeur. L'encodeur transforme les données dans un espace latent et le décodeur reconstruit les données à partir de cet espace latent. Dans le cas où cet espace latent est de dimension inférieure aux données, on parle d'architecture "sous-complète" (undercomplete). Quand la dimension de l'espace latent est supérieure à celle des données, elle est appelée "sur-complète" (overcomplete). La suite de cet article ne fera référence qu'aux architectures sous-complètes, c'est à dire qui compressent les données dans l'espace latent de plus petite dimension.

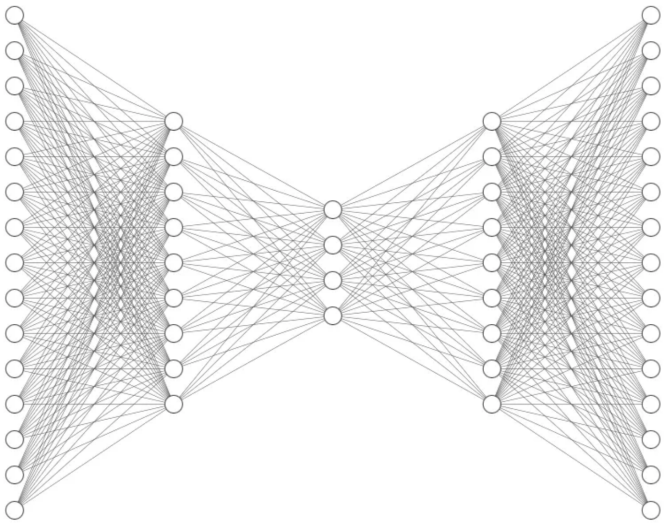


Figure 1: Architecture classique d'auto encodeur (undercomplete) utilisant des couches denses.

Fonction de perte des Auto-Encodeurs

L'objectif de l'entraînement des AE est de minimiser l'écart entre l'entrée et sa version reconstruite. Ainsi, on peut exprimer la perte globale du réseau comme étant la perte moyenne par échantillon, ainsi:

$$L_{\text{total}} = \frac{1}{m} \sum_{\text{i}=1}^{m} l(x^{(\text{i})}, \hat{x}^{(\text{i})})$$

Si l'entrée est catégorielle alors on utilise la cross-entropy pour la perte de chaque échantillon, donc:

$$l(x^{(\text{i})}, \hat{x}^{(\text{i})}) = -\sum_{\text{j}=1}^{n} [x_i \log(\widehat{x_i}) + (1 - x_i) \log(1 - \widehat{x_i})]$$

Sinon, nous utiliserons la Mean Squared Error (MSE), donc:

$$l(x^{(\text{i})}, \hat{x}^{(\text{i})}) = \frac{1}{2} \|x - \hat{x}\|^2$$

Auto-encodeur variationnel

Les auto-encodeurs variationnels (VAE) sont des modèles génératifs. Ils sont similaires aux auto-encodeurs classiques (AE) en terme d'architecture, mais leurs formulations sont très différentes.

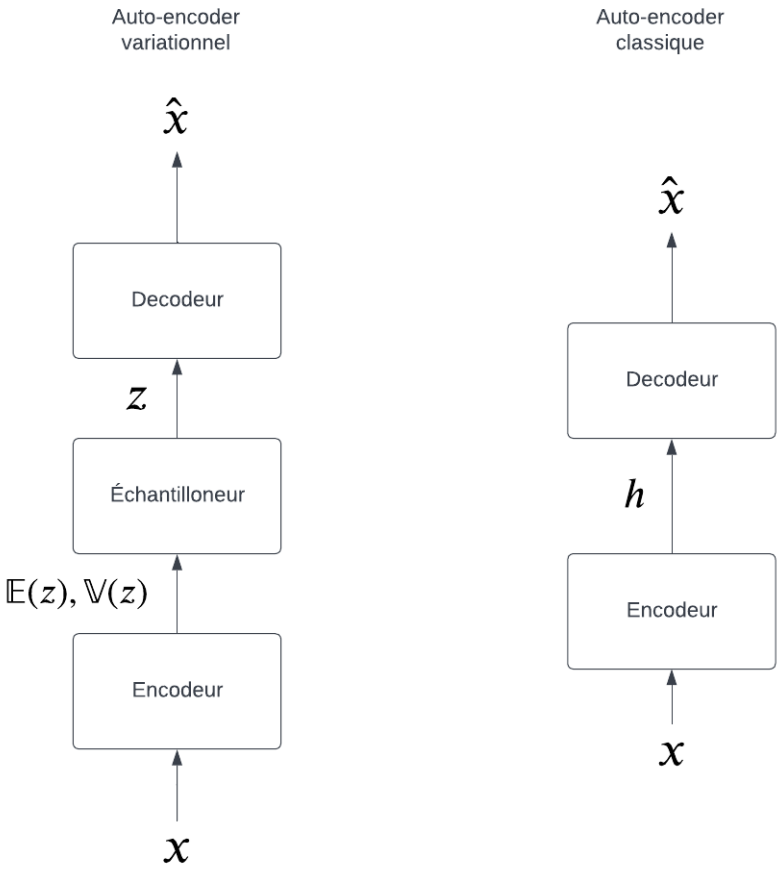


Figure 2: Comparaison VAE et AE.

Concernant l'encodeur, après avoir passé le vecteur d'entrée x à l'encodeur, l'auto-encodeur classique va générer un vecteur latent h alors que l'encodeur du VAE va générer $E(z)$ et $V(z)$, où z est la variable aléatoire suivant une loi gaussienne de moyenne $E(z)$ et de variance $V(z)$. Ensuite, par échantillonnage de $E(z)$ et $V(z)$, le VAE génère le vecteur latent z. Enfin, le vecteur latent est passé au décodeur pour générer une nouvelle donnée qui correspond à la reconstruction de l'entrée x.

Ce processus permet au VAE d'avoir un espace latent performant pour la génération.

Fonction de perte des Auto-Encodeurs variationnels

Comme tout réseau de neurones classique, on va chercher à minimiser une fonction de perte. Celle-ci est composée d'un terme de reconstruction et d'un terme de régularisation, tel que:

$$l(x, \hat{x}) = l_{\text{reconstruction}} + \beta l_{\text{KL}}(z, \text{N}(0, I_d))$$

Pour mieux comprendre l'utilité de chacun des termes, on peut visualiser chaque valeur estimée de z comme un cercle dans un espace 2D où le centre du cercle est $E(z)$ et le rayon est $V(z)$.

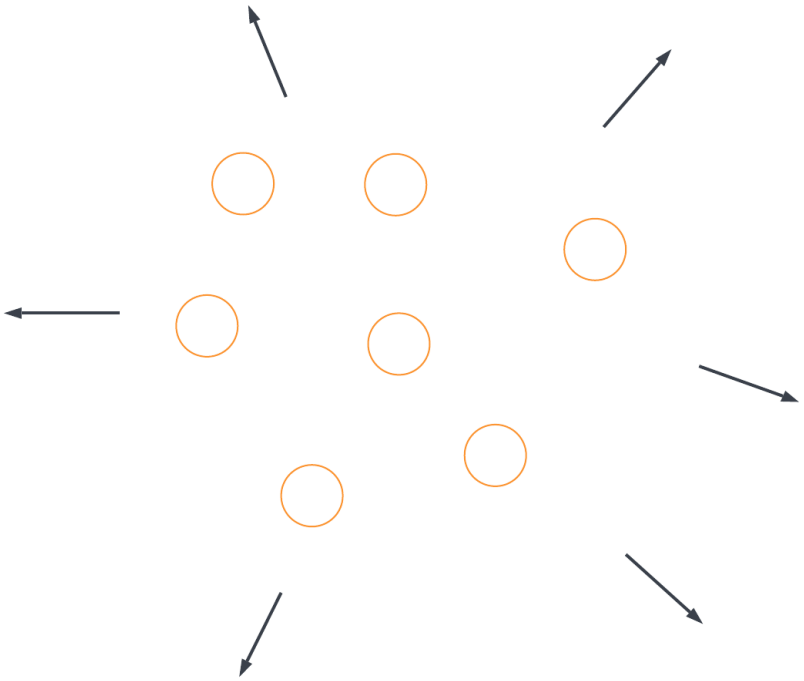


Figure 3: Visualisation de z sous forme de bulles dans l'espace latent.

Ainsi, chaque bulle représente une région estimée à z et les flèches représentent l'action du terme de reconstruction qui tend à éloigner chaque valeur estimée des autres pour éviter les chevauchements. En effet, s'il y a un chevauchement entre deux estimations (donc entre deux bulles), cela crée une ambiguïté pour la reconstruction car les points de chevauchement peuvent être mis en correspondance avec les deux entrées originales. De ce fait, le terme de reconstruction va éloigner les points. L'expression du terme de reconstruction a été définie précedemment et dépend du type des entrées. Cependant, si la fonction de perte ne se résumait qu'au terme de reconstruction, les points continueraient de s'éloigner et le système pourrait exploser. C'est pourquoi on ajoute le terme de régularisation.

Ce terme de régularisation correspond à la divergence KL (Kullback-Leibler divergence) qui mesure la distance entre la distribution latente z qui provient d'une gaussienne de moyenne $E(z)$ et de variance $V(z)$ et la distribution normale standard. Le but est de faire en sorte que la distribution z reste proche de la distribution normale, et ainsi de minimiser cette distance.

On peut écrire ce terme de régularisation comme ceci:

$$\beta l_{\mathrm{KL}}(z, \mathrm{N}(0, I_d)) = \frac{\beta}{2} \sum_{\mathrm{i}=1}^{d} \left(\mathbb{V}(z_i) - \log[\mathbb{V}(z_i)] - 1 + \mathbb{E}(z_i)^2 \right)$$

Prennons les 3 premiers termes de la somme, qui représentent une fonction en $V(z_i)$. On note ainsi:

$$v_i = \mathbb{V}(z_i) - \log[V(z_i)] - 1$$

Dont la courbe représentative est la suivante:

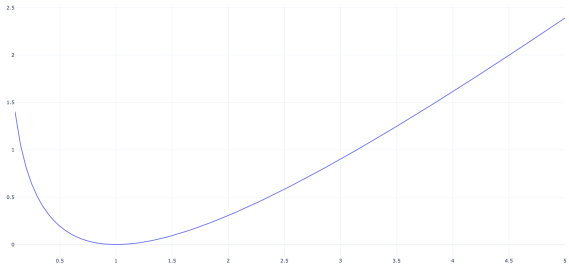


Figure 4: Fonction v_i.

On remarque que v_i est minimale quand $V(z_i)$ vaut 1. C'est à dire quand chaque valeur estimée (ou chaque point) a une variance de 1 (ou un rayon de 1). Le dernier terme de la somme $E(z_i)^2$ minimise la distance entre les valeurs estimées z_i et empêche donc l'explosion du système que pourrait causer le terme de reconstruction.

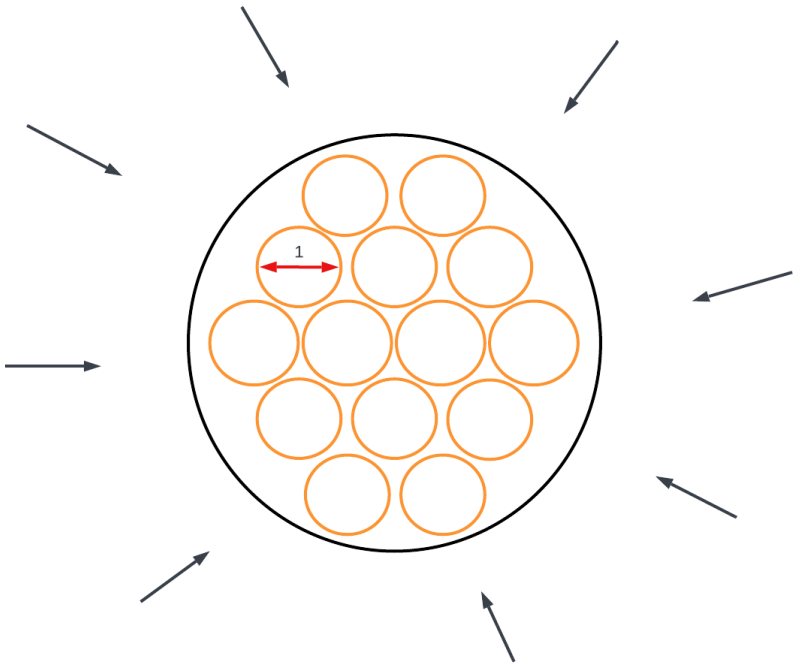


Figure 5: "bulle-de-bulles" du VAE.

On parle ainsi de l'interprétation "bulle-de-bulles" du VAE [2]. Le terme de reconstruction évite le chevauchement des valeurs estimées z_i en les éloignant, et le terme de régularisation (ie. la divergence KL) permet de maintenir la variance de chaque point autour de 1 et de contrer l'éloignement du terme de reconstruction pour éviter une explosion. Le scalaire β est un hyperparamètre qui pondère les termes dans la fonction de perte.

Reparamétrisation de l'échantillonnage

L'échantillonnage pose un problème pour la rétropropagation, de ce fait on utilise une astuce pour échantilloner z. On utilisera ainsi:

$$z = \mathbb{E}(z) + \mathbb{V}(z).\,\varepsilon$$

Avec

$$\varepsilon \rightsquigarrow N(0, I_d)$$

où I_d est l'identité dans $\mathbb{R}^d$ qui contient z et « . »" est le produit scalaire. La rétropopagation est de ce fait rendue possible.

Implémentation d'un VAE pour la génération d'électrocardiogrammes

Électrocardiogramme

L'ECG est un tracé obtenu par l'enregistrement et la transcription des courants électriques qui parcourent le coeur au cours de chaque contraction cardiaque. Il peut être schématisé comme dans la Figure 6.

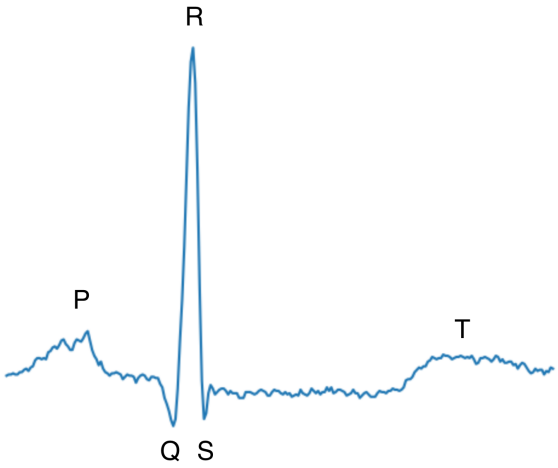


Figure 6: "bulle-de-bulles" du VAE.

Il est composé des ondes **P** et **T** ainsi que du complexe **QRS**.

Le modèle

Cet article propose une architecture d'un VAE inspiré de l'article Electrocardiogram Generation and Feature Extraction Using a Variational Autoencoder [1]. Nos données (ECG) ont une taille de 250. L'architecture de l'encodeur est définie dans la Figure 7.

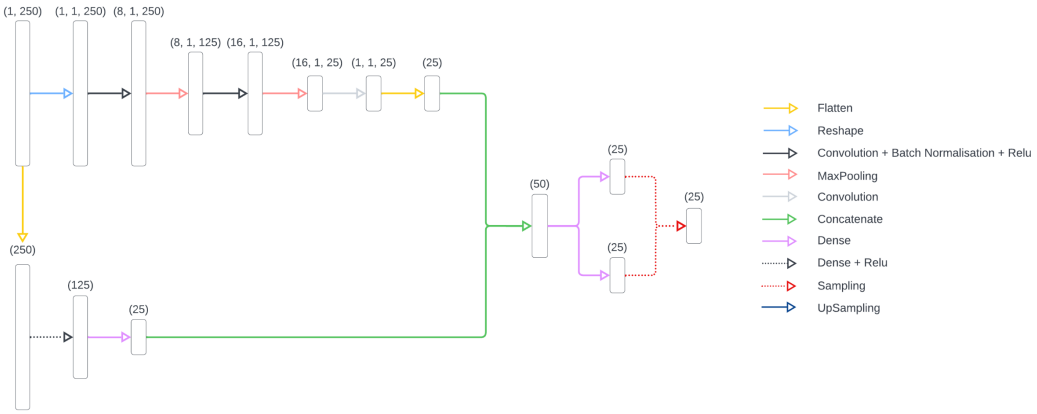


Figure 7: Architecture de l'encodeur.

L'encodeur est composé d'une partie convolutionnelle et d'une partie dense. La partie convolutionnelle consiste en une succession de convolutions, chacune suivie d'une normalisation par lots (batch normalization) et d'une fonction d'activation ReLU, puis d'un MaxPooling. Cette partie se termine par une convolution simple, suivie d'un Flatten pour obtenir finalement un vecteur de taille 25. Quant à la partie Dense, elle est composée d'une couche dense avec une fonction d'activation ReLU et d'une couche dense simple, pour obtenir un vecteur de taille 25.

Ces deux vecteurs de taille 25 sont ensuite concaténés en un vecteur de taille 50. Deux couches denses simples sont ensuite appliquées en parallèle pour obtenir deux vecteurs de taille 25. En réalité, ces deux vecteurs correspondent aux vecteurs moyenne (μ) et écart-type (σ) qui nous permettent finalement d'échantillonner le dernier vecteur de l'encodeur, qui est le vecteur latent.

L'architecture du décodeur est explicitée dans la Figure 8.

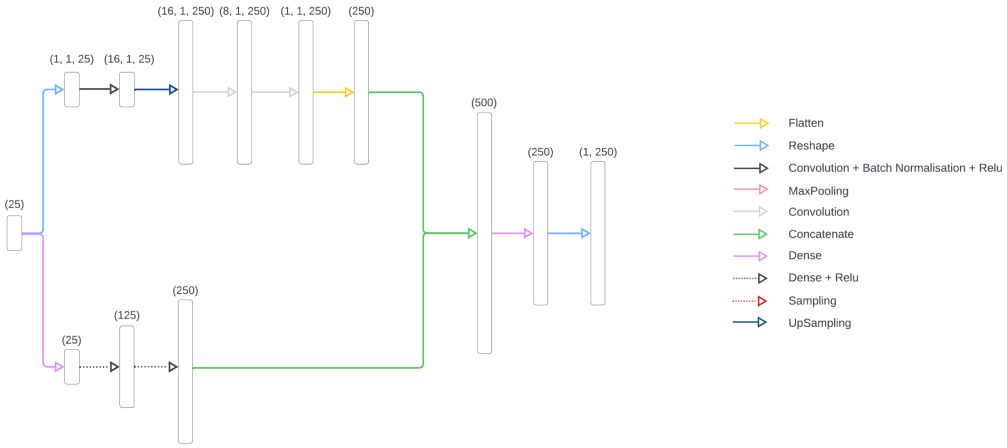


Figure 8: Architecture du décodeur.

Comme l'encodeur, le décodeur est composé d'une partie convolutionnelle et d'une partie dense. Le vecteur d'entrée est donc le vecteur latent créé par l'encodeur. La partie convolutionnelle commence par une convolution avec normalisation par lots (batch normalization) et ReLU. Ensuite, un upsampling est effectué pour revenir à une taille de 250, suivi de deux convolutions simples et d'un Flatten pour obtenir un vecteur de taille 250. La partie Dense est composée de deux couches denses avec ReLU.

Les deux vecteurs résultants de taille 250 sont ensuite concaténés en un vecteur de taille 500. Finalement, une couche dense simple permet de revenir à un vecteur de taille 250.

Implémentation avec PyTorch

L'architecture précédemment décrite est implémentée avec PyTorch. Les données des ECG sont des signaux de taille 250, et l'espace latent est choisi comme ayant une taille de 25. Ci-dessous l'encodeur.

```
class Encoder(nn.Module):

    def __init__(self):
        super(Encoder, self).__init__()
        # Couches convolutives
        self.conv1 = nn.Conv2d(1, 8, stride=1, padding=2, kernel_size=5)
        self.convbatchNorm1 = nn.BatchNorm2d(8)
        self.maxPool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(8, 16, stride=1, padding=2, kernel_size=5)
        self.convbatchNorm2 = nn.BatchNorm2d(16)
        self.maxPool2 = nn.MaxPool2d(kernel_size=7, stride=5, padding=3)
        self.conv3 = nn.Conv2d(16, 1, kernel_size=5, stride=1, padding=2)
        # Couches denses
        self.dense1 = nn.Linear(250, 125)
        self.dense2 = nn.Linear(125, 25)
        # Couches concatenation
        self.concatDense1 = nn.Linear(50, 25)
        self.concatDense2 = nn.Linear(50, 25)
        # tools
        self.N = torch.distributions.Normal(0, 1)  # standard normal distribution
        self.kl = 0  # KL divergence
    def forward(self, x):
        # Convolution
        x_conv = x.unsqueeze(0).unsqueeze(0)
        x_conv = F.relu(self.convbatchNorm1(self.conv1(x_conv)))
        x_conv = self.maxPool1(x_conv)
        x_conv = F.relu(self.convbatchNorm2(self.conv2(x_conv)))
        x_conv = self.maxPool2(x_conv)
        x_conv = self.conv3(x_conv)
        x_conv = x_conv.view(25)
        # Dense
        x = x.flatten()  # pour avoir shape (250)
        x_dense = F.relu(self.dense1(x))
        x_dense = self.dense2(x_dense)
        # Concatenation
        x_concat = torch.cat((x_conv, x_dense), 0)
        mu = self.concatDense1(x_concat)  # moyenne
        var = torch.exp(self.concatDense2(x_concat))  # variance
        # Vecteur latent
        z = mu + var * self.N.sample(mu.shape)
        self.kl = 0.5 * torch.sum(var - torch.log(var) - 1 + mu ** 2)  # KL
        return z
```

La reparamétrisation est définie par:

```
z=mu+var*self.N.sample(mu.shape)
```

Et la divergence KL est définie par:

```
0.5*torch.sum(var-torch.log(var)-1+mu**2)
```

Ce qui correspond bien à ce que l'on a expliqué plus haut, avec:

$$\beta l_{\mathrm{KL}}(z, \mathrm{N}(0, I_d)) = \frac{\beta}{2} \sum_{\mathrm{i}=1}^{d} \left(\mathbb{V}(z_i) - \log[\mathbb{V}(z_i)] - 1 + \mathbb{E}(z_i)^2 \right)$$

L'encodeur renvoie donc le vecteur latent de taille 25 échantillonné avec les vecteurs $\mathbb{E}$ et $\mathbb{V}$ et le décodeur prend en entrée ce vecteur. Le décodeur est défini comme:

```
class Decoder(nn.Module):

    def __init__(self):
        super(Decoder, self).__init__()
        # Couches convolutives
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2)
        self.convbatchNorm1 = nn.BatchNorm2d(16)
        self.upsampling1 = nn.Upsample(size=(1, 250))  # (1, 16, 1, 250)
        self.ConvUp1 = nn.ConvTranspose2d(16, 8, kernel_size=1, stride=1)
        self.ConvUp2 = nn.ConvTranspose2d(8, 1, kernel_size=1, stride=1)
        # Couches denses
        self.dense1 = nn.Linear(25, 25)
        self.dense2 = nn.Linear(25, 125)
        self.dense3 = nn.Linear(125, 250)
        # concaténation
        self.concatDense1 = nn.Linear(500, 250)

    def forward(self, z):
        # Couches convolutives
        z_conv = z.reshape((1, 25)).unsqueeze(0).unsqueeze(0)
        z_conv = F.relu(self.convbatchNorm1(self.conv1(z_conv)))
        z_conv = self.upsampling1(z_conv)
        z_conv = self.ConvUp1(z_conv)
        z_conv = self.ConvUp2(z_conv)
        z_conv = z_conv.flatten()
        # Couches denses
        z_dense = self.dense1(z)
        z_dense = F.relu(self.dense2(z_dense))
        z_dense = F.relu(self.dense3(z_dense))
        z_dense = z_dense.flatten()
        # Concaténation
        z_dense = torch.cat((z_conv, z_dense), 0)
        # Dernières couches denses
        z_final = self.concatDense1(z_dense)
        z_final = z_final.reshape(1, 250)
        return z_final
```

Ne reste plus qu'à assembler les deux parties dans l'auto encodeur:

```
class VariationalAutoencoder(nn.Module):
    def __init__(self):
        super(VariationalAutoencoder, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        z = self.encoder(x)
        return self.decoder(z)
```

Pour l'entraînement:

```
def train(vae, data_dataloader, epochs):
    loss_evolution = []
    opt = torch.optim.Adam(autoencoder.parameters())
    for _ in tqdm(range(epochs)):
        for i, (x, y) in enumerate(data_dataloader):
            opt.zero_grad()
            x_hat = vae(x)
            # Loss = MSE + divergence KL
            loss = ((x - x_hat) ** 2).sum() + autoencoder.encoder.kl
            if i == (len(x) - 1):
                # on ne veut que le dernier loss de chaque epoch
                loss_evolution.append(loss.detach().numpy())
            loss.backward()
            opt.step()
    return vae, loss_evolution
```

La fonction de perte utilisée ici comprend le terme de reconstruction qui est la MSE (car nos données sont continues) et le terme de régularisation correspondant à la divergence de Kullback-Leibler.

Génération de nouveaux ECG

La génération se fait en utilisant uniquement le décodeur du VAE une fois entrainé. Il suffit ainsi d'échantillonner un vecteur latent de taille 25 et de le donner au décodeur qui va alors construire un nouvel ECG. La distribution normale centrée réduite est contenue dans l'attribut N de l'encodeur, que l'on échantillonne avec la méthode **sample**.

```
generated_ECG = []
for i in range(n):
    latent_vector_shape = (1, 25)
    new_latent_ECG = vae_trained.encoder.N.sample(latent_vector_shape)
    generated_ECG.append(vae_trained.decoder(new_latent_ECG))
```

Ci-dessous, le génération de 30 ECG.

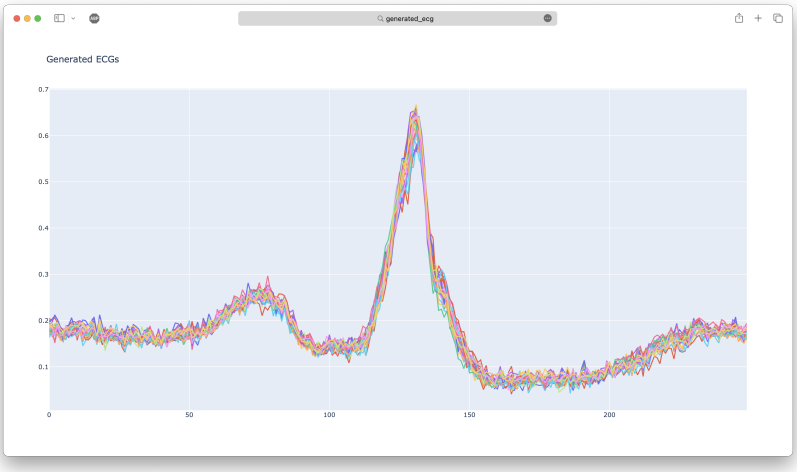


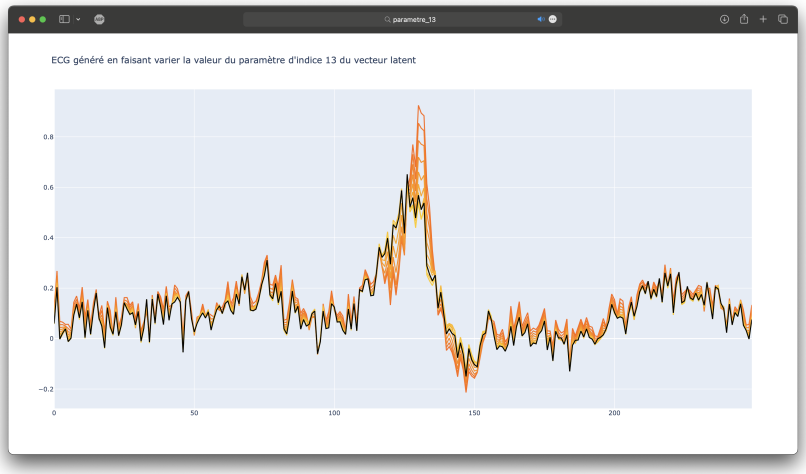
Figure 9: Superposition 30 ECG générés.

Les ondes P et T sont très reconnaissables et le complexe **QRS** est reconstruit de manière satisfaisante.

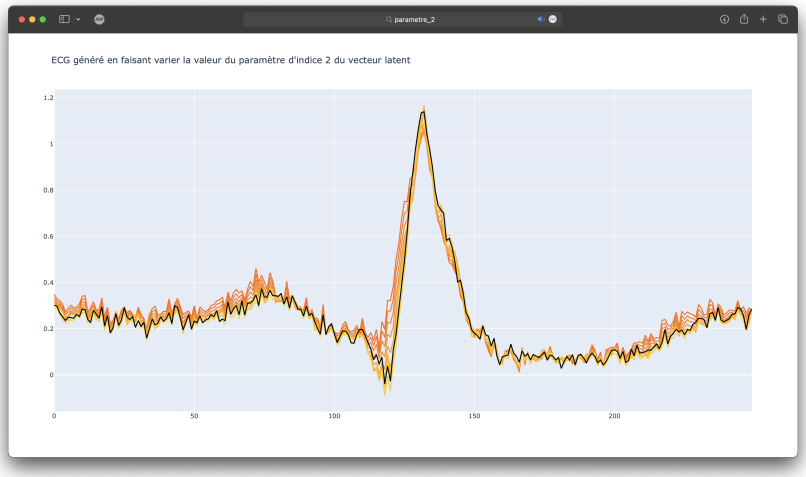
Interprétation des valeurs du vecteur latent

Chacun des paramètres du vecteur latent z permet théoriquement de coder une caractéristique de l'ECG. Ainsi, leur rôle pourrait être révélé en faisant varier chacun d'entre eux indépendemment des autres. Dans le modèle présenté dans cet article, le vecteur latent z est composé de 25 valeurs. Faisons varier chacune d'elles et tentons de découvrire leur rôle. Pour chacune des 25 valeurs, on représentera l'ECG reconstruit avec 10 valeurs possibles de la valeur sur une même figure où la courbe noire est le vecteur latent original, et les autres sont colorées du jaune au orange qui représentent la plage de valeurs prises par le paramètre.

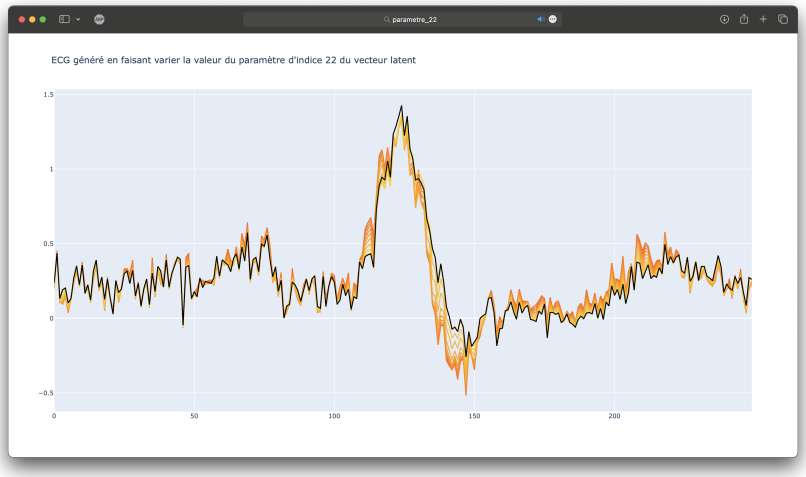
La valeur d'indice 13 du vecteur z semble coder pour la hauteur du pic R du complexe QRS.



La valeur d'indice 2 du vecteur z semble être responsable de la hauteur du pic Q du complexe QRS.



Enfin, la valeur d'indice 22 de z semble coder pour la hauteur du pic S du complexe QRS.



Références

[1] Electrocardiogram Generation and Feature Extraction Using a Variational Autoencoder V. V. Kuznetsov, V. A. Moskalenko, N. Yu. Zolotykh

[2] NYU Spring 2020 courses from Yann LeCun Y. LeCun, A. Canziani