Detecting anomalies in medical data using Generative Adversarial Networks

Justin Glibert

April 24, 2019

Contents

Abstract

The research group led by Prof Justin Dauwels here at NTU is currently working on classifying electroencephalogram recordings as coming from normal or epileptic patients. They gathered a large amount of annotated data and built a system which achieves close to 90% accuracy at detecting those patients. The dataset they use has been annotated at two different levels: The entire recordings are labeled as coming from normal or epileptic patients, and individual anomalies in those recordings have been manually identified by clinicians. Collecting the second level of annotation, the individual anomalies, was an expensive and time consuming process. Qualified doctors have to spend hours selecting those anomalies on a computer and it is clear that getting more of those individual annotations is a bottleneck when it comes to making the system they developed more robust by collecting more data.

The focus of my research has been on building a system with similar accuracy without using the annotated anomalies, just the entire recordings. Succeeding in doing so means that you can scale the system by collecting more data from hospitals (They already know which EEG recordings are coming from epileptic patients) without clinicians to annotate individual recordings.

Clinicians diagnose patients as epileptic by doing pattern matching while watching a live EEG. [1] They are often looking for interictal epileptiform discharge (IED). Those are the aforementioned individual annotated anomalies.

I investigate using Generative Adversarial Networks [2] to build an unsupervised anomaly detection model and then use the anomaly score of short time windows to classify recordings as coming from an epileptic or normal patient.

Chapter 1

Background

1.1 EEG, IED, and how to approach the problem.

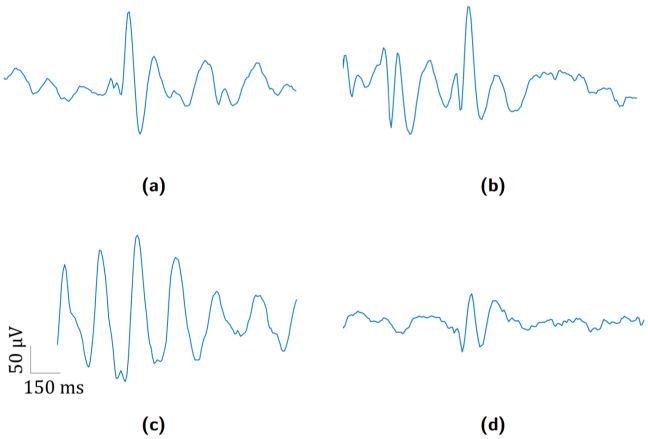
In this section I will briefly describe what an EEG is, how it relates to diagnosing epileptic patients, what IEDs are, and how you can approach the problem of classifying patients as being epileptic or not from an EEG recording.

The human electroencephalogram (EEG) was discovered by the German psychiatrist, Hans Berger, in 1929. It plays a central role in diagnosis of patients with epilepsy while usually being a non invasive procedure. An EEG is carried out by laying down electrodes on the scalp and measuring voltage fluctuations. When discretising the recording, it can be considered as Time serie.

Time series: A time series is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus it is a sequence of discrete-time data. Examples of time series are heights of ocean tides, counts of sunspots, and the daily closing value of the Dow Jones Industrial Average. [3]

Interictal epileptiform discharge (IED) are often a sign of epilepsy [1] and detecing IEDs in an EEG recording is the method that [4] and [5] used to classify recordings as coming from epileptic or normal patients.

Figure 1.1: 4 IEDs with different shapes



Detecting epileptic patients from an EEG recording can be seen, from a machine learning point of view, in a few different ways:

1. As a **Time series classification** problem where you feed the entire discretised EEG into a model. There is only one network which is trained end-to-end.

2. As a **Supervised learning** problem where you cut the recording into fixed size windows and then feed all those windows into a model which can detect IEDs (which was trained using annotated IEDs, thus in a supervised learning fashion). The number of detected IEDs (and the confidence score of those classifications) can then be fed into another model (eg: SVM) to classify the recording as coming from an epileptic patient or not. The two networks are trained separately.

3. As an **Unsupervised learning** problem where an anomaly detection model is trained on fixed size windows of normal patients (ie: Never give the model any IEDs or other anomalies usually found in EEG recordings of epileptic patients). To classify entire recordings you then feed the anomaly scores of every window into a model which can classify sequences (Time series classification using the anomaly scores) or extract hand engineered features (eg: the number of windows with an anomaly score greater than x) and then feed those features into a more general model (just like the the supervised learning method just mentioned). The two networks are trained separately.

Prof Justin Dauwels and his group followed the second approach while I investigated the third one. The obvious advantage of the third approach compared to the second one is not using the individual IEDs annotations which are expensive and time consuming to collect.

1.2 GANs

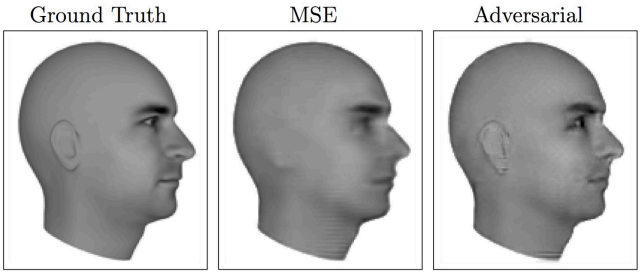
A key element in my model is a Wasserstein GAN. In this section I will introduce the vanilla GAN, list some of its flaws, and describe the transition from the GAN to the WGAN. I assume that the reader of this report is familiar with basic machine learning terminology and supervised deep learning.

A Generative Adversarial Neural Network [2] is a Deep Neural Network which has been trained in an adversarial fashion to generate some data in tensor form. The output of the network can be an image, some text, or even a fixed length 1D output (which can be interpreted as a fixed length window from a timer series).

Generative modelling as been an active area of research in machine learning and a few approaches were popular yet pretty bad (Variational Autoencoders [6] is one of those approaches)

Then came the GAN, it fixed the major problem regarding generating data with a Deep Neural net: The fact that there is not just one "right answer" and that it is fine to not average over all good answers. The following figure and explanation coming from [7] is helpful.

Figure 1.2: Visualising the GAN loss vs MSE



In this example, a model is trained to predict the next frame in a video sequence. The video depicts a computer rendering of a moving 3D model of a person's head. The image on the left shows an example of an actual frame of video, which the model would ideally predict. The image in the center shows what happens when the model is trained using mean squared error between the actual next frame and the model's predicted next frame. The model is forced to choose a single answer for what the next frame will look like. Because there are many possible futures, corresponding to slightly different positions of the head, the single answer that the model chooses corresponds to an average over many slightly different images. This causes the ears to practically vanish and the eyes to become blurry. Using an additional GAN loss, the image on the right is able to understand that there are many possible outputs, each of which is sharp and recognizable as a realistic, detailed image. [8]

One of the key innovation of the GAN is the adversarial loss. A GAN is composed of two networks trained jointly. A generator network maps a vector of random noise (usually a vector of dimension 100) to some type of data in tensor form of higher dimension (An image could be of dimension 256∗256=65536). A discriminator network takes batches of inputs composed of either real data from the dataset that you want the GAN to generate or outputs of the generator network. The discriminator then has to classify the generator outputs as "fake" and the the real data as "real". If you represent the generator network as $G(\mathbf{z})$ where $\mathbf{z}$ is a random vector (usually sampled from an uniform or gaussian distribution) and the discriminator as $D(\mathbf{x})$ where $\mathbf{x}$ is either the output of the generator or an image/sentence/fixed length window from your dataset, you can write down the GAN objective as a minimax game with the following loss function:

$$\begin{aligned}\min_G \max_D L(D,G) &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))] \\ &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1-D(x)]\end{aligned}$$

Figure 1.3: Diagram of the inner working of a GAN

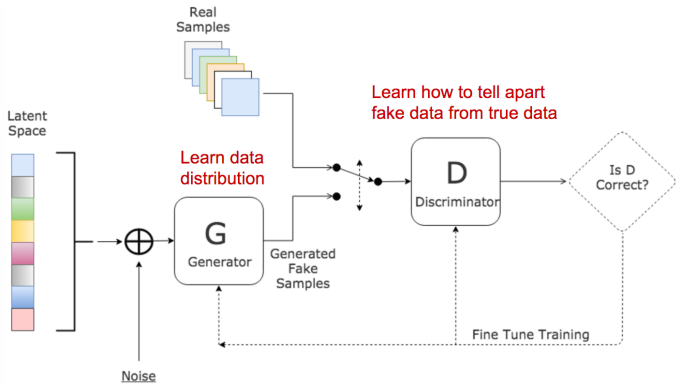


Figure 1.4: Table describing the pdf used in the loss function of a GAN

Symbol	Meaning
p_z	Data distribution over noise input z
p_g	The generator's distribution over data x
p_r	Data distribution over real sample x

The vanilla version of GANs actually does not work well. If you use some tricks that I won't describe here [9], you can get better results. The main problem with vanilla GANs however is the loss function. You can show that the loss function for any generator G and the best possible discriminator D^* can be rewritten as this:

$$L(G, D^*) = 2D_{JS}(p_r \| p_g) - 2\log 2$$

Where D_{JS} is the Jensen–Shannon Divergence. The JS Divergence is a distance function between two distributions. You may be familiar with the KL Divergence, D_{KL}, which is similar to the JS Divergence.

The loss function of the GAN is essentially the distance between the distribution of the real samples and the generated ones. You want your network to imitate the real distribution of your data and thus minimise that distance.

A GAN implementation becomes as good as its distance function (for Vanilla GAN the distance function was implicit, hidden in the minimax). The thing is, D_{JS}, the Jensen-Shannon Divergence, is not a good distance function when used in a Deep Learning context because it has discontinuities.

That means that its gradient can be zero or infinity which makes the training process unstable (and thus why we need to use the tricks listed in [9]). Let me introduce you a better distance function for probability distribution: The Wasserstein Distance.

> **Wikipedia** [10]: In mathematics, the Wasserstein or Kantorovich-Rubinstein metric or distance is a distance function defined between probability distributions on a given metric space M. Intuitively, if each distribution is viewed as a unit amount of "dirt" piled on M, the metric is the minimum "cost" of turning one pile into the other, which is assumed to be the amount of dirt that needs to be moved times the mean distance it has to be moved. Because of this analogy, the metric is known in computer science as the earth mover's distance.

The formula for the Wasserstein distance is $W(p_r, p_g)$ where:

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

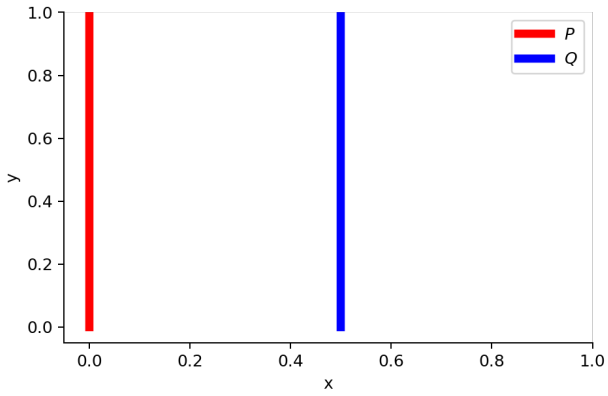
1. $\Pi(p_r, p_g)$ is the set of all possible joint probability distributions between p_r (the PDF of the distribution of the real samples in your dataset) and p_g (the PDF of the implicit distribution represented by the generator network G)

2. One joint distribution $\gamma \in \Pi(p_r, p_g)$ describes one journey where dirt will be transported. Precisely $\gamma(x, y)$ states the percentage of dirt that should be transported from point x to y so as to make x follows the same probability distribution of y.

Now let's see why this $W(p_r, p_g)$ distance is better than D_{JS} or even D_{KL}. Let's use two simple multivariate distribution as a case study.

$$\forall (x, y) \in P, x = 0 \text{ and } y \sim U(0, 1)$$
$$\forall (x, y) \in Q, x = \theta, 0 \leq \theta \leq 1 \text{ and } y \sim U(0, 1)$$

Figure 1.5: There is no overlap between P and Q when $\theta \neq 0$



When $\theta \neq 0$:

$$D_{KL}(Q\|P) = \sum_{x=\theta, y\sim U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty$$

$$D_{JS}(P,Q) = \frac{1}{2}\left(\sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{1/2} + \sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{1/2}\right) = \log 2$$

$$W(P,Q) = |\theta|$$

When $\theta = 0$, and thus the two distributions are fully overlapped:

$$D_{KL}(P\|Q) = D_{KL}(Q\|P) = D_{JS}(P,Q) = 0$$
$$W(P,Q) = 0 = |\theta|$$

D_{KL} gives us infinity when two distributions are disjoint. The value of D_{JS} has sudden discontinuities, not differentiable at $\theta = 0$. Only Wasserstein metric provides a smooth measure.

That's precisely why, if you use the Wasserstein distance as your loss function in the training of a GAN, you get much better results (gradient descent suffers from discontinuities). The loss becomes:

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w\in W} \mathbb{E}_{x\sim p_r}[f_w(x)] - \mathbb{E}_{z\sim p_r(z)}[f_w(g_\theta(z))]$$

Where $f_w(x)$ is essentially the discriminator but coming from a set of K-Lipschitz continuous functions. Again, I will not go into the details, you can refer to the WGAN the paper. [11]

We now have all the required building blocks to start describing the model, I hope you are excited.

Chapter 2

The Model

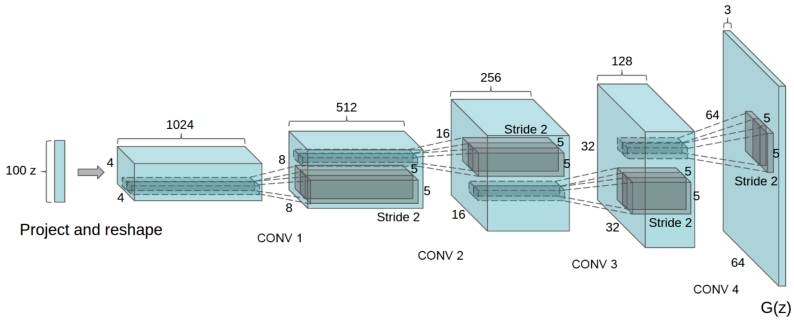
The first thing which should come to your mind is:

How can we detect anomalies with a GAN if GANs are typically used to generate data (And not do classification or regression)?

The intuition is as follows: GANs can only generate patterns they have seen before. If the samples in your dataset do not contain anomalies, then you can be sure that G will not be able to generate any either. You can then train an **Encoder**. This idea of Encoder and the izi_f training procedure that will be described later in this report comes from [12]. Remember that G takes a vector of random Gaussian noise as its input and use it to generate new samples (implicitly, what G is doing is mapping $\mathcal{N}(1, 0)$ to the real distribution p_r). You can actually go backward and, from a sample, find a vector $\mathbf{z}$ such that $G(\mathbf{z})$ will output something very similar to the sample. This Encoder network, E, can then be used to detect anomalies. As we mentioned, a GAN cannot generate anomalies if it has never seen any. Thus if you give an anomaly to E, it will find a vector $\mathbf{z}$. But if you then feed this vector $\mathbf{z}$ into G then the output will be very different from the anomaly you gave to the encoder. We can define a naive anomaly score function: $\mathrm{A(x)} = 1\frac{}{n \cdot ||\mathbf{x} - G(E(\mathbf{x}))||^2}$ where n is the dimension of the input.

Now, how do we train E, and what form does it take? Let's define the architecture of D, G, and E. I decided to use the architecture described in [13] which is now used in most GAN researchs: The Deep Convolutional GAN.

Figure 2.1: The DCGAN architecture for G. The "Project and Reshape" is basically a Fully Connected layer and every Conv layer is a Convolution Transpose. Read [14] for an intuitive explanation of Convolution Transpose.



The architecture for D and E is basically the reverse of the architecture of G. The final layer of D is a Fully Connected layer with one output node and a sigmoid as its activation function (traditional setup for binary classification). The final layer of E is a Fully Connected layer with n outputs (where n is the dimension of $\mathbf{z} \in \mathcal{R}^n$) and tanh as its activation function (We restrict the output of $\mathbf{z}$ such that its values cannot be outside one σ of the distribution used during training, $\mathcal{N}(1, 0)$).

There is another we need to figure out: What is the shape and form of the data we are going to give to the Generator, Discriminator, and Encoder? Entire EEG recordings? That will not work because those networks only accept fixed size input and output. A discretised EEG is basically a multivariate time-series. There are c channels (electrodes), each of them having n measurements or time-steps. The dataset provided by the research group of Prof Justin Dauwels consists of EEGs sampled at 128hz. Every EEG is represented by a $c*n$ matrix where every row represents the entire recording of one channel.

I decided to cut every channel into windows of size 64 (Thus every window is a $\mathcal{R}^{64}$ vector) by having a rolling window process go over each channel (I used a window step size of 32, that means there is a 50% overlap between adjacent windows). This window size and window step size are equivalent to the one used in [4] (The main paper published by the research group).

The training process for G, D, and E is as follows:

1. Preprocess the **normal patients** EEGs by first applying a CAR Mon-

tage (subtract the mean of all channels from every channel, it is commonly done when dealing with EEGs in neuroscience) and then using the rolling window process to create a **spikefree_window** dataset.

2. Train a DCGAN [13] modified to use a WGAN loss [11] on the **spikefree_window** dataset. The dimension of $\mathbf{z}$ is an hyperparameter. This process yields G and D.

3. Train an Encoder E using the izi_f procedure on the **spikefree_window** dataset.

A few details about the training procedure of D:

D is basically doing regression, it has to find the best $\mathbf{z}$ for a given x such that the reconstruction error is small. The loss for D is:

$$\mathcal{L}_{izi_f}(\mathbf{x}) = \frac{1}{n} \cdot ||\mathbf{x} - G(E(\mathbf{x}))||^2 + \frac{\kappa}{n_d} \cdot ||f(\mathbf{x}) - f(G(E(\mathbf{x})))||^2$$

As you can see, this is very similar to the naive anomaly function which was described earlier, with the addition of a second term. The first term in the sum is the **reconstruction loss** where D is penalised for the squared difference between the sample and the reconstructed one. The second term is the **discriminator feature loss**. f is actually the output of one of the last layers of D, and this output is commonly referred as a feature vector. In Deep Learning, you can actually take a very good model trained on ImageNet (Like a trained ResNet) and use the output of an intermediate layer as a feature generator. You can thus train a new model with your own data by first feeding images into the ablated ResNet and then feeding the feature vectors into a classifier. This approach works extremely well and it commonly called "embedding". Here we use one layer of D to force the regenerated sample to have a similar feature vector to the feature vector of the original sample. This yields better results and more explanations can be found in [12]. n_d is the dimension of the feature vector, κ is an hyperparameter.

izi$_\mathbf{f}$: There are two ways to train the encoder. You can sample a vector $\mathbf{z}$ from $\mathcal{N}(1,\, 0)$ and use it to generate some data x using G. You then give this x to the decoder and measure the difference between the original $\mathbf{z}$ and the output of E. This method is called ziz. izi is simply the training method we already described but without the second term in the loss. That is you do not use a discriminator's feature layer to guide the reconstruction. izi_f is the training procedure whose loss has been given earlier (Reconstruction loss and discriminator feature loss). It led to the best results in [12] and this

is the training procedure for E to I decided to use. I want to also mention that the weights of G and D are frozen during the training of E. We back-propagate through those two networks but the optimiser **does not** update their weights during the izi_f training procedure.

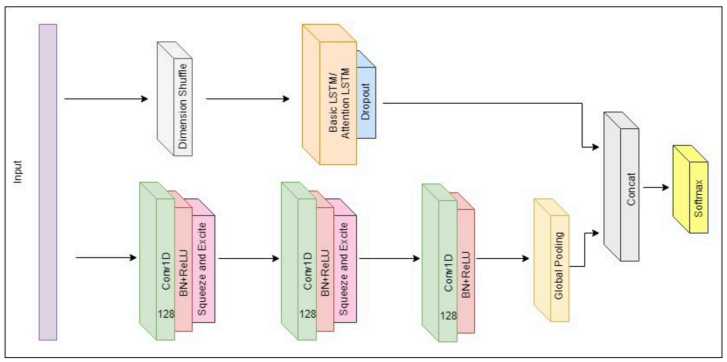
We now have a model (consisting of G, D, and E) which can compute an anomaly score for any window of size 64 (a vector $\mathbf{x} \in \mathcal{R}^{64}$). The anomaly score function is:

$$\mathcal{A}(\mathbf{x}) = \frac{1}{n} \cdot ||\mathbf{x} - G(E(\mathbf{x}))||^2 + \frac{1}{n_d} \cdot ||f(\mathbf{x}) - f(G(E(\mathbf{x})))||^2$$

The last moving part which needs to be described is a model which can classify **processed recordings**. Processed recordings are complete EEG recordings cut into windows where each window's anomaly score is computed using G, D, and E. A processed recording is represented by a $c * a$ matrix where each row of the matrix is the anomaly score of every window in the channel ($a < n$ as every window's anomaly score is just one real number). The windows are generated using the same rolling window process used to generate the **spikefree_window** dataset.

A very strong baseline in multivariate time-series classification is the Multivariate LSTM-FCNs described in this paper: [15]. A trend in time-series classification has been using Convolutional Layers instead of Recurrent Networks to classify sequences. This model, the LSTM-FCN, uses both. The trick is that the LSTM goes through the variables and not through time. The model transposes the data before feeding it into the LSTM to trick it into considering the variables as time-steps and the time-steps as variables. By doing that permutation you let the network learn relationships between the variables but force the actual processing through time to be done by CNNs which will be order of magnitudes faster. One of the disadvantage with this approach is that you need to pad your sequences with zeroes for them to all have the same length.

Figure 2.2: The LSTM-FCN architecture. [15]



We can now describe the second part of our training process:

1. Compute the anomaly score (using the aforementioned formula) of windows of size 64 and step size 32 in both **normal patients** EEGs and **epileptic patients** EEGs. This yields a new dataset, **processed_recordings**. The processed records in this dataset are comprised of $c * a$ matrices where every row of each matrix represents the entire processed recording of one channel ($a < n$ as every window's anomaly score is just one real number) of one EEG.

2. Train a LSTM-FCN, L, on the **processed_recordings** dataset. This is a binary classification setup where L has to classify processed recordings into epileptic or normal. Use oversampling to make sure that the ratio between epileptic and normal processed recordings in every batch is close to 1 (There are much more recordings of normal patients in the dataset)

The model has now been completely described. It is time to do some experiments.

Chapter 3

Experiments

3.1 WGAN

Let's now report on the experiments which have been done with the Wasserstein GAN. Training this model on CPU would take more than 24 hours which means I had to find another solution. I thus called Amazon and asked them to give me access to a p3.2xlarge instance on the Amazon Web Service EC2 cloud here in Singapore. That instance type gives you access to an Nvidia V100 GPU which brought the training time down to 5 hours. I only fiddled with two hyperparameters: The dimension of the $\mathbf{z}$ vector and wether I would "squash" the dataset and add a tanh activation function on the last layer of G.

Typically, GANs output are restricted to a range of real numbers. This makes training much more stable [8] and is a natural thing to do when you work with images (every pixel value is between 0 and 255, researchers usually rescale and translate the dataset such that 255 is mapped to 1 and 0 is mapped to -1. They can then add a tanh activation function to the G and restricts its output to this range).

However, when working with time series data, doing so is a less intuitive decision. Anomalies are not only characterised by their shape but also by their amplitude. I tried "squashing" every window from the **spike-free_window** dataset, that is translating and rescaling very window such that the peak is mapped to 1 and the lowest value is mapped to -1. This hyperparameter, "squash", means that the dataset has been squashed and an additional tanh activation function has been added to the output of G. The follow subsections and figures illustrate the training process for every pair of hyperparameters (dimension of $\mathbf{z}$, squashing).

3.1.1 WGAN. No Squash. $\mathbf{z} \in \mathcal{R}^{32}$

Figure 3.1: Discriminator loss over time

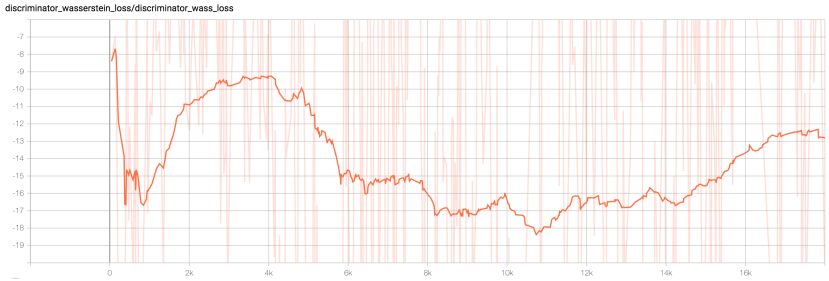


Figure 3.2: Generator loss over time. We can see that it progressively became better at tricking the discriminator

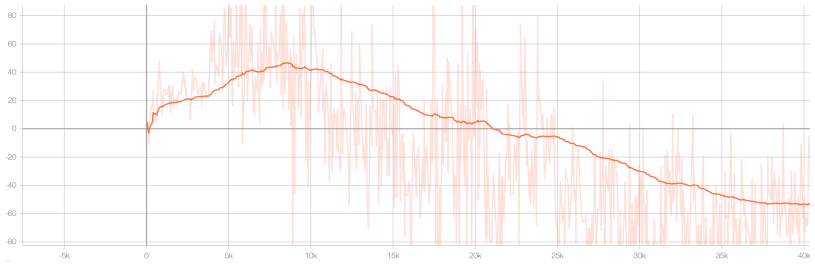
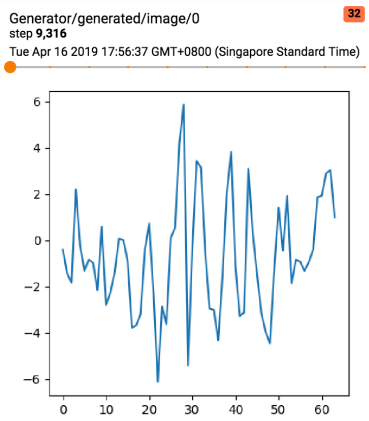
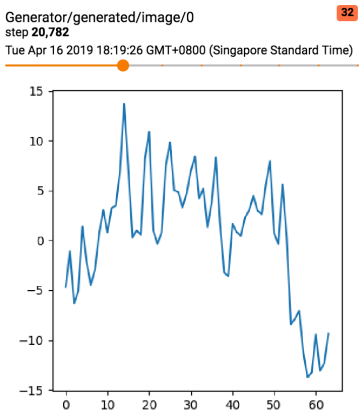


Figure 3.3: Some examples of windows generated by the GAN

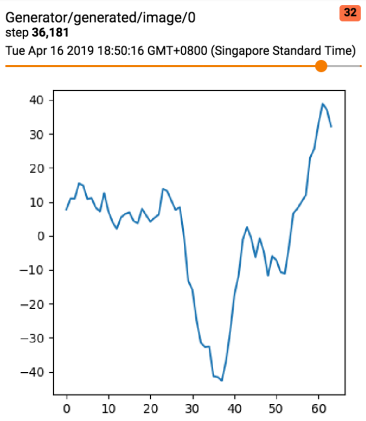
(a) Early in the training process



(b) Some times later



(c) End of the training process



3.1.2 WGAN. No Squash. $\mathbf{z} \in \mathcal{R}^{16}$

Figure 3.4: Discriminator loss over time

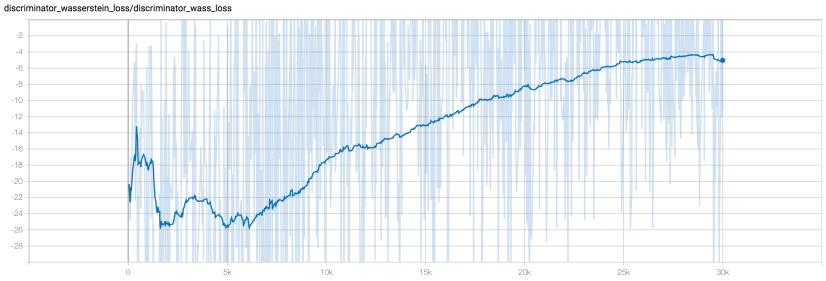


Figure 3.5: Generator loss over time. We can see that it progressively became better at tricking the discriminator

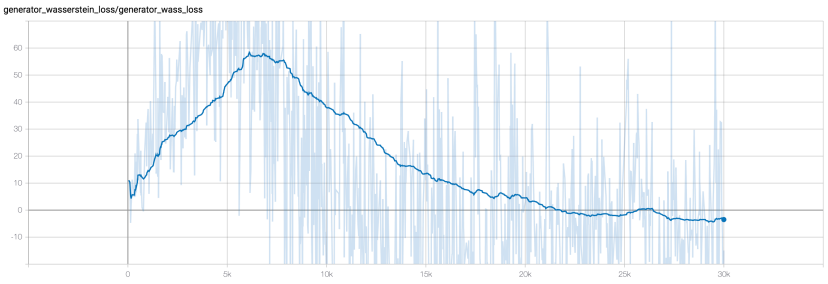
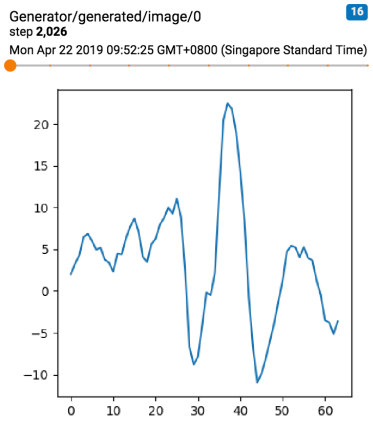
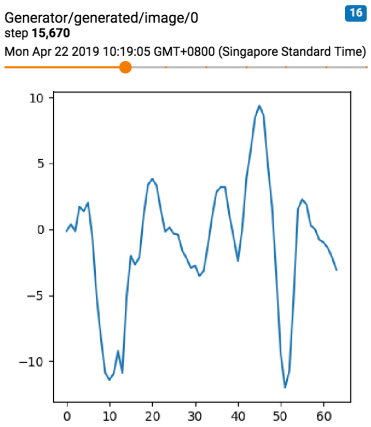


Figure 3.6: Some examples of windows generated by the GAN

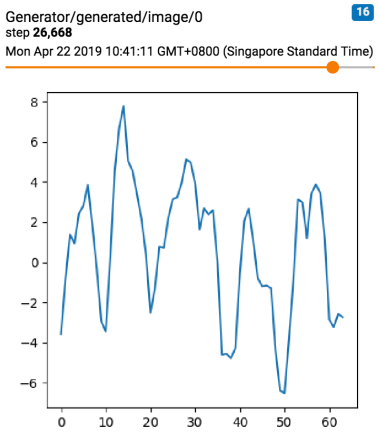
(a) Early in the training process



(b) Some times later



(c) End of the training process



3.1.3 WGAN. Squash. $\mathbf{z} \in \mathcal{R}^{32}$

Figure 3.7: Discriminator loss over time

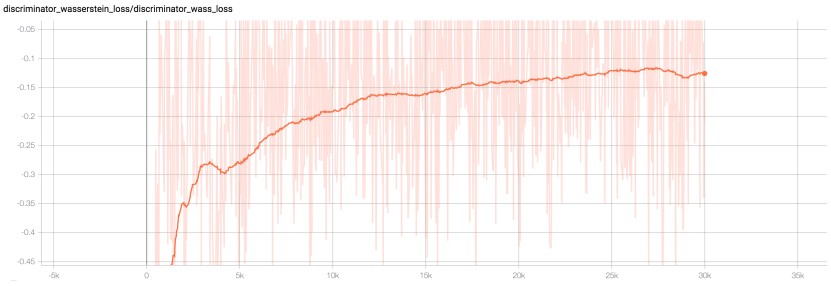


Figure 3.8: Generator loss over time

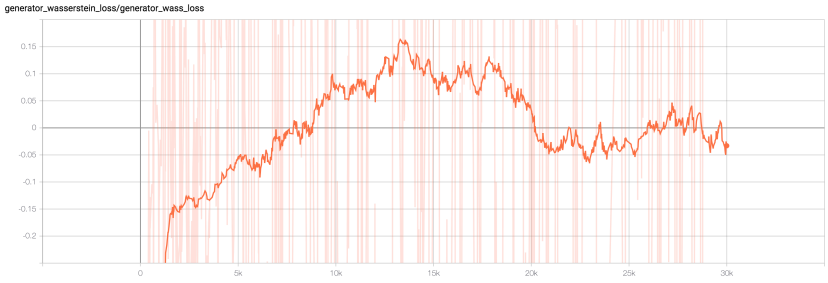
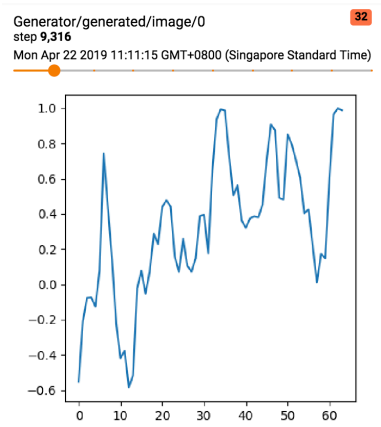
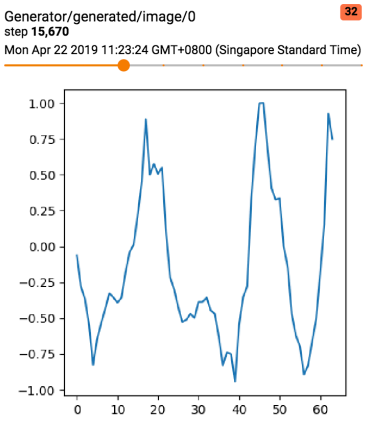


Figure 3.9: Some examples of windows generated by the GAN (Squashed as expected)

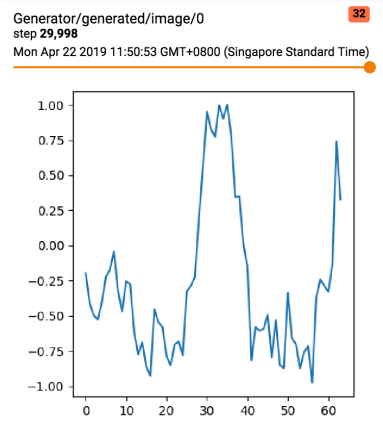
(a) Early in the training process



(b) Some times later



(c) End of the training process



3.1.4 WGAN. Squash. $\mathbf{z} \in \mathcal{R}^{16}$

Figure 3.10: Discriminator loss over time

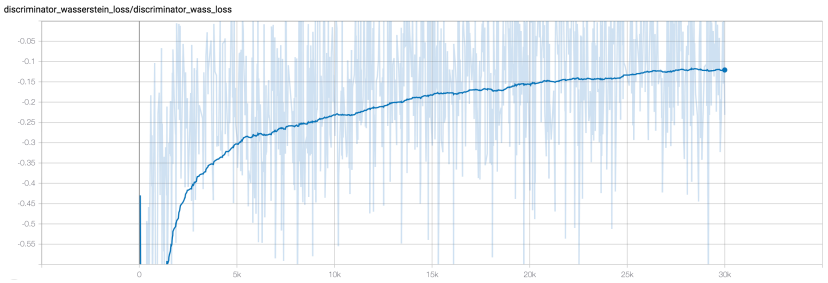


Figure 3.11: Generator loss over time

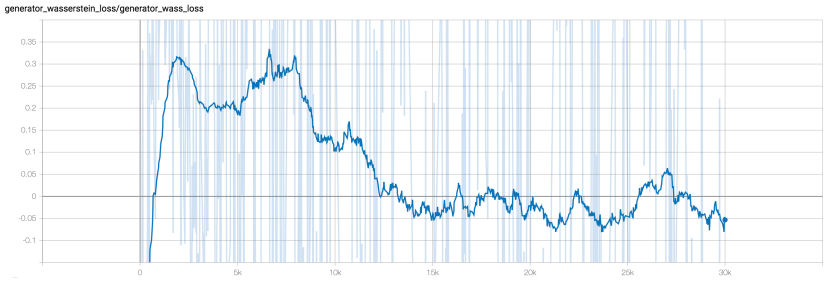
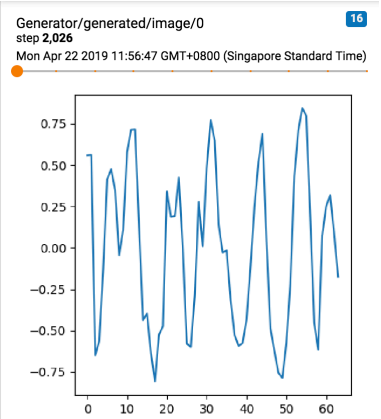
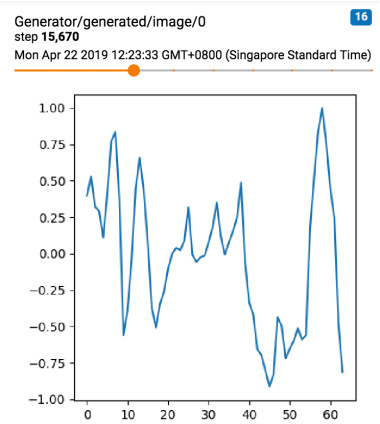


Figure 3.12: Some examples of windows generated by the GAN (Squashed as expected)

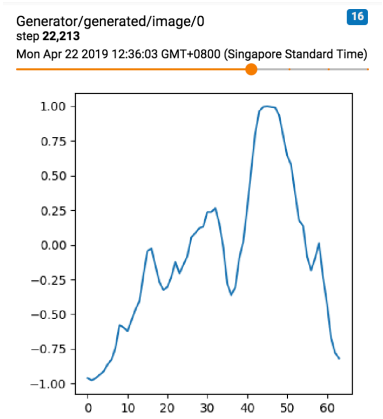
(a) Early in the training process



(b) Some times later



(c) End of the training process



3.2 Encoder

Similarly, the Encoder was trained on the EC2 instance. I did not vary any hyperparameter when training the encoder. Training one Encoder for every combination of the previous hyperparameters, the one chosen for every WGAN, took a fair amount of time. The only experiment I carried out for a subset of WGAN was not restricting the output $E(\mathbf{z})$. The researchers of [12] tried this approach too but finally decided not to drop the regularisation

process at the output of the E. That regularisation process amounts to adding a tanh activation function to the last Fully Connected layer. If you do that, the values of $\mathbf{z}$ cannot be outside one σ of the distribution used during training, $\mathcal{N}(1, 0)$.

When dropping the regularisation, the Encoder could output a vector $\mathbf{z}$ with values that the Generator had never seen before and thus game the system by using extreme $\mathbf{z}$ to generate almost any pattern. This obviously made the entire Anomaly Detector pattern useless and thus I kept the tanh function at the output of E.

Once again, The follow subsections and figures illustrate the training process of the Encoder for every pair of WGAN's hyperparameter (dimension of $\mathbf{z}$, squashing) plus a table with some statistics on the Anomaly Score computed using that pair of WGAN/Encoder on windows from the **spike-free_window** dataset, windows from the epileptic EEGs recordings, and windows with IEDs (when want to make sure that IEDs are definitely picked up as anomalies). These IEDs were obtained using the second level of annotation in the research group's dataset.

3.2.1 Encoder. No Squash. $\mathbf{z} \in \mathcal{R}^{32}$

Figure 3.13: Encoder loss over time. It converges in less than 1000 steps

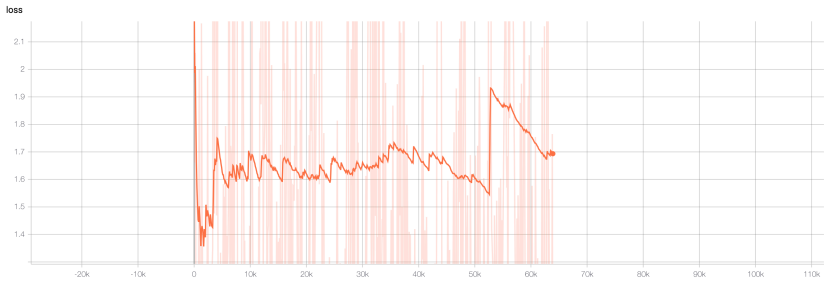
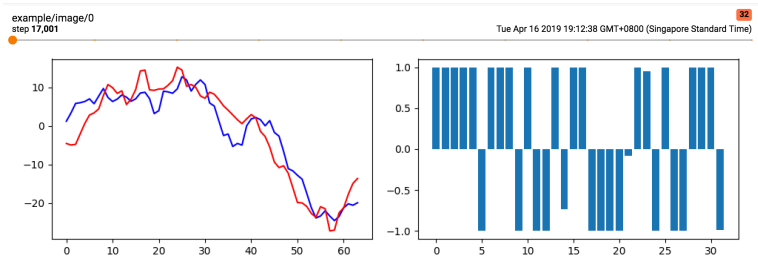


Figure 3.14: Some examples of windows generated by the GAN. The time series in blue is the input to E and the red time series is the reconstruction given by $G(E(\mathbf{x}))$. The histogram is the $\mathbf{z}$ vector generated by E.

(a) Early in the training process



(b) Some times later

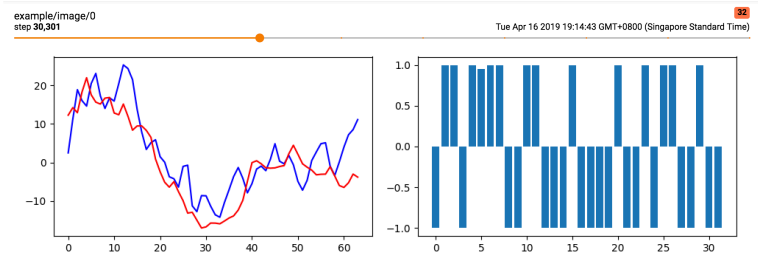


Table 3.1: Anomaly Detection results

Statistics	Spikefree	Epileptics	IEDs
Mean	53.47	74.66	**504.60**
Max	1816.24	3216.42	5319.39
Min	20.02	19.32	29.24
Median	43.66	57.88	398.65

3.2.2 Encoder. No Squash. $\mathbf{z} \in \mathcal{R}^{16}$

Figure 3.15: Encoder loss over time. It converges in less than 1000 steps

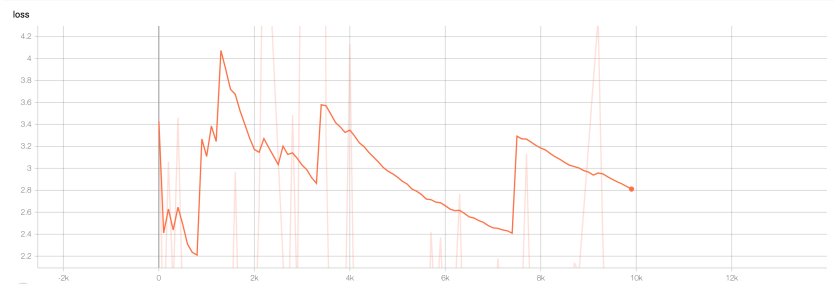
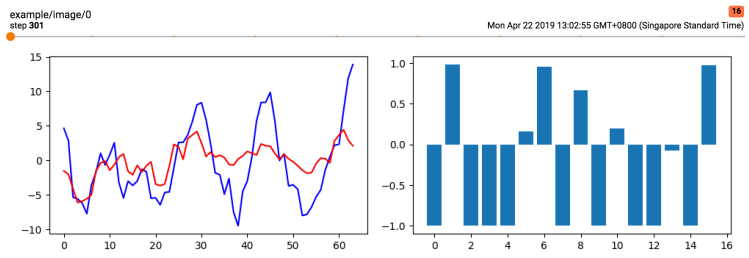


Figure 3.16: Some examples of windows generated by the GAN. The time series in blue is the input to E and the red time series is the reconstruction given by $G(E(\mathbf{x}))$. The histogram is the $\mathbf{z}$ vector generated by E.

(a) Early in the training process



(b) Some times later

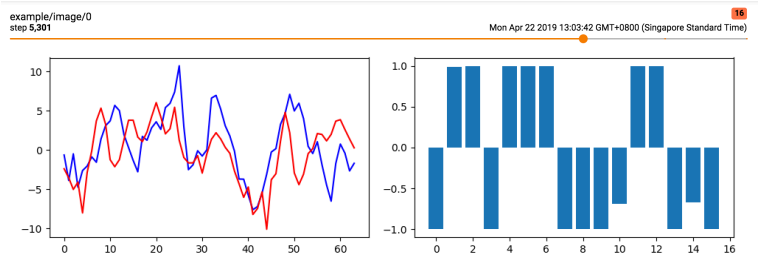


Table 3.2: Anomaly Detection results

Statistics	Spikefree	Epileptics	IEDs
Mean	53.88	79.26	**488.24**
Max	1617.46	5371.25	4945.40
Min	16.85	12.00	65.29
Median	43.07	58.89	364.61

3.2.3 Encoder. Squash. $\mathbf{z} \in \mathcal{R}^{32}$

Figure 3.17: Encoder loss over time. It converges in less than 1000 steps

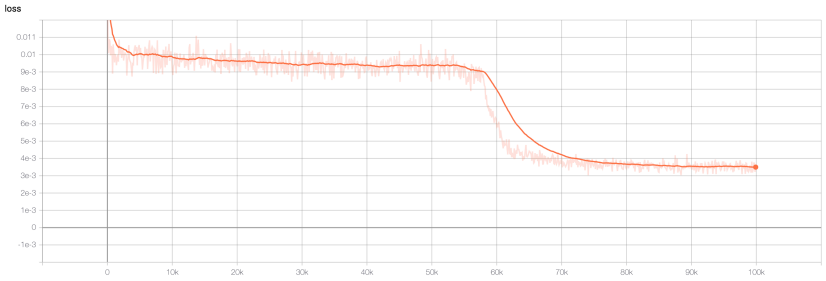
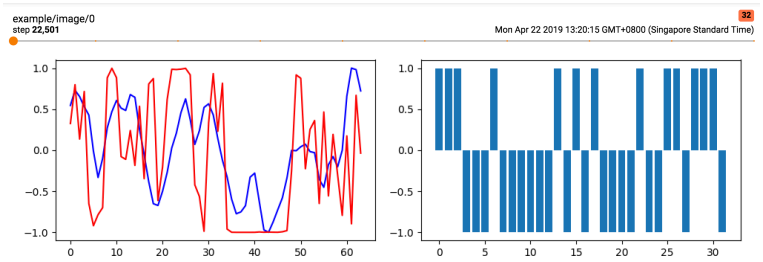


Figure 3.18: Some examples of windows generated by the GAN. The time series in blue is the input to E and the red time series is the reconstruction given by $G(E(\mathbf{x}))$. The histogram is the $\mathbf{z}$ vector generated by E.

(a) Early in the training process



(b) Some times later

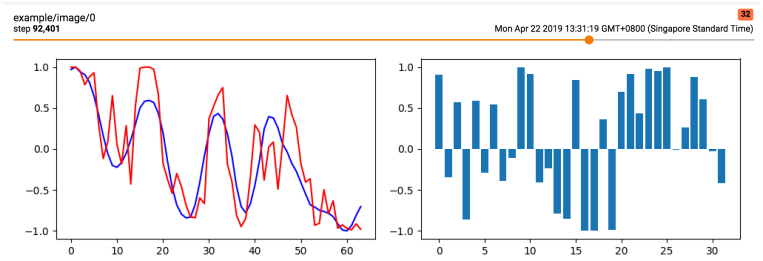


Table 3.3: Anomaly Detection results

Statistics	Spikefree	Epileptics	IEDs
Mean	1.14	1.15	**1.14**
Max	2.15	2.37	2.30
Min	0.41	0.36	0.39
Median	1.13	1.14	1.12

3.2.4 Encoder. Squash. $\mathbf{z} \in \mathcal{R}^{16}$

Figure 3.19: Encoder loss over time. It converges in less than 1000 steps

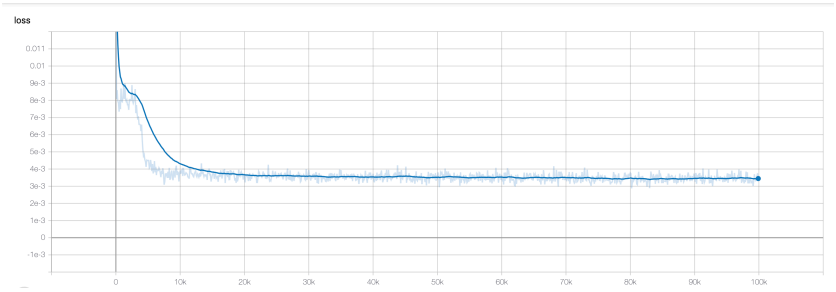
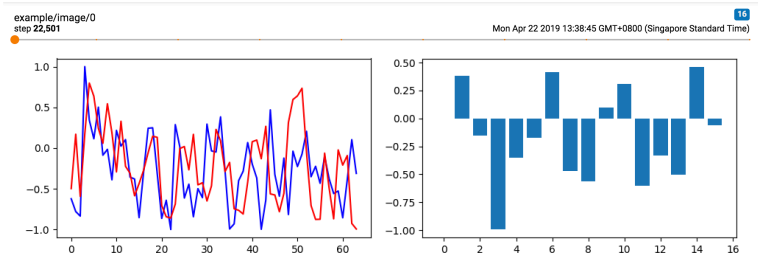


Figure 3.20: Some examples of windows generated by the GAN. The time series in blue is the input to E and the red time series is the reconstruction given by $G(E(\mathbf{x}))$. The histogram is the $\mathbf{z}$ vector generated by E.

(a) Early in the training process



(b) Some times later

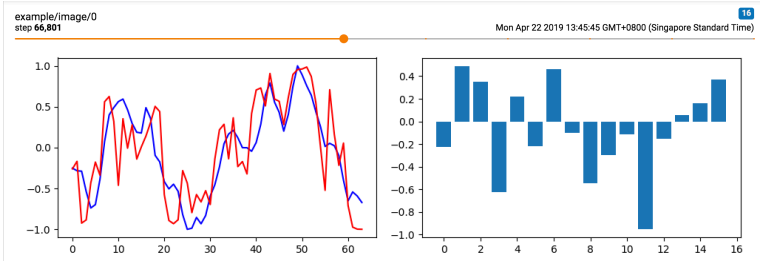


Table 3.4: Anomaly Detection results

Statistics	Spikefree	Epileptics	IEDs
Mean	0.42	0.40	**0.40**
Max	1.54	1.34	1.42
Min	0.08	0.05	0.08
Median	0.36	0.34	0.35

3.3 LSTM-FCN

Until now, every part of the model has been successful at doing what it was designed for: The WGAN generates windows which cannot be distinguished from real ones by a Discriminator network, the Encoder is quite good at finding $\mathbf{z}$ given a window, and the anomaly scores are reasonable (IEDs have an high anomaly scores while the anomaly scores of windows in spike free recordings are, in average, ten times lower than the IEDs' ones).

We first generate the **processed_recordings** dataset using the process described earlier. The WGAN/Encoder pair we will be using is the one with hyperparameters $\mathbf{z} \in \mathcal{R}^{32}$ and no Squash. This pair yielded anomaly scores with the biggest difference between the median score of IEDs and the median score of spike free windows.

I did not vary any hyperparamter of the LSTM-FCN (I used the ones in [15]). I still ran two experiments:

1. Using a **processed_recordings** dataset with recordings coming from one hospital only (The same hospital used to generate the data involved in the training of the two previous models)

2. Using a **processed_recordings** dataset with recordings coming from multiple hospitals (And retraining the WGAN + Encoder using windows generated from this new set of data)

Table 3.5: Time series classification using the LSTM-FCN results

Metric	One Hospital	Multiple Hospitals
Train Accuracy	53%	45%
Validation Accuracy	43%	42%

Chapter 4

Conclusion

Detecting anomalies in time series data works, using those anomalies to classify EEG recordings as coming from epileptic patients does not. The intuition behind why this did not work is as follows: The anomaly detection model picks up any kind of anomalies but the one used by physicians to identity epileptic patients are very specific. A time series classification model trained on top of those anomaly scores cannot distinguish from "useful" anomalies (ie: IEDS) and useless anomalies (ie: The patient moving its head and provoking electrical noises in the electrodes). It does yields bad predictions.

Bibliography

[1] S. J. M. Smith, "Eeg in the diagnosis, classification, and management of patients with epilepsy," *Journal of Neurology, Neurosurgery & Psychiatry*, vol. 76, no. suppl 2, pp. ii2–ii7, 2005.

[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *arXiv e-prints*, June 2014.

[3] "Time series - wikipedia." (Accessed on 04/17/2019).

[4] Y. RAJAMANICKAM, J. Thomas, T. Kluge, and J. Dauwels, "A deep learning scheme for automatic seizure detection from long-term scalp eeg," pp. 368–372, 10 2018.

[5] I. Ullah, M. Hussain, E.-u.-H. Qazi, and H. Aboalsamh, "An Automated System for Epilepsy Detection using EEG Brain Signals based on Deep Learning Approach," *arXiv e-prints*, Jan. 2018.

[6] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *arXiv e-prints*, Dec. 2013.

[7] W. Lotter, G. Kreiman, and D. Cox, "Unsupervised Learning of Visual Structure using Predictive Generative Networks," *arXiv e-prints*, Nov. 2015.

[8] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," *arXiv e-prints*, Dec. 2017.

[9] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved Techniques for Training GANs," *arXiv e-prints*, June 2016.

[10] "Wasserstein metric - wikipedia." `https://en.wikipedia.org/wiki/Wasserstein_metric`. (Accessed on 04/18/2019).

[11] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv e-prints*, Jan. 2017.

[12] T. Schlegl, P. Seeböck, S. M. Waldstein, G. Langs, and U. Schmidt-Erfurth, "f-anogan: Fast unsupervised anomaly detection with generative adversarial networks," *Medical Image Analysis*, vol. 54, pp. 30 – 44, 2019.

[13] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv e-prints*, Nov. 2015.

[14] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv e-prints*, Mar. 2016.

[15] F. Karim, S. Majumdar, H. Darabi, and S. Harford, "Multivariate LSTM-FCNs for Time Series Classification," *arXiv e-prints*, Jan. 2018.