“ Identification de la réponse électromagnétique des défauts non francs dans des signaux de réflectométrie ”

XU Kaiyuan

Tuteur : KAMENI Abelin, LOETE Florent

1 Juillet 2024

Table des matières

0 Introduction

0.1 Problématique

On voudrait indentifier si l'on a des défauts dans le câble :

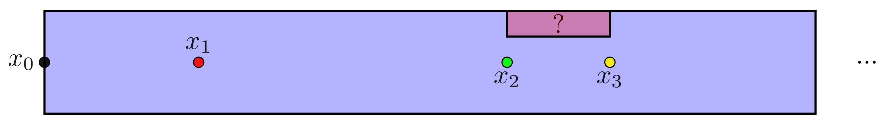


FIGURE 0.1.1 – Un câble en violet avec défaut en rouge, vue en section transversale.

on va envoyer une onde électromagnétique sous la forme gaussienne à la postion x_0 et on va mettre un détecteur à la position x_1. Et si l'on a un défaut, on note son côté gauche à x_2 et son côté droite à x_3 :

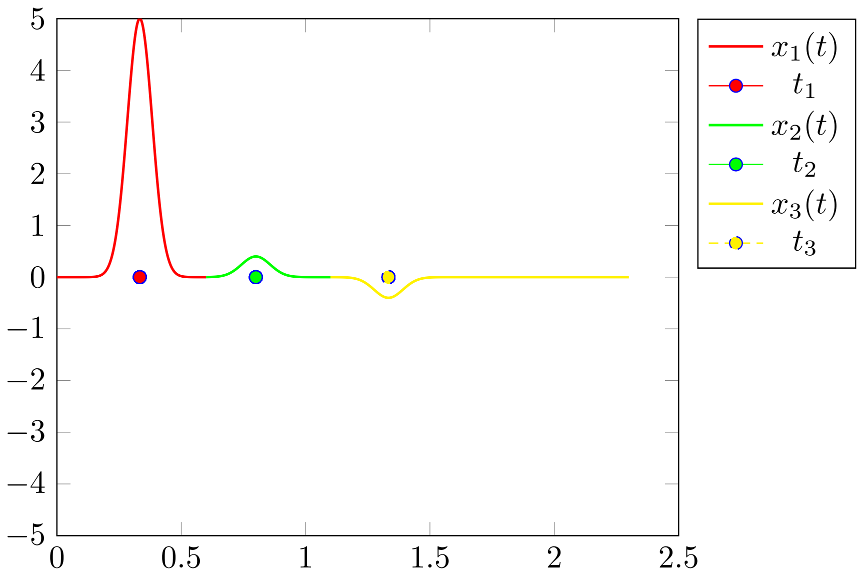


FIGURE 0.1.2 – Une illustration du signal obtenu par le capteur (axe vertical : Amplitude ; axe horizontal : Temps).

La figure 0.1.2 montre que

- On va envoyer une onde gaussienne va partir de la postion x_0. Quand elle arrive la position du détecteur x_1 on va recevoir $x_1(t)$ à l'instant t_1. C'est l'onde incidente (en rouge).
- Si on a un défaut dans ce câble, cette onde incidente gaussienne va toucher le côté gauche à x_2, donc on va recevoir une onde réflichie $x_2(t)$; Après l'onde incidente gaussienne part le côté doite à x_3, on doit recevoir une l'autre onde réflichie $x_3(t)$.

Mais dans notre expérience, il y aura du bruit dans dans le signal reçu. Notre objectif est de déterminer si l'onde réfléchie (soit $x_2(t)$ et $x_3(t)$) est présent dans le signal obtenu expérimentalement.

Au début, on va utiliser les méthodes classiques (Filtre adapté / Transformée de Fourier à fenêtre glissante) : on augmente le niveau de bruit (autrement la densité spectrale de puissance) progressivement jusqu'à ce qu'il atteint sa limite.

Puis on va construire un réseau de neurones à convolution pour aller plus loin. J'ai appris grâce à [3].

0.2 Notation

Définition 0.2.1: Transformée de Fourier

La transformée de Fourier $\tilde{x} = TF(x)$ de $x \in \mathcal{L}^1(\mathbb{R})$ est la fonction

$$TF[x] = \tilde{x}(\nu) = \int_{-\infty}^{\infty} x(t)\, e^{-\imath 2\pi \nu t} dt$$

Définition 0.2.2: Transformée de Fourier inverse

Pour une fonction $x \in \mathcal{L}^1(\mathbb{R})$, sa transformée de Fourier inverse est définie par

$$TF^{-1}[\tilde{x}] = x(t) = \int_{-\infty}^{\infty} \tilde{x}(\nu)\, e^{+\imath 2\pi\nu t} d\nu$$

0.3 La génération du bruit aléatoire gaussien

Pour la génération du bruit aléatoire gaussien avec certaine densité spectrale de puissance, on peut écrire :

```
dsp = 1.5e-5; grn = wgn(2*N, 1, dsp, 'linear');
```

est équivalent à

```
dsp = 1.5e-5; b = sqrt(dsp)*randn(1,2*N);
```

soit

```
std(grn) = std(b) = sqrt(dsp)
```

Pour le code, voir Listing 2. [5]

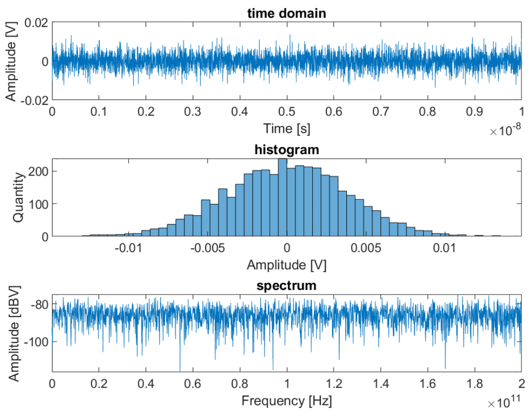


FIGURE 0.3.1 – wgn() | $dsp = 1.5 * 10^{-5}$

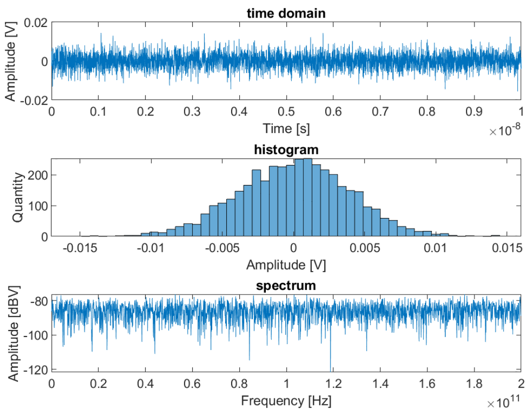


FIGURE 0.3.2 – randn() | $dsp = 1.5 * 10^{-5}$

1 Filtre adapté

1.1 Modèle physique

Définition 1.1.1: Intercorrélation

Intercorrélation temporelle $c^E_{xy}(\tau)$ des signaux x et y est la fonction

$$c^E_{xy}(\tau) = \int_{-\infty}^{+\infty} x(t)y^*(t-\tau)\ dt,$$

pour le temps contine ;

$$c^E_{xy}[k] = \sum_{n=-\infty}^{+\infty} x[n]y^*[n-k],$$

pour le temps discret.

Définition 1.1.2: Convolution

Le produit de convolution $f * g$ de deux fonction x et h

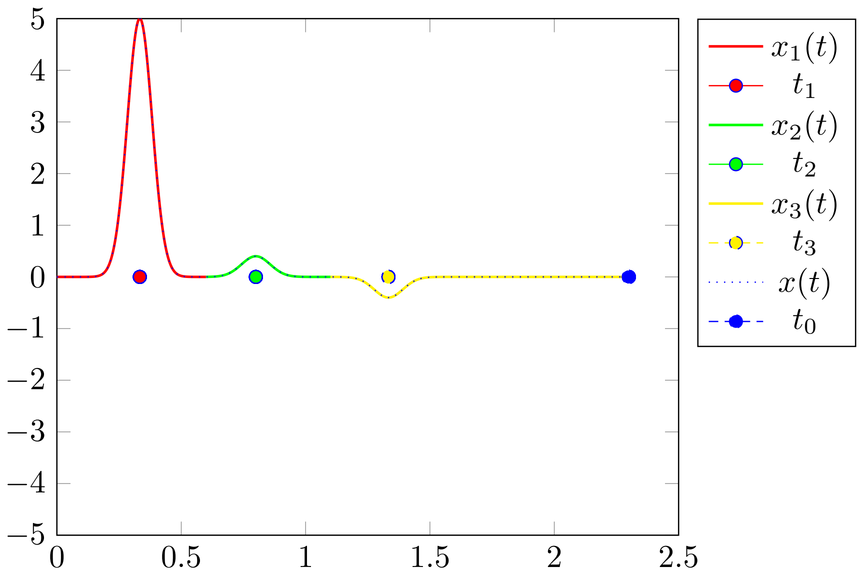
$$x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)\ d\tau$$

en particulier, on a

$$x(t) * h(t-t_0) = \int_{-\infty}^{+\infty} x(\tau)h((t-\tau)-t_0)\ d\tau$$

et

$$x(t) * h(t_0-t) = \int_{-\infty}^{+\infty} x(\tau)h(t_0-(t-\tau))\ d\tau$$



On note $s^f(t)$ le signal en sortie du filtre adapté, idéalement on a :

$$s^f(t) = \overbrace{(x_1(t)+x_2(t)+x_3(t)+b(t))}^{x(t)} * x_1^*(t_0-t)$$

Pour l'onde incidente $x_1(t) \sim \mathcal{N}(t_1, \sigma_1^2)$:

$$\begin{aligned} I_1 = x_1(t) * x_1^*(t_0-t) &= \int_{-\infty}^{+\infty} x_1(\tau)x_1^*(t_0-(t-\tau))\ d\tau \\ &= \int_{-\infty}^{+\infty} x_1(\tau)x_1^*(t_0-t+\tau)\ d\tau \\ &= c^E_{x_1x_1}(t-t_0) \end{aligned}$$

soit on doit avoir un maximum local à $t_{opt1} = t_0$;

Pour l'un des deux ondes réfléchises $x_2(t) \sim \mathcal{N}(t_2, \sigma_2^2) = +ax_1\left(\frac{\sigma_1}{\sigma_2}(t-t_2)+t_1\right) := +ax_1\left(b(t-t_2)+t_1\right)$:

$$\begin{aligned} I_2 = +ax_1\left(b(\tau-t_2)+t_1\right) * x_1^*(t_0-t) &= a\int_{-\infty}^{+\infty} x_1\left(b(t-t_2)+t_1\right)x_1^*(t_0-(t-\tau))\,d\tau \\ &= a\int_{-\infty}^{+\infty} e^{-\frac{(\tau-t_2)^2}{2\sigma_2^2}}\,e^{-\frac{((t_0-(t-\tau))-t_1)^2}{2\sigma_1^2}}\,d\tau \end{aligned}$$

comme

$$-\frac{(\tau-t_2)^2}{2\sigma_2^2} - \frac{((t_0-(t-\tau))-t_1)^2}{2\sigma_1^2} = -\frac{(\tau-t_2)^2}{2\sigma_2^2} - \frac{(\tau+t_0-t-t_1)^2}{2\sigma_1^2}$$

$$-\frac{1}{2\sigma_1^2\sigma_2^2}\left(\sigma_1^2(\tau^2-2\tau t_2+t_2^2)+\sigma_2^2(\tau^2+2\tau(t_0-t-t_1)+(t_0-t-t_1)^2)\right)$$

$$-\frac{1}{2\sigma_1^2\sigma_2^2}\left(\tau^2(\sigma_1^2+\sigma_2^2)+2\tau(-\sigma_1^2t_2+\sigma_2^2(t_0-t-t_1))+\sigma_1^2t_2^2+\sigma_2^2(t_0-t-t_1)^2\right) := -A\tau^2-2B\tau-C$$

avec

$$A = \frac{\sigma_1^2+\sigma_2^2}{2\sigma_1^2\sigma_2^2} > 0; B = \frac{-\sigma_1^2t_2+\sigma_2^2(t_0-t-t_1)}{2\sigma_1^2\sigma_2^2}; C = \frac{\sigma_1^2t_2^2+\sigma_2^2(t_0-t-t_1)^2}{2\sigma_1^2\sigma_2^2}$$

on a

$$I_2 = e^{\left(\frac{4B^2}{4A}-C\right)}\sqrt{\frac{\pi}{A}}$$

où

$$\frac{4B^2}{4A}-C = \frac{(-\sigma_1^2t_2+\sigma_2^2(t_0-t-t_1))^2}{2\sigma_1^2\sigma_2^2(\sigma_1^2+\sigma_2^2)} - \frac{\sigma_1^2t_2^2+\sigma_2^2(t_0-t-t_1)^2}{2\sigma_1^2\sigma_2^2}$$

on doit avoir un extrémal à t_{opt2} :

$$\frac{\partial}{\partial t}\left(\frac{4B^2}{4A}-C\right) = \frac{2(-\sigma_1^2t_2+\sigma_2^2(t_0-t-t_1))(-\sigma_2^2)}{2\sigma_1^2\sigma_2^2(\sigma_1^2+\sigma_2^2)} - \frac{2\sigma_2^2(t_0-t-t_1)(-1)}{2\sigma_1^2\sigma_2^2} = 0$$

$$\frac{2\sigma_1^2\sigma_2^2t_2-2\sigma_2^4(t_0-t-t_1)}{\sigma_1^2+\sigma_2^2} + 2\sigma_2^2(t_0-t-t_1) = 0$$

$$2\sigma_1^2\sigma_2^2t_2-2\sigma_2^4(t_0-t-t_1)+2(\sigma_1^2+\sigma_2^2)\sigma_2^2(t_0-t-t_1) = 0$$

$$2\sigma_1^2\sigma_2^2t_2+2\sigma_1^2\sigma_2^2(t_0-t-t_1) = 0$$

donc

$$t_{opt2} = t_0+t_2-t_1$$

Remarque 1.1.1: Intégrale de Gauss

Pour $a, b, c \in \mathbb{R}$ et $a > 0$, on a

$$\int_{-\infty}^{+\infty} e^{-ax^2+bx+c}\,dx = \int_{-\infty}^{+\infty} e^{-ax^2+bx+c-(\frac{b^2}{4a}+c)+(\frac{b^2}{4a}+c)}\,dx = e^{(\frac{b^2}{4a}+c)}\int_{-\infty}^{+\infty} e^{-(\sqrt{a}x-\frac{b}{2\sqrt{a}})^2}\,dx = e^{(\frac{b^2}{4a}+c)}\sqrt{\frac{\pi}{a}}$$

[10]

Pour l'autre onde réfléchise $x_3(t) \sim \mathcal{N}(t_3, \sigma_3^2) = -ax_1\left(\frac{\sigma_1}{\sigma_3}(t-t_3)+t_1\right) := -ax_1\left(b(t-t_3)+t_1\right)$:

$$\begin{aligned} I_3 = -ax_1\left(b(\tau-t_3)+t_1\right) * x_1^*(t_0-t) &= -a\int_{-\infty}^{+\infty} x_1\left(b(t-t_3)+t_1\right)x_1^*(t_0-(t-\tau))\,d\tau \\ &= -a\int_{-\infty}^{+\infty} e^{-\frac{(\tau-t_3)^2}{2\sigma_2^2}}\,e^{-\frac{((t_0-(t-\tau))-t_1)^2}{2\sigma_1^2}}\,d\tau \end{aligned}$$

on doit avoir un extrémal à t_{opt3} :

$$t_{opt3} = t_0+t_3-t_1$$

1.2 Simulation

Voici la fonction pour calculer l'intercorrélation :

Fonction pour calculer Intercorrélation - co_ene.m

Listing 1 – Fonction pour calculer Intercorrélation - co_ene.m

```matlab
function [k,c] = co_ene(x,y)
% function [k,c] = co_ene(x,y);
% This function calculates the energy intercorrelation of signals x and
% y in the frequency domain.
% Input:    x 1st signal to be analyzed
%           y 2nd signal to be analyzed
% Output:   k vector of indices for correlation calculation
%           (k = -length(x)+1, ....,0,...,length(x)-1)
%           c correlation calculated in k

% Transpose
x = x(:); y = y(:);

% The length of fft must be greater than 2N-1 to avoid aliasing
N = length(x);

c = real(ifft(fft(x,2*N).*conj(fft(y,2*N))));

% Keep only the lags we want and move negative lags before positive lags :
% i.e. [0 , ... , +lag_max , -lag_max , ... , 0] to [-lag_max , ... , +lag_max]
% i.e. move the second half of the data after the ifft to the front
% ref. https://github.com/ashao/matlab/blob/master/utilities/xcorr.m
c1 = c(N+2:2*N);
c2 = c(1:N);
clear c
c = [c1; c2];

k = -(N-1):(N-1); k = k(:);

end
```

pour la ligne

```
c = real(ifft(fft(x,2*N).*conj(fft(y,2*N))));
```

on utlise le théorème de Winner-Khintchine pour calculer rapidement $(2 * N)$ l'intercorrélation :

Théorème 1.2.1: Théorème de Winner-Khintchine

$$c_{xy}^{E}(\tau) = TF^{-1}\left[\tilde{x}(\nu)(\tilde{y}(\nu))^{*}\right]$$

Démonstration.

$$\begin{aligned}
c_{xy}^{E}(\tau) &= \int_{-\infty}^{+\infty} x(t)y^{*}(t-\tau)\ dt \\
&= \int_{-\infty}^{+\infty}\left[\int_{-\infty}^{\infty}\tilde{x}\left(\nu\right)e^{+\imath 2\pi\nu t}d\nu\int_{-\infty}^{\infty}e^{-\imath 2\pi\nu'\tau}\left(\tilde{y}\left(-\nu'\right)\right)^{*}e^{+\imath 2\pi\nu' t}d\nu'\right]dt \\
&= \int_{-\infty}^{+\infty}\int_{-\infty}^{\infty}\tilde{x}\left(\nu\right)\left(\tilde{y}\left(-\nu'\right)\right)^{*}e^{-\imath 2\pi\nu'\tau}\left[\int_{-\infty}^{\infty}e^{+\imath 2\pi(\nu+\nu')t}dt\right]d\nu' d\nu \\
&= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\tilde{x}\left(\nu\right)\left(\tilde{y}\left(-\nu'\right)\right)^{*}e^{-\imath 2\pi\nu'\tau}\delta(\nu+\nu')\ d\nu' d\nu \\
&= \int_{-\infty}^{+\infty}\tilde{x}\left(\nu\right)\left(\tilde{y}\left(\nu\right)\right)^{*}e^{+\imath 2\pi\nu\tau}\ d\nu \\
&= TF^{-1}\left[\tilde{x}(\nu)(\tilde{y}(\nu))^{*}\right]
\end{aligned}$$

□

Remarque 1.2.1

—

$$\begin{aligned}&\int_{-\infty}^{+\infty} y^*(t-\tau)e^{-\imath 2\pi\nu t}\,dt\\&=\int_{-\infty}^{+\infty} y^*(t')e^{-\imath 2\pi\nu(t'+\tau)}\,dt' = e^{-\imath 2\pi\nu\tau}\int_{-\infty}^{+\infty} y^*(t')e^{-\imath 2\pi\nu t'}\,dt'\\&= e^{-\imath 2\pi\nu\tau}\int_{-\infty}^{+\infty}\left(y(t')e^{-\imath 2\pi\nu t'}\right)^*\,dt' = e^{-\imath 2\pi\nu\tau}\left(\int_{-\infty}^{+\infty} y(t')e^{-\imath 2\pi\nu t'}\right)^*\,dt'\\&= e^{-\imath 2\pi\nu\tau}\left(\tilde{y}\left(-\nu'\right)\right)^*\end{aligned}$$

—

$$\int_{-\infty}^{+\infty} f(u)\delta(u+v)\,du = f(-\nu)$$

(Christophe Vignat, le 28/06/2024)

```
c1 = c(N+2:2*N); c2 = c(1:N); clear c; c = [c1; c2];
```

Car la fonction de (auto)corrélation est un signal / vecteur qui contient $2N-1$ échantillons. Avec le calcul de la TF et TF inverse on a $2N$ échantillons. On a donc fait un bourrage de zéros sur 1 point / échantillon, celui que l'on ne retient pas et qui est nul. Lors de la TF inverse on a la corrélation qui a été périodisée et calculée entre entre 0 et 2*N et on la considère entre -N+1 et N-1 d'où le passage en faisant intervenir c1 et c2.

1.2.1 Détection de la position de l'onde incidente

On ne sait pas le centre de l'onde incidente $x_1(t)$ dans le signal + noise exactement, puisque nous sommes sûrs qu'il y a $x_1(t)$ dans le signal + noise, on applique un filtre adapté $h(t)$ soit la même forme que $x_1(t)$:

$$h_1(t) = e^{-a*(t-t')^2}, \quad \text{avec } a = 3*10^{20};\ t' = 0.3*10^{-9}$$

En sortie du filtre adapté (cf. FIGURE 1.2.1 : output of the adaptive filtre (zoom)), on calule le décalage hozitontal Δt_1 entre le trait rouge (soit t_0) et le maximun. Alors on déplace le filtre adaté vers la droite de Δt_1 :

$$h_2(t) = e^{-a*(t-t'-\Delta t_1)^2}, \quad \text{avec } a = 3*10^{20};\ t' = 0.3*10^{-9}$$

maintenant, on refait le filtrage avec

$$x(t) * h_2^*(t_0 - t)$$

on doit trouver un maximum à $t = t_0$ car on nous sommes sûrs qu'il existe l'onde incidente $x_1(t)$ (cf. FIGURE 1.2.2). Le centre du filtre adapté $h_2(t)$, soit $t' + \Delta t$ aussi implique le centre de l'onde incidente $x_1(t)$.

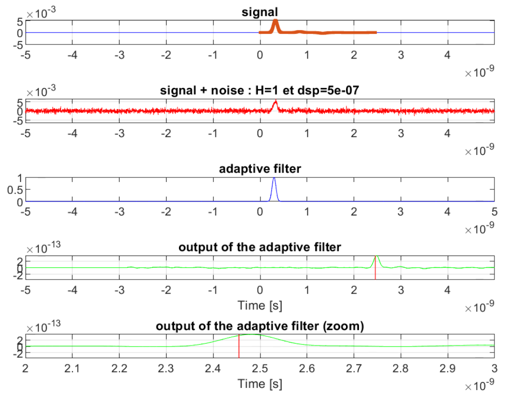


FIGURE 1.2.1 – Filtre adapté $h_1(t)$ | $dsp = 5*10^{-7}$

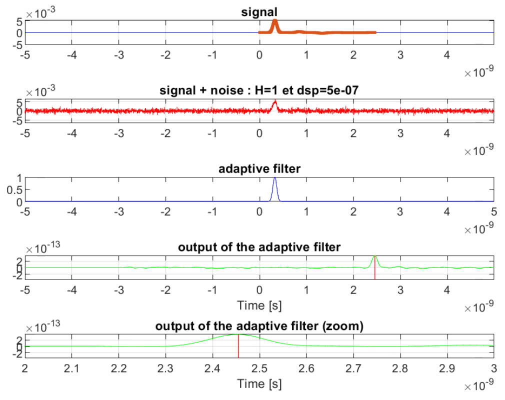


FIGURE 1.2.2 – Filtre adapté $h_2(t)$ | $dsp = 5*10^{-7}$

A partir de la postion $t' + \Delta t_1$, on peut remplacer le signal + noise par des 0 dans l'intervalle $[0, t' + \Delta t_1 + 3*\sigma_1]$ par exemple. Puis on va étulier si le signal $x(t)$ comporte des défauts ou pas.

1.2.2 Détection de la présence de l'onde réfléchie

On remplace le signal + noise avant $t = t' + \Delta t_1 + 3*\sigma_1$ par des 0 :

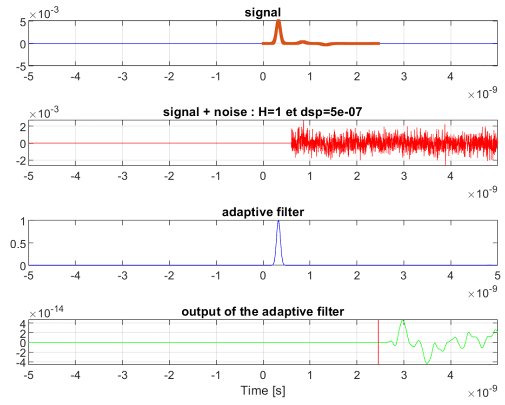


FIGURE 1.2.3 – Filtre adapté $h_2(t)$ | $dsp = 5 * 10^{-7}$

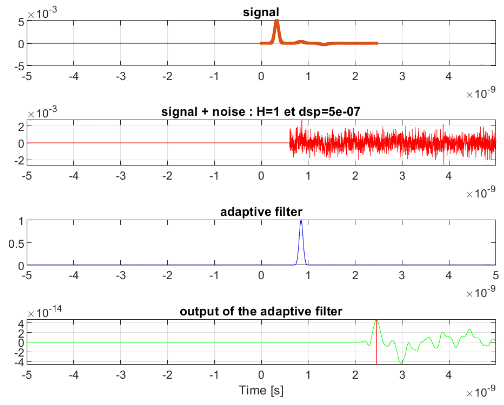


FIGURE 1.2.4 – Filtre adapté $h_3(t)$ | $dsp = 5 * 10^{-7}$

En sortie du filtre adapté (cf. FIGURE 1.2.3 : output of the adaptive filtre), on calule le décalage hozitontal Δt_2 entre le trait rouge (soit t_0) et le maximun. On déplace le filtre adaté encore une fois vers la droite de Δt_2 :

$$h_3(t) = e^{-a*(t-t'-\Delta t_1-\Delta t_2)^2}, \qquad \text{avec } a = 3*10^{20};\ t' = 0.3*10^{-9}$$

on refait le filtrage :

$$x(t) * h_3^*(t_0 - t)$$

Alors le centre du filtre adapté $h_3(t)$ (gaussian) doit être située à t_2, sinon cette méthode a atteint ses limites :

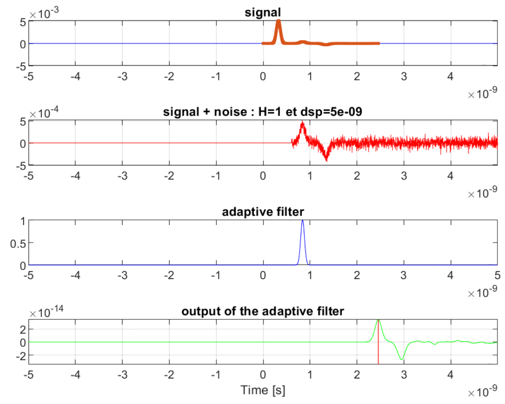


FIGURE 1.2.5 – Filtre adapté $h_3(t)$ | $dsp = 5 * 10^{-9}$

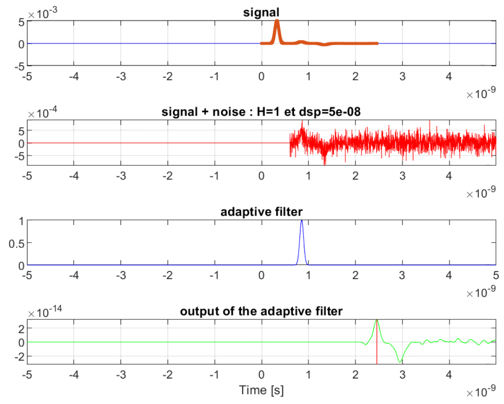


FIGURE 1.2.6 – Filtre adapté $h_3(t)$ | $dsp = 5 * 10^{-8}$

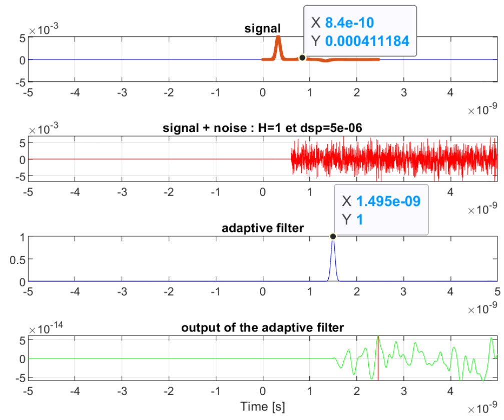


FIGURE 1.2.7 – Filtre adapté $h_3(t)$ | $dsp = 5 * 10^{-6}$

Soit cf. FIGURE 1.2.7, on voit que la position $1.495 * 10^{-9}$ impliquant par le centre du filtre adapté $h_3(t)$ ne correspond pas $t_2 = 8.4 * 10^{-10}$, cette méthode a atteint ses limites. Et pour les signaux à détecter les défauts, on peut exécuter ce programme plusieurs fois.Si la position du filtre adapté reste inchangée, cela signifie qu'il y a un défaut dans le câble. Mais si la postion du filtre adapté change alors on ne peut pas conclure. Voici un exemple d'utilisation cf. $\boxed{\textit{Listing } 4}$.

2 Transformée de fourier à fenêtre glissante (STFT)

2.1 Fenêtrage

Comme miantenant on vaudrait traiter des siganux de durée finie et de spectre bornée, alors une idée est de faire une multiplication $(\cdot)$ d'une fontion porte $x_1(t)$ en temps soit un produit de convolution $(*)$ avec la fonction sinus cardinal $\tilde{x}_1(\nu)$ dans le domaine fréquentiel, cf. la Figure dans l'Introduction.

$$x(t) \cdot x_1(t) \xrightarrow{TF} \tilde{x}(\nu) * \tilde{x}_1(\nu)$$

Mais comme le sinus cardinal s'amortit très lentement avec des lobes importantes au voisinage de l'origine, alors on propose les fonctions suivantes (fenêtres) qui sont concentrées autour de l'origine au lieu du sinus cardinal :

Exemple 2.1.1: Fenêtre triangulaire

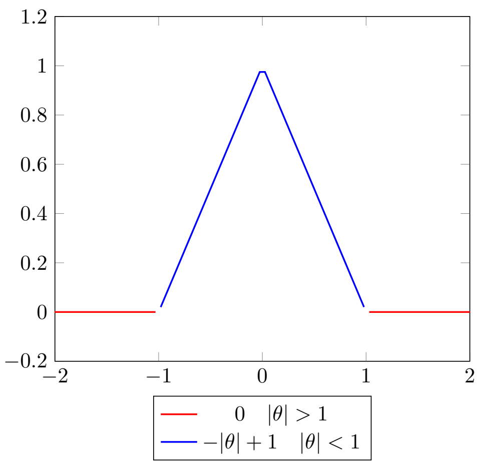
En temps :

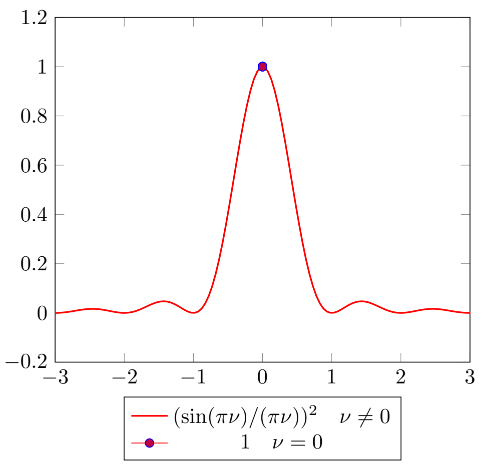
$$h(\theta) = \begin{cases} -\frac{1}{A}|\theta| + 1 & \text{Si } \theta \in [-A, A] \\ 0 & \text{Sinon} \end{cases}$$

En fréquence :

$$\tilde{h}(\nu) = A \left(\frac{\sin(\pi A \nu)}{\pi A \nu} \right)^2$$

Pour illustrer, on prend un exemple avec $A = 1$.





Exemple 2.1.2: Fenêtres de Hann et de Hamming

En temps :

$$h(\theta) = \begin{cases} \alpha + (1-\alpha)\cos\left(\frac{2\pi}{A}\theta\right) & \text{Si } \theta \in [-\frac{A}{2}, \frac{A}{2}] \\ 0 & \text{Sinon} \end{cases}$$

En fréquence :

Pour calculer la transformée de Fourier (TF) de la fenêtre de Hann ou de Hamming, on d'abord calcule la TF du cosinus :

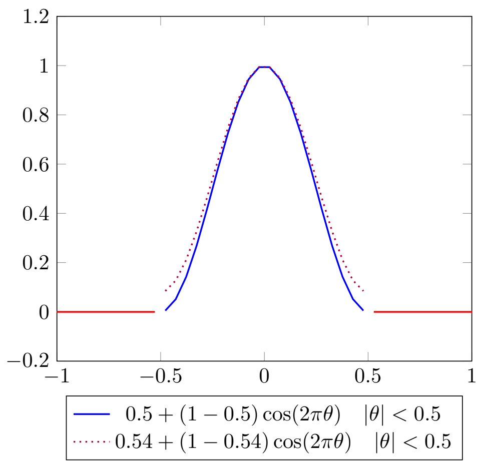
$$\int_{-\frac{A}{2}}^{+\frac{A}{2}} \cos\left(\frac{2\pi}{A}\theta\right) e^{-\imath 2\pi\nu\theta}\, d\theta = \int_{-\frac{A}{2}}^{+\frac{A}{2}} \frac{1}{2}\left[e^{+\imath\frac{2\pi}{A}\theta} + e^{-\imath\frac{2\pi}{A}\theta}\right] e^{-\imath 2\pi\nu\theta}\, d\theta = \frac{A}{2}\left[sinc(\pi - \pi A\nu) + sinc(\pi + \pi A\nu)\right]$$

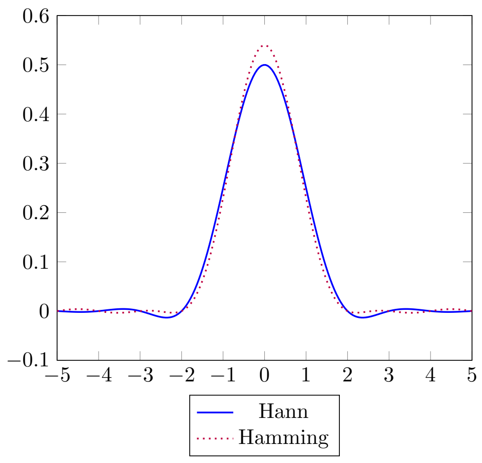
soit

$$\tilde{h}(\nu) = \alpha A sinc(\pi A \nu) + (1-\alpha)\frac{A}{2}\left[sinc(\pi - \pi A\nu) + sinc(\pi + \pi A\nu)\right]$$

Avec $\alpha = 0.5$ on a la fenêtre de Hann ; $\alpha = 0.54$ on a la fenêtre de Hamming.

Pour illustrer, on prend un exemple avec $A = 1$.





Exemple 2.1.3: Fenêtre de Gauss

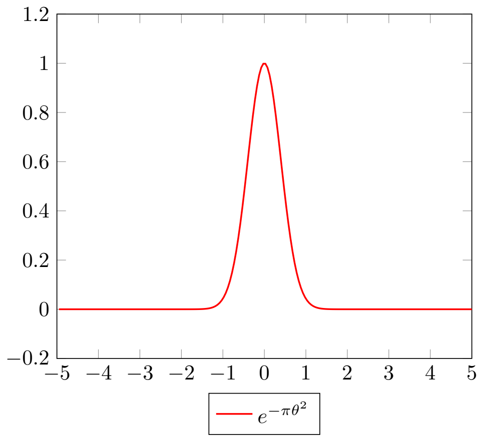
En temps :

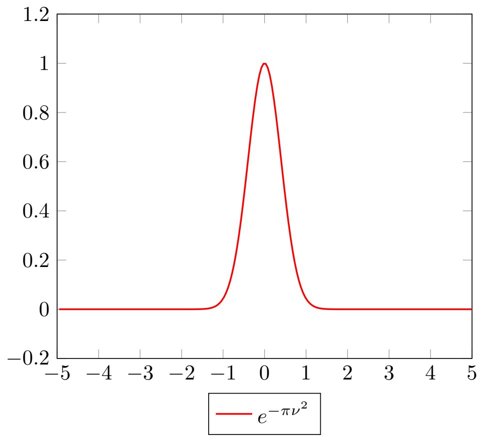
$$h(\theta) = A\ e^{-\alpha\theta^2} \quad \text{avec } \alpha > 0, A > 0$$

En fréquence :

$$\tilde{h}(\nu) = A\sqrt{\frac{\pi}{\alpha}}\ e^{-\dfrac{\pi^2}{\alpha}\nu^2}$$

Pour illustrer, on prend un exemple avec $A = 1, \alpha = \pi$.





Définition 2.1.1: Transformée de Fourier à fenêtre glissante (STFT)

On glisse une fenêtre $h(\theta)$ devant le signal $x(\theta)$, on va obtenir une famille de coeffient $G(t,\nu)$ avec t,ν réels, soit :

$$STFT_x(t,\nu) = G(t,\nu) = \int_{-\infty}^{+\infty} x(\theta)h^*(\theta - t)e^{-\imath 2\pi\nu\theta}d\theta \tag{2.1.1}$$

qui remplace les valeurs de $\tilde{x}(\nu)$. On s'appelle cette application

$$x \to G(t,\nu)$$

la **transformée de Fourier à fenêtre glissante**.

On voit que ici la fonction $h^*(\theta - t)$ décalée dans le temps est une fenêtre glissante. Le terme $h^*(\theta - t)e^{-\imath 2\pi\nu\theta}$ peut être considéré comme la réponse impulsionnelle d'un filtre sélectif en fréquence à ν. Soit on peut considérer la transformée de Fourier à fenêtre glissante commeune une série de filtres similaires, décalés dans le domaine des fréquences.

2.2 La transformée de Gabor et la formule de reconstruction

Si on a les coefficients des $G(t,\nu)$ pour toutes les valeurs réelles de t et ν, on peut reconstruire notre signal d'après Gabor :

Théorème 2.2.1: La formule de reconstruction

Soit $h \in \mathcal{L}^1(\mathbb{R}) \cap \mathcal{L}^2(\mathbb{R})$ une fenêtre telle que $|\tilde{h}|$ soit une fonction paire et $\|h\|_2 = 1$. Par exemple, Gabor a choisi une fenêtre de Gauss :

$$h(\theta) = e^{-\alpha\theta^2} \quad (\alpha > 0).$$

Pour tout signal $x \in \mathcal{L}^2(\mathbb{R})$, $(t,\nu) \in \mathbb{R} \times \mathbb{R}$, Gabor a proposé la transformation suivante :

$$G(t,\nu) = \int_{-\infty}^{+\infty} x(\theta) h^*(\theta - t) e^{-\imath 2\pi\nu\theta} d\theta$$

Alors, on a :

- $*$ Conservation de l'énergie :

$$\iint_{\mathbb{R}^2} |G(t,\nu)|^2 \, d\nu dt = \int_{-\infty}^{+\infty} |x(t)|^2 dt. \tag{2.2.1}$$

- $*$ La formule de reconstruction :

$$x(t) = \iint_{\mathbb{R}^2} G(t,\nu) h(\theta - t) e^{+\imath 2\pi\nu\theta} \, d\nu dt \tag{2.2.2}$$

au sens suivant, si

$$\hat{x}_A(t) = \int_{t\in\mathbb{R}} \int_{|\nu|\leq A} G(t,\nu) h(\theta - t) e^{+\imath 2\pi\nu\theta} \, d\nu dt$$

alors $\hat{x}_A(t) \to x$ dans $\mathcal{L}^2(\mathbb{R})$ quand $A \to +\infty$.

2.3 Principe d'incertitude (Heisenberg-Gabor)

Soit $x \in \mathcal{L}^1(\mathbb{R})$, on rappelle $\tilde{x}$ la transformée de Fourier de x la foncntion de $\mathbb{R}$ dans $\mathbb{C}$ définie par :

$$\tilde{x}(\nu) = \int_{-\infty}^{\infty} x(t) e^{-\imath 2\pi\nu t} dt$$

Le théorème suivant donne l'principe d'incertitude de Heisenberg pour la transformée de Fourier sur $\mathbb{R}$:

Théorème 2.3.1: Principe d'incertitude de Heisenberg pour la transformée de Fourier

Soit $x : \mathbb{R} \to \mathbb{C}$ une fonction de $\mathcal{C}^1(\mathbb{R})$ telle que x, x' et $x \mapsto xf(x) \in \mathcal{L}^2(\mathbb{R})$. Alors

$$\left(\int_{-\infty}^{+\infty} t^2 |x(t)|^2 \, dt\right)^{\frac{1}{2}} \left(\int_{-\infty}^{+\infty} \nu^2 |\tilde{x}(\nu)|^2 \, d\nu\right)^{\frac{1}{2}} \geq \frac{\int_{-\infty}^{+\infty} |x(t)|^2 dt}{4\pi} \tag{2.3.1}$$

On prend l'égalité si $x(t)$ est un fonction gausienne :

$$x(t) = A\, e^{-\alpha t^2}$$

Et pour la transformée de Fourier à fenêtre glissante, où $h(\theta)$ est la fenêtre, souvent une fenêtre de Hann (on va utliser dans la subsection suivante pour faire la simulation) ou une fenêtre gaussienne (Gabor) centrée autour de zéro, on rappelle $G(t,\nu) : \mathbb{R} \times \mathbb{R} \to \mathbb{C}$ telle que :

$$G(t,\nu) = \int_{-\infty}^{+\infty} x(\theta) h^*(\theta - t) e^{-\imath 2\pi\nu\theta} d\theta$$

Théorème 2.3.2: Principe d'incertitude de Heisenberg pour la transformée de Gabor

Soit $h \in \mathcal{L}^1(\mathbb{R}) \cap \mathcal{L}^2(\mathbb{R})$ une fenêtre, $x \in \mathcal{L}^2$. Alors

$$\left(\int_{-\infty}^{+\infty} t^2 |x(t)|^2 \, dt\right)^{\frac{1}{2}} \left(\int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} \nu^2 |G(t,\nu)|^2 \, dt d\nu\right)^{\frac{1}{2}} \geq \frac{(\int_{-\infty}^{+\infty} |h(\theta)|^2 d\theta)(\int_{-\infty}^{+\infty} |x(t)|^2 dt)^2}{4\pi} \tag{2.3.2}$$

Avec une démonstration proposée par Wilzokowski. [11] Le produit des résolutions temporelle et fréquentielle est minimal si $h(\theta)$ est un fonction gausienne.

2.4 Simulation

2.4.1 Revenons à l'exemple précédent : dsp = 5e-7

Nous revenons à l'exemple précédent où dsp $= 5 * 10^{-7}$, la réponse du filtre adapté (cf. FIGURE 1.2.4, en vert) montre que à priori, le signal + noise possède le signal de l'onde réfléchie. Alors en utilsant la STFT, on pourrait reconstruire ce signal réfléchi. Le signal à analyser cf. **subplot(3,1,3)** FIGURE 2.4.2 (dont onde incidente a soustraite après avoir appliqué le filtre adapté), idéalement, après le traitement, on doit obtenir cf. **subplot(3,1,2)** FIGURE 2.4.2.

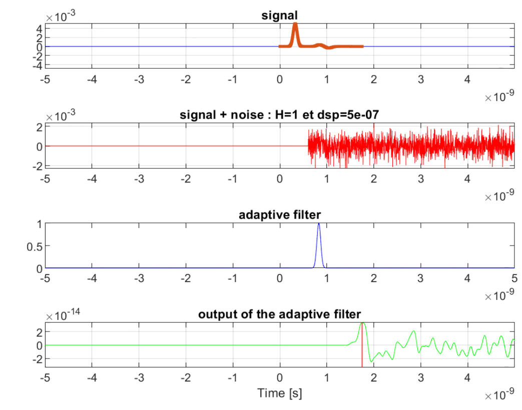


FIGURE 2.4.1 – Filtre adapté $h_3(t)$ | $dsp = 5 * 10^{-7}$

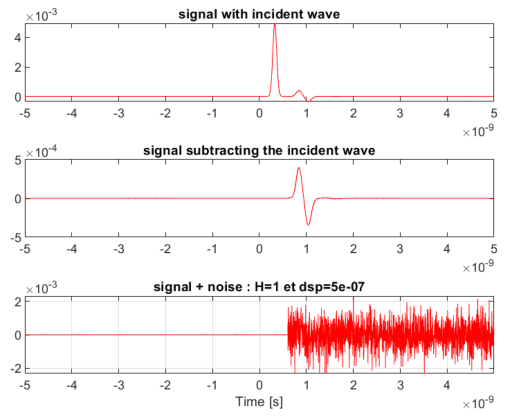


FIGURE 2.4.2 – Signal + noise | $dsp = 5 * 10^{-7}$

La commande

```
[so,fo,to] = stft(s_cut,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512); So = abs(so);
```

donne le moudule des coefficients de STFT du signal (voir la première ligne des deux figures ci-dessous). De même, la commande

```
[s,f,t] = stft(x,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512); S = abs(s);
```

donne le moudule des coefficients de STFT du signal + noise (voir la deuxième ligne des deux figures ci-dessous).

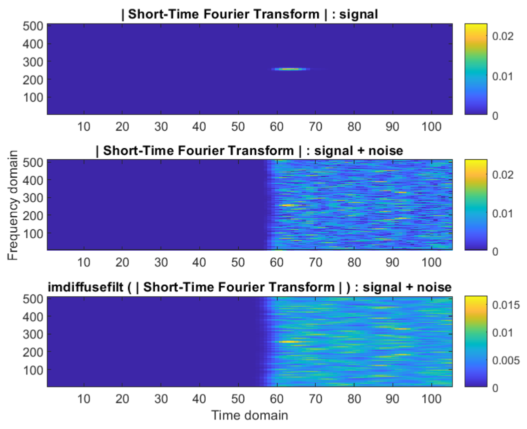


FIGURE 2.4.3 – STFT | $dsp = 5 * 10^{-7}$

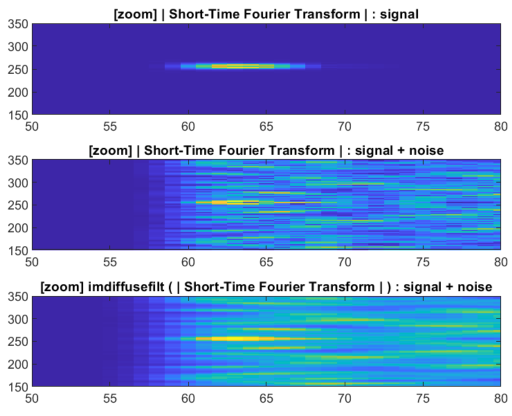


FIGURE 2.4.4 – [ZOOM] STFT | $dsp = 5 * 10^{-7}$

Le but est d'itentifier la partie du signal en la partie de signa + signal. Ceci peut être traduit en un problème de traitement d'image : on propose de faire un filtrage des images par diffusion anisotropique qui facilite la sélection ultérieure des régions / coutours (voir la troisième ligne des deux figures ci-dessus) :

```
Sd = imdiffusefilt(S);
```

On va analyser cette matrice (soit `Sd`) détecter les régions / contours.

Remarque 2.4.1

- On a choisi `FFTLength=512`, soit après STFT, l'indice de la fréquence varie entre 0 et 512. Et le signal d'onde réflichie doit se trouver au milieux (512 /2 = 256), et idéalement, être symétrique par rapport à 256.
- On n'arrive pas à localiser le signal à la fois en temps et en fréquence a cause du principe d'incertitude. Pour résoudre ce problème, une solution est de faire la transfomée en ondelettes qui introduit une fenêtre donc la taille varie avec la fréquence.
- Nous pourrions probablement utiliser 2D-CNN à partir d'ici.

Alors pour la itendification des régions / coutours, une simple méthode est d'étudier les limites au-dessus et en dessous (soit faire varier les lignes). On choisit le pixel plus intense et de plus son région traversant la ligne hozitontale de 256 , si le gradient diminue toujours alors on copie la valeur instantanée de `s(i, col_opt)` vers une matrice initialisé par des zéros pour stocker les coefficients de STFT :

```
Coef(i, col_opt) = s(i, col_opt);
```

Finalement, on obtient deux bornes : `row_down` et `row_up`, nous descendons à gauche et à droite (soit faire varier les colonnes) pour chaque pixel en ce moment.

```
for R = row_down : row_up
```

on fait les copies :

```
Coef(R, col_opt+j) = s(R, col_opt+j);
```

Une fois que on a obtenu le résultat, on fait iSTFT pour reconstruire le signal réfléchi :

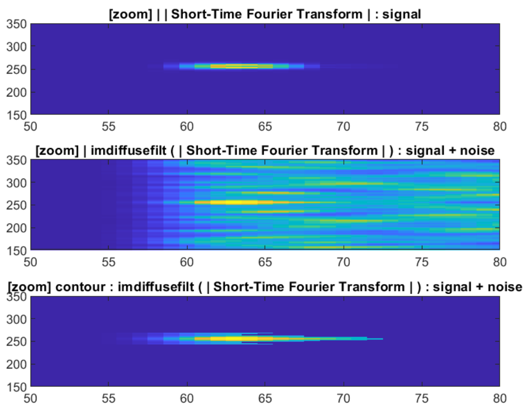


FIGURE 2.4.5 – [ZOOM] La région sélectionnée | $dsp = 5 * 10^{-7}$

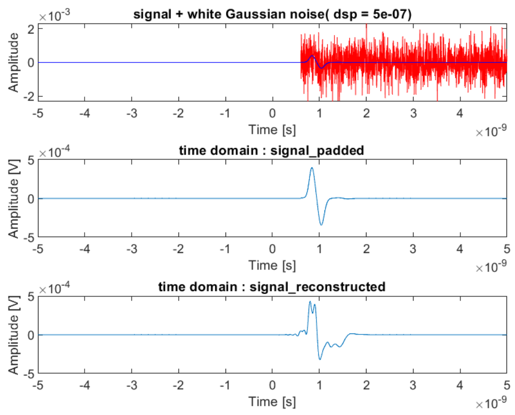


FIGURE 2.4.6 – iSTFT | $dsp = 5 * 10^{-7}$

2.4.2 Autre exemple : dsp = 5e-8

Si nous réduisons le niveau de bruit, il est évident que la reconstruction serait un signal plus précis :

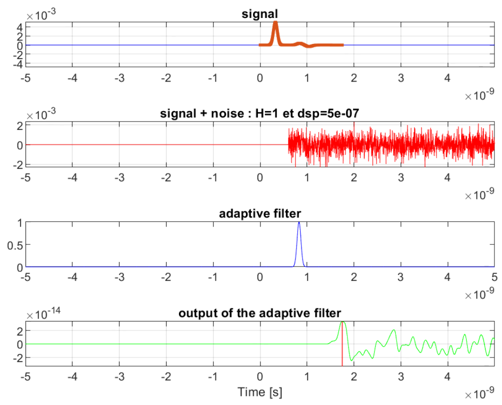


FIGURE 2.4.7 – Filtre adapté $h_3(t)$ | $dsp = 5 * 10^{-8}$

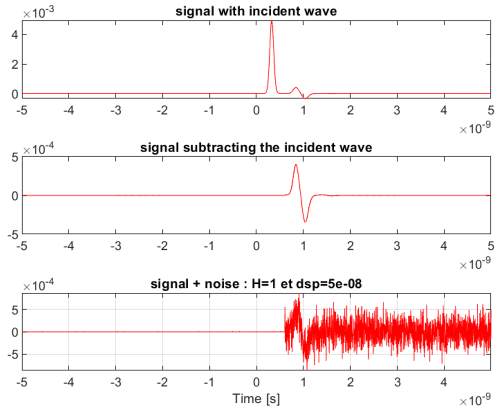


FIGURE 2.4.8 – Signal + noise | $dsp = 5 * 10^{-8}$

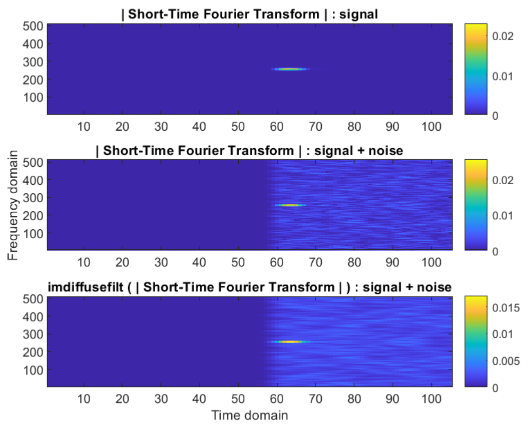


FIGURE 2.4.9 – STFT | $dsp = 5 * 10^{-8}$

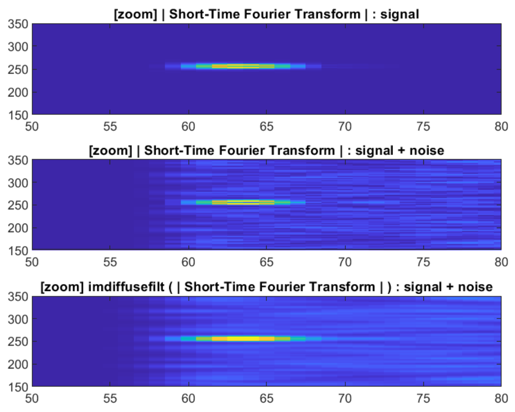


FIGURE 2.4.10 – [ZOOM] STFT | $dsp = 5 * 10^{-8}$

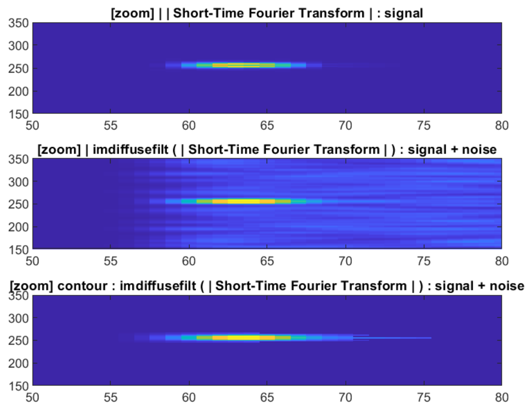


FIGURE 2.4.11 – [ZOOM] La région sélectionnée | $dsp = 5 * 10^{-8}$

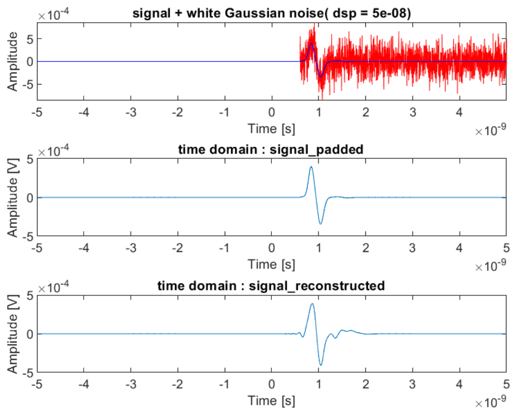


FIGURE 2.4.12 – iSTFT | $dsp = 5 * 10^{-8}$

Pour le code, voir Listing 5.

3 Convolutionnal Neural Networks in One Dimension

3.1 1D convolution for neural networks

3.1.1 Convolution

Convolutional operations help us extract the features we are interested in from the signal. Convolution is the process of taking a kernel and performing a sliding dot product with the signal. The input signal is what we want to classify or to extract features. There's is an illustration of 1D convolution operation. For example, we have a signal **x** like

1	0	0	0	-1	0	0	0	2

and the kernel **w** :

3	1	2

The first step is to flip the kenel from left to right and we align the center of **w** with the first element of **x** then we add some zeros to both sides of the signal which align the signal and kernel :

0	1	0	0	0	-1	0	0	0	2	0

2	1	3

We add each dot product which gives :

$$0 * 2 + 1 * 1 + 0 * 3 = 1$$

The next step is to slide the kernel one step to the right and do the first step again :

$$1 * 2 + 0 * 1 + 0 * 3 = 2$$

Finally the result of the convolution **y** is :

1	2	0	-3	-1	-2	0	6	2

We note the signal with m elements and elements with indexes less than 0 or greater than $m-1$ are treated as 0 during the convolution :

$$\mathbf{x} = [x_0, x_1, \ldots, x_{m-1}]$$

so the ouput **y** which has the same length as **x** :

$$\mathbf{y} = [y_0, y_1, \ldots, y_{m-1}]$$

rather than define the kernel as $\mathbf{w} = [w_0, w_1, ..., w_{n-1}]$, we make the kernel with odd elements just making some of the notation a little more convenient later :

$$\mathbf{w} = [w_{-p}, \ldots, w_0, \ldots, w_p]$$

So we have, for exameple :

$$y_0 = x_{-p}w_p + \ldots + x_{-1}w_1 + x_0w_0 + x_1w_{-1} + \ldots + x_pw_{-p} = \sum_{k=-p}^{p} x_{-k}w_k$$

$$y_1 = x_{-p+1}w_p + \ldots + x_{-1}w_2 + x_0w_1 + x_1w_0 + \ldots + x_{p+1}w_{-p} = \sum_{k=-p}^{p} x_{1-k}w_k$$

$$\vdots$$

$$y_p = x_0w_p + x_1w_{p-1} + \ldots + x_pw_0 + \ldots + x_{2p}w_{-p} = \sum_{k=-p}^{p} x_{p-k}w_k$$

$$\vdots$$

$$y_{m-1} = x_{-p+m-1}w_p + x_{-p+m}w_{p-1} + \ldots + x_{m-1}w_0 + x_mw_{-1} + \ldots + x_{p+m-1}w_{-p} = \sum_{k=-p}^{p} x_{m-1-k}w_k$$

more further, for any element y_j, we have

$$y_j = \sum_{k=-p}^{p} x_{j-k}w_k \tag{3.1.1}$$

3.1.2 Backpropagation

Neural networks learn their parameters by the backpropagation which needs the loss function to be differentiable. It means if the layer in a neural networks is differentiable, we could stack one after another meanwhile the loss function will be passed down through the layers. The backpropagation is that according to the partial derivative of the loss with respect to the inputs $\mathbf{x}$ (or $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$), we want to propagate that back to the partial derivative of the loss with respect to the inputs $\mathbf{y}$ (or $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$) by the chain rule :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

In our case, the input $\mathbf{x}$ is a linear vector, so for each element we have :

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial x_i}$$

And the output $\mathbf{y}$ is also a linear vector, so we should add each result up :

$$\frac{\partial \mathcal{L}}{\partial x_i} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{3.1.2}$$

Input gradient The input gradient is passed back to the previous layer of the network and keeps passing upwards.

According to equation (3.1.1), we have

$$\frac{\partial y_{i+k}}{\partial x_i} = w_k$$

thus we could plug this back in to the equation (3.1.2) as :

$$\begin{aligned} \boxed{\frac{\partial \mathcal{L}}{\partial x_i}} &= \sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i+k}} \frac{\partial y_{i+k}}{\partial x_i} \\ &= \boxed{\sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i+k}} w_k} \end{aligned}$$

or if we invert the vector $\mathbf{w}$ from left to right centered at zero, denoted $\overleftarrow{\mathbf{w}}$, with each element w_{-k} wrote as $\overleftarrow{w_k}$ (or $w_{-k} = \overleftarrow{w_k}$) and y_{i+k} becomes to y_{i-k}

$$\frac{\partial \mathcal{L}}{\partial x_i} = \sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i-k}} w_{-k} = \sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i-k}} \overleftarrow{w_k}$$

where $i \in [0, m-1]$. And for $i \notin [0, m-1]$ we assume that :

$$\frac{\partial \mathcal{L}}{\partial x_i} = 0$$

In this way we can regard the input gradient as the convolution between the output gradient and the reversed kernel $\overleftarrow{\mathbf{w}}$:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} * \overleftarrow{\mathbf{w}}}$$

Weight gradient Except for the input gradient, we should also calculate the weight gradient which helps to indicate the direction and the value to adjust in the kernel to decrease the loss function as low as possible by using the gradient descent. With the chain rule :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{w}}$$

In our case, the input $\mathbf{w}$ is a linear vector, so for each element we have :

$$\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w_k}$$

And the weight $\mathbf{y}$ is also a linear vector, so we should add each result up :

$$\frac{\partial \mathcal{L}}{\partial w_k} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial w_k}$$

According to equation (3.1.1), we have

$$\frac{\partial y_j}{\partial w_k} = x_{j-k}$$

thus

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_k}} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} \frac{\partial y_j}{\partial w_k}$$
$$= \boxed{\sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} x_{j-k}}$$

Also, if we invert the vector $\mathbf{x}$ from left to right centered at zero, denote $\overleftarrow{\mathbf{x}}$, with each element x_{j-k} wrote as $\overleftarrow{x}_{k-j}$ (or $x_{j-k} = \overleftarrow{x}_{k-j}$) :

$$\frac{\partial \mathcal{L}}{\partial w_k} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} x_{j-k} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} \overleftarrow{x}_{k-j}$$

where $k \in [-p, p]$. And for $k \notin [-p, p]$ we assume that :

$$\frac{\partial \mathcal{L}}{\partial w_k} = 0$$

In this way we can regard the weight gradient as the convolution between the output gradient and the reversed kernel $\overleftarrow{\mathbf{x}}$

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} * \overleftarrow{\mathbf{x}}}$$

3.1.3 Multiple channels

However, the input one-dimensional data may come in the form of multiple parallel channels. Like the Electroencephalography (EEG) can have up to 256 channels at the same time. To deal with this, we handle each channel independently : for each channel, we creat and learn its own kernel.

Mathematically, about the convolution, we just add a summation over all the channels :

$$y_j = \sum_{n=0}^{n_c-1} \sum_{k=-p}^{p} x_{c,j-k} w_{c,k}$$

We calulate the input gradient for each channel :

$$\frac{\partial \mathcal{L}}{\partial x_{c,i}} = \sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i+k}} w_{c,k}$$

also for the weight gradient :

$$\frac{\partial \mathcal{L}}{\partial w_{c,k}} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} x_{c,j-k}$$

where c is the channel index, n_c is the number of channels.

3.1.4 Bias

The last thing is that we add a learned bias (b) into the output of the convolution. The bias is a constant value which provides a constant offset to the given output :

$$y_j = b_j + \sum_{n=0}^{n_c-1} \sum_{k=-p}^{p} x_{c,j-k} w_{c,k}$$

Since b has no effect on input gradient

$$\frac{\partial \mathcal{L}}{\partial x_{c,i}} = \sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i+k}} w_{c,k}$$

nor on weight gradient :

$$\frac{\partial \mathcal{L}}{\partial w_{c,k}} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} x_{c,j-k}$$

Finally the bias gradient :

$$\frac{\partial \mathcal{L}}{\partial b_j} = \frac{\partial \mathcal{L}}{\partial y_j}$$

3.2 Coding a convolution block

3.2.1 Convolution in Python

We could use the function `conv(signal, kernel, mode = ' ')` to calculate the convolution. There are three options for the mode. We take `signal` = $\boxed{1}\boxed{2}\boxed{0}\boxed{3}$, `kernel` = $\boxed{1}\boxed{2}$ for example :

— `mode='valid'`

$$\begin{array}{cccccccl}
 & \boxed{1} & \boxed{2} & \boxed{0} & \boxed{3} & & \quad & \text{(signal)} \\
 & \boxed{2} & \boxed{1} & & & & & \text{(beginning)} \\
 & & & \boxed{2} & \boxed{1} & & & \text{(ending)} \\
 & \boxed{4} & \boxed{4} & \boxed{3} & & & & \text{(result)}
\end{array}$$

— `mode='same'`

$$\begin{array}{cccccccl}
 & \boxed{1} & \boxed{2} & \boxed{0} & \boxed{3} & & \quad & \text{(signal)} \\
 & \boxed{2} & \boxed{1} & & & & & \text{(beginning)} \\
 & & & & \boxed{2} & \boxed{1} & & \text{(ending)} \\
 & \boxed{4} & \boxed{4} & \boxed{3} & \boxed{6} & & & \text{(result)}
\end{array}$$

— `mode='full'`

$$\begin{array}{cccccccl}
 & \boxed{1} & \boxed{2} & \boxed{0} & \boxed{3} & & \quad & \text{(signal)} \\
\boxed{2} & \boxed{1} & & & & & & \text{(beginning)} \\
 & & & & \boxed{2} & \boxed{1} & & \text{(ending)} \\
 & \boxed{1} & \boxed{4} & \boxed{4} & \boxed{3} & \boxed{6} & & \text{(result)}
\end{array}$$

One implementation could be like :

Listing 6 Convolution in Python - convolve_1d.py

Firstly, we flip the kernel from left to right :

```
rev_kernel = kernel[::-1].copy()
```

Listing 7 Cross-correlation in Python - xcorr_1d.py

Then we calculate the cross-correlation starting with the beginning of the kernel and signal :

```
result[i] = np.dot(signal[i: i + n_ker], kernel)
```

This will give the same result as `conv(signal, kernel, mode = 'valid')` .

3.2.2 Initialize the convolution block

Initializer We need an initializer for deep neural networks. We could use the `LSUV` (Layer-Sequential Unit-Variance) method which provides a faster convergence, avoids gradient issues (vanishing and exploding gradients) and does not require extra hyperparameters :

```
initializer=LSUV(),
```

We should input the size of the kernel, we mention again that this should be an odd number in our case to facilitate the implementation as we told when we proved the formula, for example :

```
kernel_size=3,
```

We need to specify a number of kernels to learn, we can choose arbitrarily :

```
n_kernels=5,
```

In the convolution block, we also need to choose an optimizer, like the stochastic gradient descent with momentum :

```
optimizer=Momentum(),
```

means

$$v_t = \gamma v_{t-1} + \eta \frac{\partial}{\partial \theta} \mathcal{L}(\theta_{t-1})$$
$$\theta_t = \theta_{t-1} - v_t$$

where v_t the momentum at iteration t, θ_t the parameter at iteration t, γ the momentum parameter controls influences of past gradient (around 0.9 typically), η the learning rate.

We assign `None` (initialized value) :

```
self.n_channels = None; self.n_inputs = None; self.n_outputs = None
```

`self.weights = None;`

cause we want the convolution block determinate their values itself according to the input at the first iteration. The input will infer the number of channels, the number of inputs and so on.

We should optimize the weight and bias separately cause the weights act on individual inputs, while the biases act on combinations of outputs. Normally, we should create two different optimizers for each, like :

```
self.weight_optimizer = deepcopy(optimizer)
self.bias_optimizer = deepcopy(optimizer)
```

However, the fact is that since we have already enough gradient signal to make adjustments if we need to adjust the bias, we just create one optimizer for the weight :

```
self.weight_optimizer = deepcopy(optimizer)
```

Finally we initialize the values for the forward propagation and back-propagation :

```
self.forward_in = None; self.forward_out = None;
self.backward_in = None; self.backward_out = None;
```

Listing 8 Initialize the convolution block - __init__.py

Initialize To initialize, firstly we determinate the number of channels, number of inputs and the number of outputs by the shape of inputs :

```
self.n_channels, self.n_inputs = self.forward_in.shape;
self.n_outputs = self.n_inputs - self.kernel_size + 1
```

Next we use our initializer to create the array of weights. Noted that initializer creates an array in two dimensional so we had to flatten it by making the number of columns to be the number of channels times kernel size :

```
weights_unshaped = self.initializer.initialize(self.n_kernels, self.n_channels * self.kernel_size)
```

After that, we reshape it to get the right three dimensional array of weights that want :

```
weights = np.reshape( weights_unshaped, (self.n_kernels, self.n_channels, self.kernel_size), order='C')
```

where we use the `row-major` (`order='C'`) to make sure that it unpacks those values in the right order

```
self.weights = weights.transpose(1, 2, 0)
```

Remarque 3.2.1

Another reason that we flatten it by `self.n_channels * self.kernel_size` then reshape it back to three dimensions is that for the LSUV initializer, by given an input with a unit variance and a mean of zero it will produce outputs with a unit variance and a mean of zero. Since we will use the convolution for multiple channels, we will add the outputs by different channels up. We want the kernel across multiple channels after added together to have a unit variance with a mean of zero. That's why we flatten it in one row before the LSUV process. And then we unpack it.

Listing 9 Initialize the convolution block - initialize.py

3.2.3 Dunder methods

The `__str__` is not necessary but we could use the summary of what it generated to debug or understand the state of the objects. It notes the important parameters such as the number of inputs, number of channels and so on. It records the initializer and the weight optimizer.

Listing 10 Dunder Str function - __str__.py

3.2.4 Forward and backward pass

Forward pass Again, since

$$y_j = b_j + \sum_{n=0}^{n_c-1} \sum_{k=-p}^{p} x_{c,j-k} w_{c,k}$$

the mean part of the `forward_pass()` method is

```
self.forward_out = calculate_outputs(self.forward_in, self.weights)
```

Remarque 3.2.2: calculate_outputs()

To compute the function calculate_outputs() we need and the signal (inputs) $x_{c,j-k}$ the full set of kernels $w_{c,k}$:

```
def calculate_outputs(inputs, kernel_set):
```

We initialize the result as an array of zeros with the number of kernels in the rows, concerning the number of outputs which is equal to number inputs minus the kernel size plus one in the columns :

```
result = np.zeros((n_kernels, inputs.shape[1] - kernel_set.shape[1] + 1))
```

As there is a set of kernels to deal with, we work on one kernel at a time :

```
for i_kernel in range(n_kernels):
```

For each single row, we create a function calculate_single_kernel_output(signal, kernel), so here we need the full set of inputs but just the i-th kernel for the iteration as variables. We assign this result to relevant row :

```
result[i_kernel, :] = calculate_single_kernel_output(inputs, kernel_set[:, :, i_kernel])
```

Remarque 3.2.3: calculate_single_kernel_output()

We need the signal, a 2 dimensional array with number of channels by number of inputs and the kernel, also a 2 dimensional array with number of channels by length of the kernel :

```
def calculate_single_kernel_output(signal, kernel):
```

We initialize the result as an array of zeros, same reason, as we use the `'valid'` mode for the convolution :

```
result = np.zeros(signal.shape[1] - kernel.shape[1] + 1)
```

For each channel, we calculate the convolution of that particular channel of signal with that particular channel in the kernel :

```
result += convolve_1d(signal[i_channel, :], kernel[i_channel, :])
```

and we add them up.

Backward pass Firstly we know the gradient of the bias is equal to the gradient of the output :

$$\frac{\partial \mathcal{L}}{\partial b_j} = \frac{\partial \mathcal{L}}{\partial y_j}$$

so we can use it to compute `dL_db` if we want.

Meanwhile, to calculate weight gradient, since :

$$\frac{\partial \mathcal{L}}{\partial w_{c,k}} = \sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} x_{c,j-k}$$

we could write

```
self.dL_dw = calculate_weight_gradient(dL_dy, self.forward_in)
```

Remarque 3.2.4: calculate_weight_gradient() & calculate_single_kernel_weight_gradient()

As the same process before, we just note here by definition of the cross-correlation :

$$c_{xy}^E[k] = \sum_{n=-\infty}^{+\infty} x[n]y^*[n-k],$$

for the weight gradient :

$$\sum_{j=0}^{m-1} \frac{\partial \mathcal{L}}{\partial y_j} x_{c,j-k}$$

so we calculate the cross-correlation rather than the convention as before :

```
result[i_channel, :] = xcorr_1d(output_grad_padded, inputs[i_channel, :])
```

Another way to see is that the weight gradient is the convolution between the output gradient and the reversed kernel $\overleftarrow{\mathbf{x}}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} * \overleftarrow{\mathbf{x}}$$

it means

1. at first we flip the $\overleftarrow{\mathbf{x}}$ which gives $\mathbf{x}$
2. then we calculate the cross-correlation between $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ and $\mathbf{x}$

thus

```
result[i_channel, :] = xcorr_1d(output_grad_padded, inputs[i_channel, :])
```

To update the gradient of the weight, we send it with the weights to its own optimizer :

```
self.optimizer.update(self.weights, self.dL_dw)
```

Finally we pass the input gradient back :

$$\frac{\partial \mathcal{L}}{\partial x_{c,i}} = \sum_{k=-p}^{p} \frac{\partial \mathcal{L}}{\partial y_{i+k}} w_{c,k}$$

just write :

```
dL_dx = calculate_input_gradient(dL_dy, self.weights)
```

Remarque 3.2.5: calculate_input_gradient() & calculate_single_kernel_input_gradient()

— When we do the forward pass, we use the covolution in `purple'valid'` mode, which means the output signal will be shorter than the input signal by `kernel_size - 1`.

  Considering the backward pass, in order to compute from a shorter signal (the output) to a longer signal (the input), we use the `'full'` mode to calculate the convolution rather than the `purple'valid'` mode. So, as we mentioned before, we add `kernel_size - 1` zeros to either end of our output gradient to make sure the output gradient completely overlaps with the kernel at the beginning and at the end of the convolution :

```
output_grad_padded = np.zeros( (n_kernels, n_outputs + 2 * (kernel_size - 1)))

output_grad_padded[:, kernel_size - 1: n_outputs + kernel_size - 1]
```

— Again with the same process as before, we just note here since

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} * \overleftarrow{\mathbf{w}}$$

  thus

```
result[i_channel, :] = xcorr_1d(output_grad_padded, kernel[i_channel, :])
```

Listing 11 Forward and backward pass - forward_backward_pass.py

3.2.5 Summary

We put all of the above functions together to form a one-dimensional **Convolutional Block** (class) `Conv1D` :

Listing 12 class Conv1D - Conv1D.py

which contains

— forward pass : it takes in inputs and calculates the convolutional results with the kernels.
— backward pass : it takes in the output gradient and calculates the weight gradient, updates the weights. It calculates the input gradients then propagates the input gradients back to any previous block (layer).

3.3 Build a small convolutional neural network

3.3.1 Example of creating training and evaluation data block

Here is an example of creating training and evaluation data block, it helps to understand the principle and create the training and evaluation data for our particular case later.

We create two blocks : one is the **Training Data Block** , another is the **Evaluation Data Block** .

```
class TrainingData(object):
```

and

```
class EvaluationData(object):
```

each block contains :

1. A dunder `__init__` method for the initializer :

   the `get_data_sets()` function will return 2 generators

   ```
   self.training_data_generator, _ = get_data_sets()
   ```

   we use the first is a training data generator which will initialize the training data block ;

   ```
   _, self.evaluation_data_generator = get_data_sets()
   ```

   and we the second is the evaluation data generator which will initialize the evaluation data block.
2. A `forward_pass` :

   ```
   return next(self.training_data_generator)
   ```

   ```
   return next(self.evaluation_data_generator)
   ```

   in Python, we can call `next` on the generator and then it will give the next item in the list that it has.
3. A `backward_pass`
4. The dunder `__str__` method

after coded that, it could be connected to other blocks.

Remarque 3.3.1: About `*arg` explained by Brandon Rohrer

For the `forward_pass` and the `backward_pass` :

```
def forward_pass(self, *arg):
```

```
def backward_pass(self, *arg):
```

each accepts an argument (`*arg`). “This is not because the data loader needs a parameter, but because the structure would expect to be able to pass an argument to the forward and backward pass functions of each block, even if that argument is empty. Therefore, we need to include a parameter in the forward and backward pass functions in order to be able to accept that parameter, even if, in our case, we don’t need/use it.”

Then we create the list of signal sets with labels by the `get_blips()` function :

```
examples = get_blips()
```

each signal created has 21 elements long initialized as an one dimensional array of element 0, which contains a blip with 7 elements long might falling in a different location each time. The signal has 4 types of label defined by shape / pattern of the blip (M, V, N, H). We will generate 400 examples of signals with 100 examples for each types (flavors).

We use the list of `examples` to create two generators :

```
def training_set():
```

the training set generator

```
def evaluation_set():
```

and the evaluation set generator. Each loop, we choose randomly one of the examples from the list, copy it to the generator and put this example back to the list :

```
i_example = np.random.choice(range(len(examples)))
```

```
yield examples[i_example]
```

So each time we call the training data block or evaluation data block, like

```
training_block = TrainingData()
for _ in range(10): new_training_example = training_block.forward_pass()
```

the `forward_pass(self, *arg)` function inside will generate in a tuple with a two dimensional array one row by 21 columns and its label. Here is an example :

```
(array([[0. , 0. , 0. , 0. , 1. , 0.7, 0.4, 0.1, 0.4, 0.7, 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
0. , 0. ]]), 'M')
```

Listing 13 class TrainingData - data_loader_blips.py

3.3.2 OneHot Block, Flatten Block and ValueLogger

OneHot Block We use the one-hot block to translate each label to an one dimensional array. This class contains :

1. A dunder string method `__str__`
2. A `__init__` method as well, where we define the number of the labels and the dictionary of the label (`categories`) and the array in one-hot encoding :

   ```
   self.n_categories = n_categories; self.categories = {}; self.result = None
   ```

3. A `forward_pass` : We return an one-dimensional array consisting of zeros of length `self.n_categories`.

   ```
   self.result = np.zeros(self.n_categories)
   ```

   We check if the current label is in our dictionary or not.
   — If it is in, we will find the position of this label in the dictionary and replace 0 by 1 at the same position in the array of zeros :

   ```
   if label in self.categories.keys(): self.result[self.categories[label]] = 1
   ```

   — If not, we will add the label in the dictionary the label (`categories`). The key is `label` and the value is the number of categories so far `n_cats_so_far`.

   ```
   self.categories[label] = n_cats_so_far
   ```

   The `n_cats_so_far` is a new free position. For the previously initialised one-dimensional array, we replace 0 by 1 at this position.

   ```
   self.result[n_cats_so_far] = 1
   ```

   Finally we pass the result on

   ```
   return self.result
   ```

4. A `backward_pass` : We will enrich it later for making predictions while at the moment, we can just leave it empty.

   ```
   return values
   ```

Listing 14 class OneHot - operations.py

Flatten Block With the Flatten Block, we turn an n-dimensional array into a two dimensional array with just one row, which enables the data to be used in linear block. We need :

1. A dunder string method `__str__`
2. A `__init__` method which initialize the shape of the input :

   ```
   self.input_shape = None
   ```

3. A `forward_pass` : It notes the shape of that input

   ```
   self.input_shape = values.shape
   ```

   Then we flatten the array into one row using the `ravel()` function and we add one dimension to this one-dimensional array :

   ```
   return values.ravel()[np.newaxis, :]
   ```

4. A `backward_pass` : This is a inversion of the flattening operation. We reshape it according to the shape of the original input array (`values.shape`).

   ```
   return flat_values.reshape(self.input_shape)
   ```

Listing 15 class Flatten - operations.py

ValueLogger We create the logger for logging values. It writes down the value passed on in each iteration, appends the value to a list then we could plot it (`def report(self):`) and save all the list as a csv file (`def write(self):`).

Listing 16 class ValueLogger - logger.py

3.3.3 The blocks needed to build a small convolutional neural network

Blocks needed

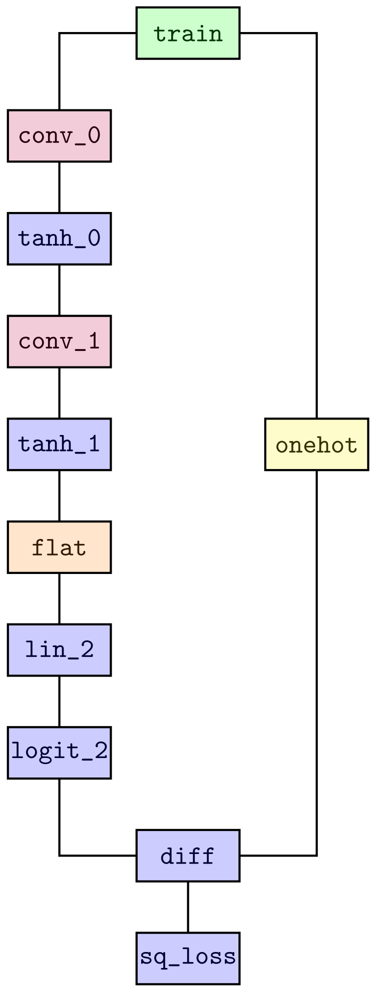


FIGURE 3.3.1 – A small convolutional neural network.

This small convolutional neural network shows that

1. We generate the training data in the training data block (`train`),
2. we pass the data to the convolution block (`conv_0`).
3. After that we use an activation function which introduces non-linearity into the networking helping the network learn complex features. Here we use hyperbolic tangent as the activation function (`tanh_0`).
4. Then we use another group of convolution block (`conv_1`) and activation function (`tanh_1`).
5. Now the data is a two dimensional array, we flatten it down to an one dimensional array (`flat`), which is used to prepare for the following step.
6. The linear block, which needs an one dimensional array as the input, combines the high-level features extracted by the convolutional blocks (`lin_2`). The linear block is like the dense block but without the activation function, normally used for the regression task.
7. Here, our example is for a classification task, we add the activation function after the linear block. We use the logistic block (`logit_2`) to compresses the output to between 0 and 1.
8. We use the one-hot block to translate each label to an one dimensional array :

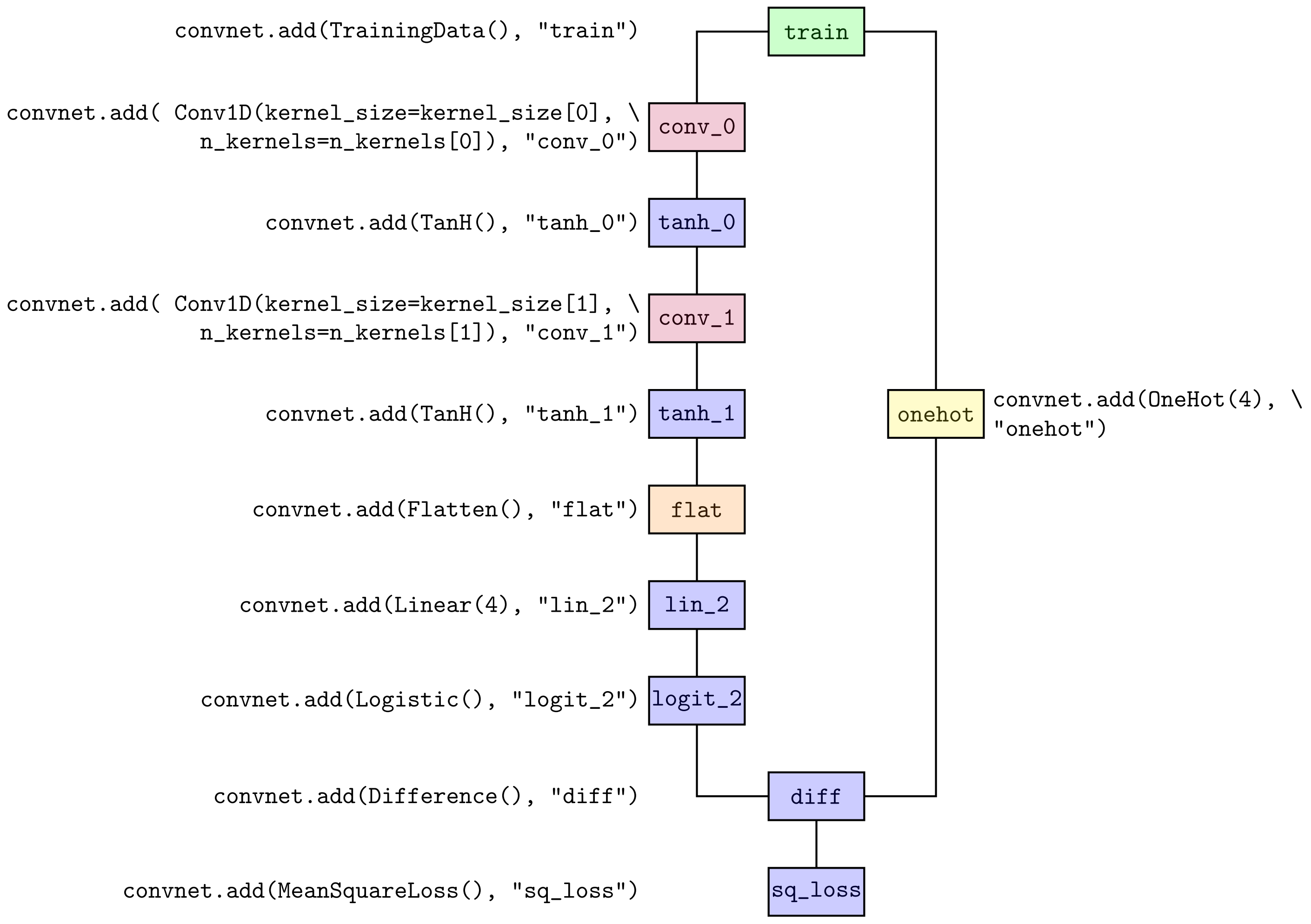
Exemple 3.3.1: one-hot encoding

In our case we have 4 types of labels : M, V, N, H, supposing the dictionary to be $\mathcal{D} = \{M, V, N, H\}$. The rule is that the array translated are zero values except for the indexes of the label as 1, for example :

$$M : \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} \quad N : \begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix}$$

9. Last but not least, we use a difference block (`diff`) to calculate the difference between the label in one-hot encoding and the output of the convolution loop.
10. We use the mean square loss function (`sq_loss`) to find the gradient of the error then we back propagate it back through the network to adjust the relevant weights so that our result after the convolution loop could match the label better.

Build a small convolutional neural network



```
convnet.add(TrainingData(), "train")
convnet.add( Conv1D(kernel_size=kernel_size[0], \
        n_kernels=n_kernels[0]), "conv_0")
convnet.add(TanH(), "tanh_0")
convnet.add( Conv1D(kernel_size=kernel_size[1], \
        n_kernels=n_kernels[1]), "conv_1")
convnet.add(TanH(), "tanh_1")
convnet.add(Flatten(), "flat")
convnet.add(Linear(4), "lin_2")
convnet.add(Logistic(), "logit_2")
convnet.add(Difference(), "diff")
convnet.add(MeanSquareLoss(), "sq_loss")
convnet.add(OneHot(4), \
"onehot")
```

FIGURE 3.3.2 – Build a small convolutional neural network.

To build this small convolutional neural network in Python

1. Firstly we initialize a structure `convent`, the short name for convolutional neural network :

    ```
    convnet = Structure()
    ```

    where the class `Structure()` works as a foundational data structure. It orchestrates the operations of a collection of blocks and their connections. We import it by

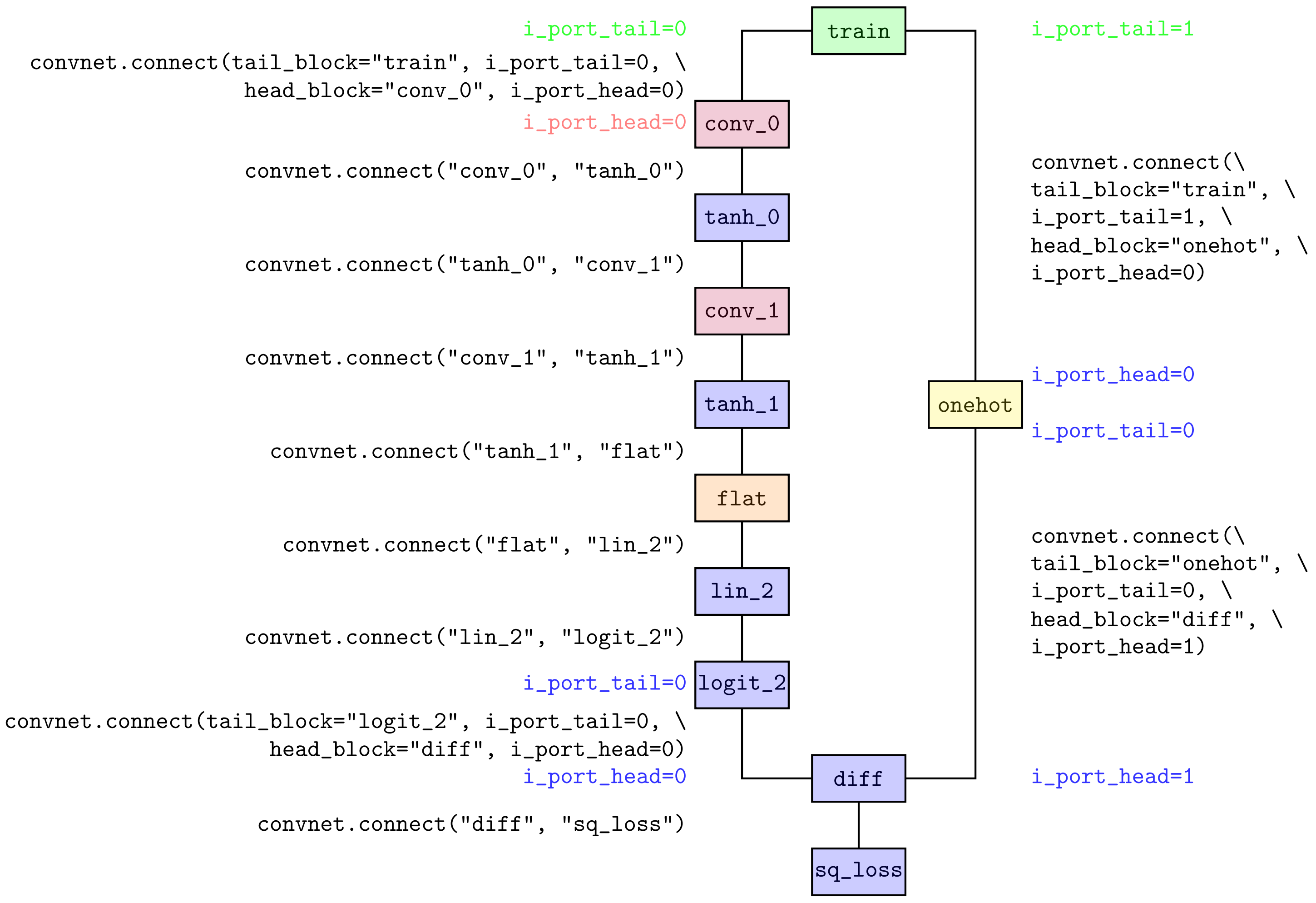
    ```
    from cottonwood.core.blocks.structure import Structure
    ```

2. We choose the size of the kernels and the number of kernels for two convolution blocks :

    ```
    kernel_size = [5, 3]; n_kernels = [15, 12]
    ```

3. We add the blocks one by one (ref. FIGURE 3.3.2).

3.3.4 Connect the blocks into a network structure



i_port_tail=0
convnet.connect(tail_block="train", i_port_tail=0, \
head_block="conv_0", i_port_head=0)
i_port_head=0
convnet.connect("conv_0", "tanh_0")
convnet.connect("tanh_0", "conv_1")
convnet.connect("conv_1", "tanh_1")
convnet.connect("tanh_1", "flat")
convnet.connect("flat", "lin_2")
convnet.connect("lin_2", "logit_2")
i_port_tail=0
convnet.connect(tail_block="logit_2", i_port_tail=0, \
head_block="diff", i_port_head=0)
i_port_head=0
convnet.connect("diff", "sq_loss")
train
conv_0
tanh_0
conv_1
tanh_1
flat
lin_2
logit_2
diff
sq_loss
onehot
i_port_tail=1
convnet.connect(\
tail_block="train", \
i_port_tail=1, \
head_block="onehot", \
i_port_head=0)
i_port_head=0
i_port_tail=0
convnet.connect(\
tail_block="onehot", \
i_port_tail=0, \
head_block="diff", \
i_port_head=1)
i_port_head=1

FIGURE 3.3.3 – Connect the blocks into a network structure.

1. We start the connection with the training block. As we told before, the output of the training block is a tuple containing the signal and its label, for example :

   ```
   (array([[ 0.  , -0.7, 0.7, 0.4, 0.  , -0.4, -0.7, 0.7, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
                                     0.  , 0.  , 0.  , 0. ]]), 'N')
   ```

   we want the signal pass from the tail of the training block to the head of the convolution block,

   ```
   convnet.connect(tail_block="train", i_port_tail=0, head_block="conv_0", i_port_head=0)
   ```

   and the label pass from the tail of the training block to the head of the one-hot block

   ```
   convnet.connect(tail_block="train", i_port_tail=1, head_block="onehot", i_port_head=0)
   ```

2. And if we don't specify the port, it configures zero for both or for either, for example :

   ```
   convnet.connect("conv_0", "tanh_0")
   ```

3. Finally, all the connections is made in our architecture (ref. FIGURE 3.3.3). The network is built.

3.3.5 Training, evaluation and reporting

Reporting We create a logger to track a value that we send to see how its value changes with time.

```
loss_logger = ValueLogger()
```

By this way, we could track the loss of the network.

Training In the training loop, we essentially need one forward pass and one backward pass for each iteration :

```
convnet.forward_pass(); convnet.backward_pass()
```

Besides we could create some loggers which are not in need :

— We use the `loss_logger` to log the loss value over time. We will be able to plot the loss value in function of the number of iterations in the end :

```
loss_logger.log_value(convnet.blocks["sq_loss"].loss)
```

— We can use the module `conv_viz.render` to visualize the current state of the convolution block.

— We can summarize the structure :

```
tb.summarize(convnet, reports_dir=reports_dir)
```

it will show all the blocks that we used and their current parameters in a text file.

— We can also visualize the network structure and the connection of the blocks :

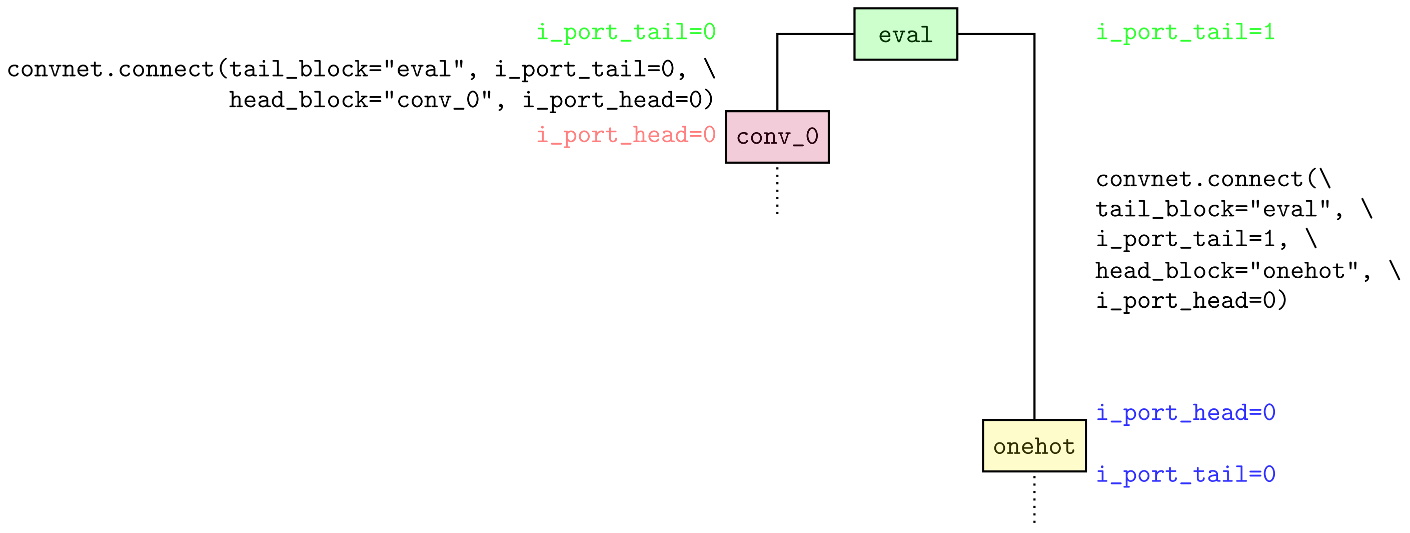
```
struct_viz.render(convnet, reports_dir)
```

in a png file.

Evaluation Concerning evaluation, we just modify a little about our network :

```
convnet.remove("train")

convnet.add(EvaluationData(), "eval")
```



```
convnet.connect(tail_block="eval", i_port_tail=0, \
                head_block="conv_0", i_port_head=0)
```

```
convnet.connect(\
tail_block="eval", \
i_port_tail=1, \
head_block="onehot", \
i_port_head=0)
```

FIGURE 3.3.4 – Evaluation.

We remove the training data block then add the evaluation data block and reconnect it to the network structure (ref. FIGURE 3.3.4).

In the evaluation loop, what we essentially need is just one forward pass :

```
convnet.forward_pass()
```

we do not use a backward pass to update the parameters : we do not need adjusting the loss gradient to minimize the loss here. We just want the data tuple pass through the convolutional neural network and get its result.

Besides, we can also create some loggers as described in the **Training** paragraph.

Listing 17 Training, evaluation and reporting - blip_demo.py

3.4 Add ReLU, Pooling, Batch Normalization

Yet, we have built a small convolutional neural network using the convolutional blocks to understand the role of each block, find out how to connect each block and how the network learn the result. However there are several pieces missing : ReLU, Pooling and Batch Normalization. In this subsection, we will add them in to make a fully fledged convolutional neural network.

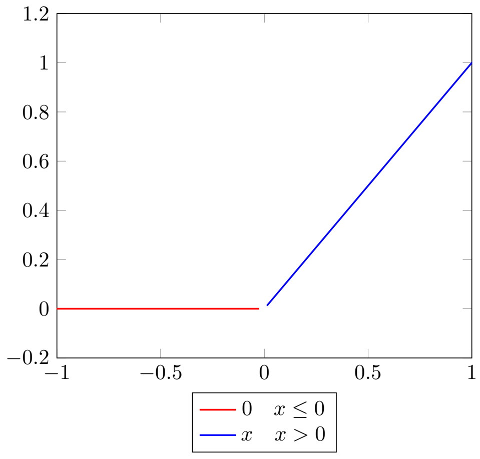
3.4.1 Rectified Linear Unit Block

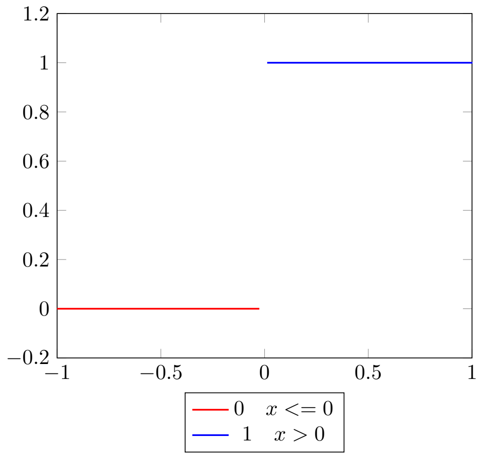
The figure below, on the left, is the image of the ReLU activation function :

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} = max(0, x)$$

and on the right, is the derivative of the ReLU activation function :

$$f'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$





To code the ReLU, we need

1. A `__init__` method to initialize the result :

```
self.result = None
```

2. A under string method `__str__` just tell we are actually in a ReLU block

```
return "rectified linear unit"
```

3. A `forward_pass` : It does a ReLU calculation : if the element is under 0, it sets it to 0 ; if not it lets it pass, we do it for the whole array :

```
self.result = np.maximum(0, values)
```

4. A `backward_pass` : We calculate the derivative of the ReLU function, i.e. the partial derivative of the output with respect to the input

```
d_relu = np.zeros(self.result.shape); d_relu[np.where(self.result > 0)] = 1
```

   Then we calculate the gradient passing backward to the preceding block using the

```
return grad * d_relu
```

Listing 18 class ReLU - activation.py

3.4.2 One dimensional max pooling Block

Pooling is the operation that we use a window to take groups of the signal and the we reduce each group to a single value. We have average pooling, which means a take the average value in each group. And the most commonly used for the convolutinal neural network, the max pooling. It works well with the ReLU activation function : if we have the possitive and negative values, it is highly likely that the positive part will be retained thus most of the information will be retained ; if we have negative value, it will be replaced by 0.

In the forward pass, we initialize the max pooling Block by the stride and window. The window is the number of elements that we take in a group. And the stride is the shift step. If we take the length of the windows to be the same as the stride size, then there is no overlap between the windows :

Exemple 3.4.1: max pooling : window = 3, stride = 3

signal : $\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 1&4&6&-1&5&3&4&7&9\\\hline\end{array}$

window = 3, stride = 3 : $\begin{array}{|c|c|c|}\hline 1&4&6\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline -1&5&3\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 4&7&9\\\hline\end{array}$

—·—·—·—·—·—·—·—·—·—·—

index of element max in each window : $\boxed{2}\ \boxed{1}\ \boxed{2}$

result : $\boxed{6}\ \boxed{5}\ \boxed{9}$

In the backward pass, we use the unpooling operation to remap the pooled gradient ($\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$) back to the dimensions of the original input signal, recovering the positional information of the original signal ($\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$) :

Exemple 3.4.2: max unpooling : window = 3, stride = 3

gradient : $\boxed{0.1}\ \boxed{0.2}\ \boxed{0.3}$

index of element max in each window : $\boxed{2}\ \boxed{1}\ \boxed{2}$

—·—·—·—·—·—·—·—·—·—·—

window = 3, stride = 3 : $\begin{array}{|c|c|c|}\hline 0&0&0.1\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 0&0.2&0\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 0&0&0.3\\\hline\end{array}$

result : $\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0&0&0.1&0&0.2&0&0&0&0.3\\\hline\end{array}$

While, it has been proved that if we choose the stride size a little bit <u>smaller</u> than the length of the window which creates a little overlap between the windows, it will be more effective in practice :

Exemple 3.4.3: max pooling : window = 3, stride = 2

signal : $\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 1&4&6&-1&5&3&4&7&9\\\hline\end{array}$

window = 3, stride = 2 : $\begin{array}{|c|c|c|}\hline 1&4&6\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 6&-1&5\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 5&3&4\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 4&7&9\\\hline\end{array}$

—·—·—·—·—·—·—·—·—·—·—

index of element max in each window : $\boxed{2}\ \boxed{0}\ \boxed{0}\ \boxed{2}$

result : $\boxed{6}\ \boxed{6}\ \boxed{5}\ \boxed{9}$

Exemple 3.4.4: max unpooling : window = 3, stride = 2

gradient : $\boxed{0.1}\ \boxed{0.2}\ \boxed{0.3}\ \boxed{0.4}$

index of element max in each window : $\boxed{2}\ \boxed{0}\ \boxed{0}\ \boxed{2}$

—·—·—·—·—·—·—·—·—·—·—

window = 3, stride = 2 : $\begin{array}{|c|c|c|}\hline 0&0&0.1\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 0.2&0&0\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 0.3&0&0\\\hline\end{array}\ \begin{array}{|c|c|c|}\hline 0&0&0.4\\\hline\end{array}$

result : $\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0&0&0.2&0&0.3&0&0&0&0.4\\\hline\end{array}$

To code the `MaxPool1D` block, we need

1. An initializer `__init__(self, stride=2, window=3)` which defines some parameters that we need later : the number of signals `self.n_signals = None`, the length of each signal `self.signal_length = None`, the length of the signal after been pooled `self.pooled_length = None` and index of the element with the maximum value in each window `self.i_max = None`.
2. To initialize these `None` values, we will write the `initialize` method. Then we initialize them according to the size of the inputs :

```
self.n_signals, self.signal_length = self.x.shape
```

We count the after been pooled as

```
(self.signal_length - self.window) // self.stride + 1
```

here `+1` means the first window ; then we calculate how much windows we can generate by moving the window from left to right with the step `self.stride`, `//` is the integer division which means if the last window doesn't match completely, we just ignore it, we only count windows that completely overlap.

And the index of the maximum value in each windows as well :

```
self.i_max = np.zeros((self.n_signals, self.pooled_length), dtype=int)
```

an array of the number of the signals by the length of signal after been pooled.

3. The dunder `__str__` method which gives a feedback of the important parameters of the initializer (or the max pooling blcok).
4. A `forward_pass` where we do the max pooling operation `def max_pool_1d(signals, i_max, window, stride, pooled_length):`
   - The number of the signal is shown by total line of the signal

     ```
     n_signals = signals.shape[0]
     ```

   - We initialize the result after the max pooling

     ```
     results = np.zeros((n_signals, pooled_length))
     ```

   - For each window possible `i_window` moving from left to right along the signal, we note down the start index `i_start` and the stop index `i_stop`. Then for each signal `i_signal`, we find the maximum value from the start index to the stop index and its index within each window `i_window`.

     ```
     results[i_signal, i_window] = np.max(signals[i_signal, i_start:i_stop])
     i_max[i_signal, i_window] = np.argmax(signals[i_signal, i_start:i_stop])
     ```

5. A `backward_pass` where we undo the max pooling operation `def max_unpool_1d(gradient, i_max, window, stride, signal_length):`
   - We read the number of the signals and the length of the signals after pooling by the gradient array

     ```
     n_signals, pooled_length = gradient.shape
     ```

   - We initialize the result of unpooling ($\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$)

     ```
     results = np.zeros((n_signals, signal_length))
     ```

   - For each signal `i_signal` and each window `i_window`, we replace the 0 by the gradient ($\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$) corresponding to the index of element max in each window.

     ```
     results[i_signal,i_window * stride + i_max[i_signal, i_window]] = gradient[i_signal, i_window]
     ```

Listing 19 class MaxPool1D - pooling.py

3.4.3 Batch Normalization

Batch Normalization was introduced by Sergey Ioffe and Christian Szegedy in 2015 [7]. It accelerates the training process and improves model stability (avoid the vanishing gradient or exploding gradient) by normalizing the inputs and performing a linear transformation at each layer.

Transformation in Forward Propagation First of all, we will change the distribution, we calculate the mean and variance of all m samples with the same characteristic

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad , \quad \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

for each sample, we centre each element of the input and let its variance to be 1

$$\widehat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \ ;$$

Then for each normalized $\widehat{x}_i$ we calculate its linear transformation by the parameters γ, β learned along with the original model parameters to restore the representation power of the network

$$y_i = \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \ .$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

```
/* One Dimensional Batch Normalization */
Input   : Batch Normalization : $\mathcal{B} = \{x_1, \ldots, x_m\}$,
          Parameters to be learned : $\gamma$, $\beta$.
Output  : Linear transformation : $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$.
1 mini-batch mean : $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^m x_i$ ;
2 mini-batch variance : $\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$ ;
3 for $i = 1$ to $m$ do
4     normalize : $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$ ;          // here $\epsilon$ is a small constant to avoid division by zero
5     scale and shift : $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ ; // trainable parameters $\gamma, \beta$ are used to perform a linear transformation
6 end
```

Backpropagation Again, neural networks learn their parameters by the backpropagation which needs the loss function to be differentiable. By the chain rule, here is the result of the gradient of the loss l with respect to all different parameters in one dimension, where $\frac{\partial l}{\partial y_i}$ depends on the the choice of activation function and we use $\frac{\partial l}{\partial y_i}$ to represent other gradients [8] :

$$\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i}\gamma \quad , \quad \frac{\partial l}{\partial \gamma} = \sum_{i=1}^m \frac{\partial l}{\partial y_i}\hat{x}_i \quad , \quad \frac{\partial l}{\partial \beta} = \sum_{i=1}^m \frac{\partial l}{\partial y_i},$$

$$\frac{\partial l}{\partial \sigma^2_B} = \sum_{i=1}^m \frac{\partial l}{\partial y_i}(x_i - \mu_B)\left(-\frac{\gamma}{2}(\sigma^2_B + \epsilon)^{-3/2}\right),$$

$$\frac{\partial l}{\partial \mu_B} = \sum_{i=1}^m \frac{\partial l}{\partial y_i}\frac{-\gamma}{\sqrt{\sigma^2_B + \epsilon}} + \frac{\partial l}{\partial \sigma^2_B}\frac{1}{m}\sum_{i=1}^m(-2)\cdot(x_i - \mu_B),$$

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i}\frac{1}{\sqrt{\sigma^2_B + \epsilon}} + \frac{\partial l}{\partial \sigma^2_B}\frac{2(x_i^{(k)} - \mu_B)}{m} + \frac{\partial l}{\partial \mu_B}\frac{1}{m}.$$

Implementation Now we want to implement the original batch normalisation introduced by Sergey Ioffe and Christian Szegedy [7]. The difficulty is to hold up to thousands of batches of inputs in memory to realize the transformation (calculating the mean and variance) and then move it forward. It's friendly to the GPU (or other parallel processing hardware) users but it is not suitable for the users running the code in their local CPU. Brandon Rohrer modified it by operating on one input at a time making it possible for batch normalisation to run on the local CPU [2]. There are two modifications :

1. The batch mean ($\mu_{\mathcal{B}}$), batch variance ($\sigma^2_{\mathcal{B}}$) and their partial derivatives ($\frac{\partial l}{\partial \sigma^2_B}$ and $\frac{\partial l}{\partial \mu_B}$) can only be calculated at the end of a batch, it means that we can't calculate them it the current batch. Thus we take their values from the previous batch.
2. We apply the updated shift (β) and scale (γ) parameters to the following batch but not the current batch. Cause we should wait for the current batch finishing, then we will get the $\frac{\partial l}{\partial \gamma}$ and $\frac{\partial l}{\partial \beta}$ to update the shift and scale.

Since we use the parameters from the previous batch but not the current batch when we calculate forward and backward passes, the match mean and variance don't deviate significantly if the shift and scale change gradually. However, their partial derivatives ($\frac{\partial l}{\partial \sigma^2_B}$ and $\frac{\partial l}{\partial \mu_B}$) may change significantly from one batch to the next. To ease this instability to the network, Brandon Rohrer calculate a running estimate (using an exponential-decay weighted average) of the population mean and variance $\mu_{\mathcal{P}}, \sigma^2_{\mathcal{P}}$ (but not the batch mean and variance) as the normalizing inputs in the forward pass. Here is the implementation of the class `OnlineBatchNormalization` where online means we operate on one input at a time then forget it but not means connecting to the Internet. We need

1. A dunder `__init__(self, minibatch_size=256, optimizer=Momentum())` method for the initializer, here the defaut minibatch size was set to 256, we choose momentum as the default optimizer. We're free to choose these default values. Inside
   — We create two optimizer one for shift (β), another for scale (γ)

   ```
   self.shift_optimizer = deepcopy(optimizer); self.scale_optimizer = deepcopy(optimizer)
   ```

   — We mention all the parameters that we will use :
   - $x_i, y_i, \widehat{x}_i, \frac{\partial l}{\partial x_i}, \beta, \gamma$

     ```
     self.x = None; self.y = None; self.z = None; self.dL_dx = None;
                    self.shift = None; self.scale = None;
     ```

   - $\mu_{\mathcal{B}}, \sigma^2_{\mathcal{B}}, \mu_{\mathcal{P}}$ for mean of all the batch seen so far, $\sigma^2_{\mathcal{P}}$

     ```
     self.batch_mean = None; self.batch_variance = None;
       self.pop_mean = None; self.pop_variance = None;
     ```

- $\frac{\partial l}{\partial \gamma}, \frac{\partial l}{\partial \beta}, \frac{\partial l}{\partial \mu_B}, \frac{\partial l}{\partial \sigma_B^2}$, and $\frac{\partial l}{\partial \mu_B}$ for the previous batch, $\frac{\partial l}{\partial \sigma_B^2}$ for the previous batch.
  `self.dL_dscale = None; self.dL_dshift = None;`
  `self.dL_dbatch_mean = None; self.dL_dbatch_variance = None;`
  `self.dL_dbatch_mean_prev = None; self.dL_dbatch_variance_prev = None;`
- sum of the x_i, sum of the squares of x_i, size of the minibatch and the counter of the current minibatch :
  `self.sum_x = None; self.sum_sq_dev = None;`
  `self.minibatch_size = minibatch_size; self.i_minibatch = 0`

2. To initialize, we use the `initialize` method to make the parameters in appropriate shape using the shape of the input.
3. The dunder `__str__` method to give a brief summary of this block (like the optimizers of β and γ).
4. A `forward_pass` : here are the key points
   — About the previous batch :
   - the batch mean $\mu_{\mathcal{B}}$
$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i,$$
```
self.batch_mean = self.sum_x / self.minibatch_size
```
   - the sample / unbiased estimate of batch variance $\sigma_{\mathcal{B}}^2$
$${\sigma_{\mathcal{B}}}^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 = \frac{1}{m-1}\left(\sum_{i=1}^{m} {x_i}^2 - \frac{\left(\sum_{i=1}^{m} x_i\right)^2}{m}\right)$$
```
self.batch_variance = ((self.sum_sq_x - self.sum_x ** 2 / self.minibatch_size) /
                        (self.minibatch_size - 1))
```
Démonstration.
$$\begin{aligned}\sigma_{\mathcal{B}}^2 &= \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 = \frac{1}{m-1}\left(\sum_{i=1}^{m} {x_i}^2 + \sum_{i=1}^{m} {\mu_{\mathcal{B}}}^2 - 2\sum_{i=1}^{m} x_i\mu_{\mathcal{B}}\right)\\ &= \frac{1}{m-1}\left(\sum_{i=1}^{m} {x_i}^2 + m{\mu_{\mathcal{B}}}^2 - 2\mu_{\mathcal{B}}\sum_{i=1}^{m} x_i\right) = \frac{1}{m-1}\left(\sum_{i=1}^{m} {x_i}^2 + m{\mu_{\mathcal{B}}}^2 - 2m{\mu_{\mathcal{B}}}^2\right)\\ &= \frac{1}{m-1}\left(\sum_{i=1}^{m} {x_i}^2 - m{\mu_{\mathcal{B}}}^2\right) = \frac{1}{m-1}\left(\sum_{i=1}^{m} {x_i}^2 - \frac{\left(\sum_{i=1}^{m} x_i\right)^2}{m}\right)\end{aligned}$$
□
   - compute the population mean $\mu_{\mathcal{P}}$ by exponential smoothing [9] [1]
$$\mu_{\mathcal{P}} \leftarrow \frac{\mu_{\mathcal{P}} + \mu_{\mathcal{B}}}{2}$$
```
self.pop_mean = self.pop_mean / 2 + self.batch_mean / 2
```
   - compute the population variance ${\sigma_{\mathcal{B}}}^2$ by exponential smoothing
$$\sigma_{\mathcal{P}}^2 \leftarrow \frac{\sigma_{\mathcal{P}}^2 + \sigma_{\mathcal{B}}^2}{2}$$
```
self.pop_variance = self.pop_variance / 2 + self.batch_variance / 2
```
   — Once the previous batch finished, it's current batch now, we compute
   - the normalization
$$\widehat{x}_i = \frac{x_i - \mu_{\mathcal{P}}}{\sqrt{\sigma_{\mathcal{P}}^2 + \epsilon}} \;;$$
```
self.z = ( (self.x - self.pop_mean) * (self.pop_variance + EPSILON) ** (-1 / 2) )
```
   - the transformation
$$y_i = \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \;.$$
```
self.y = (self.z + self.shift) * self.scale
```
5. A `backward_pass` : we just apply formulas documented by the Wikipedia [8] to the implementation.
   — We apply exactly the following formulas :

- ○

$$\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i}\gamma \quad , \quad \frac{\partial l}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i}\hat{x}_i \quad , \quad \frac{\partial l}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i},$$

```
dL_dz = dL_dy * self.scale; self.dL_dscale += dL_dy * self.z; self.dL_dshift += dL_dy;
```

- ○

$$\frac{\partial l}{\partial \sigma_B^2} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i}(x_i - \mu_B)\left(-\frac{\gamma}{2}(\sigma_B^2 + \epsilon)^{-3/2}\right),$$

```
self.dL_dbatch_variance += ( dL_dy * (self.x - self.batch_mean) * (-self.scale / 2)
    * (self.batch_variance + EPSILON) ** (-3 / 2))
```

— We notice that the term $\frac{\partial l}{\partial \sigma_B^2}$ will not be available until the current batch ends. So we replace the result of $\frac{\partial l}{\partial \sigma_B^2}$ refer to the current batch by the one from the previous batch :

- ○

$$\frac{\partial l}{\partial \mu_B} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i}\frac{-\gamma}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_B^2}\frac{1}{m}\sum_{i=1}^{m}(-2)\cdot(x_i - \mu_B),$$

```
self.dL_dbatch_mean += ( dL_dy * (-self.scale) * (self.batch_variance + EPSILON) ** (-1 /
    2) + self.dL_dbatch_variance_prev / self.minibatch_size * (-2)
    * (self.x - self.batch_mean) )
```

To calculate the input gradient $\frac{\partial l}{\partial x_i}$, we use $\frac{\partial l}{\partial \sigma_B^2}$, $\frac{\partial l}{\partial \mu_B}$ from the previous batch for the same reason :

- ○

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i}\frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_B^2}\frac{2(x_i^{(k)} - \mu_B)}{m} + \frac{\partial l}{\partial \mu_B}\frac{1}{m}.$$

```
self.dL_dx = ( dL_dz * (self.batch_variance + EPSILON) ** (-1 / 2)
+ self.dL_dbatch_variance_prev * 2 * (self.x - self.batch_mean) / self.minibatch_size
    + self.dL_dbatch_mean_prev / self.minibatch_size )
```

— Finally we update the parameters : we use the $\frac{\partial l}{\partial \gamma}$ to update the scale γ and $\frac{\partial l}{\partial \beta}$ to update the shift β :

- ○

```
self.scale_optimizer.update(self.scale, self.dL_dscale);

self.shift_optimizer.update(self.shift, self.dL_dshift);
```

We assign the current value to the previous :

- ○

```
self.dL_dbatch_mean_prev = self.dL_dbatch_mean;

self.dL_dbatch_variance_prev = self.dL_dbatch_variance;
```

And we set the all counters to be zeros.

Improvements In our implementation before, we have made two modifications according to the original batch normalization. Since we use the the partial derivatives $\frac{\partial l}{\partial \mu_B}$, $\frac{\partial l}{\partial \sigma_B^2}$ from the previous batch rather than the current batch, it may introduce instabilities to the network if their values change rapidly from batch to batch. To ease this instability effect, we calculate the population statistic by exponential smoothing. However, the following paper [4] published at the Conference on Neural Information Processing Systems (NeurIPS 2019) provides a better approach to solve this problem. In our case $\mu(x_t) = x_t, \sigma(x_t) = 0$:

— In `forward_pass`, we use these formulas (for the derivation, see [6]) to compute an affine transformation :

- ○

$$y_t = \frac{x_t - \mu_{t-1}}{\sigma_{t-1}}$$

```
self.forward_out = ( (self.forward_in - self.running_means)
    / np.sqrt(self.running_variances + EPSILON) )
```

- ○

$$\sigma_t^2 = \alpha_f \sigma_{t-1}^2 + (1 - \alpha_f)\sigma^2(x_t) + \alpha_f(1 - \alpha_f)(\mu(x_t) - \mu_{t-1})^2$$

```
self.running_variances = ( self.forward_adaptation_rate * self.running_variances
    + ( self.forward_adaptation_rate * (1 - self.forward_adaptation_rate)
        * (self.forward_in - self.running_means) ** 2 ) )
```

- ○

$$\mu_t = \alpha_f \mu_{t-1} + (1 - \alpha_f)\mu(x_t)$$

```
self.running_means += ( (1 - self.forward_adaptation_rate)
    * (self.forward_in - self.running_means) )
```

— In `backward_pass`

∘
$$\tilde{x}'_t = y'_t - (1-\alpha_{\mathrm{b}})\varepsilon^{(y)}_{t-1} y_t$$

```
corrected_output_gradient = ( uncorrected_output_gradient
- self.prescaling_error * (1 - self.backward_adaptation_rate) * self.forward_out )
```

∘
$$\varepsilon^{(y)}_t = \varepsilon^{(y)}_{t-1} + \mu(\tilde{x}'_t y_t)$$

```
self.prescaling_error += corrected_output_gradient * self.forward_out
```

∘
$$x'_t = \frac{\tilde{x}'_t}{\sigma_{t-1}} - (1-\alpha_{\mathrm{b}})\varepsilon^{(1)}_{t-1}$$

```
uncorrected_input_gradient = ( corrected_output_gradient
/ np.sqrt(self.running_variances + EPSILON) );
corrected_input_gradient = ( uncorrected_input_gradient
- (1 - self.backward_adaptation_rate) * self.postscaling_error )
```

∘
$$\varepsilon^{(1)}_t = \varepsilon^{(1)}_{t-1} + \mu(x'_t)$$

```
self.postscaling_error += corrected_input_gradient
```

3.4.4 A basic structure of convolutional neural network

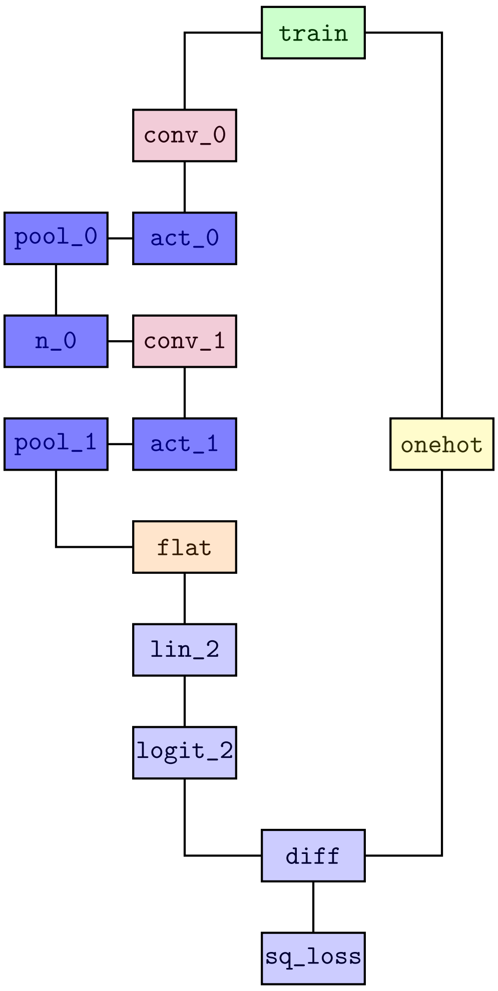


FIGURE 3.4.1 – A basic structure of convolutional neural network : ReLU, Pooling, Batch Normalization added.

We choose **ReLU** as the Activation block ; After the Activation block is the **Pooling block** ; After the first Pooling block and before the next Convolutional block, we add the **Online Normalization block**.

4 Prepare the data

4.1 Pre-process the theoretical wave : cut & pad

First of all, the data is the theoretical waveform calculated by the simulation. For each length of fault `length_fault` (2cm, 5cm and 7.5cm), we have a `.txt` file which contains 5 channels `channel_signal` : the channel 0 means the sampling points, the channel 1 to 4 refer to the different waveforms however we choose only the 3 and 4 cause 1 or 2 don't match the reality. Thus we will have $3 \times 2 = 6$ pre-processed theoretical waveforms in total. In the

$\boxed{\text{Listing 20 dn\_data\_loader.py}}$

I write two functions

`select_signal_range(length_fault, channel_signal)`

and

`get_signal_cut_pad(length_fault, channel_signal, total_point)`

to pre-process the signal/data :

1. According to the theoretical waveform, we choose only the reflected waves (the incident wave will be discarded)
2. We pad the signal : we add 0 on both sides in order to pad the signal to the length we want, here I choose the total length after pad `total_point` as 1000 points. Here we set the data to the same length because in the convolutional neural network, we need the input data to have the same length.

Here is an example illustrates how these two functions work :

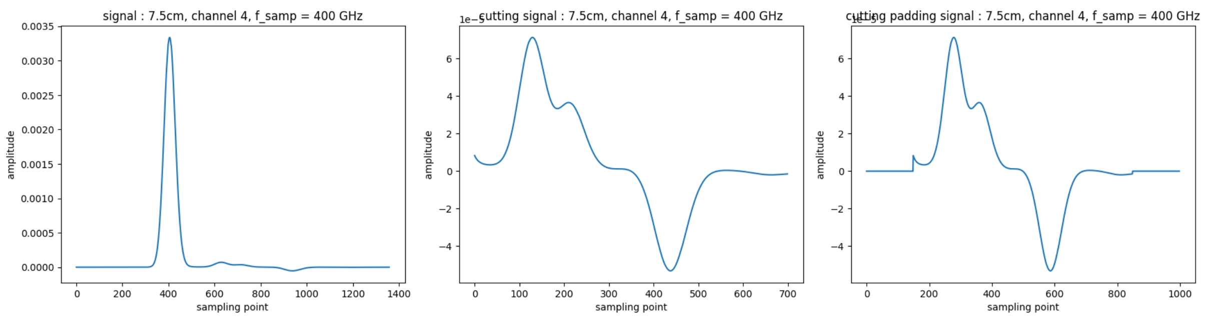


FIGURE 4.1.1 – one kind of theoretical waveform | cut : save the reflected waves | pad to 1000 points

4.2 Load waves : add Gaussian noise on the pre-processed theoretical wave

Then we will generate the training data, tuning data and testing data meanwhile we will add the Gaussian noise on each pre-processed theoretical wave. Still in the

$\boxed{\text{Listing 20 dn\_data\_loader.py}}$

1. We set the training data to occur $\frac{6}{10}$, the tuning data to occur $\frac{2}{10}$ and the testing data to occur $\frac{2}{10}$:

   `training_fraction = .6; tuning_fraction = .2; testing_fraction = .2`

2. We initialize the training data, tuning data and testing data to be the zero array :

   `training_data = []; tuning_data = []; testing_data = []`

3. We pull all the examples with different spectral densities added : here I choose the array of the spectral density as `array_dsp = np.linspace(5e-10, 5e-6, 40)`. `5e-6` cause we have seen the classic method (like Adaptive filter or STFT) reaches its limit when the spectral density takes 10^{-7}, we hope this Convolutionnal Neural Network could learn a little further :
   — For each pre-processed theoretical wave, we add the « Defect » example (signal noise + defect) :

   `examples.append((sig_noise_with_defect[np.newaxis, :], "Defect"))`

   — Meanwhile (in the same loop), we add the « Normal » example (just the noise) :

   `examples.append((sig_noise_without_defect[np.newaxis, :], "Normal"))`

— Once all added, random the full examples :

`np.random.shuffle(examples)`

4. Before divide the full examples into 3 parts, we define `class_count` as the number of total examples that we choose for each classe to put into the whole `training_data`, `tuning_data` and `testing_data`.
5. Then it will be divided into 3 groups with the fraction `training_fraction = .6; tuning_fraction = .2; testing_fraction = .2;`

All of this is the explaination of the function

```
load_waves(array_length_fault, array_channel_signal, total_point, array_dsp, nb_per_dsp)
```

used to generate the training data, tuning data and testing data.

4.3 Prepare the data for use in convolutional neural network

In order to use the generated data in the convolutional neural network, we write

Listing 21 dn_data_block.py

We write 3 classes for `TrainingData`, `TuningData` and `TestingData`, in each class :
— Concern `forward_pass(self, arg)`, we return `next(self.data)` to yield the `next` example.
— Concern `backward_pass(self, arg)`, we just let it `pass`.

5 Test & Conclusion

To test all of what we have written, we use the structure (ref. FIGURE 3.4.1) where for the first convolution block `conv_0`, I choose the kernel/window size `kernel_size` as 80 and the number of `n_kernels` as 50 ; for the second convolution block `conv_1`, I choose the kernel size `kernel_size` as 6 and the number of `n_kernels` as 5.

Here the window width is important : if the window is too small we may miss the important information while if the window is too wide the important characteristic will be diluted.

Here is the result of the loss function in $\log_{10}$ during the 250000 iterations given by the `sq_loss` block :

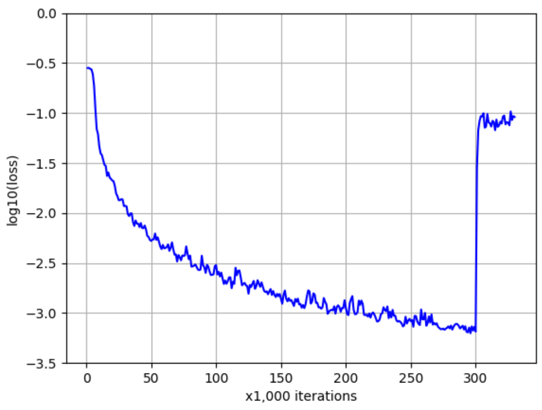


FIGURE 5.0.1 – the loss function during the iteration : training data and testing data

It is well converged that means the result after the logistic block `logit_2` well matches the label in one-hot encoding (`onehot`).

In the days to come I'm going to adjust the parameters of the convolutional neural network.

Références

[1] Brandon Rohrer. Exponential smoothing. https://e2eml.school/exponential_smoothing.html, 2020.
[2] Brandon Rohrer. Online batch normalization. https://e2eml.school/online_batch_normalization.html, 2020.
[3] Brandon Rohrer. Convolutional neural networks in one dimension. https://end-to-end-machine-learning.teachable.com/, 2024.

[4] V. Chiley, I. Sharapov, A. Kosson, U. Koster, R. Reece, S. Samaniego de la Fuente, V. Subbiah, and M. James. Online normalization for training neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[5] Emoe. Radar signal processing experiment. `https://github.com/Floyd-Fish/RadarSignalProcessing_experiment/tree/main/1-signal-generation`, 2021.

[6] T. Finch. Incremental calculation of weighted mean and variance. *University of Cambridge*, 4(11-5) :41–42, 2009.

[7] S. Ioffe and C. Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[8] Wikipedia. Backpropagation in batch normalization. `https://en.wikipedia.org/wiki/Batch_normalization#Backpropagation`, 2024.

[9] Wikipedia. Exponential smoothing. `https://en.wikipedia.org/wiki/Exponential_smoothing`, 2024.

[10] Wikipedia. Intégrale de gauss. `https://fr.wikipedia.org/wiki/Int%C3%A9grale_de_Gauss#Cas_g%C3%A9n%C3%A9rique`, 2024.

[11] E. Wilczok. New uncertainty principles for the continuous gabor transform and the continuous wavelet transform. *Documenta Mathematica*, 5 :207–226, 2000.

A Table des Programmes

Listings

B Programmes

Génération du bruit aléatoire gaussian - noise.m

Listing 2 – Génération du bruit aléatoire gaussian - noise.m

```matlab
clc; clear all; close all;

% Load data
[t,s1,s2,s3,s4]=textread('Stage\DATA_STAGE_XU\Simul_Theorique_response\DataReal2cmBBB.out');

%% Generate the Gaussian distributed random noise
% Time base parameters
Ts = t(2)-t(1);                         % Time scale
Fs = 1/Ts;                              % Sampling frequency
Tstop = 1e-8;                           % Stop time
tv = (t(1) : Ts : Tstop)';              % t : Discrete time vector
N = length(tv);                         % Points
fv = ((0:N/2-1)*1/Ts/N)';               % Discrete frequency vector

% Gaussian distributed random noise
bw = 1e3;                               % Bandwidth in Hz
k = 1.5e-8;                             % Power spectral density in W/Hz
grn = wgn(N, 1, k*bw, 'linear');        % Generate 'N' samples white Gaussian noise samples
                                        % with a total power of 'k*bw' in W
                                        % (Linear power is in watts)

% Normalized amplitude spectrum in dB
grnFftAbs = 20.*log10(abs(fft(grn))./N);

figure(1);
subplot(3, 1, 1); plot(tv, grn); title('time␣domain');
xlabel('Time␣[s]'); ylabel('Amplitude␣[V]');
subplot(3, 1, 2); histogram(grn, 50); title('histogram');
xlabel('Amplitude␣[V]'); ylabel('Quantity');
subplot(3, 1, 3); plot(fv, grnFftAbs(1:N/2)); title('spectrum');
xlabel('Frequency␣[Hz]');  ylabel('Amplitude␣[dBV]'); % set(gca, 'xscale', 'log');

%% Another way
dsp = 1.5e-5;
b = sqrt(dsp) * randn(1, N);

bFftAbs = 20.*log10(abs(fft(b))./N);

figure(2);
subplot(3, 1, 1); plot(tv, b); title('time␣domain');
xlabel('Time␣[s]'); ylabel('Amplitude␣[V]');
subplot(3, 1, 2); histogram(b, 50); title('histogram');
xlabel('Amplitude␣[V]'); ylabel('Quantity');
subplot(3, 1, 3); plot(fv, bFftAbs(1:N/2)); title('spectrum');
xlabel('Frequency␣[Hz]');  ylabel('Amplitude␣[dBV]'); % set(gca, 'xscale', 'log');
```

Fonction pour le filtre adapté - adaptive_filter.m

Listing 3 – Fonction pour le filtre adapté - adaptive_filter.m

```matlab
function [t,s,x,y,y_0,delta_t,s_cut] = adaptive_filter(H, dsp, signal, time, find_second_max, V
    )
% function [t, s, x, y, y_0] = adaptive_filter(H, dsp, signal, time, V)
%
% This function illustrates the adaptive filtering of the signal
% going from 0 to T, where the decision instant is at T.
%
% Inputs:   H       hypothesis
%                       H = 1 : useful signal present
%                       H = 0 : useful signal absent
%           dsp     power spectral density of white noise
%            signal   input signal
%            time     Discrete time vector of input signal
%            V        visualization
%                         V = 1 : signals are visualized
%                         V = 0 : signals are not visualized
%                         by default, the different signals are visualized (V = 1)
%            find_second_max : 1 (true) ; 0 (false)
%
% Outputs:   t        time
%            s        transmitted signal
%            x        received signal
%            y        output of the adaptive filter
%            y_0      output of the adaptive filter at the decision instant
%            delta_t  stores the vector of each move of the adaptive filter
%            s_cut    transmitted signal with the incident wave subtracted
%
% CECILE DURIEU, October 2009
%
% Adapted by XU Kaiyuan, June 2024

if nargin == 5
   V = 1;
end

%% Time base parameters
Te = time(2) - time(1);                        % Time scale
Fe = 1/Te;                                     % Sampling frequency

N = 2000;                                      % Total sample : 2N points i.e. N points left
    of 0, N points right of 0
t = (-N:N-1)*Te;                               % t : Discrete time vector

T = time(end);                                % Stop time of the input signal
N_T = round(T/Te);                             % Total sample of the input signal


%% Signal padded & Noise
% s = [zeros(1,N) signal zeros(1,N-N_T)];
s = [zeros(1,N+1) signal zeros(1,N-N_T-1)];     % Signal padded

% -1 / +1 cause signal begins when t = 2.5e-12, we add one 0 now it begins
% when t = 0.

% dsp = 1e-04; N = 2000;
b = sqrt(dsp)*randn(1,2*N);                    % Gaussian distributed random noise (std(b) =
    sqrt(dsp))
% grn = wgn(2*N, 1, dsp, 'linear');            % (std(grn) = sqrt(dsp))

x = H*[zeros(1,N+1) signal zeros(1,N-N_T-1)]+b; % Signal padded + Noise
[M,position] = max(x); m = min(x);

delta_t = [0, 0, 0];

s_cut = s;

for i = [1, 2, 3]

%% Choose the incident radial electric field as the adaptive filtre
a = 3*10^20; t0 = 0.3*10^(-9);
y0 = exp(-a*((time(1) : Te : time(end))-(t0+sum(delta_t(1:i)))).^2);

h = [zeros(1,N) y0 zeros(1,N-N_T)];

if find_second_max == 1
if i == 2
    index = (t0+delta_t(i))/Te + N + (5.75e-10 - 2e-11)/2/Te;
    x = [zeros(1,index-1), x(index:end)];
    s_cut = [zeros(1,index-1), s(index:end)];
end
```

```
end
%% Calculates the energy intercorrelation of the x and h signals by passing through the frequency domain
[k,c_xh] = co_ene(x,h);
y = Te*c_xh; y = y(N-N_T:3*N-N_T-1);                % output of the adaptive filter

%[c_xh] = co_ene2(x',h');
%y = c_xh; y = y(N-N_T:3*N-N_T-1);                  % output of the adaptive filter

[max_value, index] = max(y);
delta_t(i+1) =(index-N)*Te - T;

y_0 = y(N+N_T+1);                                  % output of the adaptive filter at the decision instant



if V == 1
   figure(i);

   subplot(411); plot(t,s,'b'); hold on;
   plot(time(1) : Te : time(end), signal,'.');
   title ('signal');
   axis([t(1) t(end) -max(s) max(s)]); grid;

   subplot(412); plot(t,x,'r');
   texte = ['signal␣+␣noise␣:␣H=' num2str(H) '␣et␣dsp=' num2str(dsp)];
   title(texte);
   axis([t(1) t(end) -max(x) max(x)]); grid;

   subplot(413); plot(t,h,'b');
   title ('adaptive␣filter');

   try
   subplot(414); plot(t,y,'g');
   xlabel('Time␣[s]'); title ('output␣of␣the␣adaptive␣filter'); hold on;
   M = max(y); m = min(y);
   plot(T*[1 1],5*[-1 1],'r'); hold off
   axis([t(1) t(end) -max(y) max(y)]); grid;
   catch ME
       fprintf('An␣error␣occurred:␣%s␣\n␣!␣Fail␣to␣detect␣the␣signal␣!␣\n', ME.message);
   end

%    subplot(515); plot(t,y,'g');
%    xlabel('Time [s]'); title ('output of the adaptive filter (zoom)'); hold on;
%    M = max(y); m = min(y);
%    plot(T*[1 1],5*[-1 1],'r'); hold off
%    axis([2e-9 3e-9 -max(y) max(y)]); grid;
   figure(i)


end
end
end
```

Exemple d'utlisation de la fonction pour le filtre adapté - AF_method.m

Listing 4 – Exemple d'utlisation de la fonction pour le filtre adapté - AF_method.m

```
clc; clear all; close all;

%% Load data
[t,s1,s2,s3,s4]=textread('Stage\DATA_STAGE_XU\Simul_Theorique_response\DataReal5cmBBB.out');

signal = s3;

%% Adaptive filtre
[t_,s,x,y,y_0,delta_t,s_cut] = adaptive_filter(1,5e-7,signal',t,1,1);
```

Reconstruiction du signal réfléchi STFT - STFT_method_v2.m

Listing 5 – Reconstruiction du signal réfléchi STFT - STFT_method_v2.m

```matlab
clc; clear all; close all;

% Load data
[t,s1,s2,s3,s4]=textread('Stage\DATA_STAGE_XU\Simul_Theorique_response\DataReal2cmBBB.out');

signal = s3;

% Subtracting the incident wave
H = 1; dsp = 5e-8;
[tv,s,x,y,y_0,delta_t,s_cut] = adaptive_filter(H,dsp,signal',t,1,1);

figure(1+3);
subplot(3,1,1); plot(tv,s,'r'); title('signal␣with␣incident␣wave');
subplot(3,1,2); plot(tv,s_cut,'r'); title('signal␣subtracting␣the␣incident␣wave');
subplot(3,1,3); plot(tv,x,'r');
texte = ['signal␣+␣noise␣:␣H=' num2str(H) '␣et␣dsp=' num2str(dsp)];
title(texte);
axis([tv(1) tv(end) -max(x) max(x)]); grid;
xlabel('Time␣[s]');
saveas(gcf, 'figure_4.png');

signal_padded = s_cut;
% grn+signal_padded = x ;
%% Time-frequency analysis : STFT
Ts = tv(2)-tv(1); Fs = 1/Ts;

figure(2+3);
subplot(211);
stft(s_cut,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512)
[so,fo,to] = stft(s_cut,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512);
title('Short-Time␣Fourier␣Transform␣:␣signal')
subplot(212);
stft(x,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512)
[s,f,t] = stft(x,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512);
title('Short-Time␣Fourier␣Transform␣:␣signal␣+␣noise')

saveas(gcf, 'figure_5.png');

So = abs(so);

% Anisotropic diffusion filtering of images
S = abs(s);
Sd = imdiffusefilt(S);

figure(3+3);
subplot(311); imagesc(So); set(gca, 'YDir', 'normal'); colorbar; title('|␣Short-Time␣Fourier␣Transform␣|␣:␣signal');
subplot(312); imagesc(S); set(gca, 'YDir', 'normal'); colorbar; title('|␣Short-Time␣Fourier␣Transform␣|␣:␣signal␣+␣noise');
ylabel("Frequency domain");
subplot(313); imagesc(Sd); set(gca, 'YDir', 'normal'); colorbar; title('imdiffusefilt␣(␣|␣Short-Time␣Fourier␣Transform␣|␣)␣:␣signal␣+␣noise');
xlabel("Time domain");

saveas(gcf, 'figure_6.png');

figure(4+3);
subplot(311);
imagesc(So); axis xy; axis([50 80 150 350]); title('[zoom]␣|␣Short-Time␣Fourier␣Transform␣|␣:␣signal');
subplot(312);
imagesc(S); axis xy; axis([50 80 150 350]); title('[zoom]␣|␣Short-Time␣Fourier␣Transform␣|␣:␣signal␣+␣noise');
subplot(313);
```

```
imagesc(Sd); axis xy; axis([50 80 150 350]); title('[zoom]␣imdiffusefilt␣(␣|␣Short-Time␣Fourier␣Transform␣|␣)␣:␣signal␣+␣noise');

saveas(gcf, 'figure_7.png');

%% Contour detect
[max_global,linear_index] = max(Sd(:));
[row_opt, col_opt] = ind2sub(size(Sd), linear_index);
[row, col] = size(Sd); row_middle = round(row/2);
Imag = zeros(row, col);
Imag(row_opt, col_opt) = Sd(row_opt, col_opt);

% down
i = 1;
while Sd(row_opt-i, col_opt) < Sd(row_opt-i+1, col_opt)
    %Imag(row_opt-i, col_opt) = Sd(row_opt-i, col_opt);
    i = i + 1;
end
row_down = row_opt - i ;
for i = row_middle:-1:row_down
    Imag(i, col_opt) = Sd(i, col_opt);
end

% up
row_up = 2*row_middle - row_down;
for i = row_middle:1:row_up
    Imag(i, col_opt) = Sd(i, col_opt);
end

% the rest %
% left
j = 1;
for R = row_down : row_up
    while Sd(R, col_opt-j-1) < Sd(R, col_opt-j+1)
       Imag(R, col_opt-j) = Sd(R, col_opt-j);
       j = j + 1;
    end
    j = 1;
end
% right
j = 1;
for R = row_down : row_up
    while Sd(R, col_opt+j+1) < Sd(R, col_opt+j-1)
       Imag(R, col_opt+j) = Sd(R, col_opt+j);
       j = j + 1;
    end
    j = 1;
end
figure(5+3);
subplot(311); imagesc(So); axis xy; title('|␣Short-Time␣Fourier␣Transform␣|␣:␣signal');
subplot(312); imagesc(Sd); axis xy; title('imdiffusefilt␣(␣|␣Short-Time␣Fourier␣Transform␣|␣)␣:␣signal␣+␣noise');
subplot(313); imagesc(Imag); axis xy; title('contour␣:␣imdiffusefilt␣(␣|␣Short-Time␣Fourier␣Transform␣|␣)␣:␣signal␣+␣noise');

saveas(gcf, 'figure_8.png');

figure(6+3);
subplot(311); imagesc(So); axis xy; axis([50 80 150 350]); title('[zoom]␣|␣|␣Short-Time␣Fourier␣Transform␣|␣:␣signal');
subplot(312); imagesc(Sd); axis xy; axis([50 80 150 350]); title('[zoom]␣|␣imdiffusefilt␣(␣|␣Short-Time␣Fourier␣Transform␣|␣)␣:␣signal␣+␣noise');
subplot(313); imagesc(Imag); axis xy; axis([50 80 150 350]); title('[zoom]␣contour␣:␣imdiffusefilt␣(␣|␣Short-Time␣Fourier␣Transform␣|␣)␣:␣signal␣+␣noise');

saveas(gcf, 'figure_9.png');

%% Fill each factor of STFT in contour detect
Coef = zeros(row, col);
```

```
%Coef(row_opt, col_opt) = Sd(row_opt, col_opt);
% down
i = 1;
while Sd(row_opt-i, col_opt) < Sd(row_opt-i+1, col_opt)
    %Imag(row_opt-i, col_opt) = Sd(row_opt-i, col_opt);
    i = i + 1;
end
row_down = row_opt - i ;
for i = row_middle:-1:row_down
    Coef(i, col_opt) = s(i, col_opt);
end

% up
row_up = 2*row_middle - row_down;
for i = row_middle:1:row_up
    Coef(i, col_opt) = s(i, col_opt);
end

% the rest %
% left
j = 1;
for R = row_down : row_up
    while Sd(R, col_opt-j-1) < Sd(R, col_opt-j+1)
       Coef(R, col_opt-j) = s(R, col_opt-j);
       j = j + 1;
    end
    j = 1;
end
% right
j = 1;
for R = row_down : row_up
    while Sd(R, col_opt+j+1) < Sd(R, col_opt+j-1)
       Coef(R, col_opt+j) = s(R, col_opt+j);
       j = j + 1;
    end
    j = 1;
end

%% ISTFT ideal case : signal (a test of function istft())
% [so,fo,to] = stft(signal_padded,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512);
so_reconstructed = istft(so,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512);
figure(7+3);
subplot(2, 1, 1);
plot(tv, signal_padded);
title('time␣domain␣:␣signal\_padded');
xlabel('Time␣[s]');
ylabel('Amplitude␣[V]');

subplot(2, 1, 2);
plot(tv, so_reconstructed);
%plot(0:1/Fs:(length(so_reconstructed)-1)/Fs, so_reconstructed);
title('time␣domain␣:␣signal\_reconstructed');
xlabel('Time␣[s]');
ylabel('Amplitude␣[V]');

saveas(gcf, 'figure_10.png');

%% ISTFT real case : signal reconstructed
% [so,fo,to] = stft(signal_padded,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512);
s_reconstructed = istft(Coef,Fs,Window=kaiser(256,5),OverlapLength=220,FFTLength=512);
figure(8+3);
subplot(3, 1, 1);
plot(tv,x,'red'); hold on;
plot(tv,signal_padded,'blue');
axis([tv(1) tv(end) -max(x) max(x)]);
text = ['signal␣+␣white␣Gaussian␣noise(␣dsp␣=␣', num2str(dsp), ')'];
title(text);
xlabel('Time␣[s]'); ylabel('Amplitude');

```

```
subplot(3, 1, 2);
plot(tv, signal_padded);
title('time␣domain␣:␣signal\_padded');
xlabel('Time␣[s]');
ylabel('Amplitude␣[V]');

subplot(3, 1, 3);
plot(tv, s_reconstructed);
%plot(0:1/Fs:(length(s_reconstructed)-1)/Fs, s_reconstructed);
title('time␣domain␣:␣signal\_reconstructed');
xlabel('Time␣[s]');
ylabel('Amplitude␣[V]');

saveas(gcf, 'figure_11.png');
```

Convolution in Python - convolve_1d.py

Listing 6 – Convolution in Python - convolve_1d.py

```
@njit
def convolve_1d(signal, kernel):
    """
    Numba acceleration cuts computation time down by a factor of 30.

    Head to head with NumPy's convolve(), this function comes in
    about 25% slower. Not a bad trade-off for all the flexibility
    that buys.
    """
    n_sig = signal.size
    n_ker = kernel.size
    n_conv = n_sig - n_ker + 1           # mode = 'valid'
    # Precalculating the reversed kernel cuts the computation time down
    # by a factor of 3.
    rev_kernel = kernel[::-1].copy()     # flip the kernel from left to right
    return xcorr_1d(signal, rev_kernel, n_conv)
```

Cross-correlation in Python - xcorr_1d.py

Listing 7 – Cross-correlation in Matlab - xcorr_1d.py

```
@njit
def xcorr_1d(signal, kernel, n_steps=None):
    """
    Calculate n_steps of the sliding dot product,
    a.k.a. the cross-correlation,
    between a one dimensional signal and a one dimensional kernel.

    Start with the beginning (zeroth elements) of the kernel and signal
    aligned.
    Shift the kernel up by one position each iteration.
    """
    if n_steps is None:
        n_steps = signal.size - kernel.size + 1

    result = np.zeros(n_steps, dtype=np.double)
    n_ker = kernel.size
    for i in range(n_steps):
        # Using np.dot() instead of np.sum() over the products cuts
        # the computation time down by a factor of 5.
        result[i] = np.dot(signal[i: i + n_ker], kernel)
    return result
```

Initialize the convolution block - __init__.py

Listing 8 – Initialize the convolution block - __init__.py

```
    def __init__(
        self,
        initializer=LSUV(),
        kernel_size=3,

        l1_param=None,
        l1_threshold=None,
        l2_param=None,


        n_kernels=5,
        optimizer=Momentum(learning_rate=1e-3),
    ):
        # Ensure this is odd
        self.kernel_half = int(kernel_size / 2)
        self.kernel_size = 2 * self.kernel_half + 1

        self.n_channels = None
        self.n_inputs = None
        self.n_kernels = n_kernels
        self.n_outputs = None
        self.weights = None
        ## self.bias = None


        self.l1_regularization_param = l1_param
        self.l1_regularization_threshold = l1_threshold
        self.l2_regularization_param = l2_param


        self.initializer = deepcopy(initializer)

        #### self.optimizer = deepcopy(optimizer)
        self.weight_optimizer = deepcopy(optimizer)
        ## self.weight_optimizer = deepcopy(optimizer)
        ## self.bias_optimizer = deepcopy(optimizer)

        ## self.result = None

        self.forward_in = None
        self.forward_out = None
        self.backward_in = None
        self.backward_out = None
```

Initialize the convolution block - initialize.py

Listing 9 – Initialize the convolution block - initialize.py

```
    def initialize(self):
        """
        Choose random weights for kernel values.

        The initializers expect a 2D array of weights.
        In particular the LSUV initializer will control for the variance
        along each row of the array.

        For CNNs, we would like each convolution result to have
        a variance of about 1, given an input variance of 1. Because the entire
        stack of kernels is added together, we want to treat all the
        kernel values in a stack as a single group when initializing.
        To make sure this happens, we flatten them into a single row.

        After initialization, we need to do some reshaping and swapping
        of dimensions to get the weights into the format we need.
        Dimension 0 ~ input channels (n_channels)
        Dimension 1 ~ kernel values (kernel_size)
        Dimension 2 ~ output channels (n_kernels)
        """
```

```python

        self.n_channels, self.n_inputs = self.forward_in.shape

        ## self.n_channels = self.x.shape[0]
        ## self.n_inputs = self.x.shape[1]

        self.n_outputs = self.n_inputs - self.kernel_size + 1

        weights_unshaped = self.initializer.initialize(
            self.n_kernels, self.n_channels * self.kernel_size)
        weights = np.reshape(
            weights_unshaped,
            (self.n_kernels, self.n_channels, self.kernel_size),
            order='C')
        self.weights = weights.transpose(1, 2, 0)

        # Initialize one bias parameter per output value.
        # Leave these at zero for now.

        ## self.bias = np.zeros((self.n_kernels, self.n_outputs))
```

Dunder Str function - __str__.py

Listing 10 – Dunder Str function - __str__.py

```python
        def __str__(self):
        """
        Make a descriptive, human-readable string for this layer.
        """
        str_parts = [
            "convolutional,␣one␣dimensional",
            f"number␣of␣inputs:␣{self.n_inputs}",
            f"number␣of␣channels:␣{self.n_channels}",
            f"number␣of␣outputs:␣{self.n_outputs}",
            f"number␣of␣kernels:␣{self.n_kernels}",
            f"kernel␣size:␣{self.kernel_size}␣pixels",
            f"l1␣regularization␣parameter:␣{self.l1_regularization_param}",
            f"l1␣floor␣threshold:␣{self.l1_regularization_threshold}",
            f"l2␣regularization␣parameter:␣{self.l2_regularization_param}",
            "initialization:" + tb.indent(self.initializer.__str__()),
            #### "weight optimizer:" + tb.indent(self.optimizer.__str__()),
            "weight␣optimizer:" + tb.indent(self.weight_optimizer.__str__()),
            ## "weight optimizer:" + tb.indent(self.weight_optimizer.__str__()),
            ## "bias optimizer:" + tb.indent(self.bias_optimizer.__str__()),
        ]
        return "\n".join(str_parts)
```

Forward and backward pass - forward_backward_pass.py

Listing 11 – Forward and backward pass - forward_backward_pass.py

```python
    def forward_pass(self, forward_in):
        """
        Propagate the inputs forward through the network.
        """
        # Make sure the input array is C-ordered in memory.
        # This helps the Numba code below to run the dot() function
        # much faster.
        self.forward_in = np.array(forward_in, order="C")
        if self.weights is None:
            self.initialize()

        ## self.forward_out = calculate_outputs(self.forward_in, self.weights) + self.bias
        self.forward_out = calculate_outputs(self.forward_in, self.weights)
        return self.forward_out

    def backward_pass(self, backward_in): ## dL_dy
```

```
        """
        Propagate the outputs back through the layer.
        """
        self.backward_in = backward_in
        if self.backward_in is None:
            self.backward_out = None
            return self.backward_out

        # Pad the output gradient so that it's prepared to calculate
        # the input and weight gradients.
        # Add the kernel length, less 1, to each end of axis 0.
        dL_dy = np.pad(self.backward_in, (
            (0,0),
            # (self.weights.shape[0] - 1, self.weights.shape[0] - 1)))
            (self.weights.shape[1] - 1, self.weights.shape[1] - 1)))
        ## print(self.weights.shape[1])

        if self.weight_optimizer.learning_rate > 0:
            ## print(dL_dy.shape)
            ## print(self.forward_in.shape)

            self.dL_dw = calculate_weight_gradient(dL_dy, self.forward_in)
            # l1 regularization
            if self.l1_regularization_param is not None:
                self.dL_dw += (
                    np.sign(self.weights) * self.l1_regularization_param)

            # l2 regularization
            if self.l2_regularization_param is not None:
                self.dL_dw += 2 * self.weights * self.l2_regularization_param

            self.weight_optimizer.update(self.weights, self.dL_dw)

            # Beta-LASSO normalization
            if self.l1_regularization_threshold is not None:
                weight_threshold = (
                    self.l1_regularization_threshold *
                    self.weight_optimizer.learning_rate)
                self.weights[np.where(
                    np.abs(self.weights) <= weight_threshold)] = 0


        # Bias gradient is equal to the output gradient
        ## dL_db = dL_dy
        ## dL_dw = calculate_weight_gradient(dL_dy, self.x)

        ## self.weight_optimizer.update(self.weights, dL_dw)
        ## self.bias_optimizer.update(self.bias, dL_db)

        dL_dx = calculate_input_gradient(dL_dy, self.weights)
        self.backward_out = dL_dx
        ## self.dL_dx = calculate_input_gradient(dL_dy, self.weights)
        return self.backward_out


@njit
def calculate_outputs(inputs, kernel_set):
    """
    Compute the multichannel convolutions for a collection of kernels
    and return the assembled result.

    inputs is a 2D array of floats (n_channels, n_inputs) and
    kernel_set is a 3D array of floats (n_channels, kernel_size, n_kernels)

    result will be a 2D array of floats
        (n_kernels, n_inputs - kernel_size + 1)
    """

    n_kernels = kernel_set.shape[2]
    result = np.zeros((
        n_kernels,
        inputs.shape[1] - kernel_set.shape[1] + 1))
    for i_kernel in range(n_kernels):
        result[i_kernel, :] = calculate_single_kernel_output(
            inputs, kernel_set[:, :, i_kernel])
    return result


@njit
def calculate_single_kernel_output(signal, kernel):
    """
    signal and kernel are 2 dimensional array of floats.
    Each row (dimension 0) represents a separate
    channel. signal and kernel must have the same number of rows.

    For now, all convolutions are "valid" mode, meaning that they are
    only computed for locations in which the kernel fully overlaps the
    the signal. This means that the result will be shorter than the
    signal by the (length of the kernel - 1).

    This seems like a good default behavior since it doesn't involve
    padding. Padding implies fabrication of extra data on the head and
    tail of the signal which comes with a number of pitfalls and,
    as far as I can see at the moment, not many big advantages.
    """
    result = np.zeros(signal.shape[1] - kernel.shape[1] + 1)
    for i_channel in range(signal.shape[0]):
        result += convolve_1d(signal[i_channel, :], kernel[i_channel, :])
    return result


@njit
def calculate_weight_gradient(output_grad_padded, inputs):
    """
    Compute the partial derivative of the loss function (the overall error)
    with respect to the kernel weights. This is
    a multichannel cross-correlation between output_gradients
    (the partial derivative of the loss with respect to
    the pre-activation function outputs) and the inputs (x).

    kernel_half is (kernel_width - 1) / 2
    inputs is a 2D array of floats shaped as (n_channels, n_inputs)
    output_grad  is a 2D array of floats shape as (n_kernels, n_outputs)
    n_outputs  = n_inputs - kernel_width + 1

    result will be a 3D array of floats shaped as
        (n_channels, kernel_size, n_kernels)
    """
    n_kernels = output_grad_padded.shape[0]
    n_channels = inputs.shape[0]
    ## kernel_width = inputs.shape[1] - output_grad_padded.shape[1] + 1
    kernel_width = - inputs.shape[1] + output_grad_padded.shape[1] + 1
    result = np.zeros((
        n_channels,
        kernel_width,
        n_kernels))

    for i_kernel in range(n_kernels):
        result[:, :, i_kernel] = calculate_single_kernel_weight_gradient(
            inputs, output_grad_padded[i_kernel, :])
    return result


@njit
def calculate_single_kernel_weight_gradient(inputs, output_grad_padded):
    """

    inputs is a 2D array of all the layer's inputs shaped like
```

```python
        (n_channels, n_inputs)
    output_grad is a 1D array of outputs from a single kernel shaped like
        (n_outputs) where n_inputs - n_outputs + 1 is the kernel width

    result is the single kernel weight gradients across all channels,
        shaped like (n_channels, kernel_width)
    """
    n_channels = inputs.shape[0]
    n_inputs = inputs.shape[1]
    n_outputs = output_grad_padded.size
    ## kernel_width = n_inputs - n_outputs + 1
    kernel_width = n_outputs - n_inputs + 1

    result = np.zeros((n_channels, kernel_width))
    for i_channel in range(n_channels):
        result[i_channel, :] = xcorr_1d(output_grad_padded, inputs[i_channel, :])
    return result


@njit
def calculate_input_gradient(output_grad_padded, kernel_set):
    """
    Compute the partial derivaticve of the loss function with respect to
    each of the inputs. This is a multichannel cross-correlation
    between output_gradients
    (the partial derivative of the loss with respect to
    the pre-activation function outputs) and the kernel weights.


    n_inputs  = n_outputs + kernel_width - 1
    kernel_set is a 3D array of floats shaped as
        (n_channels, kernel_size, n_kernels)
    output_grad  is a 2D array of floats shape as (n_kernels, n_outputs)

    result is shaped like inputs, a 2D array of floats
        shaped as (n_channels, n_inputs)
    """
    n_channels = kernel_set.shape[0]
    kernel_size = kernel_set.shape[1]
    n_kernels = output_grad_padded.shape[0]
    n_outputs_pad = output_grad_padded.shape[1]
    ## n_inputs = n_outputs + kernel_size - 1
    n_inputs = n_outputs_pad - kernel_size + 1

    # Pad out the output gradient so that the cross-correlation gives
    # the right number of results for the inputs.
    # Add the width of the kernel, less 1, to each end of axis 1.
    ## output_grad_padded = np.zeros(
        ## (n_kernels, n_outputs + 2 * (kernel_size - 1)))
    # The site of an infamous bug, now corrected
    # Was just:
    # output_grad_padded[ :, kernel_size - 1: n_outputs + kernel_size - 1]
    ## output_grad_padded[
        ## :, kernel_size - 1: n_outputs + kernel_size - 1] = output_grad

    result = np.zeros((n_channels, n_inputs))

    for i_kernel in range(n_kernels):
        result += calculate_single_kernel_input_gradient(
            np.copy(output_grad_padded[i_kernel, :]),
            np.copy(kernel_set[:, :, i_kernel])
        )
    return result

@njit
def calculate_single_kernel_input_gradient(output_grad_padded, kernel):
    """
    n_outputs_padded is n_outputs + 2 * (kernel_size - 1)
    kernel is the single kernel weight gradients across all channels,
        shaped like (n_channels, kernel_size)
    output_grad_padded is a 1D array of outputs from a single kernel
        shaped like (n_outputs_padded)
    n_inputs is n_outputs_padded - kernel_size + 1

    result is a 2D array of all the layer's inputs shaped like
        (n_channels, n_inputs)
    """
    n_channels = kernel.shape[0]
    kernel_size = kernel.shape[1]
    n_outputs_padded = output_grad_padded.size
    n_inputs = n_outputs_padded - kernel_size + 1

    result = np.zeros((n_channels, n_inputs))
    for i_channel in range(n_channels):
        result[i_channel, :] = xcorr_1d(
            output_grad_padded, kernel[i_channel, :])
    return result
```

class Conv1D - Conv1D.py

Listing 12 – class Conv1D - Conv1D.py

```
from copy import deepcopy
from numba import njit
import numpy as np
from cottonwood.core.initializers import LSUV
from cottonwood.core.optimizers import Adam, Momentum
import cottonwood.core.toolbox as tb


class Conv1D(object):
    """
    A one-dimensional (1D) convolutional layer, ready for training with
    backpropagation.

    For a detailed derivation of what this layer does and why:
        https://e2eml.school/convolution_one_d.html

    As input, it expects a two-dimensional (2D) numpy array of floats,
    with shape (n_channels, n_inputs) where
    n_inputs is the length of the 1D signal. This is arbitrary and
        will be specific to the data set.
    n_channels is the number of parallel channels in the inputs.
        For example a time series of 5 different stock prices would be
        a 5-channel input. A set of 128 EEG electrode recordings would
        be 128 channels.

    For output, it will produce a 2D numpy array of floats,
    with shape (n_kernels, n_outputs) where
    n_outputs is the length of the 1D signal after convolution. For now,
        all convolutions are "valid" style, meaning that they are only
        calculated for cases where the kernel overlaps completely with
        the signal. As a result, for a kernel of length n_kernel
            n_outputs = n_inputs - n_kernel + 1
    n_kernels is the number of separate kernels used. This is an
        arbitrary hyperparameter chosen during the initialization of the layer.
    """
    def __init__(
        self,
        initializer=LSUV(),
        kernel_size=3,

        l1_param=None,
        l1_threshold=None,
        l2_param=None,


        n_kernels=5,
        optimizer=Momentum(learning_rate=1e-3),
    ):
        # Ensure this is odd
        self.kernel_half = int(kernel_size / 2)
        self.kernel_size = 2 * self.kernel_half + 1

        self.n_channels = None
        self.n_inputs = None
        self.n_kernels = n_kernels
        self.n_outputs = None
        self.weights = None
        ## self.bias = None


        self.l1_regularization_param = l1_param
        self.l1_regularization_threshold = l1_threshold
        self.l2_regularization_param = l2_param


        self.initializer = deepcopy(initializer)

        #### self.optimizer = deepcopy(optimizer)
        self.weight_optimizer = deepcopy(optimizer)
        ## self.weight_optimizer = deepcopy(optimizer)
        ## self.bias_optimizer = deepcopy(optimizer)

        ## self.result = None

        self.forward_in = None
        self.forward_out = None
        self.backward_in = None
        self.backward_out = None


    def initialize(self):
        """
        Choose random weights for kernel values.

        The initializers expect a 2D array of weights.
        In particular the LSUV initializer will control for the variance
        along each row of the array.

        For CNNs, we would like each convolution result to have
        a variance of about 1, given an input variance of 1. Because the entire
        stack of kernels is added together, we want to treat all the
        kernel values in a stack as a single group when initializing.
        To make sure this happens, we flatten them into a single row.

        After initialization, we need to do some reshaping and swapping
        of dimensions to get the weights into the format we need.
        Dimension 0 ~ input channels (n_channels)
        Dimension 1 ~ kernel values (kernel_size)
        Dimension 2 ~ output channels (n_kernels)
        """

        self.n_channels, self.n_inputs = self.forward_in.shape

        ## self.n_channels = self.x.shape[0]
        ## self.n_inputs = self.x.shape[1]

        self.n_outputs = self.n_inputs - self.kernel_size + 1

        weights_unshaped = self.initializer.initialize(
            self.n_kernels, self.n_channels * self.kernel_size)
        weights = np.reshape(
            weights_unshaped,
            (self.n_kernels, self.n_channels, self.kernel_size),
            order='C')
        self.weights = weights.transpose(1, 2, 0)

        # Initialize one bias parameter per output value.
        # Leave these at zero for now.

        ## self.bias = np.zeros((self.n_kernels, self.n_outputs))

    def __str__(self):
        """
        Make a descriptive, human-readable string for this layer.
        """
        str_parts = [
            "convolutional,␣one␣dimensional",
            f"number␣of␣inputs:␣{self.n_inputs}",
            f"number␣of␣channels:␣{self.n_channels}",
            f"number␣of␣outputs:␣{self.n_outputs}",
            f"number␣of␣kernels:␣{self.n_kernels}",
            f"kernel␣size:␣{self.kernel_size}␣pixels",
            f"l1␣regularization␣parameter:␣{self.l1_regularization_param}",
            f"l1␣floor␣threshold:␣{self.l1_regularization_threshold}",
            f"l2␣regularization␣parameter:␣{self.l2_regularization_param}",
            "initialization:" + tb.indent(self.initializer.__str__()),
            #### "weight optimizer:" + tb.indent(self.optimizer.__str__()),
            "weight␣optimizer:" + tb.indent(self.weight_optimizer.__str__()),
            ## "weight optimizer:" + tb.indent(self.weight_optimizer.__str__()),
            ## "bias optimizer:" + tb.indent(self.bias_optimizer.__str__()),
        ]
        return "\n".join(str_parts)

    def forward_pass(self, forward_in):
        """
        Propagate the inputs forward through the network.
        """
        # Make sure the input array is C-ordered in memory.
        # This helps the Numba code below to run the dot() function
        # much faster.
        self.forward_in = np.array(forward_in, order="C")
        if self.weights is None:
            self.initialize()

        ## self.forward_out = calculate_outputs(self.forward_in, self.weights) + self.bias
        self.forward_out = calculate_outputs(self.forward_in, self.weights)
        return self.forward_out

    def backward_pass(self, backward_in): ## dL_dy
        """
        Propagate the outputs back through the layer.
        """
        self.backward_in = backward_in
        if self.backward_in is None:
            self.backward_out = None
            return self.backward_out

        # Pad the output gradient so that it's prepared to calculate
        # the input and weight gradients.
        # Add the kernel length, less 1, to each end of axis 0.
        dL_dy = np.pad(self.backward_in, (
            (0,0),
            # (self.weights.shape[0] - 1, self.weights.shape[0] - 1)))
            (self.weights.shape[1] - 1, self.weights.shape[1] - 1)))
        ## print(self.weights.shape[1])

        if self.weight_optimizer.learning_rate > 0:
            ## print(dL_dy.shape)
            ## print(self.forward_in.shape)

            self.dL_dw = calculate_weight_gradient(dL_dy, self.forward_in)
            # l1 regularization
            if self.l1_regularization_param is not None:
                self.dL_dw += (
                    np.sign(self.weights) * self.l1_regularization_param)

            # l2 regularization
            if self.l2_regularization_param is not None:
                self.dL_dw += 2 * self.weights * self.l2_regularization_param

            self.weight_optimizer.update(self.weights, self.dL_dw)

            # Beta-LASSO normalization
            if self.l1_regularization_threshold is not None:
                weight_threshold = (
                    self.l1_regularization_threshold *
                    self.weight_optimizer.learning_rate)
                self.weights[np.where(
                    np.abs(self.weights) <= weight_threshold)] = 0


        # Bias gradient is equal to the output gradient
        ## dL_db = dL_dy
        ## dL_dw = calculate_weight_gradient(dL_dy, self.x)

        ## self.weight_optimizer.update(self.weights, dL_dw)
        ## self.bias_optimizer.update(self.bias, dL_db)

        dL_dx = calculate_input_gradient(dL_dy, self.weights)
        self.backward_out = dL_dx
        ## self.dL_dx = calculate_input_gradient(dL_dy, self.weights)
        return self.backward_out


@njit
def calculate_outputs(inputs, kernel_set):
    """
    Compute the multichannel convolutions for a collection of kernels
    and return the assembled result.

    inputs is a 2D array of floats (n_channels, n_inputs) and
    kernel_set is a 3D array of floats (n_channels, kernel_size, n_kernels)

    result will be a 2D array of floats
        (n_kernels, n_inputs - kernel_size + 1)
    """

    n_kernels = kernel_set.shape[2]
    result = np.zeros((
        n_kernels,
        inputs.shape[1] - kernel_set.shape[1] + 1))
    for i_kernel in range(n_kernels):
        result[i_kernel, :] = calculate_single_kernel_output(
            inputs, kernel_set[:, :, i_kernel])
    return result


@njit
def calculate_single_kernel_output(signal, kernel):
    """
    signal and kernel are 2 dimensional array of floats.
    Each row (dimension 0) represents a separate
    channel. signal and kernel must have the same number of rows.

    For now, all convolutions are "valid" mode, meaning that they are
    only computed for locations in which the kernel fully overlaps the
    the signal. This means that the result will be shorter than the
    signal by the (length of the kernel - 1).

    This seems like a good default behavior since it doesn't involve
    padding. Padding implies fabrication of extra data on the head and
    tail of the signal which comes with a number of pitfalls and,
    as far as I can see at the moment, not many big advantages.
    """
    result = np.zeros(signal.shape[1] - kernel.shape[1] + 1)
    for i_channel in range(signal.shape[0]):
        result += convolve_1d(signal[i_channel, :], kernel[i_channel, :])
    return result


@njit
def calculate_weight_gradient(output_grad_padded, inputs):
    """
    Compute the partial derivative of the loss function (the overall error)
    with respect to the kernel weights. This is
    a multichannel cross-correlation between output_gradients
    (the partial derivative of the loss with respect to
    the pre-activation function outputs) and the inputs (x).

    kernel_half is (kernel_width - 1) / 2
    inputs is a 2D array of floats shaped as (n_channels, n_inputs)
    output_grad  is a 2D array of floats shape as (n_kernels, n_outputs)
    n_outputs  = n_inputs - kernel_width + 1

    result will be a 3D array of floats shaped as
        (n_channels, kernel_size, n_kernels)
    """
    n_kernels = output_grad_padded.shape[0]
    n_channels = inputs.shape[0]
    ## kernel_width = inputs.shape[1] - output_grad_padded.shape[1] + 1
    kernel_width = - inputs.shape[1] + output_grad_padded.shape[1] + 1
    result = np.zeros((
        n_channels,
        kernel_width,
        n_kernels))

    for i_kernel in range(n_kernels):
        result[:, :, i_kernel] = calculate_single_kernel_weight_gradient(
            inputs, output_grad_padded[i_kernel, :])
    return result


@njit
def calculate_single_kernel_weight_gradient(inputs, output_grad_padded):
    """

    inputs is a 2D array of all the layer's inputs shaped like
        (n_channels, n_inputs)
    output_grad is a 1D array of outputs from a single kernel shaped like
        (n_outputs) where n_inputs - n_outputs + 1 is the kernel width

    result is the single kernel weight gradients across all channels,
        shaped like (n_channels, kernel_width)
    """
    n_channels = inputs.shape[0]
    n_inputs = inputs.shape[1]
    n_outputs = output_grad_padded.size
    ## kernel_width = n_inputs - n_outputs + 1
    kernel_width = n_outputs - n_inputs + 1

    result = np.zeros((n_channels, kernel_width))
    for i_channel in range(n_channels):
        result[i_channel, :] = xcorr_1d(output_grad_padded, inputs[i_channel, :])
    return result


@njit
def calculate_input_gradient(output_grad_padded, kernel_set):
    """
    Compute the partial derivaticve of the loss function with respect to
    each of the inputs. This is a multichannel cross-correlation
    between output_gradients
    (the partial derivative of the loss with respect to
    the pre-activation function outputs) and the kernel weights.


    n_inputs  = n_outputs + kernel_width - 1
    kernel_set is a 3D array of floats shaped as
        (n_channels, kernel_size, n_kernels)
    output_grad  is a 2D array of floats shape as (n_kernels, n_outputs)

    result is shaped like inputs, a 2D array of floats
        shaped as (n_channels, n_inputs)
    """
    n_channels = kernel_set.shape[0]
    kernel_size = kernel_set.shape[1]
    n_kernels = output_grad_padded.shape[0]
    n_outputs_pad = output_grad_padded.shape[1]
    ## n_inputs = n_outputs + kernel_size - 1
    n_inputs = n_outputs_pad - kernel_size + 1

    # Pad out the output gradient so that the cross-correlation gives
    # the right number of results for the inputs.
    # Add the width of the kernel, less 1, to each end of axis 1.
    ## output_grad_padded = np.zeros(
        ## (n_kernels, n_outputs + 2 * (kernel_size - 1)))
    # The site of an infamous bug, now corrected
    # Was just:
    # output_grad_padded[ :, kernel_size - 1: n_outputs + kernel_size - 1]
    ## output_grad_padded[
        ## :, kernel_size - 1: n_outputs + kernel_size - 1] = output_grad

    result = np.zeros((n_channels, n_inputs))

    for i_kernel in range(n_kernels):
        result += calculate_single_kernel_input_gradient(
            np.copy(output_grad_padded[i_kernel, :]),
            np.copy(kernel_set[:, :, i_kernel])
        )
    return result


'''
@njit
def calculate_input_gradient(output_grad, kernel_set):
    """
    Compute the partial derivaticve of the loss function with respect to
    each of the inputs. This is a multichannel cross-correlation
    between output_gradients
    (the partial derivative of the loss with respect to
    the pre-activation function outputs) and the kernel weights.


    n_inputs  = n_outputs + kernel_width - 1
    kernel_set is a 3D array of floats shaped as
        (n_channels, kernel_size, n_kernels)
    output_grad  is a 2D array of floats shape as (n_kernels, n_outputs)

    result is shaped like inputs, a 2D array of floats
        shaped as (n_channels, n_inputs)
    """
    n_channels = kernel_set.shape[0]
    kernel_size = kernel_set.shape[1]
    n_kernels = output_grad.shape[0]
    n_outputs = output_grad.shape[1]
    n_inputs = n_outputs + kernel_size - 1

    # Pad out the output gradient so that the cross-correlation gives
    # the right number of results for the inputs.
    # Add the width of the kernel, less 1, to each end of axis 1.
    output_grad_padded = np.zeros(
        (n_kernels, n_outputs + 2 * (kernel_size - 1)))
    # The site of an infamous bug, now corrected
    # Was just:
    # output_grad_padded[ :, kernel_size - 1: n_outputs + kernel_size - 1]
    output_grad_padded[
        :, kernel_size - 1: n_outputs + kernel_size - 1] = output_grad

    result = np.zeros((n_channels, n_inputs))
    for i_kernel in range(n_kernels):
        result += calculate_single_kernel_input_gradient(
            np.copy(output_grad_padded[i_kernel, :]),
            np.copy(kernel_set[:, :, i_kernel])
        )
    return result
'''


@njit
def calculate_single_kernel_input_gradient(output_grad_padded, kernel):
    """
    n_outputs_padded is n_outputs + 2 * (kernel_size - 1)
    kernel is the single kernel weight gradients across all channels,
        shaped like (n_channels, kernel_size)
    output_grad_padded is a 1D array of outputs from a single kernel
        shaped like (n_outputs_padded)
    n_inputs is n_outputs_padded - kernel_size + 1

    result is a 2D array of all the layer's inputs shaped like
        (n_channels, n_inputs)
    """
    n_channels = kernel.shape[0]
    kernel_size = kernel.shape[1]
    n_outputs_padded = output_grad_padded.size
    n_inputs = n_outputs_padded - kernel_size + 1

    result = np.zeros((n_channels, n_inputs))
    for i_channel in range(n_channels):
        result[i_channel, :] = xcorr_1d(
            output_grad_padded, kernel[i_channel, :])
    return result


@njit
def convolve_1d(signal, kernel):
    """
    Numba acceleration cuts computation time down by a factor of 30.

    Head to head with NumPy's convolve(), this function comes in
    about 25% slower. Not a bad trade-off for all the flexibility
    that buys.
    """
    n_sig = signal.size
    n_ker = kernel.size
    n_conv = n_sig - n_ker + 1
    # Precalculating the reversed kernel cuts the computation time down
    # by a factor of 3.
    rev_kernel = kernel[::-1].copy()
    return xcorr_1d(signal, rev_kernel, n_conv)


@njit
def xcorr_1d(signal, kernel, n_steps=None):
    """
    Calculate n_steps of the sliding dot product,
    a.k.a. the cross-correlation,
    between a one dimensional signal and a one dimensional kernel.

    Start with the beginning (zeroth elements) of the kernel and signal
    aligned.
    Shift the kernel up by one position each iteration.
    """
    if n_steps is None:
        n_steps = signal.size - kernel.size + 1

    result = np.zeros(n_steps, dtype=np.double)
    n_ker = kernel.size
    for i in range(n_steps):
        # Using np.dot() instead of np.sum() over the products cuts
        # the computation time down by a factor of 5.
        result[i] = np.dot(signal[i: i + n_ker], kernel)
    return result


if __name__ == "__main__":
    layer = Conv1D()
    print(layer)
```

class TrainingData - data_loader_blips.py

Listing 13 – class TrainingData - data_loader_blips.py

```
import numpy as np


def get_data_sets():
    """
    This function creates two other functions that generate data.
    One generates a training data set and the other, an evaluation set.

    Each data point is a "blip", a sequence of zeroes with a short,
    nonzero section. Blips come in three varieties, named for the
    approximate shape they take: M, N, V, and H. Each can be inverted as well.

    To use in a script:

        import data_loader_blips as dat

        training_generator, evaluation_grenerator = dat.get_data_sets()
        new_training_example = next(training_generator)
        new_evaluation_example = next(evaluation_generator)
    """

    examples = get_blips()

    def training_set():
        while True:
            i_example = np.random.choice(range(len(examples)))
            yield examples[i_example]

    def evaluation_set():
        while True:
            i_example = np.random.choice(range(len(examples)))
            yield examples[i_example]

    return training_set(), evaluation_set()


def get_blips():
    """
    Blips have four flavors, M, N, V, and H.
    Generate equal numbers of each.
    """
    np.random.seed(87)
    blips = []
    # The length of the signal
    # example_length = 41
    example_length = 21
    # The length of the nonzero section of the signal
    blip_length = 7
    # The total number of examples to generate
    n_examples = 100

    flavors = {
        "M": np.array([1, .7, .4, .1, .4, .7, 1]),
        "V": np.array([-.1, -.4, -.7, -1, -.7, -.4, -.1]),
        "N": np.array([-.7, .7, .4, 0, -.4, -.7, .7]),
        "H": np.array([1, 0, 0, 0, 0, 0, -1]),
    }

    def generate_example(blip):
        example = np.zeros(example_length)
        i_start = np.random.choice(example_length - blip_length - 1)
        example[i_start: i_start + blip_length] = blip
        # Ensure that the example is two dimensional
        # (one channel rows by example_length cols)
        return example[np.newaxis, :]

    for _ in range(n_examples):
        # Generate tuples of (example, label)
        # This willl come in handy when it comes time to do classification
        blips.append((generate_example(flavors["M"]), "M"))
        blips.append((generate_example(flavors["V"]), "V"))
        blips.append((generate_example(flavors["N"]), "N"))
        blips.append((generate_example(flavors["H"]), "H"))

    return blips


class TrainingData(object):
    def __init__(self):
        self.training_data_generator, _ = get_data_sets()

    def __str__(self):
        return "blips␣training␣data"

    def forward_pass(self, *arg):
        return next(self.training_data_generator)

    def backward_pass(self, *arg):
        return None


class EvaluationData(object):
    def __init__(self):
        _, self.evaluation_data_generator = get_data_sets()

    def __str__(self):
        return "blips␣evaluation␣data"

    def forward_pass(self, *arg):
        return next(self.evaluation_data_generator)

    def backward_pass(self, *arg):
        return None


if __name__ == "__main__":
    """
    To run a quick test, navigate to the directory containing this module and:
        python3 -m data_loader_blips
    """
    training_block = TrainingData()
    evaluation_block = EvaluationData()
    for _ in range(10):
        new_training_example = training_block.forward_pass()
        print(new_training_example)
    for _ in range(10):
        new_evaluation_example = evaluation_block.forward_pass()
        print(new_evaluation_example)
```

class OneHot - operations.py

Listing 14 – class OneHot - operations.py

```
class OneHot(object):
    """
    Convert a string into an array of zeros with just one 1.
    """
    def __init__(self, n_categories):
        self.n_categories = n_categories
        self.categories = {}
        self.result = None

    def __str__(self):
        return "one␣hot"

    def forward_pass(self, label):
        self.result = np.zeros(self.n_categories)
        if label in self.categories.keys():
            self.result[self.categories[label]] = 1
        else:
            n_cats_so_far = len(self.categories.keys())
            if n_cats_so_far < self.n_categories:
                self.categories[label] = n_cats_so_far
            self.result[n_cats_so_far] = 1
        return self.result

    def backward_pass(self, values):
        # Pass through any values that come back this way
        return values
```

class Flatten - operations.py

Listing 15 – class Flatten - operations.py

```
class Flatten(object):
    """
    Take in an n-dimensional array and return two dimensional array
    with just one row.
    """
    def __init__(self):
        self.input_shape = None

    def __str__(self):
        return "flatten"

    def forward_pass(self, values):
        self.input_shape = values.shape
        return values.ravel()[np.newaxis, :]

    def backward_pass(self, flat_values):
        return flat_values.reshape(self.input_shape)
```

class ValueLogger - logger.py

Listing 16 – class ValueLogger - logger.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
plt.switch_backend("agg")
```

```python


class ValueLogger(object):
    """
    Track a particular value, iteration-by-iteration.
    Save the entire history of the value as a csv and
    plot the value history in a png.
    """
    def __init__(
        self,
        n_iter_report=1e4,
        log_scale=False,
        report_max=None,
        report_min=None,
        report_name=None,
        reporting_bin_size=1e3,
        reports_path="reports",
        value_name="value",
    ):
        self.value_history = []
        self.i_iter = 0
        self.log_scale = log_scale
        self.n_iter_report = n_iter_report
        self.report_min = report_min
        self.report_max = report_max
        self.reporting_bin_size = int(reporting_bin_size)
        self.value_name = value_name
        self.reports_path = reports_path
        if report_name is None:
            self.report_name = f"report_{self.value_name}"

    def log_value(self, value):
        """
        Grab a copy of the value from each iteration.
        """
        self.value_history.append(value)
        self.i_iter += 1
        if self.i_iter % self.n_iter_report == 0:
            self.report()
            self.write()

    def report(self):
        """
        Create a plot of the loss history.
        """
        n_bins = int(len(self.value_history) // self.reporting_bin_size)
        smoothed_history = []
        for i_bin in range(n_bins):
            smoothed_history.append(np.mean(self.value_history[
                i_bin * self.reporting_bin_size:
                (i_bin + 1) * self.reporting_bin_size
            ]))
        if self.log_scale:
            value_history = np.log10(np.array(smoothed_history) + 1e-10)
        else:
            value_history = np.array(smoothed_history)

        if self.report_min is None:
            ymin = np.min(value_history)
        else:
            ymin = np.minimum(self.report_min, np.min(value_history))
        if self.report_max is None:
            ymax = np.max(value_history)
        else:
            ymax = np.maximum(self.report_max, np.max(value_history))

        fig = plt.figure()
        ax = plt.gca()
        ax.plot(
            np.arange(len(value_history)) + 1,
            value_history,
            color="blue",
        )
        ax.set_xlabel(f"x{self.reporting_bin_size:,}␣iterations")
        if self.log_scale:
            ax.set_ylabel(f"log10({self.value_name})")
        else:
            ax.set_ylabel(f"{self.value_name}")
        ax.set_ylim(ymin, ymax)
        ax.grid()
        fig.savefig(os.path.join(
            self.reports_path, self.report_name + ".png"))
        plt.close()

    def write(self):
        """
        Write the value history to a csv.
        """
        with open(os.path.join(
                self.reports_path, self.report_name + ".csv"), "w") as f:
            for value in self.value_history:
                f.write(f"{value}\n")
```

Training, evaluation and reporting - blip_demo.py

Listing 17 – Training evaluation and reporting - blip_demo.py

```
import os
from cottonwood.core.blocks.activation import Logistic, TanH
from cottonwood.core.blocks.conv1d import Conv1D
from cottonwood.core.blocks.linear import Linear
from cottonwood.core.blocks.loss import MeanSquareLoss
from cottonwood.core.blocks.operations import Difference, Flatten, OneHot
from cottonwood.core.blocks.structure import Structure
from cottonwood.core.logger import ValueLogger
import cottonwood.core.toolbox as tb
from cottonwood.data.data_loader_blips import TrainingData, EvaluationData
import cottonwood.examples.convnet.conv1d_viz as conv_viz
import cottonwood.examples.simulation.visualize_structure as struct_viz


def run():
    reports_dir = os.path.join("reports", tb.date_string())
    os.makedirs(reports_dir, exist_ok=True)

    msg = f"""

Running convolutional neural network demo on the blips data set.
Look for documentation and visualizations
in the {reports_dir} directory.

"""
    print(msg)

    n_training_iter = int(1e5)
    n_evaluation_iter = int(1e5)
    n_report_interval = int(1e4)
    n_viz_interval = int(1e5)

    convnet = Structure()
    convnet.add(TrainingData(), "train")
    convnet.add(OneHot(4), "onehot")

    # Create two convolutional layers of different sizes
    kernel_size = [5, 7]
    n_kernels = [15, 12]
    convnet.add(
```

```python
        Conv1D(kernel_size=kernel_size[0], n_kernels=n_kernels[0]), "conv_0")
    convnet.add(TanH(), "tanh_0")
    convnet.add(
        Conv1D(kernel_size=kernel_size[1], n_kernels=n_kernels[1]), "conv_1")
    convnet.add(TanH(), "tanh_1")
    convnet.add(Flatten(), "flat")
    convnet.add(Linear(4), "lin_2")
    convnet.add(Logistic(), "logit_2")
    convnet.add(Difference(), "diff")
    convnet.add(MeanSquareLoss(), "sq_loss")

    convnet.connect(
        tail_block="train", i_port_tail=0, head_block="conv_0", i_port_head=0)
    convnet.connect(
        tail_block="train", i_port_tail=1, head_block="onehot", i_port_head=0)
    convnet.connect("conv_0", "tanh_0")
    convnet.connect("tanh_0", "conv_1")
    convnet.connect("conv_1", "tanh_1")
    convnet.connect("tanh_1", "flat")
    convnet.connect("flat", "lin_2")
    convnet.connect("lin_2", "logit_2")
    convnet.connect(
        tail_block="logit_2", i_port_tail=0, head_block="diff", i_port_head=0)
    convnet.connect(
        tail_block="onehot", i_port_tail=0, head_block="diff", i_port_head=1)
    convnet.connect("diff", "sq_loss")

    loss_logger = ValueLogger(
        value_name="loss",
        log_scale=True,
        n_iter_report=n_report_interval,
        report_min=-1,
        report_max=0,
        reports_path=reports_dir,
        reporting_bin_size=1e3,
    )

    for i_iter in range(n_training_iter):
        convnet.forward_pass()
        convnet.backward_pass()
        loss_logger.log_value(convnet.blocks["sq_loss"].loss)
        if (i_iter + 1) % n_viz_interval == 0:
            conv_viz.render(
                convnet.blocks["conv_0"],
                reports_dir,
                f"conv_0_{i_iter␣+␣1:07}.png")
            conv_viz.render(
                convnet.blocks["conv_1"],
                reports_dir,
                f"conv_1_{i_iter␣+␣1:07}.png")
    tb.summarize(convnet, reports_dir=reports_dir)
    struct_viz.render(convnet, reports_dir)

    convnet.remove("train")
    convnet.add(EvaluationData(), "eval")
    convnet.connect(
        tail_block="eval", i_port_tail=0, head_block="conv_0", i_port_head=0)
    convnet.connect(
        tail_block="eval", i_port_tail=1, head_block="onehot", i_port_head=0)

    for i_iter in range(n_evaluation_iter):
        convnet.forward_pass()
        loss_logger.log_value(convnet.blocks["sq_loss"].loss)
        if (i_iter + 1) % n_viz_interval == 0:
            conv_viz.render(
                convnet.blocks["conv_0"],
                reports_dir,
                f"conv_0_{n_training_iter␣+␣i_iter␣+␣1:07}.png")
            conv_viz.render(
                convnet.blocks["conv_1"],
                reports_dir,
                f"conv_1_{n_training_iter␣+␣i_iter␣+␣1:07}.png")


if __name__ == "__main__":
    run()
```

class ReLU - activation.py

Listing 18 – class ReLU - activation.py

```
class ReLU(object):
    def __init__(self):
        self.result = None

    def __str__(self):
        return "rectified␣linear␣unit"

    def forward_pass(self, values):
        self.result = np.maximum(0, values)
        return self.result

    def backward_pass(self, grad):
        d_relu = np.zeros(self.result.shape)
        d_relu[np.where(self.result > 0)] = 1
        return grad * d_relu
```

class MaxPool1D - pooling.py

Listing 19 – class MaxPool1D - pooling.py

```
from numba import njit
import numpy as np

# TODO
# AvgPool1D
# MaxPool2D
# AvgPool2D


class MaxPool1D(object):
    """
    Perform pooling, using the maximum value from each window.
    If the last window doesn't fit completely, just ignore it.

    It operates on a set of one dimensional signals.
    """
    def __init__(self, stride=2, window=3):
        self.stride = stride
        self.window = window
        self.n_signals = None
        self.signal_length = None
        self.pooled_length = None
        self.i_max = None

    def initialize(self):
        """
        Use the first set of inputs to infer the size of the remaining
        parameters.
        """
        self.n_signals, self.signal_length = self.x.shape
        self.pooled_length = (
            self.signal_length - self.window) // self.stride + 1
        self.i_max = np.zeros((self.n_signals, self.pooled_length), dtype=int)

    def __str__(self):
        str_parts = [
            "maximum␣pooling",
            f"stride:␣{self.stride}",
            f"window:␣{self.window}",
            f"number␣of␣signals:␣{self.n_signals}",
            f"signal␣length:␣{self.signal_length}",
            f"pooled␣signal␣length:␣{self.pooled_length}",
        ]
        return "\n".join(str_parts)

    def forward_pass(self, signals):
        """
        signals is a two dimensional array of shape (n_signals, signal_length).
        Each row is a separate one dimensional signal.
        """
        self.x = signals
        if self.n_signals is None:
            self.initialize()

        self.y = max_pool_1d(
            signals, self.i_max, self.window, self.stride, self.pooled_length)
        return self.y

    def backward_pass(self, dL_dy):
        """
        Transform the gradient with backpropagation and pass it back.
        gradient is a two dimensional array of shape
        (n_signals, gradient_length).
        Each row is the gradient of a separate signal.
        """
        self.dL_dy = dL_dy
        self.dL_dx = max_unpool_1d(
            self.dL_dy,
            self.i_max,
            self.window,
            self.stride,
            self.signal_length)
        return self.dL_dx


@njit
def max_pool_1d(signals, i_max, window, stride, pooled_length):
    """
    signals is a two dimensional array of shape (n_signals, signal_length).
    window is an integer, the width of the pooling window.
    stride is an integer, the size of the step each time the window shifts.
    pooled_length is the length of each signal after being pooled.

    Returns results and i_max
    Both are two dimensional arrays of shape (n_signals, pooled_length)
    results contain the maximum values from each window.
    i_max contains the location within the window
    """
    n_signals = signals.shape[0]
    results = np.zeros((n_signals, pooled_length))
    for i_window in range(pooled_length):
        i_start = i_window * stride
        i_stop = i_window * stride + window
        for i_signal in range(n_signals):
            results[i_signal, i_window] = np.max(
                signals[i_signal, i_start:i_stop])
            i_max[i_signal, i_window] = np.argmax(
                signals[i_signal, i_start:i_stop])
    return results


@njit
def max_unpool_1d(gradient, i_max, window, stride, signal_length):
    """
    gradient and i_max are two dimensional arrays
    of shape (n_signals, pooled_length).
    gradient is what needs to be unpooled, and i_max is the index within
    each window of the maximum value. It's used to assign responsibility
    for the gradient.
    window is an integer, the width of the pooling window.
    stride is an integer, the size of the step each time the window shifts.
    signal_length is the length of each signal after being unpooled.

    Returns a two dimensional result of shape (n_signals, signal_length)
    containing the unpooles gradient, ready to be pushed down to the
    previous layer.
    """
    n_signals, pooled_length = gradient.shape
    results = np.zeros((n_signals, signal_length))
    for i_signal in range(n_signals):
        for i_window in range(pooled_length):
            results[
                i_signal,
                i_window * stride + i_max[i_signal, i_window]
            ] = gradient[i_signal, i_window]
    return results
```

dn_data_loader.py

Listing 20 – dn_data_loader.py

```
import os
import matplotlib.pyplot as plt
import numpy as np

DATA_DIR = os.path.join("stage", "data", "Simul_Theorique_response_to_cut")

def select_signal_range(length_fault, channel_signal):
    '''
    the range for cutting
    '''
    range_arrays = {
        2: np.array([[230, 530], [230, 530], [230, 530], [400, 700]]),
        5: np.array([[250, 650], [240, 640], [240, 640], [480, 880]]),
        7.5: np.array([[210, 810], [210, 810], [200, 800], [500, 1200]])
    }

    range_array = range_arrays[length_fault]

    min_range, max_range = range_array[channel_signal-1, 0], range_array[channel_signal-1, 1]

    return min_range, max_range


def get_signal_cut_pad(length_fault, channel_signal, total_point):
    '''
    polt the signal cut pad : length_fault \in {2cm, 5cm, 7.5cm}, for each length_fault,
                              channel_signal \in {0, 1, 2, 3, 4} where
                              channel_signal = 0 for time ; channel_signal = 1, 2, 3, 4  for
                                  wave form
                              total_point is the length of the signal after cutting and padding
    '''

    data = np.loadtxt(os.path.join(DATA_DIR, f"{length_fault}.txt"))

    t_samp = data[1, 0] - data[0, 0]    # sampling time in s

    f_samp = 1 / t_samp                 # sampling rate in Hz

    min_range, max_range = select_signal_range(length_fault, channel_signal)

    signal_cut = data[min_range : max_range, channel_signal] # cutting signal
    zero_left = ( total_point - (max_range - min_range) ) //2

    zero_right = total_point - zero_left - (max_range - min_range)

    signal_cut_pad = np.pad(signal_cut, (zero_left, zero_right), 'constant', constant_values=0)
         # cutting padding signal

    return signal_cut_pad


def load_waves(array_length_fault, array_channel_signal, total_point, array_dsp, nb_per_dsp):
    '''
    array_length_fault : [2, 5, 7.5] in cm
    array_channel_signal : [3, 4]
    total_point : 1000
    array_dsp : np.linspace(5e-6, 5e-9, 5)
    nb_per_dsp : 20

    '''
    training_fraction = .6
    tuning_fraction = .2
    testing_fraction = .2

    training_data = []
    tuning_data = []
    testing_data = []

    # Pull all the examples
    examples = []

    for length_fault in array_length_fault : # [2, 5, 7.5]
        for channel_signal in array_channel_signal : # [3, 4]
            for _ in range(nb_per_dsp) :
                for dsp in array_dsp :
                    # signal noise + defect
                    cutting_padding_signal = get_signal_cut_pad(length_fault, channel_signal,
                        total_point)
                    sig_noise_with_defect = cutting_padding_signal + np.sqrt(dsp)*np.random.
                        randn(total_point)
                    examples.append((sig_noise_with_defect[np.newaxis, :], "Defect"))
                    # signal noise
                    sig_noise_without_defect = np.sqrt(dsp)*np.random.randn(total_point)
                    examples.append((sig_noise_without_defect[np.newaxis, :], "Normal"))

    np.random.shuffle(examples)

    # class_count = nb_per_dsp + 1
    class_count = nb_per_dsp * len(array_dsp) // 3 + 1

    n_class_training = int(class_count * training_fraction)
    n_class_tuning = int(class_count * tuning_fraction)
    n_class_testing = int(class_count * testing_fraction)

    for label in ["Defect", "Normal"]:
        class_training_data = []
        class_tuning_data = []
        class_testing_data = []

        for example in examples:
            if example[1] == label:
                roll = np.random.sample()
                if roll < training_fraction:
                    class_training_data.append(example)
                elif roll < training_fraction + tuning_fraction:
                    class_tuning_data.append(example)
                else:
                    class_testing_data.append(example)

        i_training_data = np.random.choice(
            # np.arange(len(class_training_data), dtype=np.int),
            np.arange(len(class_training_data), dtype=np.int64),
            size=n_class_training)
        for i_data in i_training_data:
            training_data.append(class_training_data[i_data])

        i_tuning_data = np.random.choice(
            # np.arange(len(class_tuning_data), dtype=np.int),
            np.arange(len(class_tuning_data), dtype=np.int64),
            size=n_class_tuning)
        for i_data in i_tuning_data:
            tuning_data.append(class_tuning_data[i_data])

        i_testing_data = np.random.choice(
            # np.arange(len(class_testing_data), dtype=np.int),
            np.arange(len(class_testing_data), dtype=np.int64),
            size=n_class_testing)
        for i_data in i_testing_data:
            testing_data.append(class_testing_data[i_data])

    return training_data, tuning_data, testing_data

def data_generator(examples):
    while True:
        i_data = np.random.choice(len(examples))
        yield examples[i_data]

def get_training_data():
    return data_generator(training_data)


def get_tuning_data():
    return data_generator(tuning_data)


def get_testing_data():
    return data_generator(testing_data)


def test():
    for _ in range(10):
        example = next(testing_data)
        plt.figure()
        plt.plot(example[0])
        plt.xlabel(example[1])
        plt.show()


array_length_fault = [2, 5, 7.5]
array_channel_signal = [3, 4]
total_point = 1000
array_dsp = np.linspace(5e-10, 5e-6, 40)
nb_per_dsp = 3

training_data, tuning_data, testing_data = load_waves(array_length_fault, array_channel_signal,
    total_point, array_dsp, nb_per_dsp)

'''

Here is the test of the function load_waves()

array_length_fault = [7.5]
array_channel_signal = [3]
total_point = 1000
array_dsp = np.linspace(5e-10, 5e-6, 40)
nb_per_dsp = 1

training_data, tuning_data, testing_data = load_waves(array_length_fault, array_channel_signal,
    total_point, array_dsp, nb_per_dsp)

    [ Note ] :
    examples = [] : we have len(array_length_fault) * len(array_channel_signal) * len(array_dsp) * nb_per_dsp * 2,
                    where 2 means 2 classes "Defect" and "Normal"
                    here we have 1 * 1 * 40 * 1 = 40 for each class ("Defect" and "Normal")
                    so we generate 40 * 2 = 80 examples

    class_count = nb_per_dsp * len(array_dsp) // 3 + 1 : the number of total examples that we choose for each classe
                                                         to put into the whole training_data, tuning_data, testing_data
                                                         here we have (1* 40) // 3 + 1 = 14 examples for each classe

                                                         then it will be divided into 3 groups with the fraction
                                                                training_fraction = .6
                                                                tuning_fraction = .2
                                                                testing_fraction = .2

    n_class_training = int(class_count * 0.6) : here we have int(14 * 0.6) = 8
    n_class_tuning = int(class_count * 0.2)   : here we have int(14 * 0.2) = 2
    n_class_testing = int(class_count * 0.2)  : here we have int(14 * 0.2) = 2

    n_training = n_class_training * 2 = 8 * 2 = 16 : 2 cause 2 classes ("Defect" and "Normal")
    n_tuning = n_class_tuning * 2 = 2 * 2 = 4      : 2 cause 2 classes ("Defect" and "Normal")
    n_testing = n_class_testing * 2 = 2 * 2 = 4    : 2 cause 2 classes ("Defect" and "Normal")

'''
'''
array_length_fault = [7.5]
array_channel_signal = [3]
total_point = 1000
array_dsp = np.linspace(5e-10, 5e-6, 40)
nb_per_dsp = 1

training_data, tuning_data, testing_data = load_waves(array_length_fault, array_channel_signal, total_point, array_dsp, nb_per_dsp)

for idx, (signal, label) in enumerate(training_data):
    total_image = len(array_length_fault) * len(array_channel_signal) * len(array_dsp) * nb_per_dsp * 2 # 2 classes for "Defect" and "Normal"
    class_count = nb_per_dsp * len(array_dsp) // 3 + 1
    n_class_training = int(class_count * 0.6) # training_fraction = 0.6
    n_training = n_class_training * 2  # 2 classes for "Defect" and "Normal"
    plt.figure()
    plt.plot(signal[0])
    plt.title(f"Training Data Nber {idx + 1} / {n_training} - {label}")
    plt.xlabel("Sample Index")
    plt.ylabel("Amplitude")
    plt.show()

for idx, (signal, label) in enumerate(tuning_data):
    total_image = len(array_length_fault) * len(array_channel_signal) * len(array_dsp) * nb_per_dsp * 2 # 2 classes for "Defect" and "Normal"
    class_count = nb_per_dsp * len(array_dsp) // 3 + 1
    n_class_tuning = int(class_count * 0.2) # tuning_fraction = 0.6
    n_tuning = n_class_tuning * 2  # 2 classes for "Defect" and "Normal"
    plt.figure()
    plt.plot(signal[0])
    plt.title(f"Tuning Data Nber {idx + 1} / {n_tuning} - {label}")
    plt.xlabel("Sample Index")
    plt.ylabel("Amplitude")
    plt.show()

for idx, (signal, label) in enumerate(testing_data):
    total_image = len(array_length_fault) * len(array_channel_signal) * len(array_dsp) * nb_per_dsp * 2 # 2 classes for "Defect" and "Normal"
    class_count = nb_per_dsp * len(array_dsp) // 3 + 1
    n_class_testing = int(class_count * 0.2) # testing_fraction = 0.6
    n_testing = n_class_testing * 2  # 2 classes for "Defect" and "Normal"
    plt.figure()
    plt.plot(signal[0])
    plt.title(f"Testing Data Nber {idx + 1} / {n_testing} - {label}")
    plt.xlabel("Sample Index")
    plt.ylabel("Amplitude")
    plt.show()

'''
```

dn_data_block.py

Listing 21 – dn_data_block.py

```
import numpy as np
## import dn_data_loader as dat
import stage.dn_data_loader as dat

class TrainingData(object):
    def __init__(self):
        self.data = dat.get_training_data()

    def __str__(self):
        return "DN␣training␣data"

    def forward_pass(self, arg):
        return next(self.data)

    def backward_pass(self, arg):
        pass


class TuningData(object):
    def __init__(self):
        self.data = dat.get_tuning_data()

    def __str__(self):
        return "DN␣tuning␣data"

    def forward_pass(self, arg):
        return next(self.data)

    def backward_pass(self, arg):
        pass


class TestingData(object):
    def __init__(self):
        self.data = dat.get_testing_data()

    def __str__(self):
        return "DN␣testing␣data"

    def forward_pass(self, arg):
        return next(self.data)

    def backward_pass(self, arg):
        pass
```