STT3030 - Cours #7

Arthur Charpentier

Automne 2024

Rappel

$\boldsymbol{X} = \{X_1, ..., X_p\}$ forment une collection de prédicteurs.
On dit que $\boldsymbol{X} \in \mathcal{X}_1 \times ... \times \mathcal{X}_p = \mathcal{X}$, où $\mathcal{X}_j$ est le domaine de X_j.
Ce sont les variables que nous utilisons pour prédire.
$Y \in \mathcal{Y}$ est une réponse.
On veut prédire la réponse Y avec les prédicteurs $\boldsymbol{X}$.

- On suppose l'existence d'une fonction $f : \mathcal{X} \to \mathcal{Y}$
- Souvent de la forme la vraie fonction $Y = f(\boldsymbol{X}) + \varepsilon$

Où ε est une supposée variabilité inexpliquée.
En apprentissage supervisé, on veut estimer f.

- Pour faire de la prédiction: $\hat{Y} = \hat{f}(\boldsymbol{X})$
- Étant donnée des prédicteurs $\boldsymbol{X}$ quelle est notre meilleure estimation de la réponse.

Bagging et Boosting

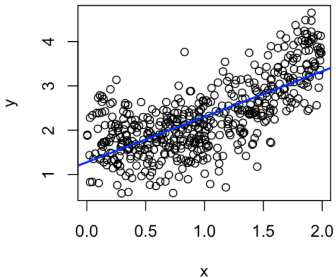
- Le **Bagging** vise à construire B modèles à partir de B bases de données obtenues par bootstrap. La **construction** de ces B modèles se fait de façon **parallèle**, c'est-à-dire de façon totalement indépendante.
- Le **Boosting** vise à construire B modèles de façon **séquentielle**: le b-ième modèle dépend du $(b-1)$-ième modèle qui dépend lui-même du $(b-2)$-ième modèle, etc. Chacun nouveau modèle est construit spécifiquement afin d'améliorer les prédictions faites par le modèle précédent.
- En français, on pourrait parler (mais en pratique, on ne le fait jamais) d'amplification ou de stimulation.

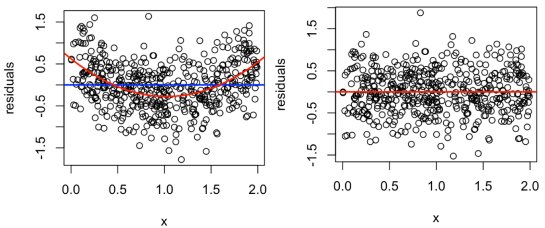
Apprendre de ses erreurs...

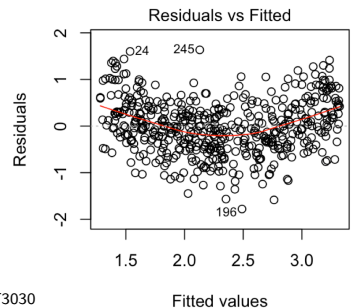
```
> df = dfr = data.frame(x=x, y=y)
> reg1 = lm(y~x, data=df)
> dfr$y = residuals(reg1)
> reg2 = lm(y~poly(x,2), data=dfr)
```

1. $y_i = \beta_0 + \beta_1 x_i + r_{1,i}$
2. $r_{1,i} = \gamma_0 + \gamma_1 x_i + \gamma_2 x_i^2 + r_{2,i}$

(etc ?)

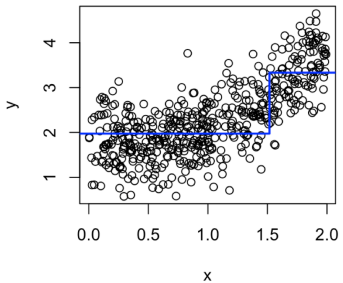


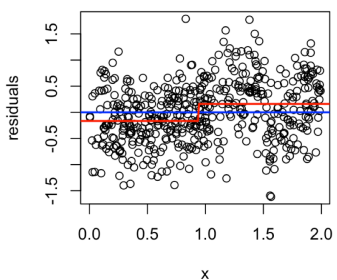


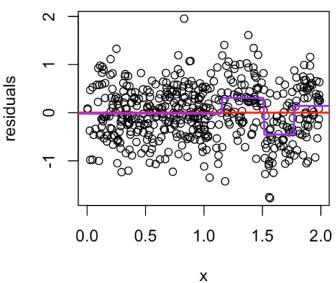


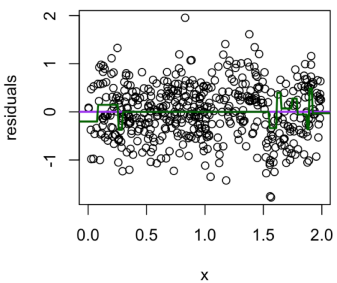
Apprendre de ses erreurs...

```
> df = dfr = data.frame(x=x, y=y)
> reg1 = rpart(y~x, data=df)
> dfr$y = residuals(reg1)
> reg2 = rpart(y~x, data=dfr)
> dfr$y = residuals(reg2)
> reg3 = rpart(y~x, data=dfr)
> dfr$y = residuals(reg3)
> reg4 = rpart(y~x, data=dfr)
```









Boosting

Algorithm 1: Boosting (version 1)

initialization : k (number of trees), γ, $f_0(\boldsymbol{x}) = \overline{y}$;
for $t = 1, 2, ...k$ **do**
    compute $r_{i,t} \leftarrow y_i - f_{t-1}(\boldsymbol{x}_i)$;
    fit a model $r_{i,t} \sim h(\boldsymbol{x}_i)$ for some tree h;
    update $f_t(\cdot) = f_{t-1}(\cdot) + \gamma h(\cdot)$

Algorithm 2: Boosting (version 2)

initialization : k (number of trees), $f_0(\boldsymbol{x}) = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} \ell(y_i, \gamma)$;
for $t = 1, 2, ...k$ **do**
    compute $r_{i,t} \leftarrow \left.\dfrac{\partial \ell(y_i, \widehat{y})}{\partial \widehat{y}}\right|_{\widehat{y}=f_{t-1}(\boldsymbol{x}_i)}$;
    fit a model $r_{i,t} \sim h(\boldsymbol{x}_i)$ for some tree h;
    update $f_t(\cdot) = f_{t-1}(\cdot) + \gamma h(\cdot)$

Algorithme AdaBoost

ou voir la mise à jour à l'aide de poids (si la classification n'est pas bonne)

On s'intéresse à un problème de classification binaire (0 ou 1) à partir d'une base de données $\mathcal{D} = \{Y_i, \mathbf{X}_i\}_{i=1,\dots,n}$.

1. Toutes les observations reçoivent un poids identique: $p_1(\mathbf{X}_i) = 1/n$, $i = 1, \dots, n$.

Algorithme AdaBoost (suite)

2. Pour l'étape $t = 1, \ldots, T$,
   - 2a. Piger au hasard (en utilisant les poids $p_t(\mathbf{X}_i)$) une base de données d'entrainement $\mathcal{D}_t = (\mathcal{D}, p_t)$.
   - 2b. Apprendre une règle de classification r_t sur cette base de données.
   - 2c. Calculer l'erreur apparente

$$\epsilon_t = p_t\left(r_t(\mathbf{X}_i) \neq Y_i\right) = \sum_{i: r_t(\mathbf{X}_i) \neq Y_i} p_t(i)$$

   pour $\mathcal{D}_t$ et évaluer $\alpha_t = 0.5 \ln((1-\epsilon_t)/\epsilon_t)$.
   - 2d. Mettre à jour les poids:

$$p_{t+1}(\mathbf{X}_i) = \begin{cases} \frac{p_t(\mathbf{X}_i)}{Z_t} e^{-\alpha_t}, & r_t(\mathbf{X}_i) = Y_i \\ \frac{p_t(\mathbf{X}_i)}{Z_t} e^{+\alpha_t}, & r_t(\mathbf{X}_i) \neq Y_i, \end{cases}$$

   où Z_t est une constante de normalisation assurant que $\sum_{i=1}^{n} p_{t+1}(\mathbf{X}_i) = 1$.

Algorithme AdaBoost (suite)

3. La classification finale faite par le modèle est alors

$$R(\mathbf{X}_i) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t r_t(\mathbf{X}_i) > 0 \\ -1, & \sum_{t=1}^{T} \alpha_t r_t(\mathbf{X}_i) < 0. \end{cases}$$

Algorithm 3: Boosting (Adaboost)

initialization : k, $\omega_i \leftarrow 1/n$;
for $t = 1, 2, ...k$ **do**
  error rate $e_t \leftarrow \dfrac{\sum_{i=1}^{n} \omega_i \mathbf{1}(y_i \neq f_{t-1}(\boldsymbol{x}_i))}{\sum_{i=1}^{n} \omega_i}$;
  set $\alpha_t \leftarrow \log(1 - e_t) - \log(e_t)$;
  update $\omega_i \leftarrow \omega_i \, e^{\alpha_t \mathbf{1}(y_i \neq f_{t-1}(\boldsymbol{x}_i))}$;
  update $f_t(\cdot) \leftarrow f_{t-1}(\cdot) + \alpha_t h(\cdot)$

Remarques

On apprend tant que $\alpha_t \geq 0$, soit $e_t < 1/5$.
Modèle classique pour h: CART (avec des poids)

- Approche théorique

Étant donnée une famille de modèles $\mathcal{H}$, on cherche à résoudre

$$h^\star = \underset{h \in \mathcal{H}}{\text{argmin}} \left\{ \mathbb{E}\big[\ell(Y, h(\boldsymbol{X}))\big] \right\}$$

ou sa version empirique

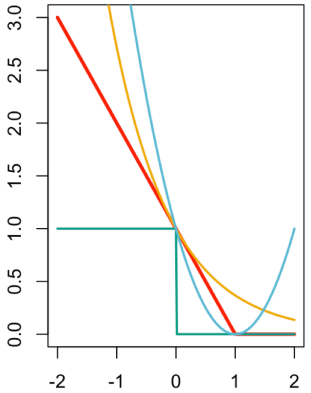
$$\widehat{h}^\star = \underset{h \in \mathcal{H}}{\text{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(\boldsymbol{x}_i)) \right\}$$

avec classiquement $\ell(y, \hat{y}) = \mathbf{1}(y = \hat{y})$.
On peut tenter de convexifier la fonction de perte, $\overline{\ell}(y, \hat{y}) = \exp(-y\hat{y})$.

Remarques

Supposons $y \in \{-1, +1\}$.
Les points sont mal classés si $y \cdot f(\boldsymbol{x} < 0)$,
on peut tracer la fonction de perte en fonction de $y\hat{y}$

- misclassification: $\mathbf{1}(y\hat{y} < 0)$
- hinge: $(1 - y\hat{y})_+$
- exponential: $\exp(-y\hat{y})$



L'algorithme Adaboost vérifie $f_t = f_{t-1} + 2\beta^\star h^\star$, où $(\beta^\star, h^\star)$ est solution de

$$(\beta^\star, h^\star) = \underset{(\beta,h)}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n} \overline{\ell}(y_i, f_{t-1}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i)) \right\}$$

Remarques

$$(\beta^\star, h^\star) = \underset{(\beta,h)}{\text{argmin}} \left\{ \sum_{i=1}^{n} \overline{\ell}(y_i, f_{t-1}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i)) \right\}$$

$$(\beta^\star, h^\star) = \underset{(\beta,h)}{\text{argmin}} \left\{ \sum_{i=1}^{n} \underbrace{\exp\big(-y_i f_{t-1}(\boldsymbol{x}_i)\big)}_{\omega_i} \exp\big(-y_i \beta h(\boldsymbol{x}_i)\big) \right\}$$

on distingue alors les i pour lesquels $y_i = h(\boldsymbol{x}_i)$ et $y_i \neq h(\boldsymbol{x}_i)$, et

$$h^\star = \underset{h \in \mathcal{H}}{\text{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \omega_i \ell(y_i, h(\boldsymbol{x}_i)) \right\}$$

Remarques

- L'erreur apparente est calculée en utilisant la distribution avec laquelle le modèle est entrainé.
- Chacune des nouvelles règles r_t est pondérée par un poids α_t qui indique l'importance que cette règle aura dans la décision finale R.
- Si T, le nombre de modèles, est *trop* grand, alors il y a un risque que l'algorithme se concentre trop vers la fin sur les cas difficiles (potentiellement du bruit) et accorde à ces derniers trop d'importance dans la décision finale R.

Bornes

- L'erreur apparente

$$\epsilon_t = p_t\left(r_t(\mathbf{X}_i) \neq Y_i\right) = \sum_{i: r_t(\mathbf{X}_i) \neq Y_i} p_t(i)$$

peut être réécrite comme

$$\epsilon_t = 0.5 - \gamma_t,$$

où γ_t est l'amélioration apportée à la prédiction par la règle r_t en comparaison avec le hasard (0.5).

Bornes (suite)

- On peut démontrer que l'erreur apparente finale (pour T) a comme borne supérieure

$$\epsilon_T \leq \exp\left(-2\sum_t \gamma_t^2\right) \leq \exp\left(-2T\gamma^2\right),$$

où $\gamma = \min(\gamma_1, \ldots, \gamma_T)$.

- Ainsi, si le weak learner fait légèrement mieux que le hasard, $\gamma_t > 0$ et l'erreur apparente finale diminue exponetiellement avec le nombre de modèles (T).

Fonction objectif

1. De façon générale, on cherche une règle $\widehat{r}$ en résolvant un problème d'optimisation empirique

$$\widehat{r} = \underset{r \in \mathcal{R}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, r(\boldsymbol{X}_i)),$$

   où ℓ est une fonction de perte quelconque.
2. Si l'optimisation est faite sur l'ensemble des fonctions r possibles et imaginables, le problème est trop complexe pour être résolu (même numériquement!).

Fonction objectif

- Une solution possible est de convexifier la fonction objectif afin de rendre le problème plus simple d'un point de vue numérique.
- L'algorithme AdaBoost répond à ce principe de minimisation pour un risque convexifié donné par

$$\ell(Y_i, r(\boldsymbol{X}_i)) = \exp\left(-Y_i r(\boldsymbol{X}_i)\right).$$

- On va également réaliser l'optimisation de façon séquentielle plutôt que globale (greedy).

Fonction objectif

On peut démontrer que la règle de classification à l'étape t de l'agorithme AdaBoost peut s'écrire comme étant

$$r_t(\mathbf{X_i}) = r_{t-1}(\mathbf{X}_i) + \alpha_t R_t(\mathbf{X}_i),$$

où

$$(\alpha_t, R_t) = \underset{(\alpha,r)\in\mathbb{R}\times\mathcal{R}}{\text{argmin}} \sum_{i=1}^{n} e^{-Y_i(r_{t-1}(\mathbf{X}_i)+\alpha r(\mathbf{X}_i))}$$

et où $\mathcal{R}$ est l'espace des choix possibles pour la règle r.

Boosting par descente de gradient fonctionnelle

1. Initialiser

$$r_0(\mathbf{X}_i) = \underset{c\in\mathbb{R}}{\text{argmin}} \frac{1}{n}\sum_{i=1}^{n} \ell(Y_i, c).$$

2. Pour les étapes $t = 1, \ldots, T$,
   2a. Calculer

$$U_i = -\left[\frac{\partial L(Y_i, r(\mathbf{X}_i)}{\partial r(\mathbf{X}_i)}\right] |_{\{r(\mathbf{X}_i)=r_{t-1}(\mathbf{X}_i)\}}, \qquad i = 1, \ldots, n.$$

   2b. Ajuster le weak learner sélectionné sur l'échantillon composé des éléments $(\mathbf{X}_1, U_1), (\mathbf{X}_2, U_2), \ldots, (\mathbf{X}_n, U_n)$ afin d'obtenir la pseudo-règle $h_t(\mathbf{X}_i)$.
   2c. Effectuer la mise à jour $r_t(\mathbf{X}_i) = r_{t-1}(\mathbf{X}_i) + \alpha h_t(\mathbf{X}_i)$.
3. La prédiction finale est donnée par $r_T(\mathbf{X}_i)$.

Remarques

- Le choix du paramètre α (taux d'apprentissage) est peu important. On recommande généralement de prendre une petite valeur (0.01 ou 0.001).
- Si on pose $\alpha = 1$ et $\ell(y, \hat{y}) = \exp(-y\hat{y})$, on retrouve (presque) l'algorithme AdaBoost.

Gradient Boosting with R, ℓ_2 loss

```
> loss_L2 = function(y, yhat) (mean(1/2*(y-yhat)^2))
> gradient_L2 = function(y, yhat) {(y-yhat)}
```

then use

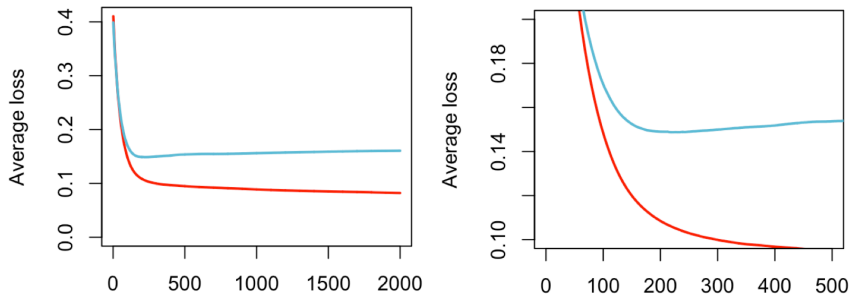
```
> grad_boost_train_valid <- function(formula, data_train, data_valid, nu
    = 0.01, stop=1000, = loss_L2) {
  data = as.data.frame(data_train)
  formula = terms.formula(formula)
  noms_X = names(data_train)[-which(names(data_train)==as.character(
    formula)[2])]
  noms_Y = names(data_train)[which(names(data_train)==as.character(
    formula)[2])]
  y = data_train[, noms_Y]
  fit_train = fit_valid = mean(y)
  data = as.data.frame(data_train[,noms_X])
  names(data) = noms_X
  data$u = grad.fun(y = y, yhat = fit_train)
  loss_train = loss_valid = c()
```

Gradient Boosting with R, ℓ_2 loss

```
  for (i in 1:stop) {
    model_weak = rpart(u  ~ ., data=data, maxdepth=3,cp=1e-9)
    fit_train = as.numeric(fit_train + nu * predict(model_weak, newdata=
    data_train))
    fit_valid = fit_valid + nu * predict(model_weak, newdata=data_valid)
    data$u = as.numeric(unlist(grad.fun(y = as.numeric(unlist(data_train[
    noms_Y])), yhat = fit_train)))
    loss_train <- c(loss_train, loss.fun(y = as.numeric(unlist(data_train
    [noms_Y])), yhat = fit_train))
    loss_valid <- c(loss_valid, loss.fun(y = as.numeric(unlist(data_valid
    [noms_Y])), yhat = fit_valid))
  }
  loss_matrix = data.frame(N = 1:stop, TRAIN = loss_train, VALID =
    loss_valid)
  return(list(loss = loss_matrix))
}
```

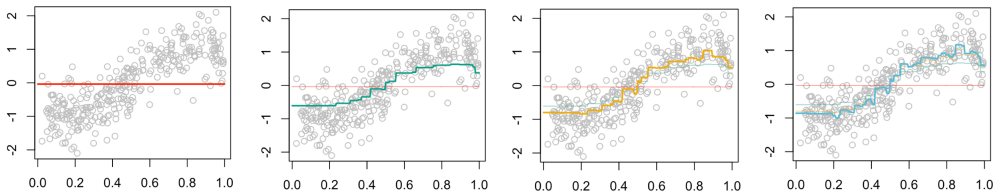
Gradient Boosting with R, ℓ_2 loss

```
> x=sort(runif(n))
> y=sin(-3*pi/4+x*pi*3/2)+rnorm(n)/2
DF = data.frame(x=x,y=y)
> set.seed(1)
> idx = sample(1:n,size = 2*n/3)
> library(rpart)
R = grad_boost_train_valid(y~x, data_train=DF[idx,], data_valid=DF[-idx
    ,], nu = .01, stop=2000,grad.fun = gradient_L2, loss.fun = loss_L2)
```



Gradient Boosting with R, ℓ_2 loss

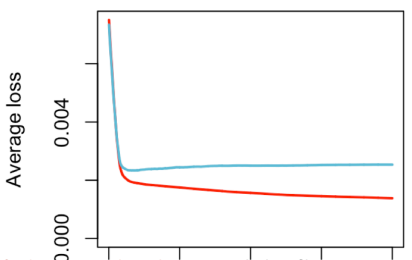
```
> x=sort(runif(n))
> y=sin(-3*pi/4+x*pi*3/2)+rnorm(n)/2
DF = data.frame(x=x,y=y)
> set.seed(1)
> idx = sample(1:n,size = 2*n/3)
> library(rpart)
R = grad_boost_train_valid(y~x, data_train=DF[idx,], data_valid=DF[-idx
    ,], nu = .01, stop=2000,grad.fun = gradient_L2, loss.fun = loss_L2)
```

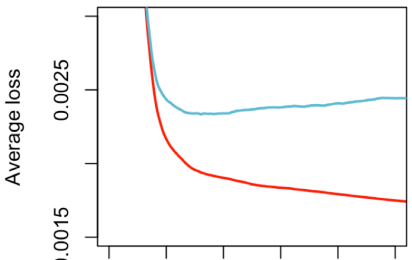


Sequential learning, with 0, 100, 200, 500 trees

Gradient Boosting with R, ℓ_1 (or Huber) loss

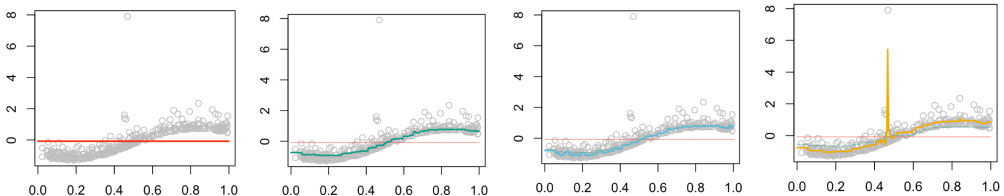
```
> loss_L_huber = function(y, yhat, delta=.01){
  L=1/2*(y-yhat)^2*(abs(y-yhat)<=delta) + delta*(abs(y-yhat)-delta/2)*(
    abs(y-yhat)>delta)
  return(mean(L))}
> gradient_L_huber = function(y, yhat, delta=.01){
  L=(y-yhat)*(abs(y-yhat)<=delta)+delta*sign(y-yhat)*(abs(y-yhat)>delta)
  return(as.numeric(L))}
> y = sin(-3*pi/4+x*pi*3/2)
> e = rlnorm(n)
> DF = data.frame(x=x,y=y+(e-mean(e))/sd(e)/2)
```





Gradient Boosting with R, ℓ_1 (or Huber) loss

```
> library(rpart)
R = grad_boost_train_valid(y~x, data_train=DF[idx,], data_valid=DF[-idx
    ,], nu = .01, stop=2000,grad.fun = gradient_L_huber, loss.fun =
    loss_L_huber)
```



Sequential learning, with 0, 100, 1000 trees,
(and 1000 trees with some ℓ_2 loss on the right)

Introduction

Un **réseau de neurones** est une fonction $\hat{f}$ pour l'apprentissage supervisé. Ces fonctions entrent alors dans le cadre des fonctions qu'on a vues à présent.

- On veut $\hat{f}(x_i) = \hat{y}_i$ avec la plus petite erreur de prédiction possible.
- Les réseaux de neurones sont extrêmements flexibles puisqu'ils ont la capacité d'apprendre par eux-mêmes des relations non linéaires et des interactions.
- Par contre, comme ils sont très variables ils sont difficiles à optimiser/apprendre.
- Malgré leurs grandes flexibilités, ils tendent à faire peu de surapprentissage (question sans réponse).

Binary Threshold Neuron & the Perceptron

If $\boldsymbol{x} \in \{0,1\}^p$, McCulloch and Pitts (1943) suggested a simple model, with threshold b

$$y_i = f\left(\sum_{j=1}^{p} x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq b)$$

where $\mathbf{1}(x \geq b) = +1$ if $x \geq b$ and 0 otherwise, or (equivalently)

$$y_i = f\left(\omega + \sum_{j=1}^{p} x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq 0)$$

with weight $\omega = -b$. The trick of adding 1 as an input was very important !
$\omega = -1$ is the or logical operator : $y_i = 1$ if $\exists j$ such that $x_{j,i} = 1$
$\omega = -p$, is the and logical operator : $y_i = 1$ if $\forall j$, $x_{j,i} = 1$

Binary Threshold Neuron & the Perceptron

but not possible for the xor logical operator : $y_i = 1$ if $x_{1,i} \neq x_{2,i}$

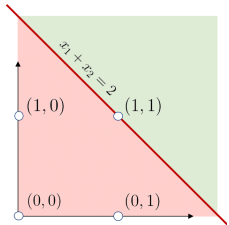
Rosenblatt (1961) considered the extension where x's are real-valued, with weight $\boldsymbol{\omega} \in \mathbb{R}^p$

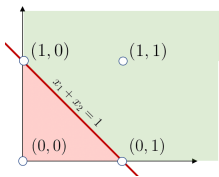
$$y_i = f\left(\sum_{j=1}^{p} \omega_j x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq b)$$

where $\mathbf{1}(x \geq b) = +1$ if $x \geq b$ and 0 otherwise, or (equivalently)

$$y_i = f\left(\omega_0 + \sum_{j=1}^{p} \omega_j x_{j,i}\right) \text{ where } f(x) = \mathbf{1}(x \geq 0)$$

with weights $\boldsymbol{\omega} \in \mathbb{R}^{p+1}$





Binary Threshold Neuron & the Perceptron

Minsky & Papert (1969) proved that perceptron were a linear separator, not very powerful

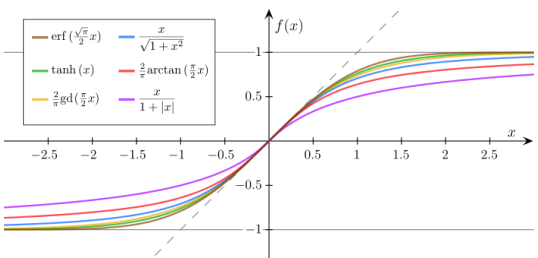
Define the sigmoid function $f(x) = \dfrac{1}{1+e^{-x}} = \dfrac{e^x}{e^x+1}$ (= logistic).

This function f is called the activation function.

If $y \in \{-1, +1\}$, one can consider the hyperbolic tangent

$$f(x)) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

or the inverse tangent function (see wikipedia).



Introduction

Un réseau de neurones est composé de **couches** de neurones. L'information entrante (les prédicteurs) passe à travers chaque couche avant de sortir (la réponse).

- Chaque couche est relativement simple: elle retourne un vecteur qui est la composition d'un produit matricielle et d'une fonction non linéaire.
- La succession de ces opérations simple permet d'estimer de complexes fonctions non-linéaires et des interactions.
- Débutons avec un simple réseau de neurones avec **une couche cachée**.

Représentation graphique

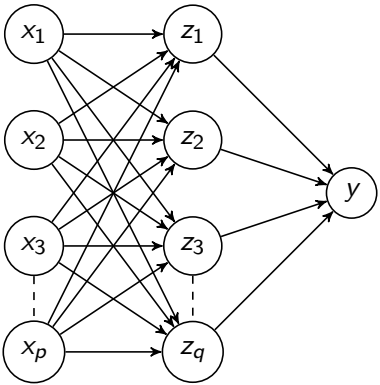


Figure 1: Représentation graphique d'un réseau de neurones. La première couche représente nos prédicteurs $\boldsymbol{x} = (x_1, \cdots, x_p)$ (couche d'entrée). La deuxième couche $\boldsymbol{z} = (z_1, \cdots, z_q)$ est une **couche cachée** de neurones. Chaque arrête représente un coefficient.

Représentation fonctionnelle

On utilise souvent la représentation graphique, celle-ci en met plein la vue pour expliquer quelque chose de relativement simple.
Chaque élément z_h de la couche caché (disons $\mathbf{z}$) est une combinaison linéaire des éléments de la couche précédente (ici les prédicteurs $\mathbf{x}$) sur laquelle nous appliquons une fonction non linéaire:

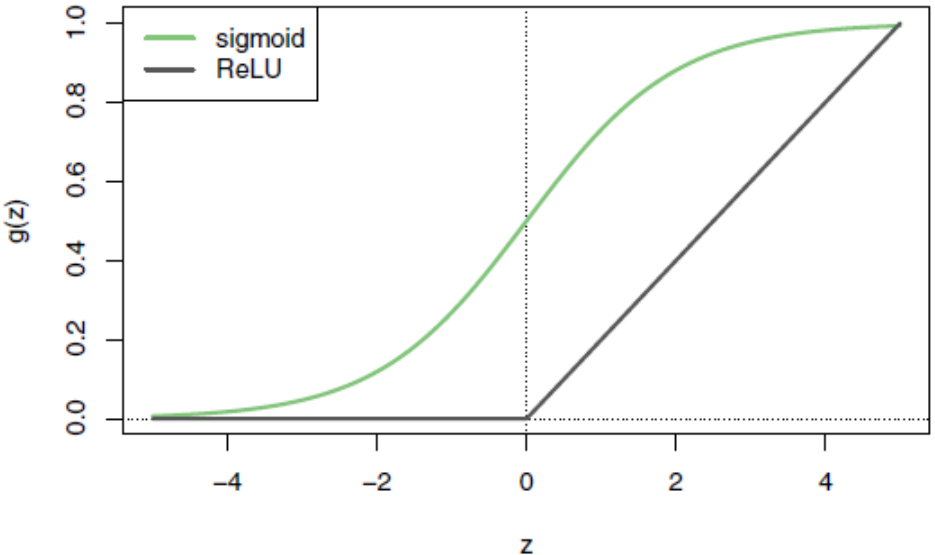
$$z_l = g(\sum_{j=p} b_{l,j}^{(1)} x_j)$$

où g est une fonction non linéaire et où les $b_j^{(1)}$ forment une collection de paramètres (coefficients à apprendre).

Fonction non linéaire

- Il est essentiel d'utiliser une fonction non linéaire entre les couches sinon ceux-ci se recombinent comme une simple combinaison linéaire à la couche suivante. (Exercices)
- C'est ce qui permet d'apprendre une fonction $\hat{f}$ non-linéaire avec interactions.
- Fonction **sigmoid** : $g(u) = \dfrac{1}{1+e^{-u}} = \dfrac{e^u}{1+e^u}$, à valeurs dans $(0, +1)$
- Fonction **ReLu** : $g(u) = (u)_+ = \max\{u, 0\}$, à valeurs dans $(0, \infty)$
- Fonction **tangente hyperbolique** : $\tanh(u) = \dfrac{e^u - e^{-u}}{e^u + e^{-u}}$, à valeurs dans $(-1, +1)$
- Il existe plusieurs fonctions non linéaires usuelles,

Fonction non linéaire



Fonction non linéaire

Les fonctions non linéaires permettent évidemment d'apprendre $\hat{f}$ non linéaire, mais aussi les interactions:

- Soit un modèle a une couche caché avec $L = 2$ et $g(u) = u^2$ pour un problème de prédiction avec deux prédicteurs (p=2).
- Supposons les paramètres appris suivants pour la couche sortie $b_0^{(2)} = 0$ $b_1^{(2)} = 1/4$ et $b_2^{(2)} = -1/4$ et pour la couche cachée : $b_{1,0}^{(1)} = 0$ $b_{1,1}^{(1)} = 1$ et $b_{1,2}^{(1)} = 1$, $b_{2,0}^{(2)} = 0$ $b_{2,1}^{(2)} = 1$ et $b_{2,2}^{(2)} = -1$.

Donc: $z_1 = g(\mathbf{b_l^1}^\top \mathbf{x}) = (0 + x_1 + x_2)^2$ et $z_2 = (0 + x_1 - x_2)^2$
Conséquemment :

$$\begin{aligned}\hat{f}(\mathbf{x}) = \hat{y} &= 0 + 1/4(0 + x_1 + x_2)^2 - 1/4(0 + x_1 - x_2)^2 \\ &= 1/4((x_1 - x_2)^2 + (x_1 - x_2)^2) = x_1 x_2\end{aligned}$$

Représentation fonctionnelle

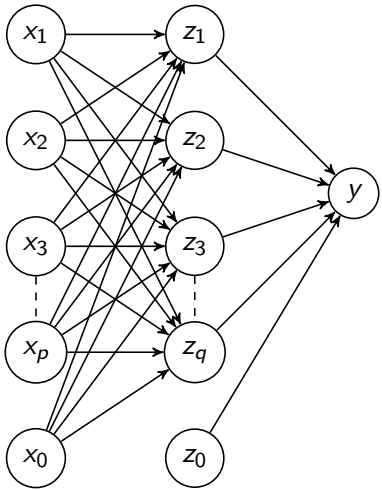
Tout comme en régression, on peut ajouter un paramètre constant (ordonné à l'origine, β_0) appelé *biais* en I.A. (on n'utilisera pas le terme biais dans le cours).

$$z_l = g(\sum_{j=p} b_{l,j}^{(1)} x_j + b_{l,0}^{(1)})$$

pour garder la représentation fonctionnelle compacte, on présuppose que $x_0 = 1 \ \forall n$ et on réécrit:

$$z_l = g(\mathbf{x}_{1\times(p+1)}\mathbf{b}_{(p+1)\times 1})$$

Représentation graphique (avec biais)



Représentation fonctionnelle

On peut rendre la notation encore plus compacte en considérant plusieurs combinaisons linéaires en simultané, donc en considérant tous les z_h, c'est-à-dire la couche cachée au complet, le vecteur $\mathbf{z}$:

$$\mathbf{z}_{1\times q} = g(\mathbf{x}_{1\times(p+1)}\mathbf{B}_{(p+1)\times q})$$

où la fonction g est appliquée aux éléments du vecteur $\mathbf{x}_{1\times(p+1)}\mathbf{B}_{(p+1)\times q}$ et où $\mathbf{B}$ est une matrice de $(p+1)\times q$ paramètres.

Représentation fonctionnelle

On peut séquentiellement passé l'information à travers plusieurs couches, par exemple, pour le modèle à une couche caché, la prochaine couche est la réponse prédite:

$$\hat{f}(\mathbf{x}) = \hat{y} = h(\mathbf{z}_{1\times q}\mathbf{B}^{(2)}_{q\times 1})$$

ce qui donne

$$\hat{f}(\mathbf{x}) = \hat{y} = h(g(\mathbf{x}_{1\times p}\mathbf{B}^{(1)}_{p\times q})\mathbf{B}^{(2)}_{q\times 1})$$

et nous avons donc $q + p * q$ paramètres à **appendre**.

Représentation graphique

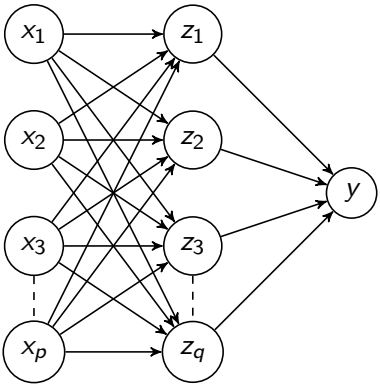


Figure 3: Représentation graphique d'un réseau de neurones. La première couche représente nos prédicteurs x. La deuxième couche z est une *couche cachée* de neurones. Chaque arrête représente un coefficient. Chaque élément z_l est une différente combinaison linéaire.

Représentation fonctionnelle

En résumé dans une application simple:

- La fonction prend en entré un vecteur de p prédicteurs $\mathbf{x}$.
- Fait q combinaisons linéaires de ces p prédicteurs avec p coefficients (paramètres qui doivent être appris).
- Ces q combinaisons linéaires sont ensuite transformé par une fonction non linéaire (déterministe, non-paramètrique) pour en extraire q nouvelles variables.
- Finalement on peut répéter le processus récursivement jusqu'à la couche sortie qui prédit y.

Réseau de neuronnes: manipulation de donnés

Nous avons:

$$\hat{f}(\mathbf{x}) = \hat{y} = h(g(\mathbf{x}_{1\times p}\mathbf{B}^{(1)}_{p\times q})\mathbf{B}^{(2)}_{q\times 1})$$
$$\Rightarrow \hat{f}(\mathbf{x}) = \hat{y} = h(\mathbf{z}_{1\times q}\mathbf{B}^{(2)}_{q\times 1})$$

Supposons h une fonction identité alors nous avons:

$$\hat{y} = B_0^{(2)} + \sum_{j=1}^{q}(B_j^{(2)} z_j)$$

Une autre manière d'interpréter ce que fait un réseau de neuronne: il manipule/modifie les prédicteurs d'entré $\mathbf{x}$ pour en faire de nouveau prédicteurs $\mathbf{z}$ qui performe mieux dans le modèle linéaire $\hat{y} = B_0^{(2)} + \sum_{j=1}^{q}(B_j^{(2)} z_j)$

Prédictions

Remarquons que l'on peut passer une matrice de données $\mathbf{x}_{n\times p}$ en entier dans ce réseau de neurones:

$$\hat{\mathbf{y}}_{n\times 1} = h(g(\mathbf{x}_{n\times p}\mathbf{B}^{(1)}_{p\times q})\mathbf{B}^{(2)}_{q\times 1})$$

(À partir de maintenant, laissons tomber la dimension des matrices et des vecteurs). Bien que le modèle soit complexe, flexible et potentiellement difficile à optimiser, prédire $\hat{y}$ étant donné, un échantillon est extrêmement rapide; les opérations qui forment le réseau sont relativement simples.

Structure de données

La forme suivante:

$$\hat{f}(\mathbf{x}) = \hat{y} = h(g(\mathbf{x}\mathbf{B}^{(1)})\mathbf{B}^{(2)})$$

est approprié pour une réponse continue $y \in \mathbb{R}$ avec h la fonction identité, et un vecteur de prédicteur aussi continue $\mathbf{x} \in \mathbb{R}^p$
Que faire si nous avons d'autres structures de variables.

Structure de données

Pour la variable réponse, nous pouvons ajuste sa taille ainsi que la fonction d'activation h.
Ainsi on perçoit la fonction d'activation h, comme une fonction lien dans les modèles linéaires généralisés.
Par exemple:

- Si on utilise la fonction sigmoid : $h(u) = \frac{1}{1+e^{-u}}$ on récupère une prédiction de type logistique.
- On aura: $\hat{f}(\mathbf{x}) = \hat{y} = \frac{1}{1+e^{\sum z_l B_l^{(2)}}} = \hat{\mathbb{P}}(y = 1|\mathbf{x})$
- On prédit 1 si $\hat{\mathbb{P}}(y = 1|\mathbf{x}) > 0.5$ et 0 sinon!

Structure de données

Pour une variable réponse catégorielle à K catégories, on utilise quelque chose de similaire.

- On définit une sortit **y** un vecteur de dimension K. $[y_1^*, ..., y_K^*]$
- On applique la fonction **softmax** qui retourne une probabilité pour chacun des k catégorie: $\hat{y}_j = h(y_j^*) = \frac{e^{y_j^*}}{\sum_{k=1}^{K} e^{y_k^*}} = \hat{\mathbb{P}}(y = j|\mathbf{x})$
- Ensuite on retourne la classe $\text{argmax}\hat{P}(y = k|\mathbf{x})$, la classe pour laquelle la probabilité est la plus élevée.

Structure de données

Pour des prédicteurs catégoriels, on utilise les variables muettes comme en régression: Supposons qu'il y a k catégorie. Nous créons un vecteur de taille $k-1$ qui contient au plus une entré exactement égale à 1, tel que;

$$X_2 = \begin{cases} \text{Classe 1} = [0, 0, ..., 0] \\ \text{Classe 2} = [1, 0, ..., 0] \\ \text{Classe 2} = [0, 1, ..., 0] \\ .... \\ \text{Classe k} = [0, 0, ..., 1] \end{cases}$$

Comme on n'a pas la contrainte de devoir inverser X, il est commun d'utiliser un vecteur de taille k à la place.

Structure de données

- Une force des réseaux de neurones est que l'on peut développer des couches ou des fonctions d'activation g spécifique à d'autres structures de données comme des images ou des séries chronologiques.
- Pour comprendre comment développer ses couches et les contraintes au développement de ceux-ci, nous devrons comprendre comment entrainer ce modèle.
- On discutera des couches spécialisées au prochain cours.

Apprentissage de réseaux de neurones

Ici nous allons encore utiliser le principe d'optimisation d'une **fonction objective**.

- En régression linéaire, on veut minimiser :

$$Obj(\hat{\beta}) = \sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2 = \mathbf{y} = ||\mathbf{y} - \mathbf{X}\hat{\beta}||^2$$

  en fonction des coefficients estimés $\hat{\beta}$.
- C'est-à-dire que l'on cherche à trouver la combinaison de coefficients $\hat{\beta}$ qui minimise $Obj(\hat{\beta})$.
- On calculer la dérivée exacte de $Obj(\hat{\beta})$ et la mettre à 0 pour trouver la solution $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

Apprentissage de réseaux de neurones

- De manière équivalente, on a vu aussi la fonction objective régularisée:

$$Obj(\hat{\beta}) = \sum_{i=1}^{n}(y_i - \sum_j \beta_j x_{ij})^2 - \lambda \sum_j |\beta_j|$$

- Avec $\hat{f}$ une fonction de type réseau de neurones, nous vous aussi minimiser l'erreur de prédiction quadratique: $Obj(\hat{\theta}) = \sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$, où θ forment la collection des paramètres du réseau de neurones: les matrices de coefficients $B^{(1)}$ et $B^{(2)}$ dans le modèle précédent.
- Comme il est impossible de calculer la dérivée analytiquement pour toute valeur des coefficients $\hat{\theta}$, nous allons approcher le point minimal de $Obj(\hat{\theta})$ progressivement à l'aide de l'algorithme du gradient.

Algorithme du gradient

- L'algorithme du gradient est une procédure d'optimisation itérative que l'on peut utiliser *presque* tout le temps pour optimiser des fonctions.
- Conçu pour l'optimisation (disons minimisation à partir de maintenant) de fonction convexe et différentiable $f(\mathbf{w})$.
- Par exemple l'erreur quadratique moyenne: $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{w}\cdot\mathbf{x}_i - y_i)^2$

Rappellons que le gradient d'une fonction différentiable $f\colon \mathbb{R}^d \to \mathbb{R}$ en $\mathbf{w}$, noté $\nabla f(\mathbf{w})$, est le vecteur des dérivées partielles, soit

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, ..., \frac{\partial f(\mathbf{w})}{\partial w_d}\right).$$

Algorithme du gradient

- On débute avec une valeur initiale pour le vecteur de paramètres, disons $\theta^{(1)} = (0, ..., 0)$.
- À chaque itération, on prend un *pas* dans la direction inverse du gradient $(-\nabla f(\hat{\theta}))$

$$\hat{\theta}^{(t+1)} = \hat{\theta}^{(t)} - \eta \nabla f(\hat{\theta}^{(t)})$$

où $\eta > 0$ est la taille du *pas* (step-size), un paramètre à déterminer (peut être fonction de t ou de $\nabla f(\hat{\theta})$) souvent à l'aide de la validation.

Algorithme du gradient

Intuitivement, puisque le gradient pointe dans la direction qui fait le plus *grandir* f alors prendre un petit pas dans la direction inverse diminue la valeur de f, donc minimise celle-ci.
Après T itérations, l'algorithme nous retourne une valeur pour le vecteur de paramètres $\mathbf{w}$

Apprentissage de réseaux de neurones

- On apprend les paramètres (disons $\hat{\theta}$) par l'algorithme du gradient (ou par une variante).
- Il nous faut donc pouvoir calculer la dérivée partielle de

$$Obj(\hat{\theta}) = \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2$$

  par rapport a chacun des b_j^l.
- Ceci est la clé: les fonctions objectives et les combinaisons linéaires furent choisies puisqu'il est facile de calculer le gradient.

Apprentissage de réseaux de neurones

Soit $f(\mathbf{x}) = h(g(\mathbf{x}_{1\times d}\mathbf{B}^{(1)}_{d\times q})\mathbf{B}^{(2)}_{q\times 1})$ et supposons que h est la fonction identité $h(x) = x$ et que g est la fonction sigmoid $g(x) = \dfrac{1}{1+e^{-x}}$. On utilise ici les règles de dérivées en chaine.

Soit $Obj = (y_i - f(x_i))^2$, alors:

$$\frac{\partial Obj}{\partial B^{(2)}_{1,q}} = -2(y_i - f(x_i))h'(\mathbf{z}_i\mathbf{B}^{(2)}_q)\mathbf{z}_{qi} \text{ et}$$

$$\frac{\partial Obj}{\partial B^{(1)}_{q,d}} = -\sum_q 2(y_i - f(x_i))h'(\mathbf{z}_i\mathbf{B}^{(2)}_q)\mathbf{B}^{(2)}_q g'(\mathbf{x}_i\mathbf{B}^{(1)}_q)x_{i,d}$$

Apprentissage de réseaux de neurones

Ces gradients sont extrêmement rapides à calculé pour deux raisons: (1) il y a une grande symétrie dans leurs calculs, et (2) les dérivés sont *pré-calculé*.

1. Plusieurs éléments dans les dérivés en chaines sont partagés par plusieurs termes, ceux-ci sont donc calculés une seule fois.
2. Les libraires *d'auto-différentiation* (https://pytorch.org/docs/stable/nn.functional.html) ont les dérivées des fonctions d'activation (g) précalculées. Seulement les fonctions dont la dérivée est précalculée peuvent être employées lorsque l'on entraine des réseaux de neurones.

Apprentissage de réseaux de neurones

Une fois ces dérivés partiels calculés. Nous pouvons employer l'algorithme du gradient pour mettre les paramètres à jour (faire un pas dans la direction opposée du gradient). Le calcule des ces dérivées partielles s'appellent *back-propagation*, un terme spécifique pour désigné le calcule des dérivées partielles dans le contexte des réseaux de neurones, processus qui part de la sortie et se propage vers l'arrière jusqu'au début du réseau.

Apprentissage de réseaux de neurones

On fait parfois référence au *forward pass* et *backward pass* lors de l'entrainement. Le *forward pass* représente la circulation de l'information de l'entré $\mathbf{x}_i$ vers la sortie $\hat{y}_i$ nous permettant de calculé $Obj(\hat{\theta})$. Ensuite le *backward pass* est équivalent au *back-propagation*, nous calculons les dérivées partielles en partant du dernier étage en remontant en arrière vers le premier étage.

Apprentissage de réseaux de neurones

- L'utilisation de l'algorithme du gradient rajoute encore plus d'hyperparamètres
- Nous avions déja le nombre de couche, le nombre de neuronnes par couche et le choix de fonction d'activation non-linéaire g (hyperparamètre standard)
- Nous avons maintenant des hyperparamètre d'optimisation comme le nombre d'itération, la taille du pas, le type d'algorithme etc...
- L'idée reste d'utiliser la validation pour déterminer les hyperparamètres mais il y en a beaucoup.
- On se contente d'un choix *correct*: un réseau de neuronne non-optimal est quand même mieux qu'une fonction linéaire!

Réseaux de neurones

On peut généraliser ces modèles de toutes sortes de manières, différentes sorties, différents formats d'entrée, différents formats (auto-encodeur), différentes contraintes sur les connexions (CNN, RNN).
Bref, on en fait ce qu'on en veut à mesure qu'on est capable de tracer le chemin des entrées aux sorties à l'aide d'opérations différentiables.
Connaître les fonctions existantes dans les librairies *auto-diff* peut nous aider à conceptualiser de nouveaux modèles faciles à tester.

Pour le prochain cours:

- Lecture: 10.1, 10.2, 10.3, 10.6 et 10.7
- Exercices: **10.10.1**, 10.10.3, exercice à la slide 14
- Installer keras (`https://tensorflow.rstudio.com/install/`)
- Lab : 10.9.1, 10.9.2 (Difficile)
- Tutorial keras : `https://tensorflow.rstudio.com/tutorials/keras/classification`

Références

Friedman, J., Hastie, T., Tibshirani, R., et al. (2001). *The elements of statistical learning*. Springer.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133.

Rosenblatt, F. (1961). Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY.