Entrainement des RN profonds

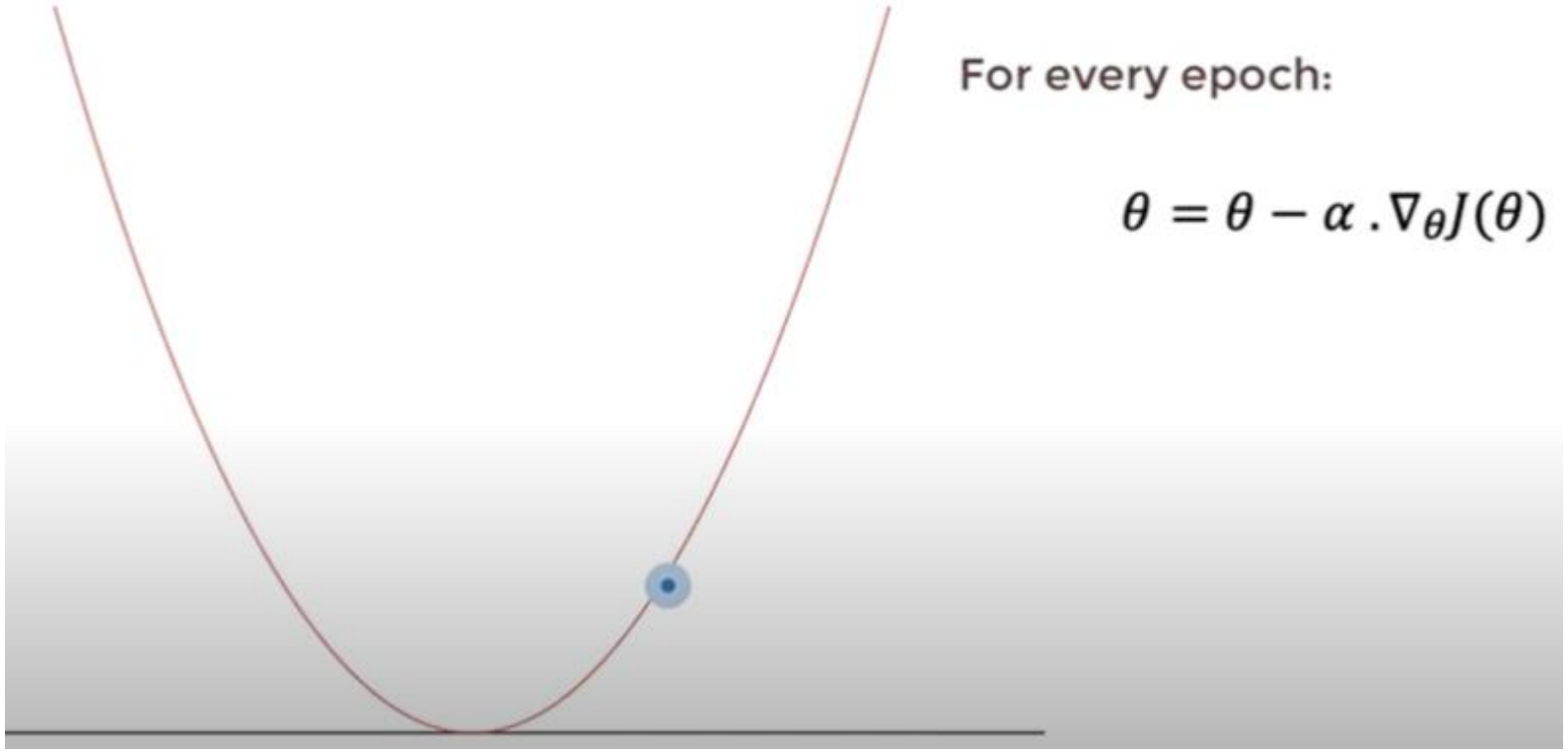
Sommaire

- Les optimiseurs
- Eviter le surajustement grâce à la régularisation
- Problèmes de disparition et d’explosion des gradients

Les optimiseurs

Gradient Descent

(Batch Gradient Descent)



For every epoch:

$$\theta = \theta - \alpha \, . \nabla_{\theta} J(\theta)$$

Exemple : cas de la régression linéaire

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Cette équation introduit des notations que nous emploierons tout au long de ce livre :

- $\hat{y}$ est la valeur prédite ($\hat{y}$ se prononce généralement « y-chapeau »),
- n est le nombre de variables,
- x_i est la valeur de la $i^{ième}$ variable,

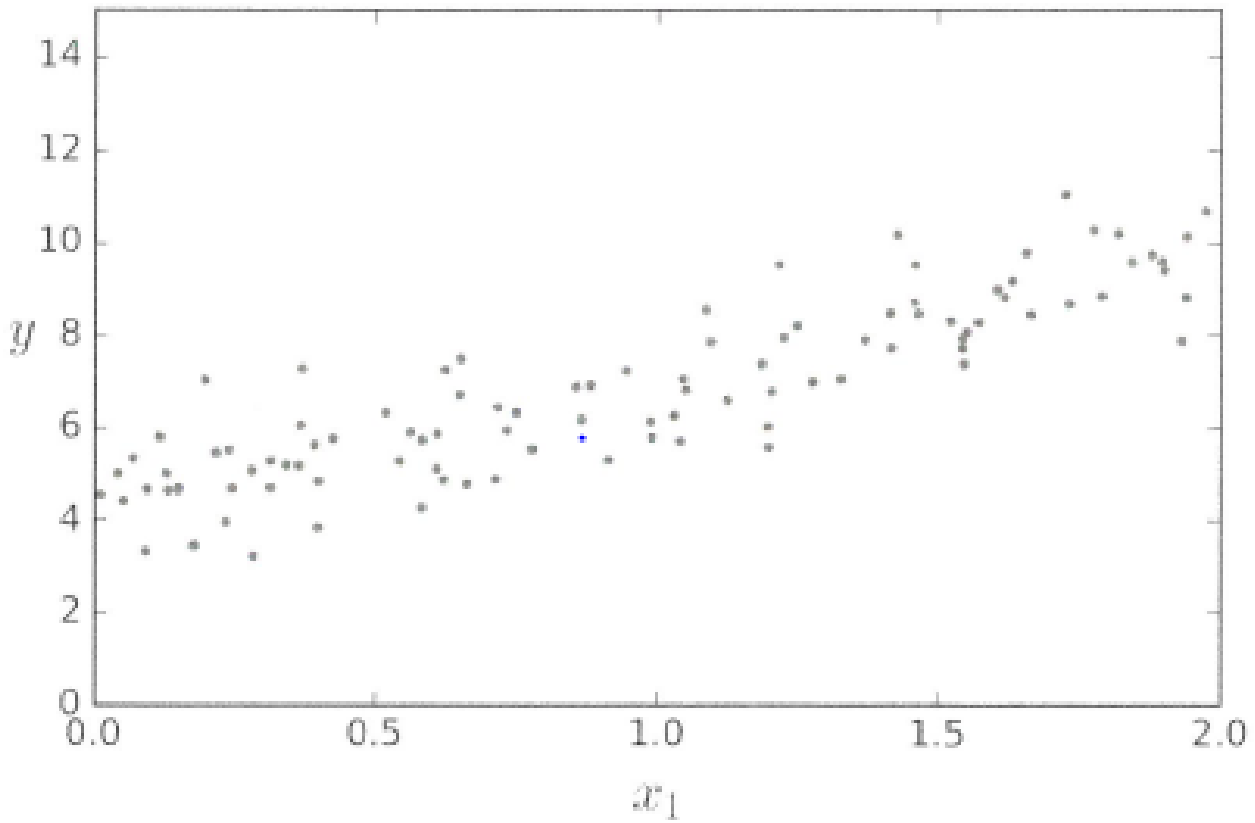
Pour entraîner un modèle de régression linéaire, il s'agit de trouver le vecteur $\boldsymbol{\theta}$ qui minimise la RMSE.

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left[h(\mathbf{x}^{(i)}) - y^{(i)}\right]^2}$$

Solution analytique par l'équation normale

Pour trouver la valeur de $\boldsymbol{\theta}$ qui minimise la fonction de coût, il s'avère qu'il existe une *solution analytique*, c'est-à-dire une formule mathématique nous fournissant directement le résultat. Celle-ci porte le nom d'*équation normale*[7].

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```



$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Voyons ce que l'équation a trouvé :

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

Solution par la Descente de gradient

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^{m} (\boldsymbol{\theta}^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \boldsymbol{\theta} - \mathbf{y})$$

Une fois que vous avez le vecteur gradient (qui pointe vers le haut), il vous suffit d'aller dans la direction opposée pour descendre. Ce qui revient à soustraire $\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$ de $\boldsymbol{\theta}$. C'est ici qu'apparaît le taux d'apprentissage η[12] : multipliez le vecteur gradient par η pour déterminer le pas de la progression vers le bas :

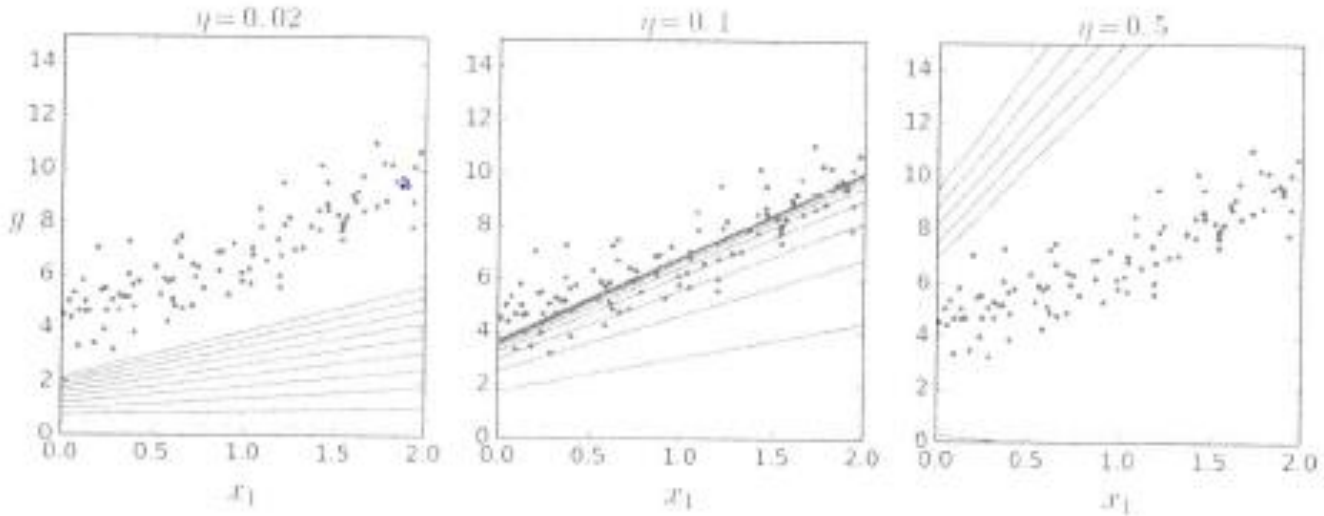
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

Solution par la Descente de gradient

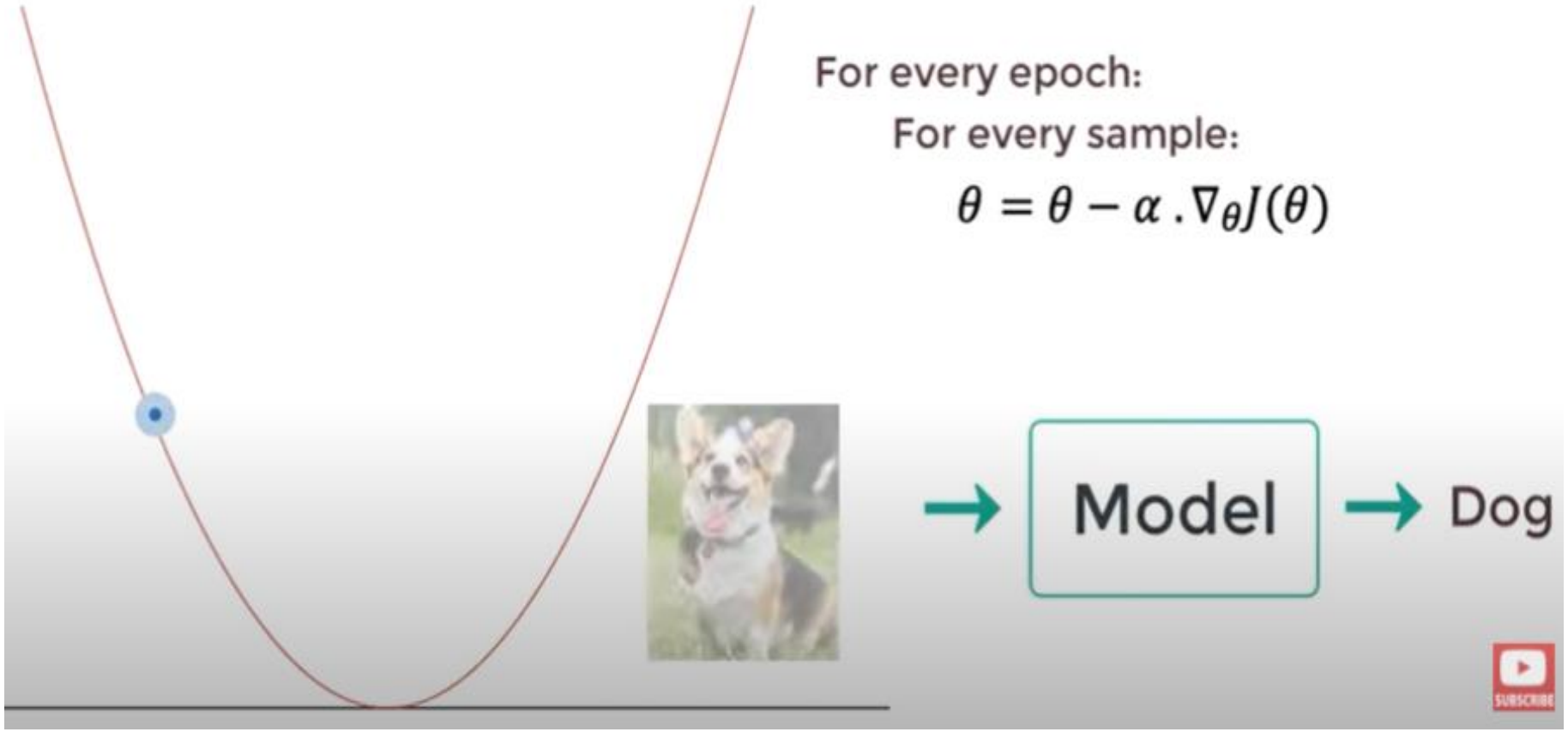
Voyons une implémentation rapide de cet algorithme :

```
eta = 0.1 # taux d'apprentissage
n_iterations = 1000
m = 100 # le nombre d'observations (len(X))

theta = np.random.randn(2,1) # initialisation aléatoire

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```



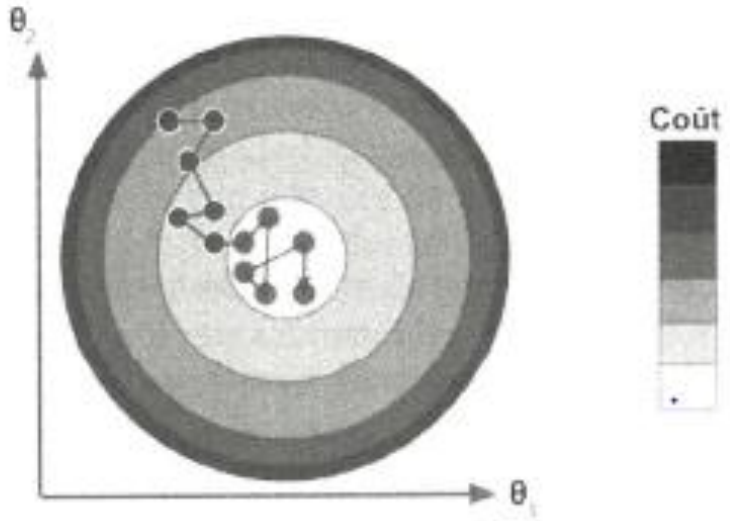
Stochastic Gradient Descent (SGD)



Solution par la SGD

Par contre, étant donné sa nature stochastique (c'est-à-dire aléatoire), cet algorithme est beaucoup moins régulier qu'une descente de gradient ordinaire: au lieu de décroître doucement jusqu'à atteindre le minimum, la fonction de coût va effectuer des sauts vers le haut et vers le bas et ne décroîtra qu'en moyenne. Au fil du temps, elle arrivera très près du minimum, mais une fois là elle continuera à effectuer des sauts aux alentours sans jamais s'arrêter (voir la figure 1.12). Par conséquent, lorsque l'algorithme est stoppé, les valeurs finales des paramètres sont bonnes, mais pas optimales.

Parmi les avantages de cette méthode, la possibilité d'effectuer des mises à jour du résultat à l'aide de nouvelles observations.



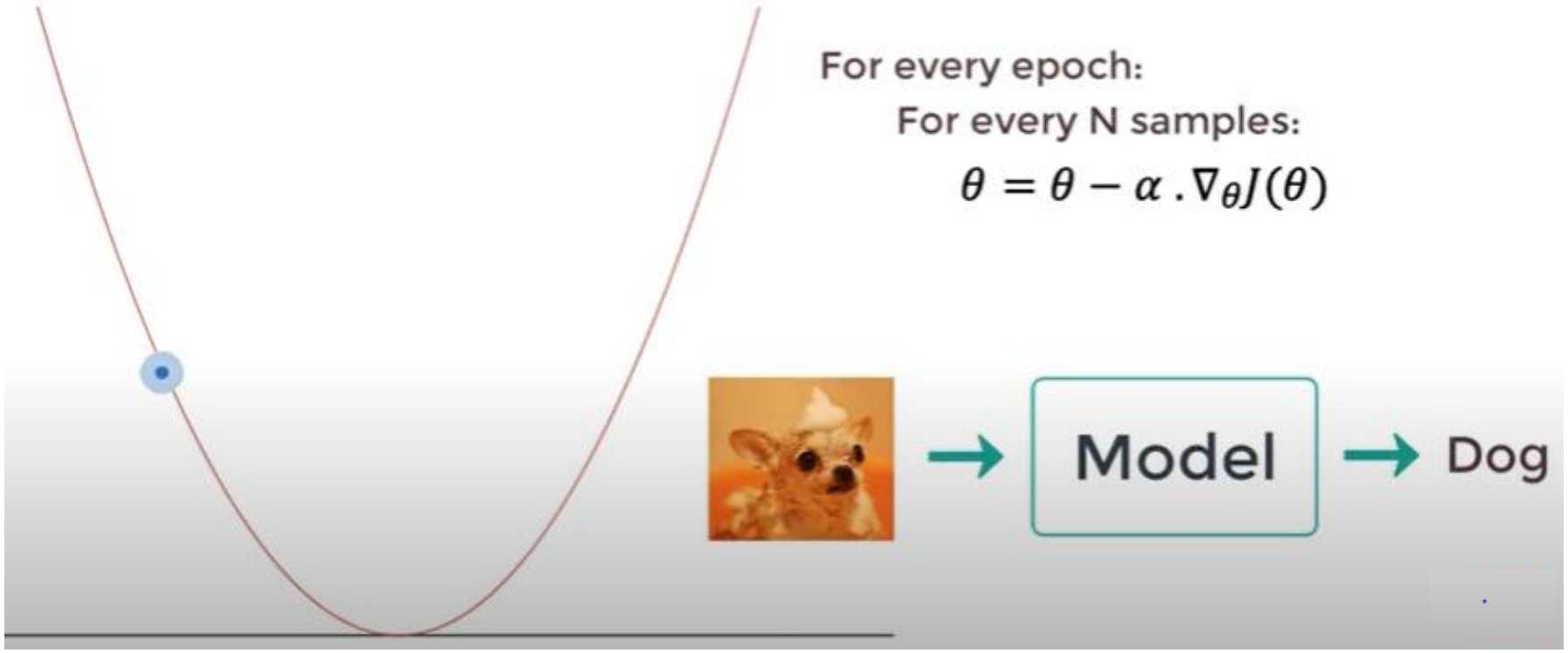
Solution par le SGD

```
n_epochs = 50
t0, t1 = 5, 50 # hyperparamètres d'échéancier d'apprentissage

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # init. aléatoire

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

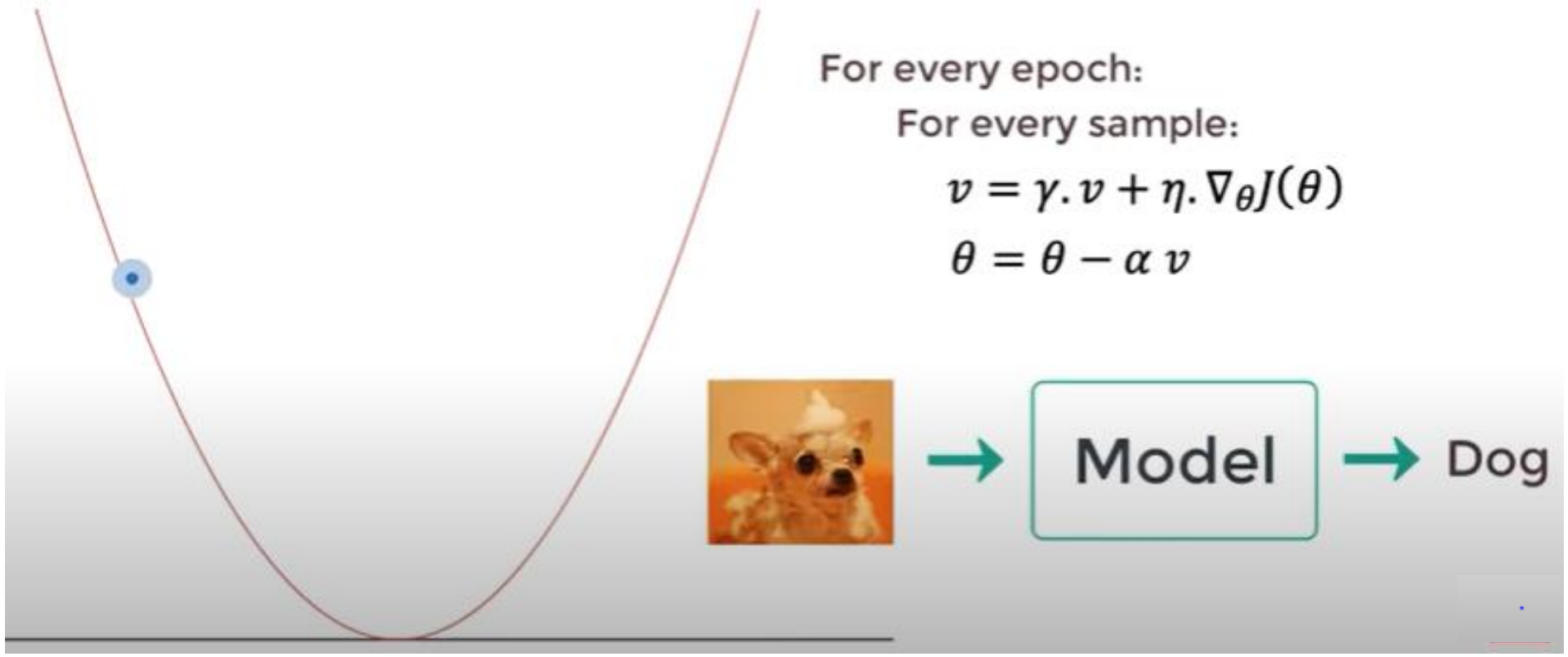
Par convention, nous effectuons des cycles de m itérations (rappelons que m est le nombre d'observations dans le jeu de données d'entraînement) : chacun de ces cycles s'appelle une *époque* (en anglais, *epoch*). Alors que le code de la descente de gradient ordinaire présenté plus haut effectue 1 000 itérations sur l'ensemble du jeu d'apprentissage, ce code-ci ne parcourt le jeu d'entraînement qu'environ 50 fois et aboutit à une solution plutôt satisfaisante :

```
>>> theta
array([[ 4.21076011],
       [ 2.74856079]])
```

Mini-Batch Gradient Descent



SGD + Momentum



SGD + Momentum (suite)

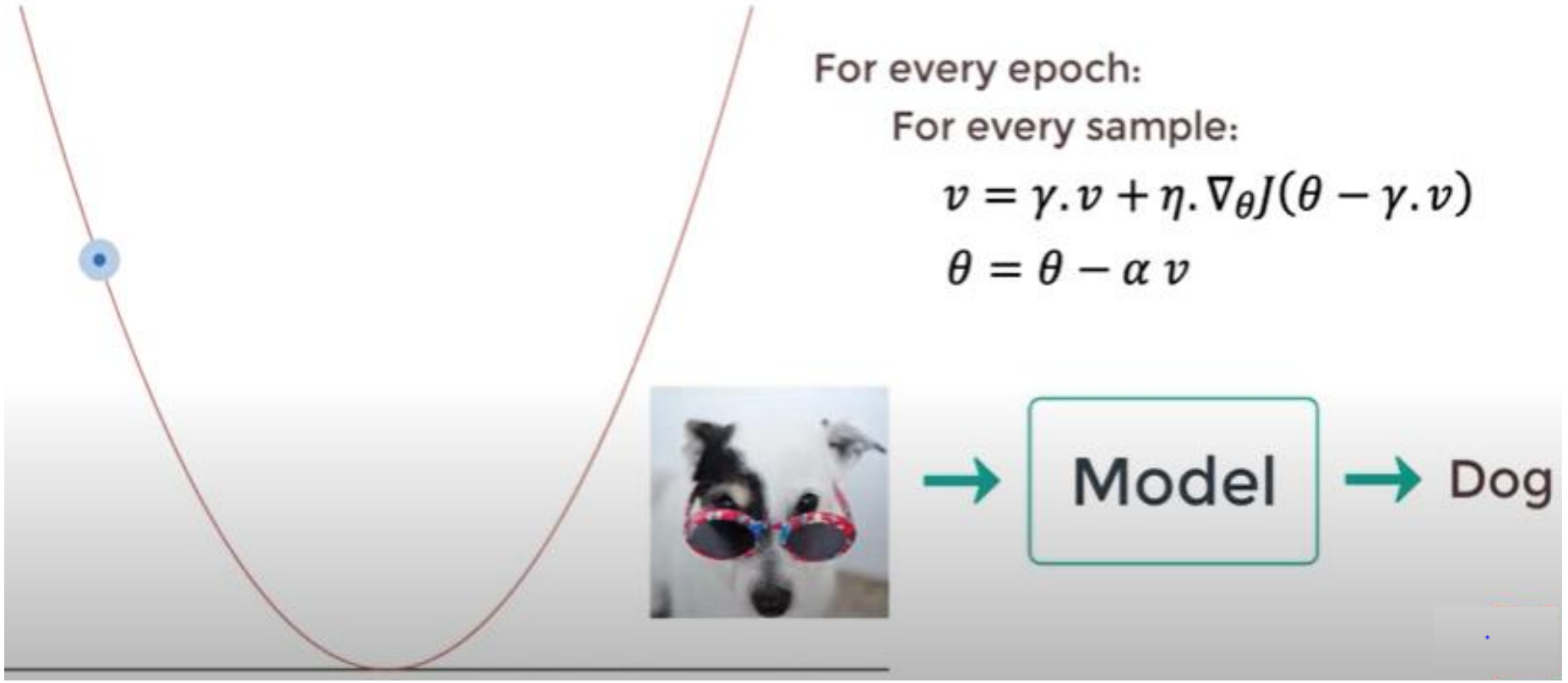
Imaginons une boule de bowling qui roule sur une surface lisse en légère pente : elle démarre lentement, mais prend rapidement de la vitesse jusqu'à atteindre une vélocité maximale (s'il y a des frottements). Voilà l'idée simple derrière la *momentum optimisation* proposée par B. Polyak (1964).

A l'opposée la descente de gradient classique ferait de petits pas réguliers vers le bas de la pente et mettrait donc plus de temps pour arriver à son extrémité. Autrement dit, le gradient est utilisé non pas comme un facteur de vitesse mais d'accélération .

Pour simuler une forme de frottement et éviter que la vitesse ne s'emballe, l'algorithme introduit un nouvel hyperparamètre γ appelé simplement inertie (*momentum*) dont la valeur doit être comprise entre 0 (frottement élevé) et 1 (aucun frottement). Une valeur fréquemment utilisée est 0.9.

Nesterov Accelerated Gradient (NAG)

(SGD + Momentum + Acceleration)

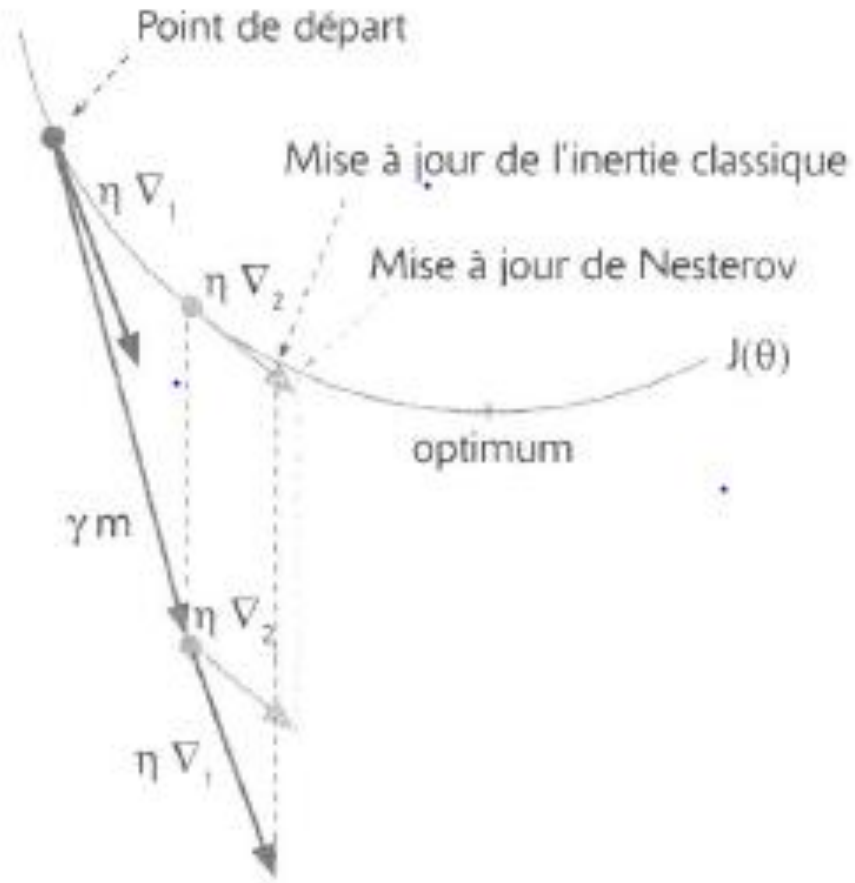


NAG

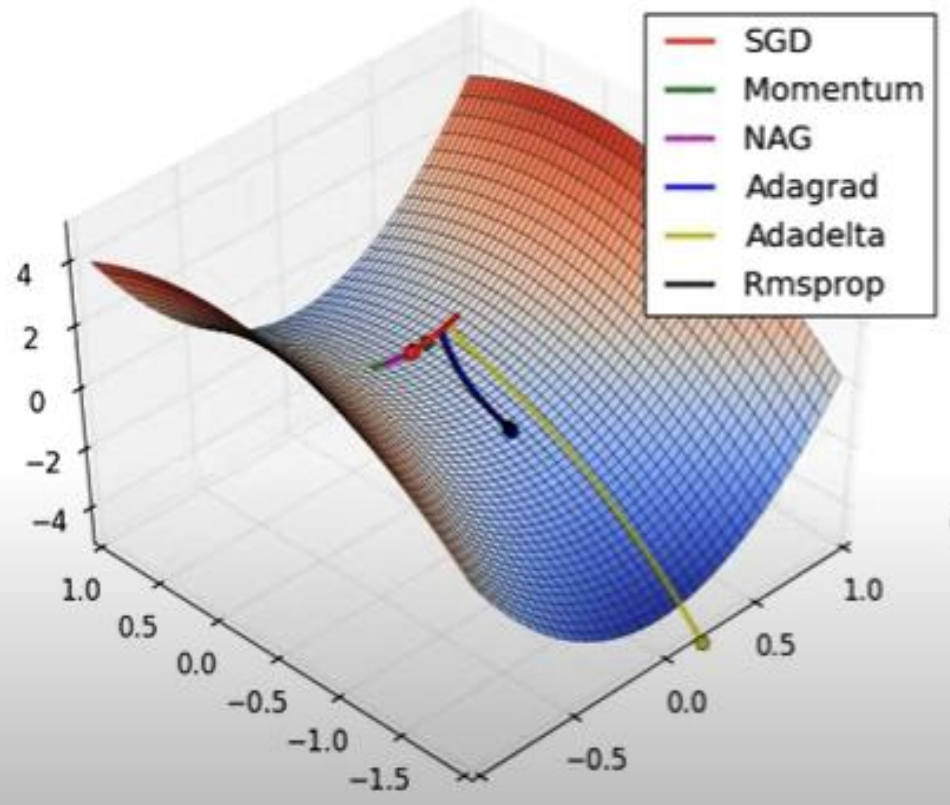
Ce petit ajustement fonctionne, car en général, le vecteur d'inertie pointe dans la bonne direction. Il sera donc plus précis d'utiliser le gradient mesuré un peu plus en avant dans cette direction que d'utiliser celui en position d'origine comme le montre la figure suivante où :

$\eta\Delta_1$ représente le gradient en Θ et $\eta\Delta_2$ représente le gradient en $\Theta - \gamma v$

Il est alors facile de constater que la mise à jour de NAG arrive un peu plus près de l'optimum



AdaGrad



For every epoch t :

For every parameter θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

$$G_{t,ii} = G_{t-1,ii} + \nabla^2_{\theta_{t,i}} J(\theta_{t,i})$$

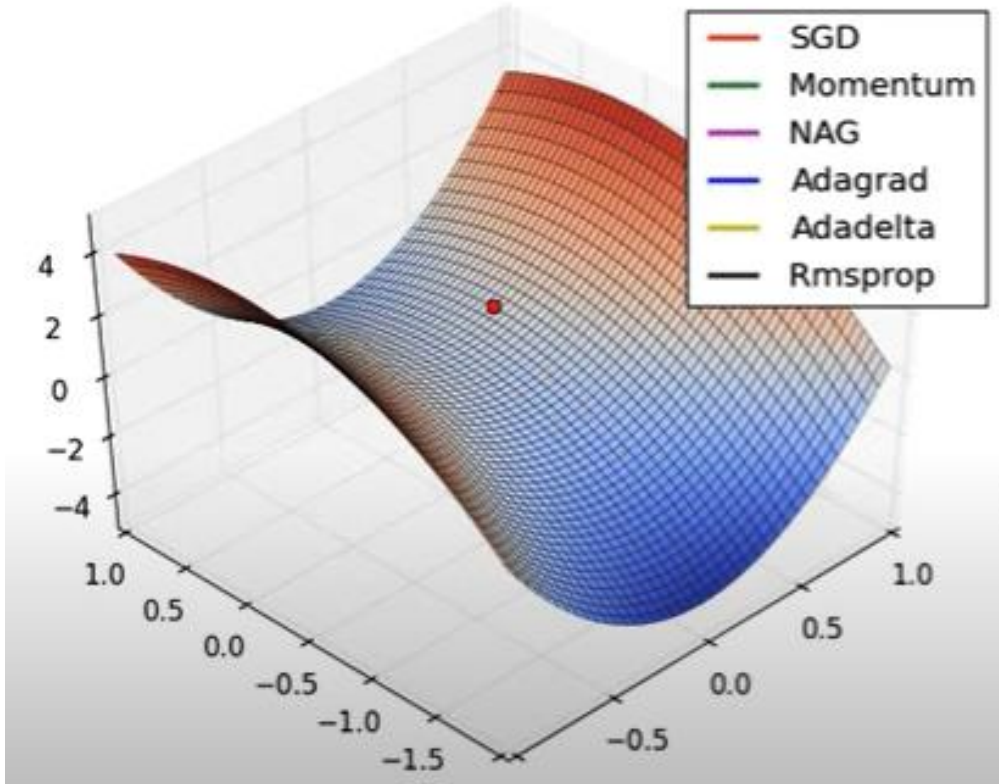
or

$$G_{t,ii} = \nabla^2_{\theta_{1,i}} J(\theta_{1,i}) + \nabla^2_{\theta_{2,i}} J(\theta_{2,i}) + \cdots + \nabla^2_{\theta_{t,i}} J(\theta_{t,i})$$

AdaGrad

En résumé, cet algorithme abaisse progressivement le taux d'apprentissage, mais il le fait plus rapidement sur les dimensions présentant une pente abrupte que pour celles dont la pente est plus douce. Nous avons donc un *taux d'apprentissage adaptatif*. Cela permet de diriger plus directement les mises à jour résultantes vers l'optimum global (voir la figure 4.7). Par ailleurs, l'algorithme exige un ajustement moindre de l'hyperparamètre η pour le taux d'apprentissage.

RMSProp



For every epoch t :

For every parameter θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}] + \epsilon}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

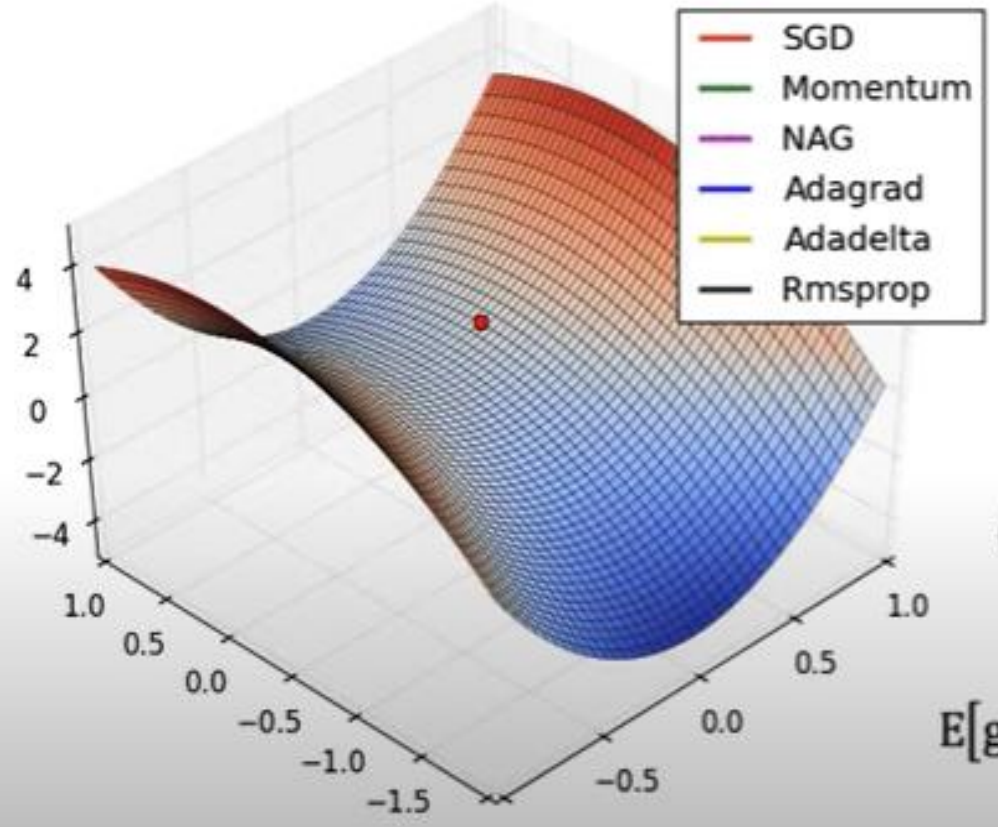
$$G_{t,ii} = \gamma G_{t-1,ii} + (1-\gamma) \nabla^2_{\theta_{t,i}} J(\theta_{t,i})$$

RMSProp

L'inconvénient d'AdaGrad est de ralentir un peu trop rapidement et de finir par ne jamais converger vers l'optimum global. L'algorithme *RMSProp*[58] corrige ce problème en cumulant uniquement les gradients issus des itérations récentes (plutôt que tous les gradients depuis le début l'entraînement). Pour cela, il utilise une moyenne mobile exponentielle au cours de la première étape

```
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])
```

Adam



For every epoch t :

For every parameter θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}] + \epsilon}} \times \mathrm{E}[g_{t,i}]$$

$$G_{t,ii} = \gamma G_{t-1,ii} + (1-\gamma)\nabla^2_{\theta_{t,i}} J(\theta_{t,i})$$

$$\mathrm{E}[g_{t,i}] = \beta E[g_{t-1,i}] + (1-\beta)\nabla_{\theta_{t,i}} J(\theta_{t,i})$$

Adam

Adam[59], pour *Adaptive Moment Estimation*, réunit les idées de l'optimisation avec inertie et de RMSProp. Il maintient, à l'instar de la première, une moyenne mobile exponentielle des gradients antérieurs et, à l'instar de la seconde, une moyenne mobile exponentielle des carrés des gradients passés

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=["accuracy"]) # Configure the model for training
```

Liens

Tutoriels :

https://ichi.pro/fr/vue-d-ensemble-des-differents-optimiseurs-pour-les-reseaux-de-neurones-247308806799555

https://towardsdatascience.com/full-review-on-optimizing-neural-network-training-with-optimizer-9c1acc4dbe78

https://ruder.io/optimizing-gradient-descent/index.html#momentum

Sur Youtube :

Optimizers - EXPLAINED!
https://www.youtube.com/watch?v=mdKjMPmcWjY

Stochastic Gradient Descent, Clearly Explained!!!
https://www.youtube.com/watch?v=vMh0zPT0tLI&t=266s&ab_channel=StatQuestwithJoshStarmer

Eviter le surajustement grâce à la régularisation

Early Stopping

A problem with training neural networks is in the choice of the number of training epochs to use.

Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the "*patience*" argument.

```
1 es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
```

The exact amount of patience will vary between models and problems. Reviewing plots of your performance measure can be very useful to get an idea of how noisy the optimization process for your model on your data may be.

Dropout

Dans le contexte du DL, la technique de régularisation la plus répandue est sans aucun doute celle du *Dropout*. Il a été prouvé que les RN les plus performants voient leur performance s'améliorer de 1 à 2% par simple ajour du Dropout.

L'algorithme est relativement simple. À chaque étape d'entraînement, chaque neurone (y compris les neurones d'entrée, mais pas les neurones de sortie) a une probabilité p d'être temporairement « éteint »[67]. Autrement dit, il sera totalement ignoré au cours de cette étape d'entraînement, mais il pourra être actif lors de la suivante (voir la figure 4.9). L'hyperparamètre p est appelé *taux d'extinction* (*dropout rate*) et il est en général fixé à 50 %.

```
model= Sequential([
    Dense(256, input_dim = 784, activation='relu'),
    Dropout(0.5),
    Dense(128, input_dim = 256, activation='relu'),
    Dense(10, activation='softmax') ])
```

Dropout

Il reste un petit détail technique qui a son importance. Supposons que p = 50 %, alors, au cours des tests, un neurone sera connecté à deux fois plus de neurones d'entrée qu'il ne l'a été (en moyenne) au cours de l'entraînement. Pour compenser cela, il faut multiplier les poids des connexions d'entrée de chaque neurone par 0,5 après l'entraînement. Dans le cas contraire, chaque neurone recevra un signal d'entrée total environ deux fois plus grand qu'au cours de l'entraînement du réseau. Il est donc peu probable qu'il se comporte correctement. Plus généralement, après l'entraînement, il faut multiplier les poids des connexions d'entrée par la *probabilité de conservation* (*keep probability*) $1 - p$. Une autre solution consiste à diviser la sortie de chaque neurone par la probabilité de conservation, pendant l'entraînement (ces deux approches ne sont pas totalement équivalentes, mais elles fonctionnent aussi bien l'une que l'autre).

Régularisation *l1* et *l2*

Comme pour les modèles linéaires simples, il est possible d'employer la régularisation *l1* et *l2* pour contraindre les poids des connexions d'un RN (mais ne général pas ses termes constants).

Lasso regression cost function (*l1*) : $$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^{n} | \theta_i |$$

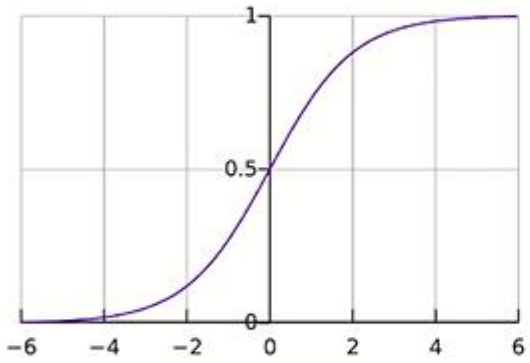
Ridge regression cost function (*l2*) : $$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

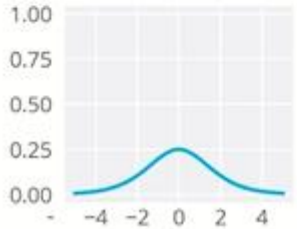
Problèmes de disparition et d’explosion des gradients

Fonctions d'activation

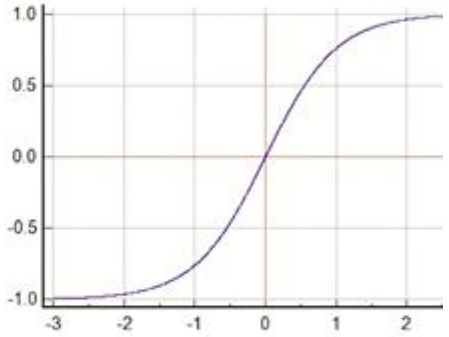
Malheureusement, alors que l'algorithme progresse vers les couches inférieures, les gradients deviennent souvent de plus en plus petits. Par conséquent, la mise à jour par descente de gradient modifie très peu les poids des connexions de la couche inférieure et l'entraînement ne converge jamais vers une bonne solution. Tel est le problème de *disparition des gradients* (*vanishing gradients*)

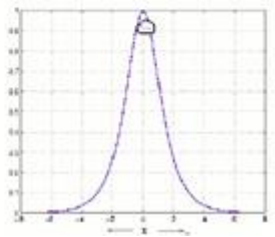
Sigmoid





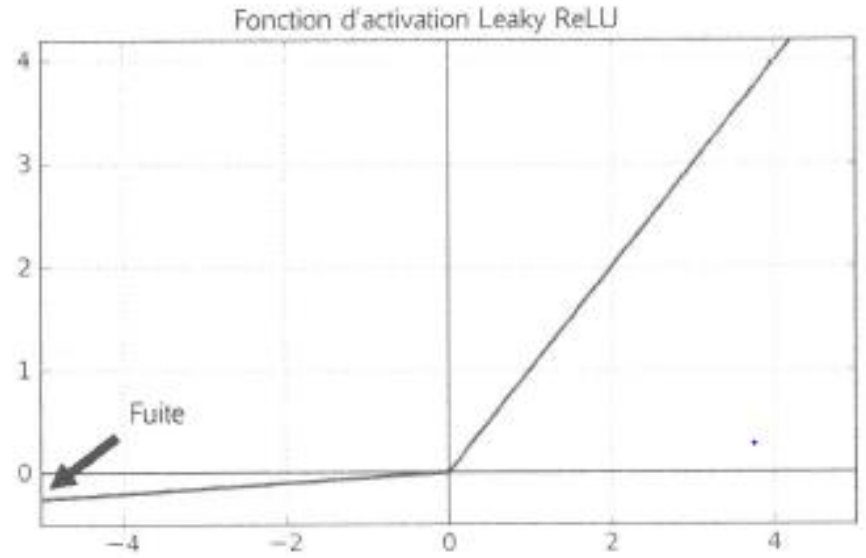
Tanh





Fonctions d'activation

Si nous examinons la fonction d'activation logistique (voir la figure 4.1), nous constatons que, lorsque les entrées deviennent grandes (en négatif ou en positif), la fonction sature en 0 ou en 1, avec une dérivée extrêmement proche de 0. Lorsque la rétropropagation intervient, elle n'a pratiquement aucun gradient à transmettre en arrière dans le réseau. En outre, le faible gradient existant est de plus en plus dilué pendant la rétropropagation, à chaque couche traversée. Il ne reste donc quasi plus rien aux couches inférieures.



Initialisation de Xavier

Dans leur article, Glorot et Bengio proposent une manière d'atténuer énormément ce problème. Le signal doit se propager correctement dans les deux directions : vers l'avant au moment des prédictions, et vers l'arrière lors de la rétropropagation des gradients. Il ne faut pas que le signal disparaisse, ni qu'il explose et sature. Pour que tout se passe proprement, les auteurs soutiennent que la variance des sorties de chaque couche doit être égale à la variance de ses entrées[43] et les gradients doivent également avoir une même variance avant et après le passage au travers d'une couche en sens inverse (les détails mathématiques se trouvent dans l'article). Il est impossible de garantir ces deux points, sauf si la couche possède un nombre égal de connexions d'entrée et de sortie. Ils ont cependant proposé un bon compromis, dont la validité a été montrée en pratique : les poids des connexions doivent être initialisés de façon aléatoire, comme dans l'équation 4.1, où $n_{\text{entrées}}$ et n_{sorties} sont les nombres de connexions en entrée et en sortie pour la couche dont les poids sont en cours d'initialisation (également appelés *fan-in* et *fan-out*). Cette stratégie d'initialisation est souvent appelée *initialisation de Xavier* (d'après le prénom de l'auteur) ou, parfois, *initialisation de Glorot*.

Initialisation de Xavier

Fonction d'activation	Distribution uniforme $[-r, r]$	Distribution normale
Logistique	$r = \sqrt{\frac{6}{n_{\text{entrées}} + n_{\text{sorties}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{entrées}} + n_{\text{sorties}}}}$
Tangente hyperbolique	$r = 4\sqrt{\frac{6}{n_{\text{entrées}} + n_{\text{sorties}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{entrées}} + n_{\text{sorties}}}}$
ReLU (et ses variantes)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{entrées}} + n_{\text{sorties}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{entrées}} + n_{\text{sorties}}}}$

Initialisation de Xavier

`GlorotNormal` class

```
tf.keras.initializers.GlorotNormal(seed=None)
```

The Glorot normal initializer, also called Xavier normal initializer.

Also available via the shortcut function `tf.keras.initializers.glorot_normal`.

Draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Batch normalisation

Dans un article[49] de 2015, Sergey Ioffe et Christian Szegedy ont proposé une technique appelée *normalisation par lots* (BN, *Batch Normalization*) pour traiter les problèmes des gradients et, plus généralement, le problème lié au fait que, pendant l'entraînement, la distribution des entrées de chaque couche évolue en fonction des changements des paramètres des couches précédentes (ce qu'ils appellent le problème de *Internal Covariate Shift*).

Leur technique consiste à ajouter une opération dans le modèle qui se contente de centrer sur zéro et normaliser ses entrées, puis à mettre à l'échelle et à décaler le résultat en utilisant deux nouveaux paramètres par couche (l'un pour la mise à l'échelle, l'autre pour le décalage). Autrement dit, cette opération permet au modèle d'apprendre l'échelle et la moyenne optimales des données. On place généralement une couche BN à la sortie de chaque couche du réseau de neurones, généralement juste avant sa fonction d'activation[50].

Batch normalisation

1. $$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2. $$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

3. $$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

4. $$z^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$$

- μ_B est la moyenne empirique, évaluée sur l'intégralité du mini-lot B.
- m_B est le nombre d'instances dans le mini-lot.
- $\mathbf{x}^{(i)}$ est ici l'entrée de la couche BN considérée.
- σ_B est l'écart-type empirique, également évalué sur l'intégralité du mini-lot.
- $\hat{\mathbf{x}}^{(i)}$ est l'entrée centrée sur zéro et normalisée.
- γ est le paramètre de mise à l'échelle pour la couche.
- β est le paramètre de décalage pour la couche.
- ϵ est un nombre minuscule pour éviter toute division par zéro (en général 10^{-3}). Il s'agit d'un *terme de lissage*.
- $z^{(i)}$ est la sortie de l'opération de normalisation : la version mise à l'échelle et décalée des entrées de la couche BN.

Batch normalisation

```
model = Sequential([
    Dense(64, input_shape=(4,), activation="relu"),
    BatchNormalization(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dense(64, activation='relu'),
    BatchNormalization(),
    Dense(64, activation='relu'),
    BatchNormalization(),
    Dense(3, activation='softmax')
]);
```

Augmentation des données

Une dernière technique de régularisation consiste à utiliser les instances d'entraînement existantes pour en générer de nouvelles, grossissant ainsi artificiellement la taille du jeu d'entraînement. Cette méthode permet de réduire le surajustement et donc d'en faire une technique de régularisation. L'astuce est de générer des instances d'entraînement réalistes. Idéalement, un être humain ne devrait pas être capable de faire la distinction entre les instances artificielles et les autres.

Par exemple, si notre modèle a pour objectif de classer des photos de champignons, nous pouvons légèrement décaler, pivoter et redimensionner chaque image du jeu d'entraînement, en variant l'importance des modifications, puis ajouter les images résultantes au jeu d'entraînement

Il est souvent préférable de générer des instances d'entraînement à la volée pendant l'entraînement plutôt que de gaspiller de l'espace de stockage et de la bande passante réseau.

Augmentation des données

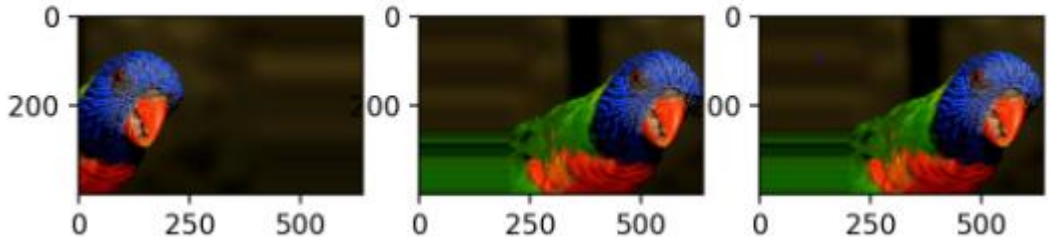
```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False, samplewise_center=False,
    featurewise_std_normalization=False, samplewise_std_normalization=False,
    zca_whitening=False, zca_epsilon=1e-06, rotation_range=0, width_shift_range=0.0,
    height_shift_range=0.0, brightness_range=None, shear_range=0.0, zoom_range=0.0,
    channel_shift_range=0.0, fill_mode='nearest', cval=0.0,
    horizontal_flip=False, vertical_flip=False, rescale=None,
    preprocessing_function=None, data_format=None, validation_split=0.0, dtype=None
)
```

Augmentation des données

Horizontal and Vertical Shift Augmentation

A shift to an image means moving all pixels of the image in one direction, such as horizontally or vertically, while keeping the image dimensions the same

```
datagen = ImageDataGenerator(width_shift_range=[-200,200])
```



Random Rotation Augmentation

A rotation augmentation randomly rotates the image clockwise by a given number of degrees from 0 to 360.

```
datagen = ImageDataGenerator(rotation_range=90)
```