Entrainement des RN profonds

Sommaire

1. Optimiseurs plus rapides
2. Eviter le surajustement grâce à la régularisation
3. Problèmes de disparition et d’explosion des gradients

1. Optimiseurs plus rapides

- SGD + Momentum

Algorithme : For every epoch:
For every sample:

$$v = \gamma . v + \eta . \nabla_\theta J(\theta)$$

$$\theta = \theta - \alpha\, v$$

- 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.

- Dans ce contexte : au lieu de simplement ajuster les paramètres en fonction du gradient à chaque itération, le *momentum* prend en compte les mises à jour précédentes, en "mémorisant" les directions de descente précédentes. Cela permet d'atteindre une convergence plus rapide, voilà l'idée simple derrière la *momentum optimisation* !

- 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.

Remarque : Pour simuler le frottement, l'algorithme introduit l'hyperparamètre γ appelé momentum . Une valeur fréquemment utilisée est 0.9.

Nesterov Accelerated Gradient (NAG)

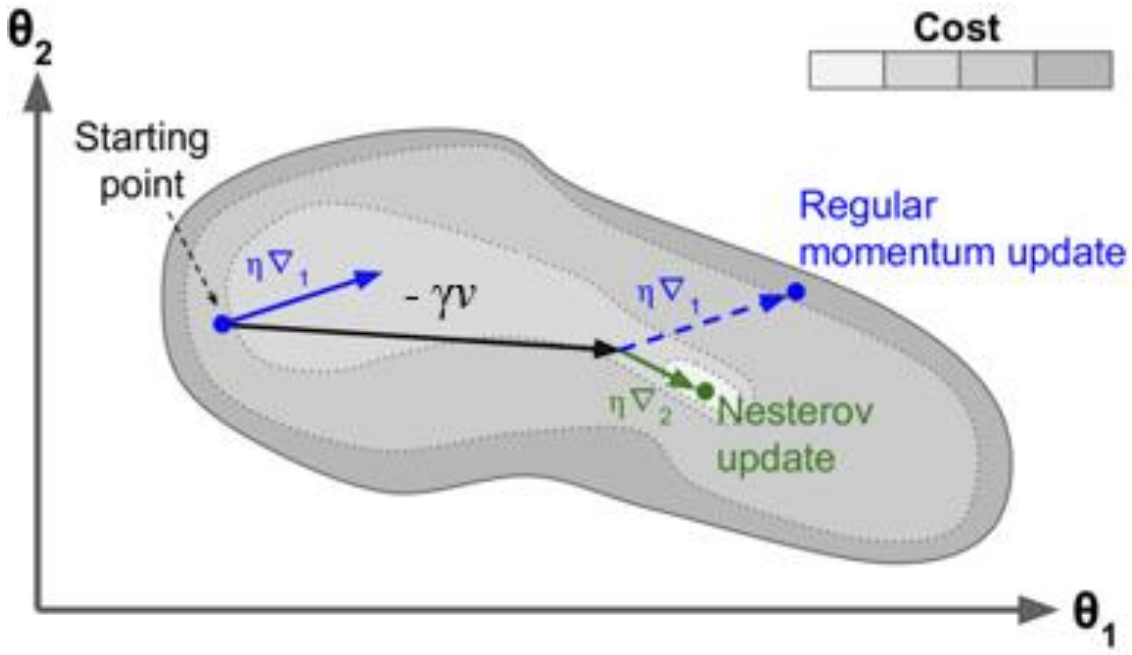
- En général, le vecteur d'inertie – γv pointe dans la bonne direction
- Il s'agit d'utiliser le gradient mesuré un peu plus en avant dans cette direction que d'utiliser celui en position d'origine : $\eta\Delta 1$ représente le gradient en Θ et $\eta\Delta 2$ représente le gradient en $\Theta - \gamma v$.
- Il est alors facile de constater que la mise à jour de NAG arrive un peu plus près de l'optimum

Algorithme : For every epoch:

For every sample:

$$v = \gamma . v + \eta . \nabla_\theta J(\theta - \gamma . v)$$

$$\theta = \theta - \alpha\, v$$



Adagrad

- La première étape accumule le carré des gradients dans le vecteur $G_{t,ii}$
- La seconde est quasi identique à la descente de gradient, mais avec une différence importante : le gradient est diminué du facteur $\sqrt{G_{t,ii}} + \epsilon$

Algorithme : 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

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

- Cet algorithme abaisse progressivement le taux d'apprentissage, mais il le fait plus rapidement sur les dimensions présentant une pente raide que pour celle dont la pente est plus douce
- Nous avons donc un taux d'apprentissage adaptatif

RMSprop

- L'inconvénient d'Adagrad est de ralentir un peu trop rapidement et de finir par ne jamais converger vers l'optimum global.
- RMSprop corrige ce problème en cumulant uniquement les gradients issus des itération récentes.
- Pour cela il utilise une moyenne exponentielle au cours de la première étape.

Algorithme : For every epoch t :

For every parameter θ_i :

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

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

Adam

- Adam pour *Adaptive Moment Estimation*, réunit les idées de l'optimisation avec inertie et RMSprop :
- Il maintient, à l'instar de la première, une moyenne mobile exponentielle des gradient antérieurs
- Et, à l'instar de la seconde, une moyenne mobile exponentielle des carrés des gradients passés

Algorithme : For every epoch t :

For every parameter θ_i:

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

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

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

2. Eviter le surajustement : Régularisation

▪ Early Stopping

- Un problème avec les NN est le choix du nombre d'epoch. L'early stopping est une technique très intuitive qui consiste à arrêter l'entraînement avant que le modèle ne commence à overfitter.

- Il s'agit de spécifier un nombre assez grand d'epoch puis d'arrêter l'entraînement une fois que les performances du modèles commencent à se détériorer (cf. *accuarcy* ou *loss*)

```
from keras.callbacks import EarlyStopping
# Set callback functions to early stop training
mycallbacks = [EarlyStopping(monitor='val_loss', patience=2)]
```

```
history = model.fit(
  train_images, #inputs
  to_categorical(train_labels), #target vector
  epochs=5, # number of epochs
  batch_size=32,
  callbacks = mycallbacks, # early stopping
  validation_data=(test_images, to_categorical(test_labels)))
```

- Le paramètre patience de la fonction Earlystopping permet de fixer un délai au déclenchement en termes de nombre d'époques sur lesquelles nous ne souhaiterions voir aucune amélioration.

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 NN les plus performants voient leur performance s'améliorer de 1 à 2% par simple ajout du Dropout.

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

- A chaque étape d'entraînement, chaque neurone (y compris les neurones d'entrées, mais pas les neurones de sorties) a une probabilité d'être temporairement éteint.
- Il sera ignoré au cours de cette étape d'entraînement, mais il pourra être actif lors de la suivante
- L'hyperparamètre est appelé taux d'extinction (dropout rate) et il est, en général de 50%.

Dropout (suite)

- Le Dropout est actif uniquement durant l'entraînement du modèle.
- Supposons que p=0.5, alors, au cours du test, 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, lors des tests, il faut multiplier les poids des connexions d'entrées par la probabilité de conservation (*keep probability*) 1-p.

Régularisation *l1, l2* et *Elastic Net*

- 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*) : $J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i|$
- Ridge regression cost function (*l2*) : $J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$
- Elastic Net : $J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2$

```
from tensorflow.keras import regularizers
layer = Dense(5, input_dim=5, kernel_regularizer=regularizers.l1(0.01))
```

Le 0.01 est le coefficient qui contrôle l'intensité de la pénalité L1.

Augmentation des données (1)

- Une dernière technique de régularisation consiste à utiliser les instances d'entraînement pour en générer de nouvelles.

- La plupart du temps toutes nos données disponibles sont déjà mobilisées. Pour contrer cela des techniques d'augmentation de données existent.

- Par exemple en computer vision, lorsque nos données sont des images, on peut créer des filtres pour modifier légèrement les couleurs. On peut pivoter les images ou étirer certains traits. Cela permet de réduire les risques d'overfitting. En plus, cette technique peut être mise en place très facilement, la plupart des outils et des librairies de machine learning proposent l'option nativement.

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

Augmentation des données (2)

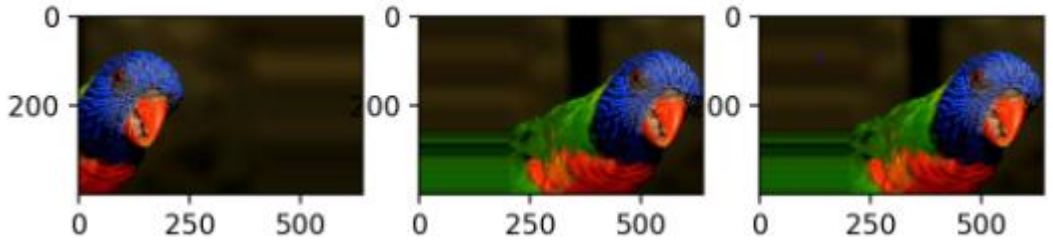
```
train_aug = ImageDataGenerator(
    rescale=1./255,
    horizontal_flip=True,
    height_shift_range=0.1,
    width_shift_range=0.1,
    brightness_range=(0.5,1.5),
    zoom_range = [1, 1.5],
)
train_aug_ds = train_aug.flow_from_directory(
    directory='./train',
    target_size=image_size,
    batch_size=batch_size,
)

model.fit(
    train_aug_ds,
    epochs=150,
    validation_data=(valid_aug_ds,),
)
```

Augmentation des données (3)

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

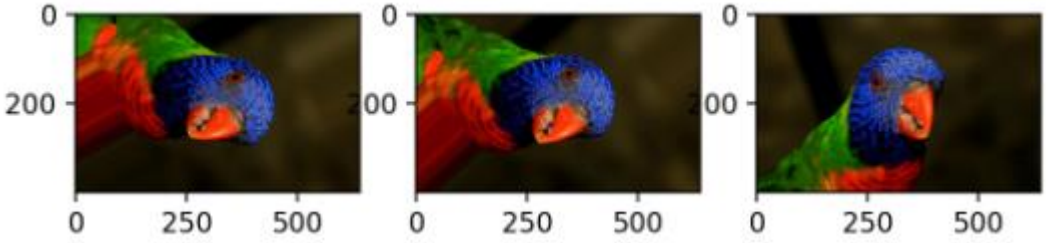
```
datagen = ImageDataGenerator(width_shift_range=[-200,200])
```



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)
```



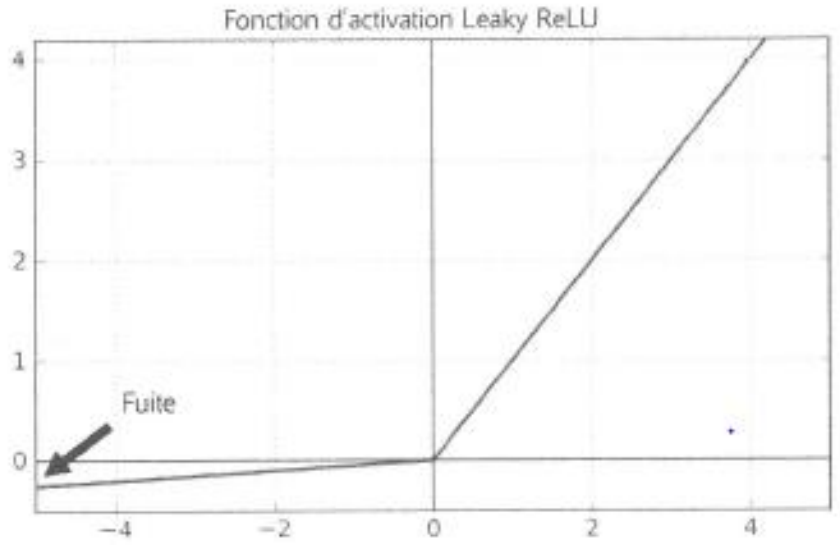
3. Problèmes de disparition et d'explosion des gradients

Fonctions d'activation

- Alors que l'algorithme progresse vers les couches inférieures, les gradients deviennent souvent de plus en plus petits : 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 : c'est le problème de disparition des gradients (*vanishing gradients*).

- Nous avons déjà vu que la fonction Tanh était meilleure que la sigmoïde vu que son gradient se rapproche de 0 beaucoup plus lentement.

- La fonction ReLu affiche elle aussi un meilleure comportement qu'elle ne sature pas pour les valeurs positives. Toutefois elle souffre d'un problème de mort des ReLu dans le cas où les entrées des neurones sont négatives
- Pour résoudre ce problème on peut employer la « leaky » ReLu.



Initialisation de Xavier (ou Glorot)

- Les initialisations **Xavier Glorot** (ou simplement Xavier) sont des méthodes populaires pour initialiser les poids dans les réseaux de neurones de manière à stabiliser la variance des activations et à faciliter l'apprentissage. Elles existent en deux versions : **Xavier Glorot Normal** et **Xavier Glorot Uniform**.

  - Xavier Glorot Normal :

    Les poids sont initialisés à partir d'une distribution **normale** (ou gaussienne) centrée sur 0, avec une variance de $\mathrm{Var}(W) = \frac{2}{\text{fan\_in}+\text{fan\_out}}$, où **fan_in** est le nombre de neurones en entrée de la couche et **fan_out** le nombre de neurones en sortie.

  - Xavier Glorot Uniform

    Les poids sont initialisés à partir d'une distribution **uniforme** sur l'intervalle $\left[-\sqrt{\frac{6}{\text{fan\_in}+\text{fan\_out}}}, \sqrt{\frac{6}{\text{fan\_in}+\text{fan\_out}}}\right]$.

Initialisation de Xavier : autres fonctions d'activation…

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)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{entrées}} + n_{\text{sorties}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{entrées}} + n_{\text{sorties}}}}$

```
model = Sequential([
    Dense(16, input_shape=(1,5), activation='relu'),
    Dense(32, activation='relu', kernel_initializer='glorot_uniform'),
    Dense(2, activation='softmax')
])
```

Batch normalisation

- La **batch normalization** est une technique utilisée dans les réseaux de neurones pour accélérer et stabiliser l'entraînement en normalisant les activations d'une couche.
- Elle a été introduite par Sergey Ioffe et Christian Szegedy (2015) dans le but de résoudre certains problèmes, notamment le problème de "covariate shift", où les distributions d'entrée changent d'une couche à l'autre au cours de l'entraînement :

  - **Normalisation des activations :** Pour chaque mini-lot d'entraînement, la batch normalization calcule la moyenne et l'écart-type des activations pour chaque dimension. Ensuite, chaque activation est normalisée.
  - **Apprentissage de paramètres de correction :** Après la normalisation, la batch normalization introduit deux nouveaux paramètres par dimension : un facteur d'échelle γ et un décalage β. Ces paramètres permettent au réseau d'apprendre la meilleure amplitude et position des activations

$$\hat{x} = \frac{x - \text{mean}}{\sqrt{\text{variance} + \epsilon}}$$

ϵ est une petite constante pour éviter la division par zéro.

$$y = \gamma \cdot \hat{x} + \beta$$

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.

Avantages de la Batch Normalisation

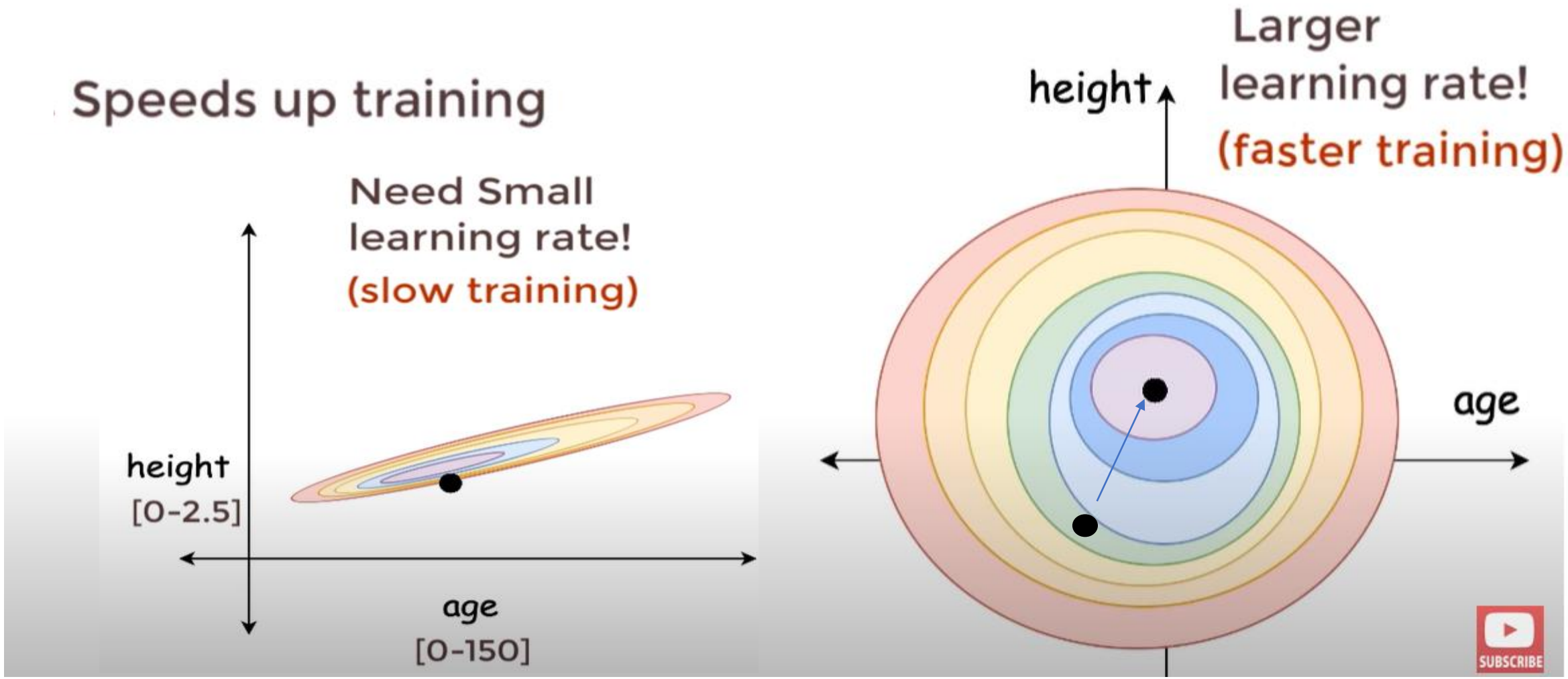
1. Accélération de l'entraînement
2. Réduction du besoin de réglage de paramètres
3. Réduction du risque de surapprentissage

Où l’appliquer ?

La BN peut être appliquée après les couches entièrement connectées ou les couches de convolution, et elle est souvent insérée avant l'application de la fonction d'activation (comme ReLU ou sigmoid).

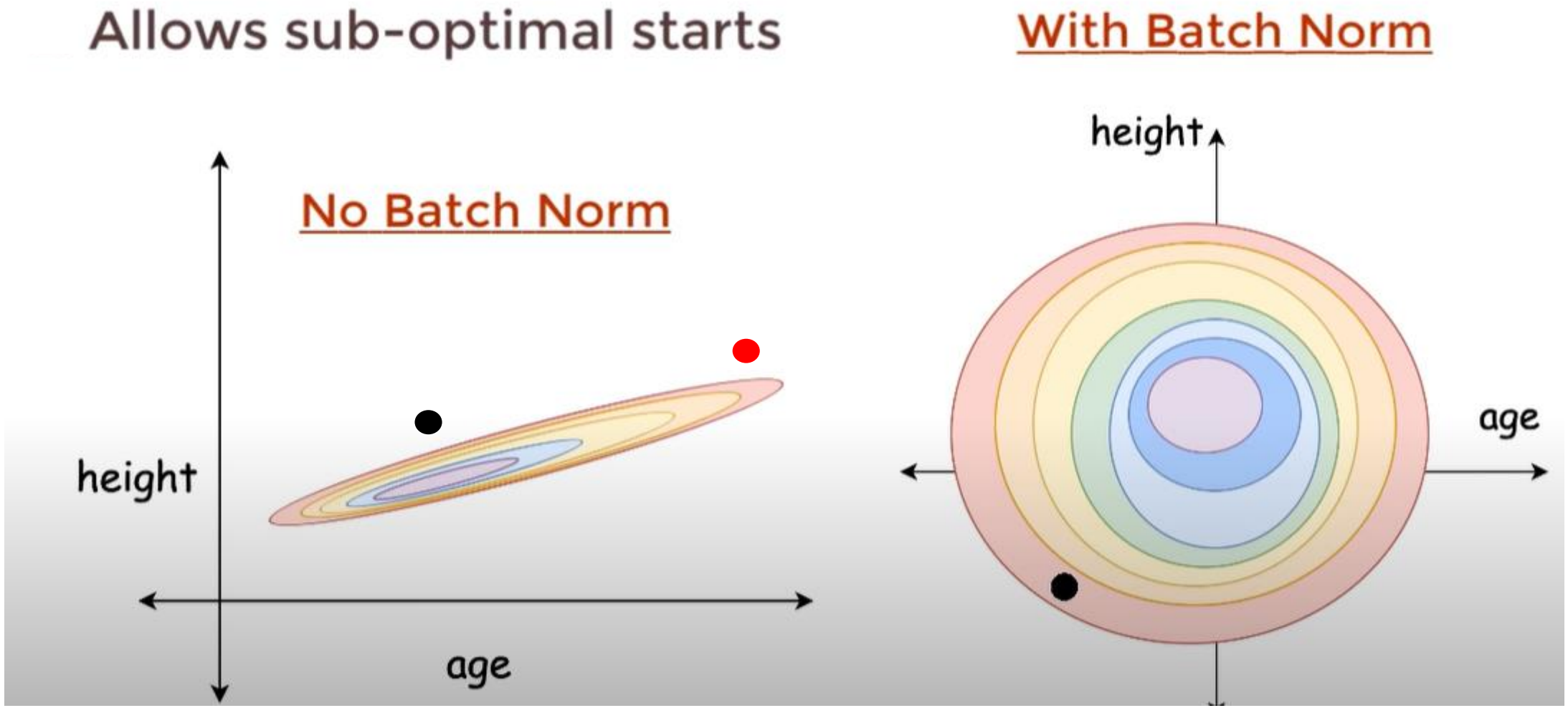
Avantage 1 : Accélération de l'entraînement

En stabilisant les distributions d'activation, la BN permet d'utiliser des taux d'apprentissage plus élevés, ce qui accélère l'apprentissage.



Avantage 2 : Réduction du besoin de réglage de paramètres

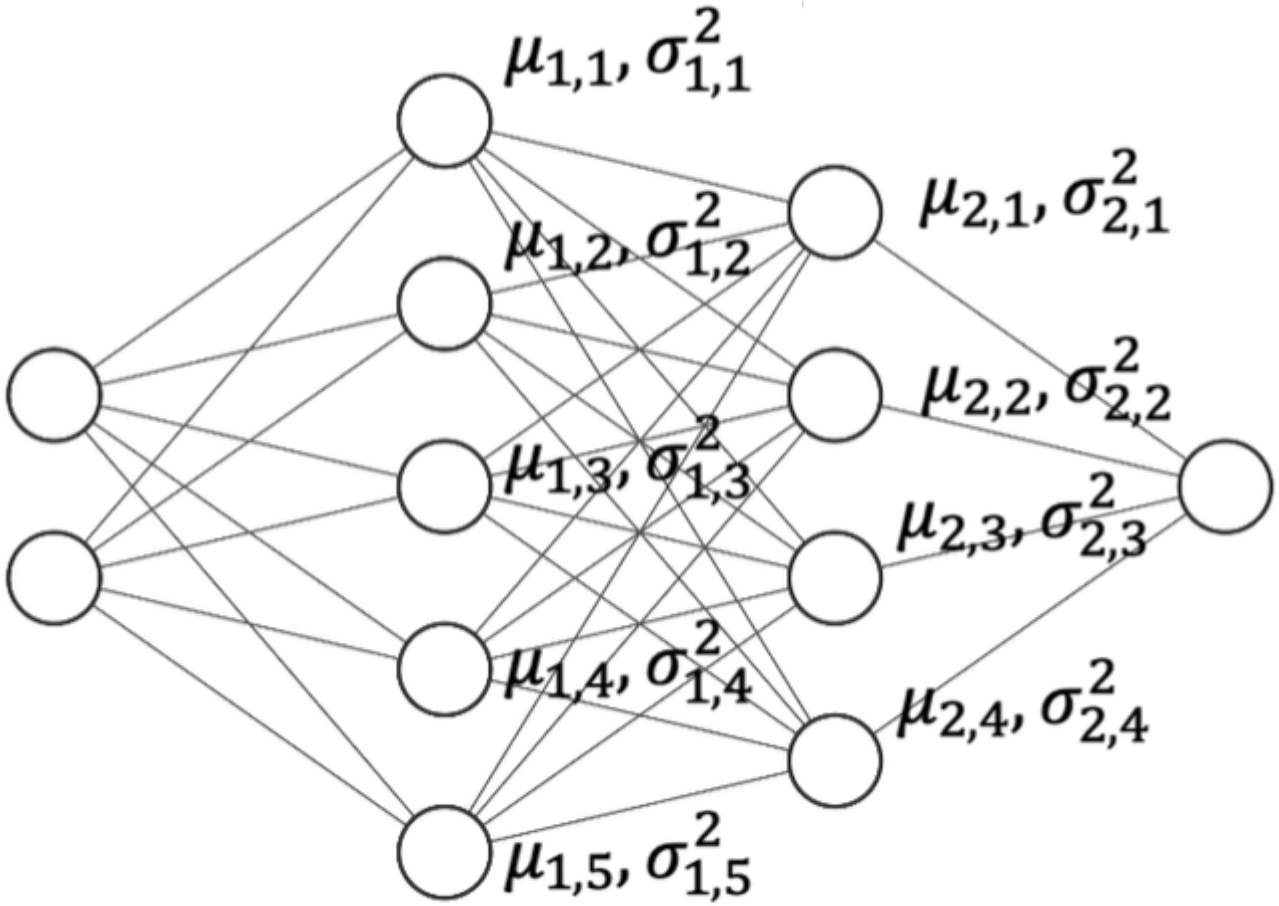
En rendant l'entraînement plus stable, la BN diminue la sensibilité aux valeurs des hyperparamètres, comme le taux d'apprentissage ou les poids initiaux. Cela permet d'éviter des configurations sous-optimales et réduit le temps consacré à l'ajustement de ces hyperparamètres.



Avantage 3 : Réduction du risque de surapprentissage

En ajoutant un bruit au processus d'entraînement, la BN peut agir comme une forme de régularisation, réduisant le besoin de techniques de régularisation supplémentaires comme le dropout.

Acts as a Regularizer (a little) → "randomness"



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')
]);
```