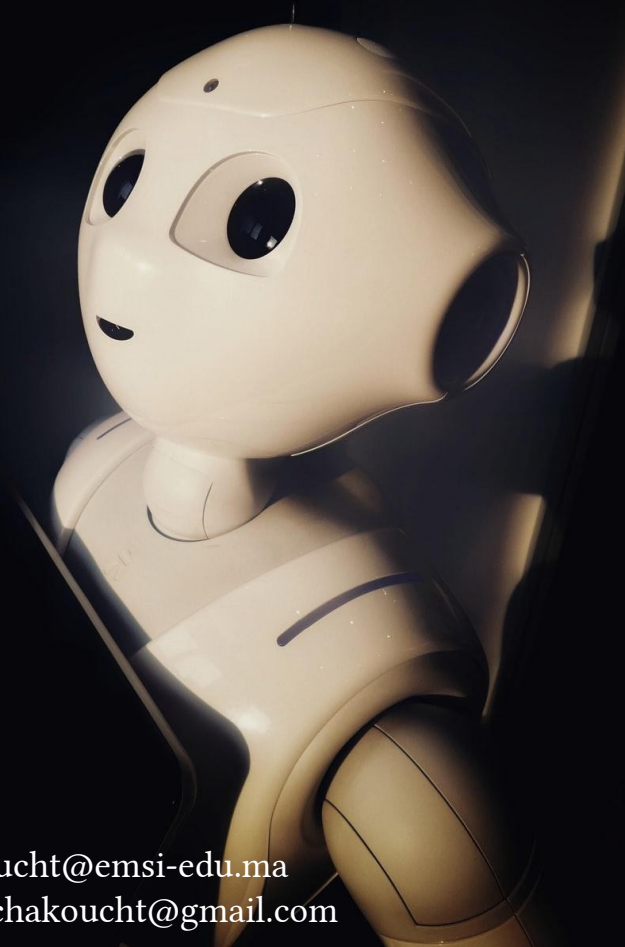


Deep Learning: 102



Aissam Outchakoucht

a.outchakoucht@emsi-edu.ma
aissam.outchakoucht@gmail.com



Recap

L'apprentissage profond est très pratique quand il s'agit de

Problèmes complexes

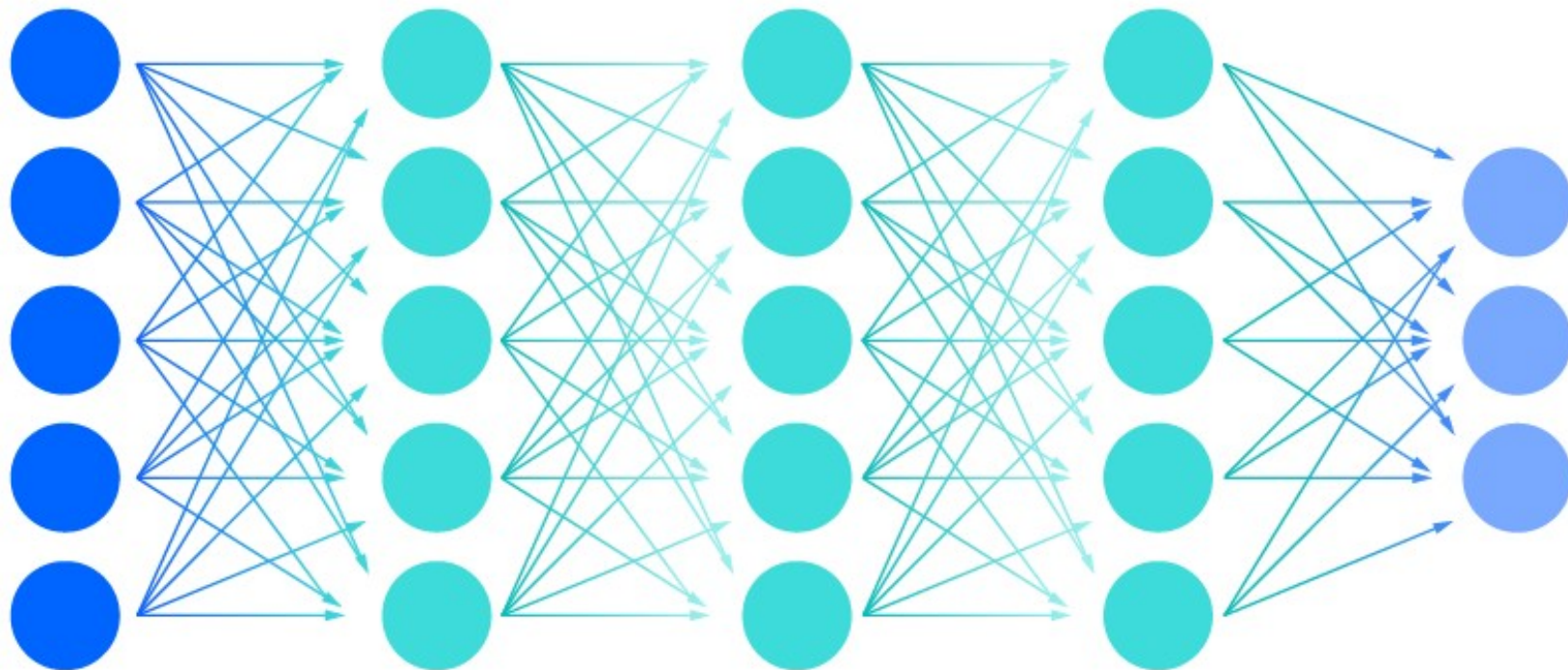
tels que la classification des images, le traitement du langage naturel et la reconnaissance vocale.

Recap

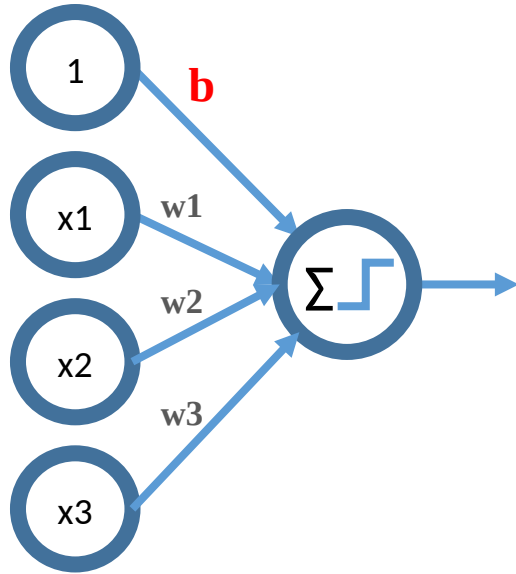
Input layer

Multiple hidden layer

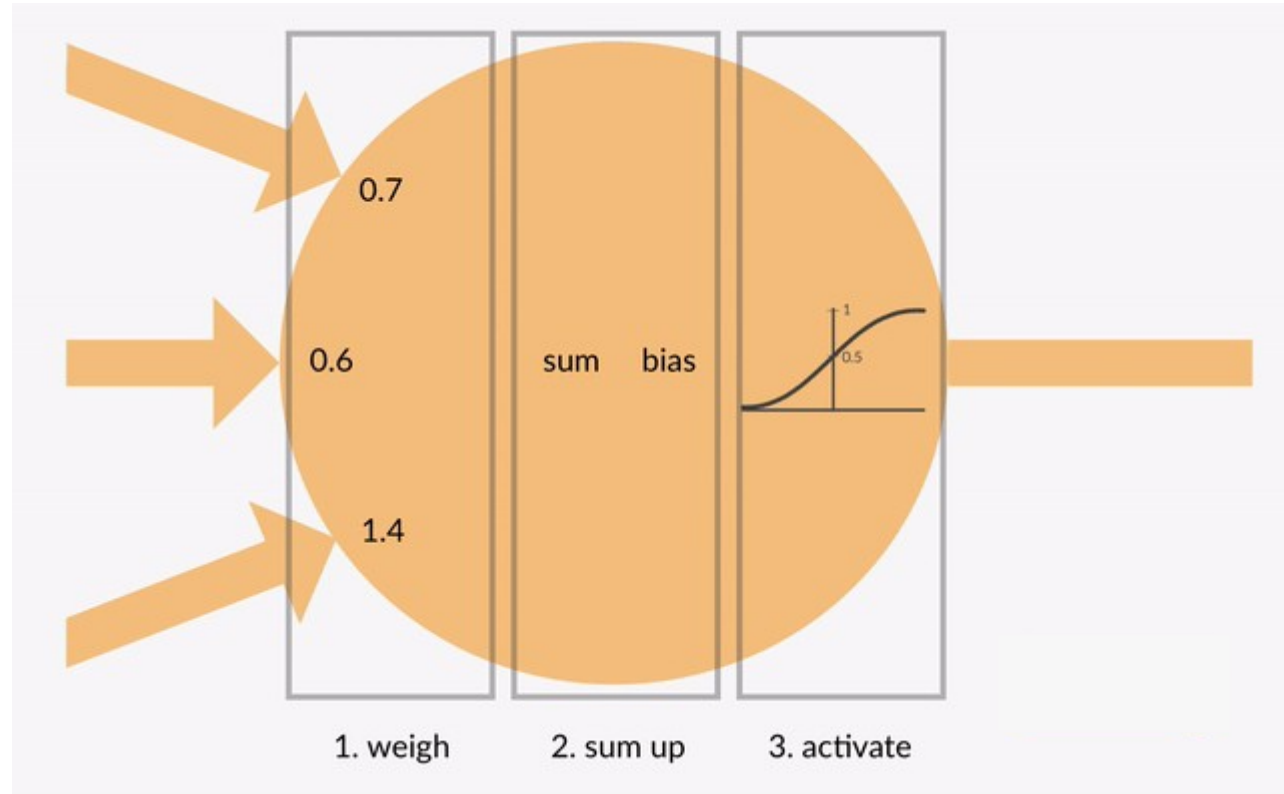
Output layer



Recap



$$F(x) = \sigma(\textcolor{green}{W} . X + \textcolor{red}{b})$$



Recap.py

```
▶ # Chargement des données  
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Label: 5



Label: 0



Label: 4



Label: 1



Label: 9



Label: 2



Label: 1



Label: 3



Label: 1



Label: 4



Label: 3



Label: 5



Label: 3



Label: 6



Label: 1



Label: 7



Label: 2



Label: 8



Label: 6



Label: 9



Recap.py

```
model = Sequential([  
    Flatten(input_shape=(28,28)),      # Aplatit les images 28x28 en vecteurs de 784 éléments  
    Dense(128, activation='relu'),      # Couche cachée avec 128 neurones et activation ReLU  
    Dense(10, activation='softmax')     # Couche de sortie avec 10 neurones (une pour chaque classe)  
])
```

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

```
# Entraînement du Modèle  
history = model.fit(X_train, y_train, epochs=10, validation_split=0.1)
```

```
Epoch 1/10  
1688/1688 ————— 6s 3ms/step - accuracy: 0.8681 - loss: 0.4688 - val_accuracy: 0.9622 - val_loss: 0.1281  
Epoch 2/10  
1688/1688 ————— 7s 4ms/step - accuracy: 0.9623 - loss: 0.1299 - val_accuracy: 0.9730 - val_loss: 0.0940  
Epoch 3/10  
1688/1688 ————— 5s 3ms/step - accuracy: 0.9747 - loss: 0.0850 - val_accuracy: 0.9730 - val_loss: 0.0933  
Epoch 4/10  
1688/1688 ————— 12s 4ms/step - accuracy: 0.9804 - loss: 0.0635 - val_accuracy: 0.9762 - val_loss: 0.0811  
Epoch 5/10  
1688/1688 ————— 5s 3ms/step - accuracy: 0.9862 - loss: 0.0457 - val_accuracy: 0.9777 - val_loss: 0.0760  
Epoch 6/10  
1688/1688 ————— 10s 3ms/step - accuracy: 0.9900 - loss: 0.0351 - val_accuracy: 0.9745 - val_loss: 0.0838  
Epoch 7/10  
1688/1688 ————— 10s 3ms/step - accuracy: 0.9916 - loss: 0.0288 - val_accuracy: 0.9757 - val_loss: 0.0860  
Epoch 8/10  
1688/1688 ————— 6s 3ms/step - accuracy: 0.9937 - loss: 0.0215 - val_accuracy: 0.9807 - val_loss: 0.0709
```

Recap.py

```
# Prédiction  
y_pred = model.predict(X_test)
```

Vrai: 7
Prédit: 7



Vrai: 1
Prédit: 1



Vrai: 0
Prédit: 0



Vrai: 5
Prédit: 5



Vrai: 2
Prédit: 2



Vrai: 4
Prédit: 4



Vrai: 6
Prédit: 6



Vrai: 9
Prédit: 9



Vrai: 1
Prédit: 1



Vrai: 9
Prédit: 9



Vrai: 9
Prédit: 9



Vrai: 7
Prédit: 7



Vrai: 0
Prédit: 0



Vrai: 5
Prédit: 6



Vrai: 0
Prédit: 0



Vrai: 3
Prédit: 3



Vrai: 4
Prédit: 4



Vrai: 9
Prédit: 9



Vrai: 1
Prédit: 1



Vrai: 4
Prédit: 4



Recap.py

```
▶ # Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=3)
```

```
# Modèle avec Early Stopping
```

```
model_early_stop = Sequential([
    Flatten(input_shape=(28,28)),
    Dense(512, activation='relu'),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
model_early_stop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
# Entraînement du modèle
```

```
history_early_stop = model_early_stop.fit(X_train, y_train, epochs=50, validation_split=0.1, callbacks=[early_stop])
```



Epoch 1/50

1688/1688 ————— 18s 10ms/step - accuracy: 0.8832 - loss: 0.3751 - val_accuracy: 0.9687 - val_loss: 0.1075

Epoch 2/50

1688/1688 ————— 20s 10ms/step - accuracy: 0.9700 - loss: 0.0995 - val_accuracy: 0.9748 - val_loss: 0.0890

Epoch 3/50

1688/1688 ————— 16s 10ms/step - accuracy: 0.9794 - loss: 0.0666 - val_accuracy: 0.9800 - val_loss: 0.0687

Epoch 4/50

1688/1688 ————— 16s 10ms/step - accuracy: 0.9836 - loss: 0.0523 - val_accuracy: 0.9762 - val_loss: 0.0857

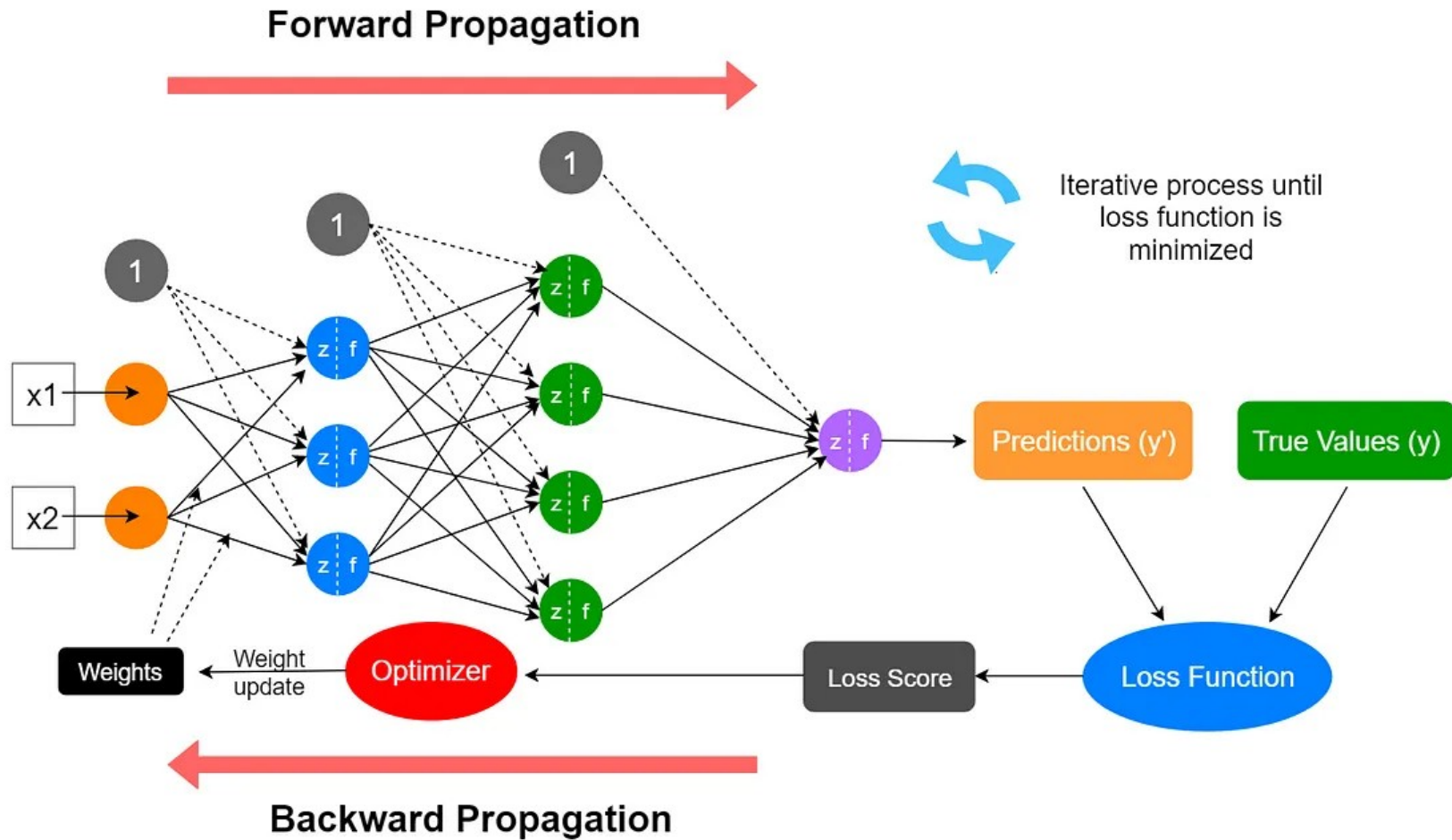
Epoch 5/50

1688/1688 ————— 16s 10ms/step - accuracy: 0.9877 - loss: 0.0419 - val_accuracy: 0.9788 - val_loss: 0.0776

Epoch 6/50

1688/1688 ————— 17s 10ms/step - accuracy: 0.9889 - loss: 0.0364 - val_accuracy: 0.9818 - val_loss: 0.0699

Recap.ppt



Recap

Hypothèse :

$$f(x) = a \cdot x + b$$

Paramètres :

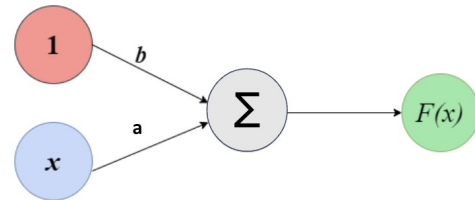
$$a, b$$

Fonction Coût :

$$J(a, b) = \frac{1}{2m} \sum_{i=1}^m \left(f(x^{(i)}) - y^{(i)} \right)^2$$

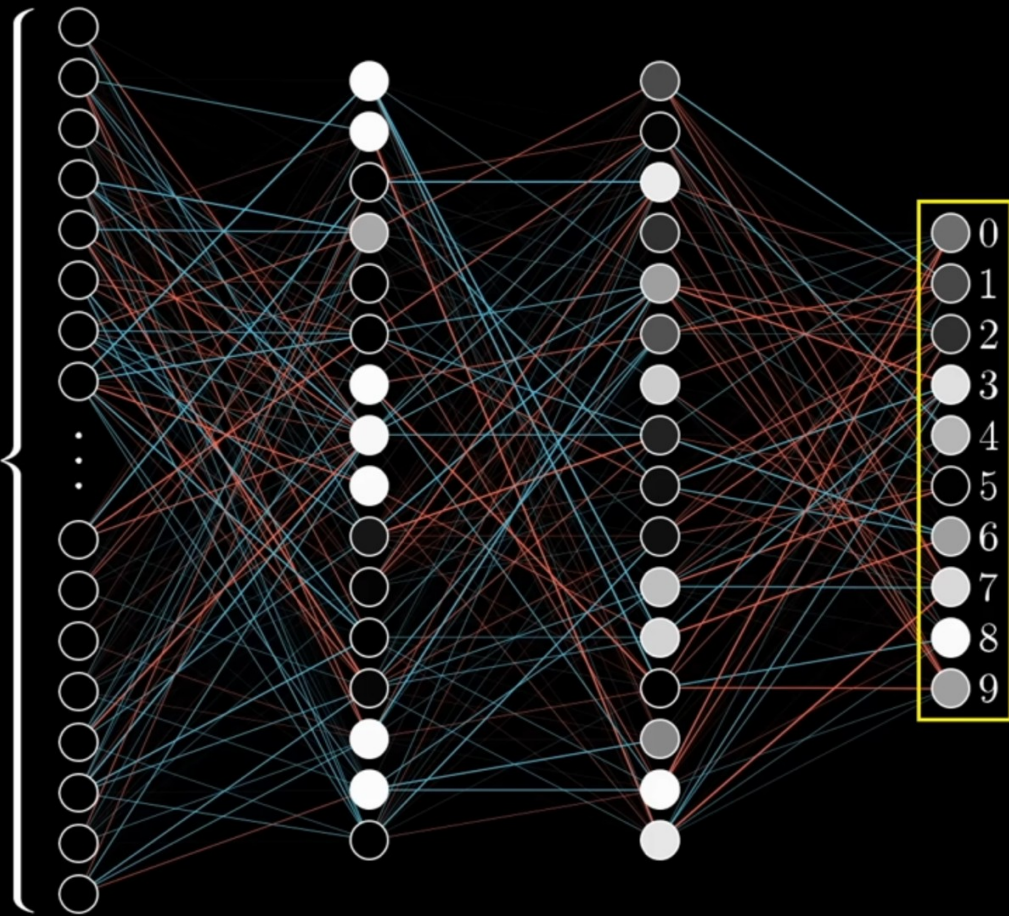
Objectif :

$$\min_{a, b} J(a, b)$$





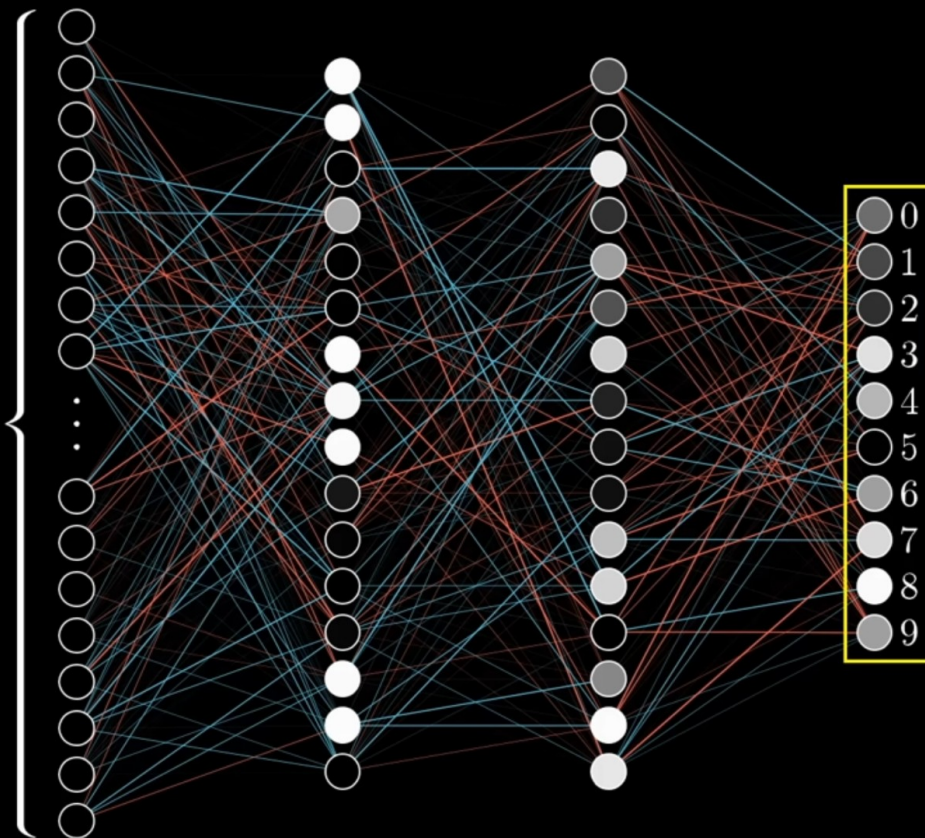
784



- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9



784



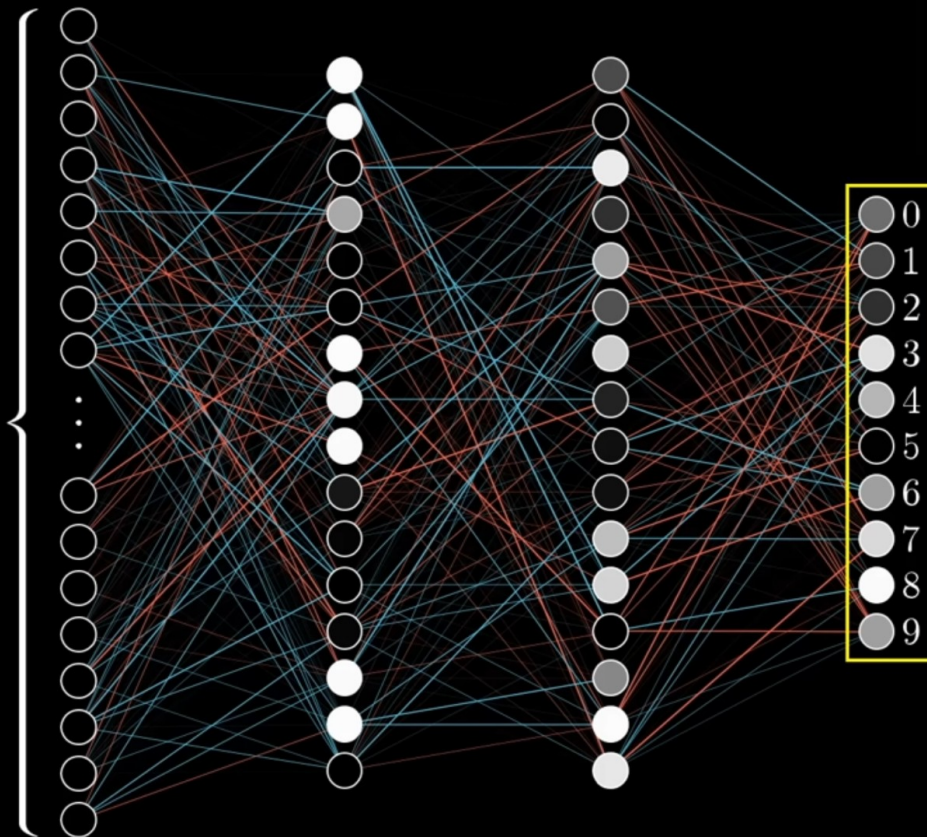
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

$$\left\{ \begin{array}{l} (0.43 - 0.00)^2 + \\ (0.28 - 0.00)^2 + \\ (0.19 - 0.00)^2 + \\ (0.88 - 1.00)^2 + \\ (0.72 - 0.00)^2 + \\ (0.01 - 0.00)^2 + \\ (0.64 - 0.00)^2 + \\ (0.86 - 0.00)^2 + \\ (0.99 - 0.00)^2 + \\ (0.63 - 0.00)^2 \end{array} \right.$$

Cost function



784

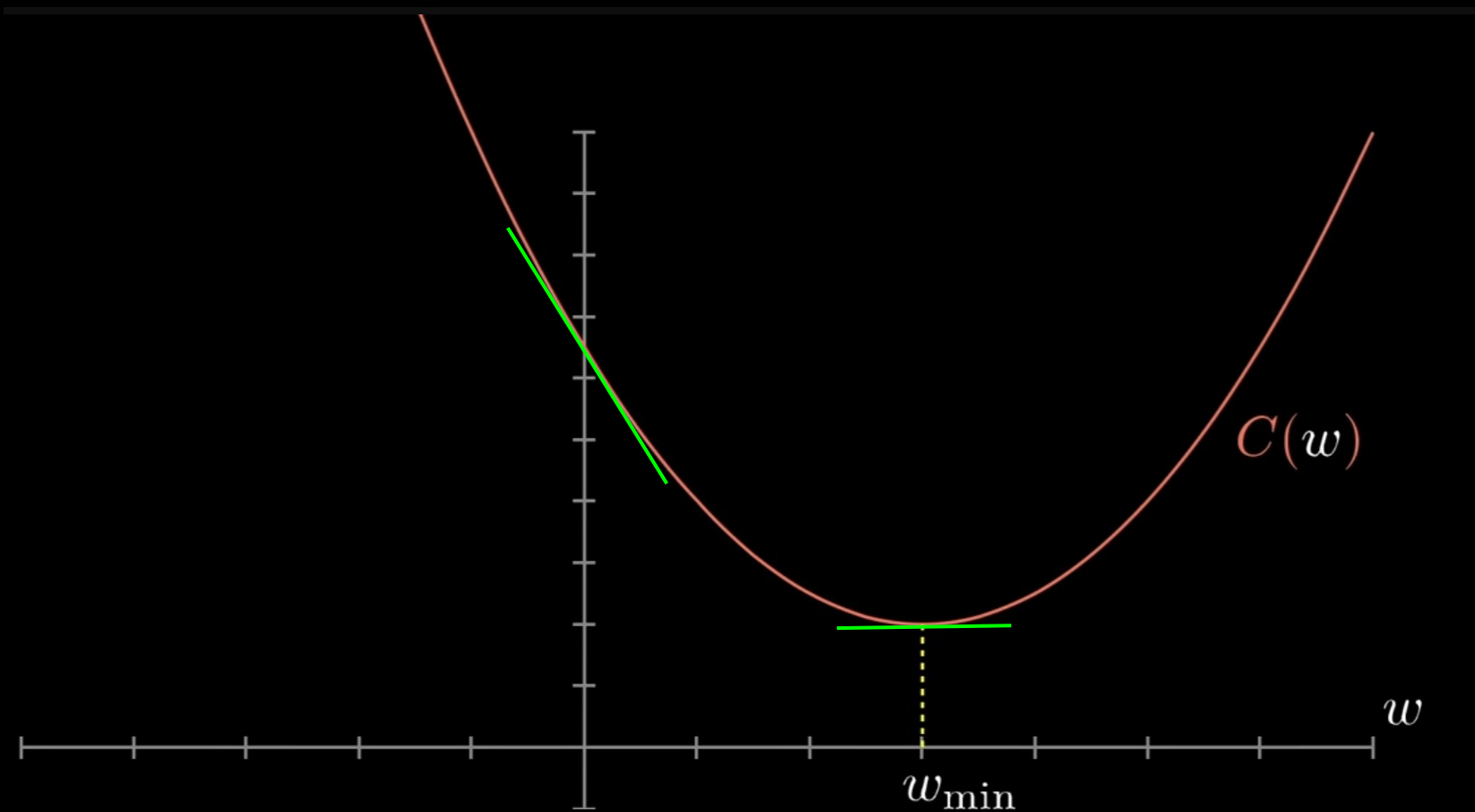


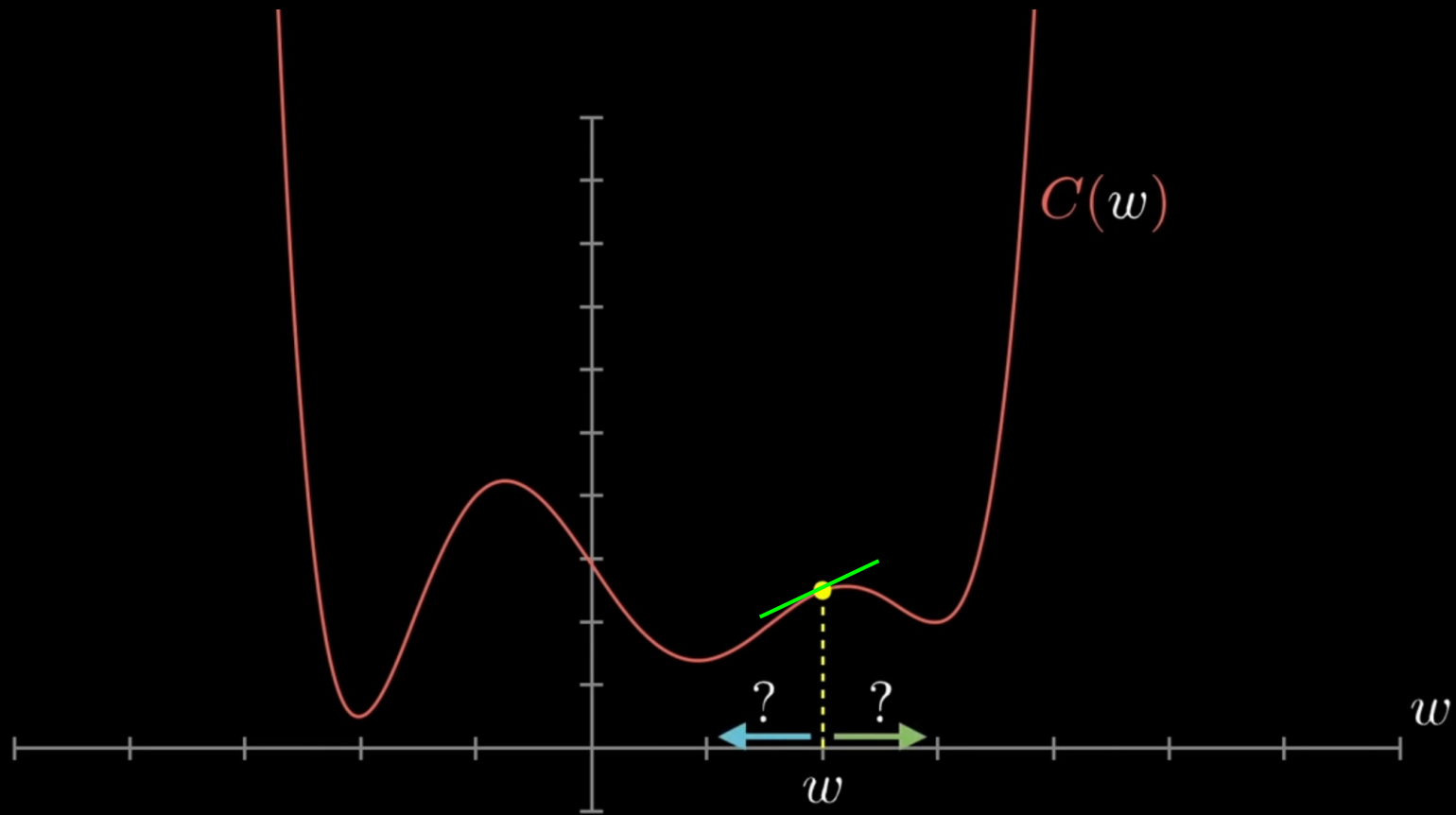
Input: 13,002 weights/biases

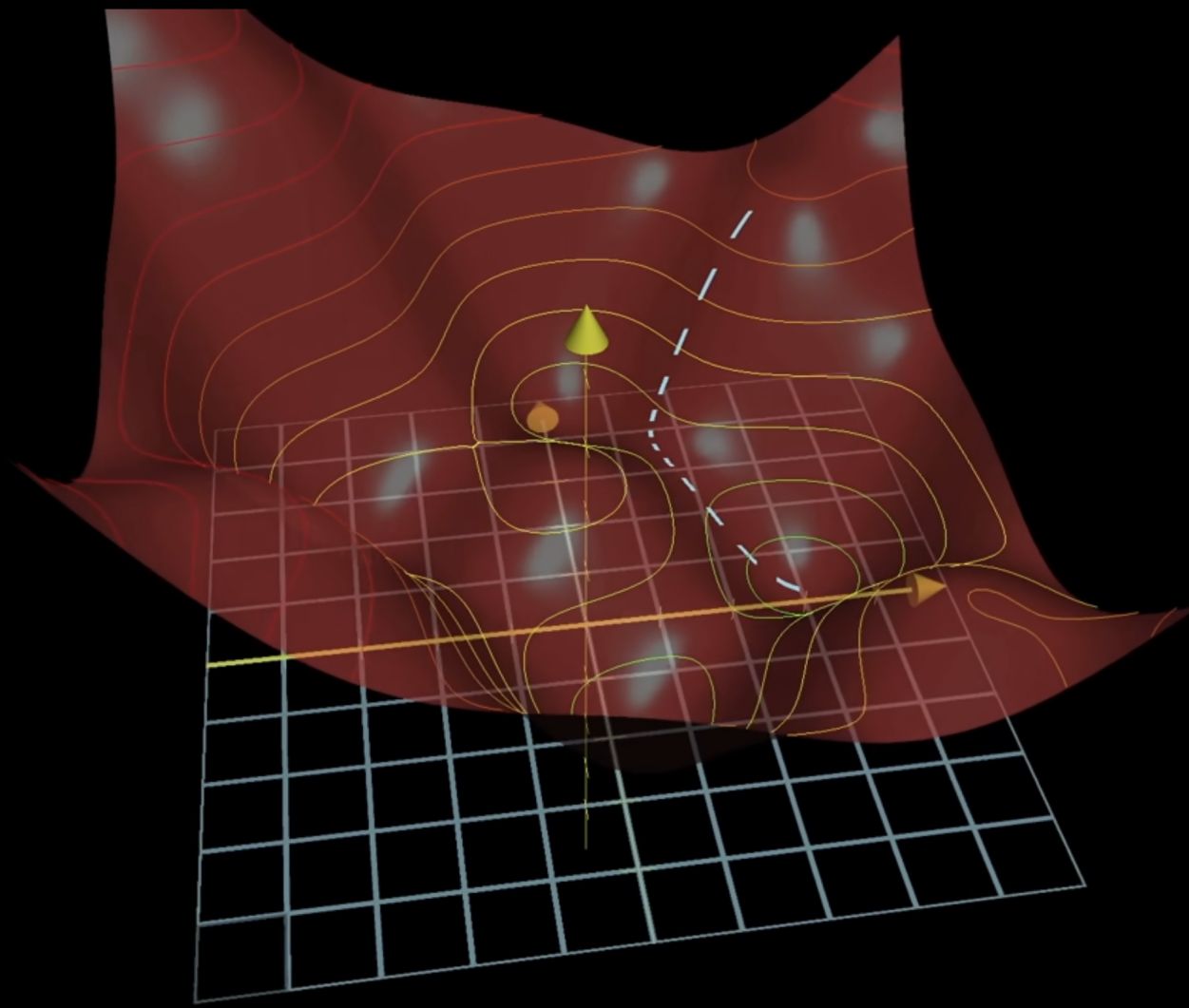
Output: 1 number (the cost)



$$J(a, b) = \frac{1}{2m} \sum_{i=1}^m \left(f(x^{(i)}) - y^{(i)} \right)^2$$







(batch) GD

La rétropropagation est l'algorithme permettant de déterminer comment un seul exemple d'entraînement devrait pousser les poids pour provoquer la diminution la plus rapide de la fonction coût.

Une véritable étape de GD consisterait à faire cela pour **tous les exemples d'entraînement** et à faire la moyenne des changements souhaités que vous obtenez, à chaque étape de la Descente de Gradient ! C'est pourquoi l'algorithme est appelé Batch Gradient Descent : il utilise le lot entier de données d'entraînement à chaque étape. En conséquence, il est terriblement **lent** sur les très grands ensembles d'entraînement.

stochastic GD

À l'extrême opposé, la méthode Stochastic Gradient Descent choisit **une instance aléatoire** dans l'ensemble d'entraînement **à chaque étape** et calcule les gradients en se basant uniquement sur cette instance unique.

Il est évident que le fait de travailler avec une seule instance à la fois rend l'algorithme beaucoup plus **rapide**, car il ne dispose que de très peu de données à manipuler à chaque itération.

En raison de sa nature stochastique (c'est-à-dire aléatoire), cet algorithme est beaucoup moins régulier que batch GD : au lieu de diminuer doucement jusqu'à atteindre le minimum, la fonction de coût rebondit de haut en bas, ne diminuant qu'en moyenne.

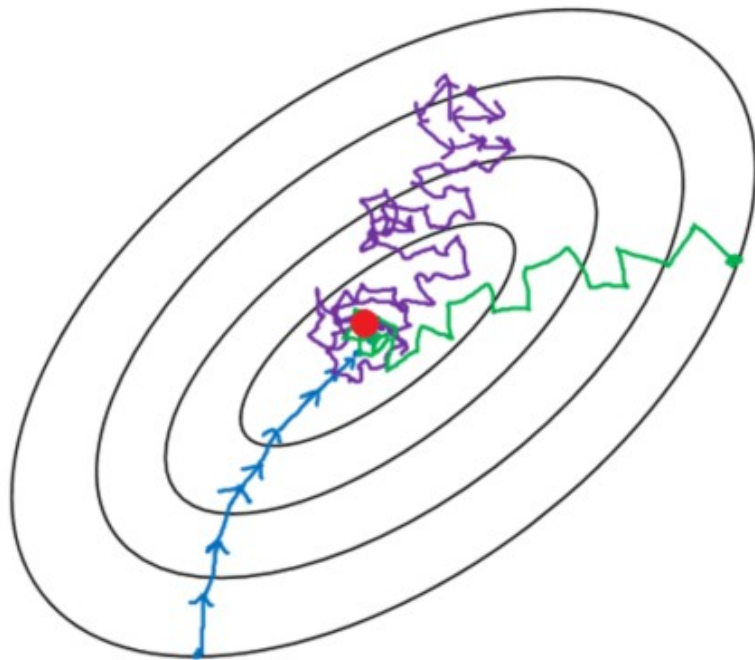
Mini batch GD

A chaque étape, au lieu de calculer les gradients sur la base de l'ensemble complet d'entraînement (comme dans Batch GD) ou sur la base d'une seule instance (comme dans Stochastic GD), **Mini-batch GD calcule les gradients sur de petits ensembles aléatoires d'instances appelés mini-batch.**

Le principal avantage de Minibatch GD par rapport à Stochastic GD est qu'on peut obtenir une augmentation des performances grâce à l'optimisation matérielle des opérations matricielles via des GPU.

Minibatch GD finira par se rapprocher un peu plus du minimum que GD stochastique - mais il peut être plus difficile pour lui d'échapper aux minima locaux

GD

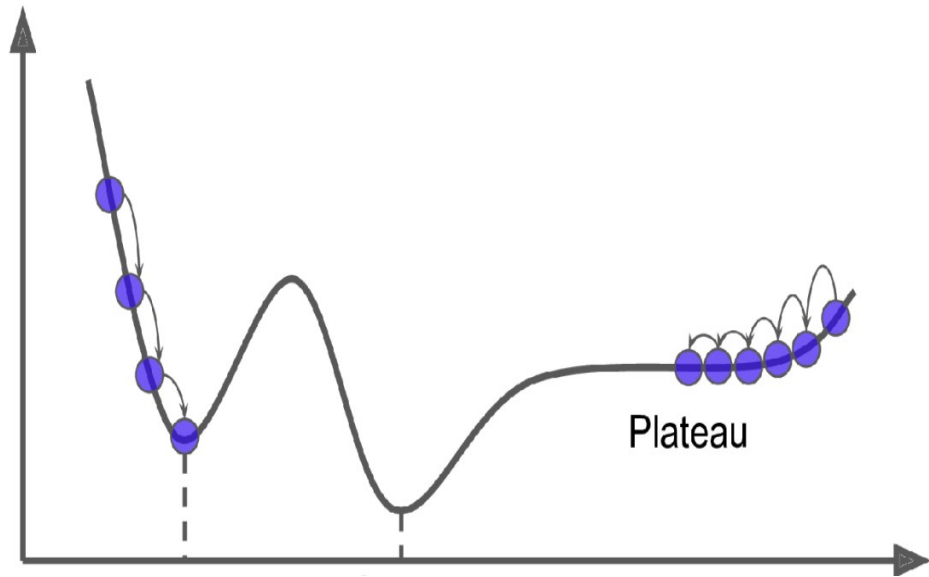


- Batch gradient descent
- Mini-batch gradient Descent
- Stochastic gradient descent

GD

Les fonctions de coût ne ressemblent pas tous à de beaux bols ordinaires (convexes).

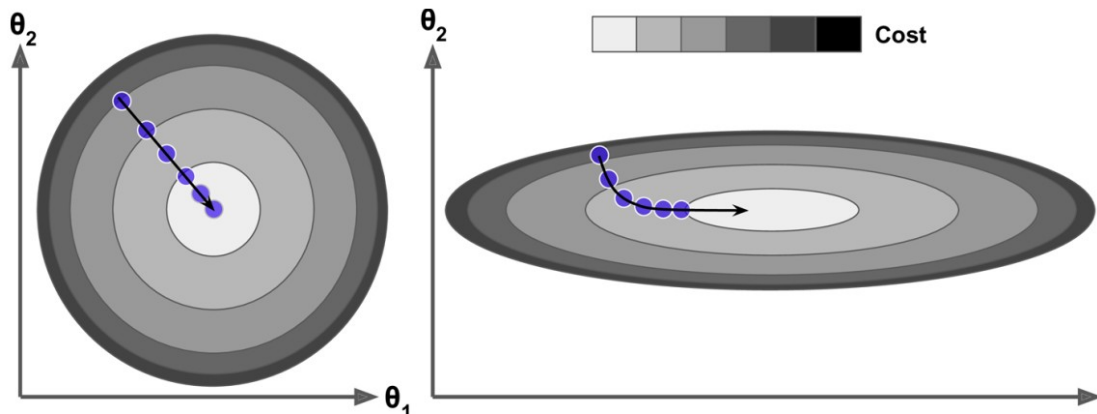
Si l'initialisation aléatoire démarre l'algorithme à gauche, alors il convergera vers un minimum local, qui n'est pas aussi bon que le minimum global. Si elle démarre à droite, alors il faudra beaucoup de temps pour traverser le plateau. Et si on s'arrête trop tôt, on n'atteindra jamais le minimum global.



Features scaling

Avant d'utiliser GD, il faut s'assurer que toutes les caractéristiques (features) ont une échelle similaire (par exemple, âge entre 0-70 et distance entre 1 et 100 000 km), sinon la convergence prendra beaucoup plus de temps.

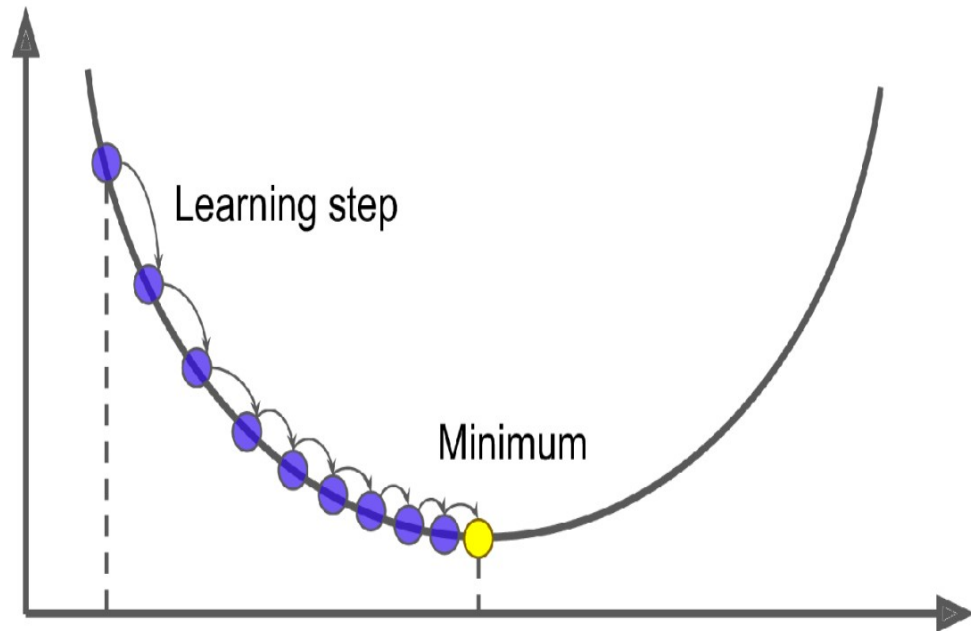
En fait, la fonction de coût a la forme d'un bol, mais il peut s'agir d'un bol allongé si les caractéristiques ont des échelles très différentes.



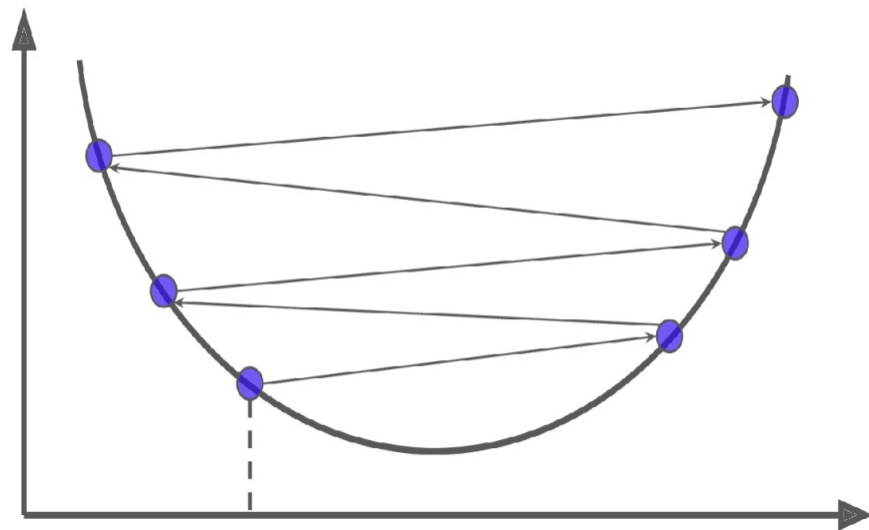
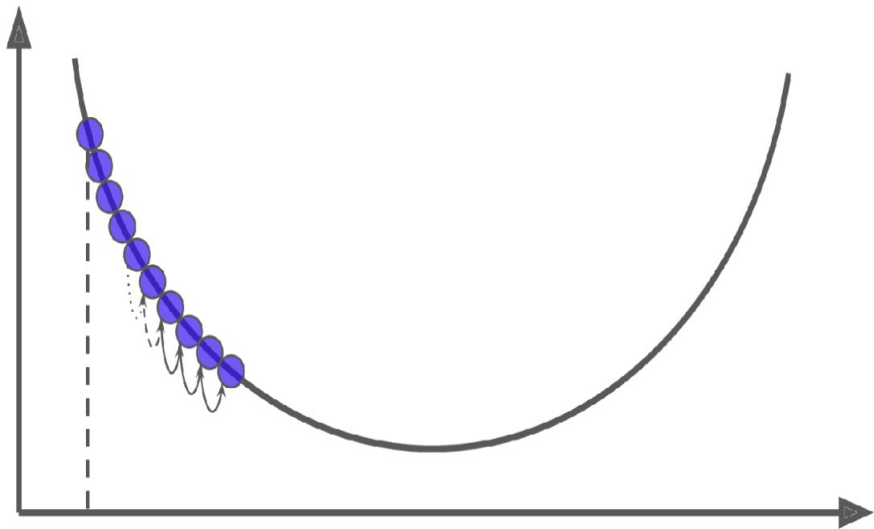
Learning rate

Concrètement, on commence l'entraînement d'un NN par remplir W avec des valeurs aléatoires (c'est ce qu'on appelle l'initialisation aléatoire).

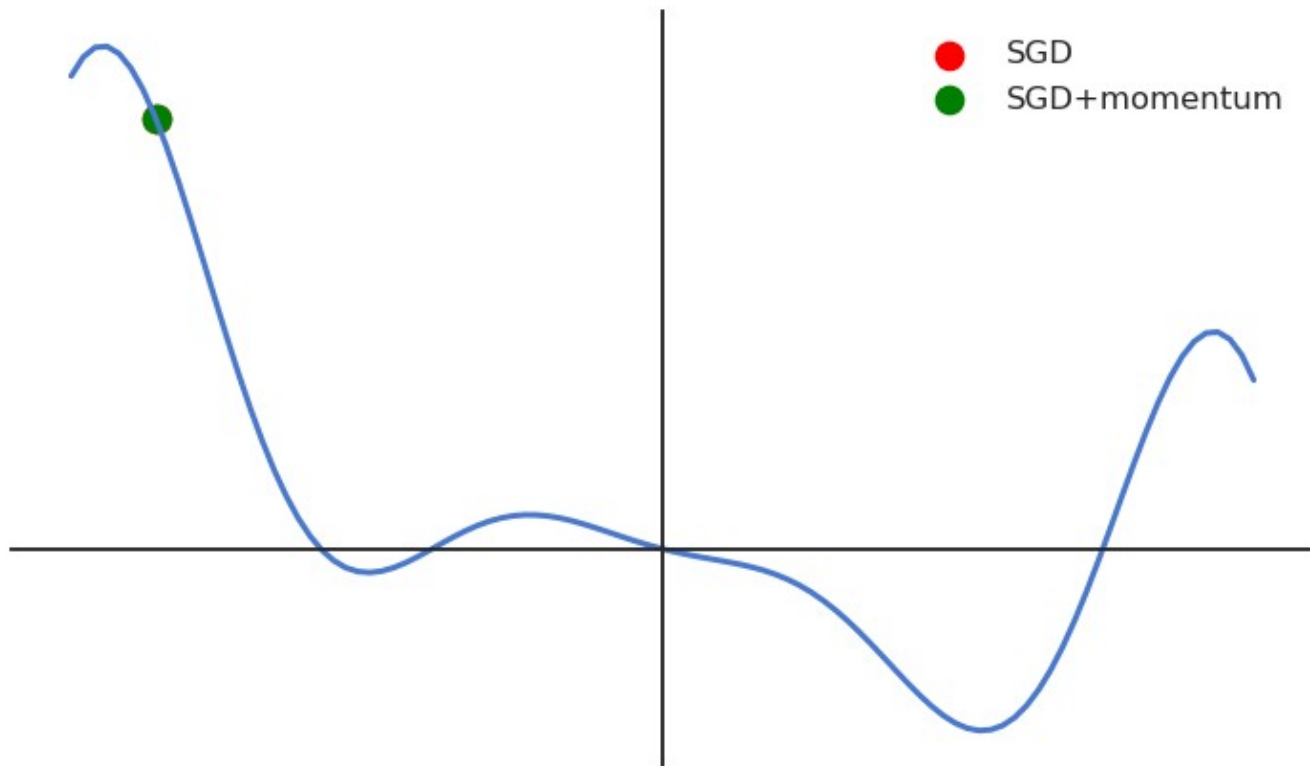
Ensuite, nous l'améliorerons progressivement, en faisant un pas à la fois, chaque pas tentant de diminuer la fonction de coût, jusqu'à ce que l'algorithme converge vers un minimum.



Learning rate

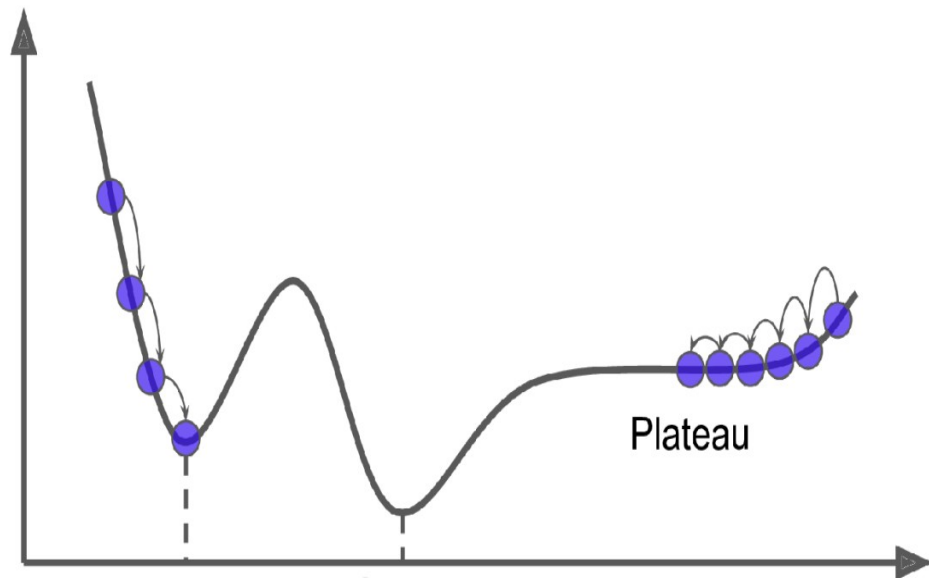


GD + Momentum



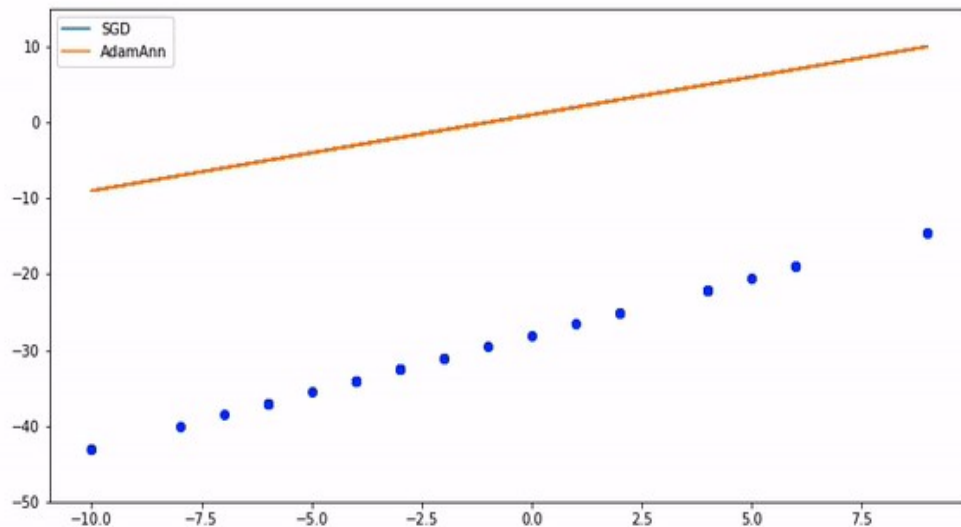
La descente de gradient avec momentum utilise la mémoire des gradients antérieurs pour lisser et accélérer la convergence.

RMSProp



RMSProp adapte la taille des mises à jour en fonction de l'ampleur des gradients récents, ce qui permet d'éviter de trop dépasser la cible lorsque ces gradients sont élevés, et de ne pas s'immobiliser lorsqu'ils sont faibles.

Adam



Adam est une méthode d'optimisation qui combine les avantages du momentum et de RMSProp. Elle conserve une moyenne mobile des gradients (comme le momentum) pour assurer la stabilité et l'orientation, et enregistre également une moyenne mobile des gradients au carré (comme RMSProp) afin d'adapter le taux d'apprentissage à l'échelle des paramètres. Cette approche duale aboutit souvent à une convergence plus rapide et plus fiable que l'utilisation d'une seule de ces méthodes.