Réseaux de Neurones & Deep Learning – 2/2

Master parcours SSD - UE Apprentissage Statistique II

Pierre Mahé - bioMérieux & Université de Grenoble-Alpes

La semaine dernière :

Architectures

- neurones artificiels et modèles linéaires
- perceptrons multi-couches
- réseaux convolutifs

⇒ **Aujourd'hui : apprentissage en pratique**

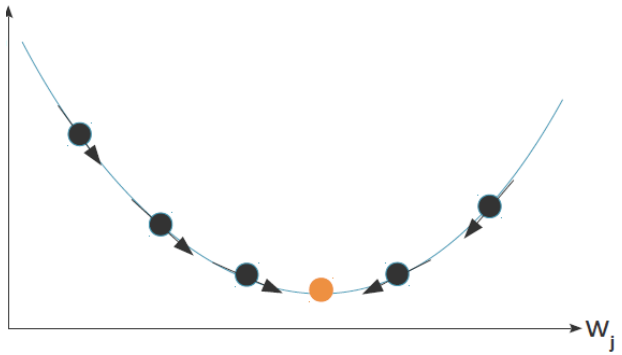
Apprentissage

- Descente de gradient & rétro-propagation
- Régularisation
- Augmentation de données
- Transfer learning
- En pratique

Apprentissage & réseaux de neurones

Apprentissage & RN : méthodes de descente de gradient stochastique (accélérées)

Descente de gradient pour minimiser $J(\mathbf{w}; \mathbf{X}, \mathbf{y})$:



$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \sum_{i=1}^{n} L\big(y_i, f(x_i)\big)$$

- $\mathbf{w}$ = paramètres du réseau
- $L(y, f(x))$ = fonction de perte

$\Rightarrow$ schéma itératif : $\boxed{w_j^{(t+1)} = w_j^{(t)} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})[j]}$

$\Rightarrow$ gradient : $\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \sum_{i=1}^{n} \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$.

Descente de gradient stochastique

Descente de gradient (classique) :

- gradient calculé sur tout le jeu de données
- $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \sum_{i=1}^{n} \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$

$\Rightarrow$ trop long si jeu de données conséquent

Descente de gradient stochastique

Descente de gradient (classique) :

- gradient calculé sur tout le jeu de données
- $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \sum_{i=1}^{n} \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$

$\Rightarrow$ trop long si jeu de données conséquent

Descente de gradient stochastique :

- gradient calculé sur un couple (x_i, y_i) pris au hasard
- $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$

Descente de gradient stochastique

Descente de gradient (classique) :

- gradient calculé sur tout le jeu de données
- $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \sum_{i=1}^{n} \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$

$\Rightarrow$ trop long si jeu de données conséquent

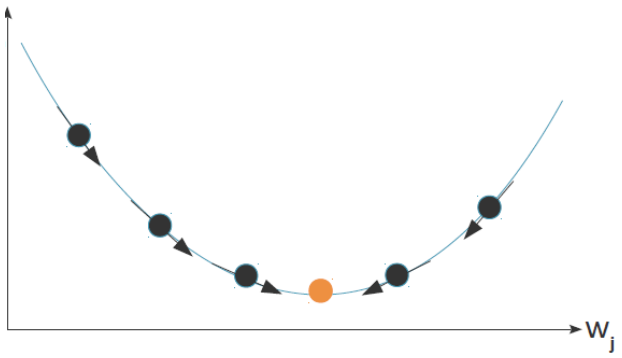
Descente de gradient stochastique :

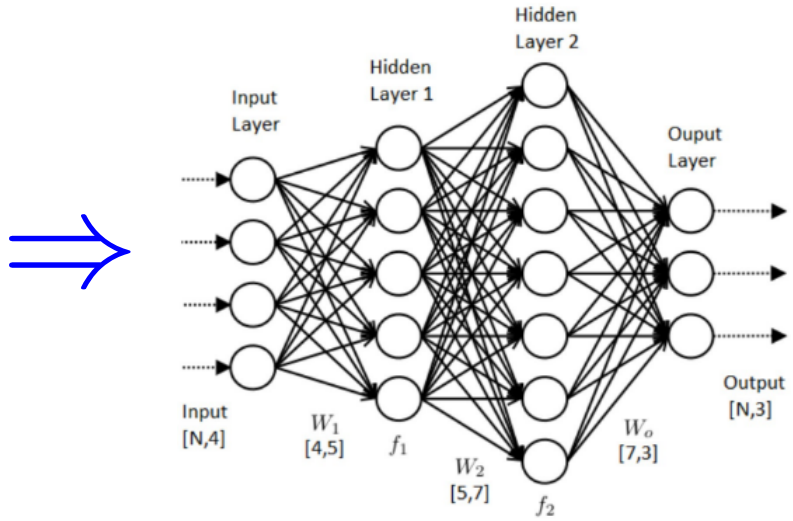
- gradient calculé sur un couple (x_i, y_i) pris au hasard
- $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$

$\Rightarrow$ en pratique : approche par (mini) batch

- gradient calculé au niveau du batch

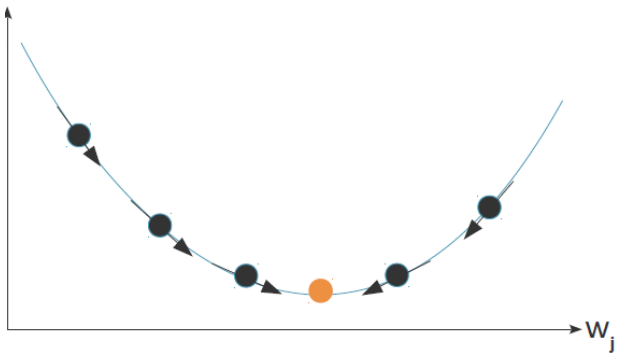
Descente de gradient et réseaux de neurones



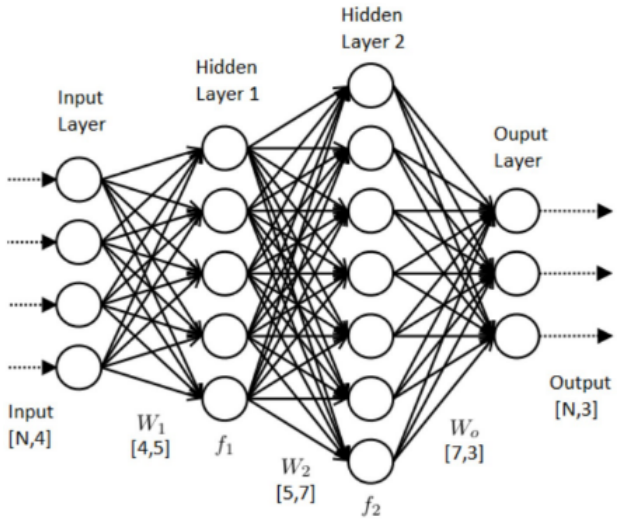


?

Descente de gradient et réseaux de neurones



$\Rightarrow$



?

$\Rightarrow$ de multiples compositions : $f(x) = h_4\Big(h_3\Big(h_2\big(h_1(x)\big)\Big)\Big)$

$\Rightarrow$ comment calculer le gradient ?

Ici :

- $h_4(z) \sim \text{softmax}(z)$
- $h_i(z) \sim \sigma(\langle w_i, z\rangle + b_i)$ pour $i = 1, ..., 3$.

Algorithme de rétro-propagation

Solution : basée sur le théorème de dérivation des fonctions composées (chain rule of calculus) :

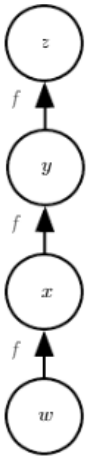
$$\text{si } y = g(x) \text{ et } z = f(y) = f\big(g(x)\big), \text{ alors } \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Algorithme de rétro-propagation

Solution : basée sur le théorème de dérivation des fonctions composées (chain rule of calculus) :

$$\text{si } y = g(x) \text{ et } z = f(y) = f\big(g(x)\big), \text{ alors } \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Illustration (tirée de Goodfellow et al. (2016)) :



$$x = f(w),\ y = f(x),\ z = f(y)$$

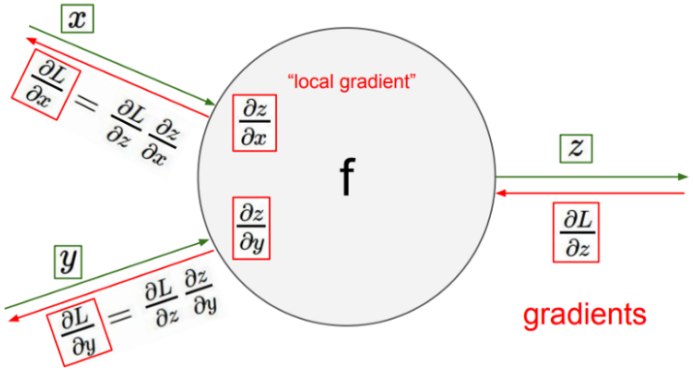
$$z = f\big(f(x)\big) \Rightarrow \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

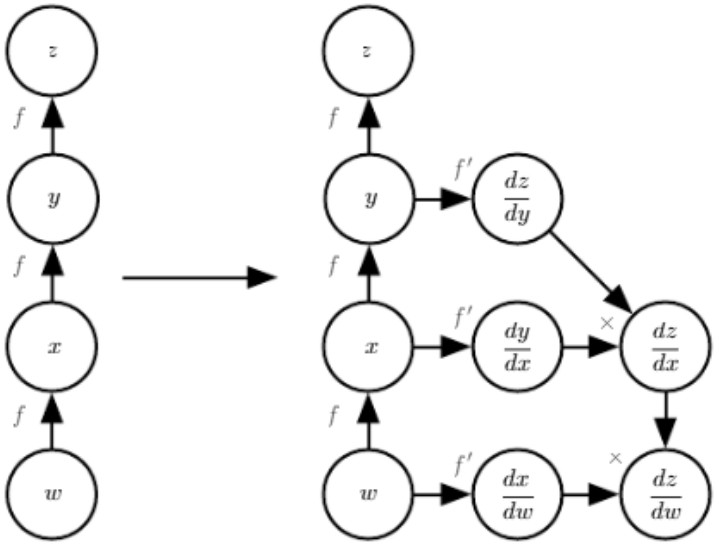
$$z = f\Big(f\big(f(w)\big)\Big) \Rightarrow \frac{dz}{dw} = \frac{dz}{dy}\frac{dy}{dx}\frac{dx}{dw}$$

$\Rightarrow$ algorithme de rétro-propagation : propager les gradients de la couche de sortie vers la couche d'entrée

Algorithme de rétro-propagation

En deux mots...





1. propagation des observations dans le réseau
2. calcul de la perte
3. rétro-propagation du gradient
4. descente de gradient et mise à jour les paramètres

Mise en oeuvre Keras

En pratique, deux étapes après avoir créé le modèle :

1. on compile le modèle
   - algo. de descente de gradient + fonction de perte [1]
2. on fitte le modèle sur les données
   - taille du batch et nombre d'epochs
   - 1 epoch = 1 passage sur tout le jeu de données

```
# create model
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=input_dim))
model.add(Dense(100, activation='relu'))
model.add(Dense(10, activation='softmax'))
# compile
model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])
# fit
model.fit(X_train_vec, Y_train, batch_size=32, epochs=10, verbose=1)
```

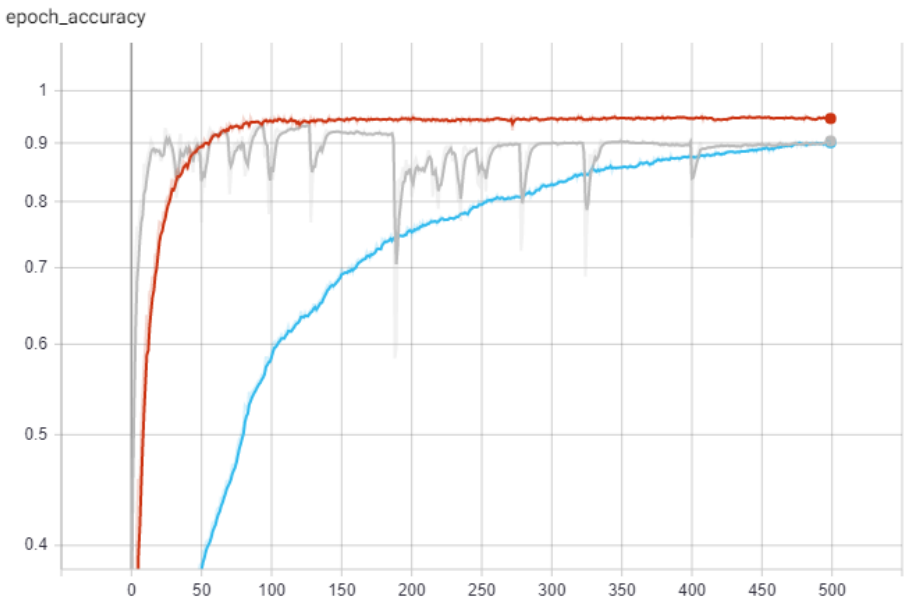
```
Epoch 1/10
60000/60000 [==============================] - 8s 130us/step - loss: 0.7074 - acc: 0.8124
Epoch 2/10
38848/60000 [==================>...........] - ETA: 2s - loss: 0.3218 - acc: 0.9092
```

1. https://keras.io/optimizers/ et https://keras.io/losses

Mise en oeuvre Keras

En pratique, un paramètre clé = le learning rate :

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} L\big(y_i, f(x_i)\big)$$



- η trop faible
- η adapté
- η trop élevé

$\Rightarrow$ à vérifier / optimiser...ainsi que l'algorithme de descente de gradient en lui même (e.g., SGD vs Adam vs RMSprop).

Apprentissage

- Descente de gradient & rétro-propagation
- Régularisation
- Augmentation de données
- Transfer learning
- En pratique

Régularisation et réseaux de neurones

Réseaux de neurones / deep-learning : nombreux paramètres

$\Rightarrow$ fort risque de sur-apprentissage.

Différentes stratégies pour régulariser un réseau de neurones

- weight-decay
- early stopping
- drop-out
- ....

Régularization et "weight decay"

Weight-decay = stratégie de régularisation "classique"

- basée sur la norme du vecteur **w** de paramètres

Régularization et "weight decay"

Weight-decay = stratégie de régularisation "classique"

- basée sur la norme du vecteur $\mathbf{w}$ de paramètres

$\Rightarrow$ fonction objective = risque empirique pénalisé :

$$\begin{aligned}\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) &= \sum_{i=1}^{n} L\big(y_i, f(x_i)\big) + \lambda\Omega(\mathbf{w}) \\ &= J(\mathbf{w};\mathbf{X},\mathbf{y}) + \lambda\Omega(\mathbf{w})\end{aligned}$$

avec typiquement $\Omega(\mathbf{w}) = \frac{1}{2}||w||_2^2$ ou $\Omega(\mathbf{w}) = ||w||_1$.

Régularization et "weight decay"

Weight-decay = stratégie de régularisation "classique"

- basée sur la norme du vecteur $\mathbf{w}$ de paramètres

$\Rightarrow$ fonction objective = risque empirique pénalisé :

$$\begin{aligned}\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) &= \sum_{i=1}^{n} L\big(y_i, f(x_i)\big) + \lambda\Omega(\mathbf{w}) \\ &= J(\mathbf{w};\mathbf{X},\mathbf{y}) + \lambda\Omega(\mathbf{w})\end{aligned}$$

avec typiquement $\Omega(\mathbf{w}) = \frac{1}{2}||w||_2^2$ ou $\Omega(\mathbf{w}) = ||w||_1$.

Conséquence : modification de la descente de gradient.

Régularization et "weight decay"

Terminologie "weight-decay" liée au cas $\Omega(\mathbf{w}) = \frac{1}{2}||w||_2^2$

- le cas classique, historique

Régularization et "weight decay"

Terminologie "weight-decay" liée au cas $\Omega(\mathbf{w}) = \frac{1}{2}||w||_2^2$

- le cas classique, historique

Fonction objective : $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \frac{\lambda}{2} \sum_{j=1}^{p} w_j^2$

Régularization et "weight decay"

Terminologie "weight-decay" liée au cas $\Omega(\mathbf{w}) = \frac{1}{2}||w||_2^2$

- le cas classique, historique

Fonction objective : $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \frac{\lambda}{2} \sum_{j=1}^{p} w_j^2$

Gradient (par rapport à w_j) : $\nabla_{w_j} \tilde{J} = \nabla_{w_j} J + \lambda w_j$

Régularization et "weight decay"

Terminologie "weight-decay" liée au cas $\Omega(\mathbf{w}) = \frac{1}{2}||w||_2^2$

- le cas classique, historique

Fonction objective : $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \frac{\lambda}{2} \sum_{j=1}^{p} w_j^2$

Gradient (par rapport à w_j) : $\nabla_{w_j} \tilde{J} = \nabla_{w_j} J + \lambda w_j$

Descente de gradient :

$$\begin{aligned} w_j^{(t+1)} &= w_j^{(t)} - \eta \nabla_{w_j} \tilde{J} \\ &= w_j^{(t)} - \eta \left(\nabla_{w_j} J + \lambda w_j^{(t)} \right) \\ &= (1 - \eta\lambda) w_j^{(t)} - \eta \nabla_{w_j} J \end{aligned}$$

$\Rightarrow$ "decay" de w_j d'un facteur $(1 - \eta\lambda)$ à chaque itération

- decay = "shrinkage"

Régularization et "weight decay"

Mise en oeuvre `Keras` : le module regularizers

- `kernel.regularizers.` { `l2`, `l1` et `l1_l2` }.
- à appliquer aux couches voulues (Dense et Conv)
- paramètre = λ

Exemple :

```python
# import
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.regularizers import l2, l1
# build model
model = Sequential()
model.add(Conv2D(6, (5,5), padding = 'same', activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(16, (5,5), activation='relu', kernel_regularizer=l2(0.01)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(120, activation='relu'))
model.add(Dense(10, activation='softmax', kernel_regularizer=l1(0.01)))
# show summary
model.summary()
```

$\Rightarrow$ couches à régulariser et valeurs de λ ... à optimiser !

Régularisation et "early stopping"

Stratégie early-stopping :

- utiliser une partie des données comme jeu de validation
- mesurer l'erreur de validation au fil des epochs
- arrêter l'apprentissage quand elle est minimale

$\Rightarrow$ très (le plus ?) classique !

2. https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-

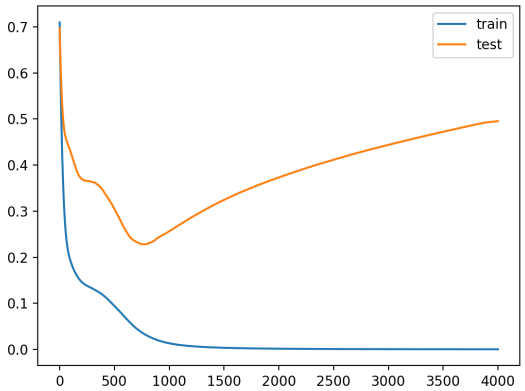
Régularisation et "early stopping"

Stratégie early-stopping :

- utiliser une partie des données comme jeu de validation
- mesurer l'erreur de validation au fil des epochs
- arrêter l'apprentissage quand elle est minimale

⇒ très (le plus ?) classique !

Comportement type sans early-stopping [2] :




```
# fit model
history = model.fit(X_train, y_train,
                    validation_data=(X_val, y_val),
                    epochs=4000)
# plot
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
plt.legend()
ptl.show()
```

⇒ objectif : s'arrêter vers ∼ 750 epochs

2. https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-

Régularisation et "early stopping"

Early stopping : facile à mettre en oeuvre grâce au mécanisme de " callbacks" de `Keras`

Callback `Keras` :

- permet d'intéragir avec le processus d'apprentissage
  - exemple type : mesurer la performance du modèle
- spécifié lors de l'appel à la méthode `fit()` du modèle
  - on peut en spécifier plusieurs (dans une liste)
- nombreux call-backs disponibles
  - early stopping, sauver le meilleur modèle, sauver les performances dans un fichier `.csv`...
  - voir `https://keras.io/callbacks/`

$\Rightarrow$ le callback "early stopping" = EarlyStopping()

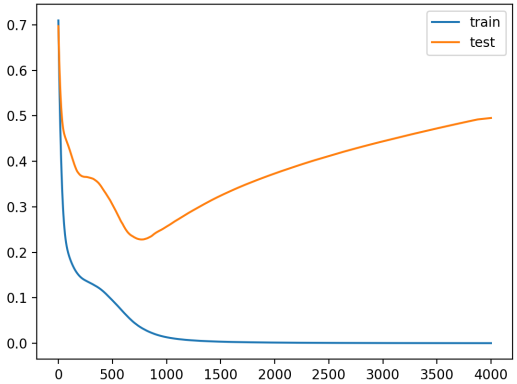
Régularisation et "early stopping"

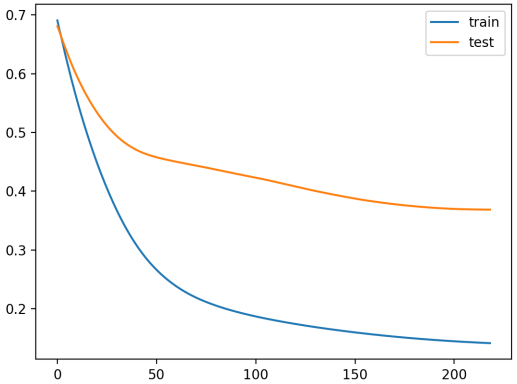
Exemple :

```
early_stop = EarlyStopping(monitor='val_loss', mode='min')
history = model.fit(X_train, y_train,
                    validation_data=(X_val, y_val),
                    epochs=4000, callbacks=[early_stop])
```

on suit la "validation loss", on s'arrête si elle est minimale

Résultat : arrêt à $\sim$ 200 epochs





$\Rightarrow$ sensible aux minimum locaux

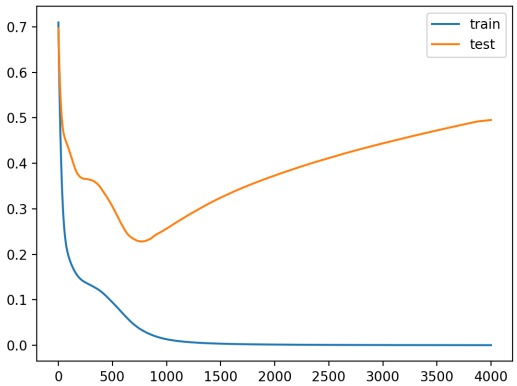
Régularisation et "early stopping"

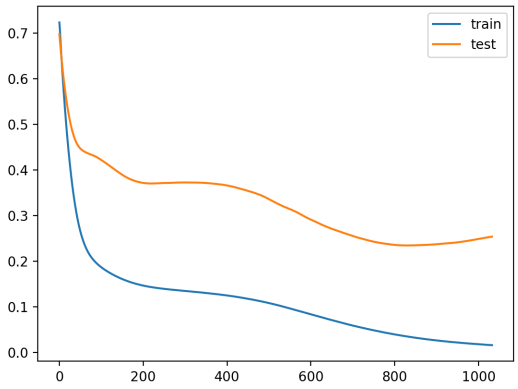
Introduction d'un critère de patience :

```
early_stop = EarlyStopping(monitor='val_loss', mode='min', patience = 200)
history = model.fit(X_train, y_train,
                    validation_data=(X_val, y_val),
                    epochs=4000, callbacks=[early_stop])
```

⇒ nombre d'epochs sans amélioration à attendre

Résultat : arrêt à $\sim$ 1000 epochs



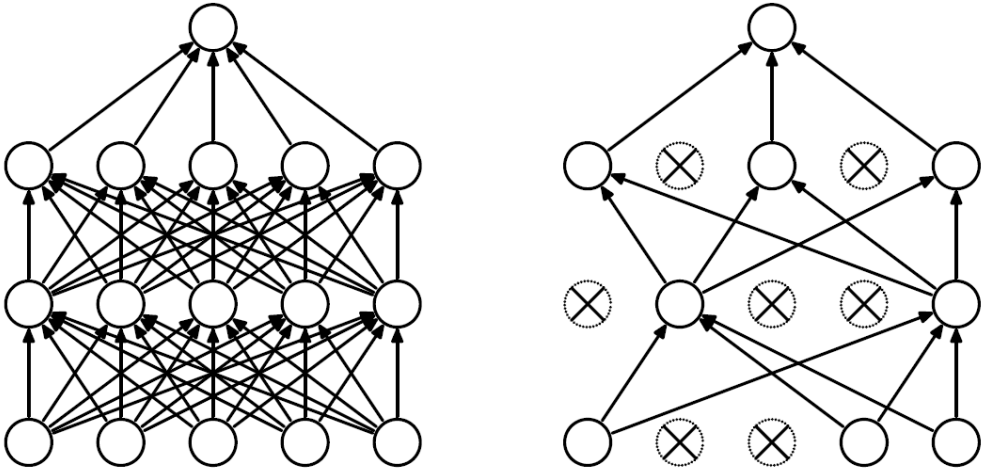


- robuste aux minimum locaux
- mais l'erreur peut augmenter pendant la "patience"

⇒ à combiner avec le callback ModelCheckpoint()

Régularisation et "dropout"

Principe du dropout : "éteindre" des neurones [3]



(a) Standard Neural Net (b) After applying dropout.

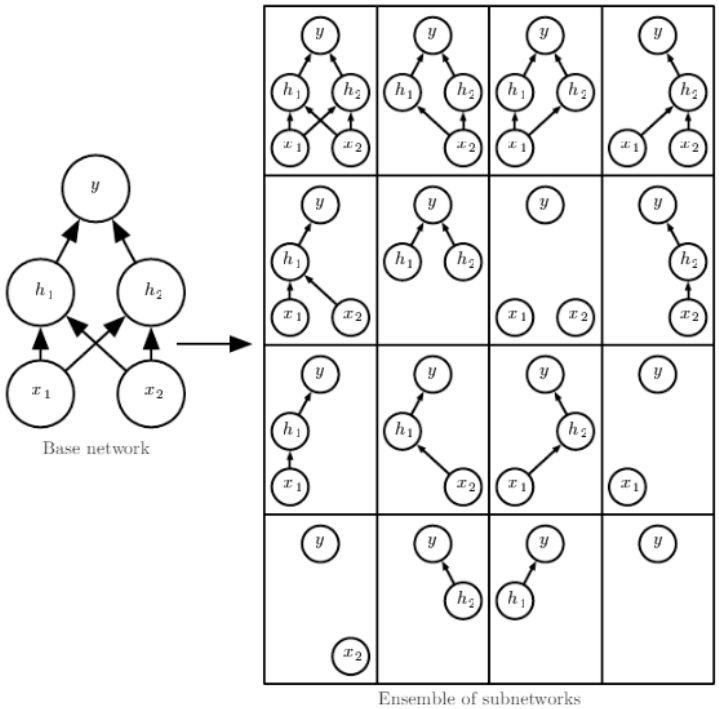
Chaque fois qu'on traite une instance lors de l'apprentissage :

- chaque neurone est "éteint" avec une probabilité p
  - qui peut être ajustée par couche
- l'instance est propagée et rétro-propagée
- elle contribue au gradient des neurones "actifs"

3. Image tirée de Srivastava et al. (2014).

Régularisation et "dropout"

Interprétation du dropout : considérer un ensemble de réseaux



- bons résultats empiriques
- justifications théoriques (Srivastava et al. (2014)).
- intuitivement : limite la co-adaptation des neurones

Régularisation et "dropout"

Mise en oeuvre `Keras` : un type de couche dédiée

```
# import
from keras.models import Sequential
from keras.layers import Dense, Dropout
# define parameters
n_classes = 10
p = 100
# build model
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=p))
model.add(Dropout(rate=0.5))
model.add(Dense(100, activation='relu'))
model.add(Dense(n_classes, activation='softmax'))
model.summary()
```

```
# import
from keras.models import Sequential
from keras.layers import Dense, Dropout
# define parameters
n_classes = 10
p = 100
# build model
model = Sequential()
model.add(Dropout(rate=0.2,input_shape=(p,)))
model.add(Dense(64, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(100, activation='relu'))
model.add(Dropout(rate=0.3))
model.add(Dense(n_classes, activation='softmax'))
model.summary()
```

- paramètre = taux de neurones d'entrée à éteindre
  - $\sim$ probabilité p
  - $+$ nombre de variables si couche d'entrée
- on peut mettre les couches où on veut
  - couche d'entrée et/ou couches internes
  - pas forcément toutes les couches
- on peut ajuster le taux dans les couches
  - e.g. 0.2 en entrée, 0.5 en interne (Goodfellow et al., 2016)

Apprentissage

- Descente de gradient & rétro-propagation
- Régularisation
- Augmentation de données
- Transfer learning
- En pratique

Augmentation de données

Augmentation de données (data augmentation) = appliquer des transformations aux données originales pour augmenter la taille du jeu d'apprentissage.

⇒ pour images : rotations, translations, changements d'échelle, symétrie, ...

https://medium.com/@pierre_guillou/data-augmentation-par-fastai-v1-e2e69e071ccc

Augmentation de données

Mise en oeuvre `Keras` : classe ImageDataGenerator

- du module `keras.preprocessing.image`

Un générateur : génère un batch à la demande pour SGD

- e.g., pour ne pas charger toutes les données en mémoire

Un image-générateur : génère un batch d'images en leur appliquant des transformations

- types de transformation spécifiés dans le constructeur
- appliquées aléatoirement à chaque image

$\Rightarrow$ s'utilisent via la méthode fit_generator() du modèle.

Exemple : `https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html`

Apprentissage

- Descente de gradient & rétro-propagation
- Régularisation
- Augmentation de données
- Transfer learning
- En pratique

Transfer learning

"Applications" `Keras` : modèles d'imagerie pré-entrainés

- (sur ImageNet)

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

$\Rightarrow$ mise à disposition pour :

- prédiction et/ou "feature extraction"
- les adapter à son problème : transfer-learning

Transfer learning

Transfer learning (ou "fine tuning") :

1. charger modèle pré-entrainé (sur ImageNet)
2. re-apprendre certaines couches sur ses données

Transfer learning

Transfer learning (ou "fine tuning") :

1. charger modèle pré-entrainé (sur ImageNet)
2. re-apprendre certaines couches sur ses données

⇒ typiquement :

- garder les couches de convolution figées
  - "feature extraction" générique
- ré-apprendre un MLP à la suite
  - couche de sortie en lien avec l'application
- éventuellement, ré-apprendre également les dernières couches de convolution

Transfer learning

Exemple : transfer-learning basé sur le modèle VGG16

```
# import
from keras.layers import Dense, Flatten
from keras.models import Sequential
from keras.applications.vgg16 import VGG16
# specify size of input data & number of classes
patch_size = 64
nb_classes = 10
# load vgg16 model
conv_base = VGG16(weights='imagenet', include_top=False,
                  input_shape = (patch_size,patch_size,3))
# create model with vgg16 + mlp
model = Sequential()
model.add(conv_base)
model.add(Flatten())
model.add(Dense(units = 100, activation = 'relu'))
model.add(Dense(nb_classes, activation = 'softmax'))
# freeze convolutional layers
conv_base.trainable = False
# show summary
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 2, 2, 512)         14714688
_________________________________________________________________
flatten_6 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_11 (Dense)             (None, 100)               204900
_________________________________________________________________
dense_12 (Dense)             (None, 10)                1010
=================================================================
Total params: 14,920,598
Trainable params: 205,910
Non-trainable params: 14,714,688
_________________________________________________________________
```

⇒ A noter :

`include_top=False`

- ne pas charger le MLP du modèle initial

`conv_base.trainable=False`

- ne pas ré-apprendre les couches de convolution

Apprentissage

- Descente de gradient & rétro-propagation
- Régularisation
- Augmentation de données
- Transfer learning
- En pratique

En pratique...

My two cents :

- pour des problèmatiques d'imagerie
  1. partir de modèles existants par transfer-learning
  2. mettre en place des procédures de data-augmentation
- essayer différentes structures de réseaux
  - nombre et paramètres des couches de convolution
  - nombre et taille des couches du MLP
- trouver une structure prometteuse
  - compromis performance et # paramètres
- optimiser les paramètres de l'apprentissage
  - paramètres du SGD, introduction de dropout, ...
- considérer beaucoup d'epochs et utiliser une combinaison de call-backs "early-stopping" et "sauvegarde du meilleur modèle"
  - EarlyStopping() et ModelCheckPoint()

Conclusion

En résumé...

Réseaux de neurones / deep-learning : des modèles performants et maintenant incontournables

- des méthodes datant des années 90
- remises au goût du jour récemment (calcul et données)

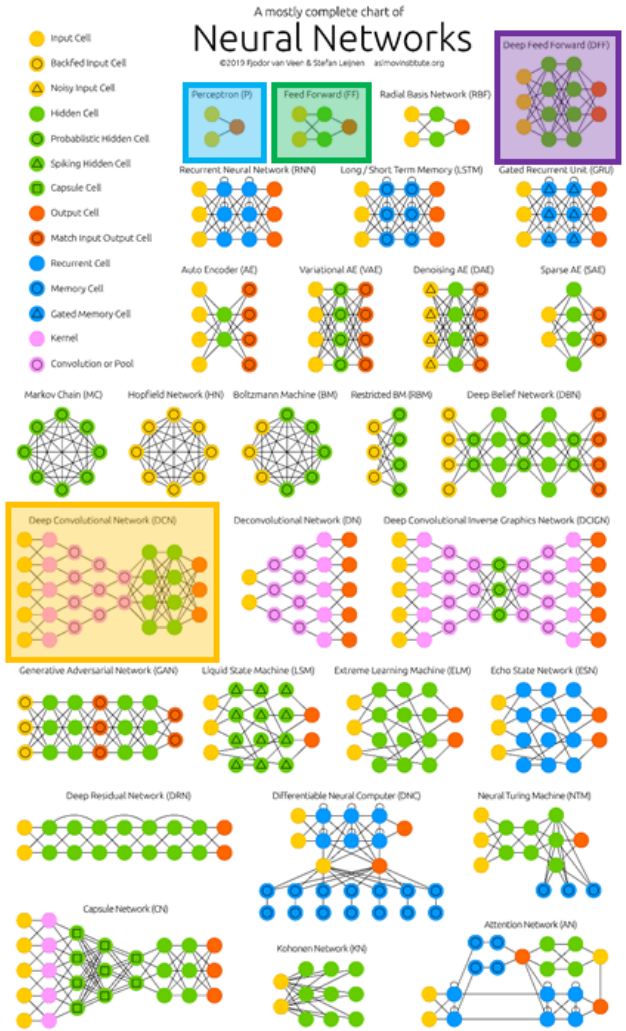
Ce cours : **une introduction !**

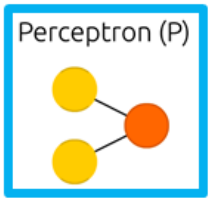
- quelques architectures clés
  - neurone artificiel, MLPs, CNNs
- quelques éléments pour l'apprentissage
  - back-propagation, SGD, régularisation, transfer-learning
- des exemples de mise en oeuvre avec `Keras`
  - API "séquentielle"

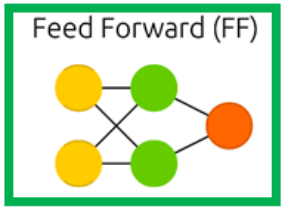
$\Rightarrow$ très simple à mettre en oeuvre / à ré-utiliser.

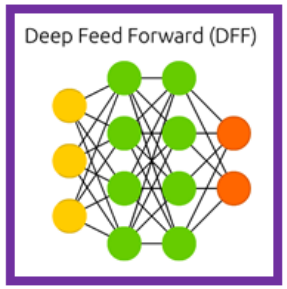
De nombreuses perspectives...

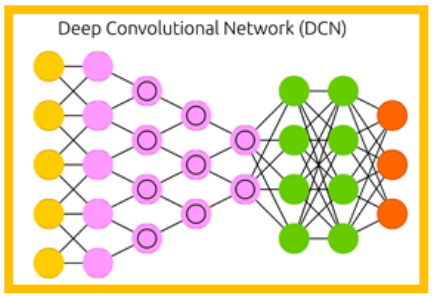
Ce cours :





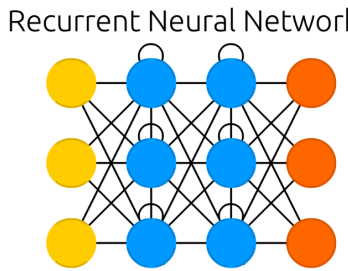


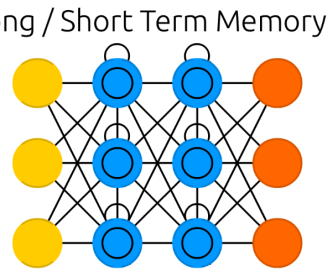




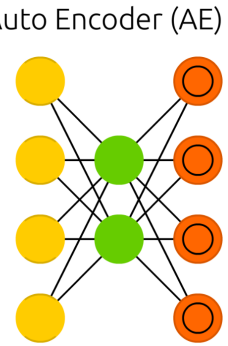
Perspectives - quelques pistes pour aller plus loin

- réseaux récurrents (type LSTM) pour l'analyse de séquences (e.g., texte, parole, séries temporelles)

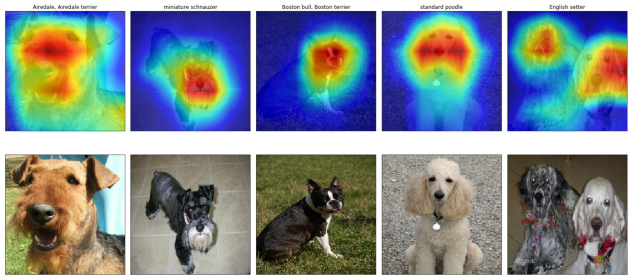




- auto-encoders pour l'analyse non-supervisée



- comment interpréter les prédictions d'un réseau de neurones (e.g., via class-activation maps)



Pour conclure

Réseaux de neurones / deep-learning : ubiquitaires

- modèles élégants et performants (et à la mode)
- approche "end to end" générique

Néanmoins : des modèles difficiles à paramétrer

- beaucoup de paramètres
- fonctions objectives non-convexes
- hyper-paramètres pas toujours évidents à régler

$\Rightarrow$ risque d'overfitting si quantité de données limitée !

Mon conseil : ne pas oublier trop vite les bonnes vieilles SVMs, forêts aléatoires et régressions (logistique) pénalisées !

Références

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15 :1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

Back-up

Théorème d'approximation universelle

Tiré d'un cours de Stéphane Canu [4] :

MLP with one hidden layer as universal approximator

Universal approximation theorem for MLP

- given any $\varepsilon > 0$
- for any continuous function f on compact subsets of $\mathbb{R}^p$
- for any admissible activation function σ (not a polynomial)
- there exists h, $W_1 \in \mathbb{R}^{p \times h}$, $b \in \mathbb{R}^h$, $c \in \mathbb{R}$ and $w_2 \in \mathbb{R}^h$ such that

$$\|f(x) - w_2\sigma(W_1 x + b) + c\|_\infty \leq \varepsilon$$

SVM, Boosting and Random Forest also

Approximation theory of the MLP model in neural networks, A Pinkus - Acta Numerica, 1999
The power of depth for feedforward neural networks, R. Eldan and O. Shamir, 2015.

4. https://github.com/StephaneCanu/Deep_learning_lecture

CNN - Architectures modernes

Depuis AlexNet [5] :

2012 Teams	%error
Supervision (Toronto)	15.3
ISI (Tokyo)	26.1
VGG (Oxford)	26.9
XRCE/INRIA	27.0
UvA (Amsterdam)	29.6
INRIA/LEAR	33.4

2013 Teams	%error
Clarifai (NYU spinoff)	11.7
NUS (singapore)	12.9
Zeiler-Fergus (NYU)	13.5
A. Howard	13.5
OverFeat (NYU)	14.1
UvA (Amsterdam)	14.2
Adobe	15.2
VGG (Oxford)	15.2
VGG (Oxford)	23.0

2014 Teams	%error
GoogLeNet	6.6
VGG (Oxford)	7.3
MSRA	8.0
A. Howard	8.1
DeeperVision	9.5
NUS-BST	9.7
TTIC-ECP	10.2
XYZ	11.2
UvA	12.1

shallow approaches

deep learning

Y. LeCun StatLearn tutorial

$\Rightarrow$ forte adhésion de la communauté "computer vision" !

5. https://github.com/StephaneCanu/Deep_learning_lecture