

Devoir4

December 9, 2019

1 IFT603 - Devoir 4

- Classifieur linéaire avec un réseau de neurones à une couche
- Fonction de perte: entropie-croisée
- Descente de gradient
- Recherche d'hyperparamètres
- Visualisation des résultats

2 Configuration

```
[10]: '''  
      Imporation des bibliothèques python générales  
      '''  
  
import numpy as np  
import matplotlib.pyplot as plt  
import itertools  
from sklearn.datasets import make_classification  
  
'''  
      Imporation des bibliothèques spécifiques au devoir  
      '''  
  
import utils  
from linear_classifier import LinearClassifier  
from two_layer_classifier import TwoLayerClassifier  
import warnings  
warnings.filterwarnings('ignore')  
  
%matplotlib inline  
plt.rcParams['figure.figsize'] = (14.0, 8.0) # set default size of plots  
  
%load_ext autoreload  
%autoreload 2
```

The autoreload extension is already loaded. To reload it, use:

```
%reload_ext autoreload
```

3 1ère partie du devoir : la classification linéaire par régression logistique

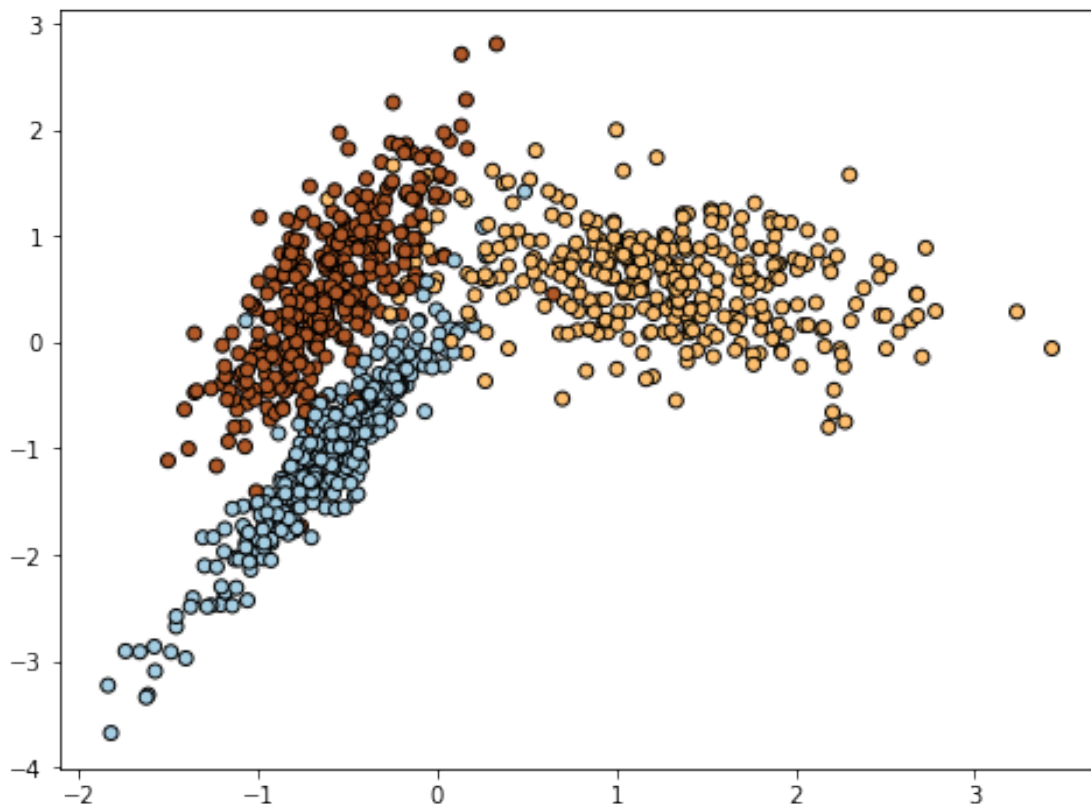
Préparation des données

Nous utilisons une fonction bien connue de sklearn pour générer un jeu de données nommée `make_classification`. Cette base de données comprend 1000 éléments distribués dans 3 classes.

```
[11]: # Générer des données
X_, y_ = make_classification(1000, n_features=2, n_redundant=0,
    ↪ n_informative=2, n_clusters_per_class=1, n_classes=3, random_state=6)

# Centrer et réduire les données (moyenne = 0, écart-type = 1)
mean = np.mean(X_, axis=0)
std = np.std(X_, axis=0)
X_ = (X_ - mean) / std

# Afficher
plt.figure(figsize=(8, 6))
plt.scatter(X_[:, 0], X_[:, 1], c=y_, edgecolors='k', cmap=plt.cm.Paired)
plt.show()
```



Séparons le jeu de données en trois parties: l'ensemble **d'entraînement**, de **validation** et de **test** (*train*, *val* et *test*).

```
[12]: num_val = 200
num_test = 200
num_train = 600
np.random.seed(1)
idx = np.random.permutation(len(X_))

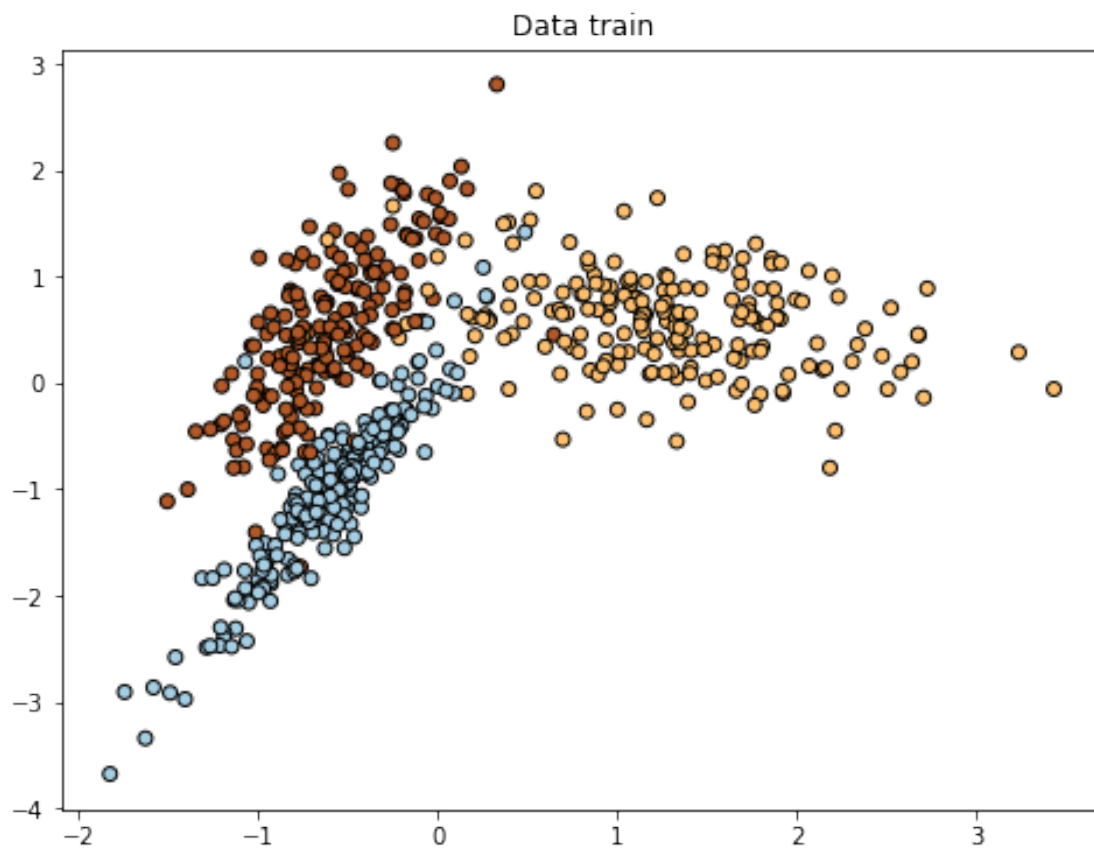
train_idx = idx[:num_train]
val_idx = idx[num_train:num_train + num_val]
test_idx = idx[-num_test:]

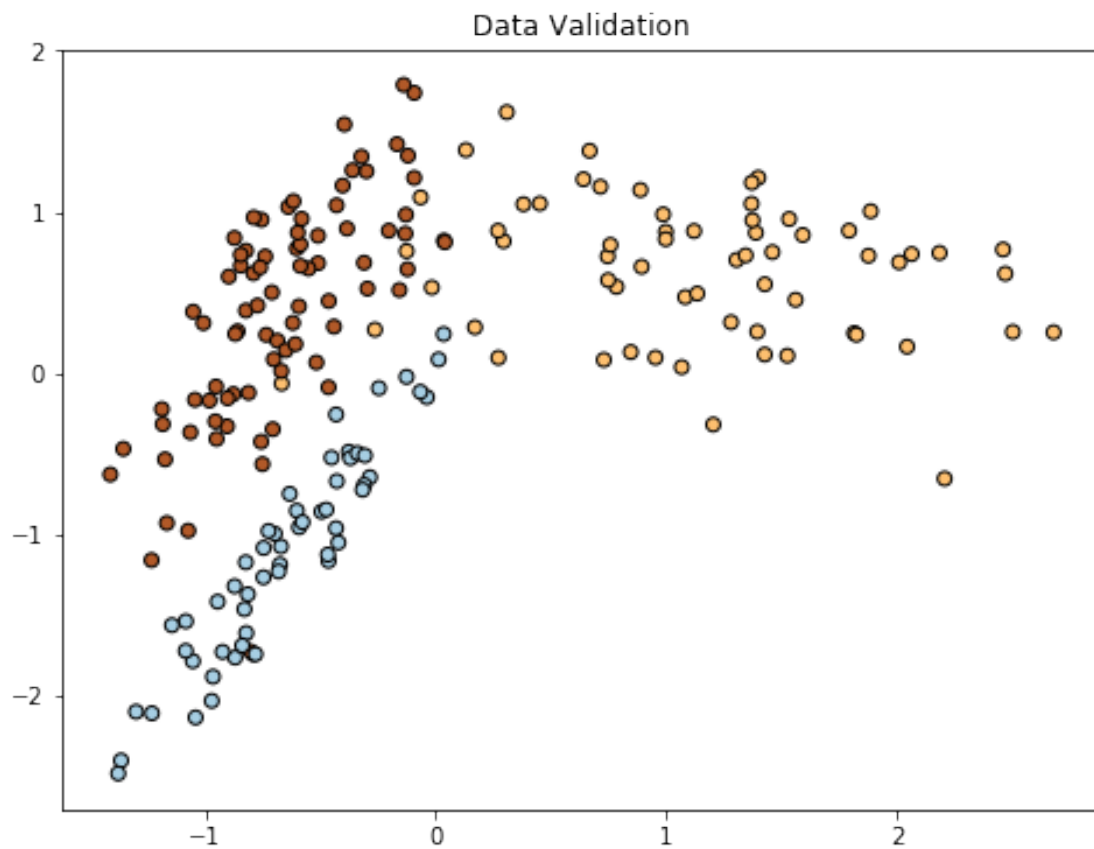
X_train = X_[train_idx]
y_train = y_[train_idx]
X_val = X_[val_idx]
y_val = y_[val_idx]
X_test = X_[test_idx]
y_test = y_[test_idx]

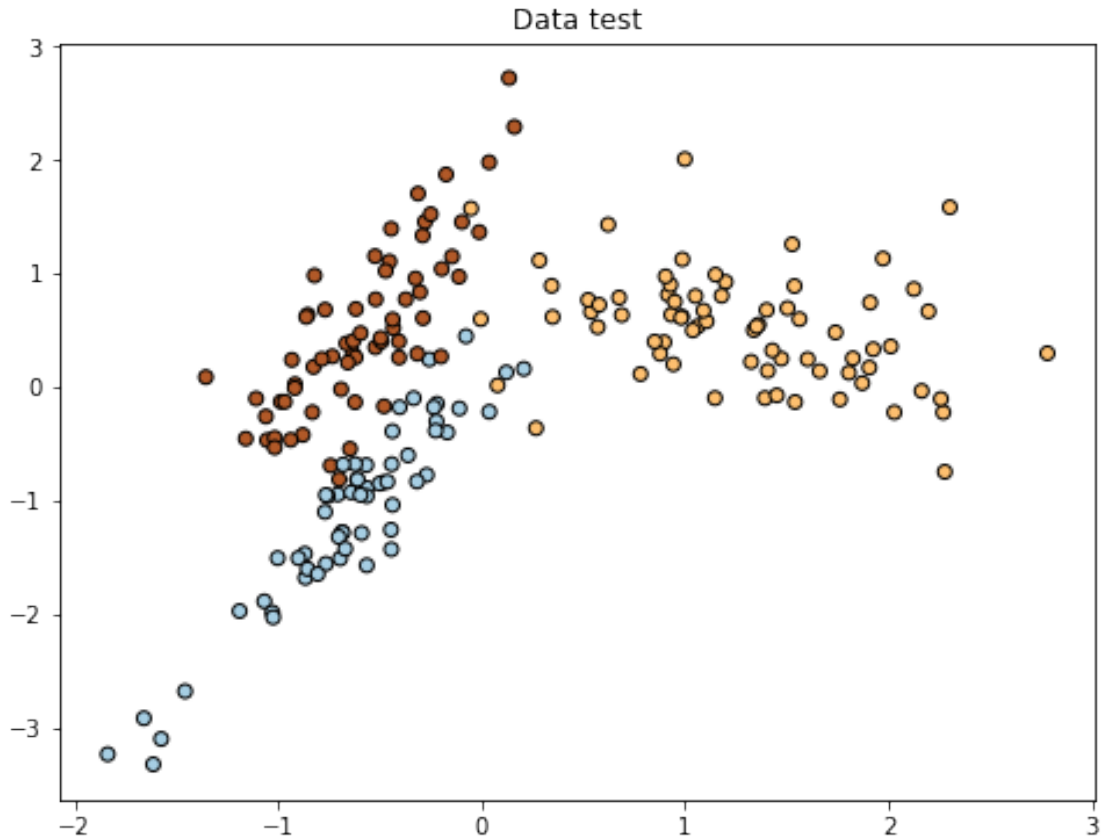
# Afficher
plt.figure(figsize=(8, 6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, edgecolors='k', cmap=plt.
    ↪cm.Paired)
plt.title('Data train')
plt.show()

plt.figure(figsize=(8, 6))
plt.scatter(X_val[:, 0], X_val[:, 1], c=y_val, edgecolors='k', cmap=plt.cm.
    ↪Paired)
plt.title('Data Validation')
plt.show()

plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, edgecolors='k', cmap=plt.cm.
    ↪Paired)
plt.title('Data test')
plt.show()
```







Vérifions que les données sont valides en entraînant un SVM de la librairie `scikit-learn`.

```
[13]: accu = utils.test_sklearn_svm(X_train, y_train, X_test, y_test)
print('Test accuracy: {:.3f}'.format(accu))
if accu < 0.7:
    print('ERREUR: L\'accuracy est trop faible. Il y a un problème avec les_
    ↪ données. Vous pouvez essayer de refaire le mélange (case ci-haut).')
```

Test accuracy: 0.950

4 Implémenter un classifieur logistique linéaire

Dans `linear_classifer.py`, implémenter la `cross_entropy_loss` (entropie croisée) ainsi que les méthodes avec l'indicatif `TODO`

Une fois fait, vérifier votre implémentation avec les cases qui suivent.

Commençons par quelques **Sanity Checks**

```
[14]: # En premier, vérifier la prédiction du modèle, la "forward pass"
# 1. Générer le modèle avec des poids W aléatoires
model = LinearClassifier(X_train, y_train, X_val, y_val, num_classes=3,
    ↪ bias=True)

# 2. Appeler la fonction qui calcule l'accuracy et la loss moyenne pour
    ↪ l'ensemble des données d'entraînement
_, loss = model.global_accuracy_and_cross_entropy_loss(X_train, y_train)

# 3. Comparer au résultat attendu
loss_attendu = -np.log(1.0/3.0) # résultat aléatoire attendu soit -log(1/
    ↪ nb_classes)
print('Sortie: {} Attendu: {}'.format(loss, loss_attendu))
if abs(loss - loss_attendu) > 0.05:
    print('ERREUR: la sortie de la fonction est incorrecte.')
else:
    print('SUCCÈS')
```

Sortie: 1.0835997088096694 Attendu: 1.0986122886681098
SUCCÈS

```
[15]: # Vérification: Vous devez pouvoir faire du surapprentissage sur quelques
    ↪ échantillons.
# Si l'accuracy reste faible, votre implémentation a un bogue.
n_check = 5
X_check = X_train[:n_check]
y_check = y_train[:n_check]
model = LinearClassifier(X_check, y_check, X_val, y_val, num_classes=3,
    ↪ bias=True)
loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = model.
    ↪ train(num_epochs=10, lr=1.0, l2_reg=0.0)
accu_train_finale = accu_train_curve[-1]
print('Accuracy d\'entraînement, devrait être 1.0: {:.3f}'.
    ↪ format(accu_train_finale))
if accu_train_finale < 0.9999:
    print('ATTENTION: L\'accuracy n\'est pas 100%.')
    utils.plot_curves(loss_train_curve, loss_val_curve, accu_train_curve,
    ↪ accu_val_curve)
else:
    print('SUCCÈS')
```

Accuracy d'entraînement, devrait être 1.0: 1.000
SUCCÈS

Lorsque ça fonctionne, maintenant testons l'effet du terme de régularisation `l2_reg`. **Augmenter le terme `l2_reg` devrait augmenter la loss et, à la limite, faire décroître l'accuracy**

```
[16]: # Prenons encore un petit échantillon et testons différentes valeurs de l2_reg
n_check = 5
X_check = X_train[:n_check]
y_check = y_train[:n_check]
model = LinearClassifier(X_check, y_check, X_val, y_val, num_classes=3,
    ↪ bias=True)

for l2_r in np.arange(0,1,0.05):
    loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = model.
    ↪ train(num_epochs=10, lr=1.0, l2_reg=l2_r)
    print('l2_reg= {:.4f} >> Loss/accuracy d\'entraînement : {:.3f} {:.3f}'.
    ↪ format(l2_r,loss_train_curve[-1],accu_train_curve[-1]))
```

```
l2_reg= 0.0000 >> Loss/accuracy d\'entraînement : 0.117 1.000
l2_reg= 0.0500 >> Loss/accuracy d\'entraînement : 0.657 0.800
l2_reg= 0.1000 >> Loss/accuracy d\'entraînement : 0.803 0.800
l2_reg= 0.1500 >> Loss/accuracy d\'entraînement : 0.914 0.800
l2_reg= 0.2000 >> Loss/accuracy d\'entraînement : 1.021 0.800
l2_reg= 0.2500 >> Loss/accuracy d\'entraînement : 1.127 0.800
l2_reg= 0.3000 >> Loss/accuracy d\'entraînement : 1.228 0.800
l2_reg= 0.3500 >> Loss/accuracy d\'entraînement : 1.322 0.800
l2_reg= 0.4000 >> Loss/accuracy d\'entraînement : 1.408 0.800
l2_reg= 0.4500 >> Loss/accuracy d\'entraînement : 1.486 0.800
l2_reg= 0.5000 >> Loss/accuracy d\'entraînement : 1.561 0.600
l2_reg= 0.5500 >> Loss/accuracy d\'entraînement : 1.636 0.600
l2_reg= 0.6000 >> Loss/accuracy d\'entraînement : 1.716 0.600
l2_reg= 0.6500 >> Loss/accuracy d\'entraînement : 1.804 0.600
l2_reg= 0.7000 >> Loss/accuracy d\'entraînement : 1.904 0.800
l2_reg= 0.7500 >> Loss/accuracy d\'entraînement : 2.004 0.800
l2_reg= 0.8000 >> Loss/accuracy d\'entraînement : 2.042 0.800
l2_reg= 0.8500 >> Loss/accuracy d\'entraînement : 2.724 0.200
l2_reg= 0.9000 >> Loss/accuracy d\'entraînement : 7.944 0.200
l2_reg= 0.9500 >> Loss/accuracy d\'entraînement : 41.568 0.200
```

5 Vous pouvez maintenant essayer d'entraîner le modèle avec les données complètes

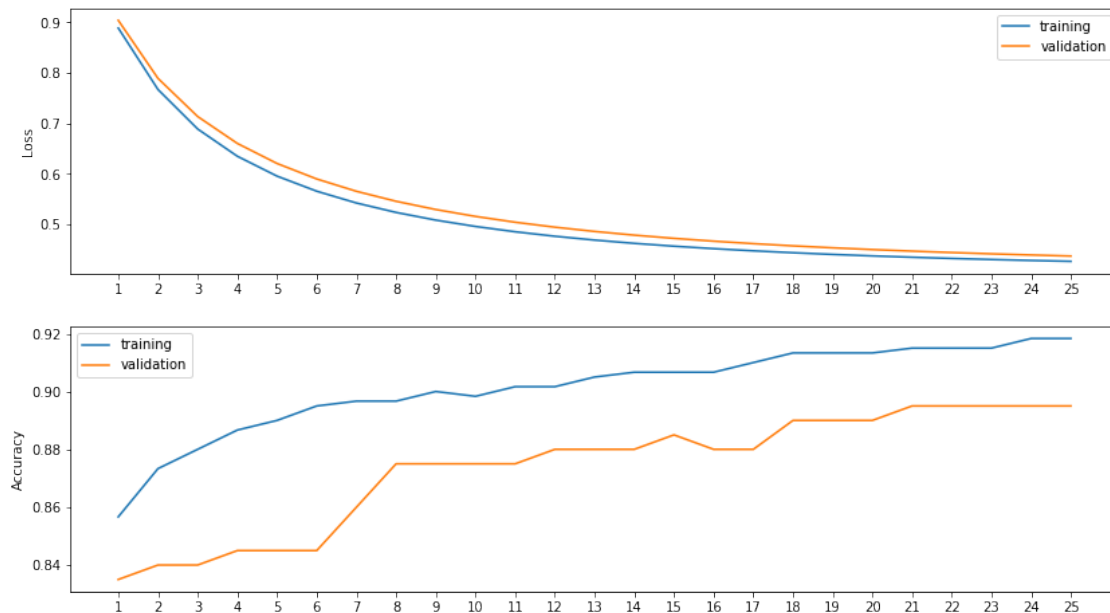
Normalement la loss devrait décroître et l'accuracy augmenter en fonction des epochs

```
[17]: # On instancie et entraîne notre modèle; cette fois-ci avec les données
    ↪ complètes.
model = LinearClassifier(X_train, y_train, X_val, y_val, num_classes=3,
    ↪ bias=True)
loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = model.
    ↪ train(lr=0.001,num_epochs=25, l2_reg=0.01)
```



```
# Illustration de la loss et de l'accuracy (le % de biens classés) à chaque
↳ itération
utils.plot_curves(loss_train_curve, loss_val_curve, accu_train_curve,
↳ accu_val_curve)

print('[Training] Loss: {:.3f} Accuracy: {:.3f}'.
↳ format(loss_train_curve[-1], accu_train_curve[-1]))
print('[Validation] Loss: {:.3f} Accuracy: {:.3f}'.format(loss_val_curve[-1],
↳ accu_val_curve[-1]))
```



```
[Training] Loss: 0.427 Accuracy: 0.918
[Validation] Loss: 0.438 Accuracy: 0.895
```

6 Recherche de meilleurs hyperparamètres

Nous allons effectuer une recherche sur ces hyperparamètres:

- **learning rate:** La longueur des pas lors de la descente de gradient
- **L2 regularization:** La pénalité sur la taille des poids dans W

```
[18]: lr_choices = [1e-2, 1e-1, 1.0, 10.0]
reg_choices = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
lr_decay = 0.995 # learning rate is multiplied by this factor after each step

best_accu = -1
```

```

best_params = None
best_model = None
best_curves = None

for lr, reg in itertools.product(lr_choices, reg_choices):
    params = (lr, reg)
    curves = model.train(num_epochs=25, lr=lr, l2_reg=reg, lr_decay=lr_decay)
    loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = curves

    val_accu = accu_val_curve[-1]
    if val_accu > best_accu:
        print('Best val accuracy: {:.3f} | lr: {:.0e} | l2_reg: {:.0e}'.
        ↪format(val_accu, lr, reg))
        best_accu = val_accu
        best_params = params
        best_model = model
        best_curves = curves

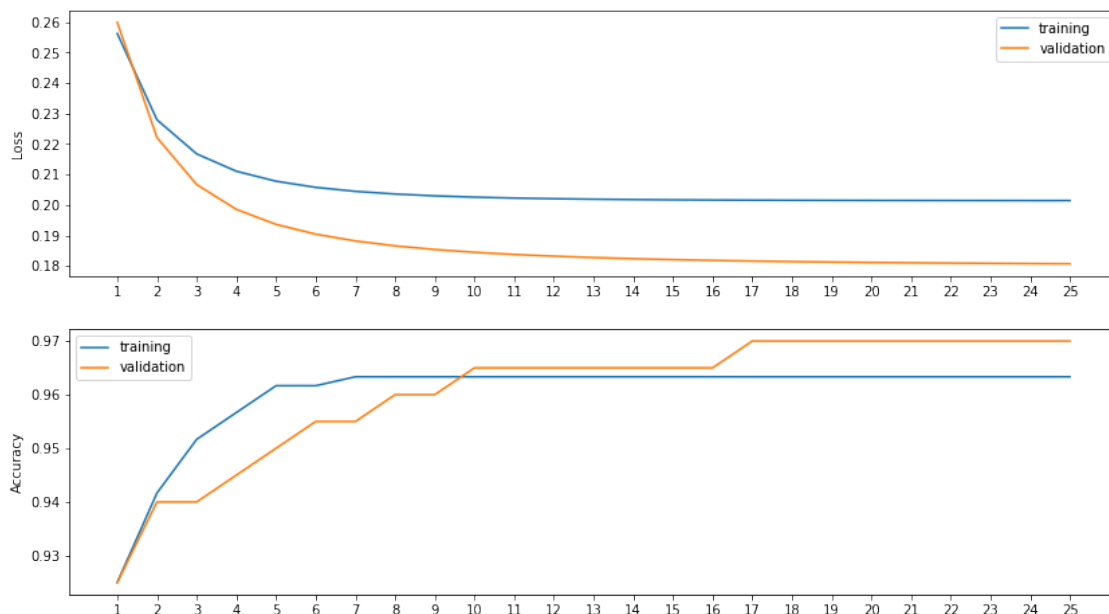
model = best_model
utils.plot_curves(*best_curves)

```

```

Best val accuracy: 0.845 | lr: 1e-02 | l2_reg: 1e-01
Best val accuracy: 0.925 | lr: 1e-02 | l2_reg: 1e-02
Best val accuracy: 0.935 | lr: 1e-02 | l2_reg: 1e-03
Best val accuracy: 0.940 | lr: 1e-02 | l2_reg: 1e-04
Best val accuracy: 0.945 | lr: 1e-01 | l2_reg: 1e-03
Best val accuracy: 0.970 | lr: 1e-01 | l2_reg: 1e-04

```



7 Vérifier la généralisation sur l'ensemble de test

```
[19]: # On ré-entraîne le modèle avec les meilleurs hyper-paramètres
lr, reg = best_params
model.train(num_epochs=25, lr=lr, l2_reg=reg, lr_decay=lr_decay)

pred = model.predict(X_test)
accu = (pred == y_test).mean()
print('Test accuracy: {:.3f}'.format(accu))
```

Test accuracy: 0.950

8 Tracer les frontières de décision

Nous allons créer une grille de points 2D qui recouvre les données, et nous allons prédire la classe pour chacun de ces points dans l'espace. Cela nous permettra de visualiser les frontières de décision apprises.

```
[20]: h = 0.01 # contrôle la résolution de la grille
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5 # Limites de la grille
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h)) #
    ↪ Créer la grille

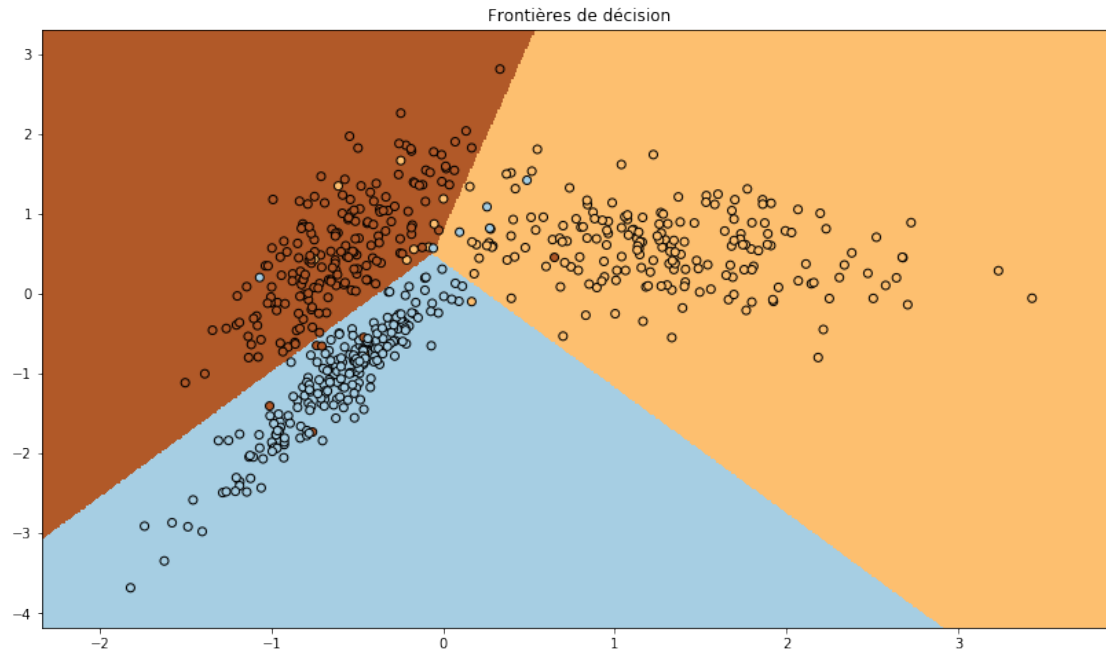
X_predict = np.c_[xx.ravel(), yy.ravel()] # Convertir la grille en une liste
    ↪ de points
Z = model.predict(X_predict) # Classifier chaque point de la grille
Z = Z.reshape(xx.shape) # Remettre en 2D

plt.figure(figsize=(14, 8))
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired) # Colorier les cases selon les
    ↪ prédictions

X_plot, y_plot = X_train, y_train
X_plot, y_plot = X_train, y_train
plt.scatter(X_plot[:, 0], X_plot[:, 1], c=y_plot, edgecolors='k', cmap=plt.cm.
    ↪ Paired) # Tracer les données

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.title('Frontières de décision')
plt.show()
```



9 2e partie du devoir: la classification non-linéaire

Nous essaierons maintenant d'apprendre des frontières de décision **non-linéaires**.

```
[21]: #Choisissez le type de données que vous voulez

# NOTE IMPORTANTE: on vous encourage à tester différentes bases de données.
→Ceci dit,
# votre solution sera testée avec Ncircles (N=4). Vous devez donc tester cette
→option.
dataset_type = 'Ncircles'
if dataset_type == 'moons':
    X_, y_ = sklearn.datasets.make_moons(n_samples=200, noise=0.5)
    num_classes = 2
elif dataset_type == 'gaussian_quantiles':
    X_, y_ = sklearn.datasets.make_gaussian_quantiles(n_samples=200,
→n_classes=2)
    num_classes = 2
elif dataset_type == '4blobs':
    d = 4
    c1a = np.random.randn(50, 2)
    c1b = np.random.randn(50, 2) + (d, d)
    c2a = np.random.randn(50, 2) + (0, d)
    c2b = np.random.randn(50, 2) + (d, 0)
```

```

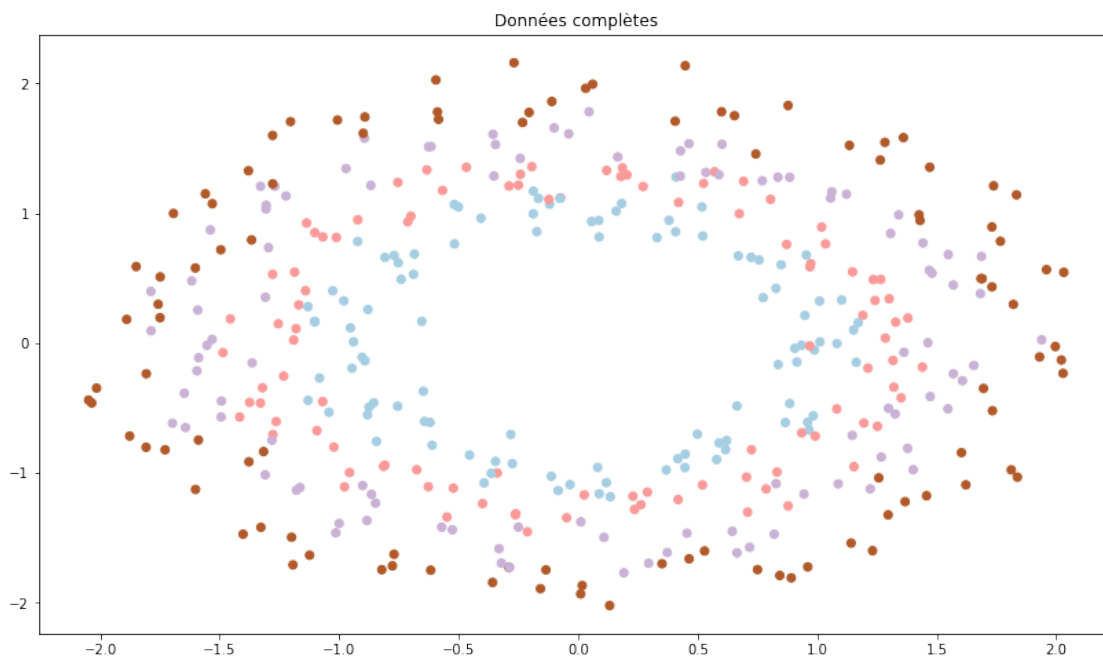
X_ = np.concatenate([c1a, c1b, c2a, c2b], axis=0)
y_ = np.array([0] * 100 + [1] * 100)
num_classes = 2
elif dataset_type == '2circles':
    X_, y_ = sklearn.datasets.make_circles(n_samples=200)
    num_classes = 2
elif dataset_type == 'Ncircles':
    samples_per_class = 100
    num_classes = 4
    angles = np.linspace(0, 2*np.pi, samples_per_class)
    radius = 1.0 + np.arange(num_classes) * 0.3
    px = np.cos(angles[:, None]) * radius[None, :] # (100, 3)
    py = np.sin(angles[:, None]) * radius[None, :] # (100, 3)
    X_ = np.stack([px, py], axis=-1).reshape((samples_per_class * num_classes, 2))
    X_ += np.random.randn(len(X_[:, 0]), 2) / 8
    y_ = np.array(list(range(num_classes)) * samples_per_class)
else:
    print('Invalid dataset type')

```

```

[23]: plt.figure()
plt.scatter(X_[:, 0], X_[:, 1], c=y_, cmap=plt.cm.Paired)
plt.title('Données complètes')
plt.show()

```



```
[24]: train_proportion = 0.5
      val_proportion = 0.2
      num_train = int(len(X_) * train_proportion)
      num_val = int(len(X_) * val_proportion)

      np.random.seed(0)
      idx = np.random.permutation(len(X_))

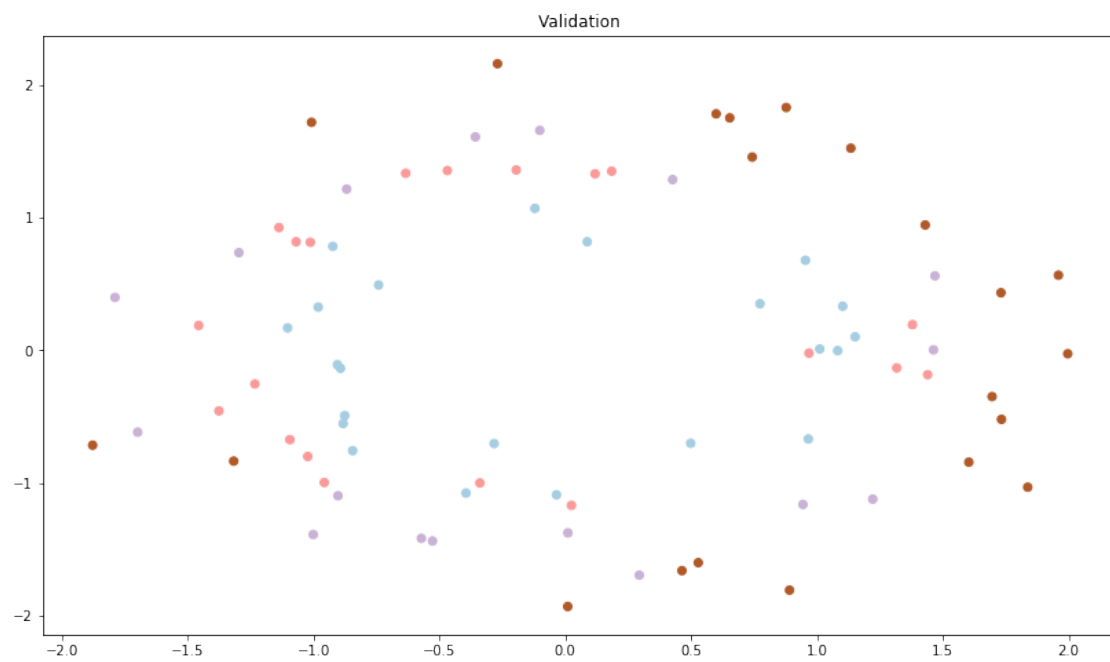
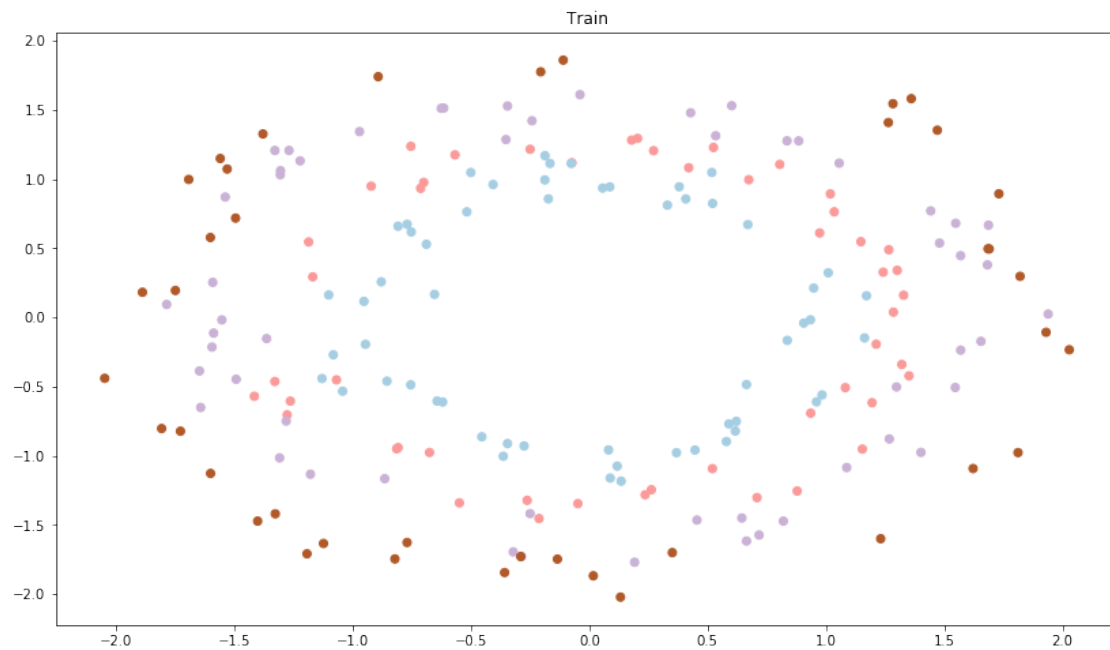
      train_idx = idx[:num_train]
      val_idx = idx[num_train:num_train + num_val]
      test_idx = idx[num_train + num_val:]

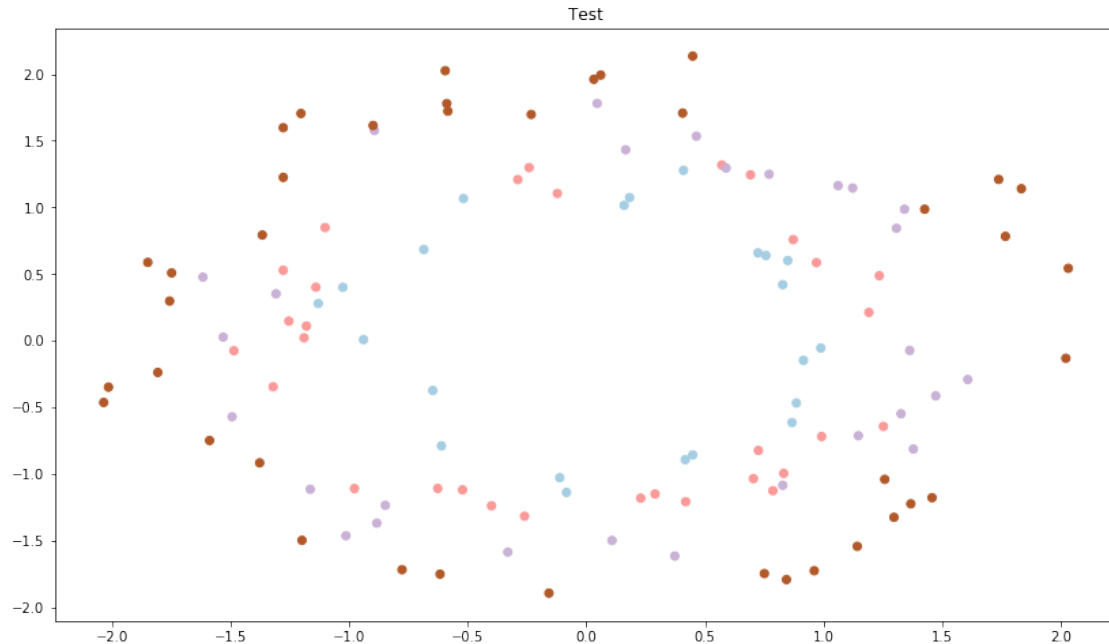
      X_train = X_[train_idx]
      y_train = y_[train_idx]
      X_val = X_[val_idx]
      y_val = y_[val_idx]
      X_test = X_[test_idx]
      y_test = y_[test_idx]

[25]: # Affichons maintenant les données d'entraînement, de validation et de test.
      plt.figure()
      plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired)
      plt.title('Train')
      plt.show()

      plt.figure()
      plt.scatter(X_val[:, 0], X_val[:, 1], c=y_val, cmap=plt.cm.Paired)
      plt.title('Validation')
      plt.show()

      plt.figure()
      plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=plt.cm.Paired)
      plt.title('Test')
      plt.show()
```





10 Entraîner avec la descente de gradient

Dans `two_layer_classifier.py`, complétez les méthodes avec l'indicatif `TODO`.

Commençons avec quelques **Sanity checks**

```
[26]: num_hidden_neurons = 10
model = TwoLayerClassifier(X_train, y_train, X_val, y_val,
                           num_features=2,
                           ↪ num_hidden_neurons=num_hidden_neurons, num_classes=num_classes)

[27]: # Vérifier que la sortie du réseau initialisé au hasard donne bien une
      ↪ prédiction égale pour chaque classe
num_hidden_neurons = 10
model = TwoLayerClassifier(X_train, y_train, X_val, y_val,
                           num_features=2,
                           ↪ num_hidden_neurons=num_hidden_neurons, num_classes=num_classes)

# 2. Appeler la fonction qui calcule l'accuracy et la loss moyenne pour
      ↪ l'ensemble des données d'entraînement
_, loss = model.global_accuracy_and_cross_entropy_loss(X_train, y_train, 0)

# 3. Comparer au résultat attendu
loss_attendu = -np.log(1.0/num_classes) # résultat aléatoire attendu soit
      ↪ -log(1/nb_classes)
```



```

print('Sortie: {} Attendu: {}'.format(loss, loss_attendu))
if abs(loss - loss_attendu) > 0.05:
    print('ERREUR: la sortie de la fonction est incorrecte.')
else:
    print('SUCCÈS')

```

Sortie: 1.386017383893022 Attendu: 1.3862943611198906
SUCCÈS

```

[28]: # Vérifier que le fait d'augmenter la régularisation L2 augmente également la
      ↪ loss
      for l2_r in np.arange(0,2,0.1):
          _, loss = model.global_accuracy_and_cross_entropy_loss(X_train,y_train,
          ↪ l2_r)
          print('l2_reg= {:.4f} >> Loss/accuracy d\'entraînement : {:.3f}'.
          ↪ format(l2_r,loss))

```

```

l2_reg= 0.0000 >> Loss/accuracy d'entraînement : 1.386
l2_reg= 0.1000 >> Loss/accuracy d'entraînement : 1.435
l2_reg= 0.2000 >> Loss/accuracy d'entraînement : 1.485
l2_reg= 0.3000 >> Loss/accuracy d'entraînement : 1.534
l2_reg= 0.4000 >> Loss/accuracy d'entraînement : 1.583
l2_reg= 0.5000 >> Loss/accuracy d'entraînement : 1.633
l2_reg= 0.6000 >> Loss/accuracy d'entraînement : 1.682
l2_reg= 0.7000 >> Loss/accuracy d'entraînement : 1.731
l2_reg= 0.8000 >> Loss/accuracy d'entraînement : 1.781
l2_reg= 0.9000 >> Loss/accuracy d'entraînement : 1.830
l2_reg= 1.0000 >> Loss/accuracy d'entraînement : 1.880
l2_reg= 1.1000 >> Loss/accuracy d'entraînement : 1.929
l2_reg= 1.2000 >> Loss/accuracy d'entraînement : 1.978
l2_reg= 1.3000 >> Loss/accuracy d'entraînement : 2.028
l2_reg= 1.4000 >> Loss/accuracy d'entraînement : 2.077
l2_reg= 1.5000 >> Loss/accuracy d'entraînement : 2.126
l2_reg= 1.6000 >> Loss/accuracy d'entraînement : 2.176
l2_reg= 1.7000 >> Loss/accuracy d'entraînement : 2.225
l2_reg= 1.8000 >> Loss/accuracy d'entraînement : 2.274
l2_reg= 1.9000 >> Loss/accuracy d'entraînement : 2.324

```

```

[29]: # Vérification: Vous devez pouvoir faire du surapprentissage sur quelques
      ↪ échantillons.
      # Si l'accuracy reste faible, votre implémentation a un bogue.
      n_check = 5
      X_check = X_train[:n_check]
      y_check = y_train[:n_check]
      model = TwoLayerClassifieur(X_check, y_check, X_val, y_val,
          num_features=2, num_hidden_neurons=num_hidden_neurons,
          ↪ num_classes=num_classes

```

```

)

loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = model.
    ↪train(num_epochs=200, lr=0.01, l2_reg=0.0)
print('Accuracy d\'entraînement, devrait être 1.0: {:.3f}'.
    ↪format(accu_train_curve[-1]))
if accu_train_curve[-1] < 0.98:
    print('ATTENTION: L\'accuracy n\'est pas 100%.')
    utils.plot_curves(loss_train_curve, loss_val_curve, accu_train_curve,
    ↪accu_val_curve)
else:
    print('SUCCÈS')

```

Accuracy d'entraînement, devrait être 1.0: 1.000
SUCCÈS

```

[30]: # Vérifier que le fait d'entraîner avec une régularisation L2 croissante
    ↪augmente la loss et, éventuellement, diminue l'accuracy
for l2_r in np.arange(0,1,0.1):
    loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = model.
    ↪train(num_epochs=200, lr=0.01, l2_reg=l2_r)
    print('l2_reg= {:.4f} >> Loss/accuracy d\'entraînement : {:.3f} {:.3f}'.
    ↪format(l2_r,loss_train_curve[-1],accu_train_curve[-1]))

```

```

l2_reg= 0.0000 >> Loss/accuracy d'entraînement : 0.189 1.000
l2_reg= 0.1000 >> Loss/accuracy d'entraînement : 1.070 0.800
l2_reg= 0.2000 >> Loss/accuracy d'entraînement : 1.370 0.800
l2_reg= 0.3000 >> Loss/accuracy d'entraînement : 1.456 0.800
l2_reg= 0.4000 >> Loss/accuracy d'entraînement : 1.389 0.800
l2_reg= 0.5000 >> Loss/accuracy d'entraînement : 1.373 0.800
l2_reg= 0.6000 >> Loss/accuracy d'entraînement : 1.368 0.800
l2_reg= 0.7000 >> Loss/accuracy d'entraînement : 1.371 0.800
l2_reg= 0.8000 >> Loss/accuracy d'entraînement : 1.374 0.400
l2_reg= 0.9000 >> Loss/accuracy d'entraînement : 1.376 0.400

```

10.1 Let's train!

```

[31]: # On instancie notre modèle; cette fois-ci avec les données complètes.
num_hidden_neurons = 20
model = TwoLayerClassifier(X_train, y_train, X_val, y_val, num_features=2,
    num_hidden_neurons=num_hidden_neurons,
    ↪num_classes=num_classes, activation='relu')

```

```

[32]: loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = model.
    ↪train(num_epochs=200, lr=1e-2, l2_reg=0.0,

```

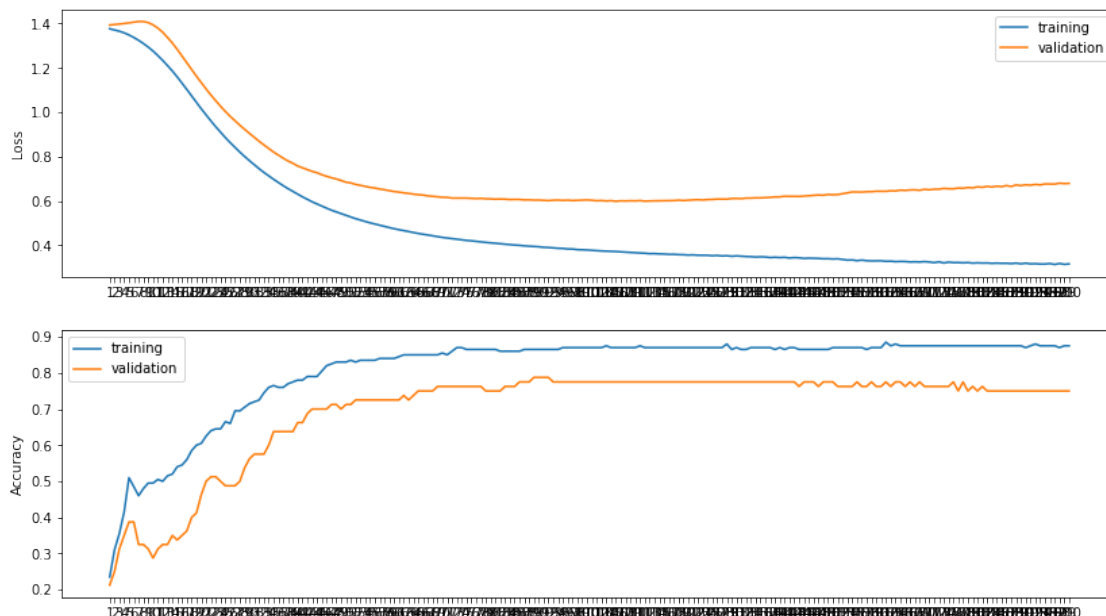
```

→ momentum=0.5)

# Illustration de la loss et de l'accuracy (le % de biens classés) à chaque
→ itération
utils.plot_curves(loss_train_curve, loss_val_curve, accu_train_curve,
→ accu_val_curve)

print('[Training] Loss: {:.3f} Accuracy: {:.3f}'.
→ format(loss_train_curve[-1], accu_train_curve[-1]))
print('[Validation] Loss: {:.3f} Accuracy: {:.3f}'.format(loss_val_curve[-1],
→ accu_val_curve[-1]))

```



```

[Training] Loss: 0.315 Accuracy: 0.875
[Validation] Loss: 0.679 Accuracy: 0.750

```

```

[33]: # Find the best hyperparameters lr and l2_reg
lr_choices = [1e-4, 1e-3, 1e-2]
reg_choices = [1e-1, 1e-2, 1e-3, 1e-4, 0]
lr_decay = 1.0 # 0.995 # learning rate is multiplied by this factor after
→ each step

best_accu = -1
best_params = None
best_model = None
best_curves = None

```

```

for lr, reg in itertools.product(lr_choices, reg_choices):
    params = (lr, reg)
    curves = model.train(num_epochs=50, lr=lr, l2_reg=reg,
↳lr_decay=lr_decay, momentum=0.5)
    loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = curves

    val_accu = accu_val_curve[-1]
    if val_accu > best_accu:
        print('Best val accuracy: {:.3f} | lr: {:.0e} | l2_reg: {:.0e}'.
↳format(val_accu, lr, reg))
        best_accu = val_accu
        best_params = params
        best_model = model
        best_curves = curves
    else:
        print('accuracy: {:.3f} | lr: {:.0e} | l2_reg: {:.0e}'.format(val_accu,
↳lr, reg))

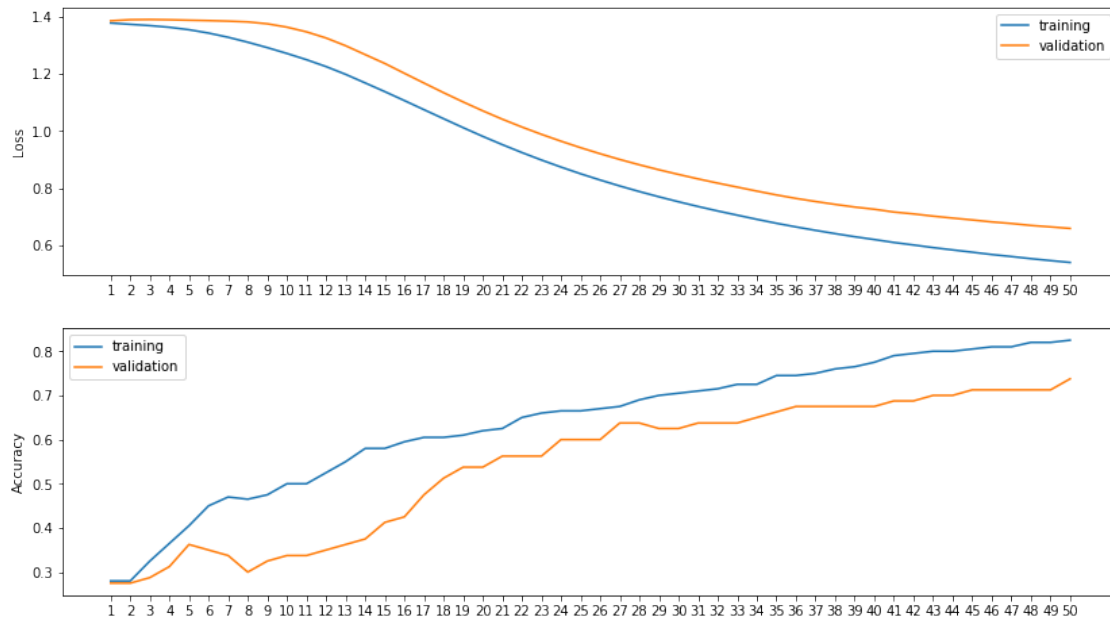
model = best_model
utils.plot_curves(*best_curves)

```

```

Best val accuracy: 0.275 | lr: 1e-04 | l2_reg: 1e-01
accuracy: 0.225 | lr: 1e-04 | l2_reg: 1e-02
accuracy: 0.212 | lr: 1e-04 | l2_reg: 1e-03
accuracy: 0.275 | lr: 1e-04 | l2_reg: 1e-04
accuracy: 0.250 | lr: 1e-04 | l2_reg: 0e+00
accuracy: 0.275 | lr: 1e-03 | l2_reg: 1e-01
Best val accuracy: 0.312 | lr: 1e-03 | l2_reg: 1e-02
accuracy: 0.300 | lr: 1e-03 | l2_reg: 1e-03
accuracy: 0.287 | lr: 1e-03 | l2_reg: 1e-04
accuracy: 0.263 | lr: 1e-03 | l2_reg: 0e+00
accuracy: 0.275 | lr: 1e-02 | l2_reg: 1e-01
Best val accuracy: 0.525 | lr: 1e-02 | l2_reg: 1e-02
Best val accuracy: 0.650 | lr: 1e-02 | l2_reg: 1e-03
Best val accuracy: 0.700 | lr: 1e-02 | l2_reg: 1e-04
Best val accuracy: 0.738 | lr: 1e-02 | l2_reg: 0e+00

```

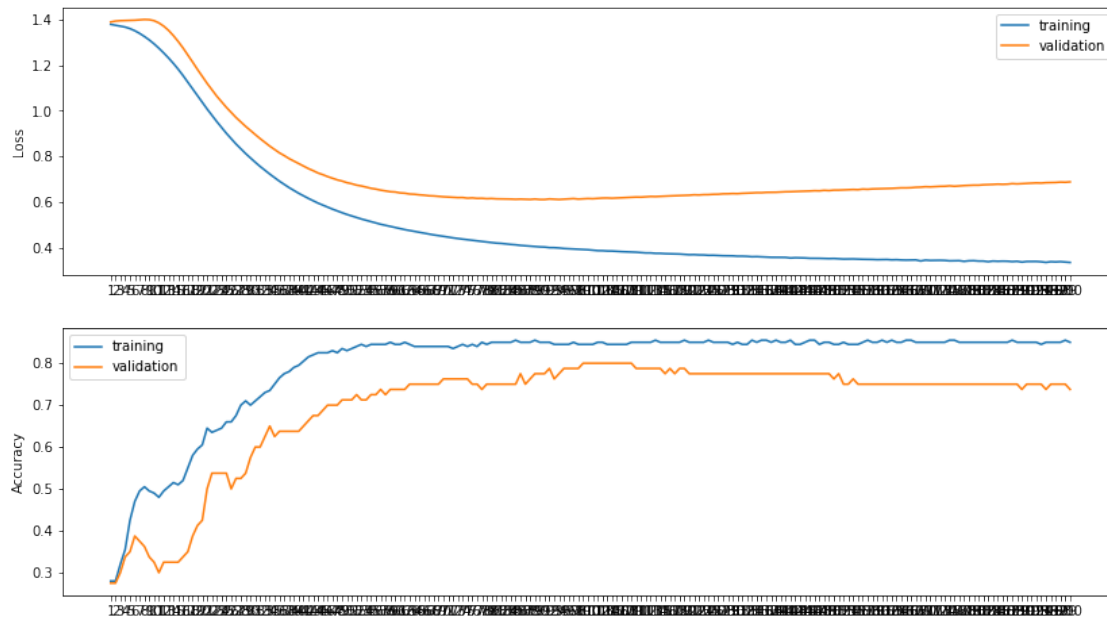


```
[34]: # On ré-entraîne le modèle avec les meilleurs hyper-paramètres
lr, reg = best_params
print(best_params)
curves = model.train(num_epochs=200, lr=lr, l2_reg=reg, momentum=0.5)
loss_train_curve, loss_val_curve, accu_train_curve, accu_val_curve = curves

pred = model.predict(X_test)
accu = (pred == y_test).mean()
print('Test accuracy: {:.3f}'.format(accu))
utils.plot_curves(*curves)
```

(0.01, 0)

Test accuracy: 0.708



11 Tracer les frontières de décision

Nous allons créer une grille de points 2D qui recouvre les données, et nous allons prédire la classe pour chacun de ces points dans l'espace. Cela nous permettra de visualiser les frontières de décision apprises.

```
[35]: # Visualisation des résultats

h = 0.05 # contrôle la résolution de la grille
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5 # Limites de la grille
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h)) #
↳ Créer la grille

X_predict = np.c_[xx.ravel(), yy.ravel()] # Convertir la grille en une liste
↳ de points
Z = model.predict(X_predict) # Classifier chaque point de la grille
Z = Z.reshape(xx.shape) # Remettre en 2D

plt.figure(figsize=(14, 8))
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired) # Colorier les cases selon les
↳ prédictions

X_plot, y_plot = X_, y_
```

```
plt.scatter(X_plot[:, 0], X_plot[:, 1], c=y_plot, edgecolors='k', cmap=plt.cm.
↪Paired) # Tracer les données

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())

plt.title('Frontières de décision')
plt.show()
```

