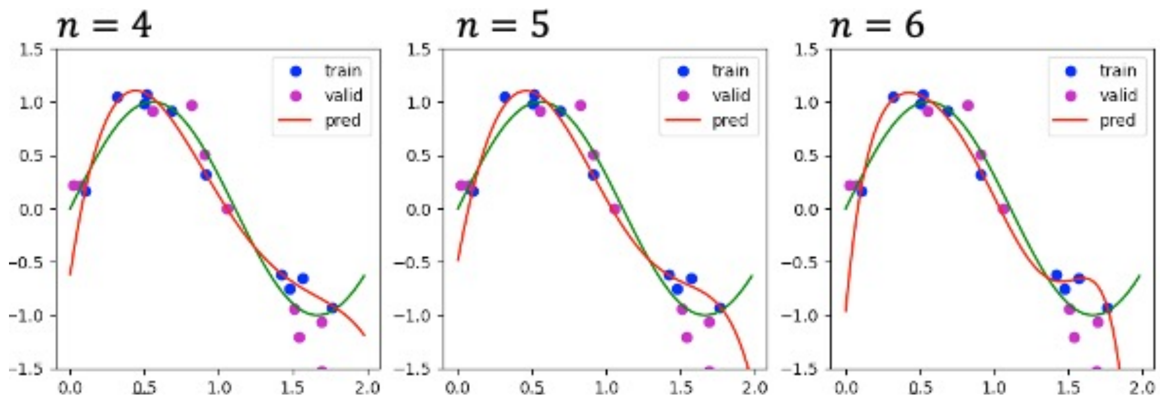
Séance de TP 1 –
Régression polynomial et descente de gradient

Sujet

Au cours de cette première séance nous allons nous intéresser à l'optimisation d'un modèle de régression polynomiale simple.
Nous comparerons l'algorithme de descente de gradient stochastique à l'algorithme des moindres carrés.

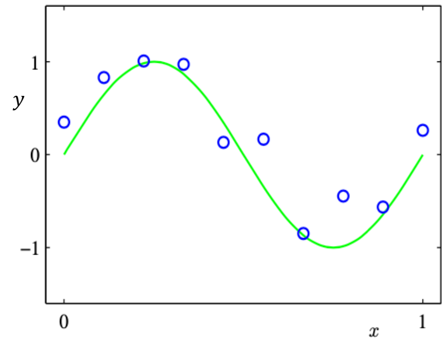
- Nous mettrons en évidence le compromis biais variance
- Nous constaterons l'intérêt d'une régularisation L2 pour les deux algorithmes
- Nous programmerons notre premier modèle neuronal avec Pytorch
- Nous observerons les conditions de convergence de l'algorithme de descente de gradient



Séance de TP 1

1- Les données

- Ce sont des données mono-dimensionnelles, sinusoïdales bruitées
- On génère un ensemble d'apprentissage de N=10,… échantillons
- On génère un ensemble de validationb de N=10,… échantillons
- Un bruit Gaussien d'amplitude variable est ajouté




```
N=10
x_train = np.random.random((N,1))*2
bruit_train = np.random.randn(N,1)*amplitude_bruit
Y_train = np.sin(2*np.pi*nu*x_train)+bruit_train
```

Séance de TP 1

1- Modèle polynomiale de régression et moindres carrés

N mesures, polynôme d'ordre m, défini par m paramètres

$$\hat{y}(k) = \theta_4 x^4 + \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$$

$$\begin{pmatrix} \hat{y}(1) \\ \cdot \\ \hat{y}(k) \\ \cdot \\ \hat{y}(N) \end{pmatrix} = \begin{pmatrix} 1\ x_{1,1} \dots x_{1,i} \dots x_{1,m} \\ \dots \\ 1\ x_{2,1} \dots x_{2,i} \dots x_{2,m} \\ \dots \\ 1\ x_{N,1} \dots x_{N,i} \dots x_{N,m} \end{pmatrix} \begin{pmatrix} \theta_0 \\ \cdot \\ \theta_i \\ \cdot \\ \theta_m \end{pmatrix}$$

$$\hat{Y} = X\theta$$

Le vecteur de paramètres optimal θ^* est celui qui minimise l'erreur MSE:

$$J(\theta) = \frac{1}{N}\sum_{k=1}^{N} e(k)^2 = \frac{1}{N}\sum_{k=1}^{N}(y(k) - \hat{y}(k))^2 = \frac{1}{N}\left(Y - \hat{Y}\right)^t\left(Y - \hat{Y}\right)$$

la solution est : $\theta^* = (X^T X)^{-1} X^T Y$

Séance de TP 1

1- Modèle polynomiale de régression et moindres carrés

Pour régulariser le modèle polynomial, et éviter le surajustement dans le cas d'un modèle sur-paramétré, on peut choisir de pénaliser les modèles qui ont des paramètres θ_i trop grand.

Pour cela on modifie le critère qui devient un critère de régression ridge, où weigh_decay

$$J(\theta) = \frac{1}{N}\sum_{k=1}^{N}(y(k) - \hat{y}(k))^2 + \lambda \sum_{k=1}^{N}(\theta_i)^2$$

$\lambda > 0$ est l'hyperparamètre qui contrôle la régression

la solution régularisée est : $\theta^* = (\lambda I + X^T X)^{-1} X^T Y$ où I est la matrice identité

```
def regression(x,Y,M,weight_decay = False, Lambda=1):
    N = np.shape(x)[0]
    X = np.ones((N,1))
    if weight_decay:
        I =np.identity(M+1)*Lambda
    else:
        I =np.zeros((M+1,M+1))
    XX = x
    for m in range(M):
        X = np.append(X,XX,axis=1)
        XX = np.multiply(XX,x)
    XT = X.T
    Theta = np.linalg.inv(I + XT @ X) @ XT @ Y
    Y_pred = X @ Theta
    return Y_pred, Theta
```

Séance de TP 1

1- Modèle polynomiale de régression et moindres carrés

Etudier le modèle polynomial pour des ordres inférieurs à 10

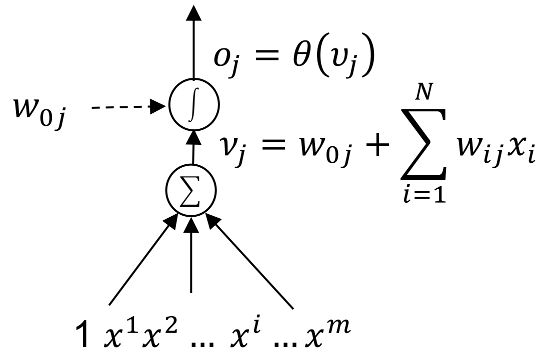
Utiliser le programme regression.py pour

- Mettre en évidence le compromis biais variance
- Mettre en évidence le sur-apprentissage
- Mettre en évidence l'intérêt de la régression ridge en donnant à λ différentes valeurs
- Evaluer le comportement de l'algorithme lorsque le nombre de mesures augmente N=100

2- Modèle polynomiale de régression et descente de gradient

On choisi maintenant de réaliser la fonction de régression polynomiale à l'aide d'un seul neurone de type perceptron, avec

- une fonction d'activation identité
- des entrées représentant les puissances de la variable x



o_j la sortie scalaire du neurone j
θ la fonction d'activation du neurone j

v_j l'activation du neurone j
w_{ij} les poids du neurone j
w_{0j} le biais

le vecteur d'entrée du neurone j

2- Modèle polynomiale de régression et descente de gradient

Etudier le neurone perceptron pour des ordres inférieurs à 7

Utiliser le programme MLP-regression.py pour

- Déterminer les propriétés de convergence de l'algorithme de descente de gradient, pour les différents ordre de la régression envisagée. Déterminer l'influence du learning rate, du nombre de d'époque
```
learning_rate = 1e-3
epochs = 5000
```
- Examiner la relation entre le critère MSE et l'amplitude du bruit
- Mettre en évidence l'intérêt du weight_decay en lui donnant différentes valeurs
```
weight_decay = 0     #, 0.01, 0.05, 0.1
```
- Evaluer le comportement de l'algorithme lorsque le nombre de mesures augmente N=100
- Comparer les solutions obtenues avec les moindres carrés et avec la descente de gradient

2. Apprentissage profond: Perceptron multicouches

2- Modèle polynomiale de régression et descente de gradient

Annexe: déclaration d'un réseaux avec pytorch

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

class NeuralNetwork(nn.Module):
    """
    Réseau de neurones avec deux couches
    Args:
        hidden_cells: nombre de neurones dans l'état caché
        input_features: nombre de features d'entrée
    """
    def __init__(self,  hidden_cells=8,input_features=1):
        super(NeuralNetwork, self).__init__()
        self.linear_stack = nn.Sequential(
            nn.Linear(input_features, hidden_cells),
            #nn.Sigmoid(), #ajout d'un activation simoid
            #nn.Linear(hidden_cells, 1), # ajout d'une autre couche
        )

    def forward(self, x):
        logits = self.linear_stack(x)
        return logits
```

2. Apprentissage profond: Perceptron multicouches

2- Modèle polynomiale de régression et descente de gradient

Annexe: boucle principale d'apprentissage

```
def train_loop(dataloader, model, loss_fn, optimizer):
    """
    Définit la boucle d'apprentissage d'une époch entière
    Returns:
        La loss moyenne sur l'époch
    """
    size = len(dataloader.dataset)
    nb_batches = len(dataloader)
    epoch_loss = 0
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X.float())
        loss = loss_fn(pred.float(), y.float())
        epoch_loss += loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 99 == 0:
            loss, current = loss.item(), (batch+1) * len(X)
            print(f"Train loss: {loss:>7f}  [{current:>5d}/{size:>5d}]\n")
    return epoch_loss / nb_batches
```

2. Apprentissage profond: Perceptron multicouches

2- Modèle polynomiale de régression et descente de gradient

Annexe: boucle de test

```
def test_loop(dataloader, model, loss_fn):
    """
    Evaluation sur le jeu de validation

    Returns:
        La loss moyenne sur le jeu de validation
    """
    size = len(list(dataloader.dataset))
    nb_batches = len(dataloader)
    test_loss = 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X.float())
            test_loss += loss_fn(pred, y).item()

    test_loss /= nb_batches
    print(f"Test loss: {test_loss:>8f} \n")
    return test_loss
```

2. Apprentissage profond: Perceptron multicouches

2- Modèle polynomiale de régression et descente de gradient

Annexe: déclaration du réseau et apprentissage

```
# définition du réseau
learning_rate = 1e-3
epochs = 5000
weight_decay = 0     #, 0.01, 0.05, 0.1]:
batch_size = 10

hidden_cells = 1 # une seule cellule pour le régresseur polynomial

model = NeuralNetwork(hidden_cells=hidden_cells,input_features=ordre)
mse_loss = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate,weight_decay=weight_decay)

# BOUCLE D'APPRENTISSAGE
for t in range(epochs):
    train_losses[ordre].append(train_loop(train_dataloader, model, mse_loss, optimizer))
    # on test à chaque itération
    test_losses[ordre].append(test_loop(valid_dataloader, model, mse_loss))
print("Done!")
```

2. Apprentissage profond: Perceptron multicouches

2- Modèle polynomiale de régression et descente de gradient

Annexe: préparation des données

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def features_poly(x,ordre):
    XX = x
    X = np.array(x)
    for i in range(1,ordre):
        XX = np.multiply(XX,x)
        X = np.append(X,XX,axis=1)
    return X

……………………………………………..


train_dataloader = DataLoader(list(zip(features_poly(x_train,ordre), Y_train)),
                                      batch_size=10, shuffle=True, drop_last=False)

valid_dataloader = DataLoader(list(zip(features_poly(x_valid,ordre), Y_valid)),
                                      batch_size=10, shuffle=False, drop_last=False)

regression_dataloader = DataLoader(list(zip(features_poly(xx,ordre), sin)),
                                          batch_size=10*N, shuffle=False, drop_last=False)
```