Étude des Fonctions de Perte par Calcul Symbolique

Gradients, Convexité et Applications

DONGMO TCHOUMENE ANITA BELVIANE - **22W2184**
DONFACK SYNTHIA CALORINE - **22U2073**
BOKOU-BOUNA-ANGE-LARISSA - **22W2188**
JIATSA ROMMEL JUNIOR - **22T2906**

Master I Data Science - Techniques d'Optimisation Avancée

2 octobre 2025

Plan

Fonctions de Perte Étudiées

Dans ce TP, nous étudions quatre fonctions de perte fondamentales :

- **Erreur Quadratique Moyenne (MSE)** - Régression
- **Entropie Croisée Binaire (BCE)** - Classification binaire
- **Entropie Croisée Catégorielle (CCE)** - Classification multi-classes
- **Perte de Huber** - Régression robuste

Objectifs :

1. Calculer les gradients analytiquement
2. Étudier la convexité
3. Visualiser et appliquer sur des données réelles

MSE : Définition et Formulation

Formulation :

$$L_{\mathsf{MSE}}(\mathbf{w}) = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

où $\hat{y}_i = \mathbf{w}^T\mathbf{x}_i$ pour la régression linéaire.

Forme vectorielle :

$$L_{\mathsf{MSE}}(\mathbf{w}) = \frac{1}{2m}||\mathbf{y} - \mathbf{X}\mathbf{w}||_2^2$$

Utilisation : Problèmes de régression, sensible aux outliers.

MSE : Calcul du Gradient par SymPy

Listing – Gradient de MSE

```python
import sympy as sp
import numpy as np

# Définition des symboles
w = sp.Matrix(sp.symbols('w1:4')) # w = [w1, w2, w3]
X = sp.Matrix(sp.symbols('x1:13')).reshape(4, 3) # 4x3
y = sp.Matrix(sp.symbols('y1:5')) # vecteur cible
m = sp.symbols('m', positive=True)

# Prédictions
y_pred = X * w

# MSE Loss
mse_loss = (1/(2*m)) * (y - y_pred).T * (y - y_pred)
mse_loss = mse_loss[0] # extraction scalaire

```

MSE : Résultats du Gradient

Gradient analytique :

$$\nabla_{\mathbf{w}} L_{\textsf{MSE}} = \frac{1}{m}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

Matrice Hessienne :

$$\mathbf{H}_{\textsf{MSE}} = \frac{1}{m}\mathbf{X}^T\mathbf{X}$$

Propriétés :

- Hessienne constante (indépendante de $\mathbf{w}$)
- $\mathbf{X}^T\mathbf{X}$ est semi-définie positive
- Si $\text{rang}(\mathbf{X}) = p$, alors strictement convexe

MSE : Convexité

Critère de convexité : Une fonction est convexe si sa Hessienne est semi-définie positive.
Pour MSE :

$$\mathbf{H}_{\mathsf{MSE}} = \frac{1}{m}\mathbf{X}^T\mathbf{X} \succeq 0$$

Preuve : Pour tout vecteur $\mathbf{v}$:

$$\mathbf{v}^T\mathbf{H}_{\mathsf{MSE}}\mathbf{v} = \frac{1}{m}\mathbf{v}^T\mathbf{X}^T\mathbf{X}\mathbf{v} = \frac{1}{m}||\mathbf{X}\mathbf{v}||_2^2 \geq 0$$

Conclusion : MSE est **convexe** avec un unique minimum global.

BCE : Définition et Formulation

Formulation :

$$L_{\mathsf{BCE}}(\mathbf{w}) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)]$$

où $\hat{p}_i = \sigma(\mathbf{w}^T\mathbf{x}_i) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}_i}}$ (fonction sigmoïde)

Utilisation : Classification binaire $(y_i \in \{0, 1\})$

Interprétation : Mesure la divergence entre les distributions prédite et réelle.

BCE : Calcul du Gradient par SymPy

Listing – Gradient de BCE

```python
# Fonction sigmoïde
def sigmoid_sympy(z):
    return 1 / (1 + sp.exp(-z))

# Logits et probabilités prédites
z = X * w # logits
p_pred = sp.Matrix([sigmoid_sympy(zi) for zi in z])

# BCE Loss (pour un échantillon)
bce_loss = 0
for i in range(len(y)):
    bce_loss += -(y[i] * sp.log(p_pred[i]) +
                  (1 - y[i]) * sp.log(1 - p_pred[i]))
bce_loss = bce_loss / m

# Gradient
```

BCE : Résultats du Gradient

Gradient analytique :

$$\nabla_{\mathbf{w}} L_{\mathsf{BCE}} = \frac{1}{m}\mathbf{X}^T(\hat{\mathbf{p}} - \mathbf{y})$$

où $\hat{\mathbf{p}} = [\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_m]^T$
Matrice Hessienne :

$$\mathbf{H}_{\mathsf{BCE}} = \frac{1}{m}\mathbf{X}^T\mathbf{D}\mathbf{X}$$

où $\mathbf{D} = \text{diag}(\hat{p}_1(1 - \hat{p}_1), \ldots, \hat{p}_m(1 - \hat{p}_m))$
Note : $\hat{p}_i(1 - \hat{p}_i) > 0$ pour $\hat{p}_i \in (0, 1)$

BCE : Convexité

Analyse de la Hessienne :

$$\mathbf{H}_{\mathsf{BCE}} = \frac{1}{m}\mathbf{X}^T\mathbf{D}\mathbf{X}$$

Matrice $\mathbf{D}$: - Diagonale avec $d_{ii} = \hat{p}_i(1 - \hat{p}_i) \geq 0$ - $\mathbf{D} \succeq 0$ (semi-définie positive)
Preuve de convexité :

$$\mathbf{v}^T\mathbf{H}_{\mathsf{BCE}}\mathbf{v} = \frac{1}{m}(\mathbf{X}\mathbf{v})^T\mathbf{D}(\mathbf{X}\mathbf{v}) \geq 0$$

Conclusion : BCE est **convexe** avec un unique minimum global.

CCE : Définition et Formulation

Formulation :

$$L_{\mathsf{CCE}}(\mathbf{W}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{i,k}\log(\hat{p}_{i,k})$$

où $\hat{p}_{i,k} = \frac{e^{z_{i,k}}}{\sum_{j=1}^{K} e^{z_{i,j}}}$ (fonction softmax)
et $z_{i,k} = \mathbf{w}_k^T \mathbf{x}_i$ (logit pour la classe k)
Utilisation : Classification multi-classes $(K > 2)$
Encodage : $\mathbf{y}_i$ en one-hot encoding

CCE : Calcul du Gradient par SymPy

Listing – Gradient de CCE

```
K = 3 # nombre de classes
W = sp.Matrix(sp.symbols('w1:10')).reshape(3, K) # matrice des poids
Y = sp.Matrix(sp.symbols('y1:13')).reshape(4, K) # one-hot encoding

# Fonction softmax
def softmax_sympy(z_vec):
    exp_z = sp.Matrix([sp.exp(zi) for zi in z_vec])
    sum_exp = sum(exp_z)
    return sp.Matrix([exp_zi / sum_exp for exp_zi in exp_z])

# Calcul des logits et probabilités
Z = X * W # logits pour chaque classe
P = sp.zeros(4, K)

for i in range(4): # pour chaque échantillon
    z_i = Z.row(i).T
```

CCE : Résultats du Gradient

Gradient analytique : Pour la classe k :

$$\nabla_{\mathbf{w}_k} L_{\mathsf{CCE}} = \frac{1}{m}\mathbf{X}^T(\hat{\mathbf{p}}_k - \mathbf{y}_k)$$

où $\hat{\mathbf{p}}_k$ et $\mathbf{y}_k$ sont les vecteurs de probabilités et labels pour la classe k.
Forme compacte :

$$\nabla_{\mathbf{W}} L_{\mathsf{CCE}} = \frac{1}{m}\mathbf{X}^T(\hat{\mathbf{P}} - \mathbf{Y})$$

Hessienne : Plus complexe, mais reste semi-définie positive.

CCE : Convexité

Propriété fondamentale : La fonction softmax avec entropie croisée est convexe par rapport aux logits.
Preuve intuitive : - La fonction $-\log(\cdot)$ est convexe - La fonction softmax préserve la convexité - La composition preserve la convexité
Hessienne :

$$\mathbf{H}_{\mathsf{CCE}} = \frac{1}{m}\mathbf{X}^T(\mathbf{D} - \hat{\mathbf{P}}\hat{\mathbf{P}}^T)\mathbf{X}$$

où $\mathbf{D}$ est diagonale avec $d_{ii} = \hat{p}_i(1 - \hat{p}_i)$.
Conclusion : CCE est **convexe**.

Huber : Définition et Formulation

Formulation :

$$L_{\textsf{Huber}}(\mathbf{w};\delta) = \frac{1}{m}\sum_{i=1}^{m} H_\delta(y_i - \hat{y}_i)$$

où :

$$H_\delta(r) = \begin{cases} \frac{1}{2}r^2 & \text{si } |r| \leq \delta \\ \delta|r| - \frac{1}{2}\delta^2 & \text{si } |r| > \delta \end{cases}$$

Paramètre δ : Seuil de transition entre comportement quadratique et linéaire.
Avantage : Robuste aux outliers (moins sensible que MSE).

Huber : Calcul du Gradient par SymPy

Listing – Gradient de Huber

```
delta = sp.symbols('delta', positive=True)

# Fonction de Huber
def huber_loss_sympy(r, delta):
    return sp.Piecewise(
        (sp.Rational(1, 2) * r**2, sp.Abs(r) <= delta),
        (delta * sp.Abs(r) - sp.Rational(1, 2) * delta**2, True)
    )

# Résidus
residuals = y - X * w

# Huber Loss totale
huber_total = sum([huber_loss_sympy(residuals[i], delta)
                   for i in range(len(residuals))]) / m

```

Huber : Résultats du Gradient

Gradient/Sous-gradient :

$$\frac{\partial H_\delta(r)}{\partial r} = \begin{cases} r & \text{si } |r| < \delta \\ \delta \cdot \text{sign}(r) & \text{si } |r| > \delta \\ [-\delta, \delta] & \text{si } r = \pm\delta \end{cases}$$

Gradient par rapport aux paramètres :

$$\nabla_{\mathbf{w}} L_{\text{Huber}} = \frac{1}{m}\mathbf{X}^T\mathbf{g}$$

où $g_i = \frac{\partial H_\delta(r_i)}{\partial r_i}$ avec $r_i = y_i - \mathbf{w}^T\mathbf{x}_i$

Huber : Convexité

Analyse de convexité :
Pour $|r| < \delta$ **:** $H_\delta(r) = \frac{1}{2}r^2$ est strictement convexe.
Pour $|r| > \delta$ **:** $H_\delta(r) = \delta|r| - \frac{1}{2}\delta^2$ est convexe (fonction linéaire par morceaux).
À $|r| = \delta$ **:** Les dérivées secondes à gauche et droite sont positives.
Hessienne généralisée : - Définie positive dans la région quadratique - Semi-définie positive globalement
Conclusion : La perte de Huber est **convexe** mais non différentiable aux points de transition.

Tableau Comparatif des Fonctions de Perte

Fonction	**Domaine**	**Convexité**	**Différentiabilité**	**Robustesse**
MSE	Régression	Convexe	C^∞	Faible
BCE	Classif. binaire	Convexe	C^∞	Moyenne
CCE	Classif. multi	Convexe	C^∞	Moyenne
Huber	Régression	Convexe	C^1 non C^2	Élevée

Points clés :

- Toutes les fonctions étudiées sont **convexes**
- Garantissent un **minimum global unique**
- Algorithmes de gradient convergent vers l'optimum
- Huber offre le meilleur compromis robustesse/efficacité

Applications Pratiques

Choix de la fonction de perte selon le contexte :

- **MSE** : Régression avec données propres, interprétation en termes de variance
- **BCE** : Classification binaire, probabilités calibrées
- **CCE** : Classification multi-classes, extension naturelle de BCE
- **Huber** : Régression robuste, présence d'outliers

Optimisation : - Gradient descent, Newton, quasi-Newton - Convergence garantie (fonctions convexes) - Choix du pas d'apprentissage critique

Conclusion

Ce que nous avons accompli :

1. **Gradients analytiques** calculés avec SymPy
2. **Propriétés de convexité** démontrées pour chaque fonction
3. **Comparaison** des caractéristiques et domaines d'application

Prochaines étapes :

- Implémentation et visualisation sur données réelles
- Comparaison empirique des performances
- Étude de l'influence des hyperparamètres (δ pour Huber)

Outils utilisés : SymPy pour le calcul symbolique, théorie de l'optimisation convexe.

Questions ?