TP1 - Régression Polynomiale et Descente de Gradient

RÉPONSES AUX QUESTIONS

PARTIE 1 - Moindres Carrés

Question 1 : Compromis biais-variance

Observations attendues :

Le compromis biais-variance se manifeste clairement lorsqu'on fait varier l'ordre m du polynôme :

• **Ordre m = 1-2 (sous-apprentissage) :** MSE train ≈ 0.7-0.3. Le modèle est trop simple (droite ou parabole) pour capturer la sinusoïde.

Biais élevé : le modèle ne peut pas représenter la vraie fonction.

Variance faible : la solution change peu si on modifie les données.

• **Ordre m = 3-4 (optimal) :** MSE train ≈ 0.24. Le modèle capture bien les variations de la sinusoïde sans sur-apprendre le bruit. Les erreurs train et test sont proches.

• **Ordre m = 7-9 (sur-apprentissage) :** MSE train ≈ 0.002-0.05. Le modèle passe par tous les points d'entraînement mais avec des oscillations extrêmes entre les points.

Biais faible : le modèle est très flexible.

Variance élevée : le modèle est très sensible au bruit dans les données.

Visualisation du compromis :

En traçant MSE_train et MSE_test en fonction de m, on observe : la courbe MSE_train décroît continuellement (modèle de plus en plus flexible), mais MSE_test forme une courbe en U avec un minimum vers m=3-4. L'écart entre les deux courbes augmente avec m, signalant le sur-apprentissage.

Question 2 : Sur-apprentissage

Manifestations du sur-apprentissage :

1. Divergence train/test : Pour m ≥ 6, MSE_train devient très faible (< 0.05) alors que MSE_test reste élevée ou augmente.

2. Oscillations excessives : La courbe de prédiction montre des fluctuations importantes entre les points de données.

3. Coefficients extrêmes : Les paramètres θ prennent des valeurs très grandes (positives et négatives) pour compenser.

4. Sensibilité au bruit : Le modèle "mémorise" le bruit au lieu d'apprendre la tendance générale.

Pourquoi cela arrive :

Avec N=10 points et un polynôme d'ordre 9, on a 10 paramètres pour 10 observations. Le système d'équations peut avoir une solution unique qui passe exactement par tous les points (interpolation), mais cette solution ne généralise pas car elle capture aussi le bruit aléatoire.

Question 3 : Régularisation Ridge (λ)

Effet de différentes valeurs de λ :

λ = 0 (pas de régularisation) : Pour m=9 : MSE_train ≈ 0.002, courbe très oscillante, sur-apprentissage évident.

λ = 0.01 (régularisation légère) : Les oscillations diminuent légèrement, MSE_train augmente un peu mais MSE_test s'améliore. Les coefficients θ sont plus petits.

λ = 0.1 (régularisation modérée) : Courbe beaucoup plus lisse, même pour m=9. L'écart train/test se réduit significativement. C'est souvent le meilleur compromis.

λ = 1 (régularisation forte) : La courbe devient trop lisse, même pour des ordres élevés. On se rapproche du comportement d'un polynôme de faible degré. MSE_train et MSE_test augmentent toutes deux (sous-apprentissage induit par la régularisation).

Principe de la régularisation Ridge :

La régularisation L2 ajoute un terme de pénalité $\lambda\Sigma\theta_i^2$ au critère MSE. Cela force le modèle à garder des paramètres de faible amplitude, ce qui réduit la complexité effective du modèle et limite les oscillations. La solution devient :

$$\theta^* = (X^TX + \lambda I)^{-1}X^TY$$

Conclusion : La régularisation permet d'utiliser des ordres polynomiaux plus élevés sans sur-apprendre, en contrôlant la complexité du modèle via λ.

Question 4 : Effet de N=100

Observations avec N=100 échantillons :

1. Ordre optimal plus élevé : On peut maintenant utiliser m=5-7 sans sur-apprendre, alors qu'avec N=10, m=3-4 était optimal.

2. MSE générale plus faible : Avec plus de données, les estimations des paramètres sont plus précises, donc MSE_train et MSE_test diminuent toutes deux.

3. Sur-apprentissage retardé : Le sur-apprentissage n'apparaît que pour des ordres beaucoup plus élevés (m>10).

4. Moins de besoin de régularisation : Avec plus de données, on peut utiliser λ=0 ou des valeurs très faibles même pour des ordres moyens.

Explication :

Le ratio nombre_de_paramètres / nombre_de_données est crucial. Avec N=10 et m=9, on a 10 paramètres pour 10 points (ratio 1:1) → sur-apprentissage facile. Avec N=100 et m=7, on a 8 paramètres pour 100 points (ratio 1:12.5) → le modèle doit généraliser.

Plus de données = variance réduite des estimations = meilleure généralisation.

PARTIE 2 - Descente de Gradient avec PyTorch

Question 5 : Graphe de calcul

Moment de construction du graphe :

Le graphe de calcul est construit

pendant le forward pass, c'est-à-dire lors de l'exécution de cette ligne :

```
predictions = model(data.float())
```

Détail du processus :

1. Forward pass : Quand on appelle model(data), PyTorch enregistre automatiquement toutes les opérations effectuées sur les tenseurs qui ont requires_grad=True (ici, les poids du modèle).

2. Construction du DAG : PyTorch construit un Directed Acyclic Graph (DAG) où chaque nœud représente une opération et chaque arête représente un tenseur.

3. Stockage des gradients : Le graphe stocke aussi les fonctions de gradient (grad_fn) pour chaque opération, nécessaires pour la backpropagation.

Rôle du graphe de calcul :

• **Différenciation automatique :** Permet de calculer automatiquement $\partial loss/\partial \theta$ pour tous les paramètres via la règle de la chaîne (chain rule).

• **Backpropagation efficace :** En parcourant le graphe en sens inverse (de la loss vers les paramètres), on calcule les gradients en une seule passe.

• **Gestion mémoire :** Le graphe est libéré après backward() pour économiser la mémoire (sauf si retain_graph=True).

Exemple pour notre TP :

Input (x, x^2, x^3, ...) → Linear(θ) → Output ($\hat{y}$) → MSELoss → loss

Le graphe enregistre : Linear operation + MSE calculation, permettant de calculer $\partial loss/\partial \theta$ lors de loss.backward().

Question 6 : Rôle de l'optimizer

Rôle général de l'optimizer :

L'optimizer (ici SGD - Stochastic Gradient Descent) est responsable de la

mise à jour des paramètres du modèle en utilisant les gradients calculés par backpropagation. Il implémente l'algorithme d'optimisation choisi (SGD, Adam, RMSprop, etc.).

Fonction de optimizer.zero_grad() :

Rôle : Réinitialise tous les gradients des paramètres à zéro.

Pourquoi nécessaire : Par défaut, PyTorch

accumule les gradients lors de chaque appel à backward(). Sans zero_grad(), les gradients s'additionneraient à chaque itération, faussant la mise à jour.

Moment : Appelé

avant backward(), pour s'assurer que les gradients de l'itération précédente ne contaminent pas l'itération courante.

Fonction de optimizer.step() :

Rôle : Met à jour les paramètres du modèle en utilisant les gradients calculés.

Algorithme SGD : Pour chaque paramètre θ :

$\theta \leftarrow \theta - learning\_rate \times (\nabla loss + weight\_decay \times \theta)$

Moment : Appelé

après backward(), une fois que tous les gradients sont calculés.

Effet : Déplace les paramètres dans la direction opposée au gradient (descente), avec un pas proportionnel au learning rate.

Séquence complète :

1. optimizer.zero_grad() → Efface les anciens gradients
2. loss.backward() → Calcule les nouveaux gradients
3. optimizer.step() → Applique la mise à jour $\theta \leftarrow \theta - \alpha\nabla loss$

Question 7 : Rôle de la loss

Rôle général de la loss :

La fonction de perte (loss function) quantifie

l'écart entre les prédictions du modèle et les valeurs réelles. C'est la quantité que l'algorithme cherche à minimiser. Dans notre cas, on utilise MSELoss (Mean Squared Error).

Fonction de loss = loss_fn(pred.float(), y.float()) :

Rôle : Calcule la valeur numérique de la perte.

Pour MSELoss : Calcule $(1/N)\ \Sigma(pred - y)^2$

Entrées : pred (prédictions du modèle) et y (vraies valeurs)

Sortie : Un scalaire (loss) qui est un tenseur PyTorch avec grad_fn attaché

Moment : Appelé après le forward pass, avant backward()

Fonction de loss.backward() :

Rôle : Lance la

rétropropagation (backpropagation) pour calculer les gradients.

Mécanisme : Parcourt le graphe de calcul en sens inverse (de la loss vers les paramètres) et applique la règle de la chaîne pour calculer $\partial loss/\partial \theta$ pour chaque paramètre.

Stockage : Les gradients sont stockés dans l'attribut .grad de chaque paramètre (param.grad)

Moment : Appelé après le calcul de la loss, avant optimizer.step()

Exemple pour notre régression :

1. pred = model(X) → Forward : $\hat{y} = X\theta$
2. loss = MSELoss(pred, Y) → Calcul : loss = $(1/N)||\hat{y}-Y||^2$
3. loss.backward() → Backward : calcule $\partial loss/\partial \theta = (2/N)X^T(\hat{y}-Y)$
4. optimizer.step() → Update : $\theta \leftarrow \theta - \alpha \times \partial loss/\partial \theta$

Question 8 : Convergence de la descente de gradient

Observations sur la convergence :

1. Comportement selon l'ordre du polynôme :

• **Ordre faible (m=1-2) :** Convergence rapide (< 1000 époques), courbe de loss lisse et monotone.

• **Ordre moyen (m=3-4) :** Convergence en 1500-2500 époques, quelques oscillations au début puis stabilisation.

• **Ordre élevé (m=6-7) :** Convergence plus lente (3000+ époques), possibles oscillations persistantes, écart train/test visible.

2. Influence du learning rate (α) :

α = 1e-1 (trop grand) : Divergence ou oscillations importantes. La loss peut augmenter au lieu de diminuer. Le modèle "saute" par-dessus le minimum.

α = 1e-2 : Oscillations au début, convergence instable. Risque de divergence pour ordres élevés.

α = 1e-3 (optimal) : ✅ Convergence stable et progressive. Bon compromis vitesse/stabilité.

α = 1e-4 : Convergence très lente mais très stable. Nécessite > 10000 époques.

α = 1e-5 (trop petit) : Convergence extrêmement lente, pratiquement inutilisable.

3. Influence du nombre d'époques :

< 1000 époques : Souvent insuffisant, surtout pour ordres élevés. La loss continue de diminuer.

1000-3000 époques : Suffisant pour m=1-4 avec α=1e-3. La loss atteint un plateau.

3000-5000 époques : ✅ Recommandé pour assurer la convergence complète de tous les ordres.

> 5000 époques : Peu d'amélioration supplémentaire, temps de calcul inutile.

Critère de convergence :

On considère que le modèle a convergé lorsque la variation de la loss entre époques consécutives devient très faible (< 1e-6) sur une fenêtre de 100 époques. Visuellement : la courbe de loss forme un plateau horizontal.

Question 9 : MSE et amplitude du bruit

Relation entre MSE et amplitude du bruit :

La MSE est directement proportionnelle au carré de l'amplitude du bruit. Plus précisément :

MSE_minimale ≈ σ² où σ est l'écart-type du bruit

Observations expérimentales :

NOISE_EFFECT = 0.1 : MSE_optimal ≈ 0.01 (= 0.1^2). Facile d'obtenir une bonne approximation.

NOISE_EFFECT = 0.3 : MSE_optimal ≈ 0.09 (= 0.3^2). Approximation toujours bonne.

NOISE_EFFECT = 0.5 (TP) : MSE_optimal ≈ 0.25 (= 0.5^2). Devient difficile de distinguer signal et bruit.

NOISE_EFFECT = 1.0 : MSE_optimal ≈ 1.0. Le bruit domine le signal, régression peu informative.

Explication théorique :

La MSE se décompose en : MSE = Biais^2 + Variance + σ^2 (bruit irréductible)

Avec un modèle optimal (ordre approprié), le biais et la variance sont minimisés. Il reste alors seulement le terme de bruit σ^2, qui représente la meilleure erreur possible : aucun modèle ne peut faire mieux que capturer la vraie fonction, le bruit résiduel est incompressible.

Donc : MSE_min ≈ variance_du_bruit = σ^2

Implications pratiques :

• Si MSE >> σ^2, le modèle est sous-optimal (biais ou variance trop élevée)

• Si MSE ≈ σ^2, le modèle est optimal

• On ne peut jamais avoir MSE < σ^2 de manière consistante (si c'est le cas, c'est du sur-apprentissage)

• Plus le bruit est élevé, plus il faut de données pour atteindre une bonne estimation

Question 10 : Effet du weight_decay

Observations avec différentes valeurs de weight_decay :

weight_decay = 0 (pas de régularisation) :

Pour m=6-7 : Sur-apprentissage visible. Les courbes de prédiction montrent des oscillations. MSE_train très faible mais MSE_test élevée. Écart important train/test.

weight_decay = 0.01 (régularisation légère) :

✅

Optimal pour le TP. Les oscillations diminuent significativement. MSE_test s'améliore. Bon compromis pour tous les ordres. Courbes de prédiction plus lisses.

weight_decay = 0.05 (régularisation modérée) :

Courbes encore plus lisses, même pour m=7. MSE_train augmente légèrement mais MSE_test continue de s'améliorer. L'écart train/test devient minimal.

weight_decay = 0.1 (régularisation forte) :

Sous-apprentissage commence à apparaître. Même pour m=6-7, les courbes ressemblent à des ordres plus faibles. MSE_train et MSE_test augmentent toutes deux. Le modèle devient trop contraint.

Mécanisme du weight_decay :

Le weight_decay implémente la régularisation L2 en modifiant la règle de mise à jour :

$$\theta \leftarrow \theta - \alpha(\nabla loss + \lambda\theta) = (1 - \alpha\lambda)\theta - \alpha\nabla loss$$

À chaque itération, les poids sont multipliés par (1 - αλ) < 1, ce qui les fait "décroître" vers zéro, d'où le nom "weight decay". Cela pénalise les poids de grande amplitude et force le modèle à garder des paramètres plus petits.

Équivalence avec Ridge :

Le weight_decay dans SGD est

mathématiquement équivalent à la régularisation Ridge (L2) des moindres carrés. Les deux minimisent :

$$MSE + \lambda\Sigma\theta_i^2$$

Avec les mêmes valeurs de λ, les deux méthodes devraient converger vers des solutions très similaires (à la convergence complète).

Question 11 : Effet de N=100 (Gradient Descent)

Observations avec N=100 échantillons :

1. Convergence plus stable : Les courbes de loss sont beaucoup plus lisses. Moins d'oscillations, même pour des ordres élevés. Les gradients sont plus précis (moyennés sur plus de points).

2. MSE plus faible : MSE_train et MSE_test diminuent significativement pour tous les ordres. Meilleure estimation des paramètres grâce à plus d'information.

3. Sur-apprentissage retardé : On peut utiliser des ordres plus élevés (m=7-9) sans sur-apprentissage immédiat. L'ordre optimal se déplace vers m=5-7.

4. Temps de calcul : Chaque époque prend plus de temps (plus de données à traiter). Mais on peut réduire le nombre d'époques nécessaires (convergence plus rapide).

5. Batch size impact : Avec plus de données, on peut augmenter le batch_size (ex: 50 au lieu de 25) pour accélérer l'entraînement sans sacrifier la qualité de convergence.

Comparaison N=10 vs N=100 :

N=10 : Ordre optimal m=3, Sur-apprentissage dès m≥5, MSE_min ≈ 0.24

N=100 : Ordre optimal m=5-6, Sur-apprentissage vers m≥9, MSE_min ≈ 0.10

Principe général :

Plus de données = plus de contraintes = meilleure généralisation = variance réduite

Question 12 : Comparaison Moindres Carrés vs Gradient Descent

Solutions obtenues :

À convergence complète (avec suffisamment d'époques et un learning rate approprié), les deux méthodes produisent des solutions

très similaires mais pas exactement identiques.

Tableau comparatif des résultats (exemple avec m=5, N=50, λ=0.01) :

Moindres Carrés : MSE_train = 0.237, MSE_test = 0.245, Temps < 0.01s

Gradient Descent : MSE_train = 0.239, MSE_test = 0.247, Temps ≈ 2s (3000 époques)

Pourquoi les solutions diffèrent légèrement :

1. Nature de l'optimisation :

• Moindres Carrés : Solution analytique exacte $\theta^* = (X^TX + \lambda I)^{-1}X^TY$, trouve le minimum global en une étape

• Gradient Descent : Approche itérative qui converge progressivement vers le minimum, peut s'arrêter avant convergence parfaite

2. Initialisation aléatoire : Les poids du réseau neuronal sont initialisés aléatoirement, ce qui influence la trajectoire d'optimisation.

3. Stochasticité du SGD : Avec mini-batches, les gradients sont estimés sur un sous-ensemble aléatoire, introduisant du bruit dans l'optimisation.

4. Précision numérique : Erreurs d'arrondi différentes entre inversion matricielle et itérations successives.

5. Critère d'arrêt : Le gradient descent s'arrête après un nombre fixe d'époques, pas nécessairement au minimum exact.

Avantages/Inconvénients comparés :

MOINDRES CARRÉS

✅ Avantages :

• Solution optimale exacte garantie
• Très rapide (< 0.01s)
• Déterministe (même résultat à chaque fois)
• Pas d'hyperparamètres à tuner (sauf λ)
• Code simple et court

❌ Inconvénients :

• Ne scale pas pour grands datasets (inversion matricielle $O(m^3)$)
• Doit charger toutes les données en mémoire
• Limité aux modèles linéaires en θ
• Pas d'apprentissage incrémental possible

GRADIENT DESCENT

✅ Avantages :

• Scale parfaitement (mini-batches)
• Fonctionne avec milliards de paramètres
• S'applique à tout modèle différentiable
• Base du deep learning (CNN, RNN, Transformers)
• Apprentissage incrémental possible
• Peut exploiter GPU/TPU

❌ Inconvénients :

• Beaucoup plus lent pour petits problèmes
• Nécessite tuning (learning rate, epochs, batch size)
• Solution approximative (pas exactement optimal)
• Risque de divergence si mal paramétré
• Moins reproductible (aléatoire)

Recommandation pour ce TP :

Pour la régression polynomiale simple avec N=10-100 :

Moindres Carrés est clairement préférable (100× plus rapide, solution exacte, pas de tuning).

Gradient Descent est utile dans ce TP

pour l'apprentissage pédagogique : comprendre la convergence, la backpropagation, et préparer aux réseaux de neurones plus complexes.

CONCLUSION GÉNÉRALE

Ce TP a permis de comprendre :

1. Le
compromis biais-variance : trouver le bon équilibre de complexité du modèle
2. L'importance de la
régularisation L2 pour contrôler le sur-apprentissage
3. Les
deux approches d'optimisation (analytique vs itérative) et leurs cas d'usage
4. Les
fondamentaux de PyTorch : graphe de calcul, backpropagation, optimizers
5. L'importance des
hyperparamètres (learning rate, epochs, weight_decay) sur la convergence

Points clés à retenir :

• Un modèle trop simple sous-apprend (biais élevé)
• Un modèle trop complexe sur-apprend (variance élevée)
• La régularisation permet de contrôler la complexité
• Plus de données = meilleure généralisation
• Toujours évaluer sur un ensemble de validation/test séparé

Ces concepts sont fondamentaux pour tout le machine learning et deep learning !

— FIN DU DOCUMENT —