29/10/2023

Perceptron, Logistic and Softmax regression

Advanced Neural Net

Ramiqi Andi

Table des matières

1. Introduction

Ce travail se penche sur deux pierres angulaires de l'apprentissage supervisé : le perceptron et la régression logistique. Bien que ces techniques semblent basiques en comparaison des réseaux de neurones profonds modernes, elles demeurent fondamentales, car elles forment la base sur laquelle repose bon nombre de méthodologies plus avancées.

Dans un premier temps, nous explorerons le perceptron, un algorithme simple mais puissant. Inspiré par les neurones biologiques, le perceptron est l'une des premières architectures d'apprentissage automatique. Nous nous plongerons dans les détails de son fonctionnement, de la propagation avant à la mise à jour des poids en passant par le calcul des gradients.

Dans la seconde section, nous aborderons la régression logistique et le classificateur Softmax avec une régularisation de norme L2. Ces techniques, bien qu'elles soient plus sophistiquées que le perceptron, reposent sur des principes similaires. La régression logistique est notamment utilisée pour les problèmes de classification binaire, tandis que le classificateur Softmax s'étend aux classifications multi-classes. La régularisation, quant à elle, est un élément crucial pour prévenir le surapprentissage et garantir la généralisation du modèle.

2. Perceptron

Le perceptron est un classificateur linéaire binaire, ce qui signifie qu'il est destiné à séparer des points de données en deux classes distinctes à l'aide d'une décision linéaire. Dans ce rapport, nous mettrons en pratique l’algorithme sur le jeu de donnée « Iris », mais nous travaillerons avec seulement 2 attributs à des fins de visualisations.

Fonctionnement du Perceptron

À sa base, le perceptron est un classificateur linéaire binaire, c'est-à-dire qu'il est conçu pour classer des données en deux catégories. Il prend un vecteur d'entrée « x », le multiplie par un ensemble de poids « w » et passe ensuite la somme résultante à travers une fonction d'activation pour produire une sortie.

L’apprentissage avec un perceptron consiste à ajuster les poids w afin de classer correctement les données d’entrainement. Cet ajustement se fait de manière itérative.

Remember that perceptron learns a set of weights

$$\mathbf{w_t} = (w1, ..., wd, w_{d+1})$$

and using these weights produces the thresholded output

$$sign(\mathbf{x_i w_t})$$

The error function of perceptron is:

$$E(\mathbf{w}) = -\sum_{\mathbf{x_i} \in M} (\mathbf{x}_i \mathbf{w}) y_i$$

where M is the set of missclassified instances and y_i is the class associated to the sample i.

The gradient of $E(\mathbf{w})$ is:

$$\nabla E(\mathbf{w}) = \sum_{\mathbf{x_i} \in M} \mathbf{x_i} y_i = -\mathbf{x}^T (y - sign(\mathbf{xw}))$$

The gradient descent direction is $-\nabla E(\mathbf{w}) = \mathbf{x}^T (y - sign(\mathbf{xw}))$ and we update the weight matrix $\mathbf{w}$ in the gradient descent direction by $\Delta \mathbf{w} = -\alpha \nabla E(\mathbf{w})$, thus:

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w} = \mathbf{w} + \alpha \mathbf{x}^T (y - sign(\mathbf{xw}))$$

where α is the learning rate.

Le perceptron apprend un ensemble de poids pour produire une sortie seuillée via la formule sign(…). Son erreur est déterminée par la somme des instances mal classées. Lors de l'apprentissage, on utilise ensuite la descente de gradient pour mettre à jour les poids. Le « Learning rate » représente la taille des étapes prises lors de l'ajustement des poids au cours de l'entraînement.

Choix du Batch

Lors de l'entraînement d'un modèle d'apprentissage automatique, en particulier les réseaux de neurones, la manière dont nous présentons les données au modèle peut grandement influencer la vitesse de convergence, la qualité du modèle final, et la consommation des ressources.

1. Online Learning

Dans cette approche, le modèle est mis à jour pour chaque exemple individuel d'entraînement. Cela signifie que pour un ensemble de données de taille N, nous aurons N mises à jour de modèle pour une époque. L'apprentissage en ligne peut être rapide et adaptatif aux changements dans les données, mais il est souvent associé à une convergence instable, car chaque exemple peut introduire beaucoup de variance dans les mises à jour.

2. Full-batch Learning

Ici, l'algorithme calcule le gradient de l'erreur sur l'ensemble du jeu de données avant d'effectuer une mise à jour des paramètres. Cela signifie que pour une époque, il n'y a qu'une seule mise à jour du modèle. Cette méthode offre une direction de mise à jour stable, car elle est basée sur l'erreur de toutes les instances. Toutefois, elle peut être très coûteuse en termes de mémoire et de temps de calcul pour des grands

ensembles de données, et elle est moins flexible face aux changements dynamiques dans les données.

3. **Mini-batch Learning**

Cette approche cherche à combiner le meilleur des deux précédents. Au lieu de mettre à jour le modèle pour chaque exemple (comme en apprentissage en ligne) ou pour l'ensemble du jeu de données (comme en full-batch), nous le mettons à jour pour un petit sous-ensemble ou "mini-batch" à la fois. La taille typique d'un mini-lot peut varier de 32 à quelques centaines d'exemples. L'apprentissage par mini-lots offre un équilibre entre l'efficacité computationnelle (exploitant souvent les opérations matricielles parallèles sur du matériel moderne) et une convergence plus stable que l'apprentissage en ligne.

Limitations

Bien que le perceptron soit un outil puissant, il présente certaines limitations. Notamment, il ne peut traiter que des données linéairement séparables. Cela signifie que si les deux classes ne peuvent être séparées par une droite (ou un hyperplan dans des dimensions plus élevées), le perceptron ne pourra pas apprendre une fonction de décision correcte, peu importe combien d'itérations d'entraînement sont effectuées.

Mise en place de Perceptron

1. **Forward pass: compute scores**
   Le "forward pass" pour le calcul des scores consiste à prendre les entrées et à les propager à travers le perceptron pour obtenir les scores bruts avant activation.

2. **Forward pass: compute loss**
   Après avoir obtenu les scores, on calcule la perte (ou l'erreur) en comparant ces scores aux valeurs attendues, ce qui nous indique à quel point le perceptron est performant.

3. **Backward pass: compute gradients**
   Le "backward pass" pour le calcul des gradients détermine comment chaque poids contribue à l'erreur globale, en calculant les dérivées partielles de la perte par rapport à ces poids.

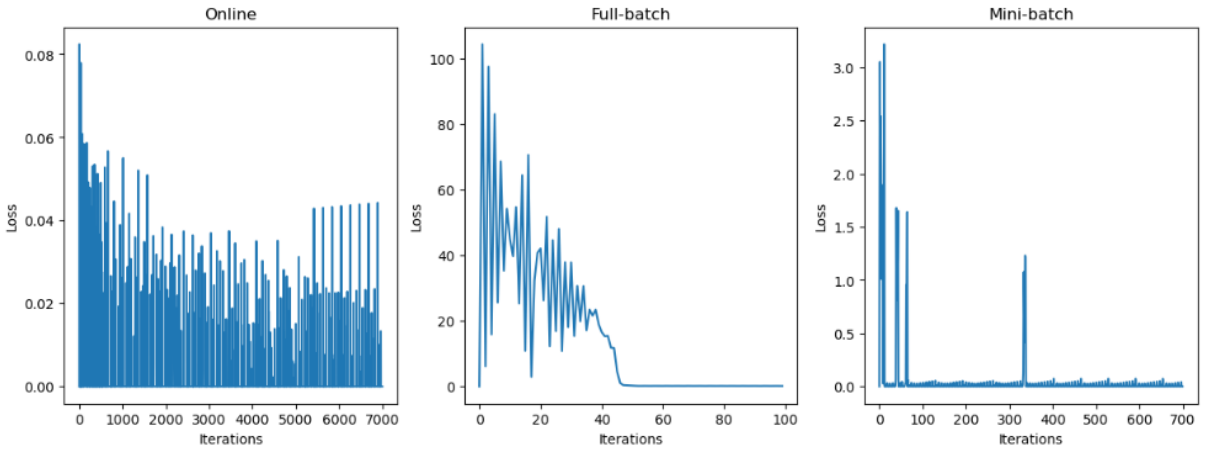
4. **Backward pass: update**
   Une fois les gradients calculés, on ajuste les poids du perceptron dans la direction opposée à ces gradients pour minimiser l'erreur.

5. **Training**
   L'entraînement implique de répéter les étapes du "forward pass" et du "backward pass" à travers plusieurs itérations (ou époques) pour affiner les poids du perceptron.

6. Visualisations

Les visualisations nous permettront d'offrir une représentation graphique des poids, des scores, de la perte ou de toute autre métrique pertinente, permettant une meilleure compréhension et interprétation des performances et de l'évolution du perceptron.

Visualisations de la perte



Une "itération" fait référence à une mise à jour unique des poids d'un modèle après avoir vu un ensemble spécifique de données. La quantité d'itérations diffère d'un graphique à l'autre en raison des différentes stratégies de batch :

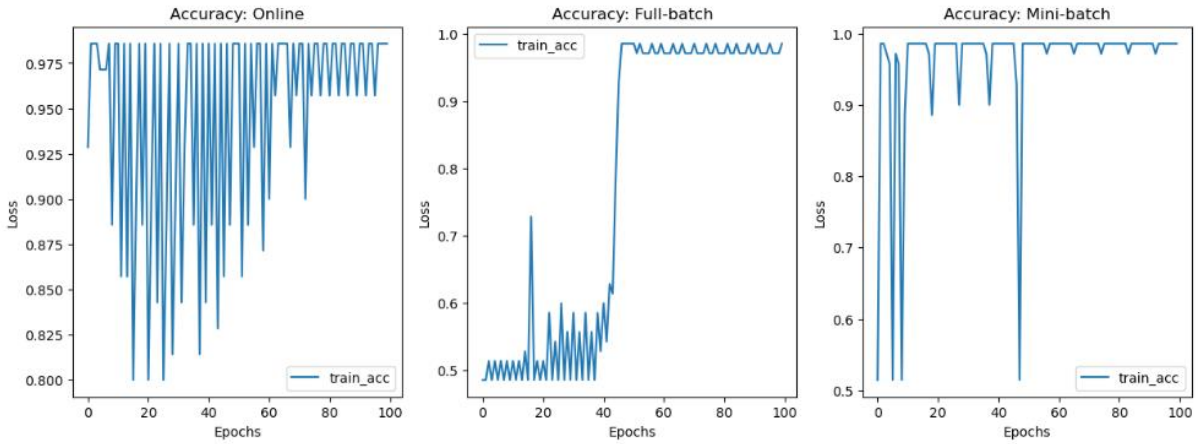
1. Online : Une itération correspond à une mise à jour après avoir vu une seule donnée. Pour un ensemble de données de taille N, N itérations constituent une époque. Ceci explique le grand nombre d'itérations sur ce graphique.

2. Full-batch : Une itération équivaut à une mise à jour après avoir vu toutes les données. Donc, pour chaque époque, il n'y a qu'une seule itération. C'est pourquoi le nombre d'itérations dans le graphique Full-batch est bien inférieur.

3. Mini-batch : Une itération correspond à une mise à jour après avoir vu un sous-ensemble (ou mini-batch) de données. Le nombre d'itérations par époque dépend de la taille du mini-batch. Le nombre total d'itérations se situe généralement entre l'apprentissage en ligne et le Full-batch.

Quant à la différence avec les "époques", une époque fait référence au processus où l'ensemble de données complet a été présenté au modèle une fois. Par conséquent, le nombre d'itérations nécessaires pour compléter une époque varie en fonction de la stratégie de batch utilisée.

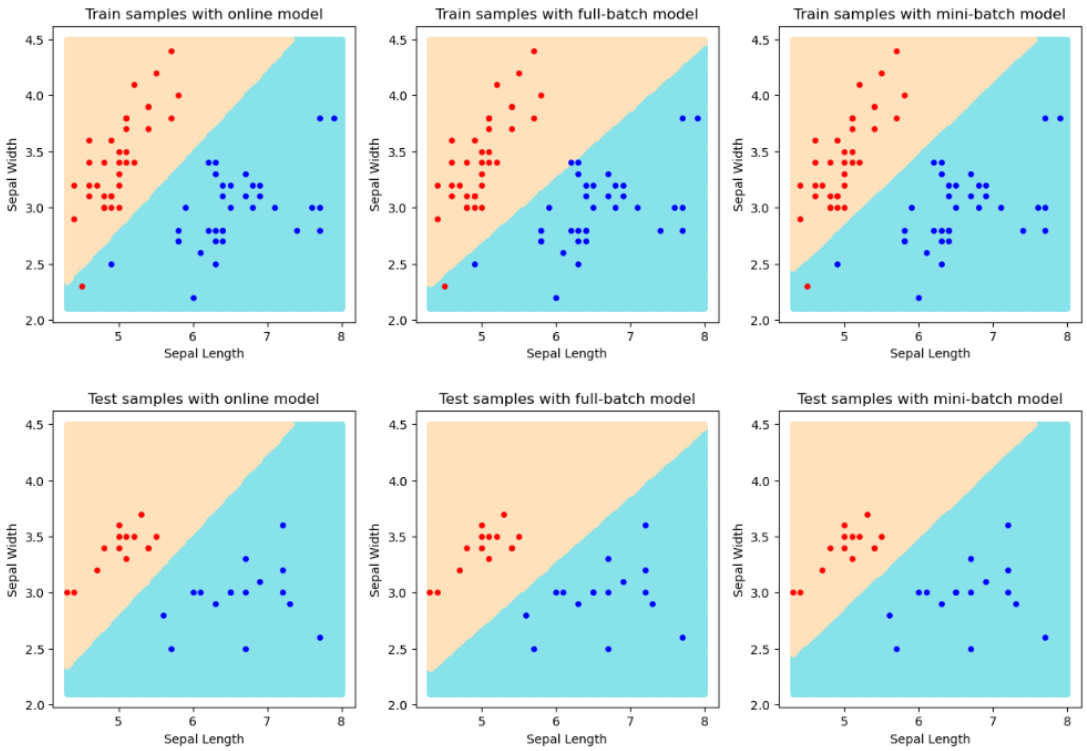
Visualisation de la précision



Les trois approches (Online, Full-batch et Mini-batch) présentent des comportements différents en raison de la manière dont elles traitent les données lors de la mise à jour des poids. Le cœur de la différence réside dans l'équilibre entre la fréquence des mises à jour des poids et la représentativité de l'erreur utilisée pour ces mises à jour. L'approche Online privilégie la fréquence, le Full-batch privilégie la précision globale, et le Mini-batch tente de trouver un équilibre entre les deux.

Selon ces observations, le modèle "Mini-batch" semble être le meilleur compromis. Il allie la rapidité de convergence du Full-batch à la précision élevée de l'Online, tout en évitant les pièges majeurs de ces deux approches.

Testing



Les graphiques illustrent comment nos trois différents modèles d'apprentissage - Online, Full-batch, et Mini-batch - classifient des données en se basant sur les caractéristiques 'Sepal Length' (Longueur du Sépale) et 'Sepal Width' (Largeur du Sépale).

1. Surfaces :

Les régions colorées dans chaque sous-graphique représentent les frontières de décision établies par chaque modèle. Ces frontières de décision délimitent les régions où le modèle prédit une certaine classe pour n'importe quel point de données donné qui tombe dans cette région. Ainsi, si un nouveau point de données était tracé à l'intérieur d'une région colorée spécifique, le modèle le classerait en fonction de la couleur de cette région.

2. Ligne entre les Surfaces :

La ligne séparant les deux surfaces colorées est la frontière de décision exacte où le modèle estime que la probabilité de chaque classe est la même. C'est le seuil où le modèle bascule sa prédiction d'une classe à l'autre.

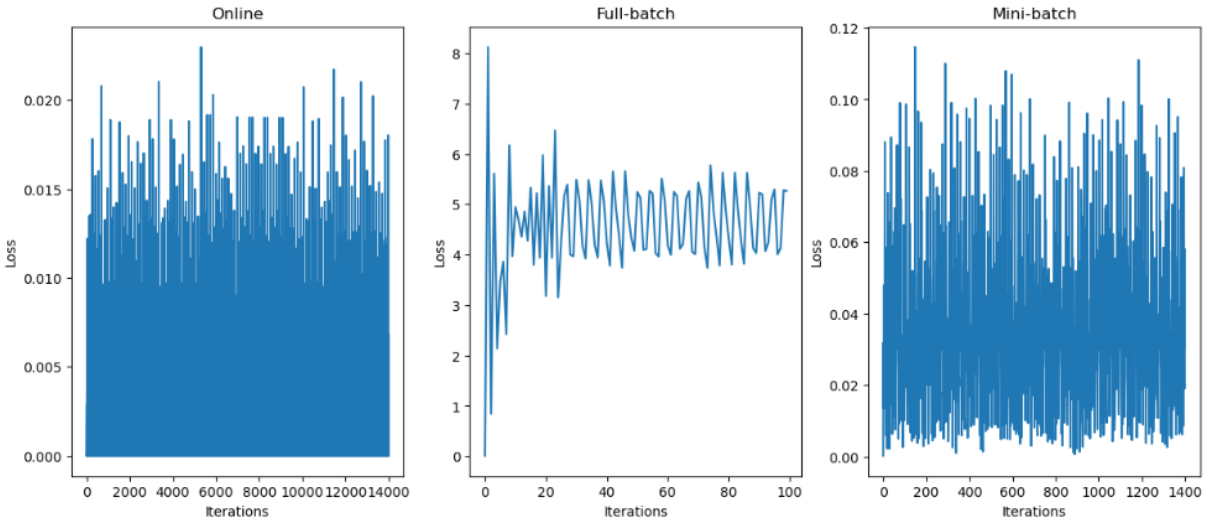
3. Points sur les Graphiques :

Les points représentés sur chaque sous-graphique symbolisent les échantillons de données. Il s'agit des instances de « Setosa » représentées par les caractéristiques 'Sepal Length' et 'Sepal Width'.

4. Couleurs :

Les différentes couleurs utilisées, à savoir le rouge et le bleu, représentent les deux classes différentes dans les données. Sur ces graphiques, chaque couleur correspond à une classe spécifique, et la couleur de chaque point indique la classe réelle de cet échantillon. De plus, la couleur des surfaces illustre la prédiction du modèle pour cette région.

Non separable problem



Pour le perceptron, qui est un modèle linéaire, cela pose un problème. Le perceptron tente de trouver le meilleur hyperplan qui sépare les données de différentes classes. Si les données ne peuvent pas être séparées linéairement, le perceptron va continuer à ajuster ses poids indéfiniment en essayant de minimiser l'erreur, sans jamais atteindre une solution parfaite.

Ces graphiques montrent que, malgré les différentes méthodes d'apprentissage, le perceptron peine à trouver une solution stable et optimale pour ce problème. Les fluctuations continuelles de la perte soulignent la difficulté à séparer linéairement les données.

3. Logistic Regression

La régression logistique est une méthode statistique couramment utilisée pour modéliser la probabilité d'un résultat binaire en fonction d'une ou de plusieurs variables indépendantes. Voici un aperçu de la régression logistique binaire. L'objectif principal est d'estimer la probabilité qu'une observation appartienne à l'une des deux catégories en fonction des variables indépendantes.

Les coefficients sont estimés en utilisant la méthode de la vraisemblance maximale. L'idée est de trouver les coefficients qui maximisent la probabilité (ou la vraisemblance) des observations réelles données.

Pour cette deuxième partie, le jeu de donée « Cifar10 » sera utilisé. Dans un premier temps, seul deux classes seront prises en compte pour la classification binaire, puis lorsque SoftMax sera utilisé, toutes les classes du dataset seront représentées.

Fonction coût :

$$\mathrm{E(w)} = -\sum_{i=i}^{N} y_i \ln \sigma(w^T x_i) - \sum_{i=i}^{N} (1 - y_i) \ln (1 - \sigma(w^T x_i))$$

Que l'on va chercher à minimiser avec une estimation de la vraisemblance maximale

Le gradient sera :

$$\nabla \mathrm{E(w)} = \sum_{i=i}^{N} (\sigma(w^T x_i) - y_i) x_i$$

- Yi : Il s'agit du label pour une instance i qui peut être 0 ou 1
- Xi : C'est le vecteur pour une instance i de notre jeu de donnée
- w : C'est le vecteur de poids du modèle
- $w^T x_i$: C'est le produit scalaire entre les poids w et le vecteur x qui nous donnera un score brut et qui sera ensuite transformé en une probabilité à l'aide de la fonction sigmoïde.
- $\sigma(w^T x_i)$: C'est la fonction sigmoïde définie tel que : $\sigma(w^T x_i) = \frac{1}{1+ e^{-(w^T x_i)}}$

Cout de la régression logistique avec régularisation L2 :

La régularisation L2 est souvent utilisée pour éviter le surajustement en pénalisant les poids qui sont trop grands. Elle est ajoutée à la fonction de coût d'origine.

$$\mathrm{E(w)} = -\sum_{i=i}^{N} y_i \ln \sigma(w^T x_i) - \sum_{i=i}^{N} (1 - y_i) \ln (1 - \sigma(w^T x_i)) + \lambda \sum_{j=i}^{M} \mathrm{w}_j^2$$

Où :

- λ est le coefficient de régularisation. C'est un hyperparamètre défini avant l'entraînement.
- Wj représente chaque composant du vecteur de poids w
- M est le nombre de caractéristiques (ou de dimensions) dans Xi

Gradient de la régression logistique avec régularisation L2 :

Le gradient de la fonction de coût précédente (avec régularisation L2) par rapport à w est :

$$\nabla \mathrm{E}(\mathrm{w}) = \sum_{i=i}^{N} (\sigma(w^T x_i) - y_i) x_i + 2\lambda w$$

La première partie provient du gradient de la fonction de coût d'origine, la deuxième est simplement le gradient de la régularisation (dérivé de $\lambda \mathrm{w}_j^2$ par rapport à wj)

Implémentation

Comme pour le perceptron, nous mettrons en place les différentes étapes dans notre code, à savoir :

Calculer les scores, calculer la loss, calculer la dérivée de la loss et prédire la classe.

Pour la partie test, nous effectuons ensuite une validation croisée pour trouver les meilleurs hyperparamètres du model. Les hyperparamètres utilisés sont le taux d'apprentissage (learning rate) et la force de régularisation (regularization strength) :

Pour chaque combinaison de taux d'apprentissage (« lr ») et de force de régularisation (« reg »), le code entraîne un modèle sur l'ensemble d'entraînement et évalue sa précision sur l'ensemble de validation. Si la précision sur l'ensemble de validation est meilleure que la précision du "meilleur modèle" précédemment observée, alors ce modèle est mis à jour comme étant le meilleur modèle. L'historique des pertes pour le meilleur modèle est également mis à jour.

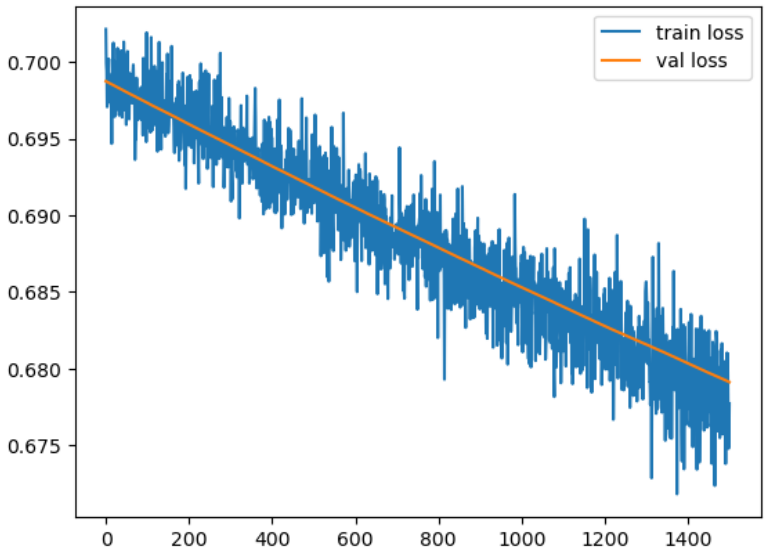
Le taux d'apprentissage (learning rate) et le régularisateur jouent tous deux des rôles cruciaux dans cet entraînement :

- Le learning rate détermine la taille des pas que l'algorithme d'optimisation fait pour atteindre un minimum (dans le cas de la minimisation de la perte). Un learning rate optimal permet à l'optimisation de converger rapidement vers un minimum sans "sauter" par-dessus, ni se coincer dans les minimas locaux.

- La régularisation est une technique utilisée pour prévenir le surapprentissage (overfitting). Elle pénalise les poids trop élevés en ajoutant un coût à la fonction de perte. Une régularisation optimale permet au modèle d'équilibrer la

complexité du modèle et la capacité d'ajustement, ce qui donne de bonnes performances à la fois sur les ensembles d'entraînement et de test.

Dans mon cas, la précision du model semble stagner à 50%, alors que la perte diminue progressivement.



La graphique illustre l'évolution de la perte (loss) du modèle pendant l'entraînement, à la fois pour l'ensemble d'entraînement (`train loss`) et l'ensemble de validation (`val loss`).

Tant la perte d'entraînement que la perte de validation semblent diminuer au fil des itérations, indiquant que le modèle apprend et s'améliore au fur et à mesure de l'entraînement. De plus, les deux pertes sont assez proches l'une de l'autre, suggérant ainsi que le modèle ne surapprend pas (overfitting) fortement à l'ensemble d'entraînement.

La perte d'entraînement et la perte de validation sont assez proches l'une de l'autre. C'est généralement un signe positif, car cela suggère que le modèle ne surapprend pas (overfitting) fortement à l'ensemble d'entraînement.

4. Regression Logistique : SoftMax

Lorsque nous avons plus de deux catégories, la régression logistique binaire ne suffit plus. Dans ce cas, nous utilisons la régression logistique multinomiale avec la fonction softmax.

Fonction de coût :

Soit « C » le nombre total de classes. La fonction de coût pour la régression logistique multinomiale (ou fonction softmax) est généralement la cross-entropy catégorielle. Elle est donnée par :

$$E(W) = -\sum_{i=1}^{N}\sum_{k=1}^{C} y_{ik} \ln\left(\frac{e^{W_k^T x_i}}{\sum_{j=1}^{C} e^{W_j^T x_i}}\right)$$

- y_{ik} : C'est 1 si l'observation i appartient à la classe k, et 0 sinon.
- W_k^T: C'est le vecteur de poids pour la classe k
- x_i : C'est le vecteur de caractéristiques pour l'observation i.

Gradient pour Softmax :

Le gradient de la fonction de coût par rapport au vecteur de poids pour la classe k est donné par :

$$\nabla E(W_k) = \sum_{i=1}^{N} (p_{ik} - y_{ik}) x_i$$

Où p_{ik} est la probabilité prédite que l'observation i appartienne à la classe k, donnée par :

$$p_{ik} = \frac{e^{W_k^T x_i}}{\sum_{j=1}^{C} e^{W_j^T x_i}}$$

La fonction de coût cherche à pénaliser les mauvaises prédictions. Si la probabilité prédite pour la vraie classe est faible, le coût sera élevé.

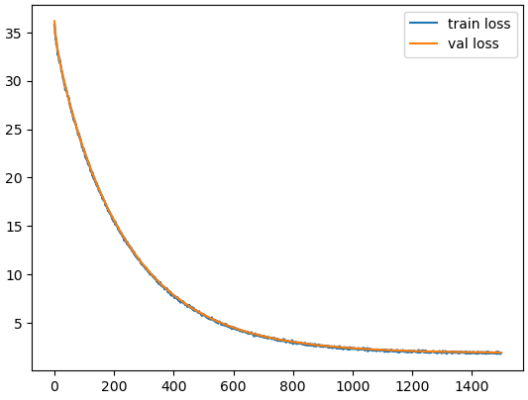
Le gradient, quant à lui, donne la direction dans laquelle nous devons ajuster nos poids w afin de minimiser la fonction de coût.

Implémentation

L'implémentation du SoftMax suit les mêmes étapes que la regression logistique binaire.

Lorsque nous entrainons notre model selon différents hyperparamètres, les résultats suivants sont obtenus :

```
lr = 5e-07, reg = 100.0
        -> train acc = 0.330, val acc = 0.297
lr = 5e-07, reg = 1000.0
        -> train acc = 0.388, val acc = 0.345
lr = 5e-07, reg = 10000.0
        -> train acc = 0.355, val acc = 0.327
lr = 5e-07, reg = 100000.0
        -> train acc = 0.281, val acc = 0.264
lr = 1e-06, reg = 100.0
        -> train acc = 0.369, val acc = 0.328
lr = 1e-06, reg = 1000.0
        -> train acc = 0.400, val acc = 0.356
lr = 1e-06, reg = 10000.0
        -> train acc = 0.350, val acc = 0.312
lr = 1e-06, reg = 100000.0
        -> train acc = 0.273, val acc = 0.258
best validation accuracy achieved during cross-validation: 0.356
```



Les meilleurs hyperparamètres semblent être « learning_rate = 1^{e}-6 » et « regularization = 1^{e}4 » nous permettant d'observer une précision de 35,5%. Les pertes du model diminuent au fur et à mesure de l'entrainement, indiquant que le model apprend effectivement à partir des données d'entrainements.

Néanmoins, bien que le modèle Softmax fournisse une base pour comprendre le comportement du jeu de données, des techniques et des architectures plus avancées seraient nécessaires pour obtenir des performances plus élevées sur CIFAR-10, tel que l'implémentation de réseau plus complexes.