Précisions : Perceptron

Nous allons éclaircir trois concepts fondamentaux pour un entraînement efficace en apprentissage profond. La version simplifié à l'extrême ne semble pas suffisamment performante.

La Fonction Softmax : Transformer des Scores en Probabilités

Imaginons que votre réseau de neurones doive identifier un chiffre manuscrit, de 0 à 9. En sortie, il ne produit pas directement des "probabilités", mais des **scores bruts** (aussi appelés **logits**) pour chaque chiffre. Ces scores peuvent être n'importe quel nombre réel (positif, négatif, grand, petit).

Le problème, c'est que ces scores ne sont pas intuitifs. Si un neurone pour le chiffre '3' donne un score de 10 et le neurone pour le chiffre '8' un score de 9, cela signifie-t-il que le réseau est très sûr que c'est un '3', ou juste un peu plus qu'un '8' ? Et que signifie un score négatif ?

C'est là qu'intervient la **fonction Softmax**. Son rôle est de transformer ces scores bruts en une **distribution de probabilités**.

Comment Softmax fonctionne-t-elle ?

Pour chaque score z_i de votre couche de sortie, Softmax calcule une valeur p_i de la manière suivante :

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^{K-1} e^{z_j}}$$

Où :

- z_i est le score (logit) du i-ème neurone de sortie.
- e est la base du logarithme naturel (environ 2.718).
- $\sum_{j=0}^{K-1} e^{z_j}$ est la somme de toutes les exponentielles des scores de sortie (K étant le nombre total de classes, ici 10 pour les chiffres 0-9).

Résultat :

- Chaque p_i sera un nombre entre 0 et 1.
- La somme de tous les p_i pour une entrée donnée sera toujours égale à 1.

Vous obtenez ainsi une véritable distribution de probabilités, où la valeur la plus élevée indique la classe prédite par le modèle. Par exemple, si le neurone '3' a une probabilité de 0.95 et le neurone '8' de 0.03, le réseau est très confiant que l'image est un '3'.

La Cross-Entropy Loss (Fonction de Perte) : Mesurer l'Erreur

Maintenant que notre réseau produit des probabilités, comment mesurons-nous à quel point il s'est trompé ? La fonction de perte (ou fonction de coût) est notre "baromètre d'erreur". Pour les problèmes de classification multi-classes comme MNIST, la **Cross-Entropy Loss** est le choix standard et le plus efficace.

Pourquoi pas une simple erreur ?

Une simple soustraction entre la prédiction et la vraie valeur serait insuffisante. La Cross-Entropy ne mesure pas seulement si la prédiction est fausse, mais aussi **à quel point le modèle est confiant dans une mauvaise prédiction**. Une erreur de Cross-Entropy élevée signifie que le modèle est très confiant dans une réponse erronée, ce qui est pire qu'une prédiction fausse mais avec une faible confiance.

Comment la Cross-Entropy fonctionne-t-elle ?

Pour un seul exemple d'entraînement, la Cross-Entropy Loss (L) est calculée ainsi :

$$L = -\sum_{i=0}^{K-1} y_i \,.\log(p_i)$$

Où :

- y_i est la "vraie" probabilité pour la classe i. Dans le cas d'une classification où chaque exemple appartient à une seule classe, y_i sera 1 pour la classe correcte et 0 pour toutes les autres (c'est ce qu'on appelle l'encodage "one-hot").
- p_i est la probabilité prédite par votre réseau pour la classe i (calculée par Softmax).
- log est le logarithme naturel.

Exemple pour comprendre :

// Si la vraie étiquette est '3' (donc $y_3 = 1$, et tous les autres $y_i = 0$)

// La formule se simplifie à : $L = -1 * \log(p_3)$ Puisque les autres termes sont $0 * \log(p_i) = 0$

// Donc, l'erreur est juste le logarithme négatif de la probabilité donnée à la *bonne* classe.

// Si p_3 est proche de 1 (très confiant et correct), $\log(p_3)$ est proche de 0 (négatif), donc L est proche de 0.

// Si p_3 est proche de 0 (pas confiant ou incorrect), $\log(p_3)$ est un grand nombre négatif, donc L est un grand nombre positif.

Minimiser cette perte revient à augmenter la probabilité de la bonne classe et à diminuer celle des mauvaises.

Rétropropagation : la Descente de Gradient

La rétropropagation (ou Backpropagation) est l'algorithme qui permet au réseau d'apprendre. C'est une application du principe de la **descente de gradient**. En gros, le réseau calcule l'erreur à la fin, puis remonte cette erreur couche par couche, en ajustant les poids et les biais pour **diminuer cette erreur**.

Imaginez que vous êtes au sommet d'une montagne (l'erreur est à son maximum) et que vous voulez descendre le plus rapidement possible. Vous regarderiez la pente (le gradient) et feriez un petit pas dans la direction la plus raide vers le bas. La rétropropagation fait exactement cela, mais pour les poids et biais.

Les étapes clés de la rétropropagation :

1. **Propagation Avant (Forward Pass) :**
    - Le réseau prend une entrée (image MNIST).
    - Il calcule les activations de la couche cachée.
    - Il calcule les scores de sortie, puis les probabilités avec Softmax.
    - Il détermine la classe prédite.
    - **(Nouveau)** Il **stocke** les entrées brutes des neurones (hiddenInputs et outputInputs) avant application des fonctions d'activation. Celles-ci sont cruciales pour le calcul des gradients.
2. **Calcul de l'Erreur de la Couche de Sortie :**
    - C'est la première étape de la "remontée" de l'erreur.
    - La dérivée de la Cross-Entropy Loss par rapport aux entrées des neurones de sortie, lorsque Softmax est utilisée, a une forme remarquablement simple : erreur_sortie_i = probabilité_prédite_i - probabilité_cible_i (one-hot)
    - Ce calcul vous donne directement la direction et la magnitude de l'ajustement nécessaire pour la couche de sortie.
3. **Mise à Jour des Poids et Biais de la Couche de Sortie :**
    - Chaque poids et biais de la dernière couche est mis à jour en fonction de l'erreur de sortie.
    - **Règle d'ajustement :**
      nouveau_poids = ancien_poids - taux_apprentissage * gradient
    - Pour les poids, le gradient est :
      erreur_sortie_i * activation_neurone_précédent_j.

- Pour les biais, le gradient est simplement erreur_sortie_i.

Extrait de code

```
// Pour chaque neurone de sortie (i)
Pour_chaque (i de 0 à 9) {
    biais_sortie[i]=biais_sortie[i]-erreur_sortie[i]*taux_apprentissage
    Pour_chaque (j de 0 à (hiddenNodes - 1)) {
        poids_cache_vers_sortie[j][i] = poids_cache_vers_sortie[j][i] -
erreur_sortie[i] * activation_cachee[j] * taux_apprentissage
    }
}
```

4. **Calcul de l'Erreur de la Couche Cachée :**
   - L'erreur est propagée "en arrière". L'erreur d'un neurone caché est la somme pondérée des erreurs des neurones de la couche suivante auxquels il est connecté, multipliée par la **dérivée de sa propre fonction d'activation**.
   - Pour la fonction ReLU, la dérivée est 1 si l'entrée du neurone était positive, et 0 si elle était négative ou nulle.

Extrait de code

```
fonction derivee_ReLU (x) {
    si (x>=0)
    alors retourne 1
    sinon retourne 0
}

// Pour chaque neurone caché (j)
Pour_chaque (j de 0 à (hiddenNodes - 1)) {
    somme_erreurs_provenant_sortie = 0
    Pour_chaque (i de 0 à 9) {
        somme_erreurs_provenant_sortie = somme_erreurs_provenant_sortie +
erreur_sortie[i] * poids_cache_vers_sortie[j][i]
    }
    // Appliquer la dérivée de ReLU à l'entrée brute du neurone caché
    erreur_cachee[j] = somme_erreurs_provenant_sortie *
derivee_ReLU(entree_brute_cachee[j])
}
```

5. **Mise à Jour des Poids et Biais de la Couche d'Entrée vers la Couche Cachée :**
    - De la même manière que pour la couche de sortie, les poids et les biais de la couche cachée sont ajustés en utilisant les hiddenErrors.

Extrait de code

```
// Pour chaque neurone caché (j)
Pour_chaque (j de 0 à (hiddenNodes - 1)) {
    biais_cache[j] = biais_cache[j] - erreur_cachee[j] * taux_apprentissage
    Pour_chaque (k de 0 à (inputNodes - 1)) {
        poids_entree_vers_cache[k][j] = poids_entree_vers_cache[k][j] -
erreur_cachee[j] * entree_reseau[k] * taux_apprentissage
    }
}
```