Documentation
Réseau de Neurones en NumPy

Implémentation from scratch

9 décembre 2025

Table des matières

1 Architecture du code

1.1 Classe `Layer`

Représente une couche individuelle du réseau.

1.1.1 Paramètres d'initialisation

— `n_input` : dimension de l'entrée
— `n_neurone` : nombre de neurones dans la couche
— `activ` : fonction d'activation (objet de type `ActivationF`)

1.1.2 Attributs principaux

— `w` : matrice de poids $\mathbb{R}^{n\_neurone \times n\_input}$
— `biais` : vecteur de biais $\mathbb{R}^{n\_neurone}$
— `x` : entrée de la couche
— `z` : sortie avant activation ($z = Wx + b$)
— `f` : sortie après activation

1.2 Classe `Neural_Network`

Réseau de neurones complet composé de plusieurs couches.

1.2.1 Paramètres d'initialisation

— `n_input_init` : dimension de l'entrée du réseau
— `nb_n_l` : liste du nombre de neurones par couche
Exemple : `[64, 32, 10]` pour 3 couches avec 64, 32 et 10 neurones
— `activ` : fonction(s) d'activation
— Soit un objet `ActivationF` unique (appliqué à toutes les couches)
— Soit une liste d'objets (un par couche)

2 Utilisation pratique

3 Méthodes d'entraînement

3.1 SGD (Stochastic Gradient Descent)

Mise à jour : $\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$

```
network.train_SGD(x_train, y_train, epochs=500, lr=0.01, batch_size=32)
```

Quand l'utiliser :
— Cas simples et petits réseaux
— Learning rate typique : 0.01 à 0.1

3.2 RMSProp

Mise à jour :

$$s_t = \beta s_{t-1} + (1-\beta) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} g_t$$

```
network.train_RMS(x_train, y_train, epochs=500, lr=0.001, batch_size=32)
```

Quand l'utiliser :
— Problèmes avec gradients de magnitudes variées
— Learning rate typique : 0.001
— $\beta = 0.9$ par défaut

3.3 Adam (Adaptive Moment Estimation)

Mise à jour :

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

```
network.train_ADAM(x_train, y_train, epochs=500, lr=0.001, batch_size
    =32)
```

Quand l'utiliser :
— **Recommandé par défaut** : le plus robuste
— Convergence rapide et stable
— Learning rate typique : 0.001
— $\beta_1 = 0.9$, $\beta_2 = 0.999$ par défaut

4 Hyperparamètres

4.1 Learning Rate (`lr`)

— **SGD** : 0.01 – 0.1
— **RMSProp/Adam** : 0.0001 – 0.01
— Si la loss explose : diminuer le learning rate
— Si la convergence est trop lente : augmenter légèrement

4.2 Batch Size

— Petits batch (16-32) : plus de bruit, meilleure généralisation
— Grands batch (128-256) : convergence plus stable, plus rapide
— Compromis typique : 32 ou 64

4.3 Nombre d'époques

— Surveiller les courbes train/val loss
— Arrêter quand val loss cesse de diminuer (early stopping)
— Typiquement : 500 à 5000 époques

5 Méthodes utiles

5.1 Prédiction

```
# Forward pass
output = network.forward(x)  # Shape: (n_output, n_samples)
```

5.2 Évaluation

```
# Calcul de la loss MSE sur un ensemble de test
test_loss = network.evaluate(x_test, y_test)
print(f"Test Loss: {test_loss:.4f}")
```

5.3 Accès aux pertes

```
# Historique des pertes
train_losses = network.train_losses  # Liste des pertes d'entrainement
val_losses = network.val_losses      # Liste des pertes de validation
```

6 Format des données

6.1 Shapes requis pour l'entraînement et la prédiction

6.1.1 Données d'entraînement

Format OBLIGATOIRE pour `train_SGD`, `train_RMS`, `train_ADAM` :

- `x_train` : $(n\_samples, n\_features)$
  - $n\_samples$: nombre d'exemples d'entraînement
  - $n\_features$: dimension de l'entrée (doit correspondre à `n_input_init`)
- `y_train` : $(n\_samples, n\_outputs)$
  - $n\_samples$: même nombre que `x_train`
  - $n\_outputs$: dimension de sortie (doit correspondre au dernier élément de `nb_n_l`)

Exemple concret :

```
# Probleme de regression : 5 features -> 1 sortie
x_train = np.random.randn(1000, 5)   # Shape: (1000, 5)
y_train = np.random.randn(1000, 1)   # Shape: (1000, 1)

network = Neural_Network(
    n_input_init=5,         # Correspond a x_train.shape[1]
    nb_n_l=[10, 5, 1],      # Dernier element (1) correspond a y_train.
        shape[1]
    activ=Activation.Sigmoid()
)

network.train_ADAM(x_train, y_train, epochs=500, lr=0.001, batch_size
    =32)
```

6.1.2 Données de validation

Format identique aux données d'entraînement :
— `x_val` : $(n\_val\_samples, n\_features)$
— `y_val` : $(n\_val\_samples, n\_outputs)$

```
# Memes dimensions que train, mais nombre d'echantillons different
x_val = np.random.randn(200, 5)   # Shape: (200, 5)
y_val = np.random.randn(200, 1)   # Shape: (200, 1)

network.train_ADAM(
    x_train, y_train,
    epochs=500, lr=0.001, batch_size=32,
    x_val=x_val,  # Optionnel
    y_val=y_val   # Optionnel
)
```

6.1.3 Prédictions avec `forward()`

Le code gère automatiquement **deux formats** :
Format 1 : Échantillon unique
— Input : $(n\_features,)$ ou $(n\_features, 1)$
— Output : $(n\_outputs, 1)$

```
# Prediction sur un seul echantillon
x_single = np.array([1.0, 2.0, 3.0, 4.0, 5.0])  # Shape: (5,)
output = network.forward(x_single)               # Shape: (1, 1)
print(output)  # [[0.73456]]
```

Format 2 : Batch
— Input : $(n\_samples, n\_features)$
— Output : $(n\_outputs, n\_samples)$ **Attention : transposé !**

```
# Prediction sur batch
x_batch = np.random.randn(100, 5)    # Shape: (100, 5)
output = network.forward(x_batch)    # Shape: (1, 100) <- Transpose !

# Pour recuperer au format (n_samples, n_outputs)
output_standard = output.T           # Shape: (100, 1)
```

6.2 Tableau récapitulatif des shapes

Méthode	Input Shape	Output Shape
`train_*`	(N, D_{in})	-
`evaluate`	(N, D_{in})	scalaire (loss)
`forward` (single)	$(D_{in},)$	$(D_{out}, 1)$
`forward` (batch)	(N, D_{in})	(D_{out}, N)

Où :
— N = nombre d'échantillons
— D_{in} = `n_input_init`
— D_{out} = dernier élément de `nb_n_l`

6.3 Erreurs courantes de shape

6.3.1 Erreur 1 : Dimensions inversées

```
# MAUVAIS : x_train avec shape (n_features, n_samples)
x_train = np.random.randn(5, 1000)   # ERREUR !

# CORRECT : shape (n_samples, n_features)
x_train = np.random.randn(1000, 5)   # OK !
```

6.3.2 Erreur 2 : y en 1D

```
# MAUVAIS : y_train en 1D
y_train = np.random.randn(1000)      # Shape: (1000,) - ERREUR !

# CORRECT : y_train en 2D
y_train = np.random.randn(1000, 1)   # Shape: (1000, 1) - OK !
# OU
y_train = y_train.reshape(-1, 1)     # Conversion rapide
```

6.3.3 Erreur 3 : Incohérence avec l'architecture

```
# Architecture : 10 entrees -> ... -> 3 sorties
network = Neural_Network(n_input_init=10, nb_n_l=[20, 3], activ=...)

# MAUVAIS : x_train a 5 features au lieu de 10
x_train = np.random.randn(1000, 5)   # ERREUR de dimension !

# MAUVAIS : y_train a 1 sortie au lieu de 3
y_train = np.random.randn(1000, 1)   # ERREUR de dimension !

# CORRECT
x_train = np.random.randn(1000, 10)  # 10 features
y_train = np.random.randn(1000, 3)   # 3 sorties
```

7 Conseils pratiques

7.1 Choix de l'architecture

1. **Nombre de couches cachées**
   — 1-2 couches : problèmes simples
   — 3-5 couches : problèmes complexes
2. **Taille des couches**
   — Commencer large, réduire progressivement
   — Exemple : [128, 64, 32, 10]
3. **Fonctions d'activation**
   — ReLU pour couches cachées (standard)
   — Sigmoid/Softmax pour la sortie selon le problème

7.2 Débogage

Loss explose :
— Réduire le learning rate (diviser par 10)
— Vérifier la normalisation des données
Pas de convergence :
— Augmenter le learning rate
— Essayer Adam au lieu de SGD
— Vérifier que les données ne sont pas constantes
Overfitting (train loss « val loss) :
— Réduire la complexité du réseau
— Ajouter du dropout (à implémenter)
— Augmenter les données d'entraînement