# Classification des Caractères Tifinagh (niveaux de gris) avec un Réseau de Neurones Multiclasses

Pr. M. BENADDY
Masters : IMSD & IAA

2024-2025

Ce TP consiste à implémenter un réseau de neurones multiclasses pour la classification des caractères Tifinagh générés à partir de la base de données AMHCD. La base de données contient 28182 images (64x64 pixels) réparties sur les 33 classes qui représente l'alphabet Tifinagh. Le modèle à implémenter en Python, qui est une adaptation du TP de la classification binaire, utilise un perceptron multicouche (MLP) avec deux couches cachées (paramètrable), des activations ReLU, et une sortie softmax. Vous devez décrire les formules mathématiques sous-jacentes.

## 1 Introduction

La reconnaissance des caractères manuscrits est un défi important en vision par ordinateur, particulièrement pour les scripts moins étudiés comme le Tifinagh, utilisé par les communautés amazighes. Dans ce projet vous devez implémenter un réseau de neurones multiclasses pour classifier ces caractères, en utilisant un MLP avec des couches de 64 et 32 neurones cachés, entraîné sur des images redimensionnées à 32x32 pixels et aplaties en vecteurs de 1024 caractéristiques.

Télécharger la base de données : `https://drive.google.com/file/d/1g03sIzi8F855KRMi0wmJesd` `view?usp=share_link`

## 2 Formules Mathématiques

Les formules clés du modèle.

### 2.1 L'opération forward

Pour une couche $l$, la sortie linéaire et l'activation sont :

$$Z^{[l]} = A^{[l-1]} W^{[l]} + b^{[l]} \tag{1}$$

$$A^{[l]} = g^{[l]}(Z^{[l]}) \tag{2}$$

où $A^{[0]} = X$, $X \in \mathbb{R}^{m \times 1024}$ est l'entrée, $W^{[l]} \in \mathbb{R}^{n^{[l-1]} \times n^{[l]}}$ les poids, et $b^{[l]} \in \mathbb{R}^{1 \times n^{[l]}}$ les biais. Pour les couches cachées ($l = 1, 2$) :

$$g^{[l]}(z) = \text{ReLU}(z) = \max(0, z) \tag{3}$$

Pour la couche de sortie ($l = 3$) :

$$A^{[3]} = \text{softmax}(Z^{[3]}), \quad \hat{y}_{i,c} = \frac{e^{z_{i,c}}}{\sum_{j=1}^{33} e^{z_{i,j}}} \tag{4}$$

## 2.2 La perte

L'entropie :

$$J = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{33} y_{i,c} \log(\hat{y}_{i,c}) \tag{5}$$

où $y_{i,c}$ est 1 si la classe $c$ est correcte, 0 sinon.

## 2.3 Précision

La précision est :

$$\text{Accuracy} = \frac{1}{m} \sum_{i=1}^{m} (\arg\max_c(\hat{y}_{i,c}) = \arg\max_c(y_{i,c})) \tag{6}$$

## 2.4 Rétropropagation

Le gradient initial pour la couche de sortie est :

$$\frac{\partial J}{\partial Z^{[3]}} = \hat{y} - Y \tag{7}$$

Pour les couches cachées ($l = 2, 1$) :

$$\frac{\partial J}{\partial Z^{[l]}} = \left( \frac{\partial J}{\partial Z^{[l+1]}} W^{[l+1]T} \right) \cdot \text{ReLU}'(Z^{[l]}) \tag{8}$$

où :

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{si } z > 0 \\ 0 & \text{sinon} \end{cases} \tag{9}$$

Les gradients des paramètres sont :

$$\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} (A^{[l-1]})^T \cdot \frac{\partial J}{\partial Z^{[l]}} \tag{10}$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J}{\partial Z^{[l]}} \tag{11}$$

Mise à jour des paramètres :

$$W^{[l]} \leftarrow W^{[l]} - \eta \frac{\partial J}{\partial W^{[l]}} \tag{12}$$

$$b^{[l]} \leftarrow b^{[l]} - \eta \frac{\partial J}{\partial b^{[l]}} \tag{13}$$

où $\eta = 0.01$.

## 3 Implémentation

Le code suivant implémente le modèle en Python, avec le chargement des données de Tifinagh partagée, le prétraitement, l'entraînement, et l'évaluation.

```python
import os
import pandas as pd
import numpy as np
import cv2
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns

# Fonctions d'activation
def relu(x):
    """
    ReLU activation: max(0, x)
    """
    assert isinstance(x, np.ndarray), "Input to ReLU must be a numpy array"
    # TODO
    assert np.all(result >= 0), "ReLU output must be non-negative"
    return result

def relu_derivative(x):
    """
    Derivative of ReLU: 1 if x > 0, else 0
    """
    assert isinstance(x, np.ndarray), "Input to ReLU derivative must be a numpy array"
    result = # TODO
    assert np.all((result == 0) | (result == 1)), "ReLU derivative must be 0 or 1"
    return result

def softmax(x):
    """
    Softmax activation: exp(x) / sum(exp(x))
    """
    assert isinstance(x, np.ndarray), "Input to softmax must be a numpy array"
    exp_x = # TODO
    result = exp_x / np.sum(exp_x, axis=1, keepdims=True)
    assert np.all((result >= 0) & (result <= 1)), "Softmax output must be in [0, 1]"
    assert np.allclose(np.sum(result, axis=1), 1), "Softmax output must sum to 1 per sample"
    return result

```

```python
# Classe MultiClassNeuralNetwork
class MultiClassNeuralNetwork:
    def __init__(self, layer_sizes, learning_rate=0.01):
        """
        Initialize the neural network with given layer sizes and learning rate.
        layer_sizes: List of integers [input_size, hidden1_size, ..., output_size]
        """
        assert isinstance(layer_sizes, list) and len(layer_sizes) >= 2, "layer_sizes must be a list with at least 2 elements"
        assert all(isinstance(size, int) and size > 0 for size in layer_sizes), "All layer sizes must be positive integers"
        assert isinstance(learning_rate, (int, float)) and learning_rate > 0, "Learning rate must be a positive number"

        self.layer_sizes = layer_sizes
        self.learning_rate = learning_rate
        self.weights = []
        self.biases = []

        # Initialisation des poids et biais
        np.random.seed(42)
        for i in range(len(layer_sizes) - 1):
            w = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * 0.01
            b = np.zeros((1, layer_sizes[i+1]))
            assert w.shape == (layer_sizes[i], layer_sizes[i+1]), f"Weight matrix {i+1} has incorrect shape"
            assert b.shape == (1, layer_sizes[i+1]), f"Bias vector {i+1} has incorrect shape"
            self.weights.append(w)
            self.biases.append(b)

    def forward(self, X):
        """
        Forward propagation: Z^{[l]} = A^{[l-1]} W^{[l]} + b^{[l]}, A^{[l]} = g(Z^{[l]})
        """
        assert isinstance(X, np.ndarray), "Input X must be a numpy array"
        assert X.shape[1] == self.layer_sizes[0], f"Input dimension ({X.shape[1]}) must match input layer size ({self.layer_sizes[0]})"

        self.activations = [X]
        self.z_values = []

```

```
        for i in range(len(self.weights) - 1):
            z = # TODO
            assert z.shape == (X.shape[0], self.layer_sizes[i+1]), f"Z^{[i+1]} has incorrect shape"
            self.z_values.append(z)
            self.activations.append(relu(z))

        z = # TODO
        assert z.shape == (X.shape[0], self.layer_sizes[-1]), "Output Z has incorrect shape"
        self.z_values.append(z)
        output = # TODO softmax call
        assert output.shape == (X.shape[0], self.layer_sizes[-1]), "Output A has incorrect shape"
        self.activations.append(output)

        return self.activations[-1]

    def compute_loss(self, y_true, y_pred):
        """
        Categorical Cross-Entropy: J = -1/m * sum(y_true * log(y_pred))
        """
        assert isinstance(y_true, np.ndarray) and isinstance(y_pred, np.ndarray), "Inputs to loss must be numpy arrays"
        assert y_true.shape == y_pred.shape, "y_true and y_pred must have the same shape"

        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        loss = # TODO
        assert not np.isnan(loss), "Loss computation resulted in NaN"
        return loss

    def compute_accuracy(self, y_true, y_pred):
        """
        Compute accuracy: proportion of correct predictions
        """
        assert isinstance(y_true, np.ndarray) and isinstance(y_pred, np.ndarray), "Inputs to accuracy must be numpy arrays"
        assert y_true.shape == y_pred.shape, "y_true and y_pred must have the same shape"

        predictions = # TODO
        true_labels = # TODO
        accuracy = # TODO
        assert 0 <= accuracy <= 1, "Accuracy must be between 0 and 1"
        return accuracy
```

```python

    def backward(self, X, y, outputs):
        """
        Backpropagation: compute dW^{[l]}, db^{[l]} for each layer
        """
        assert isinstance(X, np.ndarray) and isinstance(y, np.ndarray) and isinstance(outputs, np.ndarray), "Inputs to backward must be numpy arrays"
        assert X.shape[1] == self.layer_sizes[0], f"Input dimension ({X.shape[1]}) must match input layer size ({self.layer_sizes[0]})"
        assert y.shape == outputs.shape, "y and outputs must have the same shape"

        m = X.shape[0]
        self.d_weights = # TODO
        self.d_biases = # TODO

        dZ = outputs - y  # Gradient pour softmax + cross-entropy
        assert dZ.shape == outputs.shape, "dZ for output layer has incorrect shape"
        self.d_weights[-1] = (self.activations[-2].T @ dZ) / m
        self.d_biases[-1] = np.sum(dZ, axis=0, keepdims=True) / m

        for i in range(len(self.weights) - 2, -1, -1):
            dZ = #TODO
            assert dZ.shape == (X.shape[0], self.layer_sizes[i+1]), f"dZ^{[i+1]} has incorrect shape"
            self.d_weights[i] = #TODO
            self.d_biases[i] = #TODO

        # TODO: Ajouter une r gularisation L2 aux gradients des poids
        # dW^{[l]} += lambda * W^{[l]} / m, o  lambda est le coefficient de r gularisation

        for i in range(len(self.weights)):
            self.weights[i] -= self.learning_rate * self.d_weights[i]
            self.biases[i] -= self.learning_rate * self.d_biases[i]

    def train(self, X, y, X_val, y_val, epochs, batch_size):
        """
        Train the neural network using mini-batch SGD, with validation
        """
        assert isinstance(X, np.ndarray) and isinstance(y, np.ndarray), "X and y must be numpy arrays"
        assert isinstance(X_val, np.ndarray) and isinstance(y_val, np.ndarray), "X_val and y_val must be numpy arrays"
```

```python
        assert X.shape[1] == self.layer_sizes[0], f"Input dimension ({X.shape[1]}) must match input layer size ({self.layer_sizes[0]})"
        assert y.shape[1] == self.layer_sizes[-1], f"Output dimension ({y.shape[1]}) must match output layer size ({self.layer_sizes[-1]})"
        assert X_val.shape[1] == self.layer_sizes[0], f"Validation input dimension ({X_val.shape[1]}) must match input layer size ({self.layer_sizes[0]})"
        assert y_val.shape[1] == self.layer_sizes[-1], f"Validation output dimension ({y_val.shape[1]}) must match output layer size ({self.layer_sizes[-1]})"
        assert isinstance(epochs, int) and epochs > 0, "Epochs must be a positive integer"
        assert isinstance(batch_size, int) and batch_size > 0, "Batch size must be a positive integer"

        train_losses = []
        val_losses = []
        train_accuracies = []
        val_accuracies = []

        for epoch in range(epochs):
            indices = np.random.permutation(X.shape[0])
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            epoch_loss = 0
            for i in range(0, X.shape[0], batch_size):
                X_batch = X_shuffled[i:i+batch_size]
                y_batch = y_shuffled[i:i+batch_size]

                outputs = self.forward(X_batch)
                epoch_loss += self.compute_loss(y_batch, outputs)
                self.backward(X_batch, y_batch, outputs)

            # Calculer les pertes et accuracies
            train_loss = epoch_loss / (X.shape[0] // batch_size)
            train_pred = self.forward(X)
            train_accuracy = self.compute_accuracy(y, train_pred)
            val_pred = self.forward(X_val)
            val_loss = self.compute_loss(y_val, val_pred)
            val_accuracy = self.compute_accuracy(y_val, val_pred)

            train_losses.append(train_loss)
            val_losses.append(val_loss)
            train_accuracies.append(train_accuracy)
            val_accuracies.append(val_accuracy)

            if epoch % 10 == 0:
                print(f"Epoch {epoch}, Train Loss: {train_loss:.4f
```

```
                    }, Val Loss: {val_loss:.4f}, "
                      f"Train Acc: {train_accuracy:.4f}, Val Acc: {val_accuracy:.4f}")

        return train_losses, val_losses, train_accuracies, val_accuracies

    def predict(self, X):
        """
        Predict class labels
        """
        assert isinstance(X, np.ndarray), "Input X must be a numpy array"
        assert X.shape[1] == self.layer_sizes[0], f"Input dimension ({X.shape[1]}) must match input layer size ({self.layer_sizes[0]})"

        outputs = self.forward(X)
        predictions = np.argmax(outputs, axis=1)
        assert predictions.shape == (X.shape[0],), "Predictions have incorrect shape"
        return predictions

# D finir le chemin vers le dossier d compress
data_dir = os.path.join(os.getcwd(), 'amhcd-data-64/tifinagh-images/')
print(data_dir)
current_working_directory = os.getcwd()
print(current_working_directory)

# Charger le fichier CSV contenant les  tiquettes
try:
    labels_df = pd.read_csv(os.path.join(data_dir, 'amhcd-data-64/labels-map.csv'))
    assert 'image_path' in labels_df.columns and 'label' in labels_df.columns, "CSV must contain 'image_path' and 'label' columns"
except FileNotFoundError:
    print("labels-map.csv not found. Please check the dataset structure.")
    # Alternative : construire un DataFrame   partir des dossiers
    image_paths = []
    labels = []
    for label_dir in os.listdir(data_dir):
        label_path = os.path.join(data_dir, label_dir)
        if os.path.isdir(label_path):
            for img_name in os.listdir(label_path):
                image_paths.append(os.path.join(label_path, img_name))
                labels.append(label_dir)
    labels_df = pd.DataFrame({'image_path': image_paths, 'label':
```

```
        labels})

# V rifier le DataFrame
assert not labels_df.empty, "No data loaded. Check dataset files."
print(f"Loaded {len(labels_df)} samples with {labels_df['label'].nunique()} unique classes.")

# Encoder les  tiquettes
label_encoder = LabelEncoder()
labels_df['label_encoded'] = label_encoder.fit_transform(labels_df['label'])
num_classes = len(label_encoder.classes_)

# Fonction pour charger et pr traiter une image
def load_and_preprocess_image(image_path, target_size=(32, 32)):
    """
    Load and preprocess an image: convert to grayscale, resize, normalize
    """
    assert os.path.exists(image_path), f"Image not found: {image_path}"
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    assert img is not None, f"Failed to load image: {image_path}"
    img = cv2.resize(img, target_size)
    img = img.astype(np.float32) / 255.0  # Normalisation
    return img.flatten()  # Aplatir pour le r seau de neurones

# Charger toutes les images
X = np.array([load_and_preprocess_image(os.path.join(data_dir, path)) for path in labels_df['image_path']])
y = labels_df['label_encoded'].values

# V rifier les dimensions
assert X.shape[0] == y.shape[0], "Mismatch between number of images and labels"
assert X.shape[1] == 32 * 32, f"Expected flattened image size of {32*32}, got {X.shape[1]}"

# Diviser en ensembles d'entra nement, validation et test
X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.25, stratify=y_temp, random_state=42)

# Convertir explicitement en NumPy arrays
X_train = np.array(X_train)
X_val = np.array(X_val)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_val = np.array(y_val)
y_test = np.array(y_test)
```

```
assert X_train.shape[0] + X_val.shape[0] + X_test.shape[0] == X.shape[0], "Train-val-test split sizes must sum to total samples"

print(f"Train: {X_train.shape[0]} samples, Validation: {X_val.shape[0]} samples, Test: {X_test.shape[0]} samples")

# Encoder les  tiquettes  en one-hot pour la classification multiclasse
one_hot_encoder = OneHotEncoder(sparse_output=False)
y_train_one_hot = np.array(one_hot_encoder.fit_transform(y_train.reshape(-1, 1)))
y_val_one_hot = np.array(one_hot_encoder.transform(y_val.reshape(-1, 1)))
y_test_one_hot = np.array(one_hot_encoder.transform(y_test.reshape(-1, 1)))

# V rifier que les tableaux one-hot sont des NumPy arrays
assert isinstance(y_train_one_hot, np.ndarray), "y_train_one_hot must be a numpy array"
assert isinstance(y_val_one_hot, np.ndarray), "y_val_one_hot must be a numpy array"
assert isinstance(y_test_one_hot, np.ndarray), "y_test_one_hot must be a numpy array"

# Cr er et entra ner le mod le
layer_sizes = [X_train.shape[1], 64, 32, num_classes]  # 64 et 32 neurones cach s, 33 classes
nn = MultiClassNeuralNetwork(layer_sizes, learning_rate=0.01)
train_losses, val_losses, train_accuracies, val_accuracies = nn.train(
    X_train, y_train_one_hot, X_val, y_val_one_hot, epochs=100, batch_size=32
)

# TODO: Ajouter une validation crois e pour  valuer  la robustesse du mod le
# TODO: Impl menter l'optimiseur Adam pour une meilleure convergence

# Pr dictions et  valuation
y_pred = nn.predict(X_test)
print("\nRapport de classification (Test set) :")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))

# Matrice de confusion
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
```

```
plt.title('Matrice de confusion (Test set)')
plt.xlabel('Pr dit')
plt.ylabel('R el')
plt.savefig('confusion_matrix.png')
plt.close()

# Courbes de perte et d'accuracy
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Courbe de perte
ax1.plot(train_losses, label='Train Loss')
ax1.plot(val_losses, label='Validation Loss')
ax1.set_title('Courbe de perte')
ax1.set_xlabel(' poque ')
ax1.set_ylabel('Perte')
ax1.legend()

# Courbe d'accuracy
ax2.plot(train_accuracies, label='Train Accuracy')
ax2.plot(val_accuracies, label='Validation Accuracy')
ax2.set_title('Courbe de pr cision')
ax2.set_xlabel(' poque ')
ax2.set_ylabel('Pr cision')
ax2.legend()

plt.tight_layout()
fig.savefig('loss_accuracy_plot.png')
plt.close()
```

## 4 Visualisations

Les résultats atteints pour la configuration suivante : $32 \times 32, 64, 32, 33$

```
layer_sizes = [X_train.shape[1], 64, 32, num_classes]  # 64 et 32 neurones cach s , 33 classes
nn = MultiClassNeuralNetwork(layer_sizes, learning_rate=0.01)
train_losses, val_losses, train_accuracies, val_accuracies = nn.train(
    X_train, y_train_one_hot, X_val, y_val_one_hot, epochs=100, batch_size=32
)
```

## 5 Travail à faire

En plus de l'implémentation du modèle vous devez rédiger un rapport (article en Anglais) selon le format IMRAD déjà evoquée dans le TP de classification binaire.

NB : Le lien vers le code github doit être inclu dans le rapport.

Des améliorations incluent (Bonus) :

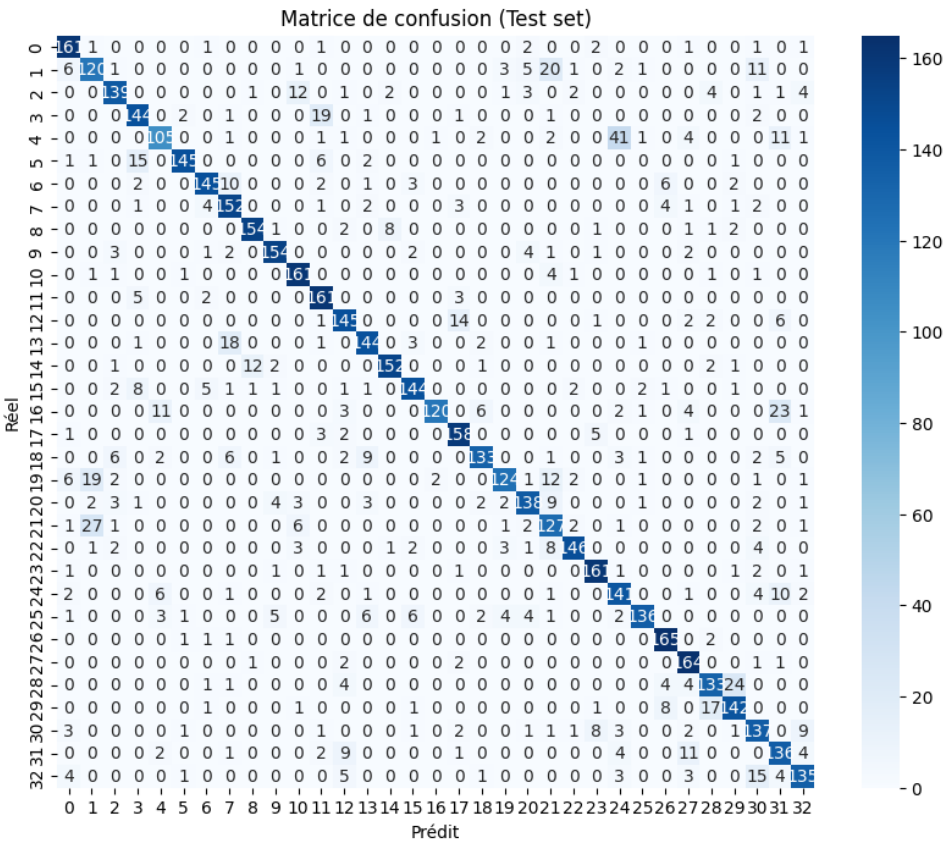


Figure 1: Matrice de confusion pour l'ensemble de test, montrant les prédictions correctes (diagonale) et les erreurs.

- **Régularisation L2** : Ajouter $\frac{\lambda}{m}W^{[l]}$ aux gradients.
- **Optimiseur Adam** : Utiliser des moments adaptatifs pour une meilleure convergence.
- **Validation croisée** : Évaluer la robustesse avec K-fold.
- **Augmentation des données** : Appliquer des rotations et translations.
- **Reprendre le travail avec la base de données AMHCD citée dans la bibliographie**

# References

[1] Benaddy, M., et al. "Amazigh Handwritten Character Database (AMHCD)." Kaggle, 2020. Disponible sur : `https://www.kaggle.com/datasets/benaddym/amazigh-handwritten-character-database-amhcd`.

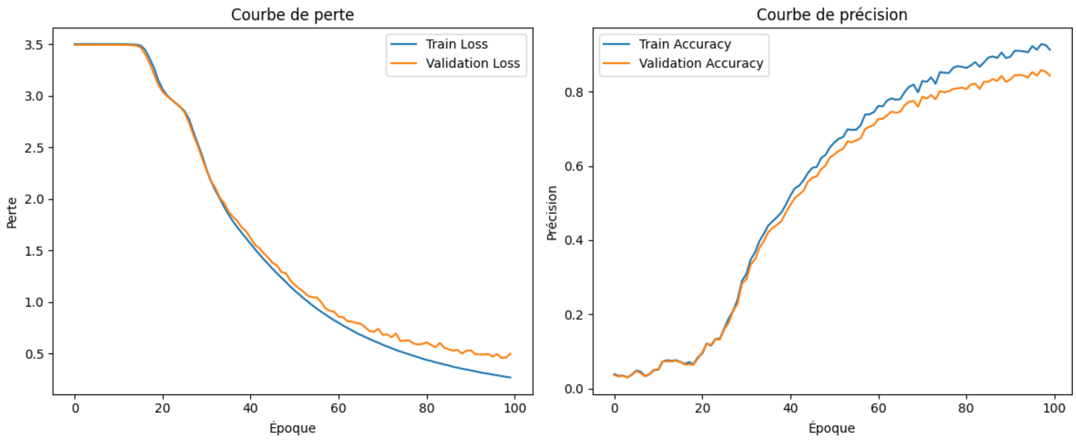


Figure 2: Courbes de perte (gauche) et de précision (droite) pour les ensembles d'entraînement et de validation.