



UNIVERSITÉ **موندبوليس**
MUNDIAPOLIS
HONORIS UNITED UNIVERSITIES

Reconnaissance de Texte Manuscrit

PRÉSENTÉ PAR :
TRAOUKI YOUNES
HANSALI BILAL

ENCADRÉ PAR :
ABDELLATIF MOUSSAID



Introduction



Problématique et Objectif



Problématique

Difficulté d'interprétation des textes manuscrits dans des scénarios comme l'archivage ou la numérisation.



Objectif

Développer un système efficace et facile d'utilisation pour transcrire des textes manuscrits en texte numérique.

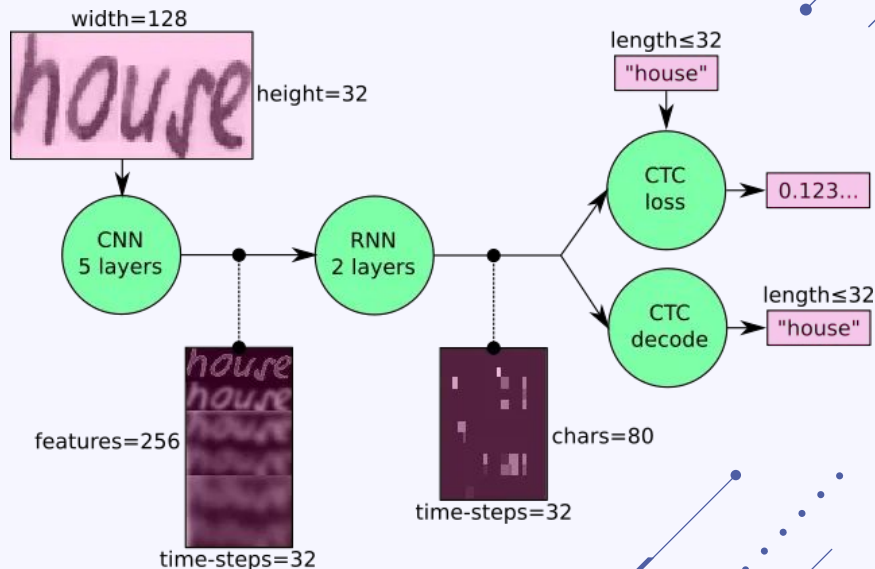
The background features abstract geometric patterns in the corners, consisting of thin blue lines, dots, and circles. A horizontal line with dots at its ends is positioned below the title.

Approches **Adoptées**

Première Approche

Architecture :

- 5 couches CNN pour extraire les caractéristiques des images.
- 2 couches RNN (LSTM) pour capturer la dépendance séquentielle des caractères.
- CTC (Connectionist Temporal Classification) pour l'alignement entre l'entrée (image) et la sortie (texte).



- Le dataset IAM

C'est une base de données utilisée pour reconnaître l'écriture manuscrite (Handwriting Text Recognition, ou HTR). Elle contient :

- 13 000 lignes de texte manuscrit écrites par 657 personnes.
- Des images de texte manuscrit en noir et blanc.
- Des annotations pour chaque mot, ligne et page entière.

Chaque image est accompagnée d'un fichier texte avec sa transcription.





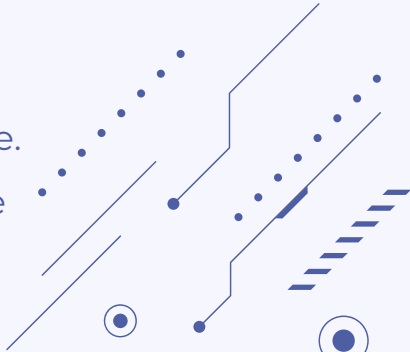
Utilisation du dataset IAM dans cette approche



■ Préparation des données :

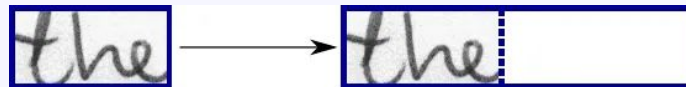
- Les images sont ajustées (taille et proportions) et leurs pixels sont normalisés (valeurs entre 0 et 1).
- Les textes manuscrits sont transformés en séquences de caractères (lettres, chiffres, etc.).

■ Entraînement :

- Les images sont utilisées comme entrées pour le modèle.
 - Les textes associés servent de réponses pour entraîner le modèle grâce à une fonction appelée CTC Loss.
- 

Données d'entrée

- **Taille de l'image d'entrée** : 128×32 en niveaux de gris.
- **Redimensionnement** : Si l'image n'a pas cette taille, elle est redimensionnée pour avoir une largeur de 128 ou une hauteur de 32, sans distorsion.
- **Placement dans une image cible** : L'image redimensionnée est copiée dans une image blanche de taille 128×32.
- **Normalisation** : Les valeurs de gris de l'image sont normalisées pour faciliter l'apprentissage du réseau neuronal.
- **Augmentation des données** :
 - L'image peut être placée aléatoirement dans l'image cible (pas forcément à gauche).
 - L'image peut être redimensionnée de manière aléatoire.



Opérations

→ CNN:

- **Entrée de l'image :** L'image est donnée au réseau CNN pour qu'il analyse et extraie des caractéristiques importantes.
- **Opérations dans chaque couche :**
 - Convolution : Un filtre est appliqué pour détecter des motifs dans l'image. Les deux premières couches utilisent un filtre 5×5 et les trois dernières utilisent un filtre 3×3 .
 - ReLU : Une fonction non linéaire est appliquée pour ne garder que les valeurs positives.
 - Pooling : Cette étape réduit la taille de l'image tout en gardant l'essentiel.
- **Effet sur l'image :** La hauteur de l'image est réduite de moitié à chaque couche, mais de nouvelles cartes de caractéristiques (canaux) sont ajoutées.
- **Sortie finale :** L'image traitée devient une carte de caractéristiques de taille 32×256 .

Opérations

→ RNN:

- **Entrée du RNN** : La séquence d'entrée contient 256 caractéristiques par étape de temps.
- **Sortie du RNN** : La sortie est une matrice de taille 32×80 .
 - **32** : Correspond aux étapes de temps.
 - **80** : Représente les 79 caractères possibles dans le jeu de données IAM, plus un caractère spécial pour les étiquettes vides (CTC).

Opérations

→ CTC:

- **Rôle du CTC (Connectionist Temporal Classification) :**
 - **Pendant l'entraînement :**
 - Le CTC reçoit la matrice de sortie du RNN et le texte attendu (vérité fondamentale).
 - Il calcule une valeur de perte pour ajuster le réseau neuronal.
 - **Pendant l'inférence (prédiction) :**
 - Le CTC reçoit uniquement la matrice de sortie du RNN.
 - Il la décode pour produire le texte final.
- **Longueur maximale des textes :**
 - Le texte attendu (vérité fondamentale) et le texte reconnu peuvent contenir jusqu'à 32 caractères.

Données de Sortie

→ CNN:

- Sortie du CNN :

La sortie est une séquence de longueur 32, où chaque étape contient 256 caractéristiques.

- Lien avec les RNN :

Ces caractéristiques sont ensuite traitées par les couches RNN pour une analyse plus approfondie.

- Corrélations détectées dans les caractéristiques :

Certaines caractéristiques correspondent déjà à des éléments de l'image d'entrée, comme :

Des caractères spécifiques (par exemple, le "e").

Des caractères répétés (par exemple, "tt").

Des propriétés visuelles des caractères, comme des boucles (présentes dans les "l" ou les "e" manuscrits).

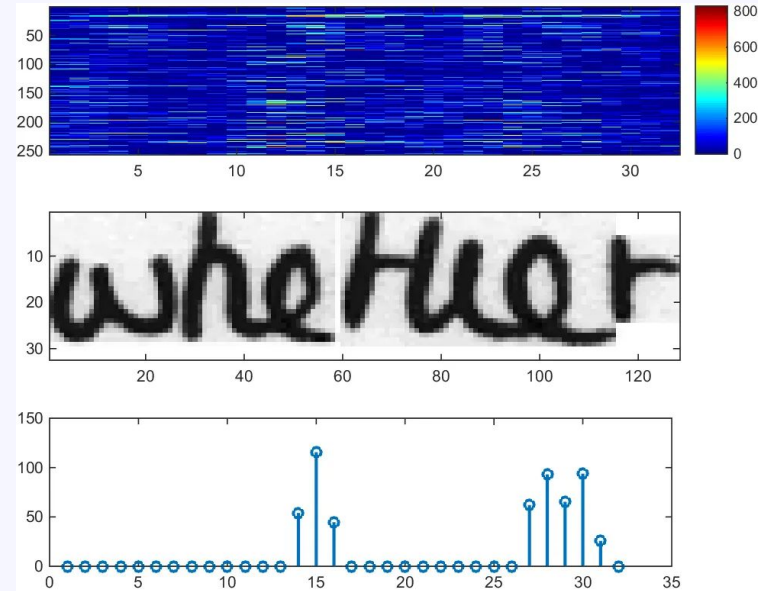
Données de Sortie

→ CNN:

En haut : 256 caractéristiques par pas de temps sont calculées par les couches CNN.

Au milieu : image d'entrée.

En bas : tracé de la 32e caractéristique, qui présente une forte corrélation avec l'occurrence du caractère « e » dans l'image.



Données de Sortie

→ RNN:

- Sortie du RNN :

La sortie est une matrice où chaque position contient des scores pour 80 caractères possibles, y compris une étiquette vide CTC (80e).

Les caractères sont représentés dans un ordre spécifique (ex. : lettres, chiffres, symboles).

- Alignement des prédictions :

Les caractères sont généralement prédits aux positions correspondantes dans l'image (exemple : le "i" est bien aligné).

Si un caractère est légèrement décalé (comme le "e"), cela n'est pas problématique grâce à l'opération CTC.

Données de Sortie

→ RNN:

- Décodage du texte avec CTC :

Étapes pour obtenir le texte final :

Prendre le caractère le plus probable à chaque position
(meilleur chemin).

Supprimer les caractères répétés.

Supprimer les espaces (étiquettes vides).

Exemple :

Sortie brute : "l---ii--tt--l---e".

Après suppression : "little".

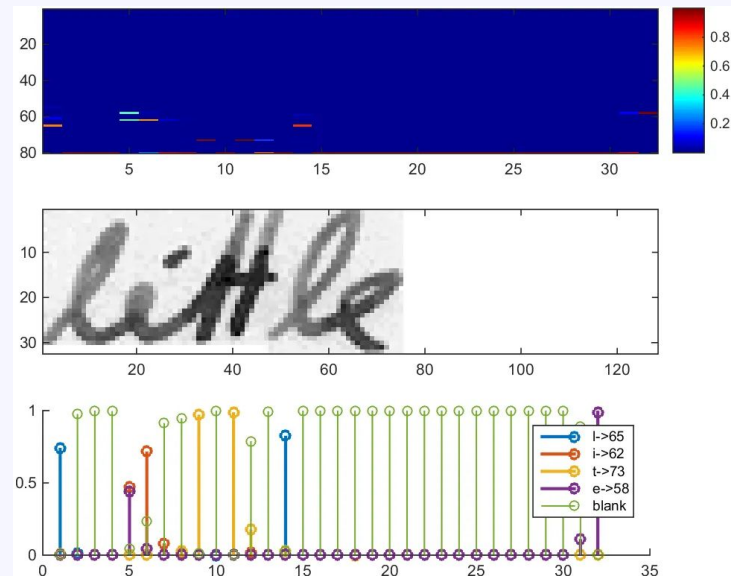
Données de Sortie

→ RNN:

En haut : matrice de sortie des couches RNN.

Au milieu : image d'entrée.

En bas : probabilités pour les caractères « l », « i », « t », « e » et l'étiquette vide CTC.



CTC Decode

→ BestPath (Greedy Decoding) :

How it works: Chooses the most probable label at each time step.

Example: Given the output probabilities for a sequence of characters over several time steps:

- **Time Step 1** : $[P(a) = 0.7, P(b) = 0.2, P(c) = 0.1]$
- **Time Step 2** : $[P(b) = 0.8, P(c) = 0.1, P(d) = 0.1]$
- **Time Step 3** : $[P(a) = 0.9, P(b) = 0.05, P(c) = 0.05]$

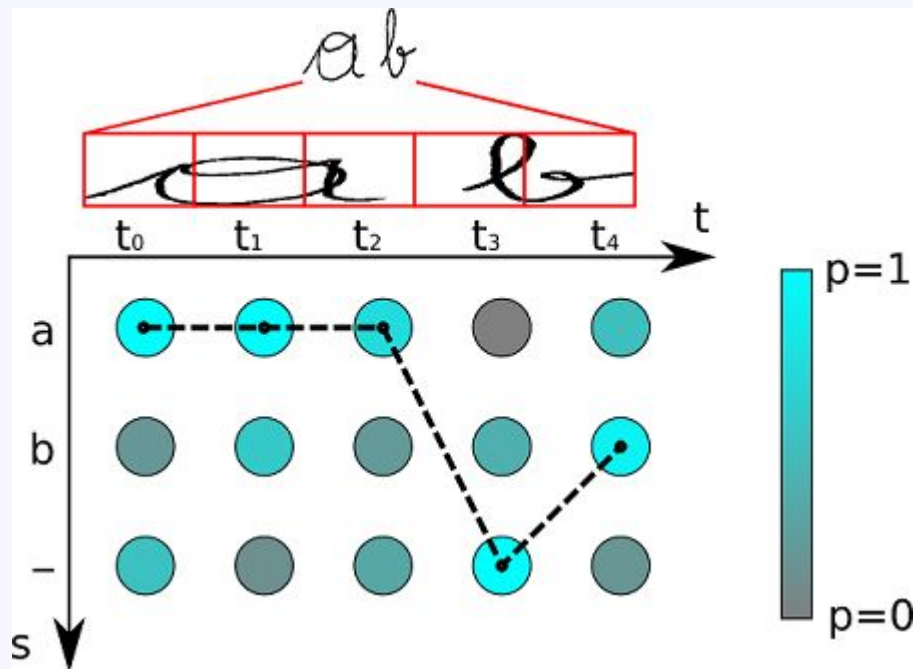
BestPath will select:

- **Step 1:** a (most probable)
- **Step 2:** b (most probable)
- **Step 3:** a (most probable)

Output: "aba"

CTC Decode

→ BestPath (Greedy Decoding) :



CTC Decode

→ BeamSearch :

How it works: Maintains a beam of the top k sequences at each step and selects the best overall sequence.

Example: Let's assume beam width $k = 2$. Given the probabilities:

- **Time Step 1** : $[P(a) = 0.7, P(b) = 0.2, P(c) = 0.1]$
- **Time Step 2** : $[P(a) = 0.6, P(i) = 0.3, P(o) = 0.1]$
- **Time Step 3** : $[P(t) = 0.8, P(o) = 0.1, P(c) = 0.1]$

CTC Decode

→ BeamSearch :

Step 1: Initialize the Beam

- At **Time Step 1**, we start with the top 2 most probable labels:
 - Sequence 1: **a** (probability = 0.7)
 - Sequence 2: **b** (probability = 0.2)

Step 2: Extend the Beam

- At **Time Step 2**, each sequence from the previous step can branch out into new possible labels, so we need to extend each of the 2 current sequences with the possible labels:
 - For Sequence 1 (a):**
 - a** → **a** (prob = $0.7 * 0.6 = 0.42$)
 - a** → **b** (prob = $0.7 * 0.3 = 0.21$)
 - a** → **c** (prob = $0.7 * 0.1 = 0.07$)

For Sequence 2 (b):

- b** → **a** (prob = $0.2 * 0.6 = 0.12$)
 - b** → **b** (prob = $0.2 * 0.3 = 0.06$)
 - b** → **c** (prob = $0.2 * 0.1 = 0.02$)
- Now, we choose the **top 2 sequences** based on the highest probabilities:
 - Top 1: **a** → **a** (0.42)
 - Top 2: **a** → **b** (0.21)
- After Step 2:**
 - Sequence 1: **a** → **a** (prob = 0.42)
 - Sequence 2: **a** → **b** (prob = 0.21)

Step 3: Extend Again

- At **Time Step 3**, we extend each of the current sequences:
 - For Sequence 1 (a → a):**
 - a** → **a** → **a** (prob = $0.42 * 0.5 = 0.21$)
 - a** → **a** → **b** (prob = $0.42 * 0.4 = 0.168$)
 - a** → **a** → **c** (prob = $0.42 * 0.1 = 0.042$)

....

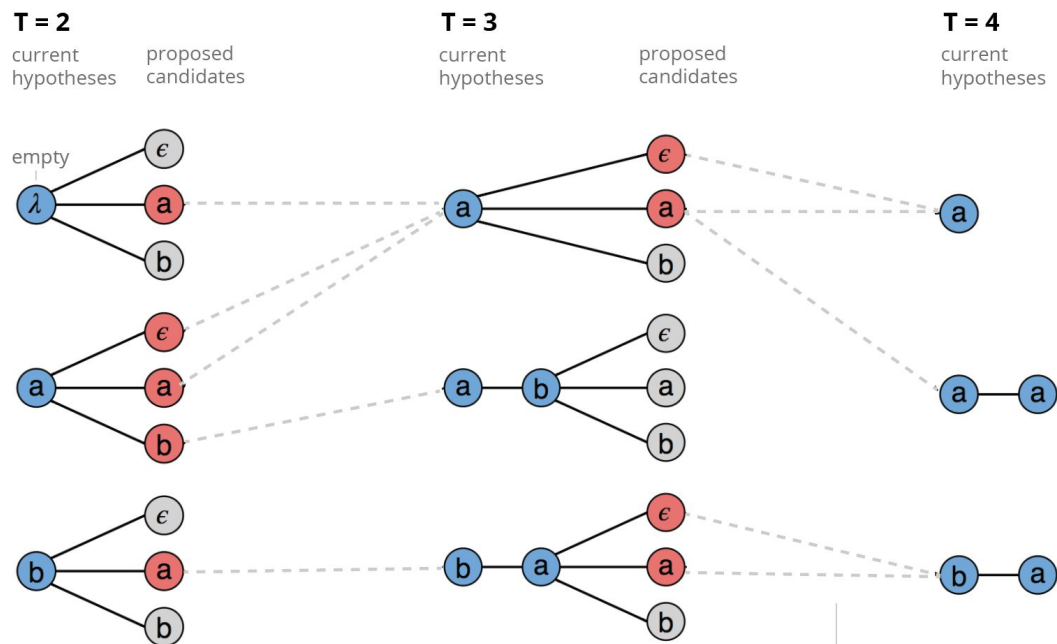
Final Output

- After the last step, BeamSearch will choose the most probable sequence:
 - The top sequence is a → a → a a probability of 0.21.**

● **Output: "aaa"**

CTC Decode

→ BeamSearch :



CTC beam search algorithm with an output alphabet $\{\epsilon, a, b\}$ and a beam size of three.

Multiple extensions can merge in to the same hypothesis.

CTC Decode

→ WordBeamSearch :

How it works: Similar to BeamSearch but **uses a lexicon** or language model to favor valid word sequences.

Example: Let's say we have a lexicon with words like "bat", "cat", and "rat", and the following output probabilities:

1. Lexicon:

This is a predefined list of valid words the system is trained on. In our example:

- **Lexicon** = ["bat", "cat", "rat"]

The goal of **WordBeamSearch** is to find the most probable sequence of **words**, not just characters.

2. Predicted Character Probabilities:

At each **time step**, the model outputs probabilities for each possible character. These probabilities represent the model's confidence in predicting a certain character at that time step. Let's assume the following output probabilities for each time step:

- **Time Step 1:** [P(b) = 0.7, P(c) = 0.2, P(r) = 0.1]
- **Time Step 2:** [P(a) = 0.6, P(i) = 0.3, P(o) = 0.1]
- **Time Step 3:** [P(t) = 0.8, P(o) = 0.1, P(c) = 0.1]

These probabilities represent the likelihood of each character at each step:

- At **Time Step 1**, the most likely character is '**b**' ($P(b) = 0.7$).
- At **Time Step 2**, the most likely character is '**a**' ($P(a) = 0.6$).
- At **Time Step 3**, the most likely character is '**t**' ($P(t) = 0.8$).

3. Step-by-Step Decoding:

Now, let's walk through how **WordBeamSearch** processes this information:

Step 1: Initialize Beam

At the first time step, the model has three possible characters to choose from: **b**, **c**, and **r**. These are the candidates that will form the first character of our word sequence.

- **Candidates after Step 1:** b, c, r

Step 2: Add New Characters

At **Time Step 2**, the model adds characters "a", "i", and "o" to each of the existing candidates. So, we now have combinations of the characters from the first and second steps

CTC Decode

→ WordBeamSearch :

- Possible sequences at Step 2:
 - **ba** (from 'b' at Step 1 + 'a' at Step 2)
 - **bi** (from 'b' + 'i')
 - **ca** (from 'c' + 'a')
 - **ri** (from 'r' + 'i')

Step 3: Add More Characters and Check Lexicon

At **Time Step 3**, the model adds characters "t", "o", and "c". Let's check how each combination of characters evolves:

- **ba** can extend to **bat** (valid word from lexicon).
- **bi** can extend to **bit**, but **bit** is not in the lexicon.
- **ca** can extend to **cat** (valid word from lexicon).
- **ri** can extend to **rit**, which is not a valid word.

Now, we have two potential valid words:

- **bat** (from ba + t)
- **cat** (from ca + t)

Choosing the Best Word Sequence:

- **WordBeamSearch** now uses the **lexicon** (or language model) to check if any of the sequences are valid words and how likely each sequence is.
- In this case, "**bat**" and "**cat**" are both valid words from the lexicon, but "**bat**" is the more probable word based on the character probabilities at each time step:
 - **P(b) = 0.7** at Time Step 1
 - **P(a) = 0.6** at Time Step 2
 - **P(t) = 0.8** at Time Step 3

Given the probabilities of these characters, "**bat**" is selected over "**cat**" because it has higher overall likelihood based on the model's predictions.

4. Final Output:

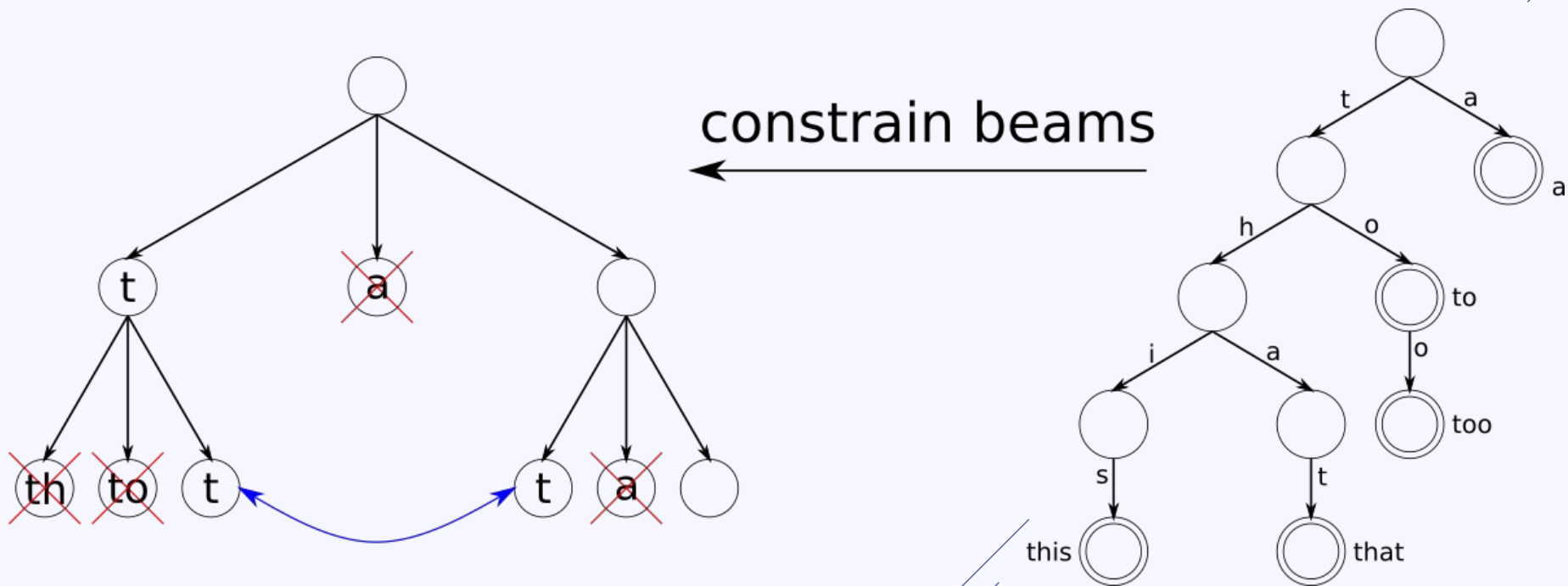
After considering all possible sequences and selecting the one with the highest probability from the lexicon, **WordBeamSearch** outputs the most probable word sequence:

- **Output: "bat"**

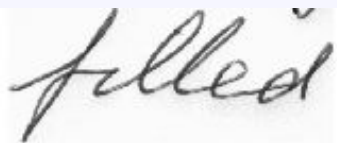
This process ensures that the decoded sequence is not just a combination of high-probability characters but a valid word sequence based on the lexicon.

CTC Decode

→ WordBeamSearch :



Use case



Best path decoding	"fuleid" ❌
Vanilla beam search	"fuleid" ❌
Word beam search	"filled" ✅

Le dictionnaire est créé automatiquement en mode entraînement et validation à partir des mots du dataset IAM et est enregistré dans `data/corpus.txt`. La liste des caractères de mots se trouve dans `model/wordCharList.txt`. La beam width est fixée à 50, conforme à celle du vanilla beam search decoding.

Résultats & Obstacles

→ Training model :

```
[OK] "who" -> "who"  
[ERR:1] "said" -> "seid"  
[ERR:1] "hullo" -> "hnllo"  
[OK] "to" -> "to"  
[ERR:2] "him" -> "tin"  
[OK] "in" -> "in"  
[OK] "the" -> "the"  
[ERR:1] "garden" -> "gardes"  
[ERR:1] "?" -> ""
```

```
Character error rate: 11.043381535038932%. Word accuracy: 74.66181061394381%.  
Character error rate not improved, best so far: 10.82091212458287%  
No more improvement for 25 epochs. Training stopped.
```

Résultats & Obstacles

→ Erreur :

tensorflow.python.framework.errors_impl.NotFoundError: Key batch_normalization_1/beta not found in checkpoint #179



younestr opened this issue 5 days ago · 1 comment



younestr commented 5 days ago

When running the testing command for this model after training it from scratch , on the words : "python main.py --img_file ..test_images/word.png" .

I manually inspected the model's snapshot , i found :

```
"tensor: batch_normalization_1_1/beta (float32) [64]
tensor: batch_normalization_1_1/beta/Adam (float32) [64]
tensor: batch_normalization_1_1/beta/Adam_1 (float32) [64]
tensor: batch_normalization_1_1/gamma (float32) [64]
tensor: batch_normalization_1_1/gamma/Adam (float32) [64]
tensor: batch_normalization_1_1/gamma/Adam_1 (float32) [64]
tensor: batch_normalization_1_1/moving_mean (float32) [64]
tensor: batch_normalization_1_1/moving_variance (float32) [64]
"
```

P.S : i also did the same with the pretrained model that's provided , but i get the same error





Deuxième Approche



Architecture

1. Input Layer:

- Shape: (32, 128, 1) for grayscale images.

2. Convolutional Layers:

- **Conv1:** 32 filters, (3, 3), SELU activation, padding 'same'.
- **Conv2:** 64 filters, (3, 3), SELU activation.
- **Conv3 & Conv4:** 128 filters, (3, 3), SELU activation.
- **Conv5 & Conv6:** 512 filters, (3, 3), SELU activation, followed by dropout.
- **Conv7 & Conv8:** 512 filters, (3, 3), SELU activation.
- **Conv9 & Conv10:** 256 filters, (3, 3), SELU activation, followed by batch normalization.

3. Max Pooling:

- **Pool4:** (2, 1) size.
- **Pool6:** (2, 1) size.

4. Lambda Layer: Squeeze the dimension.

5. Bidirectional LSTM:

- 5 layers of Bidirectional LSTMs with 128 and 512 units.

6. Dense Layer:

- 128 units with ReLU activation.

7. Output Layer:

- Softmax with $\text{len}(\text{char_list}) + 1$ units.

8. CTC Loss:

- For sequence-to-sequence training.

9. Optimizer:

- Adam with **lr=0.001**, **beta_1=0.9**, **beta_2=0.999**, gradient clipping.

Résultats & Obstacles

→ Training model :

```
=====
====
Total params: 26,924,045
Trainable params: 26,923,021
Non-trainable params: 1,024
-----
```

Résultats & Obstacles

→ Erreur :

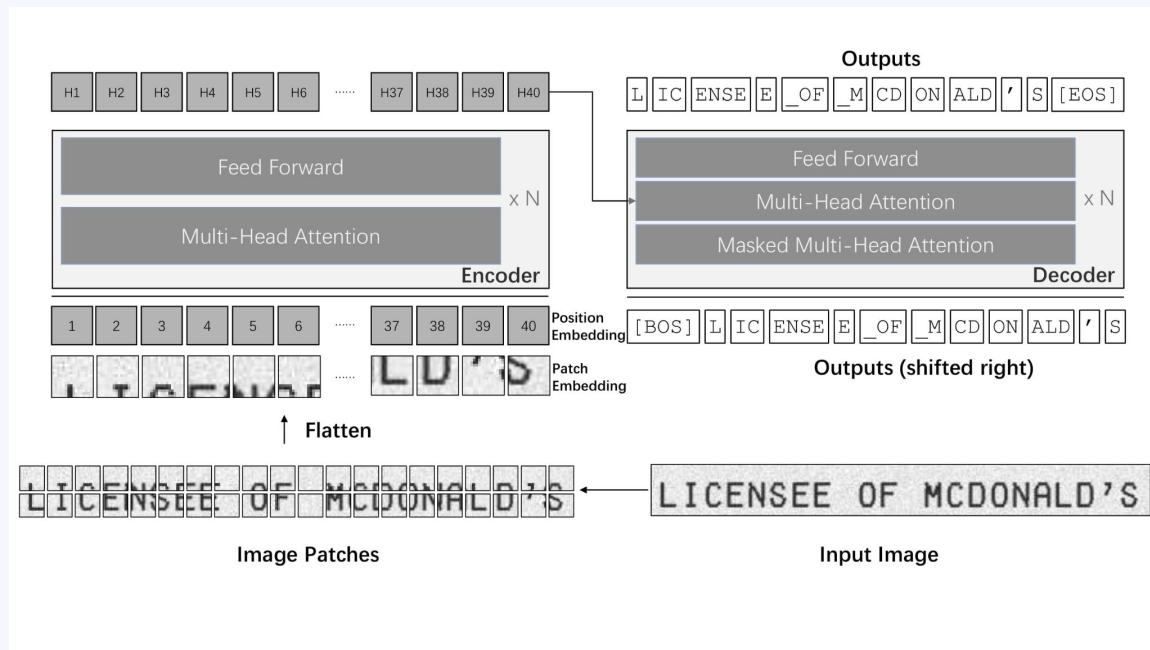
```
ValueError: Cannot assign value to variable 'output_dense/kernel:0': Shape mismatch.  
The variable shape (128, 78), and the assigned value shape (128, 77) are incompatible.
```



Troisième Approche



Architecture :




TrOCR Architecture: Utilizes the Transformer encoder-decoder structure.

- **Encoder:** Processes images into visual feature representations.
- **Decoder:** Converts visual features into textual predictions.

Step 1 : Resizing of Input Image



- The user provides an image containing text.
 - The TrOCR Processor handles **image preprocessing**:
 - **Resizing**: Adjusts the image dimensions to a format compatible with the model (e.g., 224x224 pixels).
 - **Normalization**: Converts pixel values to a range (e.g., $[0,1]$) to standardize the input for the encoder.
 - **Conversion to Tensors**: Transforms the image into a tensor format for the Vision Encoder.
- 

Step 2 : Dividing the image into patches

1- Patch Division:

- The preprocessed image tensor of size $[C, H, W]$ (where $C=3$ channels for **RGB**) is divided into **non-overlapping** square patches of size $P \times P$
- For example, with a patch size $P=16$, the image will yield $H/P * W/P$ patches.

2- Flattening:

- Each patch is flattened into a **1D vector** of size $P \times P \times C$, resulting in $N=H \times W/P^2$ **tokens** (or patches).

Step 3 : Linear Projection of patches

1- Patch Embedding:

- The flattened patch vectors are passed through a linear projection layer (fully connected layer) to project each $P^2 * C$ dimensional vector into a **D-dimensional embedding**.
- The output of this step is a sequence of **N patch embeddings, each of size D**.

2- Adding Learnable Position Embeddings:

- To encode spatial information, a learnable **position embedding vector** is added to each patch embedding.
- These embeddings help the model retain information about the **relative positions** of patches in the image.

Step 4 : Preparing tokens for the Transformer Encoder

1- Adding the Classification Token ([CLS]):

- Before passing the patch embeddings to the Transformer, a special token, known as the **classification token** or **[CLS]**, is added to the **beginning of the sequence**.
- The **[CLS]** token is a learnable vector, initialized during training, and is used to represent the entire input sequence (image patches).
- At the end of the Transformer encoder, the **[CLS]** token will **summarize the global context of the image**.

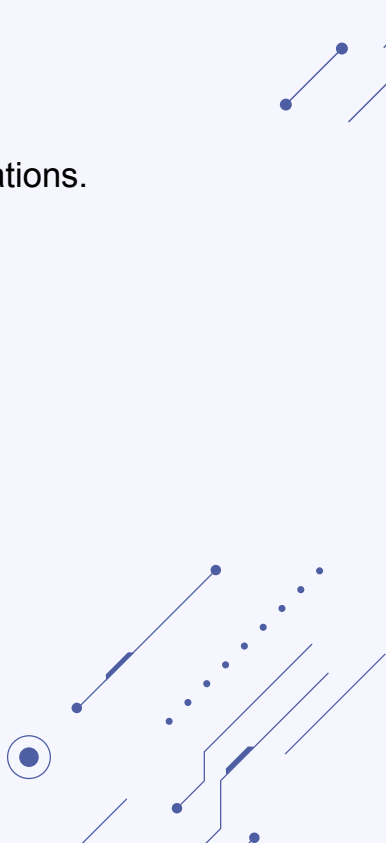
Result:

- If the image produces N patches, the sequence now has **$N+1$ tokens**, where the first token is the **[CLS]**.

Step 5 : Transformer Encoder



The Transformer encoder is the backbone of Vision Transformer (ViT). It processes the input sequence of tokens through multiple **layers** of operations. Each layer consists of two main subcomponents:

1. **Multi-Head Self-Attention (MHSA)**
 2. **Feed-Forward Network (FFN)**
- 

Step 5 : Transformer Encoder

5.1 Multi-Head Self-Attention (MHSA)

The encoder begins by computing self-attention for each token in the sequence. This process allows tokens to interact with one another, capturing both local and global relationships within the image.

Steps in MHSA:

1. Linear Transformation:

For each **token (z_i)** in the **sequence (Z)**, three linear projections are computed:

- **Query (Q)**: Represents the **token's importance when attending to others**.
- **Key (K)**: Represents the **token's relevance when being attended to**.
- **Value (V)**: Represents the **token's content or information to share**.

These projections are computed using learnable weight matrices:

- $Q = W_q \cdot Z$
- $K = W_k \cdot Z$
- $V = W_v \cdot Z$

■ Where W_q , W_k , W_v are **learnable weight matrices**

Step 5 : Transformer Encoder

2- Attention Scores:

For each **pair of tokens** (i, j), an attention score is calculated using the **dot product of their query and key vectors**:

- $\text{Attention}_{ij} = (Q_i \cdot K_j) / \sqrt{D}$

The **scaling factor** \sqrt{D} (where **D is the embedding dimension**) prevents the scores from becoming too large.

3- Softmax and Weighted Sum:

- The attention scores are passed through a softmax function to compute attention weights:
 - $\text{Weights}_{ij} = \text{softmax}(\text{Attention}_{ij})$
- The weights are then used to compute a weighted sum of the value vectors:
 - $\text{Output}_i = \sum_j (\text{Weights}_{ij} \cdot V_j)$

4- Multi-Head Attention:

Multiple attention heads operate in parallel, each independently performing the above steps with its own W_Q , W_K , and W_V matrices. The outputs from all heads **are concatenated and linearly projected back to the original embedding size**.

Step 5 : Transformer Encoder

5.2 Feed-Forward Network (FFN)

After the Multi-Head Self-Attention (MHSA) module, the token embeddings are processed through a **position-wise feed-forward network (FFN)**. This network consists of two linear layers separated by a ReLU activation function, which introduces **non-linearity and enhances the model's ability** to capture complex patterns.

FFN Process:

1. **Linear Transformation with Activation:**
 - The input token embeddings are transformed using a linear layer and a ReLU activation:
$$\text{Intermediate} = \text{ReLU}(W1 \cdot x + b1)$$
2. **Second Linear Transformation:**
 - The result is passed through another linear layer:
$$\text{Output} = \text{Intermediate} \cdot W2 + b2$$
 - $$\text{FFN}(x) = \text{ReLU}(W1 \cdot x + b1) \cdot W2 + b2$$

The FFN significantly boosts the model's expressiveness by applying these **non-linear transformations**, enabling it to learn richer features.

Step 5 : Transformer Encoder

5.3 Residual Connections and Layer Normalization

To **stabilize training**, residual connections are applied around the MHSA and FFN submodules, followed by **layer normalization**.

1. **MHSA Output:**
 - $\text{Output_MHSA} = \text{LayerNorm}(x + \text{MHSA}(x))$
2. **FFN Output:**
 - $\text{Output_FFN} = \text{LayerNorm}(\text{Output_MHSA} + \text{FFN}(\text{Output_MHSA}))$

This approach ensures that the model **preserves original information while learning new features**.

N.B : **BatchNorm** normalizes each feature within a batch of samples, while **LayerNorm** normalizes all features within each sample.

Step 6 : Encoder Output & Passing it to Decoder

After processing through MHSA, FFN, residual connections, and layer normalization, the encoder generates **contextualized embeddings**. These embeddings capture local and global features of the input sequence.

- **Final Encoder Output:** The result is a sequence of embeddings, with the **[CLS] token** summarizing the global context.
- **Passing to Decoder:** These contextualized embeddings are passed to the **decoder**, which uses them for generating output via **encoder-decoder attention**. The decoder attends to these embeddings, alongside previously generated tokens, to produce the final output sequence.

This enables the decoder to generate relevant and coherent outputs based on the encoded information.

Step 7 : Decoder process & output

The decoder uses the encoder's contextualized embeddings and previously generated tokens to produce the output sequence.

- **Masked MHSA:** Attends to previous tokens, ensuring sequential generation.
- **Encoder-Decoder Attention:** Focuses on relevant encoder outputs for generating the next token.
- **FFN:** Applies non-linear transformations to enhance token representations.

The decoder generates a **probability distribution for each token**, predicting the next token until the sequence is complete, producing the final output.

Model Pre-Trained on which datasets

→ IITK-5K :

A dataset for text recognition in images, containing 5,000 images of scene text in natural environments.



Model Pre-Trained on which datasets

→ MJSynth :

A synthetic dataset with text rendered on images in a variety of fonts and layouts to simulate real-world text recognition challenges.

(a)



(b)



Model Pre-Trained on which datasets

→ **MJSynth & SynthText:**

A synthetic dataset with text rendered on images in a variety of fonts and layouts to simulate real-world text recognition challenges.

(a)



(b)



Model Pre-Trained on which datasets

→ FUNSD:

A dataset containing scanned forms with noisy layouts, used for document structure recognition and understanding.

60-08/01 10:17 0211 310 9108 LORILLARD *** NYO 515 GEZ @H01/001

SPECIAL PROMOTION EVALUATION

2 13 5/8/95
AREA/REGION/DIVISION DATE
PROMO # 25 PROMOTIONAL PERIOD: June/July
ITEM/BRAND: Sunglasses / Menport
SCOPE: AREA REGION DIVISION X OTHER
*EXPLAIN: Special Emphasis Call

CHAIN ACCEPTANCE: N/A POOR FAIR GOOD EXCELLENT
INDEPENDENT ACCEPTANCE: POOR X FAIR GOOD EXCELLENT
CONSUMER ACCEPTANCE: POOR X FAIR GOOD EXCELLENT
EFFICIENCY RATING: POOR X FAIR GOOD EXCELLENT

COMMENTS: This item was perceived by retailers and consumers as a low quality item. Several Sales Reps reported that on occasion our consumers would pass up the deal or leave the glasses in the store.

#ITEMS/DEALS RECEIVED: 28728
WERE QUANTITIES APPROPRIATE? X YES NO
*EXPLAIN:
SHOULD PROMOTION BE REPEATED? X YES NO
IF NO, EXPLAIN:
IF YES, CAN IT BE IMPROVED? The Promotion can be improved by upgrading the quality of the sunglasses.

01/24/02

60-08/02 10:17 0211 310 9108 LORILLARD *** NYO 515 GEZ @H01/001

SPECIAL PROMOTION EVALUATION

2 13 5/8/95
AREA/REGION/DIVISION DATE
PROMO # 25 PROMOTIONAL PERIOD: June/July
ITEM/BRAND: Sunglasses / Menport
SCOPE: AREA REGION DIVISION X OTHER
*EXPLAIN: Special Emphasis Call

CHAIN ACCEPTANCE: N/A POOR FAIR GOOD EXCELLENT
INDEPENDENT ACCEPTANCE: POOR X FAIR GOOD EXCELLENT
CONSUMER ACCEPTANCE: POOR X FAIR GOOD EXCELLENT
EFFICIENCY RATING: POOR X FAIR GOOD EXCELLENT

COMMENTS: This item was perceived by retailers and consumers as a low quality item. Several Sales Reps reported that on occasion our consumers would pass up the deal or leave the glasses in the store.

#ITEMS/DEALS RECEIVED: 28728
WERE QUANTITIES APPROPRIATE? X YES NO
*EXPLAIN:
SHOULD PROMOTION BE REPEATED? X YES NO
IF NO, EXPLAIN:
IF YES, CAN IT BE IMPROVED? The Promotion can be improved by upgrading the quality of the sunglasses.

01/24/02

→ **SROIE :**

[illegible]

Implementation :

1- Entraînement sur Kaggle :

- a. J'ai utilisé **Kaggle Kernels** pour utiliser le modèle pre-trained grâce à l'accès gratuit à des GPUs puissants. Cela m'a permis de surmonter les limitations de ma machine locale.

2- Flask et Ngrok pour la transcription :

- b. **Flask** : J'ai développé une API REST avec Flask. Cette API reçoit une image via une requête HTTP, l'analyse à l'aide du modèle TrOCR et renvoie la transcription du texte extrait. Flask permet de déployer facilement des services web légers pour ce genre de tâche.
- c. **Ngrok** : Pour rendre l'API accessible en ligne, j'ai utilisé **Ngrok**. Ngrok crée un tunnel sécurisé entre mon serveur local et internet, générant une URL publique. Cela me permet d'exposer localement mon serveur Flask à l'extérieur et d'accéder à l'API depuis n'importe où, sans avoir besoin de configurer un serveur distant.

3- Réception par React :

- A. L'application **React** permet à l'utilisateur de télécharger une image, envoie la requête à l'API Flask, et affiche la transcription obtenue.



Demo

The image features a minimalist design with the word "Demo" in a large, dark blue, sans-serif font. Below the text is a horizontal line with a small dot at each end. The background is a light blue gradient, decorated with various abstract geometric elements: two parallel diagonal lines with a small circle at the end in the top right; a single diagonal line with a dot at the end in the middle right; and a complex arrangement of lines, dots, and a small circle in the bottom right corner.



Implementation dans Raspberry Pi 5



The background features abstract geometric elements in a muted blue color. In the top right, there are two parallel diagonal lines and a small circle with a dot inside. Further right, another set of parallel lines is accompanied by two dots. In the bottom right corner, a more complex arrangement includes a circle with a dot, several parallel lines, and a series of dots forming a diagonal path. A horizontal line with dots at its ends is positioned below the main text.

Demo



Conclusion





Merci ! :D

