Practical work 5:
Transformer Encoder-Decoder, CNN-Transformer, and Vision Transformer
for digit sequence images recognition

All along this classroom work we will train then test a Transformer encoder-decoder Network for the recognition of handwritten digit sequences. We will use the same dataset as for the RNN-CTC available in the file "MNIST_5digitsDifficile.pkl".

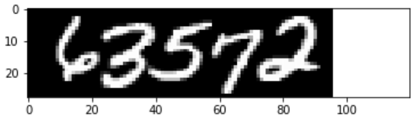


Figure 1: One example from the MNIST-5 digits difficile

This classroom work was developed from the tutorial available at this link:
https://towardsdatascience.com/a-detailed-guide-to-pytorchs-nn-transformer-module-c80afbc9ffb1

The transformer encoder-decoder architecture

The Encoder-Decoder architecture depicted on figure below achieves the encoding of the sequence of pixel columns of size *28* into a sequence of embeddings of size *hidden_size* of same length. Then, the decoder starts predicting the next output symbol, knowing the input representation provided by the encoder and its previous output predictions Y_{t-1}.
The encoder-decodeur bloc is implemented in the pytorch library through the class *pytorch.nn.Transformer()*. The input of this bloc are input embeddings (*x_embedding*) and output embeddings (y*_embedding*) that need to be positional encoded respectively.
The architecture is defined by :

- *hidden_size :* size of the internal representations, both on the encoder side and decoder side.
- *Num_layers* : number of encoder and decoder transformer layers (they can be different on the encoder and decoder side)
- *Num_heads* : number of heads in the encoder and decoder side.
- *Dropout*: the fraction of droped out (freezed) parameters at each update of the parameter. This prevents from over fitting the training dataset during training.
- *Num_classes*: number of classes to be discriminated. Notice that in addition to the 10 digit classes two additional tokens are required. The START_TOKEN (10) which is a token that provides the decoder with the appropriate context to start decoding, and the END_TOKEN which allows to stop the decoding process once it has been predicted. All in all 12 classes are required for the task at end.

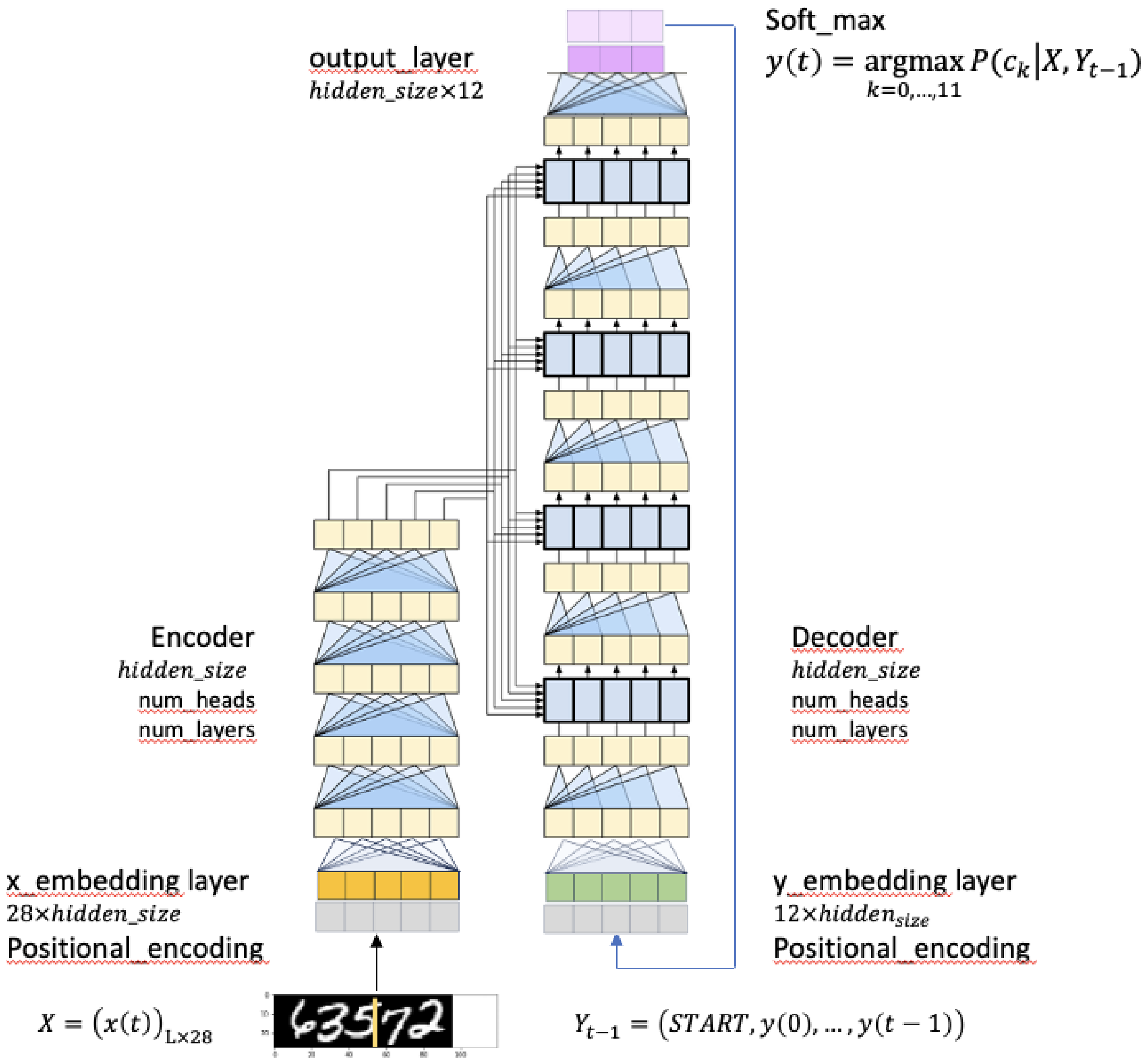


Figure 2: the encoder decoder architecture.

Working with the Google Colab environment

For this classroom work you need the three following files that can be downloaded from your environement:

```
TRANFORMER.py
Main-Transformer.ipynb
MNIST_5digitsDifficile.pkl
```

They must be put into your google drive ***ColabNotebooks*** directory. The trained model files are store on your local Colab directory : *"/content/sample_data"*

1- Accessing your Google Drive files

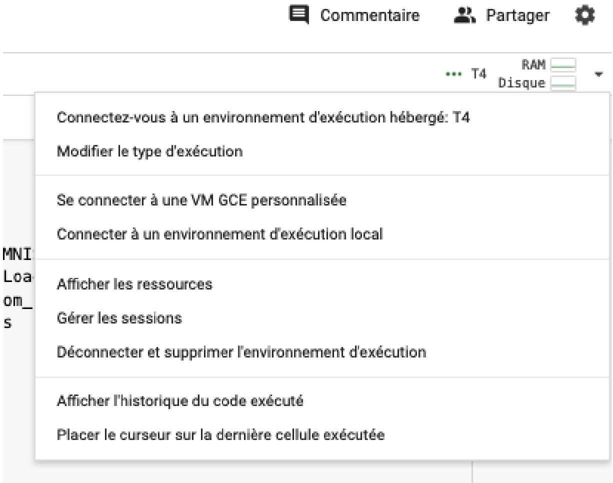
Your drive directory can be mounted on your Colab local directory *"/content/drive"* through the following command lines included in the `Main-Transformer.ipynb` file.

```
from google.colab import drive
!mkdir -p drive
drive.mount('/content/drive',force_remount=True)
#!cd /content/drive/MyDrive/ColabNotebooks
```

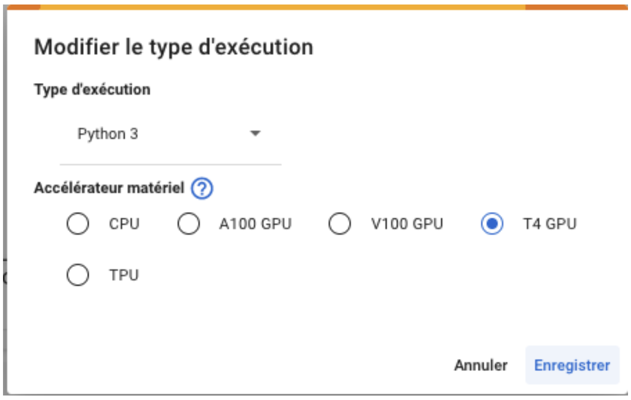
2- Configuration of the Colab processing power

Training transformer architecture requires a lot of training data as well as powerful computing infrastructure. In your Colab environment you can ask for more computation power for free if available. At least Colab may provide you with T4 Nvidia GPU if some of them are available. Also note that this free computing pawer is accessible for a limited amount of time (2 hours or so). Therefore, you may get disconnected from this computing environment at any time without being warned. Afterwards you may come back to computing with a CPU.

To ask for GPU select "Modifier le type d'exécution"


Commentaire
Partager
T4
RAM
Disque
Connectez-vous à un environnement d'exécution hébergé: T4
Modifier le type d'exécution
Se connecter à une VM GCE personnalisée
Connecter à un environnement d'exécution local
Afficher les ressources
Gérer les sessions
Déconnecter et supprimer l'environnement d'exécution
Afficher l'historique du code exécuté
Placer le curseur sur la dernière cellule exécutée
MNI
Loa
om_
s

Then select your processor



Modifier le type d'exécution

Type d'exécution

Python 3

Accélérateur matériel

○ CPU ○ A100 GPU ○ V100 GPU ● T4 GPU

○ TPU

Accessing the TRANSFORMER module from your Google Drive

```
sys.path.append('/content/drive/MyDrive/ColabNotebooks')
import TRANSFORMER
```

Your ColabNoteBooks directory should content the TRANSFORMER.py module as well as the Main-Transformer.ipynt notebook.

Architecture implementation details with Pytorch

The file **TRANFORMER.py** is the module code that implements the encoder-decoder architecture. It is composed of the following methods and classes.

Transformer class

```
        self.positional_encoding_layer = PositionalEncoding(dim_model = self.hidden_size,
                                                            dropout_p = self.dropout,
                                                            max_len = self.max_length)
        self.x_embedding = nn.Linear(config['input_features'],config['hidden_size'])
        self.y_embedding = nn.Embedding(config['num_classes'],config['hidden_size'])
        self.transformer = nn.Transformer(d_model=config['hidden_size'],
                                                      nhead=config['num_heads'],
                                                      num_encoder_layers =config['num_layers'],
                                                      num_decoder_layers =config['num_layers'],
                                                      dim_feedforward = config['hidden_size'],
                                                      dropout=config['dropout'],
                                                      batch_first = True)
        # Output layer for text prediction
        self.output_layer = nn.Linear(config['hidden_size'], config['num_classes'])
```

```
    def forward(self, x,y,y_output_mask,src_key_padding_mask,tgt_key_padding_mask):
        # Embedding + positional encoding - Out size = (batch_size, sequence length, dim_model)
        #x = self.x_embedding(x) * math.sqrt(self.hidden_size)
        x = self.x_embedding(x)
        x = self.positional_encoding_layer(x)
        y = self.y_embedding(y) * math.sqrt(self.hidden_size)
        y = self.positional_encoding_layer(y)
        # Transformer blocks - Out size = (sequence length, batch_size, num_tokens)
        transformer_out = self.transformer(x, y,
                                           tgt_mask = y_output_mask,
                                           src_key_padding_mask=src_key_padding_mask,
                                           tgt_key_padding_mask=tgt_key_padding_mask)
        output = self.output_layer(transformer_out)
        return output
```

Other methods

```
def train_loop(dataloader, model, loss_fn, optimizer):
    print("Training loop...")
    size = len(dataloader.dataset)
    nb_batches = len(dataloader)
    epoch_loss = 0
    model.train()

    for batch, (X, y,X_l,y_l) in tqdm(enumerate(dataloader)):
        X = X.to(model.DEVICE)
        y = y.to(model.DEVICE)
        y_input = y[:,:-1]# on retire le END_TOKEN car nest jamais traité en entrée
        y_output = y[:,1:]# on ne cherche jamais à predire le START_TOKEN

        l = y_input.size(1) # la longueur maximale d'un élément du batch
        y_output_mask = model.get_tgt_mask(l).to(model.DEVICE)
        x_padding_mask = (X[:,:,0] == model.pad_idx).to(model.DEVICE)
        y_input_padding_mask = (y_input == model.pad_idx).to(model.DEVICE)

        # Compute prediction and loss
        pred = model(X.float(), y_input,
                     y_output_mask,
                     x_padding_mask,
                     y_input_padding_mask)
        pred = pred.permute(1, 2, 0)
        y_output = y_output.permute(1, 0)
        loss = loss_fn(pred, y_output)
        epoch_loss += loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return epoch_loss / nb_batches
```

```
def valid_loop(dataloader, model, loss_fn):
.........................
    return valid_loss
```

The file **Main-Transformer.py**

```
if __name__ == '__main__':

    TRAINING = True  # Training if True Testing otherwise
    SHOW = False

    device = torch.device("cpu")
    if torch.backends.mps.is_available():
        device = torch.device("mps")
    elif torch.cuda.is_available():
        device = torch.device("cuda")
    print("Device :",device)

    config = {
        ‘w_width':5,
        'w_stride':5,
        'input_features':140,
        'batch_size':256,
        'num_epochs':1500,
        'learning_rate':1e-4,
        'num_classes':12, # START_TOKEN + END_TOKEN
        'hidden_size':64,
        'num_heads':4,
        'num_layers':8,
        'dropout':0.5,
        'pad_idx':-1,
        'max_length':140,
        'START_TOKEN':10,
        'END_TOKEN':11,
    }
```

```
    dir_name = "/content/sample_data/"
    model_name = 
"Transformer_"+str(config['input_features'])+"_"+str(config['hidden_size'])+"_"+str(config['num_head
s'])+"_"+str(N_train)
    model_name =dir_name+model_name
```

```
    my_transformer = TRANSFORMER.Transformer(config,device).to(device)
    loss_fn = torch.nn.CrossEntropyLoss(ignore_index=config['pad_idx'],reduction='mean')
    optimizer = torch.optim.Adam(my_transformer.parameters(),lr=config['learning_rate'])

    train_loss = []
    valid_loss = []
    for e in range(config['num_epochs']):
        train_loss.append(TRANSFORMER.train_loop(train_dataloader,
                                                 my_transformer,
                                                 loss_fn,
                                                 optimizer))
```

Tranformer Encoder-Decoder (T)

- Train a transformer with a (w_width = 5, w_stride = 5) sliding window on a small dataset (train = 10 000, valid = 1000). Observe the problems encountered. Convergence, and performance.
- Train a transformer with a (w_width = 5, w_stride = 5) sliding window on a large dataset (train = 54 000, valid = 6000). Observe the problems encountered. Convergence, and performance.

Hybride CNN-Transformer architecture (CNN-T)

- Implement a modified version of the transformer which introduces some CNN encoding layers before the transformer bloc.

- Show how CNN layers help convergence by providing a better embedding of the input image.

Vision Transformer architecture (ViT)

- Implement your own vision transformer with a square $n \times n$ pixels sliding window and appropriate 2D stride, with a 2D positional encoding. Compare the performance obtained with the hybride CNN-Transformer one.