UNICAMP

MT862

Tópicos em Tratamento Matemático de Imagens e Inteligência Computacional

Neural Machine Translation with Attention

Projeto 2

Professor: João Batista Florindo

Natália França dos Reis
Vitor Hugo Miranda Mourão

Parte I: Introdução e Metodologia

1. Introdução

A tradução automática, tem evoluído significativamente com a popularização das redes neurais profundas. Duas inovações chave nesse campo são as Redes Neurais Recorrentes *Long Short-Term Memory* (LSTM) e a Arquitetura de Atenção.

A arquitetura de Atenção foi introduzida para melhorar a tradução automática. Ela permite que o modelo se concentre em partes específicas da entrada ao produzir cada palavra da saída. Isso é especialmente útil em tradução, onde cada palavra na saída pode estar relacionada a diferentes partes da entrada. A atenção ajuda a resolver o problema de "*bottleneck*", em redes neurais, especialmente em modelos de sequência para sequência como os usados em tradução automática. Isso se refere à limitação de ter que codificar toda a informação de uma longa sequência de entrada (como uma frase ou parágrafo) em um único vetor de estado fixo. Este vetor é então usado pelo decodificador para gerar a sequência de saída, permitindo que o decodificador tenha acesso a todo o texto de entrada, em vez de apenas a uma representação fixa [1].

Combinando LSTMs e a arquitetura de atenção, os modelos de Rede Neural de Tradução (NMT, do inglês, *Neural Machine Translation*) podem capturar a semântica de textos longos e traduzi-los com maior precisão.

O modelo sequência-a-sequência com LSTMs e atenção é composto por duas partes principais: o codificador, que processa o texto de entrada e cria uma representação contextual, e o decodificador, que gera a tradução, focando seletivamente em diferentes partes do texto de entrada graças ao mecanismo de atenção [1].

Neste projeto, adotaremos a abordagem baseada em arquiteturas de tradução para treinar uma rede que possa identificar e converter datas de diversos formatos escritos por humanos (em inglês), para um formato padronizado. A tarefa central é ensinar a rede a traduzir com cada vez mais precisão as datas estabelecidas. Este procedimento iterativo permitirá uma compreensão mais profunda sobre como diferentes durações de treinamento impactam a capacidade da rede de aprender com eficácia.

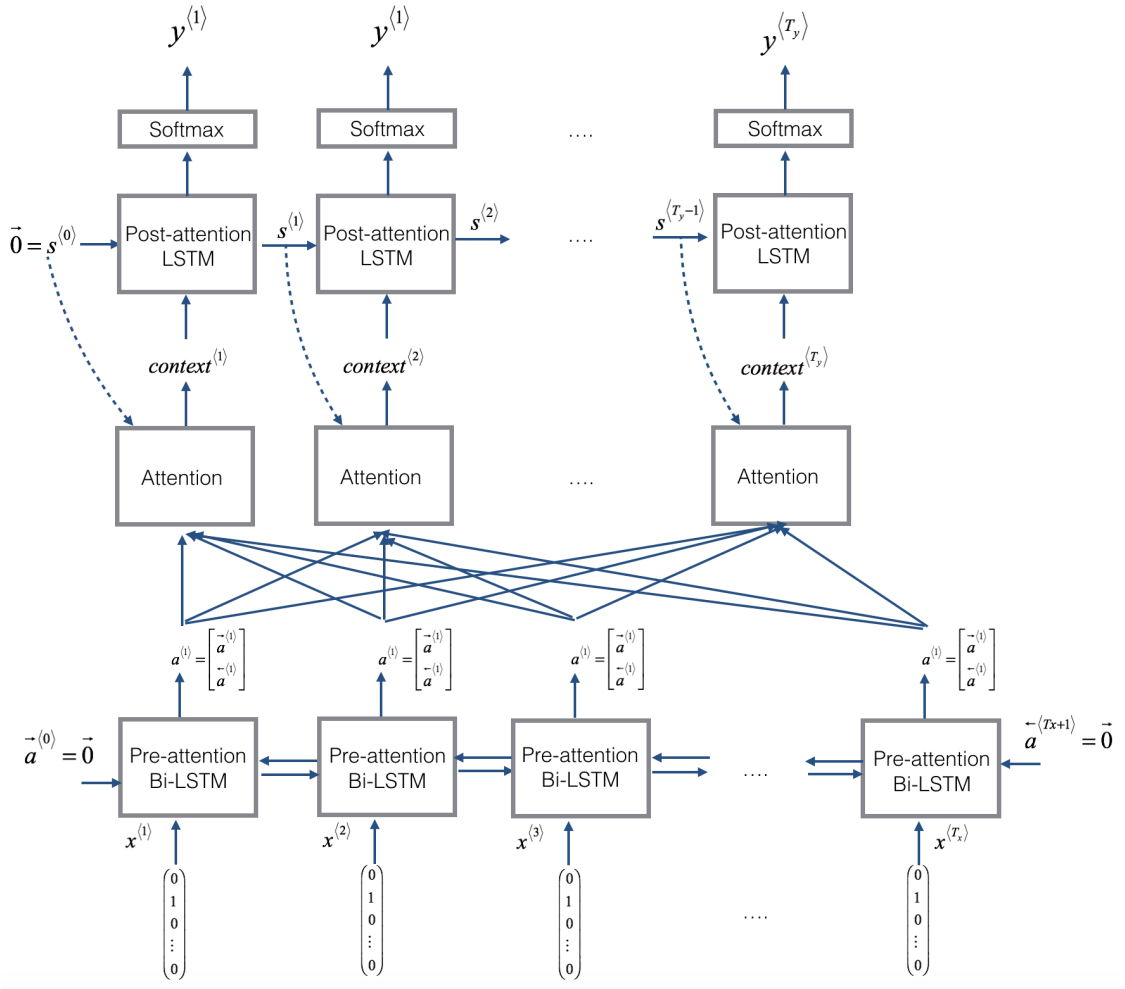
2. Arquitetura de Tradução

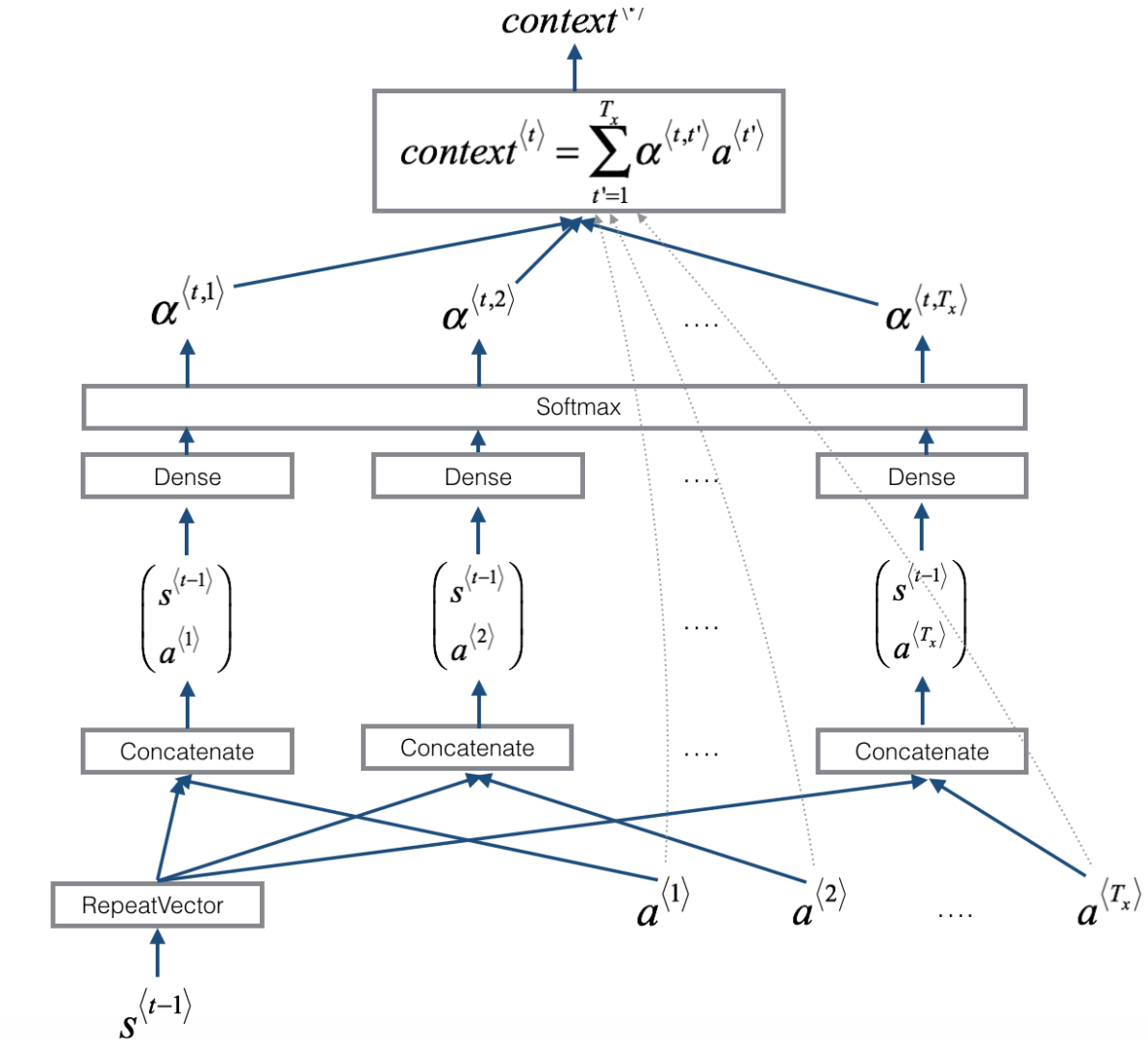
A NMT adota uma arquitetura codificador-decodificador, que consiste em duas redes neurais recorrentes. De acordo com [2], a rede do codificador modela a semântica da frase fonte e a transforma em representação vetorial, a partir da qual a rede decodificadora gera a tradução alvo. O mecanismo de atenção tornou-se um componente indispensável em NMT, que permite ao modelo compor dinamicamente

a representação da fonte para cada intervalo de tempo durante a decodificação, em vez de uma representação única e estática. Especificamente, o modelo de atenção mostra quais palavras-fonte o modelo deve se concentrar para prever a próxima palavra-alvo.

A arquitetura da rede de tradução usada neste projeto está representada na imagem a abaixo:

Figura 1 - Arquitetura da rede neural de tradução com atenção.





Fonte: https://jmyao17.github.io/Machine_Learning/Sequence/attention.html

2.1 NMT

Neste projeto, utilizamos uma arquitetura que integra LSTM bidirecional com mecanismos de atenção. Este design é crucial para entender o contexto complexo e garantir uma tradução mais precisa e contextualizada. Seguem abaixo mais detalhes sobre a composição dessa rede.

Pre-attention Bi-LSTM

A base da nossa arquitetura é um LSTM bidirecional (Bi-LSTM), que processa a entrada textual em ambas as direções. Esta camada é fundamental para capturar as dependências de contexto, tanto anteriores quanto posteriores, para cada ponto no tempo na sequência de entrada. Isso nos permite ter uma representação mais rica e completa do texto de entrada.

Mecanismo de Atenção
Acima da camada Bi-LSTM, incorporamos um mecanismo de atenção. Este componente é importante para conectar as camadas de pré-atenção e pós-atenção. Ele calcula pesos de atenção que são aplicados às saídas do Bi-LSTM, destacando partes da entrada que são mais relevantes para a previsão de cada palavra na sequência de saída. Isso permite a concentração em partes específicas do texto de entrada, melhorando significativamente a qualidade da tradução.

Post-attention LSTM
Após o mecanismo de atenção, temos a camada "Post-attention LSTM". Esta camada recebe o vetor de contexto, formado a partir das saídas ponderadas do Bi-LSTM, e o estado oculto anterior para gerar o novo estado oculto. Essa etapa é fundamental para que a rede direcione seu foco para as partes mais relevantes do texto de entrada ao decidir a saída atual.

Softmax
A última camada da arquitetura é a Softmax. Aplicamos esta camada a cada saída do Post-attention LSTM para obter uma distribuição de probabilidade sobre o conjunto de tokens possíveis para a próxima palavra na sequência de saída. Isso ajuda a selecionar a palavra mais provável a cada passo.

2.2 Mecanismo de Atenção
RepeatVector* e *Concatenate
Para alinhar os estados ocultos do decodificador com a sequência de tempo da entrada, utilizamos a operação *RepeatVector*. Isso permite que cada estado oculto do decodificador seja comparado com todos os estados ocultos codificados da entrada. Após isso, realizamos a concatenação de cada estado oculto codificado com o estado oculto repetido do decodificador, preparando-os para o cálculo da pontuação de atenção.

***Dense* e *Softmax* para Pontuação de Atenção**
Os estados concatenados são então passados por uma camada totalmente conectada (densa), que aprende a projetar esses estados em um espaço útil para calcular a pontuação de atenção. As pontuações de atenção são normalizadas usando a função softmax para cada passo de tempo, resultando em pesos de atenção que indicam a importância relativa de cada parte da entrada.

Cálculo do Vetor de Contexto
Finalmente, o vetor de contexto para cada momento no tempo é calculado como uma soma ponderada dos estados ocultos codificados. Os pesos de atenção servem como coeficientes de ponderação, permitindo que a rede concentre-se nas informações mais pertinentes da entrada ao produzir a saída.

3. Metodologia

A Rede Neural de Tradução foi implementada de acordo com trechos de código passados em sala de aula com algumas adaptações que permitem uma análise mais detalhada dos resultados do algoritmo. Uma dessas alterações foi a criação de um conjunto de dados de teste com 1000 entradas utilizando a biblioteca *“faker”*, esse conjunto de dados tem a mesma estrutura que o conjunto de dados usado como treinamento, o que difere o conjunto de teste do treino é que são gerados aleatoriamente mil outras diferentes datas que serão inseridas no modo preditivo do modelo já treinado, a saída então é comparada com as instâncias *“corretas”* também geradas pela biblioteca *faker.*

Com a possibilidade de gerar dados de teste foram realizados diferentes treinamentos da NMT, variando a quantidade de épocas e fixando os conjuntos de dados de treino e de teste. Dessa forma, foi possível gerar um gráfico da relação acurácia por número de épocas. Além do gráfico obtido, também foi possível identificar em cada iteração do algoritmo em diferentes épocas onde ocorriam os erros de predição.

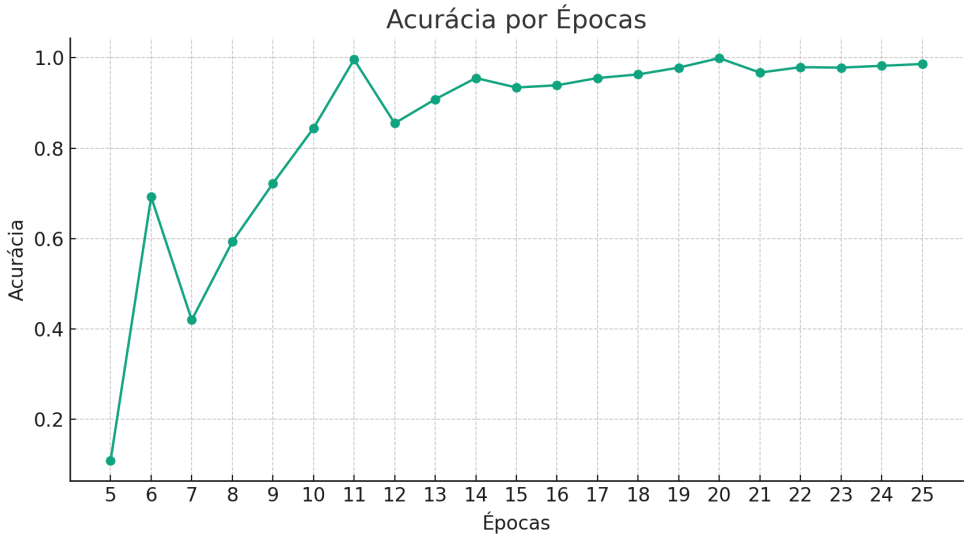
A análise de mapa de atenção foi incorporada à metodologia para fornecer uma visualização concreta do mecanismo de atenção da Rede Neural de Tradução. Essa técnica permite observar as áreas de foco do modelo durante a predição, destacando quais partes da entrada são consideradas mais relevantes para cada passo da sequência de saída. Ao implementar essa metodologia, foi possível não apenas rastrear a progressão do aprendizado ao longo das épocas de treinamento, mas também compreender como as diferentes partes das datas de entrada influenciam a geração das datas de saída.

Parte II: Contextualização e Análise de Resultados

1. Resultados e Discussão

Através de diversas iterações variando o número de épocas obtemos a curva presente na **Figura 2** a seguir.

Figura 2 - Progressão da Acurácia ao longo de 25 Épocas de Treinamento de um Modelo de Rede Neural com Mecanismos de Atenção para Tradução de Datas.



Além da análise de acurácia do modelo dependente do número de épocas utilizadas no treinamento, a **Tabela 1** a seguir apresenta alguns exemplos de erros cometidos durante a predição do modelo. No anexo I é apresentada a **Tabela 2** que detalha os valores obtidos nessa fase de análise.

Tabela 1 - Comparação entre as expressões de datas corretas com as preditas pelo modelo de rede neural.

Fonte	Expressão Correta	Expressão Predita
tuesday march 3 2015	2015-03-03	2013-03-33
1/31/70	1970-01-31	1970-12-11
monday december 3 2001	2001-12-03	2010-12-33
oct 21 1990	1990-10-21	1990-10-22
13 oct 2022	2022-10-13	2022-11-13
1/19/01	2001-01-19	2011-09-11
08.07.88	1988-08-08	1988-07-08
18 06 22	2022-06-18	2022-04-17

Podemos observar que ao aumentar o número de épocas a acurácia do modelo aumenta consideravelmente, sendo que à partir de 13 épocas já obtemos uma acurácia maior que 90% e com 17 épocas a acurácia já ultrapassa os 95%. Como nossa rede de tradução leva a um formato de máquina pré-especificado podemos comparar os resultados obtidos com a verdadeira tradução e verificar os

erros. O primeiro caso, é que o modelo previu “33” como sendo a tradução correta de um dia, obviamente isto está errado e uma possível verificação seria passar filtros no output que verificam datas não existentes (dias maiores que 31, meses maiores que 12, etc). Também se verificam erros de datas com números próximos (12 quando deveria ser 11, 22 quando deveria ser 21, etc) e repetição/alternação de caracteres. Esses últimos casos são mais difíceis de identificar quando a implementação já é automática, fazendo com que para que o modelo seja aceitável a sua acurácia deva ser extremamente elevada.

Nas **Figuras 3 e 4** a seguir, apresentamos um gráfico de mapa de atenção, uma ferramenta visual essencial que ilustra como o nosso modelo de rede neural com mecanismo de atenção pondera diferentes partes da entrada ao realizar a tarefa de tradução de datas. O gráfico destaca as conexões entre a sequência de entrada e a sequência de saída, mostrando as áreas onde o modelo concentra sua “atenção” para prever cada parte da data corretamente. As intensidades das cores no mapa indicam o peso da atenção, com tons mais escuros representando um foco maior. Este mapa não apenas fornece *insights* sobre o processo interno de decisão do modelo, mas também serve como um diagnóstico para entender e melhorar seu desempenho, identificando onde o modelo está alinhado com as expectativas e onde podem ocorrer desvios como mostrado nas **Figuras 3 e 4.**

Ambas as figuras foram geradas a partir do mesmo código. Porém, a **Figura 3** mostra o caso onde o modelo previu corretamente a saída, enquanto a **Figura 4** mostra o mapa de atenção para o caso onde o modelo previu erroneamente a saída. Como dito anteriormente ambas as figuras foram geradas pelo mesmo código, mas em com treinamentos realizados em separado (Reiniciando o ambiente de programação). Logo observamos que também há um efeito “aleatório” sobre o treinamento do modelo, que pode levar a resultados diferentes na predição.

Figura 3 - Mapa de atenção que prevê corretamente a saída da rede.

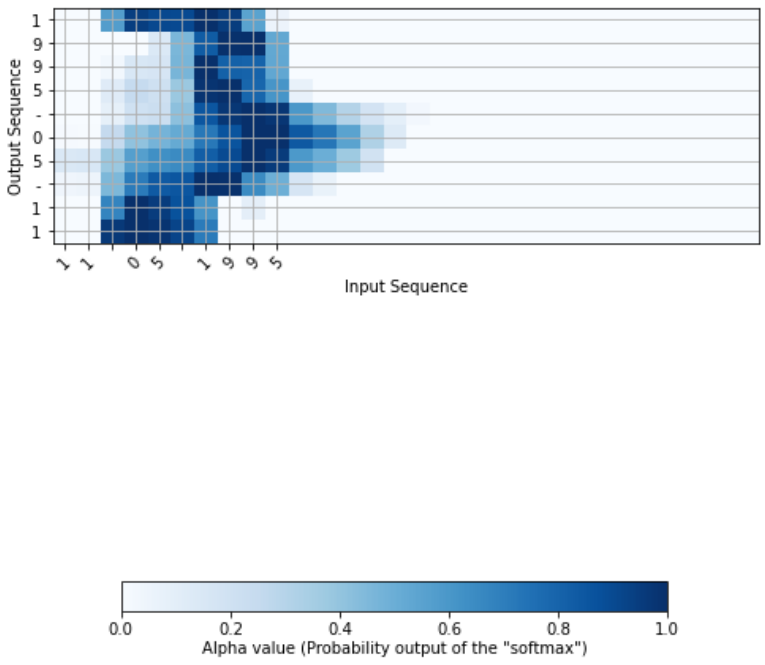
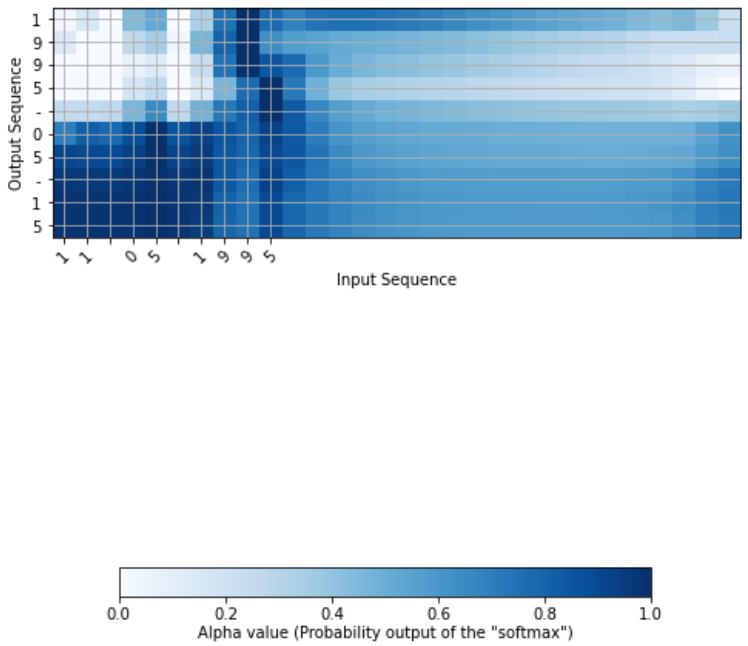


Figura 4 - Mapa de atenção que prevê erroneamente a saída da rede.



Também podemos observar na **Figura 3** que existem muitos espaços em branco o que revela uma esparsidade dos pesos de atenção normalizados, já na **Figura 4** há um certo equilíbrio nos pesos de atenção dos primeiros caracteres de entrada, o que pode ter causado o erro na predição. De qualquer forma, os mapas de atenção produzidos não parecem ser adequados ao proposto no projeto e os autores não conseguiram identificar onde o problema está acontecendo.

2. Conclusão

Tarefas complexas de tradução podem ser impraticáveis de se resolver por programação baseada em regras, uma alternativa para resolução dessas tarefas é o uso de redes neurais. Essas por sua vez também enfrentam dificuldades quando lidam com longas cadeias de caracteres. Mecanismos de atenção auxiliam as redes neurais a reter informações de contexto e gerar traduções mais acuradas.

Neste trabalho identificamos o aumento rápido da acurácia de uma rede neural de tradução usando um mecanismo de atenção, mostrando que mesmo em poucas épocas o algoritmo atinge níveis de acurácia satisfatórios para diversas aplicações.

Foram apresentados mapas de atenção que auxiliam na visualização das relações entre valores de entrada e saída da rede neural. Porém, o presente trabalho não identificou a origem de alguns problemas relacionados aos cálculos e/ou representação dos pesos de atenção.

De modo geral, o uso de redes neurais de tradução mostra uma abordagem rápida e eficiente para tarefas de tradução de entrada de datas humanas para o

formato de máquina. Em trabalhos futuros a tarefa pode ser dividida em três: i) identificação do dia, ii) identificação do mês e iii) identificação do ano nas datas e concatenação no formato desejado, dessa forma os modelos poderão se “especificar” mais ainda numa tarefa e gerar resultados mais acurados. Contudo, essa abordagem não seria recomendável para uma tradução mais generalizada.

Referências

[1] BAHDANAU, Dzmitry; CHO, Kyunghyun; BENGIO, Yoshua. Neural machine translation by jointly learning to align and translate. **arXiv preprint arXiv:1409.0473,** 2014.

[2] HOU, Long; ZHANG, Jiajun; ZONG, Chengqing. Look-ahead attention for generation in neural machine translation. In: **Natural Language Processing and Chinese Computing: 6th CCF International Conference, NLPCC 2017, Dalian, China, November 8–12, 2017, Proceedings 6**. Springer International Publishing, 2018. p. 211-223.

ANEXO I

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 18 13:00:52 2023

@author: Natalia França dos Reis & Vitor Hugo Miranda Mourao
"""

from  tensorflow.keras.layers  import  Bidirectional,  Concatenate,
Dot, Input, LSTM
from tensorflow.keras.layers import RepeatVector, Dense, Activation
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Model
import tensorflow.keras.backend as K
import tensorflow as tf
import numpy as np
import pickle

from faker import Faker
import random
from tqdm import tqdm
from babel.dates import format_date
import matplotlib.pyplot as plt

def softmax(x, axis=1):
    """Softmax activation function.
    # Arguments
        x : Tensor.
         axis: Integer, axis along which the softmax normalization
is applied.
    # Returns
        Tensor, output of softmax transformation.
    # Raises
        ValueError: In case `dim(x) == 1`.
    """
    ndim = K.ndim(x)
    if ndim == 2:
        return K.softmax(x)
    elif ndim > 2:
        e = K.exp(x - K.max(x, axis=axis, keepdims=True))
        s = K.sum(e, axis=axis, keepdims=True)
        return e / s
    else:
         raise ValueError('Cannot apply softmax to a tensor that is
1D')

def string_to_int(string, length, vocab):
```

```
    """
    Converts all strings in the vocabulary into a list of integers representing the positions of the
    input string's characters in the "vocab"

    Arguments:
    string -- input string, e.g. 'Wed 10 Jul 2007'
    length -- the number of time steps you'd like, determines if the output will be padded or cut
    vocab -- vocabulary, dictionary used to index every character of your "string"

    Returns:
        rep -- list of integers (or '<unk>') (size = length) representing the position of the string's character in the vocabulary
    """

    #make lower to standardize
    string = string.lower()
    string = string.replace(',','')

    if len(string) > length:
        string = string[:length]

    rep = list(map(lambda x: vocab.get(x, '<unk>'), string))

    if len(string) < length:
        rep += [vocab['<pad>']] * (length - len(string))

    return rep

def preprocess_data(dataset, human_vocab, machine_vocab, Tx, Ty):

    X, Y = zip(*dataset)

    X = np.array([string_to_int(i, Tx, human_vocab) for i in X])
    Y = [string_to_int(t, Ty, machine_vocab) for t in Y]

        Xoh = np.array(list(map(lambda x: to_categorical(x, num_classes=len(human_vocab)), X)))
        Yoh = np.array(list(map(lambda x: to_categorical(x, num_classes=len(machine_vocab)), Y)))

    return X, np.array(Y), Xoh, Yoh

# Define the file paths
base_path_n = "C:/Users/usuario/Desktop/DL_project/"
```

```
base_path_v = "C:/Users/vitor/Downloads/Pos grad/Doutorado/MT862 -
Deep Learning/projeto 2/files/"
file_paths_n = {
    'dataset':  base_path_n + "dataset.pkl",
    'human_vocab': base_path_n + "human_vocab.pkl",
    'machine_vocab': base_path_n + "machine_vocab.pkl",
    'inv_machine_vocab': base_path_n + "inv_machine_vocab.pkl"
}
file_paths_v = {
    'dataset': base_path_v + "dataset.pkl",
    'human_vocab': base_path_v + "human_vocab.pkl",
    'machine_vocab': base_path_v + "machine_vocab.pkl",
    'inv_machine_vocab': base_path_v + "inv_machine_vocab.pkl"
}

# Load the data from the files in a single line using dictionary
comprehension
data = {name: pickle.load(open(path, 'rb')) for name, path in
file_paths_n.items()}

dataset = data['dataset']
human_vocab = data['human_vocab']
machine_vocab = data['machine_vocab']
inv_machine_vocab = data['inv_machine_vocab']

Tx = 30
Ty = 10

tf.config.list_physical_devices(device_type='GPU')

X, Y, Xoh, Yoh = preprocess_data(dataset,human_vocab,machine_vocab,
Tx, Ty)

##### Attention #####
repeator = RepeatVector(Tx)
concatenator = Concatenate(axis = -1)
densor1 = Dense(10, activation = "tanh")
densor2 = Dense(1, activation = "relu")
activator = Activation(softmax, name = "attention_weights")
dotor = Dot(axes = 1)

def one_step_attention(a, s_prev):
    s_prev = repeator(s_prev)
    concat = concatenator([a,s_prev])
    e = densor1(concat)
    energies = densor2(e)
    alphas = activator(energies)
    context = dotor([alphas,a])
```

```
    return context

n_a = 32
n_s = 64

post_activation_LSTM_cell = LSTM(n_s, return_state = True)
output_layer = Dense(len(machine_vocab), activation = softmax)

def modelf(Tx, Ty, n_a, n_s, human_vocab_size, machine_vocab_size):
    X = Input(shape = (Tx, human_vocab_size))
    s0 = Input(shape = (n_s,), name = "s0")
    c0 = Input(shape = (n_s,), name = "c0")
    s = s0
    c = c0

    outputs = []
    a = Bidirectional(LSTM(n_a,return_sequences = True))(X)

    for t in range(Ty):
        context = one_step_attention(a, s)
         s, _, C = post_activation_LSTM_cell(context, initial_state
= [s,c])
        out = output_layer(s)
        outputs.append(out)
    model = Model(inputs = [X,s0,c0], outputs = outputs)
    return model

model    =    modelf(Tx,    Ty,    n_a,    n_s,    len(human_vocab),
len(machine_vocab))
# model.summary()

opt = Adam(learning_rate = 0.005, beta_1 = 0.9, beta_2 = 0.999,
weight_decay = 0.01)
model.compile(optimizer = opt, loss = "categorical_crossentropy",
metrics = ["accuracy"])

s0 = np.zeros((len(X), n_s))
c0 = np.zeros((len(X), n_s))

outputs = list(Yoh.swapaxes(0,1))
model.fit([Xoh, s0, c0], outputs, epochs = 25, batch_size = 100)

##### Model Prediction ######

fake = Faker()
Faker.seed(1992)
random.seed(1996)
```

```
# Define format of the data we would like to generate
FORMATS = ['short',
           'medium',
           'long',
           'full',
           'full',
           'full',
           'full',
           'full',
           'full',
           'full',
           'full',
           'full',
           'full',
           'full',
           'd MMM YYY',
           'd MMMM YYY',
           'dd MMM YYY',
           'd MMM, YYY',
           'd MMMM, YYY',
           'dd, MMM YYY',
           'd MM YY',
           'd MMMM YYY',
           'MMMM d YYY',
           'MMMM d, YYY',
           'dd.MM.YY']

# change this if you want it to work with another language
LOCALES = ['en_US']

def load_date():
    """
        Loads some fake dates
          :returns: tuple containing human readable string, machine
readable string, and date object
    """
    dt = fake.date_object()

    try:
                                human_readable    =    format_date(dt,
format=random.choice(FORMATS),              locale='en_US')       #
locale=random.choice(LOCALES))
        human_readable = human_readable.lower()
        human_readable = human_readable.replace(',','')
        machine_readable = dt.isoformat()

    except AttributeError as e:
        return None, None, None
```

```
    return human_readable, machine_readable, dt

def load_dataset(m):
    """
        Loads a dataset with m examples and vocabularies
        :m: the number of examples to generate
    """

    human_vocab = set()
    machine_vocab = set()
    dataset = []


    for i in tqdm(range(m)):
        h, m, _ = load_date()
        if h is not None:
            dataset.append((h, m))
            human_vocab.update(tuple(h))
            machine_vocab.update(tuple(m))

    return dataset

m = 1000
dataset_test = load_dataset(m)

EXAMPLES = [example[0] for example in dataset_test]
TARGETS = [example[1] for example in dataset_test]
s00 = np.zeros((1, n_s))
c00 = np.zeros((1, n_s))

correct_predictions = 0

for i, example in enumerate(EXAMPLES):
    print(i,"\n")
    # Convert the raw date string into a one-hot encoded version
    source = string_to_int(example, Tx, human_vocab)
        source = np.array(list(map(lambda x: to_categorical(x,
num_classes=len(human_vocab)), source)))
    source = np.expand_dims(source, axis=0)

    # Predict the output using the model
    prediction = model.predict([source, s00, c00])
    prediction = np.argmax(prediction, axis=-1)

    # Iterate over the Ty dimension to get predictions
    output = [inv_machine_vocab[int(i)] for i in prediction]
    predicted_date = ''.join(output)
```

```
    # Compare the predicted date to the actual date
    actual_date = TARGETS[i]
    if predicted_date == actual_date:
        correct_predictions += 1
    else:
        print("source:", example)
        print("output:", predicted_date)
        print("actual:", actual_date, "\n")

# Calculate the accuracy
accuracy = correct_predictions / len(EXAMPLES)
print("Accuracy:", accuracy)

##### PLOT #####

def int_to_string(ints, inv_vocab):
    """
    Output a machine readable list of characters based on a list of indexes in the machine's vocabulary

    Arguments:
     ints -- list of integers representing indexes in the machine's vocabulary
      inv_vocab -- dictionary mapping machine readable indexes to machine readable characters

    Returns:
     l -- list of characters corresponding to the indexes of ints thanks to the inv_vocab mapping
    """

    l = [inv_vocab[i] for i in ints]
    return l

def plot_attention_map(modelx, input_vocabulary, inv_output_vocabulary, text, n_s = 128, num = 7):
    """
    Plot the attention map.

    """
    attention_map = np.zeros((10, 30))
    layer = modelx.get_layer('attention_weights')

    Ty, Tx = attention_map.shape

    human_vocab_size = 37
```

```
    # Well, this is cumbersome but this version of tensorflow-keras
has a bug that affects the
    # reuse of layers in a model with the functional API.
    # So, I have to recreate the model based on the functional
    # components and connect then one by one.
    # ideally it can be done simply like this:
    # layer = modelx.layers[num]
      # f = Model(modelx.inputs, [layer.get_output_at(t) for t in
range(Ty)])
    #

    X = modelx.inputs[0]
    s0 = modelx.inputs[1]
    c0 = modelx.inputs[2]
    s = s0
    c = s0

    a = modelx.layers[2](X)
    outputs = []

    for t in range(Ty):
        s_prev = s
        s_prev = modelx.layers[3](s_prev)
        concat = modelx.layers[4]([a, s_prev])
        e = modelx.layers[5](concat)
        energies = modelx.layers[6](e)
        alphas = modelx.layers[7](energies)
        context = modelx.layers[8]([alphas, a])
        # Don't forget to pass: initial_state = [hidden state, cell
state] (≈ 1 line)
          s, _, c = modelx.layers[10](context, initial_state = [s,
c])
        outputs.append(energies)

    f = Model(inputs=[X, s0, c0], outputs = outputs)


    s0 = np.zeros((1, n_s))
    c0 = np.zeros((1, n_s))
                encoded    =    np.array(string_to_int(text,    Tx,
input_vocabulary)).reshape((1, 30))
        encoded  =  np.array(list(map(lambda  x:  to_categorical(x,
num_classes=len(input_vocabulary)), encoded)))


    r = f([encoded, s0, c0])

    for t in range(Ty):
```

```
        for t_prime in range(Tx):
            attention_map[t][t_prime] = r[t][0, t_prime]

    # Normalize attention map
    row_max = attention_map.max(axis=1)
    attention_map = attention_map / row_max[:, None]

    prediction = modelx.predict([encoded, s0, c0])

    predicted_text = []
    for i in range(len(prediction)):
                predicted_text.append(int(np.argmax(prediction[i],
axis=1)))

    predicted_text = list(predicted_text)
              predicted_text    =    int_to_string(predicted_text,
inv_output_vocabulary)
    text_ = list(text)

    # get the lengths of the string
    input_length = len(text)
    output_length = Ty

    # Plot the attention_map
    plt.clf()
    f = plt.figure(figsize=(8, 8.5))
    ax = f.add_subplot(1, 1, 1)

    # add image
         i   =   ax.imshow(attention_map,   interpolation='nearest',
cmap='Blues')

    # add colorbar
    cbaxes = f.add_axes([0.2, 0, 0.6, 0.03])
    cbar = f.colorbar(i, cax=cbaxes, orientation='horizontal')
      cbar.ax.set_xlabel('Alpha value (Probability output of the
"softmax")', labelpad=2)

    # add labels
    ax.set_yticks(range(output_length))
    ax.set_yticklabels(predicted_text[:output_length])

    ax.set_xticks(range(input_length))
    ax.set_xticklabels(text_[:input_length], rotation=45)

    ax.set_xlabel('Input Sequence')
    ax.set_ylabel('Output Sequence')
```

```
    # add grid and legend
    ax.grid()

    f.savefig("attention_map.png")

    return attention_map

text = "tuesday 09 october 1993"
# Call the function with the correct parameters
plot_attention_map(model, human_vocab, inv_machine_vocab, text, n_s
= 64, num = 7)
```

ANEXO II

Tabela 2 - Valores de Acurácia e alguns tipos de erros encontrados para diferentes épocas.

Épocas	Acurácia	Exemplo de erro
5	0.109	source: tuesday march 3 2015 output: 2013-03-33 actual: 2015-03-03
6	0.692	source: 1/31/70 output: 1970-12-11 actual: 1970-01-31
7	0.420	source: monday december 3 2001 output: 2010-12-33 actual: 2001-12-03
8	0.593	source: 1/19/01 output: 2011-09-11 actual: 2001-01-19
9	0.722	source: oct 21 1990 output: 1990-10-22 actual: 1990-10-21
10	0.844	source: 13 oct 2022 output: 2022-11-13 actual: 2022-10-13
11	0.996	source: 2/17/75 output: 1975-02-27 actual: 1975-02-17
12	0.855	source: 25 03 19 output: 2019-05-25 actual: 2019-03-25
13	0.908	source: 1/31/70 output: 1970-01-33 actual: 1970-01-31
14	0.955	source: 1/17/91 output: 1991-01-18 actual: 1991-01-17
15	0.934	source: february 22 1988 output: 1988-12-22 actual: 1988-02-22
16	0.939	source: 18 06 22 output: 2022-04-17 actual: 2022-06-18
17	0.955	source: 20 oct 2009 output: 2019-10-20

		actual: 2009-10-20
18	0.963	source: 7/10/98 output: 1998-08-10 actual: 1998-07-10
19	0.978	source: 1/19/01 output: 2021-01-19 actual: 2001-01-19
20	0.999	source: december 31 2008 output: 2008-12-31 actual: 2007-12-31
21	0.967	source: 2/17/75 output: 1975-03-17 actual: 1975-02-17
22	0.979	source: monday april 29 1985 output: 1985-05-29 actual: 1985-04-29
23	0.978	source: 1/31/70 output: 1970-11-31 actual: 1970-01-31
24	0.982	source: 1/17/91 output: 1991-01-19 actual: 1991-01-17
25	0.986	source: 08.07.88 output: 1988-08-08 actual: 1988-07-08