UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET

KATEDRA ZA RAČUNARSTVO I INFORMATIKU

Lossless enkoder/dekoder podataka korišćenjem rekurentnih neuronskih mreža (RNN) i aritmetičkog kodiranja

Student: Arsenije Arsenijević

Predmet: Soft kompjuting

NIŠ, 2020

1. Uvod

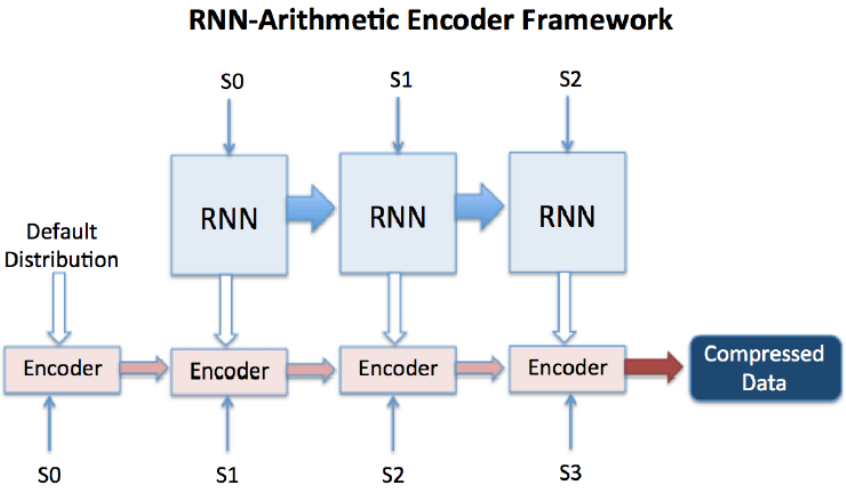
Lossless enkoder/dekoder podataka je onaj sistem koji nakon dekodiranja enkodiranih podataka vraća podatke u izvornom obliku, bez gubitka informacija. Upotreba može biti mnogostruka, od enkodiranja podataka radi čuvanja privatnosti, enkodiranja podataka pre slanja kroz mrežu (bilo u svrsi zaštite podataka ili smanjivanja obima podataka – kompresovanja – radi manjeg opterećenja mreže), itd.

Jedno od predloženih rešenja koje koristi neuronske mreže je opisano u radu [1]. U pomenutom radu se koristi rekurentna neuronska mreža (eng. *Recurent Neural Network* – u daljem tekstu RNN) za predviđanje *uslovne distibucije verovatnoće* (eng. *Conditional Probability Distribution* – u daljem tekstu *izlaz iz mreže*) trenutnog karaktera na osnovu *n* prethodnih karaktera i taj izlaz iz mreže se koristi u bloku aritmetičkog kodera za kodiranje trenutnog karaktera.

Ovaj rad predstavlja izveštaj o pokušaju implementacije prethodno opisanog sistema u okviru projekta iz predmeta *Soft kompjuting* (eng. *Soft Computing*) na master akademskim studijama Elektronskog fakulteta u Nišu. Rad se fokusira samo na rad dela sa neuronskim mrežama, svi ostali delovi će samo biti kratko pomenuti i dati bez objašnjenja.

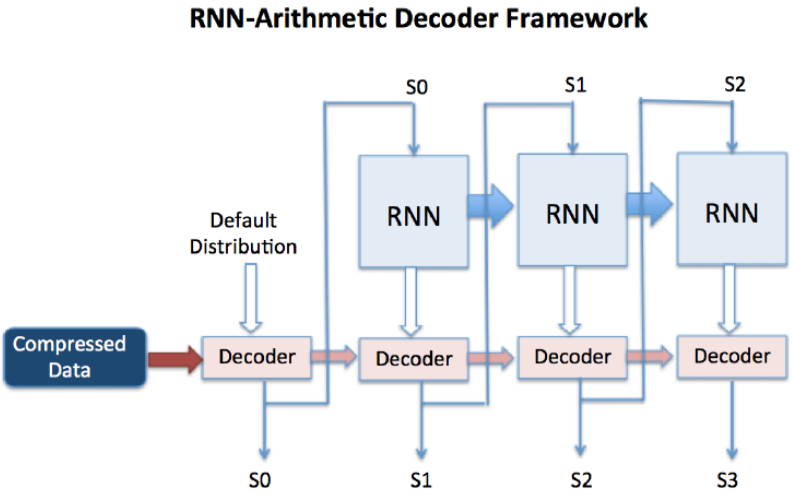
2. Ideja projekta

Osnovna ideja je implementirati sledeći sistem:



Ilustracija 1

Kao što se vidi na *Ilustraciji 1*, sistem se sastoji od dva osnovna dela: RNN blok i blok aritmetičkog enkodera. Kako za prvi karakter mreža ne može dati nikakvu pretpostavku jer ne postoje prethodni karakteri, koristi se podrazumevana distribucija (eng. *Default Distribution*) za prvi karakter koja ne dolazi iz neuronske mreže. Ova vrednost će kasnije biti važna kako bismo dekodirali taj prvi karakter.



Ilustracija 2

Ilustracija 2 pokazuje izgled sistema u slučaju dekodiranja prethodno enkodiranih podataka. Ponovo se vidi podrazumevana distribucija za dekodiranje prvog karaktera. Napomenućemo da postoje 4 „toka podataka'': 1) tok distribucije – izlaz iz RNN i ulaz u enkoder/dekoder; 2) tok karaktera – ulaz karaktera direktno u blokove RNN i enkoder/dekoder; 3) interni tok metapodataka između ćelija/iteracija u RNN i 4) interni tok metapodataka između iteracija enkodera/dekodera.

3. Implementacija

Za implementaciju su korišćeni *Python 3.7* i njegove interne biblioteke, zatim biblioteka *numpy* (https://numpy.org/) za olakšani rad sa podacima i na kraju *tensorflow API* (https://www.tensorflow.org/) i *keras* (https://keras.io/) za kreiranje i modelovanje neuronske mreže. Korišćeno okruženje je *PyCharm 2020.2*.

Sledi kompletna funkcionalna implementacija (bez komentara i bespotrebnih linija koda) sa objašnjenjima koda. Cela implementacija je organizovana u jednom *Python* fajlu - *main.py*, osim dodatna 3 Python fajla koji služe kao paketi za *black-box* rad sa aritmetičkim kodiranjem. Za aritmetičko kodiranje je korišćena *open-source* pokazna implementacija (https://github.com/nayuki/Reference-arithmetic-coding).

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = ""
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
from arithmeticcompress import ArithmeticCompress
from arithmeticdecompress import ArithmeticDecompress

SEED = 7
VERBOSE = 0
DEFAULT_PROBABILITY_DISTRIBUTION_ESTIMATE_FIRST = 10.0
DEFAULT_PROBABILITY_DISTRIBUTION_ESTIMATE_SECOND = 15.0

np.random.seed(SEED)
tf.random.set_seed(SEED)

tf.config.set_visible_devices([], 'GPU')
```

Prethodni kod pokazuje importovanje biblioteka koje su korišćene, zatim gašenje podrške za korišćenje grafičke kartice i prosleđivanje konstantnog *SEED*-a bibliotekama *numpy* i *tensorflow*. Poslednja dva opisana koraka su tu radi eliminisanja nedeterminističnosti sistema što je neophodno iz razloga što je potrebno da mreža bude identično početno konfigurisana u početnom trenutku pre enkriptovanja, odnosno dekriptovanja, kako bi vraćala identične rezultate pri identičnim uzastopnim ulazima pri enkriptovanju, odnosno dekriptovanju.

```
class LstmEncoder:
    data = []
    num_epochs = 1
    batch_size = 1
    time_steps = 1
    num_features = 1
    dictionary_size = 256
```

Kompletna implementacija je odrađena u okviru jedne klase *LstmEncoder* (vrši funkcije i enkodera i dekodera). Nakon definisanja zaglavlja klase definisane su i neke konstantne vrednosti i prazna lista za ulazne podatke (*data*).

Broj epoha je 1 zato što se vrši samo jedan prolaz kroz podatke. Teorijski je moguće vršiti i više prolaza, ali bi onda bilo neophodno čuvati poslednje stanje mreže (sve skrivene parametre koje mreža sadrži, kao što su težine neurona) zajedno sa enkriptovanim podacima. U ovoj implementaciji, vrši se samo jedan prolaz kroz podatke.

Veličina *batch*-a je takođe 1 iz razloga što dajemo jedan po jedan karakter na ulaz mreže.

Broj *vremenskih koraka* (eng. *Time Steps*) zavisi od veličine *batch*-a i ukupnog broja podataka i broja *feature*-a i u opštem slučaju bi ovde trebao biti ukupan broj podataka (karaktera, bajtova) – jer koristimo 1 i 1 za oba navedena. Ovde je pokazno inicijalizovan na 1 jer nećemo koristiti funkcije iz *tensorflow API*-a koje automatski vrše iteriranje *batch*-eva jer je nama potreban izlaz iz svakog *batch*-a, ne samo iz epohe – umesto toga ručno ćemo iterirati kroz *batch*-eve.

Broj *feature*-a je inicijalizovan na 1 jer koristimo prave vrednosti karaktera pri kreiranju, odnosno ne vršimo *one-hot-encoding*. Sitnim izmenama u kodu bismo mogli ubaciti *one-hot-encoding* čime bi broj feature-a prešao sa 1 na 256 – jer to predstavlja mogućih broj karaktera koji se mogu naći u ulaznom fajlu, kako radimo na nivou bajtova. S tim na umu, veličina rečnika je postavljena na 256. *Feature* nam je ovde u suštini *integer* vrednost bajta.

```
    def __init__(self, from_file, to_file):
        self.from_file = from_file
        self.to_file = to_file
```

Prethodni kod predstavlja konstruktor klase koji prima kao parametre i čuva putanje do ulaznog fajla (koji će se enkriptovati/dekriptovati) i izlaznog fajla (koji će nastati pri enkripciji/dekripciji).

```
    def load_data(self):
        with open(self.from_file, 'rb') as file:
            byte = file.read(1)
            while byte:
                self.data.append(byte[0])
                byte = file.read(1)
        self.total_series_len = len(self.data)
```

Prethodni kod predstavlja funkciju *load_data* koja služi da učita podatke iz fajla koji će se enkriptovati (ne koristi se kod dekripcije, kako Aritmetički dekoder čita ulazni fajl bit po bit – isto kao što ga i upisuje Aritmetički enkoder). Kako je ovo samo eksperimentalan/pokazni kod, učitavamo sve podatke odjednom – u opštem slučaju bismo za velike fajlove zbog očuvanja memorije pravili generator ulaza, koji bi vraćao deo po deo iz fajla. Ovde to nije potrebno. Kod se čini malo prenatrpan za jednostavno čitanje fajlova, ali takav je zbog konverzije bajta u njegov *integer* parnjak (opseg 0 - 255). Takođe čuvamo i broj karaktera koji smo učitali.

```
    def one_hot_encode(self, array_of_data, num_of_classes):
        return tf.one_hot(array_of_data, num_of_classes,
                          dtype=tf.dtypes.int32)
```

Prethodni kod pokazuje funkciju (koja je u klasi, ali realno može biti i statička/globalna) koja bi služila za *one-hot* enkodiranje podataka kada bismo koristili *one-hot-encoding*. Koristi *tensorflow API* funkciju sa identičnim parametrima i dodatnim parametrom koji naglašava da je tip u izlaznom *encoding*-u potrebno biti *integer*.

```
def prepare_data(self):
    data_array = list(self.data)
    #data_array = self.one_hot_encode(data_array,
                                      self.dictionary_size)
    #self.num_features = self.dictionary_size
    real_time_steps = self.total_series_len // self.batch_size
                      // self.num_features
    data_final = np.array(data_array)
                 .reshape((self.batch_size, real_time_steps,
                           self.num_features))
    return data_final
```

Prethodni kod pokazuje funkciju pripreme učitanih podataka za ulaz u neuronsku mrežu. Prvo se pravi kopija podataka (preventiva u slučaju da kasnije vršimo neku izmenu podataka). Zatim vidimo dve zakomentarisane linije koda (iza znaka #) koje bi samo bilo potrebno odkomentarisati kada bismo koristili *one-hot-encoding*. Zatim sledi određivanje pravih vremenskih koraka (jer je konstantna jedinica samo sa jedan batch) a na osnovu ukupnog broja podataka, veličine *batch*-a i broja *feature*-a. Zatim se podaci wrap-uju u numpy listu i vrši se reorganizacija podataka (*reshape*) kako bi bili u pogodnom obliku za davanje na ulaz u neuronsku mrežu (standardan oblik je trodimenziona struktura sa dimenzijama (veličina-*batch*-a, broj-vremenskih-koraka, broj-*feature*-a)).

```
def build_model(self, data_final=None):
    model = keras.Sequential(name='eNcoder-network')
    model.add(layers.LSTM(self.num_features,
                          name='eNcoder-lstm-1',
                          stateful=True,
                          return_sequences=True,
                          batch_input_shape=(self.batch_size,
                                             None,
                                             self.num_features)))
    model.add(layers.Dense(self.dictionary_size,
                           activation='softmax',
                           name='eNcoder-dense-1'))
    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy')
    return model
```

Prethodni kod predstavlja srž, odnosno izgled/arhitekturu neuronske mreže koju koristimo. Vidimo da se koristi *keras*-ov sekvencijalni model. Model sadrži dva skrivena sloja. Mreža i slojevi su dobili određena imena koja su apsolutno inkonsekventna za implementaciju.

Prvi sloj, *LSTM* definiše tip RNN mreže koju koristimo. U suštini ovde može biti i npr. *GRU* ili neki drugi tip ali trenutna implementacija koristi *LSTM*. *LSTM* predstavlja skraćenicu od *Long-Short-Term-Memory* i to je tip veštačkih neurona koji je bolje osmišljen od običnog *Vanilla RNN* neurona u smislu da se bori donekle sa problemom nestajućeg gradijenta – gde je gradijent dinamička veličina kojom se dinamički ispravljaju težine neurona u mreži, a vremenom taj gradijent može opasti na previše nisku vrednost i to se naziva *problem nestajućeg gradijenta* (eng. *Vanishing Gradient Problem*) koji tišti sve tipove RNN-a. Prvi parametar u sloju je *units* (koji je ovde podrazumevan pa se ne navodi kao ostali) i on predstavlja broj neurona u ovom sloju; ovde je taj broj simbolički podešen na broj *feature*-a jer je znamo da je broj *feature*-a 1 u trenutnoj implementaciji i želimo da koristimo samo jedan neuron. Broj neurona u suštini ne zavisi od broja *feature*-a, samo smo hteli takvu mrežu u ovom slučaju. Sledeći bitan parametar je *stateful* koji je postavljen na vrednost *True* što je izuzetno bitno kako bi mreža pamtila rezultate iz prethodnih iteracija (osim ako ne pozovemo *reset_states* ali to nećemo raditi). Ovaj parametar nam dozvoljava da mreža uči iz ulaza kao iz jedne velike vremenske serije. Sledeći parametar je *return_sequences* koji je ovde inkonsekventan jer nam je broj *unit*-a postavljen na 1 (*num_features*) – a u opštem slučaju je bitan da bi sloj na izlazu dao izlaz iz svakon neurona (kao sekvencu) umesto samo izlaz iz poslednjeg neurona. Ovo je uvek bitno ako LSTM sloj nije poslednji sloj u mreži kako bi mreža stvarno bila sekvencijalna i da svi slojevi dobiju sve informacije. Sledeći parametar je *batch_input_shape* koji je potrebno ispuniti zbog *stateful=True* parametra jer ako je *stateful* mreža, neophodno je da LSTM sloj zna veličinu *batch*-a. Kao *batch_input_shape* jer dat isti oblik kao što smo reorganizovali ulazne podatke, s tim što je broj *batch*-eva stavljen kao *None* da bismo dozvolili dinamički broj *batch*-eva (u našem slučaju je i to uvek 1).

Drugi sloj predstavlja *Dense* sloj koji je inicijalizovan sa brojem *unit*-a na broj karaktera u rečniku, kao aktivaciona funkcija je postavljena *softmax* – koja skuplja izlaz iz prethodnog sloja na opseg [0,1].

Zatim se model kompajlira sa postavljenim optimajzerom na *adam* što predstavlja tip/način ažuriranja težina u mreži. Kao *loss* funkcija je odabrana *sparse_categorical_crossentropy* (u slučaju korišćenja *one-hot-encoding* bila bi ista funkcija bez prefiksa *sparse_*) koja je pogodna za korišćenje u multiklasnoj klasifikaciji – što naša mreža i predstavlja, model multiklasne klasifikacije sa 256 mogućih klasa.

Izgled mreže:

```
Model: "eNcoder-network"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
eNcoder-lstm-1 (LSTM)        (1, None, 1)              12
_________________________________________________________________
eNcoder-dense-1 (Dense)      (1, None, 256)            512
================================================================
Total params:524   Trainable params:524   Non-trainable params:0
_________________________________________________________________
```

```
def compress(self, model, data_final):
    enc = ArithmeticCompress(self.to_file)
    enc.start()

    enc.compress_next(DEFAULT_PROB_DISTR_EST_1, self.data[0])
    enc.compress_next(DEFAULT_PROB_DISTR_EST_2, self.data[1])

    for i in range(until-2):
        inp = data_final[:,i,:].reshape((self.batch_size,
                    self.time_steps, self.num_features))
        tar = data_final[:,i+1,:].reshape((self.batch_size,
                    self.time_steps, self.num_features))

        output = model.train_on_batch(x=inp, y=tar,
                                      reset_metrics=False)

        enc.compress_next(new_freq + 1, self.data[i+2])

    enc.stop()
```

Došli smo do enkodiranja (ovde nazvanog kompresovanje). Prethodni kod predstavlja funkciju koja enkodira sve podatke iz ulaznog fajla. Prva linija je inicijalizacija *black-box* aritmetičkog enkodera (*ArithmeticCompress*) kojem se kao parametar pri inicijalizaciji ovde daje u koji fajl da upisuje enkodirane podatke. Zatim se pokreću unutrašnji mehanizmi aritmetičkog enkodera. Sledeće dve linije predstavljaju ono početno enkodiranje prvog karaktera (u našem slučaju početno enkodiramo 2 karaktera sa unapred poznatim distribucijama – konstante sa početka fajla). Zatim pokrećemo iteriranje kroz podatke gde uzimamo trenutni i sledeći podatak, reorganizujemo u odgovajući oblik, treniramo mrežu nad samo te dve informacije bez resetovanja prethodnih vrednosti (parametri sve govore) i zatim koristimo izlaznu vrednost *loss* funkcije kao distribuciju da enkodiramo treći karakter (primetimo da enkodiramo treći na osnovu predikcije drugog iz prvog karaktera (i istorije) – ovo je mana koju je autor svesno ubacio u kod u trenutnom nedostatku valjanijeg rešenja).

Nakon iteriranja kroz sve podatke zaustavljamo unutrašnje mehanizme aritmetičkog enkodera i proces enkripcije je završen.

```
def decompress(self, model):
    dec = ArithmeticDecompress(self.from_file, self.to_file)
    dec.start()

    symbol1 = dec.decompress_next(DEFAULT_PROB_DISTR_EST_1)
    symbol2 = dec.decompress_next(DEFAULT_PROB_DISTR_EST_2)

    while symbol2 != 256:
        inp = np.array([symbol1]).reshape((self.batch_size,
                                           self.time_steps,
                                           self.num_features))
        tar = np.array([symbol2]).reshape((self.batch_size,
                                           self.time_steps,
                                           self.num_features))

        output = model.train_on_batch(x=inp, y=tar,
                                      reset_metrics=False)

        symbol_temp = dec.decompress_next(output + 1)
        symbol1 = symbol2
        symbol2 = symbol_temp

    dec.stop()
```

Apsolutno analogno enkodiranju (kompresovanju), dekodiranje (dekompresovanje) iz prethodnog koda počinje inicijalizacijom i pokretanjem aritmetičkog dekodera (*ArithmeticDecompress*) – s tim što dekoderu dajemo i ulazni i izlazni fajl, odnosno dekoder čita iz ulaznog fajla bitove i upisuje u izlazni fajl bajtove, takva je implementacija. Na osnovu konstantnih distribucija dekriptujemo prva dva simbola/bajta. Na osnovu ta dva bajta (njihovoh reorganizujućeg oblika) se vrši treniranje mreže sa identičnim parametrima kao u enkodiranju. Treći simbol se dekodira u dekriptoru na osnovu izlaza *loss* funkcije iz mreže. Drugi simbol se postavlja kao prvi, treći kao drugi, i iteracije se ponavljaju sve do trenutka kada dekriptor ne vrati vrednost 256 (vrednost veću od maksimalne u našem rečniku – u ovom slučaju to će biti 256). Zaustavljaju se mehanizmi dekriptora.

Sledi primer korišćenja enkodera:

```
lstmEnc = LstmEncoder(from_file=f'data/arbitrary/pdf.pdf',
    to_file=f'data/compressed/compressed_pdf_loss1.bin')
lstmEnc.load_data()
data = lstmEnc.prepare_data()
model = lstmEnc.build_model(data)
lstmEnc.compress(model, data)
```

Sledi primer korišćenja dekodera:

```
lstmEnc = LstmEncoder(from_file=f'data/compressed/pdf_loss1.bin',
    to_file=f'data/decompressed/pdf1.pdf')
model = lstmEnc.build_model()
lstmEnc.decompress(model)
```

4. Zaključak

Ovaj sistem predstavlja izuzetno zanimljiv način kompresije podataka gde je neuronskoj mreži prepušteno traženje veza između podataka umesto korišćenja klasičnih statističkih metoda. Mada prikazana implementacija rešenja ne funkcioniše iz dva razloga, prvi što se vrši enkripcija trećeg podatka na osnovu predviđanja drugog i što odabrano *black-box* rešenje za aritmetički koder ne odrađuje dekripciju.

Sistem uspešno kriptuje podatke i u nekim test slučajevima je obim enkriptovanih podataka bio manji, a u nekim veći od početnih podataka. Znači da sistem apsolutno kriptuje podatke, samo ih možda kompresuje ili ne. Kompresija bi se verovatno povećala pažljivim podešavanjem parametara sistema jer je prikazan sistem ipak laički u odnosu na to kako bi stvarno trebao izgledati – a sve u svrsi učenja i ograničenja resursa.

Neophodno je napomenuti i da je sistem radio izuzetno sporo u toku enkripcije podataka. Obzirom da u tekućem modelu nema mnogo parametara (samo 524), očekivano je da bi uvođenjem komplikovanijih slojeva treniranje bilo još zahtevnije, i previđanje autora je da je ovo najveća mana ovakvog sistema (brzina, vreme izvršenja) čak i u funkcionalnom obliku.

Svakako bi funkcionalni oblik rešenja trebao biti u mogućnosti naći dublje veze u podacima od klasičnih statističkih metoda i prognoza autora je da će funkcionalno rešenje dati odlične rezultate.

Kod projekta i početna verzija ovog dokumenta biće dostupni i na:

https://github.com/ArxyaArsa/master-LstmCompress

Literatura

[1] Mohit Goyal et al., “DeepZip: Lossless Data Compression Using Recurrent Neural Networks,” ArXiv:1811.08162 [Cs, Eess, q-Bio], November 20, 2018, http://arxiv.org/abs/1811.08162.

[2] https://github.com/kedartatwawadi/NN_compression

[3] J. Schmidhuber and S. Heil, “Sequential Neural Text Compression,” IEEE Transactions on Neural Networks 7, no. 1 (January 1996): 142–46, https://doi.org/10.1109/72.478398.

[4] Mahoney, Matthew. (2000). Fast Text Compression with Neural Networks.

[5] http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[6] https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

[7] Alex Graves, “Generating Sequences With Recurrent Neural Networks,” ArXiv:1308.0850 [Cs], June 5, 2014, http://arxiv.org/abs/1308.0850.