

Stranice predmeta Duboko učenje (FER)

3. vježba: modeliranje nizova povratnim neuronskim mrežama

U trećoj laboratorijskoj vježbi bavimo se povratnim neuronskim mrežama i problemom modeliranja jezika na razini slova.

Kao što su konvolucijske neuronske mreže arhitektura specijalizirana za obrađivanje slika kroz iterativno povećavajuće lokalne filtre, povratne neuronske mreže su arhitektura specijalizirana za obrađivanje nizova podataka u smjeru jedne dimenzije (simuliranje protoka vremena). Intuitivan i motivirajući primjer ovoga je tekst, u kojemu možemo zamisliti trenutni smisao rečenice kao funkciju svih dosadašnjih riječi, te za slučaj jezika europskog podrijetla - možemo pretpostaviti da se trenutni smisao mijenja od početka rečenice prema kraju, u smjeru s lijeva na desno.

Glavna prednost povratnih neuronskih mreža ne nalazi se u obrađivanju postojećih nizova, već u generiranju novih slijedova na temelju naučene reprezentacije u skrivenom sloju ili na temelju naučenih parametara mreže. Slijedovi generirani (i obrađivani) pomoću povratnih neuronskih mreža mogu imati varijabilnu duljinu, za razliku od generiraja slika pomoću konvolucijskih mreža, u kojima dimenzije izlaza (generirane slike) moraju biti unaprijed poznate. U zadnje vrijeme adaptacije konvolucijskih slojeva i mreža za projekciju podataka varijabilne duljine u reprezentaciju skrivenog sloja uzimaju maha te konkuriraju povratnim modelima, no generiranje je još u domeni povratnih mreža.

Statističko modeliranje jezika je problem aproksimacije vjerojatnosne distribucije preko niza riječi. Za zadani niz riječi \mathbf{w} "The quick brown fox jumped" duljine $n = 5$, jezični model pridodaje vjerojatnost $P(\mathbf{w}) = P(w_1, \dots, w_n)$ tom nizu riječi, koju interpretiramo kao vjerojatnost da se taj niz pojavi u stvarnom jeziku.

Standardni statistički modeli jezika radi izračunljivosti i rijetkosti podataka ne mogu modelirati proizvoljno duge slijedove, te prihvaćaju aproksimaciju prozorom riječi (tzv. kontekst) konstantne duljine. Takvi jezični modeli se zovu modelima n-grama.

Primjerice, za $n = 5$ i jezični model 5-grama, vjerojatnost pojavljivanja prethodno navedenog niza bi bila:

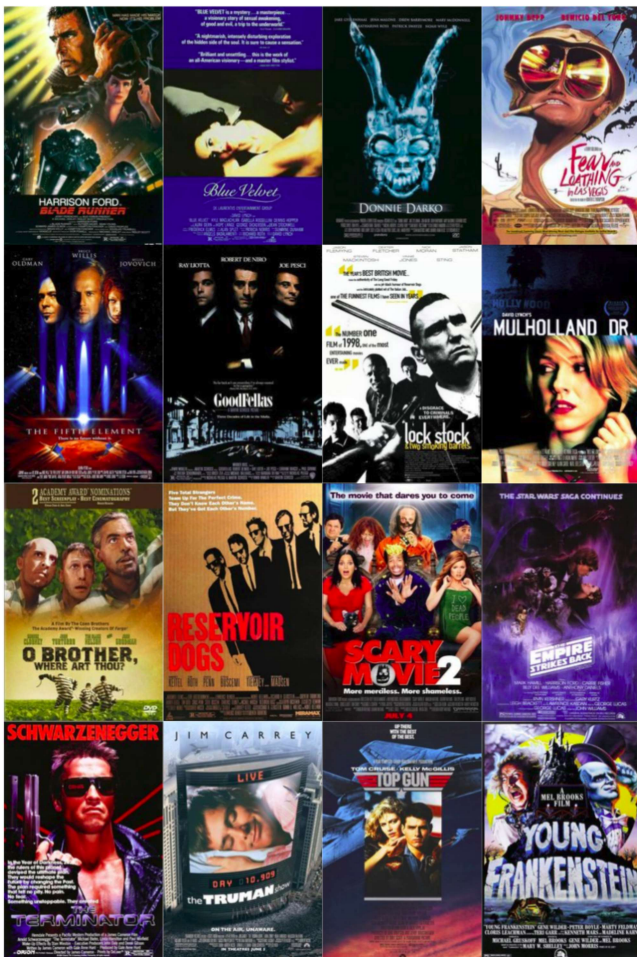
$$P(\mathbf{w}) = P(\text{The})P(\text{quick}|\text{The})P(\text{brown}|\text{quick}, \text{The}) \dots$$

Kao što možemo primjetiti, jezični modeli poput navedenog čuvaju kontekst samo unutar kratkog prozora od pet riječi teško pamte dulje veze unutar teksta. Usprkos tome, jezični modeli bazirani na n-gramima (5-gramima) su bili state-of-the-art do pred par godina.

Povratne neuronske mreže su problemima izračunljivosti i rijetkosti doskočile s dvije optimizacije - prva od njih je čuvanje pomičnog prosjeka (*engl. running average*) dosadašnjeg niza riječi u skrivenom sloju, dok je druga korištenje projekcije riječi u tzv. ugnježđeni (*engl. embedding*) sloj.

Skup podataka: Cornell Movie–Dialogs Corpus

Zadatak u laboratorijskoj vježbi je naučiti jezični model na domeni dijaloga iz filmova. Originalni skup podataka možete skinuti [ovdje](#), dok je verzija koju ćete koristiti u laboratorijskoj vježbi podskup ukupnog datasea. Primjeri filmova iz podskupa su u nastavku:



Primjer teksta dijaloga iz filmova u formatu iz skupa podataka:

ICE:

You're a hell of a flyer. You can be my wingman any time.

MAVERICK:

No. You can be mine

Svi razgovori u skupu podataka će pratiti navedeni format - ime osobe napisano tiskanim slovima praćeno s dvotočkom i simbolom novog reda, te tekst koji osoba izgovara praćen s dva simbola novog reda. Dijalozi različitih filmova i scena nisu odvojeni na poseban način.

Uz podskup skupa podataka dostupna je i skripta `select_preprocess.py` koja iz cijelog skupa podataka odabire podskup naslova naveden u kontrolnoj tekstualnoj datoteci, te odrađuje predprocesiranje i zapisivanje dijaloga iz podskupa naslova. Skriptu možete naći [ovdje](#).

Predprocesiranje koje se provodi nad podacima je odbacivanje svih znakova osim alfanumeričkih te interpunkcije. Znakovi koji ostaju nakon predprocesiranja su:

```
!',. `: ? 0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c  
d e f g h i j k l m n o p q r s t u v w x y z
```

te razmak i znak novog reda.

Zadatak 1: učitavanje podataka i batching

Prvi zadatak u laboratorijskoj vježbi je modul za učitavanje tekstualnog skupa podataka te njegova podjela u *minibatcheve*. Razred za učitavanje skupa podataka implementirajte i spremite u datoteku pod nazivom `dataset.py`. Imenovanje samog razreda i metoda ostavljamo vama na izbor.

Modul kao argumente prima lokaciju skupa podataka na disku (datoteka u `txt` formatu), veličinu minibatcha te duljinu niza podataka (broj koraka koji se povratna mreža odmata). Modul se može podijeliti u tri konceptualna dijela:

1.1: Predprocesiranje podataka

Kako bi mogli koristiti tekstualne podatke u povratnim neuronskim mrežama, nužno je obaviti njihovo mapiranje u prostor projeva (u ovom slučaju diskretni). To ćemo napraviti tako da svakom znaku dodijelimo jedan broj (identifikator - *id*). U praksi, id-jevi se tekstualnim podacima dodijeljuju po principu “češći je manji”, tj. znakovi (ili riječi) koje su češće imaju niži id.

Za predprocesiranje podataka potrebno je implementirati funkciju koja za zadani tekst računa mapiranja (slovo - id) po principu “češći je manji”. Osim toga, potrebno je implementirati funkciju koja zadani proizvoljni niz slova pretvara u niz id-jeva, te funkciju koja zadani proizvoljni niz id-jeva pretvara u niz slova. Kosturi ovih funkcija mogli bi izgledati kao u nastavku:

```

# ...
# Code is nested in class definition, indentation is not representative.
# "np" stands for numpy.

def preprocess(self, input_file):
    with open(input_file, "r") as f:
        data = f.read().decode("utf-8") # python 2

    # count and sort most frequent characters

    # self.sorted_chars contains just the characters ordered descending by f
    self.char2id = dict(zip(self.sorted_chars, range(len(self.sorted_chars)))
    # reverse the mapping
    self.id2char = {k:v for v,k in self.char2id.items()}
    # convert the data to ids
    self.x = np.array(list(map(self.char2id.get, data)))

def encode(self, sequence):
    # returns the sequence encoded as integers
    pass

def decode(self, encoded_sequence):
    # returns the sequence decoded as letters
    pass

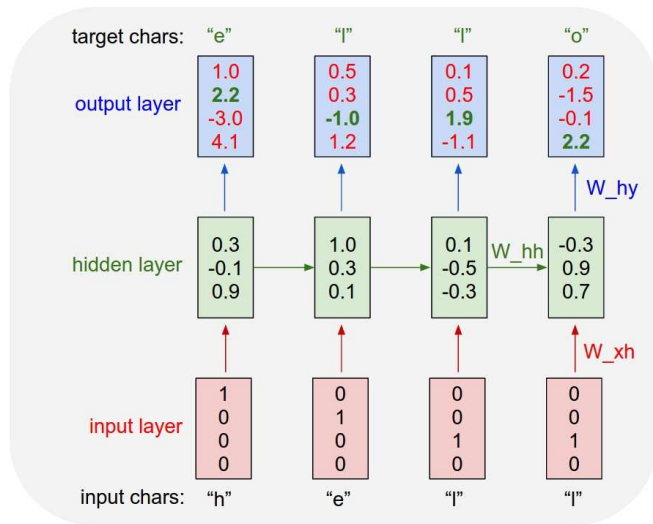
# ...

```

1.2: Podjela u minibatcheve

Pri podjeli u minibatcheve naš skup podataka razdvajamo u parove `x, y` veličine `batch_size`. No, u ovom slučaju možda nije jasno što nam je `y`. U zadatku modeliranja jezika na temelju trenutnog ulaznog znaka i prethodno viđenih znakova pokušavamo predvidjeti *idućí* znak. Prema tome, za ulaz x_t naš izlaz y_t koji pokušavamo predvidjeti je upravo x_{t+1} .

Primjer ovoga vidljiv je na idućoj slici (preuzeto s [“The Unreasonable Effectiveness of Recurrent Neural Networks”](#)):



Prema tome, vaš zadatak u ovom koraku je implementirati funkciju koja će na temelju postojećeg dataseta pretvorenog u niz id-jeva taj dataset pretvoriti u niz batcheva veličine `batch_size` te duljine `sequence_length`. Kostur ove funkcije bi mogao izgledati kao u nastavku:

```
# ...
# Code is nested in class definition, indentation is not representative.

def create_minibatches(self):
    self.num_batches = int(len(self.x) / (self.batch_size * self.sequence_length))

    # Is all the data going to be present in the batches? Why?
    # What happens if we select a batch size and sequence length larger than the total data?

    #####
    # Convert data to batches
    #####

    pass
```

Način spremanja i reprezentacije batcheva je prepušten vašem izboru.

1.3: Iteracija po minibatchevima

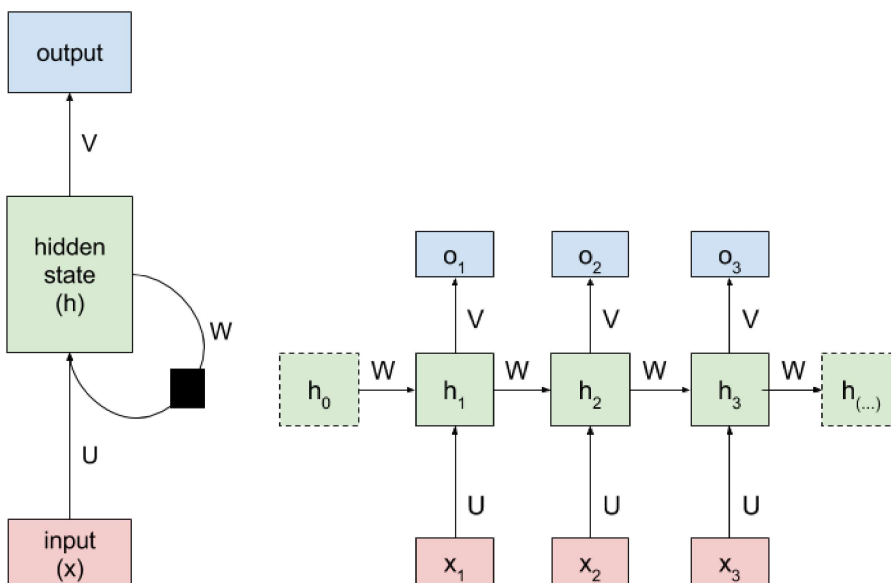
Razred za učitavanje skupa podataka treba sadržavati i logiku za iteraciju po batchevima skupa podataka. Pri ovome treba obraćati pozornost na činjenicu da se batchevi ne smiju miješati (*engl. shuffle*) unutar ni između batcheva, budući da radimo s tekstualnim podacima, te je njihov redoslijed bitan. Prema tome, redoslijed batcheva u svakoj epohi biti će identičan. Kostur ove funkcije bi mogao izgledati kao u nastavku:

```
# ...
# Code is nested in class definition, indentation is not representative.
def next_minibatch(self):
    # ...

    batch_x, batch_y = None, None
    # handling batch pointer & reset
    # new_epoch is a boolean indicating if the batch pointer was reset
    # in this function call
    return new_epoch, batch_x, batch_y
```

Zadatak 2: obična jednoslojna povratna neuronska mreža

Kao što smo spomenuli u uvodu laboratorijske vježbe, povratne neuronske mreže su specijalizirane za procesiranje usmjerenih nizova - no još nismo dotaknuli na koji način funkcioniraju. Rad neuronskih mreža najlakše je predočiti s iduća dva grafa:



U prvom grafu povratna neuronska mreža je zapisana kompaktno uz povratnu vezu s vremenskim odmakom (crni kvadrat simbolizira korištenje vrijednosti iz prošlog vremenskog koraka), dok je u drugom grafu prikazana tzv. razmotana (*engl. unrolled*) povratna mreža. Bitna stvar za primjetiti je da su težinske matrice (U, W, V) **neovisne** o vremenskom koraku!

Ovo je primjer ideje dijeljenja parametara prisutne i u konvolucijskim neuronskim mrežama - zahvaljujući kojoj je moguće obrađivati nizove varijabilnih i velikih dužina.

Matematički zapis izraza koji se izvode u grafovima je u nastavku:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b})$$

$$\mathbf{o}^{(t)} = \mathbf{V}\mathbf{h}^{(t)} + \mathbf{c}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

Inicijalizacija početnog skrivenog stanja $\mathbf{h}^{(0)}$ se u jednostavnim primjenama ostavlja nasumičnosti ili inicijalizira na nule - no u praksi je bolje tretirati početno stanje kao parametar algoritma koji učimo. Unutar okvira laboratorijske vježbe dovoljno je inicijalizirati ga na vektor nula.

Parametri ove mreže su, prema tome matrice \mathbf{U} (*ulaz*), \mathbf{V} (*van*) i \mathbf{W} (*skriVeni-skriVeni*) te vektori pristranosti \mathbf{b} i \mathbf{c} .

Pri modeliranju jezika kao izlaz mreže u svakom vremenskom koraku ćemo imati klasifikacijski problem (koje je iduće slovo?), te ćemo koristiti gubitak unakrsne entropije s kojim ste se upoznali u prethodnoj laboratorijskoj vježbi.

Za podsjetnik, u nastavku su jednačba unakrsne entropije te njenog gradijenta, prilagođene notacijom za jednačbe povratnih neuronskih mreža.

$$L = - \sum_{i=1}^C \mathbf{y}_i \log(\hat{\mathbf{y}}_i)$$

$$\frac{\partial L}{\partial \mathbf{o}} = \hat{\mathbf{y}} - \mathbf{y}$$

Pri čemu je $\hat{\mathbf{y}}$ vektor vjerojatnosti dobiven kao izlaz mreže, dok je \mathbf{y} stvarna distribucija razreda (naješće, one-hot vektor s jedinicom na lokaciji točnog razreda).

2.1: Implementacija povratne neuronske mreže

Obična povratna neuronska mreža ima tri različita hiperparametra - veličinu skrivenog sloja, duljinu vremenskog odmatanja te stopu učenja, te prethodno navedene matrice i vektore parametara. Algoritam učenja koji trebate implementirati je Adagrad.

Uz pretpostavku standardne precizosti decimalnih brojeva (single precision, 4 bytes), veličine skrivenog sloja 500, zadatka modeliranja jezika s vokabularom veličine 70 (dimenzija ulaza te izlaza) - koliko je ukupno memorijsko zauzeće bez overheada programskog jezika? Koji parametar zauzima najviše prostora? Ponovite istu računicu za vokabular veličine 10000.

Navedene veličine vokabulara su slične onima koje ćete imati za modele na razini znakova (70 - u praksi bliže 150), te za modele na razini riječi (red veličine 10^5).

Predloženi početni hiperparametri vaše implementacije su veličina skrivenog sloja 100, duljina vremenskog odmatanja 30 te stopa učenja $1e-1$.

Implementaciju povratne neuronske mreže možemo podijeliti u 4 konceptualna dijela kao u nastavku:

2.1.1: Inicijalizacija parametara

Hiperparametre inicijalizirajte po preporučenim vrijednostima radi reproducibilnosti, dok matrice parametara inicijalizirajte nasumično iz gaussove distribucije s standardnom devijacijom $1e-2$. Vektore pristranosti inicijalizirajte na nule.

Primjer inicijalizacije parametara bi mogao izgledati kao u nastavku:

```
# ...
# Code is nested in class definition, indentation is not representative.
# "np" stands for numpy
def __init__(self, hidden_size, sequence_length, vocab_size, learning_rate):
    self.hidden_size = hidden_size
    self.sequence_length = sequence_length
    self.vocab_size = vocab_size
    self.learning_rate = learning_rate

    self.U = None # ... input projection
    self.W = None # ... hidden-to-hidden projection
    self.b = None # ... input bias

    self.V = None # ... output projection
    self.c = None # ... output bias

    # memory of past gradients - rolling sum of squares for Adagrad
    self.memory_U, self.memory_W, self.memory_V = np.zeros_like(self.U), np.
    self.memory_b, self.memory_c = np.zeros_like(self.b), np.zeros_like(self
```

Pri inicijalizaciji vektora pristranosti podsjetite se dodati redundantnu dimenziju kako bi numpy prepoznao dimenziju broadcastinga. Zbrajanje dimenzija ($H \times D + H$) neće nužno producirati željene rezultate, te je lakše inicijalizirati vektore pristranosti na $H \times 1$.

2.1.2: Prolaz unaprijed

Prolaz unaprijed povratne neuronske mreže možete zamisliti kao petlju u kojoj iterativno obrađujemo vremenske korake. Prolaz unaprijed prema tome možemo konceptualno

razdvojiti u funkciju koja obrađuje jedan jedini korak unaprijed kroz vrijeme, te funkciju koja obrađuje cijeli vremenski slijed te iterativno zove prethodnu.

Kosturi tih funkcija bi mogli izgledati kao u nastavku (kao u većini paketa za duboko učenje):

```
# ...
# Code is nested in class definition, indentation is not representative.

def rnn_step_forward(self, x, h_prev, U, W, b):
    # A single time step forward of a recurrent neural network with a
    # hyperbolic tangent nonlinearity.

    # x - input data (minibatch size x input dimension)
    # h_prev - previous hidden state (minibatch size x hidden size)
    # U - input projection matrix (input dimension x hidden size)
    # W - hidden to hidden projection matrix (hidden size x hidden size)
    # b - bias of shape (hidden size x 1)

    h_current, cache = None, None

    # return the new hidden state and a tuple of values needed for the backw

    return h_current, cache

def rnn_forward(self, x, h0, U, W, b):
    # Full unroll forward of the recurrent neural network with a
    # hyperbolic tangent nonlinearity

    # x - input data for the whole time-series (minibatch size x sequence_le
    # h0 - initial hidden state (minibatch size x hidden size)
    # U - input projection matrix (input dimension x hidden size)
    # W - hidden to hidden projection matrix (hidden size x hidden size)
    # b - bias of shape (hidden size x 1)

    h, cache = None, None

    # return the hidden states for the whole time series (T+1) and a tuple o

    return h, cache
```

Primjetite da iako parametri U, W i b postoje u razredu (self.U, self.W, self.b), zbog reproducibilnosti te numeričke provjere operacija preporuča se, no nije nužno, držati ih kao parametre funkcija. U slučaju da se koriste vrijednosti iz razreda (ili drukčija implementacija

povratne neuronske mreže), provjera implementacije je na studentu. U suprotnom, ispravno je primjetiti da ove funkcije mogu biti statičke (`@staticmethod` dekorator).

2.1.3: Prolaz unatrag

Prolaz unatrag neuronske mreže je konceptualno sličan prolazu unaprijed, uz izmjenu da se iteracija vrši od zadnjeg vremenskog koraka prema prvom. Prolaz unatrag se vrši algoritmom propagacije unatrag kroz vrijeme (*engl. backpropagation through time, BPTT*). Kosturi funkcija za propagaciju unatrag mogu izgledati kao u nastavku:

```
# ...
# Code is nested in class definition, indentation is not representative.

def rnn_step_backward(self, grad_next, cache):
    # A single time step backward of a recurrent neural network with a
    # hyperbolic tangent nonlinearity.

    # grad_next - upstream gradient of the loss with respect to the next hid
    # cache - cached information from the forward pass

    dh_prev, dU, dW, db = None, None, None, None

    # compute and return gradients with respect to each parameter
    # HINT: you can use the chain rule to compute the derivative of the
    # hyperbolic tangent function and use it to compute the gradient
    # with respect to the remaining parameters

    return dh_prev, dU, dW, db

def rnn_backward(self, dh, cache):
    # Full unroll forward of the recurrent neural network with a
    # hyperbolic tangent nonlinearity

    dU, dW, db = None, None, None

    # compute and return gradients with respect to each parameter
    # for the whole time series.
    # Why are we not computing the gradient with respect to inputs (x)?

    return dU, dW, db
```

Kako biste spriječili problem eksplodirajućih gradijenata, prije nego što primijenite ukupni gradijent kroz sve korake na parametre, koristite podrezivanje vrijednosti gradijenata po komponentama (gradient clipping). Preporuča se koristiti metoda `np.clip()`, te vrijednosti ostavljati u intervalu od -5 do 5.

2.1.4: Petlja učenja

Unutar petlje učenja potrebno je povezati povratnu neuronsku mrežu s ulaznim podacima, izračunati gubitak na izlazu mreže te vršiti postupak optimizacije parametara. Idejno, neuronska mreža bi trebala biti neovisna o podacima - tako da je sasvim u redu odvojiti kontrolu iteriranja po epohama i minibatchevima izvan razreda mreže, dok će mreža obrađivati pojedine korake optimizacije.

Implementacija kostura funkcije za izračun gubitka bi mogla izgledati ovako:

```
# ...
# Code is nested in class definition, indentation is not representative.

def output(h, V, c):
    # Calculate the output probabilities of the network

def output_loss_and_grads(self, h, V, c, y):
    # Calculate the loss of the network for each of the outputs

    # h - hidden states of the network for each timestep.
    #     the dimensionality of h is (batch size x sequence length x hidden
    # V - the output projection matrix of dimension hidden size x vocabulary
    # c - the output bias of dimension vocabulary size x 1
    # y - the true class distribution - a tensor of dimension
    #     batch_size x sequence_length x vocabulary size - you need to do th
    #     passing the argument. A fast way to create a one-hot vector from
    #     an id could be something like the following code:

    # y[batch_id][timestep] = np.zeros((vocabulary_size, 1))
    # y[batch_id][timestep][batch_y[timestep]] = 1

    #     where y might be a list or a dictionary.

    loss, dh, dV, dc = None, None, None, None
    # calculate the output (o) - unnormalized log probabilities of classes
    # calculate yhat - softmax of the output
    # calculate the cross-entropy loss
    # calculate the derivative of the cross-entropy softmax loss with respect
    # calculate the gradients with respect to the output parameters V and c
    # calculate the gradients with respect to the hidden layer h
```

```
return loss, dh, dV, dc
```

Implementacija kostura funkcije za ažuriranje parametara modela (težina) bi mogla izgledati ovako:

```
# ...
# Code is nested in class definition, indentation is not representative.

# The inputs to the function are just indicative since the variables are mos

def update(self, dU, dW, db, dV, dc,
            U, W, b, V, c,
            memory_U, memory_W, memory_b, memory_V, memory_c):

    # update memory matrices
    # perform the Adagrad update of parameters
    pass
```

Implementacija kostura petlje za kontrolu toka podataka te iteriranja optimizacijskog procesa bi mogla izgledati kao u nastavku:

```
# ...
# code not necessarily nested in class definition

def run_language_model(dataset, max_epochs, hidden_size=100, sequence_length

    vocab_size = len(dataset.sorted_chars)
    RNN = None # initialize the recurrent network

    current_epoch = 0
    batch = 0

    h0 = np.zeros((hidden_size, 1))

    average_loss = 0

    while current_epoch < max_epochs:
        e, x, y = dataset.next_minibatch()

        if e:
            current_epoch += 1
```

```

h0 = np.zeros((hidden_size, 1))
# why do we reset the hidden state here?

# One-hot transform the x and y batches
x_oh, y_oh = None, None

# Run the recurrent network on the current batch
# Since we are using windows of a short length of characters,
# the step function should return the hidden state at the end
# of the unroll. You should then use that hidden state as the
# input for the next minibatch. In this way, we artificially
# preserve context between batches.
loss, h0 = RNN.step(h0, x_oh, y_oh)

if batch % sample_every == 0:
    # run sampling (2.2)
    pass
batch += 1

```

Funkcija `step` kojoj kostur nije opisan sadržava logiku pokretanja jednog prolaza unaprijed i unatrag povratne neuronske mreže. Kao ulaze prima vektor početnog stanja skrivenog sloja, ohe-hot kodirane ulaze i izlaze dimenzija $B \times T \times V$, pri čemu je B veličina minibatcha, T broj vremenskih koraka a V veličina vokabulara. Kao izlaze dobivamo gubitak u tom koraku te skriveno stanje iz **zadnjeg koraka**, koje spremamo kao novo početno stanje.

2.2: Uzorkovanje podataka na temelju naučene mreže

Kod treniranja mreže smo implicitno uzimali idući znak niza umjesto predikciju mreže kao ulaz u idućem vremenskom koraku. Ovaj pristup nije primjenjiv za vrijeme testiranja (kad nam nije dostupan točni idući element niza). Iz ovog razloga greška na skupu za treniranje je umjetno drastično manja od greške na skupu za testiranje (validaciju).

Unutar okvira laboratorijske vježbe nećemo ulaziti u dubinu uzorkovanja nizova, već ćemo obraditi najosnovniji pristup. Svaki `sample_every` minibatcheva pokrenuti ćemo metodu za uzorkovanje iz naše povratne neuronske mreže na idući način:


1. Spremite trenutno skriveno stanje mreže (`h_train`)
2. Inicijalizirajte prazno skriveno stanje (`h0_sample`)
3. Definirajte `seed` niz znakova koji će “zagrijati” mrežu- npr: `seed = 'HAN:\nIs that good or bad?\n\n'`
4. Definirajte duljinu niza za koji ćete uzorkovati (`n_sample=300`)
5. Pokrenite mrežu unaprijed na svim znakovima iz niza “seed”
6. Vratite uzorkovan niz podataka do duljine `n_sample - len(seed)`

Kostur funkcije za uzorkovanje bi mogao (ali ne mora) izgledati kao u nastavku:

```
# ...  
# code not necessarily nested in class definition  
def sample(seed, n_sample):  
    h0, seed_onehot, sample = None, None, None  
    # inicijalizirati h0 na vektor nula  
    # seed string pretvoriti u one-hot reprezentaciju ulaza  
  
    return sample
```

Zadatak 3: višeslojna povratna neuronska mreža u tensorflowu

U duhu blagdana, kod riješenog 3. zadatka će biti dostupan uskoro za eksperimentiranje, te se zadatak neće bodovati!

 dlunizg
ivan.kreso@fer.hr