Μεταπτυχιακό Στην Τεχνητή Νοημοσύνη

Επεξεργασία Φυσικής Γλώσσας

Εργασία 3

Πετρόπουλος Παναγιώτης

panos.petr1@gmail.com

Περιεχόμενα

Transition-Based Dependency Parser 2
- Ερώτημα 1 2
- Ερώτημα 2 2
- Ερώτημα 3 3
- Ερώτημα 4 4
- Ερώτημα 5 4
- Ερώτημα 6 5
- Aggregated Results 6

Graph-Based Dependency Parser 7
- Ερώτημα 1 7
- Ερώτημα 2 7
- Ερώτημα 3 8
- Ερώτημα 4 8
- Aggregated Results 9

Σύγκριση & Συμπεράσματα 10

Transition-Based Dependency Parser

Σε αυτήν την προσέγγιση έχουμε έναν parser ο οποίος προσπαθεί να βρει σχέσεις εξάρτησης μεταξύ λέξεων. Για να το πετύχει αυτό έχει ένα stack με τις λέξεις κάθε φορά (σε κάθε κατάσταση) και ένα input buffer όπου αποτελεί μια στοίβα με όλες τις λέξεις. Κάθε φορά τραβάει 2 λέξεις από το buffer και τις βάζει στο stack. Εξετάζοντας με ένα classifier (Oracle) την πιθανότητα εξάρτησης, διαγράφεται το στοιχείο από το buffer και προβλέπει την εξάρτηση μέσω του stack.

Οι ενέργειες που εκτελούνται για αυτό είναι οι παρακάτω:

- LEFTARC: Οι πρώτες 2 λέξεις της στοίβας ενώνονται με σχέση εξάρτησης με head την πρώτη λέξη. Η δεύτερη λέξη απομακρύνεται από την στοίβα
- RIGHTARC: Οι πρώτες 2 λέξεις της στοίβας ενώνονται με σχέση εξάρτησης με head την δεύτερη λέξη. Η πρώτη λέξη απομακρύνεται από την στοίβα
- SHIFT: Η επόμενη λέξη του input buffer εισάγεται στη στοίβα

Ερώτημα 1

Μετά την απλή εκτέλεση του κώδικα έχουμε το παρακάτω UAS score.

```
================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 68535853.15it/s]
- test UAS: 89.15
Done!
```

Η τιμή του UAS score που επιτυγχάνεται στο Test set είναι 89.15.

Ερώτημα 2

Αλλάζοντας τον κώδικα στο αρχείο *parser_model.py* και συγκεκριμένα διαγράφοντας το υπάρχον Embedding Layer και προσθέτοντας ένα δικό μας, όπως θα δούμε στην παρακάτω εικόνα, μπορούμε να πετύχουμε την δημιουργία ενός Embedding Layer το οποίο θα μαθαίνει τα Embeddings και τα βάρη αυτών κατά την φάση του training.

```
self.emb = nn.Embedding(embeddings.shape[0],self.embed_size)
```

Για να ολοκληρώσουμε όμως την υλοποίηση αυτού του Layer, θα πρέπει στην Forward συνάρτηση του Δικτύου μας, να εφαρμόσουμε το δικό μας Layer σβήνοντας το είδη υπάρχον.

```
x = self.emb(w)
```

Αφού είμαστε έτοιμοι με τον κώδικα, ξεκινάμε την εκτέλεσή του, ώστε να εκπαιδευτεί το μοντέλο μας.

Παρακάτω το αποτέλεσμα του UAS score για το Test set.

```
================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 51229508.77it/s]
- test UAS: 87.80
Done!
```

Η τιμή του UAS score που επιτυγχάνεται στο Test set είναι 87.80.

Η τιμή του UAS έπεσε σε σχέση με το πρώτο ερώτημα, καθώς τα Embeddings των tokens στην ερώτηση 2, μαθαίνονται κατά την φάση του training, σε αντίθεση με το ερώτημα 1 που έχουμε είδη κάποια προ-εκπαιδευμένα Embeddings και γίνονται απλά update τα βάρη αυτών. Πιθανώς, για το ερώτημα 2 να μπορούσαμε να προσεγγίσουμε το score του προηγούμενου ερωτήματος με παραπάνω εποχές στο training.

Ερώτημα 3

Αναλύοντας τον κώδικα που έχει υλοποιηθεί, μπορεί κανείς να καταλάβει ότι το πλήθος των Features που χρησιμοποιούνται για την φάση του Training σε κάθε εποχή, είναι 36.

18 features προκύπτουν από τα tokens και επιπλέον 18 από τα POS Tags.

Για να αφαιρέσουμε τα POS tags, αρχικά θα πρέπει να αντικαταστήσουμε την αρχικοποίηση των αριθμών των features από 36 σε 18, στο αρχείο *parser_model.py* . Αυτό το κάνουμε γιατί θέλουμε να αποφύγουμε προβλήματα σχετικά με τα Dimensions στην είσοδο του κάθε Layer.

Σαν τελευταίο βήμα απομένει να θέσουμε την Boolean μεταβλητή use_pos σε False. Για να γίνει αυτό θα πρέπει να επεξεργαστούμε το αρχείο *parser_utils.py* και συγκεκριμένα στην κλάση Config.

Έπειτα εκτελούμε τον κώδικα και προκύπτει το παρακάτω αποτέλεσμα.

```
================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 64126282.96it/s]
- test UAS: 86.87
Done!
```

Η τιμή του UAS score που επιτυγχάνεται στο Test set είναι 86.87.

Συγκριτικά με το πρώτο ερώτημα μπορεί κανείς να συμπεράνει ότι χωρίς το POS tagging η απόδοση του μοντέλου μειώνεται. Άρα το POS tagging βοηθάει αρκετά το μοντέλο μας να ανιχνεύσει τις συσχετίσεις μεταξύ λέξεων. Πράγμα το οποίο είναι λογικό καθώς το POS tagging μας ορίζει τι μέρος του λόγου είναι ένα token και επομένως είναι μια παραπάνω πληροφορία. Για παράδειγμα, κατά αυτόν τον τρόπο το μοντέλο μας είναι σε θέση να καταλάβει την σχέση εξάρτησης μεταξύ 2 λέξεων, όπου η μια είναι Ουσιαστικό και η άλλη ρήμα. Η μια ακολουθεί την άλλη για ένα συγκεκριμένο νόημα εντός της πρότασης.

Ερώτημα 4

Διατηρώντας τον κώδικα που τροποποιήθηκε στο ερώτημα 3, εκπαιδεύουμε το μοντέλο μας μόνο με λέξεις. Στη συνέχεια για να κάνουμε freeze τα Embeddings, με σκοπό να μην γίνονται update χρησιμοποιούμε το παρακάτω κομμάτι κώδικα στην συνάρτηση train του αρχείου *run.py*.

```
parser.model.pretrained_embeddings.weight.requires_grad = False
```

Σύμφωνα με τον παραπάνω κώδικα, πρακτικά λέμε στο μοντέλο μας ότι δεν χρειάζεται τα gradients για τα συγκεκριμένα βάρη και άρα δεν θα τα κάνει update κατά την φάση του gradient descent.

Ένας άλλος τρόπος για να επιτευχθεί το freezing είναι να τροποποιήσουμε το αρχείο *parser_model.py*

Και συγκεκριμένα στο σημείο που ορίζουμε την αρχιτεκτονική του μοντέλου με τον παρακάτω κώδικα.

```
self.freezed_embeddings = nn.Embedding.from_pretrained(self.pretrained_embeddings.weight,freeze=True)
```

Έπειτα από τις αλλαγές τρέχουμε τον κώδικα και παίρνουμε το παρακάτω αποτέλεσμα.

```
================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 58538249.74it/s]
- test UAS: 84.14
Done!
```

Η τιμή του UAS score που επιτυγχάνεται στο Test set είναι 84.14.

Η απόδοση έπεσε σε σχέση με τα προηγούμενα ερωτήματα , πράγμα το οποίο είναι λογικό γιατί τα βάρη των Embeddings δεν ανανεώνονται.

Ερώτημα 5

Τροποποιώντας το αρχείο *parser_model.py* και προσθέτοντας στην αρχιτεκτονική του δικτύου το παρακάτω κομμάτι κώδικα, προσθέτουμε το επιπλέον Layer διάστασης 100.

```
self.embed_to_hidden2 = nn.Linear(self.n_features*self.embed_size, 100, bias=True)
nn.init.xavier_uniform_(self.embed_to_hidden2.weight) #in-place function
```

Για να μπορέσουμε όμως να το εκπαιδεύσουμε, θα πρέπει και στην forward συνάρτηση να προσθέσουμε το παρακάτω κομμάτι κώδικα.

```
h1 = F.relu(self.embed_to_hidden2(embeddings))
h2 = F.relu(self.embed_to_hidden(h1))
```

Όπου το πρώτο Layer διάστασης 100 δέχεται ως είσοδο το output του Embedding Layer και το δεύτερο Linear Layer δέχεται είσοδο από το πρώτο Linear. Σε κάθε έξοδο όπως βλέπουμε παραπάνω εφαρμόζεται και η ReLU συνάρτηση ενεργοποίησης, η οποία βοηθάει στο να αποφύγουμε το vanishing gradient problem.

Τέλος εκτελούμε τον κώδικα και έχουμε τα παρακάτω αποτελέσματα.

```
================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 54891596.93it/s]
- test UAS: 88.71
Done!
```

Έχοντας υπόψιν μας ότι έχουμε 36 features, pretrained Embeddings τα οποία ανανεώνονται (not freezed) βλέπουμε ότι το score σε σχέση με το ερώτημα 1 είναι ελάχιστα πιο κάτω στο 88.71.

Αυτό μπορεί να συνέβη, διότι όσο προσθέτουμε Layers, προσθέτουμε και περισσότερες μαθηματικές πράξεις και άρα μπορεί η πληροφορία να αρχίζει να χάνεται. Θα πρέπει λοιπόν να κάνουμε parameter tuning γιατί στην ουσία οι παράμετροι έμειναν ίδιοι, ενώ εμείς αλλάξαμε αρχιτεκτονική. Επίσης πιθανώς να χρειάζεται παραπάνω εποχές εκπαίδευσης. Η διαφορά ωστόσο δεν είναι μεγάλη και θα μπορούσε κανείς να την θεωρήσει αμελητέα.

Ερώτημα 6

Για αυτό το ερώτημα δημιουργούμε μια κλάση η οποία υλοποιεί την συνάρτηση ενεργοποίησης Cube.

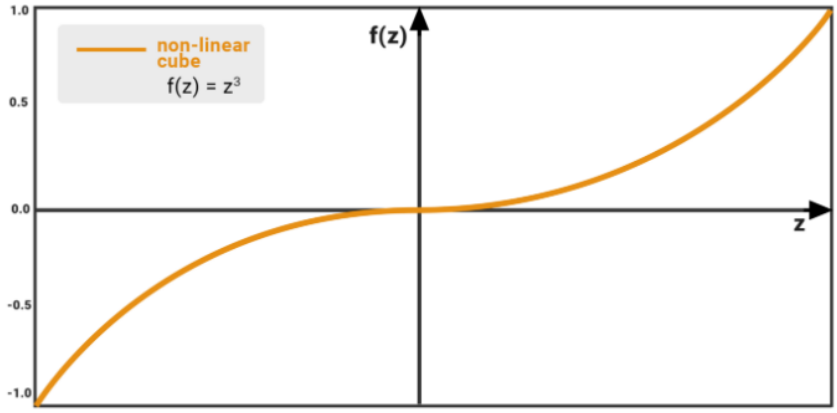
```
class Cube(nn.Module):

    def __init__(self):
        '''
        Init method.
        '''
        super().__init__() # init the base class

    def forward(self, input):
        '''
        Forward pass of the function.
        '''
        return torch.pow(input, 3)
```

Έπειτα εφαρμόζουμε αυτήν την συνάρτηση στην forward συνάρτηση του μοντέλου μας αντικαθιστώντας την ReLU.

Η Γραφική παράσταση της Cube ($f(x) = x^3$) παρουσιάζεται παρακάτω.



Η crossEntropyLoss συνάρτηση κατά το training υλοποιεί στην ουσία και την Softmax οπότε δεν είναι απαραίτητο η εφαρμογή Softmax στο Output του μοντέλου. Άρα έχουμε το παρακάτω αποτέλεσμα.

```
================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 47877538.32it/s]
- test UAS: 87.56
Done!
```

Το score είναι 87.56, πιθανώς θα ήθελε παραπάνω εποχές εκπαίδευσης.

Aggregated Results

Approach	UAS Score
Base line Implementation	**89.15**
Without Pre-Trained Embeddings	87.8
Without POS Tags	86.87
Without POS Tags - Freeze Embeddings	84.14
With Additional Linear Layer	88.71
With Cube Activation Function	87.56

Τα scores είναι αρκετά κοντά μεταξύ τους και σε σχέση με το base line.

Graph-Based Dependency Parser

Σε αυτήν την προσέγγιση δημιουργείται ένας πλήρες συνδεδεμένο Γράφος με κόμβους και ακμές, όπου κάθε κόμβος αποτελεί μια λέξη και κάθε ακμή περιέχει πληροφορία σχετικά με POS tags, Embeddings, Lemmas κλπ.

Τέλος για κάθε ακμή προκύπτει ένα score από κάποιον Classifier όπου το μεγαλύτερο score σημαίνει μεγαλύτερη σχέση εξάρτησης ανάμεσα σε 2 κόμβους του Γράφου (λέξεις).

Note: Για λόγους επεξεργαστικής ισχύος και συντομίας, η διαδικασία του Training έγινε για 3 εποχές. Άρα τα αποτελέσματα θα μπορούσαν να ήταν καλύτερα.

Ερώτημα 1

Αφού αλλάξουμε το - - n_lstm_layer argument και του θέσουμε την τιμή 1, ώστε να έχουμε ένα μόνο στρώμα του BiLSTM, εκτελούμε τον κώδικα με την παρακάτω εντολή:

$ python main.py --train_path data/train.conll --dev_path data/dev.conll --epochs 3 --lr 1e-3 --w_emb_dim 100 --pos_emb_dim 25 --lstm_hid_dim 125 --mlp_hid_dim 100 --n_lstm_layers 1

Παρακάτω παρουσιάζονται τα αποτελέσματα που προκύπτουν από το test set.

```
test results:
Metric              |Score |
--------------------+------+
las                 | 93.65|
uas                 | 92.15|
```

Ερώτημα 2

Για να χρησιμοποιήσουμε τα external embeddings Glove-6B-100d, θα πρέπει αρχικά να τροποποιήσουμε τον υπάρχοντα κώδικα. Συγκεκριμένα στο αρχείο *model.py* και εκεί που ορίζεται η αρχιτεκτονική του Δικτύου, υπάρχει το Layer που διαβάζει από external embeddings. Τροποποιούμε την συνάρτηση *from_pretrained()* και θέτουμε το Boolean όρισμα freeze True.

```
if ex_w_vec is not None and ext_emb_w2i is not None:
    # load external word embeddings
    self.ex_emb_flag = True
    self.ex_emb_w2i = ext_emb_w2i
    self.ex_word_emb = nn.Embedding.from_pretrained(ex_w_vec, freeze=True)
```

Κατά αυτόν τον τρόπο δεν θα γίνονται update τα Embeddings και τα βάρη στην φάση του training.

Για την εκτέλεση του κώδικα τρέχουμε την παρακάτω εντολή στο command line:

$ python drive/MyDrive/NLP/Exe3/main.py --train_path data/train.conll --dev_path data/dev.conll --ext_emb glove.6b/glove.6B.100d --epochs 3 --lr 1e-3 --w_emb_dim 100 --pos_emb_dim 25 --lstm_hid_dim 125 --mlp_hid_dim 100 --n_lstm_layers 1

Στην ουσία χρησιμοποιούμε και ένα επιπλέον argument στην εντολή, το –ext_emb.

Παρακάτω παρουσιάζονται τα αποτελέσματα στο test set.

```
test results:
Metric              |Score |
--------------------+------+
las                 | 93.33|
uas                 | 91.78|
```

Ερώτημα 3

Για να χρησιμοποιήσουμε την συνάρτηση ενεργοποίησης ReLU θα πρέπει αρχικά να αντικαταστήσουμε την tanh στο αρχείο *model.py.* Τροποποιούμε τον κώδικα όπως φαίνεται στην παρακάτω εικόνα.

```
self.slp_out_arc = nn.Sequential(
    # nn.Tanh(),
    nn.ReLU(),
    nn.Linear(mlp_hid_dim, 1, bias=False)
)
```

Έπειτα τον εκτελούμε και προκύπτει το παρακάτω αποτέλεσμα για το test set.

```
Metric              |Score |
--------------------+------+
las                 | 93.73|
uas                 | 92.10|
```

Ερώτημα 4

Για να εφαρμόσουμε το μοντέλο BERT έναντι του BiLSTM, θα πρέπει να ορίσουμε στην αρχιτεκτονική το BERT σαν tokenizer και μετά σαν μοντέλο ώστε να πάρουμε από αυτό τα Embeddings του.

Αντί λοιπόν να χρησιμοποιήσουμε τα pretrained Embeddings θα χρησιμοποιήσουμε το Output του μοντέλου BERT.

Το BERT έχε εκπαιδευτεί να δέχεται ένα συγκεκριμένο pattern στην είσοδό του όπως παρακάτω:

[CLS] + TEXT + [SEP]. Το [SEP] διαχωρίζει τις προτάσεις. Έτσι λοιπόν θα πρέπει να φτιάξουμε την συγκεκριμένη μορφή. Τα δεδομένα μας όμως είναι ήδη tokenized άρα δεν χρειάζεται να πάρουμε και token από το BERT.

Τέλος, Θα πρέπει να λάβουμε υπόψιν μας και τις διαστάσεις των Embeddings στην forward function.

Ακολουθεί παράδειγμα κώδικα που έγινε Import στο Colab από το Drive.

```
print([w[0] for w in sentence]+['[SEP]'])
# tokkens = BertTokenizer.tokenize(sentence)
print(pos_tags)
toks = ['[CLS]']+[w[0] for w in sentence]+['[SEP]']
pad_token = '[PAD]'
toks = toks + ['[PAD]']*(100-len(toks))
pos_tags = pos_tags+['[PAD]']*(25-len(pos_tags))
print(toks)
indexed_tokens = self.tokenizer.convert_tokens_to_ids(tokens=toks)
indexed_pos = self.tokenizer.convert_tokens_to_ids(tokens=[w[0] for w in pos_tags])
segments_ids = [1] * len(sentence)
segments_pos_ids = [1] * len(pos_tags)
tokens_tensor = torch.tensor([indexed_tokens]).to(self.device)
segments_tensors = torch.tensor([segments_ids]).to(self.device)
pos_tensor = torch.tensor([indexed_pos]).to(self.device)
segments_pos_tensors = torch.tensor([segments_pos_ids]).to(self.device)

input_vectors = torch.cat((tokens_tensor, pos_tensor), dim=-1)
input_seg_vectors = torch.cat((segments_tensors, segments_pos_tensors), dim=-1)
# print(tokens_tensor)
# print(segments_tensors)
# print(input_vectors.squeeze(0).shape)


outputs = self.encoder(input_vectors)
hidden_states = outputs[2]

token_embeddings = torch.stack(hidden_states, dim=0)

print(token_embeddings.size())
token_embeddings = torch.squeeze(token_embeddings, dim=1)

print(token_embeddings.size())
```

Σημείωση: Τα pos_tags όπως αναφέρονται στο paper, δεν συμπεριλαμβάνονται στον τελικό κώδικα και στον πείραμα.

Αφού εκτελέσουμε τον κώδικα έχουμε το παρακάτω αποτέλεσμα.

```
Metric              |Score |
--------------------+------+
las                 | 92.46|
uas                 | 90.48|
```

Αυτό το αποτέλεσμα προέκυψε από 2 εποχές training καθώς δεν πρόλαβα λόγω Limitations του Colab. Θεωρητικά στις ίδιες τιμές θα κυμαίνεται και το αποτέλεσμα στο test set.

Aggregated Results

Approach	UAS Score	LAS Score
Base line Implementation with 1 Layer BiLSTM	92.15	93.65
Trained Embeddings	91.78	93.33
With ReLU activation Function in MLPs	92.1	93.73
Embeddings only from BERT model	90.48	92.46

Συγκριτικά όλες οι τιμές είναι παρόμοιες με καλύτερη απόδοση και στα 2 scores ο Parser με την εφαρμογή της ReLU. Αυτό διότι σε σχέση με την Tahn, ή Relu κάνει τους υπολογισμού πιο εύκολους και μας γλυτώνει από το vanishing gradient problem. Δηλαδή, Οι παράγωγοι κατά το back propagation δεν πρόκειται να είναι πάνω από κάποιες τιμές, με αποτέλεσμα να εξαφανίζονται τα βάρη. Στην Tahn, λαμβάνονται υπόψιν οι αρνητικές τιμές ενώ στην Relu δεν έχουμε αρνητικές τιμές καθώς όλες αυτές αντιλαμβάνονται ως μηδέν.

Σύγκριση & Συμπεράσματα

Σύμφωνα με τα αποτελέσματα παραπάνω οι Graph-based dependency parsers, πέτυχαν καλύτερα και υψηλότερα scores.

Σε γενικές γραμμές οι Transition-based dependency parsers τείνουν να εκπαιδεύονται πιο γρήγορα και πιο εύκολα, ενώ οι Graph-based dependency parsers μπορούν να παρέχουν πιο ακριβή αποτελέσματα.

Οι Transition-based dependency parsers εκπαιδεύονται γρήγορα και εύκολα επειδή βασίζονται σε έναν απλό αλγόριθμο που κάνει προβλέψεις μία κάθε φορά, χωρίς να χρειάζεται να λάβει υπόψη το πλήρες δέντρο ανάλυσης. Αυτό τα καθιστά ιδιαίτερα κατάλληλα για διαδικτυακές και incremental parsing εργασίες. Οι Graph-based dependency parsers, από την άλλη πλευρά, μπορούν να παρέχουν πιο ακριβή αποτελέσματα επειδή λαμβάνουν υπόψη το πλήρες δέντρο ανάλυσης (parsing tree) όταν κάνουν προβλέψεις. Αυτό τους επιτρέπει να συλλαμβάνουν εξαρτήσεις μεγάλων “αποστάσεων” που μπορεί να παραλείψουν οι Transition-based dependency parsers.