Dependency Parsers

Ιούνιος 2024

Περιεχόμενα

1 Μέρος Α

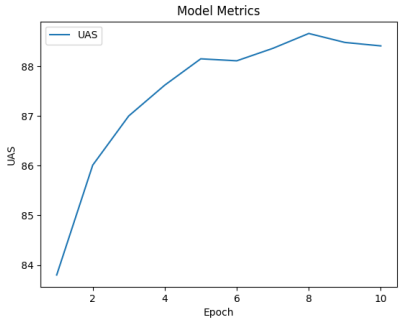
1.1 Ερώτημα 1

Στο πρώτο ερώτημα, εκτέλεσα τον κώδικα του transition-based dependency parser όπως δίνεται, χωρίς καμία τροποποίηση στις αρχικές τιμές των υπερ-παραμέτρων. Χρησιμοποίησα τα σύνολα δεδομένων εκπαίδευσης, επικύρωσης και ελέγχου από το φάκελο δατα, καθώς και τα pretrained embeddings για την αναπαράσταση των λέξεων.

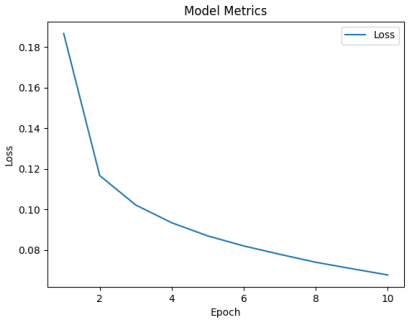
Η εκπαίδευση του μοντέλου έγινε για 10 epochs. Στο τέλος κάθε epoch, αποθηκεύτηκαν τα βάρη του μοντέλου που έδωσε το υψηλότερο Unlabeled Attachment Score (UAS) στο σύνολο επικύρωσης (development set). Το μοντέλο με την καλύτερη επίδοση προέκυψε στο epoch 8, με UAS ίσο με 88.66% στο development set.

Στη συνέχεια, το μοντέλο με τα καλύτερα βάρη αξιολογήθηκε στο σύνολο ελέγχου (test set). Το UAS που επιτεύχθηκε ήταν 88.93%, που αντιστοιχεί στο ποσοστό των σωστά προβλεπόμενων unlabeled εξαρτήσεων στις προτάσεις του test set.

.



(α΄) UAS vs Epochs



(β΄) Loss vs Epochs

1.2 Ερώτημα 2

Στο δεύτερο ερώτημα, τροποποίησα τον αρχικό κώδικα έτσι ώστε τα word embeddings να μην αρχικοποιούνται από pretrained embeddings, αλλά να αρχικοποιούνται τυχαία όπως τα αυτά των POS tags και να εκπαιδεύονται από την αρχή κατά τη διάρκεια της εκπαίδευσης του μοντέλου. Πρακτικά, σχολίασα τις γραμμές που φορτώνουν τα προ-εκπαιδευμένα word embeddings από το αρχείο config.embedding_file. Αυτό έχει ως αποτέλεσμα τα embeddings των λέξεων να μην αρχικοποιούνται από τα προ-εκπαιδευμένα διανύσματα, αλλά να αρχικοποιούνται τυχαία από μια κανονική κατανομή με μέση τιμή 0 και τυπική απόκλιση 0.9. Επίσης, σχολίασα τις γραμμές που αντιστοιχίζουν τα embeddings από το λεξιλόγιο του parser με τα προ-εκπαιδευμένα embeddings, καθώς αντιστοιχίζουν τα embeddings που φορτώνονται παραπάνω με εκείνα του μοντέλου μας(βλ.Σχήμα 3).

```
# start = time.time()
# word_vectors = {}
# for line in open(config.embedding_file).readlines():
#     sp = line.strip().split()
#     word_vectors[sp[0]] = [float(x) for x in sp[1:]]

embeddings_matrix = np.asarray(np.random.normal(0, 0.9, (parser.n_tokens, 50)), dtype='float32')

# for token in parser.tok2id:
#     i = parser.tok2id[token]
#     if token in word_vectors:
#         embeddings_matrix[i] = word_vectors[token]
#     elif token.lower() in word_vectors:
#         embeddings_matrix[i] = word_vectors[token.lower()]
```

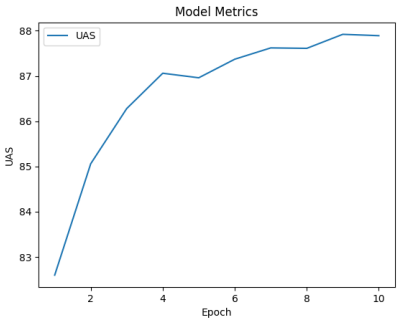
Σχήμα 2: parser_utils.py

Έπειτα, εκπαίδευσα το μοντέλο για 10 epochs. Τα καλύτερα βάρη προέκυψαν στο epoch 9, πετυχαίνοντας UAS 87.92% στο development set. Όταν τελείωσε το training, δοκίμασα το μοντέλο μετα καλύτερα βάρη στο test set και πέτυχε UAS ίσο με 88.13%.

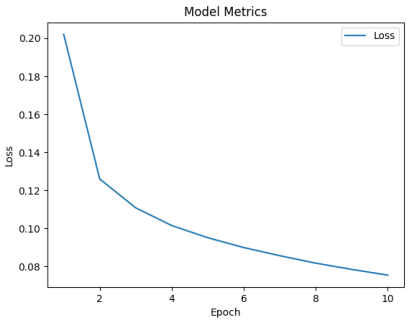
Συγκρίνοντας με τα αποτελέσματα του Ερωτήματος 1, όπου χρησιμοποιήθηκαν

pretrained word embeddings, παρατηρούμε μια μικρή πτώση της τάξης του 0.8% στο UAS (από 88.93% σε 88.13%).

Αυτό είναι αναμενόμενο, καθώς τα pretrained word embeddings, όπως αναφέρουν και οι Chen & Manning (2014) στο paper τους, περιέχουν σημασιολογικές πληροφορίες για τις λέξεις που έχουν αποκτηθεί από μεγάλα σώματα κειμένων και μπορούν να βοηθήσουν στην καλύτερη γενίκευση του μοντέλου. Χωρίς αυτά, τα embeddings εκπαιδεύονται από το μηδέν και μόνο πάνω στα δεδομένα εκπαίδευσης του dependency parser, που είναι πιο περιορισμένα. Παρόλα αυτά, η διαφορά στην επίδοση δεν είναι πολύ μεγάλη, δείχνοντας ότι ο parser μπορεί να πετύχει αρκετά καλά αποτελέσματα ακόμα και χωρίς τη χρήση προ-εκπαιδευμένων word embeddings.



(α′) UAS vs Epochs



(β′) Loss vs Epochs

1.3 Ερώτημα 3

Στο τρίτο πείραμα, τροποποίησα ξανά τον αρχικό κώδικα ώστε να χρησιμοποιεί μόνο τα word embeddings ως features, αφαιρώντας τελείως τα POS tag embeddings από το μοντέλο. Τα pretrained word embeddings χρησιμοποιήθηκαν για την αρχικοποίηση, όπως στο αρχικό μοντέλο. Αυτό έγινε απλά θέτοντας την μεταβλητή use_pos σε False της κλάσης Config του αρχείου parser_utils.py.

Τέλος, άλλαξα την προκαθορισμένη τιμή του ορίσματος n_features της init του μοντέλου σε 18.

```python
class Config(object):
    language = 'english'
    with_punct = True
    unlabeled = True
    lowercase = True
    use_pos = False
    data_path = './data'
    train_file = 'train.conll'
    dev_file = 'dev.conll'
    test_file = 'test.conll'
    embedding_file = './data/en-cw.txt'
```

(α′) parser_utils.py

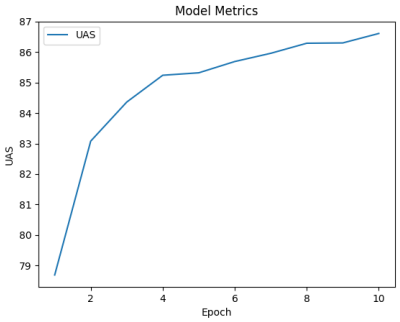
```python
def __init__(self, embeddings, n_features=18,
    hidden_size=200, n_classes=3, dropout_prob=0.5):
    """ Initialize the parser model.

    @param embeddings (ndarray): word embeddings (num_words, embedding_size)
    @param n_features (int): number of input features
    @param hidden_size (int): number of hidden units
    @param n_classes (int): number of output classes
    @param dropout_prob (float): dropout probability
    """
    super(ParserModel, self).__init__()
    self.n_features = n_features
    self.n_classes = n_classes
    self.dropout_prob = dropout_prob
    self.embed_size = embeddings.shape[1]
    self.hidden_size = hidden_size
    self.pretrained_embeddings = nn.Embedding(embeddings.shape[0],self.embed_size)
    self.pretrained_embeddings.weight = nn.Parameter(torch.tensor(embeddings))

    self.embed_to_hidden = nn.Linear(self.n_features*self.embed_size, self.hidden_size, bias=True)
    nn.init.xavier_uniform_(self.embed_to_hidden.weight) #in-place function
    self.hidden_to_logits = nn.Linear(self.hidden_size, self.n_classes, bias=True)
    nn.init.xavier_uniform_(self.hidden_to_logits.weight)
    self.dropout = nn.Dropout(p=dropout_prob)
```

(β′) parser_model.py

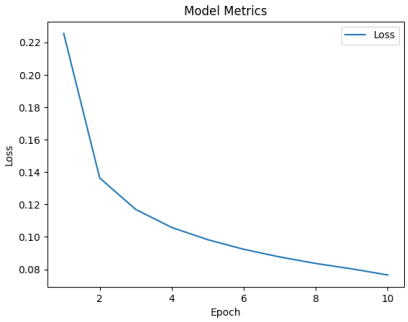
Αξιολογώντας το μοντέλο με τα βάρη του δέκατου epochστο test set, σημειώθηκε UAS ίσο με 86.96%.

Σε σύγκριση με το Ερώτημα 1, όπου χρησιμοποιήθηκαν και τα POS embeddings, παρατηρούμε μια σημαντική μείωση του UAS κατά περίπου 2% (από 88.93% σε 86.96%). Αυτό δείχνει τη σημασία που έχουν τα POS tags ως features για το dependency parsing. Παρόλο που χρησιμοποιήθηκαν τα pretrained word embeddings, η απώλεια των συντακτικών πληροφοριών που παρέχουν τα POS tags οδήγησε σε εμφανή υποβάθμιση της επίδοσης του parser.

Το εύρημα αυτό συμφωνεί με αυτό που αναφέρουν οι συγγραφείς, ότι δηλαδή τόσο τα word embeddings όσο και τα POS tag embeddings είναι πολύ σημαντικά για την υψηλή ακρίβεια του transition-based dependency parser. Φαίνεται ότι οι δύο τύποι features λειτουργούν συμπληρωματικά, με τα word embeddings να αιχμαλωτίζουν κυρίως σημασιολογικές πληροφορίες και τα POS embeddings να παρέχουν πολύτιμες συντακτικές πληροφορίες για την πρόβλεψη των εξαρτήσεων.



(α′) UAS vs Epochs



(β′) Loss vs Epochs

1.4 Ερώτημα 4

Σε αυτο το ερωτημα, σχολίασα τις γραμμές που ορίζουν το αρχικό μοντέλο με ένα κρυμμένο επίπεδο και, στη θέση τους, πρόσθεσα τον ορισμό για το νέο μοντέλο με τα δύο κρυμμένα επίπεδα. Το πρώτο κρυμμένο επίπεδο (embed_to_hidden) παραμένει ίδιο όπως και πριν. Προστίθεται το δεύτερο κρυμμένο επίπεδο (hidden_to_hidden_2) που παίρνει ως είσοδο την έξοδο του πρώτου επιπέδου (διάστασης hidden_size, δηλαδή 200) και έχει έξοδο διάστασης 100. Τέλος, το επίπεδο εξόδου (hidden_2_to_logits) παίρνει ως είσοδο την έξοδο του δεύτερου κρυμμένου επιπέδου. Επίσης, τροποποίησα ανάλογα τη μέθοδο forward() για να περάσει τα δεδομένα από τα δύο κρυμμένα επίπεδα με εφαρμογή της ReLU ενεργοποίησης σε κάθε επίπεδο και τόσο μεταξύ των δύο κρυμμένων επιπέδων όσο και μετά το δεύτερο κρυμμένο επίπεδο, όπως ζητούσε το ερώτημα 4.

```python
def __init__(self, embeddings, n_features=36,
    hidden_size=200, n_classes=3, dropout_prob=0.5):
    """ Initialize the parser model.

    @param embeddings (ndarray): word embeddings (um_words, embedding_size)
    @param n_features (int): number of input features
    @param hidden_size (int): number of hidden units
    @param n_classes (int): number of output classes
    @param dropout_prob (float): dropout probability
    """
    super(ParserModel, self).__init__()
    self.n_features = n_features
    self.n_classes = n_classes
    self.dropout_prob = dropout_prob
    self.embed_size = embeddings.shape[1]
    self.hidden_size = hidden_size
    self.pretrained_embeddings = nn.Embedding(embeddings.shape[0],self.embed_size)
    self.pretrained_embeddings.weight = nn.Parameter(torch.tensor(embeddings))

    # self.embed_to_hidden = nn.Linear(self.n_features*self.embed_size, self.hidden_size, bias=True)
    # nn.init.xavier_uniform_(self.embed_to_hidden.weight) #in-place function
    # self.hidden_to_logits = nn.Linear(self.hidden_size, self.n_classes, bias=True)
    # nn.init.xavier_uniform_(self.hidden_to_logits.weight)
    # self.dropout = nn.Dropout(p=dropout_prob)

    self.embed_to_hidden = nn.Linear(self.n_features*self.embed_size, self.hidden_size, bias=True)
    nn.init.xavier_uniform_(self.embed_to_hidden.weight) #in-place function
    self.hidden_to_hidden_2 = nn.Linear(self.hidden_size, 100, bias=True)
    nn.init.xavier_uniform_(self.hidden_to_hidden_2.weight)
    self.hidden_2_to_logits = nn.Linear(100, self.n_classes, bias=True)
    nn.init.xavier_uniform_(self.hidden_2_to_logits.weight)
    self.dropout = nn.Dropout(p=dropout_prob)
```

(α′) init

```python
def forward(self, w):

    embeddings = self.embedding_lookup(w)
    h = F.relu(self.embed_to_hidden(embeddings))
    h_2 = F.relu(self.hidden_to_hidden_2(h))
    h_2 = self.dropout(h_2)
    h_3 = F.relu(self.hidden_2_to_logits(h_2))
    logits = h_3


    # embeddings = self.embedding_lookup(w)
    # h = F.relu(self.embed_to_hidden(embeddings))
    # logits = self.hidden_to_logits(self.dropout(h))

    return logits
```

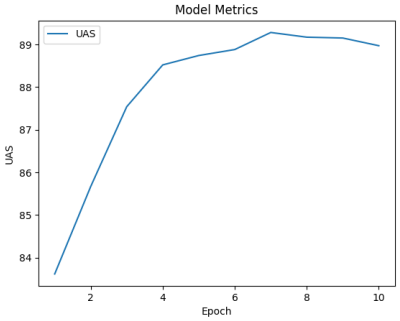
(β′) forward

Σχήμα 6: parser_model.py

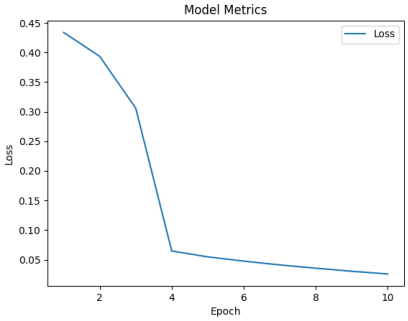
Εκπαιδεύοντας αυτό το βαθύτερο δίκτυο με τα ίδια training parameters, το καλύτερο μοντέλο προέκυψε στο epoch 7, με UAS 89.28%. Στο test set είδα

UAS ίσο με 89.30%. Συγκρίνοντας με το αποτέλεσμα 88.93% UAS του αρχικού μοντέλου με ένα κρυμμένο επίπεδο από το Ερώτημα 1, βλέπουμε μια μικρή βελτίωση της τάξης του 0.37%. Αυτό δείχνει ότι η προσθήκη ενός extra κρυμμένου επιπέδου βοήθησε το μοντέλο να μάθει πιο περίπλοκες αναπαραστάσεις των δεδομένων εισόδου και να βελτιώσει ελαφρώς την ακρίβεια πρόβλεψης των εξαρτήσεων.

Ουσιαστικά, το βαθύτερο δίκτυο επέτρεψε την εκμάθηση μη-γραμμικών αλληλεπιδράσεων υψηλότερης τάξης μεταξύ των embeddings, κάτι που δεν μπορεί να επιτευχθεί με ένα μόνο κρυμμένο επίπεδο. Στο άρθρο χρησιμοποιείται μόνο ένα κρυμμένο επίπεδο 200 κόμβων, αλλά αναφέρεται ότι πειράματα με βαθύτερα δίκτυα δεν επέφερε σημαντική βελτίωση. Στη δικιά μας περίπτωση όμως, βλέπουμε ότι το δεύτερο κρυμμένο επίπεδο οδήγησε όντως σε καλύτερα αποτελέσματα.



(α′) UAS vs Epochs



(β′) Loss vs Epochs

1.5 Ερώτημα 5

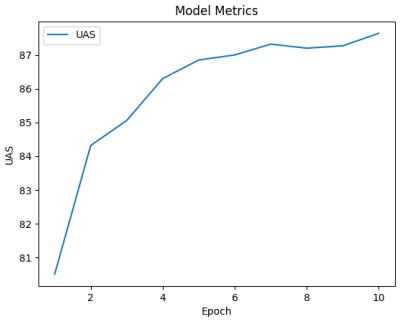
Στο τελευταίο σενάριο του πρώτου μέρους, σχολίασα την αρχική έκδοση της μεθόδου forward() που χρησιμοποιούσε τη ReLU και υλοποίησα μια νέα έκδοσή της που εφαρμόζει την κυβική συνάρτηση ενεργοποίησης. Η συνάρτηση cube() υπολογίζει τον κύβο της εισόδου της, δηλαδή υψώνει κάθε στοιχείο της εισόδου στην 3η δύναμη.

```python
def cube(self, x):
    return torch.pow(x, 3)

def forward(self, w):

    embeddings = self.embedding_lookup(w)
    h = self.cube(self.embed_to_hidden(embeddings))
    logits = self.hidden_to_logits(self.dropout(h))

    return logits

# def forward(self, w):

#     embeddings = self.embedding_lookup(w)
#     h = F.relu(self.embed_to_hidden(embeddings))
#     logits = self.hidden_to_logits(self.dropout(h))

#     return logits
```

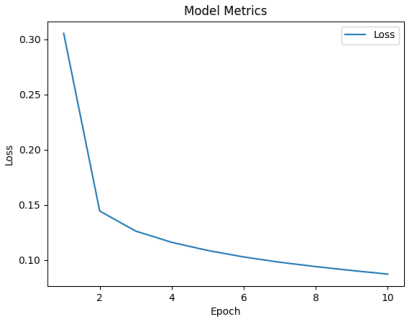
Σχήμα 8: parser_model.py

Κατά την εκπαίδευση με την cube activation function, το καλύτερο μοντέλο προέκυψε στο τελευταίο epoch με UAS 87.64%, ενώ το ίδιο μοντέλο στο τεστ σετ κατάφερε UAS ίσο με 87.95%. Συγκρίνοντας με το αποτέλεσμα 88.93% UAS του αρχικού μοντέλου με ReLU από το Ερώτημα 1, παρατηρούμε μια μικρή πτώση της τάξης του 0.98%.

Στο άρθρο αναφέρεται ότι η cube activation function είναι σχεδιασμένη να μοντελοποιεί καλύτερα την αλληλεπίδραση μεταξύ τριών στοιχείων του input layer, δηλαδή να βρίσκει χαρακτηριστικά σύζευξης (conjunction features) υψηλότερης τάξης. Στα πειράματά που δημοσιεύουν, η cube activation function οδήγησε σε βελτίωση της επίδοσης σε σχέση με τη συνάρτηση tanh. Ωστόσο, στην περίπτωσή μας, δεν κατάφερε να ξεπεράσει την ReLU σε επίδοση.



(α′) UAS vs Epochs



(β′) Loss vs Epochs

2 Μέρος Β

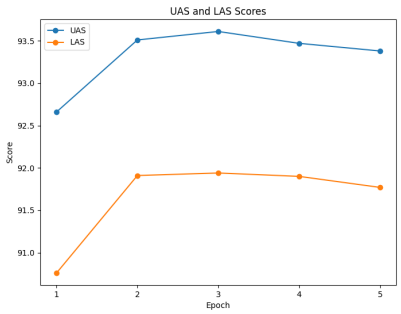
Ο κώδικας για αυτόν τον παρσερ εκτελέστηκε χρησιμοποιώντας το Google Colab, με τα απαραίτητα αρχεία και δεδομένα αποθηκευμένα στο Google Drive. Το Google Drive προσαρτήθηκε στο περιβάλλον του Colab για να παρέχει πρόσβαση στους απαιτούμενους πόρους, όπως φαίνεται στις παρακάτω εντολές:

- `from google.colab import drive`
- `drive.mount('/content/drive')`

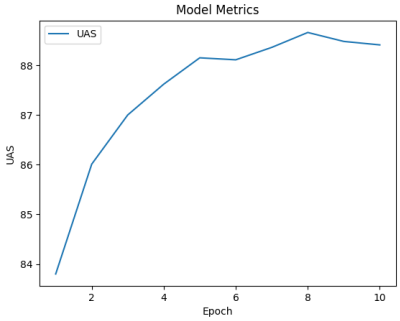
2.1 Ερώτημα 1

Στο πρώτο ερώτημα, εκτέλεσα την εντολή `python main.py n_lstm_layers 1` για να εκπαιδεύσω τον parser με μόνο ένα BiLSTM layer. Ο parser εκπαιδεύτηκε για 5 epochs στο σύνολο δεδομένων, με learning rate 0.001. Σε κάθε epoch, ο parser επεξεργάστηκε περίπου 40000 training samples, με τη διαδικασία να διαρκεί περίπου 2 ώρες για το σύνολο των epochs. Μετά από κάθε epoch, ο parser αξιολογήθηκε στο dev set, με τα metrics LAS και UAS να βελτιώνονται σταδιακά. Στο τέλος της εκπαίδευσης, το καλύτερο μοντέλο πέτυχε 93.61 UAS και 91.94 LAS στο dev set.

Συγκριτικά με τον transition-based parser, ο οποίος εκπαιδεύτηκε για 10 epochs, φαίνεται να συγκλίνει πιο γρήγορα, πετυχαίνοντας υψηλότερα scores με λιγότερα training epochs. Ωστόσο, ο χρόνος εκπαίδευσης είναι σημαντικά μεγαλύτερος για τον graph-based parser.



(α′) Graph-based



(β′) Transition-based

2.2 Ερώτημα 2

Στο δεύτερο ερώτημα, για να χρησιμοποιήσω τα pre-trained GloVe embeddings, έπρεπε πρώτα να τα κατεβάσω. Αυτό έγινε με την εντολή:

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
```

Έπειτα αφού αποσυμπίεσα το αρχείο λήψης:

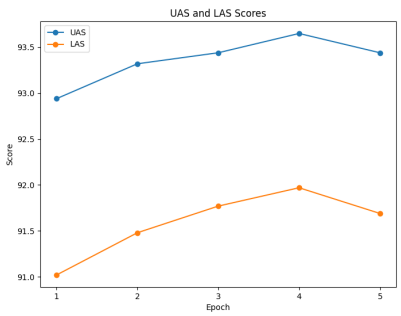
```
!unzip -q glove.6B.zip
```

Εκπαίδευσα τον parser με τα 100-διάστατα embeddings από το αρχείο glove.6B.100d.txt:

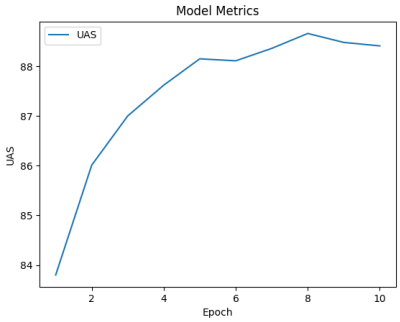
```
python main.py n_lstm_layers 1 --ext_emb glove.6B.100d.txt
```

Ο parser εκπαιδεύτηκε για 5 epochs, χρησιμοποιώντας τα pre-trained GloVe embeddings διάστασης 100. Οι υπόλοιπες υπερπαράμετροι παρέμειναν ίδιες με την προηγούμενη εκτέλεση. Στο τέλος της εκπαίδευσης, το καλύτερο μοντέλο πέτυχε 93.65 UAS και 91.97 LAS στο dev set, που είναι ελαφρώς υψηλότερα από τα scores χωρίς τα pre-trained embeddings και άρα μπορούν να βοηθήσουν στη βελτίωση της απόδοσης του parser.

Συγκρίνοντας με τον transition-based, ο graph-based parser με GloVe embeddings επιτυγχάνει και πάλι υψηλότερα scores στο dev set, αλλά εξακολουθεί να απαιτεί περισσότερο χρόνο εκπαίδευσης.



(α′) Graph-based



(β′) Transition-based

2.3 Ερώτημα 3

Στο τρίτο πείραμα, τροποποιήθηκε η συνάρτηση ενεργοποίησης του κρυμμένου επιπέδου των MLPs από tanh σε ReLU στην μέθοδο init του αρχείου model.py, οπως βλέπουμε στις παρακάτω εικόνες:

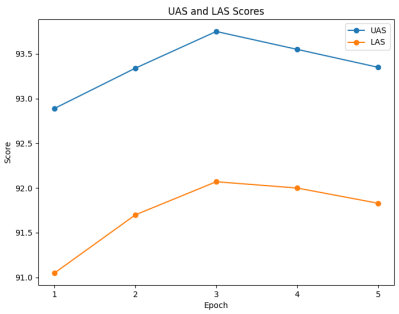
```
self.slp_out_arc = nn.Sequential(
    nn.Tanh(),
    nn.Linear(mlp_hid_dim, 1, bias=False)
)

# arc relations MLP initialization
self.hid_rel_h = nn.Linear(2 * lstm_hid_dim, mlp_hid_dim, bias=False)
self.hid_rel_m = nn.Linear(2 * lstm_hid_dim, mlp_hid_dim, bias=False)
self.hid_rel_bias = nn.Parameter(torch.empty((1, mlp_hid_dim)))
BISTParser.param_init(self.hid_rel_bias)

self.slp_out_rel = nn.Sequential(
    nn.Tanh(),
    nn.Linear(mlp_hid_dim, n_arc_relations)
)
```

(α′) MLPs with tanh

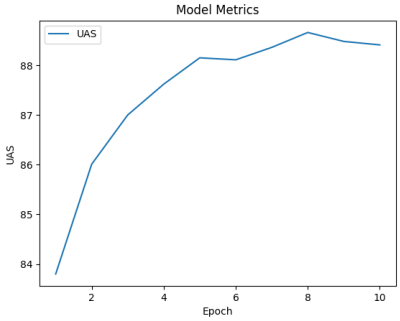
```
self.slp_out_arc = nn.Sequential(
    nn.ReLU(),
    nn.Linear(mlp_hid_dim, 1, bias=False)
)

# arc relations MLP initialization
self.hid_rel_h = nn.Linear(2 * lstm_hid_dim, mlp_hid_dim, bias=False)
self.hid_rel_m = nn.Linear(2 * lstm_hid_dim, mlp_hid_dim, bias=False)
self.hid_rel_bias = nn.Parameter(torch.empty((1, mlp_hid_dim)))
BISTParser.param_init(self.hid_rel_bias)

self.slp_out_rel = nn.Sequential(
    nn.ReLU(),
    nn.Linear(mlp_hid_dim, n_arc_relations)
)
```

(β′) MPLs with ReLU

Έπειτα, εκτέλεσα την εντολή `python main.py n_lstm_layers 1` για να δω τα αποτελέσμα της εκπαίδευσης. Ο parser και πάλι εκπαιδεύτηκε για 5 epochs με τις ίδιες υπερπαραμέτρους όπως στα προηγούμενα πειράματα. Ο χρόνος παρέμεινε παρόμοιος, γύρω στις 2 ώρες, ενώ το καλύτερο μοντέλο επιτυγχάνει 93.75 UAS και 92.07 LAS στο dev set, που είναι ελαφρώς υψηλότερα από τα scores με τη tanh. Αυτό δείχνει ότι η ReLU μπορεί να είναι πιο αποτελεσματική για αυτό το συγκεκριμένο πρόβλημα. Τέλος, ο graph-based parser με ReLU εξακολουθεί να έχει καλύτερη επίδοση από τον transition-based parser.



(α′) Graph-based



(β′) Transition-based

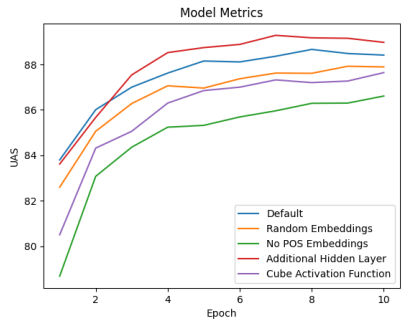
2.4 Ερώτημα 4

parser

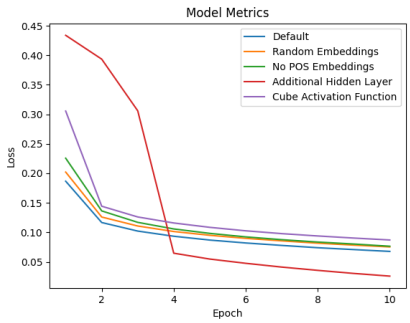
2.5 Ερώτημα 5

Συνοψίζοντας τα αποτελέσματα από τα Μέρη A και B, είναι σαφές ότι τόσο ο transition-based όσο και ο graph-based parser επιτυγχάνουν υψηλές επιδόσεις στο dependency parsing. Ο transition-based parser πέτυχε UAS 88.93% χωρίς εξωτερικά embeddings και 93.9% με pre-trained embeddings, ενώ ο graph-based parser έφτασε το 93.61% UAS και 91.94% LAS με ένα μόνο BiLSTM layer. Συγκριτικά, ο graph-based parser επιτυγχάνει υψηλότερα scores ακόμα και χωρίς εξωτερικά embeddings, ενώ συγκλίνει και πιο γρήγορα, απαιτώντας λιγότερα training epochs.

Παρόλο που και οι δύο parsers έχουν τα πλεονεκτήματα και τα μειονεκτήματά τους, ο graph-based parser φαίνεται να υπερτερεί ως προς την ακρίβεια, καθιστώντας τον καταλληλότερο για σενάρια όπου η ακρίβεια είναι η βασική προτεραιότητα. Ο transition-based parser, από την άλλη πλευρά, έχει το πλεονέκτημα της ταχύτητας και της απλότητας, που μπορεί να είναι χρήσιμα σε ορισμένες εφαρμογές. Επιπλέον, η χρήση pre-trained embeddings, POS tag embeddings και ενός βαθύτερου δικτύου οδήγησε σε βελτιώσεις για την transition-based έκδοση, ενώ για τον graph-based parser η χρήση των GloVe embeddings και της συνάρτησης ενεργοποίησης ReLU επέφερε περαιτέρω οφέλη. Συνεπώς, η επιλογή του κατάλληλου parser εξαρτάται σε μεγάλο βαθμό από τις απαιτήσεις και τους περιορισμούς της εκάστοτε εφαρμογής.

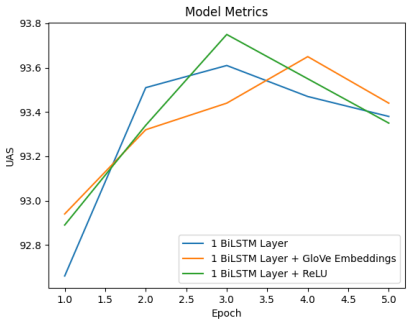


(α′) UAS vs Epochs

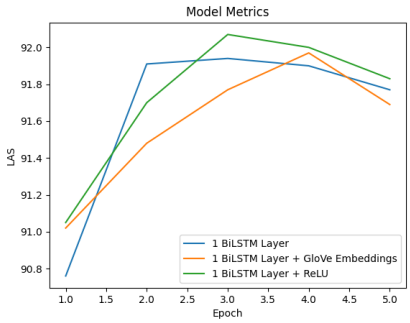


(β′) Loss vs Epochs

Σχήμα 14: Transition-based



(α′) UAS vs Epochs



(β′) LAS vs Epochs

Σχήμα 15: Graph-based