Υπολογιστική Νοημοσύνη

Αναφορά εργασίας,
στα Βαθιά Νευρωνικά Δίκτυα

Αλέξανδρος Τρεμόπουλος

Μας ζητήθηκε να υλοποιήσουμε ένα νευρωνικό δίκτυο βαθιάς μάθησης. Στην αναφορά αυτή παρουσιάζεται το dataset, η προ-επεξεργασία του, οι μηχανές βαθιάς μάθησης που χρησιμοποίησα πάνω σε αυτό, και τα αναλυτικά της αποτελέσματα.

Dataset: IMdB dataset/ The Large Movie Review Dataset. Αυτό το dataset, περιέχει 25.000 κριτικές ταινιών (καλές ή κακές), για training και άλλες 25.000 για testing. Είναι μία NLP εφαρμογή για sentiment analysis. Σκοπός είναι να βρεθεί αν μια δοθέντα κριτική ταινίας έχει θετικό ή αρνητικό συναίσθημα.

Χρησιμοποιώ python και keras, από το οποίο, παίρνω και το imdb dataset.

Το dataset έχει δύο κλάσεις ([0,1]) για καλό και κακό sentiment στην κριτική. Επίσης περιέχει 88.585 ξεχωριστές λέξεις και οι κριτικές έχουν μ.ο. 234,76 λέξεις με διασπορά 172.91. Αυτά φαίνονται από τις εξής γραμμές κώδικα:

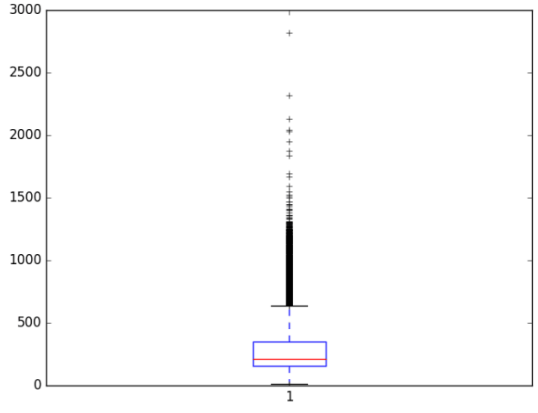
```
# summarize size
print("Training data: ")
print(X_train.shape)
print(y_train.shape)

# Summarize number of classes
print("Classes: ")
print(numpy.unique(y_train))

# Summarize number of words
print("Number of words: ")
print(len(numpy.unique(numpy.hstack(X_train))))

# Summarize review length
print("Review length: ")
result = [len(x) for x in X_train]
print("Mean %.2f words (%f)" % (numpy.mean(result), numpy.std(result)))
```

Με τη χρήση ενός boxplot για το μέγεθος των κριτικών σε λέξεις, καταλαβαίνουμε ότι θα καλύψουμε την πλειοψηφία της κατανομής αν χρησιμοποιούμε 400 με 500 λέξεις.



Προ-επεξεργασία:

Κάνοντας load το dataset δίνεις σε μεταβλητή τον αριθμό των πιο συχνών λέξεων που θα χρησιμοποιήσεις. Θα παρουσιάσω δοκιμές με διάφορα πλήθη.

```
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

Έπειτα θα κάνουμε pad τις κριτικές στις 500 λέξεις αρχικά, για να έχουν όλες οι κριτικές το ίδιο μέγεθος, δηλαδή όσες κριτικές είναι παραπάνω από 500 λέξεις θα κοπούν στις 500, και όσες είναι λιγότερες θα συμπληρωθούν με κενά για να γίνουν 500.

```
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
```

Έπειτα θα φτιάξουμε ένα Embedding layer. Με αυτό τον τρόπο οι λέξεις γίνονται encoded ως διανύσματα πραγματικών τιμών σε ένα χώρο υψηλής διάστασης, όπου οι ομοιότητα μεταξύ των λέξεων όσο αφορά το νόημα τους, μεταφράζεται στο πόσο κοντά βρίσκονται στο διανυσματικό χώρο. Έτσι οι ξεχωριστές λέξεις, γίνονται mapped σε διανύσματα συνεχών αριθμών.

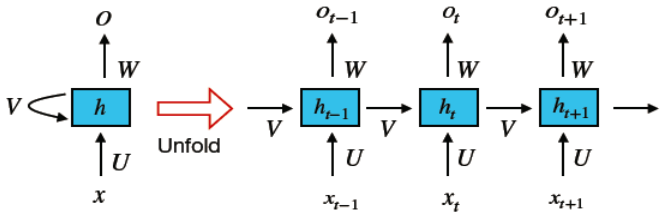
Το Embedding Layer του keras, χρειάζεται 3 παραμέτρους. Πρώτα, χρειάζεται το μέγεθος του λεξιλογίου, το μέγιστο δηλαδή αριθμό ξεχωριστών λέξεων. Η 2η παράμετρος υποδεικνύει το μέγεθος των embedding διανυσμάτων, το οποίο είναι πάντα 32 και τέλος χρειάζεται το μέγεθος της κάθε πρότασης.

```
embedding_layer = Embedding(top_words, 32, input_length=max_words)
```

Δημιουργείται ένας πίνακας που κάνει map integers σε embedding διανύσματα και όταν το δίκτυο εκπαιδευτεί μπορούμε να πάρουμε τα βάρη του embedding layer.

RNN-LSTM:

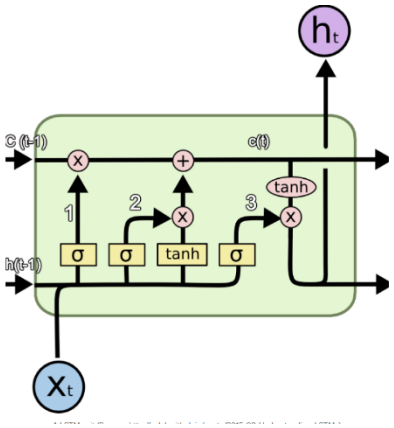
Αρχικά στην εργασία δοκιμάζω ένα RNN(Recurrent Neural Network) και συγκεκριμένα την εκδοχή LSTM (Long Short Term Memory). Τα RNN χρησιμοποιούνται για διαδοχικές πληροφορίες. Ένα τέτοιο δίκτυο αναδρομικά κάνει υπολογισμούς σε όλα τα instances μιας input ακολουθίας βάση των προηγούμενων υπολογισμένων αποτελεσμάτων. Αυτές οι ακολουθίες τροφοδοτούνται διαδοχικά, μία προς μία, σε ένα επαναλαμβανόμενο unit.



Έτσι ένα RNN, έχει την ικανότητα να αποστηθίσει τα αποτελέσματα προηγούμενων υπολογισμών και να χρησιμοποιήσει αυτή την πληροφορία στον τρέχον υπολογισμό.

Στα RNN η πληροφορία μπαίνει συνεχώς σε λούπα με αποτέλεσμα πολύ μεγάλα updates στα βάρη του μοντέλο του νευρωνικού δικτύου. Αυτό οδηγεί σε ένα μη-σταθερό δίκτυο. Η εκδοχή **LSTM**, διορθώνει αυτό το πρόβλημα, χρησιμοποιώντας τρεις πύλες για τη διαδικασία της αποστήθισης.

Θα αναφερθεί ο τρόπος λειτουργίας των 3 πυλών.



1: Η αρχιτεκτονική του LSTM το επιτρέπει να ξεχάσει τις άχρηστες πληροφορίες. Παίρνει το τρέχον input (x(t))και το output του προηγούμενου LSTM unit και χρησιμοποιώντας τη σιγμοειδή συνάρτηση (sigmoid layer) αποφασίζει ποια μέρη από το παλιό output(h(t-1)) θα διαγράψει (βάζοντας output 0). Αυτή η πύλη ονομάζεται forget gate f(t). Η έξοδος αυτής της πύλης είναι f(t)*c(t-1), όπου c(t-1), είναι η μνήμη από το τελευταίο LSTM unit.

2: Εδώ θα αποφασιστεί ποια πληροφορία από το νέο input x(t) θα αποθηκευτεί στο cell state. Έχει πάλι ένα sigmoid layer που αποφασίζει ποια μέρη της νέας πληροφορίας θα ανανεωθούν ή θα αγνοηθούν. Το tanh layer φτιάχνει ένα διάνυσμα με όλες τις πιθανές τιμές από το νέο input. Αυτά τα δύο πολλαπλασιάζονται για να φτιάξουν το νέο cell state. Αυτή η νέα μνήμη προστίθεται στην παλιά, c(t-1), για να δώσει την c(t).

3: Τέλος, αποφασίζεται τι θα βγει στο output. Ένα sigmoid layer, αποφασίζει ποια μέρη του cell state, θα βγουν στο output.

Έπειτα το cell state περνάει από tanh παράγοντας όλες τις πιθανές τιμές και πολλαπλασιάζονται με την έξοδο του sigmoid gate, έτσι ώστε να εμφανιστούν μόνο τα μέρη που αποφασίστηκαν. Το μοντέλο δεν μαθαίνει την απάντηση, από την άμεση εξάρτηση των τελευταίων αποτελεσμάτων, αλλά από μια long term εξάρτηση. Έτσι η διαφορά του LSTM από το RNN, είναι ότι το μοντέλο μαθαίνει ποιες πληροφορίες να αποθηκεύσει στην long term μνήμη και τι να ξεφορτωθεί.

Για το δίκτυο χρησιμοποιούνται από το keras.models το Sequential και από το keras.layers τα Dense, Dropout και LSTM.

Το **δίκτυο** αρχίζει ορίζοντας το μοντέλο σε Sequential, έτσι ώστε να προστεθεί σειριακά η στοίβα από layers. Το πρώτο layer θα είναι το Embedding layer που αναλύθηκε παραπάνω. Έπειτα θα προστεθεί το LSTM layer, και τελευταίο layer (output) θα είναι ένα Dense, το οποίο θα έχει ένα νευρώνα και το χρειαζόμαστε ώστε να κατατάσσει στο τέλος τις προτάσεις σε positive /negative sentiment. Μετά το embedding layer και μετά το LSTM layer, μπορούν να προστεθούν και dropout layers.

LSTM layer: Μία παράμετρος, η οποία είναι ο αριθμός των νευρώνων.

Dropout layer: Ο σκοπός είναι να μειωθεί το overfitting μετατρέποντας ένα μικρό μέρος των δεδομένων σε 0. Μπορεί να πάρει τιμές στο εύρος [0.0,1.0). Αν το dropout layer έχει τιμή σχεδόν

1.0, σημαίνει ότι δεν έχουμε καθόλου dropout, γιατί διατηρείται το 100% των inputs, ενώ αν έχει 0.0 το layer δεν βγάζει κανένα output, αφού διατηρείται το 0% των inputs.

Dense layer: Η activation function είναι σχεδόν πάντα sigmoid, γιατί θέλουμε να παίρνει η έξοδος, τιμές, από το 0 στο 1. Πρέπει να προβλεφθεί η πιθανότητα ως έξοδος, για το αν η πρόταση έχει positive/negative sentiment. Δοκιμές που γίναν με relu, έβγαλαν random αποτελέσματα.

```
model = Sequential()
model.add(embedding_layer)
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Αφού προστεθούν τα layers, το μοντέλο πρέπει να γίνει compile.

Χρησιμοποιούνται 2 διαφορετικά **loss functions**.

MSE: Αυτός είναι ο μαθηματικός τύπος, όπου υπολογίζεται ο μέσος του αθροίσματος για όλα τα δείγματα εκπαίδευσης, της τετραγωνικής διαφοράς μεταξύ της πραγματικής και της προσεγγίσιμης τιμής. Ο τετραγωνισμός χρειάζεται για να μην υπάρχουν αρνητικά πρόσημα και δίνει επίσης περισσότερο βάρος σε μεγαλύτερες διαφορές.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Binary Crossentropy: Από τη στιγμή που γίνεται training σε binary classifier (μπορεί να γίνει και σε multi label), είναι πιο λογική η χρήση αυτής τη συνάρτησης. Εφαρμόζεται όταν μια ερώτηση μπορεί να απαντηθεί με 2 μόνο επιλογές.

$$\mathrm{Loss} = -\frac{1}{\text{output size}}\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

,όπου $\hat{y}_i$ είναι η i-οστή βαθμωτή τιμή του output μοντέλου, και y_i είναι η αντίστοιχη τιμή στόχος, και output size είναι το πλήθος των βαθμωτών τιμών του output μοντέλου. Το loss function αυτό είναι ταιριαστό μόνο με τη sigmoid συνάρτηση στο dense layer και για αυτό το λόγο αν μπει π.χ. relu, τα αποτελέσματα είναι random.

Optimizers:

Stochastic Gradient Descent: Οπουδήποτε χρησιμοποιήθηκε, είχα random αποτελέσματα.Για αυτό το λόγο χρησιμοποιώ adam.

Adam: Στον adam optimizer το learning rate διατηρείται για κάθε παράμετρο και προσαρμόζεται ξεχωριστά κατά τη διάρκεια της εκπαίδευσης. Σε αντίθεση με τον SGD, που διατηρεί ένα μοναδικό learning rate για όλες τις αλλαγές βαρών και αυτό δεν αλλάζει κατά τη διάρκεια της εκπαίδευσης. Αυτή η μέθοδος υπολογίζει ξεχωριστά προσαρμοσμένα learning rates για διαφορετικές παραμέτρους από προσεγγίσεις του πρώτου και του δεύτερου moment των gradients. Το όνομα Adam παράγεται από το adaptive moment estimation.

Προσθέτω στην αρχή, πριν υλοποιηθεί το δίκτυο, numpy.random.seed(7), με σκοπό να είναι οι δοκιμές όσο πιο σταθερές γίνεται (με ίδιες παραμέτρους κυρίως).

Επίσης για το fitting του μοντέλου χωρίζω τα δεδομένα για να κάνω validation και μετά κάνω το μοντέλο evaluate στα training data, και στα test data.

```
batch_size = 1024
num_epochs = 8
X_valid, y_valid = X_train[:batch_size], y_train[:batch_size]
X_train2, y_train2 = X_train[batch_size:], y_train[batch_size:]
history = model.fit(X_train2, y_train2, validation_data=(X_valid, y_valid), batch_size=batch_size, epochs=num_epochs)

scores = model.evaluate(X_train, y_train, verbose=0)
print('Train accuracy:', scores[1])

scores = model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy:', scores[1])
```

Τέλος χρησιμοποιώντας τη βιβλιοθήκη matlplotlib (pyplot) βγάζω γραφήματα για την εξέλιξη του loss και του accuracy στην πάροδο των εποχών.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('No. epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Δοκιμές:

Αρχίζω τις δοκιμές σε δίκτυο χρησιμοποιώντας μόνο LSTM. Αρχικά, όπως αναφέρθηκε και πριν δομώ με τον εξής τρόπο το δίκτυο:

-embedding layer

-dropout

-lstm

-Dropout

-dense(1,sigmoid)

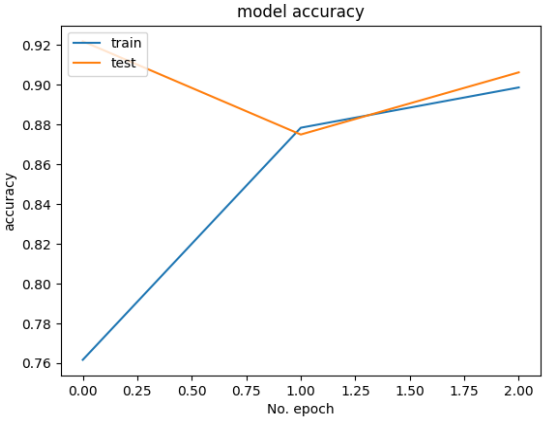
Και στο compile loss='binary_crossentropy', optimizer ='adam'

Και σύμφωνα με τη δομή του embedding layer που εξηγήθηκε παραπάνω, βάζω top_words = 5000, max_words=500.

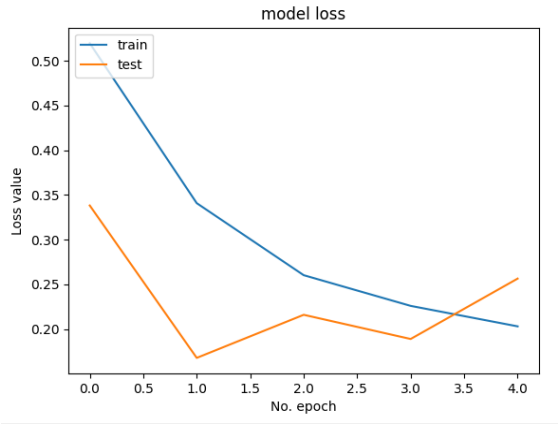
Παρακάτω ο πίνακας αυτών των δοκιμών, όπου τα **dropout** layers είναι στο **0.2**. Στο lstm layer δίνω κυρίως **100 νευρώνες.** Σε αυτό το πλήθος νευρώνων, δοκιμάζοντα batches 64/128/256/512/1024. Γενικά οι χρόνοι των δοκιμών στο lstm είναι σχετικά μεγάλοι.

lstm	Batch	Epochs	Sec	Train acc	Test acc
50	64	3	208	0.9024	0.8577
75	64	3	361	0.9264	0.8718
100	64	3	522	0.9324	0.8780
100	64	5	838	0.9354	0.8688
100	128	5	542	0.9441	0.8756
100	**256**	**3**	**528**	**0.9219**	**0.8792**
100	256	4	651	0.9284	0.8763
100	256	5	809	0.9398	0.8737
100	512	3	399	0.8895	0.8651
100	512	4	536	0.9025	0.8662
100	512	5	677	0.9321	0.8761
100	1024	4	463	0.8574	0.8391
100	1024	5	621	0.8963	0.8593
100	1024	6	737	0.9090	0.8674
120	64	3	619	0.9259	0.8772
150	64	3	721	0.9292	0.8754
200	64	3	1041	0.7636	0.7382

Αρχικά στα 64 batches, με 3 εποχές συγκλίνει σε ένα από τα καλύτερα test acc. , 87.8% σε 522 sec.

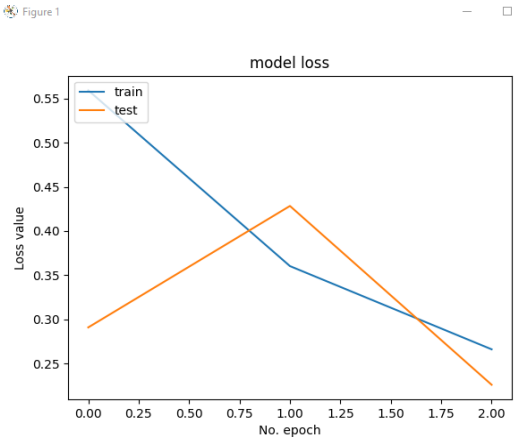


Αυξάνοντας τις εποχές(5) αυξάνεται το loss (σχήμα από κάτω).

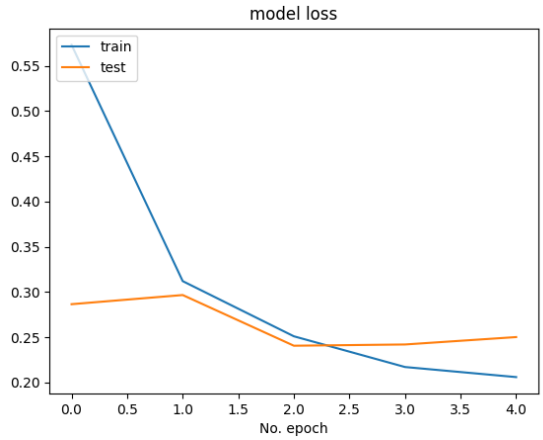


Παραμένοντας στους 100 νευρώνες.

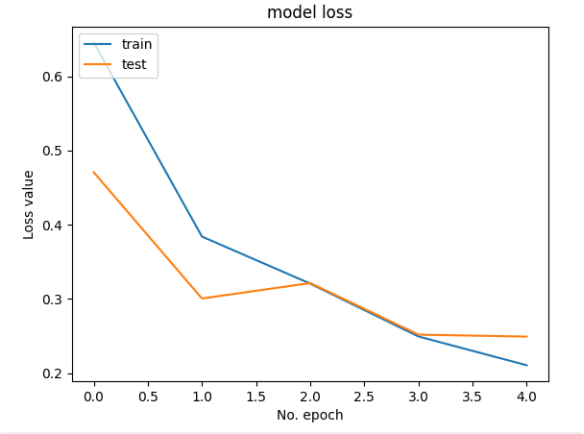
Στα 256 batches, πάλι στις 3 εποχές, βλέπουμε σύγκλιση στο καλύτερο test acc (87.92%) σε 528 sec. Καλύτερο και από πριν. Στο γράφημα του loss, βλέπουμε την εξέλιξη.



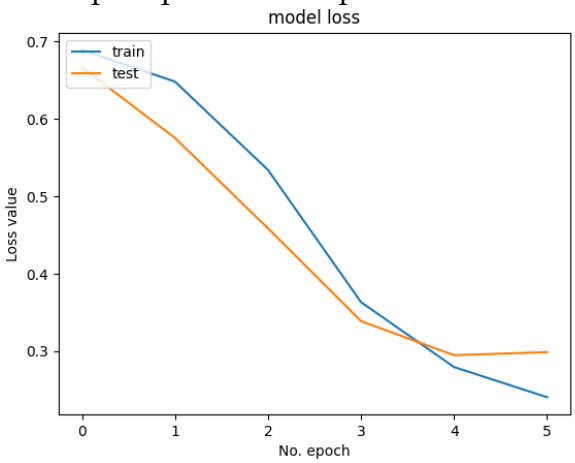
Πηγαίνοντας στις 4 εποχές τα αποτελέσματα είναι κοντά, αλλά στις 5 εποχές αρχίζει το overfitting, καθώς όπως φαίνεται στο γράφημα το testing loss έχει αρχίσει να αυξάνεται καθώς το training loss συνεχίζει να πέφτει.



Στο 512 batch size πάλι στα 3 εποχές τα πάει καλύτερα χωρίς να κάνει overfitting, φτάνοντας 86.51% σε 399 sec. Στις 5 εποχές το test acc, είναι βελτιωμένο αλλά από το γράφημα βλέπουμε ότι έχει αρχίσει ένα ‘ελαφρύ’ overfitting.



Στο 1024 batch size δεν έχουμε κάποια αξιοσημείωτη βελτίωση, και μάλιστα στις 6 εποχές που φτάνει 86.74% χρειάζεται 737 sec, όπου αρχίζει σιγά σιγά πάλι η υπερεκπαίδευση.



Έτσι στους **100** νευρώνες **το καλύτερο result** είναι στα 256 batch size, **87.92**% σε 528 sec.

Για να δοκιμαστεί διαφορετικό πλήθος νευρώνων, κρατάω batch size 64 και 3 epochs, για να κάνω συγκρίσεις. Τα προηγούμενα αποτελέσματα ήταν train acc= 93.24%, test acc=87.80% σε 522 sec

Αρχικά πάλι στους 100 νευρώνες, βγάζω το dropout, βλέπω ότι πέφτει ο χρόνος(480), αλλά πέφτει και το accuracy(86.96%). Το σημαντικό όμως με τη χρήση του dropout είναι να αποφευχθεί η υπερεκπαίδευση.

Δοκιμάζω στους 50/75/120/150/200 νευρώνες. Τα νούμερα φαίνονται στον πίνακα. Όσο αυξάνονται οι νευρώνες, αυξάνεται και ο χρόνος, χωρίς όμως να είναι καλύτερα τα αποτελέσματα, όπως στους 200 νευρώνες, που σε 1041 sec, έχει μόνο 73.82% test acc. Οι πιο πολλοί νευρώνες, χρειάζονται παραπάνω χρόνο να εκπαιδευτούν. Με πολλές εποχές ίσως φτάσουν λίγο καλύτερα αποτελέσματα, αλλά δεν αξίζει από άποψη χρόνου, γιατί είναι πολύ μεγαλύτερος. Στους 75 και 120 νευρώνες, που είναι πολύ κοντά στους 100 τα accuracies είναι πολύ κοντά. Ο χρόνος στους 75 νευρώνες είναι 361 sec(από 522) αλλά το test acc είναι 87.18% από 87.8%.

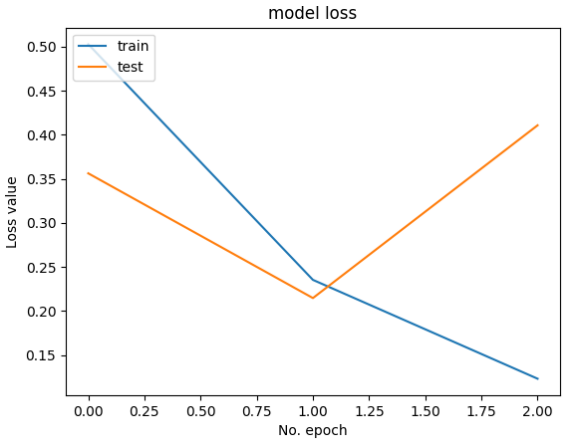
Αυτό που δοκιμάζω πάλι στα 64 batch size/2 epochs, είναι να αλλάξω το πλήθος των διαφορετικών λέξεων (top_words) , και το μέγεθος της κάθε κριτικής (max_words).

Οπότε για top_words=10.000 και max_words=500, έχω test acc 95.12%, train acc = 87.26%, σε 532 sec. Οπότε αν αυξήσω κατά 5.000 τις λέξεις που κρατάω σε παρόμοιο χρόνο, έχω μια μικρή πτώση στο test acc και αύξηση του train acc.

Με top_words=10.000 και max_words=1.000, ο χρόνος αυξάνεται δραματικά και πάει σε 1834 sec, χωρίς κάποια βελτίωση στα αποτελέσματα train 93.52% test 87.22%.

Top_words=15.000 και max_words=500. Εδώ σε 516 sec, 96.21% train acc, 86.88% test acc. Πάλι παρόμοιο χρόνο, πτώση μικρή σε test, και άνοδο σε train..Δεν προσφέρει κάτι αυτή η αλλαγή.

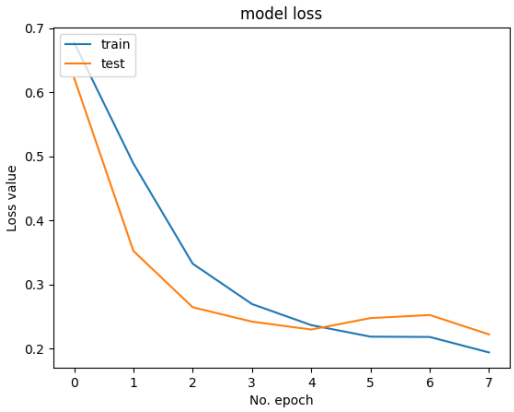
Για να επιβεβαιώσω ότι η αύξηση των top_words δεν προσφέρει βελτίωση, δοκιμάζω 80.000, σχεδόν όλες δηλαδή τις διαφορετικές λέξεις και τα αποτελέσματα είναι παρόμοια 542sec train 98.27%, test 86.73%. Υπάρχει όμως μεγάλη υπερεκπαίδευση, όπως φαίνεται στο σχήμα:



Οπότε παραμένω στις 5.000 διαφορετικές λέξεις, σε όλες τις υπόλοιπες δοκιμές. Μιας και παρατηρήθηκε συχνά overfitting, αυξάνω τα **dropout layers** σε **0.5.** Κάτω οι δοκιμές.

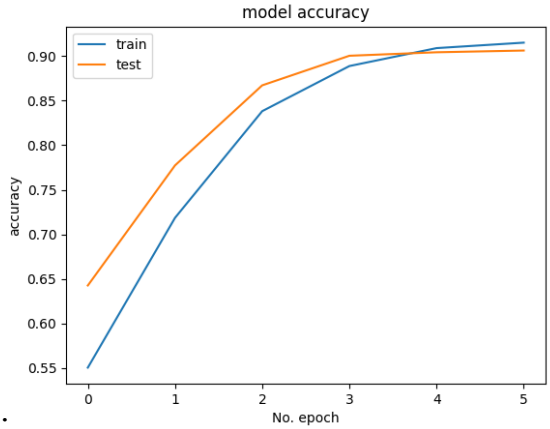
lstm	Batch	Epochs	Sec	Train acc	Test acc
100	32	2	323	0.8756	0.8493
100	32	4	545	0.9201	0.8693
100	32	5	682	0.9267	0.874
100	32	6	820	0.9426	0.8754
100	32	7	951	0.9493	0.8723
100	32	8	1068	0.9021	0.8414
100	64	3	527	0.8812	0.8419
100	64	2	361	0.9053	0.8781
100	512	3	436	0.8766	0.8543
100	512	4	533	0.8739	0.8551
100	512	5	641	0.91	0.8741
100	512	6	793	0.935	0.8832
100	**512**	**7**	**932**	**0.9454**	**0.8838**
100	512	8	1068	0.9405	0.8799

Κάνω δοκιμές μόνο στους 100 νευρώνες, στους οποίους πριν είδα τα καλύτερα αποτελέσματα, και σε 32/64/512 batch size, ώστε να γίνουν οι συγκρίσεις με τον παραπάνω πίνακα. Είναι ευδιάκριτο ότι τώρα χρειάζονται παραπάνω εποχές για να γίνει υπερεκπαίδευση του μοντέλου. Πριν, με 512 batch size, χρειάστηκαν 5 εποχές για να αρχίσει, τώρα συμβαίνει στις 8 εποχές, όπως φαίνεται και στο γράφημα.

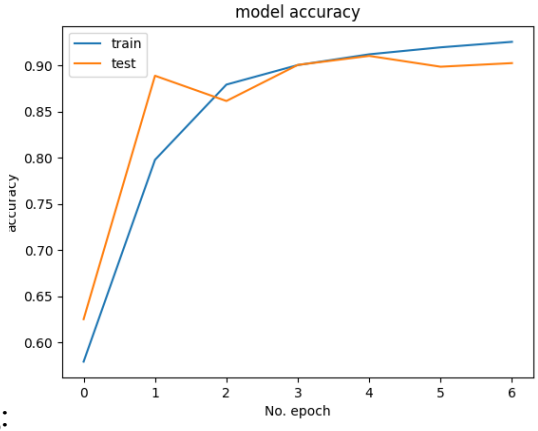


Εδώ όμως βρίσκονται οι μόνες δοκιμές μέχρι τώρα, που φτάνουν το **88%**. Συγκεκριμένα πάλι με batch size 512, στις 6 και στις 7 εποχές, αλλά σε 793 και 932 δεύτερα αντίστοιχα. Απαιτείται πολύ παραπάνω χρόνος.

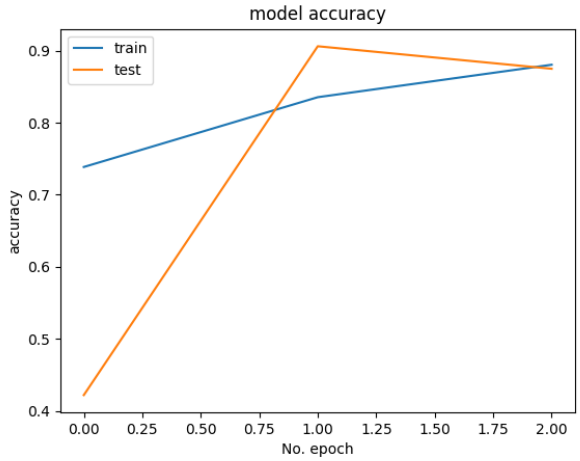
6 εποχές:



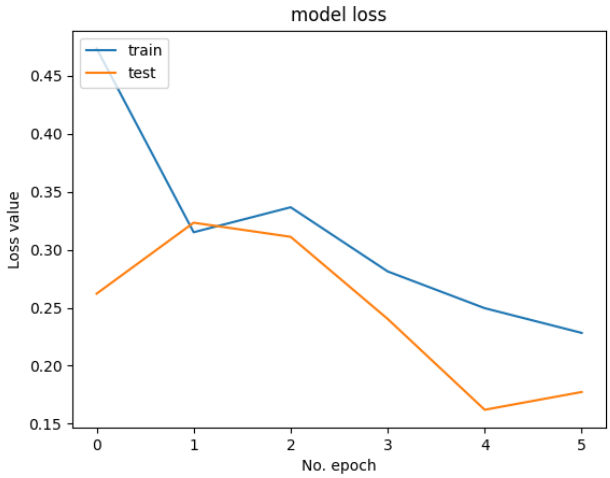
7 εποχές:



Στα 64 batch size, είναι αξιοσημείωτο, ότι στις 3 εποχές υπάρχει αρκετή πτώση στο test acc, απ' ότι στις 2 εποχές .(βλ. γράφημα κάτω)



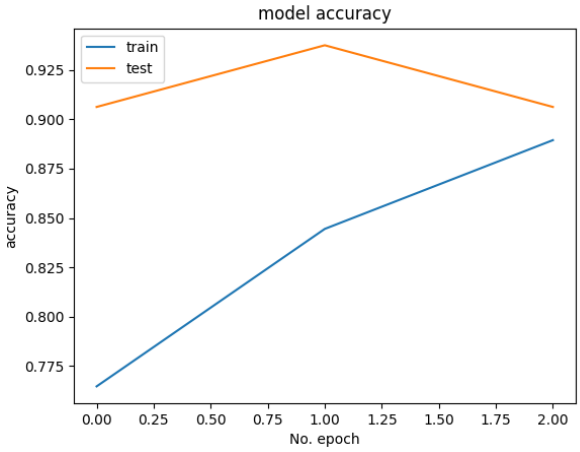
Τέλος στα 32 batch size, συνεχίζει και βελτιώνεται μέχρι τις 6 εποχές.(βλ. γράφημα)



Στη συνέχεια δοκιμάζω να αλλάξω το loss function από binary_crossentropy σε **mse**, κρατώντας σαν optimizer τον adam. Τα dropouts είναι στο 0.2 για να γίνουν καλύτερες συγκρίσεις πάλι με τον αρχικό πίνακα.

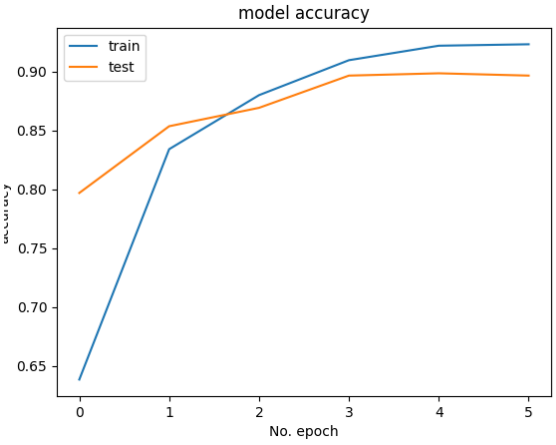
lstm	Batch	Epochs	Sec	Train acc	Test acc
100	64	3	534	0.9013	0.8514
100	64	5	834	0.8859	0.8451
100	128	3	377	0.9027	0.8640
100	256	3	542	0.9211	0.8790
100	256	5	778	0.9407	0.8764
100	512	3	409	0.9124	0.8598
100	512	5	715	0.9327	0.8765
100	512	6	801	0.9257	0.8664

Για batch size 64 οι αποδόσεις είναι χειρότερες. Π.χ. στις 3 εποχές τώρα το test acc είναι 85.14%, ενώ με binary_crossentropy, ήταν 87.8%. Οι χρόνοι είναι παρόμοιοι, και πάλι γίνεται υπερεκπαίδευση όπως φαίνεται στο γράφημα:



Στα 256 batch size τα αποτελέσματα είναι σχεδόν ταυτόσημα, με διαφορά από 0.01% έως 0.09% και σε train και σε test acc.

Τέλος στα 512 batch size, πάλι δεν έχει κάποια αξιοσημείωτη διαφορά. Εκπαιδεύεται σωστά μέχρι και τις 5 εποχές, όπου συγκλίνει με test acc 87.65%, ενώ στον αρχικό μας πίνακα ήταν 87.61% και μάλιστα εδώ σε 30 δεύτερα πιο γρήγορα.



Παρατηρούμε λοιπόν, ότι αυτή η αλλαγή του loss function, δεν μας προσφέρει κάτι σημαντικό.

Convolutional Neural Network (CNN)

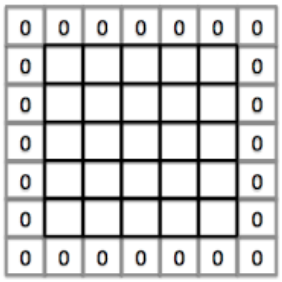
Ένα CNN, λειτουργεί διαφορετικά από ένα κανονικό neural network. Κάθε layer, προσπαθεί να βρει ένα pattern ή χρήσιμες πληροφορίες στα δεδομένα.

Convolution (συνέλιξη): Είναι ένας μαθηματικός συνδυασμός δύο σχέσεων για να παράγουν μια

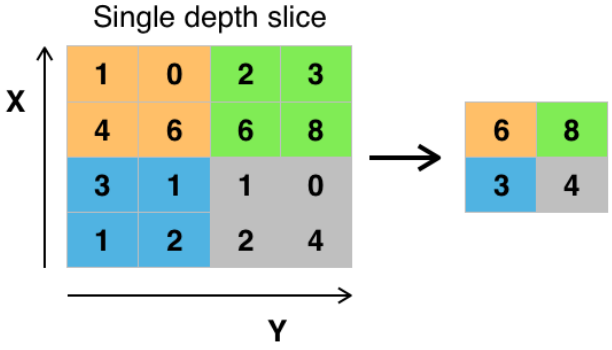
τρίτη σχέση. Ενώνει δύο sets πληροφοριών.

Convolution σε input.: Γίνεται slide της συνέλιξης πάνω στα input data για να εξάγουμε features εφαρμόζοντας ένα φίλτρο/ πυρήνα. Με αυτό το sliding filter, συνδέονται κάποιες παράμετροι, όπως πόσο input να δεχτεί τη φορά και σε τι έκταση θα έπρεπε να αλληλεπικαλύπτεται το input. Όταν εφαρμοστεί τον φίλτρο και γίνουν generate πολλαπλά feature maps, μια activation function δίνεται στο output για να παράγει μια μη-γραμμική σχέση. Συνήθως αυτή η συνάρτηση είναι η ReLu.

Έπειτα γίνεται padding περιτριγυρίζοντας το input, έτσι ώστε το feature map να μην συρρικνωθεί και αν γίνει αυτό, χρήσιμες πληροφορίες πάνω από τα όρια θα αρχίσουν να χάνονται. Επίσης μπορεί να βελτίωση την απόδοση.



Χρειάζεται όμως και κάτι για να μειώσει τους πολλούς υπολογισμούς του CNN και να μην υπερεκπαιδευτούν τα data. Για αυτό χρησιμοποιείται ένα pooling layer, μετά από ένα convolution layer, το οποίο μειώνει την πολυπλοκότητα των διαστάσεων κρατώντας τις σημαντικές πληροφορίες. Θα μπορούσε να είναι ένα max pooling layer, που βρίσκει το maximum του pool και το στέλνει στο επόμενο layer, όπως φαίνεται στη εικόνα από κάτω.



Καμία φορά χρησιμοποιείται Flatten να για να μετατρέψει τα δεδομένα σε μικρότερη διάσταση.

Τελευταίο layer, είναι ένα fully connected layer, όπου κάθε κόμβος ενός layer, συνδέεται με κάθε κόμβο του άλλου layer.

Όσο αφορά τα NLP προβλήματα, έχουμε το embedding layer, όπως αυτό έχει εξηγηθεί. Έτσι αρχίζουμε το μοντέλο πάλι με αυτό, και στη συνέχεια γίνεται slide το filter/kernel πάνω σε αυτά

τα embeddings για να βρεθούν συνελίξεις.

Οι βιβλιοθήκες που χρησιμοποιούνται είναι οι εξής:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
```

Παρακάτω screenshot, από τον κώδικα:

```
model = Sequential()
model.add(embedding_layer)
model.add(Conv1D(filters=64, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Έτσι, όπως αναφέρθηκε, μετά το embedding layer, προστίθεται το convolutional layer και το max-pooling layer.

Το convolutional layer, είναι μονοδιάστατο (**Conv1D**), μιας και έχουμε κείμενο και όχι εικόνες, όπως και στο max-pooling (**MaxPooling1D**). Οι παράμετροι του Conv1D: το kernel size καθορίζει το μέγεθος του sliding window, ενώ τα filters είναι το πόσα διαφορετικά παράθυρα θα έχει, όλα με το ίδιο μέγεθος, δηλαδή το kernel size. Το πόσα διαφορετικά αποτελέσματα ή κανάλια θέλω να πραχθούν. Άρα, άμα έχω filters=32, θα έχω ως αποτέλεσμα 100 διαφορετικές συνελίξεις. Το padding='same', σημαίνει ότι το μέγεθος του output feature-maps είναι το ίδιο με το input feature-map. Τέλος το activation function, παράγει μια μη-γραμμική σχέση.

Το MaxPooling1D παίρνει μία παράμετρο, το pool_size. Αν αυτό έχει μέγεθος 2, μειώνει στη μέση το μέγεθος των features maps, από το convolutional layer.

Μετά προστίθεται το Flatten, το οποίο θα επιπεδοποιήσει (flatten) τα δεδομένα που του έρχονται(1D), ώστε να τα δώσει στο dense layer, έτοιμα.

Υπάρχει μετά ένα ακόμη dense layer (fully connected), όπου θα δοκιμαστεί και LSTM layer στη θέση του και το τελευταίο dense layer, έχει πάντα ένα νευρώνα σαν παράμετρο, επειδή γίνεται binary classification, και πρέπει να καταταχθούν οι προτάσεις σε θετικές/αρνητικές.

Μετά ακολουθεί το fit και το compile, όπως αναλύθηκαν και στο LSTM δίκτυο.

Οι **αρχικές** μου **δοκιμές** γίνοντα με τις εξής τιμές:

```
model.add(Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
```

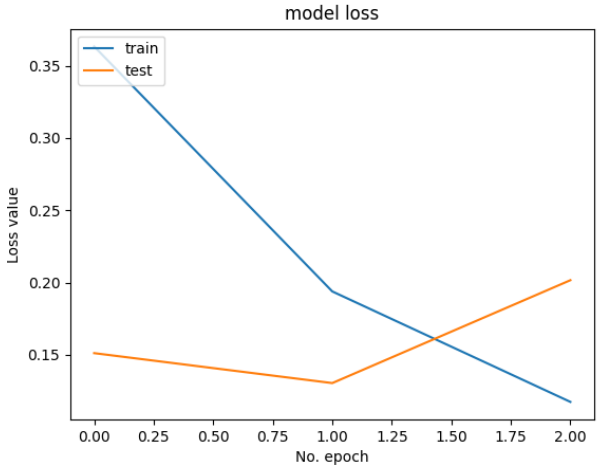
Έπειτα θα γίνουν δοκιμές, με διαφορετικά filters/kernel size/activation functions/ νευρώνες στο dense, όπως και διαφορετικούς optimizers και loss functions.

Καταρχάς, να αναφερθεί, ότι αν μπει σαν optimizer ο sgd, και σαν activation function στο Conv1D sigmoid τα αποτελέσματα είναι random.

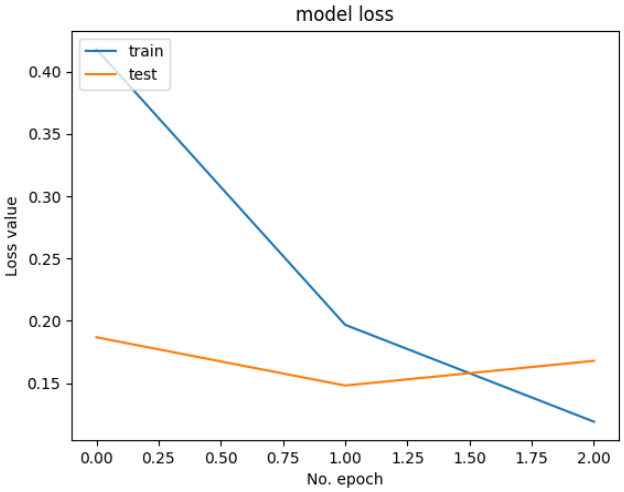
Οπότε παρακάτω είναι ο πίνακας, με τις παραμέτρους του screenshot. Οι χρόνοι στο CNN, είναι πολύ μικροί από 17, μέχρι 52 δεύτερα, πάρα πολύ πιο γρήγορα από το απλό LSTM, δηλαδή.

Batch size	Epochs	Sec	Train acc	Test acc
32	**1**	**22**	**0.9403**	**0.8915**
32	2	35	0.9790	0.8865
32	3	46	0.9816	0.8726
64	1	20	0.9363	0.8886
64	2	28	0.9632	0.8804
64	3	38	0.9873	0.8756
64	5	52	0.9998	0.8723
128	1	17	0.9297	0.8878
128	2	25	0.9349	0.8830
512	1	15	0.7563	0.7397
512	2	20	0.89	0.8677
512	3	25	0.9381	0.8831
512	4	32	0.9574	0.8819
512	5	36	0.94	0.8755
1024	3	26	0.897	0.8705
1024	4	30	0.9058	0.8752
1024	5	36	0.9308	0.8867
1024	6	40	0.9502	0.8796

Ας δούμε τα αποτελέσματα, για τα διαφορετικά batch sizes. Εδώ συναντάμε το καλύτερο αποτέλεσμα σε όλες τις δοκιμές που θα ακολουθήσουν. 32 batch size και μόνο ΜΙΑ εποχή, που σε 22 sec έχει test acc 89.15%. Στις 2 εποχές συνεχίζεται η εκπαίδευση, αλλά με λίγο λιγότερο acc, στα 88.65%, ενώ στις 3 εποχές αρχίζει το overfitting, όπως φαίνεται στο γράφημα:

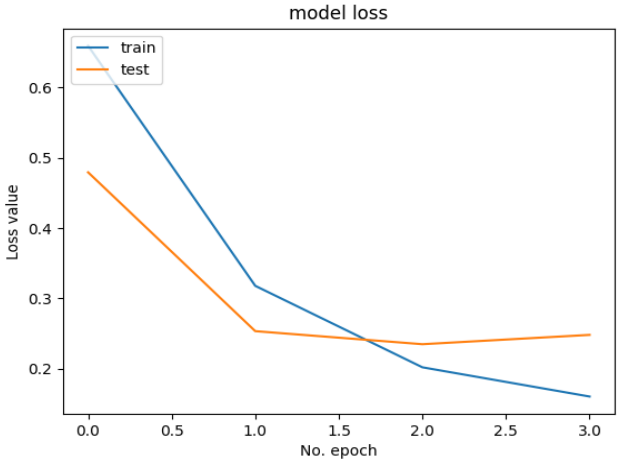


Για 64 batch size, πάλι στη μία εποχή τα πάει καλύτερα, με test acc 88.86% σε 20 δεύτερα και το overfitting να αρχίζει πάλι στις εποχές:

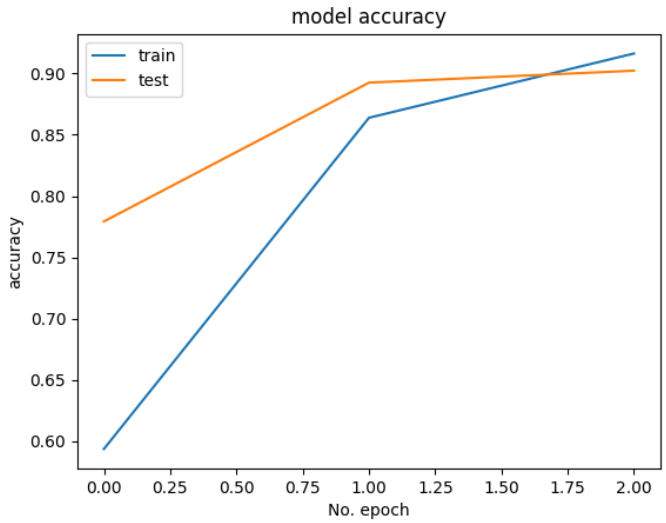


Το μοτίβο αυτό συνεχίζει στο 128 batch size, δηλαδή στη μία εποχή τα πάει καλύτερα με 88.78% σε 25’.

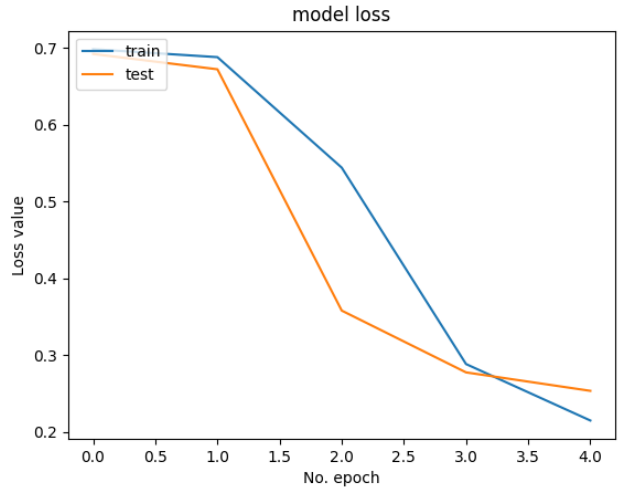
Με batch size 512, υπερεκπαίδευση αρχίζει στις 4 εποχές:



Επίσης καλύτερα τα πάει στις 3 εποχές με 88.31% test acc σε 25 sec:



Αντίστοιχα, για 1024 batch size, στις 5 εποχές overfitting, παρά το ότι εδώ είναι το καλύτερο test acc, για αυτό το batch size (88.67%):

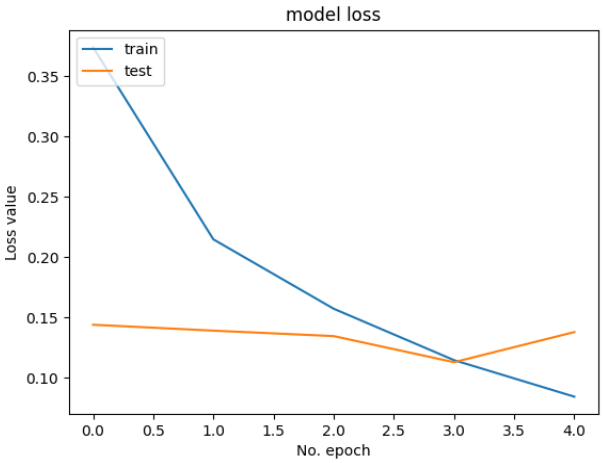


,ενώ συγκλίνει στις 4 εποχές, με test acc 87.52% σε 30 sec.

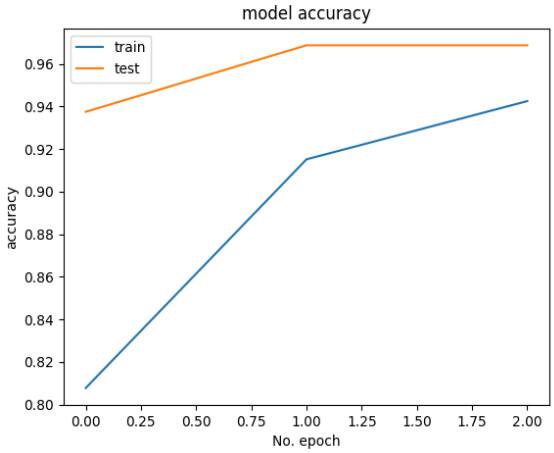
- Η πρώτη αλλαγή που κάνω είναι να προσθέσω, ένα **dropout layer (0.5)** μετά το convolutional layer, για να μειώσω-καθυστερήσω το overfiting.

Batch size	Epochs	Sec	Train acc	Test acc
32	1	24	0.9104	0.8752
32	2	40	0.9435	0.8679
32	3	52	0.9801	0.8835
32	4	67	0.9895	0.8809
32	5	82	0.9913	0.8725
64	3	44	0.9785	0.884
64	5	67	0.9931	0.8793
64	6	79	0.9936	0.8703
512	5	48	0.9291	0.8637

Αυτή τη φορά με 32 batch size, overfitting παρατηρούμε στις 5 εποχές:



,ενώ καλύτερο αποτέλεσμα έχουμε στις 3 εποχές 88.35% σε 52 δεύτερα.

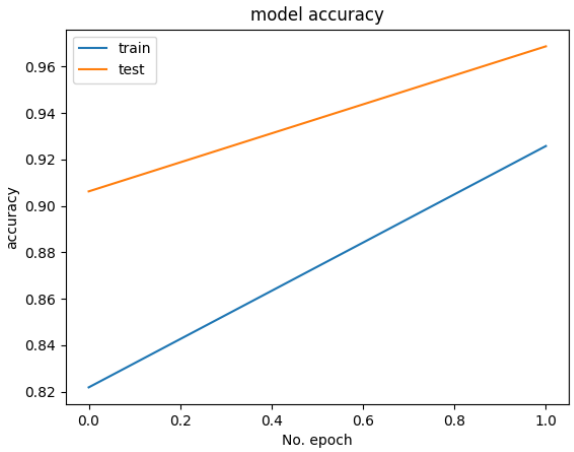


Χωρίς dropout, φτάσαμε σε λίγο καλύτερα αποτελέσματα σε λιγότερο χρόνο.

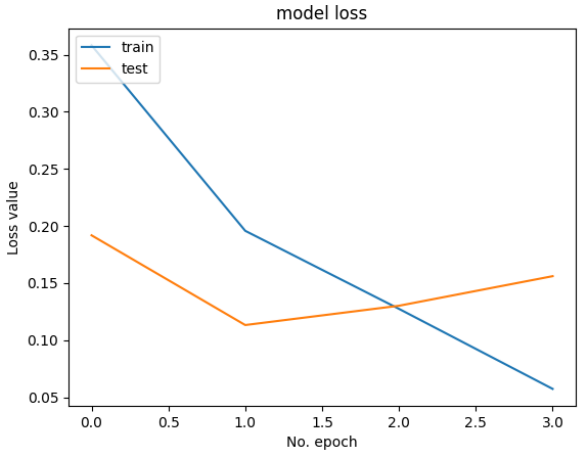
- Γυρνώντας πάλι στα Conv1D (32 filters, kernel size 3, relu),χωρίς dropout, αλλάζω την **συνάρτηση ενεργοποίησης στο dense layer σε sigmoid** από relu. Επίσης κρατάω adam/binary-crossentropy. Εδώ είναι κάποιες δοκιμές:

Batch size	Epochs	Sec	Train acc	Test acc
32	1	22	0.9196	0.8733
32	2	35	0.9702	0.8891
32	3	48	0.9883	0.882
32	4	59	0.997	0.8733
512	3	26	0.9395	0.8841
512	4	31	0.943	0.8816

Παρατηρείται το 2ο μέχρι στιγμής και το 3ο συνολικά καλύτερο στις CNN δοκιμές, πάλι στα 32 batch size, στις 2 εποχές αυτή τη φορά, αντί για τη 1, που ήταν νωρίτερα. Έτσι σε 35 sec, συγκλίνει στο 88.91% test acc:



Παρά το γεγονός ότι στις 3 εποχές συνεχίζει να εκπαιδεύεται, το test acc, έχει μια μικρή πτώση (88,2%), ενώ στις 4 εποχές αρχίζει το overfitting:

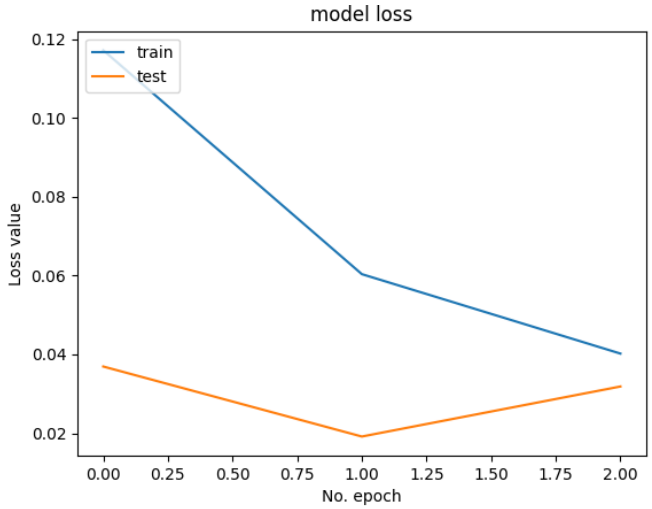


Η διαφορά με τη relu που είχαμε πριν, είναι ότι αρχίζει να υπερεκπαιδεύεται μια εποχή νωρίτερα στα 32 batch size, αλλά για 512, πάλι στις 4 εποχές, όπου τα test acc είναι σχεδόν ίδια, δηλαδή στις 3 εποχές ήταν 0.1% καλύτερο ενώ στις 4 εποχές 0.03% μικρότερο. Δηλαδή είναι ελάχιστες οι μεταβολές.

- Γυρνάω πάλι στις προηγούμενες παραμέτρους με relu στο dense, και αυτό που αλλάζω είναι το loss function, σε **mse**, με κάποιες δοκιμές στον παρακάτω πίνακα:

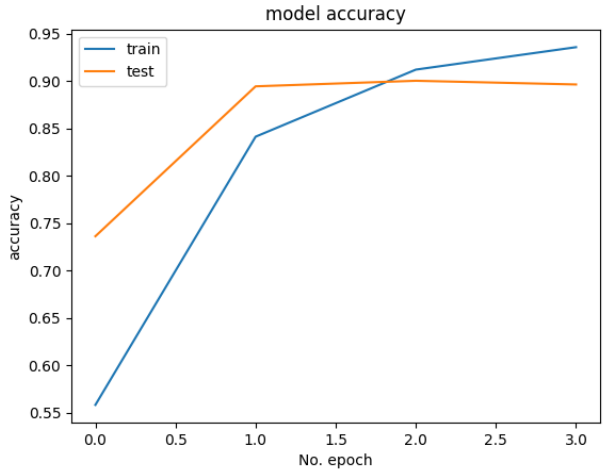
Batch size	Epochs	Sec	Train acc	Test acc
32	1	23	0.9261	0.8808
32	2	35	0.9656	0.8856
32	3	47	0.9759	0.8825
64	3	36	0.9774	0.8809
64	4	45	0.9863	0.8794
512	4	31	0.9536	0.883
1024	5	34	0.9387	0.8857

Εδώ στα 32 batch size, στις 1,2 εποχές υπάρχει μια μικρή πτώση στο test acc σε παρόμοιους χρόνους και πάλι στη 3η εποχή αρχίζει η υπερεκπαίδευση, όπως φαίνεται στο γράφημα:



Στα 64 batch size, πάλι δεν έχει κάποια αξιοσημείωτη διαφορά, με τα αποτελέσματα να είναι πολύ κοντά.

Για 512 batch size, στις 4 εποχές, πήγε από 88.19% στα 88.3%:



Παρόμοια συμπεριφέρεται το μοντέλο με 1024/5 για παραμέτρους.

- Κρατάω το αρχικό μοντέλο πάλι για να γίνουν καλύτερες συγκρίσεις και επειδή εκεί είναι και η καλύτερη δοκιμή από θέμα test acc, μέχρι τώρα.

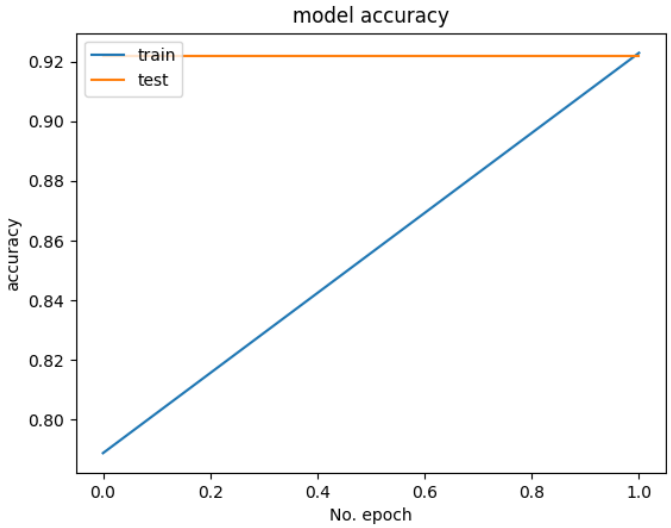
Έτσι αυτή τη φορά, αλλάζω το **kernel size** σε **5**. Αυτό που παρατηρώ, είναι ότι στις εποχές του πίνακα, η συμπεριφορά είναι παρόμοια όσο αφορά το test acc, το χρόνο και το πότε γίνεται υπερεκπαίδευση.

Batch size	Epochs	Sec	Train acc	Test acc
32	1	24	0.9356	0.8891
32	2	35	0.9768	0.8868
64	2	30	0.9687	0.889

64	3	39	0.9751	0.8878
512	4	33	0.9424	0.8829
512	5	39	0.9629	0.8755

Πάλι όμως με 32 batch size, και 1 εποχή έχουμε 88.91%, το 3ο καλύτερο αποτέλεσμα σε 24'.

64 batch size και 2 εποχές 88.9% :



Αυτό που παρατηρείται και εδώ, είναι ότι ενώ το μοντέλο συνεχίζει να εκπαιδεύεται και συγκλίνει στις 3 εποχές, δεν είναι εκεί το καλύτερο result.

Στα 512 batch size, πάλι παρόμοια τα πράγματα.

Επόμενη αλλαγή που κάνω, είναι να αυξήσω το max pooling σε 4, από 2:

Batch size	Epochs	Sec	Train acc	Test acc
32	1	20	0.9393	0.8899
32	2	29	0.9642	0.8822
64	2	23	0.9651	0.8881
64	3	31	0.9857	0.8841
512	3	23	0.9332	0.887
512	4	27	0.9479	0.8862
512	5	32	0.9797	0.8814

Γενικά πάλι κυμαίνονται σε ίδια πλαίσια τα νούμερα. Εδώ είναι το 2ο καλύτερο test acc, με 88.99% σε 20 δεύτερα, με 32/1. Στις ίδιες παραμέτρους είχαμε το καλύτερο αποτέλεσμα όλου του CNN, και όλων των δοκιμών σε αυτή την εργασία και παρά που τα νούμερα είναι παρόμοια, κρατάω σαν καλύτερο το αρχικό μοντέλο, λόγω του ότι έχει ακόμη την καλύτερη δοκιμή.

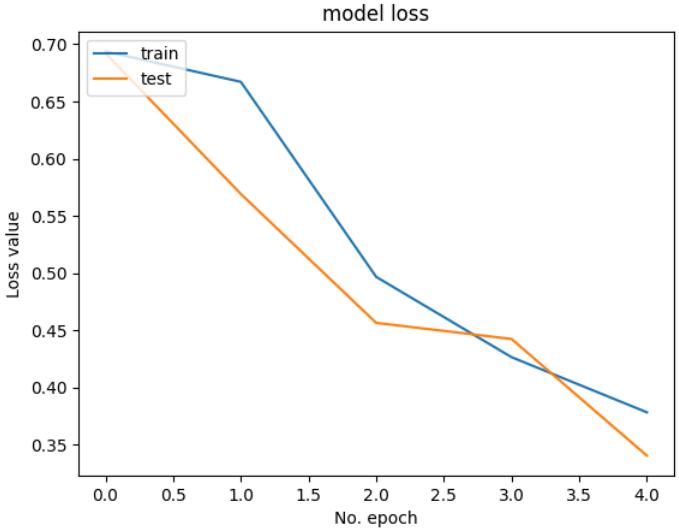
Στα 64 batch size, εδώ όμως ,είμαστε καλύτερα, που στις 2 εποχές τώρα είναι στο 88.81% ενώ στο αρχικό μοντέλο ήταν 88.04%.

Μικρές βελτιώσεις του 0.3% φαίνεται και για batch size 512.

- Άλλη μία παραλλαγή που κάνω είναι να αλλάξω το **πλήθος των νευρώνων στο dense layer.** Αρχικά τους μειώνω σε **100** από 250:

Batch size	Epochs	Sec	Train acc	Test acc
32	1	19	0.9345	0.8877
32	2	28	0.9509	0.8855
64	2	25	0.9368	0.8782
64	3	31	0.9679	0.882
64	4	40	0.9798	0.8778
512	3	24	0.9361	0.8845
512	5	33	0.9344	0.8746

Εδώ παρατηρώ μια καθυστέρηση στις εποχές για overfitting, συγκεκριμένα στο batch 512 στις 5 εποχές το loss κυμαίνεται έτσι:



Ενώ πριν είχε αρχίσει καλά το overfitting σε αυτές τις εποχές. Το ίδιο συμβαίνει και για batch 64, όπου τώρα αρχίζει στις 4 εποχές, ενώ πριν στις 3.

Τα αποτελέσματα είναι περίπου όπως και στις τελευταίες παραλλαγές, δηλαδή για 32 batch size, είναι λίγο κατώτερα, ενώ για 64, λίγο καλύτερα.

Οι χρόνοι είναι λίγα δευτερόλεπτα μικρότεροι, κάτι που ήτα αναμενόμενο.

- Αυτή τη φορά θα αυξήσω τους **νευρώνες σε 500**:

Batch size	Epochs	Sec	Train acc	Test acc
32	1	29	0.9336	0.8841
32	2	46	0.9762	0.8845
32	3	64	0.9907	0.8799

64	2	37	0.9729	0.884
64	3	49	0.9923	0.8732
128	2	30	0.9396	0.8856
128	3	38	0.9842	0.8825
512	3	29	0.9447	0.884
512	4	34	0.9341	0.8802

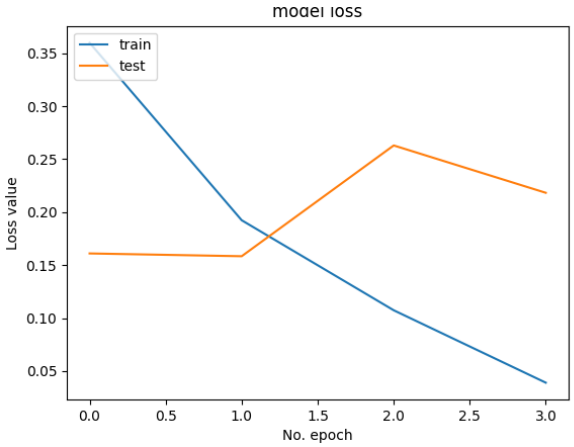
Εν συντομία και εδώ φαίνεται να αντέχει μια εποχή παραπάνω πριν υπερεκπαιδευτεί το μοντέλο και οι χρόνοι είναι ελάχιστα μεγαλύτεροι, που είναι λογικό. Τα αποτελέσματα είναι λίγο κατώτερα όμως, αλλά όχι σημαντικά.

- Πηγαίνοντας πάλι στο αρχικό μοντέλο, **αυξάνω τα filters**, από 32 σε **64**.

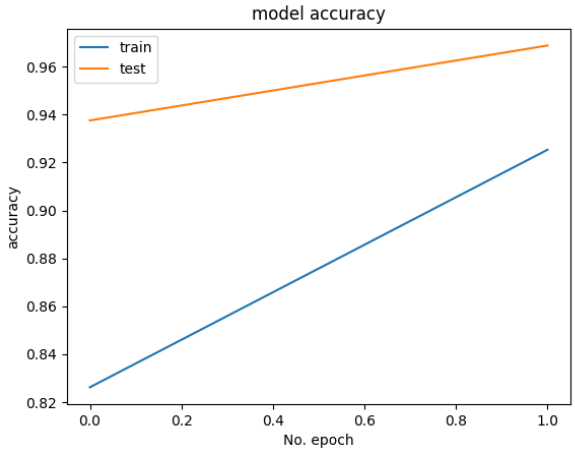
Batch size	Epochs	Sec	Train acc	Test acc
32	1	32	0.9387	0.8879
32	2	51	0.9649	0.8873
32	3	72	0.9775	0.8838
32	4	89	0.9962	0.8754
64	3	52	0.9815	0.8812
64	4	66	0.9795	0.8751
512	3	39	0.9295	0.8808
512	4	48	0.9336	0.8812
1024	5	53	0.9228	0.8817
1024	6	60	0.9429	0.8804
1024	7	69	0.9473	0.8756

Αυτή τη φορά οι χρόνοι αυξάνονται, όχι πολύ, αλλά αισθητά.

Για 32 batch, υπάρχει μια μεγάλη άνοδος στο loss, και μετά ξανά πτώση:

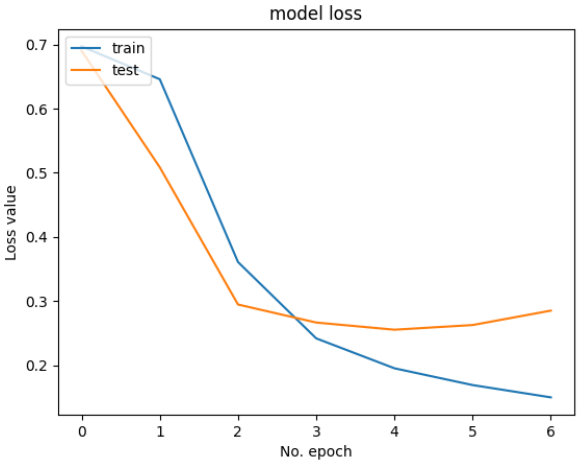


Φτάνει στο 88.73% στις 2 εποχές, που είναι καλό αλλά όχι στα καλύτερα:



Στα 64 batch, φαίνεται μικρή βελτίωση, αλλά για 512 και 1024, φαίνεται μια πτώση σε σύγκριση με τον πίνακα του αρχικού μοντέλου.

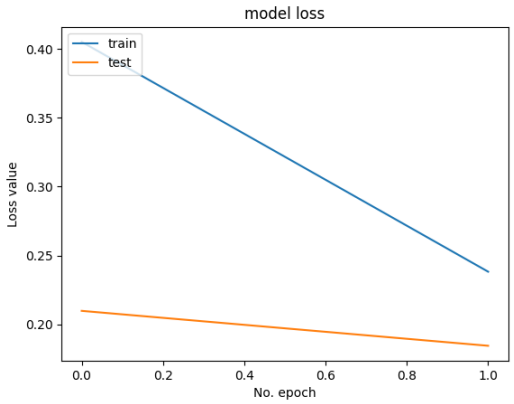
Γράφημα 1024/7 όπου υπάρχει overfitting:



- Τέλος, αυτό που δοκιμάζω είναι να αντικαταστήσω το dense layer, με ένα **lstm layer των 100 νευρώνων**:

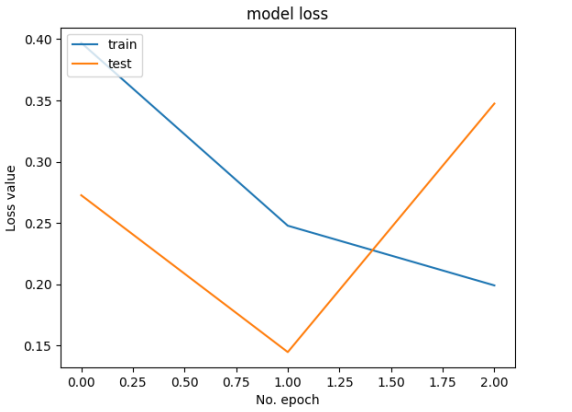
Batch size	Epochs	Sec	Train acc	Test acc
32	1	114	0.9028	0.8684
32	2	185	0.9323	0.8769
32	3	243	0.9125	0.8503
64	2	153	0.9428	0.8864
64	3	208	0.9501	0.882
512	3	189	0.9234	0.8801
512	4	248	0.9273	0.8823
512	5	294	0.9346	0.8815
512	6	343	0.9596	0.8768
1024	6	416	0.9427	0.8786
1024	7	482	0.9407	0.8773
1024	8	598	0.9478	0.8740

Οι χρόνοι είναι πολύ μεγαλύτεροι και το μέγιστο test acc, είναι 88.64% (64/2):

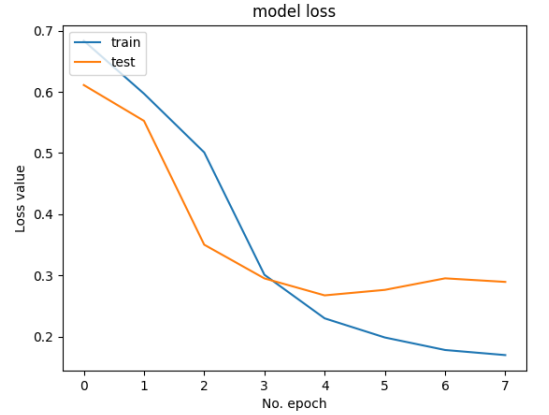


Δεν υπάρχει δηλαδή κανένα όφελος σε αυτή τη δοκιμή, γιατί κάνεις **πολλά λεπτά παραπάνω** ,χωρίς κανένα όφελος από οπτική accuracy.

32/3 πολύ μεγάλο overfitting:



Με 1024 batch, αρχίζει στις 8 εποχές η υπερεκπαίδευση:



Συμπέρασμα:

Στο **LSTM** δίκτυο το μέγιστο test acc φτάνεται μαζί με dropout 0.5, με 100 νευρώνες, 512 batch size και 7 εποχές. Αλλά απαιτεί 932 δευτερόλεπτα…

Όσο αφορά το **CNN** δίκτυο, μετά από αλλαγές παραμέτρων σε filters/ kernel size/ activation function του Conv1D και του dense layer/ με FCN και LSTM layer μετά τη συνέλιξη/ διαφορετικά loss functions/optimizers, **το καλύτερο result** και μάλιστα από την αρχή με τις πιο «απλές» παραμέτρους, βρέθηκε με αυτές τις παραμέτρους:

```
model = Sequential()
model.add(embedding_layer)
model.add(Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Και βάζοντας 32 batch size και μία μόνο εποχή. Έτσι σαν αποτέλεσμα έχουμε 89.15% test acc, σε 22 μόνο δευτερόλεπτα. Αυτή είναι η καλύτερη δοκιμή.

Υ.Γ. : Στο αρχείο κώδικα που σας στέλνω υπάρχουν όλες οι απαραίτητες βιβλιοθήκες. Αν τρέξει το αρχείο θα τρέξει το πιο γρήγορο νευρωνικό που αναφέρθηκες πιο πάνω. Σε σχόλια υπάρχει και το LSTM και στα screenshots που έχω στην αναφορά φαίνεται ακριβώς πως πρέπει να γραφτεί στον κώδικα. Και σε αυτό γίνονται όλες οι αλλαγές παραμέτρων για τις υπόλοιπες δοκιμές. Σε σχόλια υπάρχουν και κάποιες γραμμές κώδικα, που αναλύουν τα δεδομένα από το dataset, τα οποία εξήγησα όταν περιέγραφα το dataset.