KNN algorithm on SVHN Dataset

Καρυοφυλλιά Πριάκου

December 16, 2022

1 Updates

Για να ελαχιστοποιήσω προσωρινά το υπολογιστικό κόστος και να διευκολυνθώ στους πειραματισμούς μου, επέλεξα απ' το dataset τα πρώτα 1000 δεδομένα. Παρατηρώ οτι το accuracy μου μειώνεται περίπου κατά $20,65\%$.
accuracy = 0.24185617701290718

```
test_imagesC = test_images[0:1000,:]
train_imagesC = train_images[0:1000, :]  #Gia dataset 1000 stoixeiwn
test_labelsC = test_labels[0:1000]
train_labelsC = train_labels[0:1000]


#testRIres = np.reshape(testRI,(18315,3072))
knn = OneVsRestClassifier(KNeighborsClassifier(n_neighbors=3, weights='distance'))
knn.fit(train_imagesC, train_labelsC)
acc = knn.score(train_imagesC,train_labelsC)

print("pixel accuracy: {:.2f}%".format(acc * 100))
```

```
pixel accuracy: 100.00%
```

```
#todo mean squared error
pred = knn.predict(test_images)
accuracy = accuracy_score(test_labels, pred)
print(accuracy)
```

```
0.24185617701290718
```

2 NN with Keras

- **RGB σε Gray**

How do I convert RGB to grayscale?

The Average method takes the average value of R, G, and B as the grayscale value.

- Grayscale = (R + G + B) / 3.
- Grayscale = R / 3 + G / 3 + B / 3.
- Grayscale = 0.299R + 0.587G + 0.114B.
- Y = 0.299R + 0.587G + 0.114B U'= (B-Y)*0.565 V'= (R-Y)*0.713.
- Grayscale = Y.

```
def rgb2gray(rgb):

    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray
```

Αφού έκανα την εικόνα απο rgb σε gray μένει να μετατρέψω τους πίνακες απο $[x, y, rgb, samples]$ σε $[samples, x, y]$ για να εκπαιδευτούν στο μοντέλο μου. $x, y = 32pixel$

```
def images3d(data):
    images = [] #create empty list
    for i in range(0, data.shape[3]): #73257 gia train images, 26032 gia test
        images.append(rgb2gray(data[:, :, :, i])) #to append einai gia lists
    return np.asarray(images) #metatroph list se numpy array
```

Μορφή πινάκων πριν και μετά τον μετασχηματισμό τους:

```
(32, 32, 3, 73257) Train images
(32, 32, 3, 26032) Test images
(73257,) train labels
(26032,) test labels
----after format------
(73257, 32, 32) Train images
(26032, 32, 32) Test images
(73257,) Train labels
(26032,) Test labels
```

- **Κανονικοποίηση μεταβλητών**. $2^8 = 256$ πιθανότητες γαι κάθε χρώμα.

```
train_imagesNN, test_imagesNN = train_images / 255, test_images / 255
train_labelsNN, test_labelsNN = train_labels, test_labels
```

- **Δημιουργία μοντέλου με keras**. Ξεκινάω με ένα input layer, 3 hid-

den layers(100 outputs) και ενα output(11 outputs όσες και οι κλασεις μου) layer.Τα keras.layers.dense εκτελούν την πράξη $output = activation(dot(input, kernel) + bias)$ όπου activation χρησιμοποιώ την συνάρτηση RELU και kernel είναι ο πίνακας των βαρών. Εκτελώ για **10 εποχές.**

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 flatten_4 (Flatten)          (None, 1024)              0

 dense_20 (Dense)             (None, 300)               307500

 dense_21 (Dense)             (None, 100)               30100

 dense_22 (Dense)             (None, 100)               10100

 dense_23 (Dense)             (None, 11)                1111

=================================================================
Total params: 348,811
Trainable params: 348,811
Non-trainable params: 0
_________________________________________________________________
```

accuracy = 0.1160, validation accuracy = 0.1107

Αλλάζω την συνάρτηση του τελευταίου επιπέδου εξόδου απο RELU σε Softmax. Η συνάρτηση softmax

$$f(x) = \frac{e^x}{\sum e^x} \tag{1}$$

χρησιμοποιείται κατα βάση στη τελευταίο layer γιατί παράγει ως έξοδο μία **κανονικοποιημένη κατανομή πιθανοτήτων.** Δηλαδή κανονικοποιεί όλες τις εισόδους στο διάστημα [0,1] και επιπλέον το άθροισμα των τιμών του διανύσματος ισούται με 1. Όπως βλέπουμε δίνει πολύ καλύτερα αποτελέσματα σε σχέση με την RELU που παράγει τυχαίες πραγματικές τιμές για κάθε κλάση.

accuracy= 0.7538, validation accuracy 0.6945

Προσθέτω samples για forwardpass. Ο χρόνος εκτέλεσης όπως είναι αναμενόμενο διπλασιάζεται.

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 flatten_6 (Flatten)          (None, 1024)              0

 dense_28 (Dense)             (None, 500)               512500

 dense_29 (Dense)             (None, 200)               100200

 dense_30 (Dense)             (None, 200)               40200

 dense_31 (Dense)             (None, 11)                2211

=================================================================
Total params: 655,111
Trainable params: 655,111
Non-trainable params: 0
_________________________________________________________________
```

accuracy= 0.7654, validation accuracy 0.7275

Προσθέτω 1 hidden layer

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 flatten_7 (Flatten)          (None, 1024)              0

 dense_32 (Dense)             (None, 500)               512500

 dense_33 (Dense)             (None, 200)               100200

 dense_34 (Dense)             (None, 200)               40200

 dense_35 (Dense)             (None, 200)               40200

 dense_36 (Dense)             (None, 11)                2211

=================================================================
Total params: 695,311
Trainable params: 695,311
Non-trainable params: 0
_________________________________________________________________
```

accuracy= 0.7762, validation accuracy= 0.6892

Προσθέτω κιαλλο hidden layer.

Layer (type)	Output Shape	Param #
flatten_8 (Flatten)	(None, 1024)	0
dense_37 (Dense)	(None, 500)	512500
dense_38 (Dense)	(None, 200)	100200
dense_39 (Dense)	(None, 200)	40200
dense_40 (Dense)	(None, 200)	40200
dense_41 (Dense)	(None, 200)	40200
dense_42 (Dense)	(None, 11)	2211

Total params: 735,511
Trainable params: 735,511
Non-trainable params: 0

accuracy= 0.7788, validation accuracy= 0.6570
Παρατηρώ συγκριτικά μικρή αύξηση στο accuracy και συγκριτικά μεγάλη μείωση του value accuracy, οπότε παραμένω στα 4 hidden layers. Αυτό ωφείλεται στο οτι τα σφάλματα στην φάση του backpropagation που επιστρέφουν γίνονται πολύ μικρά, και η εκπαίδευση του μοντέλου γίνεται λιγότερο αποτελεσματική αυξάνοντας άσκοπα την πολυπλοκότητα.

Οι δοκιμές μου γινόντουσαν με τον gradient descent optimizer:
$\theta = \theta - \alpha \cdot \bigtriangledown J(\theta)$
Ο gradient descent διατηρεί ενα learning rate(α) για κάθε ανανέωση των βαρών και δεν αλλάζει κατα την διάρκεια της εκπαίδευσης. Ο **Adams optimizer** προσαρμώζει το learning rate αποθηκεύοντας τον εκθετικά μειωμένο μέσο των προηγούμενων τετραγωνικών παραγώγων v_t και τον εκθετικά μειωμένο μέσο των προηγούμενων παραγώγων m_t:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t \tag{4}$$

οπου b_1 expotentioal decay rate for first moment estimates(eg 0.9) ,b_2 exponential decay rate for the second-moment estimates (eg 0.999) και ϵ ένας πολύ μικρός αριθμός για να αποφευχθεί η διαίρεση με το 0.

```
model.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam', metrics = ['accuracy'])
```

accuracy = 0.7996 , validation accuracy = 0.7714

Batch size. To default batch size του keras είναι 32. To batch size είναι ο αριθμός των samples που θα χρησιμοποιηθούν σε ένα forward/backward pass πριν το update των παραμέτρων του μοντέλου. Αυξάνω το batch size απο 32 σε 50:

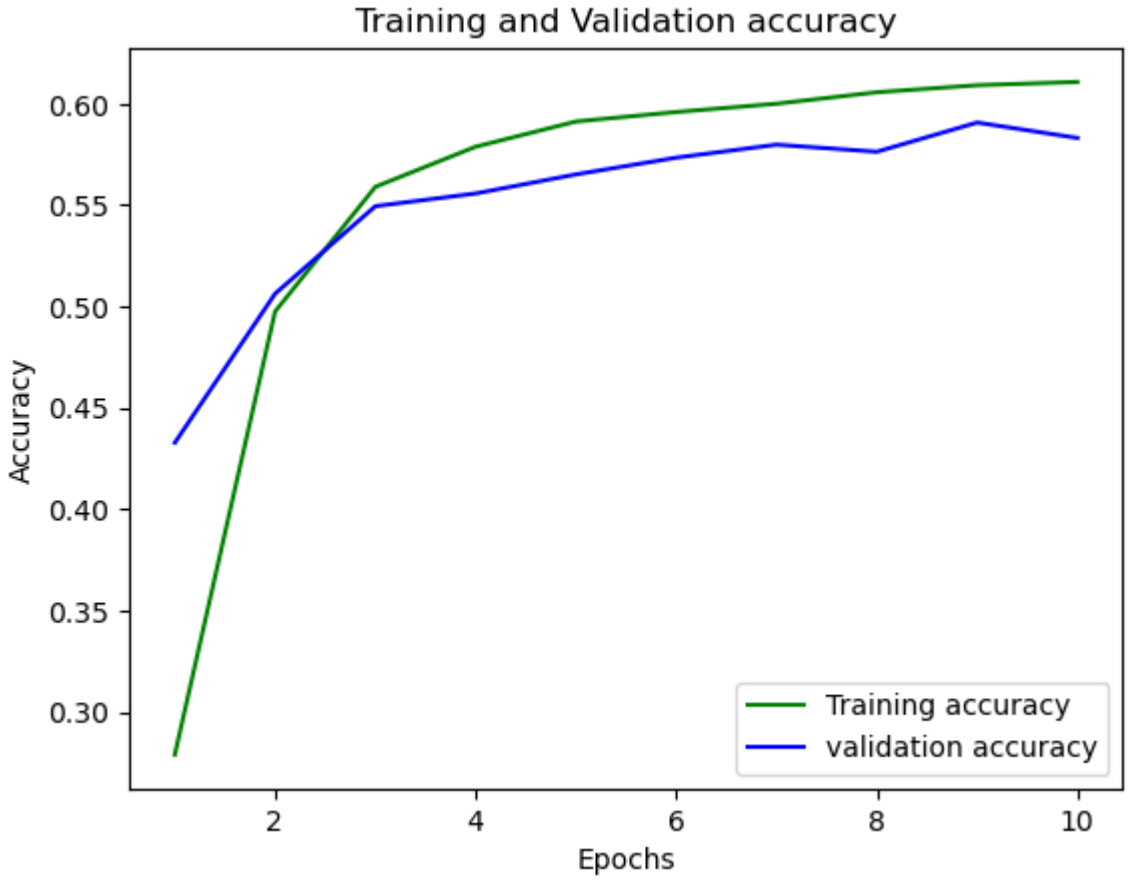
```
history = model.fit(train_imagesNN,train_labelsNN,epochs = 10, validation_data = (test_imagesNN, test_labelsNN), batch_size=50)
```

accuracy = 0.7765, validation accuracy= 0.7466
To accuracy μειώθηκε μου σημαίνει οτι το νευρωνικό δίκτυο αρχίζει να υπερεκπαιδεύεται (poor generalization).

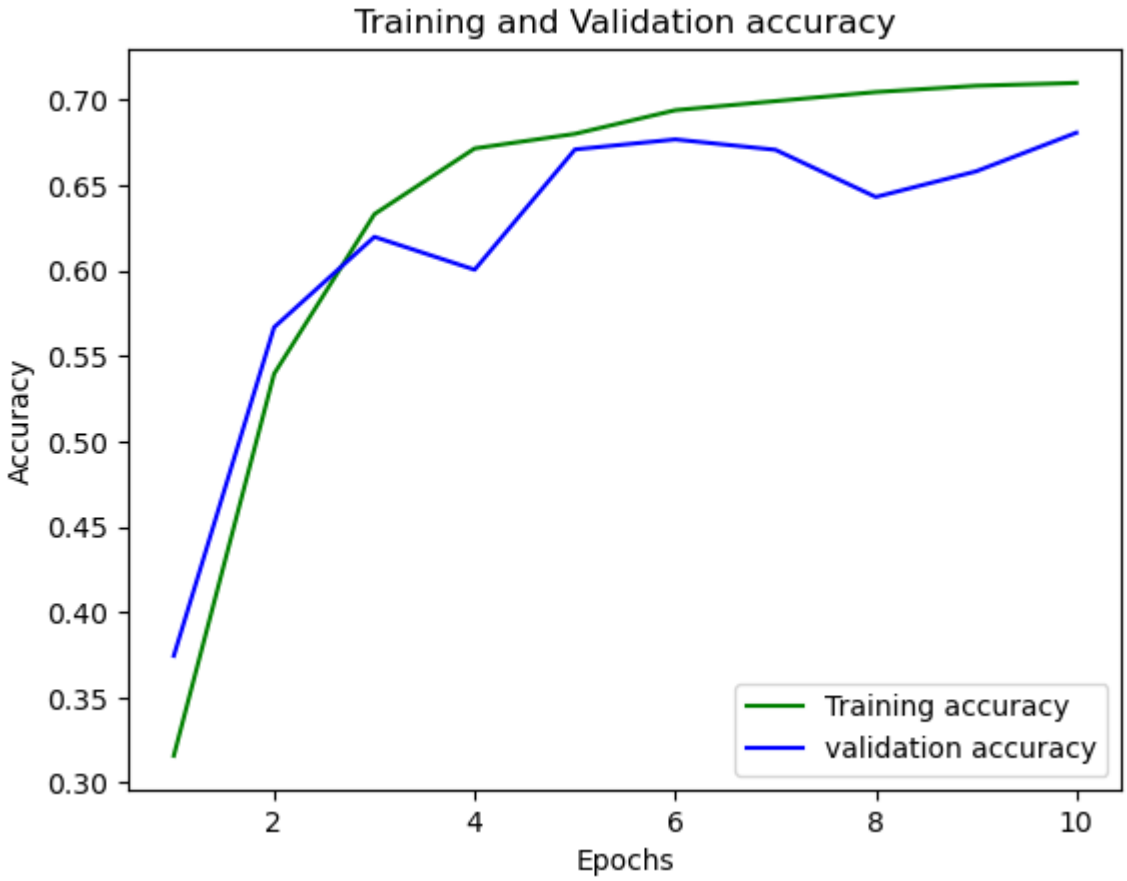
batch size=25 .

accuracy=0.6108, validation accuracy= 0.05832

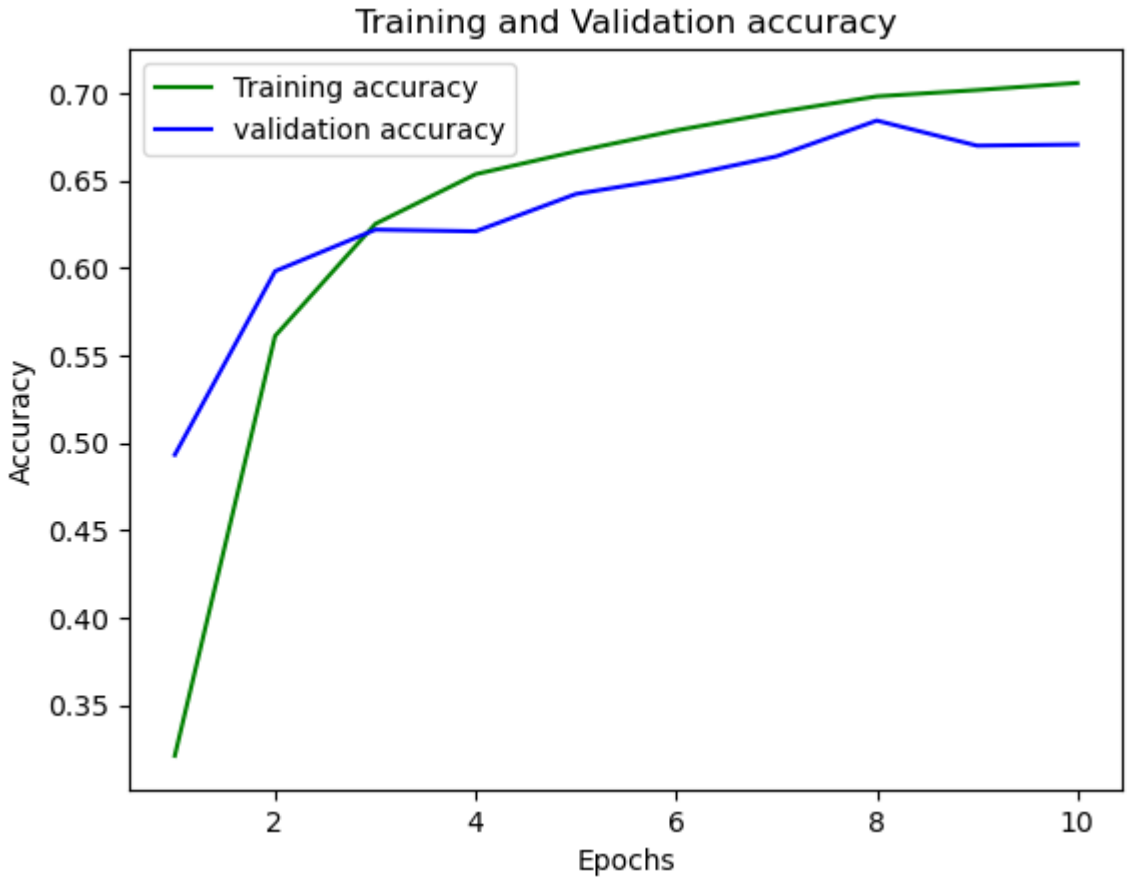
batch size= 50



accuracy=0.7609, validation accuracy= 0.7268
Batch size= 35

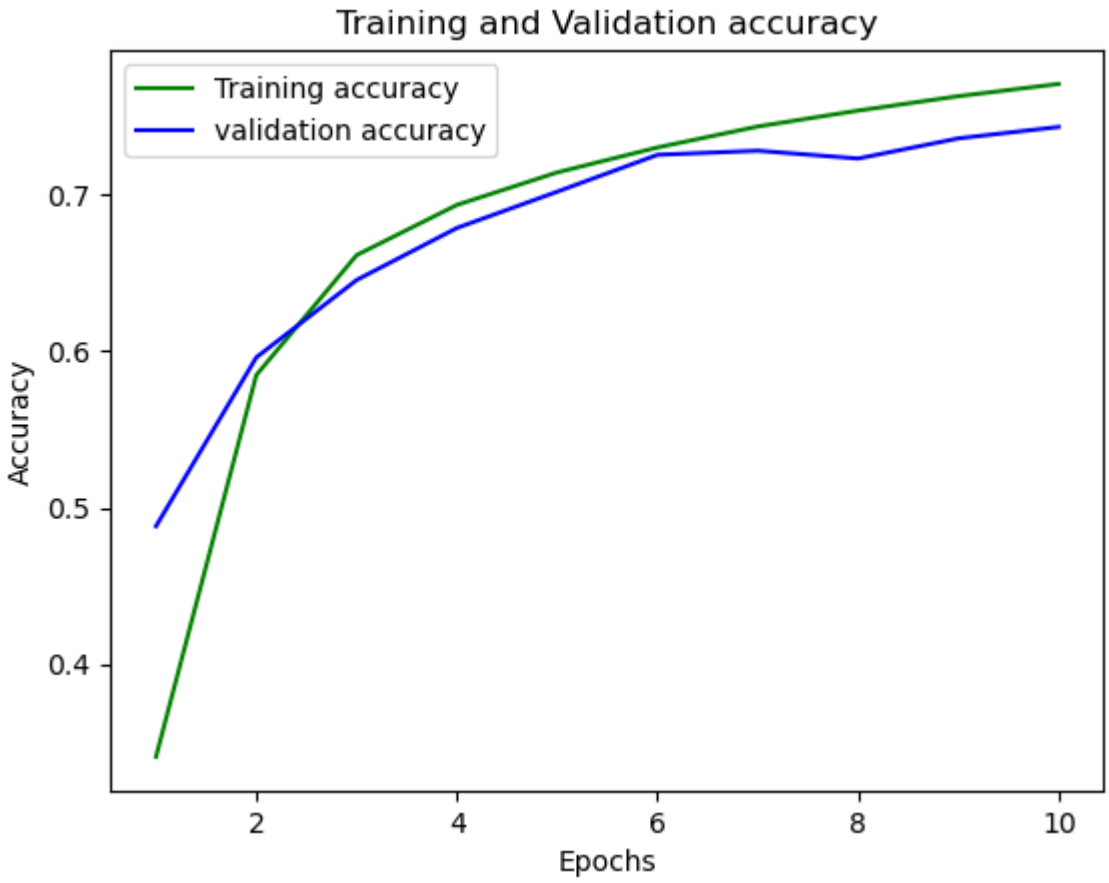


accuracy=0.7099, validation accuracy= 0.6808
Batch size= 37

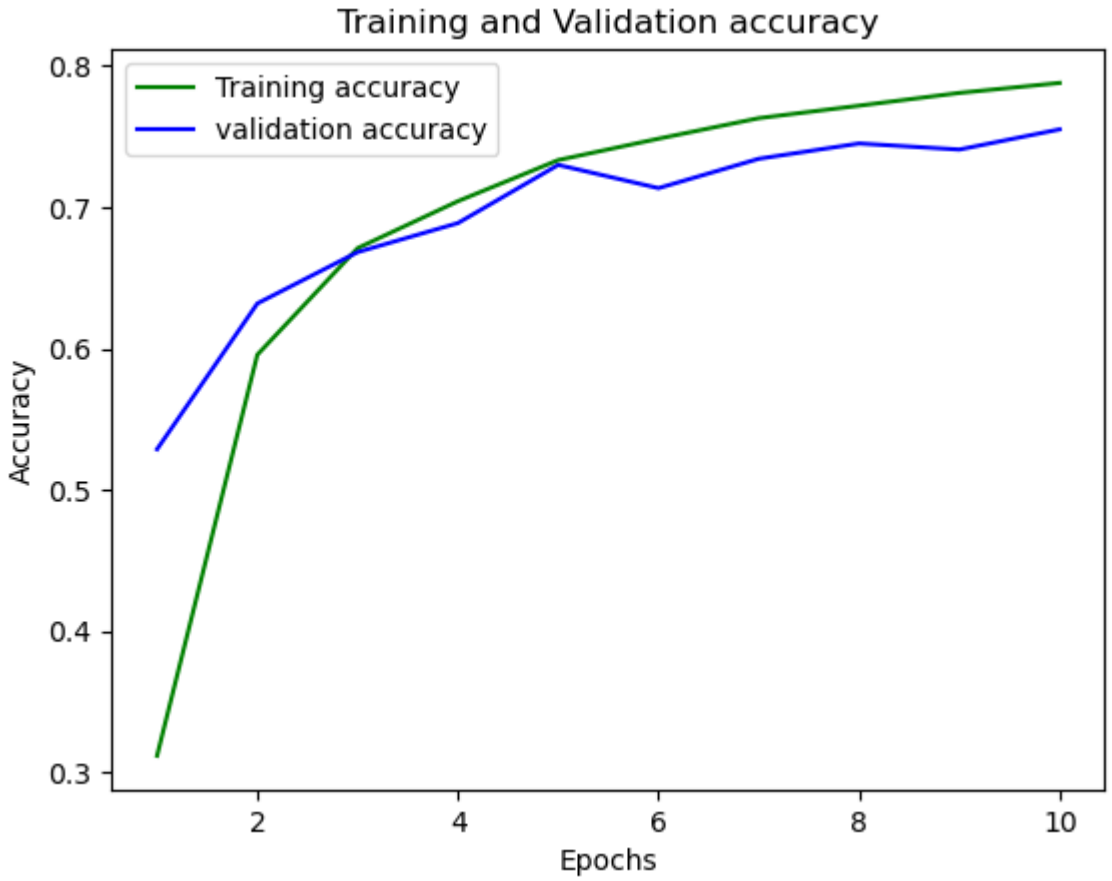


accuracy=0.7057, validation accuracy= 0.6705
Batch size= 32

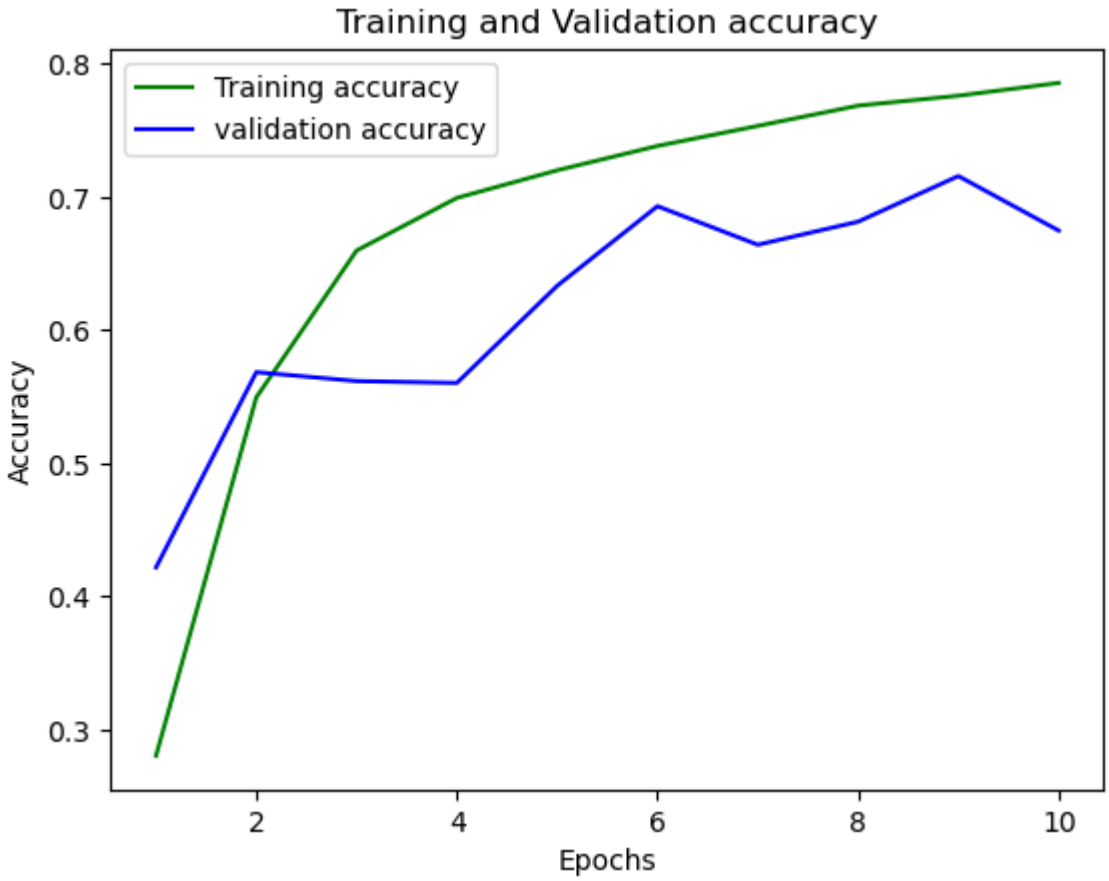
accuracy=0.6795, validation accuracy= 0.6404
Batch size= 60



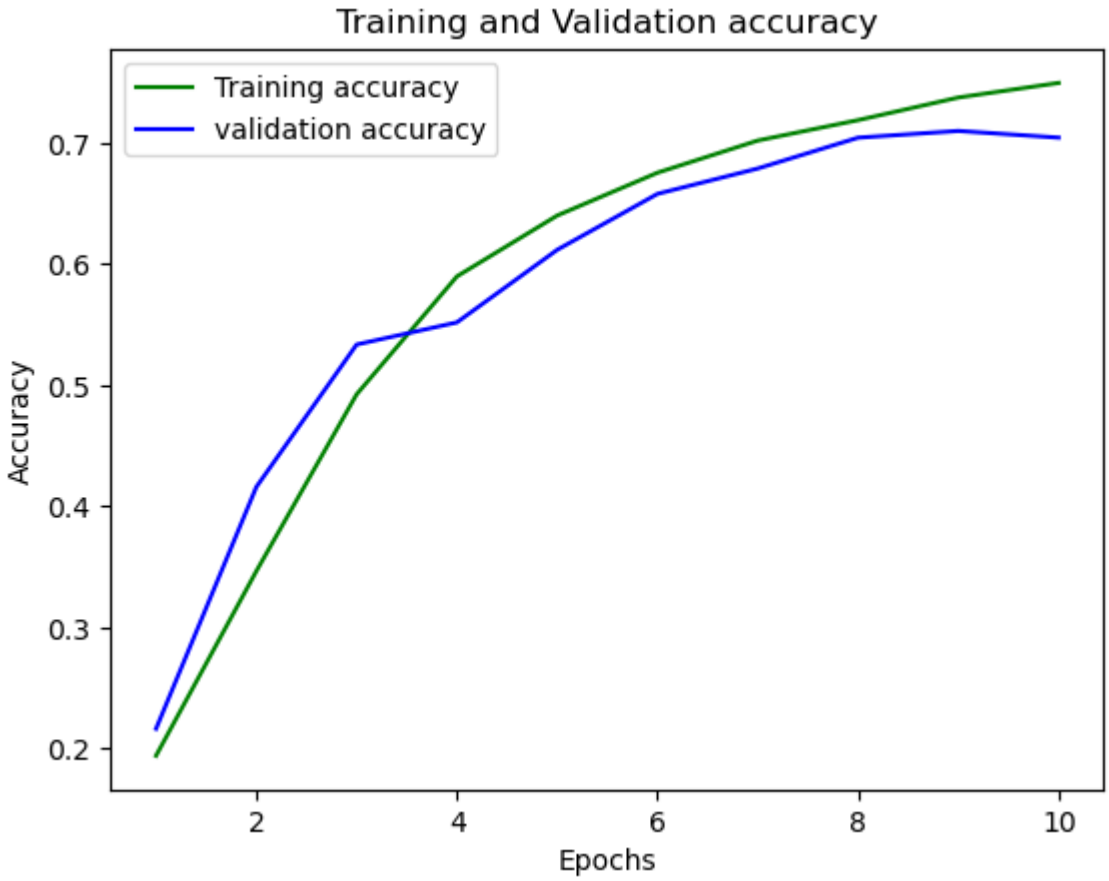
accuracy=0.7705, validation accuracy= 0.7430
Batch size= 150



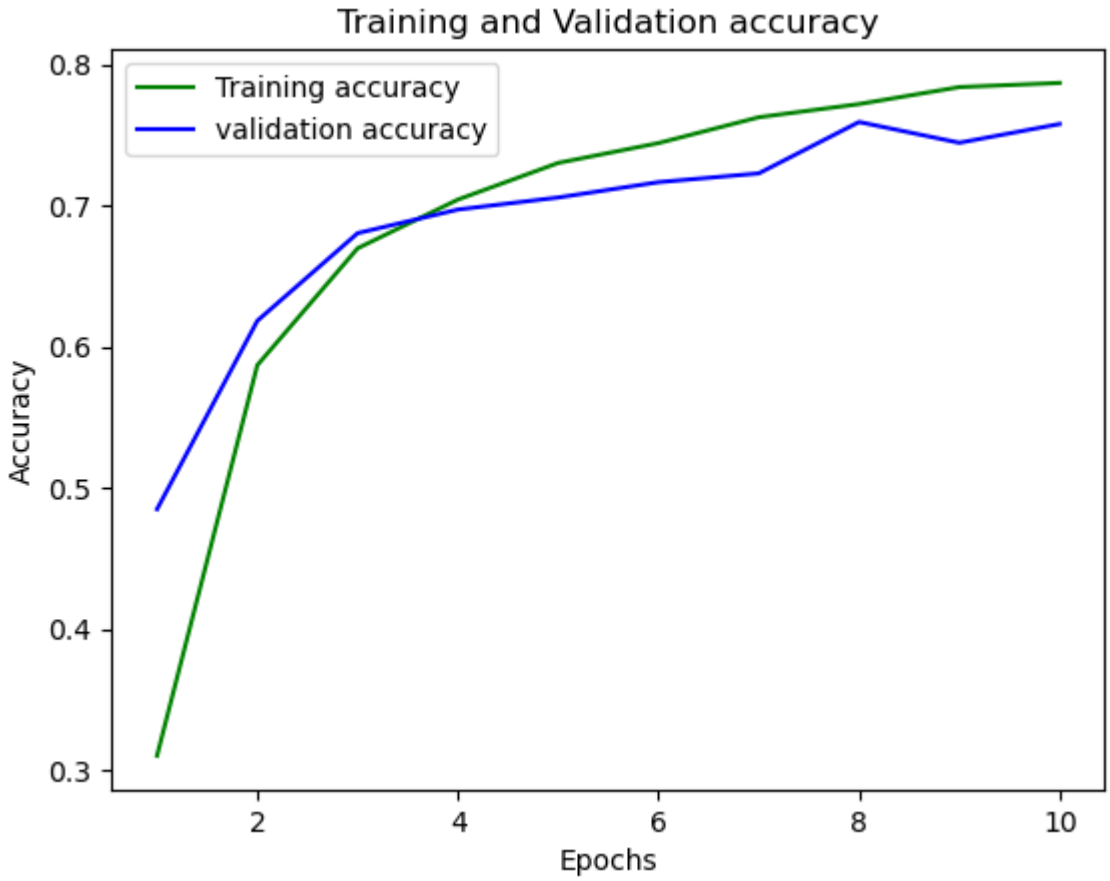
accuracy=0.7880, validation accuracy= 0.7551
Batch size= 250



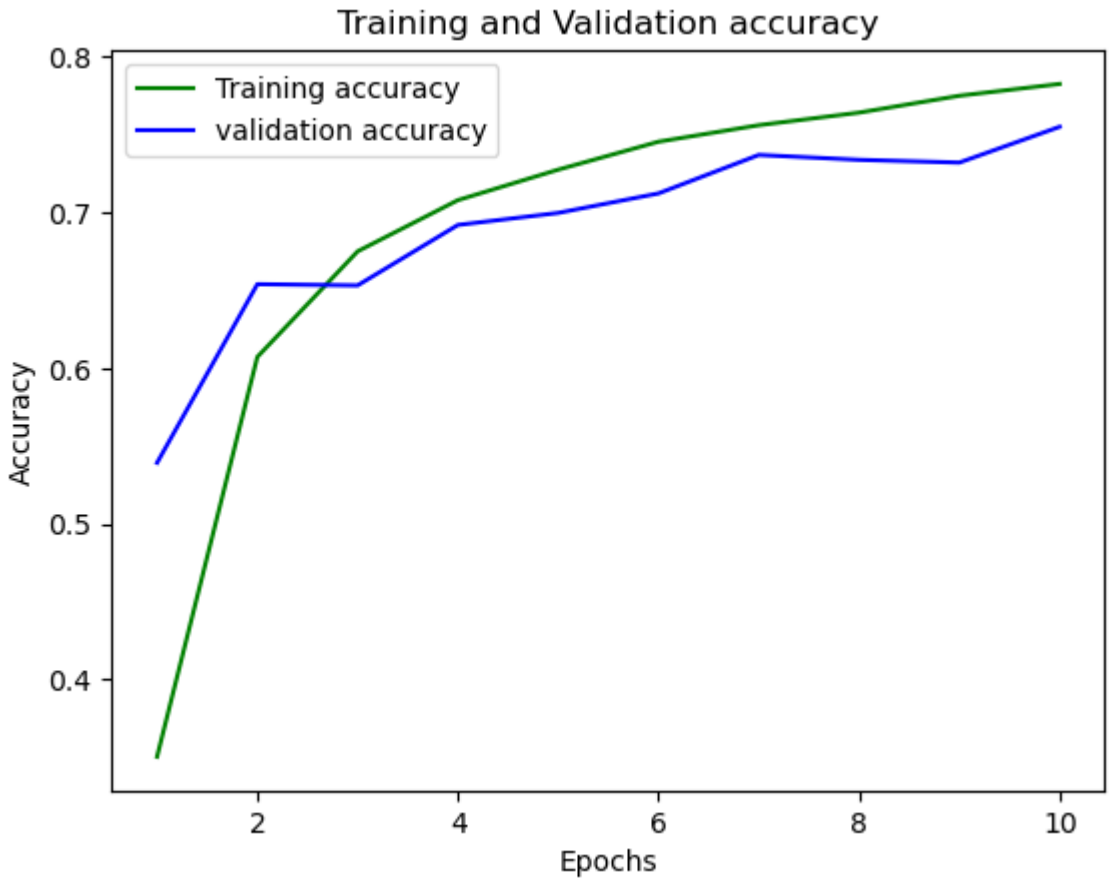
accuracy=0.7855, validation accuracy=0.6747
Batch size= 1000



accuracy=0.7495, validation accuracy= 0.7043
Batch size= 160



accuracy=0.7869, validation accuracy= 0.7578
Batch size= 100



accuracy=0.7824, validation accuracy= 0.7549

Συγκέντρωση γραφημάτων

- Παρατηρώ οτι για πολύ μικρό batch size έχω μικρό accuracy και validation accuracy(πχ 25). Όσο μεγαλώνει το batch size μεγαλώνει το accuracy και το validation accuracy μέχρι ένα σημείο που αρχίζει το validation accuracy να μειώνεται(1000 batch size), και βλέπουμε πως το μοντέλο έχει αρχίσει να δυσκολεύεται να κάνει "γενίκευση", έχουμε overfitting.

Figure 1: batch size = 25

Figure 2: batch size = 32

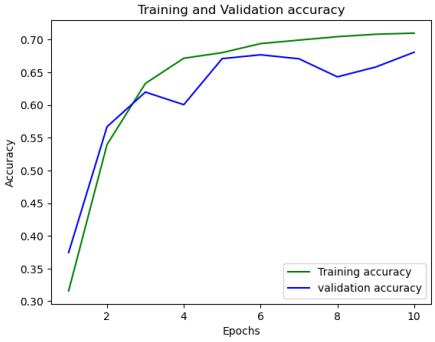


Figure 3: batch size = 35

Figure 4: batch size = 37

Figure 5: batch size = 50

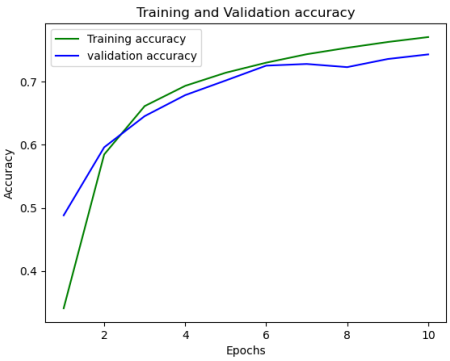


Figure 6: batch size = 60

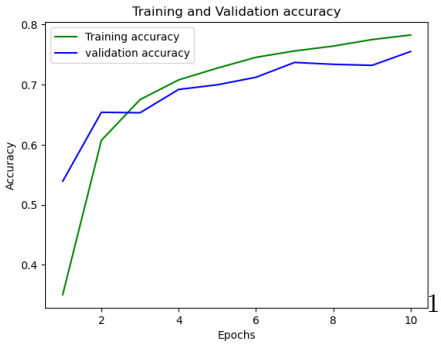


Figure 7: batch size = 100

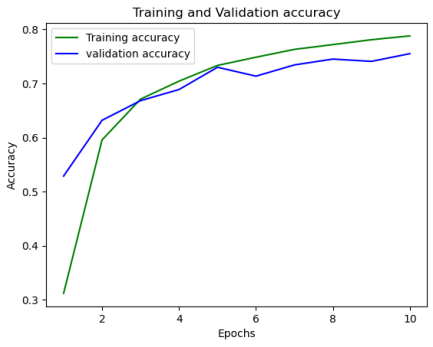


Figure 8: batch size = 150

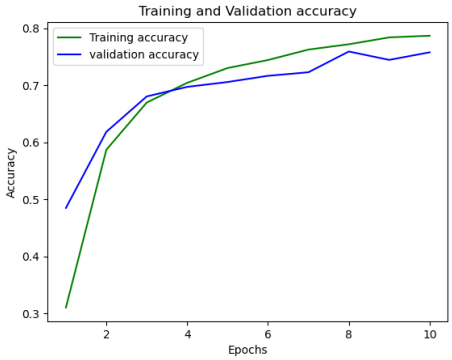


Figure 9: batch size = 160

Figure 10: batch size = 250