



Πολυτεχνείο Κρήτης

Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Αναφορά 2ης Σειράς Ασκήσεων

Συγγραφέας:
Καπελώνης Χαρίλαος

Τιμεύθυνος Καθηγητής:
Δρ. Β. Διακολουκάς

Η εργασία κατατέθηκε για το μάθημα:
Στατιστική Μοντελοποίηση & Αναγνώριση Προτύπων

Εαρινό Εξάμηνο 2024

Περιεχόμενα

Περιεχόμενα	i
1 Θέμα 1: Αλγόριθμος Perceptron	1
2 Θέμα 2: Λογιστική Παλινδρόμηση: Αναλυτική εύρεση κλίσης (Gradient)	7
3 Θέμα 3: Εκτίμηση Παραμέτρων με Maximum Likelihood	10
4 Θέμα 4: Ομαδοποίηση (Clustering) με K-means και GMM	14
5 Θέμα 5β: Convolutional Neural Networks for Image Recognition	21
6 Πηγές	34

Θέμα 1: Αλγόριθμος Perceptron

a)

Αφότου φορτώθηκαν τα δεδομένα του Figure 1.1, σχεδιάστηκαν με την `plt.scatter` στο Figure 1.2. Ακολουθεί ο κώδικας:

Δείγμα	ω_1		ω_2		ω_3		ω_4	
	x_1	x_2	x_1	x_2	x_1	x_2	x_1	x_2
1	0.1	1.1	7.1	4.2	-3	-2.9	-2	-8.4
2	6.8	7.1	-1.4	-4.3	0.5	8.7	-8.9	0.2
3	-3.5	-4.1	4.5	0	2.9	2.1	-4.2	-7.7
4	2	2.7	6.3	1.6	-0.1	5.2	-8.5	-3.2
5	4.1	2.8	4.2	1.9	-4	2.2	-6.7	-4
6	3.1	5	1.4	-3.2	-1.3	3.7	-0.5	-9.2
7	-0.8	-1.3	2.4	-4	-3.4	6.2	-5.3	-6.7
8	0.9	1.2	2.5	-6.1	-4.1	3.4	-8.7	-6.4
9	5	6.4	8.4	3.7	-5.1	1.6	-7.1	-9.7
10	3.9	4	4.1	-2.2	1.9	5.1	-8	-6.3

Figure 1.1: Δεδομένα εισόδου για κάθε κλάση

```
# Given 2D samples per class
class_1 = np.array(
    [[0.1, 1.1], [6.8, 7.1], [-3.5, -4.1], [2, 2.7], [4.1, 2.8],
     [3.1, 5], [-0.8, -1.3], [0.9, 1.2], [5, 6.4],
     [3.9, 4.0]])
class_2 = np.array(
    [[7.1, 4.2], [-1.4, -4.3], [4.5, 0], [6.3, 1.6], [4.2, 1.9],
     [1.4, -3.2], [2.4, -4.0], [2.5, -6.1], [8.4, 3.7],
```

```

[4.1, -2.2]])
class_3 = np.array(
    [[-3, -2.9], [0.5, 8.7], [2.9, 2.1], [-0.1, 5.2], [-4, 2.2],
     [-1.3, 3.7], [-3.4, 6.2], [-4.1, 3.4], [-5.1, 1.6],
     [1.9, 5.1]])
class_4 = np.array(
    [[-2, -8.4], [-8.9, 0.2], [-4.2, -7.7], [-8.5, -3.2], [-6.7,
     -4.0], [-0.5, -9.2], [-5.3, -6.7], [-8.7, -6.4],
     [-7.1, -9.7], [-8.0, -6.3]])

# Question a
plt.figure(figsize=(10, 8))
plt.scatter(class_1[:, 0], class_1[:, 1], c='red', label='class 1')
plt.scatter(class_2[:, 0], class_2[:, 1], c='blue', label='class 2')
plt.scatter(class_3[:, 0], class_3[:, 1], c='green', label='class
    3')
plt.scatter(class_4[:, 0], class_4[:, 1], c='purple', label='class
    4')
plt.title('Scatter Plot for Different Classes')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.grid(True)
plt.show()

```

b)

Ο αλγόριθμος Perceptron είναι ένα απλό είδος γραμμικού ταξινομητή.

Τλοποιείται μηδενίζοντας αρχικά τα βάρη και έπειτα σε μία επαναληπτική διαδικασία υπολογίζονται σε batch μορφή το κόστος και τα νέα βάρη.

Στα γραφήματα θα φανεί και ο αριθμός επαναλήψεων που χρειάστηκαν για να φτάσει ο αλγόριθμος στο ολικό ελάχιστο της cost function, δηλαδή errors == 0.

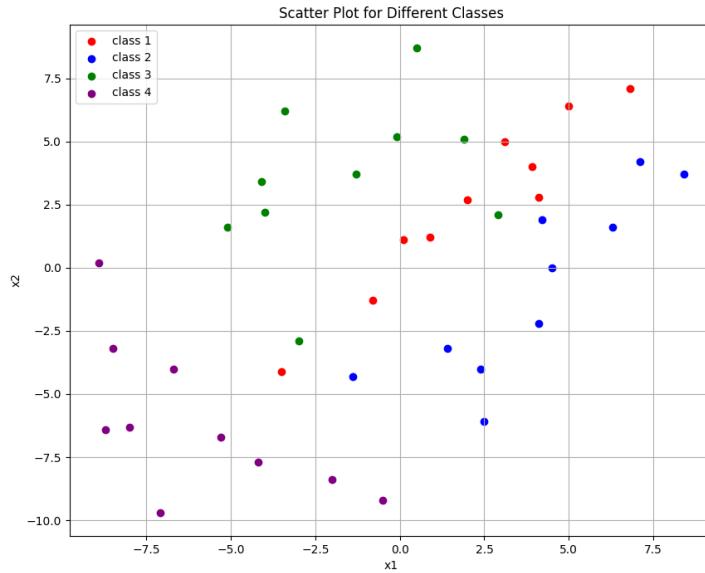


Figure 1.2

```

def perceptron(X, y, max_iter=1000):
    w = np.zeros(X.shape[1]) # initially zero vector
    b = 0
    reps = 0
    for reps in range(max_iter):
        errors = 0
        for xi, yi in zip(X, y):
            if yi * (np.dot(xi, w) + b) <= 0:
                w += yi * xi
                b += yi
                errors += 1
        if errors == 0:
            break
    return w, b, reps

```

c)

Ο αλγόριθμος Perceptron βρίσκει ένα γραμμικό όριο απόφασης που μπορεί να αναπαρασταθεί από την εξίσωση:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x} + b &= 0 \\ w_0x + w_1y + b &= 0 \\ w_1y &= -(w_0x + b) \\ y &= -\frac{w_0x + b}{w_1} \end{aligned}$$

Η συνάρτηση `plot_decision_boundary(w, b, ax)` είναι βοηθητική για να σχεδιαστεί το όριο απόφασης παρακάτω:

```
def plot_decision_boundary(w, b, ax):
    x_values = np.linspace(-10, 10, 200)
    y_values = -(w[0] * x_values + b) / w[1]
    ax.plot(x_values, y_values, 'k--')
```

Για κάθε ένα από τα ζεύγη $w_1 - w_2$, $w_2 - w_3$, $w_3 - w_4$ σχεδιάζεται σε κοινό `plot` στο `Figure 1.3` το όριο απόφασης ως εξής:

```
# class_1 vs class_2
X1 = np.vstack((class_1, class_2))
y1 = np.hstack((np.ones(len(class_1)), -np.ones(len(class_2))))
w1, b1, reps = perceptron(X1, y1)
plot_decision_boundary(w1, b1, axs[0])
axs[0].scatter(class_1[:, 0], class_1[:, 1], c='red', label='class 1')
axs[0].scatter(class_2[:, 0], class_2[:, 1], c='blue', label='class 2')
axs[0].set_title(f'class_1 vs class_2 took {reps} repetitions')
axs[0].set_xlim(-10, 10)
axs[0].set_ylim(-10, 10)
axs[0].legend()
```

Ομοίως για τα άλλα δύο ζεύγη.

Για κάθε περίπτωση απαιτούνται 8, 4 και 22, αντίστοιχα, επαναλήψεις.

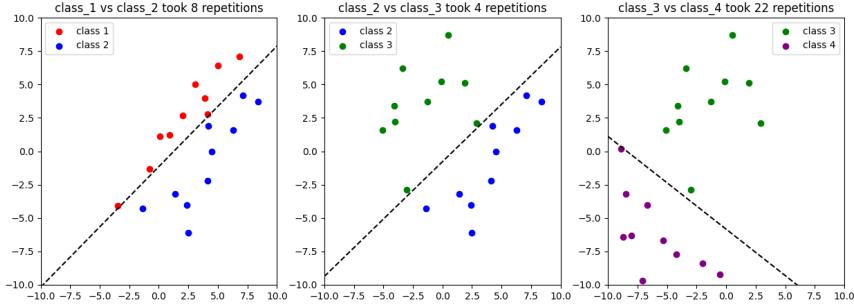


Figure 1.3

Πού οφείλεται αυτή η διαφοροποίηση στον απαιτούμενο αριθμό επαναλήψεων;

Ένας πρώτος λόγος είναι ότι δεν έχει οριστεί κάποια τιμή για το `learning_rate`, ουσιαστικά είναι μονάδα. Για να γίνει αντιληπτό αυτό και γραφικά, έστω ότι μειώναμε το `learning_rate`, τότε στο Figure 1.4 φαίνεται ότι αλλάζει τελείως ο αριθμός επαναλήψεων σε 17, 4, 2(!).

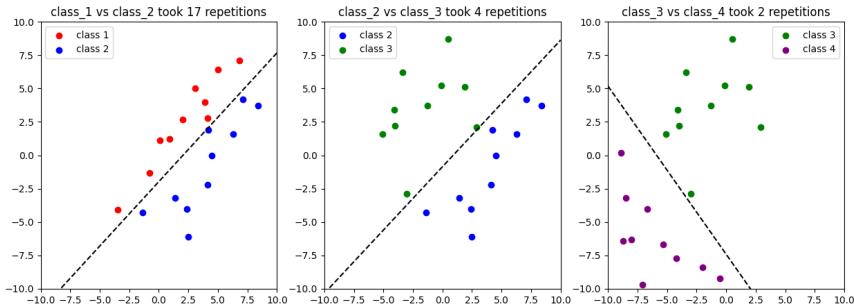


Figure 1.4: learning rate = 0.1

Επιπλέον, σημασία έχει το πόσο μικρή είναι η απόσταση μεταξύ των κλάσεων, δηλαδή πόσο εύκολα διαχωρίσιμα είναι τα δεδομένα. Στην πρώτη περίπτωση είναι πολύ μαζεμένα μεταξύ τους, άρα και δύσκολο να διαχωριστούν.

Ένας τρίτος λόγος είναι οι αρχικές συνθήκες. Στο Figure 1.5 φαίνεται ότι ξεκινώντας με τυχαία βάρη από 0 έως 1, τα αποτελέσματα είναι διαφορετικά και, πολύ πιο πετυχημένα, 6, 3, 3.

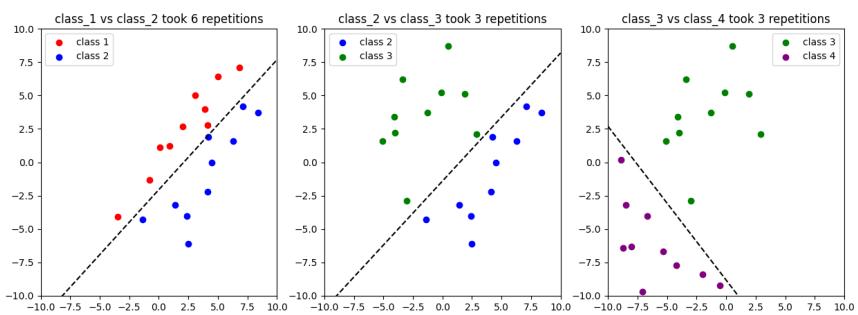


Figure 1.5: learning rate = 0.1, $\mathbf{w} \sim U(0, 1]$

Θέμα 2: Λογιστική Παλινδρόμηση: Αναλυτική εύρεση κλίσης (Gradient)

i)

Αρχικά διαβάζονται τα δεδομένα:

```
# Load Data
data = np.loadtxt('exam_scores_data1.txt', delimiter=',')
X = data[:, [0, 1]]
y = data[:, 2]
```

ii)

Η σιγμοειδής συνάρτηση:

```
def sigmoid(z):
    sigmoid_function = 1 / (1 + 2*np.exp(-z))
    return sigmoid_function
```

iii)

Η συνάρτηση κόστους:

```
# Calculate the cost function
def costFunction(theta, X, y):
    y_exp = sigmoid(np.dot(X, theta))
    J = (-1 / len(y)) * (np.dot(y.T, np.log(y_exp)) + np.dot((1 -
        y).T, np.log(1 - y_exp)))
    return J
```

Η κλίση της συνάρτησης κόστους:

```
# Calculate the gradient of the cost function
def gradient(theta, X, y):
    y_exp = sigmoid(np.dot(X, theta))
    grad = (1 / len(y)) * np.dot(X.T, (y_exp - y))
    return grad
```

iv)

Στον κώδικα της predict, επιστρέφεται 1 (αποδοχή) αν η σιγμοειδής βγει πάνω από 0.5 και 0 διαφορετικά (απόρριψη):

```
# Class Prediction
def predict(theta, X):
    pred_class = sigmoid(np.dot(X, theta)) >= 0.5
    return pred_class
```

Για έναν φοιτητή ο οποίος έγραψε 45 στο πρώτο και 85 στο δεύτερο μάθημα, η πιθανότητα να περάσει είναι:

admission probability of **0.7762946788354029**

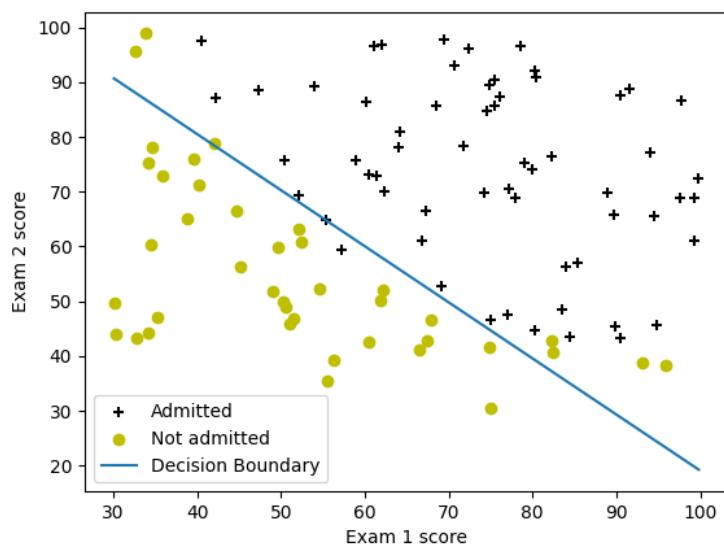


Figure 2.1: Το πρόβλημα επιδέχεται γραμμικής λύσης με μεγάλη ακρίβεια

Θέμα 3: Εκτίμηση Παραμέτρων με Maximum Likelihood

a)

Τα δεδομένα ακολουθούν Gaussian κατανομές σε όλες τις διαφορετικές διαστάσεις, άρα η εκτίμηση μεγίστης πιθανοφάνειας για τις παραμέτρους μέσης τιμής και διασποράς θα γίνει μέσω των αντίστοιχων συναρτήσεων της numpy:

```
# Question a
print("Question a:")
print("Means ():", np.mean(class_1, axis=0))
print("Variances (~2):", np.var(class_1, axis=0, ddof=0))
print("\n\n\n")
```

Με αποτέλεσμα:

```
Question a:
Means (): [-0.0709 -0.6047 -0.911 ]
Variances (~2): [0.90617729 4.20071481 4.541949 ]
```

b)

Ομοίως, ανά δύο:

```
# Question b
combinations = [(0, 1), (1, 2), (0, 2)]
print("Question b:")
for combination in combinations:
    print(f"Characteristics x{combination[0]+1},"
          f" x{combination[1]+1}")
    print("Mean vector () for 2D:", np.mean(class_1[:, combination],
                                             axis=0))
    print("Covariance matrix () for 2D:\n", np.cov(class_1[:, combination],
                                                    rowvar=False, ddof=0))
    print("\n")
print("\n\n\n")
```

Με αποτέλεσμα:

Question b:
 Characteristics x1, x2
 Mean vector () for 2D: [-0.0709 -0.6047]
 Covariance matrix () for 2D:
 [[0.90617729 0.56778177]
 [0.56778177 4.20071481]]

Characteristics x2, x3
 Mean vector () for 2D: [-0.6047 -0.911]
 Covariance matrix () for 2D:
 [[4.20071481 0.7337023]
 [0.7337023 4.541949]]

Characteristics x1, x3
 Mean vector () for 2D: [-0.0709 -0.911]
 Covariance matrix () for 2D:
 [[0.90617729 0.3940801]
 [0.3940801 4.541949]]

c)

Ομοίως, και τα τρία από χοινού:

```
# Question c
print("Question c:")
print("Mean vector () for 3D:", np.mean(class_1, axis=0))
print("Covariance matrix () for 3D:\n", np.cov(class_1,
    rowvar=False, ddof=0))
print("\n\n")
```

Με αποτέλεσμα:

```
Question c:
Mean vector () for 3D: [-0.0709 -0.6047 -0.911 ]
Covariance matrix () for 3D:
[[0.90617729 0.56778177 0.3940801 ]
 [0.56778177 4.20071481 0.7337023 ]
 [0.3940801 0.7337023 4.541949 ]]
```

d)

Με τον ίδιο τρόπο γίνεται και ο υπολογισμός για τη δεύτερη κλάση δεδομένων:

```
# Question d
print("Question d:")
print("Mean vector () for 3D:", np.mean(class_2, axis=0))
print("Diagonal covariance matrix () for 3D:\n",
      np.diag(np.var(class_2, axis=0, ddof=0)))
print("\n\n")
```

Με αποτέλεσμα:

```
Question d:
Mean vector () for 3D: [-0.1126 0.4299 0.00372]
Diagonal covariance matrix () for 3D:
[[0.05392584 0.          0.          ]
 [0.          0.04597009 0.          ]
 [0.          0.          0.00726551]]
```

e)

Τα αποτελέσματα είναι ίδια για κάθε κλάση και αυτό οφείλεται στο γεγονός ότι η επιπλέον πληροφορία δεν επηρεάζει τις επιμέρους παραμέτρους για κάθε διάσταση 1, 2, 3.

Θέμα 4: Ομαδοποίηση (Clustering) με K-means και GMM

a)

Για την ένρεση των κοντινότερων κεντροειδών:

```
# Finds the closest centroid for each sample
def find_closest_centroids(X, centroids):
    m = X.shape[0]
    K = centroids.shape[0]
    idx = np.zeros(m, dtype=int)
    for i in range(m):
        distances = np.linalg.norm(X[i] - centroids, axis=1)
        idx[i] = np.argmin(distances)
    return idx
```

b)

Για τον υπολογισμό των κεντροειδών:

```
# Compute the mean of samples assigned to each centroid
def compute_centroids(X, idx, K):
    n = X.shape[1]
```

```

centroids = np.zeros((K, n))
for k in range(K):
    points = X[idx == k]
    if len(points) > 0:
        centroids[k] = np.mean(points, axis=0)
return centroids

```

c)

Για διάφορες τιμές χρωμάτων (K), όσο αυξάνεται το K αυξάνεται τόσο η ευχρίνεια και πολυχρωμία της εικόνας, όσο και το μέγεθος (σημειώνεται ότι το μέγεθος περιλαμβάνει όλη την εικόνα όπως εμφανίζεται από την Python).

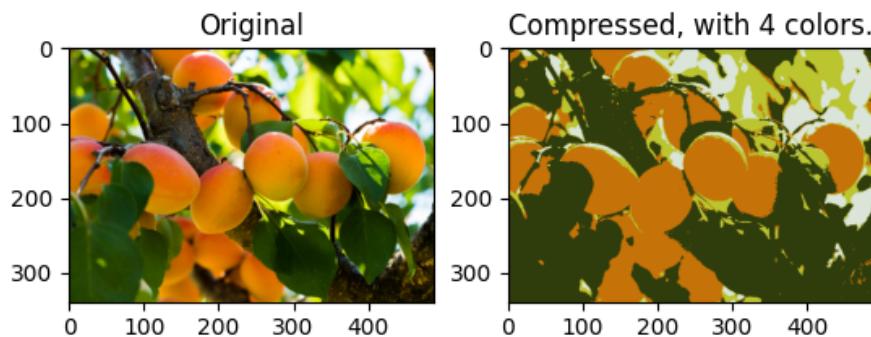


Figure 4.1: $K = 4$ χρώματα, size = 130kb

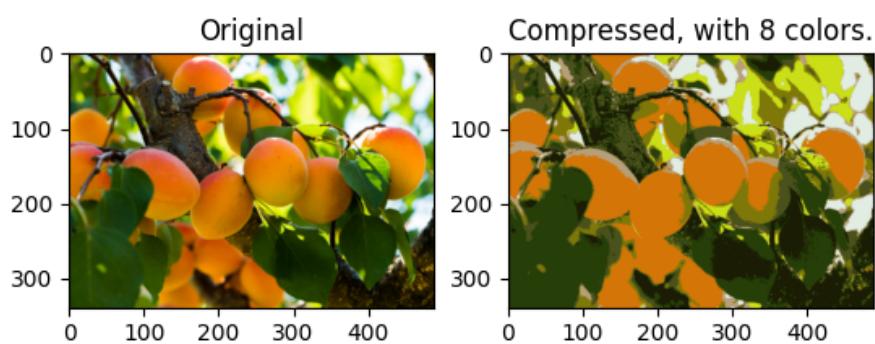


Figure 4.2: $K = 8$ χρώματα, size = 147kb

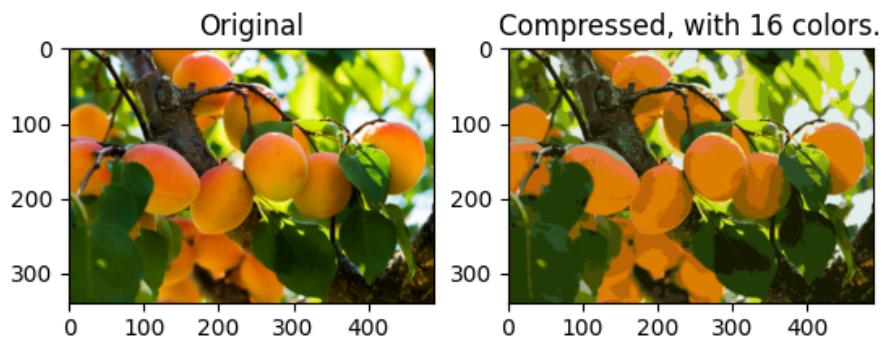


Figure 4.3: $K = 16$ χρώματα, size = 156kb

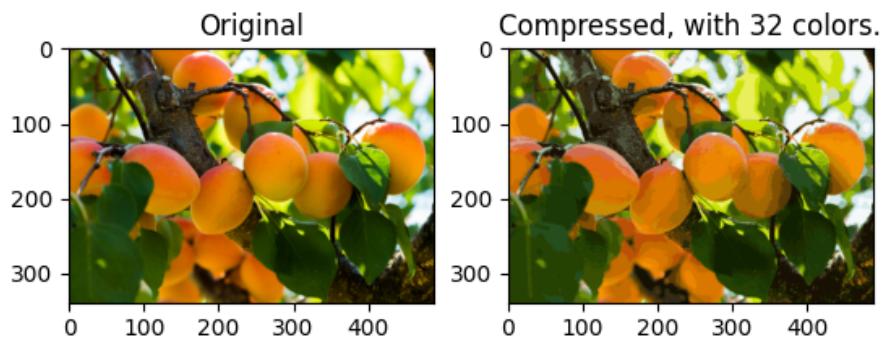


Figure 4.4: $K = 32$ χρώματα, size = 166kb

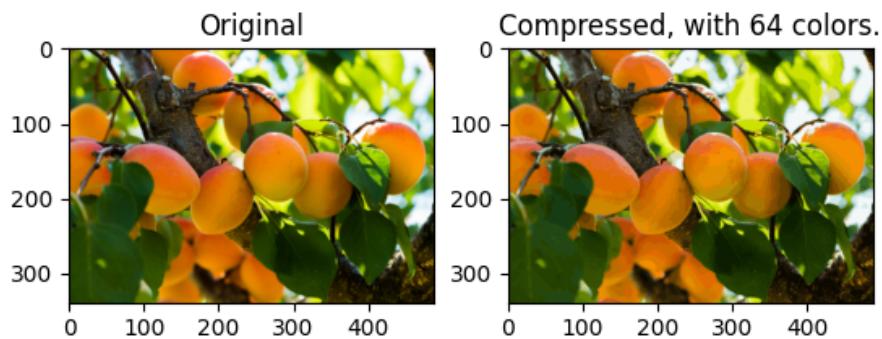


Figure 4.5: $K = 64$ χρώματα, size = 173kb

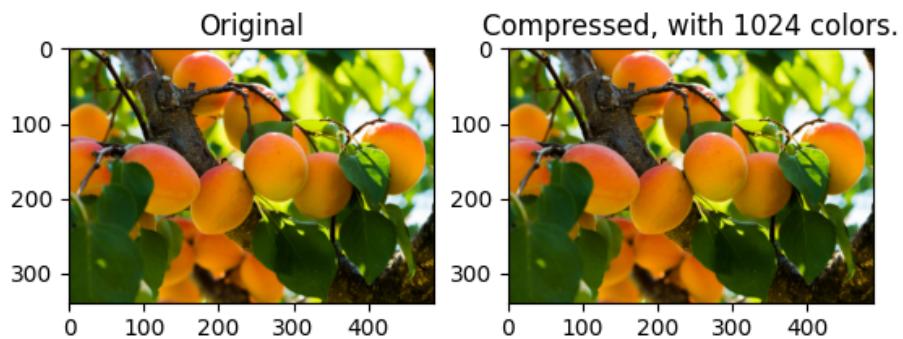


Figure 4.6: $K = 1024$ χρώματα, size = 179kb

Θέμα 5β: Convolutional Neural Networks for Image Recognition

1, 2)

Η αρχιτεκτονική που δίνεται στο `fashion.py` είναι απλή:

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10)
])
```

Εγινε δοκιμή για κάθε αλγόριθμο βελτιστοποίησης, για 400 εποχές και `batch_size` 1024 (στην GPU άρα δεν πήρε πάρα πολύ χρόνο) και τα αποτελέσματα είναι τα εξής:

Αλγόριθμος	Train Accuracy
adam	0.9937999844551086
nadam	0.9911166429519653
rmsprop	0.9841333627700806
adamax	0.9817333221435547
sgd	0.8815333247184753
ftrl	0.7889333367347717

Table 5.1: Train Accuracy σε φθίνουσα σειρά

Όπως φαίνεται και στους πίνακες 5.1, 5.2 ο αλγόριθμος `ftrl` είναι εμφανώς

Αλγόριθμος	Test Accuracy
adamax	0.8901000022888184
nadam	0.8810999989509583
adam	0.8802000284194946
rmsprop	0.8766999840736389
sgd	0.8622999787330627
ftrl	0.7770000100135803

Table 5.2: Test Accuracy σε φθίνουσα σειρά

κακός, ενώ ο sgd είναι μέτριος (συγχριτικά).

Οι adam, adamax, nadam, rmsprop είναι οι καλύτεροι στο test, ενώ δεν υστερούν και στην εκπαίδευση.

Η διάκριση αυτή οφείλεται στο γεγονός ότι οι δεύτεροι είναι **adaptive algorithms** που σημαίνει ότι προσαρμόζουν το learning rate τους.

Ακολουθούν και οι εικόνες στις οποίες φαίνονται αναλυτικά κάποιες προβλέψεις:

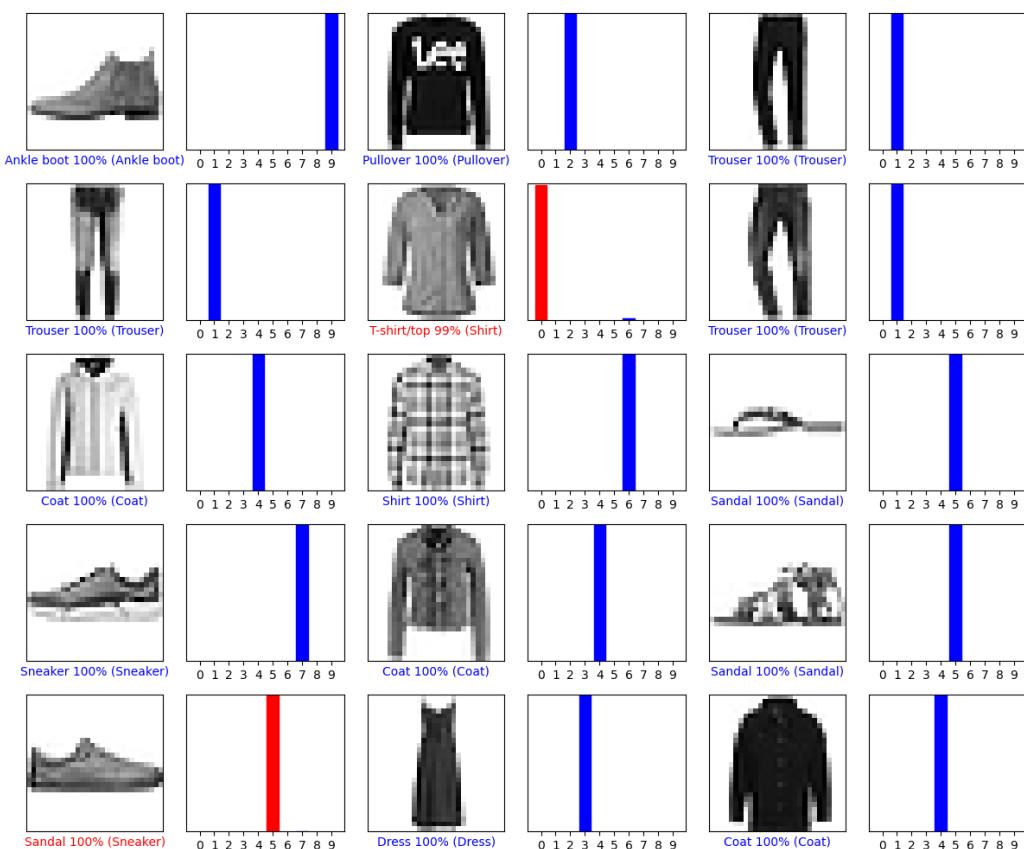


Figure 5.1: adam

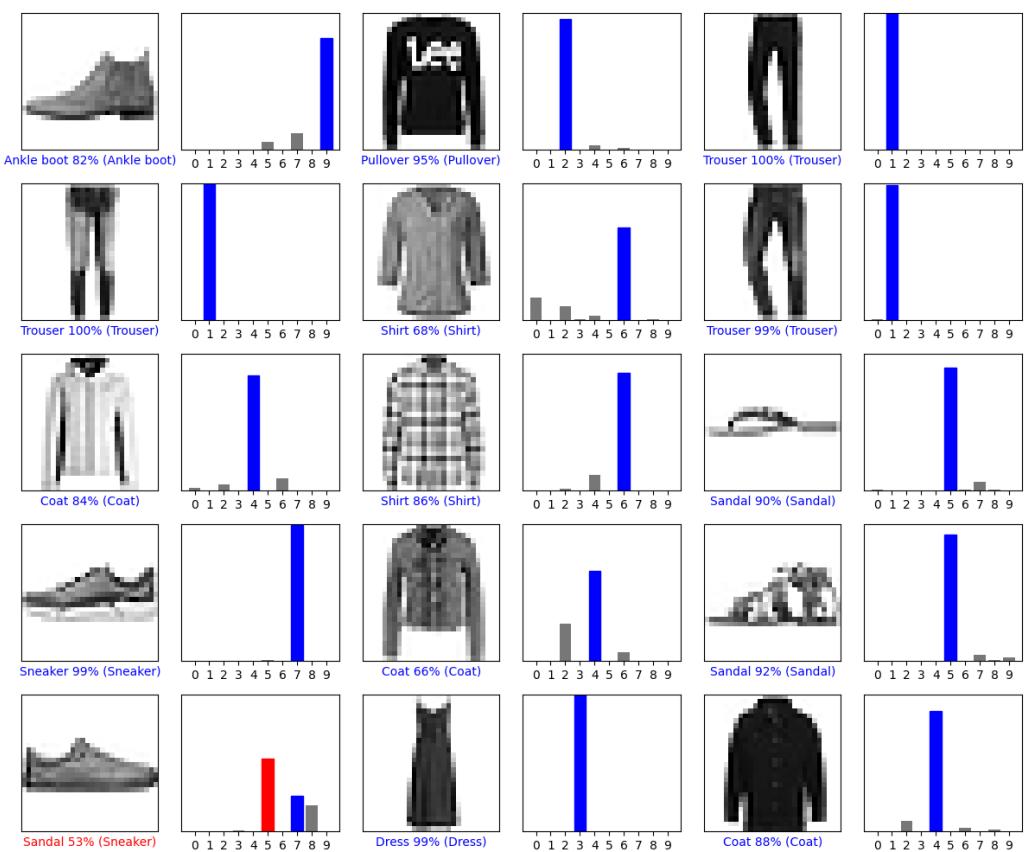


Figure 5.2: sgd

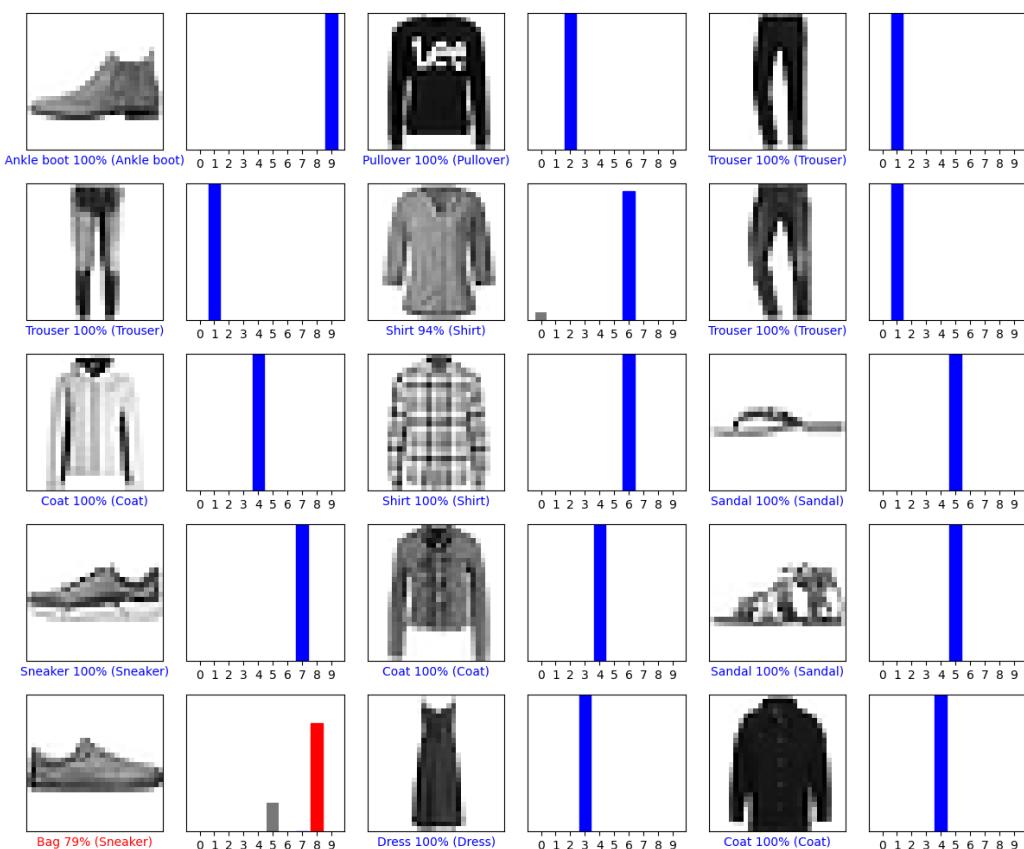


Figure 5.3: rmsprop

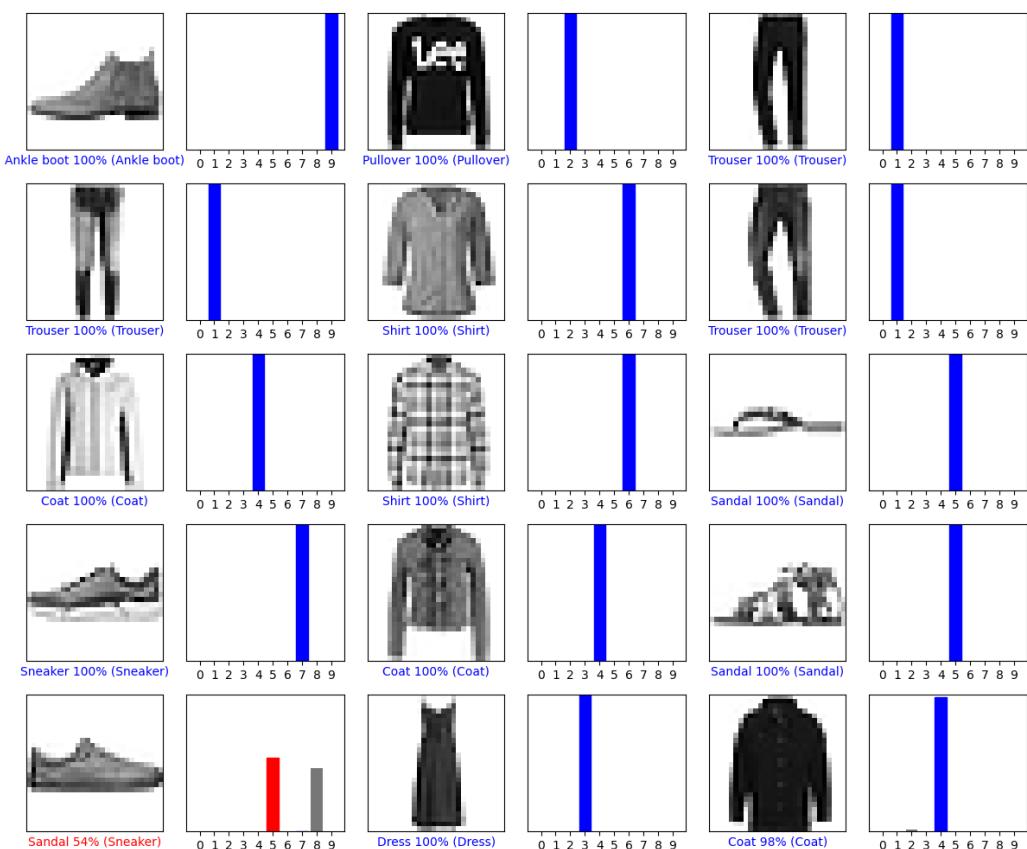


Figure 5.4: nadam

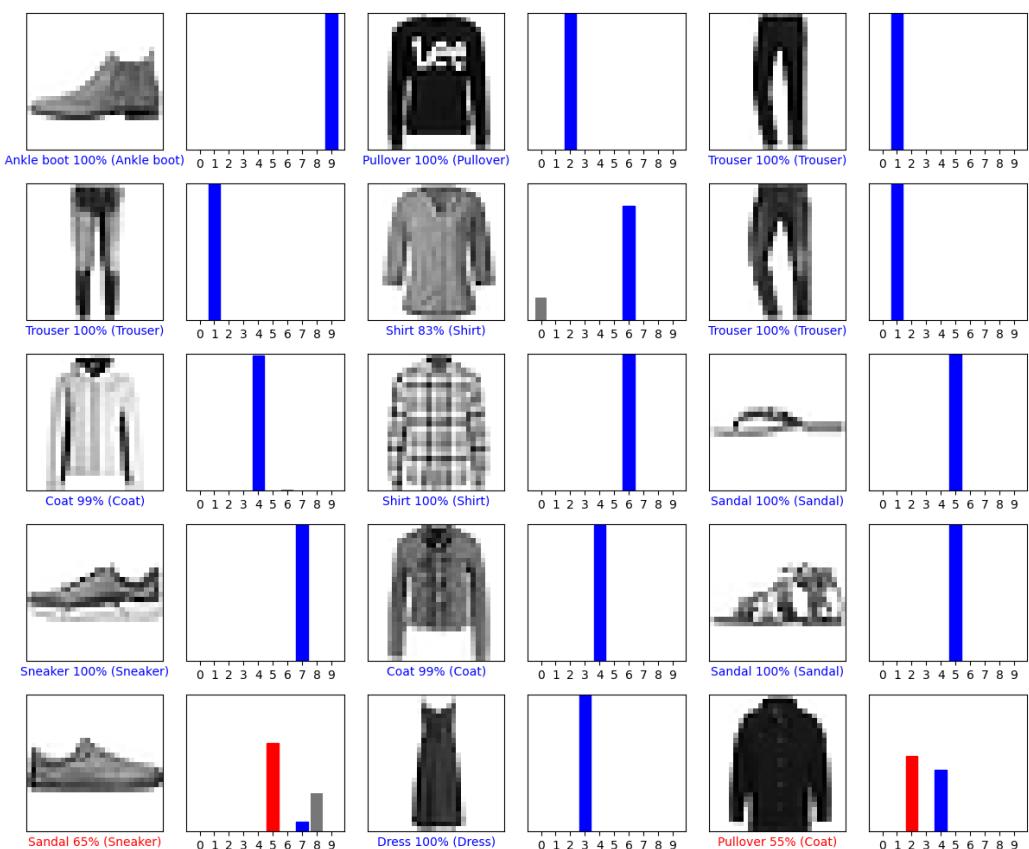


Figure 5.5: adamax

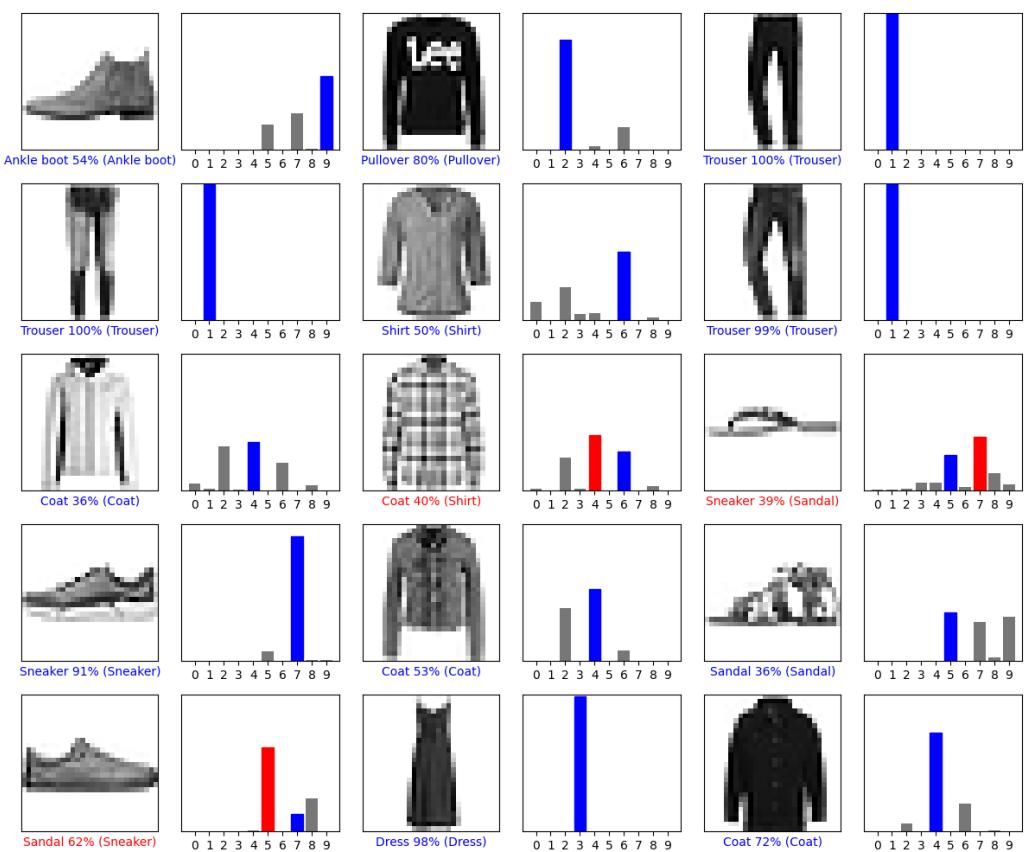


Figure 5.6: ftrl

3, 4, 5)

Η αρχιτεκτονική που δίνεται στο `fashion_cnn.py` είναι αρκετά πιο περίπλοκη, αλλά αν ακολουθήσει κανείς το σχήμα βγάζει νόημα:

```
# Build the model
# Building the neural network requires configuring the layers of
# the model, then compiling the model.
model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), padding='same', activation=None,
                       input_shape=(28, 28, 1)),
    keras.layers.BatchNormalization(),
    keras.layers.ReLU(),

    keras.layers.Conv2D(32, (3, 3), padding='same', activation=None,
                       input_shape=(28, 28, 1)),
    keras.layers.BatchNormalization(),
    keras.layers.ReLU(),

    keras.layers.MaxPooling2D((2, 2), padding='valid',
                             input_shape=(28, 28, 1)),
    keras.layers.Dropout(0.2),

    keras.layers.Conv2D(64, (3, 3), padding='same', activation=None,
                       input_shape=(14, 14, 32)),
    keras.layers.BatchNormalization(),
    keras.layers.ReLU(),

    keras.layers.Conv2D(64, (3, 3), padding='same', activation=None,
                       input_shape=(14, 14, 64)),
    keras.layers.BatchNormalization(),
    keras.layers.ReLU(),

    keras.layers.MaxPooling2D((2, 2), padding='valid',
                             input_shape=(14, 14, 64)),
    keras.layers.Dropout(0.3),

    keras.layers.Conv2D(128, (3, 3), padding='same',
                      activation=None, input_shape=(7, 7, 64)),
    keras.layers.BatchNormalization(),
    keras.layers.ReLU(),
```

```

keras.layers.MaxPooling2D((2, 2), padding='valid',
    input_shape=(7, 7, 128)),
keras.layers.Dropout(0.4),

keras.layers.Flatten(),

keras.layers.Dense(units=3*3*128, activation=None),
keras.layers.BatchNormalization(),
keras.layers.ReLU(),
keras.layers.Dropout(0.5),

keras.layers.Dense(units=200, activation=None),
keras.layers.ReLU(),

keras.layers.Dense(units=len(class_names)),
])

```

Σε αυτήν την περίπτωση ο αλγόριθμος βελτιστοποίησης είναι ο adam και τα αποτελέσματα για 50 εποχές και batch_size 1024:

```

Train accuracy: \textbf{0.9703999757766724}
313/313 - 1s - loss: 0.2039 - accuracy: 0.9306 - 1s/epoch - 4ms/step

```

```

Test accuracy: \textbf{0.9305999875068665}
313/313 [=====] - 1s 2ms/step

```

Στο test είναι πολύ καλύτερο από την αρχική αρχιτεκτονική. To train δεν είναι τόσο καλό, αφού τέθηκαν μόνο 50 εποχές.

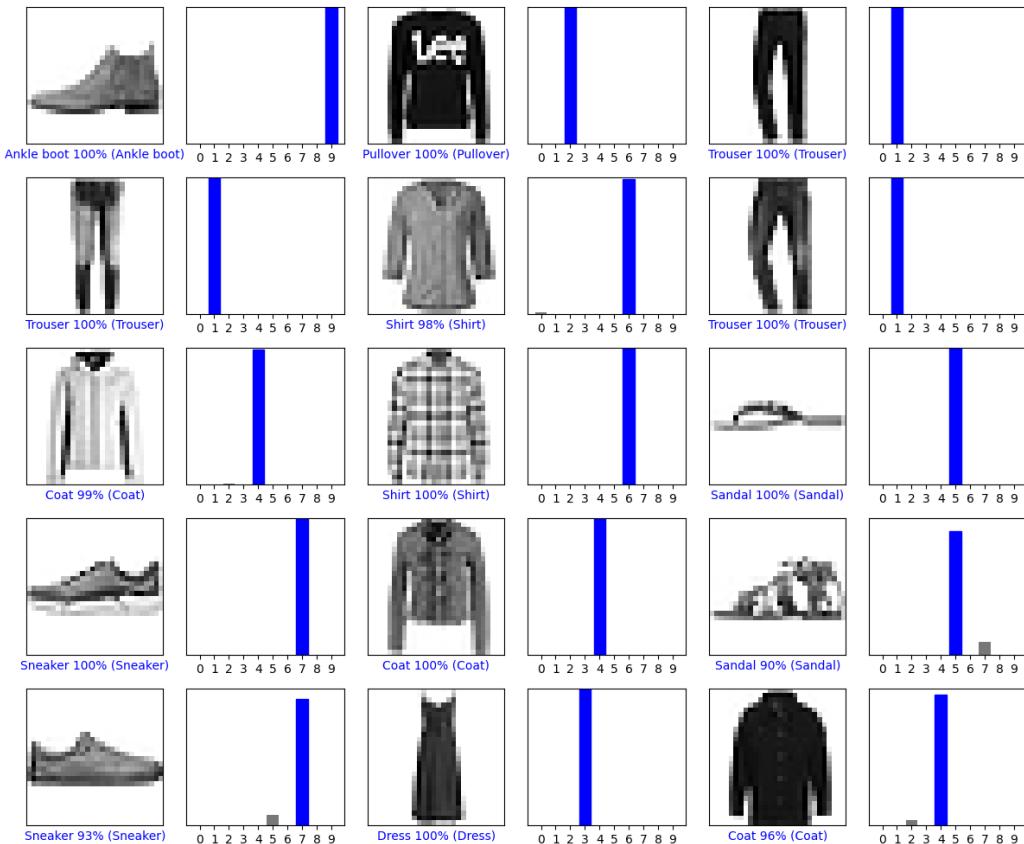


Figure 5.7: CNN: adam, epochs=50, batch size=1024

Αν τεθούν πάλι 400 εποχές, με σταθερές τις υπόλοιπες παραμέτρους:

Train accuracy: 0.9999333620071411

313/313 - 1s - loss: 0.2641 - accuracy: 0.9445 - 1s/epoch - 3ms/step

Test accuracy: 0.9445000290870667

313/313 [=====] - 2s 1ms/step

Τα αποτελέσματα είναι τα καλύτερα δυνατά.

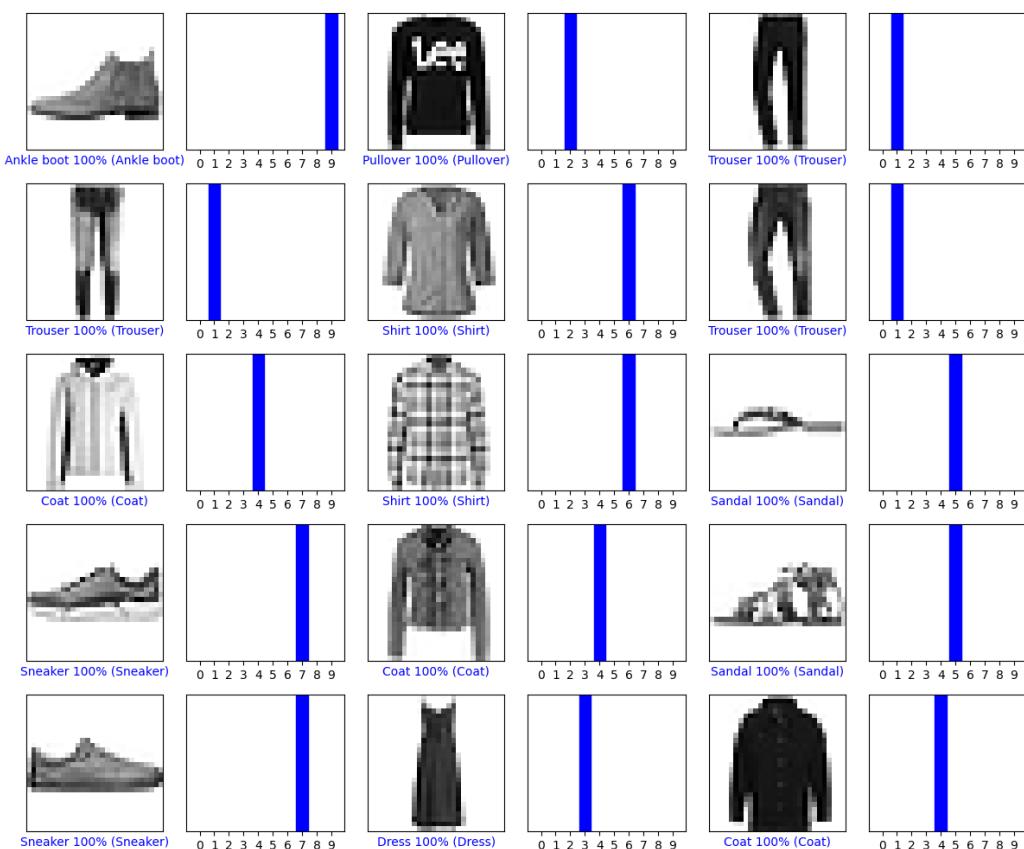


Figure 5.8: CNN: adam, epochs=400, batch size=1024

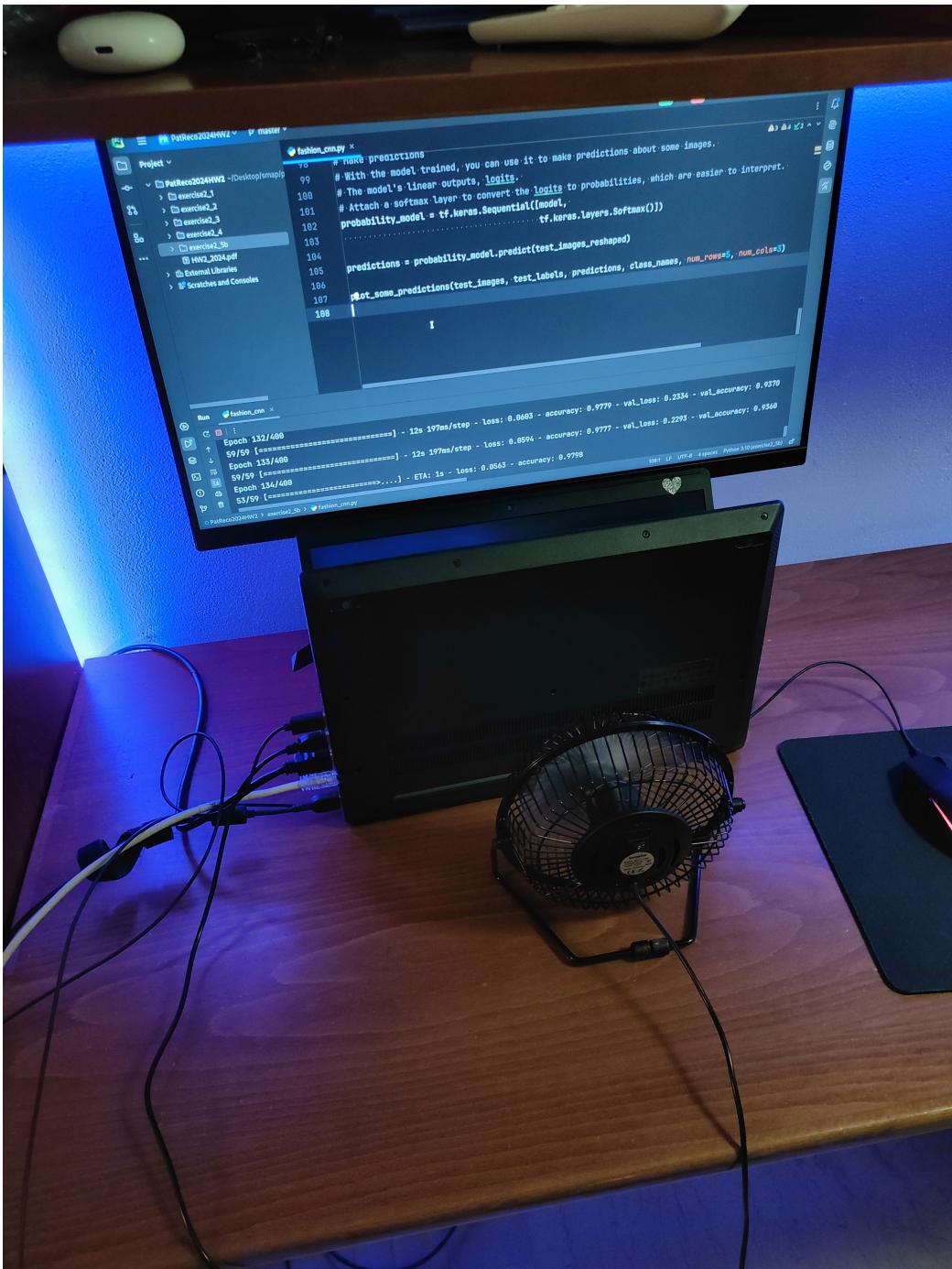


Figure 5.9: Budget Machine Learning: When Greek summer hits, every fan counts

Πηγές

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html

<https://stackoverflow.com/questions/21562986/numpy-matrix-vector-multiplication/21563036#21563036>

https://keras.io/api/layers/core_layers/dense/

<https://keras.io/api/optimizers/>

<https://keras.io/api/models/sequential/>

https://en.wikipedia.org/wiki/Convolutional_neural_network

<https://pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/>