2 Η ΥΠΟΧΡΕΩΤΙΚΗ ΕΡΓΑΣΙΑ ΣΤΟ ΜΑΘΗΜΑ «ΝΕΥΡΩΝΙΚΑ ΔΙΚΤΥΑ – ΒΑΘΙΑ ΜΑΘΗΣΗ»

Η παρούσα εργασία ασχολείται με την υλοποίηση ενός **Support Vector Machine** σε python, με στόχο τον διαχωρισμό 2 κλάσεων των εικόνων του dataset CIFAR-10. Συγκεκριμένα, το μοντέλο θα εκπαιδευτεί για τον διαχωρισμό τον κλάσεων "cat" και "dog"

Η υλοποίηση του SVM γίνετε χωρίς την χρήση εξωτερικών βιβλιοθηκών εκτός από NumPy και CuPy για gpu acceleration. Ακολουθούν τα βήματα υλοποίησης.

Binary Classification with Linear Kernel

1) Κάνω load to CIFAR-10 dataset με την βοήθεια του **tensorflow.keras.datasets**
2) Επιλέγω ποιες 2 από τις 10 κλάσεις θέλω να διαχωρίσω, απαλείφοντας τις άλλες 8.
3) Ορίζω τις τιμές των labels για την πρώτη κλάση(dog) σαν **+1** και για την δεύτερη(cat) **-1.**
4) Κάνω flatten τις φωτογραφίες από 3D σε 1D και κανονικοποιώ από [0,255] σε [0,1].
5) Ακολουθεί η υλοποίηση των συναρτήσεων του SVM

def compute_loss (w, X, y, reg_lambda):

Επέλεξα **Hinge loss** για loss function με **L2** regularization.

1) $Hinge_{loss} = \frac{1}{n}\sum_{i=1}^{n} \max\left(0{,}1 - y_i f(x)\right)$, όπου y_i η πραγματική τιμή, $f(x)$ η εκτίμηση του μοντέλου που περιγράφετε σαν $f(x) = w^T x_i$

2) $L2_{loss} = \frac{\lambda}{2}||w||^2$ ή $\frac{\lambda}{2}\sum_{j=1}^{d} w_j^2$,

Με **total loss** το άθροισμά τους

```
def compute_loss(w, X, y, reg_lambda):
    n_samples = X.shape[0]
    distances = cp.zeros(n_samples)  # Αρχικοποίηση
    for i in range(n_samples):
        dot_product = cp.dot(w.T, X[i])  # Υπολογισμός w^T * x_i
        distances[i] = 1 - y[i] * dot_product
    distances = cp.maximum(0, distances)  # max(0, distance)
    hinge_loss = cp.mean(distances)  # Μέσος όρος των αποστάσεων (hinge loss)
    reg_loss = 0.5 * reg_lambda * cp.sum(w ** 2)
    # Συνολική απώλεια
    total_loss = hinge_loss + reg_loss
    return total_loss
```

Επεξήγηση κώδικα:

Με βάση τους ορισμούς που οριστήκαν πριν, για κάθε δείγμα $\boldsymbol{i}$, υπολογίζεται το $\boldsymbol{w}^T\boldsymbol{x}_i$, δηλαδή το εσωτερικό γινόμενο των βαρών $\boldsymbol{w}$ και του διανύσματος χαρακτηριστικών $\boldsymbol{x}_i$. Υπολογίζω την απόσταση ως $\mathbf{1} - \boldsymbol{y}_i \cdot (\boldsymbol{w}^T\boldsymbol{x}_i)$, Αν το δείγμα ταξινομήθηκε σωστά, αυτή η τιμή είναι αρνητική ή μηδέν. Φιλτράρω τις αποστάσεις ώστε να κρατάω μόνο τις θετικές (λάθος ταξινομημένα), αυτό διασφαλίζει ότι τα σωστά ταξινομημένα δείγματα δεν συνεισφέρουν στην απώλεια. Βρίσκω τον μέσο όρο των φιλτραρισμένων αποστάσεων ορίζοντας το **hinge_loss**. Έπειτα υπολογίζω το L2 **reg_loss** όπως έχει οριστεί, προσθέτοντας το **hinge_loss** και το **reg_loss**, επιστρέφω την συνολική απώλεια.

def compute_gradient (w, X, y, reg_lambda):

Σύμφωνα με το source, η παράγωγος της hinge loss ως προς $\boldsymbol{w}_i$ για $\boldsymbol{x}_i$ είναι:

$$\frac{\partial \boldsymbol{l}_{hinge}}{\partial \boldsymbol{w}_i} = \begin{cases} -\boldsymbol{y} \cdot \boldsymbol{x}_i, & \mathbf{1} - \boldsymbol{y} \cdot (\boldsymbol{w}^T\boldsymbol{x}_i) > \mathbf{0} \\ \mathbf{0}, & \boldsymbol{x}_i \geq \mathbf{0} \end{cases}$$

Γνωρίζοντας επίσης πως η παράγωγος του $L2_{loss}$ εύκολα υπολογίζεται σε:

$$\frac{\partial(\frac{1}{2}\lambda \boldsymbol{w}_j^2)}{\partial \boldsymbol{w}_i} = \lambda \boldsymbol{w}_j$$

```python
def compute_gradient(w, X, y, reg_lambda):
    n_samples, n_features = X.shape
    grad_hinge = cp.zeros(n_features)  # Αρχικοποίηση
    for i in range(n_samples):
        dot_product = cp.dot(w.T, X[i])
        distance = 1 - y[i] * dot_product
        if distance > 0:
            grad_hinge += -y[i] * X[i]
    # Υπολογισμός του μέσου όρου του gradient hinge loss
    grad_hinge /= n_samples
    # Υπολογισμός του regularization gradient
    grad_reg = reg_lambda * w
    # Συνολικό gradient
    grad_w = grad_reg + grad_hinge
    return grad_w
```

Επεξήγηση κώδικα:

Όμοιος με πριν, υπολογίζω το $\mathbf{1} - \boldsymbol{y}_i \cdot (\boldsymbol{w}^T\boldsymbol{x}_i)$, ελέγχοντας αν είναι μεγαλύτερο του 0, αν είναι προσθέτω $-\boldsymbol{y} \cdot \boldsymbol{x}_i$ στο **grad_hindge**. Υπολογίζω τον μέσο όρο του των παραγώγων και τον προσθέτω στην παράγωγο της κανονικοποίησης. Έτσι επιστρέφω το συνολικό gradient .

def train_svm(X, y, learning_rate, reg_lambda, num_epochs):

```
def train_svm(X, y, learning_rate, reg_lambda, num_epochs):
    n_samples, n_features = X.shape
    w = cp.zeros(n_features)
    pbar = tqdm(range(1, num_epochs + 1), desc='Training Progress')
    for epoch in pbar:
        grad_w = compute_gradient(w, X, y, reg_lambda)
        w -= learning_rate * grad_w
        loss = compute_loss(w, X, y, reg_lambda)
        pbar.set_description(f'Epoch {epoch}, Loss: {loss:.4f}')
    return w
```

Επεξήγηση κώδικα:

Αρχικά, αρχικοποιώ τα βάρη με 0 και με την βοήθεια της βιβλιοθήκης **tqdm** για την παρακολούθηση της προόδου, μπαίνω στην λούπα εκπαίδευσης για συγκεκριμένες εποχές. Υπολογίζεται το gradient της συνάρτησης απώλειας (Hinge Loss + regularization) καλώντας τη **compute_gradient**. Ενημερώνω τα βάρη σύμφωνα με gradient descend. Υπολογίζω το loss και το εμφανίζω για κάθε εποχη. Τέλος επιστρέφω τα τελικά βάρη.

def predict(X, w):

```
def predict(X, w):
    n_samples = X.shape[0]
    predictions = cp.zeros(n_samples)
    for i in range(n_samples):
        dot_product = cp.dot(w.T, X[i])
        predictions[i] = cp.sign(dot_product)
    return predictions
```

Επεξήγηση κώδικα:

Υπολογίζω το $\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}}$ για κάθε δείγμα $\boldsymbol{x}_{\boldsymbol{i}}$ ξεχωριστά, αν το $\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}} < \mathbf{0}$, τότε η πρόβλεψη είναι -1 και +1 για $\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}} > \mathbf{0}$. Επιστρέφω τον πίνακα **predictions** που έχει τις προβλέψεις για κάθε δείγμα.

ΑΠΟΤΕΛΕΣΜΑΤΑ: ✓ 11m 7.7s

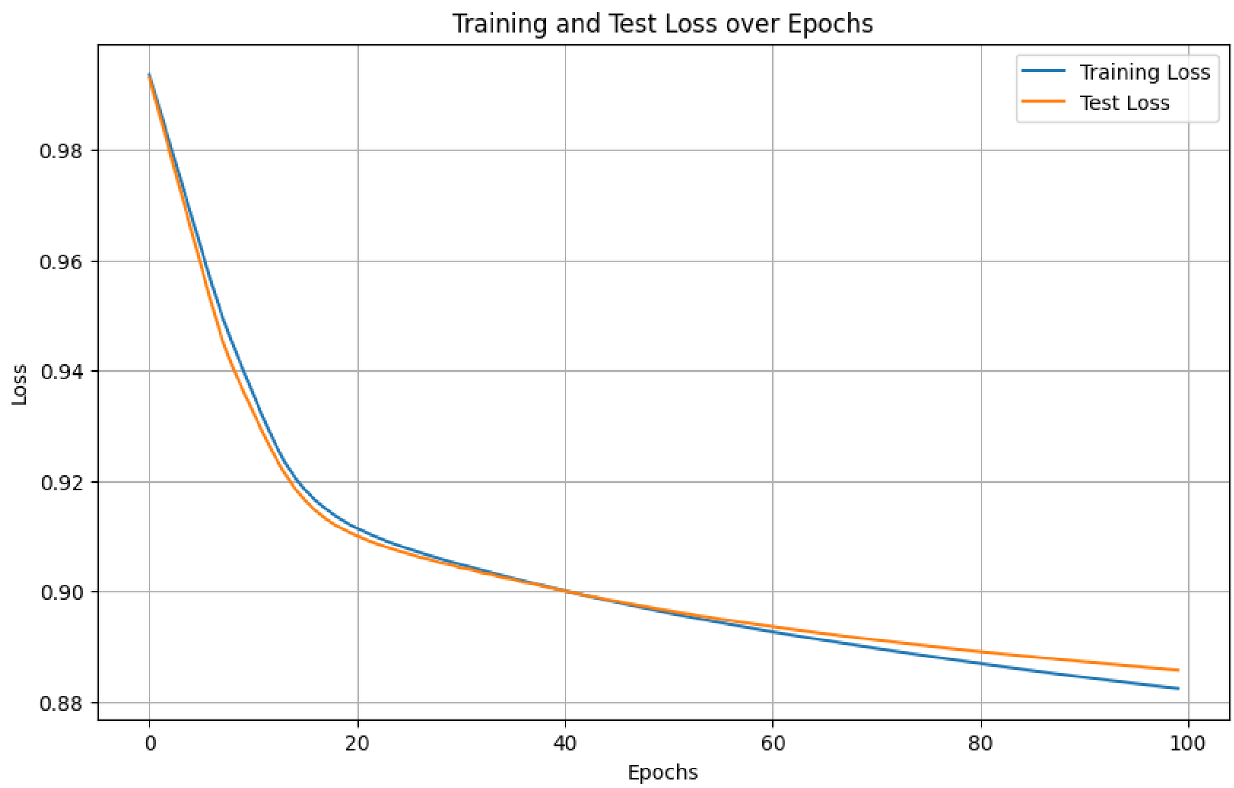
Το μοντέλο εκπαιδευτικέ για ταξινόμηση των κλάσεων "cat" και "dog"

Υπερπαραμέτροι:

- learning_rate = 0.01
- reg_lambda = 0.001
- num_epochs = 100

Test Accuracy: 58.55%

Training Accuracy: 59.06%




```
Test Set Classification Report:
              precision    recall  f1-score   support

    Class -1       0.59      0.55      0.57      1000
     Class 1       0.58      0.62      0.60      1000

    accuracy                           0.59      2000
   macro avg       0.59      0.59      0.59      2000
weighted avg       0.59      0.59      0.59      2000
```

ΒΕΛΙΣΤΟΠΟΙΗΣΕΙΣ

Η εκπαίδευση με for-loops αυξάνει σημαντικά την πολυπλοκότητα, ακολουθεί υλοποίηση με παραλληλισμό:

```python
def compute_loss(w, X, y, reg_lambda):
    dot_products = cp.dot(X, w)
    distances = 1 - y * dot_products
    distances = cp.maximum(0, distances)
    hinge_loss = cp.mean(distances)
    reg_loss = 0.5 * reg_lambda * cp.sum(w ** 2)
    total_loss = hinge_loss + reg_loss
    return total_loss
```

Έτσι, Όλες οι αποστάσεις $\mathbf{1} - \boldsymbol{y_i} \cdot (\boldsymbol{w^T x_i})$, υπολογίζονται ταυτόχρονα μέσω του `cp.dot(X, w)`, χωρίς τη χρήση βρόχου. Η πράξη `y * dot_products` εκτελείται διανυσματικά για όλα τα δείγματα και όχι ξεχωριστά για κάθε δείγμα.

```
def compute_gradient(w, X, y, reg_lambda):
    n_samples = X.shape[0]
    dot_products = cp.dot(X, w)
    distances = 1 - y * dot_products
    misclassified = distances > 0
    grad_hinge = -cp.dot((misclassified * y).T, X) / n_samples
    grad_reg = reg_lambda * w
    grad_w = grad_reg + grad_hinge
    return grad_w
```

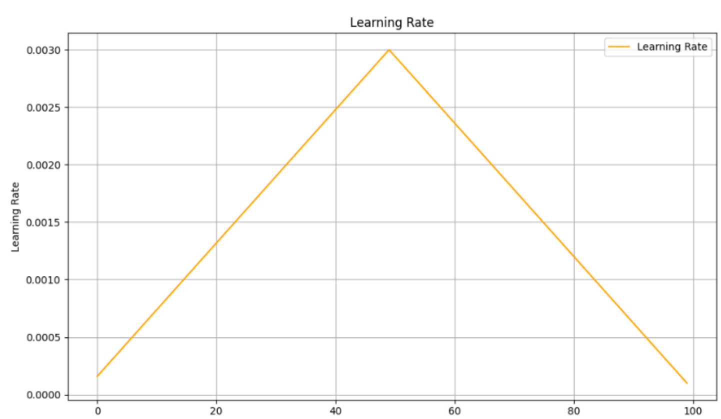
Όμοια με πριν υπολογίζει τις αποστάσεις $\mathbf{1} - \boldsymbol{y_i} \cdot (\boldsymbol{w^T x_i})$, δημιουργώ boolean array **misclassified** για το φιλτράρισμα των δειγμάτων. Το misclassified * y, διαβεβεώνει πως τα δείγματα που παραβιάζουν το margin εχουν τιμή $\boldsymbol{y_i}$**,** ενώ τα άλλα **0.** Υπολογίζω το εσωτερικό γινόμενο των labels misclassified * y με τον πίνακα **X,** αυτό είναι ισοδύναμο με την πρόσθεση $-\boldsymbol{y} \cdot \boldsymbol{x_i}$ για κάθε δείγμα που έκανα στην προηγούμενη υλοποίηση. Η υπόλοιπη λογική είναι όμοια με πριν.

Με αυτόν το τρόπο μείωσα τον χρόνο εκπαίδευσης από ✓ 11m 7.7s σε ✓ 6.3s ! Χωρίς αλλαγή στα αποτελέσματα. Λόγο του ότι δουλεύω σε μόνο δύο κλάσεις του cifar-10, δεν χρειάστηκε υλοποίηση batches για θέματα μνήμης.

Υλοποίηση LR scheduler

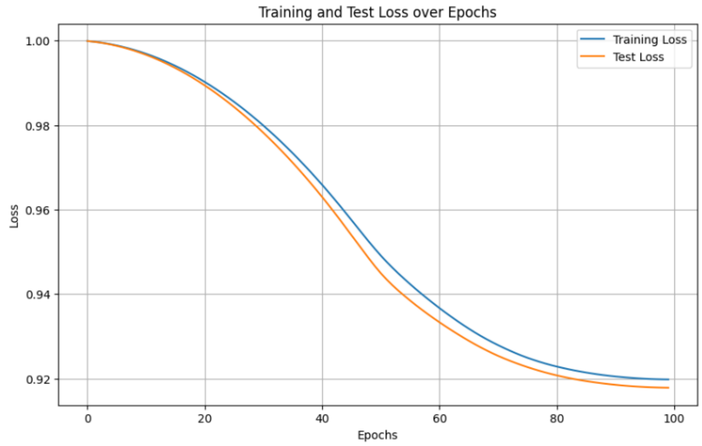
```
def up_then_down_scheduler(epoch, total_epochs, ramp_up_epochs, min_lr, max_lr):
    if epoch <= ramp_up_epochs:
        # Linear ramp-up
        lr = min_lr + (max_lr - min_lr) * (epoch / ramp_up_epochs)
    else:
        # Linear ramp-down
        lr = max_lr - (max_lr - min_lr) * ((epoch - ramp_up_epochs) / (total_epochs - ramp_up_epochs))
    return lr
```

- min_lr = 0.0001
- max_lr = 0.003
- ramp_up_epochs = 50
- total_epochs = 100
- reg_lambda = 0.001
- Training Accuracy: 56.74%
- Test Accuracy: 56.60%



Training Accuracy: 56.74%

Test Accuracy: 56.60%



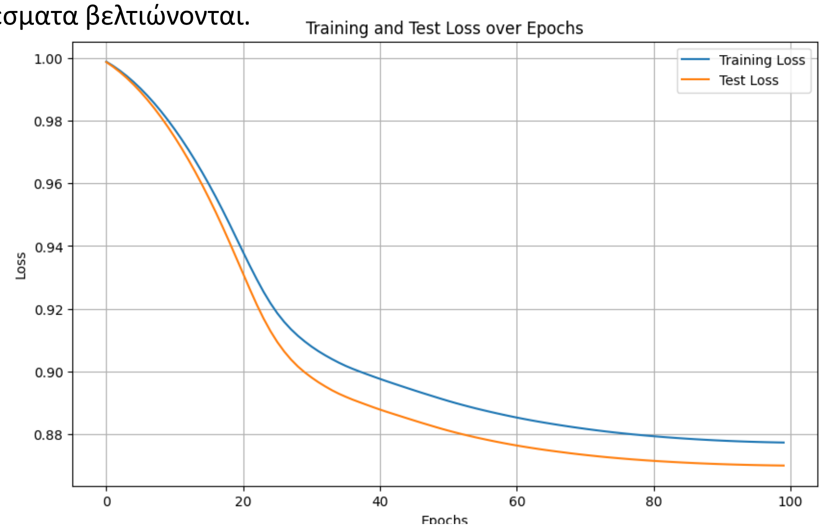
Προσθήκη bias & Standardization

Σύμφωνα με το [source](), το bias είναι σημαντικό στα SVMs, χωρίς αυτό ο classifier πάντα τέμνει την αρχή τον αξόνων. Τρέχοντας την νέα υλοποίηση παίρνω ακριβός τα παρόμοια αποτελέσματα. Αυτό σημαίνει πως τα δεδομένα μου είναι σχεδόν κεντραρισμένα. Σύμφωνα με το ChatGPT μπορώ να τυποποιήσω (standardize) τα αρχικά δεδομένα, δηλαδή να εξασφαλίζω πως τα αρχικά δεδομένα έχουν μέσο όρο 0 και τυπική απόκλιση 1. Όντος τα αποτελέσματα βελτιώνονται.

- min_lr = 0.0001
- max_lr = 0.001
- ramp_up_epochs = 50
- total_epochs = 100
- reg_lambda = 0.001

✓ 6.3s

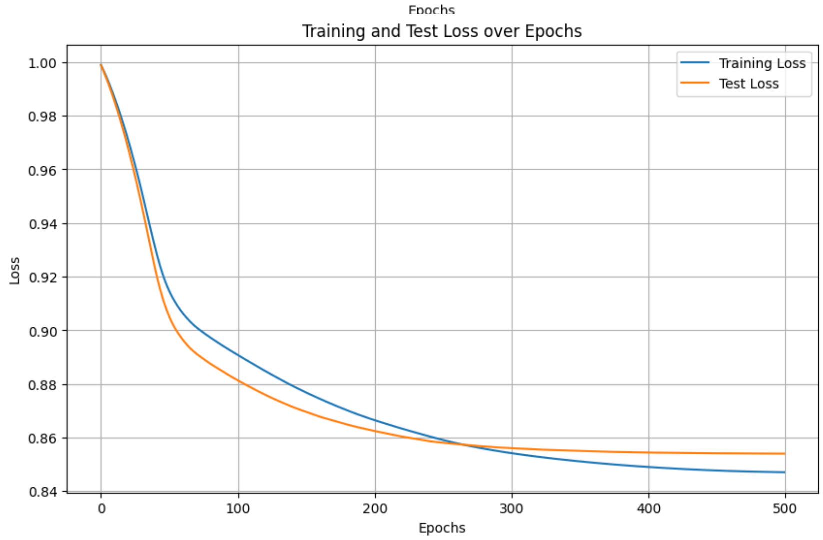
```
Training Accuracy: 60.18%
Test Accuracy: 60.05%
```



- min_lr = 0.0001
- max_lr = 0.001
- ramp_up_epochs = 250
- total_epochs = 500
- reg_lambda = 0.0001

✓ 32.7s

```
Training Accuracy: 61.98%
Test Accuracy: 61.75%
```



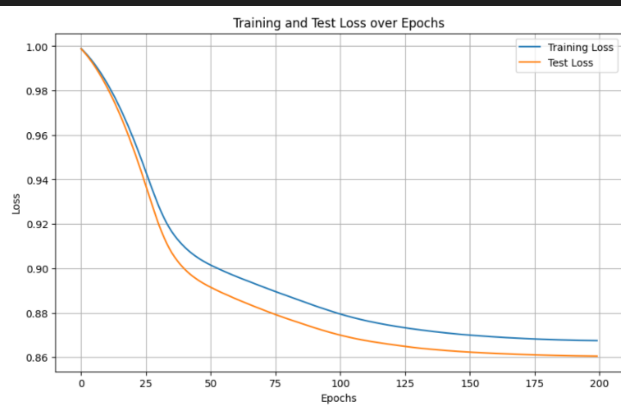
Προσθήκη Horizontal Flip

Θα εκμεταλλευτώ την γρήγορη σύγκλιση του μοντέλου, και θα εφαρμόσω horizontal flip σε **όλα** τα αρχικά μου δεδομένα, κρατώντας τα original. Ως αποτέλεσμα το δεδομένα εκπαίδευσης να έχουν διπλασιαστεί σε πλήθος, δηλαδή έχω όλα τα original και όλα τα original flipped. Αυτό θα προσφέρει περισσότερα δεδομένα ώστε το μοντέλο μου να «μάθει» καλύτερα

```
# Reshape back to image dimensions for augmentation
x_train_rh = x_train.reshape(-1, 32, 32, 3)
x_train_flipped = x_train_rh[:, :, ::-1, :]
# Combine original and flipped images
x_train_augmented = cp.vstack([x_train_rh, x_train_flipped])
# Duplicate labels for flipped images
y_train_augmented = cp.hstack([y_train, y_train])
# Reshape augmented training data back to flattened format
x_train = x_train_augmented.reshape(x_train_augmented.shape[0], -1)
y_train = y_train_augmented
```

- min_lr = 0.0001
- max_lr = 0.001
- ramp_up_epochs = 100
- total_epochs = 200
- reg_lambda = 0.0001

✓ 42.5s

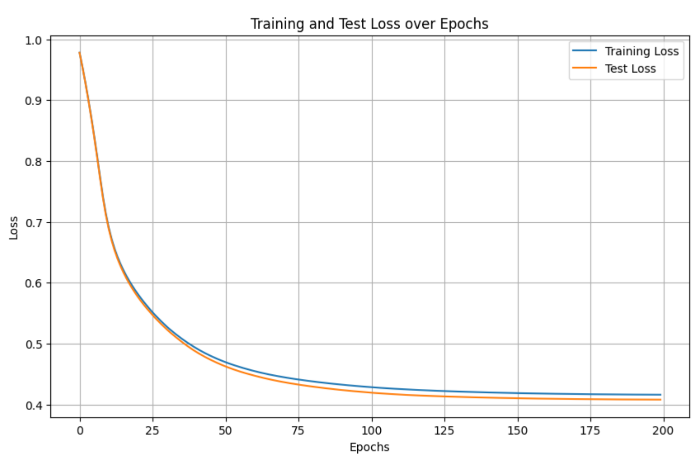



```
Training Accuracy: 60.53%
Test Accuracy: 60.55%
```

Για σκοπούς σύγκρισης τρέχω το μοντέλο μου για διαχωρισμό των κλάσεων «dog» και «airplane»

- min_lr = 0.0001
- max_lr = 0.001
- ramp_up_epochs = 100
- total_epochs = 200
- reg_lambda = 0.0001

✓ 41.8s




```
Training Accuracy: 84.06%
Test Accuracy: 83.90%
```

Λογικά αποτελέσματα, εφόσον οι κλάσεις «dog» και «cat» έχουν όμοια χαρακτηριστικά.

Προσθήκη Random Crop

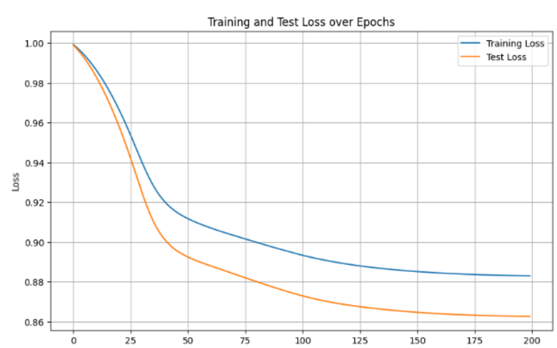
Όμοια με την **Horizontal Flip** δημιουργώ ένα ακόμη copy του original dataset και εφαρμόζω crop

```
def random_crop_with_padding(images, crop_size=(32, 32), padding=4):
    images = cp.asarray(images)
    # Get image dimensions
    size, height, width, channels = images.shape
    # Create a padded image array
    padded_height = height + 2 * padding
    padded_width = width + 2 * padding
    padded_images = cp.zeros((size, padded_height, padded_width, channels), dtype=images.dtype)
    # Copy original images into the center of the padded images
    padded_images[:, padding:padding+height, padding:padding+width, :] = images
    # Initialize an array to store cropped images
    cropped_images = cp.empty((size, *crop_size, channels), dtype=images.dtype)
    # Randomly crop each image
    for i in range(size):
        h_start = np.random.randint(0, 2 * padding + 1)
        w_start = np.random.randint(0, 2 * padding + 1)
        cropped_images[i] = padded_images[i, h_start:h_start+crop_size[0], w_start:w_start+crop_size[1],
:]

    return cropped_images
```

Παίρνω το πλήθος εικόνων **size** το **height**, **width** και τα **channels**. Υπολογίζω τις νέες διαστάσεις με το padding πάνω, κάτω, δεξια και αριστερά, και γεμίζω τον πίνακα με μηδενικά. Τοποθετώ τις original εικόνες στο κέντρο του padded πίνακα μου, έχοντας padding γύρω τους. Αρχικοποιώ τον **cropped_immages** με τις σωστές διαστάσεις. Μετά για κάθε εικόνα επιλέγονται τυχαίες αρχικές θέσεις **h_start** για ύψος και **w_start** για πλάτος, με περιορισμούς ώστε να διασφαλίσουμε ότι το crop δεν θα υπερβεί το ύψος της padded εικόνας. Από τη padded εικόνα γίνεται αποκοπή **cropped_images** ξεκινώντας από τις θέσεις **h_start** και **w_start** και με διαστάσεις που ορίζονται από το **crop_size**. Επιστρέφω τις cropped εικόνες.

- min_lr = 0.0001
- max_lr = 0.001
- ramp_up_epochs = 100
- total_epochs = 200
- reg_lambda = 0.0001

✓ 1m 9.3s

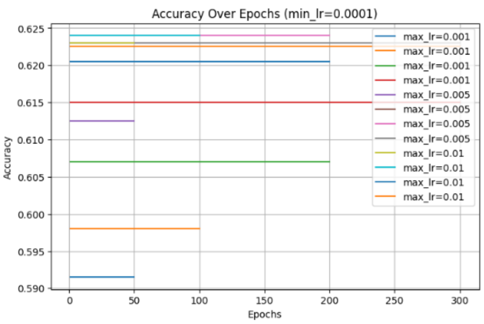


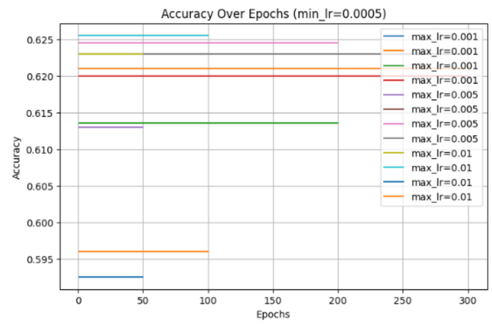

```
Training Accuracy: 59.52%
Test Accuracy: 60.45%
```

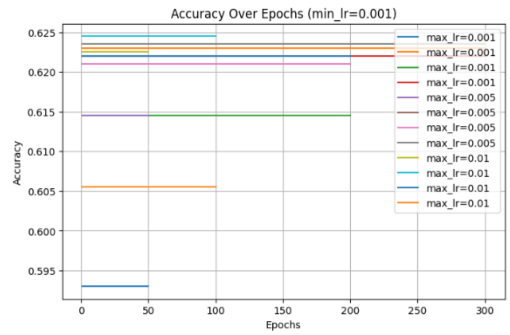
Hyperparameter Tuning

Τρέχω το μοντέλο για διαφορετικά **min_lr** & **max_lr**, για διαφορετικό αριθμό εποχών

Αποτελέσματα:





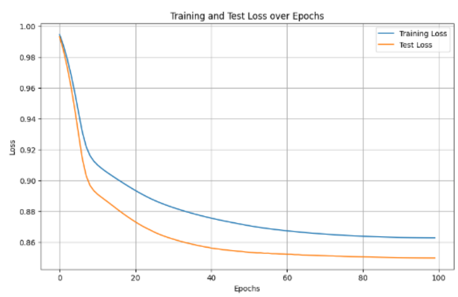


Βρίσκω πως οι καλύτερες Υπερπαραμέτροι είναι:

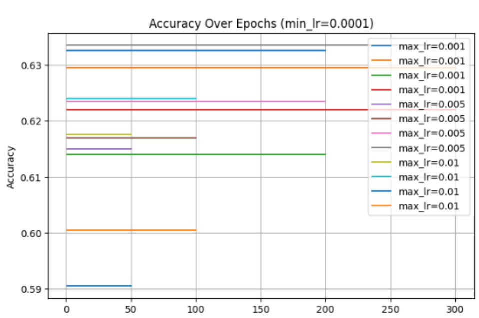
- min_lr = 0.0005
- max_lr = 0.01
- ramp_up_epochs = 50
- total_epochs = 100
- reg_lambda = 0.001

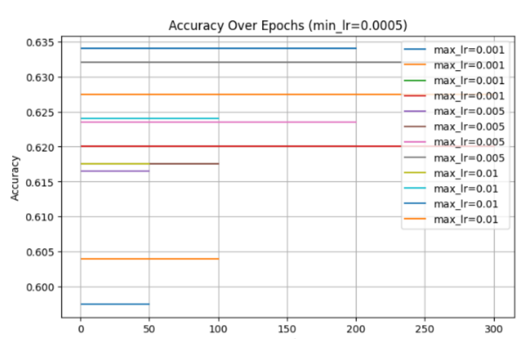
Τρέχοντας το μοντέλο με αποκλειστικά αυτές τις παραμέτρους :

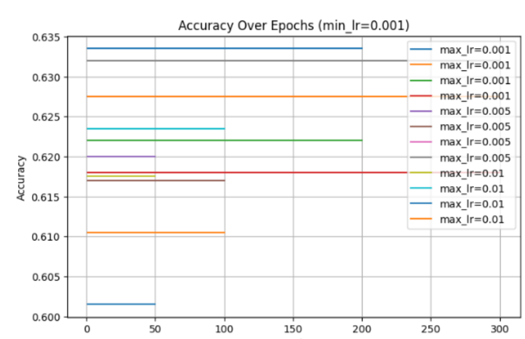
```
Training Accuracy: 61.10%
Test Accuracy: 62.45%
```



Θα ξανατρέξω το script του hyper parameter tuning αυτή την φορά χωρίς data augmentation/preprocessing, αφήνοντας μόνο το standardization



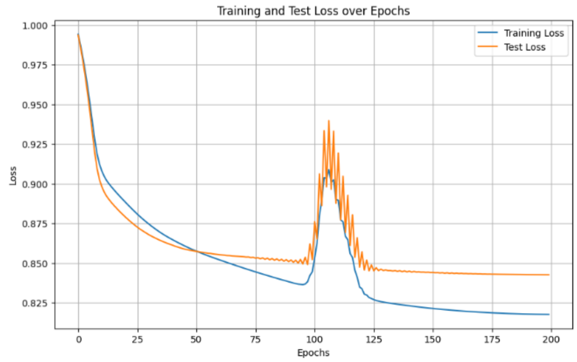




Βέλτιστες υπερπαραμέτροι:

✓ 12.6s

- min_lr = 0.0005
- max_lr = 0.01
- ramp_up_epochs = 100
- total_epochs = 200
- reg_lambda = 0.001




```
Training Accuracy: 63.69%
Test Accuracy: 63.40%
```

Καταλήγω πως το μοντέλο αποδίδει καλύτερα αλλά και γρηγορότερα, όταν εφαρμόζω **μόνο** standardization στην είσοδο.

Binary Classification with Polynomial Kernel

Θα υλοποιήσω Dual Form SVM και σύμφωνα με source

$$\underset{w,b}{minimize}\ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\zeta_i$$

$$such\ that\ y_i\left(w^T.X_i + b\right) \geq 1 - \zeta_i\ and\ \ \zeta_i \geq 0\quad For\ i = 1, 2, \cdots, n$$

«where zeta is the distance of a misclassified point from its correct hyperplane»

«If the value of C is very high then we try to minimize the number of misclassified points drastically which results in overfitting, and with decrease in value of C there will be underfitting»

$$\underset{\alpha}{maximize}\sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j\left(X_i^T.X_j\right)$$

$$s.t\quad 0 \leq \alpha_i \leq C\quad and\quad \sum_{i=1}^{n}\alpha_i y_i = 0$$

Συνάρτηση απόφασης

$$f(x) = \sum_{j=1}^{n}\alpha_j y_j K\left(x_j, x\right) + b$$

Με βάση το source2, η ενημέρωση του $\boldsymbol{a_j}$ στην εκπαίδευση βασίζετε εδώ:

$$\frac{\partial w}{\partial a_i} = 1 - y_i \sum_{j-1}^{n} a_j y_j K(x_i, x_j) = 1 - y_i f(x_i)$$

Για την ενημέρωση του $\boldsymbol{b}$ **,** source3, εύκολα βρίσκω ξεκινώντας από

$y_i f(x_i) = 1$, καταλήγω σε $b_i = y_i - \sum_{j=1}^{m} a_j y_j K(x_{j,x_i})$

Ορίζοντας το τελικό $\boldsymbol{b}$**,** ως τον μέσο όρο των $\boldsymbol{b_i}$.

```
def polynomial_kernel(X, Y, degree=3, coef0=1):
    return (cp.dot(X, Y.T) + coef0) ** degree
```

Υλοποιεί τον ορισμό $K(X, Y) = (X \cdot Y^T + c)^d$

```
class PolySVM:
```

Ορίζω κλαση

```
ef __init__(self, C=1.0, epochs=100, degree=3, coef0=1):
        self.kernel = polynomial_kernel
        self.C = C
        self.epochs = epochs
        self.degree = degree
        self.coef0 = coef0
        self.alphas = None
        self.b = 0
        self.K = None
        self.X = None
        self.y = None
        self.support_vectors = None
        self.support_alphas = None
        self.support_y = None
```

- Αρχικοποιώ τον πυρήνα με βάση `def polynomial_kernel` που ορίστηκε πριν.
- Αρχικοποιώ το **C**, που έχει οριστεί πάνω.
- **self.K,** αποθηκεύει τον πίνακα kernel, ουσιαστικά $\boldsymbol{K[i,j] = K(x_i, x_j)}$
- **self.support_vectors,** τα δεδομένα εισόδου $\boldsymbol{x_i}$ για τα οποία $\boldsymbol{a_i > 0}$ μετά την εκπαίδευση.
- **self.support_alphas,** Αποθηκεύει τις παραμέτρους $\boldsymbol{a_i}$ που αντιστοιχούν στα support vectors.
- **self.support_y**, Αποθηκεύει τις ετικέτες $\boldsymbol{y_i}$ που αντιστοιχούν στα support vectors.

```
def fit(self, X, y):
        self.X = X
        self.y = y
```

```
        m, n = X.shape
        self.alphas = cp.zeros(m)
        self.b = 0
        self.K = self.kernel(X, X, degree=self.degree, coef0=self.coef0)

        for epoch in tqdm(range(self.epochs), desc='Training Epochs'):
            for i in range(m):
                # Compute the decision function for xi
                f_i = cp.sum(self.alphas * self.y * self.K[:, i]) + self.b
                E_i = f_i - self.y[i]

                # Compute gradient w.r.t alpha_i
                gradient = 1 - self.y[i] * f_i

                # Update rule
                if (self.y[i] * f_i < 1):
                    # Gradient for alpha_i
                    grad_alpha_i = gradient  # Since dual objective is to maximize
                    # Update alpha_i
                    self.alphas[i] += self.lr * grad_alpha_i
                    # Clip alpha_i to [0, C]
                    self.alphas[i] = cp.clip(self.alphas[i], 0, self.C)
                else:
                    # If no update needed
                    continue

            # Update bias term b
            # Identify support vectors (0 < alpha_i < C)
            support_vector_indices = cp.where((self.alphas > 1e-5) & (self.alphas < self.C))[0]
            if len(support_vector_indices) > 0:
                # Compute the sum over all alpha_j * y_j * K(x_j, x_i) for each support vector i
                # Reshape alphas * y for broadcasting
                sum_alpha_y_K = cp.sum((self.alphas * self.y)[:, cp.newaxis] * self.K[:,
support_vector_indices], axis=0)

                # Compute b as the mean of (y_i - sum_alpha_y_K) over support vectors
                self.b = cp.mean(y[support_vector_indices] - sum_alpha_y_K)
            else:
                self.b = 0

        # Extract support vectors after training
        self.support_vector_indices = cp.where(self.alphas > 1e-5)[0]
        self.support_vectors = self.X[self.support_vector_indices]
        self.support_alphas = self.alphas[self.support_vector_indices]
        self.support_y = self.y[self.support_vector_indices]
```

1. Υπολογίζω τον **self.K** μία φορά στην αρχή για την χρήση του στον βρόχο
2. Αρχικοποιώ τα $\boldsymbol{\alpha_i}$ και το $\boldsymbol{b}$ με μηδενικά
3. Μπαίνω σε loop για όλα τα δείγματα $\boldsymbol{x_i}$ για κάθε εποχη
   a. Υπολογίζω το $\boldsymbol{f(x_i) = \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) + b}$
   b. Υπολογίζω το $\boldsymbol{1 - y_i f(x_i)}$ και ελέγχω αν είναι < 1
   c. Αν είναι < 1
      i. Ενημερώνω το $\boldsymbol{\alpha_j}$ με βάση την τιμή του $\boldsymbol{1 - y_i f(x_i)}$
      ii. και το περιορίζω στο διάστημα [0,C]
   d. Μετά την ενημέρωση όλων των $\boldsymbol{\alpha_j}$ σε μια εποχή γεμίζω το **support_vector_indices**
   e. Για αυτά τα διαστήματα ενημερώνω το $\boldsymbol{b}$
4. Γεμίζω όλα τα **support_vectors**

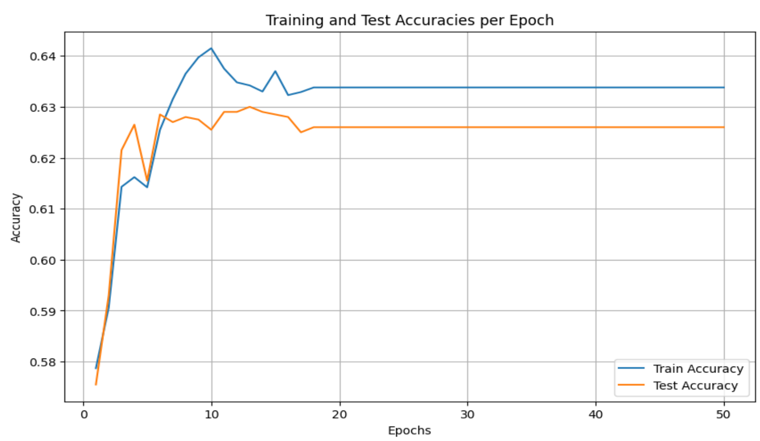
```
def decision_function(self, X):
        K = self.kernel(X, self.support_vectors, degree=self.degree, coef0=self.coef0)
        return cp.dot(K, self.support_alphas * self.support_y) + self.b

def predict(self, X):
       return cp.sign(self.decision_function(X))
```

1. Υπολογίζω το Kernel με βάση τα support_vectors δηλαδή $\boldsymbol{K(x_{test}, x_{support})}$.
2. Με την βοήθεια των άλλων support vectors υπολογίζεται η συνάρτηση απόφασης. $f(x_{test}) = \sum_{i=1}^{s} a_i y_i K(x_i, x_{test}) + b$

Παραδείγματα εκτέλεσης

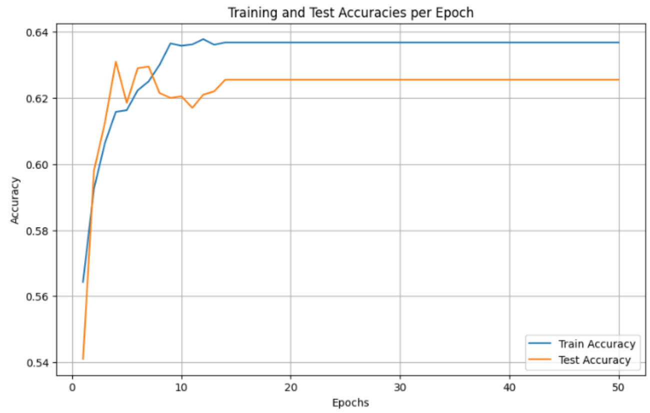
- C=3.0
- tol=1e-3
- epochs=50
- learning_rate=0.001
- degree=3
- coef0=1


```
Training Accuracy: 63.38%
Test Accuracy: 62.60%
```



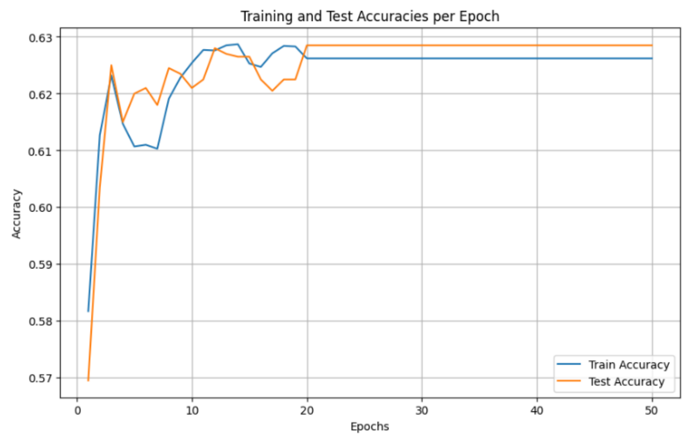
- C=3.0,
- tol=1e-3
- epochs=50
- learning_rate=0.001
- degree=4
- coef0=1

```
Training Accuracy: 63.68%
Test Accuracy: 62.55%
```



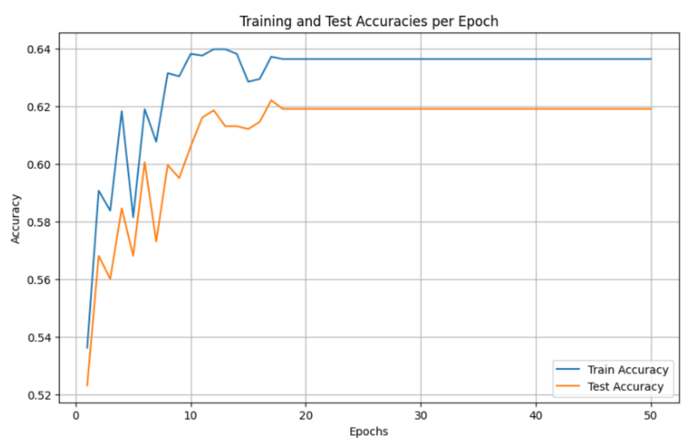
- C=1.0, ✓ 3m 58.7s
- tol=1e-3
- epochs=50
- learning_rate=0.01
- degree=3
- coef0=1

```
Training Accuracy: 62.62%
Test Accuracy: 62.85%
```

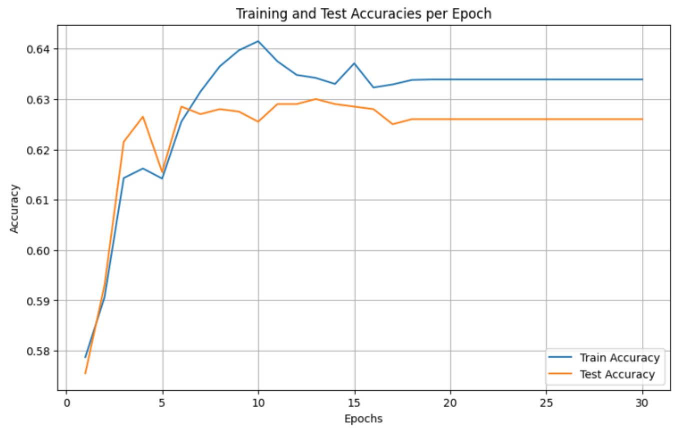


- C=0.01 ✓ 4m 12.8s
- tol=1e-3
- epochs=50
- learning_rate=0.01
- degree=3
- coef0=1

```
Training Accuracy: 63.63%
Test Accuracy: 61.90%
```



- C=10 ✓ 2m 22.5s
- tol=1e-3
- epochs=30
- learning_rate=0.001
- degree=3
- coef0=1

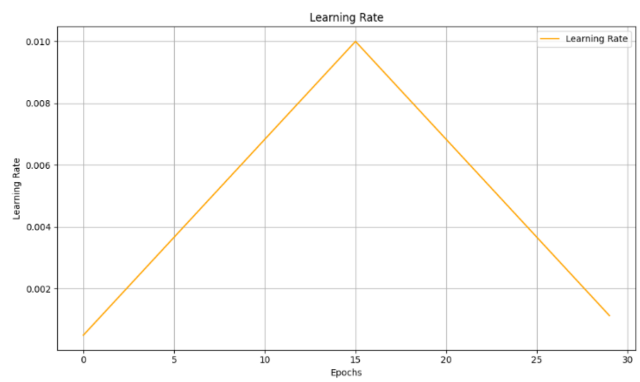


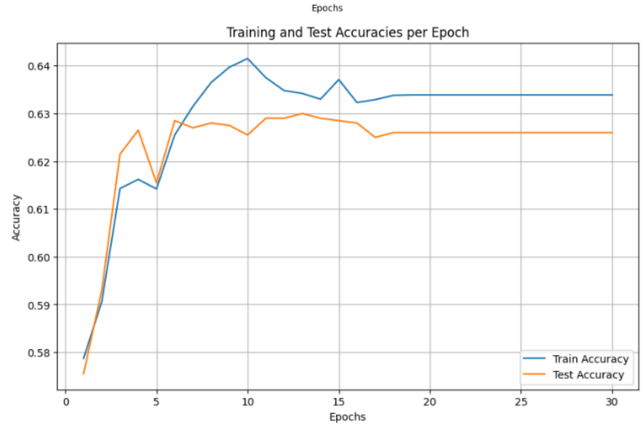

```
Training Accuracy: 63.39%
Test Accuracy: 62.60%
```

Προσθήκη Learning Rate Scheduler στο μοντέλο

Ο scheduler είναι ο ίδιος που χρησιμοποιήθηκε στο Linear μοντέλο

- C=10
- tol=1e-3
- epochs=30
- degree=3
- coef0=1
- min_lr = 0.0005
- max_lr = 0.01
- ramp_up_epochs = 15

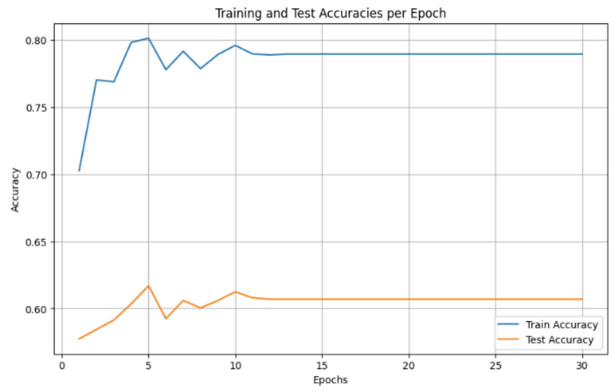





```
Training Accuracy: 63.39%
Test Accuracy: 62.60%
```

Προσθήκη Standardization στα αρχικά δεδομένα

- C=10, 2m 9.1s
- epochs=30,
- degree=3,
- coef0=1
- min_lr = 0.0005
- max_lr = 0.01
- ramp_up_epochs = 15




```
Training Accuracy: 78.93%
Test Accuracy: 60.70%
```

Hyperparameter Tuning

Έτρεξα το μοντέλο για τις εξής τιμές:

- `C_values = [1e-3, 1e-2, 1e-1, 1, 10, 100]`
- `degree_values = [2, 3, 4, 5]`
- `coef0_values = [0, 2.5, 5, 7.5, 10]`

Έτρεξα όλους τους πιθανούς συνδυασμούς αυτών των υπερπαραμέτρων για 30 εποχές και min_lr=0.0005, max_lr=0.001, ramp_up_epochs=15. Από τα 120 διαφορετικά μοντέλα τα top 5 είναι:

Top 5 Hyperparameter Combinations:

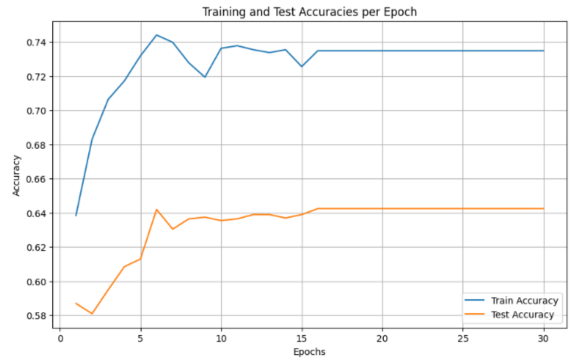
	C	Degree	coef0	Test Accuracy	Train Accuracy
0	10.000	2	7.5	0.6425	0.7349
1	0.001	2	5.0	0.6370	0.7380
2	100.000	2	7.5	0.6350	0.7060
3	0.100	2	2.5	0.6340	0.7426
4	1.000	2	5.0	0.6335	0.7371

Best Hyperparameters:
C=10.0, Degree=2, coef0=7.5

238m 2.8s

Τρέχοντας το καλύτερο ξεχωριστά επιβεβαιώνω πως:

- C=10, ✓ 2m 0.9s
- epochs=30,
- degree=2,
- coef0=7.5
- min_lr = 0.0005
- max_lr = 0.001
- ramp_up_epochs = 15




```
Training Accuracy: 73.49%
Test Accuracy: 64.25%
```

SMO approach

Binary Classification With RBF kernel Using SMO

Αλλάζοντας το Kernel function σε RBF της προηγούμενης υλοποίησης (polynomial kernel), εμφάνισε προβλήματα, με αποτέλεσμα να πρέπει να αλλάξω την λογική της κλάσης. Με την βοήθεια του paper, αλλάζω την υλοποίηση μου σε SMO

```
def __init__(self, C=1.0, tol=1e-3, max_passes=5, kernel='rbf', gamma=0.05):
        self.C = C
        self.tol = tol
        self.max_passes = max_passes
        self.kernel = kernel
        self.gamma = gamma
        self.alphas = None
        self.b = 0
        self.X = None
        self.y = None
        self.K = None  # Kernel matrix
```

Initialization των παραμέτρων του RBF

```
def compute_kernel_matrix(self, X):
        if self.kernel == 'rbf':
            # Efficient RBF Kernel computation
            X_norm = cp.sum(X ** 2, axis=1).reshape(-1, 1)
            K = cp.exp(-self.gamma * (X_norm + X_norm.T - 2 * cp.dot(X, X.T)))
            return K
```

Ορισμός του Kernel σε RBF

```
def fit(self, X, y, x_test=None, y_test=None):
        self.X = X
        self.y = y
        m, n = X.shape
        self.alphas = cp.zeros(m)
        self.b = 0
        self.K = self.compute_kernel_matrix(X)

        passes = 0
        iter = 0
        max_iter = 1000
        train_accuracies = []
        test_accuracies = []
        while passes < self.max_passes and iter < max_iter:
            num_changed_alphas = 0
            for i in range(m):
                E_i = self.decision_function_single(i) - y[i]
```

```
                # Check if example violates KKT conditions
                if ( (y[i]*E_i < -self.tol and self.alphas[i] < self.C) or
                     (y[i]*E_i > self.tol and self.alphas[i] > 0) ):

                    # Select j != i randomly
                    j = cp.random.randint(0, m)
                    while j == i:
                        j = cp.random.randint(0, m)

                    E_j = self.decision_function_single(j) - y[j]

                    alpha_i_old = self.alphas[i].copy()
                    alpha_j_old = self.alphas[j].copy()

                    # Compute L and H
                    if y[i] != y[j]:
                        L = max(0, self.alphas[j] - self.alphas[i])
                        H = min(self.C, self.C + self.alphas[j] - self.alphas[i])
                    else:
                        L = max(0, self.alphas[i] + self.alphas[j] - self.C)
                        H = min(self.C, self.alphas[i] + self.alphas[j])
                    if L == H:
                        continue

                    # Compute eta
                    eta = self.K[i,i] + self.K[j,j] - 2.0 * self.K[i,j]
                    if eta <= 0:
                        continue

                    # Update alpha_j
                    self.alphas[j] += y[j] * (E_i - E_j) / eta
                    # Clip alpha_j
                    self.alphas[j] = cp.clip(self.alphas[j], L, H)

                    if cp.abs(self.alphas[j] - alpha_j_old) < 1e-5:
                        continue

                    # Update alpha_i
                    self.alphas[i] += y[i] * y[j] * (alpha_j_old - self.alphas[j])

                    # Compute b1 and b2
                    b1 = self.b - E_i - y[i]*(self.alphas[i] - alpha_i_old)*self.K[i,i] -
y[j]*(self.alphas[j] - alpha_j_old)*self.K[i,j]
                    b2 = self.b - E_j - y[i]*(self.alphas[i] - alpha_i_old)*self.K[i,j] -
y[j]*(self.alphas[j] - alpha_j_old)*self.K[j,j]
```

```
                # Update b
                if 0 < self.alphas[i] < self.C:
                    self.b = b1
                elif 0 < self.alphas[j] < self.C:
                    self.b = b2
                else:
                    self.b = (b1 + b2) / 2.0

                num_changed_alphas += 1

        if num_changed_alphas == 0:
            passes += 1
        else:
            passes = 0
        iter += 1
```

- Υπολογίζω το kernel matrix με **compute_kernel_matrix**
- Μέσα στην λούπα για κάθε στοιχείο υπολογίζω το σφάλμα **Ei=f(xi)−yi** οπού **f(xi)** συνάρτηση απόφασης του στοιχείου i.
- Γίνετε έλεγχος των συνθηκών KTT
- Επιλέγω ένα τυχαίο δείγμα j επιλέγεται ώστε j≠i
- Υπολογίζω το **Ej=f(xj)−yj**
- Υπολογίζω τα όρια L και H, όπως αναφέρονται στο paper
- Υπολογίζω το **η** και αν πληρεί τις προϋποθέσεις συνεχίζω στο επόμενο **i,** αν όχι συνεχίζω
- Ενημερώνω το **aj** περιορίζοντας το σε [L,H], με βάση αυτό ενημερώνω και το **ai**
- Ενημερώνω και το b με βάση την μεθοδολογία του paper.
- Αν δεν ενημερωθεί κανένα **ai** για 5 iteration τότε τερματίζω.

Για να βρω βέλτιστες υπερπαραμέτρους, έτρεξα το SVC classifier του scikit-learn με GridSearchCV με 5-fold cross-validation για:
C: [0.2, 0.3, 0.4, 0.5] , gamma: [gamma, gamma_withVar], Σε υποσύνολο των αρχικών δεδομένων.

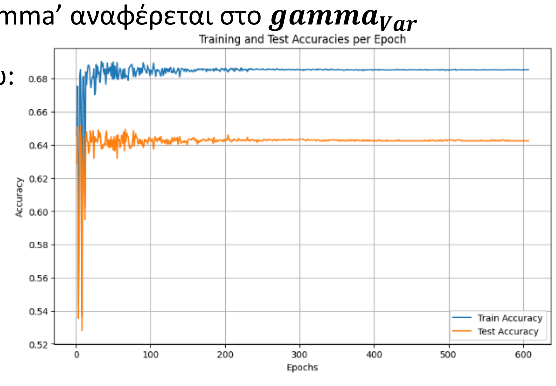
$$gamma = \frac{1}{num_{samples}} \qquad gamma_{Var} = \frac{1}{Var(x_{test})num_{samples}},$$

<u>Αποτελέσματα:</u>

Best Parameters: {'C': 0.3, 'gamma': 0.00033039790870810l4}, το ‘gamma’ αναφέρεται στο $\boldsymbol{gamma_{Var}}$

Τρέχοντας την δική μου υλοποίηση με αυτές τις παραμέτρους παίρνω:
`Training Accuracy: 68.54%`

`Test Accuracy: 64.25%` **`execution time: 102mins`**



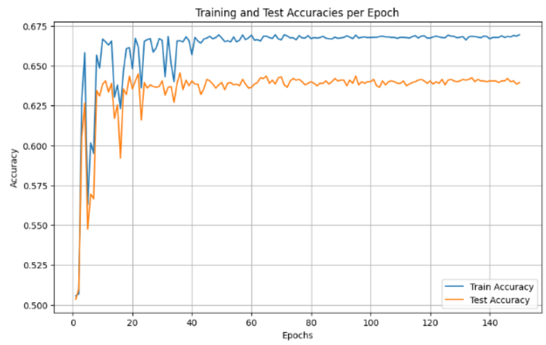
Με βάση του γραφήματος φαίνεται πως μετά το 150-200 iteration το accuracy σταθεροποιείτε. Λόγο χρόνου, το μοντέλο μου θα γίνεται trained σε 150 iterations

Ξαναέτρεξα το GridSearchCV για τις παραμέτρους:
C: [0.01, 0.1, 0.2, 0.5, 1, 2, 5, 10], **gamma**: [gamma, gamma_withVar, 0.0001, 0.001, 0.01, 0.1, 1, 10]

Best Parameters: {'C': 0.5, 'gamma': 0.0001, 'kernel': 'rbf'}

Ξανατρέχω το δικό μου μοντέλο σε όλα τα δεδομένα για 150 iterations:

`Training Accuracy: 66.95%`
`Test Accuracy: 63.95%`

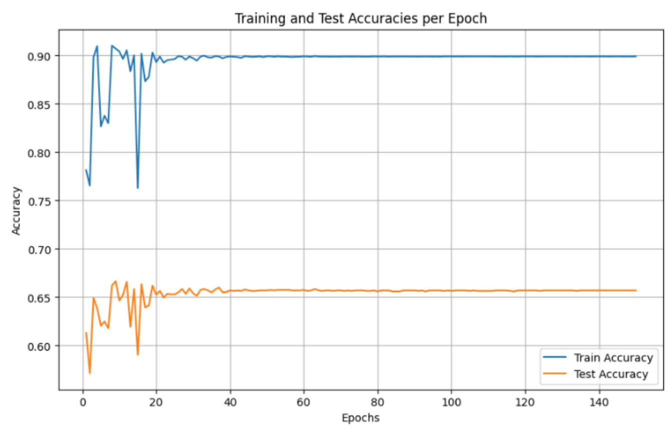


<u>Υπερπαραμέτροι</u>

- C=0.5
- gamma=0.001

34m 28.9s

`Training Accuracy: 89.91%`
`Test Accuracy: 65.70%`

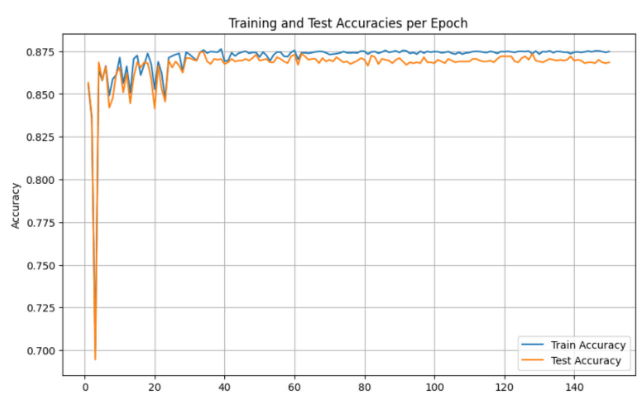


Αυξάνοντας το gamma παρακολουθείται overfitting, με ελάχιστη βελτίωση στο **`Test Accuracy`**

Ας Τρέξουμε το μοντέλο για διαχωρισμό κλάσεων των «airplane» και «cat», για σύγκριση

`Training Accuracy: 87.48%`
`Test Accuracy: 86.85%`

33m 24.3s

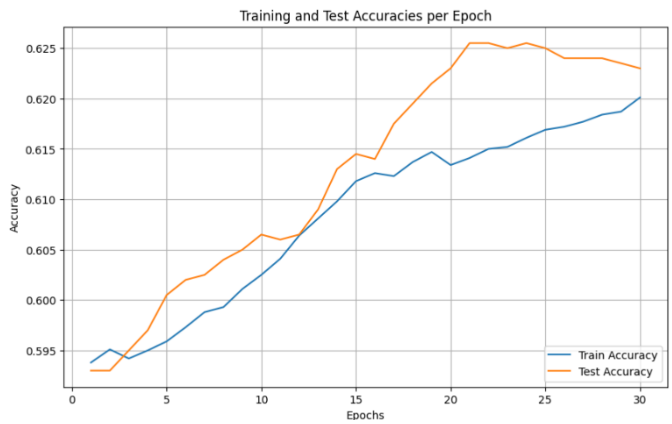


Πίσω σε Dual form SVM

Όπως είχα προαναφέρει, το RBF kernel παρουσίαζε προβλήματα στο dual form implementation που είχα υλοποιήσει στο polynomial. Τελικά το gamma που έδινα ως είσοδο, δεν ήταν αρκετά χαμηλό και το μοντέλο δεν «μάθαινε». Ακολουθούν αποτελέσματα με αυτήν την μέθοδο με RBF kernel:

- C=0.5 

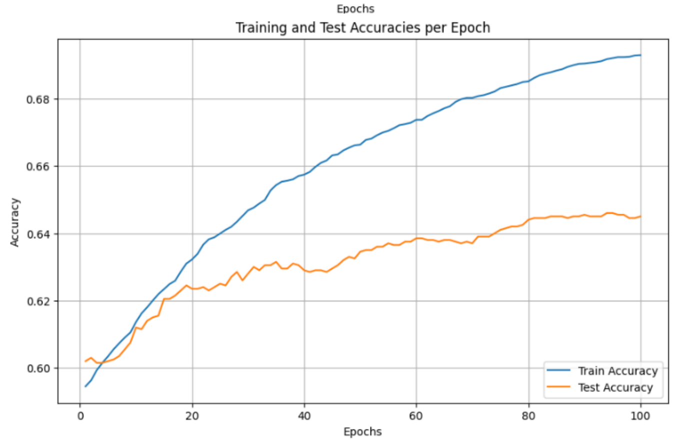
- epochs=30
- min_lr=0.0005
- max_lr=0.001
- ramp_up_epochs=15
- gamma= 0.0001



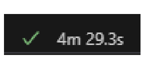
Training Accuracy: 62.01%
Test Accuracy: 62.30%

- C=0.5 

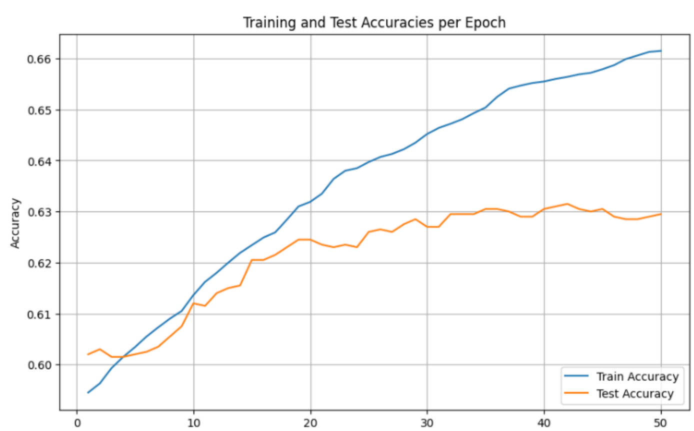
- epochs=100
- min_lr=0.0005
- max_lr=0.001
- ramp_up_epochs=15
- gamma= $\frac{1}{Var(x_{test})num_{samples}}$



Training Accuracy: 69.29%
Test Accuracy: 64.50%

- C=0.5 

- epochs=50
- min_lr=0.0005
- max_lr=0.001
- ramp_up_epochs=15
- gamma= $\frac{1}{num_{samples}}$



Training Accuracy: 66.15%

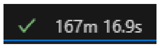
Test Accuracy: 62.95%

Συνοψίζοντας, έχω δημιουργήσει:

1. Primal form linear kernel με SGD
2. Dual form polynomial kernel με Gradient Ascent
3. Dual form RBF kernel με Gradient Ascent
4. SMO με RBF kernel

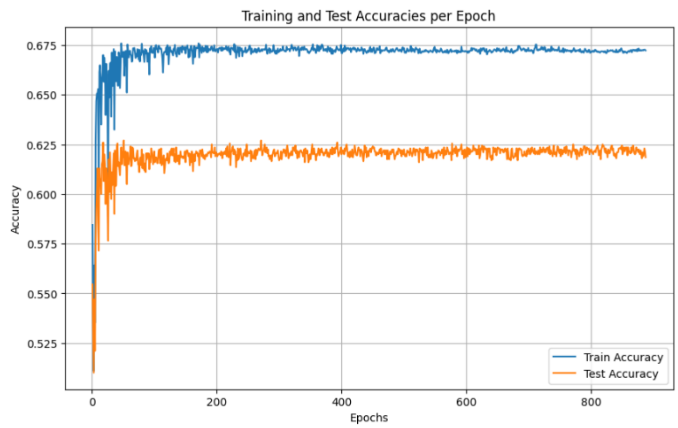
Θα υλοποιήσω επίσης SMO μοντέλο για Linear και Polynomial kernel, απλός αλλάζοντας την συνάρτηση του kernel.

SMO with Linear Kernel

- C=0.001 

- tol=1e-5

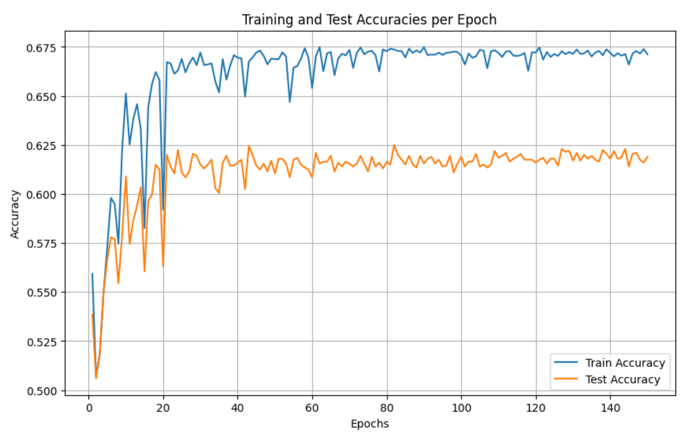
Training Accuracy: 67.23%
Test Accuracy: 61.85%



Λόγο χρόνου σύγκλισης, Η εκπαίδευση τερματίζει στα 150 iterations

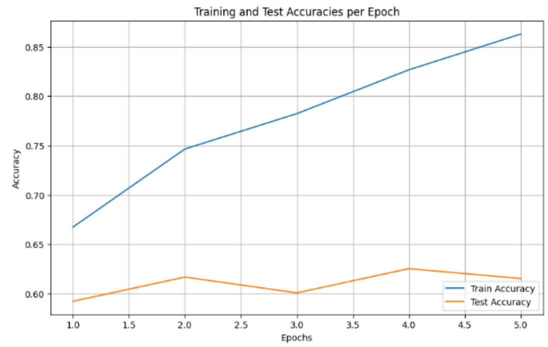
- C=0.001
- Tol=1e-3

Training Accuracy: 67.12%
Test Accuracy: 61.90%



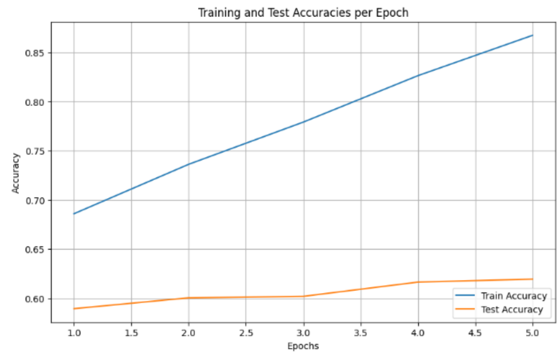
SMO with Polynomial Kernel

- C=1.0 ✓ 3m 21.0s
- tol=1e-3
- degree=3
- coef0=1



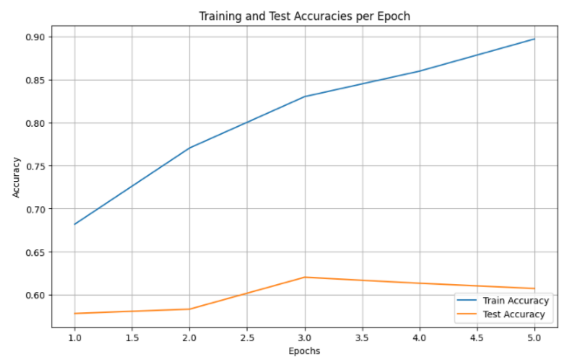
Training Accuracy: 86.28%
Test Accuracy: 61.55%

- C=1.0 ✓ 3m 45.4s
- tol=1e-5
- degree=3
- coef0=1



Training Accuracy: 86.73%
Test Accuracy: 61.95%

- C=1.0 ✓ 3m 23.5s
- tol=1e-3
- degree=4
- coef0=1



Training Accuracy: 89.69%
Test Accuracy: 60.75%

MLP Model

Υλοποίηση MLP (pytorch) με ένα κρυφό επίπεδο, loss function Hinge loss με SGD.

Ακολουθώ το ίδιο preprocessing στα αρχικά δεδομένα όπως τα προηγούμενα μοντέλα.

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MLP, self).__init__()
```

```
        self.hidden_layer = nn.Linear(input_size, hidden_size)
        self.output_layer = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = torch.relu(self.hidden_layer(x))
        x = self.output_layer(x)
        return x
```

Στο **`def __init__`** γίνετε ο ορισμός των στρωμάτων του δικτύου

- **self.hidden_layer = nn.Linear(input_size, hidden_size)**: Δημιουργεί ένα fully connected layer που παίρνει δεδομένα με μέγεθος input_size και τα μετασχηματίζει σε διαστάσεις hidden_size. Αυτό είναι το κρυφό επίπεδο.
- **self.output_layer = nn.Linear(hidden_size, 1)**: Δημιουργεί ένα γραμμικό επίπεδο που παίρνει δεδομένα με μέγεθος hidden_size και τα μετασχηματίζει σε έξοδο.

Στο **`def forward,`** ορίζω το forward pass

- **x = torch.relu(self.hidden_layer(x))**: Τα δεδομένα περνούν από το hidden_layer και εφαρμόζεται η συνάρτηση ενεργοποίησης ReLU
- **x = self.output_layer(x)**: Το αποτέλεσμα από το προηγούμενο βήμα περνάει από το output_layer για να παραχθεί η τελική πρόβλεψη.

```
input_size = x_train.shape[1]
hidden_size = 256
model = MLP(input_size, hidden_size)
```

Εδώ ορίζω των αριθμό των νευρώνων στο κρυφό επίπεδο, και αρχικοποιώ το μοντέλο μου

```
class HingeLoss(nn.Module):
    def __init__(self):
        super(HingeLoss, self).__init__()

    def forward(self, outputs, labels):
        outputs = outputs.view(-1)
        labels = labels.view(-1)
        loss = torch.mean(torch.clamp(1 - outputs * labels, min=0))
        return loss
```

Ορίζω το Loss function

- def forward(self, outputs, labels): Αναδιαμορφώνω τα outputs και τα labels σε μονοδιάστατο πίνακα, και με βάση τον ορισμό του Hinge Loss επιστρέφω τον μέσο όρο.

```
criterion = HingeLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

Ορίζω την συνάρτηση του Loss Function. Όπως και τον optimizer που χρησιμοποιεί SGD με σταθερό Learning rate 0.001

```
num_epochs = 70
train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * inputs.size(0)
        predictions = torch.sign(outputs).view(-1)
        correct += (predictions == labels.view(-1)).sum().item()
        total += labels.size(0)

    avg_loss = total_loss / total
    accuracy = correct / total
    train_losses.append(avg_loss)
    train_accuracies.append(accuracy)

    # Evaluate on test data
    model.eval()
    total_loss_test = 0
    correct_test = 0
    total_test = 0


    for inputs_test, labels_test in test_loader:
        outputs_test = model(inputs_test)
```

```
            loss_test = criterion(outputs_test, labels_test)
            total_loss_test += loss_test.item() * inputs_test.size(0)
            predictions_test = torch.sign(outputs_test).view(-1)
            correct_test += (predictions_test == labels_test.view(-1)).sum().item()
            total_test += labels_test.size(0)

    avg_loss_test = total_loss_test / total_test
    accuracy_test = correct_test / total_test
    test_losses.append(avg_loss_test)
    test_accuracies.append(accuracy_test)

    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_loss:.4f}, Train Acc: {accuracy*100:.2f}%, Test Loss: {avg_loss_test:.4f}, Test Acc: {accuracy_test*100:.2f}%")
```

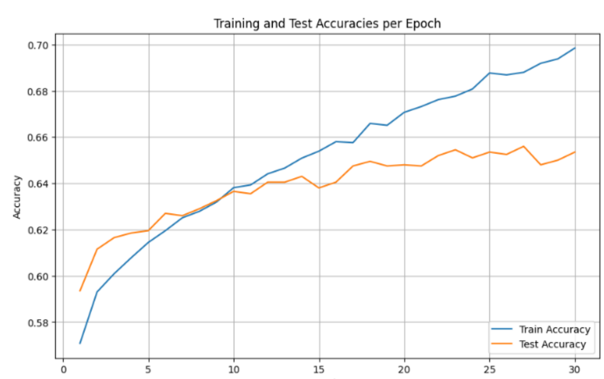
Μετά τον ορισμό τον εποχών και την αρχικοποίηση των λιστών, μπαίνω στον βρόχο εκπαίδευσης

1. Βάζω το μοντέλο σε λειτουργεία εκπαίδευσης
2. Εσωτερικός βρόχος
    a. Περνάω τα δεδομένα εισόδου και τις ετικέτες (labels) από τον train_loader.
    b. Μηδενίζω τα gradients από το προηγούμενο βήμα.
    c. Υπολογίζω τις προβλέψεις του μοντέλου.
    d. Υπολογίζω την απώλεια χρησιμοποιώντας Hinge Loss που ορίστηκε πριν.
    e. Υπολογίζω τους βαθμούς για την ενημέρωση των παραμέτρων.
    f. Ενημερώνω τις παραμέτρους του μοντέλου σύμφωνα με τον optimizer.
3. Συγκεντρώνω την απώλεια για όλα τα δείγματα στο batch.
4. Υπολογίζω τις προβλέψεις.
5. Υπολογίζω πόσες σωστές προβλέψεις έγιναν, την μέση απώλεια και την ακρίβεια.
6. Θέτω το μοντέλο σε λειτουργεία αξιολόγησης.
7. Συλλέγονται οι μέσες τιμές απώλειας και ακρίβειας για τα test data.

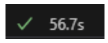
Παραδείγματα εκτέλεσης MLP

- lr=0.001 ✓ 23.9s
- hidden_size = 256
- num_epochs = 30

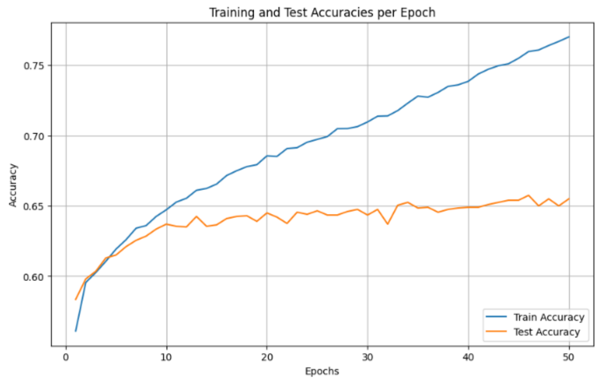
Final Training Accuracy: 70.19%
Final Test Accuracy: 65.35%



- lr=0.001
- hidden_size = 512
- num_epochs = 50



Final Training Accuracy: 77.42%
Final Test Accuracy: 65.50%



Hyperparameter Tuning for MLP

Εφαρμόζω τεχνική Random Search στις τιμές:

- learning_rate: [0.01, 0.001, 0.0001],
- hidden_size: [128, 256, 512],
- batch_size: [32, 64, 128],
- optimizer: ['SGD', 'Adam'],
- num_epochs: [30]

<u>Αποτελέσματα:</u>

```
Top 5 Hyperparameter Combinations:
Rank 1: Test Accuracy: 65.85% with parameters: {'learning_rate': 0.0001,
'hidden_size': 512, 'batch_size': 128, 'optimizer': 'Adam', 'num_epochs': 30}
Rank 2: Test Accuracy: 65.05% with parameters: {'learning_rate': 0.0001,
'hidden_size': 512, 'batch_size': 128, 'optimizer': 'Adam', 'num_epochs': 30}
Rank 3: Test Accuracy: 64.65% with parameters: {'learning_rate': 0.0001,
'hidden_size': 512, 'batch_size': 128, 'optimizer': 'Adam', 'num_epochs': 30}
Rank 4: Test Accuracy: 64.65% with parameters: {'learning_rate': 0.01,
'hidden_size': 128, 'batch_size': 64, 'optimizer': 'SGD', 'num_epochs': 30}
Rank 5: Test Accuracy: 64.30% with parameters: {'learning_rate': 0.0001,
'hidden_size': 512, 'batch_size': 32, 'optimizer': 'Adam', 'num_epochs': 30}
```

KNN and NCC

Χρησιμοποιώντας τα KNeighborsClassifier και NearestCentroid της sklrearn εξετάζω τα αποτελέσματα για διαχωρισμό κλάσεων «cat» και «dog»,

KNN

K-NN 1 Neighbour Test Accuracy: 57.85%

K-NN 3 Neighbours Test Accuracy: 57.50%

NCC

Nearest Centroid Test Accuracy: 57.85%

Σύγκριση των Μοντέλων

Ακολουθούν όλα τα μοντέλα SVM που υλοποιηθήκαν με το καλύτερο αποτέλεσμα τους στο test set.

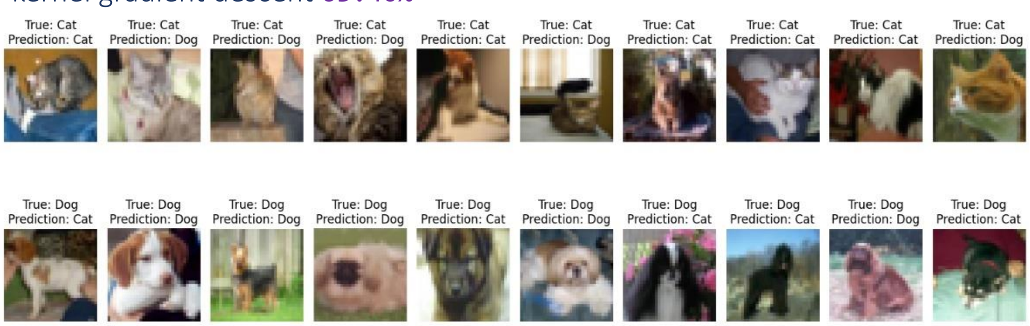
1. Linear kernel gradient descent 63.40% (with hyperparameter tuning)
2. Linear kernel SMO 61.90%
3. Polynomial kernel gradient ascent 64.25% (with hyperparameter tuning)
4. Polynomial kernel SMO 61.95%
5. RBF kernel gradient ascent 64.50%
6. RBF kerel SMO 65.70% (with hyperparameter tuning)
7. MLP 65.85% (with hyperparameter tuning)
8. K-NN 1 57.85%
9. K-NN 3 57.50%
10. NCC 57.85%

Αυτά τα αποτελέσματα δείχνουν πως οι κλάσεις «cat» και «dog», έχουν αρκετά όμοια χαρακτηριστικά και είναι αρκετά δύσκολα στην ομαδοποίηση. Επίσης, αποδείξαμε ότι οι μη γραμμικές μέθοδοι και τα νευρωνικά δίκτυα υπερτερούν στο δεδομένο πρόβλημα, αλλά όχι σε σημαντικό βαθμό όπως θα περίμενε κανείς.

Παραδείγματα

Εδώ θα κάνουμε μια ανακεφαλαίωση δίνοντας επίσης παραδείγματα κατηγοριοποίησης «cat» και «dog». Συγκρίνοντας το accuracy με τον διαχωρισμό «airplane» με «cat» με ίδιες υπερπαραμέτρους

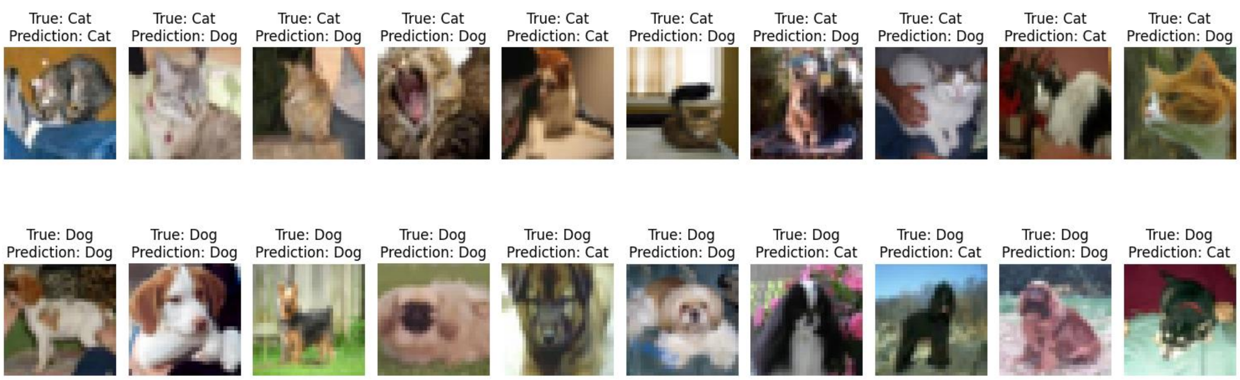
Linear kernel gradient descent 63.40%



Airplane-Cat classification

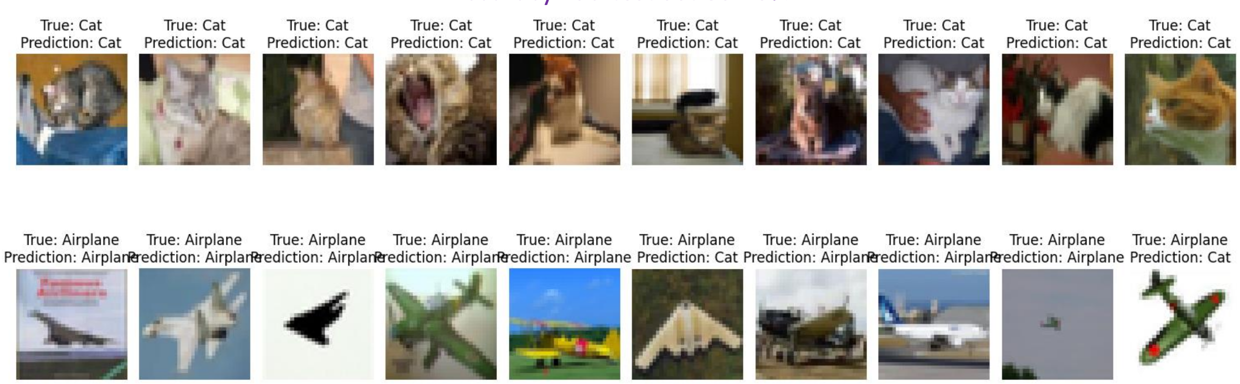
Accuracy του test set 84.45%

Linear kernel SMO 61.90%



Airplane-Cat classification

Accuracy του test set 83.75%

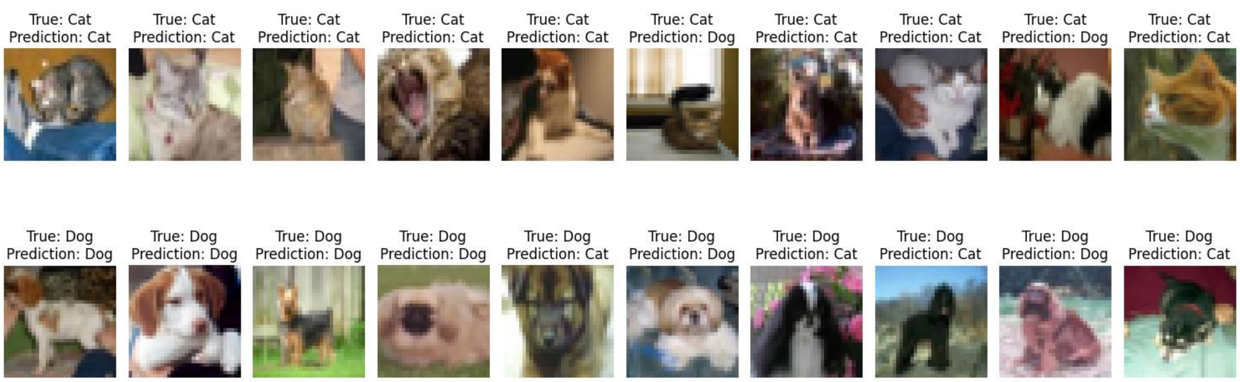


Polynomial kernel gradient ascent 64.25%

Airplane-Cat classification

Accuracy του test set 76.45%

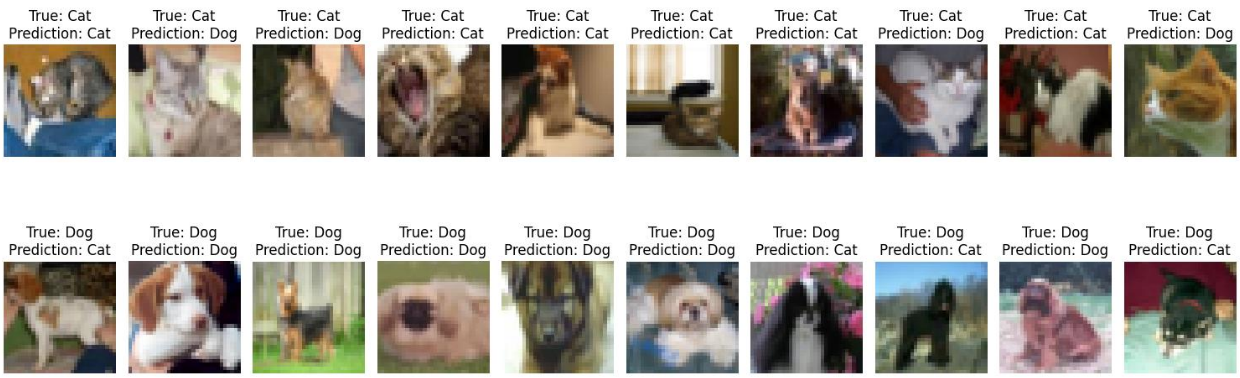
Polynomial kernel SMO 61.95%



Airplane-Cat classification

Accuracy του test set 82.95%

RBF kernel gradient ascent 64.50%



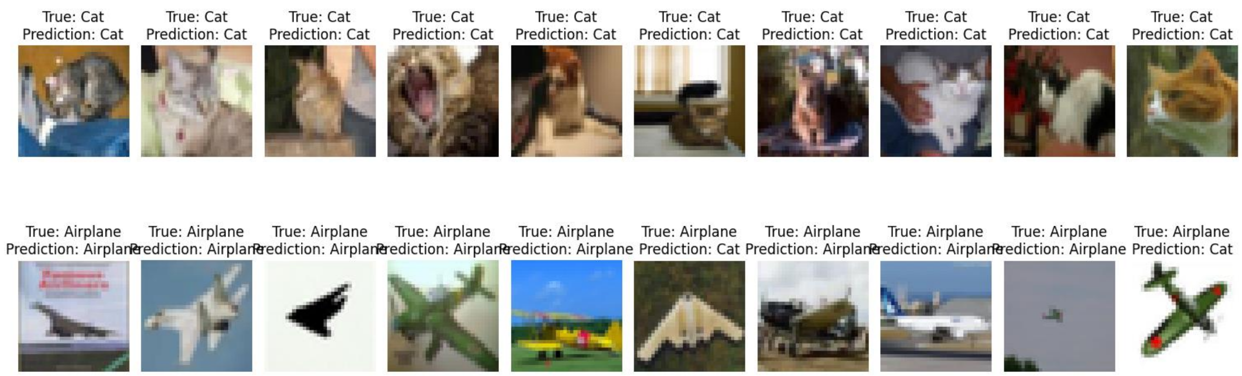
Airplane-Cat classification

Accuracy του test set 84.90%

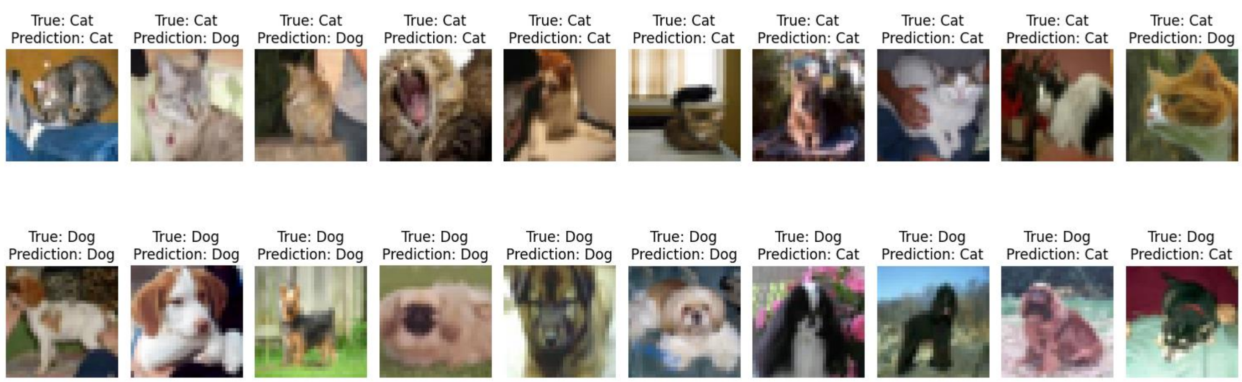
RBF kerel SMO 65.70%

Airplane-Cat classification

Accuracy του test set 86.90%



MLP 65.85%



Airplane-Cat classification

Accuracy του test set 89.65%

K-NN 1/3 57.85%/57.50%

Airplane-Cat classification: KNN 1/3 83.50%/82.95%

NCC 57.85%

Airplane-Cat classification: 74.45%