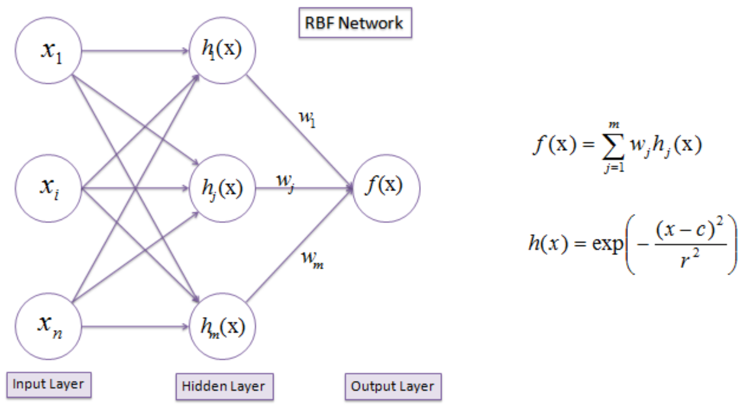
Radial Basis Function Neural Network

Comparison between RBF nn, k-nearest Neighbour, and Nearest Class Centroid

Iasonas Kakandris



A report presented for the course of
Neural Networks: Deep learning

Aristotle University of Thessaloniki
Faculty of Electrical and Computer Engineering
5/1/2024

Contents

1 Introduction

The goal of this project is to create a Radial basis function neural network and test its ability to categorize photos from the CIFAR-10 dataset. More specifically we will create 2 implementations of the RBF neural network and we will train them with the 50,000 training dataset of the CIFAR-10.In the end, we will compare the accuracy of the RBF mentioned above nns with the accuracy of the K-Nearest Neighbor and Centroid Neighbor algorithms to find the best results.

The understanding of the theory as well as part of the code was based on [Sai] and [Zog]

2 Theory

To fully understand the code implementation used, we need to understand the theory behind the RBF nn.

A Radial Basis Function Neural Network (RBF NN) is an artificial neural network that uses radial basis functions as activation functions. The network is typically composed of three layers: an input layer, a hidden layer with radial basis function neurons, and an output layer.

2.1 Architecture

2.1.1 Input Layer

The input layer receives the input features, denoted as

$$X = [x_1, x_2, ..., x_n]$$

where n is the number of input features

2.1.2 Hidden Layer

The hidden layer consists of radial basis function neurons. Each neuron calculates its output based on the Euclidean distance between the input and a center, and this output is passed through a radial basis function. The output h_i of the i-th hidden neuron is calculated as follows:

$$h_i(X) = \exp(-\frac{\|X - C_i\|^2}{2\sigma_i^2})$$

where C_i is the center of the i-th neuron and σ_i is the width of the Gaussian function (controls the spread of the function

2.1.3 Output Layer

The output layer combines the outputs of the hidden layer to produce the final output. The output y_k of the k-th output neuron is calculated as a weighted sum of the hidden layer outputs:

$$y_k = \sum_{i=1}^{M} w_{ki} h_i(X) + b_k$$

w_{ki} are the weights connecting the hidden layer to the output layer, M is the number of hidden neurons, and b_k is the bias for the k-th output neuron.

2.2 Training

2.2.1 Centers and Widths

The centers C_i and widths σ_i of the radial basis functions are usually determined during training. Common methods include using k-means clustering on the input data to find center locations and setting widths based on the spread of the clusters or using random centers

2.2.2 Weights and Biases

The weights w_{ki} and biases b_k in the output layer is learned through a training process that minimizes a loss function. Common loss functions include mean squared error for regression problems or cross-entropy for classification problems.

2.2.3 Training Algorithm

Training often involves iterative optimization algorithms such as gradient descent or variants. The gradients are computed using backpropagation, similar to other neural network types.

3 Code description

3.1 GaussianRBF Class

The GaussianRBF class encapsulates the functionality of the Gaussian Radial Basis Function (RBF), a crucial component in neural network architectures. This class offers methods to calculate the Gaussian RBF values based on input data, center points, and a width parameter. The Gaussian RBF is employed as a radial basis function in neural networks, contributing to non-linearity in the model.

```
class GaussianRBF:
    def __init__(self):
        pass

    def gaussian_rbf(self, x, center, sigma):
        return np.exp(-cdist(x, center, 'sqeuclidean') / (2 * sigma**2))
```

3.2 CentersInitializer Class

The CentersInitializer class plays a pivotal role in initializing the centers used in the Radial Basis Function (RBF) neural network. It provides methods for three types of center initialization: random initialization K-means clustering-based initialization and K-means per class initialization. These centers serve as anchor points for the RBF units, influencing the network's ability to capture complex patterns in the data.

```
class CentersInitializer:
    def __init__(self):
        pass

    def random_centers(self, n_centers, X_train):
        center_indices = np.random.choice(X_train.shape[0], n_centers, replace=False)
        rbf_centers = X_train[center_indices]
        return rbf_centers

    def k_means_centers(self, X, max_iters, n_centers):
        km = KMeans(n_clusters=n_centers, max_iter=max_iters, verbose=0)
        km.fit(X)
        return km.cluster_centers_

    def k_means_centers_per_class(self, X_train, y_train, max_iters, n_centers):
        unique_classes = np.unique(y_train)
        class_centers = {}

        for class_label in unique_classes:
            # Extract data for the current class
            class_data = X_train[y_train == class_label]

            # Apply k-means
            km = KMeans(n_clusters=n_centers, max_iter=max_iters, verbose=0)
            km.fit(class_data)

            # Save cluster centers
            class_centers[class_label] = km.cluster_centers_

        return class_centers
```

3.3 RBFNeuralNetwork Class

The RBFNeuralNetwork class represents a neural network architecture that incorporates Radial Basis Function (RBF) units. This class inherits functionality from both the GaussianRBF and CentersInitializer classes. It is responsible for training the neural network on input data, selecting or initializing RBF centers, and employing logistic regression for classification. The number of RBF centers and the width parameter are adjustable parameters, allowing customization based on the specific requirements of the problem at hand.

```
class RBFNeuralNetwork_torch(GaussianRBF_torch, CentersInitializer):
  def __init__(self, n_centers, rbf_width):
      self.n_centers = n_centers
      self.rbf_width = rbf_width
      self.rbf_centers = None
      self.logistic_regression = None

  def fit(self, X_train, y_train, center_initializer='random', lr=0.01, epochs=100 ,
      weight_decay = 1e-3 , optimizer = 'adam'):
      device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

      if center_initializer == 'random':
          self.rbf_centers = torch.tensor(self.random_centers(n_centers=self.n_centers,
              X_train=X_train),
                                          dtype=torch.float32, device=device)
      elif center_initializer == 'k_means':
          self.rbf_centers = torch.tensor(self.k_means_centers(X_train, max_iters=300,
              n_centers=self.n_centers),
                                          dtype=torch.float32, device=device)
      elif center_initializer == 'k_means_per_class':
        class_centers = self.k_means_centers_per_class(X_train, y_train, max_iters=300,
            n_centers=self.n_centers)
        # Use the class centers as the RBF centers
        self.rbf_centers = torch.tensor(np.concatenate(list(class_centers.values())),
            dtype=torch.float32, device=device)

      rbf_outputs_train = self.gaussian_rbf(torch.tensor(X_train, dtype=torch.float32,
          device=device),
                                            self.rbf_centers, self.rbf_width)

      input_dim = rbf_outputs_train.shape[1]
      output_dim = len(np.unique(y_train))
      model = nn.Linear(input_dim, output_dim)
      criterion = nn.CrossEntropyLoss()
      if optimizer == 'adam':
          optimizer = optim.Adam(model.parameters(), lr=lr , weight_decay= weight_decay)
      elif optimizer == 'sgd' :
          optimizer = optim.SGD(model.parameters(), lr=lr)

      X_train_tensor = torch.tensor(X_train, dtype=torch.float32, device=device)
      y_train_tensor = torch.tensor(y_train, dtype=torch.long, device=device)

      dataset = TensorDataset(X_train_tensor, y_train_tensor)
      dataloader = DataLoader(dataset, batch_size=128 , shuffle=True) #, batch_size=128

      for epoch in range(epochs):
          for inputs, labels in dataloader:
              optimizer.zero_grad()
              rbf_outputs = self.gaussian_rbf(inputs, self.rbf_centers, self.rbf_width)
              logits = model(rbf_outputs)
              loss = criterion(logits, labels)
              loss.backward()
              optimizer.step()
            if epoch % 10 == 0 or epoch == (epochs - 1) :
                with torch.no_grad():
                    rbf_outputs_test = self.gaussian_rbf(torch.tensor(X_train,
                        dtype=torch.float32, device=device),
                                                   self.rbf_centers, self.rbf_width)
                    logits_test = model(rbf_outputs_test)
                    y_pred = torch.argmax(logits_test, dim=1).cpu().numpy()
                    accuracy = accuracy_score(y_train, y_pred)
                    print(f"Epoch {epoch}, Accuracy: {accuracy * 100:.2f}%")
                    # Calculate and print the loss
                    loss_value = criterion(logits_test, torch.tensor(y_train,
                        dtype=torch.long, device=device)).item()
                    print(f"Epoch {epoch}, Loss: {loss_value:.4f}")
                    cm = confusion_matrix(y_train, y_pred)
                    class_names = ['airplane', 'automobile', 'bird', 'cat',
                                 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']


                    heatmap_1 = sns.heatmap(
                         cm, annot=True, fmt='d', cmap='YlGnBu', xticklabels=class_names,
                             yticklabels=class_names)
                    heatmap_1.set_title("confusion matrix")
                    plt.xlabel('Predicted')
                    plt.ylabel('Actual')
                     # print(classification_report(batch_y, output_for_matrix))
                    plt.show()


        self.logistic_regression = model

    def predict(self, X_test):
        rbf_outputs_test = self.gaussian_rbf(torch.tensor(X_test, dtype=torch.float32),
                                         self.rbf_centers, self.rbf_width)
        logits = self.logistic_regression(rbf_outputs_test)
        return torch.argmax(logits, dim=1).cpu().numpy()
```

3.4 GaussianRBF_torch Class

The GaussianRBF_torch class is designed for PyTorch and focuses on implementing the Gaussian Radial Basis Function (RBF) using PyTorch operations. The Gaussian rbf method computes the RBF values for a given input tensor, center tensor, and a specified sigma. PyTorch's torch.cdist and torch.exp functions are utilized to efficiently calculate the pairwise distances and exponentiation, respectively. This class serves as a crucial component for the RBF neural network implemented in PyTorch.

```
    class GaussianRBF_torch:
    def __init__(self):
        pass

    @staticmethod
    def gaussian_rbf(x, center, sigma):
        return torch.exp(-torch.cdist(x, center, p=2) / (2 * sigma**2))
```

3.5 RBFNeuralNetwork_torch Class

The RBFNeuralNetwork_torch class is a PyTorch-based implementation of a neural network incorporating Radial Basis Function (RBF) units. It inherits functionality from the GaussianRBF_torch class and the previously mentioned CentersInitializer class. This class is responsible for training the neural network using PyTorch's capabilities, with options for different optimizers such as Adam or SGD. The training process involves initializing or selecting RBF centers, calculating RBF layer outputs, defining a linear layer for logistic regression, and optimizing the model parameters.

The fit method trains the model using the specified training data, labels, and other hyperparameters such as learning rate, epochs, weight decay, and optimizer choice. The PyTorch DataLoader is employed for efficient batch processing during training. The model's accuracy is printed periodically during training to monitor its progress.

The predict method is responsible for generating predictions on new data using the trained model. It calculates the RBF layer outputs for the test data and applies the logistic regression layer to obtain predictions. The resulting predictions are converted from PyTorch tensors to NumPy arrays for compatibility with other Python libraries.

```
class RBFNeuralNetwork_torch(GaussianRBF_torch, CentersInitializer):
def __init__(self, n_centers, rbf_width):
    self.n_centers = n_centers
    self.rbf_width = rbf_width
    self.rbf_centers = None
    self.logistic_regression = None

def fit(self, X_train, y_train, center_initializer='random', lr=0.01, epochs=100 ,
     weight_decay = 1e-3 , optimizer = 'adam'):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    if center_initializer == 'random':
        self.rbf_centers = torch.tensor(self.random_centers(n_centers=self.n_centers,
            X_train=X_train),
                                        dtype=torch.float32, device=device)
    elif center_initializer == 'k_means':
        self.rbf_centers = torch.tensor(self.k_means_centers(X_train, max_iters=300,
            n_centers=self.n_centers),
                                        dtype=torch.float32, device=device)

    rbf_outputs_train = self.gaussian_rbf(torch.tensor(X_train, dtype=torch.float32,
        device=device),
                                          self.rbf_centers, self.rbf_width)

    input_dim = rbf_outputs_train.shape[1]
    output_dim = len(np.unique(y_train))
    model = nn.Linear(input_dim, output_dim)
    criterion = nn.CrossEntropyLoss()
    if optimizer == 'adam':
        optimizer = optim.Adam(model.parameters(), lr=lr , weight_decay= weight_decay)
    elif optimizer == 'sgd' :
        optimizer = optim.SGD(model.parameters(), lr=lr)

    X_train_tensor = torch.tensor(X_train, dtype=torch.float32, device=device)
    y_train_tensor = torch.tensor(y_train, dtype=torch.long, device=device)

    dataset = TensorDataset(X_train_tensor, y_train_tensor)
    dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

    for epoch in range(epochs):
        for inputs, labels in dataloader:
            optimizer.zero_grad()
            rbf_outputs = self.gaussian_rbf(inputs, self.rbf_centers, self.rbf_width)
            logits = model(rbf_outputs)
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()

        if epoch % 10 == 0:
            with torch.no_grad():
```

```
            rbf_outputs_test = self.gaussian_rbf(torch.tensor(X_train,
                dtype=torch.float32, device=device),
                                            self.rbf_centers, self.rbf_width)
            logits_test = model(rbf_outputs_test)
            y_pred = torch.argmax(logits_test, dim=1).cpu().numpy()
            accuracy = accuracy_score(y_train, y_pred)
            print(f"Epoch {epoch}, Accuracy: {accuracy * 100:.2f}%")

        self.logistic_regression = model

    def predict(self, X_test):
        rbf_outputs_test = self.gaussian_rbf(torch.tensor(X_test, dtype=torch.float32),
                                            self.rbf_centers, self.rbf_width)
        logits = self.logistic_regression(rbf_outputs_test)
        return torch.argmax(logits, dim=1).cpu().numpy()
```

4 Results

The limit on the number of iterations in logistic regression is set to 1000 and the max iterations for the k-means is set to 300. This was to prevent the program from taking too long to process the data. The downside is the lack of full optimization of the centers, the weights and the biases

Furthermore to further optimize our code we used Principal component analysis (PCA) to lower the dimensionality of our images and speed up the process of the data. Unfortunately, this was applied after some initial attempts to find the best values of our parameters. As a result some elapsed times (especially on the k-means fit) are big.

For the PCA we used number of components to be **500** and the total Explained Variance is **0.98** or **98%**

4.1 RBFNeuralNetwork Class

4.1.1 Random center initializing

Note: The titles of the confusion matrices are wrong. The sigmoid kernel part of the title comes from the previous part of the project

4.1.1.1 n_centers=500, rbf_width=10.0 Training Accuracy: **46.10%**
Test Accuracy: **45.53%**
Elapsed time **152.99s**
The classification report can be seen in Table 1

Class	Precision	Recall	F1-Score	Support
0	0.53	0.49	0.51	1000
1	0.51	0.55	0.53	1000
2	0.36	0.30	0.33	1000
3	0.34	0.30	0.32	1000
4	0.42	0.37	0.39	1000
5	0.40	0.35	0.37	1000
6	0.44	0.56	0.50	1000
7	0.48	0.46	0.47	1000
8	0.54	0.63	0.58	1000
9	0.48	0.53	0.50	1000
Accuracy			0.46	10000
Macro Avg	0.45	0.46	0.45	10000
Weighted Avg	0.45	0.46	0.45	10000

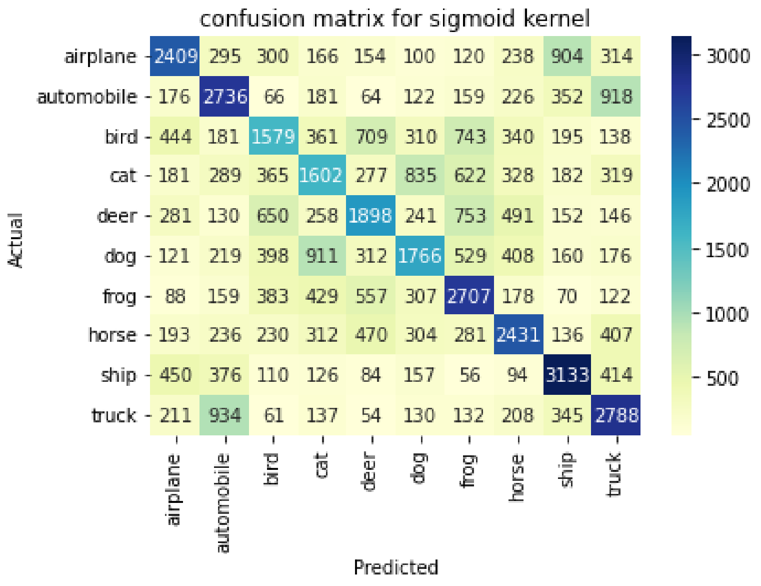
Table 1: Classification Report for the RBF Model 1

The confusion matrix for the training data can be seen in Figure 1
The confusion matrix for the test data can be seen in Figure 2

4.1.1.2 n_centers=500, rbf_width=0.5 Training Accuracy: **10.92%**
Test Accuracy: **10.00%**
Elapsed time **52.28s**

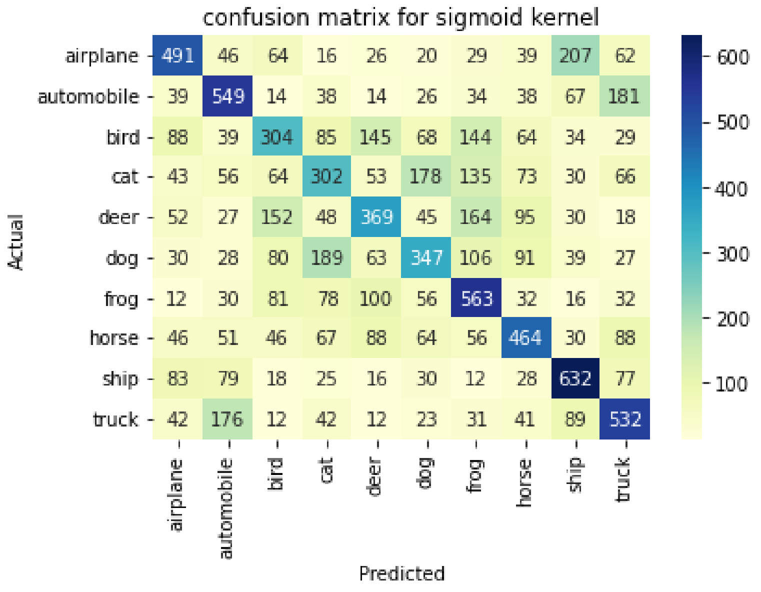
The confusion matrix for the test data can be seen in Figure 3

4.1.1.3 n_centers=500, rbf_width=5 Training Accuracy: **37.34%**
Test Accuracy: **37.01%**
Elapsed time **76.32s**
The classification report can be seen in Table 2



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	2409	295	300	166	154	100	120	238	904	314
automobile	176	2736	66	181	64	122	159	226	352	918
bird	444	181	1579	361	709	310	743	340	195	138
cat	181	289	365	1602	277	835	622	328	182	319
deer	281	130	650	258	1898	241	753	491	152	146
dog	121	219	398	911	312	1766	529	408	160	176
frog	88	159	383	429	557	307	2707	178	70	122
horse	193	236	230	312	470	304	281	2431	136	407
ship	450	376	110	126	84	157	56	94	3133	414
truck	211	934	61	137	54	130	132	208	345	2788

Figure 1: training cm model 1



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	491	46	64	16	26	20	29	39	207	62
automobile	39	549	14	38	14	26	34	38	67	181
bird	88	39	304	85	145	68	144	64	34	29
cat	43	56	64	302	53	178	135	73	30	66
deer	52	27	152	48	369	45	164	95	30	18
dog	30	28	80	189	63	347	106	91	39	27
frog	12	30	81	78	100	56	563	32	16	32
horse	46	51	46	67	88	64	56	464	30	88
ship	83	79	18	25	16	30	12	28	632	77
truck	42	176	12	42	12	23	31	41	89	532

Figure 2: test cm model 1

The confusion matrix for the test data can be seen in Figure 4

4.1.1.4 n_centers=100, rbf_width=5

Training Accuracy: **31.55%**
Test Accuracy: **31.81%**
Elapsed time **22.66s**
The classification report can be seen in Table 3

4.1.1.5 n_centers=100, rbf_width=10

Training Accuracy: **40.08%**
Test Accuracy: **39.96%**
Elapsed time **39.26s**
The classification report can be seen in Table 4

The confusion matrix for the test data can be seen in Figure 5

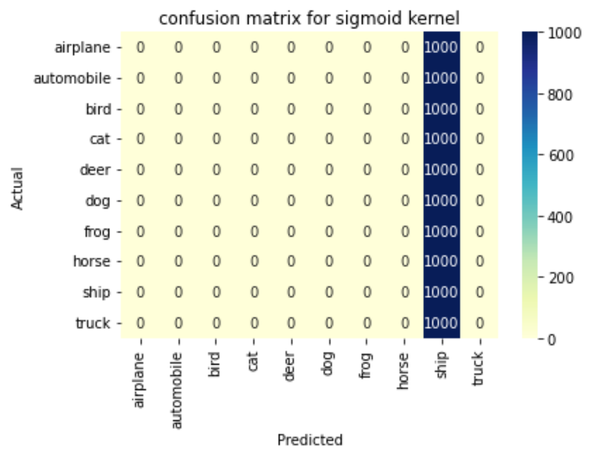


Figure 3: test c=500 s=0.5

Class	Precision	Recall	F1-Score	Support
0	0.51	0.40	0.45	1000
1	0.27	0.52	0.36	1000
2	0.33	0.20	0.25	1000
3	0.28	0.17	0.21	1000
4	0.42	0.29	0.34	1000
5	0.31	0.30	0.31	1000
6	0.44	0.45	0.44	1000
7	0.37	0.30	0.33	1000
8	0.52	0.51	0.51	1000
9	0.35	0.56	0.43	1000
Accuracy			0.37	10000
Macro Avg	0.38	0.37	0.36	10000
Weighted Avg	0.38	0.37	0.36	10000

Table 2: Classification Report for n_centers=500, rbf_width=5

Class	Precision	Recall	F1-Score	Support
0	0.45	0.34	0.38	1000
1	0.25	0.40	0.30	1000
2	0.29	0.17	0.21	1000
3	0.19	0.06	0.09	1000
4	0.38	0.29	0.33	1000
5	0.28	0.23	0.25	1000
6	0.39	0.39	0.39	1000
7	0.29	0.24	0.26	1000
8	0.44	0.45	0.45	1000
9	0.27	0.62	0.38	1000
Accuracy			0.32	10000
Macro Avg	0.32	0.32	0.30	10000
Weighted Avg	0.32	0.32	0.30	10000

Table 3: Classification Report for n_centers=100, rbf_width=5

4.1.1.6 n_centers=1000, rbf_width=10

Training Accuracy: **47.81%**
Test Accuracy: **46.92%**

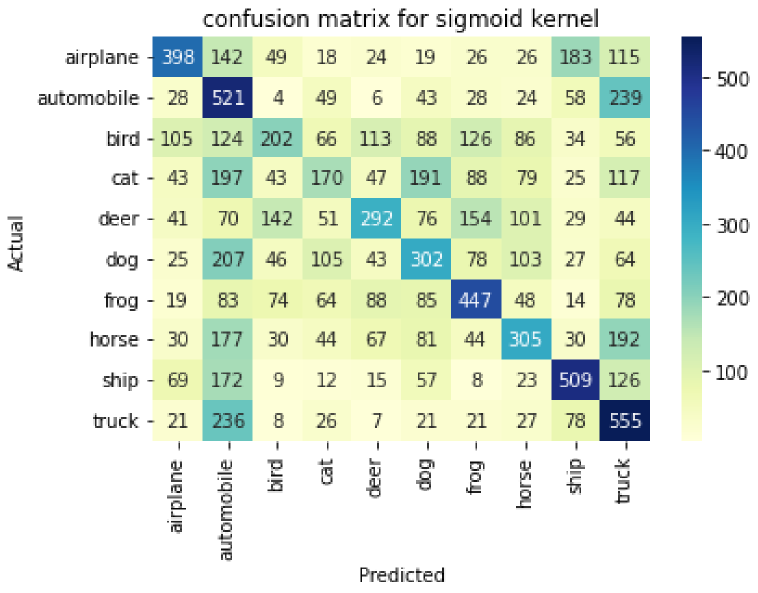


Figure 4: test c=500 s=5

Class	Precision	Recall	F1-Score	Support
0	0.48	0.44	0.46	1000
1	0.42	0.46	0.44	1000
2	0.31	0.26	0.28	1000
3	0.31	0.23	0.26	1000
4	0.38	0.33	0.35	1000
5	0.36	0.36	0.36	1000
6	0.40	0.50	0.45	1000
7	0.40	0.37	0.38	1000
8	0.47	0.55	0.50	1000
9	0.42	0.50	0.46	1000
Accuracy			0.40	10000
Macro Avg	0.39	0.40	0.39	10000
Weighted Avg	0.39	0.40	0.39	10000

Table 4: Classification Report for c = 100 s = 10

Elapsed time **313.51s**
The classification report can be seen in Table 5
The confusion matrix for the test data can be seen in Figure 6

4.1.1.7 Other runs During our quest to find the best parameters for the classification, different values of the parameters were used other than the ones mentioned in the previous section. On table 6 are the accuracy scores of these runs alongside these values.

4.1.2 K-means initializer

For this initializer, we used sometimes batched and sometimes the whole dataset. The reason behind this is the time needed to create centers for the whole batch and the simultaneous diminishing returns in accuracy. It is mentioned explicitly in each paragraph if we are using batched or the whole dataset.

4.1.2.1 n_centers=500, rbf_width=10 Training Accuracy: **44.11%**
Test Accuracy: **43.63%**

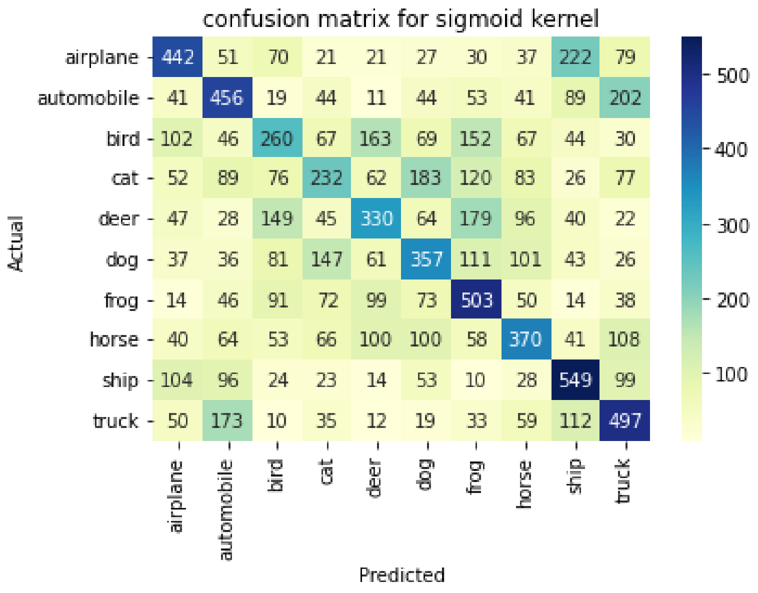


Figure 5: test c=100 s=10

Class	Precision	Recall	F1-Score	Support
0	0.54	0.51	0.52	1000
1	0.53	0.56	0.54	1000
2	0.38	0.32	0.35	1000
3	0.36	0.31	0.33	1000
4	0.44	0.38	0.41	1000
5	0.42	0.36	0.39	1000
6	0.45	0.57	0.51	1000
7	0.49	0.49	0.49	1000
8	0.54	0.64	0.59	1000
9	0.50	0.55	0.52	1000
Accuracy			0.47	10000
Macro Avg	0.46	0.47	0.46	10000
Weighted Avg	0.46	0.47	0.46	10000

Table 5: Classification Report for c = 1000 s = 10

Table 6: Accuracy Results for Different Variance and Centers

Variance	Centers	Accuracy (%)
8	10	29.15
8	500	44.98
8	1000	46.98
20	500	42.4
10	10	30.39

Elapsed time **3431.12s**
We are using the whole dataset
The classification report can be seen in Table 7

The confusion matrix for the train data can be seen in Figure 7

The confusion matrix for the test data can be seen in Figure 8

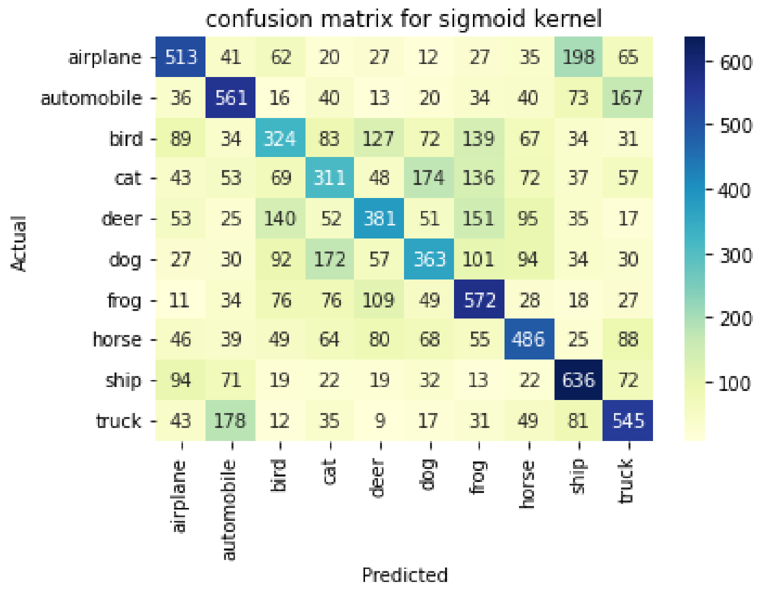


Figure 6: test c=1000 s=10

Class	Precision	Recall	F1-Score	Support
0	0.52	0.46	0.49	1000
1	0.48	0.53	0.50	1000
2	0.34	0.28	0.31	1000
3	0.34	0.29	0.31	1000
4	0.39	0.36	0.37	1000
5	0.39	0.33	0.36	1000
6	0.43	0.55	0.48	1000
7	0.45	0.44	0.44	1000
8	0.51	0.61	0.55	1000
9	0.46	0.52	0.49	1000
Accuracy			0.44	10000
Macro Avg	0.43	0.44	0.43	10000
Weighted Avg	0.43	0.44	0.43	10000

Table 7: Classification Report for k-means c=500 s=10

4.1.2.2 n_centers=100, rbf_width=10

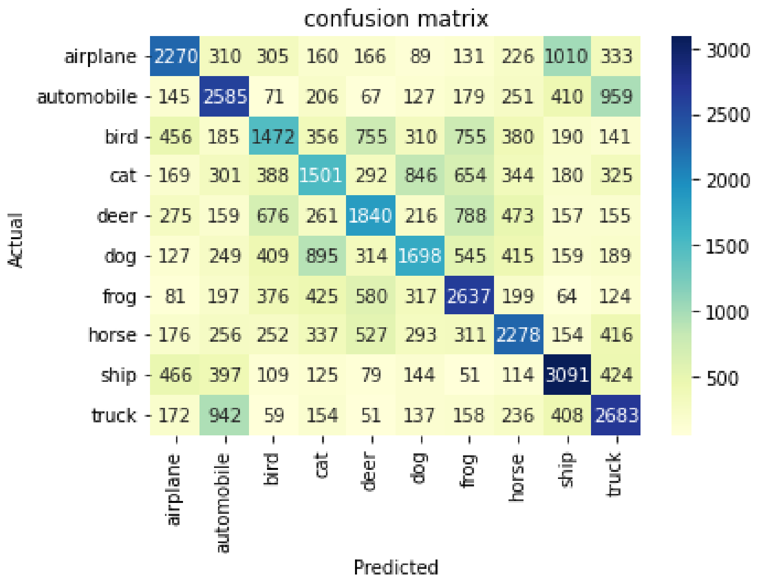
Training Accuracy: **39.47%**
Test Accuracy: **39.61%**
Elapsed time **1637s**
We are using the whole dataset
The classification report can be seen in Table 8
The confusion matrix for the test data can be seen in Figure 9

4.1.2.3 n_centers=10, rbf_width=10

Training Accuracy: **29.30%**
Test Accuracy: **30.30%**
Elapsed time **208.29s**
We are using the whole dataset
The confusion matrix for the test data can be seen in Figure 10

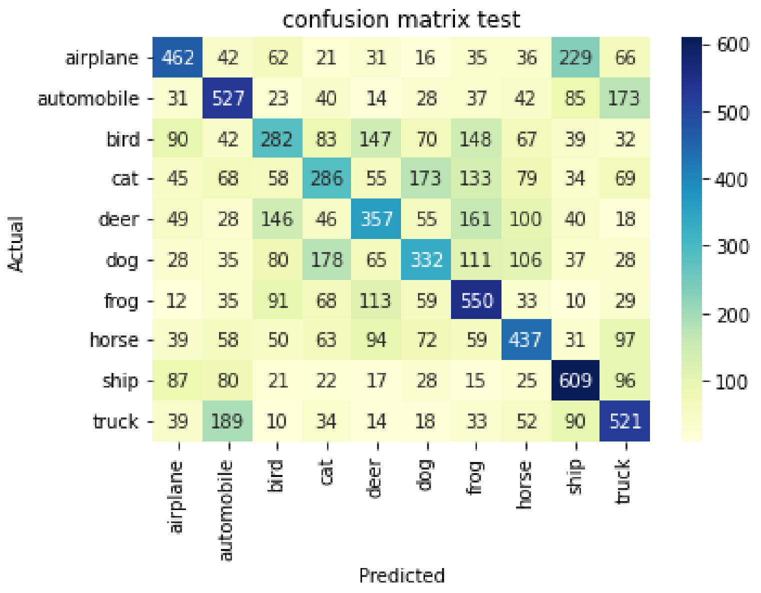
4.1.2.4 Other runs

A table with different values for the parameters can be seen in Table 9



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	2270	310	305	160	166	89	131	226	1010	333
automobile	145	2585	71	206	67	127	179	251	410	959
bird	456	185	1472	356	755	310	755	380	190	141
cat	169	301	388	1501	292	846	654	344	180	325
deer	275	159	676	261	1840	216	788	473	157	155
dog	127	249	409	895	314	1698	545	415	159	189
frog	81	197	376	425	580	317	2637	199	64	124
horse	176	256	252	337	527	293	311	2278	154	416
ship	466	397	109	125	79	144	51	114	3091	424
truck	172	942	59	154	51	137	158	236	408	2683

Figure 7: k-means c=500 s=10



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	462	42	62	21	31	16	35	36	229	66
automobile	31	527	23	40	14	28	37	42	85	173
bird	90	42	282	83	147	70	148	67	39	32
cat	45	68	58	286	55	173	133	79	34	69
deer	49	28	146	46	357	55	161	100	40	18
dog	28	35	80	178	65	332	111	106	37	28
frog	12	35	91	68	113	59	550	33	10	29
horse	39	58	50	63	94	72	59	437	31	97
ship	87	80	21	22	17	28	15	25	609	96
truck	39	189	10	34	14	18	33	52	90	521

Figure 8: test k-means c=500 s=10

4.2 RBFNeuralNetwork Torch Class

150 epochs were used each time to have a common factor for comparison between the other parameters. We also used optimizer='adam' as it seemed to have better results and it is working better with our type of classification

4.2.1 Random initilizer

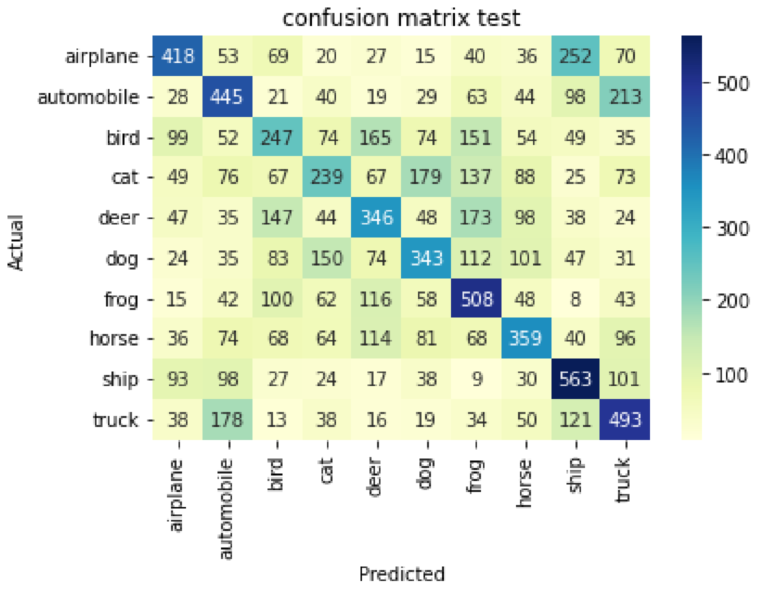
4.2.1.1 n_centers=500, rbf_width=10 Accuracy: **21.17%**

elapsed time **218.67s**
We can see in Figure 11 the training data accuracy during the epochs. We see fluctuations that indicate wrong parameter values.

We can see that the accuracy is too low so we will try other values for the parameters

Class	Precision	Recall	F1-Score	Support
0	0.49	0.42	0.45	1000
1	0.41	0.45	0.43	1000
2	0.29	0.25	0.27	1000
3	0.32	0.24	0.27	1000
4	0.36	0.35	0.35	1000
5	0.39	0.34	0.36	1000
6	0.39	0.51	0.44	1000
7	0.40	0.36	0.38	1000
8	0.45	0.56	0.50	1000
9	0.42	0.49	0.45	1000
Accuracy			0.40	10000
Macro Avg	0.39	0.40	0.39	10000
Weighted Avg	0.39	0.40	0.39	10000

Table 8: Classification Report for k-means c=100 s=10



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	418	53	69	20	27	15	40	36	252	70
automobile	28	445	21	40	19	29	63	44	98	213
bird	99	52	247	74	165	74	151	54	49	35
cat	49	76	67	239	67	179	137	88	25	73
deer	47	35	147	44	346	48	173	98	38	24
dog	24	35	83	150	74	343	112	101	47	31
frog	15	42	100	62	116	58	508	48	8	43
horse	36	74	68	64	114	81	68	359	40	96
ship	93	98	27	24	17	38	9	30	563	101
truck	38	178	13	38	16	19	34	50	121	493

Figure 9: test k-means c=100 s=10

Table 9: Accuracy Results for Different Variance and Centers with K-means

Batching	Variance	Centers	Accuracy (%)
Batch	8	10	29.29
Batch	8	20	32.53
Whole	8	10	29.47
Batch	8	50	36.98
Batch	8	100	39.45
Batch	8	200	41.88

4.2.1.2 n_centers=500, rbf_width=2

Accuracy: **44.18%** Elapsed time: **219.74s**
From Figure 12 we can see that the loss and the accuracy have greatly improved. This is the best result we can get by using a random initializer.

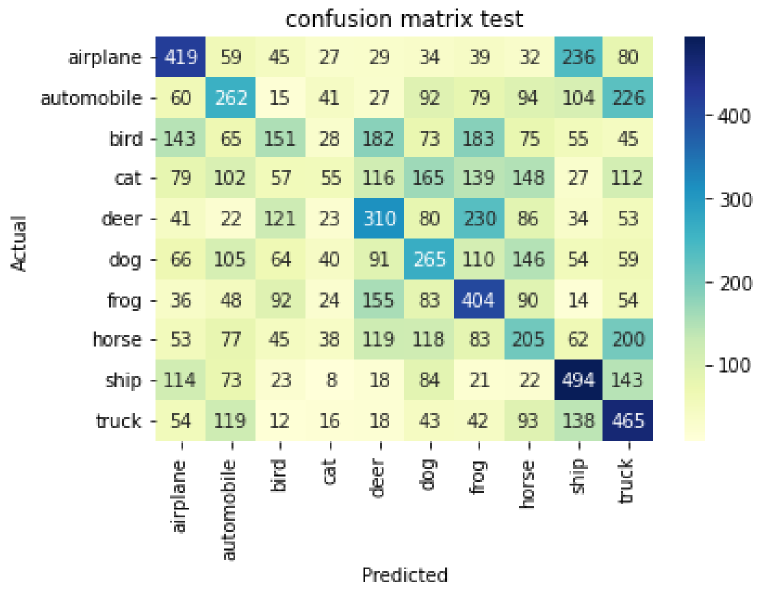


Figure 10: test k-means c=10 s=10

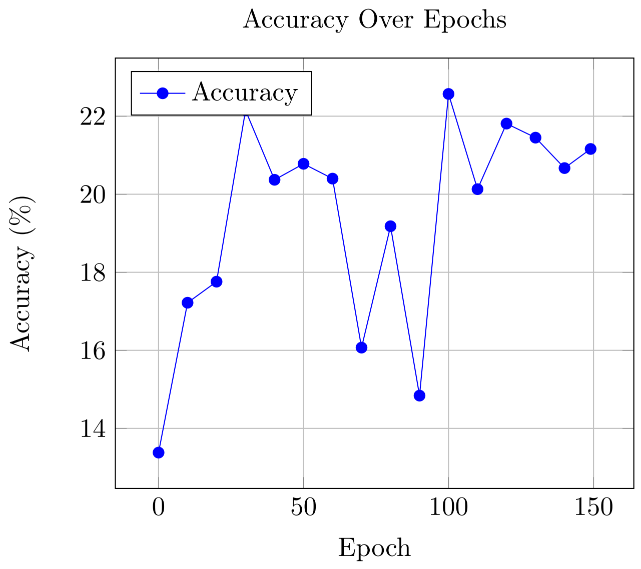


Figure 11: Accuracy Over Epochs n_centers=500, rbf_width=10

4.2.2 K-means initializer

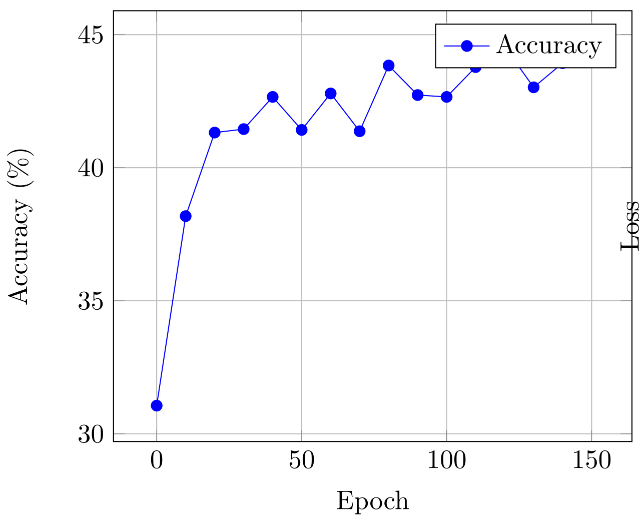
We will use 500 centers and rbf width 2 as they concluded with better results when they were used in the random initializer

4.2.2.1 lr=0.01 , weight_decay = 1e-6 Accuracy test: **42.98%**

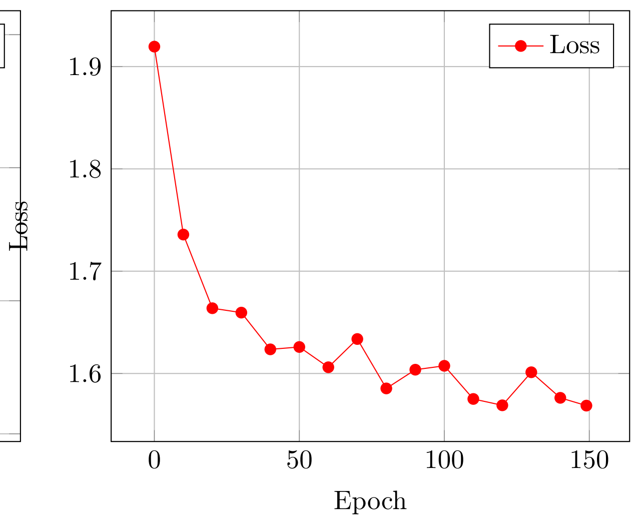
Elapsed time: **454.57s** We can see in Figure 13 the training data accuracy during the epochs. It seems that we are getting close to our best score so far without any fluctuations.

4.2.3 K-means initializer per class

This method was created specifically for the torch class as it seems that the k-means have better results when using this class. The way it works is by creating n_centers for each class separately, minimizing the loss faster and increasing the accuracy. After some tests, it was found that 100 centers per class

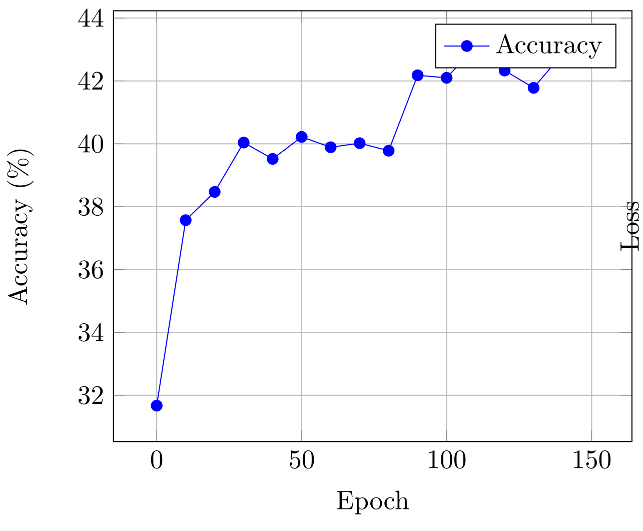


(a) Epoch Accuracy

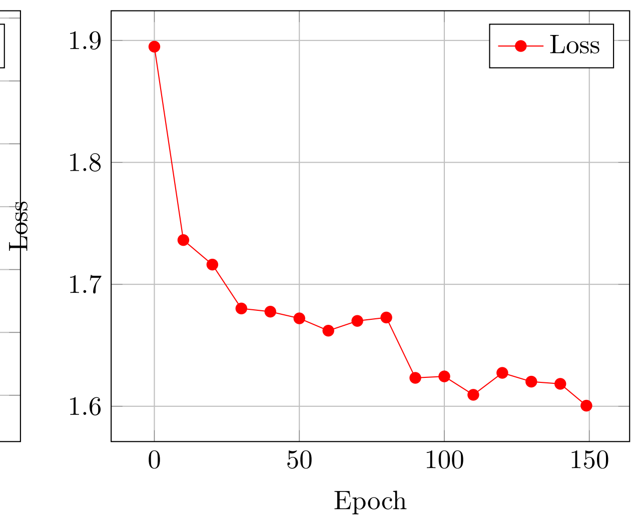


(b) Epoch Loss

Figure 12: Epoch Accuracy and Loss (label: $n\_centers = 500, rbf\_width = 2\_random\_init$)



(a) Epoch Accuracy



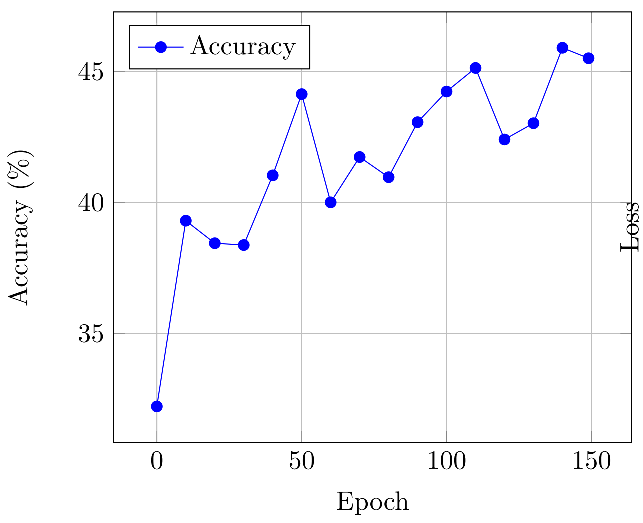
(b) Epoch Loss

Figure 13: Epoch Accuracy and Loss K-means initializer lr=0.01 , weight_decay $= 1e-6$

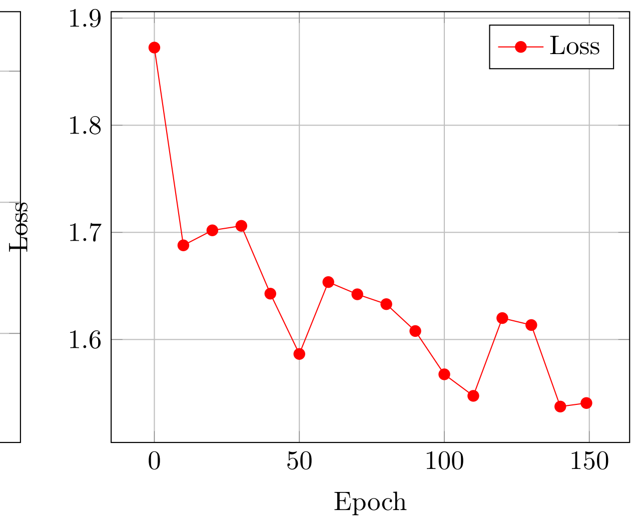
(500 in total) and an rbf width of 2 have the best results. We will change the learning rate and the weight decay to find the optimal solution.

4.2.3.1 lr=0.01,weight_decay = 1e-6 Accuracy for test data **44.44%**
Elapsed time: **303.76s**
We can see in Figure 14 the loss and the accuracy of the training data during each epoch

4.2.3.2 lr=0.008 weight 1e-6 These parameters had the best results. The accuracy was **45.29%** and the loss from the training data **1.5321**. Elapsed time **290.37s** We can see in Figure 15 the loss and the accuracy of the training data during each epoch. This is the best result so far.

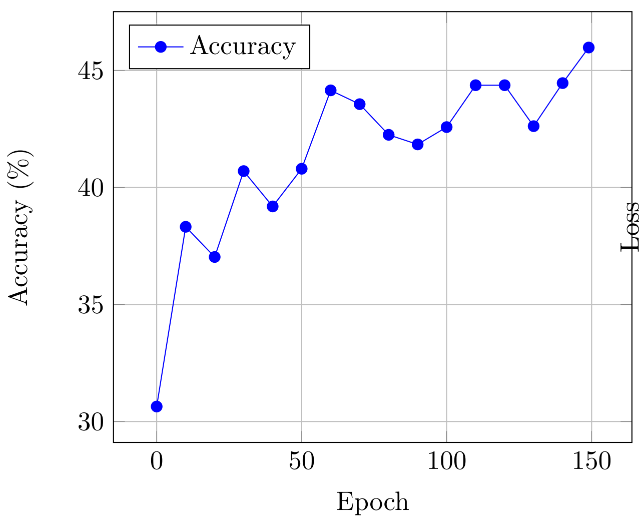


(a) Epoch Accuracy

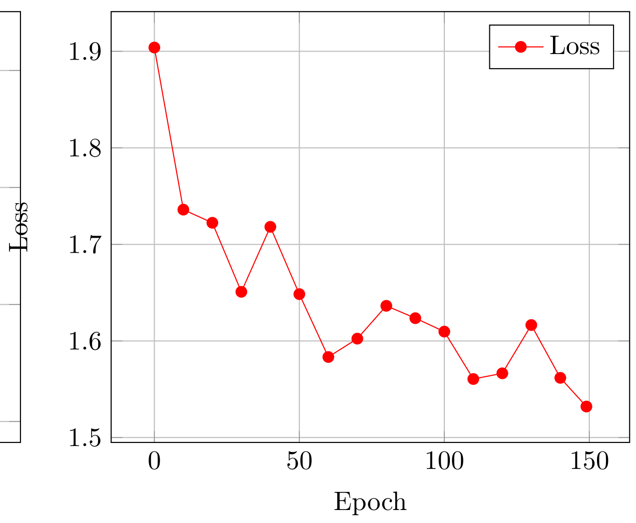


(b) Epoch Loss

Figure 14: Epoch Accuracy and Loss lr=0.01,weight_decay = 1e-6 K-means per class



(a) Epoch Accuracy



(b) Epoch Loss

Figure 15: Epoch Accuracy and Loss lr=0.008 weight 1e-6

5 Conclusion

- In the case of the vanilla RBF neural network, the random centers initializer took less time to process the data and the accuracy was similar, and sometimes better than the k-means initializer. more specifically for 500 centers and 10 variances, the random initializer took **152.99s** while the k-means initializer took **3431.12s** while the accuracy for the first was **45.53%** while the later's was **43.63%**.
- To combat the lateness during the training of the k means initializer, we used the PCA method to reduce the dimensions of each data. This significantly improved our time. Using again 500 centers and 10 variance, the time it took was **344s** which is **10 times** faster than without using the PCA.
- The best results were given using: **1000 centers** , **8 variance** with the **simple RBF neural network**. Accuracy of **46.98 %**. The number of iterations for this result reached our limit (

1000 iterations) which may indicate that better results could be achieved using a higher upper limit but this could further delay our results.

- The best result for the torch RBF neural network was given by using the **adam optimizer, learning rate 0.008 and weight decay 1e-6** . The result was **45.29%** which is close to our best result in general.

- Although the 2 RBF classes should behave similarly, using the Adam optimizer and due to the different updates of the weights and the biases, the sigma and the number of centers parameters were largely different in the 2 classes to achieve the best results. More specifically the vanilla rbf needed a variance of 8 and 1000 centers for the best results while the torch rbf needed a variance of 2 and 500 centers in total.

- Similar to the vanilla rbf, the torch rbf has similar results using the random and the k-means initializer. Although a slight increase in accuracy and a decrease in loss when using the k-means by a class initializer, generally the results don't differ that much. The random initializer takes less time to complete than expected due to its simplistic approach to finding the centers and classifying the data.

- From the previous projects, we had concluded that the accuracy score for the 1-nearest neighbor was **35.39%** the 3-nearest neighbors **33.03%** and the nearest centroid **27.74%**. These results are lower than our RBF neural network which was **46.98 %** and that means that by comparison, the neural network is better at classifying the CIFAR-10 dataset. Still, our classic neural network seems to be coping better at **48%** accuracy. The polynomial kernel SVM with the batched data had a score of **44.8%**, really close to the other classification methods with only a portion of the data.

References

[Sai] Ashish Saini. Radial basis function networks (rbfns) with python 3: A comprehensive guide.

[Zog] Raouf Zoghbi. Rbf neural network python.