NEURAL NETWORKS

Projects Presentation

Author:

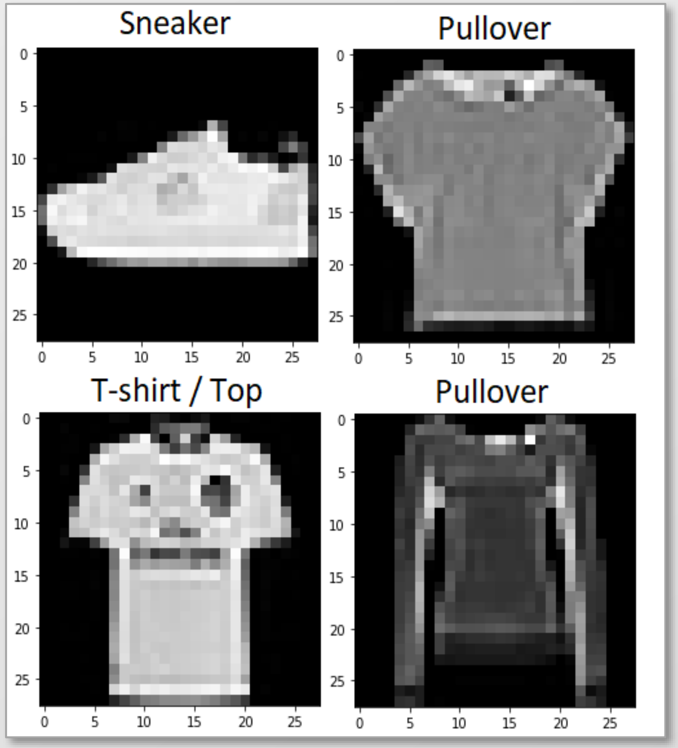
Sotiris Ftiakas 3076

Programming Language:

Python

Dataset

- Fashion – MNIST
- 28 x 28 grayscale clothing images (784 pixels total)
- Pixels take values from 0 - 255, representing their darkness
- 10 different labels (T-shirt/top, Trouser, Pullover, etc.)
- CSV file
- 60.000 training examples
- 10.000 test examples



K-Nearest Neighbors

- Function with 2 parameters: k, distance_metric

  k: Number of neighbors

  distance_metric: (Euclidean, Manhattan and Cosine Similarity supported)

- Experiment Results:

K	Training Examples	Metric	Accuracy	Time
3	5.000	Euclidean	0.81	44 s
3	60.000	Euclidean	0.86	563 s
3	60.000	Cosine Similarity	0.86	895 s
1	5.000	Euclidean	0.80	39 s
1	**60.000**	**Euclidean**	**0.86**	**525 s**
1	60.000	Cosine Similarity	0.86	828 s

Nearest Class Centroid

- Function with 2 parameters: classes, distance_metric

  classes: Number of different classes/labels

  distance_metric: (Euclidean, Manhattan and Cosine Similarity supported)

- Experiment Results:

Training Examples	Metric	Accuracy	Time
5.000	Cosine Similarity	0.68	8 s
5.000	Euclidean	0.69	1 s
5.000	Manhattan	0.61	1 s
60.000	Cosine Similarity	0.68	310 s
60.000	**Euclidean**	**0.69**	**15 s**
60.000	Manhattan	0.61	15 s

Multilayer Perceptron

Attempt to build my MLP from scratch:

- Initialize weights and biases (random weights, 0 biases)
- Sigmoid as activation function at layers
- Forward propagation, pass training examples through layers
- Back propagation, minimize loss function with gradient descent
- Update weights and biases at the end of each epoch
- <u>Runs, but doesn't correctly update weights and biases.</u>

Use of sklearn library:

- MLPClassifier( )
- Experimentation with its parameters

```python
# The forward propagation function

def forward_prop(model,a0):

    # Load parameters from model
    w1, b1, w2, b2, w3, b3 = model['w1'], model['b1'], model['w2'], model['b2'], model['w3'],model['b3']

    # First linear step = input layer x times the dot product of the weights + our bias b
    u1 = a0.dot(w1) + b1

    # First activation function
    a1 = sigmoid(u1)

    # Second linear step
    u2 = a1.dot(w2) + b2

    # Second activation function
    a2 = sigmoid(u2)

    # Third linear step
    u3 = a2.dot(w3) + b3

    # For the Third linear activation function (last layer) either the sigmoid or softmax should be used.
    a3 = sigmoid(u3)

    #Store results and return them

    fp_results = {'a0':a0, 'u1':u1, 'a1':a1, 'u2':u2, 'a2':a2, 'u3':u3, 'a3':a3}
    return fp_results
```

```python
# The backward propagation function

def backward_prop(model,fp_results,y):

    # Load parameters from model
    w1, b1, w2, b2, w3, b3 = model['w1'], model['b1'], model['w2'], model['b2'], model['w3'],model['b3']

    # Load forward propagation results
    a0, a1, a2, a3 = fp_results['a0'], fp_results['a1'], fp_results['a2'], fp_results['a3']

    u1, u2, u3 = fp_results['u1'], fp_results['u2'], fp_results['u3']

    # Get number of samples
    m = y.shape[0]

    # Calculate δ for output layer
    delta3 = 2*loss(y,a3)/m*sigmoid_prime(u3)

    dw3 = (a2.T).dot(delta3)
    db3 = np.sum(delta3, axis=0)

    # Calculate δ for hidden layer
    delta2 = np.multiply(delta3.dot(w3.T), sigmoid_prime(a2))

    dw2 = np.dot(a1.T, delta2)
    db2 = np.sum(delta2, axis=0)

    # Calculate δ for input layer
    delta1 = np.multiply(delta2.dot(w2.T), sigmoid_prime(a1))

    dw1 = np.dot(a0.T, delta1)
    db1 = np.sum(delta1,axis=0)

    # Store gradients
    grads = {'dw3':dw3, 'db3':db3, 'dw2':dw2,'db2':db2,'dw1':dw1,'db1':db1}
    return grads
```

Multilayer Perceptron

- **Best model's parameters:**

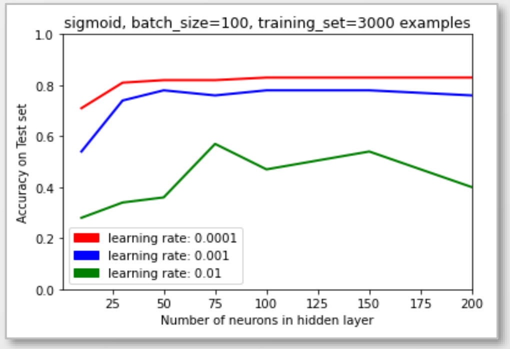
  1 hidden layer of 100 neurons

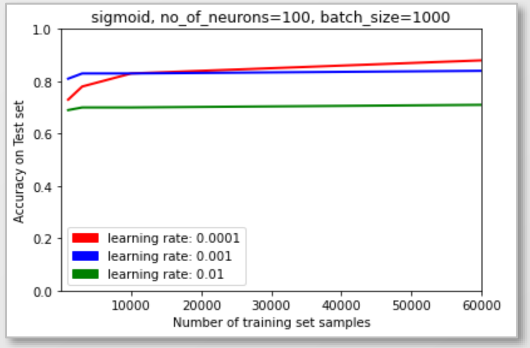
  sigmoid as activation function

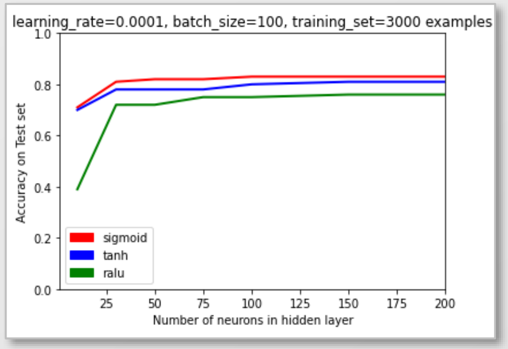
  batch size = 750

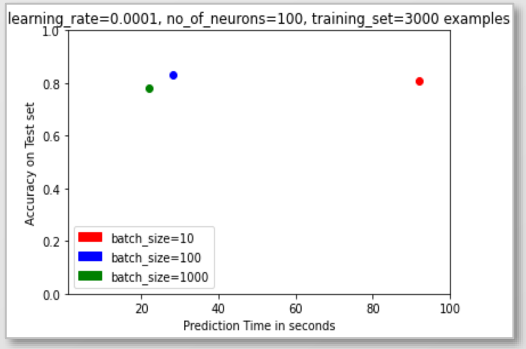
  learning rate = 0.0002

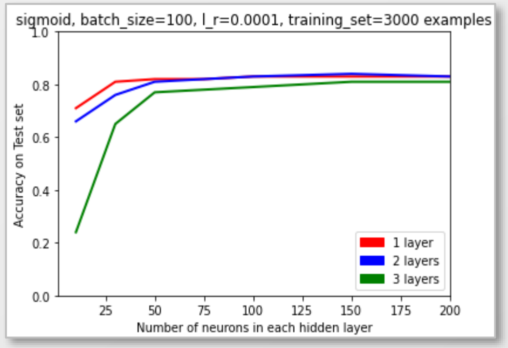
- **Accuracy on Test Set:** 0.88
- **Time:** 197 s

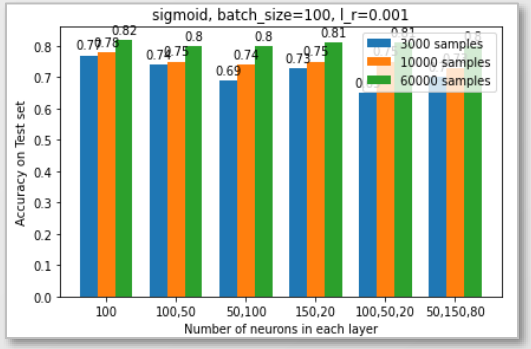


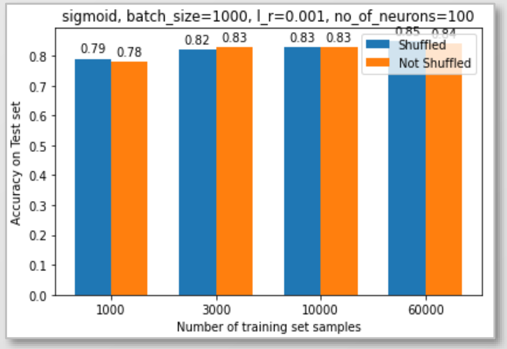


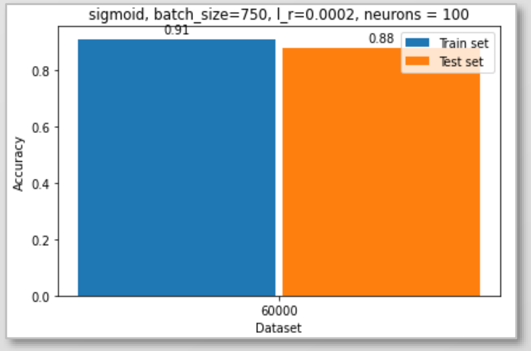












Support Vector Machine

Use of sklearn library:

SVC( )

- Kernel type (rbf, polynomial, linear, sigmoid)
- Polynomial kernel degree
- C value (Positive regularization parameter)
- Gamma value (Only for rbf, polynomial or sigmoid)

LinearSVC( )

- Penalty type (L1, L2)
- Optimization (Dual, Primal)

Additional changes:

- Normalization between 0 – 1
- PCA dimensionality reduction

Support Vector Machine

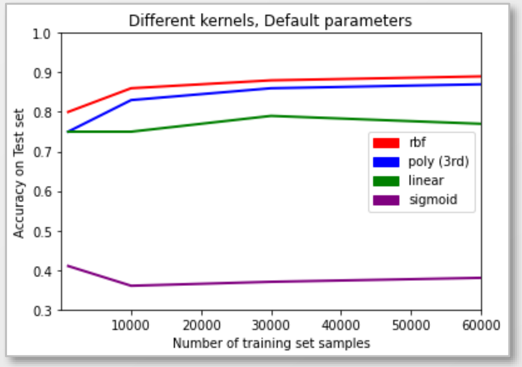
- **Best model's parameters:**

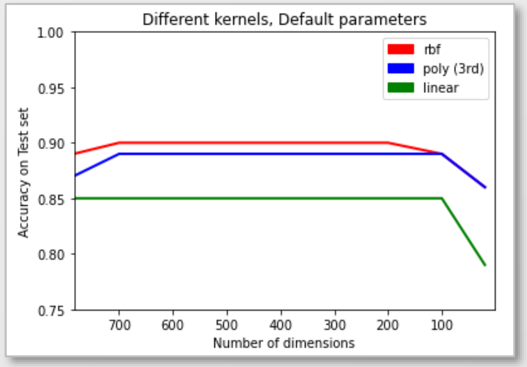
  RBF Kernel

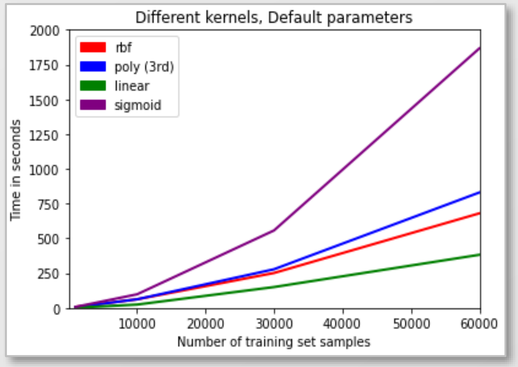
  PCA = 200 dim

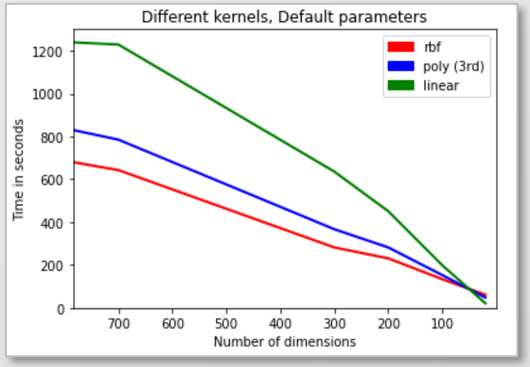
  C = 10

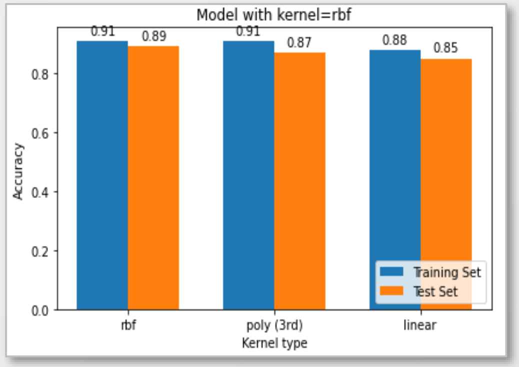
- **Accuracy on Test Set:** 0.91
- **Time:** 203 s

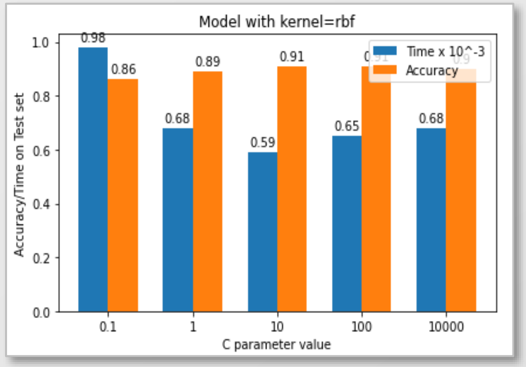


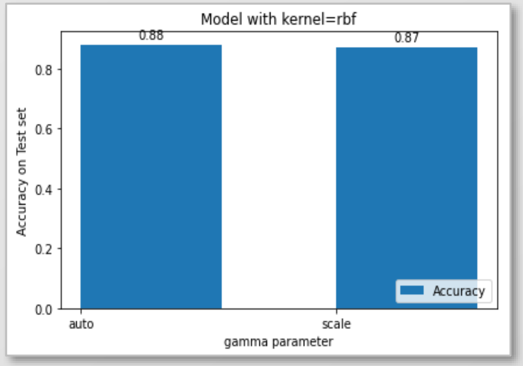


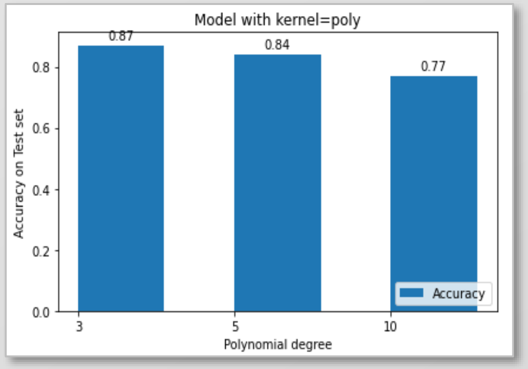












Radial Basis Function Neural Network

RBFNN implementation from scratch

- Choose centers/hidden neurons (random, k-means)
- Define our radial basis function ($e^{-\beta * D^2}$)
- β = optimization parameter, D = distance of a point from a center
- Different β values [($\frac{1}{std^2}$), ($\frac{\sqrt{2*k}}{Dmax}$)]
- std = Standard Deviation of a cluster, Dmax = max. distance between 2 centers
- Pass examples through hidden layer and get matrix A.
- Find and update weights using Least Squares Linear Regression: $w = \left(\mathrm{A}^{\mathrm{T}}\mathrm{A}\right)^{-1}\mathrm{A}^{\mathrm{T}}\mathrm{y}$
- Pass test data from RBFNN and get predictions

Radial Basis Function Neural Network

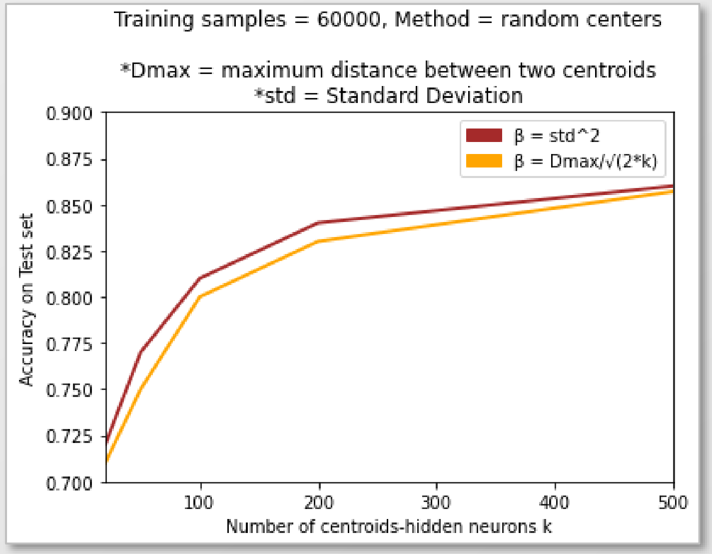
- **Best model's parameters:**

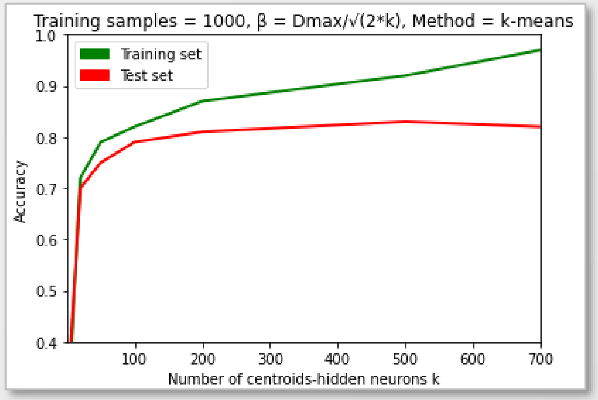
  k = 500 hidden neurons

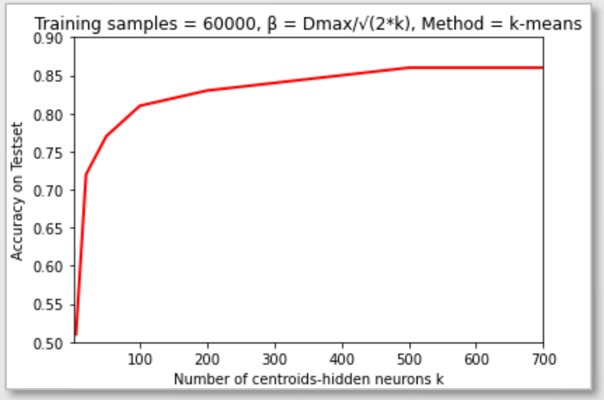
  $$\beta = \frac{1}{std^2}$$

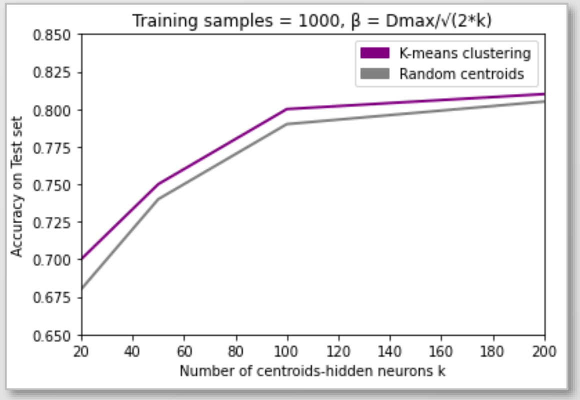
  Method = k-means

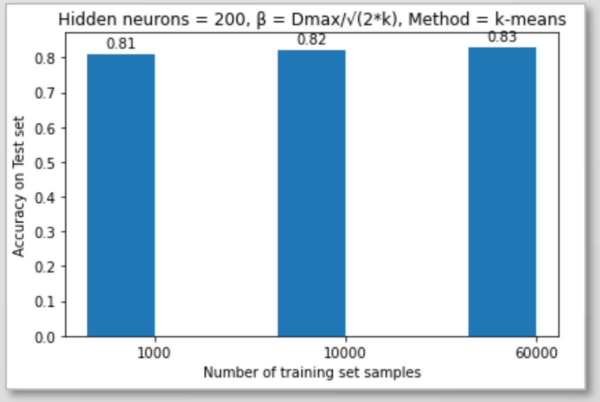
- **Accuracy on Test Set:** 0.86
- **Time:** 7 hours (theoretically way less)











Conclusion & Results

Algorithm	Accuracy	Speed	Speed in Theory
K-NN	86%	~ 9 minutes	Slow Testing
NCC	69%	~ 15 seconds	Fastest but poor results
MLP	88%	~ 3 minutes	Fast
SVM	91%	~ 3 minutes (with PCA)	Average
RBFNN	86%	~ 7 hours (from scratch, not optimized)	Faster than MLP