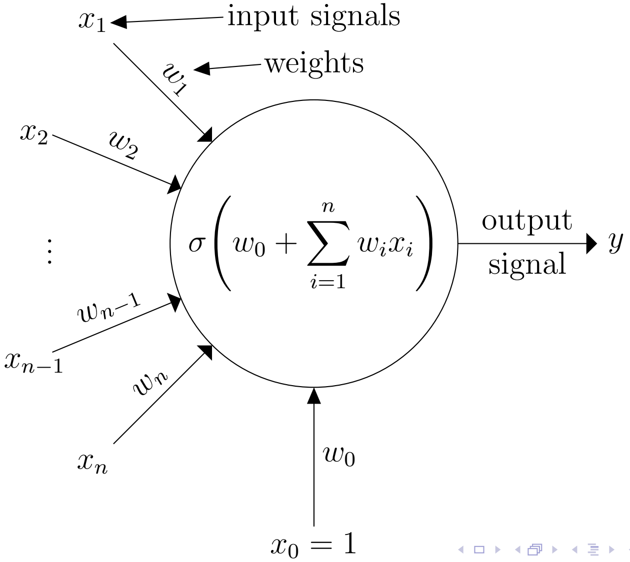
12. Neural Networks

Bruce E. Shapiro

Getting Started in Machine Learning

Last revised: April 22, 2019

The Perceptron



The Perceptron Performs Linear Separation

Example: Two-feature data

- Let $\sigma(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$
- Denote output function by z; input features by x, y; let $n = 2$:

$$z = \sigma\left(w_0 + w_1 x + w_2 y\right)$$

- Decision boundary occurs at $w_0 + w_1 x + w_2 y = 0$, i.e.,

$$y = -\frac{w_0}{w_2} + -\frac{w_1}{w_2} x$$

Learning Rule

- Submit data points $(\mathbf{x}, y)$ to neural network one at a time
  - $\mathbf{x}$ is a feature vector
  - y is the predicted (training) value

Learning Rule

- Submit data points $(\mathbf{x}, y)$ to neural network one at a time
  - $\mathbf{x}$ is a feature vector
  - y is the predicted (training) value
- Compute output as $y_p = \sigma(w_0 + \mathbf{w} \cdot \mathbf{x})$
- Compute prediction error as $\epsilon = y_p - y$

Learning Rule

- Submit data points $(\mathbf{x}, y)$ to neural network one at a time
  - $\mathbf{x}$ is a feature vector
  - y is the predicted (training) value
- Compute output as $y_p = \sigma(w_0 + \mathbf{w} \cdot \mathbf{x})$
- Compute prediction error as $\epsilon = y_p - y$
- Update the weight vector as $\Delta\mathbf{w} = \eta\epsilon\mathbf{x}$
  - Update is proportional to magnitude of input (biology)
  - Update is proportional to magnitude of error
  - η is a small (less than 1) learning rate

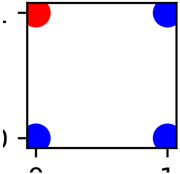
Learning Rule

- Submit data points $(\mathbf{x}, y)$ to neural network one at a time
  - $\mathbf{x}$ is a feature vector
  - y is the predicted (training) value
- Compute output as $y_p = \sigma(w_0 + \mathbf{w} \cdot \mathbf{x})$
- Compute prediction error as $\epsilon = y_p - y$
- Update the weight vector as $\Delta\mathbf{w} = \eta\epsilon\mathbf{x}$
  - Update is proportional to magnitude of input (biology)
  - Update is proportional to magnitude of error
  - η is a small (less than 1) learning rate
- Iterate until there are no further updates
  - May require cycling through input data
  - Final result will depend on initial randomization of weights
  - Result is not unique

Implementation: Two Features

```
data= [((0, 0), 1), ((0, 1), 0), ((1, 0), 1), ((1, 1), 1)]

colors=["red","blue"]
for (x,y),cl in data:
    plt.scatter(x,y,color=colors[cl],s=100)
```

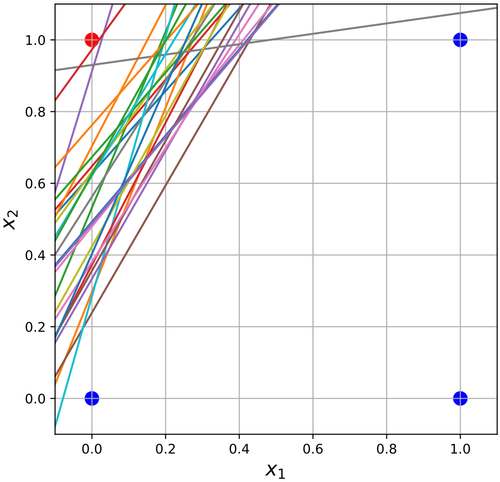


Implementation: Two Features

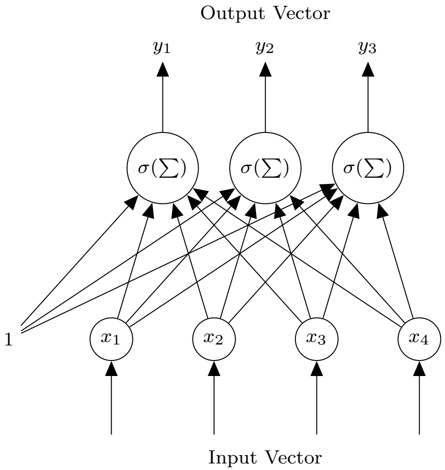
```
def sigma(x): # step function
    return(0 if x < 0 else 1)
```

```
def perceptron(data, steps,eta=.5):
    w=np.random.random(3)
    xvecs = [np.array([1,*x]) for x,y in data]
    yvals = [y for x,y in data]
    for j in range(steps):
        for xvec, y in zip(xvecs,yvals):
            output = sigma(xvec.dot(w))
            err=y-output
            w+=err*eta*xvec
    return(w)
```

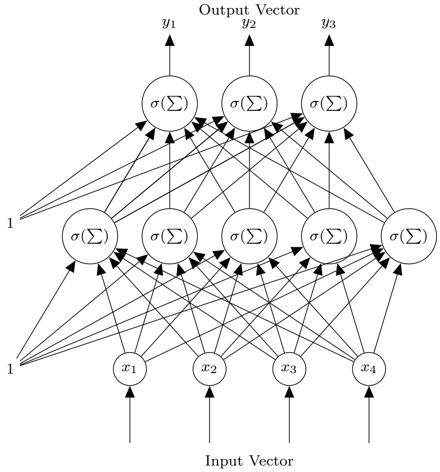
Example With Two Features, 25 Runs



Single Layer Perceptron: Multiple Output



Multilayer Perceptron: Nonlinear Separation Possible



Back-Propagation of Weights

$$\mathcal{E} = \frac{1}{2}\sum(y_i' - y_i)^2$$

Objective Function

Back-Propagation of Weights

$$\mathcal{E} = \frac{1}{2} \sum (y_i' - y_i)^2$$ Objective Function

$$u_j = \sigma\left(\sum_{k=1}^{p} w_{ki} u_k \right)$$ Layer Output

Back-Propagation of Weights

$$\mathcal{E} = \frac{1}{2}\sum(y_i' - y_i)^2$$

Objective Function

$$u_j = \sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$$

Layer Output

$$\frac{\partial \mathcal{E}}{\partial w_{ij}} = \frac{\partial \mathcal{E}}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}}$$

Optimize Weights

Back-Propagation of Weights

$$\mathcal{E} = \frac{1}{2}\sum(y_i' - y_i)^2 \qquad \text{Objective Function}$$

$$u_j = \sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right) \qquad \text{Layer Output}$$

$$\frac{\partial\mathcal{E}}{\partial w_{ij}} = \frac{\partial\mathcal{E}}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}} \qquad \text{Optimize Weights}$$

$$\frac{\partial\mathcal{E}}{\partial u_j} = \frac{\partial}{\partial y_j}\frac{1}{2}\sum(y_i' - y_i)^2 = y_j' - y_j \qquad \text{On output layer } u_j = y_j$$

Back-Propagation of Weights

$$\mathcal{E} = \frac{1}{2} \sum (y_i' - y_i)^2$$ Objective Function

$$u_j = \sigma\left(\sum_{k=1}^{p} w_{ki} u_k \right)$$ Layer Output

$$\frac{\partial \mathcal{E}}{\partial w_{ij}} = \frac{\partial \mathcal{E}}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$ Optimize Weights

$$\frac{\partial \mathcal{E}}{\partial u_j} = \frac{\partial}{\partial y_j} \frac{1}{2} \sum (y_i' - y_i)^2 = y_j' - y_j$$ On output layer $u_j = y_j$

$$\frac{\partial u_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sigma\left(\sum_{k=1}^{p} w_{ki} u_k \right)$$

Back-Propagation of Weights

$$\mathcal{E} = \frac{1}{2}\sum(y_i' - y_i)^2 \qquad \text{Objective Function}$$

$$u_j = \sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right) \qquad \text{Layer Output}$$

$$\frac{\partial\mathcal{E}}{\partial w_{ij}} = \frac{\partial\mathcal{E}}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}} \qquad \text{Optimize Weights}$$

$$\frac{\partial\mathcal{E}}{\partial u_j} = \frac{\partial}{\partial y_j}\frac{1}{2}\sum(y_i' - y_i)^2 = y_j' - y_j \qquad \text{On output layer } u_j = y_j$$

$$\frac{\partial u_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$$

$$\frac{\partial\mathcal{E}}{\partial w_{ij}} = (y_j' - y_j)\frac{\partial\sigma}{\partial w_{ij}}$$

Learning Rule - Output Layers

To find $\dfrac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$, assume that $\sigma(x) = \dfrac{1}{1+e^{-x}}$. Then:

Learning Rule - Output Layers

To find $\dfrac{\partial}{\partial w_{ij}} \sigma\left(\sum_{k=1}^{p} w_{ki} u_k\right)$, assume that $\sigma(x) = \dfrac{1}{1+e^{-x}}$. Then:

$$1 - \sigma(x) = 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1-e^{-x}}$$

Learning Rule - Output Layers

To find $\dfrac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$, assume that $\sigma(x) = \dfrac{1}{1+e^{-x}}$. Then:

$$1 - \sigma(x) = 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1-e^{-x}}$$

$$\frac{\partial\sigma(x)}{\partial x} = \frac{\partial}{\partial x}\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}\sigma(x)}{1+e^{-x}}$$

Learning Rule - Output Layers

To find $\frac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$, assume that $\sigma(x) = \frac{1}{1+e^{-x}}$. Then:

$$
\begin{aligned}
1-\sigma(x) &= 1-\frac{1}{1+e^{-x}} = \frac{e^{-x}}{1-e^{-x}} \\
\frac{\partial\sigma(x)}{\partial x} &= \frac{\partial}{\partial x}\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}\sigma(x)}{1+e^{-x}} \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}
$$

Learning Rule - Output Layers

To find $\frac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$, assume that $\sigma(x) = \frac{1}{1+e^{-x}}$. Then:

$$\begin{aligned}
1-\sigma(x) &= 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1-e^{-x}} \\
\frac{\partial\sigma(x)}{\partial x} &= \frac{\partial}{\partial x}\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}\sigma(x)}{1+e^{-x}} \\
&= \sigma(x)(1-\sigma(x)) \\
\frac{\partial u_j}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right) = u_j(1-u_j)u_i
\end{aligned}$$

Learning Rule - Output Layers

To find $\frac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right)$, assume that $\sigma(x) = \frac{1}{1+e^{-x}}$. Then:

$$
\begin{aligned}
1-\sigma(x) &= 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1-e^{-x}} \\
\frac{\partial\sigma(x)}{\partial x} &= \frac{\partial}{\partial x}\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}\sigma(x)}{1+e^{-x}} \\
&= \sigma(x)(1-\sigma(x)) \\
\frac{\partial u_j}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}\sigma\left(\sum_{k=1}^{p} w_{ki}u_k\right) = u_j(1-u_j)u_i \\
\frac{\partial \mathcal{E}}{\partial w_{ij}} &= \underbrace{(y'_j - y_j)}_{\partial\mathcal{E}/\partial u_j}\ \underbrace{\frac{\partial\sigma}{\partial w_{ij}}}_{\partial u_j/\partial w_{ij}} = y_j(1-y_j)(y'_j - y_j)u_i
\end{aligned}
$$

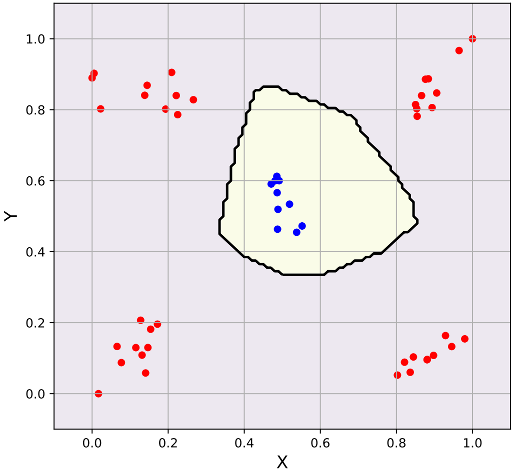
Backprop Learning Rule (summary)

$$\Delta w_{ij} = -\eta \delta_j y_i$$

$$\delta_j = \begin{cases} y_j(1 - y_j)(y'_j - y_j) & \text{for the output layer} \\ y_j(1 - y_j)(\sum \delta_k w_{jk}) & \text{for hidden layers*} \end{cases}$$

*Derivation = Exercise

NonLinear Separation - 150/150/150 Network



Classification Example: Identification of Red Wine

Read Wine Data Files. Files are semi-colon delimited.

```
import pandas as pd
red=pd.read_csv("https://archive.ics.uci.edu/ml/
  machine-learning-databases/wine-quality/
  winequality-red.csv", sep=";")
white=pd.read_csv("https://archive.ics.uci.edu/ml/
  machine-learning-databases/wine-quality/
  winequality-white.csv",sep=";")
```

Classification Example: Identification of Red Wine

Read Wine Data Files. Files are semi-colon delimited.

```
import pandas as pd
red=pd.read_csv("https://archive.ics.uci.edu/ml/
  machine-learning-databases/wine-quality/
  winequality-red.csv", sep=";")
white=pd.read_csv("https://archive.ics.uci.edu/ml/
  machine-learning-databases/wine-quality/
  winequality-white.csv",sep=";")
```

Convert to numpy arrays, after removing `quality` column:

```
import numpy as np
XRED=np.array(red.drop(columns=["quality"]))
XWHITE=np.array(white.drop(columns=["quality"]))
```

Classification Example: Identification of Red Wine

Read Wine Data Files. Files are semi-colon delimited.

```
import pandas as pd
red=pd.read_csv("https://archive.ics.uci.edu/ml/
  machine-learning-databases/wine-quality/
  winequality-red.csv", sep=";")
white=pd.read_csv("https://archive.ics.uci.edu/ml/
  machine-learning-databases/wine-quality/
  winequality-white.csv",sep=";")
```

Convert to numpy arrays, after removing **`quality`** column:

```
import numpy as np
XRED=np.array(red.drop(columns=["quality"]))
XWHITE=np.array(white.drop(columns=["quality"]))
```

Define X and Y data arrays for sklearn:

```
X=np.vstack([XRED,XWHITE])
Y=[*(len(XRED)*[1]),*(len(XWHITE)*[0])]
```

Build, Train, and Test 3-4-4 Neural Network

Train Network:

```
from sklearn.neural_network import MLPClassifier as ANN
from sklearn.model_selection import train_test_split
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
r = ANN(solver='lbfgs', alpha=1e-5,
          hidden_layer_sizes=(3,4,4), random_state=1)
r.fit(XTRAIN,YTRAIN)
```

Build, Train, and Test 3-4-4 Neural Network

Train Network:

```
from sklearn.neural_network import MLPClassifier as ANN
from sklearn.model_selection import train_test_split
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
r = ANN(solver='lbfgs', alpha=1e-5,
          hidden_layer_sizes=(3,4,4), random_state=1)
r.fit(XTRAIN,YTRAIN)
```

Make predictions using test data:

```
YP=r.predict(XTEST)
```

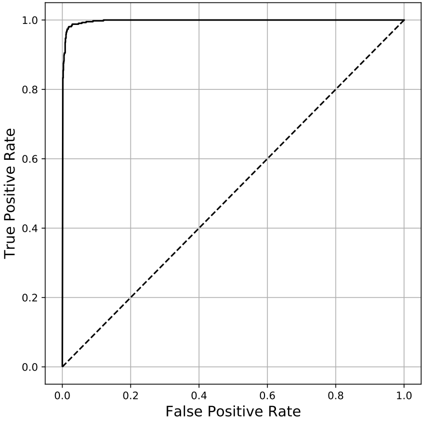
Evaluate predictions:

```
from sklearn.metrics import recall_score, \
  precision_score, roc_auc_score, accuracy_score,\
  confusion_matrix
print(confusion_matrix(YTEST,YP))
print("Accuracy= ",accuracy_score(YTEST,YP))
print("Recall=   ",recall_score(YTEST,YP))
print("Precision=",precision_score(YTEST,YP) )
```

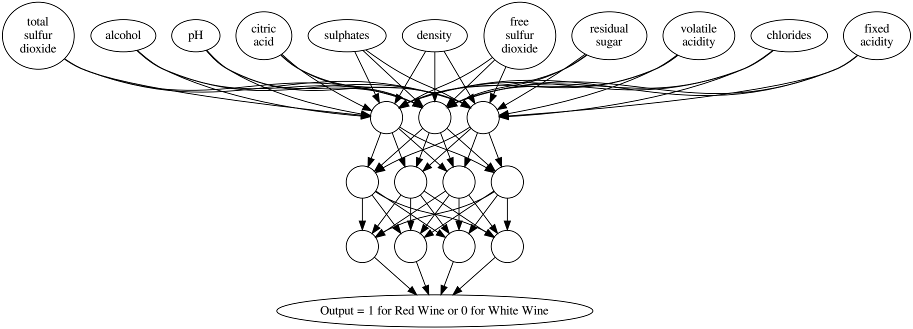
```
[[1172   25]
 [  32  396]]
Accuracy=  0.9649230769230769
Recall=    0.9252336448598131
Precision= 0.9406175771971497
```

ROC Curve for ANN Wine Comparison

```
YPROB=r.predict_proba(XTEST)[:,1]
fpr, tpr, threshold = roc_curve(YTEST,YPROB)
plt.plot(fpr,tpr, c="k")
# ... other plotting commands omitted (see notebook)
```



Visualization with Graphviz



- Wine data has many features:

```
print(red.columns)
```

```
Index(['fixed acidity', 'volatile acidity',
 'citric acid', 'residual sugar',
 'chlorides', 'free sulfur dioxide',
 'total sulfur dioxide', 'density',
 'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

- The ANN does not tell us how important each of these features are relative to one another.
- We can use logistic regression to compare the features.

- Define a function to perform logistic regression on a single feature, and return the X and Y data points for the ROC curve for that feature

```
from sklearn.linear_model import LogisticRegression as LR
def LRC(column, X,Y):
    x=X[:,column].reshape(-1,1)
    xtrain,xtest,ytrain,ytest=train_test_split(x,Y)
    model=LR().fit(xtrain,ytrain)
    YP=model.predict_proba(xtest)[:,1]
    FPR,TPR,TH=roc_curve(ytest,YP)
    return(FPR,TPR)
```

- continued ...

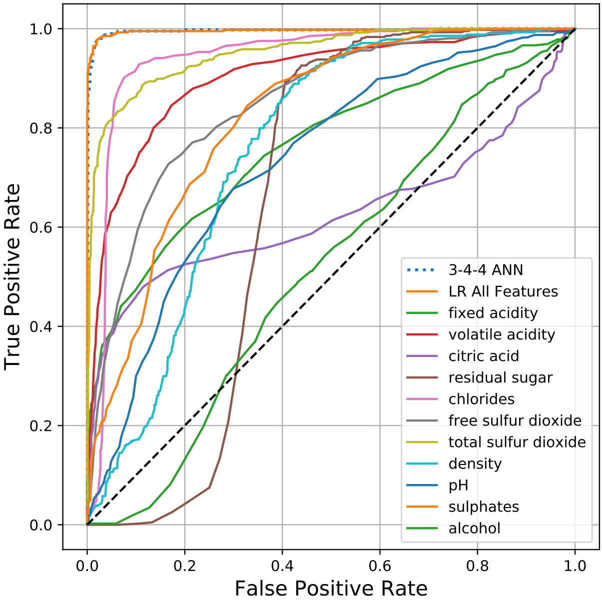
- In a single cell
    - Do single feature logistic regression for each column
    - Do multi-feature logistic regression
    - Plot the ANN and all logistic regression ROC curves

```
# single feature plots
names=red.columns()
for j in range(11):
    FPR,TPR=LRC(j,X,Y)
    plt.plot(FPR,TPR,label=names[j])

# ANN ROC curve (saved from earlier work)
plt.plot(fpr,tpr, ls=":",lw=2,label="3-4-4 ANN")

# multifeature neural net
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
lrmodel=LR().fit(XTRAIN,YTRAIN)
YPLR=lrmodel.predict_proba(XTEST)[:,1]
LRfpr, LRtpr, LRthreshold = roc_curve(YTEST, YPLR)
plt.plot(LRfpr,LRtpr,label="LR All Features")

# ... additional plotting functions omitted ...
```



1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
3-4-4 ANN
LR All Features
fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol

ANN for Regression

- Read MPG data file

```
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
  machine-learning-databases/auto-mpg/auto-mpg.data",
  header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight",
  "accel","model","origin","carname")
data = data.dropna(axis=0)
```

- Define X and Y data sets

```
X=data[["cyl", "displ","hp","weight","accel"]]
Y=data["mpg"]
```

- continued ...

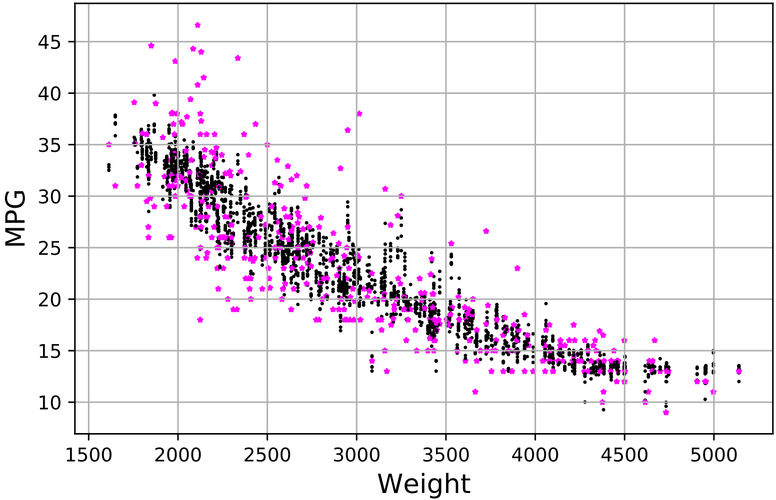
- Libraries:

```
from sklearn.neural_network import MLPRegressor as REG
from sklearn.preprocessing import MinMaxScaler
```

- Data should be scaled to unit length
- Code block for a single run:

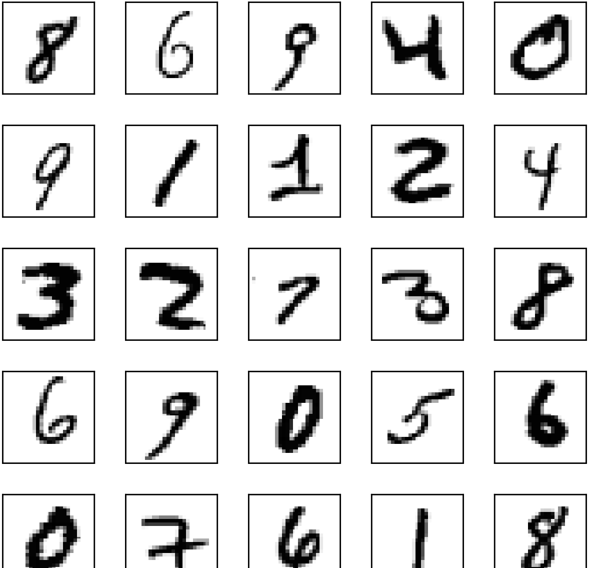
```
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
    XTRAINT=scaler.fit_transform(XTRAIN)
    XTESTT=scaler.transform(XTEST)
        r = REG(solver='lbfgs', alpha=1e-5,
                random_state=1,
        hidden_layer_sizes=(5,5,5))
    r.fit(XTRAINT,YTRAIN)
    YP=r.predict(XTESTT)
    #
    # .. ? plot point or calculate error
```

- Repeat many times ...



black:predictions; magenta: observations

Multiclass Data - MNIST Digits

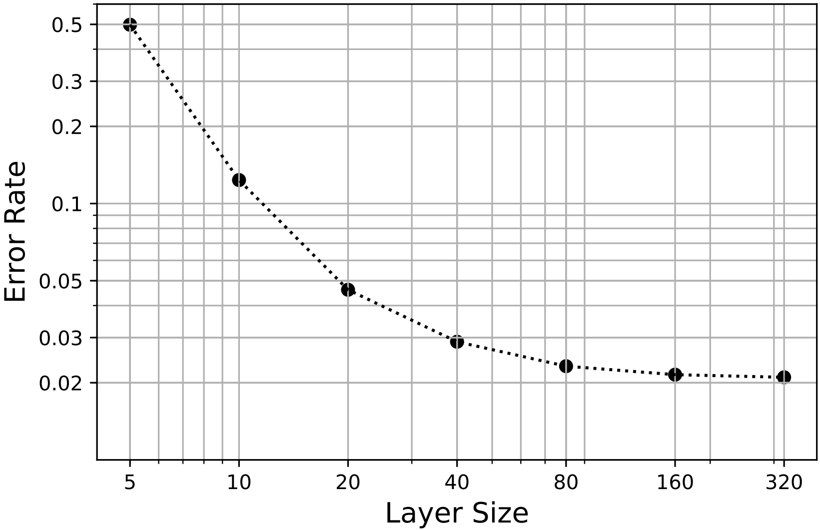


- 60000 Training images, 10000 Test images
- Each image is 28x28 pixels
- To use in an ANN convert to a $28^2 = 784$ element array of pixels.
- See notebook for code to read and convert data files

Example for 10-10-10 ANN

```
r = ANN(solver='lbfgs', alpha=1e-5,
          hidden_layer_sizes=(10,10,10), random_state=1)
r.fit(XTRAINT,YTRAIN)
YP=r.predict(XTESTT)
print(confusion_matrix(YTEST,YP))
print("Accuracy= ",accuracy_score(YTEST,YP))
```

```
[[ 944    0    1    3    2   12   12    3    1    2]
 [   0 1091    5   10    1    2    1    1   23    1]
 [  13    1  902   48   14    1   16   17   15    5]
 [   7    4   30  869    1   48    2   14   25   10]
 [   2    2    5    0  878    2   18    0    7   68]
 [  17    0   12   68   15  684    9    8   65   14]
 [  24    1   12    0   13   15  880    0   13    0]
 [   5   11   22    7    2    1    0  918    0   62]
 [   4   21    9   27   33   89   13    2  764   12]
 [   8    6    1    6   97   23    3   21    9  835]]
Accuracy=  0.8765
```


0.5
0.3
0.2
0.1
0.05
0.03
0.02
Error Rate
5
10
20
40
80
160
320
Layer Size

References

1. Wine and MPG data from: Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository http://archive.ics.uci.edu/ml. Irvine, CA: University of California, School of Information and Computer Science.
2. LeCun, Y et. al. (1998) Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11):2278-2324 (MNIST Paper) Data files at http://yann.lecun.com/exdb/mnist/.