Neural Nets

Data Mining for Business Analytics in Python

Shmueli, Bruce, Gedeck & Patel

 rev 11/27/19

Import Functionality Needed

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.neural_network import MLPClassifier

from dmba import classificationSummary
```

Neural Networks

- A flexible data-driven method
- Can be used for classification, prediction and feature extraction
- It is the basis for deep learning
- It is a powerful technique that lies behind many artificial intelligence applications such as image and voice recognition
- Overfitting is a major danger with NN

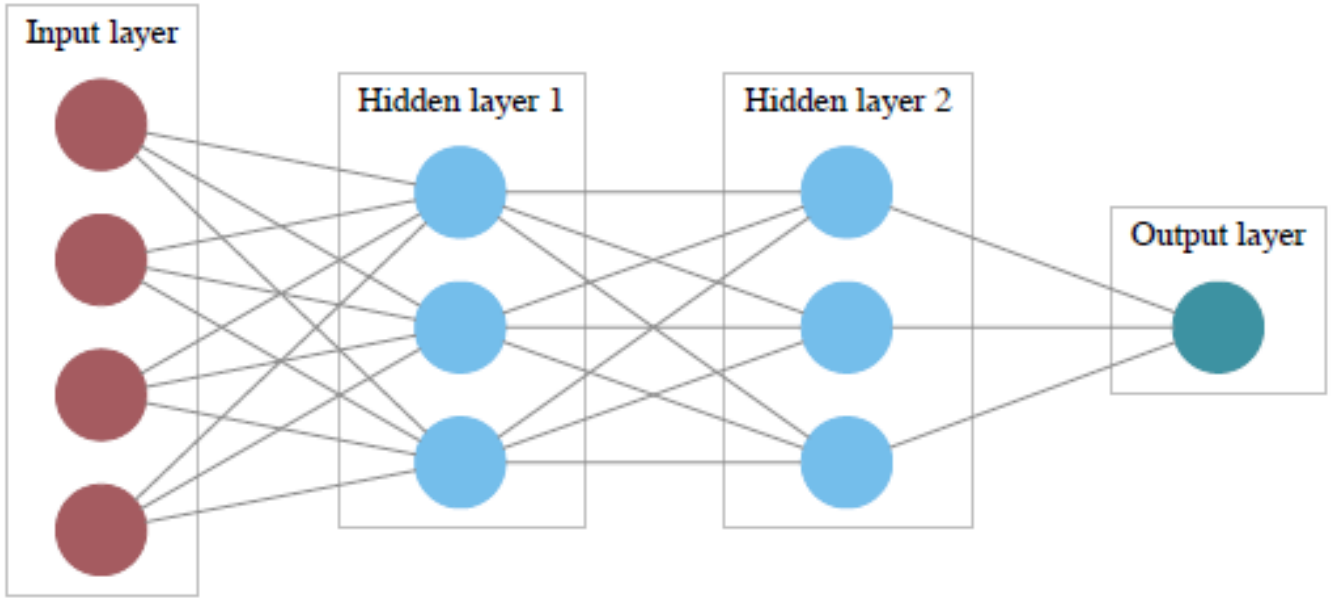
Basic Idea

- Combine input information in a complex & flexible neural net “model”
- Model “coefficients” are continually tweaked in an iterative process
- The network’s interim performance in classification and prediction informs successive tweaks

Network Structure

- Multiple layers
  - Input layers (raw observations)
  - Hidden layers
  - Output layer
- Nodes
- Weights (like coefficients, subject to iterative adjustment)
- Bias values (also subject to iterative adjustment)

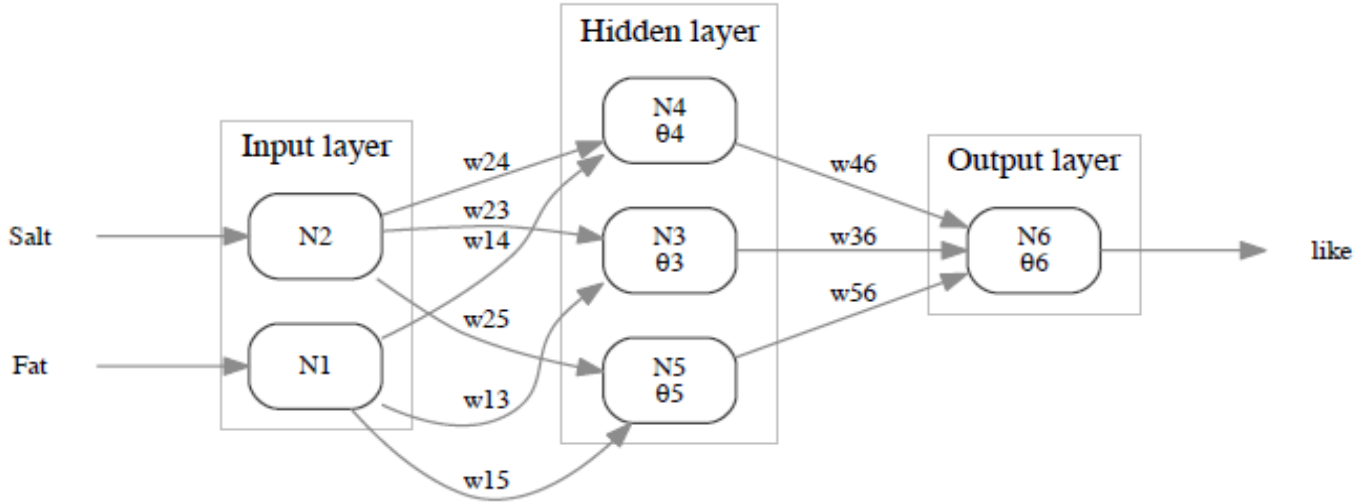
Schematic Diagram



Tiny Example

Using fat & salt content to predict consumer acceptance of cheese

Obs.	Fat Score	Salt Score	Opinion
1	0.2	0.9	like
2	0.1	0.1	dislike
3	0.2	0.4	dislike
4	0.2	0.5	dislike
5	0.4	0.5	like
6	0.3	0.8	like



Rectangles are nodes, w_{ij} on arrows are weights, and Θ_j are node bias values, a constant that controls the level of contribution of node j.

Moving Through the Network

The Input Layer

- For input layer, input = output
- E.g., for record #1:
  Fat input = output = 0.2
  Salt input = output = 0.9

- Output of input layer = input into hidden layer

The Hidden Layer

- In this example, it has 3 nodes
- Each node receives as input the output of all input nodes
- Output of each hidden node is some function of the weighted sum of inputs

The Hidden Layer

- The function g, also called a transfer function or activation function is some monotone function. Examples include the linear function [g(s) = bs], an exponential function [g(s) = exp (bs)], and a logistic/sigmoidal function [g(s) = $1/(1 + e^{-s})$]. This last function used to be by far the most popular one in neural networks. Its practical value arises from the fact that it has a squashing effect on very small or very large values but is almost linear in the range where the value of the function is between 0.1 and 0.9. Deep learning uses mostly the ReLU (rectified linear unit) activation function or variants of it. This function is identical to the linear function, but set to zero for s < 0. If we use a logistic activation function, we can write the output of node j in the hidden layer as:

$$\text{Output}_j = g\left(\theta_j + \sum_{i=1}^{p} w_{ij}x_i\right) = \frac{1}{1 + e^{-(\theta_j + \sum_{i=1}^{p} w_{ij}x_i)}}.$$

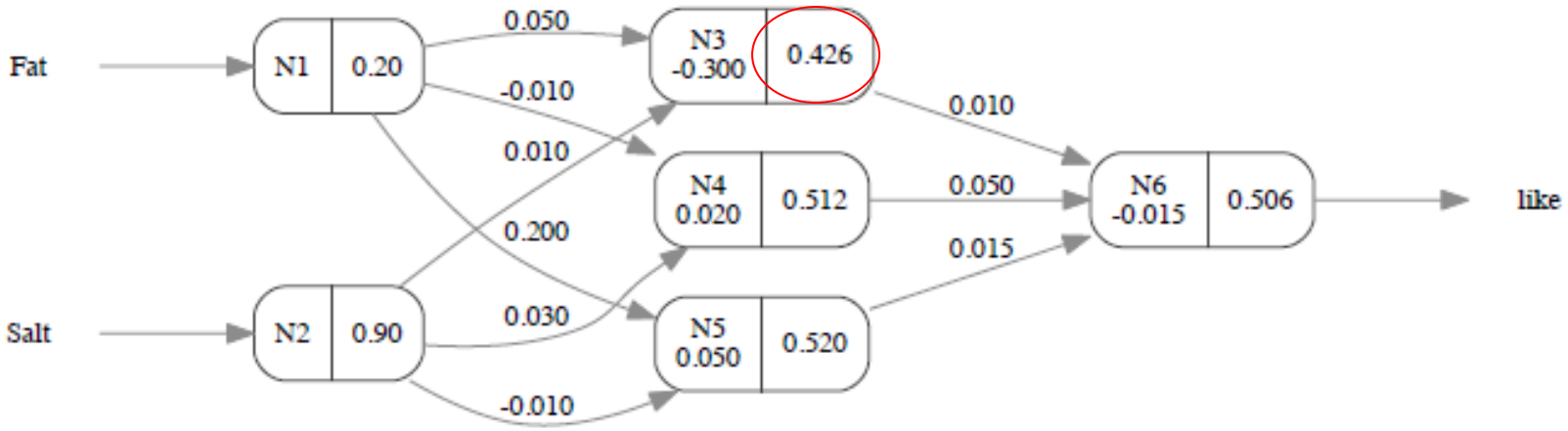
1. **Output**$_j$: This is the probability that the instance belongs to class j (usually between 0 and 1 due to the logistic function).
2. $g(\cdot)$: Represents the logistic (or sigmoid) function, which maps any real-valued number into the range (0, 1).
3. θ_j: This is the bias term for class j.
4. $\sum_{i=1}^{p} w_{ij}x_i$: A weighted sum of the input features x_i, where w_{ij} represents the weight associated with feature x_i for class j.
5. **Logistic Function**: The logistic function $\frac{1}{1+e^{-(\theta_j + \sum_{i=1}^{p} w_{ij}x_i)}}$ squashes the output between 0 and 1, allowing it to be interpreted as a probability.

The Weights

- The weights ϑ (theta) and w are typically initialized to random values in the range -0.05 to +0.05 (around zero)
- Equivalent to a model with random prediction (in other words, no predictive value)
- These initial weights are used in the first round of training

Initial Pass of the Network

Node outputs (on the right within node) using first record in the example, and logistic function



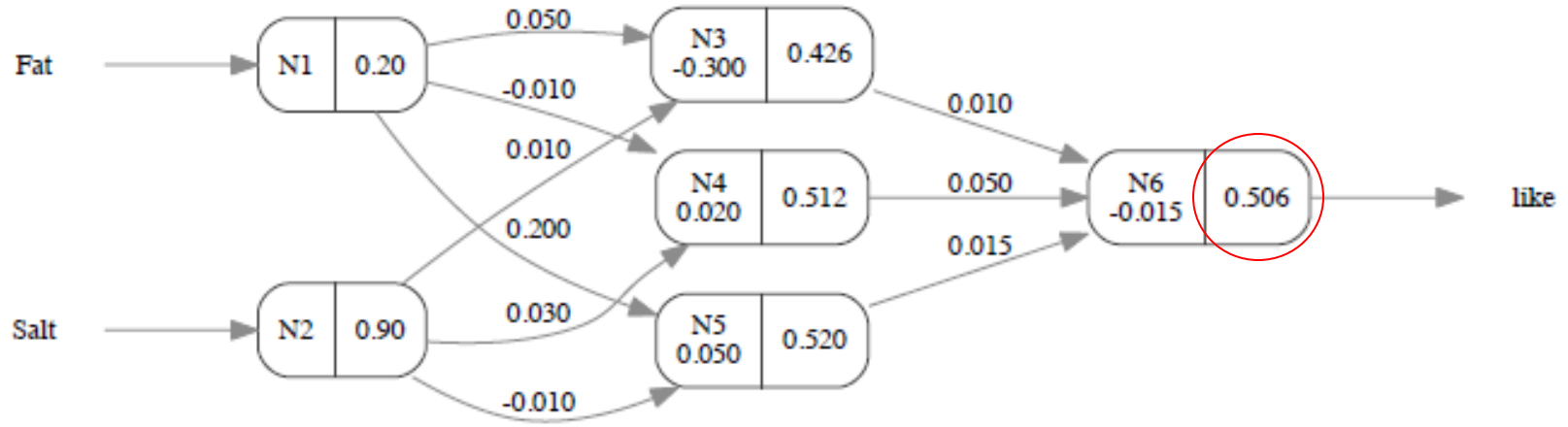
Calculations at hidden node 3:

$$\text{Output}_{N3} = \frac{1}{1 + e^{-[-0.3+(0.05)(0.2)+(0.01)(0.9)]}} = 0.43$$

NOTE: If there is more than one hidden layer, the same calculation applies, except that the input values for the second, third, and so on, hidden layers would be the output of the preceding hidden layer. This means that the number of input values into a certain node is equal to the number of nodes in the preceding layer. (If there was an additional hidden layer in our example, its nodes would receive input from the three nodes in the first hidden layer.)

Output Layer

The output of the last hidden layer becomes input for the output layer



$$\text{Output}_{N6} = \frac{1}{1 + e^{-[-0.015 + (0.01)(0.43) + (0.05)(0.51) + (0.015)(0.52)]}} = 0.506.$$

Mapping the output to a classification

Output = 0.506,

just slightly in excess of 0.5, so classification, at this early stage, is “like”

Relation to Linear Regression

A net with a single output node and no hidden layers, where g is the identity function, takes the same form as a linear regression model:

$$\hat{y} = \Theta + \sum_{i=1}^{p} w_i x_i$$

This is exactly equivalent to the formulation of a multiple linear regression! This means that a neural network with no hidden layers, a single output node, and an identity function g searches only for linear relationships between the outcome and the predictors.

Training the Model

(Estimating θ_j and w_{ij} (bias and weights)
that lead to the best predictive results)

Preprocessing Steps

- Scale variables to 0-1
- Categorical variables
- If equidistant categories, map to equidistant interval points in 0-1 range (Example for four ordinal categories: [0, 0.25, 0.5, 1])
- Otherwise, create m-1 dummy variables
- Transform highly-skewed variables (e.g., log)

Initial Pass Through Network

- Goal: Find weights that yield best predictions
- The process described above is repeated for all records
- At each record compare prediction to actual
- Difference is the error for the output node
- Error is propagated back and distributed to all the hidden nodes and used to update their weights

Back Propagation ("back-prop")

- The most popular method for using model errors to update weights ("learning"). Errors are computed from the last layer (the output layer) back to the hidden layers.
- Output from output node k: $\hat{y}_k$
- y_k is 1 or 0 depending on whether the actual class of the observation coincides with node k's label or not
- Error associated with that node:

$$err_k = \hat{y}_k(1 - \hat{y}_k)(y_k - \hat{y}_k)$$

Note: This is similar to the ordinary definition of an error $(y_k - \hat{y}_k)$, multiplied by a correction factor

Error is Used to Update Weights

$$\theta_j^{new} = \theta_j^{old} + l(err_j)$$

$$\mathrm{w}_j^{new} = w_j^{old} + l(err_j)$$

l = Constant between 0 and 1, which controls the amount of change in weights from one iteration to the next. It reflects the "learning rate" or "weight decay parameter".

In our example, the error associated with output node N6 for the first record is $(0.506)(1 - 0.506)(1 - 0.506) = 0.123$. This error is then used to compute the errors associated with the hidden layer nodes, and those bias and weights are updated accordingly using a formula similar to equation on this page. Two methods are called *case updating* and *batch updating*.

Why It Works

- Big errors lead to big changes in weights
- Small errors leave weights relatively unchanged
- Over thousands of updates, a given weight keeps changing until the error associated with that weight is negligible, at which point weights change little

Python Packages for Neural Nets

Most common for basic neural nets:

- `scikit-learn`

For deep learning:

- `tensorflow`
- `keras`
- `pytorch`

Code for Tiny Example

```
example_df = pd.read_csv('TinyData.csv')
predictors = ['Fat', 'Salt']
outcome = 'Acceptance'
X = example_df[predictors]
y = example_df[outcome]
classes = sorted(y.unique())

# Using MLPClassifier (Multi-Layer Perceptron) in scikit-learn
clf = MLPClassifier(hidden_layer_sizes=3, activation='logistic',
      solver='lbfgs', random_state=1)
clf.fit(X, y)
clf.predict(X)
```

- activation='logistic': This parameter determines the activation function used in the hidden layers. 'logistic' typically refers to the logistic sigmoid activation function, which is commonly used in neural networks. Other options include 'relu' (rectified linear unit) and 'tanh' (hyperbolic tangent).
- The abbreviation "LBFGS" stands for "Limited-memory Broyden-Fletcher-Goldfarb-Shanno." It's a popular optimization algorithm used in numerical optimization and machine learning, including neural network training. LBFGS is an iterative optimization method designed to find the minimum of a multivariate function, which is particularly useful for problems like training neural networks.

Another Example: If you want two hidden layers with three neurons in the first layer and two neurons in the second layer use:

```
clf = MLPClassifier(hidden_layer_sizes=(3,2),activation='logistic',solver='lbfgs',random_state=1)
```

Code for Tiny Example

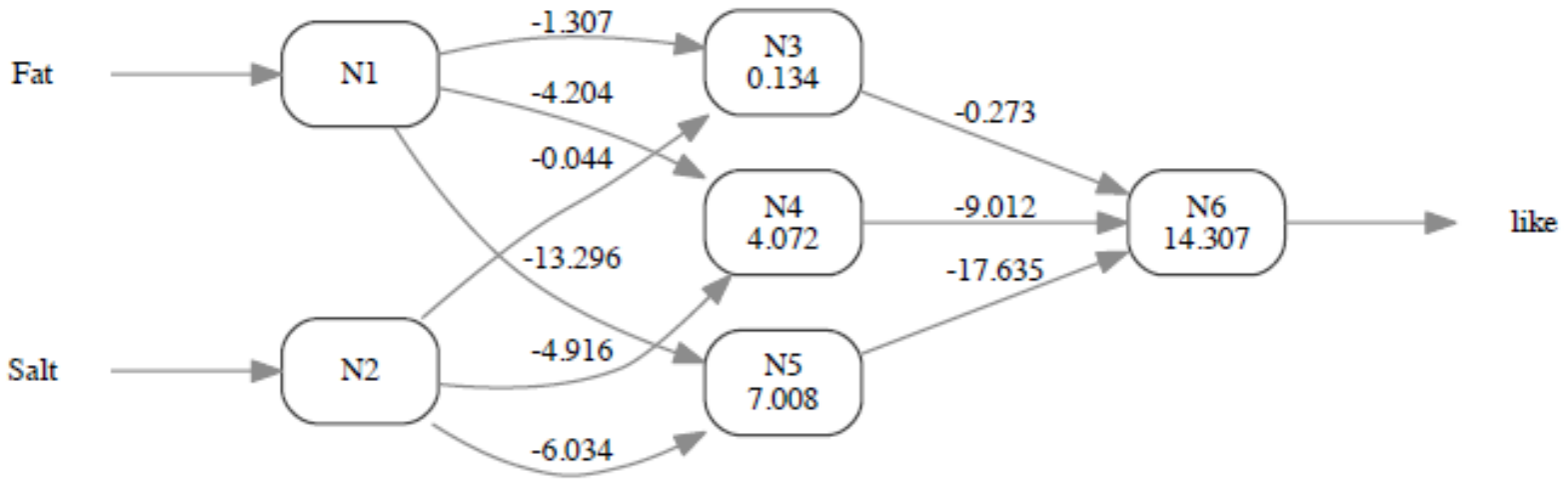
```
# Look at network structure
print('Intercepts') # bias values
print(clf.intercepts_)
print('Weights')
print(clf.coefs_)

Intercepts
[array([0.13368045, 4.07247552, 7.00768104]),
array([14.30748676])]
Weights
[array([
[ -1.30656481, -4.20427792, -13.29587332],
[ -0.04399727, -4.91606924, -6.03356987]
]),
array([
[ -0.27348313],
[ -9.01211573],
[-17.63504694]
])]
```

Predictions

```
# Prediction
print(pd.concat([
    example_df,
    pd.DataFrame(clf.predict_proba(X), columns=classes)
], axis=1))
```

```
       Fat Salt Acceptance   dislike    like
0       0.2 0.9    like      0.000490  0.999510
1       0.1 0.1  dislike     0.999994  0.000006
2       0.2 0.4  dislike     0.999741  0.000259
3       0.2 0.5  dislike     0.997368  0.002632
4       0.4 0.5    like      0.002133  0.997867
5       0.3 0.8    like      0.000075  0.999925
```

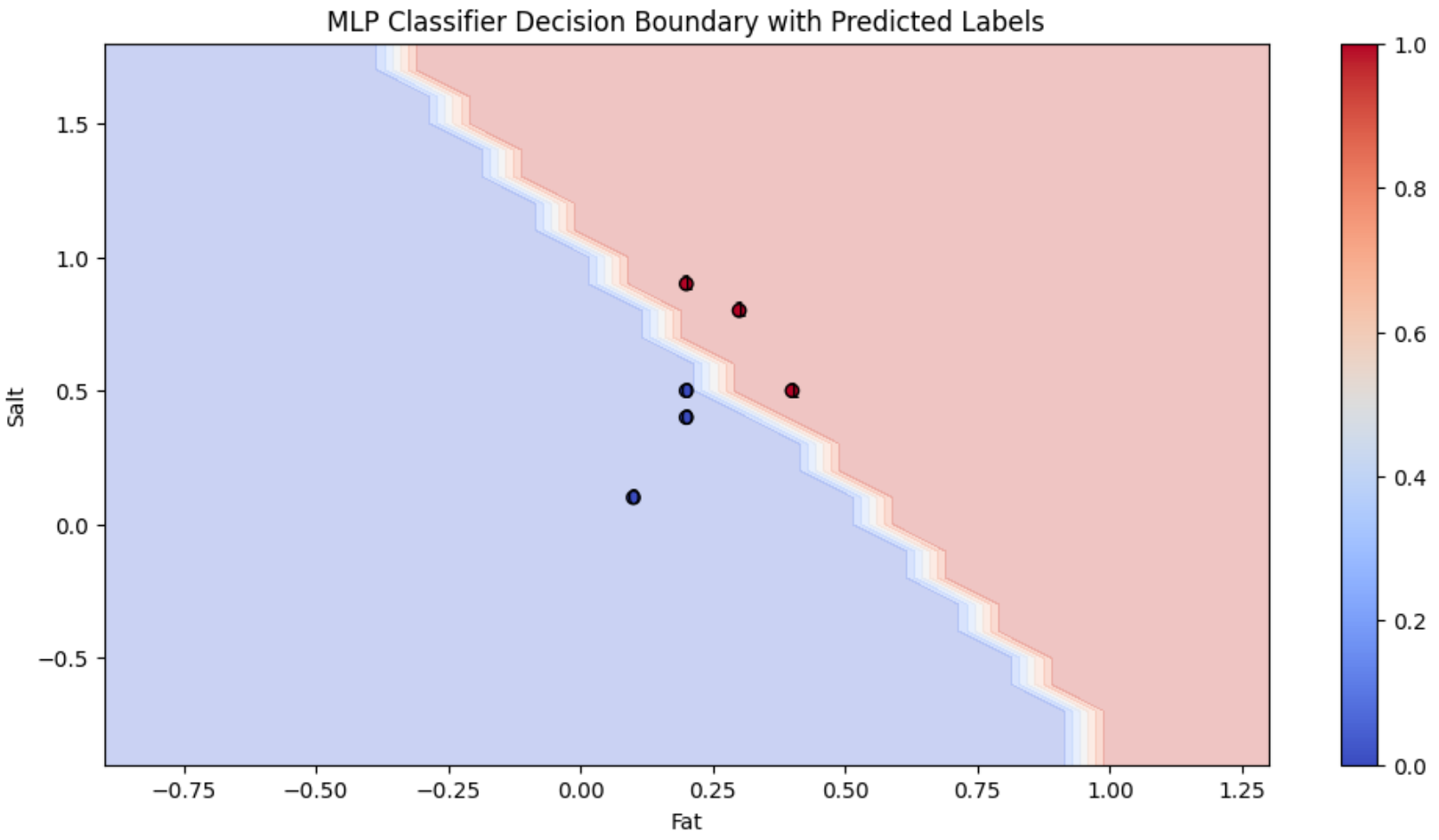
Tiny Example - Final Weights



Tiny Example - Figures

```
import numpy as np; import pandas as pd; import matplotlib.pyplot as plt; from sklearn.neural_network import MLPClassifier; from sklearn.preprocessing import LabelEncoder
example_df = pd.read_csv('TinyData.csv')
predictors = ['Fat', 'Salt']; outcome = 'Acceptance'
X = example_df[predictors]; y = example_df[outcome]
# Step 1: Ensure X is numeric, and encode y if needed
X = X.apply(pd.to_numeric, errors='coerce') # Convert non-numeric values to NaN
# Encode target variable if not already numeric
le = LabelEncoder(); y = le.fit_transform(y)
# Step 2: Check for and handle NaN or infinite values in X and y
if X.isnull().values.any() or np.isinf(X.values).any():
print("NaN or infinite values found in features; handling them.")
X = X.fillna(0) # Replace NaNs with 0 or use another strategy like dropna
# Step 3: Reset index to ensure alignment between X and y if any rows were dropped
X = X.reset_index(drop=True)
y = pd.Series(y).reset_index(drop=True) # Convert y back to Series for compatibility
# Initialize and train the MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(3,), activation='logistic', solver='lbfgs', random_state=1)
clf.fit(X, y)
# Display intercepts and weights
print('Intercepts:', clf.intercepts_)
print('Weights:', clf.coefs_)
# Visualization setup: Create a mesh grid for plotting decision boundaries
x_min, x_max = X['Fat'].min() - 1, X['Fat'].max() + 1
y_min, y_max = X['Salt'].min() - 1, X['Salt'].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
# Predict on the mesh grid to visualize decision boundary
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
# Plot the decision boundary
plt.figure(figsize=(12, 6))
plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)
plt.scatter(X['Fat'], X['Salt'], c=y, marker='o', edgecolor='k', cmap=plt.cm.coolwarm)
# Add predicted values as text annotations
for i, (x_val, y_val) in enumerate(zip(X['Fat'], X['Salt'])):
# Predict using a DataFrame to maintain feature names
input_data = pd.DataFrame([[x_val, y_val]], columns=predictors) # Keep as DataFrame
pred = clf.predict(input_data)[0]
plt.text(x_val, y_val, str(pred), color='black', ha='center', va='center')
plt.xlabel('Fat')
plt.ylabel('Salt')
plt.title("MLP Classifier Decision Boundary with Predicted Labels")
plt.colorbar()
plt.show()
```

Tiny Example - Figures



Common Criteria to Stop the Updating

- When weights change very little from one iteration to the next
- When the misclassification rate reaches a required threshold
- When a limit on runs is reached

Avoiding Overfitting

With sufficient iterations, neural net can easily overfit the data causing the error rate on validation data (and most important, on new data) to be too large.

To avoid overfitting:

- Track error in validation data or via cross-validation
- Limit iterations (using argument max_iter)
- Limit complexity of network

(Reduce the number of neurons in hidden layers as well as number of hidden layers, regularization, higher learning rate for a quicker convergence, early stop, use of automated architecture search methods like neural architecture search-NAS)

User Inputs

Specify Network Architecture

Number of hidden layers

- Most popular – one hidden layer (use argument `hidden_layer_sizes`)

Size of "number of nodes" in hidden layer(s)

- More nodes capture complexity, but increase chances of overfit (use argument `hidden_layer_sizes`)

Number of output nodes

- For classification with m classes, use *m* or *m-1* nodes
- For numerical prediction use one (unless we are interested in predicting more than one function)

Learning Rate (argument `learning_rate`)

- Low values "downweight" the new information from errors at each iteration
- This slows learning, but reduces tendency to overfit to local structure
- Helps tone down the effect of outliers on the weights and avoids getting stuck in local optima.
- Han and Kamber (2001) suggest the more concrete rule of thumb of setting l = 1/(current number of iterations), l = 1 then l = 0.5 , … , to zero

Advantages and Disadvantage

- ✓ Good predictive ability
- ✓ Can capture complex relationships
- ✓ No need to specify a model

- ❖ Considered a “black box” prediction machine, with no insight into relationships between predictors and outcome
- ❖ NN is highly dependent on the quality of its input, so the choice of predictors should be done carefully, using domain knowledge, variable selection, and dimension reduction techniques before using the network
- ❖ Heavy computational requirements if there are many variables (additional variables dramatically increase the number of weights to calculate)

Deep Learning

- Deep learning involves complex networks with many layers, incorporating processes for dimension reduction and feature discovery.
- The statistical and machine learning models in the reference book - including standard neural nets - work where you have informative predictors (purchase information, bank account information, # of rooms in a house, etc.)
- In rapidly-growing applications of voice and image recognition, you have high numbers of “low-level” granular predictors - pixel values, wave amplitudes, uninformative at this low level

Deep Learning

The most active application area for neural nets

- In image recognition, pixel values are predictors, and there might be 100,000+ predictors – big data! (voice recognition similar)
- Deep neural nets with many layers ("neural nets on steroids") have facilitated revolutionary breakthroughs in image/voice recognition, and in artificial intelligence (AI)
- Key is the ability to self-learn features ("unsupervised")
- For example, clustering could separate the pixels in this 1" by 1" football field image into the "green field" and "yard marker" areas without knowing that those concepts exist
- From there, the concept of a boundary, or "edge" emerges
- Successive stages move from identification of local, simple features to more global & complex features

Convolutional Neural Net

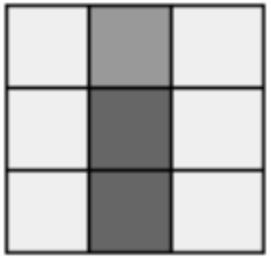
example in image recognition

- Convolution: One key innovation in the area of algorithmic design, associated with deep learning, extending the ability of simple neural networks to perform the tasks of supervised and unsupervised learning
- In a standard neural network, each predictor gets its own weight at each layer of the network. A convolution, by contrast, selects a subset of predictors (pixels), and applies the same operation to the entire subset. It is this grouping that fosters the automated discovery of features. Recall that the data in image recognition tasks consist of a large number of pixel values, which, in a black and white image, range from 0 (black) to 255 (white). Since we are interested in detecting the black lines and shadings, we will reverse this to 255 for black and 0 for white.

Convolutional Neural Net

well-suited for image recognition

- A popular deep learning implementation is a convolutional neural net (CNN)
- Need to aggregate predictors (pixels)
- Rather than have weights for each pixel, group pixels together and apply the same operation: “convolution”
- Common aggregation is a 3 x 3 pixel area, for example the small area around this man’s lower chin



Enlargement of area

25	200	25
25	225	25
25	225	25

Pixel values (higher number = darker)

Apply the Convolution

Convolution operation is “multiply the pixel matrix by the filter matrix” then sum

0	1	0
0	1	0
0	1	0

x

25	200	25
25	225	25
25	225	25

0*25 + 1*200 + 0*25 +
0*25 + 1*225 + 0*25 +
0*25 + 1*225 + 0*25
= 650

Filter matrix that is good at identifying center vertical lines (we will see why shortly)

Pixel values

Sum = 650; this is higher than any other arrangement of the filter matrix, because pixel values are highest in central column. After processing the entire image, the filter produces a feature map (or activation map) composed of all the calculated sum values.

Continue the Convolution

- The filter matrix moves across the image, storing its result, yielding a smaller matrix whose values indicate the presence or absence of a vertical line.
- Similar filters can detect horizontal lines, curves, borders - *hyper-local features*
- Further convolutions can be applied to these local features
- Result: Multi-dimensional matrix, or *tensor*, of higher-level features

Local Feature Map

- The “vertical line detector” filter moves across and down the original image matrix, recalculating and producing a single output each time. We end up with a smaller matrix; how much smaller depends on whether the filter moves one pixel at a time, two, or more. While the original image values were simply individual pixel values, the new, smaller matrix is a map of features, answering the question “is there a vertical line in this section?”

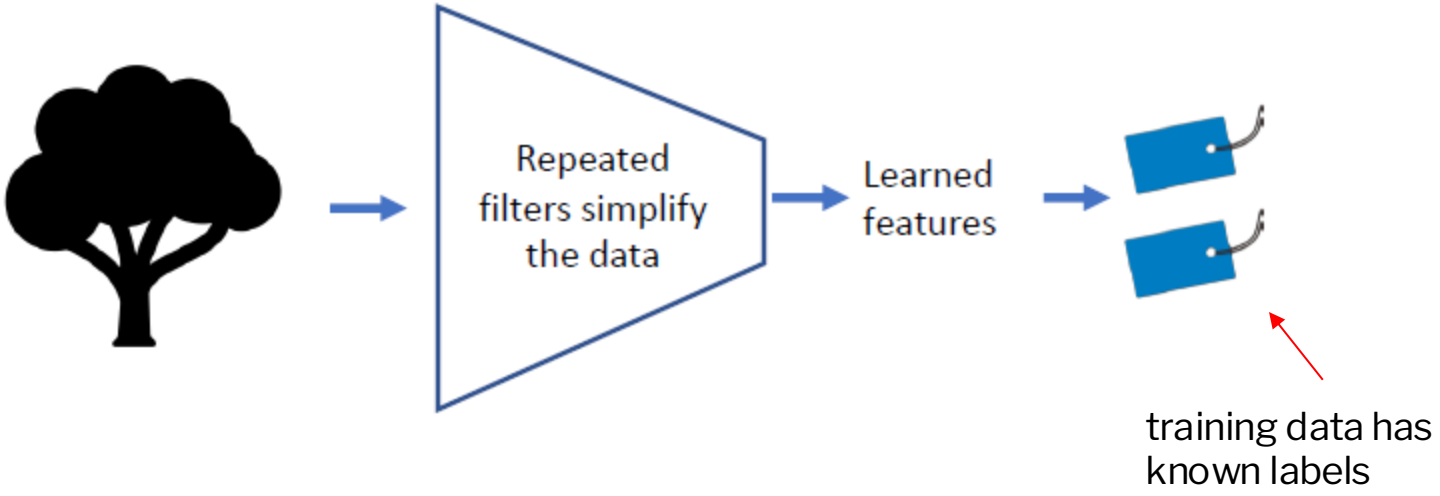
- The fact that the frame for the convolution is relatively small means that the overall operation can identify features that are local in character. We could imagine other local filters to discover horizontal lines, diagonal lines, curves, boundaries, etc. Further layers of different convolutional operations, taking these local feature maps as inputs, can then successively build up higher level features (corners, rectangles, circles, etc.).

- The first feature map is of vertical lines; we could repeat the process to identify horizontal lines and diagonal lines. We could also imagine filters to identify boundaries between light and dark areas. Then, having produced a set of initial low-level feature maps, the process could repeat, except this time working with these feature maps instead of the original pixel values. This iterative process continues, building up multidimensional matrix maps, or tensors, of higher and higher level features. As the process proceeds, the matrix representation of higher level features becomes somewhat abstract, so it is not necessarily possible to peer into a deep network and identify, for example, an eye.

The Learning Process

How does the net learn which convolutions to do?

- In supervised learning, the net retains those convolutions and features which are successful in labeling (tagging) images
- Note that the feature-learning process yields a reduced (simpler) set of features than the original set of pixel values



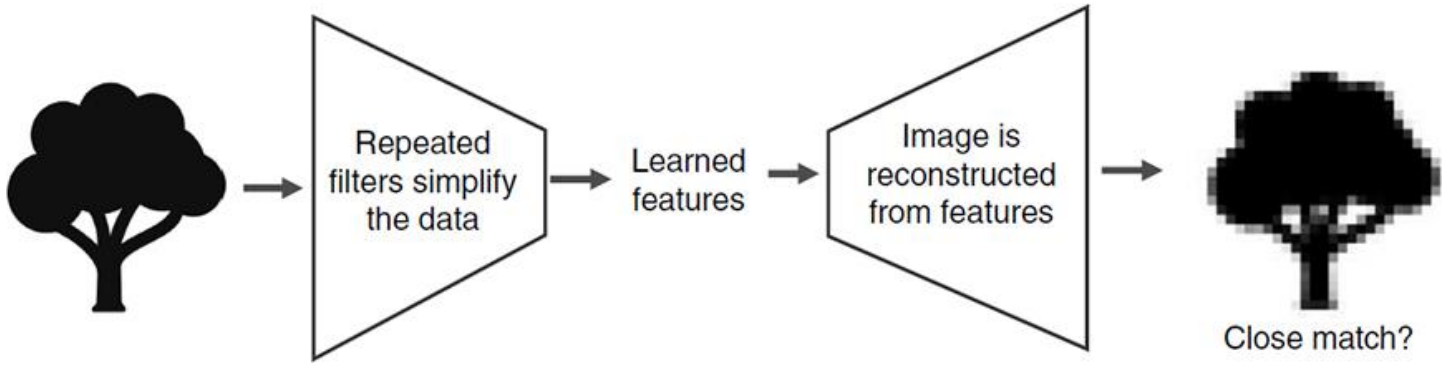
The Learning Process

- In a supervised learning setting, the network keeps building up features up to the highest level, which might be the goal of the learning task. Consider the task of deciding whether an image contains a face. You have a training set of labeled images with faces, and images without faces. The training process yields convolutions that identify hierarchies of features (e.g., edges > circles > eyes) that lead to success in the classification process. Other hierarchies that the net might conceivably encounter (e.g., edges > rectangles > houses) get dropped because they do not contribute to success in the identification of faces. Sometimes it is the case that the output of a single neuron in the network is an effective classifier, an indication that this neuron codes for the feature you are focusing on.

Unsupervised Learning

Autoencoding

- Deep learning nets can learn higher level features even when there are no labels to guide the process
- The net adds a process to take the high level features and generate an image (Famous examples include identifying images with faces, and identifying dogs and cats in images)
- The generated image is compared to the original image and the net retains the architecture that produces the best matches



Unsupervised Learning

Autoencoding

- Up to the learned features point (the bottleneck in the previous image), the network is similar to the supervised network.
- Once it has developed the learned features, which are a low-dimension representation of the data, it expands those features into an image by a reverse process. The output image is compared to the input image, and if they are not similar, the network keeps working (using the same backpropagation method we discussed earlier). Once the network reliably produces output images that are similar to the inputs, the process stops.

Check these out

- https://playground.tensorflow.org/
- Video-1
- Video-2