

Artificial Neural Network (ANN)

by J. Wunderlich Ph.D.

Class Lecture and suggested semester project option

After reviewing everything below, and listening to the accompanying lectures in class, you may want to use Matlab and the deep learning tool box discussed below and available on Elizabethtown College computers, or just using the principles discussed below implemented by you in some programming language, to build an ANN for your own data and problem definition. You should not use the exact forest-fire predicting example below, however you can create the prediction or inference ANN for real estate prices as discussed in the first video. And if you build on something you find in GitHub, make sure you give full credit to the creator of the original code that you tweak.

Most ANN's are trained with INPUT VARIABLES ("FEATURES" , "STIMULUS") and corresponding OUTPUT OBSERVATIONS ("DESIRED RESPONSES"). A SHALLOW ANN has one hidden layer. A **DEEP** ANN has multiple layers.

Artificial Neural Networks (ANN's) can be used for PREDICTION or for INFERENCE once they have been trained:

WATCH(9min): https://www.youtube.com/watch?v=uh_k1jD35K8



[Inference vs. Prediction: An Overview](#)

Subscribe to RichardOnData here:

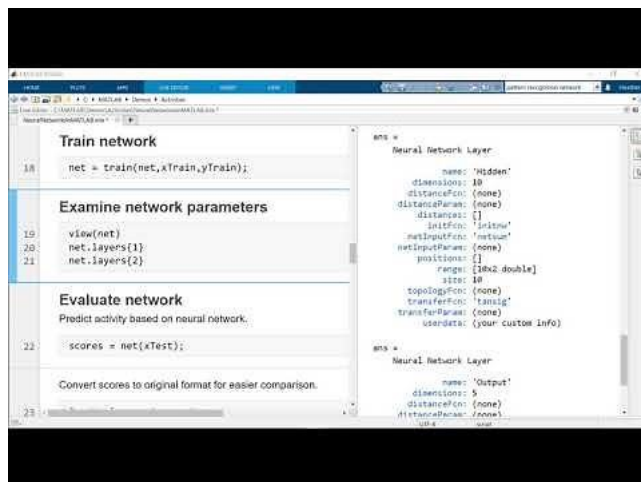
https://www.youtube.com/channel/UCKPyg5gsnt6h0aA8EBw3i6A?sub_confirmation=1 In this video I go over the difference between inference and prediction, in the statistical modeling and machine learning context. It happens all the time - clients have requests to incorporate machine learning and/or statistical ...

www.youtube.com

PREDICTION in above example for real estate, is where you want to predict, like Zillow does, what the price of a house should be on the market based on training an ANN with known data, like present comparable sales of homes in the area, or historical prices of homes sold in the area, based on inputs of training variables like how many bedrooms, how many bathrooms, the crime rate in the area, the distance to a body of water, the square footage of the building, the size of the lot, if there is a pool or hot tub, parking facilities, etc..

INFERENCE in this above example for real estate, would be where the ANN, after it has been trained, can tell you how much your home value will go up based on, for example, adding a bedroom, adding a bathroom, adding a pool or hot tub, adding a garage, if the crime rate were to somehow go down in the area, if you could somehow influence local politics and change the location of a highway offramp or extension of the municipal sewer system, etc. Recall that this type of inference is somewhat different than the way inference is implemented in inference engines and knowledge bases in rule based expert systems in traditional symbolic artificial intelligence with rules, traceable code making decisions, confidence values assigned to rules and user input, etc., we have discussed in other lectures

WATCH(4min) by Mathworks (the company that makes Matlab): <https://www.youtube.com/watch?v=6T2yYTSw8z0>

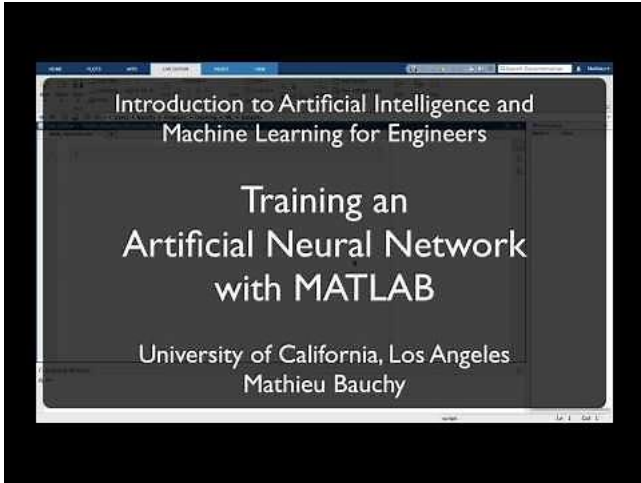


[Getting Started with Neural Networks Using MATLAB](#)

A neural network is an adaptive system that learns by using interconnected nodes. Neural networks are useful in many applications: you can use them for clustering, classification, regression, and time-series predictions. In this video, you'll walk through an example that shows what neural networks are and how to work with them in MATLAB

The following example is for **PREDICTING when forest fires will be likely** (i.e., the **Percentage of a chance of fire**), based on a training set of 1,500 weather data, separated into three different variables: **temperature, humidity, and wind speed**.

WATCH (81min, but only watch first 62min): <https://www.youtube.com/watch?v=xOzh6PMk21I>



[Training an Artificial Neural Network with Matlab – Machine Learning for Engineers](#)

This video is part of the "Artificial Intelligence and Machine Learning for Engineers" course offered at the University of California, Los Angeles (UCLA). This course introduces ML/AI theory and applications that are relevant to engineering, with a focus on practical machine learning for civil engineering. This tutorial illustrates how to use ...

www.youtube.com

Load the raw weather data from the `fire.csv` file (a Comma-Separated-Values file), into a matrix. And for this example, all the input data is originally formatted in columns, but will later need to be converted into rows to fit the input format desired by Matlab functions. See Matlab help for function "readmatrix":

<https://www.mathworks.com/help/matlab/ref/readmatrix.html>

Also define a matrix of inputs "x" with its all the rows, from beginning to the end, designated in the "x = data(:,1:3);" by the ":" by itself, with nothing on either side of the ":", and the columns of this matrix named "data" is being defined by the "1:3" for the three input variables. And the output Y variable (a vector) comes from the fourth column of the input data. The variable "m" is the length of the vector y, which is the number of desired output observations (responses) in this original data set, for later use.:

Import the data

```
data = readmatrix('fire.csv');  
x = data(:,1:3);  
y = data(:,4);  
m = length(y);
```

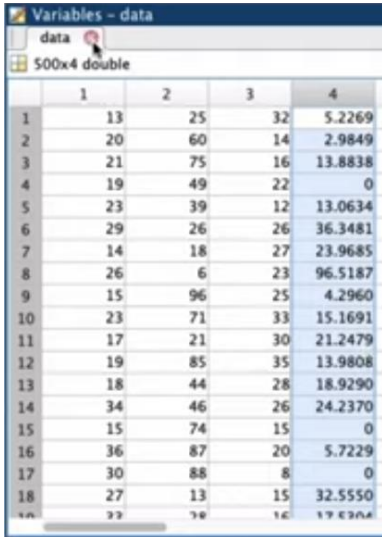
Description

`A = readmatrix(filename)` creates an array by reading column-oriented data from a file. The `readmatrix` function performs automatic detection of import parameters for your file.

`readmatrix` determines the file format from the file extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsx`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

For files containing mixed numeric and text data, `readmatrix` imports the data as a numeric array by default.

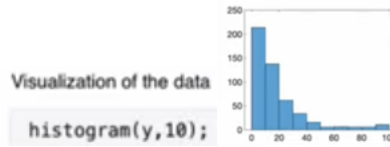


	1	2	3	4
1	13	25	32	5.2269
2	20	60	14	2.9849
3	21	75	16	13.8838
4	19	49	22	0
5	23	39	12	13.0634
6	29	26	26	36.3481
7	14	18	27	23.9685
8	26	6	23	96.5187
9	15	96	25	4.2960
10	23	71	33	15.1691
11	17	21	30	21.2479
12	19	85	35	13.9808
13	18	44	28	18.9290
14	34	46	26	24.2370
15	15	74	15	0
16	36	87	20	5.7229
17	30	88	8	0
18	27	13	15	32.5550

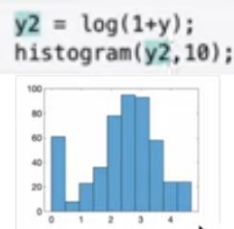
Before training the ANN, it's best to PRECONDITION both the input and output data.

Let's first look at **CONDITIONING THE OUTPUT**:

- First, a histogram (bar chart) is made to look at the raw y data, with "10" in the Matlab function indicating to group the data in "bins" of ten Data points:



- So we see above in the histogram that the bar on the left is very high, indicating that there are many combinations of the three input variables that result in close to 0% chance of a fire; however we see on the far right that there are a number of combinations of input variables that can almost guarantee a fire. This kind of distribution of results is very much skewed to the left and therefore the output would be better if a log form of it was created.
- So, below, we transform "y" onto a LOG scale, and add one ("1") so that there is no error if the value of the original "y" is zero -- i.e., log(0) is an error. A logarithmic scale is a nonlinear scale used when analyzing a large range of quantities. Instead of increasing in equal increments, each interval is increased by a factor of the base of the logarithm; Typically, base ten:



- And now we see a more uniformly distributed distribution of the raw output data, which is a better form of the data for the ANN to learn.

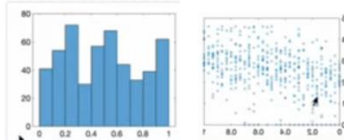
Then look at **CONDITIONING THE INPUTS**:

- We **NORMALIZE** the input data below, for each input variable, by dividing all the inputs of the variable, by the range of values of all of the inputs of that variable, from minimum to maximum (i.e., the Max - the Min). This is done so each of them contributes the same relative range of values to the stimulus of the neural network, otherwise the variable with the largest magnitude could contribute a disproportionate amount.

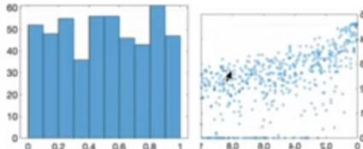
```
for i = 1:3
    x2(:,i) = (x(:,i)-min(x(:,i)))/(max(x(:,i))-min(x(:,i)));
end
histogram(x2(:,1),10);
```

And then do histogram (a bar chart), and x,y plot of each of the three input variables, to visualize the normalized inputs, and LOG-transformed output:

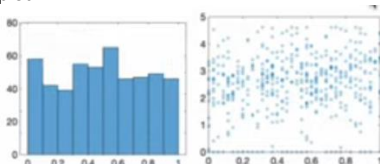
`histogram(x2(:,1),10);` and `plot(x2(:,1),y2,'o');` to see Normalized TEMPERATURE inputs (in Celcius), with no clear visual relationship between x and y showing on the plot. (the 'o' just says plot little circles):



and `histogram(x2(:,2),10);` and `plot(x2(:,2),y2,'o');` to see Normalized RELATIVE HUMIDITY inputs (in Percent), with somewhat of an inverse relationship showing between x and y (i.e., lack of Relative humidity causing a fire) on the plot:



and `histogram(x2(:,3),10);` and `plot(x2(:,3),y2,'o');` to see Normalized WINDSPEED inputs (in KPH), with no clear visual relationship between x and y showing on the plot:



TRAIN the ANN:

Although training sets are typically listed with columns of INPUT VARIABLES ("FEATURES" , "STIMULUS") and a column of OUTPUT OBSERVATIONS ("DESIRED RESPONSES"), Matlab likes everything in rows, so you need to take the transpose of the matrix that contains all of the inputs so that everything is rotated 90° into rows of inputs within the resulting transpose matrix, the single quote (') after the variable is Matlab's way of transposing a matrix:

Train an artificial neural network (ANN)

```
xt = x2';
yt = y2';
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 30/100;
net.divideParam.testRatio = 0/100;
[net,tr] = train(net, xt, yt);
```

Description

`net = fitnet(hiddenSizes)` returns a function fitting neural network with a hidden layer size of `hiddenSizes`.

`net = fitnet(hiddenSizes, trainFcn)` returns a function fitting neural network with a hidden layer size of `hiddenSizes` and training function, sp

The ANN architecture defaults to one hidden layer, and then in this example the user is specifying that there are 10 neurons in the hidden layer (even though that is the default number). However, if you were to instead do:

```
net = fitnet(hiddenSizes, trainFcn)
```

you could pick different training methods, like for example "momentum" where the ANN will drive faster towards Minima on the error surface based on recent previous learning steps.

Also shown above, breaking the initial data into three sets of data:

- The **TRAINING SET** is used to change the weights within the neural network as it learns to compromise such that it satisfies all of the stimulus/desired-response pairs of the training set
- **VALIDATION SET** is used to optimize hyperparameters (for example the number of hidden layers and the number of neurons in each in their). The default neural network has one hidden layer with 10 neurons. This is a **SHALLOW** ANN since it only has one hidden layer. A **DEEP** ANN has multiple layers.
- The **TEST SET**... Which eventually will be taken from the training set, not the validation set, since that was tuned exactly for optimization of the architecture.

"`[net, tr] = train(net, xt, yt);`" above will yield a new "net" function, a trained network! .. that will be created by the "train" function, with the "tr" being a returned Training Record, including:

- a variable "`trainInd`" (see below) which is a vector of 350 elements (Indices) from the 500 original elements in the training set... i.e. the 70% specified above in "`net.divideParam.trainRatio = 70/100;`" that tells this Matlab toolbox program to randomly pick 70% of the original raw data set to be used for training the ANN.
- a variable "`valInd`" (see below) which is a vector of 150 elements (Indices) from the 500 original elements in the training set... i.e. the 30% specified above in "`net.divideParam.valRatio = 30/100;`" that tells this Matlab toolbox program to randomly pick 30% of the original raw data set to be used for validating (optimizing the architecture) of the ANN.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 30/100;
net.divideParam.testRatio = 0/100;

Command Window

trainFcn: 'trainlm'
trainParam: [1x1 struct]
performFcn: 'mse'
performParam: [1x1 struct]
derivFcn: 'defaultderiv'
divideFcn: 'dividerand'
divideMode: 'sample'
divideParam: [1x1 struct]
trainInd: [1x350 double]
valInd: [1x150 double]
```

The following window will open when the "**train**" function is executed:



See more on Matlab training functions, including a way to use GPU's instead of CPU's, here: <https://www.mathworks.com/help/deeplearning/ref/network.train.html#d123e185236>

Now that the ANN has been trained using the TRAINING SET (where here it is a subset of all of the original data set), we can access the performance of this ANN by comparing the actual outputs of the trained ANN for all of the original data set, to all of the outputs listed in the initial data set. This is the **Root_MEAN_SQUARED_ERROR (RME)** that is being minimized as the ANN learns.

This is calculated by taking the square root of the total of each of the differences squared, one by one, between what the trained neural network outputs "yTrain", and the original desired outputs "yTrainTrue" of everything in the training set, evaluating one example at a time, which is what the ".^2" does:

Performance of the ANN network

```
yTrain = net(xt(:,tr.trainInd));
yTrainTrue = yt(tr.trainInd);
sqrt(mean((yTrain - yTrainTrue).^2))
```

ans = 0.7736

But recall that we had transformed the output using $\log(y+1)$ of the original data, and trained the ANN with that, so now we should undo this when comparing the prediction outputs to the desired output. This is done by raising these outputs as an exponent (in base 10) and then subtracting one:

Performance of the ANN network

```
yTrain = exp(net(xt(:,tr.trainInd)))-1;
yTrainTrue = exp(yt(tr.trainInd))-1;
sqrt(mean((yTrain - yTrainTrue).^2))
```

ans = 8.4265

So therefore this neural network can predict the **percentage chance of fire** with only an 8.4265% ERROR.

And when we do the same thing using the validation set, we see an 8.5969% ERROR:

```
yVal = exp(net(xt(:,tr.valInd)))-1;
yValTrue = exp(yt(tr.valInd))-1;
sqrt(mean((yVal - yValTrue).^2))
```

ans = 8.5969

Now redo everything while making the number of neurons in the hidden layer a variable that is optimized:

Optimize the number of neurons in the hidden layer

```

for i = 1:60
    % defining the architecture of the ANN
    hiddenLayerSize = i;
    net = fitnet(hiddenLayerSize);
    net.divideParam.trainRatio = 70/100;
    net.divideParam.valRatio = 30/100;
    net.divideParam.testRatio = 0/100;

    % training the ANN
    [net,tr] = train(net, xt, yt);

    % determine the error of the ANN
    yTrain = exp(net(xt(:,tr.trainInd)))-1;
    yValTrue = exp(yt(tr.valInd))-1;
    yTrainTrue = exp(yt(tr.trainInd))-1;
    yVal = exp(net(xt(:,tr.valInd)))-1;
    rmse_train(i) = sqrt(mean((yTrain - yTrainTrue).^2)) % RMSE of training set
    rmse_val(i) = sqrt(mean((yVal - yValTrue).^2)) % RMSE of validation set
end
    
```

where "rmse_train(i)" and "rmse_val(i)" are vectors created for plots.

Watch the architecture and parameters change as you execute the above code from time index 55:50 to 55:27:

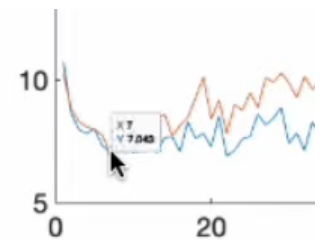
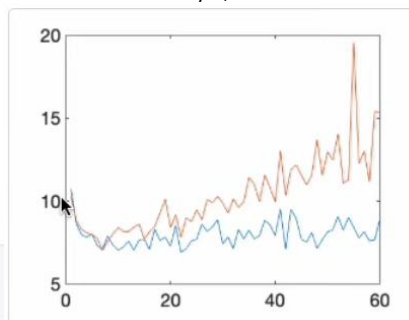


And you will see that there is **UNDERFITTING** when there is not enough neurons in the hidden layer, and **OVERFITTING** there are too many:

Select the optimal number of neurons in the hidden layer

```

plot(1:60,rmse_train); hold on;
plot(1:60,rmse_val); hold off;
    
```



In the graph above (top curve is for validation set) you can see that both the validation set and training set have too high of an error (both 10%) when there are only a couple neurons in the hidden there, and then the error for both of them decreases when there are approximately a half dozen neurons in the hidden layer (the x axis);

and then when the number of hidden neurons gets to 30 or 40, **the difference in validation set error becomes higher than the training set which indicates OVERFITTING, and therefore the network will begin to not be able to predict well when presented with new stimulus input that is not part of the training set.**

And so, for this example it would be better to have an extremely simple architecture with only a few hidden neurons than to have way too many such that the ANN loses its ability to accurately predict when presented new never-seen-before stimulus.

For this architecture the best number of neurons in the hidden layer is seven:

Overfitting in Machine Learning

Overfitting refers to a model that models the training data too well.

Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the model's ability to generalize.

See more on overfitting here: <https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/>

And why we train with noise here: <https://towardsdatascience.com/noise-its-not-always-annoying-1bd5f0f240f>

Now go back and plug the number 7 into the code above:

Train an artificial neural network (ANN)

```
xt = x2';
yt = y2';
hiddenLayerSize = 7;
net = fitnet(hiddenLayerSize);
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 30/100;
net.divideParam.testRatio = 0/100;
[net,tr] = train(net, xt, yt);
```

Performance of the ANN network

```
yTrain = exp(net(xt(:,tr.trainInd))))-1;
yTrainTrue = exp(yt(tr.trainInd))-1;
sqrt(mean((yTrain - yTrainTrue).^2))
yVal = exp(net(xt(:,tr.valInd))))-1;
yValTrue = exp(yt(tr.valInd))-1;
sqrt(mean((yVal - yValTrue).^2))
```

ans = 7.2009

ans = 6.5232

To see improved performance.

For this course stop at time 1:02 in the video.