**What is the difference between Multi Label and Multi-class classification problem?**

Multiclass classificationmeans a classification task with more than two classes; e.g., classify a set of images of fruits which may be oranges, apples, or pears. Multiclass classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time.

Multilabel classification assigns to each sample a set of target labels. This can be thought of as predicting properties of a data-point that are not mutually exclusive, such as topics that are relevant for a document. A text might be about any of religion, politics, finance or education at the same time or none of these.

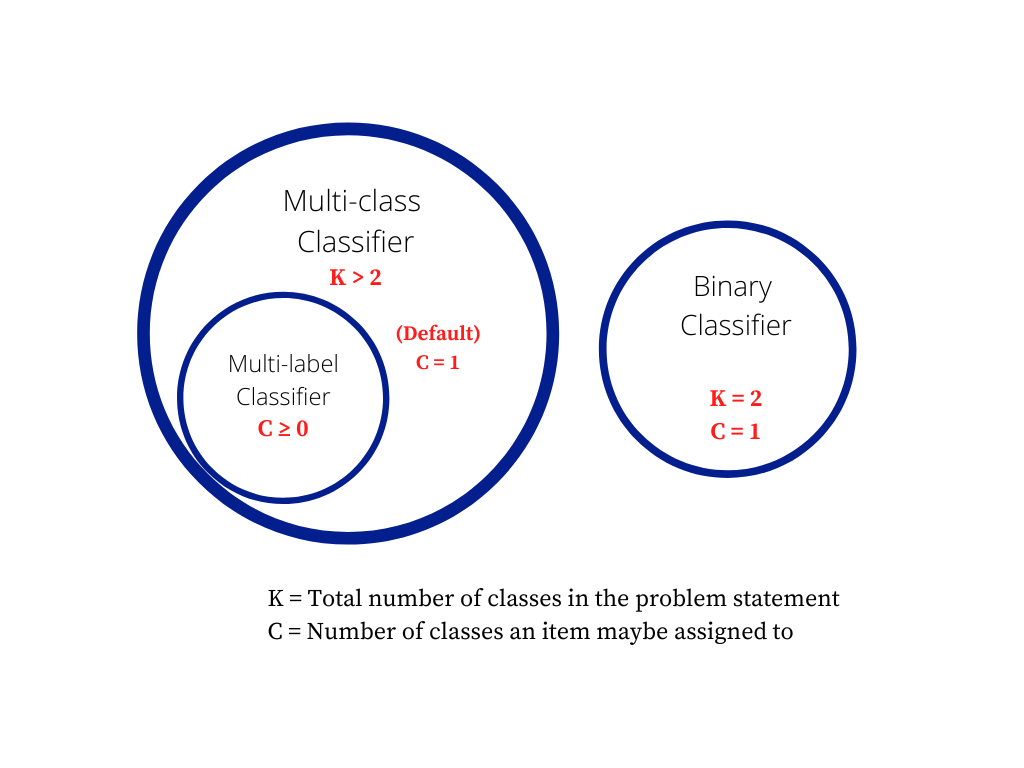
Taken from http://scikit-learn.org/stable/modules/multiclass.html

Look at these contrasts:

* Multi-class vs Binary-class is the question of the number of classes your classifier is modeling. In theory, a binary classifier is much simpler than multi-class, so it's important to make this distinction. For example, the Support vector machine (SVM) can trivially learn a hyperplane to separate two classes, but 3 or more classes makes it complex. In the neural networks, we commonly use Sigmoid for binary, but Softmax for multi-class as the last layer of the model.
* Multi-label vs Single-Label is the question of how many classes any object or example can belong to. In the neural networks, if single label is needed we use a single Softmax layer as the last layer, thus learning a single probability distribution that spans across all classes. If the multi-label classification is needed, we use multiple Sigmoids on the last layer, thus learning separate distribution for each class.

Remarks: we combine multilabel with multiclass, in fact, it is safe to assume that all multi-label are multi-class classifiers. When we have a binary classifier (say positive v/s negative classes), we wouldn't usually assign both labels or no-label at the same time! We usually convert such scenarios to a multi-class classifier where classes are one of {positive, negative, both, none}. Hence multi-label AND binary classifier is not practical, and it is safe to assume all multilabel are multiclass.

On the other side, not all Multi-class classifiers are multi-label classifiers and we shouldn't assume it unless explicitly stated.

****

**What should the final activation layer be?**

The probabilities produced by a sigmoid are independent, and are not constrained to sum to one: 0.37 + 0.77 + 0.48 + 0.91 = 2.53. That’s because the sigmoid looks at each raw output value separately.

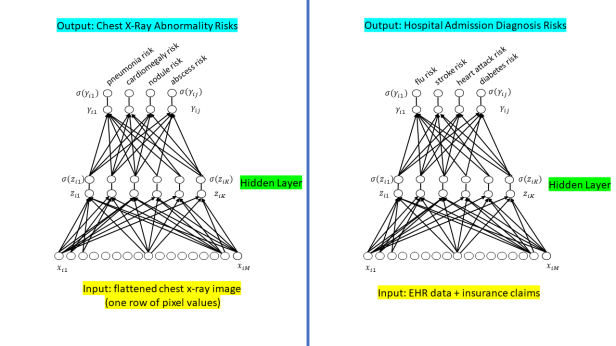
In contrast, the outputs of a softmax are all interrelated. The probabilities produced by a softmax will always sum to one by design: 0.04 + 0.21 + 0.05 + 0.70 = 1.00. Thus, if we are using a softmax, in order for the probability of one class to increase, the probabilities of at least one of the other classes has to decrease by an equivalent amount.

**Sigmoid Examples: Chest X-Rays and Hospital Admission**

Chest X-Rays: A single chest x-ray could show many different medical conditions at the same time. If we build a classifier for chest x-rays, we want that classifier to be able to indicate that multiple conditions are present. Here’s a chest x-ray image showing both pneumonia and abscess, and the corresponding label, which you’ll notice has multiple “ones” in it:

Hospital Admission: Given a patient’s health records, we might want to predict whether that patient will be admitted to the hospital in the future. We can frame this as a classification problem: classify a patient’s past health record according to their future hospital admission diagnoses (if any.) The patient might be admitted for multiple diseases, so there is possibly more than one right answer.

Diagrams: The picture below shows two feedforward neural networks, corresponding to these two classification problems. At the end, a sigmoid function is applied to the raw output values to obtain the final probabilities and allow for more than one correct answer – because a chest x-ray can contain multiple abnormalities, and a patient might be admitted to the hospital for multiple diseases.



**Softmax Examples: Handwritten Digits and Irises**

Handwritten Digits: If we are classifying images of handwritten digits (the MNIST data set), we want to force the classifier to choose only one identity for the digit by using the softmax function. After all, a picture of the number 8 is only the number 8; it cannot be the number 7 at the same time.

***Based on the above explanation the choice for final activation layer for our problem will be Sigmoid***

For a multi-class classification problem we use softmax activation function. It is because we want to maximize the probability of a single class and softmax ensures that the sum of the probabilities is one. However we use sigmoid activation function for the output layer in multi-label classification setting. What sigmoid does is that it allows you to have a high probability for all your classes or some of them, or none of them.

**What should the loss function for a multi class classification problem using neural networks?**

For a multi-class classification problem we often use *categorical\_crossentropy* loss. This is useful since we are interested to approximate the true data distribution (where only one class is true). However in multi label classification setting we formulate the objective function like a binary classifier where each neuron(*y\_train.shape[1]*) in the output layer is responsible for one vs all class classification. *binary\_crossentropy* is suited for binary classification and thus used for multi-label classification.

**How should the multi classes in a multi label of the Y-variable ( Target variable) be encoded for multi label classification problem for neural networks?**

We usually one hot encode our labels for multi-class classification problem. By one hot encoding we represent the categorical variables as binary vectors. We first map categorical values to integer values. Then, each integer value is represented as a binary vector where all values are zero except the index of the integer, which is marked with a 1.

However we know that multi-label classification problem we can have any number of classes associated with it. We strongly assume that the labels are mutually exclusive. Thus instead of one hot encoding we do **multi label binarization**. Here the label (can have multiple classes) is transformed into a binary vector such that all values are zero except the indexes associated for each class in that label, which is marked with a 1.

We can easily implement this as shown below:

from sklearn.preprocessing import MultiLabelBinarizer

# Create MultiLabelBinarizer object

mlb = MultiLabelBinarizer()

# One-hot encode data

mlb.fit\_transform(y)

**How many hidden layers, how many nodes in hidden layers?**

I realize this question has been answered, but I don't think the extant answer really engages the question beyond pointing to a link generally related to the question's subject matter. In particular, the link describes one technique for programmatic network configuration, but that is not a "[a] standard and accepted method" for network configuration.

By following a small set of clear rules, one can programmatically set a competent network architecture (i.e., the number and type of neuronal layers and the number of neurons comprising each layer). Following this schema this will give you a competent architecture but probably not an optimal one.

But once this network is initialized, you can iteratively tune the configuration during training using a number of ancillary algorithms; one family of these works by pruning nodes based on (small) values of the weight vector after a certain number of training epochs--in other words, eliminating unnecessary/redundant nodes (more on this below).

So every NN has three types of layers: input, hidden, and output.

Creating the NN architecture therefore means coming up with values for the number of layers of each type and the number of nodes in each of these layers.

The Input Layer

Simple--every NN has exactly one of them--no exceptions that I'm aware of.

With respect to the number of neurons comprising this layer, this parameter is completely and uniquely determined once you know the shape of your training data. Specifically, the number of neurons comprising that layer is equal to the number of features (columns) in your data. Some NN configurations add one additional node for a bias term.

The Output Layer

Like the Input layer, every NN has exactly one output layer. Determining its size (number of neurons) is simple; it is completely determined by the chosen model configuration.

Is your NN going running in Machine Mode or Regression Mode (the ML convention of using a term that is also used in statistics but assigning a different meaning to it is very confusing). Machine mode: returns a class label (e.g., "Premium Account"/"Basic Account"). Regression Mode returns a value (e.g., price).

If the NN is a regressor, then the output layer has a single node.

If the NN is a classifier, then it also has a single node unless softmax is used in which case the output layer has one node per class label in your model.

The Hidden Layers

So those few rules set the number of layers and size (neurons/layer) for both the input and output layers. That leaves the hidden layers.

How many hidden layers? Well if your data is linearly separable (which you often know by the time you begin coding a NN) then you don't need any hidden layers at all. Of course, you don't need an NN to resolve your data either, but it will still do the job.

Beyond that, as you probably know, there's a mountain of commentary on the question of hidden layer configuration in NNs (see the insanely thorough and insightful [NN FAQ](http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html) for an [excellent summary](http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html) of that commentary). One issue within this subject on which there is a consensus is the performance difference from adding additional hidden layers: the situations in which performance improves with a second (or third, etc.) hidden layer are very few. One hidden layer is sufficient for the large majority of problems.

So what about size of the hidden layer(s)--how many neurons? There are some empirically-derived rules-of-thumb, of these, the most commonly relied on is 'the optimal size of the hidden layer is usually between the size of the input and size of the output layers'. Jeff Heaton, author of [Introduction to Neural Networks in Java](https://www.heatonresearch.com/book/) offers a few more.

In sum, for most problems, one could probably get decent performance (even without a second optimization step) by setting the hidden layer configuration using just two rules: (i) number of hidden layers equals one; and (ii) the number of neurons in that layer is the mean of the neurons in the input and output layers.

Optimization of the Network Configuration

Pruning describes a set of techniques to trim network size (by nodes not layers) to improve computational performance and sometimes resolution performance. The gist of these techniques is removing nodes from the network during training by identifying those nodes which, if removed from the network, would not noticeably affect network performance (i.e., resolution of the data). (Even without using a formal pruning technique, you can get a rough idea of which nodes are not important by looking at your weight matrix after training; look weights very close to zero--it's the nodes on either end of those weights that are often removed during pruning.) Obviously, if you use a pruning algorithm during training then begin with a network configuration that is more likely to have excess (i.e., 'prunable') nodes--in other words, when deciding on a network architecture, err on the side of more neurons, if you add a pruning step.

Put another way, by applying a pruning algorithm to your network during training, you can approach optimal network configuration; whether you can do that in a single "up-front" (such as a genetic-algorithm-based algorithm) I don't know, though I do know that for now, this two-step optimization is more common.

There's one additional rule of thumb that helps for supervised learning problems. You can usually prevent over-fitting if you keep your number of neurons below:

*N**h*=*N**s**/*(*α*∗(*N**i*+*N**o*))

*N**i* = number of input neurons.  
*N**o* = number of output neurons.  
*N**s* = number of samples in training data set.  
*α* = an arbitrary scaling factor usually 2-10.

[Others recommend](http://www.solver.com/training-artificial-neural-network-intro) setting *α*

to a value between 5 and 10, but I find a value of 2 will often work without overfitting. You can think of *α*

as the effective branching factor or number of nonzero weights for each neuron. Dropout layers will bring the "effective" branching factor way down from the actual mean branching factor for your network.

As explained by this [excellent NN Design text](http://hagan.okstate.edu/NNDesign.pdf" \l "page=469), you want to limit the number of free parameters in your model (its [degree](https://stats.stackexchange.com/q/57027/15974) or number of nonzero weights) to a small portion of the degrees of freedom in your data. The degrees of freedom in your data is the number samples \* degrees of freedom (dimensions) in each sample or *N**s*∗(*N**i*+*N**o*)

(assuming they're all independent). So *α*

is a way to indicate how general you want your model to be, or how much you want to prevent overfitting.

For an automated procedure you'd start with an *α*

of 2 (twice as many degrees of freedom in your training data as your model) and work your way up to 10 if the error (loss) for your training dataset is significantly smaller than for your test dataset.

The Number of Hidden Layers

There are really two decisions that must be made regarding the hidden layers: how many hidden layers to actually have in the neural network and how many neurons will be in each of these layers. We will first examine how to determine the number of hidden layers to use with the neural network.

Problems that require two hidden layers are rarely encountered. However, neural networks with two hidden layers can represent functions with any kind of shape. There is currently no theoretical reason to use neural networks with any more than two hidden layers. In fact, for many practical problems, there is no reason to use any more than one hidden layer. Table 5.1 summarizes the capabilities of neural network architectures with various hidden layers.

Table 5.1: Determining the Number of Hidden Layers

| Number of Hidden Layers | Result |

0 - Only capable of representing linear separable functions or decisions.

1 - Can approximate any function that contains a continuous mapping

from one finite space to another.

2 - Can represent an arbitrary decision boundary to arbitrary accuracy

with rational activation functions and can approximate any smooth

mapping to any accuracy.

Deciding the number of hidden neuron layers is only a small part of the problem. You must also determine how many neurons will be in each of these hidden layers. This process is covered in the next section.

The Number of Neurons in the Hidden Layers

Deciding the number of neurons in the hidden layers is a very important part of deciding your overall neural network architecture. Though these layers do not directly interact with the external environment, they have a tremendous influence on the final output. Both the number of hidden layers and the number of neurons in each of these hidden layers must be carefully considered.

Using too few neurons in the hidden layers will result in something called underfitting. Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set.

Using too many neurons in the hidden layers can result in several problems. First, too many neurons in the hidden layers may result in overfitting. Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers. A second problem can occur even when the training data is sufficient. An inordinately large number of neurons in the hidden layers can increase the time it takes to train the network. The amount of training time can increase to the point that it is impossible to adequately train the neural network. Obviously, some compromise must be reached between too many and too few neurons in the hidden layers.

There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers, such as the following:

* The number of hidden neurons should be between the size of the input layer and the size of the output layer.
* The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
* The number of hidden neurons should be less than twice the size of the input layer.

These three rules provide a starting point for you to consider. Ultimately, the selection of an architecture for your neural network will come down to trial and error. But what exactly is meant by trial and error? You do not want to start throwing random numbers of layers and neurons at your network. To do so would be very time consuming. Chapter 8, “Pruning a Neural Network” will explore various ways to determine an optimal structure for a neural network.

**What is epoch vs. iteration in neural network training?**

[**https://stackoverflow.com/questions/4752626/epoch-vs-iteration-when-training-neural-networks**](https://stackoverflow.com/questions/4752626/epoch-vs-iteration-when-training-neural-networks)

**In the neural network terminology:**

* one epoch = one forward pass and one backward pass of all the training examples
* batch size = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
* number of iterations = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

For example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

**\* Model-1 iteration 2 will be with Cross-validation**

**\* Experiment with different model performance metrics**