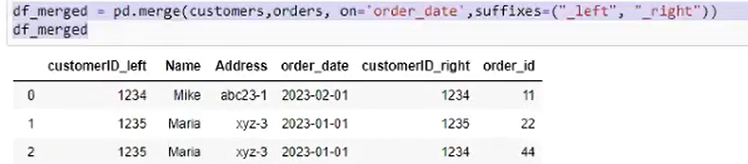
**Lecture # 8 : Tensor Flow by Sir Nasir on 04Mar23**

[**https://www.youtube.com/watch?v=nYmkECchD0c**](https://www.youtube.com/watch?v=nYmkECchD0c)

**Questions from Previous Lectures**

**Q 1: Why we use suffix in the Pandas Merge method >**

In Pandas if we merge two tables on the basis of a column for example roll no is the column in both tables and we are merging on the basis of the roll no column then it will appear only once in the results and rest of the columns will be added in the data frame results. But if there are more than one common columns like rollno and cnic number then in such cases we added suffix while merging on the basis of roll no so that in the result it will show left table column and right table column of cnic. For example :



In the above example we have merged two sets customers and orders on the basis of order\_date and mentioned suffixes as \_left and \_right. So as there is customerid column in both sets so on one set it added customerid\_left and on other id we set customerid\_right.

**Q : What is the Difference between Labeled and Un Labelled Data**

Consider the diabeties dataset. There are columns with different heading that represents different types of values. Now we have to predict on the basis of given data either this patient is diabetic or not. Now there are many columns for input to neuron so we can call then x1 , x2 … and it will predict result which we can call as y. then it will also compare with original y and calculate loss. So if loss is high then system will move back and updates weights to improve the results and again this process is repeated until the loss is according to our requirement. Now this type of data is labeled data as it is mentioned in the column heading what type of values are given in that column.

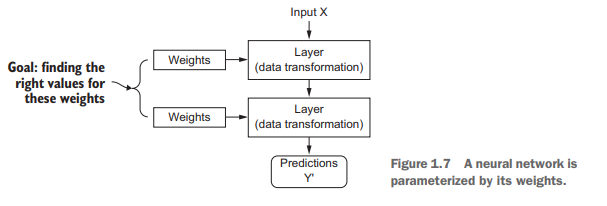
In cases where this label is not available then it is known as un labelled data. In such cases we use algorithms like clustering and many more to do the work.

**Quick Recap of Last Lecture**

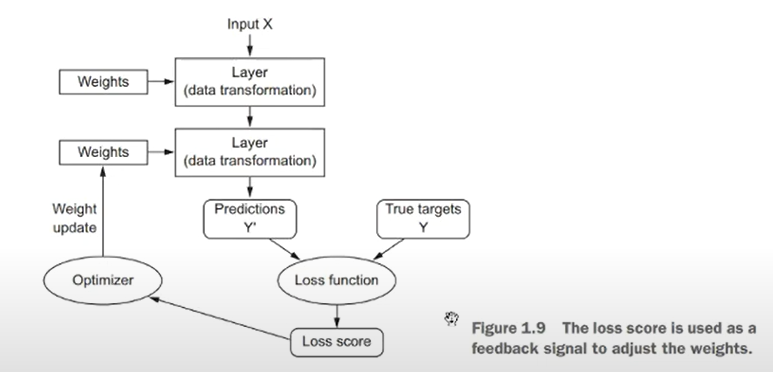
* Understanding how deep learning works, in three figures (Page 9, Topic 1.1.5) of book Deep Learning with Python

machine learning is about mapping inputs (such as images) to targets (such as the label “cat”), which is done by observing many examples of input and targets. You also know that deep neural networks do this input-to-target mapping via a deep sequence of simple data transformations (layers) and that these data transformations are learned by exposure to examples.

*The specification of what a layer does to its input data is stored in the layer’s weights, which in essence are a bunch of numbers.* In technical terms, we’d say that the transformation implemented by a layer is parameterized by its weights. Weights are also sometimes called the parameters of a layer. In this context, learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets. But here’s the thing: a deep neural network can contain tens of millions of parameters. Finding the correct value for all of them may seem like a daunting task, especially given that modifying the value of one parameter will affect the behavior of all the others.



To control something, first you need to be able to observe it. To control the output of a neural network, you need to be able to measure how far this output is from what you expected. *This is the job of the loss function of the network, also called the objective function*. The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example:



*The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score.*  This adjustment is the job of the optimizer, which implements what’s called the *Backpropagation algorithm: the central algorithm in deep learning*.

Optimizer Process : Initially, the weights of the *network are assigned random values*, so the network merely *implements a series of random transformations*. Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high. But with every example the network processes, the weights are adjusted a little in the correct direction, and the loss score decreases.

Note: At start to judge the direction system uses differential equations to resolve either it has to go in forward direction or backward direction and how many steps he needs to take. Step value is normally fixed.

*This is the training loop, which, repeated a sufficient number of times (typically tens of iterations over thousands of examples)*, yields weight values that minimize the loss function. *A network with a minimal loss is one for which the outputs are as close as they can be to the targets: a trained network*. Once again, it’s a simple mechanism that, once scaled, ends up looking like magic.

Brief History of Machine Learning :

* Probabilistic modeling
* Early neural networks
* Kernel methods
* Decision trees, random forests, and gradient boosting machine
* Back to neural network

**Mathematical Building Blocks of Neural Network**

**How to Install TensorFlow**

Start the Jupyter Notebook . From the home page of jupyter notebook Click

New🡪 Terminal Window

From there run the command from the jupyter notebook terminal window : **conda install -c conda-forge tensorflow**

**The architecture we are studying is known as ANN Artificial Neural Network**

**It is a general neural network that can process numbers, text, images, voices , videos and any other data.**

**Now after sometime new neural network were created which were specific for some specific type of data. Which are also generated from the same ANN. For example : To process images use CNN , for text use RNN for generative images use GAN etc**

**By using specialized neural network for that type of data will increase the accuracy .**

**The dataset we are using is free from noises , missing values but if still the result does not come according to our requirements then we can add another layer as**

**Network.add(layers.dense(256,activation=’relu’))**

**To make the network ready for training, we need to pick three more things, as part of the compilation step:**

1. ***A loss function*—How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.**
2. ***An optimizer*—The mechanism through which the network will update itself based on the data it sees and its loss function.**
3. ***Metrics to monitor during training and testing*—Here, we’ll only care about accuracy (the fraction of the images that were correctly classified).**

**Before training, we’ll preprocess the data by reshaping it into the shape the network expects and scaling it so that all values are in the [0, 1] interval. Previously, our training images, for instance, were stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval. We transform it into a float32 array of shape (60000, 28 \* 28) with values between 0 and 1.**

**# So we need to transform our 3d Dataset into 2d having 60k rows and 784 columns**

*train\_images = train\_images.reshape((60000, 28 \* 28))*

*train\_images = train\_images.astype('float32') / 255*

*test\_images = test\_images.reshape((10000, 28 \* 28))*

*test\_images = test\_images.astype('float32') / 255*

# Preparing the Labels

We also need to categorically encode the labels, a step that’s explained in chapter 3.

*from keras.utils import to\_categorical*

*train\_labels = to\_categorical(train\_labels)*

*test\_labels = to\_categorical(test\_labels)*

*# Train the network*

*We’re now ready to train the network, which in Keras is done via a call to the network’s fit method—we fit the model to its training data:*

*network.fit(train\_images, train\_labels, epochs=5, batch\_size=128)*

*Note : Batch size means how much data it should fetch at a time..*

We quickly reach an accuracy of 0.989 (98.9%) on the training data. Now let’s # check that the model performs well on the test set, too:

*test\_loss, test\_acc = network.evaluate(test\_images, test\_labels)*

*print('test\_acc:', test\_acc)*

The test-set accuracy turns out to be 97.8%—that’s quite a bit lower than the training set accuracy. This gap between # training accuracy and test accuracy is an example of overfitting: the fact that machine-learning models tend to perform worse on new data than on their training data.