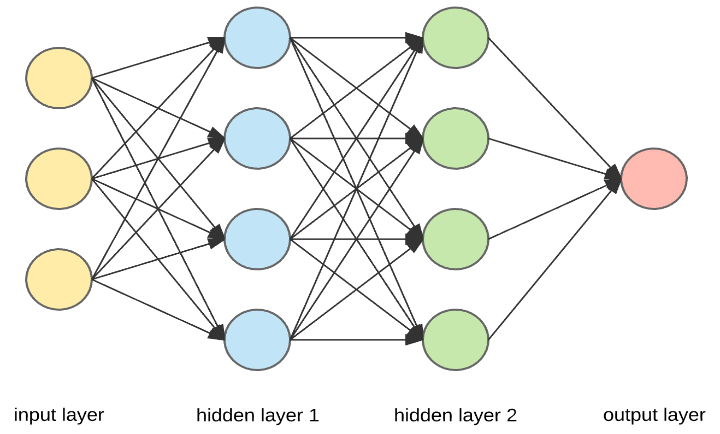
**Programming with R for Data Analysis and Machine Learning – Fall 2020**

**Siddhartha Ireddy, R#11684309, Project**

* **Introduction to Neural network**

Is a machine learning algorithm, which is built on the principle of the organization and functioning of biological neural networks. This concept arose in an attempt to simulate the processes occurring in the brain by Warren McCulloch and Walter Pitts in 1943.

Neural networks consist of individual units called **neurons**. Neurons are located in a series of groups — **layers.** Neurons in each layer are connected to neurons of the next layer. Data comes from the input layer to the output layer along these compounds. Each individual node performs a simple mathematical calculation. Тhen it transmits its data to all the nodes it is connected to.

**

* **Convolutional neural networks and image classification**

Convolutional neural networks (CNN) is a special architecture of artificial neural networks, proposed by Yann LeCun in 1988. CNN uses some features of the visual cortex. One of the most popular uses of this architecture is image classification. For example Facebook uses CNN for automatic tagging algorithms, Amazon — for generating product recommendations and Google — for search through among users’ photos.

In [deep learning](https://en.wikipedia.org/wiki/Deep_learning), a convolutional neural network (CNN, or ConvNet) is a class of [deep neural networks](https://en.wikipedia.org/wiki/Deep_neural_network), most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and [translation invariance](https://en.wikipedia.org/wiki/Translation_invariance) characteristics. They have applications in [image and video recognition](https://en.wikipedia.org/wiki/Computer_vision), [recommender systems](https://en.wikipedia.org/wiki/Recommender_system), [image classification](https://en.wikipedia.org/wiki/Image_classification), [medical image analysis](https://en.wikipedia.org/wiki/Medical_image_computing), [natural language processing](https://en.wikipedia.org/wiki/Natural_language_processing), [brain-computer interfaces](https://en.wikipedia.org/wiki/Brain%E2%80%93computer_interface), and financial [time series](https://en.wikipedia.org/wiki/Time_series).

CNNs are [regularized](https://en.wikipedia.org/wiki/Regularization_(mathematics)) versions of [multilayer perceptron](https://en.wikipedia.org/wiki/Multilayer_perceptron)s. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to [overfitting](https://en.wikipedia.org/wiki/Overfitting) data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

The name “convolutional neural network” indicates that the network employs a mathematical operation called [convolution](https://en.wikipedia.org/wiki/Convolution). Convolutional networks are a specialized type of neural networks that use convolution in place of general matrix multiplication in at least one of their layers

A convolutional neural network consists of an input and an output layer, as well as multiple [hidden layers](https://en.wikipedia.org/wiki/Multilayer_perceptron#Layers). The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other [dot product](https://en.wikipedia.org/wiki/Dot_product). The activation function is commonly a [ReLU layer](https://en.wikipedia.org/wiki/Rectifier_(neural_networks)), and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final [convolution](https://en.wikipedia.org/wiki/Convolution).

1. **Convolutional Layer:**

A convolutional layer within a neural network should have the following attributes:

* Convolutional kernels defined by a width and height (hyper-parameters).
* The number of input channels and output channels (hyper-parameter).
* The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map.

Convolutional layers convolve the input and pass its result to the next layer. [Fully connected feedforward neural networks](https://en.wikipedia.org/wiki/Multilayer_perceptron) can be used to learn features as well as classify data but it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable.

The convolution operation reduces the number of free parameters, allowing the network to be deeper with fewer parameters. By using regularized weights over fewer parameters, the vanishing gradient and exploding gradient problems seen during [backpropagation](https://en.wikipedia.org/wiki/Backpropagation) in traditional neural networks are avoided.

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable [filters](https://en.wikipedia.org/wiki/Filter_(signal_processing)) (or [kernels](https://en.wikipedia.org/wiki/Kernel_(image_processing))), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is [convolved](https://en.wikipedia.org/wiki/Convolution) across the width and height of the input volume, computing the [dot product](https://en.wikipedia.org/wiki/Dot_product) between the entries of the filter and the input and producing a 2-dimensional [activation map](https://en.wikipedia.org/wiki/Activation_function) of that filter. As a result, the network learns filters that activate when it detects some specific type of [feature](https://en.wikipedia.org/wiki/Feature_(machine_learning)) at some spatial position in the input.

Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

**

*Source Google*

1. **Pooling layer:**

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer.

Pooling is a form of non-linear [down-sampling](https://en.wikipedia.org/wiki/Downsampling_(signal_processing)). There are several non-linear functions to implement pooling among which max pooling is the most common. It [partitions](https://en.wikipedia.org/wiki/Partition_of_a_set) the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum.

The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, [memory footprint](https://en.wikipedia.org/wiki/Memory_footprint) and amount of computation in the network, and hence to also control [overfitting](https://en.wikipedia.org/wiki/Overfitting). It is common to periodically insert a pooling layer between successive convolutional layers (each one typically followed by a [ReLU layer](https://en.wikipedia.org/wiki/Convolutional_neural_network#ReLU_layer)) in a CNN architecture.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 down samples at every depth slice in the input by 2 along both width and height, discarding 75% of the activations:

In addition to max pooling, pooling units can use other functions, such as [average](https://en.wikipedia.org/wiki/Average) pooling or [ℓ2-norm](https://en.wikipedia.org/wiki/Euclidean_norm) pooling. While Max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Average pooling uses the average value from each of a cluster of neurons at the prior layer.

Pooling is an important component of convolutional neural networks for [object detection](https://en.wikipedia.org/wiki/Object_detection) based on Fast R-CNN architecture.

1. **Fully connected layer:**

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional [multi-layer perceptron](https://en.wikipedia.org/wiki/Multi-layer_perceptron) neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images. After several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer.

1. **Dataset Used:**

German Traffic Sign Recognition Benchmark(GTSRB) dataset at:

<http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset>

The GTSRB dataset, compiled published by the real-time computer vision research group in Institut für Neuroinformatik, was originally used for a competition of classifying single images of traffic signs. It consists of a training set of 39,209 labeled images and a testing test of 12,630 unlabeled images. The training dataset contains 43 classes i.e 43 types of traffic signs.

The dataset can be downloaded by below link and they represent 43 classes of images:

<http://benchmark.ini.rub.de/Dataset/GTSRB_Final_Training_Images.zip>

* **Methodology**

1. **About Dataset:**

* The image files are in PPM (short for portable pixmap) format.
* The number of images from each class ranges from 210 to 2250.
* Each image contains one traffic sign.
* The sizes of the images are not uniform, ranging from 15\*15 to 250\*250 pixels, and images are not square.
* Images contain a border of up to 10% around the actual sign. Thus the sign is not centered with the image.
* It is an unbalanced multi-class classification problem.

*Chart, bar chart, histogram

Description automatically generated*

*Unbalanced multi-class classification problem.*

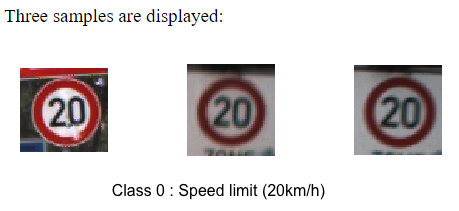
1. **Obtaining the channels of Red, Green and Blue of each Image:**

We obtain a pixmapRGB object with attributes red, green, and blue (which are the pixels for each of the three channels), as well as size, which is the width and height of the image.

Original Image:



Class 0, Original Image,



Three samples of Class 0: obtained from original Image, Red, Green and Blue respectively.

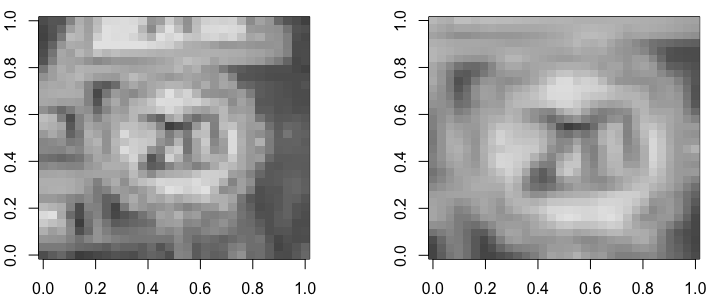
1. **Preprocessing Function:**

Preprocessing function for a raw image includes separating the ROI and resizing it to 32\*32(Obtaining the ROI and Resizing)

The signs, our regions of interest (ROI), are not centered within the images, whose sizes unfortunately vary. As a result, we need to separate the ROI from the image and standardize its size (resizing it to 32\*32) before we can analyze and classify the data. We resort to the annotations provided along with the images.

Resize function is from the EBImage package:

* EBImage is an R package which provides general purpose functionality for the reading, writing, processing and analysis of images.
* EBImage offers tools to segment cells and extract quantitative cellular descriptors.
* This facilitates the use of other tools in the R environment for signal processing, statistical modeling, machine learning and visualization with image data.

**

*Resize on Red Channel*

We process all the three channels(Red, Green and Blue).

Based on these three channels, we construct the feature space by:

Since Discarding any channel might result in loss of information. Simply stacking them up could lead to redundancy. We combine three channels into one and see to be a better solution.

In the color world, Y'UV is an encoding system that encrypts brightness information separately from color information. Y'UV represents human perception of color in terms of three components: Y' as the luminance (brightness), and U and V as the chrominance (color). Y'UV can be converted from RGB using:

* Y' = 0.299R + 0.587G + 0.114B
* U = 0.492(B - Y')
* V = 0.877(R - Y')

For our feature space, we can only take the brightness channel Y'.

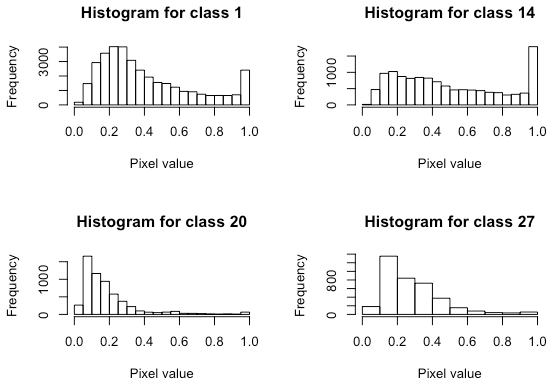
We get the whole preprocessed Image as (ROI + resize + conversion to Y') the entire labeled dataset.

Output Processed Image: Y' = 0.299R + 0.587G + 0.114B

1. **Exploratory Analysis:**

We perform Exploratory analysis on the distribution of features, that is, the pixels of the image in the dataset. We take the 16 pixels from the central 4\*4 block (222nd to 225th, 254th to 257th, 286th to 289th, and 318th to 321st) in each image from class 1 (Speed limit=30km/h), Class 14 (Stop), Class 20 (Dangerous curve to the right), and Class 27 (Pedestrians). We display their histograms:

The brightness of the central pixels is distributed differently among these four classes. For instance, the majority of the central pixels from class 20 are dark, as the sign has a thick black stroke through the center; while in class 14, the stop sign has a white stroke near the central area. Pixels taken from other positions can also be distinctly distributed among different classes.

**

***Display of Histogram***

1. **Keras Framework:**

* Keras is an open-source library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.
* Keras Sequential Model
* Built-in CNN, RNN, and autoencoder models as well as support classes and methods (metrics, optimizers, regularizers, visualization, and so on), which enable easy and fast prototyping.
* Allowing the same code to run seamlessly on CPU and GPU.
* Excellent modularity and extensibility. These allow for customized network architectures: Multiple input, multiple output, layer sharing, model sharing, memory-based network

1. **Data Preparation:**

* We prepare the input data for Keras modeling by reshaping the training, validation, and testing feature matrix.
* The input pixels are already of values ranging from 0 to 1, so we did not perform any rescaling.
* We convert training, validation, and testing target vectors (integers from 0 to 42) into a binary class matrix (one-hot encoded) as required by the Keras classification models.

1. **Define the model:**

* We used Keras sequential model
* Then Add first set of convolutional layers (hidden 2D convolutional, ReLu nonlinear layer and the pooling layer) Parameters: Filter = 32, kernel size = c(5, 5), stride = (2,2)pool size = c(2,2)
* Used pipe (%>%) operator to add layers to the Keras sequential model.
* Add second set of convolutional, ReLu nonlinear, and pooling layer. Parameters: Filter = 32, kernel size = c(5, 5), stride = (2,2)pool size = c(2,2).
* Flatten the resulting feature maps from the previous convolution layers and feed into a dense layer:
* connect to a soft max layer containing 43 output units:
* can use the summary() function to view the details of the model

1. **Optimizer:**

we use the optimizer **Stochastic gradient descent** (**SGD**) with the same learning rate = 0.005 and momentum = 0.9.

* **Results and Conclusion:**

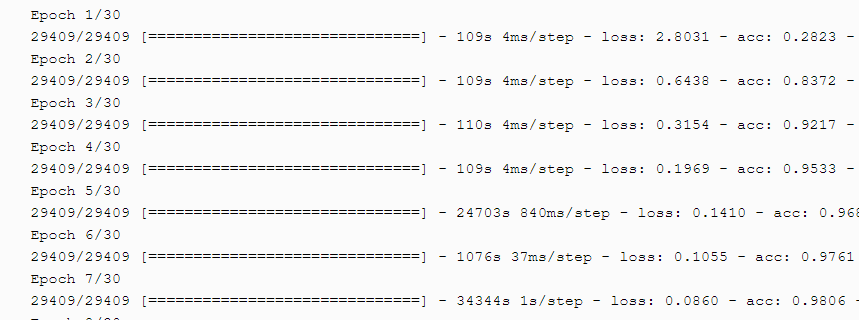
We got the accuracy of 98% on Test dataset and the accuracy of 99% on Validation dataset. After 30 epochs, the model is well trained, with 98.65% accuracy achieved on the testing set and 99% on validation dataset.

Though the accuracy obtained is this good on both the test and validation data set it has been only achieved using two layers of Convolution networks.

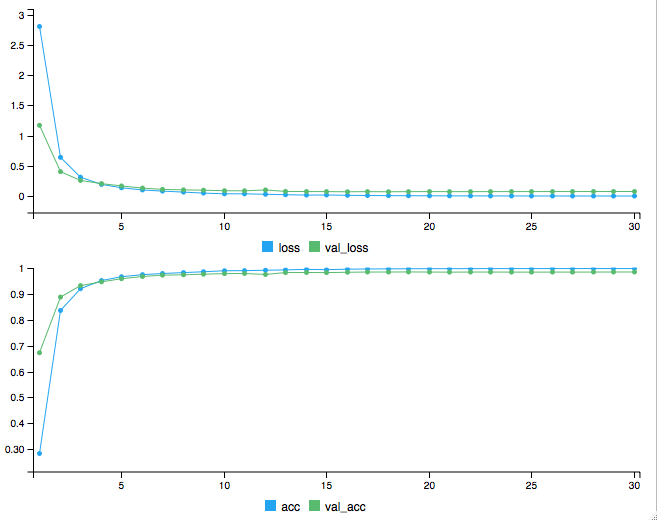
Since the variation in data is too unbalanced, the number of classes being 43 and such hue amount of data set have been used probably this accuracy have been witnessed.

We do not claim this model does give the same amount of accuracy on other datasets used when training and testing with this mode. However tuning the parameters like momentum, number of epoches and the learning rate can give the reasonable amount of accuracy.

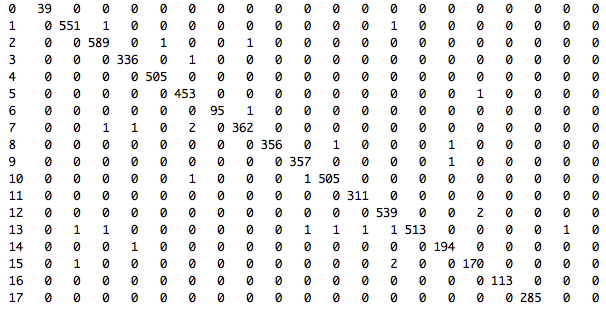
Hence, we propose future improvements to this model, Many different existing models like VGG 16, VGG 50, Alexnet, Imagenet etc can be employed. Also, many regularization techniques like Dropout, Early stopping, Local Normalization, Data Augmentation etc can be employed in the same model for further improvements on different datasets.

**

***Epochs during training***

**

***Accuracy and Loss****,*



***Confusion Matrix for Test data set***

* **Future Work:**

**Regularization:**

Overfitting is a very serious problem for all machine learning and deep learning problems. You can get to understand this is happening when your model is doing well on training data, but not able to replicate that performance on test data.

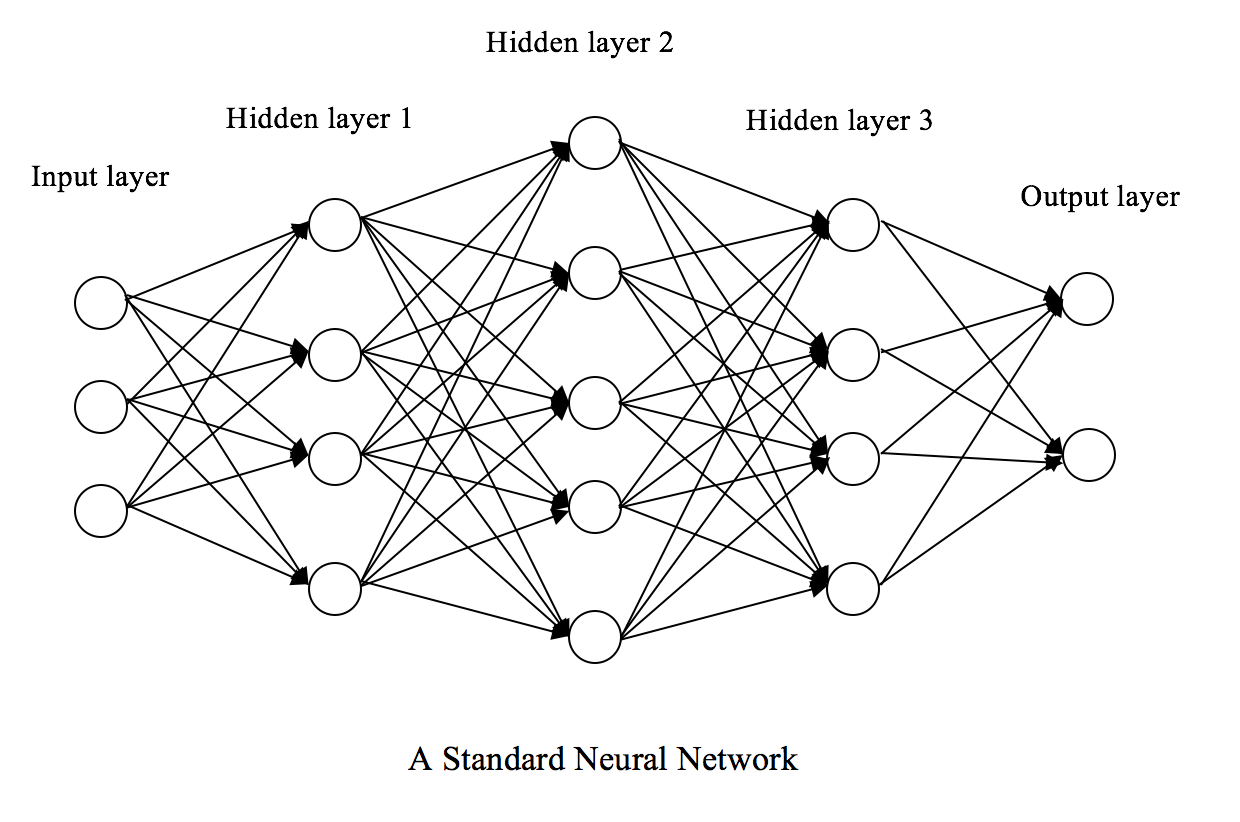
The general set of strategies against this curse of overfitting is called regularization and drop out, Local Response Normalization, Data Augumentation,Early stopping such techniques.

1. **Reducing overfitting with dropout**

We are going to employ regularization in our Keras solution, specifically dropout.

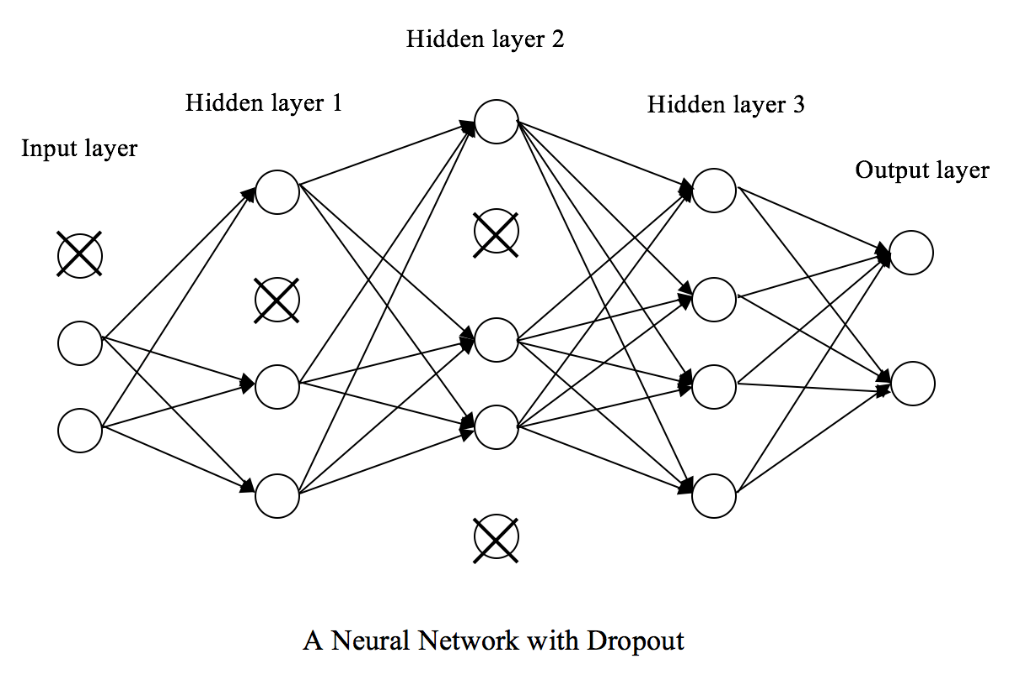
Dropout is a regularization technique in neural networks initially proposed by Geoffrey Hinton et. al. in 2012 (Improving Neural Networks by Preventing Co-adaptation of Feature Detectors in Neural and Evolutionary Computing). As the name implies, it ignores a small subset of neurons (can be hidden or visible) that are randomly selected in a neural network during training. The dropped-out neurons temporarily make no contribution to the activation of downstream neurons or the weight updates to neurons on backward pass.

In a standard neural network, neurons are co-dependent among neighboring neurons during training. And weights of neurons are tuned for a particular context within the network, which restricts the individual power of each neuron. Such reliance on context may cause the model to be too specialized to training data. When some neurons in the network are not considered, the weights of neurons become less sensitive to those of other neurons. Neurons are forces to learn useful information more independently. Co-adaptation on training data is penalized.

**

***Simple Neural Network,***

Employing dropout is simple. During the training phase and in a layer with dropout rate p, for each iteration, we randomly switch off a fraction p of neurons. In the testing phase, we use all neurons but scale their activations by a factor of q = 1 - p, in order to account for the dropped-out activations in the training phase.

******

***Neural Network with Dropout,***

1. **Early Stopping:**

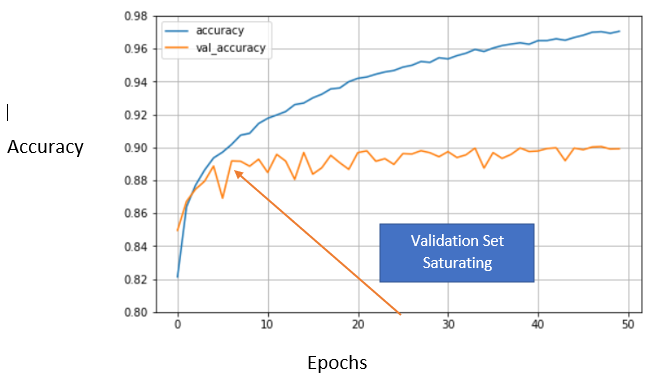
* Early Stopping monitors the performance of the model for every epoch on a held-out validation set during the training
* terminate the training conditional on the validation performance.
* stop training as soon as the validation error reaches a minimum.

The model tries to chase the loss function on the training data, by tuning the parameters. Now, we keep another set of data as the validation set and as we go on training, we keep a record of the loss function on the validation data, and when we see that there is no improvement on the validation set, we stop, rather than going all the epochs. This strategy of stopping early based on the validation set performance is called **Early Stopping.**

In the below Figure it can be seen that:

* The training set accuracy continues to increase, through all the Epochs
* The validation set accuracy, however, saturates between 8 to 10 epochs. This is where the model can be stopped training.

Early Stopping, hence does not only protect against overfitting but needs considerably less number of Epoch to train.

**

*Source Google*

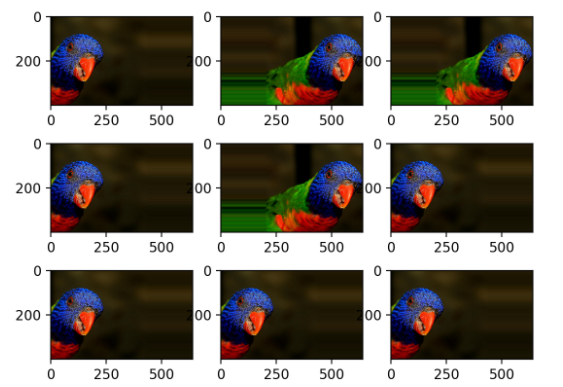
1. **Data augmentation:**

Data augmentation simply means expanding the size of the existing data that we feed to the supervised learning models in order to compensate for the cost of further data collection and labeling.

There are many ways to augment data in computer vision. The simplest one is probably flipping an image horizontally or vertically.

Achieved by:

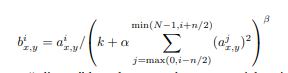
* [ImageDataGenerator class](https://keras.io/preprocessing/image/).
* Increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data.
* It acts as a regularizer and helps reduce overfitting when training a machine learning model.

**

*Source Google*

1. **Local Response Normalization:**

* Local Response Normalization.
* ImageNet Classification with Deep Convolutional Neural Networks.



*Source Google*

* **Source Code:**

**#Importing Libraries and Loading the data**

library('pixmap')  
image <- read.pnm('GTSRB/Final\_Training/Images/00000/00000\_00002.ppm',cellres=1)  
red\_matrix <- matrix(image@red, nrow = image@size[1], ncol = image@size[2])  
green\_matrix <- matrix(image@green, nrow = image@size[1], ncol = image@size[2])  
blue\_matrix <- matrix(image@blue, nrow = image@size[1], ncol = image@size[2])  
plot(image, main=sprintf("Original"))  
rotate <- function(x) t(apply(x, 2, rev))  
par(mfrow=c(1, 3))  
image(rotate(red\_matrix), col = grey.colors(255), main=sprintf("Red"))  
image(rotate(green\_matrix), col = grey.colors(255), main=sprintf("Green"))  
image(rotate(blue\_matrix), col = grey.colors(255), main=sprintf("Blue"))  
plot\_samples <- function(training\_path, class, num\_sample){  
       classes <- c("Speed limit (20km/h)", "Speed limit (30km/h)",    
                     "Speed limit (50km/h)", "Speed limit (60km/h)",  
                      "Speed limit (70km/h)", "Speed limit (80km/h)",    
                     "End of speed limit (80km/h)", "Speed limit (100km/h)",  
                    "Speed limit (120km/h)",  "No passing",  "No passing for vehicles over 3.5 metric tons",    
                     "Right-of-way at the next intersection",  "Priority road", "Yield", "Stop", "No vehicles",    
                     "Vehicles over 3.5 metric tons prohibited", "No entry", "General caution", "Dangerous curve to the left", "Dangerous curve to the right", "Double curve", " Bumpy road", "Slippery road",    
                     "Road narrows on the right", "Road work", "Traffic signals", "Pedestrians", "Children    
                   crossing", "Bicycles crossing",    
                     "Beware of ice/snow", "Wild animals crossing",    
                     "End of all speed and passing limits",    
                     "Turn right ahead", "Turn left ahead", "Ahead only",    
                     "Go straight or right", "Go straight or left",    
                     "Keep right", "Keep left", "Roundabout mandatory",    
                     "End of no passing", "End of no passing by vehicles over 3.5 metric    
                    tons")  
       if (class<10) {  
           path <- paste(training\_path, "0000", class, "/", sep="")  
         } else {  
             path <- paste(training\_path, "000", class, "/", sep="")  
         }  
       par(mfrow=c(1, num\_sample))  
       # Randomly display num\_sample samples  
         all\_files <- list.files(path = path)  
         title <- paste('Class', class, ':', classes[class+1])  
         print(paste(title, "          (", length(all\_files),    
                      " samples)", sep=""))  
         files <- sample(all\_files, num\_sample)  
         for (file in files) {  
             image <- read.pnm(paste(path, file, sep=""), cellres=1)  
             plot(image)  
           }  
         mtext(title, side = 3, line = -23, outer = TRUE)  
}  
training\_path <- "GTSRB/Final\_Training/Images/"  
plot\_samples(training\_path, 12, 3)

# for(i in 0:42) {  
#  plot\_samples(training\_path, i, 3)  
#}  
   
BiocManager::install()  
BiocManager::valid()  
BiocManager::install("EBImage")  
library("EBImage")  
roi\_resize <- function(input\_matrix, roi){  
       roi\_matrix <- input\_matrix[roi[1, 'Roi.Y1']:roi[1, 'Roi.Y2'],    
                                   roi[1, 'Roi.X1']:roi[1, 'Roi.X2']]  
       return(resize(roi\_matrix,32, 32))  
}

**#Preprocessing Function**

annotation <- read.csv(file="GTSRB/Final\_Training/Images/00000/GT-00000.csv", header=TRUE, sep=";")  
roi = annotation[3, ]  
red\_matrix\_cropped <- roi\_resize(red\_matrix, roi)  
par(mfrow=c(1, 2))  
image(rotate(red\_matrix), col = grey.colors(255) , main=sprintf("Original"))  
image(rotate(red\_matrix\_cropped), col = grey.colors(255) , main=sprintf("Preprocessed"))  
  
load\_labeled\_data <- function(training\_path, classes){  
   # Initialize the pixel features X and target y  
     X <- matrix(, nrow = 0, ncol = 32\*32)  
     y <- vector()  
     # Load images from each of the 43 classes  
        for(i in classes) {  
            print(paste('Loading images from class', i))  
            if (i<10) {  
                annotation\_path <- paste(training\_path, "0000", i, "/GT-0000",    
                                           i, ".csv", sep="")  
               path <- paste(training\_path, "0000", i, "/", sep="")  
             } else {  
                  annotation\_path <- paste(training\_path, "000", i, "/GT-000", i, ".csv", sep="")  
                  path <- paste(training\_path, "000", i, "/", sep="")  
           }  
           annotation <- read.csv(file=annotation\_path, header=TRUE,    
                                     sep=";")  
       
             for (row in 1:nrow(annotation)) {  
                 # Read each image  
                   image\_path <- paste(path, annotation[row, "Filename"], sep="")  
                   image <- read.pnm(image\_path, cellres=1)  
                   # Parse RGB color space  
                    red\_matrix <- matrix(image@red, nrow = image@size[1], ncol = image@size[2])  
                    green\_matrix <- matrix(image@green, nrow = image@size[1], ncol = image@size[2])  
                    blue\_matrix <- matrix(image@blue, nrow = image@size[1], ncol = image@size[2])  
                      # Crop ROI and resize  
                        red\_matrix\_cropped <- roi\_resize(red\_matrix, annotation[row, ])  
                        green\_matrix\_cropped <- roi\_resize(green\_matrix, annotation[row, ])  
                        blue\_matrix\_cropped <- roi\_resize(blue\_matrix, annotation[row, ])  
                      # Convert to brightness, e.g. Y' channel  
                        x <- 0.299 \* red\_matrix\_cropped + 0.587 \* green\_matrix\_cropped + 0.114 \* blue\_matrix\_cropped  
                        X <- rbind(X, matrix(x, 1, 32\*32))  
                          y <- c(y, i)  
                        }  
           }  
         return(list("x" = X, "y" = y))  
}  
  
#classes <- 0:42  
#data <- load\_labeled\_data(training\_path, classes)

**#Exploratory Data Analysis**

# Save the data object to a file  
#saveRDS(data, file = "43 classes.rds")  
# Restore the data object  
data <- readRDS(file = "43 classes.rds")  
data.x <- data$x  
data.y <- data$y  
dim(data.x)  
summary(as.factor(data.y))  
  
central\_block <- c(222:225, 254:257, 286:289, 318:321)  
par(mfrow=c(2, 2))  
for(i in c(1, 14, 20, 27)) {  
   hist(c(as.matrix(data.x[data.y==i, central\_block])),  main=sprintf("Histogram for class %d", i), xlab="Pixel brightness")  
 }

**#Splitting of Data**

set.seed(42)  
partition <- createDataPartition(data$y, p = 0.80, list = F)  
partition <- partition[sample(nrow(partition)),]  
trainingdata.x <- data.x[partition, ]  
trainingdata.y <- data.y[partition]  
test.x <- data.x[-partition, ]  
test.y <- data.y[-partition ]  
  
set.seed(42)  
partitiontraining <- createDataPartition(y = trainingdata.y, p = 0.75, list = F)  
partitiontraining <- partitiontraining[sample(nrow(partitiontraining)),]  
training.x <- trainingdata.x[partitiontraining, ]  
training.y <- trainingdata.y[partitiontraining ]  
validation.x <- trainingdata.x[-partitiontraining, ]  
validation.y <- trainingdata.y[-partitiontraining]  
  
data\_train.x <- data.x[partition,]  
data\_train.y <- data.y[partition]  
data\_validate.x <- trainingdata.x[-partitiontraining, ]  
data\_validate.y <- trainingdata.y[-partitiontraining ]  
data\_test.x <- data.x[-partition,]  
data\_test.y <- data.y[-partition]

#if (!require("keras"))  
 #  devtools::install\_github("rstudio/keras")  
library(keras)  
#install\_keras()  
#install.packages("keras")  
  
x\_train <- data\_train.x  
dim(x\_train) <- c(nrow(data\_train.x), 32, 32, 1)  
x\_test <- data\_test.x  
dim(x\_test) <- c(nrow(data\_test.x), 32, 32, 1)  
#####  
x\_val <- data\_validate.x  
dim(x\_val) <- c(nrow(data\_validate.x), 32, 32, 1)  
####  
y\_train <- to\_categorical(data\_train.y, num\_classes = 43)  
y\_test <- to\_categorical(data\_test.y, num\_classes = 43)  
###  
y\_val <- to\_categorical(data\_validate.y, num\_classes = 43)

**## Defining the Model**

#use\_session\_with\_seed(42)

model <- keras\_model\_sequential()

model %>% layer\_conv\_2d(filter = 32, kernel\_size = c(5,5),  
         input\_shape = c(32, 32, 1)) %>%  
         layer\_activation("relu") %>%  
         layer\_max\_pooling\_2d(pool\_size = c(2,2)) %>%  
     
# Second hidden convolutional layer layer  
      layer\_conv\_2d(filter = 64, kernel\_size = c(5,5)) %>%  
      layer\_activation("relu") %>%  
      layer\_max\_pooling\_2d(pool\_size = c(2,2)) %>%  
      layer\_flatten() %>%  
      layer\_dense(1000) %>%  
      layer\_activation("relu") %>%  
      layer\_dense(43) %>%  
      layer\_activation("softmax")  
  
summary(model)  
opt <- optimizer\_sgd(lr = 0.005, momentum = 0.9)  
model %>% compile(loss = "categorical\_crossentropy", optimizer = opt, metrics = "accuracy")  
model %>% fit(x\_train, y\_train, batch\_size = 100, epochs = 30,  
              validation\_data = list(x\_test, y\_test), shuffle = FALSE)  
model %>% evaluate(x\_test, y\_test,verbose = 0)  
y\_pred <- model %>% predict\_classes(x\_test)  
y\_pred  
#dim(y\_pred)  
#dim(y\_test)  
model %>% evaluate(x\_val, y\_val,verbose = 0)  
model %>% evaluate(x\_test, y\_test,verbose = 0)  
model %>% predict\_classes(x\_val)  
model %>% predict\_classes(x\_test)