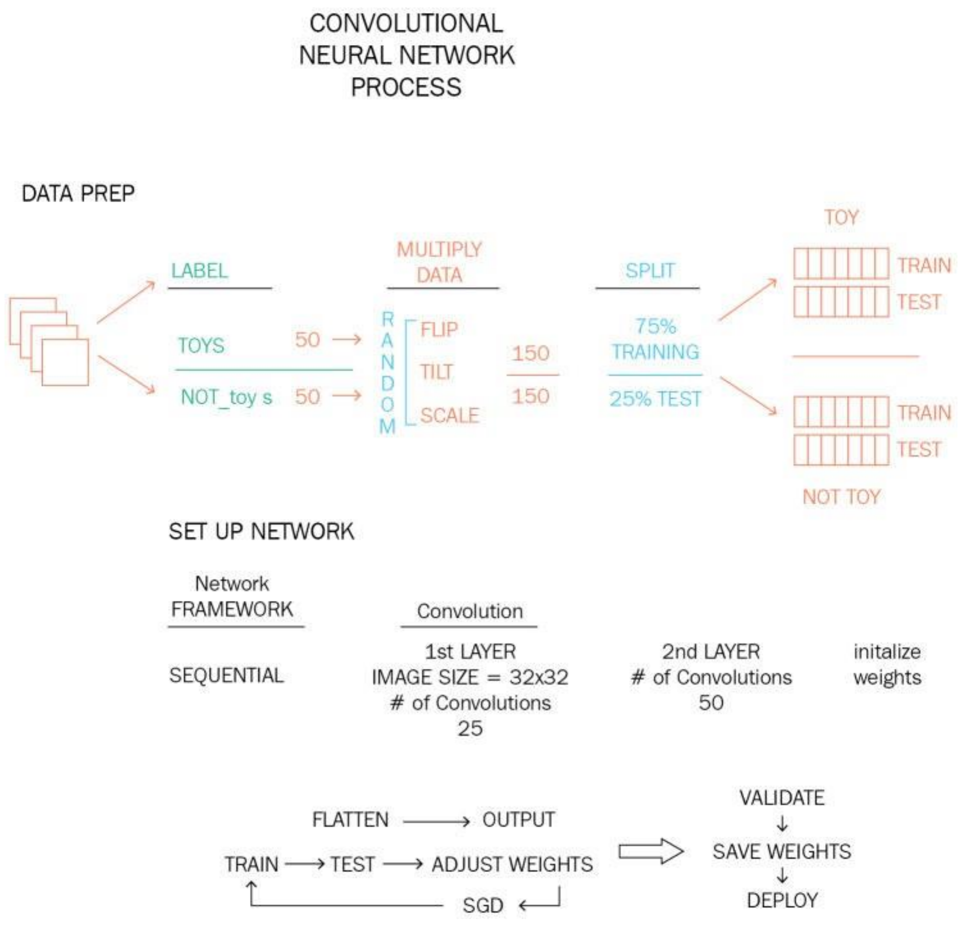
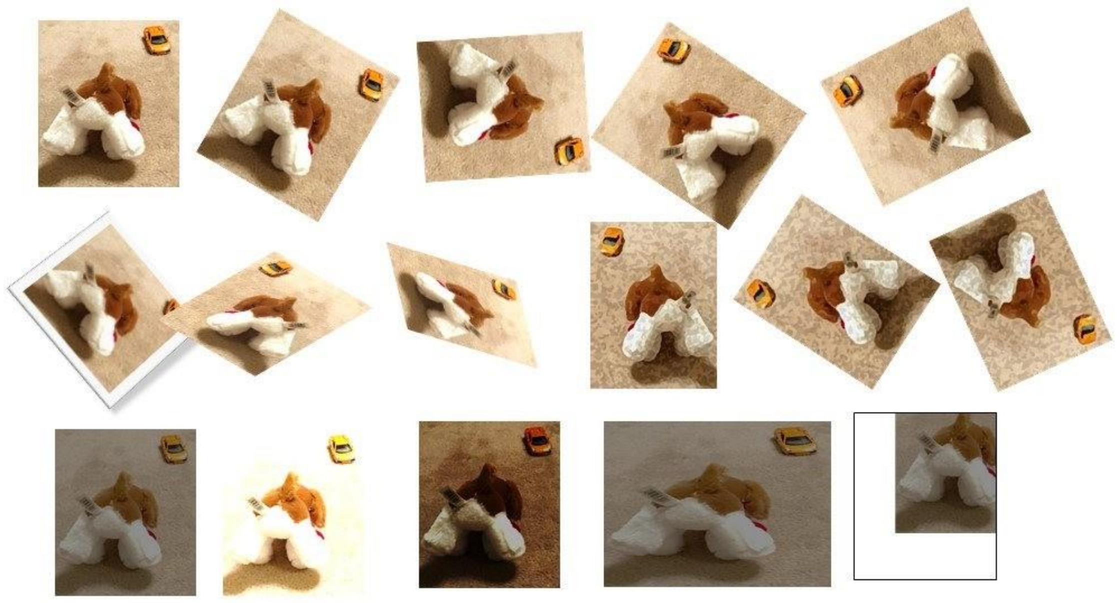
The convolution neural network process

I want to provide you with an end-to-end look at what we will be doing in the code for the rest of this chapter. Remember that we are building a **convolutional neural network** (**CNN**) that examines objects in a video frame and outputs if one or more toys are in the image, and where they are:



Here is an overview of the process:

1. Prepare a training set of images of the room with and without toys.
2. Label the area of the images that contain toys – in a separate folder.
3. Label images that don't contain toys – in a separate folder.
4. Break the training set into two parts: a set we use to train the network, and a set we use to test the network.
5. We will be building two programs: the training program that runs on our desktop computer, and trains the network, and the working program that uses the trained network to find toys.
6. Program 1: Train the network (`toyTrainNetwork.py`).
7. We take each image and multiply its training value by randomly scaling, rotating, and flipping (mirroring) the images. This increases our training set 20 fold:


Data Augmentation works by random rotations, flips, mirrors, noise, shears, and lighting

1. We build our CNN network with a convolution layer, a maxpooling layer, another convolution layer, another maxpooling layer, then a fully connected layer, and an output layer. This type of CNN is call a **LeNet**.
2. Now, we scale all our images down to reduce the amount of processing.
3. The network is initialized with random weights.
4. We present a labeled image to the network. We get an answer that is either right or wrong. If it is right, we reinforce the weights on the inputs that contributed to this answer by incrementing them (the training value). If the answer is wrong, we reduce the weights instead. In neural networks, the error between the desired result and the actual result is called **loss**.
5. Repeat for each image.
6. Now, we test the network by running the testing set of images – which are pictures of the same toys, but that were not in the training set. We see what sort of output we get over this set (how many wrong, how many right). If this answer is above 90%, we stop. Otherwise, we go back and run all the training images again.
7. Once we are happy with the results – and we should need between 100 and 500 iterations to get there – we stop and store the weights that we ended up with in the training network. This is our *trained* CNN.
8. Program 2: Find the toys (`ToyFindNetwork.pyp`).
9. Now, we *deploy* the trained network by loading it and using our video images from the live robot to look for toys. We will get a probability of an image having a toy from 0 to 100%. We scan the input video image in sections and find which sections contain toys. If we are not happy with this network, we can reload this network into the training program and train it some more.
10. Done!

Now, let's cover this in detail, step by step. We have a bit more theory to cover before we start writing the code.

Our first task is to prepare a training set. We put the camera on the robot, and drive the robot around using the teleoperation interface (or just by pushing it around by hand), snapping still photos every foot or so. We need to do at least two passes of this: one with toys in the room, and one without toys in the room. We need about 200 pictures each way, and more is better. We also need to do a set of pictures in the daytime with natural light, and at night, if your room changes lighting between day and night. This affords us several advantages: we are using the same room and the same camera to find the toys, and under the same lighting conditions.

Now, we need to label the images. I used a small program called **LabelImg**, which is available at `github/tzutalin/labelImg`. This program creates boundaries of labeled objects in the ImageNet data format. ImageNet is a popular dataset used in competitions between image classifers, and has 1,000 types of objects labeled. We have to do this ourselves.
Download the binary version of LabelImg and start it up. Use the `Open Dir` command to select the directory with your images of toys. It will present the first picture.

The process for us is fairly straightforward. We look at each picture in turn, and draw a box around any toy objects. Use the *W* shortcut key with each box. The label dialog box opens up. On the first occasion, you will have to type `toy` as the label. We label these – as you might guess – `toy`. We also draw boxes around objects that are not toys – and label them `not_toy`. This is going to take a while. We put the finished product, which is an XML file, into a file folder labeled `data`. We will need to convert the XML file into a CSV (comma separated values) file to input into our classifier training program.

In the next section, we will write our Keras/Tensorflow-based network training program, but, for the moment, let's talk about what we are going to do first. The program will read in the images one at a time, and then create additional images by randomly rotating, scaling, flipping, translating, and adding noise to the images. Now, our 200 pictures will do the work of 2,000 pictures.

We create our **convolutional neural network** (**CNN**) with an AI development package called **Keras**. The first step is to build a **sequential** network framework to incorporate the layers. Sequential networks are the most common topology used for neural networks, indicating that the network is composed of layers each connected to the layer above and the layer below, and nowhere else. We are building a five-layer network. The first network is a **convolution layer**. We'll use 20 convolutions and a 5 x 5 convolution matrix, which looks at the neighboring two pixels around each pixel to develop features. Keras will determine which convolution kernels to use. We specify the padding to match the convolution size, which adds two rows of pixels to all four sides of our image, so that we don't get *out of range* errors. Next, we have to add the **activation function** for this layer, which will be the **Rectifier Linear Unit**, or **ReLU** function, which just forwards all values greater than zero, and returns zero otherwise.

The next layer is a **maxpooling** layer, which we will set as 2 x 2 with a stride of 2 x 2. This converts blocks of four pixels into one pixel, and takes the maximum value out of the four pixels, which conserves features in the image. This layer reduces the image size to one- quarter of its original size, which increases the speed of our network, and lets the next layer work to recognize larger features in the image. If the first layer recognized tires on our toy cars, the second layer after maxpooling will look for windows, hoods, and car trunks.

The third layer is again a **convolution** using the reduced sized image as input. This convolution layer needs to have twice as many convolutions as the first layer, so it has 40 convolutions and again uses a 5 x 5 kernel size. We again use the ReLU activation function for this layer.

We add a 2 x 2 maxpooling layer after the second convolution, just like the first one.

Now, we are going to **flatten** our data into a single set of numbers in preparation for generating our output. We use the Keras flatten function and then add a **dense** or fully connected layer for layer five. We have to get from an image to a binary `yes` or `no` value, and so we flatten the data to convert the image data to a string of numbers.

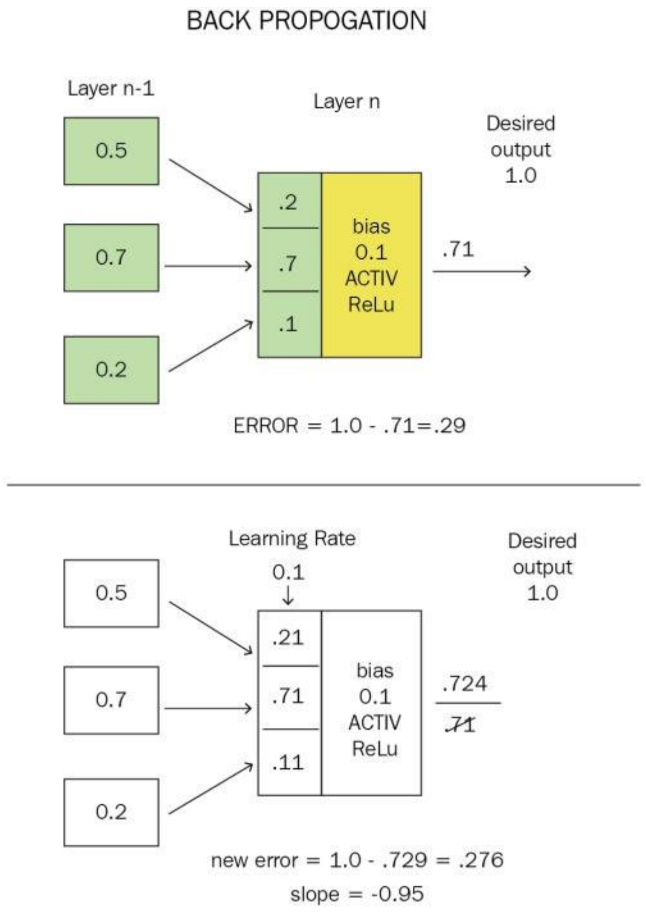
The final layer is our output layer, which just has two neurons – one for **toy** and one for **not toy**, which are our two classes of objects we will be identifying. We give the network constructor our number of classes – two – and set its activation to SoftMax. **Softmax** converts the outputs of each of our classes to be a number between 0 and 1, and then divides the outputs so that the sum of all the outputs equals `1`. Since we only have two classes, we could have used a `sigmoid` function, but the SoftMax activation function is more generic and lets us add more classes later if we want. Our output will be two numbers – the predicted probability that the image contains a toy, and the probability that the image does not contain a toy. The total will add up to `1`, so if the `toy` probability is `0.9`, then the `not_toy` probability will be `0.1`.

The final two steps in building our network are to specify our `loss`, or `error` function, and to choose a **training optimizer.** The `loss` function is how we compute the error, or loss, from the network. Each pass, the network will be presented with an image that either contains a toy, or does not. We labeled the images so that the program knows which is which – I like to think of this as the "truth" value. The network analyzes the image and produces our two numbers – the "toy" and "not toy" values. Our error is the difference between the truth value and the predicted value. If the toy value is true, or 1.0, and the prediction is toy = 0.9, then our error is 0.1. If the image did not contain a toy, and we still got 0.9 as the result, then the error would be 0.9. Since we use a lot of data to come to that one number, we have to have a way of portioning the error out to the individual neurons. We will select the **binary cross entropy** loss calculation, which has been shown to work well with this sort of problem where there are only two classes.

How we use that information in our training is the role of the **training optimizer.** We will use the **ADAM** optimizer for this example, which is an improved version of **stochastic gradient descent** (**SGD**) training. SGD is another of those simple concepts with a fancy name. **Stochastic** just means **random**. What we want to do is tweek the weights of our neurons to give a better answer than we got the first time – this is what we are training, by adjusting the weights. We want to change the weights a small amount – but in which direction? We want to change the weights in the direction that improves the answer – it makes the prediction closer to the truth.

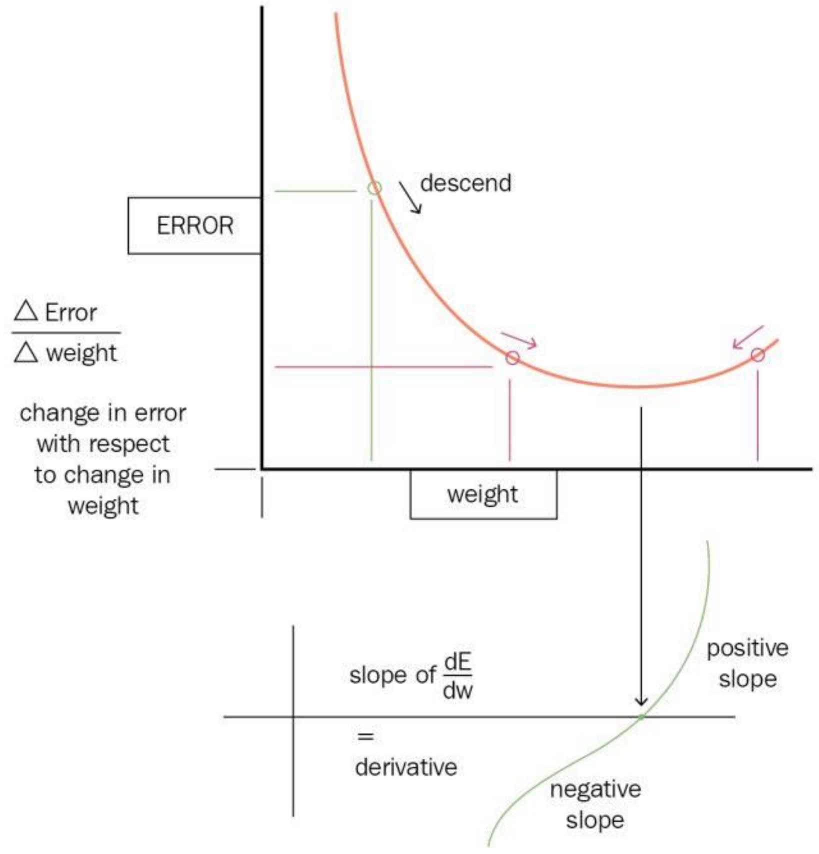
Let's do a little thought experiment. We have a neuron that we know is producing the wrong answer, and needs adjusting. We add a small amount to the weight and see how the answer changes. It gets slightly worse – the number is further away from the correct answer. So we subtract a small amount instead – and, as you might think, the answer gets better. We have reduced the amount of error slightly. If we made a graph of the error produced by the neuron, we are moving towards an error of zero, or we are descending the graph toward some minimum value. Another way of saying descending is that the slope of the line is negative – going toward zero. The amount of the slope can be called a **gradient** – just as you would refer to the slope or steepness of a hill as the gradient. We can calculate the partial derivative (in other words, the slope of the line near this point) and that tells us the slope of the line.

The way we go about adjusting the weights on the network as a whole is called **backpropogation**. That is because, as you might surmise, we have to start at the end of the network – where we know what the answer is supposed to be – and work our way toward the beginning. We have to calculate the contribution of each neuron to the answer we want, and adjust it a small amount (the learning rate) in the right direction to move toward the correct answer every time. We go back to the idea of a neuron – we have inputs, weights for each inputs, a bias, and then an activation function to produce an output. If we know what the output is, we can work backward through the neuron to adjust the weights. Let's take a simple example. Here is a neuron with three inputs, Y1, Y2, and Y3. We have three weights – W1, W2, and W3. We'll have the bias, B, and our activation function, D, which is the ReLU rectifier. Our inputs are 0.2, 0.7, and 0.02. The weights are 0.3, 0.2, and 0.5. Our bias is 0.3, and the desired output is 1.0. We calculate the sum of the inputs and weights and we get a value of 0.21. Adding our bias, we get 0.51. The ReLU function passes any value greater than zero, so the activated output of this neuron is 0.51. Our desired value is 1.0, which comes from the truth (label) data. So, our error is 0.49. If we add the training rate value to each weight, what happens? Observe the following diagram:

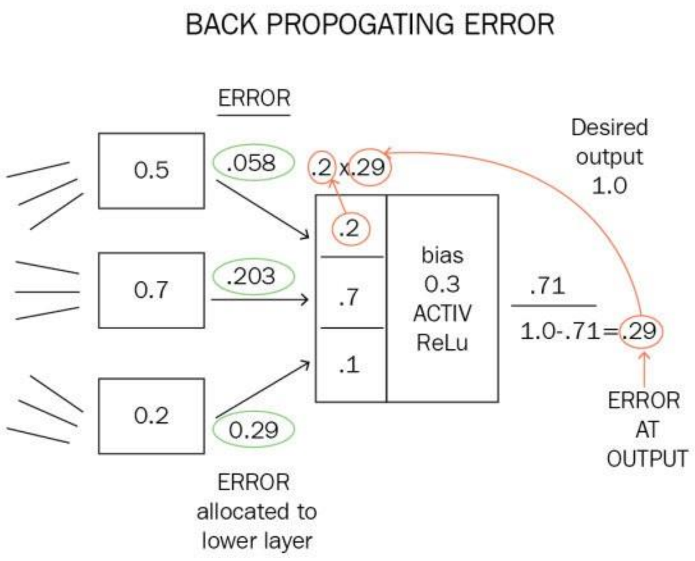


The output value now goes up to 0.5192. Our error goes down to 0.4808. We are on the right trail! The gradient of our error slope is *(.4808-.49) / 1 = -0.97*. The 1 is because we just have one training sample so far. Where does the stocahastic part come from? Our recognition network may have 50 million neurons. We can't be doing all of this math for each one. So we take a random sampling of inputs rather than all of them to determine whether our training is positive or negative.

In math terms, the slope of an equation is provided by the derivative of that equation. So, in practice, backpropagation takes the partial derivative of the error between training epochs to determine the slope of the error, and thus determine whether we are training our network correctly. As the slope gets smaller, we reduce our training rate to a smaller number to get closer and closer to the correct answer:


descend
ERROR
$\frac{\triangle \text{Error}}{\triangle \text{weight}}$
change in error with respect to change in weight
weight
slope of $\frac{dE}{dw}$
= derivative
positive slope
negative slope

Our next problem is: how do we go up layers. We can see here at the output neuron how we determine error – just the label value minus the output of the network. How do we apply this information to the previous layer? Each neuron's contribution to the error is proportional to its weight. We divide the error by the weight of each input, and that value is now the applied error of the next neuron up the chain. Then, we can recompute their weights and so on. You start to see why neural networks take so much compute power:



We backpropagate the error back up the network from the end back to the beginning. Then, we start all over again with the next cycle