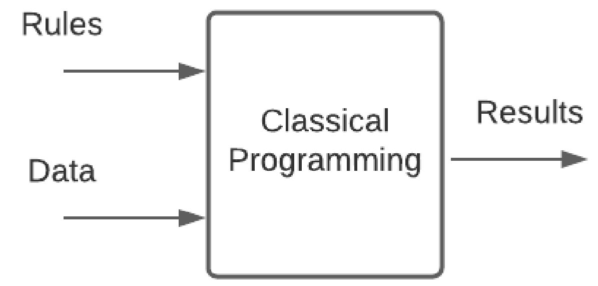
Machine Learning
Artificial Neural Networks

Deep Learning

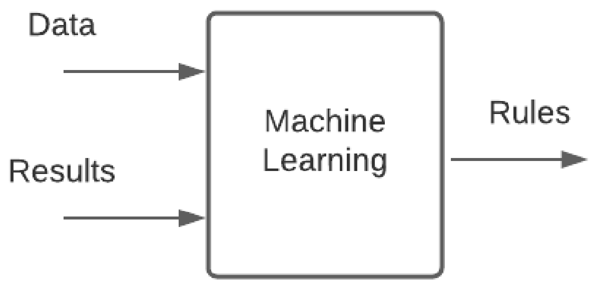
In classical problem solving, you would write the code for solving a problem and run the program with relevant data in order to produce desired results as shown in Figure 1.

Figure 1: Classical Approach to Problem Solving



In problem solving through machine learning, selected data and expected results are provided as input to the learning neural network which goes through a learning process by using the inputs. The learning produces rules for solving the problem. The trained network then becomes the problem solver for new data of the same type as the data used in training as shown in Figure 2.

Figure 2: Machine Learning Approach to Discovering Solutions to Problems



In deep learning, the neural network model learns to perform classification tasks directly from images, text, or sound. The term “deep” refers to the number of layers in the network—the more layers, the deeper the network. Traditional neural networks contain only 2 or 3 layers, while deep networks can have hundreds.

Deep learning is a subtype of machine learning. With machine learning, you manually extract the relevant features of an image. With deep learning, you feed the raw images directly into a deep neural network that learns the features automatically.

Some examples of deep learning situation are:

1. Identify faces (or more generally image categorization)
2. Read handwritten digits and texts
3. Recognize speech (no more transcribing interviews yourself)
4. Translate languages
5. Play computer games
6. Control self-driving cars (and other types of robots)

Deep learning is proving very useful in applications such as face recognition, text translation, voice recognition, and advanced driver assistance systems, including lane classification and traffic sign recognition.

A deep learning neural network consists of an input layer, several hidden layers, and an output layer. The layers are interconnected via nodes, or neurons, with each hidden layer using the output of the previous layer as its input.

Deep learning takes place by exposing the network to several different categories of labeled objects such as pictures. Once trained, we can present to the network as input a new picture. The network then uses its experience with the objects that it was exposed to in training to identify this new picture.

Let's say we have a set of images where each image contains one of four different categories of object, and we want the deep learning network to automatically recognize which object is in each image. We label the images in order to have training data for the network.

Deep learning often requires hundreds of thousands or millions of images for the best results. It's also computationally intensive and requires a high-performance graphical processing unit (GPU) in order to effectively process these large numbers within a reasonable time. Desktop computers with only CPU can be used to build deep learning networks of less complexity. Of course, such computers can be used with a pre-trained network.

An example of a powerful pre-trained network is AlexNet. It is a convolutional neural network (CNN), designed by University of Toronto scientist Alex Krizhevsky in collaboration Ilya

Sutskever under the guidance Professor Geoffrey Hinton, a cognitive psychologist and computer scientist, most noted for his work on artificial neural networks.
AlexNet is 8 layers deep. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. It can be accessed through the Internet. Another example of a pre-trained network is GoogLeNet. It is a convolutional neural network that is 22 layers deep.
A convolution is simply application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.
The innovation in convolutional neural networks is the ability to automatically run a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images.
The following is a simple demonstration of access and operation AlexNet using MATLAB, an environment without requiring GPU on user desktop computer.
The code used is simple and almost self-explanatory (text following a % is a comment for benefit of the user and need not be type in running code statement.

<u>Step 1: Clear the camera and identify it as the webcam to be accessed by AlexNet</u>

```
clear camera; % Clear the camera from any pre-existing images
camera = webcam; % Connect to the camera
```

<u>Step 2: Access AlexNet operating environment</u>

```
nnet = alexnet; % Access AlexNet
```

<u>Step 3: Take s picture of the object in front of webcam</u>

```
picture = camera.snapshot ; % Take a picture
```

<u>Step 4: Define the image size, and AlexNet assign a label to what found</u>

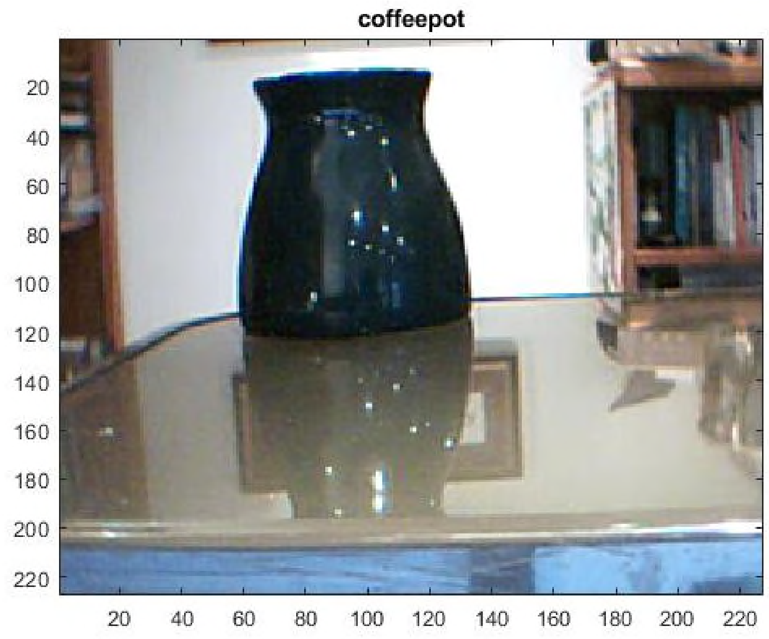
```
% Resize the picture, 227*227=51529 picture elements, each with brightness between 0 and 255
% Brightness value of 0 is black color, brightness of 255 is white color (maximum color value)
picture = imresize(picture,[227,227]);

label = classify(nnet, picture);  % Classify the picture
```

Step5: Display the picture title as determined by AlexNet

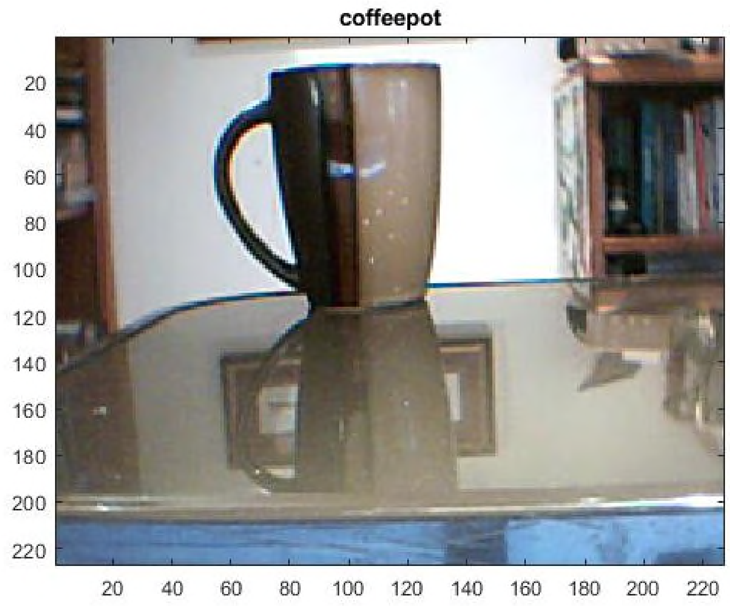
```
image(picture); % Show the picture
title(char(label)); % Show the label
```

Get ready for a snapshot from you webcam before the snapshot command. Just for fun, I executed this step while sitting in a high back leather chair with some clutter behind it. After executing step 5 commands, AlexNet identified the picture as barber shop. Close but not quite there.

Next time for the same steps, I allowed the webcam to take picture of a fancy mug which also be used as flower pot. Here is how AlextNet classified it.



I would say that AlexNet is quite a good recognizer of objects. I thought, I should stop fooling around, I allowed AlexNet to take a picture of a bonafide coffee mug. Here is how it classified it.



Looks like a coffee pot. AlexNet got rightfully confused because I had placed the coffee mug on a reflecting surface making it look like a coffee pot.

Next time, I will place the object on a non-reflecting surface just be nice to AlexNet.

The point to note is that a picture consists of picture elements. Each element is called a pixel. These are small little dots that make up the image. We need to make sure that the elements that really represent what we are trying to recognize are identified and not the noise, the elements that are distracting us from the object of our attention. Filtering functions are applied to reduce the effect of noise.

The following example shows how to create and train a simple convolutional neural network (convnet/CNN) for classification using deep learning. The term convolute means to fold, roll or twist together. In convnet each layer of neurons affects the adjacent layer of neurons. Convnet uses a learning algorithm particularly suited to image processing. Rather than working with global features of an image, it breaks the image into local features. After learning a pattern in one location, it can recognize it anywhere. This makes processing of data more efficient. Convnets can take an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. Convnets require fewer training samples to learn representations that have generalization power. The amount of pre-processing required in a convnet is much lower as compared to other classification algorithms. Convnet can learn spatial hierarchies in the patterns it recognizes. The first

convolution layer learns small local patterns such as edges, the second layer recognizes larger patterns made of the features of the first layer, and so on. It allows convnets to learn increasingly complex and abstract visual concepts.
The inspiration for convnets appears to have come from the structure of the human visual cortex and connectivity patterns of neurons in the human brain.
A convolution is the application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image. Convolutional neural networks are essential tools for deep learning, and are especially suited for image recognition.

Building Blocks of Deep Learning Networks

1. Drawing a batch of training samples X and corresponding targets (expected outputs) y
   You develop an appropriate representation of input data. For example, if we have 10,000 images of digits from 0 to 9, called labels, we may code each image as a (28,28) array of pixels, each pixel having an eight-bit integer values between 0 and 255 indicating it brightness. The basic data structure (data container) used in machine learning is called a tensor. A tensor is a multidimensional array. The dimensions may vary from 0 for a scalar to an appropriate number for network layers, referred to as the tensor rank. The dimensions are called axes, on the lines of conventional thinking of axes in arrays. A matrix of rows and columns is a 2D tensor. Packing 2D tensors in an array leads to a 3D tensor and so on. Each tensor has three attributes rank, shape and data type. For example, 10,000 samples of input digits as arrays of 28-by-28 integer data type, we denote the tensor shape as (10000, 28, 28). The product of 28-by-28 is 784. The input is converted into a representation as (10000, 784) and assigning values to each point in the image. The data value of each point is transformed into fractions of 32-bit floating point data type.   This is achieved by dividing the integer values of 0 to 255 in the 28-by-28 array with the maximum of 255, resulting in gloating point values between 0 and 1 for network processing. This form of representation happens at the network input layer.
2. In forward pass (propagation) through the network, you feed the network tensor of inputs X, and targets y to obtain y_pred, the values predicted by the network for specified targets. The

output of each neuron can be expressed as, output=relu(dot (W, input) + b) where W, input, and b are the tensors for weights, input data (X, y), and bias respectively. Generally, the initial values of weights are chosen randomly and the bias is set at zero. These parameters get adjusted automatically for subsequent forward passes on the basis of results from the back propagation through the network. Here relu, rectified linear unit, is an activation function that is applied to sum of tensor product of inputs and associated weights plus the bias vector. We can define relu function operation as relu(x)=max(x,0) i.e. the actual calculated output is x if x is greater than zero, otherwise the output is simply zero. This computational operation occurs at every node, and in some cases at selected nodes. A node is simply a computational module. Therefore, operations at all nodes of a layer in forward pass can be performed in parallel. In large networks, parallel processing becomes a necessity.

3. The network applies a loss function to determine the mismatch between y and y_pred. The goal is to minimize it. The loss function is one of the important components of neural networks. Loss is nothing but a prediction error of neural net. The method to calculate the loss is called a loss function. In simple words, the loss is used to calculate the gradients, the rates of change and the amounts of change during a forward pass. Instead of just using the gradient of the current step to guide the forward pass, momentum also accumulates the gradient of the past steps to determine the direction to go. The gradients are used to update the parameters of the neural net. This is how a neural net is trained.
4. Performing a backward pass (propagation), calculating the amount and the rate of change (gradient) for all hidden network layers. Backpropagation algorithm is probably the most fundamental building block in a neural network. The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the network model's parameters (weights and biases).
5. Making changes in node parameters can be performed simultaneously at all nodes of all layers, making massive parallel processing a necessity. Parameters are modified for reducing the loss, making W=W-stepsize*gradient. This process is known as the stochastic gradient descent with momentum (SGDM) optimization, and the algorithm used is called an optimizer. Gradient, in plain terms means slope or slant of a surface. So gradient descent literally means descending a slope to reach the lowest point on that surface**.** The network has

gone through a learning experience that can be applied to successive steps. The amount by which the weights are updated during training is referred to as the step size or the "learning rate." Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0. Step size of the algorithm plays a critical role. It determines the subset of the local optima that the algorithm can converge to, and it specifies the magnitude of the oscillations if the algorithm converges to an orbit. There is relationship between the step size of the algorithm and what is obtained as a solution.

6. The output of the last hidden layer becomes the input of the output layer. If we have ten classes to identify as in the case of ten numerical digits then the output layer contains 10 units. The output is shaped through 10-way softmax layer. It returns an array of ten probability scores (with a sum of 1). Each score will be the probability that the current digit image presented belongs to one-of-ten-digit classes, 0, 1, 2…9. Softmax is an activation function, like relu and sigmoid. The term softmax is used because this activation function represents a smooth version of the winner-takes-all activation model in which the unit with the largest input has output value of 1 while all other units have output value of 0.

Example: Simple Deep Learning Classification Example in MATLAB

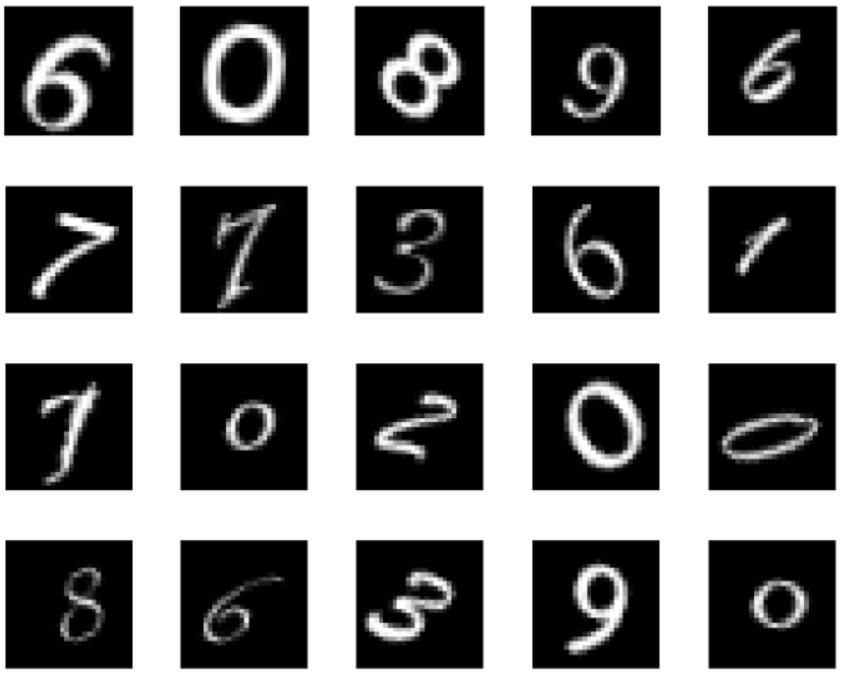
Broadly speaking, the operational steps are:

1. Load and explore image data.
2. Define the network architecture.
3. Specify training options.
4. Train the network.
5. Predict the labels of new data and calculate the classification accuracy.
6. Load and Explore Image Data

MATLAB programming environment is being used in this example, rather than Python, because it allows to build deep learning networks without the need to have a high-power graphical processing unit (GPU) in your computer. These GPUs are expensive. Examples are Titan X and GTX 1080 Ti. As an alternative, one could use Google Cloud instances or Amazon Web Services (AWS). However, they involve usage-based associated costs.

The goal here is to understand the structure and operations of deep learning networks and not necessarily to focus on the coding required for it.

Step 1: Load the following image as input data



Each digit is a picture of 28x28 pixels. So, in total each image of a digit has 28*28=784 pixels. Each pixel takes a value between 0 and 255 (based on RGB code). Each image consists of an array of 784 numbers as inputs to the network.

Coding is shown in boxes and comments or explanations appear before/after each box.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...

    'nndatasets','DigitDataset');

imds = imageDatastore(digitDatasetPath, ...

    'IncludeSubfolders', true,'LabelSource','foldernames');

figure;
perm = randperm(10000,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end
```

Data is labeled automatically and stored. This image datastore enables you to store large amounts of data, including data that does not fit in memory, and read batches of images efficiently during training of a convolutional neural network.

Step 2: Calculate the number of images in each category

```
labelCount = countEachLabel(imds)
```

The output is label count.

```
labelCount = 10×2 table

    Label    Count
    _____    _____

      0      1000
      1      1000
      2      1000
      3      1000
      4      1000
      5      1000
      6      1000
      7      1000
      8      1000
      9      1000
```

Label count is a table that contains the label names and the number of images for each label. The datastore contains 1000 images for each of the digits 0-9, for a total of 10000 images. You can specify the number of classes in the last fully connected layer of your network as the OutputSize argument.

Step 3: Specify the image size

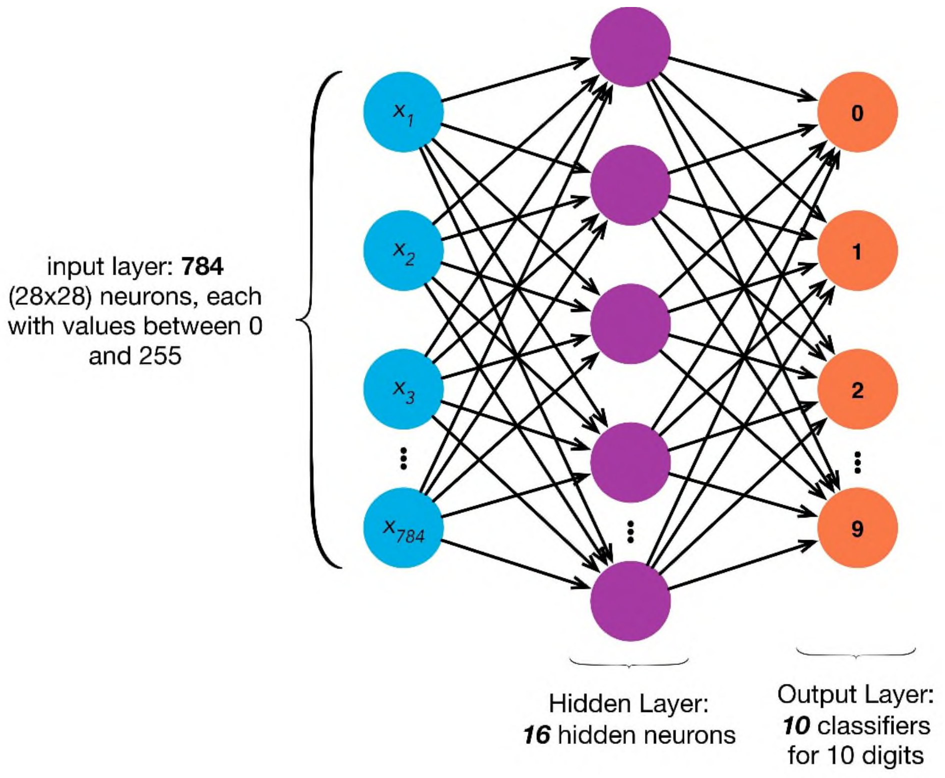
```
img = readimage(imds,1);
size(img)

img = readimage(imds,1);
>> size(img)
```

The output is:

```
ans =
   28    28
ans = 1×2
   28    28
```

Each image is 28-by-28-by-1 array of pixels. Therefore, there are 28*28=784 pixels with grayscale brightness values from 0 to 255 for images labeled from 0 to 9. The computer can't really "see" a digit like we humans do, but if we dissect the image into an array of 784 numbers with values like [0, 0, 180, 16, 230, …, 4, 77, 0, 0, 0], then we can feed this array into our neural network. The computer can't understand an image by "seeing" it, but it can understand and analyze the pixel values that represent an image.

Figure 3: Visualization of inputs, hidden, and output layers


input layer: **784** (28x28) neurons, each with values between 0 and 255
x_1
x_2
x_3
x_{784}
0
1
2
9
Hidden Layer: ***16*** hidden neurons
Output Layer: ***10*** classifiers for 10 digits

Let the set of inputs X be designated as x_1, x_2, x_3……x_m. These are known as m features. A feature is just a variable that has influence to a specific outcome. These features are multiplied by their corresponding set W of weights w_1, w_2, w_3, ……..,, w_m and then summed into what is known as dot product, described as:

$$W \cdot X = w_1x_1 + w_2x_2 + \ldots + w_mx_m = \sum_{i=1}^{m} w_ix_i$$

Added to it is a bias value b_i. There are usually 2 or more hidden layers in a deep learning network. At each hidden layer of n number of neurons, the inputs are multiplied by the corresponding weights and summed. There are n such summations called dot products. Subsequently, this summed value plus the corresponding bias, b, is operated upon by an activation function to create the desired effect of activating or not activating a neuron in following hidden layer of neurons.

Step 4: Specify the training data sets

```
label. splitEachLabel splits the datastore digitData into two new
datastores, trainDigitData and valDigitData.
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');
```

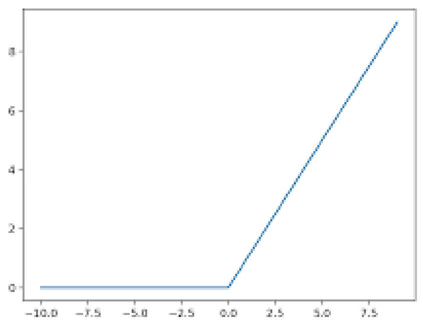
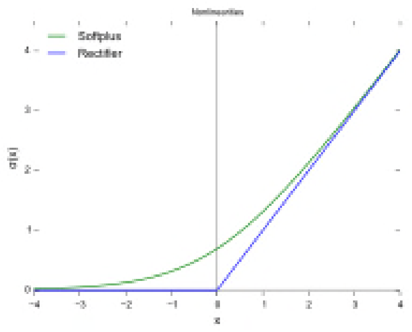
Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each

Step 5: Define the convnet architecture

```
layers = [
    imageInputLayer([28 28 1])
      convolution2dLayer(3,8,'Padding','same')
      batchNormalizationLayer
      reluLayer
      maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
      reluLayer

    maxPooling2dLayer(2,'Stride',2)

convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
      reluLayer

    fullyConnectedLayer(10)
      softmaxLayer
      classificationLayer];
```

Image Input Layer is where you specify the image size, which, in this case, is 28-by-28-by-1. These numbers correspond to the height, width, and the channel size. The digit data consists of grayscale images, so the channel size (color channel) is 1. For a color image, the channel size is 3, corresponding to the RGB values. You do not need to shuffle the data because trainNetwork, by default, shuffles the data at the beginning of training. trainNetwork can also automatically shuffle the data at the beginning of every epoch during training.
Convolutional Layer’s first argument is filterSize, which is the height and width of the filters the training function uses while scanning along the images. In this example, the number 3 indicates that the filter size is 3-by-3. You can specify different sizes for the height and width of the filter. The second argument is the number of filters, numFilters, which is the number of neurons that connect to the same region of the input. This parameter determines the number of feature maps. Use the 'Padding' name-value pair to add padding to the input feature map. For a convolutional layer with a default stride of 1, 'same' padding ensures that the spatial output size is the same as the input size. You can also define the stride and learning rates for this layer using name-value pair arguments of convolution2dLayer.
Batch Normalization Layer Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization.
use batchNormalizationLayer to create a batch normalization layer.
The batch normalization layer is followed by a nonlinear activation function. The most common activation function is the rectified linear unit (ReLU). Use relu to create a ReLU layer.

Figure 4: Neuron Input to Output Activation Functions

ReLU is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.

Max Pooling Layer Convolutional layers (with activation functions) are sometimes followed by a down-sampling operation that reduces the spatial size of the feature map and removes redundant spatial information. Down-sampling makes it possible to increase the number of filters in deeper convolutional layers without increasing the required amount of computation per layer. One way of down-sampling is using a max pooling, which you create using maxPooling2dLayer. The max pooling layer returns the maximum values of rectangular regions of inputs, specified by the first argument, poolSize. In this example, the size of the rectangular region is [2,2]. The 'Stride' name-value pair argument specifies the step size that the training function takes as it scans along the input.

Fully Connected Layer The convolutional and down-sampling layers are followed by one or more fully connected layers. As its name suggests, a fully connected layer is a layer in which the neurons connect to all the neurons in the preceding layer. This layer combines all the features learned by the previous layers across the image to identify the larger patterns. The last fully connected layer combines the features to classify the images. Therefore,
the OutputSize parameter in the last fully connected layer is equal to the number of classes in the target data. In this example, the output size is 10, corresponding to the 10 classes.

Use fullyConnectedLayer to create a fully connected layer.

Softmax Layer The softmax activation function normalizes the output of the fully connected layer. The output of the softmax layer consists of positive numbers that sum to one, which can then be used as classification probabilities by the classification layer. Create a softmax layer using the softmaxLayer function after the last fully connected layer.

Classification Layer The final layer is the classification layer. This layer uses the probabilities returned by the softmax activation function for each input to assign the input to one of the mutually exclusive classes and compute the loss. To create a classification layer,
use classificationLayer.

Step 6: Specify Training Options

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

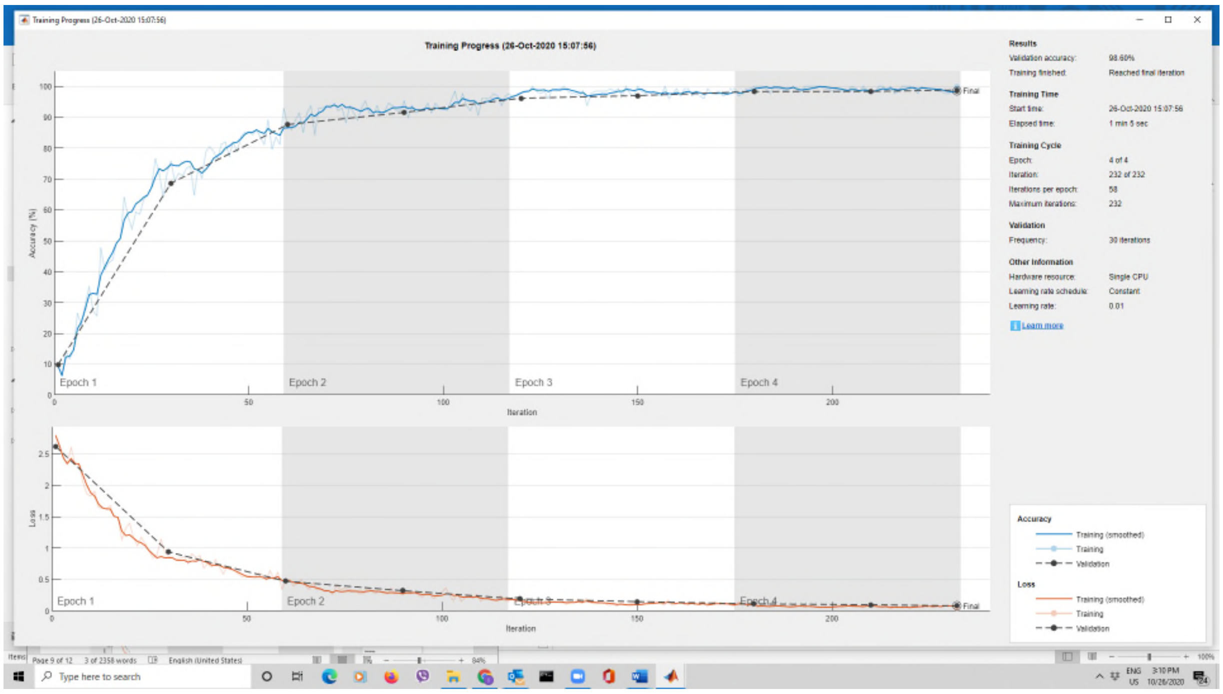
After defining the network structure, specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial step size (learning rate) of 0.01. Set the maximum number of epochs to 4. An epoch is a full training cycle on the entire training data set. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights. Turn on the training progress plot, and turn off the command window output.

Step 7: Train network using training data

```
net = trainNetwork(imdsTrain,layers,options);
```

Train the network using the architecture defined by layers, the training data, and the training options.

The resulting output is:



Results

Validation accuracy:	98.60%
Training finished:	Reached final iteration

Training Time

Start time:	26-Oct-2020 15:07:56
Elapsed time:	1 min 5 sec

Training Cycle

Epoch:	4 of 4
Iteration:	232 of 232
Iterations per epoch:	58
Maximum iterations:	232

Validation

Frequency:	30 iterations

Other Information

Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.01

Learn more

The picture is a video showing the accuracy of output as the network goes though the learning epochs until the loss function value is reduced to near zero. An epoch is a full pass through the entire data set.

By default, trainNetwork uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the 'ExecutionEnvironment' name-value pair argument of trainingOptions.

The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy.

When you train a network for deep learning, it is often useful to monitor the training progress. By plotting various metrics during training, you can learn how the training is progressing. For example, you can determine if and how quickly the network accuracy is improving, and whether the network is starting to overfit the training data.

When you specify ‘training-progress’ as the ‘Plots’ value in trainingOptions and start network training, trainNetwork creates a figure and displays training metrics at every iteration. Each iteration is an estimation of the gradient and an update of the network parameters. If you specify validation data in trainingOptions, then the figure shows validation metrics each time trainNetwork validates the network. The figure plots the following:

- Training accuracy — Classification accuracy on each individual mini-batch.
- Smoothed training accuracy — Smoothed training accuracy, obtained by applying a smoothing algorithm to the training accuracy. It is less noisy than the unsmoothed accuracy, making it easier to spot trends.
- Validation accuracy — Classification accuracy on the entire validation set (specified using trainingOptions).
- Training loss, smoothed training loss, and validation loss **—** The loss on each mini-batch, its smoothed version, and the loss on the validation set, respectively. If the final layer of your network is a classificationLayer, then the loss function is the cross-entropy loss. For regression networks, the figure plots the root mean square error (RMSE) instead of the accuracy.

During the training, you can stop the training and return the current state of the network by clicking the stop button in the top-right corner. For example, you might want to stop training when the accuracy of the network reaches a plateau and it is clear that the accuracy is no longer improving. After you click the stop button, it can take a while for the training to complete. Once training is complete, trainNetwork returns the trained network.

When training finishes, view the results showing the final validation accuracy and the reason that training finished. The final validation metrics are labeled final in the plots. If your network contains batch normalization layers, then the final validation metrics are often different from the validation metrics evaluated during training. This is because batch normalization layers in the final network perform different operations than during training.

The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

Step 8: Classify validation images and compute accuracy

Predict the labels of the validation data using the trained network, and calculate the final validation accuracy. Accuracy is the fraction of labels that the network predicts correctly. In this case, more than 99% of the predicted labels match the true labels of the validation set.

```
YPred = classify(net,imdsValidation);
YValidation = imdsValidation.Labels;
accuracy = sum(YPred == YValidation)/numel(YValidation)
```

The output is:

```
accuracy = 0.9988
```

Sources:
https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f
https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
Chollet, Francois, Deep Learning with Python, Manning, NY, 2018.
https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks