ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ

Νευρωνικά Δίκτυα

ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ #2

Διδάσκων Καθηγητής: Δαλιάνης Σωτήριος

ΕΚΠ. ΕΤΟΣ 2022-2023

1 Σκοπός της Άσκησης

Η άσκηση αυτή αποσκοπεί στο να εξοικειώσει τους φοιτητές στην δημιουργία, εκπαίδευση, και αξιολόγηση απλών και σύνθετων Νευρωνικών Δικτύων σε εφαρμογές ταξινόμησης, πρόβλεψης χρονοσειρών και αναγνώρισης προτύπων, με εφαρμογές στο Matlab.

2 Υπόβαθρο – Προετοιμασία

- Βασικές γνώσεις προγραμματισμού σε περιβάλλον MATLAB ή OCTAVE.
- Κατανόηση βασικών αρχών στατιστικής ανάλυσης.
- Εγκατάσταση του περιβάλλοντος MATLAB.
- Εγκατάσταση του Statistics and Machine Learning Toolbox του περιβάλλοντος MATLAB.
- Εγκατάσταση των εφαρμογών Machine Learning and Deep Learning του περιβάλλοντος MATLAB.
- Εγκατάσταση της εφαρμογής Deep Network Designer και σύνδεση του matlab με προκατασκευασμένα νευρωνικά δύκτια όπως το `googlenet`, alexnet ή άλλο δίκτυο.

3 Εκτέλεση της Άσκησης

Επιλέξτε 2 Data sets από τις βιβλιοθήκες του MATLAB με διαφορετικά χαρακτηριστικά και εισάγετέ τα στο matlab. Προσαρμόστε τα data ανάλογα με το νευρονικό δίκτυο που χρησιμοποιείτε

Στο Α μέρος της άσκησης:

Χρησιμοποιώντας τις εφαρμογές GUI του Machine Learning Toolbox, - Data clustering (nctool) - Time-series analysis (ntstool), να δημιουργήσετε ανάλογα νευρωνικά δίκτυα και να τα εκπαιδεύσετε με κατάλληλα Data sets. Αξιολογήστε τα δίκτυα παρουσιάζοντας τα γραφήματα. Επιλέξτε διαφορετικές παραμέτρους υπολογισμού και να παρουσιάστε τα αποτελέσματα. Να αναλύσετε σύντομα τα αποτελέσματα σχετικά με τα δεδομένα και τις μεθόδους που χρησιμοποιήσατε.

Στο Β μέρος της άσκησης:

Β.1. Χρησιμοποιώντας την εφαρμογή Deep Network Designer να εισάγετε ένα προ-εκπαιδευμένο δίκτυο, αφού συνδέσετε το matlab με προκατασκευασμένα νευρωνικά δίκτια όπως το `googlenet`, alexnet ή άλλο δίκτυο. Να μελετήστε το δίκτυο και να περιγράψετε βασικά χαρακτηριστικά του.

Β.2 Με το δίκτυο που έχετε φορτώσει δοκιμάστε να αναγνωρίσετε χαρακτηριστικό από δικές σας εικόνες με κατάλληλο μέγεθος ή από βιβλιοθήκες.

Αποθηκεύσατε χαρακτηριστικά αποτελέσματα και γραφήματα σχετικά με την εκπαίδευση και αξιοπιστία του νευρονικού δικτύου. Σε ασκήσεις με μεγάλο αριθμό δεδομένων και μεταβλητών συνιστάτε να αποθηκεύτε το workspace σε αρχείο .mat.

Η παρουσίαση της άσκησης γίνεται σε ένα ενιαίο αρχείο world doc ή pdf που περιλαμβάνει σύντομη περιγραφή του προβλήματος, τον κώδικα που χρησιμοποιήσατε, τα αποτελέσματα και τα γραφήματα. Τα αρχεία ανεβαίνουν στο e-class μέχρι την καταληκτική ημερομηνία παράδοσης της άσκησης.

Μέρος Α. Διαχωρισμός κλάσεων με νευρωνικά δίκτυα S.O.M. στο MATLAB

1. Cluster Data with a Self-Organizing Map

Clustering data is another excellent application for neural networks. This process involves grouping data by similarity. For example, you might perform:

- Market segmentation by grouping people according to their buying patterns
- Data mining by partitioning data into related subsets
- Bioinformatic analysis by grouping genes with related expression patterns

Suppose that you want to cluster flower types according to petal length, petal width, sepal length, and sepal width. You have 150 example cases for which you have these four measurements.

As with function fitting and pattern recognition, there are two ways to solve this problem:

- Use the `nctool` GUI, as described in Using the Neural Network Clustering App.
- Use a command-line solution, as described in Using Command-Line Functions.

Defining a Problem

To define a clustering problem, simply arrange Q input vectors to be clustered as columns in an input matrix (see "Data Structures" for a detailed description of data formatting for static and time series data). For instance, you might want to cluster this set of 10 two-element vectors:

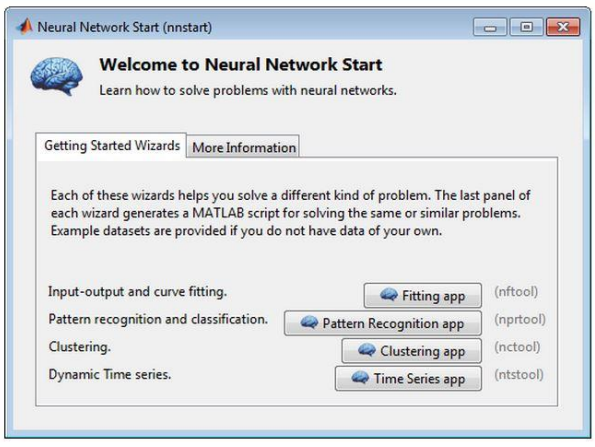
```
inputs = [7 0 6 2 6 5 6 1 0 1; 6 2 5 0 7 5 5 1 2 2]
```

The next section shows how to train a network using the `nctool` GUI.

Using the Neural Network Clustering App

1. If needed, open the Neural Network Start GUI with this command:

```
nnstart
```



2. Click **Clustering app** to open the Neural Network Clustering App. (You can also use the command `nctool`.)

3. Click **Next**. The Select Data window appears.

4. Click **Load Example Data Set**. The Clustering Data Set Chooser window appears.

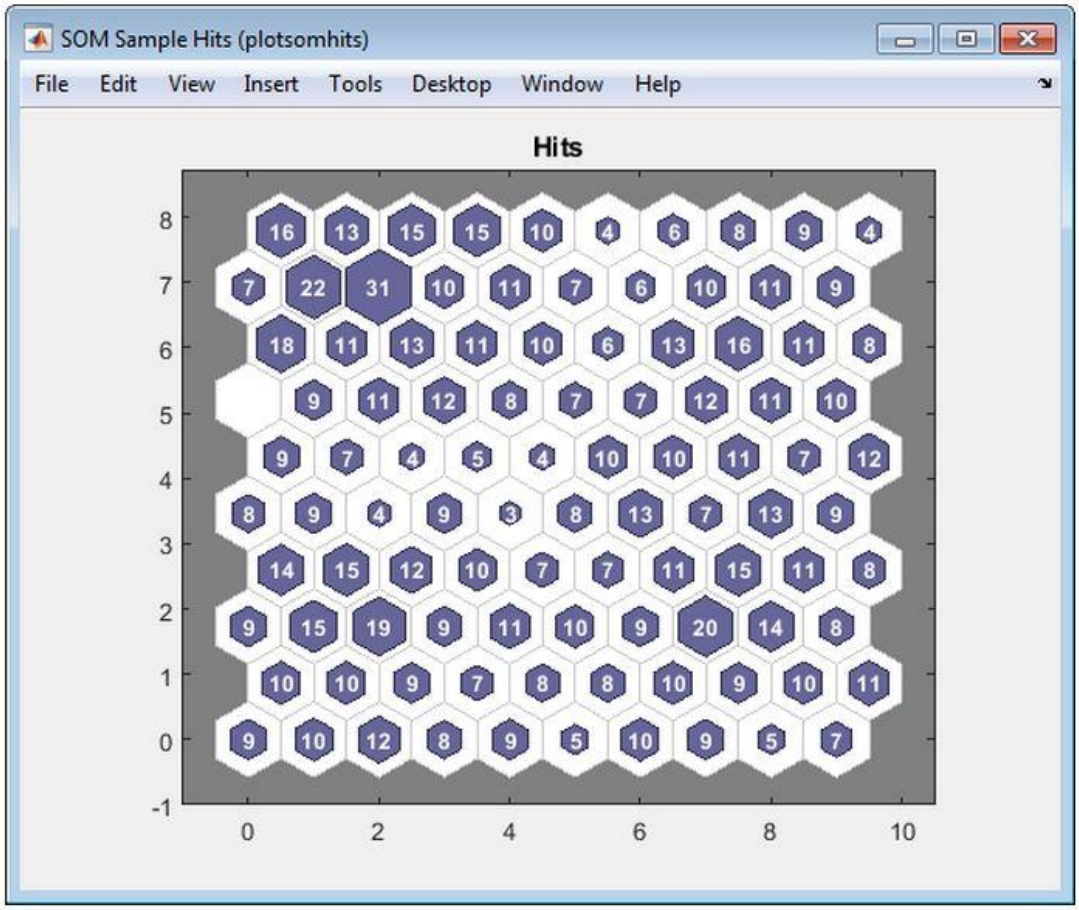
5. In this window, select **Simple Clusters**, and click **Import**. You return to the Select Data window.
6. Click **Next** to continue to the Network Size window, shown in the following figure.

   For clustering problems, the self-organizing feature map (SOM) is the most commonly used network, because after the network has been trained, there are many visualization tools that can be used to analyze the resulting clusters. This network has one layer, with neurons organized in a grid. (For more information on the SOM, see "Self-Organizing Feature Maps".) When creating the network, you specify the numbers of rows and columns in the grid. Here, the number of rows and columns is set to `10`. The total number of neurons is 100. You can change this number in another run if you want.

7. Click **Next**. The Train Network window appears.

8. Click **Train**.

   The training runs for the maximum number of epochs, which is 200.
9. For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. Investigate some of the visualization tools for the SOM. Under the **Plots** pane, click **SOM Sample Hits**.



The default topology of the SOM is hexagonal. This figure shows the neuron locations in the topology, and indicates how many of the training data are associated with each of the neurons (cluster centers). The topology is a 10-by-10 grid, so there are 100 neurons. The maximum number of hits associated with any neuron is 31. Thus, there are 31 input vectors in that cluster.

10. You can also visualize the SOM by displaying weight planes (also referred to as *component planes*). Click **SOM Weight Planes** in the Neural Network Clustering App.

    This figure shows a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Darker colors represent larger weights.) If the connection patterns of two inputs were very similar, you can assume that the inputs are highly correlated. In this case, input 1 has connections that are very different than those of input 2.

11. In the Neural Network Clustering App, click **Next** to evaluate the network.

    At this point you can test the network against new data.

If you are dissatisfied with the network's performance on the original or new data, you can increase the number of neurons, or perhaps get a larger training data set.

12. When you are satisfied with the network performance, click **Next**.
13. Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs or deploy the network with MATLAB Compiler tools and other MATLAB and Simulink code generation tools.

14. Use the buttons on this screen to save your results.

- You can click **Simple Script** or **Advanced Script** to create MATLAB® code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In Using Command-Line Functions, you will investigate the generated scripts in more detail.
- You can also save the network as `net` in the workspace. You can perform additional tests on it or put it to work on new inputs.

15. When you have generated scripts and saved your results, click **Finish**.

Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created in step 14 of the previous section.

```
% Solve a Clustering Problem with a Self-Organizing Map
% Script generated by NCTOOL
%
% This script assumes these variables are defined:
%
%   simpleclusterInputs - input data.

inputs = simpleclusterInputs;

% Create a Self-Organizing Map
dimension1 = 10;
dimension2 = 10;
net = selforgmap([dimension1 dimension2]);

% Train the Network
[net,tr] = train(net,inputs);

% Test the Network
```

```
outputs = net(inputs);

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
% figure, plotsomtop(net)
% figure, plotsomnc(net)
% figure, plotsomnd(net)
% figure, plotsomplanes(net)
% figure, plotsomhits(net,inputs)
% figure, plotsompos(net,inputs)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the training process. In this case, let's follow each of the steps in the script.

1. The script assumes that the input vectors are already loaded into the workspace. To show the command-line operations, you can use a different data set than you used for the GUI operation. Use the flower data set as an example. The iris data set consists of 150 four-element input vectors.

   ```
     load iris_dataset
   inputs = irisInputs;
   ```

2. Create a network. For this example, you use a self-organizing map (SOM). This network has one layer, with the neurons organized in a grid. (For more information, see "Self-Organizing Feature Maps".) When creating the network with `selforgmap`, you specify the number of rows and columns in the grid:

   ```
   dimension1 = 10;
   dimension2 = 10;
   net = selforgmap([dimension1 dimension2]);
   ```

3. Train the network. The SOM network uses the default batch SOM algorithm for training.

   ```
   [net,tr] = train(net,inputs);
   ```

4. During training, the training window opens and displays the training progress. To interrupt training at any point, click **Stop Training**.

5. Test the network. After the network has been trained, you can use it to compute the network outputs.

   ```
   outputs = net(inputs);
   ```

6. View the network diagram.

   ```
   view(net)
   ```

7. For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs

space in the two dimensions of the network topology. The default SOM topology is hexagonal; to view it, enter the following commands.

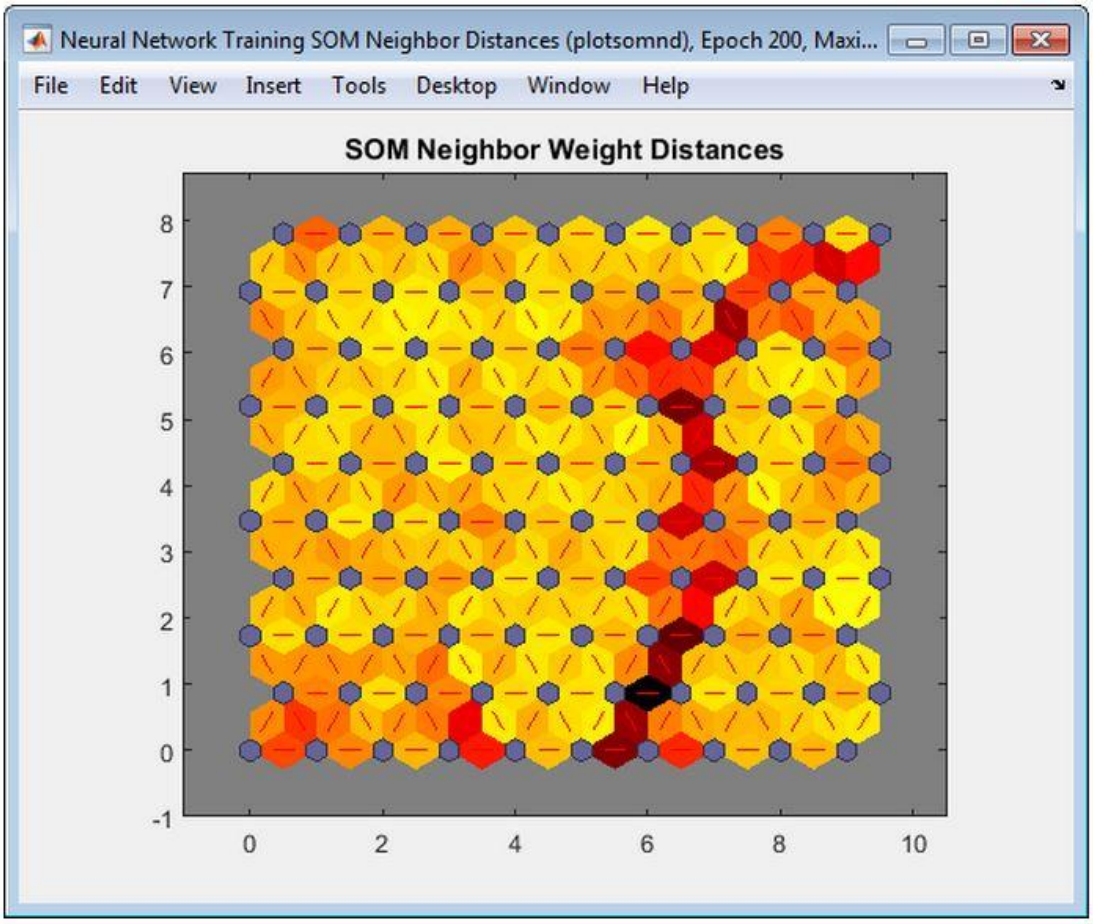
```
figure, plotsomtop(net)
```

In this figure, each of the hexagons represents a neuron. The grid is 10-by-10, so there are a total of 100 neurons in this network. There are four elements in each input vector, so the input space is four-dimensional. The weight vectors (cluster centers) fall within this space.

Because this SOM has a two-dimensional topology, you can visualize in two dimensions the relationships among the four-dimensional cluster centers. One visualization tool for the SOM is the *weight distance matrix* (also called the *U-matrix*).

8. To view the U-matrix, click **SOM Neighbor Distances** in the training window.

   In this figure, the blue hexagons represent the neurons. The red lines connect neighboring neurons. The colors in the regions containing the red lines indicate the distances between neurons. The darker colors represent larger distances, and the lighter colors represent smaller distances. A band of dark segments crosses from the lower-center region to the upper-right region. The SOM network appears to have clustered the flowers into two distinct groups.



To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the SOM weight position plot) and watch it animate.
- Plot from the command line with functions such as `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`. (For more information on using these functions, see their reference pages.)

Also, see the advanced script for more options, when training from the command line. https://www.mathworks.com/help/deeplearning/

Μέρος Β. Αναγνώριση εικόνων και χαρακτηριστικών με μεθόδους Deep Learning

2. Deep Learning in MATLAB

What Is Deep Learning?

Deep learning is a branch of machine learning that teaches computers to do what comes naturally to humans: learn from experience. Machine learning algorithms use computational methods to "learn" information directly from data without relying on a predetermined equation as a model. Deep learning is especially suited for image recognition, which is important for solving problems such as facial recognition, motion detection, and many advanced driver assistance technologies such as autonomous driving, lane detection, pedestrian detection, and autonomous parking.

Deep Learning Toolbox™ provides simple MATLAB® commands for creating and interconnecting the layers of a deep neural network. Examples and pretrained networks make it easy to use MATLAB for deep learning, even without knowledge of advanced computer vision algorithms or neural networks.

For a free hands-on introduction to practical deep learning methods, see Deep Learning Onramp.

Deep Learning Toolbox

Deep Learning Toolbox™ provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps. You can use convolutional neural networks (ConvNets, CNNs) and long short-term memory (LSTM) networks to perform classification and regression on image, time-series, and text data. You can build network architectures such as generative adversarial networks (GANs) and Siamese networks using automatic differentiation, custom training loops, and shared weights. With the Deep Network Designer app, you can design, analyze, and train networks graphically. The Experiment Manager app helps you manage multiple deep learning experiments, keep track of training parameters, analyze results, and compare code from different experiments. You can visualize layer activations and graphically monitor training progress.

You can import networks and layer graphs from TensorFlow™ 2, TensorFlow-Keras, and PyTorch®, the ONNX™ (Open Neural Network Exchange) model format, and Caffe. You can also export Deep Learning Toolbox networks and layer graphs to TensorFlow 2 and the ONNX model format. The toolbox supports transfer learning with DarkNet-53, ResNet-50, NASNet, SqueezeNet and many other pretrained models.

You can speed up training on a single- or multiple-GPU workstation (with Parallel Computing Toolbox™), or scale up to clusters and clouds, including NVIDIA® GPU Cloud and Amazon EC2® GPU instances (with MATLAB® Parallel Server™).

Tutorials

App Workflows

- **Get Started with Deep Network Designer**
  This example shows how to use Deep Network Designer to adapt a pretrained GoogLeNet network to classify a new collection of images.
- **Get Started with Transfer Learning**
  This example shows how to use transfer learning to retrain SqueezeNet, a pretrained convolutional neural network, to classify a new set of images.
- **Create Simple Image Classification Network Using Deep Network Designer**
  This example shows how to create and train a simple convolutional neural network for deep learning classification using Deep Network Designer.
- **Create Simple Sequence Classification Network Using Deep Network Designer**
  This example shows how to create a simple long short-term memory (LSTM) classification network using Deep Network Designer.

Command-Line Workflows

- **Try Deep Learning in 10 Lines of MATLAB Code**
  Learn how to use deep learning to identify objects on a live webcam with the SqueezeNet pretrained network.
- **Classify Image Using Pretrained Network**
  This example shows how to classify an image using the pretrained deep convolutional neural network GoogLeNet.
- **Create Simple Image Classification Network**
  This example shows how to create and train a simple convolutional neural network for deep learning classification.

https://www.mathworks.com/help/deeplearning/getting-started-with-deep-learning-toolbox.html?s_tid=CRUX_lftnav

https://www.mathworks.com/videos/what-is-deep-learning-toolbox--1535667599631.html?s_tid=vid_pers_recs

https://www.mathworks.com/videos/deep-learning-for-signals-and-sound-1544467789023.html?s_tid=vid_pers_recs

What Do You Want to Do?	Learn More
Perform transfer learning to fine-tune a network with your data	Start Deep Learning Faster Using Transfer Learning **Tip** Fine-tuning a pretrained network to learn a new task is typically much faster and easier than training a new network.
Classify images with pretrained networks	Pretrained Deep Neural Networks
Create a new deep neural network for classification or regression	Create Simple Deep Learning Network for Classification Train Convolutional Neural Network for Regression
Resize, rotate, or preprocess images for training or prediction	Preprocess Images for Deep Learning
Label your image data automatically based on folder names, or interactively using an app	Train Network for Image Classification Image Labeler
Create deep learning networks for sequence and time series data.	Sequence Classification Using Deep Learning Time Series Forecasting Using Deep Learning
Classify each pixel of an image (for example, road, car, pedestrian)	Semantic Segmentation Basics (Computer Vision Toolbox)
Detect and recognize objects in images	Deep Learning, Semantic Segmentation, and Detection (Computer Vision Toolbox)
Classify text data	Classify Text Data Using Deep Learning
Classify audio data for speech recognition	Speech Command Recognition Using Deep Learning
Visualize what features networks have learned	Deep Dream Images Using AlexNet Visualize Activations of a Convolutional Neural Network
Train on CPU, GPU, multiple GPUs, in parallel on your desktop or on clusters in the cloud, and work with data sets too large to fit in memory	Deep Learning with Big Data on GPUs and in Parallel

To learn more about deep learning application areas, including automated driving, see Deep Learning Applications.

To choose whether to use a pretrained network or create a new deep network, consider the scenarios in this table.

	Use a Pretrained Network for Transfer Learning	Create a New Deep Network
Training Data	Hundreds to thousands of labeled images (small)	Thousands to millions of labeled images
Computation	Moderate computation (GPU optional)	Compute intensive (requires GPU for speed)
Training Time	Seconds to minutes	Days to weeks for real problems
Model Accuracy	Good, depends on the pretrained model	High, but can overfit to small data sets

For more information, see Choose Network Architecture.

Deep learning uses neural networks to learn useful representations of features directly from data. Neural networks combine multiple nonlinear processing layers, using simple elements operating in parallel and inspired by biological nervous systems. Deep learning models can achieve state-of-the-art accuracy in object classification, sometimes exceeding human-level performance.

You train models using a large set of labeled data and neural network architectures that contain many layers, usually including some convolutional layers. Training these models is computationally intensive and you can usually accelerate training by using a high performance GPU. This diagram shows how convolutional neural networks combine layers that automatically learn features from many images to classify new images.

Many deep learning applications use image files, and sometimes millions of image files. To access many image files for deep learning efficiently, MATLAB provides the `imageDatastore` function. Use this function to:

- Automatically read batches of images for faster processing in machine learning and computer vision applications
- Import data from image collections that are too large to fit in memory
- Label your image data automatically based on folder names

Try Deep Learning in 10 Lines of MATLAB Code

This example shows how to use deep learning to identify objects on a live webcam using only 10 lines of MATLAB code. Try the example to see how simple it is to get started with deep learning in MATLAB.

1. Run these commands to get the downloads if needed, connect to the webcam, and get a pretrained neural network.
2.
```
camera = webcam; % Connect to the camera
net = alexnet;   % Load the neural network
```
If you need to install the `webcam` and `alexnet` add-ons, a message from each function appears with a link to help you download the free add-ons using Add-On Explorer. Alternatively, see Deep Learning Toolbox Model *for AlexNet Network* and MATLAB Support Package for USB Webcams.

After you install Deep Learning Toolbox Model *for AlexNet Network*, you can use it to classify images. AlexNet is a pretrained convolutional neural network (CNN) that has been trained on more than a million images and can classify images into 1000 object categories (for example, keyboard, mouse, coffee mug, pencil, and many animals).

3. Run the following code to show and classify live images. Point the webcam at an object and the neural network reports what class of object it thinks the webcam is showing. It will keep classifying images until you press **Ctrl**+**C**. The code resizes the image for the network using `imresize`.

```
while true
    im = snapshot(camera);       % Take a picture
    image(im);                   % Show the picture
    im = imresize(im,[227 227]); % Resize the picture for alexnet
    label = classify(net,im);    % Classify the picture
    title(char(label));          % Show the class label
    drawnow
end
```

In this example, the network correctly classifies a coffee mug. Experiment with objects in your surroundings to see how accurate the network is.

To watch a video of this example,

https://www.mathworks.com/videos/deep-learning-in-11-lines-of-matlab-code-1481229977318.html

see Deep Learning in 11 Lines of MATLAB Code.

To learn how to extend this example and show the probability scores of classes, see Classify Webcam Images Using Deep Learning.

For next steps in deep learning, you can use the pretrained network for other tasks. Solve new classification problems on your image data with transfer learning or feature extraction. For examples, see Start Deep Learning Faster Using Transfer Learning and Train Classifiers Using Features Extracted from Pretrained Networks. To try other pretrained networks, see Pretrained Deep Neural Networks.

3. Pretrained Deep Neural Networks

You can take a pretrained image classification network that has already learned to extract powerful and informative features from natural images and use it as a starting point to learn a new task. The majority of the pretrained networks are trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [2]. These networks have been trained on more than a million images and can classify images into 1000 object categories, such as keyboard, coffee mug, pencil, and many animals. Using a pretrained network with transfer learning is typically much faster and easier than training a network from scratch.

You can use previously trained networks for the following tasks:

Purpose	Description
Classification	Apply pretrained networks directly to classification problems. To classify a new image, use `classify`. For an example showing how to use a pretrained network for classification, see Classify Image Using GoogLeNet.
Feature Extraction	Use a pretrained network as a feature extractor by using the layer activations as features. You can use these activations as features to train another machine learning model, such as a support vector machine (SVM). For more information, see Feature Extraction. For an example, see Extract Image Features Using Pretrained Network.
Transfer Learning	Take layers from a network trained on a large data set and fine-tune on a new data set. For more information, see Transfer Learning. For a simple example, see Get Started with Transfer Learning. To try more pretrained networks, seeTrain Deep Learning Network to Classify New Images.

Load Pretrained Networks

Use functions such as `googlenet` to get links to download pretrained networks from the Add-On Explorer. The following table lists the available pretrained networks trained on ImageNet and some of their properties. The network depth is defined as the largest number of sequential convolutional or fully connected layers on a path from the input layer to the output layer. The inputs to all networks are RGB images.

Network	Depth	Size	Parameters (Millions)	Image Input Size
`alexnet`	8	227 MB	61.0	227-by-227
`vgg16`	16	515 MB	138	224-by-224
`vgg19`	19	535 MB	144	224-by-224
`squeezenet`	18	4.6 MB	1.24	227-by-227
`googlenet`	22	27 MB	7.0	224-by-224
`inceptionv3`	48	89 MB	23.9	299-by-299
`densenet201`	201	77 MB	20.0	224-by-224
`mobilenetv2`	54	13 MB	3.5	224-by-224
`resnet18`	18	44 MB	11.7	224-by-224
`resnet50`	50	96 MB	25.6	224-by-224
`resnet101`	101	167 MB	44.6	224-by-224
`xception`	71	85 MB	22.9	299-by-299
`inceptionresnetv2`	164	209 MB	55.9	299-by-299

GoogLeNet Trained on Places365

The standard GoogLeNet network is trained on the ImageNet data set but you can also load a network trained on the Places365 data set [3] [4]. The network trained on

Places365 classifies images into 365 different place categories, such as field, park, runway, and lobby. To load a pretrained GoogLeNet network trained on the Places365 data set, use `googlenet('Weights','places365')`. When performing transfer learning to perform a new task, the most common approach is to use networks pretrained on ImageNet. If the new task is similar to classifying scenes, then using the network trained on Places365 could give higher accuracies.

Compare Pretrained Networks

Pretrained networks have different characteristics that matter when choosing a network to apply to your problem. The most important characteristics are network accuracy, speed, and size. Choosing a network is generally a tradeoff between these characteristics.

Tip

To get started with transfer learning, try choosing one of the faster networks, such as SqueezeNet or GoogLeNet. You can then iterate quickly and try out different settings such as data preprocessing steps and training options. Once you have a feeling of which settings work well, try a more accurate network such as Inception-v3 or a ResNet and see if that improves your results.

Use the plot below to compare the ImageNet validation accuracy with the time required to make a prediction using the network. A good network has a high accuracy and is fast. The plot displays the classification accuracy versus the prediction time when using a modern GPU (an NVIDIA® TITAN Xp) and a mini-batch size of 64. The prediction time is measured relative to the fastest network. The area of each marker is proportional to the size of the network on disk.

A network is *Pareto efficient* if there is no other network that is better on all the metrics being compared, in this case accuracy and prediction time. The set of all Pareto efficient networks is called the *Pareto frontier*. The Pareto frontier contains all the networks that are not worse than another network on both metrics. The plot connects the networks that are on the Pareto frontier in the plane of accuracy and prediction time. All networks except AlexNet, VGG-16, VGG-19, Xception, and DenseNet-201 are on the Pareto frontier.

Note

The plot below only shows an indication of the relative speeds of the different networks. The exact prediction and training iteration times depend on the hardware and mini-batch size that you use.

The classification accuracy on the ImageNet validation set is the most common way to measure the accuracy of networks trained on ImageNet. Networks that are accurate on ImageNet are also often accurate when you apply them to other natural image data sets using transfer learning or feature extraction. This generalization is possible because the networks have learned to extract powerful and informative features from natural images that generalize to other similar data sets. However, high accuracy on ImageNet does not always transfer directly to other tasks, so it is a good idea to try multiple networks.

If you want to perform prediction using constrained hardware or distribute networks over the Internet, then also consider the size of the network on disk and in memory.

Network Accuracy

There are multiple ways to calculate the classification accuracy on the ImageNet validation set and different sources use different methods. Sometimes an ensemble of multiple models is used and sometimes each image is evaluated multiple times using multiple crops. Sometimes the top-5 accuracy instead of the standard (top-1) accuracy is quoted. Because of these differences, it is often not possible to directly compare the accuracies from different sources. The accuracies of pretrained networks in Deep Learning Toolbox™ are standard (top-1) accuracies using a single model and single central image crop.

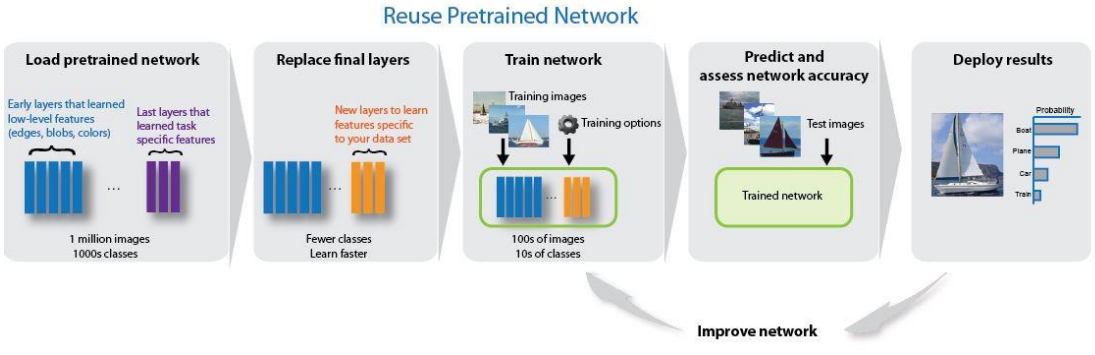
Feature Extraction

Feature extraction is an easy and fast way to use the power of deep learning without investing time and effort into training a full network. Because it only requires a single pass over the training images, it is especially useful if you do not have a GPU. You extract learned image features using a pretrained network, and then use those features to train a classifier, such as a support vector machine using `fitcsvm`.

Try feature extraction when your new data set is very small. Since you only train a simple classifier on the extracted features, training is fast. It is also unlikely that fine-tuning deeper layers of the network improves the accuracy since there is little data to learn from.

- If your data is very similar to the original data, then the more specific features extracted deeper in the network are likely to be useful for the new task.
- If your data is very different from the original data, then the features extracted deeper in the network might be less useful for your task. Try training the final classifier on more general features extracted from an earlier network layer. If the new data set is large, then you can also try training a network from scratch.

ResNets are often good feature extractors. For an example showing how to use a pretrained network for feature extraction, see Extract Image Features Using Pretrained Network.



Transfer Learning

You can fine-tune deeper layers in the network by training the network on your new data set with the pretrained network as a starting point. Fine-tuning a network with transfer learning is often faster and easier than constructing and training a new network. The network has already learned a rich set of image features, but when you

fine-tune the network it can learn features specific to your new data set. If you have a very large data set, then transfer learning might not be faster than training from scratch.

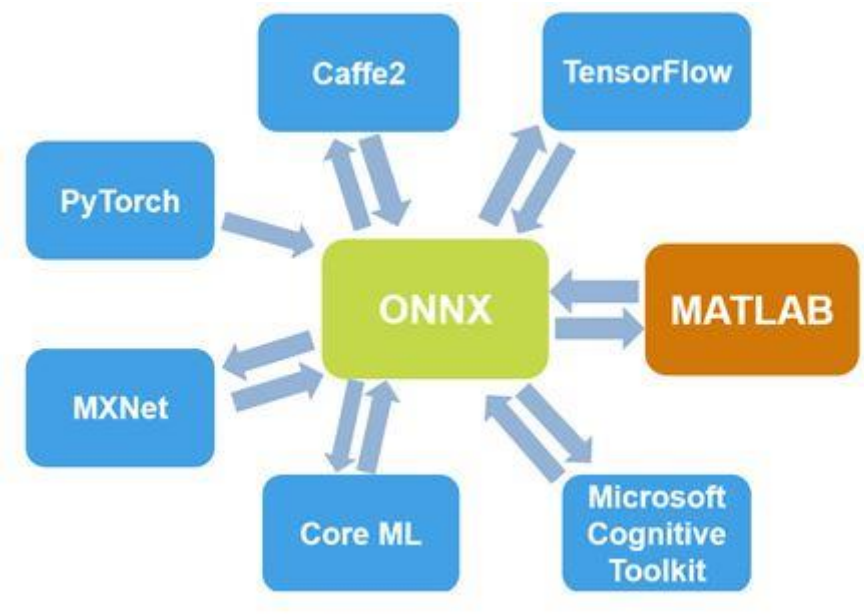
Tip

Fine-tuning a network often gives the highest accuracy. For very small data sets (fewer than about 20 images per class), try feature extraction instead.

Fine-tuning a network is slower and requires more effort than simple feature extraction, but since the network can learn to extract a different set of features, the final network is often more accurate. Fine-tuning usually works better than feature extraction as long as the new data set is not very small, because then the network has data to learn new features from. For examples showing how to perform transfer learning, see Transfer Learning with Deep Network Designer and Train Deep Learning Network to Classify New Images.

Import and Export Networks

You can import networks and network architectures from TensorFlow®-Keras, Caffe, and the ONNX™ (Open Neural Network Exchange) model format. You can also export trained networks to the ONNX model format.



Import from Keras

Import pretrained networks from TensorFlow-Keras by using `importKerasNetwork`. You can import the network and weights either from the same HDF5 (.h5) file or separate HDF5 and JSON (.json) files. For more information, see `importKerasNetwork`.

Import network architectures from TensorFlow-Keras by using `importKerasLayers`. You can import the network architecture, either with or without weights. You can import the network architecture and weights either from the same HDF5 (.h5) file or separate HDF5 and JSON (.json) files. For more information, see `importKerasLayers`.

Import from Caffe

Import pretrained networks from Caffe by using the `importCaffeNetwork` function. There are many pretrained networks available in Caffe Model Zoo[5]. Download the desired `.prototxt` and `.caffemodel` files and use `importCaffeNetwork` to import the pretrained network into MATLAB®. For more information, see `importCaffeNetwork`.

You can import network architectures of Caffe networks. Download the desired `.prototxt` file and use `importCaffeLayers` to import the network layers into MATLAB. For more information, see `importCaffeLayers`.

Export to and Import from ONNX

By using ONNX as an intermediate format, you can interoperate with other deep learning frameworks that support ONNX model export or import, such as TensorFlow, PyTorch, Caffe2, Microsoft® Cognitive Toolkit (CNTK), Core ML, and Apache MXNet™.

Export a trained Deep Learning Toolbox network to the ONNX model format by using the `exportONNXNetwork` function. You can then import the ONNX model to other deep learning frameworks that support ONXX model import.

Import pretrained networks from ONNX using `importONNXNetwork` and import network architectures with or without weights using`importONNXLayers`.

References

[1] *ImageNet*. http://www.image-net.org

[2] Russakovsky, O., Deng, J., Su, H., et al. “ImageNet Large Scale Visual Recognition Challenge.” *International Journal of Computer Vision (IJCV)*. Vol 115, Issue 3, 2015, pp. 211–252

[3] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Antonio Torralba, and Aude Oliva. "Places: An image database for deep scene understanding." *arXiv preprint arXiv:1610.02055* (2016).

[4] *Places*. http://places2.csail.mit.edu/

[5] *Caffe Model Zoo*. http://caffe.berkeleyvision.org/model_zoo.html

See Also

`alexnet` | `densenet201` | `exportONNXNetwork` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` |`importKerasNetwork` | `importONNXLayers` | `importONNXNetwork` | `inceptionresnetv2` | `inceptionv3` | `resnet101` | `resnet18` | `resnet50` |`squeezenet` | `vgg16` | `vgg19`

Related Topics

- Deep Learning in MATLAB
- Extract Image Features Using Pretrained Network
- Classify Image Using GoogLeNet
- Train Deep Learning Network to Classify New Images
- Visualize Features of a Convolutional Neural Network
- Visualize Activations of a Convolutional Neural Network
- Deep Dream Images Using AlexNet

4. Παράδειγμα: Deep Learning for Image Classification

Avi's pick of the week is The Deep Learning Toolbox Team. AlexNet is a pre-trained 1000-class image classifier using deep learning more specifically a convolutional neural networks (CNN). The support package provides easy access to this powerful model to help quickly get started with deep learning in MATLAB.

Contents

Access the pre-trained model in MATLAB

Once you have downloaded and installed the support package, you can load the pre-trained model into MATLAB.

```
net = alexnet
net =

  SeriesNetwork with properties:

    Layers: [25×1 nnet.cnn.layer.Layer]
```

View network architecture

Now let's take a quick look at the structure of the deep neural network layers.

```
net.Layers
ans =

  25x1 Layer array with layers:

     1   'data'     Image Input                   227x227x3 images
with 'zerocenter' normalization
     2   'conv1'    Convolution                   96 11x11x3
convolutions with stride [4  4] and padding [0  0]
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel
normalization with 5 channels per element
     5   'pool1'    Max Pooling                   3x3 max pooling
with stride [2  2] and padding [0  0]
     6   'conv2'    Convolution                   256 5x5x48
convolutions with stride [1  1] and padding [2  2]
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel
normalization with 5 channels per element
```

```
     9   'pool2'    Max Pooling                  3x3 max pooling
with stride [2  2] and padding [0  0]
    10   'conv3'    Convolution                  384 3x3x256
convolutions with stride [1  1] and padding [1  1]
    11   'relu3'    ReLU                         ReLU
    12   'conv4'    Convolution                  384 3x3x192
convolutions with stride [1  1] and padding [1  1]
    13   'relu4'    ReLU                         ReLU
    14   'conv5'    Convolution                  256 3x3x192
convolutions with stride [1  1] and padding [1  1]
    15   'relu5'    ReLU                         ReLU
    16   'pool5'    Max Pooling                  3x3 max pooling
with stride [2  2] and padding [0  0]
    17   'fc6'      Fully Connected              4096 fully
connected layer
    18   'relu6'    ReLU                         ReLU
    19   'drop6'    Dropout                      50% dropout
    20   'fc7'      Fully Connected              4096 fully
connected layer
    21   'relu7'    ReLU                         ReLU
    22   'drop7'    Dropout                      50% dropout
    23   'fc8'      Fully Connected              1000 fully
connected layer
    24   'prob'     Softmax                      softmax
    25   'output'   Classification Output        cross-entropy with
'tench', 'goldfish', and 998 other classes
```

Classify image

Now lets try and classify an image using deep learning. We need to resize the input images to match the input of the network that is 227 x 227 pixels before classifying the image.

```
I = imread('Keyboard.jpg');

% Adjust size of the image
sz = net.Layers(1).InputSize
I = imresize(I,[sz(1) sz(2)]);

% Classify the image using AlexNet
label = classify(net, I)

% Show the image and the classification results
figure
imshow(I)
text(10,20,char(label),'Color','white')
sz =

   227   227     3


label =

     typewriter keyboard
```

5. Deep Network Designer App

Edit and build deep learning networks

Description

The Deep Network Designer app lets you build, visualize, and edit deep learning networks. Using this app, you can:

- Import pretrained networks and edit them for transfer learning.
- Import and edit networks and build new networks.
- Drag and drop to add new layers and create new connections.
- View and edit layer properties.
- Analyze the network to ensure you define the architecture correctly, and detect problems before training.
- Generate MATLAB® code.

  After you finish designing a network, you can export it to the workspace, where you can save or train the network.

Open the Deep Network Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `deepNetworkDesigner`.

6. Image Processing and Computer Vision with MATLAB

Semantic segmentation, object detection, and image recognition. Computer vision applications integrated with deep learning provide advanced algorithms with deep learning accuracy. MATLAB® provides an environment to design, create, and integrate deep learning models with computer vision applications.

You can easily get started with specialized functionality for computer vision such as:

- Image and video labeling apps
- Image datastore to handle large amounts of data for training, testing, and validation
- Image and computer vision-specific preprocessing techniques
- Ability to import deep learning models from TensorFlow™-Keras and PyTorch for image recognition

https://www.mathworks.com/solutions/deep-learning/deep-learning-computer-vision.html

Deep Learning for Computer Vision

In this presentation, you'll discover how to use computer vision and image processing techniques in MATLAB to solve practical image analysis, automation, and detection problems using real-world examples. Explore the latest features in image processing and computer vision such as interactive apps, new image enhancement algorithms, data preprocessing for deep learning, and 3D algorithms.

https://www.mathworks.com/products/image.html

https://www.mathworks.com/videos/image-processing-made-easy-81718.html

https://www.mathworks.com/videos/image-processing-and-computer-vision-with-matlab-1524489939916.html

Appendix 1.

SAR Target Classification using Deep Learning

This example uses: Deep Learning Toolbox

This example shows how to create and train a simple convolution neural network to classify SAR targets using deep learning.

Deep learning is a powerful technique that can be used to train robust classifier. It has shown its effectiveness in diverse areas ranging from image analysis to natural language processing. These developments have huge potential for SAR data analysis and SAR technology in general, slowly being realized. A major task for SAR-related algorithms has long been object detection and classification, which is called automatic target recognition (ATR). Here we used a simple convolution neural network to train and classify SAR targets using Deep Learning Toolbox™.

The Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

This example demonstrates how to:

- Download Data set.
- Load and analyze image data.
- Splitting and augmentation of the data.
- Define the network architecture.
- Train the network.
- Predict the labels of new data and calculate the classification accuracy.

To illustrate this workflow, we will use the Moving and Stationary Target Acquisition and Recognition (MSTAR) Mixed Targets dataset published by the Air Force

Research Laboratory [1]. Our goal is to develop a model to classify ground targets based on SAR imagery.

Download Data set

This example uses MSTAR target dataset contains 8688 SAR images from 7 ground vehicle and a calibration target. The data was collected using an X-band sensor in spotlight mode, with a 1-foot resolution. The type of target we used are BMP2 (Infantry Fighting Vehicle), BTR70 (armored car), and T72 (tank). The images were captured at two different depression angles 15 degree and 17 degree with 190 ~ 300 different aspect versions, which are full aspect coverage over 360 degree. Optical images and SAR images of these three types of targets and their replicate targets are shown below in figure.

Download the dataset from the given URL using the `helperDownloadMSTARTargetData` helper function, defined at the end of this example. The size of data set is 28MiB.

```
outputFolder = pwd;
dataURL = ['https://ssd.mathworks.com/supportfiles/radar/data/' ...
    'MSTAR_TargetData.tar.gz'];
helperDownloadMSTARTargetData(outputFolder,dataURL);
```

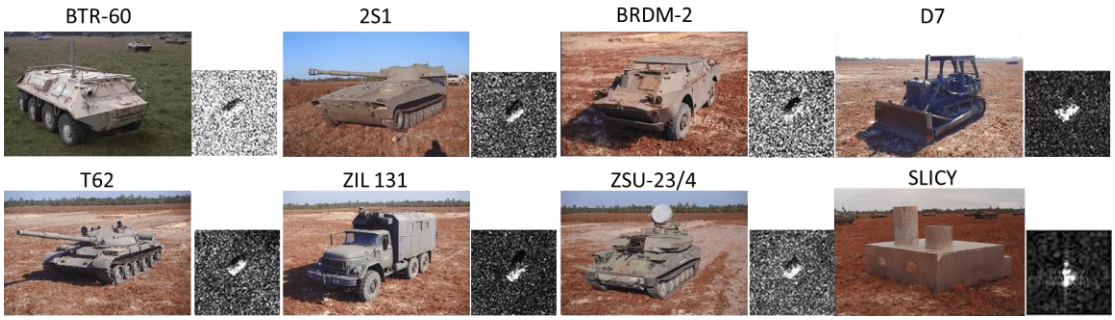
Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser and extract the file. If you do so, change the outputFolder variable in the code to the location of the downloaded file.

Load and analyze Image Data

Load the SAR image data as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `imageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

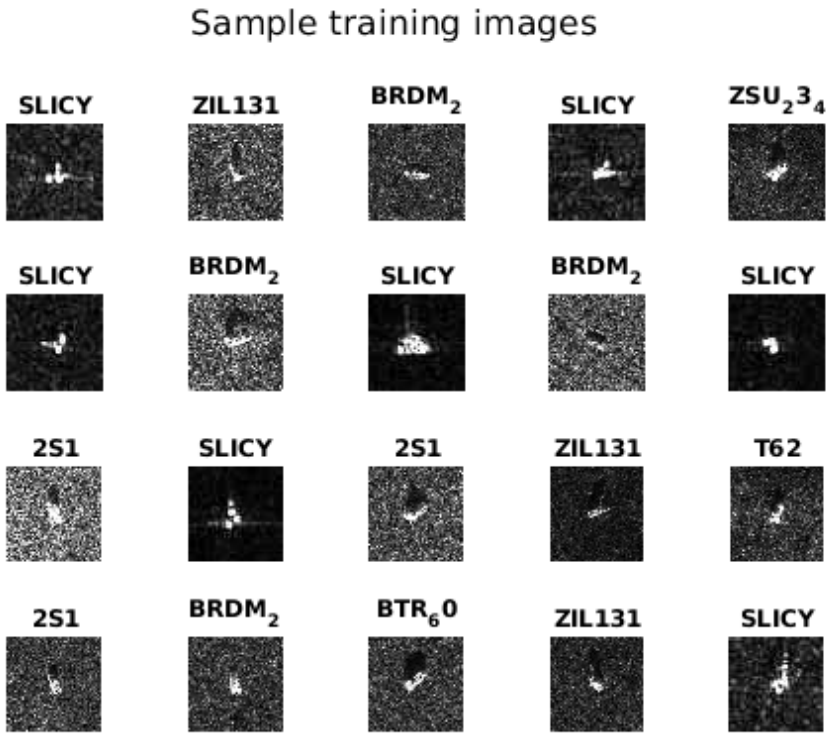
```
sarDatasetPath = fullfile(pwd,'Data');
imds = imageDatastore(sarDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
```

The MSTAR dataset contains SAR returns from 7 ground vehicles and a calibration target. Optical images and SAR images for these 8 targets are shown below



Let's explore the datastore by randomly displaying some chip images.

```
rng(0)
figure
% Shuffle the datastore.
imds = shuffle(imds);
for i = 1:20
    subplot(4,5,i)
    img = read(imds);
    imshow(img)
    title(imds.Labels(i))
    sgtitle('Sample training images')
end
```



The `imds` variable now contains the images and the category labels associated with each image. The labels are automatically assigned from the folder names of the image files. Use `countEachLabel` to summarize the number of images per category.

```
labelCount = countEachLabel(imds)
labelCount=8×2 table
     Label       Count
    ________     _____

    2S1          1164
    BRDM_2       1415
    BTR_60        451
    D7            573
    SLICY        2539
    T62           572
    ZIL131        573
    ZSU_23_4     1401
```

First, specify the network input size. When choosing the network input size, consider the memory constraint of your system and the computation cost incurred in training.

```
imgSize = [128,128,1];
```

Create Datastore Object for Training, Validation and Testing

Divide the data into training, validation and test sets. We will use 80% of our dataset for training, 10% for model validation during training, and 10% for testing after training. `splitEachLabel` splits the datastore `imds` into three new datastores, `imdsTrain`, `imdsValidation`, and `imdsTest`. In doing so, it considers the varying number of images of different classes, so that the training, validation, and test sets have the same proportion of each class.

```
trainingPct = 0.8;
validationPct = 0.1;
[imdsTrain,imdsValidation,imdsTest] = splitEachLabel(imds,...
    trainingPct,validationPct,'randomize');
```

Data Augmentation

The images in the datastore do not have a consistent size. To train the images with our network, the image size must match the size of the network's input layer. Instead of resizing the images ourselves, we can use an `augmentedImageDatastore`, which will automatically resize the images before passing them into the network. The `augmentedImageDatastore` can also be used to apply transformations, such as rotation, reflection, or scaling, to the input images. This is useful to keep the network from overfitting to our data.

```
auimdsTrain = augmentedImageDatastore(imgSize, imdsTrain);
auimdsValidation = augmentedImageDatastore(imgSize, imdsValidation);
auimdsTest = augmentedImageDatastore(imgSize, imdsTest);
```

Define Network Architecture

Define the convolutional neural network architecture.

```
layers = createNetwork(imgSize);
```

Train Network

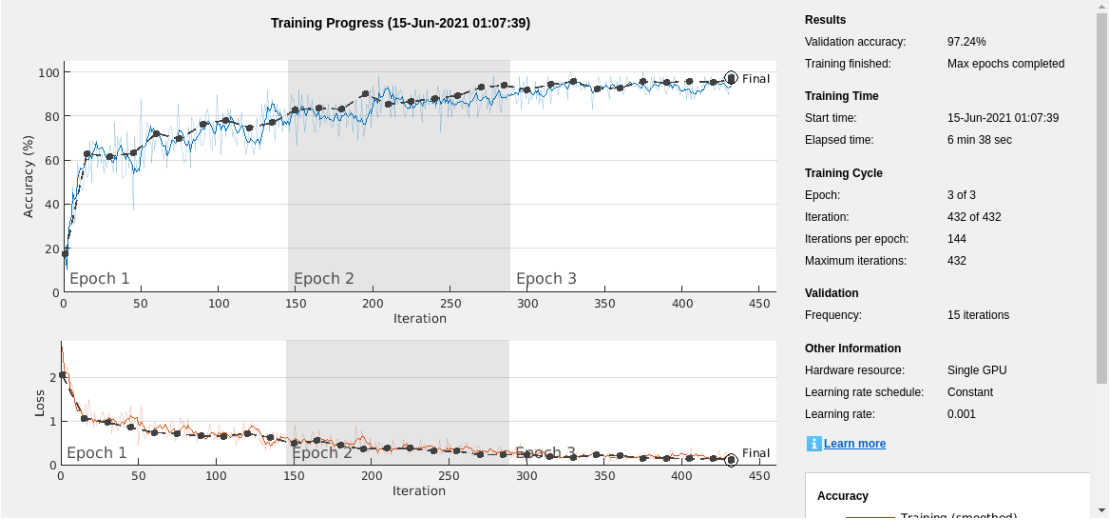
After defining the network structure, use `trainingOptions` (Deep Learning Toolbox) to specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.001. We set the maximum number of epochs to 3. An epoch is a full training cycle 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. We set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.001, ...
    'MaxEpochs',3, ...
    'Shuffle','every-epoch', ...
    'MiniBatchSize',48,...
    'ValidationData',auimdsValidation, ...
    'ValidationFrequency',15, ...
    'Verbose',false, ...
    'CheckpointPath',tempdir,...
    'Plots','training-progress');
```

Train the network using the architecture defined by `layers`, the training data, and the training options. 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). For information about the supported compute capabilities, see GPU Support by Release (Parallel Computing Toolbox). 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. For more information on the training progress plot, see Monitor Deep Learning Training Progress (Deep Learning Toolbox). The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

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



The training process is displayed in the image above. The dark blue line on the upper plot indicates the model's accuracy on the training data, while the black dashed line indicates the model's accuracy on the validation data (separate from training). The validation accuracy is more than 97%, which is quite good for an eight-class classifier. Furthermore, note that the validation accuracy and training accuracy are similar. This indicates that we have a robust classifier. When the training accuracy is much higher than the validation accuracy, the model is overfitting to (i.e. memorizing) the training data.

Classify Test Images and Compute Accuracy

Predict the labels of the validation data using the trained network and calculate the final accuracy. Accuracy is the fraction of labels that the network predicts correctly.

```
YPred = classify(net,auimdsTest);
YTest = imdsTest.Labels;

accuracy = sum(YPred == YTest)/numel(YTest)
accuracy = 0.9827
```

The test accuracy is very close to the validation accuracy, giving us confidence in the model's predictive ability.

We can use a confusion matrix to study the model's classification behavior in greater detail. A strong center diagonal represents accurate predictions. Ideally, we would like to see small, randomly located values off the diagonal. Large values off the diagonal could indicate specific scenarios where the model struggles.

```
figure
cm = confusionchart(YPred, YTest);
cm.RowSummary = 'row-normalized';
cm.Title = 'SAR Target Classification Confusion Matrix';
```

Out of the eight classes, the model appears to struggle the most with correctly classifying the ZSU-23/4. The ZSU-23/4 and 2S1 have very similar SAR images and hence we observe some misclassification by our trained model. However, it is still able to achieve greater than 90% accuracy for the class.

Helper Function

The function `createNetwork` takes as input the image input size `imgSize` and returns a convolution neural network. See below for a description of what each layer type does.

Image Input Layer — An `imageInputLayer` (Deep Learning Toolbox) is where you specify the image size. These numbers correspond to the height, width, and the channel size. The SAR image 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 — In the convolutional layer, the 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` (Deep Learning Toolbox).

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` (Deep Learning Toolbox) to create a batch normalization layer.

ReLU Layer — The batch normalization layer is followed by a nonlinear activation function. The most common activation function is the rectified linear unit (ReLU). Use `reluLayer` (Deep Learning Toolbox) to create a ReLU layer.

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` (Deep Learning Toolbox). 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` (Deep Learning Toolbox) 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` (Deep Learning Toolbox) 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` (Deep Learning Toolbox).

```
function layers = createNetwork(imgSize)
    layers = [
        imageInputLayer([imgSize(1) imgSize(2) 1])      % Input Layer
        convolution2dLayer(3,32,'Padding','same')       % Convolution
Layer
        reluLayer                                       % Relu Layer
        convolution2dLayer(3,32,'Padding','same')
        batchNormalizationLayer                         % Batch
normalization Layer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)                 % Max Pooling
Layer

        convolution2dLayer(3,64,'Padding','same')
        reluLayer
        convolution2dLayer(3,64,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(3,128,'Padding','same')
        reluLayer
        convolution2dLayer(3,128,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(3,256,'Padding','same')
        reluLayer
```

```
        convolution2dLayer(3,256,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(6,512)
        reluLayer

        dropoutLayer(0.5)                                   % Dropout
Layer
        fullyConnectedLayer(512)                            % Fully
connected Layer.
        reluLayer
        fullyConnectedLayer(8)
        softmaxLayer                                        % Softmax
Layer
        classificationLayer                                 %
Classification Layer
        ];
end

function helperDownloadMSTARTargetData(outputFolder,DataURL)
% Download the data set from the given URL to the output folder.

    radarDataTarFile =
fullfile(outputFolder,'MSTAR_TargetData.tar.gz');

    if ~exist(radarDataTarFile,'file')

        disp('Downloading MSTAR Target data (28 MiB)...');
        websave(radarDataTarFile,DataURL);
        untar(radarDataTarFile,outputFolder);
    end
end
```

References

[1] MSTAR Dataset. https://www.sdms.afrl.af.mil/index.php?collection=mstar