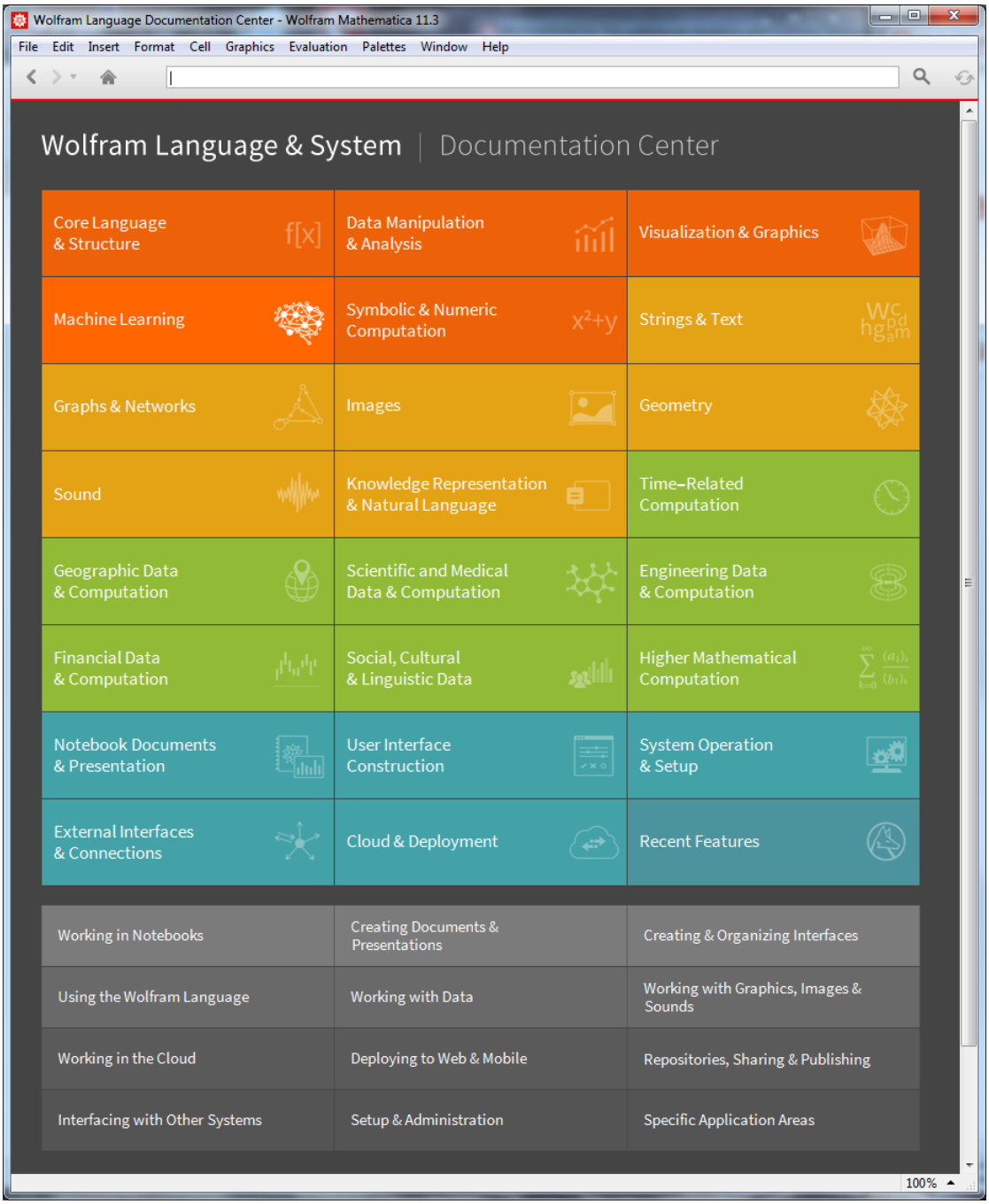
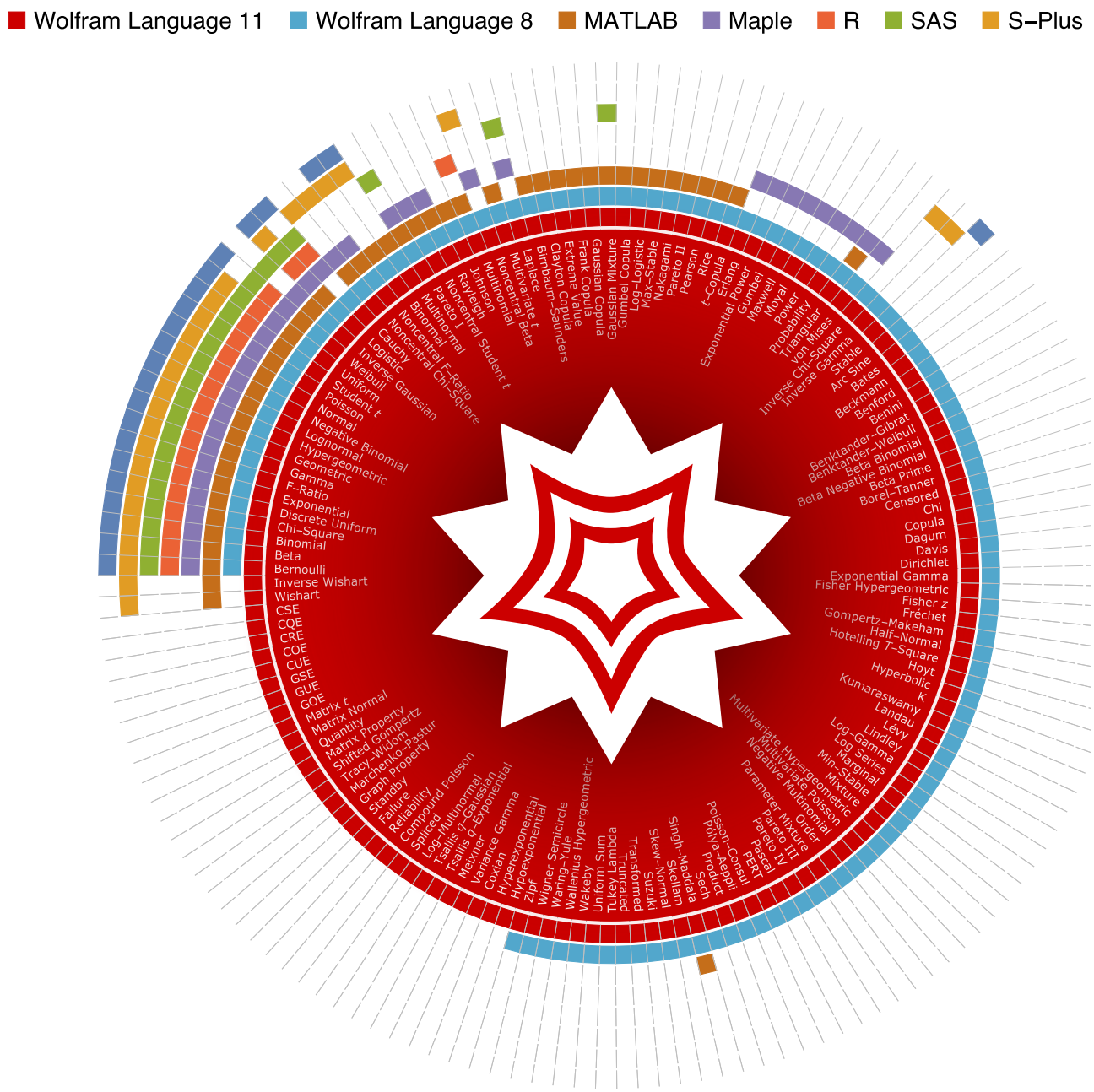
Predictive Analytics World Machine Learning and Neural Networks using the Wolfram Language


Wolfram Language Documentation Center - Wolfram Mathematica 11.3
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
Wolfram Language & System | Documentation Center
Core Language & Structure
Data Manipulation & Analysis
Visualization & Graphics
Machine Learning
Symbolic & Numeric Computation
Strings & Text
Graphs & Networks
Images
Geometry
Sound
Knowledge Representation & Natural Language
Time-Related Computation
Geographic Data & Computation
Scientific and Medical Data & Computation
Engineering Data & Computation
Financial Data & Computation
Social, Cultural & Linguistic Data
Higher Mathematical Computation
Notebook Documents & Presentation
User Interface Construction
System Operation & Setup
External Interfaces & Connections
Cloud & Deployment
Recent Features
Working in Notebooks
Creating Documents & Presentations
Creating & Organizing Interfaces
Using the Wolfram Language
Working with Data
Working with Graphics, Images & Sounds
Working in the Cloud
Deploying to Web & Mobile
Repositories, Sharing & Publishing
Interfacing with Other Systems
Setup & Administration
Specific Application Areas
100%

Statistical Distributions underlie Data Predictions

Mathematica has more distributions than any other programming language

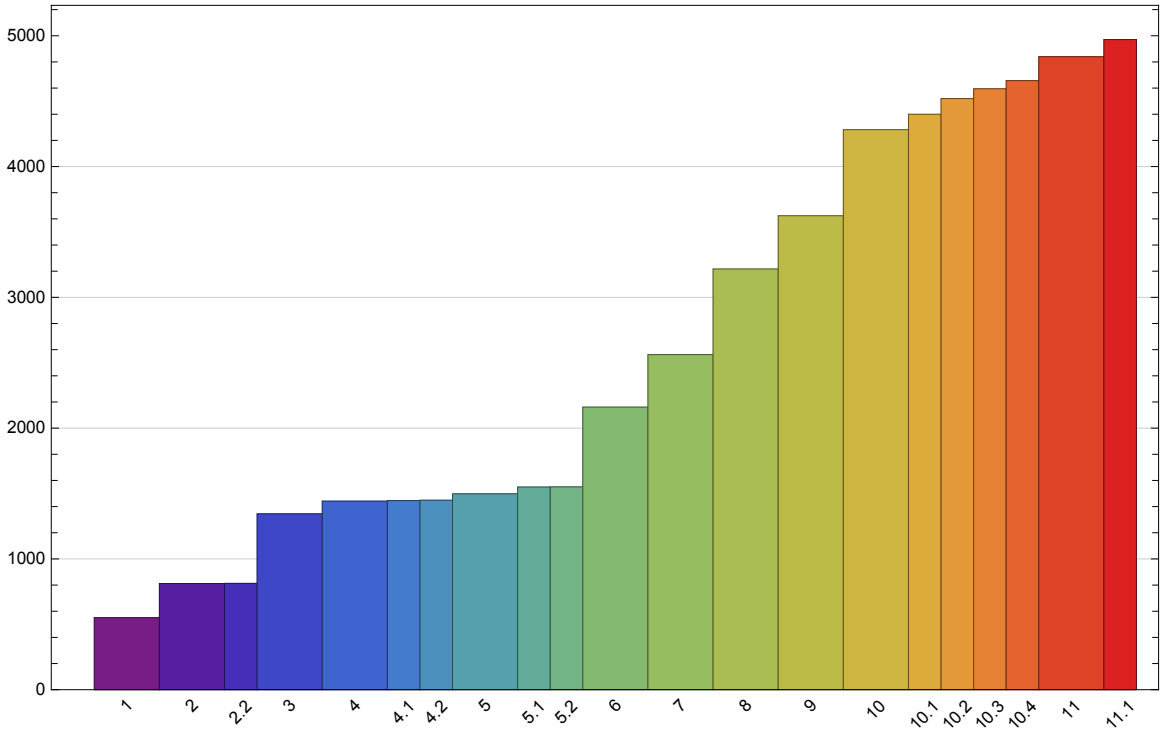


© The Wolfram Companies. Trademarks: Wolfram, Wolfram Language. All other trademarks, service marks, registered trademarks and registered service marks are the tive owners.

WolframLanguage is constantly growing to adapt to new

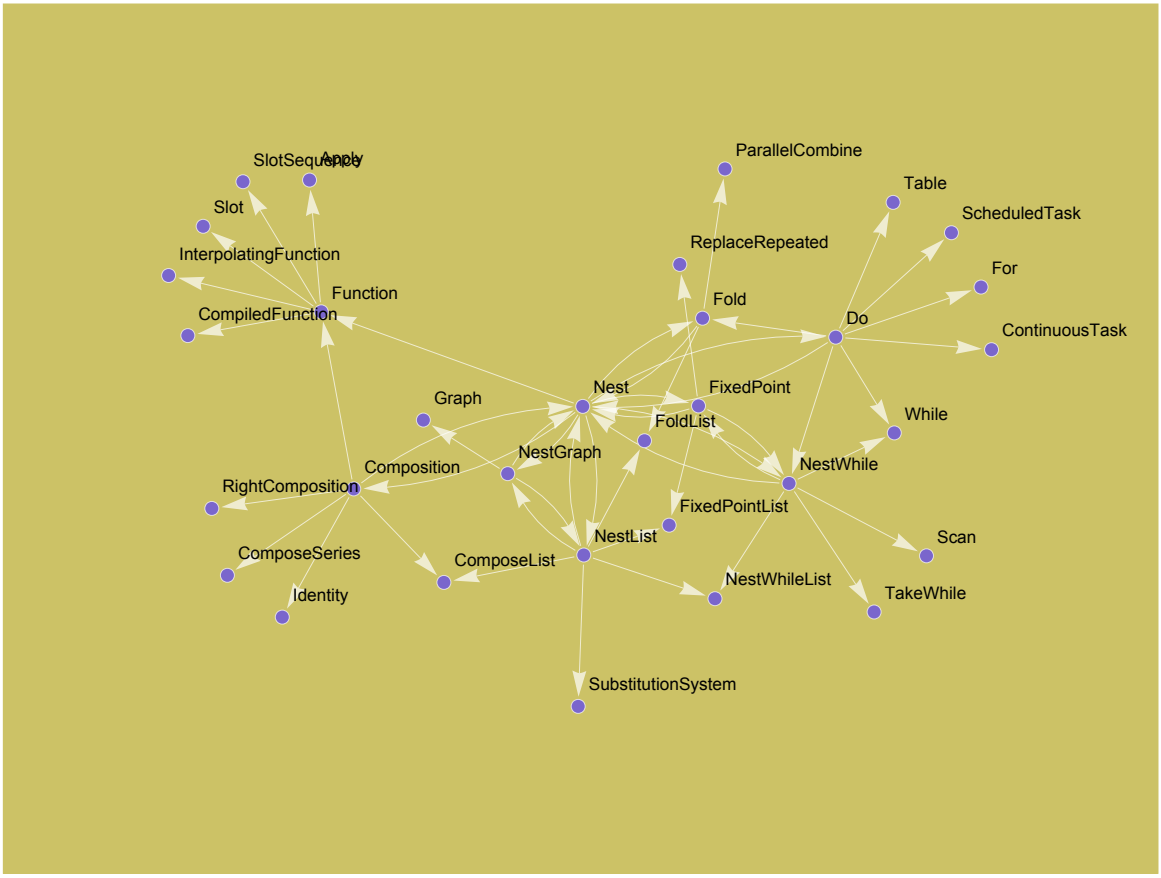
knowledge

All Releases of Mathematica



WolframLanguage knows about itself:
Generate a network of "nearby" functions in the Wolfram Language:

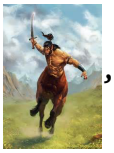
```
NestGraph[WolframLanguageData[#, related symbols] &, Nest WOLFRAM LANGUAGE SYMBOL, 2,
 VertexLabels → "Name", GraphStyle → "BackgroundGold", PlotRangePadding → 1, ImageSize → 600]
```



Inbuilt Classify

Classify Images

Train a classifier on 32 images of legendary creatures:

In[ ]:=
```
legendary = Classify[<|
  "Griffin" → {  ,  ,  ,
     ,  ,  ,  ,  },
  "Centaur" → {  ,  ,  ,  ,
     ,  ,  ,  },
  "Dragon" → {  ,  ,  ,  ,
     ,  ,  ,  },
  "Unicorn" → {  ,  ,  ,  ,  ,
     ,  ,  |>]
```

Out[ ]=
```
ClassifierFunction[   Input type: Image
                                                Classes: Centaur, Dragon, Griffin, Unicorn ]
```

Use the classifier to recognize unseen creatures:

In[ ]:= `legendary[{`  `,`  `,`  `,`  `}]`

Out[ ]= {Griffin, Centaur, Dragon, Unicorn}

Classify variables in data sets

Load the "Titanic" dataset, which contains a list of *Titanic* passengers with their age, sex, ticket class, and survival:

In[15]:=
```
dataset = ExampleData[{"MachineLearning", "Titanic"}, "Data"];
```

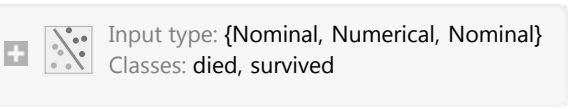
Visualize a sample of the dataset:

In[16]:=
```
RandomSample[dataset, 10] // TableForm
```

Out[16]//TableForm=
```
{3rd, Missing[], male} → died
{2nd, 19., male} → died
{2nd, 50., male} → died
{3rd, 14., male} → died
{3rd, 30., female} → died
{1st, 18., female} → survived
{1st, Missing[], male} → died
{3rd, 16., male} → died
{3rd, 23., male} → died
{1st, 49., male} → survived
```

Train a logistic classifier on this dataset:

In[17]:=
```
c = Classify[dataset, Method → "LogisticRegression"]
```

Out[17]= `ClassifierFunction[`  `]`

Calculate the survival probability of a 10-year-old girl traveling in third class:

In[18]:=
```
c[{"3rd", 10, "female"}, "Probability" → "survived"]
```

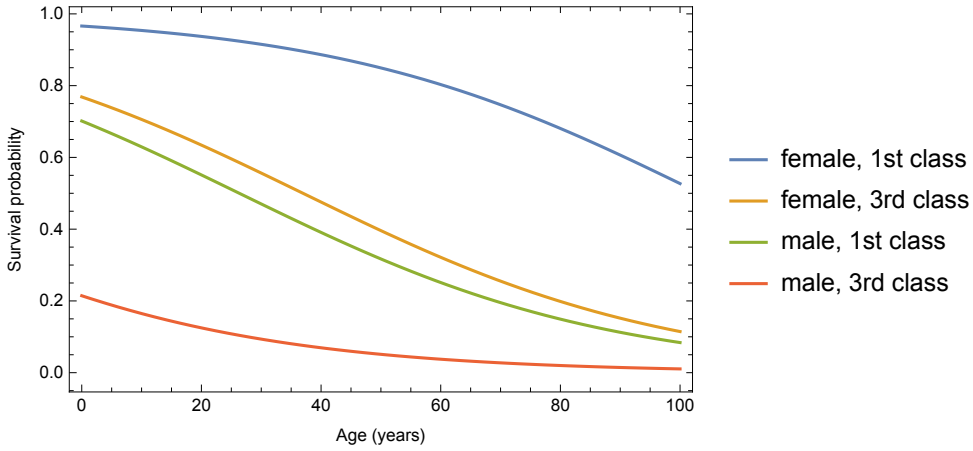
Out[18]= 0.705805

Plot the survival probability as a function of age for some combinations of "class" and "sex":

In[19]:=
```
p[class_, age_, sex_] := c[{class, age, sex}, {"Probability", "survived"}];
```

In[20]:=
```
Plot[{p["1st", x, "female"], p["3rd", x, "female"],
  p["1st", x, "male"], p["3rd", x, "male"]}, {x, 0, 100}, PlotLegends →
  {"female, 1st class", "female, 3rd class", "male, 1st class", "male, 3rd class"},
 Frame → True, FrameLabel → {"Age (years)", "Survival probability"}, Exclusions → None]
```

Out[20]=



Inbuilt Predict

Predict temperature distributions in different cities for different months

Load a dataset of the average monthly temperature as a function of the city, the year, and the month:

In[21]:=
```
dataset = RandomSample[{#2, ToExpression[#3], #4} → (#1 - 32) / 1.8 & @@@
    ExampleData[{"Statistics", "USCityTemperature"}]];
```

Visualize a sample of the dataset:

In[22]:=
```
RandomSample[dataset, 5] // TableForm
```

Out[22]//TableForm=
```
{Lincoln, 1964, August} → 22.2222
{Lincoln, 1964, March} → 1.83333
{Lincoln, 1971, June} → 25.5556
{Newark, 1967, August} → 23.0556
{Lincoln, 1965, September} → 15.6111
```

Train a linear predictor on the dataset:

In[23]:=
```
p = Predict[dataset, Method → "LinearRegression"]
```

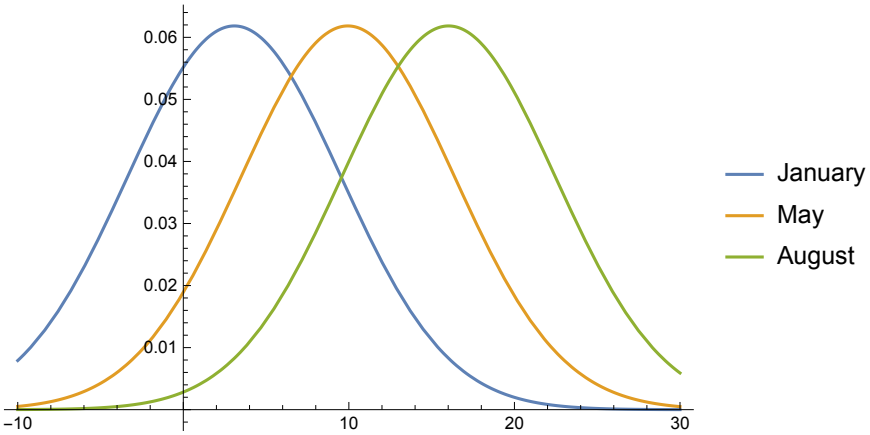
Out[23]=
```
PredictorFunction[
```

Input type: {Nominal, Numerical, Nominal}
Method: LinearRegression
]

Plot the predicted temperature distribution of the city "Lincoln" in 2020 for different months:

In[24]:=
```
Plot[{
  PDF[p[{"Lincoln", 2020, "January"}, "Distribution"], x],
  PDF[p[{"Lincoln", 2020, "May"}, "Distribution"], x],
  PDF[p[{"Lincoln", 2020, "August"}, "Distribution"], x]
 }, {x, -10, 30}, PlotLegends → {"January", "May", "August"}]
```

Out[24]=



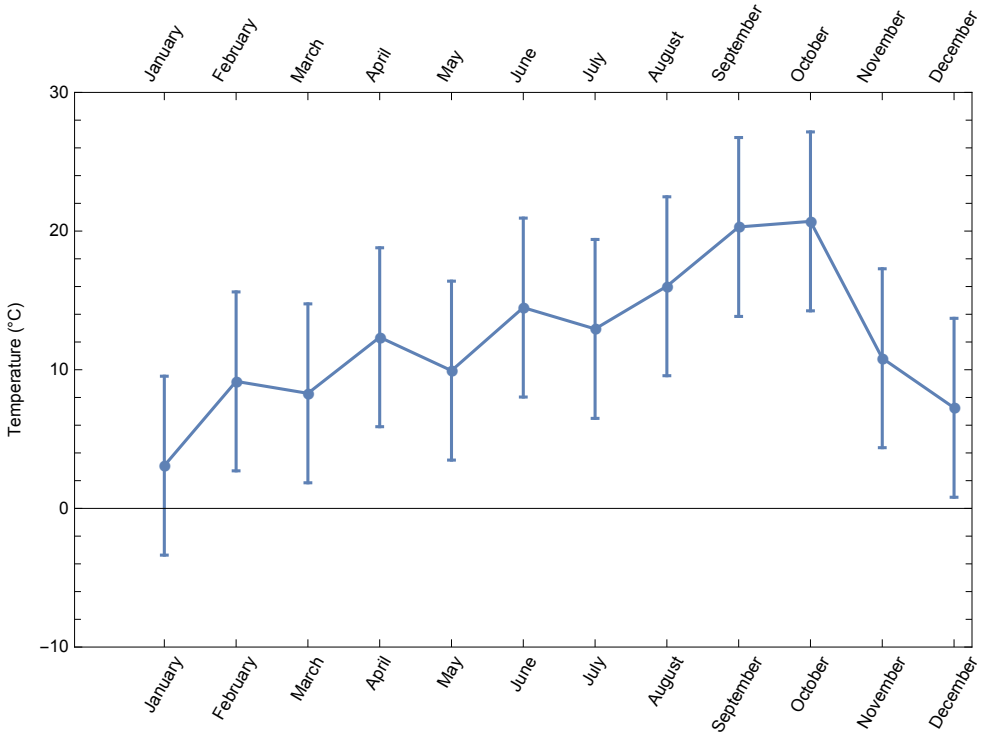
For every month, plot the predicted temperature and its error bar (standard deviation):

In[25]:=
```
months = {"January", "February", "March", "April", "May", "June",
   "July", "August", "September", "October", "November", "December"};
```

In[26]:=
```
distributions = MapIndexed[
   {First[#2], p[{"Lincoln", 2020, #1}, "Distribution"]} &, months];
```

In[27]:=
```
Needs["ErrorBarPlots`"]
ErrorListPlot[
 {{#1, #2[[1]]}, ErrorBar[#2[[2]]]} & @@@ distributions,
 FrameTicks → {Transpose[{Range[12], Rotate[#, Pi / 3] & /@ months}], Automatic},
 PlotMarkers → Automatic, Frame → True, Joined → True,
 PlotRange → {-10, 30}, FrameLabel → {None, "Temperature (°C)"}]
```

Out[28]=



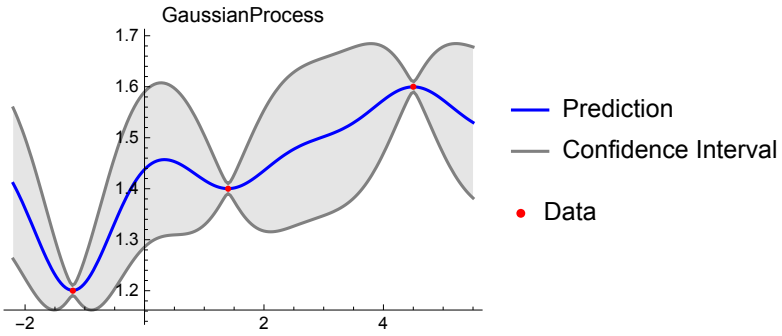
Create a function to visualize the predictions of a given method after learning from 1D data:

In[29]:=
```
visualizePrediction[data_, method_] := Module[
   {p, predictionplot, dataplot, xs},
   dataplot = ListPlot[List @@@ data, PlotStyle → Red, PlotLegends → {"Data"}];
   xs = data[[All, 1]];
   p = Predict[data, Method → method];
   predictionplot = Plot[{
      p[x],
      p[x] + StandardDeviation[p[x, "Distribution"]],
      p[x] - StandardDeviation[p[x, "Distribution"]]
     }, {x, Min[xs] - 1, Max[xs] + 1}, PlotStyle → {Blue, Gray, Gray},
     Filling → {2 → {3}}, Exclusions → False, PerformanceGoal → "Speed",
     PlotLegends → {"Prediction", "Confidence Interval"}];
   Show[predictionplot, dataplot, PlotLabel → method, ImageSize → 250]
  ];
```

Try the function with the "GaussianProcess" method on a simple dataset:

In[30]:=
```
visualizePrediction[{-1.2 → 1.2, 1.4 → 1.4, 3.1 → 1.8, 4.5 → 1.6}, "GaussianProcess"]
```

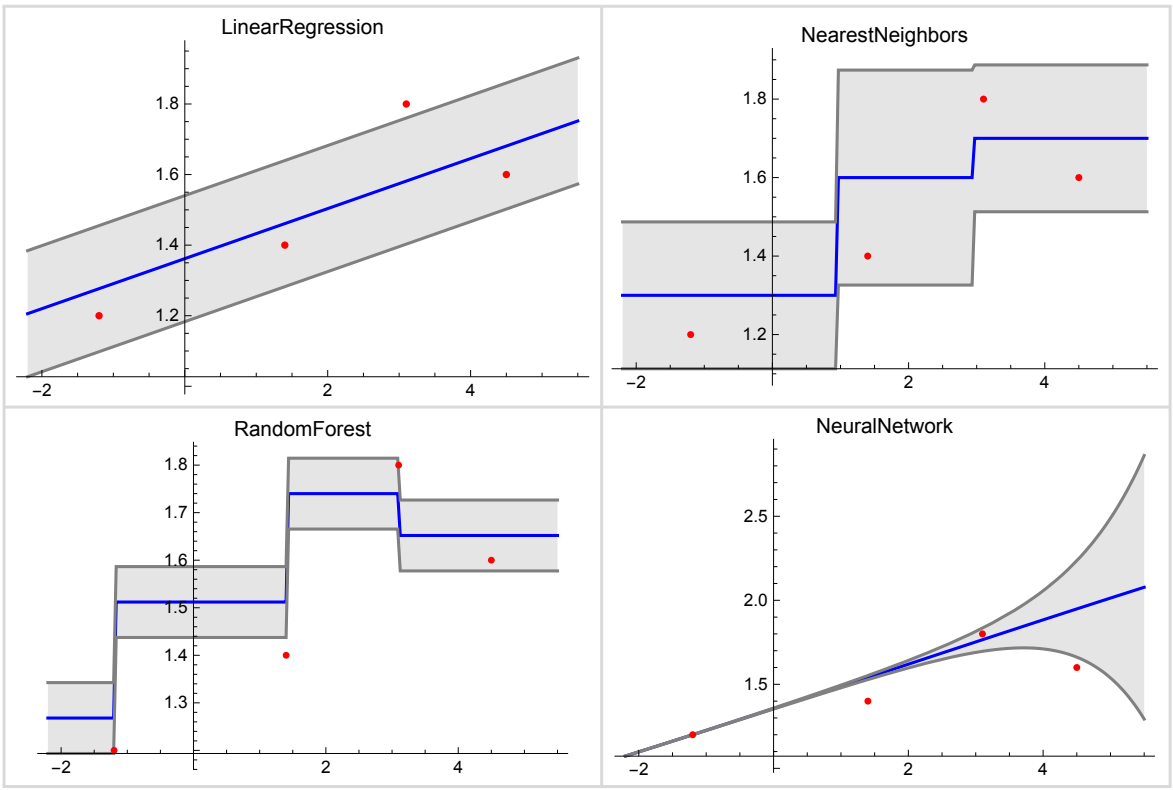
Out[30]=



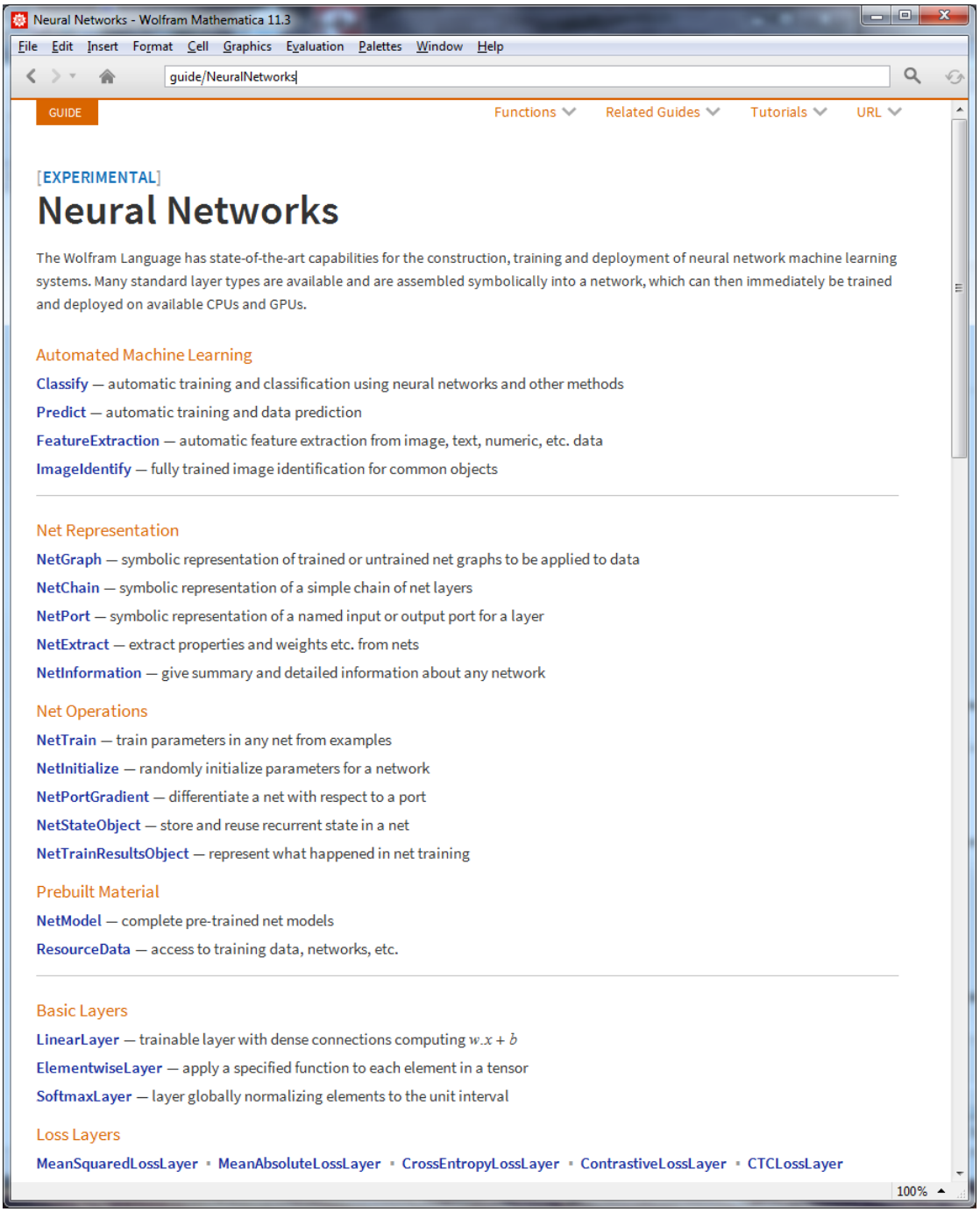
Visualize the prediction of other methods:

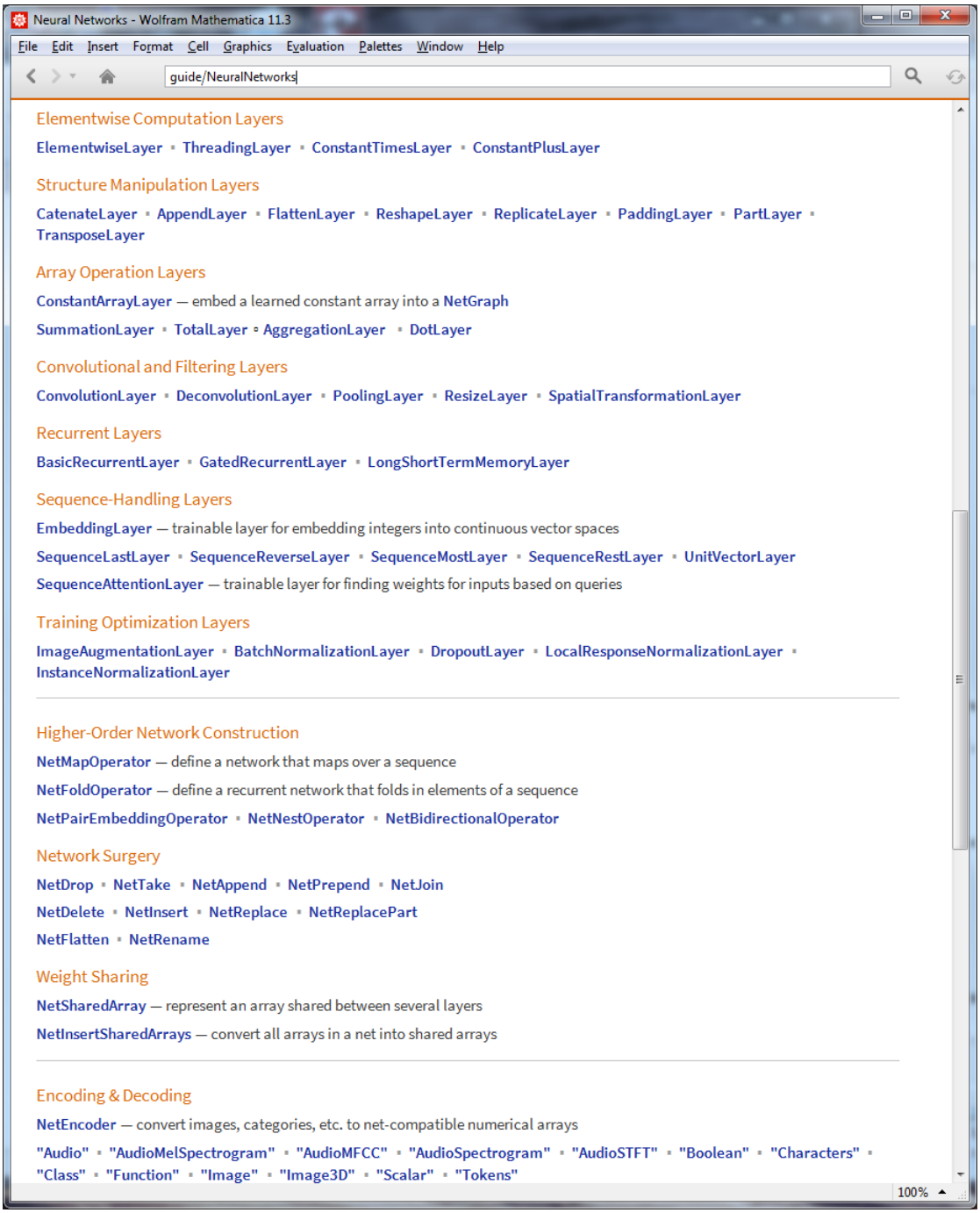
In[31]:=
```
Grid[
 Partition[visualizePrediction[{-1.2 → 1.2, 1.4 → 1.4, 3.1 → 1.8, 4.5 → 1.6}, #][[1, 1]] & /@
   {"LinearRegression", "NearestNeighbors", "RandomForest", "NeuralNetwork"}, 2],
 Frame → All, FrameStyle → LightGray]
```

Out[31]=



Neural Networks are inbuilt





Top Level Representation of Neural Networks: Simple Examples

Construct a chain:

In[6]:=
```
net = NetChain[{LinearLayer[3], Tanh, LinearLayer[4], Ramp, LinearLayer[5]}, "Input" → 2]
```

Out[6]=
```
NetChain[ uninitialized  Input port: vector (size: 2)
                         Output port: vector (size: 5)
                         Number of layers: 5 ]
```

The layers used to construct a NetChain can be extracted using Normal:

In[7]:=
```
Normal[net]
```

Out[7]=
```
{LinearLayer[ uninitialized  Input: vector (size: 2)
                             Output: vector (size: 3) ],

 ElementwiseLayer[  Function: Tanh
                    Output: vector (size: 3) ],

 LinearLayer[ uninitialized  Input: vector (size: 3)
                             Output: vector (size: 4) ],

 ElementwiseLayer[  Function: Ramp
                    Output: vector (size: 4) ],

 LinearLayer[ uninitialized  Input: vector (size: 4)
                             Output: vector (size: 5) ]}
```

NetChain objects can be viewed graphically as levels in a NetGraph:

In[8]:=
```
NetGraph[net]
```

Out[8]=
```
NetGraph[ uninitialized  Input port: vector (size: 2)
                         Output port: vector (size: 5)
                         Number of layers: 5 ]
```

Construct a new network consisting of the first two layers:

In[9]:=
```
NetTake[net, 2]
```

Out[9]=
```
NetChain[ uninitialized  Input port: vector (size: 2)
                         Output port: vector (size: 3)
                         Number of layers: 2 ]
```

Construct a new network consisting of the last two layers:

In[10]:=
```
NetTake[net, -2]
```

Out[10]=
```
NetChain[ uninitialized | Input port: vector (size: 4)
                          Output port: vector (size: 5)
                          Number of layers: 2 ]
```

In[12]:=
```
net = NetInitialize[net]
```

Out[12]=
```
NetChain[ | Input port: vector (size: 2)
          | Output port: vector (size: 5)
          | Number of layers: 5 ]
```

Apply the net to an input:

In[14]:=
```
net[{1, 5}]
```

Out[14]=
```
{0.351691, -0.304539, 1.25159, 0.577513, -0.173458}
```

Create a net with two outputs:

In[•]:=
```
net = NetGraph[{Ramp, LogisticSigmoid},
  {1 → 2, 1 → NetPort["Output1"], 2 → NetPort["Output2"]}]
```

Out[•]=
```
NetGraph[ | Input port: tensor
          | Number of outputs: 2
          | Number of layers: 2 ]
```

Apply the net to an input:

In[•]:=
```
net[{-1, 0, 1}]
```

Out[•]=
```
⟨|Output1 → {0., 0., 1.}, Output2 → {0.5, 0.5, 0.731059}|⟩
```

Get the output from a specific port:

In[•]:=
```
net[{-1, 0, 1}, "Output1"]
```

Out[•]=
```
{0., 0., 1.}
```

Top Level Representation of Neural Network Information

Count the number of layers in a net:

In[32]:=
```
net = NetChain[{2, Ramp, 3, Ramp, 4}, "Input" → 1]
```

Out[32]=
```
NetChain[ uninitialized
          Input      vector (size: 1)
        1 LinearLayer vector (size: 2)
        2 Ramp       vector (size: 2)
        3 LinearLayer vector (size: 3)
        4 Ramp       vector (size: 3)
        5 LinearLayer vector (size: 4)
          Output     vector (size: 4) ]
```

```
In[ ]:= NetInformation[net, "LayersCount"]
Out[ ]= 5
```

Count the number of arrays in a net:

```
In[ ]:= NetInformation[net, "ArraysCount"]
Out[ ]= 6
```

Give the dimensions of all arrays in the net:

```
In[33]:= NetInformation[net, "ArraysDimensions"]
Out[33]= ⟨| {1, Biases} → {2}, {1, Weights} → {2, 1}, {3, Biases} → {3},
 {3, Weights} → {3, 2}, {5, Biases} → {4}, {5, Weights} → {4, 3} |⟩
```

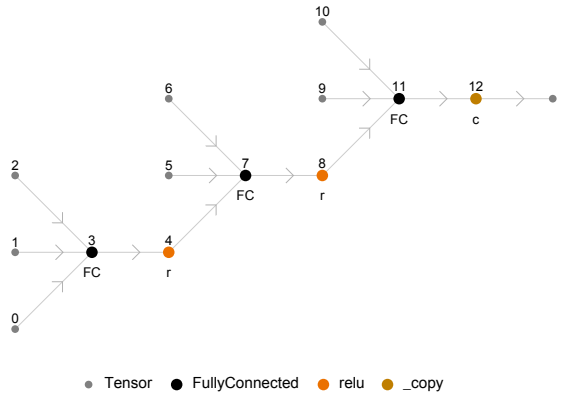
Count the types of layers that are present:

```
In[34]:= NetInformation[net, "LayerTypeCounts"]
Out[34]= ⟨| LinearLayer → 3, ElementwiseLayer → 2 |⟩
```

Plot the underlying operations of the net when compiled to "MXNet":

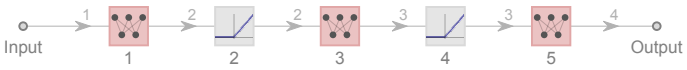
```
In[35]:= NetInformation[net, "MXNetNodeGraphPlot"]
```

Out[35]=




```
In[36]:= NetInformation[net, "SummaryGraphic"]
```

Out[36]=



Obtain a NetTrainResultsObject for a training session:

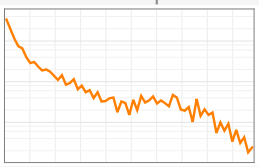
In[37]:=
```
net = NetModel["LeNet"];
data = RandomSample[ResourceData["MNIST"], 1000];
results = NetTrain[net, data, All, MaxTrainingRounds → 20]
```

Out[39]=

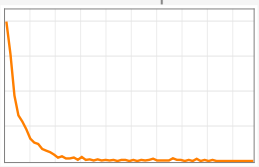
NetTrainResultsObject

Total training time:	24 s
Total rounds:	20
Total batches:	320
Batch size:	64
Method:	ADAM
Final round loss:	0.00184
Final round error:	0%

Loss evolution plot:



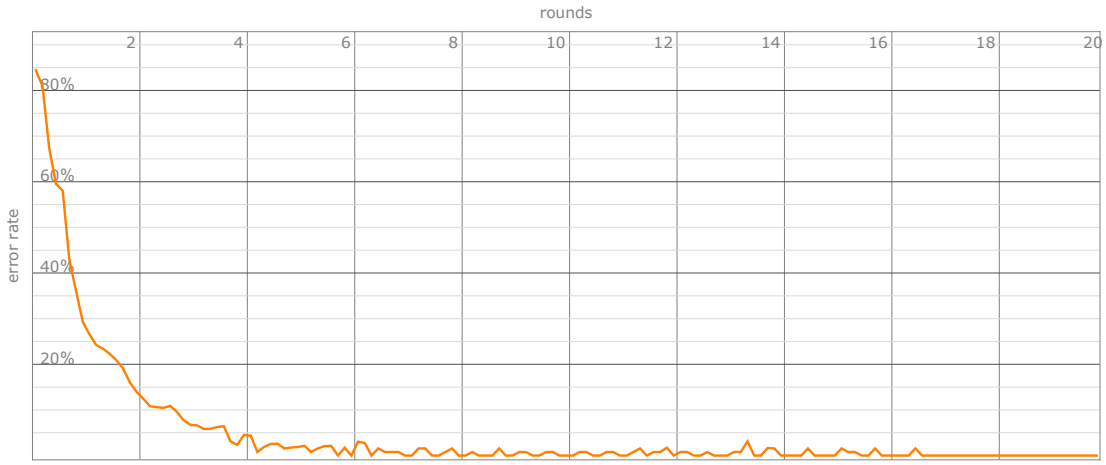
Error evolution plot:



Query the results object for specific properties:

In[40]:=
```
results["ErrorRateEvolutionPlot"]
```

Out[40]=



In[41]:=
```
results["FinalRoundErrorRate"]
```

Out[41]=
```
0.
```

In[42]:=
```
results["TotalTrainingTime"]
```

Out[42]=
```
23.9441
```

In[43]:=
```
results["TrainedNet"]
```

Out[43]=
```
NetChain[
```

		image
	Input	3–tensor (size: 1×28×28)
1	ConvolutionLayer	3–tensor (size: 20×24×24)
2	Ramp	3–tensor (size: 20×24×24)
3	PoolingLayer	3–tensor (size: 20×12×12)
4	ConvolutionLayer	3–tensor (size: 50×8×8)
5	Ramp	3–tensor (size: 50×8×8)
6	PoolingLayer	3–tensor (size: 50×4×4)
7	FlattenLayer	vector (size: 800)
8	LinearLayer	vector (size: 500)
9	Ramp	vector (size: 500)
10	LinearLayer	vector (size: 10)
11	SoftmaxLayer	vector (size: 10)
	Output	class

```
]
```

Get a list of all available properties:

In[44]:=
```
results["Properties"]
```

Out[44]=
```
{BatchErrorRateList, BatchLossList, BatchSize, CheckpointingFiles,
 CompactErrorRateEvolutionPlot, CompactLossEvolutionPlot, ErrorRateEvolutionPlot,
 EvolutionPlots, FinalLearningRate, FinalRoundErrorRate, FinalRoundLoss,
 FinalValidationErrorRate, FinalValidationLoss, InitialLearningRate,
 LossEvolutionPlot, LowestValidationErrorRate, LowestValidationLoss,
 LowestValidationRound, MeanBatchesPerSecond, MeanInputsPerSecond,
 NetTrainInputForm, OptimizationMethod, RoundErrorRateList, RoundLossList,
 TotalBatches, TotalInputs, TotalRounds, TotalTrainingTime, TrainedNet, TrainingNet,
 ValidationErrorRateList, ValidationErrorRateSeries, ValidationLossList,
 ValidationLossSeries, VersionNumber, WeightsLearningRateMultipliers}
```

Training Neural Networks

Train a three-layer network to learn a 2D function:

In[45]:=
```
data = Flatten@Table[{x, y} → Exp[-Norm[{x, y}]], {x, -3, 3, .005}, {y, -3, 3, .005}];
net = NetChain[{32, Tanh, 1}];
trained = NetTrain[net, data, BatchSize → 1024]
```

Out[47]=
```
NetChain[
```

	Input	vector (size: 2)
1	LinearLayer	vector (size: 32)
2	Tanh	vector (size: 32)
3	LinearLayer	vector (size: 1)
	Output	scalar

```
]
```

Evaluate the network on an input:

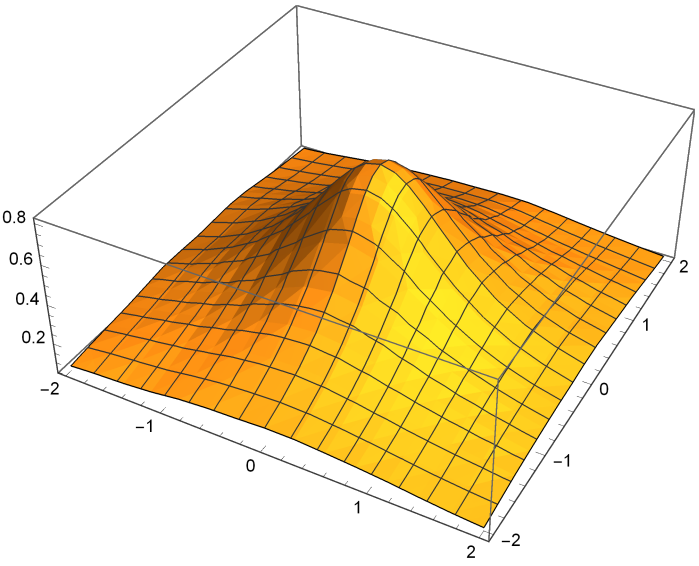
In[48]:=
```
trained[{1, 0}]
```

Out[48]=
```
0.37573
```

Plot the prediction of the net as a function of x and y:

In[49]:=
```
Plot3D[trained[{x, y}], {x, -2, 2}, {y, -2, 2}, NormalsFunction → None]
```

Out[49]=



Show the difference that results from different Initializations:

Construct a base network that takes vector inputs of size 2 and produces vector outputs of size 3:

In[ ]:=
```
net = NetChain[{30, Sin, 3, Tanh, 3, LogisticSigmoid}, "Input" → 2]
```

Out[ ]=

NetChain [uninitialized
Input port: vector (size: 2)
Output port: vector (size: 3)
Number of layers: 6]

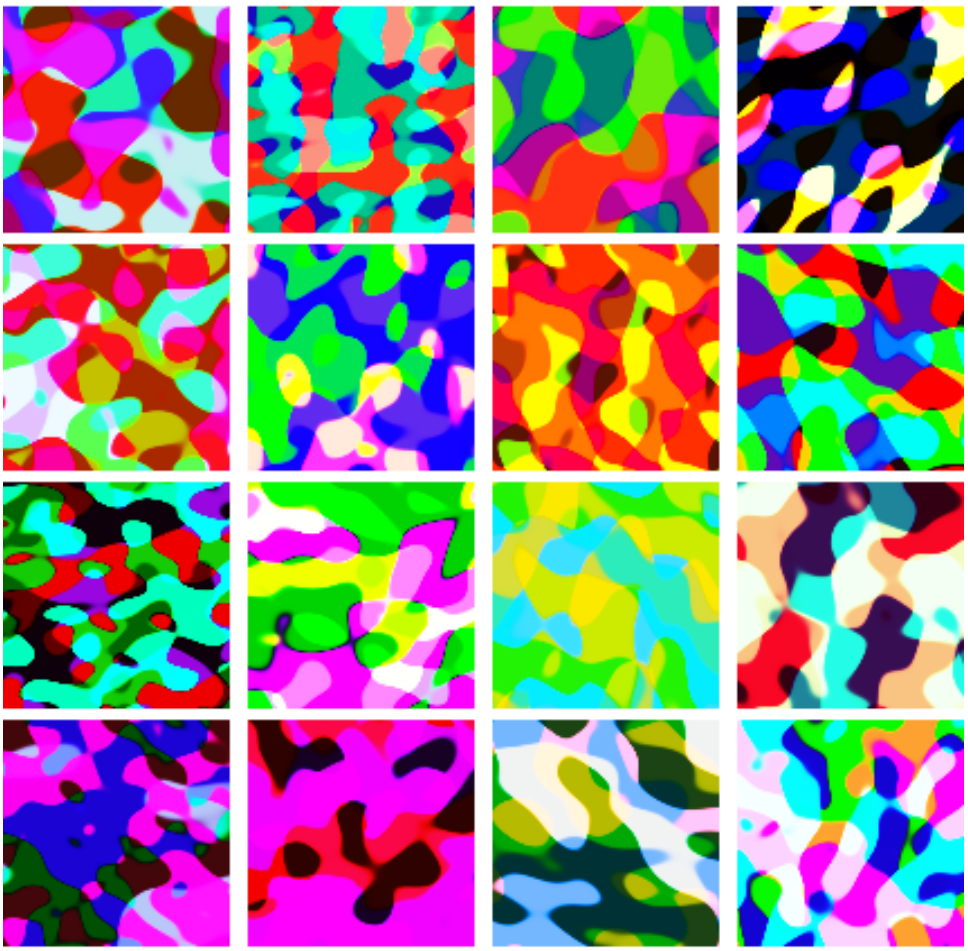
Make a table of 16 randomly initialized copies of the base network:

In[ ]:=
```
nets = Table[
   NetInitialize[net, Method → {"Random", "Weights" → 3, "Biases" → 2}], 16];
```

Plot them in a gallery by treating them as functions mapping (x, y) positions to (r, g, b) color values:

In[ ]:=
```
row = Range[-2, 2, 0.04];
coords = Tuples[row, 2];
plot[net_] := Image[Partition[net[coords], Length[row]]];
Multicolumn@Table[plot[net], {net, nets}]
```

Out[ ]=



Convert a test image into a training set, in which pixel positions (x, y) are mapped to color values (r, g, b):

In[ ]:=
```
img =
```

```
;
```

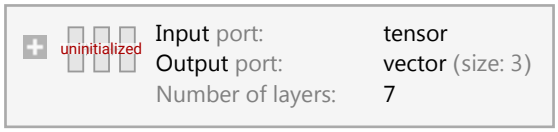
In[ ]:=
```
dims = Reverse@ImageDimensions[img];
rules = Flatten@MapIndexed[(2 (#2 - 1.) / (dims - 1) - 1.) → #1 &, ImageData@img, {2}];
```

In[ ]:=
```
RandomChoice[rules]
```

Out[ ]=
```
{0.472803, -0.435737} → {0.678431, 0.309804, 0.298039}
```

Create a network to predict the color based on pixel position:

In[ ]:=
```
chain = NetChain[{100, Ramp, 250, Ramp, 10, LogisticSigmoid, 3}]
```

Out[ ]=
```
NetChain[
```


```
]
```

Train the network:

In[ ]:=
```
trained = NetTrain[chain, rules, MaxTrainingRounds → 100];
```

Use the network to predict the entire original image:

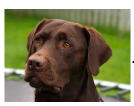
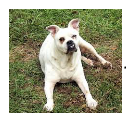
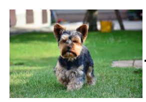
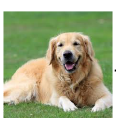
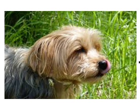
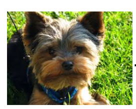
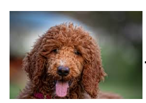
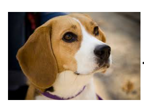
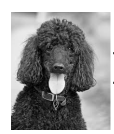
In[∘]:=
```
{h, v} = Range[-1, 1, 2. / #] & /@ (dims);
coords = Tuples[{h, v}];
Image[Partition[trained@coords, Length[v]]]
```

Out[∘]=


Feature Extraction

Extract Nearest Images

Construct a dataset of dog images:

In[∘]:= **dataset = {**  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  **};**

Train an extractor function from this dataset:

In[∘]:=
```
fe = FeatureExtraction[dataset]
```

Out[∘]= FeatureExtractorFunction[Input type: Image
Output type: NumericalVector (length: 20)]

Generate a NearestFunction on the extracted features of the dataset:

```
In[ ]:= nf = Nearest[fe[dataset] → Automatic]
```

```
Out[ ]= NearestFunction[ Data points: 21
                         Input dimension: 20 ]
```

Using the NearestFunction , construct a function that displays the nearest image of the dataset:

```
In[ ]:= nearestdog = dataset[[First@nf[fe[#]]]] &
```

```
Out[ ]= dataset⟦First[nf[fe[#1]]]⟧ &
```

Use this function on images that are not in the dataset:

```
In[ ]:= nearestdog[ ]
```

```
Out[ ]=
```

```
In[ ]:= nearestdog[ ]
```

```
Out[ ]=
```

```
In[ ]:= nearestdog[ ]
```

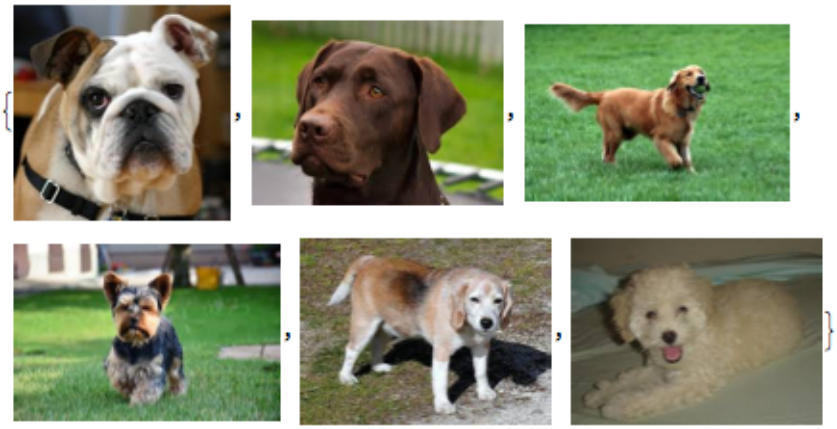
```
Out[ ]=
```

This feature extractor function can also be used to delete image pairs that are too similar:

```
In[ ]:= features = # → fe[#] & /@ dataset;
```

In[•]:=
```
First /@ DeleteDuplicates[features, CosineDistance[#1[[2]], #2[[2]]] < 1 &]
```

Out[•]=



Extract Nearest Text

Load the text of *Alice in Wonderland*:

In[•]:=
```
alice = ExampleData[{"Text", "AliceInWonderland"}];
```

Split the text into sentences:

In[•]:=
```
sentences = TextSentences[alice];
```

Train a feature extractor on these sentences:

In[•]:=
```
fe = FeatureExtraction[sentences]
```

Out[•]=
```
FeatureExtractorFunction[ Input type: Text
                          Output type: NumericalVector (length: 549) ]
```

Generate a NearestFunction with the sentences' features:

In[•]:=
```
nf = Nearest[fe[sentences] → Automatic]
```

Out[•]=
```
NearestFunction[ Data points: 553
                 Input dimension: 549 ]
```

Using the NearestFunction, construct a function that displays the nearest sentence in *Alice in Wonderland*:

In[•]:=
```
nearestalice = sentences[[First@nf[fe[#]]]] &
```

Out[•]=
```
sentences⟦First[nf[fe[#1]]]⟧ &
```

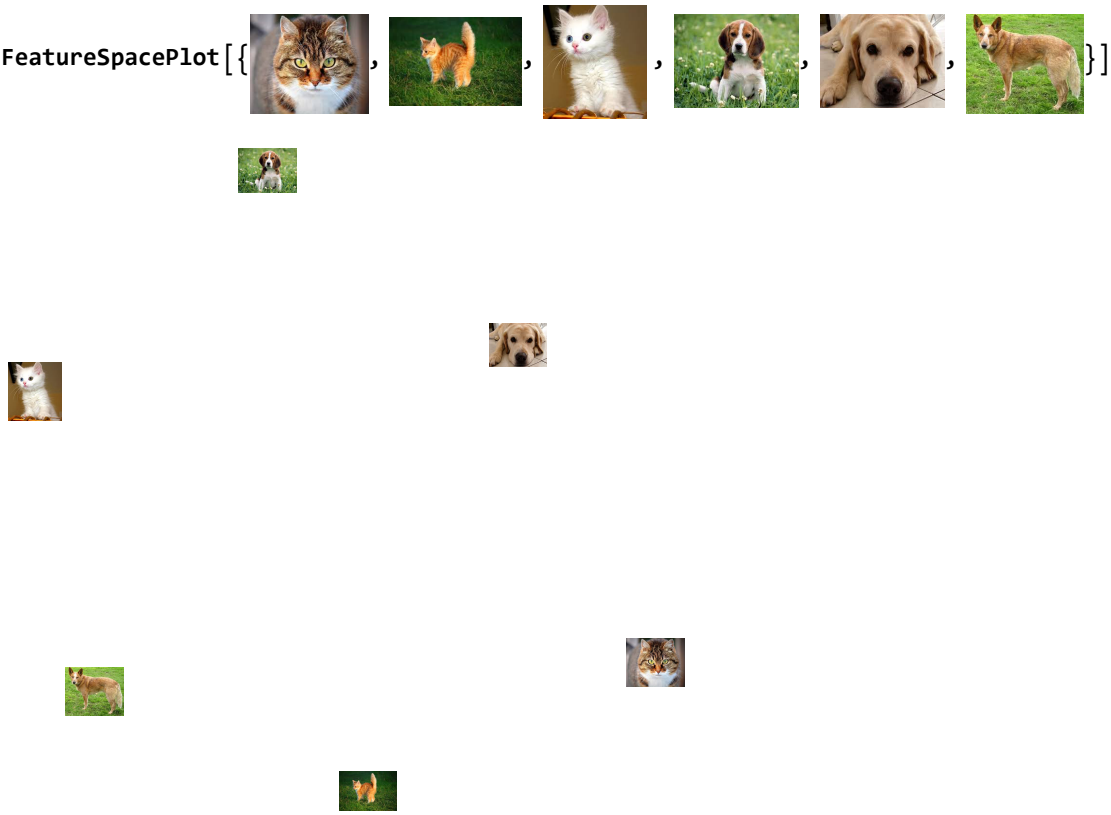
Use this function with a few queries:

```
In[ ]:= nearestalice["Alice and the Rabbit"]
Out[ ]= Alice knew it was the Rabbit coming to look for her and she trembled
          till she shook the house, quite forgetting that she was now about a
          thousand times as large as the Rabbit and had no reason to be afraid of it.

In[ ]:= nearestalice["Alice and the Queen"]
Out[ ]= When the procession came opposite to Alice, they all
          stopped and looked at her, and the Queen said severely, "Who is this?"

In[ ]:= nearestalice["Off her head"]
Out[ ]= "Off with her head!" the Queen shouted at the top of her voice.
```

Feature Spaces

Plot the features extracted from images:

```
In[ ]:= FeatureSpacePlot[{ , , , , , }]
```

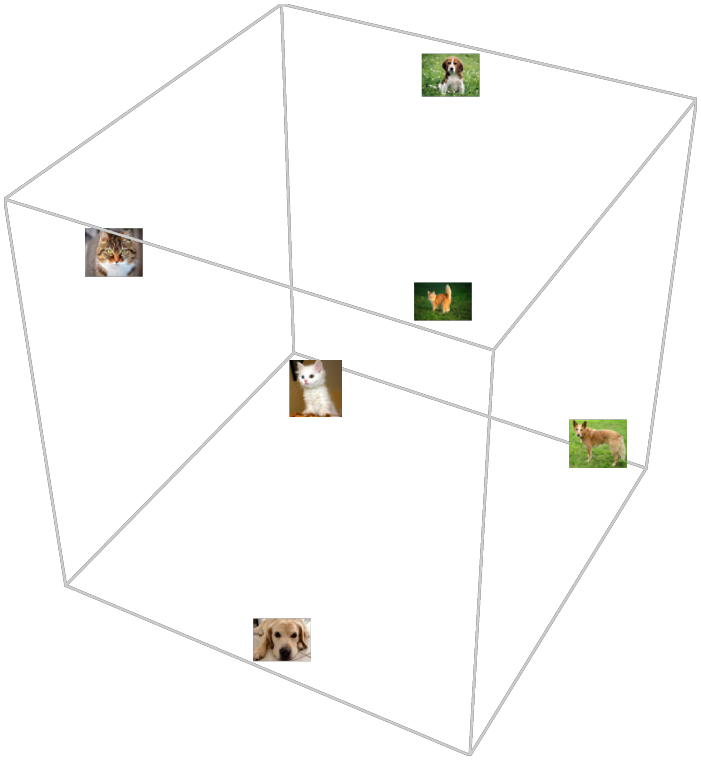


```
Out[ ]=
```

Plot the features extracted from images:

In[ ]:= `FeatureSpacePlot3D[{`  ,  ,  ,  ,  ,  `}]`

Out[ ]=



Use a gray background for the plot:

In[ ]:=
```
FeatureSpacePlot3D[EntityValue[ Europe COUNTRIES , "FlagImage"], Background → Gray]
```

Out[ ]=

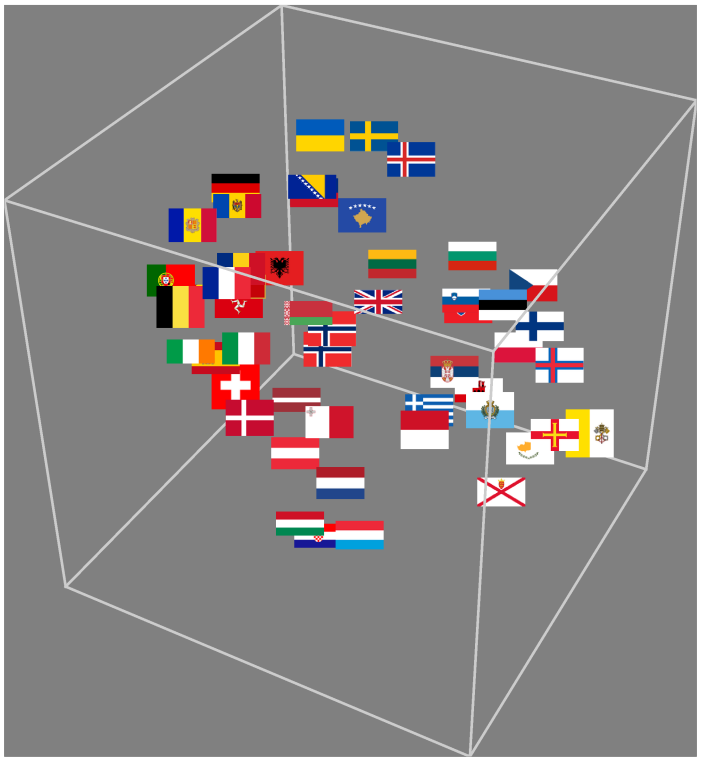


Image Recognition

ImageIdentify

Import Google image search for “jewelery”

```
jewelery =
  URLExecute["https://www.google.ie/search?q=jewelery&source=lnms&tbm=isch&sa=X&ved=0
     ahUKEwiBperDmLjRAhXJCsAKHZtEDNQQ_AUICCgB&biw=1920&bih=949", "Images"];
```

Identify objects in images:

```
Grid[Partition[Column[{#, ImageIdentify[#]}, Center] & /@ jewelery, 5]]
```


tambourine


bangle


necklace


snake


necklace


necklace


earring


lace


snowflake


locket


necklet


person


knot


Japanese apricot


tricycle


necklace


locket


plastron


lace


bolo tie

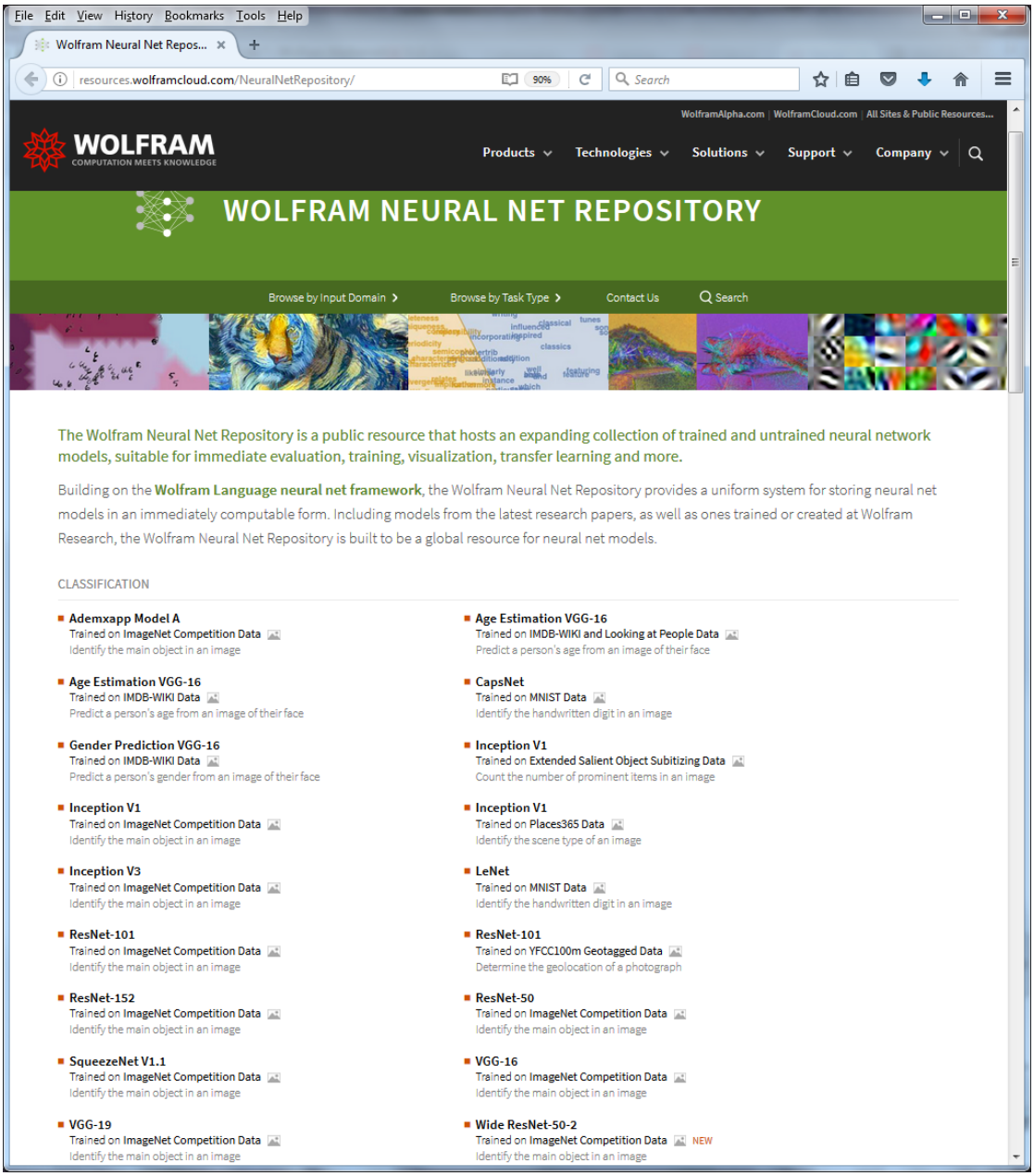
Get ten different identifications, along with their probabilities:

```
ImageIdentify[ <image> , car WORD , 10, "Probability"]
```

```
⟨| station wagon → 0.428902, limousine → 0.380134, coupe → 0.179555,
  sport utility vehicle → 0.0068486, convertible → 0.001783, automobile → 0.999995 |⟩
```

Big Data Neural Networks

Neural Net Repository



Neural Net Repository accessed by ResourceData objects and implemented by NetModel

Obtain a list of all available models:

```
In[•]:= NetModel[]
```

```
Out[•]= {2D Face Alignment Net Trained on 300W Large Pose Data,
 3D Face Alignment Net Trained on 300W Large Pose Data,
 AdaIN-Style Trained on MS-COCO and Painter by Numbers Data,
 Ademxapp Model A Trained on ImageNet Competition Data,
 Age Estimation VGG-16 Trained on IMDB-WIKI and Looking at People Data,
 Age Estimation VGG-16 Trained on IMDB-WIKI Data,
 ColorNet Image Colorization Trained on ImageNet Competition Data,
 ColorNet Image Colorization Trained on Places Data,
 ConceptNet Numberbatch Word Vectors V17.06 (Raw Model),
 Gender Prediction VGG-16 Trained on IMDB-WIKI Data,
 GloVe 100-Dimensional Word Vectors Trained on Tweets,
 GloVe 100-Dimensional Word Vectors Trained on Wikipedia and Gigaword 5 Data,
 GloVe 200-Dimensional Word Vectors Trained on Tweets,
 GloVe 25-Dimensional Word Vectors Trained on Tweets,
 GloVe 300-Dimensional Word Vectors Trained on Common Crawl 42B,
 GloVe 300-Dimensional Word Vectors Trained on Common Crawl 840B,
 GloVe 300-Dimensional Word Vectors Trained on Wikipedia and Gigaword 5 Data,
 GloVe 50-Dimensional Word Vectors Trained on Tweets,
 GloVe 50-Dimensional Word Vectors Trained on Wikipedia and Gigaword 5 Data,
 Inception V1 Trained on Extended Salient Object Subitizing Data,
 Inception V1 Trained on ImageNet Competition Data,
 Inception V1 Trained on Places365 Data,
 Inception V3 Trained on ImageNet Competition Data, LeNet, LeNet Trained on MNIST Data,
 Pix2pix Photo-to-Street-Map Translation, Pix2pix Street-Map-to-Photo Translation,
 ResNet-101 Trained on Augmented CASIA-WebFace Data,
 ResNet-101 Trained on ImageNet Competition Data,
 ResNet-101 Trained on YFCC100m Geotagged Data,
 ResNet-152 Trained on ImageNet Competition Data,
 ResNet-50 Trained on ImageNet Competition Data,
 Single-Image Depth Perception Net Trained on Depth in the Wild Data,
 Single-Image Depth Perception Net Trained on NYU Depth V2 and Depth in the Wild Data,
 Single-Image Depth Perception Net Trained on NYU Depth V2 Data,
 SqueezeNet V1.1 Trained on ImageNet Competition Data,
 Unguided Volumetric Regression Net for 3D Face Reconstruction,
 Vanilla CNN for Facial Landmark Regression, VGG-16 Trained on ImageNet Competition Data,
 VGG-19 Trained on ImageNet Competition Data,
 Wolfram ImageIdentify Net for WL 11.1, Yahoo Open NSFW Model V1}
```

Obtain the trained version of a specific neural net:

```
In[ ]:= net = NetModel["LeNet Trained on MNIST Data"]
```

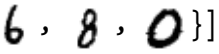
```
Out[ ]= NetChain[
```

		image
	Input	3–tensor (size: 1 × 28 × 28)
1	ConvolutionLayer	3–tensor (size: 20 × 24 × 24)
2	Ramp	3–tensor (size: 20 × 24 × 24)
3	PoolingLayer	3–tensor (size: 20 × 12 × 12)
4	ConvolutionLayer	3–tensor (size: 50 × 8 × 8)
5	Ramp	3–tensor (size: 50 × 8 × 8)
6	PoolingLayer	3–tensor (size: 50 × 4 × 4)
7	FlattenLayer	vector (size: 800)
8	LinearLayer	vector (size: 500)
9	Ramp	vector (size: 500)
10	LinearLayer	vector (size: 10)
11	SoftmaxLayer	vector (size: 10)
	Output	class

```
]
```

Apply the trained net to a set of inputs:

```
In[ ]:= net[{ , , }]
```



```
Out[ ]= {6, 8, 0}
```

Produce class probabilities for a single input:

```
In[ ]:= net[ , "Probabilities"]
```

```
Out[ ]= ⟨| 0 → 7.28386 × 10^-13, 1 → 1.8159 × 10^-16, 2 → 2.15518 × 10^-12,
 3 → 3.2635 × 10^-12, 4 → 9.74528 × 10^-15, 5 → 6.61678 × 10^-10,
 6 → 3.45442 × 10^-13, 7 → 3.57894 × 10^-20, 8 → 1., 9 → 2.42621 × 10^-12 |⟩
```