Building and Training Basic Neural Networks

Wolfram Research
training@wolfram.com

Overview

Network Data

Neural networks are built up using tensors. So the first step is to convert different data types to tensors.

- Tensors
- Encoders and Decoders

Network Structure

- Layers
- Network Constructors

Network Training

This section focuses on training the networks in an optimized fashion.

- Basic Theory
- Loss Layers

Performing Logistic Regression on Real-World Data

- Basic Layers
- Network Construction
- Accuracy Estimation

LeNet Trained on Handwritten Digits

This section of the talk focuses on logistic regression using neural networks for classification and training the LeNet (Yann LeCun) on handwritten digits. For each of the examples, each layer, encoders/decoders and constructors are considered.

- Basic Layers
- Network Construction
- Accuracy Estimation

Network Data: Tensors

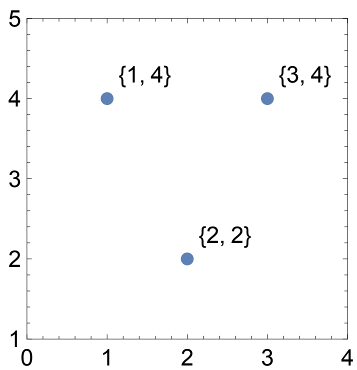
Ranks of Tensors

Tensors are multidimensional arrays.

- Rank 0 (scalars): 0.0
- Rank 1 (vectors): {0.0, 1.0}
- Rank 2 (matrices): {{1.,2.,3.} , {3., 2., 1.}}
- Rank-*n* tensors: {... {... {1., 2., 3.}...}...}

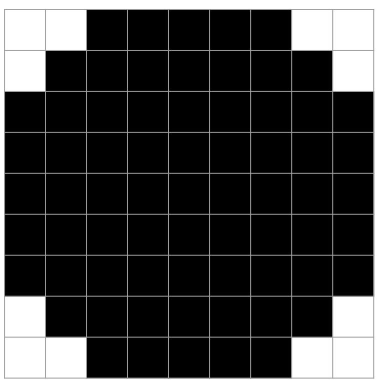
Examples of Tensors:

- Vectors as coordinates of points:



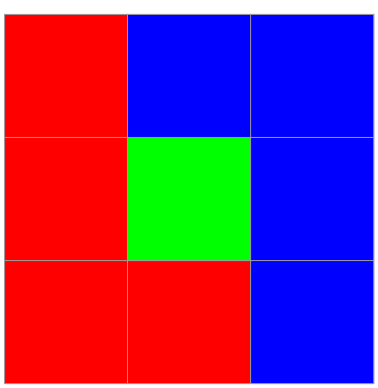
$$\approx \quad \begin{matrix} (\;1 & 4\;) \\ (\;3 & 4\;) \\ (\;2 & 2\;) \end{matrix}$$

- Matrices as grayscale images:



$$\approx \quad \begin{pmatrix} 0&0&1&1&1&1&1&0&0 \\ 0&1&1&1&1&1&1&1&0 \\ 1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1&0 \\ 0&0&1&1&1&1&1&0&0 \end{pmatrix}$$

- Rank-3 tensors as a colored image:



$$\approx \quad \left(\begin{pmatrix} 1&0&0 \\ 1&0&0 \\ 1&1&0 \end{pmatrix} \begin{pmatrix} 0&0&0 \\ 0&1&0 \\ 0&0&0 \end{pmatrix} \begin{pmatrix} 0&1&1 \\ 0&0&1 \\ 0&0&1 \end{pmatrix} \right)$$

Network Data: Encoders and Decoders

Class Encoders and Decoders

Encode the user-defined class (here the class of male and females):

```
In[•]:= enc = NetEncoder[{"Class", {"male", "female"}}]

Out[•]= $Aborted
```

Use it to create the input tensor:

```
In[•]:= enc[{"male", "female", "female"}]

Out[•]= {1, 2, 2}
```

Create the decoder for the same classes:

```
In[•]:= dec = NetDecoder[{"Class", {"male", "female"}}]

Out[•]= NetDecoder[ Type: Class
                    Input: vector (size: 2) ]
```

Use a probabilistic approach to decode the output (make sure each of the tensors to be decoded follows the dimension specification of the decoder):

```
In[•]:= dec[{{1.9, 1.9}, {0.9, 0.7}}]

Out[•]= {female, male}
```

Image Encoder

In[•]:= image =  ;

```
In[•]:= enc = NetEncoder[{"Image", ImageDimensions[image], ColorSpace → "Grayscale"}]

Out[•]= NetEncoder[ Type: Image
                    Output: array (size: 1 × 213 × 236) ]

In[•]:= encoded = enc[image];

In[•]:= dec = NetDecoder[{"Image", ColorSpace -> "Grayscale"}]

Out[•]= NetDecoder[ Type: Image
                    Input: array (rank: 3) ]
```

In[•]:=
```
dec[encoded]
```

Out[•]=

Layers

A neural network is a biologically inspired model and consists of “layers” in between the input and output tensors.

The implemented layers in the Wolfram Language™ are the following:

Layer Type	Layers
Basic Layers	LinearLayer | ElementwiseLayer | SoftmaxLayer
Elementwise Computation Layers	ElementwiseLayer | ThreadingLayer | ConstantTimesLayer | ConstantPlusPlayer
Convolution and Filtering Layers	ConvolutionLayer | DeconvolutionLayer | PoolingLayer | ResizeLayer | SpecialTransformationLayer
Training optimization Layers	ImageAugmentationLayer | BatchNormalizationLayer | DropoutLayer | LocalResponseNormalizationLayer | InstanceNormalizationLayer
Structure Manipulation Layers	CatenateLayer | FlattenLayer | ReshapeLayer | ReplicateLayer | PaddingLayer | PartLayer | TransposeLayer
Array Operation Layers	ConstantArrayLayer | SummationLayer | TotalLayer | AggregationLayer | DotLayer
Recurrent Layers	BasicRecurrentLayer | GatedRecurrentLayer | LongShortTermMemoryLayer
Sequence-Handling Layers	EmbeddingLayer | SequenceLastLayer | SequenceReverseLayer | SequenceMostLayer | SequenceRestLayer | SequenceAttentionLayer | UnitVectorLayer

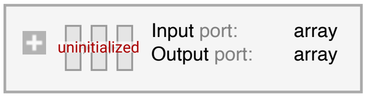
Network Constructors

NetChain

NetChain can be used to connect different layers in a chain-like fashion to create a neural net.

Create a simple chain performing logistic regression:

In[•]:=
```
net = NetChain[{LinearLayer[], ElementwiseLayer[LogisticSigmoid]}]
```

Out[•]= NetChain[


]

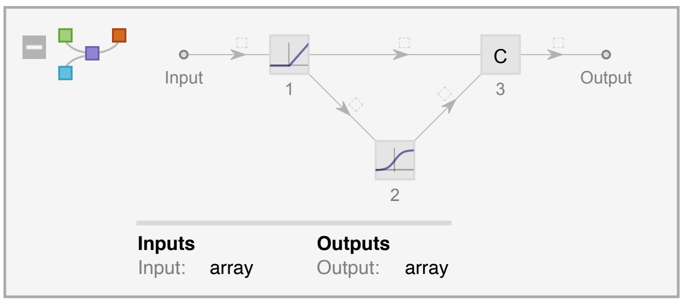
NetGraph

NetGraph can be used to connect different layers to create a graphical neural network.

Concatenating output from intermediate layers:

NetGraph[layer list, topology list: input(s)→outputs(s)]

In[•]:=
```
net = NetGraph[{Ramp, LogisticSigmoid, CatenateLayer[]}, {1 → 2, {1, 2} → 3}]
```

Out[•]= NetGraph[


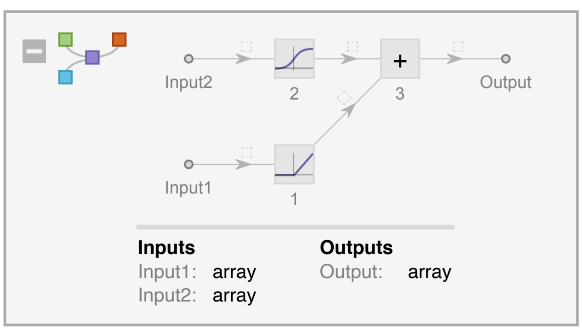
]

NetPort

NetPort represents the specified port for a layer in a **NetGraph** or similar structure.

Creating a simple total **NetGraph** with two inputs:

In[•]:=
```
net = NetGraph[{Ramp, LogisticSigmoid, TotalLayer[]}, {NetPort["Input1"] → 1, NetPort["Input2"] → 2, {1, 2} → 3}]
```

Out[•]= NetGraph[


]

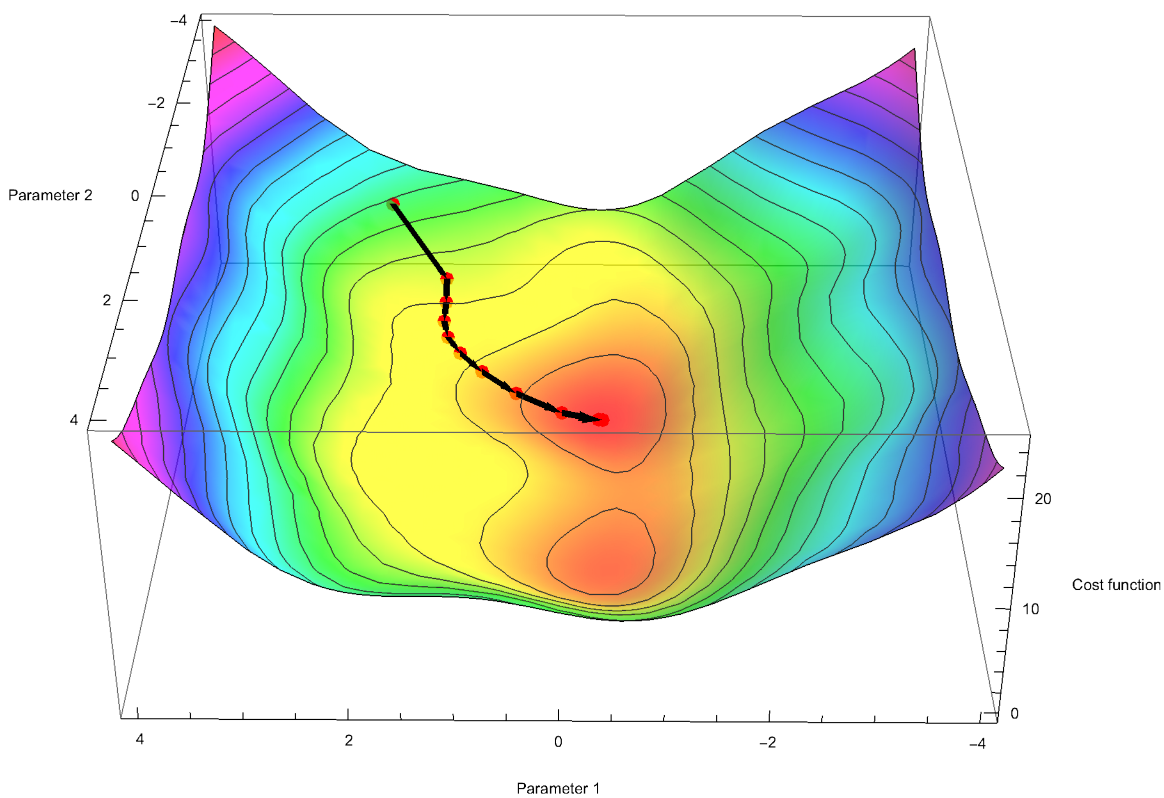
Network Training: Basic Theory

Neural Networks Learn on Training Data

- Neural networks are data-driven algorithms.
- You can think of training data as implicitly specifying a function, where training “tunes” a net to approximate this function.

Basic Theory

- Training a network finds a (local) optimum of the loss as a function of the learned parameters.
- To train one parameter x, the *error* or *loss* ϵ of the network is computed on subsets (batches) of the training data. The chain rule is used to *back-propagate* the error ϵ through the network.
- Updating the parameter $x \to x - r\,\partial x$ is known as *gradient descent* with learning rate r. Here is an example that shows how the error (cost function) is minimized with respect to the two parameters, using gradient descent.



- The parameters of the network gradually converge on a local optimum, i.e. the error is minimized.
- In practice, more complicated update schemes like ADAM are used.

The aim of the training (optimization) is to:

- Evaluate the vector of class scores f, that is a function of the initial weights and inputs, given, the dataset of pairs of input and class,
- Minimize the loss function, L, which comprises of regularization loss and data loss.
- The data loss computes the differences between the scores f and the labels y (given by Loss Layers in the Wolfram Language).
- The regularization loss is only a function of the weights or the parameters

So the question now is:
a) How do we initialize the weights?
b) What method to use for parameter updates?
c) What hyperparameter to use?

NetInitialize

Answers a) How to initialize the parameters?

NetIntiailize[net] gives a net in which all uninitialized learnable parameters in *net* have been given initial values. By default, all methods initialize bias vectors to zero.

- Possible settings for Method include:

"Xavier"	choose weights to preserve variance of random tensors propagated through affine layers
"Orthogonal"	choose weights to be orthogonal matrices
"Random"	choose weights from a given univariate distribution
"Identity"	choose weights so as to preserve components of tensors when propogated through affine layers

◻ Random :
- Initialize the weights of the neurons to small numbers to break symmetry.
- Assures that they are unique
- You can separately initialize the weight matrices and vectors using the suboptions present

W ∝ RandomVariate[NormalDistribution[0,0.1/0.01]]

- For the method "Random", the following suboptions are supported:

"Weights"	NormalDistribution[0, 1]	random distribution to use to initialize weight matrices
"Biases"	None	random distribution to use to initialize vectors

◻ Identity:
- In the above approach outputs from a randomly initialized neuron has a variance that grows with the number of inputs.
- Normalize the variance of each neuron’s output to 1 by scaling its weight vector by the square root of the number of inputs.
- Initialize the net using the "Identity" method, which results in a net that attempts to preserve the components of tensors as they pass through linear layers

W ∝ RandomVariate[UniformDistribution[-1/ $\sqrt{n_{in}}$,1/ $\sqrt{n_{in}}$]]

◻ Xavier

W ∝ RandomVariate[UniformDistribution[-r,r]]

- For the method "Xavier", the following suboptions are supported:

"FactorType"	"Mean"	one of "In", "Out", or "Mean"
"Distribution"	"Normal"	either "Normal" or "Uniform"

http : // machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2010_GlorotB10.pdf (Xavier et.al)

Tanh: RandomVariate[UniformDistribution[-r,r]] ; r = $\sqrt{\frac{6}{n_{in}+n_{out}}}$

Sigmoid: RandomVariate[UniformDistribution[-r,r]] ; r =4 $\sqrt{\frac{6}{n_{in}+n_{out}}}$

https://arxiv.org/pdf/1502.01852v1.pdf (He. et.al)

ReLU: RandomVariate[UniformDistribution[-r,r]] ; r = $\sqrt{\frac{2}{n_{in}}}$

◻ Orthogonal:
- First initialize the weight matrix, then perform SingularValueDecomposition.
- W_{init} = RandomReal[{0,0.1}, {n,m}]; {W_{fin},w,v} = SingularValueDecomposition[W_{init}]; W_{fin} is your required weight

NetTrain: Methods

Answers b) What method to use for parameter update?

Stochastic Gradient Descent:

▫ Vanilla Update (not used in the Wolfram Language, here only for teaching purpose) :
- Update parameters in the negative direction of the gradient.
- Parameters x and the gradient dx, form used is: x += - learning_rate * dx

▫ Momentum Update (not used in the Wolfram Language, here only for teaching purpose) : :
- Better convergence for rates on deep networks.
- The update has the form:

v = μ * v - learning_rate*dx # integrate velocity
x += v # integrate position

Here we see an introduction of a v variable that is initialized at zero, and an additional hyperparameter μ, which is entered in the Wolfram Language (as Momentum)

▪ For the method "SGD", the following additional suboptions are supported:

"Momentum"	0.93	how much to preserve the when updating the derivati

▫ Nesterov Momentum Update: In this method of updating the momentum, we are looking one-step ahead, and as a result we have a little bit better guess for the step (SGD in Wolfram Language)
v_prev = v # back this up
v = μ * v - learning_rate * dx # velocity update stays the same
x += - μ * v_prev + (1 + μ) * v # position update changes form

▫ AdaGrad: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

cache += dx**2
x += - learning_rate * dx / (Sqrt(cache) + ϵ)
The smoothing term ϵ (~1e-4 to 1e-8) avoids division by zero.

▫ RMSProp: Geoff Hinton’s Coursera class http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (Slide 29)
(RMSProp in Wolfram Language)
cache += Beta* cache + (1 - Beta) * dx**2
x += - Beta * dx / (Sqrt(cache) + ϵ)
Here, Beta is a hyperparameter and typical values are [0.9, 0.99, 0.999].

▫ Adam: https://arxiv.org/pdf/1412.6980.pdf
(Adam in Wolfram Language)
m = Beta1*m + (1-Beta1)*dx
v = Beta2*v + (1-Beta2)*(dx**2)
x += - learning_rate * m / (Sqrt(v) + ϵ)
Recommended values in the paper are ϵ = 1e-8, beta1 = 0.9, beta2 = 0.999.

NetTrain: Regularization and Hyperparameter Optimization

Regularization

It refers to a process of introducing additional information in order to solve an ill-posed problem or to prevent over fitting. As a result of regularization, the total loss function gets modified to include the data loss as well as the regularization loss. There are different ways you can obtain regularization for your neural networks:

▫ L2 Regularization:

- For all weighs ω in the network, we add the term $\frac{1}{2}\lambda\,\omega^2$ to the objective, where λ is the regularization strength.
- Heavily penalizes peaky weight vectors and preferring diffuse weight vectors.

▫ Gradient and Weight Clipping

▫ Network Layers specific for regularization (e.g. DropoutLayer)

Hyperparameter Optimization

Neural networks training involve many hyperparameter settings. The most common hyperparameters include:

- the initial learning rate
- learning rate decay schedule (such as the decay constant)
- regularization strength (L2 penalty)

Certain aspects of hyperparameter optimization are:

Hyperparameter ranges: A typical sampling of the learning rate would look as follows: learning_rate = 10^UniformDistribution(-6, 1)
Random Search: http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf
It is better to search randomly than on grid.

In the Wolfram Language, the hyperparameters can be obtained from suboptions for methods:

- Suboptions for specific methods can be specified using Method → {"*method*", opt_1 -
suboptions are supported for all methods:

"LearningRate"	Automatic	the size of steps to take the derivative
"LearningRateSchedule"	Automatic	how to scale the learnir progresses
"L2Regularization"	None	the global loss associat of all learned tensors
"GradientClipping"	None	the magnitude above w should be clipped
"WeightClipping"	None	the magnitude above w be clipped

Loss Layers

Loss Layer Types

The loss layer specifies the functional form of loss to be computed, which measures the compatibility between a prediction (e.g. the class scores in classification) and the actual value of the label. The data loss takes the form of an average over the data losses for every individual example. In the iteration of training the network, this explicit loss function is minimized by tuning the parameters (via back propagation). The implemented loss layers are as follows:

- **MeanAbsoluteLossLayer**: represents a loss layer that computes the mean absolute difference between the `"Input"` port and `"Target"` port (Poisson type used for regression problem):

```
data = <|"Input" → {1, 2, 3}, "Target" → {3, 2, 1}|>; MeanAbsoluteLossLayer[][data]
meanAbsoluteLoss = N[Mean[Flatten[Abs[#Input - #Target]]]] &;
meanAbsoluteLoss[data]
```

Out[•]= 1.33333

Out[•]= 1.33333

- **MeanSquaredLossLayer**: represents a loss layer that computes the mean squared difference between the `"Input"` port and `"Target"` port (Gaussian type used for regression problem):

```
In[•]:= data = <|"Input" → {1, 2, 3}, "Target" → {3, 2, 1}|>;
MeanSquaredLossLayer[][data]
meanSquaredLoss = N[Mean[Flatten[(#Input - #Target) ^ 2]]] &;
meanSquaredLoss[data] // N
```

Out[•]= 2.66667

Out[•]= 2.66667

- **CrossEntropyLossLayer**: represents a net layer that computes the information-theoretic distance by comparing probabilities with specified target values (used for classification problem):

```
In[•]:= ceLoss = -Total[#Target * Log@#Input] &; data = <|"Input" → {0.1, 0.2, 0.7}, "Target" → {0.1, 0.3, 0.6}|>;
ceLoss[data]
CrossEntropyLossLayer["Probabilities"][data]
```

Out[•]= 0.927095

Out[•]= 0.927095

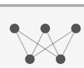
When a loss layer is chosen automatically for a port, the loss layer to use is based on the layer within the net whose output is connected to the port. For **SoftmaxLayer** and **LogisticSigmoid, CrossEntropyLossLayer** is chosen; for non-loss layers, **MeanSquaredLossLayer** is chosen. If a loss layer is specified, it is used unchanged.

Performing Logistic Regression with Real-World Data: Basic Layers

LinearLayer

A **LinearLayer** is essentially an affine transformation $x \to Ax + b$, where A is the “weight matrix” and the vector b is the “bias vector”. This type of layer has a lot of parameters, and hence is used toward the end of the network (after the dimensions of the tensor have been reduced). To understand the **LinearLayer**, it is easiest to construct a net with a single **LinearLayer**, which is solving a simple linear approximation:

```
In[•]:= data = {2, 10, 3};
layer = NetInitialize@LinearLayer[2, "Input" → 3]
layer[data]
```

Out[•]= LinearLayer[ Input: vector (size: 3) Output: vector (size: 2)]

Out[•]= {7.08937, -4.08089}

```
In[•]:= linear[data_, weight_, bias_] := Dot[weight, data] + bias
```

In[•]:=
```
linear[data, NetExtract[layer, "Weights"], NetExtract[layer, "Biases"]]
```

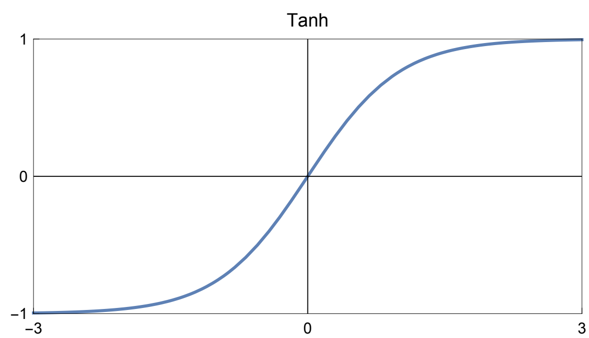
Out[•]=
```
NumericArray[ Type: Real32  Dimensions: {2, 3} ].{2, 10, 3} + NumericArray[ Type: Real32  Dimensions: {2} ]
```

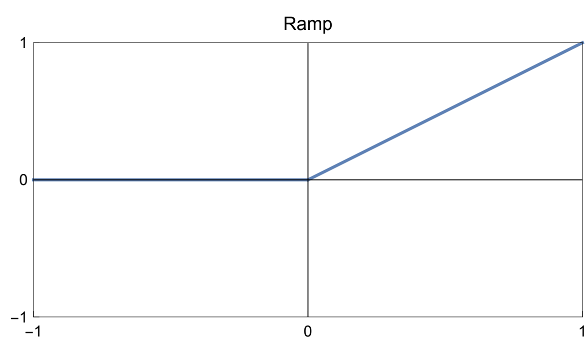
ElementwiseLayer

The **ElementwiseLayer** applies a unary function *f* to every element of the input tensor. The function *f* is referred to as the activation function in the literature; commonly used activation functions are **LogisticSigmoid, Tanh** and **Ramp** (Rectified Linear Unit, ReLU):

In[•]:=
```
Row[Table[
  {min, max} = {-1, 1} * If[f === Ramp, 1, 3];
  Plot[f[x], {x, min, max}, PlotLabel → f, ImageSize → 300, Frame → True, PlotRange → {{min, max}, {-1, 1}}, AspectRatio → 0.5,
   FrameTicks → {{{-1, 0, 1}, {}}, {{min, 0, max}, {}}}], {f, {LogisticSigmoid, Tanh, Ramp}}], Spacer[180]]
```

Out[•]=





In[•]:=
```
elem = ElementwiseLayer[Tanh]; data = RandomReal[{0, 10}, 10];
elem[data]
```

Out[•]=
```
{0.999993, 0.969795, 0.999623, 1., 0.999935, 0.998672, 1., 0.999306, 0.9942, 0.999992}
```

As expected, it applies **Tanh** to the data:

```
In[ ]:= Tanh[data]

Out[ ]= {0.999993, 0.969795, 0.999623, 1., 0.999935, 0.998672, 1., 0.999306, 0.9942, 0.999992}
```

SoftmaxLayer

The softmax classifier takes an input (a vector) and returns normalized class probabilities. The element x_i in a vector is converted to $e^{x_i} / \sum_j e^{x_j}$, and generally the innermost dimension is used as the normalization dimension.

Apply the **SoftmaxLayer** on a list of random data to get probabilistic interpretations. On a list:

```
In[ ]:= data = RandomReal[{0, 10}, 10]
        SoftmaxLayer[]@data
        Total@SoftmaxLayer[]@data

Out[ ]= {0.721154, 1.09536, 5.24204, 1.68959, 2.11553, 8.23392, 7.36807, 4.38206, 7.98477, 7.07688}

Out[ ]= {0.000210757, 0.000306407, 0.0193722, 0.000555099, 0.00084987, 0.385954, 0.162368, 0.00819771, 0.300835, 0.12135}

Out[ ]= 1.
```

See what **SoftmaxLayer** is actually evaluating by using the functional approach:

```
In[ ]:= fsoftmax[x_] := N@Exp[x] / Total[Exp[x], {-1}];
        fsoftmax[data]
        Total@fsoftmax[data]

Out[ ]= {0.000210757, 0.000306407, 0.0193722, 0.000555099, 0.00084987, 0.385954, 0.162368, 0.00819771, 0.300835, 0.12135}

Out[ ]= 1.
```

Performing Logistic Regression with Real-World Data: Constructing the Network

This example uses the *Titanic* dataset to perform logistic regression with both categorical and numerical input data. The dataset contains information for the passengers traveling on the *Titanic*: the class of travel, their age, their gender and if they survived or not.

Get the data from the Wolfram server, delete incomplete entries, and split the data into a training and a test dataset:

```
In[•]:= titanicdata = ExampleData[{"Dataset", "Titanic"}];
        titanicdata = DeleteMissing[titanicdata, 1, 2];
        {trainingData, testData} = TakeDrop[RandomSample@titanicdata, 800]
```

Out[•]= {

class	age	sex	survived
1st	58	male	False
3rd	25	male	False
2nd	26	male	True
2nd	36	male	False
2nd	47	male	False
1st	28	female	True
2nd	19	male	False
2nd	55	female	True
2nd	3	male	True
3rd	28	male	False
3rd	19	male	False
2nd	29	female	True
3rd	31	male	False
1st	61	male	False
3rd	3	female	False
1st	27	male	True
3rd	21	female	False
1st	42	male	False
3rd	20	male	False
3rd	9	female	False

rows 1–20 of 800

,

class	age	sex	survived
3rd	35	male	False
1st	21	female	True
3rd	21	male	False
3rd	41	male	False
3rd	19	male	False
1st	31	female	True
3rd	37	male	False
3rd	21	male	False
3rd	19	male	False
1st	54	female	True
2nd	39	male	False
3rd	40	male	False
3rd	32	male	False
2nd	33	male	False
2nd	29	male	False
2nd	40	female	True
2nd	8	male	True
2nd	20	female	True
1st	37	male	False
3rd	25	male	True

rows 1–20 of 246

}

In the next step, encoders for each of the features are created: class (with 3 values), sex (with 2 values) and survived can use the built-in Boolean encoder:

```
In[•]:= classEncoder = NetEncoder[{"Class", {"1st", "2nd", "3rd"}, "UnitVector"}]
        genderEncoder = NetEncoder[{"Class", {"male", "female"}, "UnitVector"}]
```

Out[•]= NetEncoder[Type: Class Output: vector (size: 3)]

Out[•]= NetEncoder[Type: Class Output: vector (size: 2)]

Combine all the input from the different class encoders, label the input ports, and connect to the first layer:

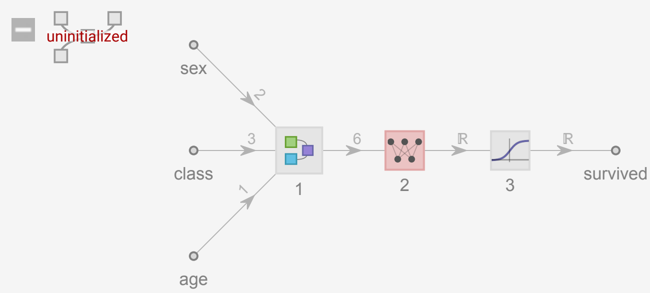
In[∘]:=
```
net1 = NetGraph[{CatenateLayer[]}, {{NetPort["class"], NetPort["age"], NetPort["sex"]} → 1},
  "class" → classEncoder, "age" → "Scalar", "sex" → genderEncoder]
```

Out[∘]= NetGraph[Number of inputs: 3 Output port: vector (size: 6)]

In the second step, the layers that perform the logistic regression are added. Take the already existing **NetGraph** that was built for inputting the layers and add the necessary layers for logistic regression. Connect the **LogisticSigmoid** to the appropriate decoder so that it outputs a Boolean to indicate if the passenger survived or not:

In[∘]:=
```
net2 = NetGraph[{net1, LinearLayer[], LogisticSigmoid}, {1 → 2 → 3 -> NetPort["survived"]}, "survived" → "Boolean"]
```

Out[∘]= NetGraph[



Inputs		**Outputs**	
age:	scalar	survived:	boolean
class:	class		
sex:	class		

]

Performing Logistic Regression with Real-World Data: Training, Predicting and Accuracy

Train the network using **NetTrain**. **MaxTrainingRounds** is specified as an option to specify how many times the data is revisited. As the training progresses, you can see the training progress report that shows the update and displays the error function:

In[∘]:=
```
trainingData⟦1⟧
```

Out[∘]=

class	1st
age	58
sex	male
survived	False

In[∘]:=
```
trained = NetTrain[net2, trainingData, MaxTrainingRounds → 1000]
```

Out[∘]= NetGraph[Number of inputs: 3 survived port: boolean]

In[∘]:=
```
Export[FileNameJoin[{NotebookDirectory[], "trained.wlnet"}], trained]
```

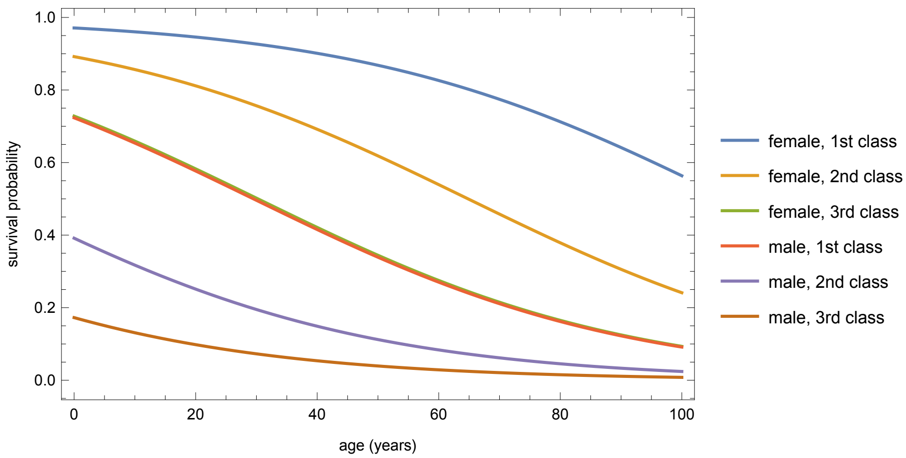
Out[∘]=
```
/Users/robertjfrey/Documents/Work/Stony Brook
  University/AMS/QF/public_html/Instruction/Spring2020/AMS512/Class09/trained.wlnet
```

Once the network is trained, you can enter the input features for fictitious passengers and find the probability for their survival:

```
In[•]:= {trained[<|"class" → "1st", "age" → 20, "sex" → "female"|>], trained[<|"class" → "1st", "age" → 20, "sex" → "female"|>, None]}
        {trained[<|"class" → "3rd", "age" → 30, "sex" → "male"|>], trained[<|"class" → "3rd", "age" → 30, "sex" → "male"|>, None]}
Out[•]= {True, 0.945925}
Out[•]= {False, 0.0728203}
```

You can take a further step and plot the probability of survival with age for all the variations in class and gender:

```
In[•]:= p[class_, age_, sex_] := trained[<|"class" -> class, "age" -> age, "sex" -> sex|>, None];
        Plot[{p["1st", x, "female"], p["2nd", x, "female"], p["3rd", x, "female"],
          p["1st", x, "male"], p["2nd", x, "male"], p["3rd", x, "male"]}, {x, 0, 100},
         PlotLegends → {"female, 1st class", "female, 2nd class", "female, 3rd class", "male, 1st class",
           "male, 2nd class", "male, 3rd class"}, Frame → True, FrameLabel → {"age (years)", "survival probability"}]
Out[•]=
```



Assess the accuracy of the network by testing it against the test dataset created. Compare the accuracy with the built-in **Classify** function:

```
In[•]:= cm = ClassifierMeasurements[trained, testData → "survived", "Accuracy"]
Out[•]= 0.784553
```

```
In[•]:= cf = Classify[trainingData → "survived"];
        ClassifierMeasurements[cf, testData → "survived", "Accuracy"]
Out[•]= 0.776423
```

LeNet explained

LeNet is a simple convolution network that performs feature extraction that can be used to classify an image:

```
NetChain[
{

(*STEP 1: FETAURE EXTRACTION*)

(*FIRST CONVOLUTION BLOCK*)
ConvolutionLayer[20,3],     (*first convolution => 20 feature images*)
ElementwiseLayer[Ramp],     (*activation function (ReLU) => non-linearity, sparsity*)
(*FIRST POOLING BLOCK*)
PoolingLayer[2,2],          (*max pooling => downsampling*)
 (*SECOND CONVOLUTION BLOCK*)
ConvolutionLayer[50,3],     (*second convolution => 50 feature images*)
ElementwiseLayer[Ramp],     (*activation function (ReLU) => non-linearity, sparsity*)
(*SECOND POOLING BLOCK*)
PoolingLayer[2,2],          (*max pooling => downsampling*)

FlattenLayer[],             (*flattening => images to vector*)

(*STEP 2: COMPUTING CLASS PROBABILITY*)

(*FULLY-CONNECTED BLOCK 1*)
LinearLayer[500],          (*first fully connected layer => feature vector from image features*)
ElementwiseLayer[Ramp],    (*activation function (ReLU) => non-linearity, sparsity*)

(*FULLY-CONNECTED BLOCK 2*)
LinearLayer[10],           (*second fully connected layer => class prediction*)
(*PROBABILITY COMPUTATIONS*)
SoftmaxLayer[]              (*normalization*)
},

"Input"  → NetEncoder[{"Image", {32, 32}}],  (*encoder => image to tensor*)
"Output" → NetDecoder[{"Class", classes}]  (*decoder => tensor to class*)
];
```

LeNet: Network Training (with Options and Properties)

Network Data

In this example, LeNet is trained on the handwritten digits from the MNIST database. In the Wolfram Language database, the digits are classified into training and test datasets. **ResourceData** can be used to access the data:

In[∘]:=
```
trainingData = ResourceData[ResourceObject["MNIST"], "TrainingData"];
testData = ResourceData[ResourceObject["MNIST"], "TestData"];
```

In[∘]:=
```
trainingData[[1]]
```

Out[∘]=  → 0

Constructing the Network

In[∘]:=
```
lenet = NetChain[{
    ConvolutionLayer[20, 5], Ramp, PoolingLayer[2, 2],
    ConvolutionLayer[50, 5], Ramp, PoolingLayer[2, 2],
    FlattenLayer[], 500, Ramp, 10, SoftmaxLayer[]},
   "Output" → NetDecoder[{"Class", Range[0, 9]}],
   "Input" → NetEncoder[{"Image", {28, 28}, "Grayscale"}]
  ]
```

Out[∘]= NetChain[uninitialized | Input port: image | Output port: class]

Training the Network, with Various Options

Train the network now on the training data (with different options to make the process faster):

If a simple **NetTrain** is performed, you will see that the training takes approximately 40 minutes:

In[∘]:=
```
trained = NetTrain[lenet, trainingData]
```

Out[∘]= NetChain[Input port: image | Output port: class]

Try to improve the speed by changing the number of **MaxTrainingRounds** to 2 (reduces training time by a factor of 5) and changing the **BatchSize,** which in turn improves the number of inputs per second:

In[∘]:=
```
trained = NetTrain[lenet, trainingData, MaxTrainingRounds → 2, BatchSize → 512 ]
```

Training Progress

```
ressIndicator[NeuralNetworks`Private`NetTrain`$progressFrac
 progress | Round[100 NeuralNetworks`Private`NetTrain`$
    total | NeuralNetworks`Private`NetTrain`$absoluteBat
     time | GeneralUtilities`TimeString[Round[NeuralNetw
   method | ADAM optimizer, batch size 512, CPU
          | current     round     If[NeuralNetw
     loss | Lookup[NeuraLookup[NeuralIf[NeuralNetw
    error | Which[LookupWhich[LookupIf[NeuralNetw
loss
NeuralNetworks`Private`NetTrain`$lastLossPlotBoxes
error rate
NeuralNetworks`Private`NetTrain`$lastMetricPlotBoxes
Stop
```

Finally, if you have a GPU, the training goes much faster; in this case the training completes in 20s:

```
trained = NetTrain[lenet, trainingData, MaxTrainingRounds → 2, BatchSize → 512, TargetDevice → "GPU" ]
```

Obtaining Different Properties

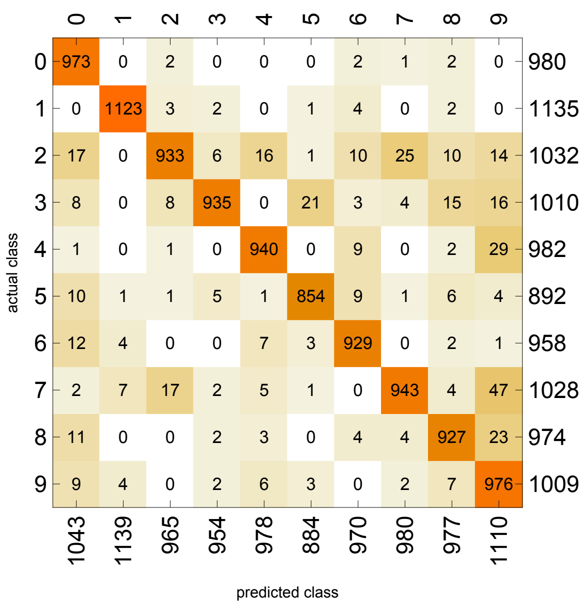
In[•]:=
```
cm = ClassifierMeasurements[trained, testData]
```

Out[•]=
```
ClassifierMeasurementsObject[ Classifier: Net
                              Number of test examples: 10000 ]
```

Data not in notebook; Store now »

In[•]:=
```
cm["ConfusionMatrixPlot"]
```

Out[•]=



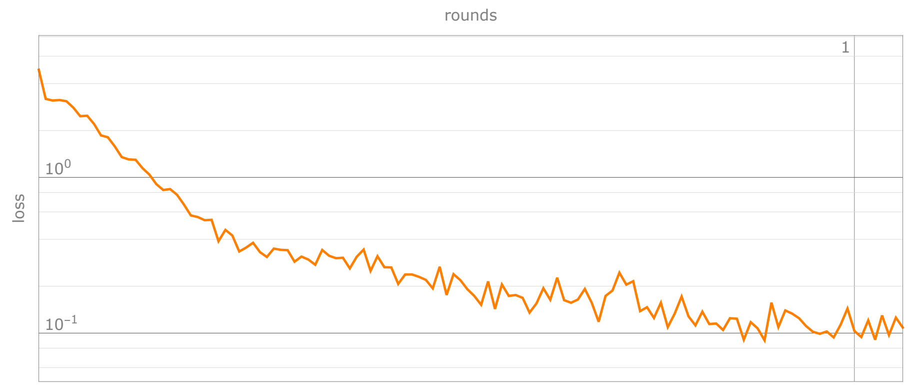
In[•]:=
```
cm["Accuracy"]
```

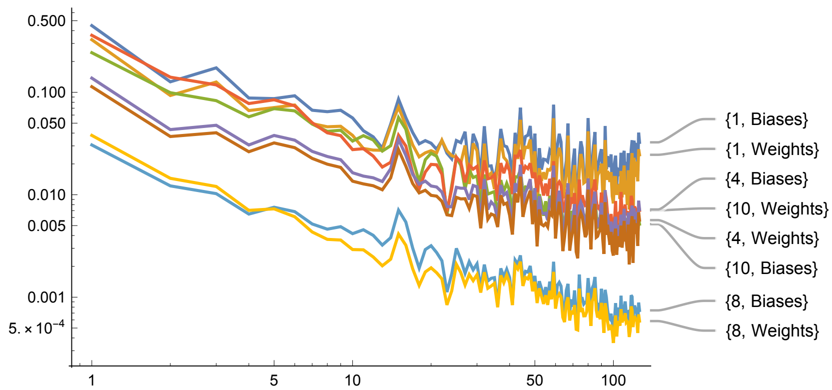
Out[•]=
```
0.9533
```

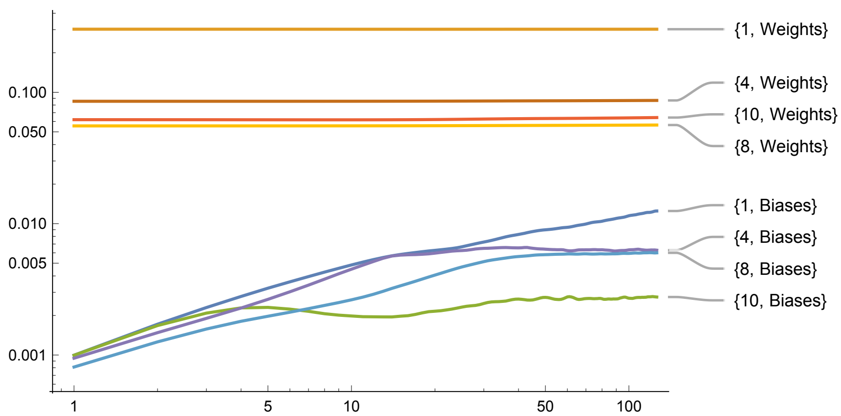
In[•]:=
```
result = NetTrain[lenet, trainingData,
   {"LossEvolutionPlot", "RMSGradientsEvolutionPlot", "RMSWeightsEvolutionPlot"}, MaxTrainingRounds → 2, BatchSize → 512];
```

In[]:=
```
Column@result
```

Out[]=







Exporting a Trained Net

```
Export["trainedLenet.wlnet", trainedLenet]
```

Out[]= testnet.wlnet

```
trained = Import["trainedLenet.wlnet"]
```

Out[]= NetChain[Input port: image | Output port: class | Number of layers: 11]

In[•]:=
```
cm = ClassifierMeasurements[trained, testData]
```

Out[•]=
```
ClassifierMeasurementsObject[ Classifier: Net
                              Number of test examples: 10000 ]
Data not in notebook; Store now »
```

In[•]:=
```
cm["Accuracy"]
```

Out[•]=
```
0.9533
```

Glossary

BatchSize	CatenateLayer	CellAnnotation	Class	ClassifierMeasurements
Classify	ColorSpace	ConfusionMatrixPlot	ConvolutionLayer	DeleteMissing
Dot	ElementwiseLayer	ExampleData	Exp	Flatten
FlattenLayer	Frame	FrameLabel	Image	ImageDimensions
Input	LinearLayer	LogisticSigmoid	LossEvolutionPlot	MaxTrainingRounds
Mean	MeanAbsoluteLossLayer	MeanSquaredLossLayer	N	NetChain
NetDecoder	NetEncoder	NetExtract	NetInitialize	NetPort
Output	Plot	PlotLegends	PoolingLayer	Ramp
RandomReal	RandomSample	ResourceData	ResourceObject	RMSGradientEvolutionPlot
RMSWeightEvolutionPlot	SoftmaxLayer	TakeDrop	Tanh	Target
TargetDevice	Total	TotalLayer		