Tests and Examples of Machine Learning in Wolfram Coding Architecture

Numerical tests of Linear Layer Networks

Lets train a Linearly layered network to perform multiplication by 2. We'll train one 'trainedRough' by doing approximations of multiplication of 2 and another by doing exact multiplication. Then we'll apply them to untrained sets of data and see how each does.

```
trainedExact = NetTrain[LinearLayer[], {1 → 2.0, 2 → 4.0, 3 → 6.0, 4 → 8.0}]
```

```
Starting training.

Optimization Method: SGD

Batch Size: 4

  %   round   batch      round      batch    learning    inputs      time       time

     /10000    /1         loss       loss        rate    /second   elapsed      left

 53    5281     1       0.0000     0.0000    6.83*^-3     12747        2s         2s

Training complete.
```

```
Out[∘]= LinearLayer[ Input:  real
                     Output: real ]
```

```
trainedRough = NetTrain[LinearLayer[], {1 → 1.9, 2 → 4.1, 3 → 6.0, 4 → 8.1}]
```

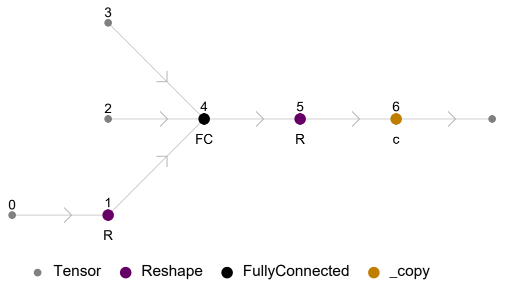
```
Starting training.

Optimization Method: SGD

Batch Size: 4

  %   round   batch      round      batch    learning    inputs      time       time

     /10000    /1         loss       loss        rate    /second   elapsed      left

 52    5190     1     3.75*^-3   3.75*^-3    6.90*^-3     12491        2s         2s

Training complete.
```

```
Out[∘]= LinearLayer[ Input:  real
                     Output: real ]
```

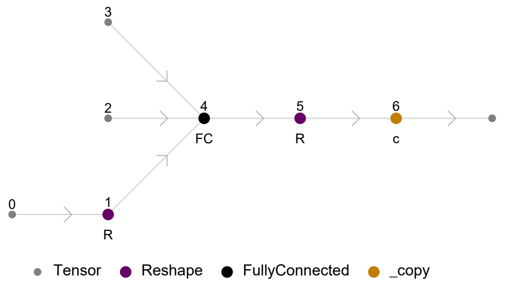
We can now plot these nets:

```
NetInformation[trainedExact, "MXNetNodeGraphPlot"]
NetInformation[trainedRough, "MXNetNodeGraphPlot"]
```

Out[∘]=



Out[∘]=



```
trainedExact[2]
```

Out[∘]=
```
4.
```

```
UntrainedDataInput = {3, 5, 2, 7, 1, 8, 100, 13, π, 2.4, 0.5};
Print["Input Data (each number run through the network individually): ",
 UntrainedDataInput]
Print["trainedExact gives ", trainedExact[UntrainedDataInput]]
Print["trainedExact Error is: ",
 (2 * UntrainedDataInput) - trainedExact[UntrainedDataInput]]
Print["trainedRough gives ", trainedRough[UntrainedDataInput]]
Print["trainedRough Error is: ",
 (2 * UntrainedDataInput) - trainedRough[UntrainedDataInput]]
Print[Style[
StringJoin["i.e. trainedExact runs ", ToString[UntrainedDataInput[[1]]],
   " through its network and gets ", ToString[trainedExact[UntrainedDataInput][[1]]],
   " while trainedRough runs ", ToString[UntrainedDataInput[[1]]],
   " through its network and gets ", ToString[trainedRough[UntrainedDataInput][[1]]] ]
,
  Bold,
  16] ]
```

```
Input Data (each number run through the network individually):
 {3, 5, 2, 7, 1, 8, 100, 13, π, 2.4, 0.5}

trainedExact gives {6., 10., 4., 14., 2., 16., 200., 26., 6.28319, 4.8, 1.}

trainedExact Error is: {0., 0., 0., 0., 0., 0., 0., 0., -1.74846×10^-7, -1.90735×10^-7, -1.19209×10^-7}

trainedRough gives {6.05, 10.15, 4., 14.25, 1.95, 16.3, 204.9, 26.55, 6.34027, 4.82, 0.925}

trainedRough Error is: {-0.0500002, -0.150001, 0., -0.250001,
  0.0500003, -0.300001, -4.90001, -0.550001, -0.05708, -0.0200002, 0.0750004}
```

i.e. trainedExact runs 3 through its network and gets 6.
while trainedRough runs 3 through its network and gets 6.05

Minor errors come in in the last terms, all of which have decimal parts, of the exactly trained network. Machine precision on mathematica typically around 10^-8, about the order of the error we see here.

We have more significant errors in the roughly trained network. Can see that the error grows with the size of the input, averaging roughly 1.5 -2.5% of the input. We have exactitude when the input is exactly 2. We show below (see plot 4 and its description) that this exactitude is an artefact of the training used.

Descriptions of plots:
Plot 1 - shows absolute error increases linearly
Plot 2 - shows relative error crosses exactitude point, perhaps logarithmic
Plot 3 - Extend scale of plot to up to x=100, appears to show relative error approaching an asymptote

Below plot 3 we calculate the relative error for inputs of 10^6 and 10^10, obtaining 0.0250001 both time. This indicates that the relative error likely approaches an asymptote of roughly 2.5%

Immediately below we run some statistical calculations on the ratios of the input to output training values. We may note that although the mean of the ratios is about 0.6 % off, whereas we have the aforementioned 2.5 % error asymptote. Also we can note the deviation measures are roughly the order of the error, with the first Median deviation being particularly close.

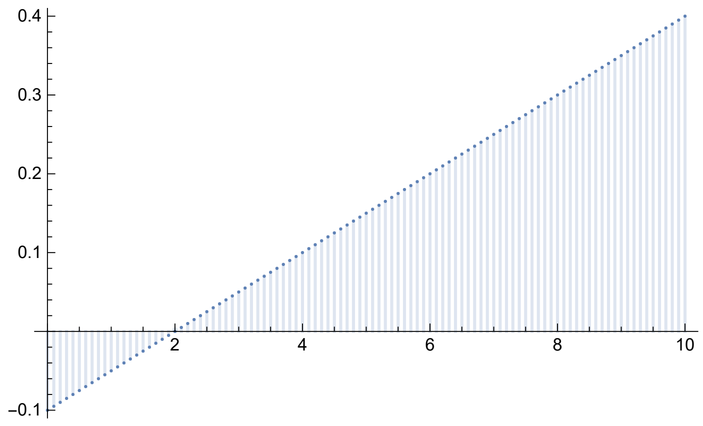
```
RoughRatios = {1.9/1, 4.1/2, 6.0/3, 8.1/4};
Mean[RoughRatios]
MeanDeviation[RoughRatios]
MeanDeviation[(RoughRatios^(-1))]
StandardDeviation[(RoughRatios)]
StandardDeviation[(RoughRatios^(-1))]
MedianDeviation[(RoughRatios)]
MedianDeviation[(RoughRatios^(-1))]
```

```
Out[•]= 1.99375

Out[•]= 0.046875

Out[•]= 0.0121644

Out[•]= 0.0657489

Out[•]= 0.0169662

Out[•]= 0.025

Out[•]= 0.00609756
```

```
Style["Plot 1, absolute error:", Bold]
DiscretePlot[trainedRough[x] - (2 * x), {x, 0, 10, 0.1}]
Style["Plot 2, error as a portion of expected value:", Bold]
DiscretePlot[(trainedRough[x] - (2 * x)) / (2 * x), {x, 0, 10, 0.1}]
Style["Plot 3, error as a portion of expected value, larger scale:", Bold]
DiscretePlot[(trainedRough[x] - (2 * x)) / (2 * x), {x, 0, 100, 0.1}, AxesOrigin → {0, 0}]
```

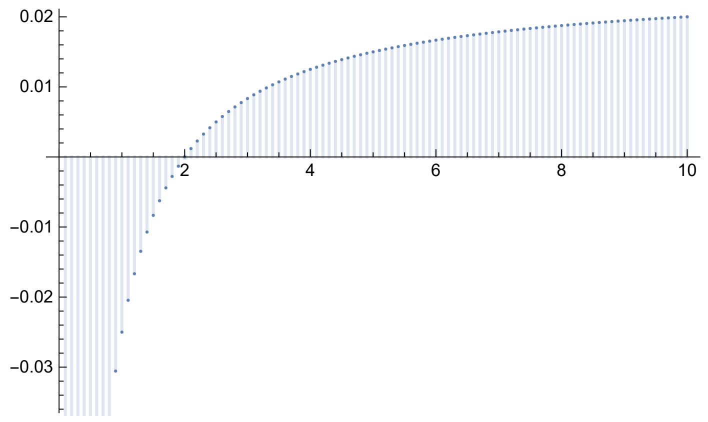
Out[•]= **Plot 1, absolute error:**

Out[•]=



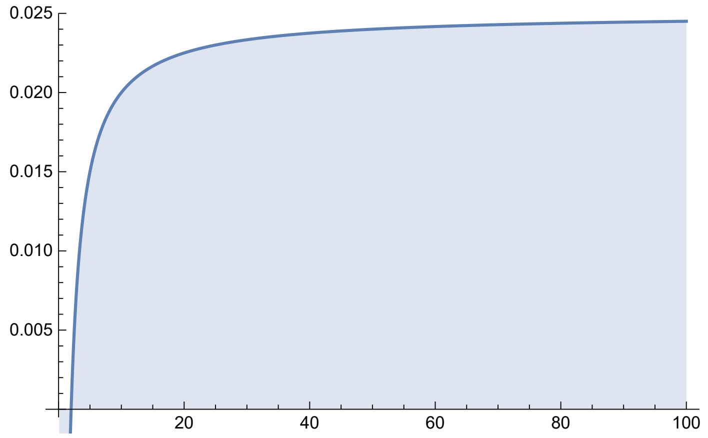
Out[•]= **Plot 2, error as a portion of expected value:**

```
Power: Infinite expression 1/0. encountered.
```

Out[ ]=


```
Out[ ]= Plot 3, error as a portion of expected value, larger scale:
```

Power: Infinite expression $\frac{1}{0.}$ encountered.

Out[ ]=


```
(trainedRough[(10^1)] - (2 * (10^1))) / (2 * (10^1))
Out[ ]= 0.0200001

(trainedRough[(10^6)] - (2 * (10^6))) / (2 * (10^6))
(trainedRough[(10^10)] - (2 * (10^10))) / (2 * (10^10))
Out[ ]= 0.0250001

Out[ ]= 0.0250001
```

Below we change the training solely by making 2 -> 4.0 rather than 2 -> 4.1, as we used to train the original 'rough' network. This defines a trained network 'trainedRough2'. We can see that the point of exactitude is shifted, so the fact that trainedRough gives 4 with an input of 2 seems to be just an artefact of the training.

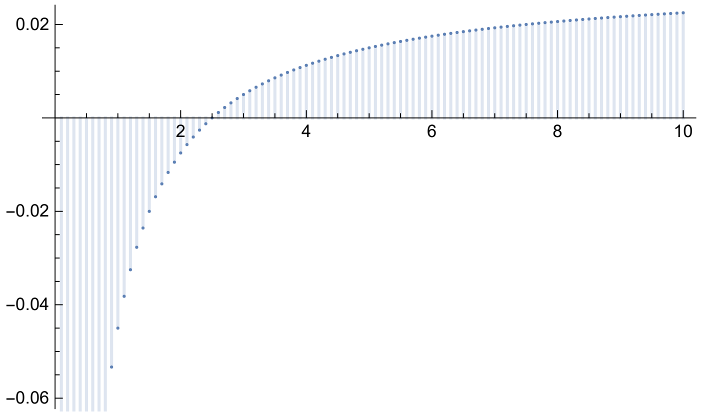
```
trainedRough2 = NetTrain[LinearLayer[], {1 → 1.9, 2 → 4.0, 3 → 6.0, 4 → 8.1}]
Style["Plot 4, new network trainedRough2 error as a portion of expected value:", Bold]
DiscretePlot[(trainedRough2[x] - (2 * x)) / (2 * x), {x, 0, 10, 0.1}]
```

```
Starting training.

Optimization Method: SGD

Batch Size: 4

  %    round  batch      round      batch    learning     inputs       time        time

      /10000     /1       loss       loss        rate    /second    elapsed        left

 53     5344      1   5.00*^-4   5.00*^-4   6.79*^-3      12862         2s          2s

Training complete.
```

Out[∘]= LinearLayer[Input: real Output: real]

Out[∘]= **Plot 4, new network trainedRough2 error as a portion of expected value:**

Power: Infinite expression $\frac{1}{0.}$ encountered.

Out[∘]=


Tests of Image Recognition

We'll now look at the image recognition of handwritten numbers sourced from one of Mathematica's pre-trained neural nets. We retrieve these nets using the function Netmodel["Name of data set"]:

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

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

Source of training data:

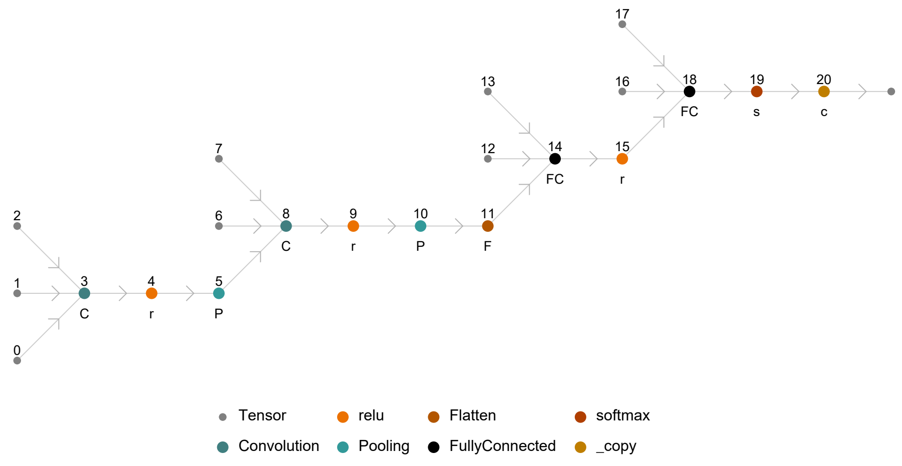
```
NetModel["LeNet Trained on MNIST Data", "SourceMetadata"]
```

Out[∘]= ⟨|Citation →
 Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-Based Learning Applied to Document Recognition," Proceedings of the IEEE, 86(11), 2278-2324 (1998),
 Source → http://yann.lecun.com/exdb/lenet, Date → Year: 1998 |⟩

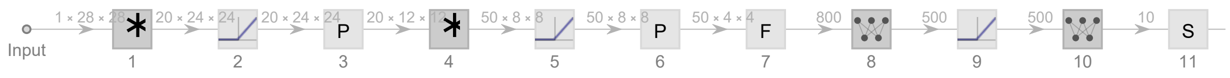
We can now plot the net and a 'summary graphic' of its underlying structure of its:

```
NetInformation[net, "MXNetNodeGraphPlot"]
NetInformation[net, "SummaryGraphic"]
```

Out[ ]=



Out[ ]=



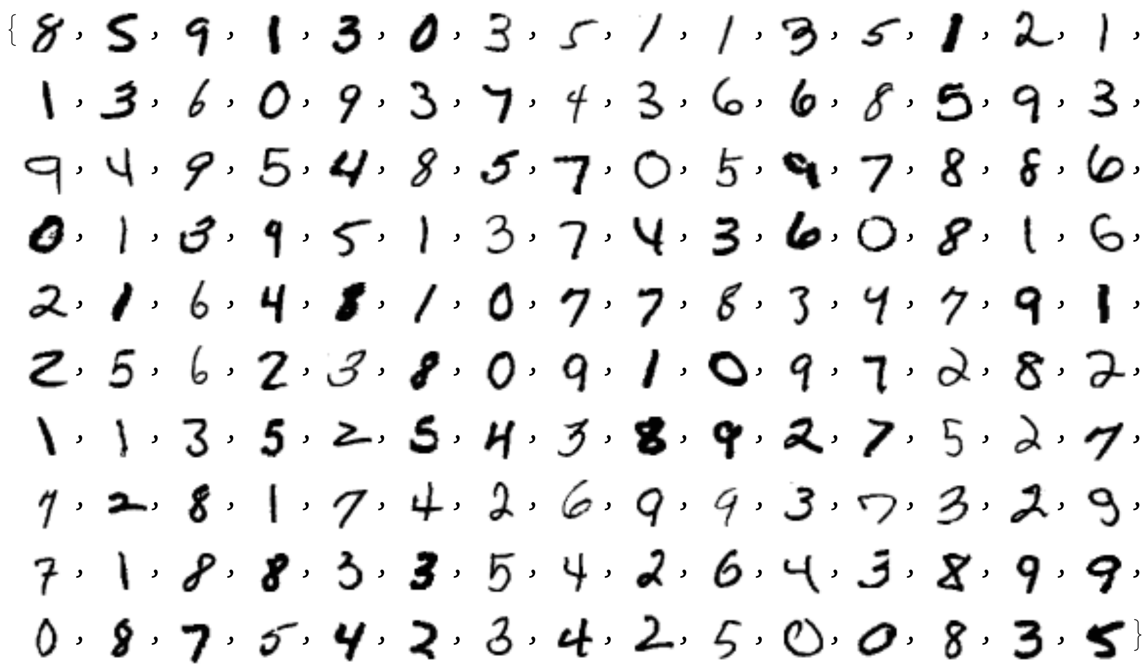
According to the documentation this network has 98.5 % accuracy on the MNIST data set.
(https://resources.wolframcloud.com/NeuralNetRepository/resources/LeNet-Trained-on-MNIST-Data)

This set has exactly 60,000 samples of numbers. Some are relatively poor quality (smudged, mis-shapen, faded, etc.). Additionally some of the numbers are stylized in different ways, such as a 4 or 7 with a horizontal dash, or 2's with curls or corners at the bottom. Looking through it, it can be hard for a person to distinguish some of the numbers, they are by no means perfect samples. The fact that 98.5% accuracy is retained seems better under these circumstances.

Lets look at a random sample of the data from this set:

```
sample = Keys[RandomSample[ResourceData["MNIST"], 150]]
```

Out[•]=



Now lets have the net tell us what it thinks some of the inputs are by having it give a list of probabilities for each number and then having it give the number with the maximal probability, implemented as shown

```
sample[[1]]
net[sample[[1]], "Probabilities"]
net[sample[[1]]]

sample[[2]]
net[sample[[2]], "Probabilities"]
net[sample[[2]]]

sample[[3]]
net[sample[[3]], "Probabilities"]
net[sample[[3]]]
```

Out[∘]=

Out[∘]= $\langle| 0 \to 2.12545 \times 10^{-14}, 1 \to 2.67648 \times 10^{-18}, 2 \to 9.53796 \times 10^{-11}, 3 \to 1.7027 \times 10^{-13}, 4 \to 6.31805 \times 10^{-14}, 5 \to 1.7638 \times 10^{-11}, 6 \to 2.95915 \times 10^{-17}, 7 \to 5.85696 \times 10^{-18}, 8 \to 1., 9 \to 1.75954 \times 10^{-12} |\rangle$

Out[∘]= 8

Out[∘]=

Out[∘]= $\langle| 0 \to 2.02122 \times 10^{-14}, 1 \to 1.90318 \times 10^{-22}, 2 \to 2.22699 \times 10^{-14}, 3 \to 1.20302 \times 10^{-12}, 4 \to 2.58977 \times 10^{-21}, 5 \to 1., 6 \to 1.06772 \times 10^{-14}, 7 \to 1.95586 \times 10^{-19}, 8 \to 1.49662 \times 10^{-12}, 9 \to 3.80188 \times 10^{-9} |\rangle$

Out[∘]= 5

Out[∘]=

Out[∘]= $\langle| 0 \to 8.56797 \times 10^{-17}, 1 \to 9.65988 \times 10^{-16}, 2 \to 6.46527 \times 10^{-14}, 3 \to 3.8998 \times 10^{-9}, 4 \to 1.01521 \times 10^{-6}, 5 \to 7.33252 \times 10^{-10}, 6 \to 2.77507 \times 10^{-23}, 7 \to 1.36636 \times 10^{-11}, 8 \to 1.67658 \times 10^{-13}, 9 \to 0.999999 |\rangle$

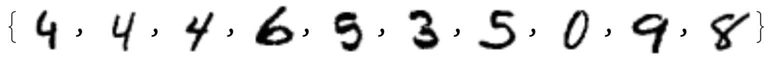
Out[∘]= 9

Yielding the supposesedly correct answer in all cases.
(Note that, since 'sample' is randomly generated, your results may differ if you run this yourself)

I say 'supposedly' since it could be the case that any of the above numbers could just be a very messed-up writing of a different intended number. We can see above that for the nine the network basically thinks there's a roughly one in a million chance the writer could have messed up a 4 to look like this.

But these are ones it was trained against, lets now test it against the test data included in MNIST. There are 10,000 such test values, all of which were not included in the training data.
Define a set of test data:

```
testsample = Keys[RandomSample[ResourceData[ResourceObject["MNIST"], "TestData"], 10]]
```

Out[∘]= 

We'll now run explicit tests of the first element and have it

```
testsample[[1]]
net[testsample[[1]], "Probabilities"]
net[testsample[[1]]]

net[testsample]
```

Out[∘]=

```
Out[∘]= ⟨|0 → 4.7888 × 10^-13, 1 → 8.46037 × 10^-10, 2 → 4.5134 × 10^-11, 3 → 1.68332 × 10^-10, 4 → 0.769862,
         5 → 3.0078 × 10^-9, 6 → 2.38514 × 10^-14, 7 → 1.54175 × 10^-10, 8 → 2.06709 × 10^-11, 9 → 0.230138|⟩

Out[∘]= 4

Out[∘]= {4, 4, 4, 6, 5, 3, 5, 0, 9, 8}
```

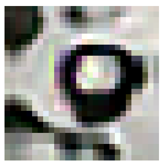
Can see that they all seem to match up correctly, even though the 'testsample' images are by no means ideal. Look at the first five, it is very close to being a 9. Indeed, this is reflected in the probability output (immediately below), with the net giving the first five a 10% chance of being a 9.
Can start to see the power of even these relatively simple nets. Not only would we have a hard time discerning some of these numbers, but we would not be able to quantify the probability of this 5 being a 9, removing some bias. If, for instance, a person saw that 5 at the end of a price tag, they might assume right away that its a 9, due to the commonality of prices ending in 9's.

```
net[testsample[[5]], "Probabilities"]
```

```
Out[∘]= ⟨|0 → 3.8421 × 10^-10, 1 → 4.24606 × 10^-10, 2 → 1.25446 × 10^-9, 3 → 0.00715046, 4 → 2.02839 × 10^-8,
         5 → 0.891099, 6 → 1.97093 × 10^-11, 7 → 3.27355 × 10^-8, 8 → 1.81144 × 10^-6, 9 → 0.101748|⟩
```

Now lets try to mess with the network a bit by inserting less-sensical images. We'll use Mathematica's ImageRestyle function to create a new image that's a mixture of a picture of a 4 and a zero and test it.

```
fourandzeromixed = ImageRestyle[testsample[[1]], testsample[[9]]]
```

Out[∘]= 

```
net[fourandzeromixed, "Probabilities"]
```

```
Out[∘]= ⟨|0 → 0.044661, 1 → 0.000465248, 2 → 0.00541036, 3 → 0.00630587, 4 → 0.0041097,
         5 → 0.217542, 6 → 0.00211606, 7 → 0.000180982, 8 → 0.338815, 9 → 0.380394|⟩
```

So we can see that the network is pretty uncertain of what the image is, but assigns the highest probability of 38% to it being a 9, with 8 close behind. This makes a bit of sense, it has a single circle as a defining feature while also having some extraneous bit (these parts are why it only assigns a value of 4% to

0).

```
CloudExport[ EvaluationNotebook[],
 CloudObject[https://www.wolframcloud.com/objects/8beb227a-a079-49c2-8a5e-1618b6b9daf5]]
```

```
Out[ ]= CloudObject[https://www.wolframcloud.com/objects/7b4eee08-a0a5-4995-a0bd-b596a7759634]
```

```
In[ ]:= SystemOpen[CloudObject[
    "https://www.wolframcloud.com/objects/8beb227a-a079-49c2-8a5e-1618b6b9daf5"][[1]]]
```