Neural Network Exercise

Kim Mingi (xp2425@khu.ac.kr)

Reference

A. Wolfram Tutorial "Introduction to Neural Nets" https://reference.wolfram.com/language/tutorial/NeuralNetworksIntroduction.html#1969481202

- MNIST handwriting examples

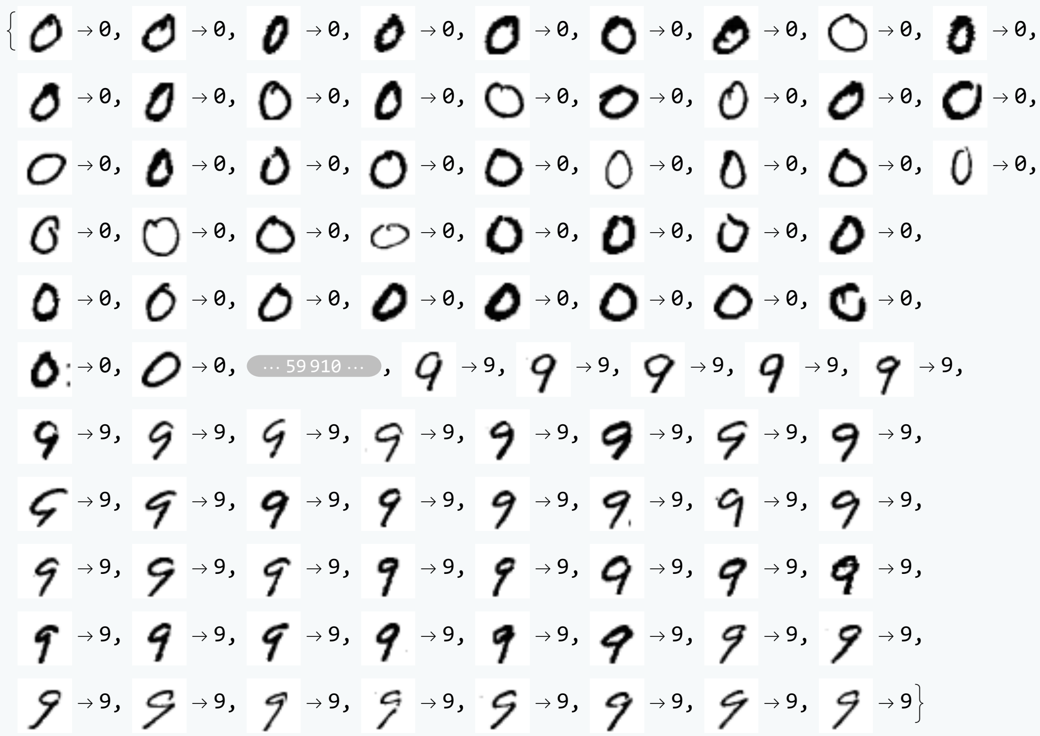
In[1]:=
```
ResourceSearch["MNIST"]
```

Out[1]=
```
{ResourceObject[ Name: FashionMNIST »
                 Type: DataResource
                 Description: A small MNIST–like fashion product image dataset ],
 ResourceObject[ Name: MNIST »
                 Type: DataResource
                 Description: Database of handwritten digits commonly used for training image proces
 ResourceObject[ Name: LeNet Trained on MNIST Data »
                 Type: NeuralNet
                 Description: Identify the handwritten digit in an image ],
 ResourceObject[ Name: CapsNet Trained on MNIST Data »
                 Type: NeuralNet
                 Description: Identify the handwritten digit in an image ]}
```

In[2]:=

```
trainingData = ResourceData["MNIST", "TrainingData"]
```

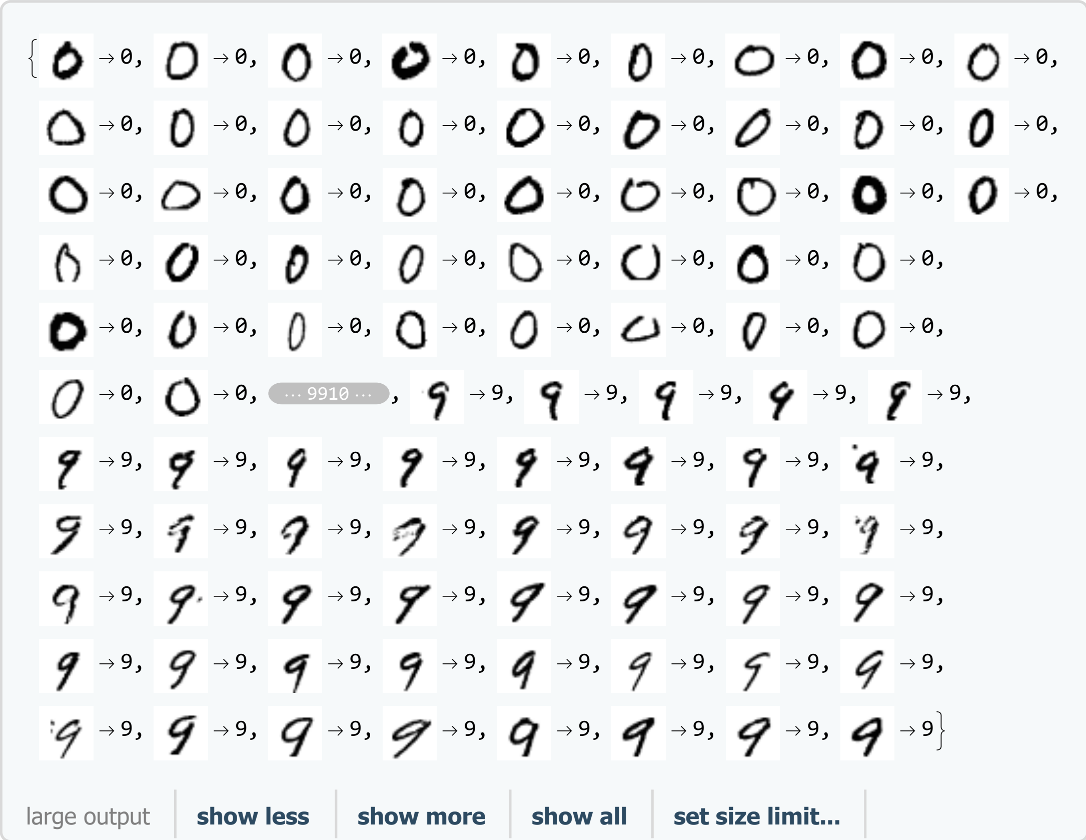
Out[2]=



large output | **show less** | **show more** | **show all** | **set size limit...**

In[3]:=
```
testData = ResourceData["MNIST", "TestData"]
```

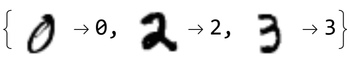
Out[3]=



Pick a few random examples

In[4]:=
```
RandomSample[trainingData, 3]
```

Out[4]=



1. Introduction to NN layers

Layers in Mathematica

In[15]:= `?*Layer`

▼ System`

AggregationLayer	LongShortTermMemoryLayer
BasicRecurrentLayer	MeanAbsoluteLossLayer
BatchNormalizationLayer	MeanSquaredLossLayer
CatenateLayer	PaddingLayer
ConstantArrayLayer	PartLayer
ConstantPlusLayer	PoolingLayer
ConstantTimesLayer	ReplicateLayer
ContrastiveLossLayer	ReshapeLayer
ConvolutionLayer	ResizeLayer
CrossEntropyLossLayer	SequenceAttentionLayer
DeconvolutionLayer	SequenceLastLayer
DotLayer	SequenceMostLayer
DotPlusLayer	SequenceRestLayer
DropoutLayer	SequenceReverseLayer
ElementwiseLayer	SoftmaxLayer
EmbeddingLayer	SpatialTransformationLayer
FlattenLayer	SummationLayer
GatedRecurrentLayer	ThreadingLayer
ImageAugmentationLayer	TotalLayer
InstanceNormalizationLayer	TransposeLayer
LinearLayer	UnitVectorLayer
LocalResponseNormalizationLayer	

Create a LinearLayer that takes as input a vector of length 2 and produces as output a vector of length 3

In[18]:= `linear = LinearLayer[3, "Input" → 2]`

Out[18]= LinearLayer[uninitialized

Parameters

OutputDimensions:	3

Arrays

Weights:	matrix (size: 3×2)
Biases:	optional vector (size: 3)

Ports

Input:	vector (size: 2)
Output:	vector (size: 3)

]

In[19]:= `linear[{2, 3}]`

Out[19]= `$Failed`

Above layer is an ‘uninitialized layer’ which is to say, the learning is not yet started. By using NetInitialize, we can initialize the layer

In[20]:=
```
initLinear = NetInitialize[linear]
```

Out[20]=
```
LinearLayer[
  Parameters
    OutputDimensions:   3
  Arrays
    Weights:            matrix (size: 3×2)
    Biases:             vector (size: 3)
  Ports
    Input:              vector (size: 2)
    Output:             vector (size: 3)
]
```

In[22]:=
```
initLinear[{{2, 3}, {2, 4}, {1, 2}}]
```

Out[22]=
```
{{2.38762, 1.40096, 0.639938}, {3.027, 1.76667, 0.264892}, {1.5135, 0.883337, 0.132446}}
```

Without pesky calculations, we can easily ‘extract’ the weight and biases.

In[27]:=
```
W = NetExtract[initLinear, "Weights"];
W // MatrixForm
```

Out[28]//MatrixForm=

$$\begin{pmatrix} 0.234746 & 0.639378 \\ 0.151909 & 0.365714 \\ 0.882538 & -0.375046 \end{pmatrix}$$

In[24]:=
```
NetExtract[initLinear, "Biases"]
```

Out[24]=
```
{0., 0., 0.}
```

In[29]:=
```
Dot[W, {2, 3}]
```

Out[29]=
```
{2.38762, 1.40096, 0.639938}
```

LinearLayer has only one input, on the other hand, some layers have more than two inputs; MeanSquaredLossLayer needs the *input* and the *target* as two inputs

In[30]:=
```
msloss = MeanSquaredLossLayer[]
```

Out[30]=
```
MeanSquaredLossLayer[
  Parameters
    none
  Ports
    Input:     tensor
    Target:    tensor
    Loss:      real
]
```

Apply the layer to two inputs

In[31]:=
```
msloss[<|"Input" → {1, 2, 3}, "Target" → {4, 0, 4}|>]
```

Out[31]=
```
4.66667
```

2. Net Encoders & Net Decoders

Before we train or test the data, it should be transformed to numeric tensor, which is a NetEncoder’s job; to translate the data to numeric tensors. Let’s see some of the simple examples

In[33]:=
```
imageenc = NetEncoder[{"Image", {12, 12}, "ColorSpace" → "Grayscale"}]
```

Out[33]= NetEncoder[

Type:	Image
ImageSize:	{12, 12}
ColorSpace:	Grayscale
ColorChannels:	1
MeanImage:	None
VarianceImage:	None
Output:	3-tensor (size: 1 × 12 × 12)

]

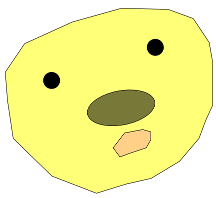
Above encoder produces a 1×12×12 tensor

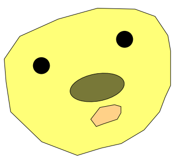
In[36]:= imageenc[] // MatrixForm

Out[36]//MatrixForm=

1.	1.	1.	0.992157	0.996078	1.	1
1.	1.	0.996078	1.	1.	1.	0.99
1.	0.996078	1.	0.878431	0.945098	1.	0.99
0.992157	1.	0.839216	0.0470588	0.615686	1.	0.97
1.	0.992157	0.266667	0.	0.694118	1.	0.49
1.	0.843137	0.0156863	0.403922	0.960784	0.976471	0.12
1.	0.737255	0.	0.619608	0.843137	0.509804	0.03
1.	0.898039	0.14902	0.	0.00392157	0.00392157	0.02
0.996078	1.	0.937255	0.752941	0.694118	0.796078	0.90
1.	0.996078	1.	1.	1.	0.996078	1
1.	1.	0.996078	0.988235	0.988235	0.988235	0.98
1.	1.	1.	1.	1.	1.	1

Let's encode some arbitrary images!



In[37]:= img =  ;

In[38]:=
```
ImageDimensions[img]
```

Out[38]= {360, 322}

In[40]:=
```
imageenc[img] // MatrixForm
```

Out[40]//MatrixForm=

1.	1.	1.	1.	1.	1.	1.	1.	1.
1.	1.	1.	1.	0.74902	0.498039	0.	0.	0.250
1.	1.	1.	0.74902	0.	0.	0.	0.	0.
1.	1.	0.74902	0.	0.	0.	0.	0.	0.
1.	1.	0.25098	0.	0.	0.	0.	0.	0.
1.	1.	0.	0.	0.	0.	0.	0.	0.
1.	1.	0.	0.	0.	0.	0.	0.	0.
1.	1.	0.498039	0.	0.	0.	0.	0.	0.
1.	1.	0.74902	0.	0.	0.	0.	0.	0.
1.	1.	1.	0.74902	0.25098	0.	0.	0.498039	0.749
1.	1.	1.	1.	1.	1.	1.	1.	1.
1.	1.	1.	1.	1.	1.	1.	1.	1.

In[41]:=
```
imageenc[img] // Dimensions
```
Out[41]=
```
{1, 12, 12}
```

We can attach encoder to a layer

In[42]:=
```
pool = PoolingLayer[10, "Input" → NetEncoder[{"Image", 64}]]
```
Out[42]=
```
PoolingLayer[
  Parameters
  KernelSize:      {10, 10}
  Stride:          {1, 1}
  PaddingSize:     {0, 0}
  Function:        Max
  Dimensionality:  2
  Ports
  Input:           image
  Output:          3-tensor (size: 3×55×55)
]
```

Then we can directly apply the image the layer

In[48]:=
```
pool[img] // Shallow
```
Out[48]//Shallow=
```
{{{«55»}, {«55»}, {«55»}, {«55»}, {«55»},
  {«55»}, {«55»}, {«55»}, {«55»}, {«55»}, «45»},
 {{«55»}, {«55»}, {«55»}, {«55»}, {«55»}, {«55»}, {«55»},
  {«55»}, {«55»}, {«55»}, «45»}, {{«55»}, {«55»}, {«55»},
  {«55»}, {«55»}, {«55»}, {«55»}, {«55»}, {«55»}, {«55»}, «45»}}
```

Convert the output back to an image

In[51]:=
```
Image[%, Interleaving → False]
```
Out[51]=

Finally create the final image encoder for MNIST, whose grayscale images are of 28×28.

In[64]:=
```
mnistEnc = NetEncoder[{"Image", {28, 28}, "ColorSpace" → "Grayscale"}]
```
Out[64]=
```
NetEncoder[
  Type:           Image
  ImageSize:      {28, 28}
  ColorSpace:     Grayscale
  ColorChannels:  1
  MeanImage:      None
  VarianceImage:  None
  Output:         3-tensor (size: 1×28×28)
]
```

In[54]:=
```
trainingData[[1, 1]] // ImageDimensions
```
Out[54]=
```
{28, 28}
```

Next, create a decoder for MNIST handwritings. Conversely, decoder translate the numeric tensor to some class. In case of MNIST, there are ten classes, from 0 to 9.

In[56]:=
```
mnistDec = NetDecoder[{"Class", Range[0, 9]}]
```

Out[56]=
```
NetDecoder[ Type: Class
            Labels: {≪1≫}
            Dimensions: 10 ]
```

In[58]:=
```
mnistDec[{0, 0, 0, 0, 0, 0, 0, 0, 0, 1}]
```

Out[58]=
```
9
```

In[60]:=
```
mnistDec[{0.1, 0.3, 0.6, 0, 0, 0, 0, 0, 0, 0}, "TopProbabilities"]
```

Out[60]=
```
{2 → 0.6, 1 → 0.3, 0 → 0.1}
```

In[61]:=
```
mnistDec[{0.1, 0.3, 0.6, 0, 0, 0, 0, 0, 0, 0}, "Entropy"]
```

Out[61]=
```
0.897946
```

Similarly to encoder, we can attach a decoder to the layer

In[62]:=
```
pool = PoolingLayer[10, "Input" → NetEncoder[{"Image", 64}], "Output" → NetDecoder["Image"]]
```

Out[62]=
```
PoolingLayer[ Parameters
              KernelSize: {10, 10}
              Stride: {1, 1}
              PaddingSize: {0, 0}
              Function: Max
              Dimensionality: 2 ]
```

In[63]:=
```
pool[img]
```

Out[63]=

Now construct NN to concatenate the layers together.

In[65]:=
```
uninitializedLenet =
  NetChain[{ConvolutionLayer[20, 5], ElementwiseLayer[Ramp], PoolingLayer[2, 2],
    ConvolutionLayer[50, 5], ElementwiseLayer[Ramp], PoolingLayer[2, 2],
    FlattenLayer[], LinearLayer[500], ElementwiseLayer[Ramp], LinearLayer[10],
    SoftmaxLayer[]}, "Input" → mnistEnc, "Output" → mnistDec]
```

Out[65]= 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

(uninitialized)

ElementwiseLayer (2)

Parameters

Function: Ramp

Ports

Input: 3–tensor (size: 20 × 24 × 24)

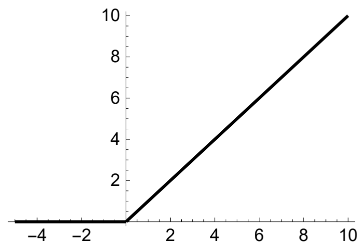
Output: 3–tensor (size: 20 × 24 × 24)

]

Note: Ramp: ReLU

In[84]:=
```
Plot[Ramp[x], {x, -5, 10}]
```

Out[84]=



In[66]:=
```
lenet = NetInitialize[uninitializedLenet]
```

Out[66]= 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

]

In[78]:= **samples = RandomSample[trainingData, 10]**

Out[78]= { → 0,  → 6,  → 1,  → 1,
 → 1,  → 4,  → 4,  → 0,  → 1,  → 9}

In[81]:= **samplelist = Table[samples[[i, 1]], {i, 10}]**

Out[81]= {, , , , , , , , , }

In[82]:= **lenet[samplelist]**

Out[82]= {8, 8, 8, 7, 8, 8, 8, 8, 8, 8}

Pretty stupid...