Contrastive Representation Learning

Author: Tom Lieber (alltom.com)

Background

Contrastive learning is the process of learning an embedding in which 1) items in the same class are close together and 2) items in different classes are far apart. This notebook implements one of the simplest algorithms for doing so using the MNIST dataset of handwritten digits.

Useful links

- Contrastive representation learning tutorial by Lilian Weng
- Mathematica's built-in contrastive loss layer, which I don't use because I wanted to try triplet loss

Parameters

Dimensionality of the embedding space (min 2):

```
In[241]:= embeddingDims = 3;
```

When debugging, it's useful to only use *some* of the digit classes, so this lets you set the number of digit classes to include (min 1, max 10):

```
In[242]:= digitCount = 10;
```

The algorithm below does no hard negative mining, so a large batch size is necessary to ensure that as many hard negatives are randomly included as possible:

```
In[243]:= batchSize = 4096;
```

Training only stops when validation loss fails to improve for this number of batches:

```
In[244]:= trainingPatience = 5;
```

The number of dimensions to use in the visualizations of the resulting embedding space if embeddingDims is greater than 3. Setting this to 2 visualizes using a colored voronoi diagram. Setting it to 3 renders a 3-D point cloud.

```
In[245]:= reducedDimension = 3;
```

How many examples to pull from each class in order to generate the visualization. Mathematica starts falling over if this goes above a few thousand.

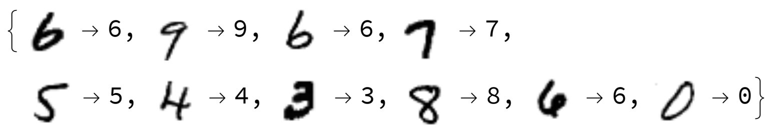
```
In[246]:= sampleCountPerClass = Ceiling[3000 / digitCount];
```

Dataset

```
In[247]:= datasetFilter[ds_] := Select[ds, #[[2]] < digitCount &];
```

```
In[248]:= mnist := ResourceObject["MNIST"];
          mnistTrainSplit = datasetFilter@ResourceData[mnist, "TrainingData"];
          mnistTestSplit = datasetFilter@ResourceData[mnist, "TestData"];
```

```
In[251]:= RandomSample[mnistTrainSplit, 10]
```



```
Out[251]= { → 6,  → 9,  → 6,  → 7,
            → 5,  → 4,  → 3,  → 8,  → 6,  → 0}
```

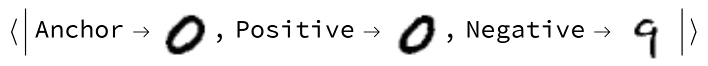
Group the graining examples by digit, producing lists like {{{img → 0}, {img → 0}, …}, {{img → 1}, {img → 1}, …}}. This representation is useful throughout the notebook.

```
In[252]:= bydigitTrain = GatherBy[mnistTrainSplit, #[[2]] &];
          bydigitTest = GatherBy[mnistTestSplit, #[[2]] &];
```

Triplet generation

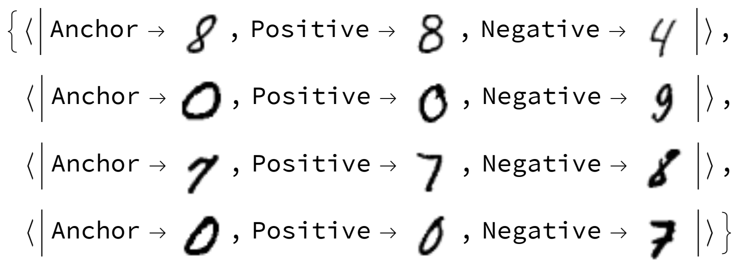
A triplet contains a random example (anchor), another example from the same class (positive), and a random example from a different class (negative).

```
In[254]:= sampleTriple[bydigit_] :=
            Block[{anchorClass, negativeClass, anchor, positive, negative},
             {anchorClass, negativeClass} = RandomSample[bydigit, 2];
             {anchor, positive} = RandomSample[anchorClass, 2];
             {negative} = RandomSample[negativeClass, 1];
             <|"Anchor" → anchor[[1]], "Positive" → positive[[1]], "Negative" → negative[[1]] |>];
          sampleTriple[bydigitTrain]
```



```
Out[255]= ⟨|Anchor → , Positive → , Negative → |⟩
```

```
In[256]:= sampleTripleBatch[bydigit_, n_] := Table[sampleTriple[bydigit], n];
          sampleTripleBatch[bydigitTrain, 4]
```



```
Out[257]= {⟨|Anchor → , Positive → , Negative → |⟩,
           ⟨|Anchor → , Positive → , Negative → |⟩,
           ⟨|Anchor → , Positive → , Negative → |⟩,
           ⟨|Anchor → , Positive → , Negative → |⟩}
```

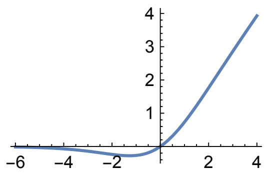
Model definition

Layers

Swish is like Ramp with some extra curvature for that extra oomph.

```
In[258]:= swish = ElementwiseLayer[# * LogisticSigmoid[#] &];
          Plot[swish[x], {x, -6, 4}]
```

Out[259]=



squaredDistanceLayer accepts two vectors and outputs their squared Euclidean distance.

```
In[260]:= squaredDistanceLayer =
            NetChain[{ThreadingLayer[Subtract], ElementwiseLayer[#^2 &], SummationLayer[]}]
```

```
Out[260]= NetChain[ Number of inputs:   2
                    Output port:        real ]
```

Define a projection from the input image to the embedding space. This could be fancier, but for MNIST, it doesn't matter.

```
In[261]:= embeddingLayer = NetChain[{256, swish, 256, swish, embeddingDims}]
```

```
Out[261]= NetChain[ uninitialized  Input port:    array
                                   Output port:   vector (size: 3) ]
```

The network

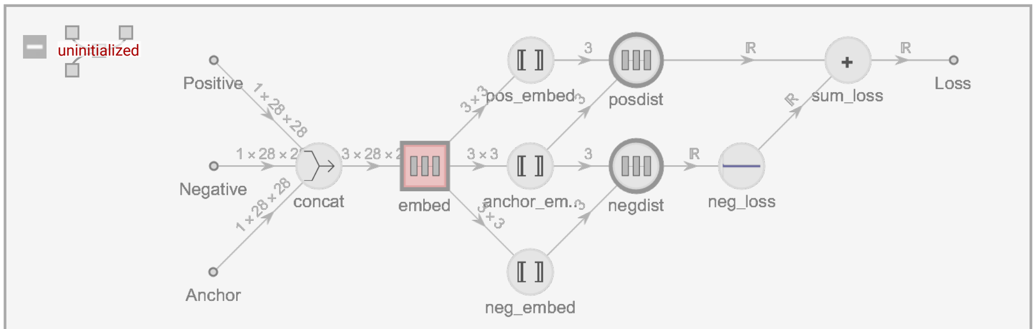
The network's only output is the loss. Its purpose is to find weights for the embedding subnetwork that minimize the loss. We'll extract the embedding subnetwork in the next section.

The closer the anchor and positive example embeddings are, the lower the loss. The further the anchor and negative example embeddings are, the lower the loss (up to a certain distance, after which it's 0).

If we referred to embeddingLayer in the network three separate times, then each instance would get its own independent weights. To get just one embedding subnetwork, we must use NetMapOperator to create only one instance of embeddingLayer. The three images are concatenated, independently fed to embeddingLayer, and their embeddings are concatenated to form the output, which is deconstructed using PartLayers. (What would we do if we needed to use the output of one embedding as the input for a second embedding operation? I don't know.)

```
In[262]:= net = NetGraph[
   <|
    "concat" → CatenateLayer[],
    "embed" → NetMapOperator[embeddingLayer],
    "anchor_embed" → PartLayer[1],
    "pos_embed" → PartLayer[2],
    "neg_embed" → PartLayer[3],
    "posdist" → squaredDistanceLayer,
    "negdist" → squaredDistanceLayer,
    "neg_loss" → ElementwiseLayer[Max[0, 0.1 - Sqrt[#]] ^2 &],
    "sum_loss" → ThreadingLayer[Plus]
    |>,
   {
    NetPort["Anchor"] → "concat",
    NetPort["Positive"] → "concat",
    NetPort["Negative"] → "concat",
    "concat" → "embed",
    "embed" → "anchor_embed",
    "embed" → "pos_embed",
    "embed" → "neg_embed",
    "anchor_embed" → "posdist",
    "anchor_embed" → "negdist",
    "pos_embed" → "posdist",
    "neg_embed" → "negdist" → "neg_loss" → "sum_loss",
    "posdist" → "sum_loss" → NetPort["Loss"]
   },
   "Anchor" → NetEncoder[{"Image", {28, 28}, "Grayscale"}],
   "Positive" → NetEncoder[{"Image", {28, 28}, "Grayscale"}],
   "Negative" → NetEncoder[{"Image", {28, 28}, "Grayscale"}]
  ]
```

Out[262]= NetGraph[



Input Ports
Anchor: image
Positive: image
Negative: image

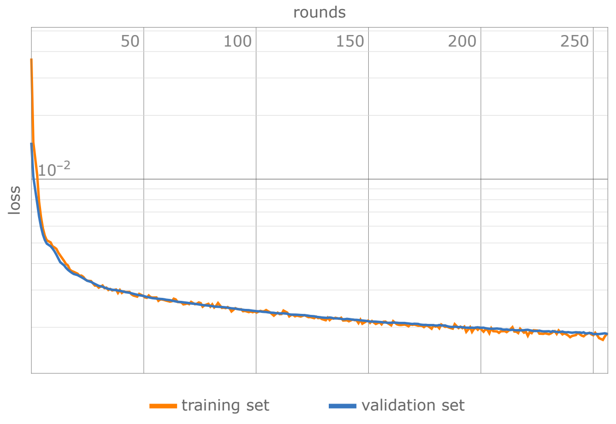
Output Port
Loss: real

]

Train

```
In[263]:= trained = NetTrain[
            net,
            sampleTripleBatch[bydigitTrain, #BatchSize] &,
            All,
            ValidationSet → sampleTripleBatch[bydigitTest, batchSize],
            BatchSize → batchSize,
            TrainingStoppingCriterion → <|
              "Criterion" → "Loss", "Patience" → trainingPatience|>,
            MaxTrainingRounds → Infinity
           ]
```

```
Out[263]= NetTrainResultsObject[
  NetTrain Results
        summary   batches: 256, rounds: 256,
                  time: 6.1min, examples/s: 2866
           data   training examples: 4096,
                  validation examples: 4096,
                  processed examples: 1048576,
                  skipped examples: 0
         method   ADAM optimizer, batch size 4096, CPU
          round   loss: 1.86×10^-3
     validation   loss: 1.87×10^-3
]
```



Extract the trained embedding sub-network by pulling it from the NetMapOperator named “embed”. There must be a nicer way to grab it, but I don’t know what it is.

```
In[264]:= trainedEmbeddingSubnet =
            NetExtract[NetExtract[trained["TrainedNet"], "embed"], "Net"];
          embedder = NetChain[{ReshapeLayer[{28, 28}], trainedEmbeddingSubnet},
            "Input" → NetEncoder[{"Image", {28, 28}, "Grayscale"}]]
```

```
Out[265]= NetChain[  Input port:    image
                     Output port:   vector (size: 3) ]
```

Evaluate

A color map that I found slightly more legible than the built-in default:

```
In[266]:= colors = {"#e6194b", "#3cb44b", "#ffe119", "#4363d8",
             "#f58231", "#911eb4", "#46f0f0", "#f032e6", "#bcf60c", "#fabebe",
             "#008080", "#e6beff", "#9a6324", "#fffac8", "#800000", "#aaffc3",
             "#808000", "#ffd8b1", "#000075", "#808080", "#ffffff", "#000000"};
```

```
In[267]:= embedBydigit[bydigit_] := Map[Map[embedder[#[[1]]] → #[[2]] &, #] &, bydigit]
          sampleBydigit[arr_] := Map[RandomSample[#, sampleCountPerClass] &, arr]
```

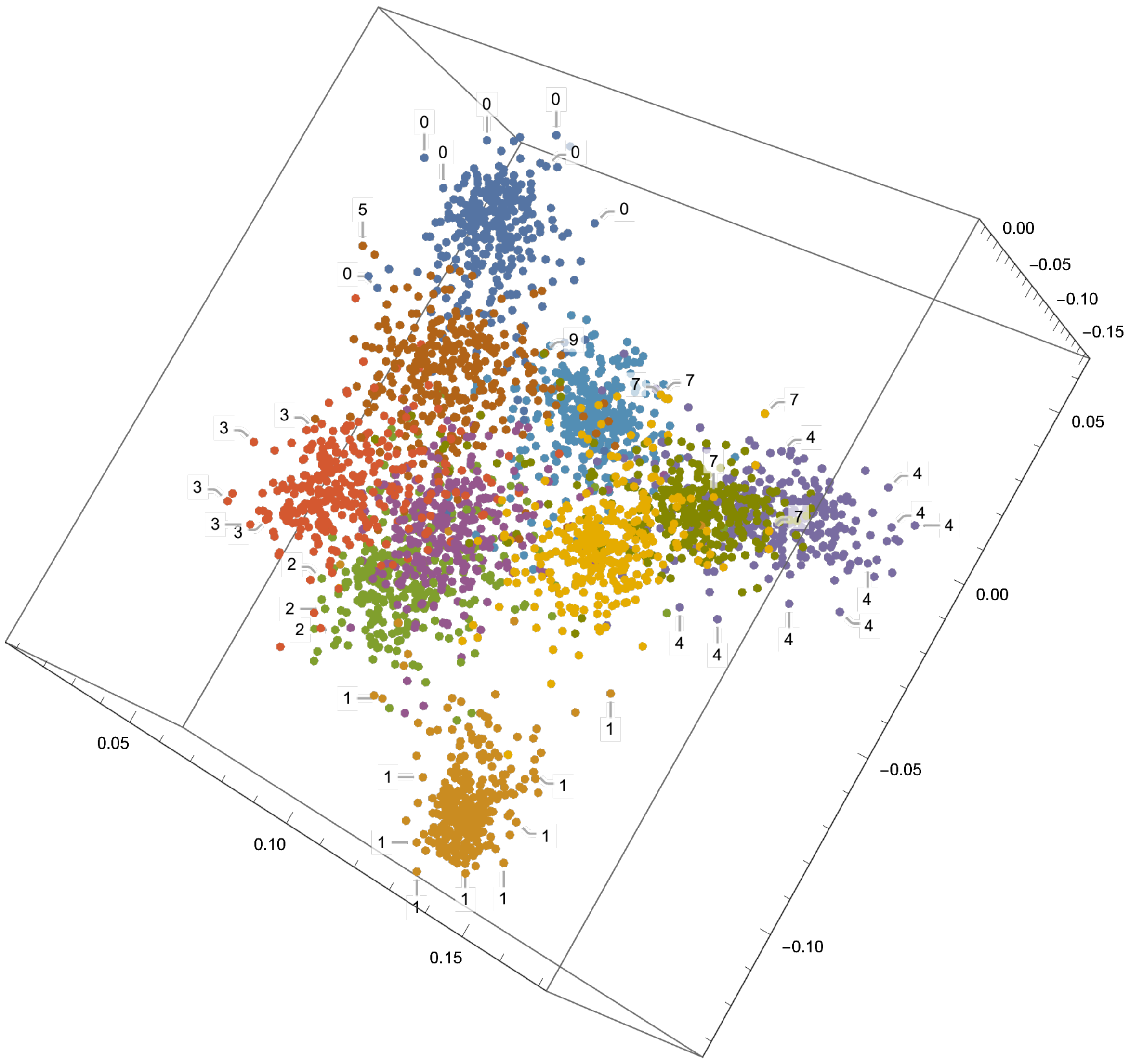
This function visualizes an embedding in 2-D by rendering a Voronoi diagram of the examples in embedding space and giving each class a separate color.

```
In[269]:= voronoi[bydigitEmbeddings_] :=
           Block[{sample, flatSample, cellColors, v, cellIndices},
            sample = sampleBydigit[bydigitEmbeddings];
            flatSample = Catenate[sample];
            cellColors = Map[Opacity[1, RGBColor[#]] &, colors[[Values@flatSample + 1]]];
            v = VoronoiMesh[Keys@flatSample,
              BaseStyle → Transparent, MeshCellStyle → {1 → Opacity[0]}];
            cellIndices = #[[2]] & /@ Region`Mesh`MeshMemberCellIndex[v][Keys@flatSample];
            v = SetProperty[{v, {2, cellIndices}}, MeshCellStyle → cellColors];
            Show[v, ListPlot[sample, PlotStyle → colors]]
           ]
```

Visualize the embedding space according to the config at the top. 2-D is voronoi, 3-D is point cloud. When the embedding space has more than 3 dimensions, SVD is used to reduce its dimensionality to reducedDimension first.

```
Switch[embeddingDims,
 2, voronoi[embedBydigit@sampleBydigit@bydigitTrain],
 3, ListPointPlot3D[embedBydigit@sampleBydigit@bydigitTrain, PlotStyle → colors],
 _, Block[{allEmbeddings, bydigitEmbeddingsLowdim},
  allEmbeddings = Catenate[embedBydigit@bydigitTrain];
  bydigitEmbeddingsLowdim = sampleBydigit@
    GatherBy[Thread[DimensionReduce[Keys[allEmbeddings], reducedDimension,
        Method → "LatentSemanticAnalysis"] → Values[allEmbeddings]], #⟦2⟧ &];
  Switch[reducedDimension,
   2, voronoi[bydigitEmbeddingsLowdim],
   3, ListPointPlot3D[bydigitEmbeddingsLowdim, PlotStyle → colors]
  ]
 ]
]
```

Out[270]=



0.00
−0.05
−0.10
−0.15
0.05
0.00
−0.05
−0.10
0.05
0.10
0.15
0
5
3
2
1
9
7
4