The Handwritten Digits Model

Introduction

This tutorial explores how to estimate the parameters of the handwritten digits model. It does so by "opening the black box" using an interactive session. The objective is to delve into how to conduct an estimation rather than to find the absolute best search algorithm. The results are good, but they could (have) certainly be bettered.

Along the way, brief functions are defined which:

- define the components of the model and show how the model feeds forward to make predictions;
- calculate the gradient efficiently using back propagation;
- demonstrate how the gradient can be applied in steps to reduce the objective;
- define epochs to more efficiently use the data in the image dataset; and
- twiddle the knobs to vary the run parameters.

This is an interactive session using APL. You can execute all the statements presented with little more than copy and paste. You're encouraged to explore on your own along the way.

The APL environment

Why APL? This provides a rich language ideally suited to the manipulation of matrices and higher order arrays. It is mature, stable, well documented and well understood. It's also interactive, which means that you can break a big problem down into smaller steps and examine what's happening along the way.

All of the text in the `APL385 Unicode` font is executable in APL. The particular APL used here is Dyalog APL 17.1 with:

```
⎕io←0
⎕pp←6
⎕rl←16807
]boxing on
```

Dyalog APL is freely available for non-commercial use at www.dyalog.com.

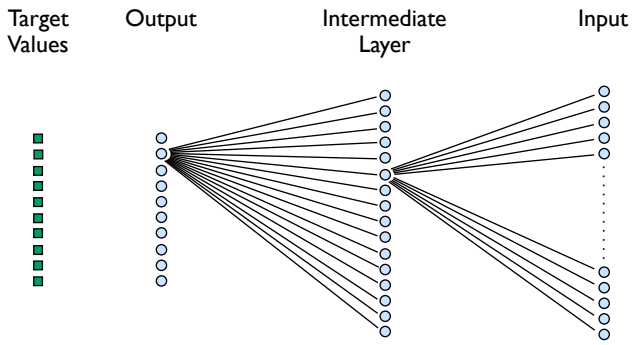
Definitions

This tutorial makes use of a few utility functions. These are defined here with brief comment.

```
num←{×/⍴⍵}                                  Number
sum←{+/,⍵}                                  Sum
mean←{(sum÷num)⍵}                           Mean
sop←{+/,⍺×⍵}                                Sum of product
ssq←{sop⍨⍵}                                 Sum of squares
image←{28 28⍴(⊂⍵>0.5)⌷' *'}                 Rough display of an image sample
timer←{⎕DL⍵ ⋄ ⎕DL←⍵ ⋄ 0.001×⎕DL-⍨1⌷⎕AI}      Execution time in seconds
correct←{0.01 rnd 0.01×sum TestLabels∧⍉{⍵=⌈/⍵}⍤1⊢  % correct
  ⍉ff ⍵,⊂TestImages}
incorrect←{⍸∨⌿TestLabels≠⍉{⍵=⌈/⍵}⍤1⊢        Indices of incorrect matches
  ⍉ff ⍵,⊂TestImages}
```

The Model

The model used is based on the three layer model described more fully in [0]. It has an input layer, one intermediate layer and an output layer. The input layer is a vector of `784` black and white pixel intensities on a scale from `0` to `1`. The intermediate layer is a vector of length `16` and the output layer is a boolean vector of length `10`.



The transitions between layers incorporate biases and an activation function:

```
tf←{activate ⍺+.×1,⍵}                       Transition function.
```

The intermediate layer is calculated from the input layer by `intermediate←p0 tf input` where `p0` is a transition matrix of shape `16 785`. The output layer is produced from the intermediate layer in a similar way with `output←p1 tf intermediate` where `p1` is a transition matrix of shape `10 17`.

There are a number of possibilities for the activation function. We'll consider three:

```
sigmoid←{÷1+*-⍵}                            Sigmoid 1 / 1+e^-x
relu←{0⌈⍵}                                  Rectified Linear Unit
tanh←{7○⍵}                                  Hyperbolic tangent
```

For each activation function we'll need to know its derivative. The corresponding derivatives are:

```
dsigmoid←{{⍵×1-⍵}sigmoid ⍵}
drelu←{0<⍵}
dtanh←{{(1+⍵)×1-⍵}tanh ⍵}
```

The target is a boolean vector of length `10` with a single `1` marking the digit. The model's corresponding predictions are made with the feed forward function:

```
ff←{⊃tf/⍵}
```

Note that the feed forward function `ff` handles a matrix argument as input, as long as the number of samples appears as the final axis of its argument.

The objective function is the measure chosen to show how far a set of parameters deviates from optimal. We'll consider two:

```
leastsquares←{ssq target-ff ⍵,⊂input}


cren←{(α×⍟⍵)+(1-α)×⍟1-⍵}
crossentropy←{-sum target cren ff ⍵,⊂input}
```

The gradient function is based on the three layer gradient function from [0]. It is defined for the least squares objective as:

```
r←gr_lsq(p1 p0 target input);s;t;z0;a0;z1;a1;R;e1;e0

⍝ Gradient for least squares objective

  a0←activate z0←p0+.×s←1,input
  a1←activate z1←p1+.×t←1,a0
  R←2×a1-target
  e1←R×dactivate z1
  e0←((⍉0 1↓p1)+.×e1)×dactivate z0
  r←(e1+.×⍉t)(e0+.×⍉s)
  r÷←¯1↑2↑(⍴input),1
```

For the cross entropy objective, the gradient function is the same with just one change. The line calculating `R` needs to be: `R←(a1-target)÷a1×1-a1`.

Note that `gr_lsq` returns the mean gradient of all the samples in `input`. This standardizes the magnitude of the gradient's value, making it possible to compare gradients derived from inputs with differing numbers of samples.

Estimating the Parameters with Least Squares

Let's start with some manual examination of the behaviour of the model with a least squares objective. First we'll initialize the structure and transition matrices; then set the objective function, its gradient, the activation function and its derivative.

```
      ⎕rl←16807
      structure←10 16 784
      ρ¨p←(1+1↓structure)÷⍨¨(¯1↓structure){?(⍺,⍵+1)ρ0}¨1↓structure
```

10 17	16 785

```
      obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
      target input←TrainingLabels TrainingImages
```

As a reference point, let's note the value for the objective function if we make no prediction at all (i.e. the output is all zero) is:

```
      ssq target
60000
```

And if we make a prediction with the randomly generated weights `p`:

```
      (obj,correct)p
186791 10.09
```

This tells us that making a prediction with this set of random weights actually produces a worse objective value than if we'd made no guess at all. The number of correct guesses is about 10% which is as we'd expect from pure guesswork (as there are `10` possible values in the output layer).

Iterating with a gradient

Let's now calculate the gradient and see the effect of taking some steps. We'll start with something really small to be true to the Taylor expansion assumption:

```
      g←gr p,target input
      obj p-0.000001×g
186791
```

So much for that idea. It appears we've taken a really, really tiny step, producing no noticeable change in the objective. Let's be bolder:

```
      obj p←p-0.001×g
186612
```

That's definitely an improvement. We've moved the objective down, if only by `0.1`%. Let's see if we can get a range of values that might help us get a better feeling for the `k` parameter.

```
    k←0.001
    ↑{(⍵,obj)p-⍵×g}¨k×10*⍳5
 0.001 186433
 0.01  184831
 0.1   169591
 1      85578.1
10      54246
```

We can certainly see there's progress to be made. Perhaps we should plunge right in and set `k←10`? Maybe not. This is just the beginning, so we should be cautious. Let's work with `k←1`. We'll update the parameters and calculate a new gradient:

```
    k←1
    p-←k×g
    g←gr p,target input
```

And again examine a spread of `k` values:

```
    ↑{(⍵,obj)p-⍵×g}¨k×10*¯2+⍳5
  0.01 85245
  0.1  82288.4
  1    60117.1
 10    59970.9
100    60000
```

Once more, it is tempting to set `k←10`. However, there's no rush. Let's be patient and keep `k←1`. This is probably a good point to formalize what happens when we take a step:

```
    step←{⍵-k×gr ⍵,target input}
```

This takes a vector of transition matrices and calculates the gradient for the entire training dataset. This is used to adjust the transition matrices using the global parameter `k`. Let's take one more step:

```
    obj p←step p
60117.1
```

And in order to take multiple steps:

```
    obj p←step⍣4⊢p
54159.9
```

Let's do another `4` steps and reassess the `k` parameter:

```
    obj p←step⍣4⊢p
54052.8
    g←gr p,target input
    ↑{(⍵,obj)p-⍵×g}¨k×10*¯2+⍳5
  0.01 54052.7
  0.1  54052.2
  1    54047.1
 10    54047.8
100    56222.8
```

This confirms that a `k` value of `1` is working well. Let's press on with another ten steps.

```
    (obj,correct)p←step⍣10⊢p
54030.7 11.35
```

Let's take stock of where we've got to. According to my counting, we've done `21` steps with some progress. The objective has come down to `54030.7` and the percentage of correct guesses has increased to `11.35%`. That's not earth-shattering. It is understandable however. Early on in an estimation, the parameters can be expected to be far from any minimum of the objective. The gradient at such great distance is not necessarily pointing in a very productive direction. Progress should be expected to be slow until the estimation gets closer to the minimum point.

What about performance? There's not much point in writing code to solve a problem if it takes forever to run. Fortunately, that's not an issue so far. For example, the last executed line above takes `10` steps, each step calculating the gradients for both transition matrices using all `60000` samples, updates the transition matrices and calculates the sum of squares of the residuals. It takes just under `12` seconds.

Examining the gradient

It's interesting to get an idea of what the gradient looks like. That's not easy when there are `12730` elements. But we can tease out some bits of information.

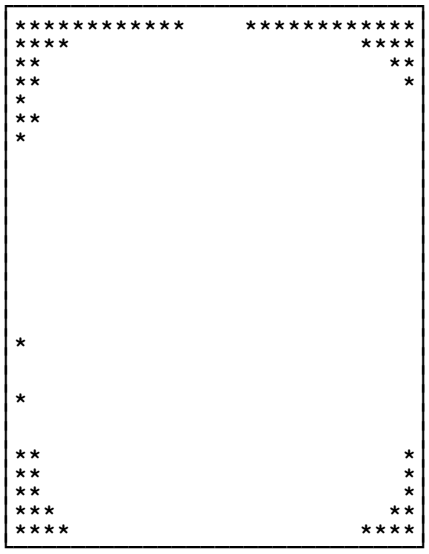
Zeros are important. When they occur, they stop an element in one layer contributing to a change in an element of the next layer. If we start with the matrix `g0` that transforms the inputs into the intermediate layer:

```
    g1 g0←(⊂0 1)↓¨gr p,target input                    Ignoring the biases
    ∪0+.=g0
16 0
```

This tells us that the columns of the `g0` gradient are either all zero or all non-zero.

Which elements of the original image grid are "dead"? By that we mean, they have a zero gradient and are unchanging. We can use the `image` function to display this:

```
    ⊂image 0∧.=g0
```



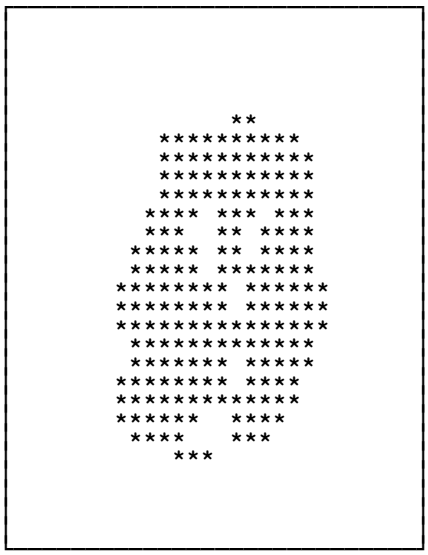
This makes sense. As the MNIST images are centred in the `28` by `28` grid, we'd expect some of the peripheral cells not to be involved at all in making predictions.

What about the second gradient `g1`?

```
    0∊g1                                                    No zeros in this matrix.
0
```

So, which parts of the grid are contributing the most to the changes brought about by the `g0` gradient? Here's a way to look at that:

```
    t←ssq⍤1⊢⍉g0
    ⊂image t∊200↑t[⍒t]
```



As we'd expect, the central cells are the biggest contributors.

Batches and epochs

Estimating the transition parameters is time consuming, largely due to the number of calculations it takes to calculate the gradient. Here's a timing for the gradient of the entire training dataset:

```
    timer'g←gr p,target input'
1.184
```

Although this demonstrates the speed of APL's matrix operations, it could still turn out to be bad news if we have to do many iterations. An alternative technique, which works well in practice, is to use a cruder approximation to the gradient which can be calculated more quickly. This simply uses just a portion of the training images. Here's a timing for a gradient calculation using `1`% of the training dataset.

```
    timer'g←gr p,(10 600↑TrainingLabels)(784 600↑TrainingImages)'
0.011
```

If the approximation to the gradient holds up, then this makes it possible to proceed with many more steps in the same amount of time. The algorithm goes like this: choose a batch at random from the training images as `input` and `target`, calculate the gradient, take a step and repeat until all the training images have been used once. This is known as an epoch and is coded as follows:

```
r←epoch(p1 p0 k batch);t;n;s;c;x;input;target
  t←1↓⍴TrainingImages ⋄ n←⌊t÷batch ⋄ s←(n,batch)⍴t?t
  c←0
  :While c<n
     x←c⌷s
     input←TrainingImages[;x]
     target←TrainingLabels[;x]
     p1 p0-←k×gr p1 p0 target input
     c+←1
  :EndWhile
  r←p1 p0 k batch
```

`epoch` takes the two parameter matrices `p1` and `p0`, the `k` step factor and the batch size as arguments. (Make sure to choose `batch` as a divisor of `60000`.) `epoch` runs through all of the training data once, calculating the gradient and taking a step for each batch. The result is similar to the argument but with the latest revised values for `p1` and `p0`. Here's a session using `epoch`.

```
     ⎕rl←16807
     structure←10 16 784
     p←(1+1↓structure)÷⍨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
     obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
     (obj,correct)p
186791 10.09
     k←1 ⋄ batch←600
     obj 2⍴p←epoch p,k batch
53637.3
     obj 2⍴p←epoch p
44263.4
     obj 2⍴p←epoch p
31410.6
     obj 2⍴p←epoch p
23878.9
     obj 2⍴p←epoch p
18463.9
     (obj,correct)2⍴p←epoch p
15092.9 88.63
```

We're clearly making good progress. After running six epochs, the objective has come down to about `8`% of its starting value and the percentage of correct predictions has risen from `10`% to `88`%. We've been fortunate in our knob twiddling.

Running multiple epochs

Note that it's possible to run multiple epochs quite succinctly. This is because the result of one use of `epoch` is suitable as the argument to a consequent use of `epoch`. For example, the following statement executes three epochs:

```
     (obj,correct)≢pp←epoch⍣3⊢p
10996
```

Alternatively, we can run multiple epochs with a halting criterion. The following runs `epoch` repeatedly until successive results have objective functions differing by less than `5`.

```
     halt←{(obj ≢⍺)<5+obj ≢⍵}
     (obj,correct)≢epoch⍣halt p
10373.7
```

A word of warning. Choosing a halting criterion based on the objective function has to be done with care. Often, early on in an estimation the objective function may come down very slowly before picking up speed later. In some cases the approximate gradient used may actually cause an increase in the objective. This sort of behaviour could lead to early termination. A hybrid approach is probably a good idea: run enough epochs until the percentage of correct guesses is better than 90%, then use a halting criterion similar to `halt` defined above.

Varying the batch size

We have chosen a batch size of `600` so far. Not a lot of thought went into this. Perhaps it will be useful to have a larger `batch`? Let's compare the improvements made with a selection of batch sizes: `300`, `600`, `3000`, `6000`.

```
     (obj,correct)≢epoch(3⊃p),300
9927.68 91.28
     (obj,correct)≢epoch(3⊃p),600
10391.2 91
     (obj,correct)≢epoch(3⊃p),3000
10847.2 90.68
     (obj,correct)≢epoch(3⊃p),6000
10913.9 90.6
```

This rough review suggests choosing smaller batch sizes. We'll continue with `batch←300`.

```
     p←(3⊃p),300
```

Completing the estimation

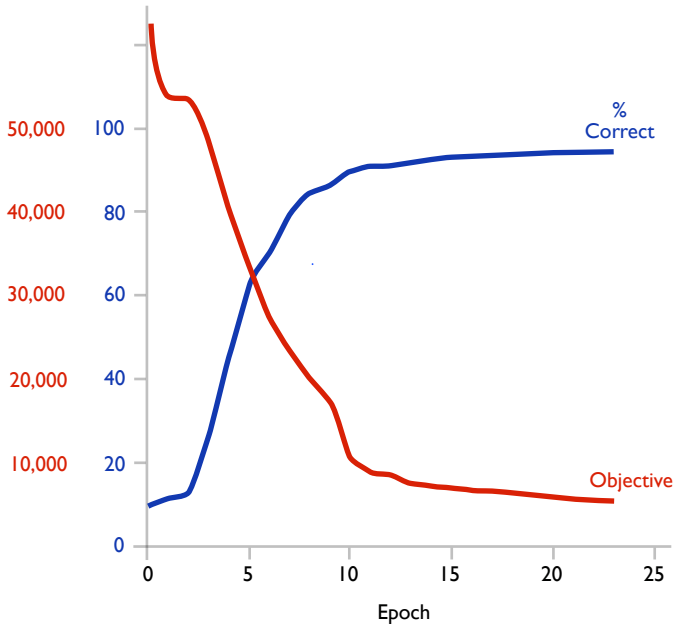
We're now ready to continue the estimation. But, while we're at it, let's get a feel for performance. An elapsed time per `epoch` is probably a good measure:

```
     0.25×timer'p←epoch⍣4⊢p'
1.1825
     (obj,correct)≢p
8402.16 92.15
```

That's `1.2` seconds for each epoch of `100` steps. Let's do another `10` epochs.

```
      (obj,correct)2⍴⍪epoch⍤10⊢p
6688.83 93.54
```

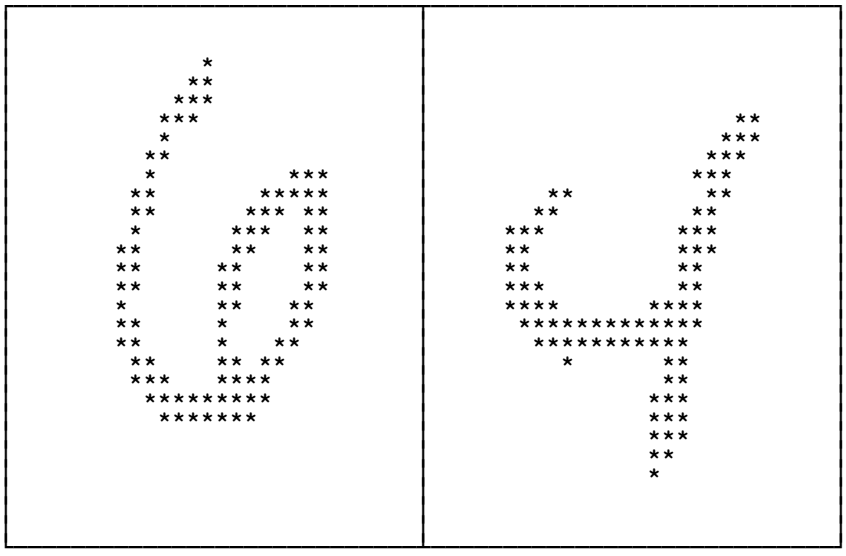
For our purposes, this is a pretty good place to pause. We've run 23 epochs and have taken the objective down from 186791 to 6688. The percentage of correct guesses has increased to better than 93%. Total run time is less than 30 seconds. Here's a graph of the results.



The model's results

The final value of p that we calculated has the parameters that we can use to make predictions with about 94% accuracy. Let's check a couple of examples:

```
      image¨↓⍉TestImages[;66 67]
```



```
predicted←{{⊃⍒⍵}⍤1⊢⍉⍤2 ff(2ρp),⊂⍵}
```
Predicted digits.

```
observed←{{⊃⍒⍵}⍤1⊢⍉⍤2⍵}
```
Observed digits

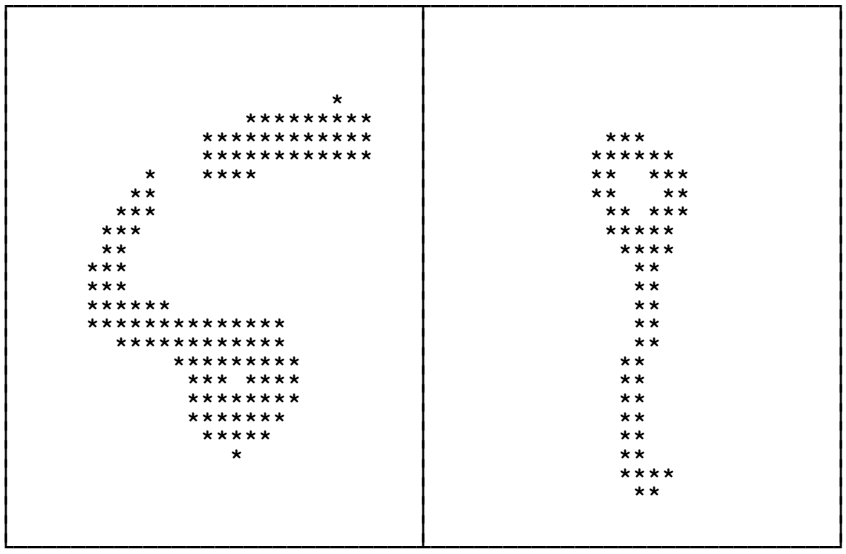
```
    predicted TestImages[;66 67]
6 4
    observed TestImages[;66 67]
6 4
```

What about the images that we failed to match?

```
    ρbad←incorrect 2ρp
646
    20ρbad
8 33 38 66 77 119 124 149 187 217 233 241 247 259 290 300 313 320 321 340
```

Here's what two of the incorrectly guessed images look like:

```
    image⍤1⍉⍤2 TestImages[;8 320]
```



These images are definitely a bit vague and the model's predictions (`6` and `8`) are not out of this world. The desired answers are `5` and `9`.

```
    predicted TestImages[;8 320]
6 8
```
Predicted

```
    observed TestLabels[;8 320]
5 9
```
Observed

Improving Performance

Profiling

Dyalog APL provides a nice tool, `⎕profile`, for tracking where the CPU time gets used. For example, we could break down the execution of `epoch` as follows:

```
⎕RL←16807
structure←10 16 784
p←(1+1↓structure)÷⍨¯1↓structure{?(⍺,⍵+1)⍴0}¨1↓structure
obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←sigmoid ⋄ dactivate←dsigmoid

⎕profile'clear' ⋄ ⎕profile'start'
t←epoch p
⎕profile'stop'


]Profile summary -code -lines                          (slightly edited)
```

```
Total time: 1289.2 msec

Element                    msec      %  Calls  Code
#.epoch[7]                700.2   54.3    200  p1 p0-←k×gr p1 p0 target input
#.epoch[5]                581.9   45.1    200  input←TrainingImages[;x]
#.gr_lsq[9]               330.9   25.7    200  r←(e1+.×⍉t)(e0+.×⍉s)
#.gr_lsq[4]               317.2   24.6    200  a0←activate z0←p0+.×s←1,input
#.dactivate[0]             18.0    1.4    800  dactivate←{{⍵×1-⍵}sigmoid ⍵}
#.activate[0]              17.7    1.4    400  activate←{÷1+*-⍵}
#.gr_lsq[8]                16.0    1.2    200  e0←((⍉0 1↓p1)+.×e1)×dactivate z0
#.sigmoid[0]               15.1    1.2    400  sigmoid←{÷1+*-⍵}
#.gr_lsq[5]                11.2    0.9    200  a1←activate z1←p1+.×t←1,a0
#.gr_lsq[10]               10.4    0.8    200  r÷←¯1↑2↑(⍴input),1
#.gr_lsq[7]                 7.5    0.6    200  e1←R×dactivate z1
#.epoch[6]                  3.7    0.3    200  target←TrainingLabels[;x]
                                               ⋄ n←⌊t÷batch ⋄ s←(n,batch)⍴t?t
#.gr_lsq[6]                 1.2    0.1    200  R←2×a1-target
#.epoch[4]                  0.4    0.0    200  x←c⎕s
#.epoch[3]                  0.2    0.0    201  :While c<n
```

Most of this comes as no surprise. The majority of time is spent calculating the gradient and revising the parameter arrays. However, the second call on line `5` of `epoch` does catch the eye. This is where the random batches are set up. Perhaps this code can be improved. Here's a revised `epoch` function. It removes the indexed selection from the loop, doing most of the work up front once.

An improved `epoch` function

```
r←epoch(p1 p0 k batch);t;n;s;c;x;input;target
  t←1↓⍴TrainingImages ⋄ n←⌊t÷batch ⋄ s←(n,batch)⍴t?t
  input←TrainingImages[;s] ⋄ target←TrainingLabels[;s]
  c←0
:While c<n
    p1 p0-←k×gr p1 p0(target[;c;])(input[;c;])
    c+←1
:EndWhile
  r←p1 p0 k batch
```

Here's what the CPU profile looks like now:

```
⎕profile'clear' ⋄ ⎕profile'start'
t←epoch p
⎕profile'stop'
```

```
]Profile summary -code -lines                    (slightly edited)
```

```
Total time: 1029.1 msec

Element              msec      %  Calls  Code
#.epoch[5]          811.7   78.9    200  p1 p0-←k×gr p1 p0(target[;c;])(input[;c;])
#.gr_lsq[9]         332.5   32.3    200  r←(e1+.×⍉t)(e0+.×⍉s)
#.gr_lsq[4]         316.8   30.8    200  a0←activate z0←p0+.×s←1,input
#.epoch[2]          214.4   20.8      1  input←TrainingImages[;s] ⋄ target←TrainingLabels[;s]
#.dactivate[0]       18.1    1.8    800  dactivate←{{ω×1-ω}sigmoid ω}
#.activate[0]        17.6    1.7    400  activate←{÷1+*-ω}
#.gr_lsq[8]          16.0    1.6    200  e0←((⍉0 1↓p1)+.×e1)×dactivate z0
#.sigmoid[0]         15.2    1.5    400  sigmoid←{÷1+*-ω}
#.gr_lsq[5]          11.2    1.1    200  a1←activate z1←p1+.×t←1,a0
#.gr_lsq[10]         10.3    1.0    200  r÷←¯1↑2↑(⍴input),1
#.gr_lsq[7]           7.6    0.7    200  e1←R×dactivate z1
#.epoch[1]            1.3    0.1      1  t←¯1↑⍴TrainingImages
                                          ⋄ n←1⌈⌊t÷batch ⋄ s←(n,batch)⍴t?t
#.gr_lsq[6]           1.2    0.1    200  R←2×a1-target
#.epoch[6]           0.2    0.0    200  c+←1
#.epoch[4]           0.2    0.0    201  :While c<n
#.epoch[7]           0.1    0.0    200  :EndWhile
```

That's about a 20% improvement. Of course, we could keep going ... but later.

Variations

Increasing the size of the intermediate layer

We've used a hidden layer so far with 16 elements. Can we improve our results with more elements in this layer? Here's how to do this for a layer with 30 elements. This run does 50 epochs with k←1 and a batch size of 200.

```
      ⎕rl←16807
      structure←10 30 784
      p←(1+1↓structure)÷⍨¯1↓structure{?(⍺,⍵+1)⍴0}¨1↓structure
      obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
      timer'p←epoch⍣10⊢p,1 200'
13.013
      (obj,correct)2⍴⍪p
6247.57 93.82
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
4821.93 94.83
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
4145.85 95.35
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
3765.94 95.78
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
3407.54 96.01
```

After 50 epochs and approximately 85 seconds, we have correct predictions for 96% of the images. That's a useful improvement. Of course, it raises new questions (Why 30? Why not 45?) which might be interesting to study, but not here, not now.

Using the cross entropy objective

Let's now do the estimation using the cross entropy objective. To do so, we'll use a variation of gr_lsq which uses the appropriate R value. It's gr_cren and it looks like this:

```
r←gr_cren(p1 p0 target input);s;t;z0;a0;z1;a1;R;e1;e0

⍝ Gradient for cross entropy objective

  a0←activate z0←p0+.×s←1,⍪input
  a1←activate z1←p1+.×t←1,⍪a0
  R←(a1-target)÷a1×1-a1
  e1←R×dactivate z1
  e0←((⍉0 1↓p1)+.×e1)×dactivate z0
  r←(e1+.×⍉t)(e0+.×⍉s)
  r÷←¯1↑2↑(⍴input),1
```

Now we can run an estimation as follows:

```
      ⎕rl←16807
      structure←10 30 784
      p←(1+1↓structure)÷⍨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
      obj←crossentropy ⋄ gr←gr_cren ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
      timer'p←epoch⍣10⊢p,1 200'
12.964
      (obj,correct)2⍴⍪p
16593.9 95.5
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
12813.9 96.27
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
10838.3 96.56
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
9565.48 96.56
      (obj,correct)2⍴⍪p←epoch⍣10⊢p
8679.53 96.43
```

These results are revealing. Significantly, the estimation pretty well converges after `30` epochs with a slightly improved correct percentage. And, a little surprisingly, although further epochs do manage to reduce the objective, they do not improve the percentage of correct predictions.

What is the comparable least squares objective for this set of parameters?

```
      leastsquares 2⍴⍪p
2156.74
```

Using other activation functions

In the modelling we've done so far, we've used the sigmoid function `{÷1+*-⍵}` as the activation function. This has had the effect of forcing the values output in each layer to have values between `0` and `1`. We can show that with:

```
      sigmoid ¯10 ¯5 0 5 10
0.0000453979 0.00669285 0.5 0.993307 0.999955
```

This has had a "dampening" effect on calculated values, ensuring that they (particularly the gradient) do not get too large.

Relu activation

There are other possibilities for the activation function. One commonly used is the Rectified Linear Unit, which simply forces negative values to be zero. I confess to not knowing the theoretical appeal of this activation function but it gets a lot of use in practice, deservedly so. We can set things up with:

```
      ⎕rl←16807
      structure←10 30 784
      p←(1+1↓structure)÷⍨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
      obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←relu ⋄ dactivate←drelu
```

The first task is to find a suitable value for `k`, the learning rate:

```
    g←gr p,target input
    ↑{(⍵,obj)p-⍵×g}¨0.08 0.09 0.1 0.15 0.2
0.08 54460.6
0.09 54376.2
0.1  54311.9
0.15 54304.6
0.2  54852.5
```

Let's proceed with `k←0.1` and a batch size of `300`.

```
    (obj,correct)2⍴p←p,0.1 300
55821.1 15.54
```

This seems to be making some progress. Let's run an entire epoch:

```
    timer'p←epoch p'
1.175
    (obj,correct)2⍴p
14486 89.24
```

That's excellent. After just a single epoch, we're up to `89`% correct predictions. Continuing on:

```
    (obj,correct)2⍴p←epoch⍣10⊢p
5825.38 95.59
    (obj,correct)2⍴p←epoch⍣10⊢p
4716.07 96.23
    (obj,correct)2⍴p←epoch⍣10⊢p
4173.42 96.68
    (obj,correct)2⍴p←epoch⍣10⊢p
3916.19 96.81
    (obj,correct)2⍴p←epoch⍣10⊢p
3665.91 96.94
```

So, after `51` epochs we've got up to close to `97`%. Not bad, for about `60` seconds processing on a quite ordinary desktop computer.

Tanh activation

We can run a similar estimation for the hyperbolic tangent activation, as follows:

```
      ⎕rl←16807
      structure←10 30 784
      p←(1+1↓structure)÷⍨¯1↓structure{?(⍺,⍵+1)⍴0}¨1↓structure
      obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←tanh ⋄ dactivate←dtanh

      (obj,correct)2⍴p←p,0.1 300
55825.6 15.52
      timer'p←epoch p'
1.275
      (obj,correct)2⍴p
27012.6 83.36
      (obj,correct)2⍴p←epoch⍣10⊢p
12353.8 92.2
      (obj,correct)2⍴p←epoch⍣10⊢p
9906.35 92.88
      (obj,correct)2⍴p←epoch⍣10⊢p
9621.64 92.94
      (obj,correct)2⍴p←epoch⍣10⊢p
8992.91 93.12
      (obj,correct)2⍴p←epoch⍣10⊢p
9228.26 93.27
      (obj,correct)2⍴p←epoch⍣10⊢p
9146.85 93.42
```

This result seems to be not quite as good as the `relu` or `sigmoid` activations. It's a bit slower to get going and seems to take longer to make progress. But, we put no effort into finding a good learning rate or batch size and, with better values, perhaps it would have marched on to produce better results. Food for thought and experimentation.

Conclusion

We've done three fairly full estimations here. One with a `sigmoid` activation function and `50` epochs giving close to `96`% correct predictions; one with a `relu` activation function and `51` epochs giving close to `97`% correct predictions; and lastly one using the `tanh` activation (`93`%). Other variations are possible (e.g. alternative objective functions, more or larger layers) but, at the end of the day, I suspect that it's difficult to get much beyond `97`% with this model.

Perhaps the model is less than perfect? That's very likely and you could spend all day (or the rest of your life) varying the technique and the parameters of the estimation, but ultimately you will hit a limit. To go further you need a better model. Which takes us to convolutional networks [1].

References

[0] "*Machine Learning, An Interactive Approach*", M. Powell, https://aplwiki.com/images/1/16/3_Machine_Learning.pdf

[1] "*Convolutional Networks*", M. Powell, unpublished.

Mike Powell
Victoria, BC, Canada
April 2020