Machine Learning,
An Interactive Approach

Introduction

Machine learning is about estimating the parameters of a system. Very often, if you're studying this field, the theory is expressed in conventional mathematical notation and the computer execution is shown in Python.

At times, understanding the theory may be difficult and you'd like to get help from your computer. The snag is that you will need to translate ∑ summations, subscripts and ∂ derivatives into Python code. Not an insurmountable challenge, but certainly a distraction from the task at hand. That's problem number one.

Faced with something that's only partially understood, you may turn to other sources. Perhaps a different author can resolve the difficulty? But this is where the second problem arises: you're likely to be faced with different notation.

Isn't it better to have one notation, written and understood by all authors and directly (as in copy and paste) executable on a computer? That's exactly what APL offers.

This tutorial takes machine learning as its example. It explores the pattern recognition model for handwritten digits using data from the Modified National Institute of Standards and Technology database. This tutorial focuses on deriving the analytic gradients in APL and establishing them as part of the back propagation algorithm. A following tutorial [3] uses these results to conduct a full model estimation interactively.

The APL environment

Why APL? Because it is 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.

Shape analysis

It's useful to take an expression and work though its functional happenings to determine the shape of its result. Included here is an informal technique to record the steps in this process. It requires a convention.

> *If text is in red, it should be read (in APL) as "an object of shape ...".*

For example:

```
     3×2 3 4
2 3 4
```

A scalar multiplying a rank 3 object results in a rank 3 object of the same shape.

```
     (2 3 4⍴6){⍺∘.×⍵}⍤¯2⊢2 3 5 6⍴7
»    2 3 4{⍺∘.×⍵}⍤¯2⊢2 3 5 6
»    2 3,(4∘.×5 6)
2 3 4 5 6
```

Outer products in a rank 2 frame.

Definitions

This tutorial makes use of a number of 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
rnd←{⍺×⌊0.5+⍵÷⍺}                            ⍝ Round
image←{28 28⍴(⊂⍵>0.5)⌷' *'}                 ⍝ Display an image sample
disp←{⊂⍤2⊢⍵}                                ⍝ Display higher rank array
comp←{a b←⍷⍨⍵ ⋄ mean 0=0.99 1.01⍷a÷b+2×a×b=0} ⍝ Compare two similar values
timer←{ΔΔj←1⌷⎕ai ⋄ ΔΔk←⍎⍵ ⋄ 0.001×ΔΔj-⍨⍣¯1⌷⎕ai} ⍝ Execution time in seconds
cells←{⊂⍤⍺⊢⍵}                               ⍝ All the ⍺-cells of ⍵
id←{(⍵,⍵)⍴1,(×/⍵)⍴0}                        ⍝ Identity function
diag←{s←⍴⍵ ⋄ (s,s)⍴(,id s)\⍵}               ⍝ Diagonalize a vector
rscan←{1≡≢⍵:⍵ ⋄ (⊂(⊃⍵)⍺⍺⊃t),t←⍺⍺ 1↓⍵}       ⍝ Reverse scan
mp←{(,⍤⍺⍺⊢⍺)+.×,[⍺⍺↑⍳⍴⍴⍵]⍵}                ⍝ Extended matrix product
tip←{⍺ sop⍤((⍴⍴⍺)⌊⍴⍴⍵)⊢⍵}                  ⍝ Total inner product
sh←{(⍺⌽⍳⍴⍴⍵)⍉⍵}                             ⍝ Shift axes to the right
```

Random numbers

In various places, values are chosen from a uniform distribution. The seed for the random number generator is set initially and at several other places along the way. If you follow along, but repeat or omit some statements, you may find that your random values differ slightly.

Recognizing Handwritten Digits

The model discussed here uses the training and test data from the Modified National Institute of Standards and Technology ("MNIST") database (see Appendix C for a source of the data) to make predictions of handwritten digits `0-9` each provided as a `28` by `28` bitmap. Commonly, a prediction is made for each sample in the input, by applying several matrix transforms resulting in a vector with `10` values. The result is a bit like a probability distribution and the index of the largest value is taken to be the best guess for the input. Usually the transformations are performed with rank `2` matrices. The challenging task is the learning part, the estimation of the transformation matrices.

The images to be used are kept as a matrix, `TrainingImages`, of shape `784 60000` with each column holding the `28` by `28` pixel intensity values (floating point `0` to `1`) for a single image. The digit corresponding to each column of `TrainingImages` is kept as a boolean of length `10` in the corresponding column of `Training-Labels`.

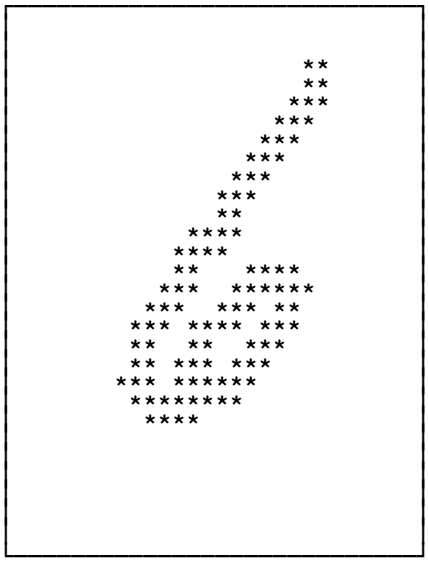
```
      ⍴¨TrainingImages TrainingLabels
```

784 60000	10 60000

```
      TrainingImages TestImages÷←255
```

Here's an image from this database.

```
      ⊂image 999[]⍤1⊢TrainingImages
```



The model

To get started, we'll use a model with 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 vector of length `10`.

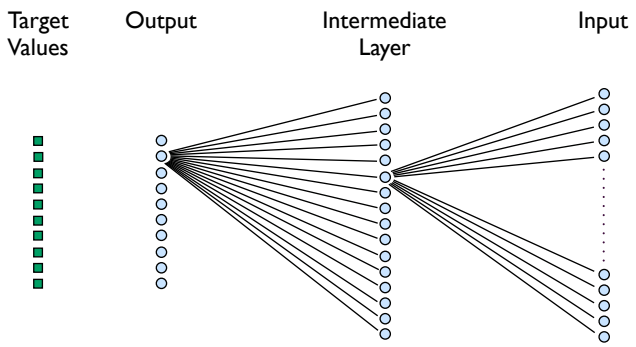


Figure 1

The transition between layers incorporates both biases and an activation function:

```
tf←{activate α+.×1,ω}
```
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`.

A commonly used activation function is the sigmoid function. (This is just one of several possibilities. Appendix B provides definitions and derivatives for nine of the most common.)

```
sigmoid←{÷1+*-ω}
```
$1 / 1 + e^{-x}$

```
dsigmoid←{{ω×1-ω}sigmoid ω}
```
It's derivative

The target is a boolean vector of length `10` with a single `1` marking the target digit.

Here's an example of the model acting on a single sample:

```
input←TrainingImages[;999]
target←TrainingLabels[;999]
activate←sigmoid
p0←?16 785⍴0
```
First transition matrix

```
p1←?10 17⍴0
```
Second transition matrix

```
⍝output←p1 tf p0 tf input
10
```

A simple way to record the structure of a network is to just record the number of items in each of the layers. For the network in Figure 1, this would be:

```
structure←10 16 784
```

Then we can generate initial values for the transition matrix parameters with:

```
    ρ¨p1 p0←(¯1↓structure){?(α,ω+1)ρ0}¨1↓structure
```

10 17	16 785

A convenient way to calculate the output is with the feed forward function:

```
    ff←{⊃tf/ω}
```
Feed forward

And, if we need to retain the values in all the layers, we can do so with:

```
    lff←{tf rscan ω}
```
Layered feed forward

```
    ρff p1 p0 input
10
    ρ¨lff p1 p0 input
```

10	16	784

This type of model is known as a model with fully connected layers.

Objective functions

Estimation of the model parameters requires the definition of an objective function and a variational procedure to iteratively adjust the parameters to minimize the value of that objective function. A number of objective functions are commonly in use. We'll consider the two most popular, least squares and cross entropy. Both are defined in terms of the `target` values and the feed forward output from the model:

The least squares objective is:

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

The cross entropy objective is:

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

For example:

```
    target←TrainingLabels[;999]
    input←TrainingImages[;999]
    (leastsquares,crossentropy)p1 p0
8.99637 77.8501
```

Measuring the closeness of fit

How do we measure the closeness of fit between the predicted values and the targets? Let's look at the sort of values we are working with so far:

```
    output←ff p1 p0 input
    2 10⍴target,output
0        0        0        0        0        0       1       0       0       0
0.999838 0.999861 0.999761 0.999946 0.999872 0.99975 0.99994 0.99965 0.99964 0.999868
```

The `target` values are a boolean vector with a single `1` positioned to indicate the digit. The `output` values have been coerced by the `activate` function to be between `0` and `1` but are all disappointingly close to the upper limit. The reason for this is that the matrix multiplication by `p1` or `p0` adds up lots of small contributions to make a large value. `activate` then scales this to be close to `1`.

It would be helpful to choose smaller initial values for `p1` and `p0`. There are many ways to do this but simply dividing by the number of elements contributing to each layer works quite well:

```
     p1 p0←(1+1↓structure)÷⍤(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
     output←ff p1 p0 input
     2 10⍴target,output
0        0        0        0        0        0        1        0        0        0
0.578667 0.576268 0.562872 0.57146 0.566322 0.554667 0.566124 0.574293 0.577999 0.561077
     0.01 rnd target-ff p1 p0 input
¯0.58 ¯0.58 ¯0.56 ¯0.57 ¯0.57 ¯0.55 0.43 ¯0.57 ¯0.58 ¯0.5
     (leastsquares,crossentropy)p1 p0
3.10566 8.1514
```

Adjusting the parameters

In order to improve the accuracy of a network's predictions we need to reduce the value of the objective function. Usually this means using the gradient of the objective function with respect to the parameters.

We're looking for the gradient of the objective function with respect to the two transition matrices `p1` and `p0`. This means that we must calculate gradients separately for `p1` and `p0` and then reassemble the results. We can do the calculations as a numeric approximation (using the `∆` operator defined below) with:

```
     ⍴g0←{obj p1 ⍵}∆ p0
16 785
     ⍴g1←{obj ⍵ p0}∆ p1
10 17
```

Once we have calculated the gradient, we are in a position to adjust the parameters. The technique we will use takes a step in the direction of the gradient in such a way that a reduction in the `obj` function is guaranteed. Here's how this works. Consider the first order Taylor expansion of a function `f` about a point `x`. The function `f` returns a scalar result for a vector argument and its derivative `f ∆ x` will return a vector of the same length as `x`. The value of `f` at a nearby point `x+dx`, where `dx` is small, is given by:

```
f x+dx  ↔  (f x) + dx+.×f ∆ x
```

If we choose a value for `dx`, small relative to `x`, and of the form `-k×f ∆ x`, with `k>0`, then the change in value of `f` is of the form `-k×v+.×v` which is guaranteed to be negative.

Numeric Approximation of the Gradient

Frames and cells

The result of a function `f` is, in general, made up of a frame and individual results. Let's label the shapes of these parts as `fr` and `sir`. If `f` is of rank `k`, we can determine `fr` and `sir` for an argument `x` as follows:

```
fr←⍴c←k cells x
sir←⍴⊃f⍤c
```

Let's follow this through with an example:

```
f←{+/⍵} ⋄ k←1 ⋄ x←?2 4 3⍴9
⍝c←k cells x                                    Break up x into cells of rank k
```

6 6 3	6 5 7	0 3 8	0 8 7
5 1 7	4 6 4	7 0 6	6 1 6

```
    f⍤c
15 18 11 15
13 14 13 13
```

The function derived by the derivative operator behaves in exactly the same way. It has the same rank as that of the original function. However, as it is a derivative, the individual results produced have more structure – the values being the sensitivity of the function's result to changes in the elements of the values in each `k`-cell. Let's use a different function `g` with the unchanged values for `x` and `c` to illustrate:

```
g←{⍵×+/⍵}
dg←{(⍵∘.+0×⍵)+(+/⍵)×⍳⍴⍵}                        dg is the derivative of g
k←1                                             g and dg are both rank 1 functions
⍝r←dg⍤c
```

21 6 6 6 21 6 3 3 18	24 6 6 5 23 5 7 7 25	11 0 0 3 14 3 8 8 19	15 0 0 8 23 8 7 7 22
18 5 5 1 14 1 7 7 20	18 4 4 6 20 6 4 4 18	20 7 7 0 13 0 6 6 19	19 6 6 1 14 1 6 6 19

The shape of each individual result is now a matrix. The shape of the overall result can be thought of as three pieces joined together: the shape of the frame, the shape of the function `g` applied to an individual cell and the shape of the individual cell itself. That is:

```
    ⍴⍤r
2 4 3 3
    (⍴c),(⍴g⊃c),⍴⊃c
2 4 3 3
```

The numeric derivative operator

An operator for the derivative, `∆`, is discussed in [0] and [1]. We'll use that definition here.

```
r←(f ∆)x;c;p;q;dx;d;j;n;sf;sx;t
 ⍝ Derivative of function f at x.
 ⍝ Assumes that the application of f to x does not produce a frame.
 ⍝ Coding comments:
 ⍝   Uses a loop to reduce memory usage.
 ⍝   Careless regard by f for the locals used here could be fatal.
 sf←⍴p←f x ⋄ sx←⍴x ⋄ dx←0.000001×{⍵+⍵=0}x←,x
 r←(×/sx,sf)⍴p ⋄ c←0 ⋄ j←0
 :While c<⍴dx
     q←x ⋄ d←c⌷dx ⋄ (c⌷q)+←d
     n←⍴t←,((f sx⍴q)-p)÷d ⋄ r[j+⍳n]←t
     c+←1 ⋄ j+←n
 :EndWhile
 r←(sx,sf)⍴r ⋄ r←((⍴sf)⍉⍳⍴⍴r)⍉r
```

The `∆` operator takes a function as its left argument, producing a derived function. The result is formed by applying the derived function to the right argument, which is expected to be an unboxed array.

This definition comes with a caveat. It expects to be used with arguments that do not exceed the rank of its function left argument. If a situation arises where the rank of the argument does exceed that of the function, then `∆` should be applied to cells of the argument with the rank `⍤` operator.

Performance of the numeric derivative ∆

We could dive in and attempt calculations for the gradients (`g0` and `g1`) using `∆` for a large number of samples. But this runs into a problem: there are many parameters. `p0` has `16×785` and `p1` has `10×17` – `12730` in all. The derivative with respect to `p0` is an array of shape `16 785`. Each element of this array requires, at minimum, one execution of the `ff` function. If `ff` takes any appreciable time to execute, and it does with `60000` training samples available, then the gradient calculation will be far too slow. Here's a timing of one execution of `ff` with the full set of training data (on a late 2014 iMac):

```
    timer'ff p1 p0 TrainingImages'
0.632
```

Even with APL's fast matrix operations and lots of memory, the calculation of a single gradient numerically with respect to `p0` using all of the training data is going to take something like `12730×0.632` seconds, more than two hours. And then we expect to include this in an iterative search algorithm. A most unpleasant prospect.

Fortunately, it's possible to simplify the differentiations analytically using the derivative rules [0] and this can reduce the computation time significantly.

The Analytic Gradients

The two layer model

Let's start looking at the simplest case. We'll assume that we're using a cross entropy objective function with just an input and output layer, a sigmoid activation function and a single sample.

```
      ⎕rl←16807
      obj←crossentropy ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
      structure←10 784
      pp←⊃(1+1↓structure)÷⍤(¯1↓structure){?(⍺,⍵+1)⍴0}⍤1⊢1↓structure
10 785
      batch←?60000
      pinput←TrainingImages[;batch]
784
      ptarget←TrainingLabels[;batch]
10
      obj⊂p
7.1514
```

The gradient

The derivative of the cross entropy objective function with respect to `p` is:

```
      g←{obj⊂⍵}∆ p
»     {-sum target∘cren ff ⍵ input}∆ p.................................................[0]
```

This is a derivative of a composition of the functions `{-sum ⍵}` and `{target∘cren ff ⍵ input}`. Writing `a←ff p input` this may be expanded as:

```
»     (-sum ∆ target∘cren a)+.×{target∘cren ff ⍵ input}∆ p
```

As the derivative of `sum` is `{(⍴⍵)⍴1}`, this becomes:

```
»     (-(⍴target)⍴1)+.×{target∘cren ff ⍵ input}∆ p
»     -+⌿{target∘cren ff ⍵ input}∆ p
```

Expanding this as the derivative of the composition of `{target∘cren ⍵}` and `{ff ⍵ input}`, we get:

```
»     (-+⌿(target∘cren ∆ a)+.×{ff ⍵ input}∆ p
»     (-+⌿({(target×⍤⍵)+(1-target)×⍤1-⍵}∆ a)+.×{ff ⍵ input}∆ p
```

We can simplify this by replacing `+⌿{(target×⍤⍵)+(1-target)×⍤1-⍵}∆ a` with its analytic derivative `(target-a)÷a×1-a`.

```
»     ((a-target)÷a×1-a)+.×{ff ⍵ input}∆ p                    Cross entropy
```

Note that, if we had chosen `leastsquares` as the objective, we would have arrived at a slightly different expression for the gradient.

```
      (2×a-target)+.×{ff ⍵ input}∆ p                          Least squares
```

With either objective, the left hand term is a constant. For simplicity let's call this `R` with the understanding that it has a different definition depending on the choice of objective.

```
»    R+.×{⍵ tf input}⍙ p ..................................................... [1]
»    R+.×{activate ⍵+.×1¯1,input}⍙ p
```

Expanding the derivative as a composition, with `z←p+.×1¯1,input`, we have:

```
»    R+.×(activate ⍙ z)+.×{⍵+.×1¯1,input}⍙ p
```

Changing the order of execution of the inner products:

```
»    (R+.×activate ⍙ z)+.×{⍵+.×1¯1,input}⍙ p
 10      10 10          10 10 785
```

As we have assumed rank `0` activation functions, the term `activate ⍙ z` for a vector argument `z` is a diagonal matrix. This makes it possible to replace `R+.×activate ⍙ z` with `R×dactivate z`:

```
»    (R×dactivate z)+.×{⍵+.×1¯1,input}⍙ p ..................................... [2]
 10     10           10 10 785
```

The right hand term `{⍵+.×1¯1,input}⍙ p` is the derivative of a matrix product of `{⍵}` and the constant `1¯1,input`. We can expand this with `(f+.×g)⍙ ↔ {¯2 sh(2 sh f ⍙ ⍵)+.×g ⍵}` giving:

```
»    (R×dactivate z)+.×¯2 sh(2 sh idρp)+.×1¯1,input
»    (R×dactivate z)+.×¯2 sh(idρp)+.×1¯1,input
```

The right hand term `¯2 sh(idρp)+.×1¯1,input` is `10 10 785`. Because of its size, it's a little hard to display. Let's work with a smaller example:

```
     a←1 4 6 ⋄ b←3 2 6 1
     disp ¯2 sh(id 3 4)+.×a
```

3 2 6 1	0 0 0 0	0 0 0 0
0 0 0 0	3 2 6 1	0 0 0 0
0 0 0 0	0 0 0 0	3 2 6 1

It's clear that effect of this expression is to produce `3` copies of a `3` by `4` array. Each copy has the right argument as a single row in the array with the rest of the elements being `0`. When this used in the inner product with `a`, it produces:

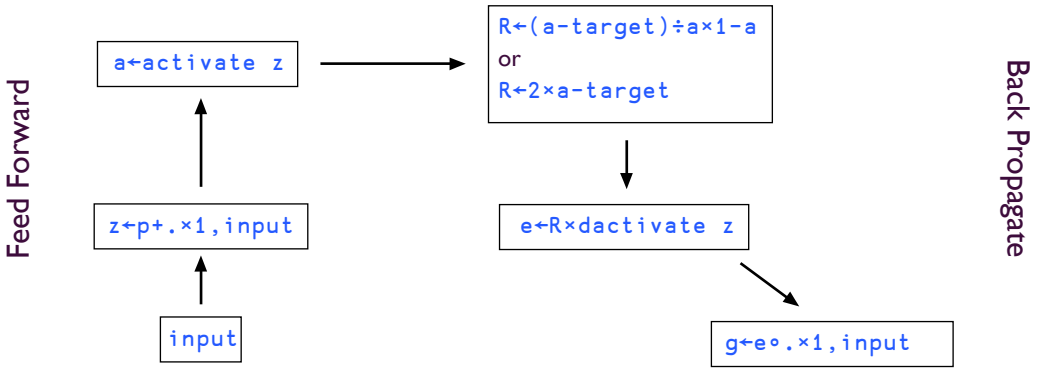
```
     a+.×¯2 sh(id 3 4)+.×b
 3  2  6 1
12  8 24 4
18 12 36 6
```

which is nothing more than `a∘.×b`. Using this, we have finally:

```
»    (R×dactivate z)∘.×1¯1,input ............................................. [3]
```

Back propagation

We can follow the path through these calculations, as follows:



On the left we have the feed forward section. The `input` is transformed by the parameter matrix (which includes biases) to produce *weighted inputs* (often labelled as *z*). Each *weighted input* is in turn transformed with the `activate` function to produce an *activation* (often labelled as *a*). For a vector `input`, the *weighted input* and *activation* are also vectors. If there are more layers, *a* and *z* are subscripted. The final *activation* value is the output of the model.

On the right is the back propagation section. This shows how the gradients are calculated from values already calculated during the feed forward section. The *error* term `e` is derived in part from the similar value one layer up the chain: `e` is derived from `R`. That's why this is called back propagation. Note that once the error value is known, calculation of the gradient `g` is straightforward.

Including a hidden layer

We can extend the model to include an additional layer with, for example:

```
    structure←10 16 784
    p¨p1 p0←(1+1↓structure)÷¨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
```

10 17	16 785

Now that we have three layers in total, there are two parameter matrices and two gradients. The output of the model is given by:

```
    pff p1 p0 input
10
```

The definitions of the objective functions `leastsquares` and `crossentropy` are unchanged:

```
    (leastsquares,crossentropy)p1 p0
3.06855 8.07564
```

The two gradients are now `{obj ⍵ p0}∆ p1` (the `p1` gradient) and `{obj p1 ⍵}∆ p0` (the `p0` gradient).

The `p1` *gradient*

The `p1` gradient is straightforward, requiring only a revision to the value for its input. It is:

```
    a1←p1 tf a0←p0 tf input
    R←(a1-target)÷a1×1-a1
    g1←{obj ⍵ p0}⍙ p1
»   (R×dactivate p1+.×1¯⍪a0)∘.×1¯⍪a0
»   (R×dactivate z1)∘.×1¯⍪a0                                with z1←p1+.×1¯⍪a0
```

The `p0` *gradient*

The derivative of the objective function with respect to `p0` is:

```
    g0←{obj p1 ⍵}⍙ p0
»   {-sum target∘cren ff p1 ⍵ input}⍙ p0.....................................[4]
```

The simplification of this equation first follows the steps used in equations `[0]` and `[1]` above:

```
»   R+.×{p1 tf ⍵ tf input}⍙ p0
    10       10 16 785
```

This is a derivative of a composition of `{p1 tf ⍵}` and `{⍵ tf input}`. The `{⍵ tf input}` function works with matrices, producing vectors for use by `{p1 tf ⍵}`. This means that `{p1 tf ⍵}` is of rank `1` and `{⍵ tf input}` is of rank `2`. The expression may be expanded with `(f g)⍙ ↔ (f ⍙ g)(1 mp)g ⍙`. We then have:

```
»   R+.×({p1 tf ⍵}⍙ a0)+.×{⍵ tf input}⍙ p0
»   R+.×({activate p1+.×1¯⍪⍵}⍙ a0)+.×{⍵ tf input}⍙ p0
    10           10 16                 16 16 785
```

Expanding the `{activate p1+.×1¯⍪⍵}⍙ a0` term as the derivative of the composition of `{activate ⍵}` and `{p1+.×1¯⍪⍵}`, we have:

```
»   R+.×(activate ⍙ z1)+.×({p1+.×1¯⍪⍵}⍙ a0)+.×{⍵ tf input}⍙ p0
    10       10 10             10 16            16 16 785
```

We can combine the first two terms with the same justification as was made for equation `[2]` above:

```
»   (R×dactivate z1)+.×({p1+.×1¯⍪⍵}⍙ a0)+.×{⍵ tf input}⍙ p0
         10                10 16            16 16 785
```

The middle term `{p1+.×1¯⍪⍵}⍙ a0` can be simplified by noting that it is equivalent to `{p[0;]+(0 1↓p1)+.×⍵}⍙ a0`, which just becomes `0 1↓p1`.

```
»   (R×dactivate z1)+.×(0 1↓p1)+.×{⍵ tf input}⍙ p0
         10             10 16        16 16 785
```

We've seen the right hand term before. It was dealt with in the discussion of equation `[2]` above. Drawing on those results, the derivative expression becomes, with `z0←p0+.×1¯⍪input` :

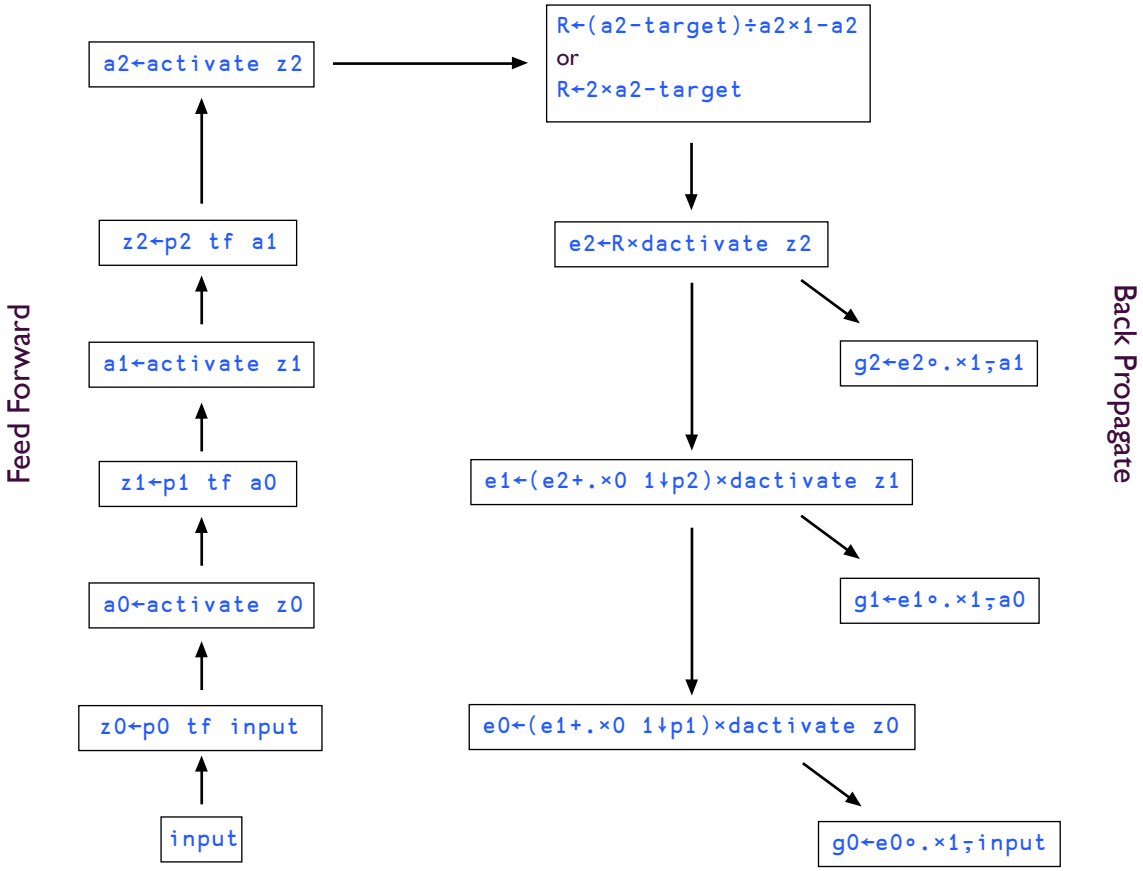
```
»   (((R×dactivate z1)+.×0 1↓p1)×dactivate z0)∘.×1¯⍪input ..................[5]
```

A second hidden layer

The model we've looked at so far has three layers, corresponding to two transition matrices, `p1` and `p0`. What would the gradients look like if we had a fourth layer with a new transition matrix `p2`? Without going through the analysis, here's what these three gradients are:

```
      structure←10 5 16 784
      p2 p1 p0←(1+1↓structure)÷⍤(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
      a0←activate z0←p0+.×1,⍪input
      a1←activate z1←p1+.×1,⍪a0
      a2←activate z2←p2+.×1,⍪a1
      R←(a2-target)÷a2×1-a2
      g0←{obj p2 p1 ⍵}∆ p0
»     ((((R×dactivate z2)+.×0 1↓p2)×dactivate z1)+.×0 1↓p1)×dactivate z0)∘.×1,⍪input ..... [6]
      g1←{obj p2 ⍵ p0}∆ p1
»     (((R×dactivate z2)+.×0 1↓p2)×dactivate z1)∘.×1,⍪a0 ............................... [7]
      g2←{obj ⍵ p1 p0}∆ p2
»     (R×dactivate z2)∘.×1,⍪a1 ..................................................... [8]
```

It helps to see the relationships between these equations in a diagram. The figure below draws on the description by Nielsen in [2] and shows how this works.



Samples in batches

At this point, we could compute values for the parameters that will improve the model's predictive ability. As we've got plenty of parameters to work with we should be able to eliminate the residuals all together. But this would only be for one sample. What we are looking for are values for the parameters that will do the best job making correct predictions with any `28` by `28` image. To do that, we need to estimate the parameters using all the information in the `TrainingImages` dataset.

One approach is simply to rely on the rank operator to extend the calculations for a vector `input` to a matrix of say `n` samples. The result for `g0` as an example might then be of shape `n,16 785`. But we still have to decide what to do with `n` values for each gradient. About the only reasonable answer to that question is to use the average. It turns out that the functions we have so far defined need need only slight changes to handle a batch of samples, without recourse to the rank operator. In particular:

```
n←8
batch←n?60000
input←TrainingImages[;batch]
target←TrainingLabels[;batch]
structure←10 5 16 784
p2 p1 p0←(1+1↓structure)÷⍤(¯1↓structure){?(⍺,⍵+1)⍴0}⍤1↓structure
a0←activate z0←p0+.×1,input
a1←activate z1←p1+.×1,a0
a2←activate z2←p2+.×1,a1
R←(a2-target)÷a2×1-a2
e2←R×dactivate z2
e1←((⍉0 1↓p2)+.×e2)×dactivate z1
e0←((⍉0 1↓p1)+.×e1)×dactivate z0
g2←e2+.×⍉1,a1
g1←e1+.×⍉1,a0
g0←e0+.×⍉1,input
g2 g1 g0÷←n
```

Both `input` and `target` have `n` as the last dimension

`e1` and `e0` are revised to handle multiple samples

The gradients are summed with an inner rather than an outer product

Different activation functions

So far, we've kept the activation function quite general. Where it appears, we've used `activate` and `dactivate` rather than referring to specific functions. Now's the time to introduce more of the commonly used activation functions. Here are nine:

Activation function	Definition	Derivative	Rank
Sigmoid	`sigmoid←{÷1+*-⍵}`	`dsigmoid←{{⍵×1-⍵}sigmoid ⍵}`	0
Hyperbolic Tangent	`tanh←{7○⍵}` or `{(*⍵){(⍺-⍵)÷⍺+⍵}*-⍵}`	`dtanh←{{(1+⍵)×1-⍵}tanh ⍵}`	
Softmax	`softmax←{(⊢÷sum)*⍵}`	`dsoftmax←{(diag-⊢∘.×⊢)softmax ⍵}`	1
Softsign	`softsign←{⍵÷1+|⍵}`	`dsoftsign←{{⍵×⍵}÷1+|⍵}`	0
Softplus	`softplus←{⍟1+*⍵}`	`dsoftplus←sigmoid`	0
Rectified Linear Unit	`relu←{0⌈⍵}`	`drelu←{⍵>0}`	0
Rectified Linear Unit 6	`relu6←{6⌊0⌈⍵}`	`drelu6←{(⍵>0)∧⍵≤6}`	0
Exponential Linear Unit	`elu←{(⍵×⍵>0)-(1-*⍵)×⍵≤0}`	`delu←{(*⍵)*⍵≤0}`	0
Leaky Rectified Linear Unit	`lrelu←{⍵×0.01*⍵≤0}`	`dlrelu←{0.01*⍵≤0}`	0

(Note that the derivations of the derivative expressions are in Appendix B.)

The equations we've constructed so far for the gradients have all assumed that the activation function is of rank `0`. This is important as it makes possible the nice simplification which we incorporated back at equation `[2]`. However, it does mean that the `softmax` activation doesn't fit this formulation. It requires some special attention and we'll put off dealing with that until another time.

Using an alternative activation function just requires setting `activate` and `dactivate` to their appropriate values. For example, to use the hyperbolic tangent activation:

```
activate←tanh ⋄ dactivate←dtanh
```

Performance

Of the three layer model

A very important reason for deriving the analytic gradients is to improve performance. Let's measure the improvement in the gradient calculation. We'll use a three layer model with a cross entropy objective and sigmoid activation.

```
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
```

```
    n←100
    batch←n?60000
    input←TrainingImages[;batch]
    target←TrainingLabels[;batch]
    structure←10 16 784
    p1 p0←(1+1↓structure)÷¨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
    obj←crossentropy ⋄ gr←gr_cren ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
    timer'a←n÷¨{obj ⍵ p0}∆ p1 ⋄ b←n÷¨{obj p1 ⍵}∆ p0'
6.6
    timer'g1 g0←gr p1 p0 target input'
0.001
    comp a g1
1
    comp b g0
0.922611
```

That's a speed up by a factor of perhaps `6600`.

(Note that there is a significant difference between `g0` and `b`. This is not a cause for alarm. A manual examination of these two gradients show them to be close enough to confirm that we're doing the right calculations. The difference is likely due to the errors accumulated in `b`, produced by the numeric gradient.)

With a larger batch size

Let's check and see how well the analytic gradient behaves with a larger batch size. For this test we'll use the entire `TrainingImages` dataset but will give up on calculating the numeric approximation to the gradient.

```
    structure←10 16 784
    p1 p0←(1+1↓structure)÷⍨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
    timer'g1 g0←gr p1 p0 TrainingLabels TrainingImages'
1.386
```

That's `1.4` seconds on a 2014 iMac to calculate gradients for both transition matrices with the entire `60000` training database. Earlier, we estimated that one calculation of the numeric gradient might take two hours with all of the training data. We're now able to get this done in `1.4` seconds – a speed up by a factor of about `5000`.

With a least squares objective

In order to use the least squares function as the objective, we need to make a slight adjustment to the gradient function `gr`. Recall that earlier we used a constant `R` understanding that it would have a different definition depending on the objective function selected. Now is the time to make sure that `R` gets the correct value.

```
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
```

As we'd expect, the execution time for the least squares gradient is much the same:

```
    structure←10 16 784
    p1 p0←(1+1↓structure)÷⍨(¯1↓structure){?(⍺,⍵+1)⍴0}¨1↓structure
    obj←leastsquares ⋄ gr←gr_lsq ⋄ activate←sigmoid ⋄ dactivate←dsigmoid
    timer'g1 g0←gr p1 p0 TrainingLabels TrainingImages'
1.187
```

Conclusion

This tutorial shows how the APL derivative rules can be applied to simplify the calculation of gradients used in machine learning. The simplification results in improved accuracy, simpler coding and a considerable improvement in performance, perhaps by a factor of `6600`.

The examples presented here are for networks of fully connected layers with a `leastsquares` or `crossentropy` objectives and rank `0` activation functions.

Most of the focus here is on the gradient function, which differs slightly depending on the choice of objective and activation functions. The tutorial points out where the variations lie, but makes no attempt to consolidate the variations into one set of code. There's plenty of time for that later.

The next tutorial shows how the gradient results can be used in an estimation procedure with the MNIST handwritten digits model.

Appendix A – Derivatives

The derivative rules

Name	Definition	Rule	Note
Taylor expansion	`f x+dx`	`(f x)+dx{⍺ tip f ⍙ ⍵}⍤rf⊢x`	`rf` is the rank of `f`
Sum	`(f+g)⍙`	`f ⍙ + g ⍙`	
Difference	`(f-g)⍙`	`f ⍙ - g ⍙`	
Product	`(f×g)⍙`	`(f xp g ⍙)+g xp f ⍙`	
Quotient	`(f÷g)⍙`	`((f ⍙)-(f÷g)xp g ⍙)xp(÷g)`	
Outer Product	`(f∘.×g)⍙`	`(f∘.×g ⍙)+order⍉f ⍙∘.×g`	
Composition	`(f g)⍙`	`(f ⍙ g)(n mp)g ⍙`	`m←⍴⍴g x`
Inverse	`fi ⍙`	`⌹(f ⍙ fi)`	
Matrix multiplication	`(f+.×g)⍙`	`(f+.×g ⍙)+(-n)sh(n sh f ⍙)+.×g`	`n←⍴⍴x`

Common derivatives

Name	Definition	Rank	Derivative
`sum`	`{+/,⍵}`	∞	`{(⍴⍵)⍴1}`
`num`	`{×/⍴⍵}`	∞	`{(⍴⍵)⍴0}`
`mean`	`{(sum÷num)⍵}`	∞	`{(⍴⍵)⍴÷num ⍵}`
`max`	`{⌈/,⍵}`	∞	`{⍵=max ⍵}`
`min`	`{⌊/,⍵}`	∞	`{⍵=min ⍵}`
`zeromax`	`{0⌈⍵}`	0	`(0<⍵}`
`zeromin`	`{0⌊⍵}`	0	`{0≥⍵}`
`exp`	`{*⍵}`	0	`{*⍵}`
`ln`	`{⍟⍵)`	0	`{÷⍵}`

Appendix B
Derivatives of the Activation Functions

Sigmoid

```
      f←{1} ⋄ g←{1+*-⍵}
      dsigmoid x                                  for scalar x as sigmoid is rank 0
»     (f÷g)⍙ x
»     (((f ⍙)-(f÷g)×g ⍙)×(÷g))x                   Quotient rule
»     ((0-(f÷g)×g ⍙)×(÷g))x                       As f ⍙ ↔ 0
»     ((-(sigmoid x)×g ⍙ x)×÷g x                  As f÷g ↔ sigmoid
»     ((-(sigmoid x)×-*-x)×sigmoid x              As g ⍙ x ↔ -*-x
»     (((sigmoid x)×*-x)×sigmoid x
»     (1-sigmoid x)×sigmoid x
»     {⍵×1-⍵}sigmoid x ..........................................................[9]
```

Hyperbolic Tangent

```
      f←{(*⍵)-*-⍵} ⋄ g←{(*⍵)+*-⍵}
      dtanh x                                     for scalar x as tanh is of rank 0
»     (f÷g)⍙ x
»     (((f ⍙)-(f÷g)×g ⍙)×(÷g))x                   Quotient rule
»     ((g-(f÷g)×f)÷g)x                            As f ⍙ ↔ g and g ⍙ ↔ f
»     (1-(f÷g)×f÷g)x
»     {(1+⍵)×1-⍵}tanh x...........................................................[10]
```

Softmax

```
      f←{*⍵} ⋄ g←{(⍴⍵)⍴+/f ⍵}
      dsoftmax x                                  for vector x as softmax is of rank 1
»     (f÷g)⍙ x
»     (((f ⍙)-(f÷g)⍤p g ⍙)⍤p(÷g))x                Quotient rule
»     ((f ⍙ x)⍤p(÷g x))-(((f x)÷g x)⍤p(2⍴⍴x)⍴f x)⍤p÷g x
»     (diag(f x)÷g x)                             As f ⍙ x ↔ diag f x
      -((f x)÷g x)⍤p(2⍴⍴x)⍴(f x)÷g x              and g ⍙ x ↔ (2⍴⍴x)⍴f x
»     (diag(f÷g)x)-((f÷g)x)⍤p(2⍴⍴x)⍴(f÷g)x
»     (diag(f÷g)x)-((f÷g)x)∘.×(f÷g)x              As a ⍤p(2⍴⍴a)⍴a ↔ a∘.×a
»     {(diag-⊢∘.×⊢)softmax ⍵}x ...................................................[11]
```

Softsign

```
    dsoftsign x                                          for scalar x as softsign is of rank 0
»   {ω÷1+|ω}∆ x
»   (x≤0)⌷({ω÷1+ω}∆ x),{ω÷1-ω}∆ x
»   (x≤0)⌷((1-(x÷1+x)×1)÷1+x),(1-(x÷1-x)×¯1)÷1-x          Quotient rule, twice
»   (x≤0)⌷(÷×⍨1+x),÷×⍨1-x
»   {ω×ω}÷1+|x                                                                    [12]
```

Softplus

```
    dsoftplus x                                          for scalar x as softplus is of rank 0
»   {⍟1+*ω}∆ x
»   (÷1+*x)××x                                          Composition of {⍟ω} and {1+*ω}
»   {÷1+*-ω}x
»   sigmoid x.....................................................................[13]
```

Rectified Linear Unit

```
    drelu x                                              for scalar x as relu is of rank 0
```

The `relu` function is the constant `0` for values `≤0` and a line of unit slope for positive values. Thus, it's derivative is:

```
»   {0<ω}x........................................................................[14]
```

Rectified Linear Unit 6

```
    drelu6 x                                             for scalar x as relu6 is of rank 0
```

The `relu6` function is the constant `0` for values `≤0` or `>6` and a line of unit slope in between. Thus, it's derivative is:

```
»   {(ω>0)∧ω≤6}x..................................................................[15]
```

Exponential Linear Unit

```
    delu x                                               for scalar x as elu is of rank 0
```

The `elu` function is `(*x)-1` for values `≤0` and a line of unit slope for positive values. Thus, it's derivative is:

```
»   {(*ω)*ω≤0}x...................................................................[16]
```

Leaky Rectified Linear Unit

```
    dlrelu x                                             for scalar x as lrelu is of rank 0
```

The `lrelu` function has two straight lines meeting at `x=0`. The gradient is `0.01` for negative values and `1` for positive values. Thus, it's derivative is:

```
»    {0.01*ω≤0}x..................................................................[17]
```

Appendix C – MNIST data

The MNIST database

The MNIST database of handwritten digits (available from http://yann.lecun.com/exdb/mnist) is a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centred in a fixed-size image.

If you download the four .gz files and unzip them (which may happen automatically) you should get the following files:

```
t10k-images.idx3-ubyte                    7,840,016 Bytes
t10k-labels.idx1-ubyte                    10,008 Bytes
train-images.idx3-ubyte                   47,040,016 Bytes
train-labels.idx1-ubyte                   60,008 Bytes
```

Both the training and test data come with two files each. One of the files has image representations as `28` by `28` arrays of grayscale integers between `0` (white) and `255` (black); the other has labels corresponding to their values, `0` to `9`. Details of the file formats are provided below.

Here's a function to read these files:

```
read←{tn←⍵ ⎕NTIE 0 ⋄ d←⎕NREAD tn,11,2↑⎕NSIZE tn ⋄ tn←⎕NUNTIE tn
      m n←2⊥⍣¯2 32⍴d
      m=2051:2⊥1 2 3 0⍉(n,(2⊥32⍴64↓d),(2⊥32⍴96↓d),8)⍴128↓d
      m=2049:2⊥⍉(n,8)⍴64↓d}
    folder←'./NMIST Data/'
    ⍴TestImages←⍉(10000 784⍴read folder,'t10k-images.idx3-ubyte')÷255
784 10000
    ⍴TestLabels←(⍳10)∘.=read folder,'t10k-labels.idx1-ubyte'
10 10000
    ⍴TrainingImages←⍉(60000 784⍴read folder,'train-images.idx3-ubyte')÷255
784 60000
    ⍴TrainingLabels←(⍳10)∘.=read folder,'train-labels.idx1-ubyte'
10 60000
```

File formats

Training labels (`train-labels-idx1-ubyte`):

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
```

Training images (`train-images-idx3-ubyte`):

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

Test labels (`t10k-labels-idx1-ubyte`):

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  10000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
```

Test images (`t10k-images-idx3-ubyte`):

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  10000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black). The labels values are 0 to 9.

References

[0] "*The Derivative Rules*", M. Powell, https://aplwiki.com/images/f/f8/2_The_Derivative_Rules.pdf

[1] "*The Derivative Revisited*", M. Powell, https://aplwiki.com/images/f/f9/1_The_Derivative_Revisited.pdf

[2] "*Neural Networks and Deep Learning*", Michael Nielsen, Determination Press, 2015.

[3] "*The Handwritten Digits Model*", M. Powell, https://aplwiki.com/images/f/fa/4_The_Handwritten_Digits_Model.pdf

Mike Powell
Victoria, BC, Canada
April 2020