2-Layer Neural Network

Genji Ohara

July 4, 2023

1 Preamble

```
module NeuralNetwork where

import Numeric.LinearAlgebra
import Prelude hiding ((<>))

type Vec = Vector R
type Mat = Matrix R
```

2 Constants

```
inputSize   :: Int
hiddenSize  :: Int
outputSize  :: Int
inputSize   = 784
hiddenSize  = 50
outputSize  = 10

w1_start    :: Int
w1_size     :: Int
w2_start    :: Int
w2_size     :: Int
b2_start    :: Int
weight_size :: Int
w1_start    = 0
w1_size     = inputSize * hiddenSize
w2_start    = w1_size + hiddenSize
w2_size     = hiddenSize * outputSize
b2_start    = w2_start + w2_size
weight_size = w1_size + hiddenSize + w2_size + outputSize
```

3 Layers

3.1 Affine

3.1.1 forward

```
affine :: Mat → Vec → Mat → Mat
affine w b x = x <> w + asRow b

affineDX :: Mat → Mat → Mat
affineDX w dout = dout <> (tr w)

affineDW :: Mat → Mat → [R]
affineDW x dout = (matToList $ (tr x) <> dout) ++ (toList $ sum $
    toRows dout)
```

3.2 Activation Function

3.2.1 ReLU

$$\mathrm{ReLU}(x) = \max(x, 0)$$

$$\mathrm{ReLU}(X) = \begin{bmatrix} \mathrm{ReLU}(x_{11}) & \cdots & \mathrm{ReLU}(x_{1N}) \\ \vdots & \ddots & \vdots \\ \mathrm{ReLU}(x_{N1}) & \cdots & \mathrm{ReLU}(x_{NN}) \end{bmatrix}$$

```
relu :: Mat → Mat
relu = cmap (max 0)

reluBackward :: Mat → Mat → Mat
reluBackward dout x = dout * mask
    where mask = cmap (λ_x → if _x > 0 then 1 else 0) x
```

3.2.2 Sigmoid

$$\mathrm{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

```
sigmoid :: R → R
sigmoid x = 1 / (1 + exp(-x))
```

3.3 Cross Entropy Error

$$\mathrm{CEE}(\boldsymbol{y}, \boldsymbol{t}) = -\boldsymbol{t}^T \begin{bmatrix} \ln y_1 \\ \vdots \\ \ln y_D \end{bmatrix}$$

$$\mathrm{CEE}(Y, T) = \sum_{i=1}^{N} \mathrm{CEE}(\boldsymbol{y}_i, \boldsymbol{t}_i)$$

```
sumCrossEntropyError :: [Vec] → [Vec] → R
sumCrossEntropyError [] _ = 0
sumCrossEntropyError _ [] = 0
sumCrossEntropyError (y:ys) (t:ts) = -t <.> (cmap log y) +
    sumCrossEntropyError ys ts

crossEntropyError :: Mat → Mat → R
crossEntropyError y t = sumCrossEntropyError ys ts / batchSize
    where
        ys = toRows y
        ts = toRows t
        batchSize = fromIntegral $ length ys
```

3.4 Softmax

3.4.1 Softmax

$$\exp(\boldsymbol{x}) = \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_N} \end{bmatrix}$$

$$\mathrm{softmax}(\boldsymbol{x}) = \frac{\exp(\boldsymbol{x})}{\|\exp(\boldsymbol{x})\|_1} = \frac{\exp(\boldsymbol{x} - \boldsymbol{c})}{\|\exp(\boldsymbol{x} - \boldsymbol{c})\|_1}$$

$$\mathrm{softmax}(X) = \begin{bmatrix} \mathrm{softmax}(\boldsymbol{x}_{:1}) & \cdots & \mathrm{softmax}(\boldsymbol{x}_{:N}) \end{bmatrix}$$

```
softmaxVec :: Vec → Vec
softmaxVec xVec = scale (1 / norm_1 expVec) expVec
    where
        c       = maxElement xVec
        cVec    = vector $ take (size xVec) [c,c..]
        expVec  = cmap exp $ xVec - cVec

softmax :: Mat → Mat
softmax = fromRows ∘ (map softmaxVec) ∘ toRows
```

3.4.2 Softmax with Loss

```
softmaxWithLoss :: Mat → Mat → R
softmaxWithLoss x t = crossEntropyError (softmax x) t

softmaxWithLossBackward :: Mat → Mat → Mat
softmaxWithLossBackward y t = (y - t) / (scalar $ fromIntegral $
    rows y)
```

3.5 Loss Function

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{w}; X, T) &= \mathrm{softmaxWithLoss}(\hat{Y}, T) \\
&= \mathrm{CEE}(\mathrm{softmax}(\hat{Y}), T)
\end{aligned}
$$

```
loss :: Vec → Mat → Mat → R
loss w x t = softmaxWithLoss (forwardProp w x) t
```

3.6 One-Hot Vector

```
oneHotList :: Int → Int → [R]
oneHotList len idx =
    if len == 0
        then []
        else
            if idx == 0
                then 1 : oneHotList (len - 1) (idx - 1)
                else 0 : oneHotList (len - 1) (idx - 1)

oneHotVector :: Int → Int → Vec
oneHotVector len idx = vector $ oneHotList len idx

oneHotMat :: Int → [Int] → Mat
oneHotMat len labelList = fromRows $ map (oneHotVector len)
    labelList
```

3.7 Forward Propagetion

```
-- 出力層の活性化関数(softmax)を適用する直前まで計算
forwardProp :: Vec → Mat → Mat
forwardProp weight x = affine w2 b2 $ relu $ affine w1 b1 x
```

```
    where
        w1 = reshape hiddenSize $ subVector w1_start w1_size weight
            :: Mat
        w2 = reshape outputSize $ subVector w2_start w2_size weight
            :: Mat
        b1 = subVector w1_size  hiddenSize weight
        b2 = subVector b2_start outputSize weight
```

3.8 Prediction

```
predict :: Vec → Mat → Mat
predict w x = oneHotMat outputSize $ map maxIndex $ toRows $
    forwardProp w x
```

3.9 Gradient

```
numericalGradientList :: Int → (Vec → R) → Vec → [R]
numericalGradientList idx f x =
    if idx == size x
        then []
        else
            let h  = 1e-4
                dx = cmap (* h) $ oneHotVector (size x) idx
                x1 = x + dx
                x2 = x - dx
            in (f(x1) - f(x2)) / (2 * h) : numericalGradientList (
                idx + 1) f x

numericalGradient :: (Vec → R) → Vec → Vec
numericalGradient f = vector ∘ (numericalGradientList 0 f)

matToList :: Mat → [R]
matToList = concat ∘ toLists

gradient :: Vec → Mat → Mat → Vec
gradient weight x t =

    let w1 = reshape hiddenSize $ subVector w1_start w1_size weight
        :: Mat
        w2 = reshape outputSize $ subVector w2_start w2_size weight
            :: Mat
        b1 = subVector w1_size  hiddenSize weight
        b2 = subVector b2_start outputSize weight

        -- forward propagation
        a1 = affine w1 b1 x
        y1 = relu a1
```

```
        y2 = softmax $ affine w2 b2 y1

        -- backward propagation
        da2 = softmaxWithLossBackward y2 t
        dx2 = affineDX w2 da2
        dw2 = affineDW y1 da2
        da1 = reluBackward dx2 a1
        dw1 = affineDW x da1

    in fromList $ dw1 ++ dw2
--

gradientCheck :: Vec → Mat → Mat → R
gradientCheck w x t =
    let num_grad    = numericalGradient (λ_w → loss _w x t) w
        grad        = gradient w x t
        err_sum     = sum $ map abs $ toList $ num_grad - grad
    in err_sum / (fromIntegral $ length $ toList grad)
```

3.10 Learning

```
learn :: Vec → Mat → Mat → R → R → Vec
learn weight x t learningRate iterNum =
    if iterNum == 0
        then weight
        else learn new_w x t learningRate (iterNum - 1)
            where
                new_w = weight - (cmap (* learningRate) $ gradient
                    weight x t)

testAccuracy :: Vec → Mat → Mat → R
testAccuracy w x t = scoreSum / (fromIntegral $ rows x)
    where scoreSum = sumElements $ takeDiag $ (predict w x) <> (tr t
        )
```