Haskell-ML

Genji Ohara

March 20, 2024

Contents

Chapter 1

Introduction

1.1 About This Book

This book is a collection of Haskell code for machine learning. This PDF file is generated from haskell-ml.lhs written in Literate Haskell format. You can compile it as both Haskell and LaTeX source code. I write this book to learn Haskell and machine learning and hope it will be helpful for those who have the same interest.

1.2 Prerequisites

We use the following libraries:

- `Prelude` for basic functions
- `Numeric.LinearAlgebra` for matrix operations
- `Data.CSV` for reading CSV files
- `Text.ParserCombinators.Parsec` for parsing CSV files
- `System.Random` for random number generation
- `Data.List` for list operations

```haskell
import Prelude hiding ((<>))
import Numeric.LinearAlgebra
import Data.CSV
import Text.ParserCombinators.Parsec
import System.Random
import Data.List
```

We use the following type aliases:

- `R` for `Double`
- `Vec` for `Vector R`
- `Mat` for `Matrix R`

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

We define the some spaces as follows:

Feature Space	$\mathcal{F} = \mathbb{R}^D$
Label Space	$\mathcal{L} = \{0, 1, \ldots, L-1\}$
Data Space	$\mathcal{D} = \mathcal{F} \times \mathcal{L}$

```haskell
type Feature    = [Double]
type Label      = Int
data DataPoint = DataPoint {
    dFeature :: Feature,
    dLabel   :: Label
} deriving Show
data RegDataPoint = RegDataPoint {
    rdFeature :: Feature,
    rdLabel   :: Double
} deriving Show
```

1.3 Entry Point

You can test all methods in this book by compiling haskell-ml.lhs as a Haskell source code.

```haskell
main :: IO()
main = do
    testDT
    testLinReg
    testNN
```

1.4 Data Processing

1.4.1 Read Data

We need to read external datasets for input to models.

```
type DataSet    = [DataPoint]
type RegDataSet = [RegDataPoint]

readClsDataFromCSV :: String -> IO DataSet
readClsDataFromCSV fileName = do
    rawData <- parseFromFile csvFile fileName
    return $ either (\_ -> []) processClsData rawData

readRegDataFromCSV :: String -> IO RegDataSet
readRegDataFromCSV fileName = do
    rawData <- parseFromFile csvFile fileName
    return $ either (\_ -> []) processRegData rawData
```

1.4.2 Process Data

We need following steps to process data:

$$
\begin{tikzcd}
 & \texttt{rawData} \arrow[dl, "\texttt{init}"'] \arrow[dr, "\texttt{last.transpose}"] & \\
\texttt{rawFeat} \arrow[d, "\texttt{map (map read)}"'] & & \texttt{rawLabs} \arrow[d, "\texttt{strLabelToIntLabel}"] \\
\texttt{feats} \arrow[dr] & & \texttt{labs} \arrow[dl] \\
 & \text{Processed Data} &
\end{tikzcd}
$$

concatDataPoint

```
processClsData :: [[String]] -> [DataPoint]
processClsData rawData = concatClsDataPoint feats labs
    where
        rawLabs = (last . transpose) rawData
        feats   = map (map (read :: String -> Double) . init) $ rawData
        labs    = strLabelToIntLabel rawLabs

processRegData :: [[String]] -> [RegDataPoint]
processRegData rawData = concatRegDataPoint feats labs
    where
        rawLabs = (last . transpose) rawData
        feats   = map (map (read :: String -> R) . init) $ rawData
        labs    = map (read :: String -> R) rawLabs

strLabelToIntLabel :: [String] -> [Int]
strLabelToIntLabel strLabels = map (maybeToInt . labelToIndex) strLabels
    where
```

```
        labelToIndex l = findIndex (l ==) $ nub strLabels
        maybeToInt Nothing = 0
        maybeToInt (Just a) = a

concatClsDataPoint :: [[Double]] -> [Int] -> [DataPoint]
concatClsDataPoint (f:fs) (l:ls) = DataPoint f l : concatClsDataPoint fs ls
concatClsDataPoint [] _ = []
concatClsDataPoint _ [] = []

concatRegDataPoint :: [[Double]] -> [Double] -> [RegDataPoint]
concatRegDataPoint (f:fs) (l:ls) = RegDataPoint f l : concatRegDataPoint fs ls
concatRegDataPoint [] _ = []
concatRegDataPoint _ [] = []
```

1.4.3 Split Data

We need to split the dataset into training and test datasets.

```
splitDataset :: DataSet -> R -> (DataSet, DataSet)
splitDataset dataSet rate = (trainData, testData)
    where
        trainData = take (round $ rate * fromIntegral (length dataSet)) dataSet
        testData  = drop (round $ rate * fromIntegral (length dataSet)) dataSet

splitRegDataset :: RegDataSet -> R -> (RegDataSet, RegDataSet)
splitRegDataset dataSet rate = (trainData, testData)
    where
        trainData = take (round $ rate * fromIntegral (length dataSet)) dataSet
        testData  = drop (round $ rate * fromIntegral (length dataSet)) dataSet
```

1.5 Some Utilities

```
listToString :: [R] -> String
listToString [] = ""
listToString (r:rs) = show r ++ " " ++ listToString rs

vecToString :: Vec -> String
vecToString = listToString . toList

vecsToString :: [Vec] -> String
vecsToString [] = ""
vecsToString (r:rs) = (vecToString r) ++ "\n" ++ (vecsToString rs)

```

```
matToString :: Mat -> String
matToString = vecsToString . toRows

concatMatAndVec :: Mat -> Vec -> Mat
concatMatAndVec x v = fromColumns $ toColumns x ++ [v]
```

Chapter 2

Linear Model

2.1 Linear Regression

Linear regression is a very simple classifier.

2.1.1 Setting

Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$, where $\boldsymbol{x}_i \in \mathbb{R}^D$ is a feature vector and $y_i \in \{0, 1\}$ is a label,

$$\boldsymbol{X} \triangleq \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \quad \boldsymbol{y} \triangleq \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.1}$$

2.1.2 Model

We get the estimated label $\hat{y}$ from the feature vector $\boldsymbol{x}$ as follows:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{2.2}$$

We transform eq. (2.2) by adding a bias term:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + w_0 = \begin{bmatrix} w_0 & \boldsymbol{w}^T \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{x}}. \tag{2.3}$$

```
predictLinReg :: Vec -> Vec -> R
predictLinReg tw x = tw <.> (vector $ [1.0] ++ toList x)

predictLinRegMat :: Vec -> Mat -> Vec
predictLinRegMat tw x = fromList $ map (predictLinReg tw) $ toRows x
```

2.1.3 Problem

We want to find the weight $\tilde{\boldsymbol{w}}$ that minimizes the objective:

$$E(\tilde{\boldsymbol{w}}) = \|\boldsymbol{y} - \tilde{\boldsymbol{X}}\tilde{\boldsymbol{w}}\|^2 + \lambda\|\tilde{\boldsymbol{w}}\|^2. \tag{2.4}$$

where

$$\tilde{\boldsymbol{X}} \triangleq \begin{bmatrix} \tilde{\boldsymbol{x}}_1^T \\ \tilde{\boldsymbol{x}}_2^T \\ \vdots \\ \tilde{\boldsymbol{x}}_N^T \end{bmatrix} = \begin{bmatrix} 1 & \\ 1 & \\ \vdots & \boldsymbol{X} \\ 1 & \end{bmatrix} \tag{2.5}$$

```
addBias :: Mat -> Mat
addBias x = fromColumns $ [bias] ++ (toColumns x)
    where bias = vector $ take (rows x) [1,1..]
```

2.1.4 Fitting

Gradient of the objective eq. (2.4) is

$$\nabla E(\tilde{\boldsymbol{w}}) = 2\left[\left(\tilde{\boldsymbol{X}}^T\tilde{\boldsymbol{X}} + \lambda I\right)\tilde{\boldsymbol{w}} - \tilde{\boldsymbol{X}}^T\boldsymbol{y}\right]. \tag{2.6}$$

Therefore

$$\underset{\tilde{\boldsymbol{w}}}{\operatorname{argmin}}\; E(\tilde{\boldsymbol{w}}) = \left(\tilde{\boldsymbol{X}}^T\tilde{\boldsymbol{X}} + \lambda I\right)^{-1}\tilde{\boldsymbol{X}}^T\boldsymbol{y} \tag{2.7}$$

```
fit :: Mat -> Vec -> R -> Vec
fit x_til y lambda = (inv a) #> ((tr x_til) #> y)
  where a = (tr x_til) <> x_til + (scale lambda $ ident $ cols x_til)
```

2.1.5 Test

We use iris dataset for testing.

```
testLinReg :: IO()
testLinReg = do
    putStrLn "Linear Regression"
    dataSet <- readRegDataFromCSV "data/housing.csv"
    let splittedData = splitRegDataset dataSet 0.8
    let trainData = fst splittedData
    let testData = snd splittedData
    let x = fromRows $ map (vector . rdFeature) trainData
    let y = vector $ map rdLabel trainData
    let x_til = addBias x
```

```
    let w = fit x_til y 0.1
    let x_test = fromRows $ map (vector . rdFeature) testData
    let y_test = vector $ map rdLabel testData
    let d_y = y_test - (predictLinRegMat w x_test)
    let mse = (d_y <.> d_y) / (fromIntegral $ rows x_test)
    print mse
    writeFile "output/linreg.dat" $ show mse
```

Output:

```
31.971956716480754
```

2.2 Logistic Regression

2.2.1 Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.8}$$

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

2.2.2 Prediction

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{x}) \tag{2.9}$$

```
predictLogReg :: Vec -> Vec -> R
predictLogReg w x = sigmoid $ w <.> x
```

2.2.3 Fitting

We minimize the objective:

$$E(\boldsymbol{w}) = -\sum_{i=1}^{N} [t_i \ln \hat{y}_i + (1-t_i)\ln(1-\hat{y}_i)] + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{2.10}$$

Gradient:

$$\nabla E(\boldsymbol{w}) = \boldsymbol{X}^T(\hat{\boldsymbol{y}} - \boldsymbol{t}) + \lambda \boldsymbol{w} \tag{2.11}$$

```
-- lossLogReg :: Vec -> Vec -> Vec -> R -> R
-- lossLogReg w x t lambda = sumCrossEntropyError ys ts + lambda * (norm_2 w) ^ 2
--     where
--         ys = map (predictLogReg w) $ toRows x
```

```
--         ts = toList t

gradientLogReg :: Vec -> Mat -> Vec -> R -> Vec
gradientLogReg w x t lambda = (tr x) #> (ys - t) + scale lambda w
    where
        ys = fromList $ map (predictLogReg w) $ toRows x
```

2.2.4 Stochastic Gradient Descent

We update the weight as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla E(\boldsymbol{w}) \tag{2.12}$$

By iterating eq. (2.12), we can minimize the objective eq. (2.10).

```
sgd :: Vec -> Mat -> Vec -> R -> R -> Vec
sgd weight x t learningRate iterNum =
    if iterNum == 0
    then weight
    else sgd new_w x t learningRate (iterNum - 1)
    where
    new_w = weight - (cmap (* learningRate) $ gradientLogReg weight x t 0.1)
```

2.2.5 Test

Chapter 3

Tree Model

3.1 Decision Tree

3.1.1 Constants

```
featureNum :: Int
featureNum = 4

labelNum :: Int
labelNum = 3
```

3.1.2 Tree Structure

Literal

```
data Literal = Literal Int Double
-- data Literal = Literal {
    --      lFeatureIdx :: Int,
    --      lValue :: Double
    -- }

instance Show Literal where
    show (Literal i v) = "Feature[" ++ (show i) ++ "] < " ++ (show v)
```

Split

```
data Split = Split {
    sLiteral :: Literal,
    sScore :: Double
  } deriving Show
```

```

instance Eq Split where
    (Split _ s) == (Split _ s') = s == s'

instance Ord Split where
    compare (Split _ s) (Split _ s') = compare s s'
```

Tree

```
data Tree = Leaf Int String | Node Literal Tree Tree String
-- data Tree = Leaf {label :: Int, id :: String} |
--              Node {literal :: Literal, left :: Tree, right :: Tree, id :: String}
```

3.1.3 Output Tree

```
instance Show Tree where
    show tree = treeToString tree 0

treeToString :: Tree -> Int -> String
treeToString (Leaf l _) depth =
    branchToString depth ++ "class: " ++ (show l) ++ "\n"
treeToString (Node literal leftTree rightTree _) depth =
    let str1 = branchToString depth ++ show literal ++ "\n"
        str2 = treeToString leftTree (depth + 1)
        str3 = branchToString depth ++ "!" ++ show literal ++ "\n"
        str4 = treeToString rightTree $ depth + 1
    in str1 ++ str2 ++ str3 ++ str4

branchToString :: Int -> String
branchToString depth = "|" ++ (concat $ replicate depth "   |") ++ "--- "
```

Listing 3.1: Example of CLI output

```
|--- Feature[2] < 1.9
|   |--- class: 0
|--- !Feature[2] < 1.9
|   |--- Feature[3] < 1.7
|   |   |--- Feature[2] < 4.9
|   |   |   |--- Feature[3] < 1.6
|   |   |   |   |--- class: 1
|   |   |   |--- !Feature[3] < 1.6
|   |   |   |   |--- class: 2
|   |   |--- !Feature[2] < 4.9
```

```
|    |    |    |--- Feature[3] < 1.5
|    |    |    |    |--- class: 2
|    |    |    |--- !Feature[3] < 1.5
|    |    |    |    |--- Feature[0] < 6.7
|    |    |    |    |    |--- class: 1
|    |    |    |    |--- !Feature[0] < 6.7
|    |    |    |    |    |--- class: 2
|    |--- !Feature[3] < 1.7
|    |    |--- Feature[2] < 4.8
|    |    |    |--- Feature[0] < 5.9
|    |    |    |    |--- class: 1
|    |    |    |--- !Feature[0] < 5.9
|    |    |    |    |--- class: 2
|    |    |--- !Feature[2] < 4.8
|    |    |    |--- class: 2
```

3.1.4 Gini Impurity

Class Ratio

Label Set	$L = \{y \mid (\boldsymbol{x}, y) \in D\}$	
Label Count	$c_l(L) = \sum_{i \in L} \mathbb{I}[i = l],$	$\boldsymbol{c}(L) = \sum_{i \in L} \mathrm{onehot}(i)$
Class Ratio	$p_l(L) = \frac{c_l(L)}{\lvert L \rvert},$	$\boldsymbol{p}(L) = \frac{\boldsymbol{c}(L)}{\lVert \boldsymbol{c}(L) \rVert_1}$

```haskell
labelCount :: [Label] -> Vec
labelCount = sum . (map $ oneHotVector labelNum)

classRatio :: [Label] -> Vec
classRatio labelList = scale (1 / (norm_1 countVec)) $ countVec
    where countVec = labelCount labelList
```

Gini Impurity

$$\mathrm{Gini}(L) = 1 - \sum_{l=0}^{L-1} p_l(L)^2 = 1 - \lVert \boldsymbol{p}(L) \rVert_2^2$$

```haskell
gini :: [Label] -> Double
gini labelList = 1.0 - (norm_2 $ classRatio labelList) ^ 2
```

3.1.5 Search Best Split

Split Data

$$D_l(D, i, v) = \{(\boldsymbol{x}, y) \in D \mid x_i < v\}$$
$$D_r(D, i, v) = \{(\boldsymbol{x}, y) \in D \mid x_i \geq v\}$$

```
splitData :: DataSet -> Literal -> [DataSet]
splitData dataSet (Literal i v) = [lData, rData]
    where
        lData = [(DataPoint x y) | (DataPoint x y) <- dataSet, x !! i <= v]
        rData = [(DataPoint x y) | (DataPoint x y) <- dataSet, x !! i >  v]
```

Score Splitted Data

$$\mathrm{score}(D, i, v) = \frac{|D_l|}{|D|}\mathrm{gini}\left[D_l(D, i, v)\right] + \frac{|D_r|}{|D|}\mathrm{gini}\left[D_r(D, i, v)\right]$$

```
scoreLiteral :: DataSet -> Literal -> Split
scoreLiteral dataSet literal = Split literal score
    where
        score = sum $ map (weightedGini (length dataSet)) $ labelSet
        labelSet = map (map dLabel) $ splitData dataSet literal

weightedGini :: Int -> [Label] -> Double
weightedGini wholeSize labelSet = (gini labelSet) * dblDataSize / dblWholeSize
  where
    dblDataSize     = fromIntegral $ length labelSet
    dblWholeSize    = fromIntegral wholeSize
```

Search Best Split

$$\underset{i,v}{\mathrm{argmin}}\,\mathrm{score}(D, i, v)$$

```
bestSplitAtFeature :: DataSet -> Int -> Split
bestSplitAtFeature dataSet i = myMin splitList
    where
        splitList   = [scoreLiteral dataSet l | l <- literalList]
        literalList = [Literal i (x !! i) | (DataPoint x _) <- dataSet]

bestSplit :: DataSet -> Split
bestSplit dataSet = myMin splitList
    where splitList = [bestSplitAtFeature dataSet f | f <- [0,1..featureNum-1]]
```

3.1.6 Grow Tree

Grow Tree

```haskell
growTree :: DataSet -> Int -> Int -> String -> Tree
growTree dataSet depth maxDepth nodeId =
    if stopGrowing
    then Leaf (majorLabel dataSet) nodeId
    else Node literal leftTree rightTree nodeId
    where
        literal         = sLiteral $ bestSplit dataSet
        leftTree        = growTree lData (depth + 1) maxDepth (nodeId ++ "l")
        rightTree       = growTree rData (depth + 1) maxDepth (nodeId ++ "r")
        [lData, rData]  = splitData dataSet literal
        stopGrowing =
            depth == maxDepth ||
            gini [y | (DataPoint _ y) <- dataSet] == 0 ||
            length lData == 0 || length rData == 0
```

Stop Growing

$$\text{majorLabel}(D) = \underset{l \in \mathcal{L}}{\text{argmax}} \sum_{(\boldsymbol{x},y) \in D} \mathbb{I}\left[y = l\right]$$

```haskell
majorLabel :: DataSet -> Label
majorLabel dataSet = maxIndex $ labelCount [y | (DataPoint _ y) <- dataSet]
```

3.1.7 Output Tree in GraphViz

```haskell
labelToStringForGraphViz :: Tree -> String
labelToStringForGraphViz (Leaf l leafId) =
    leafId ++ " [label=\"Class: " ++ (show l) ++ "\"]\n"
labelToStringForGraphViz (Node (Literal i v) left right nodeId) =
    nodeId ++ " [shape=box,label=\"Feature[" ++ (show i) ++ "] < " ++ (show v) ++ "
        \"]\n" ++
    labelToStringForGraphViz left ++ labelToStringForGraphViz right

nodeToStringForGraphViz :: Tree -> String
nodeToStringForGraphViz (Leaf _ leafId) = leafId ++ ";\n"
nodeToStringForGraphViz (Node _ left right nodeId) =
    nodeId ++ " -- " ++ nodeToStringForGraphViz left ++
    nodeId ++ " -- " ++ nodeToStringForGraphViz right

treeToStringForGraphViz :: Tree -> String
```

```
treeToStringForGraphViz tree =
    "graph Tree {\n" ++ labelToStringForGraphViz tree ++ nodeToStringForGraphViz tree
       ++ "}"
```

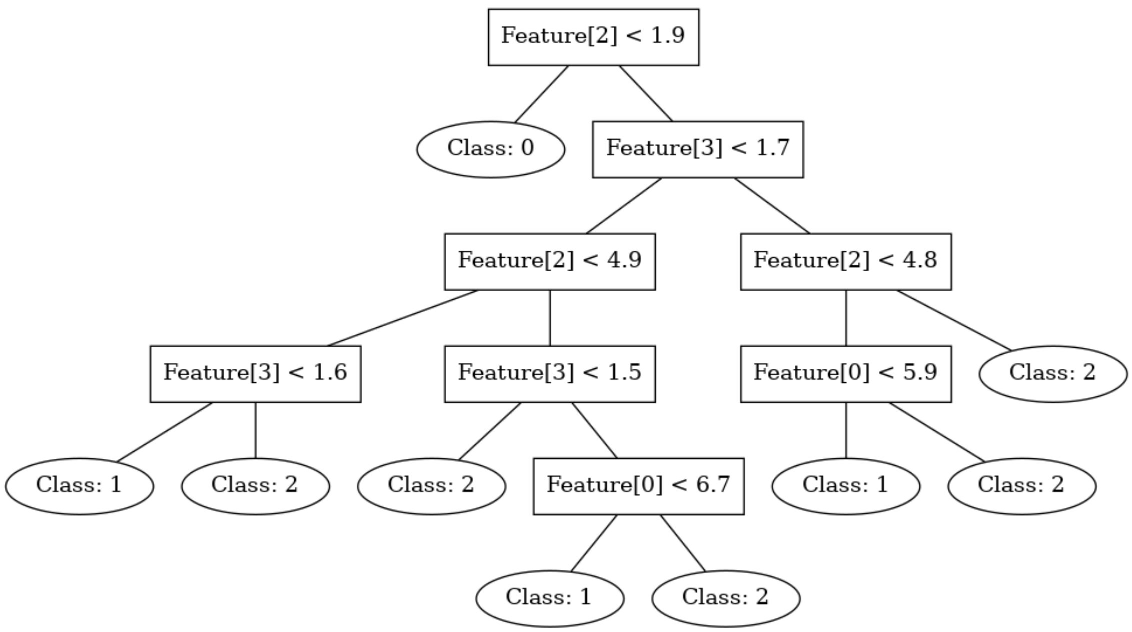

Figure 3.1: Example of GraphViz output

3.1.8 Other Functions

Algorithm

```
myMin :: [Split] -> Split
myMin splitList = foldr min (Split (Literal 0 0) 2) splitList

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
```

3.1.9 Test

```
testDT :: IO()
testDT = do
    dataSet <- readClsDataFromCSV "data/iris/iris.data"
    let tree = growTree dataSet 0 10 "n"
    let treeStr = show tree
    putStrLn treeStr
    writeFile "output/output-tree" treeStr
    writeFile "output/tree.dot" $ treeToStringForGraphViz tree
```

Chapter 4

Neural Network

4.1 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
```

4.2 Layers

4.2.1 Affine

forward

```
affine :: Mat -> Vec -> Mat -> Mat
```

```haskell
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)
```

4.2.2 Activation Function

ReLU

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

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

```haskell
relu :: Mat -> Mat
relu = cmap (max 0)

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

Sigmoid

See eq. (2.8).

4.2.3 Cross Entropy Error

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

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

```haskell
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
```

4.2.4 Softmax

Softmax

$$
\begin{aligned}
\exp(\boldsymbol{x}) &= \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_N} \end{bmatrix} \\
\operatorname{softmax}(\boldsymbol{x}) &= \frac{\exp(\boldsymbol{x})}{\|\exp(\boldsymbol{x})\|_1} = \frac{\exp(\boldsymbol{x}-\boldsymbol{c})}{\|\exp(\boldsymbol{x}-\boldsymbol{c})\|_1} \\
\operatorname{softmax}(X) &= \begin{bmatrix} \operatorname{softmax}(\boldsymbol{x}_{:1}) & \cdots & \operatorname{softmax}(\boldsymbol{x}_{:N}) \end{bmatrix}
\end{aligned}
$$

```
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
```

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)
```

4.2.5 Loss Function

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

```haskell
loss :: Vec -> Mat -> Mat -> R
loss w x t = softmaxWithLoss (forwardProp w x) t
```

4.2.6 Forward Propagetion

```haskell
--       (softmax)
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
```

4.2.7 Prediction

```haskell
predict :: Vec -> Mat -> Mat
predict w x = oneHotMat outputSize $ map maxIndex $ toRows $ forwardProp w x
```

4.2.8 Gradient

```haskell
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 (\_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)
```

4.2.9 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)
```

4.2.10 Test

```
testNN :: IO()
testNN = do
    cs1 <- readFile "dataset/train_data.dat"
    cs2 <- readFile "dataset/test_data.dat"
    cs3 <- readFile "dataset/train_label.dat"
    cs4 <- readFile "dataset/test_label.dat"

    let batchSize  = 100

    let trainDataList   = map read $ lines cs1
    let testDataList    = map read $ lines cs2
    let trainLabelList  = map read $ lines cs3
    let testLabelList   = map read $ lines cs4

    weight <- flatten <$> randn 1 weight_size

    let trainData   = matrix inputSize $ take (batchSize * inputSize) trainDataList
    let testData    = matrix inputSize testDataList
    let trainLabel  = oneHotMat outputSize $ take batchSize trainLabelList
    let testLabel   = oneHotMat outputSize testLabelList

    putStr "Now Loading Training Data...\n"
    putStr "size of train data  : "
    print $ size trainData
    putStr "size of train label : "
    print $ size trainLabel
    putStr "size of test data   : "
    print $ size testData
    putStr "size of test label  : "
    print $ size testLabel

    -- putStr "Gradient Check      : "
    -- print $ gradientCheck weight x t

    let learningRate = 0.1
    let iterNum = 100
    let newW = learn weight trainData trainLabel learningRate iterNum

    print $ testAccuracy newW trainData trainLabel
```

```
    print $ testAccuracy newW testData testLabel
```