Google Classroom Code: mhxgl24

Classification, Dataloaders, Activation Functions (sigmoid), loss functions (log loss, cross entropy), TensorBoard

Deep Learning (DS-5006)

Dr. Adeel Mumtaz

Lecture 5

Fall, 2022

National University
Of Computer and Emerging Sciences

Contents

- Review
  - NN Concepts summary
  - Multiple Regression
  - PyTorch Summary
  - Quiz-2
  - A Simple Regression Problem (Pytorch Implementation)
  - Home Task (backprop)
- A simple binary classification problem (Theory)
  - NN Architecture
  - Why Activation function is required
    - Sigmoid Activation
  - Why MSE can't be used as loss function
    - Log loss or logistic loss
  - GD for logistic regression
- A simple binary classification problem (PyTorch)
  - scikit-learn library
  - Feature/Data Scaling
  - Datasets and Dataloaders
  - Model Class
  - Mode Visualization using TorchSummary
  - PyTorch BCELoss
  - Batch wise Training Loop
  - Evaluation Metrics
  - Model Checkpointing
  - TensorBoard
- Summary
- Home Task

NN Summary Review

- Data Set, Training, Validation, Test
- NN as function Approximators
- Cost/Loss Function
  - MSE Loss for regression
- Architecture of NN
- Parameters
- Training Loop
- Optimizer
- Learning Rate
- Types of Gradient Descent
  - Epoch
  - Batch
- Loading/Saving Model

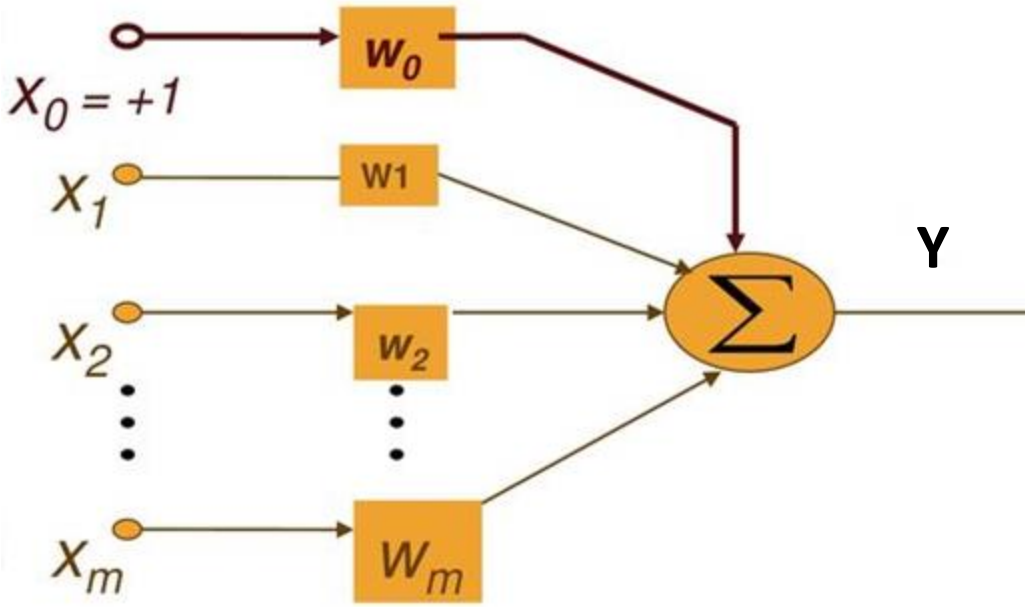
MULTIPLE REGRESSION

Multiple regression

- Multiple regression is a technique that can be used to analyze the relationship between a single dependent variable and several independent variables

Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
…	…	…	…	…

Model



Vectorization

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} & \cdots & x_n^{(3)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}_{m \times (n+1)}$$

$$\hat{Y} = XW$$

$$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}_{n+1}$$

$$Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}_{m}$$

Gradient Descent (Vectorization)

$$\mathcal{L}(X,W) = \frac{1}{m}\sum_{i=1}^{m}\left(W^T x^{(i)} - y^{(i)}\right)^2$$

$$dW = \frac{\partial \mathcal{L}(X,W)}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}(x^{(i)}W - y^{(i)})\,x^{(i)}$$

$$W_{new} = W_{old} - \alpha * dW$$

$$b_{new} = ?$$

Scalar derivative			Vector derivative		
$f(x)$	$\rightarrow$	$\frac{\mathrm{d}f}{\mathrm{d}x}$	$f(\mathbf{x})$	$\rightarrow$	$\frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}}$
bx	$\rightarrow$	b	$\mathbf{x}^T\mathbf{B}$	$\rightarrow$	$\mathbf{B}$
bx	$\rightarrow$	b	$\mathbf{x}^T\mathbf{b}$	$\rightarrow$	$\mathbf{b}$
x^2	$\rightarrow$	$2x$	$\mathbf{x}^T\mathbf{x}$	$\rightarrow$	$2\mathbf{x}$
bx^2	$\rightarrow$	$2bx$	$\mathbf{x}^T\mathbf{B}\mathbf{x}$	$\rightarrow$	$2\mathbf{B}\mathbf{x}$

PYTORCH SUMMARY

Summary-1

```
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
tensor = torch.randn((2, 3, 4), dtype=torch.float)
```

```
print(tensor.size(), tensor.shape)
```

```
same_matrix = matrix.view(1, 6)
```

```
another_matrix = matrix.view(1, 6).clone().detach()
```

```
x_train_tensor = torch.as_tensor(x_train)
```

```
dummy_tensor.numpy()
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

```
x_train_tensor = torch.as_tensor(x_train).float().to(device)
```

```
back_to_numpy = x_train_tensor.cpu().numpy()
```

Summary-2

```
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
```

```
loss = (error ** 2).mean()
```

```
loss.backward()
```

```
print(b.grad, w.grad)
```

```
b.grad.zero_(), w.grad.zero_()
```

```
with torch.no_grad():
    b -= lr * b.grad
    w -= lr * w.grad
```

Summary-3

- Optimizer
  - opt=optim.SGD(parameters)
  - opt.step()
  - opt.zero_grad()
- Loss Fuction
  - nn.MSELoss()
  - loss.backward()
- Model Class
  - __init__(self)
  - forward(self, x)
  - Defining Model Parameters
  - Model state_dict()
  - Model device
  - Nested Model

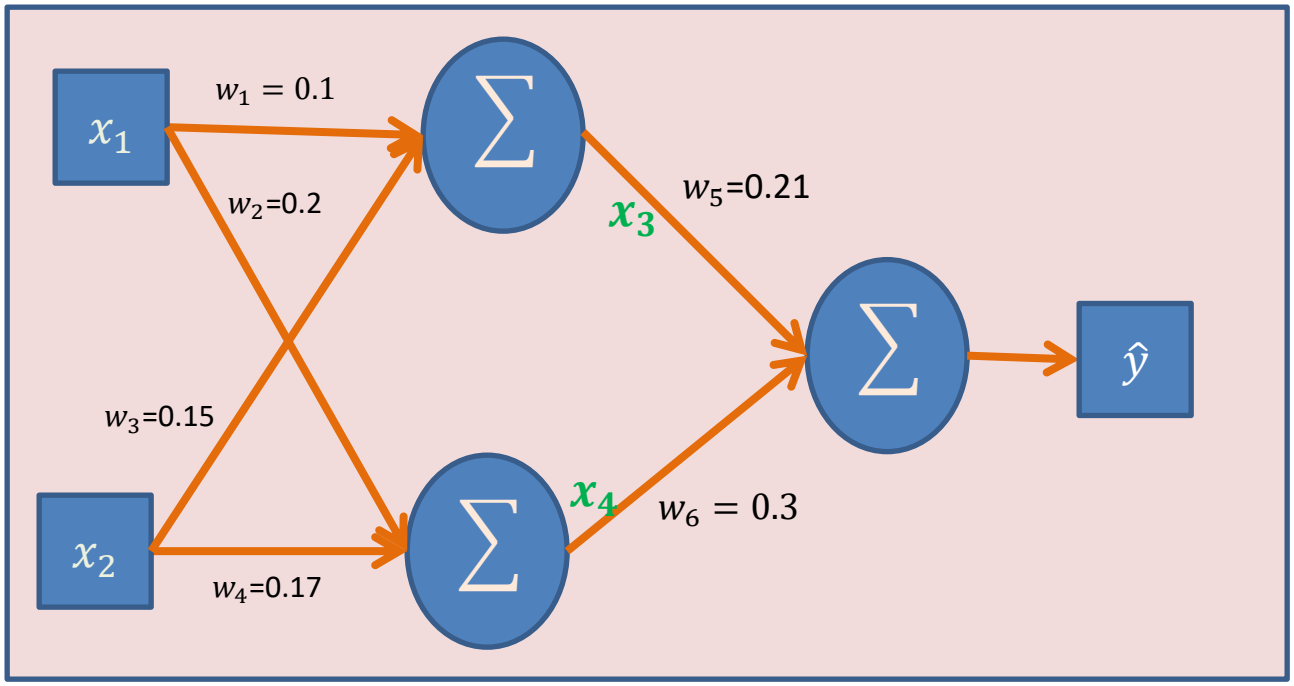
Forward & Backprop

DYNAMIC COMPUTATION GRAPH

Graded Home Task (backprop)

Training Data

x_1	x_2	y
-2	3	5
1	-2	-3

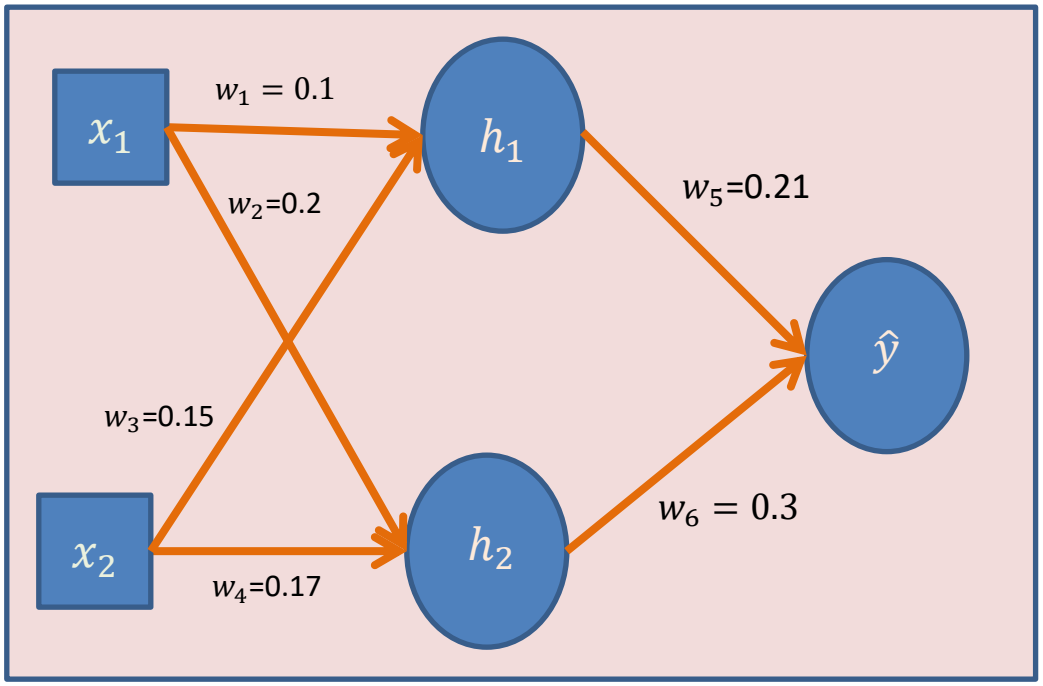


Assuming MSE Loss function, Given above training samples calculate

1. Forward Pass (values of $x_3, x_4, \hat{y}$)
2. Calculate Loss value
3. Calculate gradients of w_1 to w_6 with respect to loss function using backprop/chain rule
4. Update w_1 to w_6 using SGD optimizer with lr=0.1
5. Write PyTorch Code using Autograd, built-in loss function and optimizer
6. Compare manual and PyTorch output

Quiz-2

x_1	x_2	y
-2	3	5

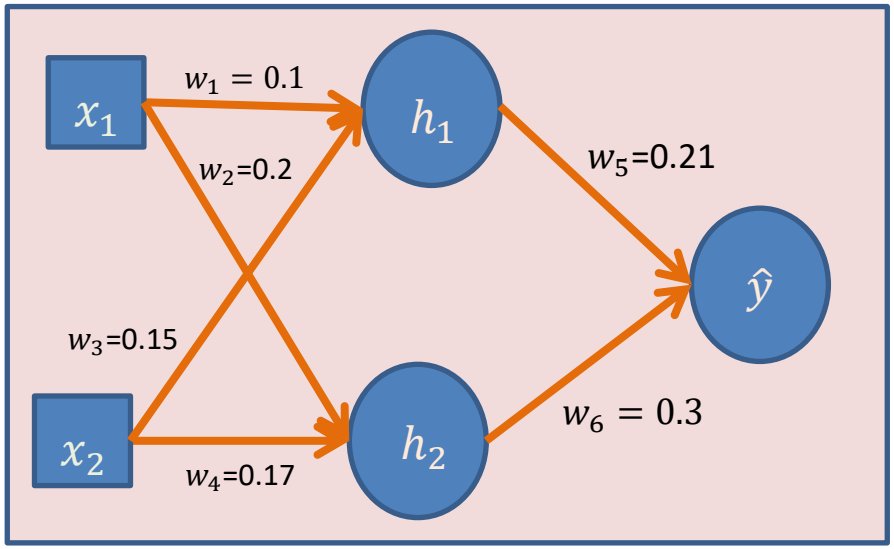


Assuming MSE Loss function, Given above training sample calculate

1. Forward Pass (values of $h_1, h_2, \hat{y}$)
2. Calculate Loss value
3. Calculate gradients of w_1 and w_5 with respect to loss function
4. Update w_1 and w_5 using SGD optimizer with lr=0.1
5. Write PyTorch Code using Autograd, built-in loss function and optimizer

Quiz-2 (solution)

x_1	x_2	y
-2	3	5



Step1: Forward Pass:

$$h_1 = x_1 * w_1 + x_2 * w_3 = 0.25$$
$$h_2 = x_1 * w_2 + x_2 * w_4 = 0.11$$
$$\hat{y} = h_1 * w_5 + h_2 * w_6 = 0.0855$$

Step2: Computing Loss

$$L = (\hat{y} - y)^2 = 24.1523$$

Step3: Computing Gradients

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_5} = 2(\hat{y} - y) * h_1 = 2(0.0855 - 5) * 0.25 = -2.45725$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial h_1} * \frac{\partial h_1}{\partial w_1} = 2(\hat{y} - y) * w_5 * x_1 = 2(0.0855 - 5) * 0.21 * (-2) = 4.12818$$

Step4: Parameter update using SGD

$$w_{1new} = w_1 - \alpha * \frac{\partial L}{\partial w_1} = 0.1 - 0.1 * 4.12818 = \mathbf{-0.3128}$$

$$w_{5new} = w_5 - \alpha * \frac{\partial L}{\partial w_5} = 0.21 - 0.1 * (-2.45725) = \mathbf{0.4557}$$

Quiz-2 (Solution)

```python
import torch

#forward pass
x=torch.tensor([-2,3],dtype=torch.float)
wh= torch.tensor([[0.1,0.2],[0.15,0.17]],dtype=torch.float,requires_grad=True)
h= torch.matmul(wh.t(),x)
wo=torch.tensor([0.21,0.3],dtype=torch.float,requires_grad=True)
yhat = torch.matmul(wo.t(),h)

#backward pass
loss_func=torch.nn.MSELoss()
y=torch.tensor([5],dtype=torch.float)
loss=loss_func(yhat,y)
loss.backward()
print(wh.grad)
print(wo.grad)

opt=torch.optim.SGD(params=[wh,wo],lr=0.1)
opt.step()

print(wh)
print(wo)
```

Output (grad)

```
tensor([[ 4.1282,  5.8974],
        [-6.1923, -8.8461]])
tensor([-2.4573, -1.0812])
```

Output (new params)

```
tensor([[-0.3128, -0.3897],
        [ 0.7692,  1.0546]],
tensor([0.4557, 0.4081], req
```

Step3: Computing Gradients

$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_5} = 2(\hat{y} - y) * h_1 = 2(0.0855 - 5) * 0.25$=-2.45725

$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial h_1} * \frac{\partial h_1}{\partial w_1} = 2(\hat{y} - y) * w_5 * x_1 = 2(0.0855 - 5) * 0.21 * (-2)$=4.12818

Step4: Parameter update using SGD

$$w_{1new} = w_1 - \alpha * \frac{\partial L}{\partial w_1} = 0.1 - 0.1 * 4.12818 = \mathbf{-0.3128}$$

$$w_{5new} = w_5 - \alpha * \frac{\partial L}{\partial w_5} = 0.21 - 0.1 * (-2.45725) = \mathbf{0.4557}$$

A SIMPLE REGRESSION PROBLEM (PYTORCH IMPLEMENTATION)

Imports, Device, Seed

```
1    import numpy as np
2    import matplotlib.pyplot as plt
3    import torch
4    from torchviz import make_dot
```

```
7    device='cuda' if torch.cuda.is_available() else 'cpu'
8    torch.manual_seed(100)
```

Model Class

```
10  class SimpleRGNet(torch.nn.Module):
11      def __init__(self):
12          super().__init__()
13          self.linear = torch.nn.Linear(1,1,bias=True)
14      def forward(self,x):
15          return self.linear(x)
16
```

Model, optimizer and loss initialization

```
40    model=SimpleRGNet().to(device)
41    paramList=list(model.parameters())
42    stateDict=model.state_dict()
43    print(paramList)
44    print(stateDict)
45
46    lr=0.1
47    optimizer=torch.optim.SGD(model.parameters(),lr=lr)
48    lossfnc = torch.nn.MSELoss(reduce="mean")
```

Data Preparation

```
20  true_w=2
21  true_b=1
22  N=100
23  #data generation
24  np.random.seed(100)
25  x=np.random.rand(N,1)
26  epsilon=0.1*np.random.randn(N,1)
27  y=true_w*x+true_b+epsilon
28  #data split
29  idx=np.arange(N)
30  np.random.shuffle(idx)
31  idx_train=idx[:int(0.8*N)]
32  idx_test=idx[int(0.8*N):]
33  x_train, y_train = x[idx_train],y[idx_train]
34  x_val, y_val = x[idx_test],y[idx_test]
35  x_train_tensor=torch.as_tensor(x_train).float().to(device)
36  y_train_tensor=torch.as_tensor(y_train).float().to(device)
37  x_val_tensor=torch.as_tensor(x_val).float().to(device)
38  y_val_tensor=torch.as_tensor(y_val).float().to(device)
```

Training Loop

```
53  trainLosses=[]
54  valLosses=[]
55  for i in range(1000):
56      model.train()
57      #forward pass
58      yhat=model(x_train_tensor)
59      loss=lossfnc(yhat,y_train_tensor)
60      trainLosses.append(loss.item())
61      #make_dot(loss).view()
62      loss.backward()
63      optimizer.step()
64      optimizer.zero_grad()
65      stateDict=model.state_dict()
66      w=stateDict['linear.weight']
67      b=stateDict['linear.bias']
68      w=w.item()
69      b=b.item()
70
71      model.eval()
72      with torch.no_grad():
73          #val MSE loss
74          yhatval=model(x_val_tensor)
75          valLoss=lossfnc(yhatval,y_val_tensor)
76          valLosses.append(valLoss.item())
77      #stopping condition
78      if(valLoss.item()<0.0001):
79          break
80      print(f'train loss={loss.item()}, val loss={valLoss.item()}, w={w}, b={b}')
```

PyTorch Summary

- Data & Parameters as Tensors
- Moving from numpy to Tensor to GPU
- PyTorch Model Class
- Autograd using backward()
- Built-in Loss Functions
- Built-in optimizers
- Much Simplified Training Loop
- Getting Ready for deep models training using large datasets?

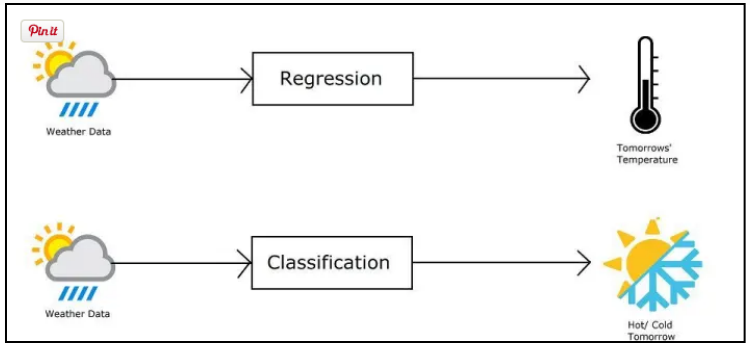
A SIMPLE BINARY CLASSIFICATION PROBLEM (THEORY)

Binary Classification

- Supervised learning algorithm that categorizes new observations into one of two classes

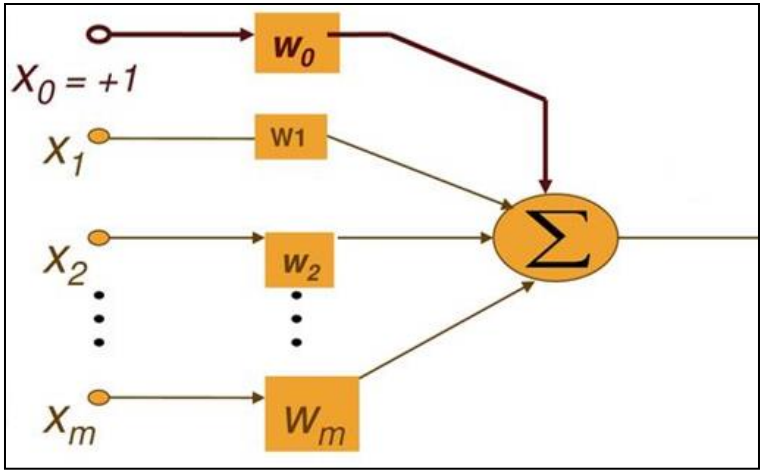
Application	Observation	0	1
Medical Diagnosis	Patient	Healthy	Diseased
Email Analysis	Email	Not Spam	Spam
Financial Data Analysis	Transaction	Not Fraud	Fraud
Marketing	Website visitor	Won't Buy	Will Buy
Image Classification	Image	Hotdog	Not Hotdog

- Logistic Regression
- k-Nearest Neighbors
- Decision Trees
- Support Vector Machine
- Naive Bayes



Binary Classification

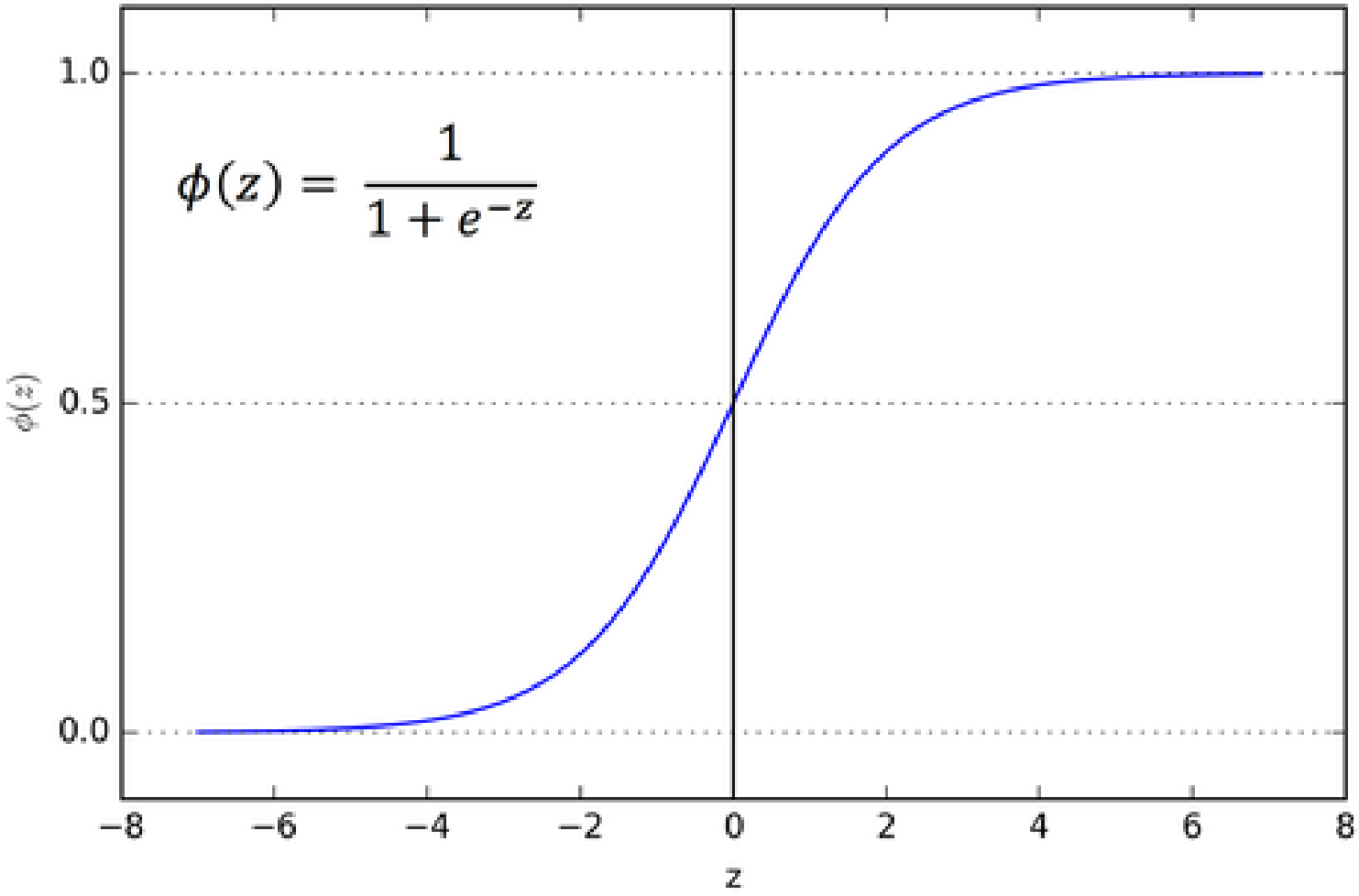
- Why Linear Regression is not suitable for classification?
  - Linear Regression deals with continuous values at output whereas classification problems requires discrete values



Solution (Activation Function)

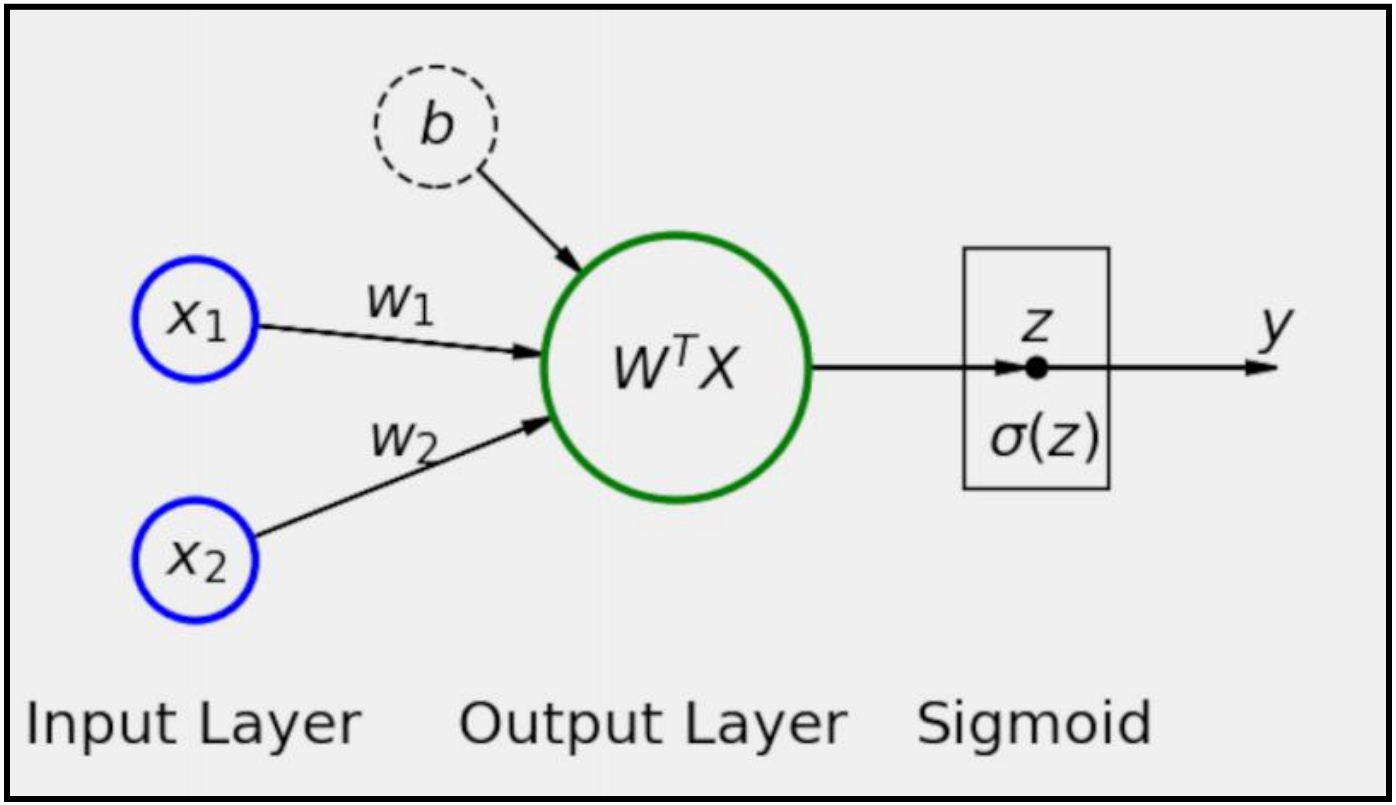
- Sigmoid function
  - Mathematical function having a characteristic that can take any real value and **map it to between 0 to 1** shaped like the letter "S".
  - The sigmoid function also called a logistic function.
  - $\sigma(Z) = \frac{1}{1+e^{-z}}$
- We can threshold the probability output of sigmoid for classification
  - E.g. if threshold=0.5
  - Class = A if $\sigma(Z) \geq 0.5$
  - Class = B if $\sigma(Z) < 0.5$

Sigmoid Activation



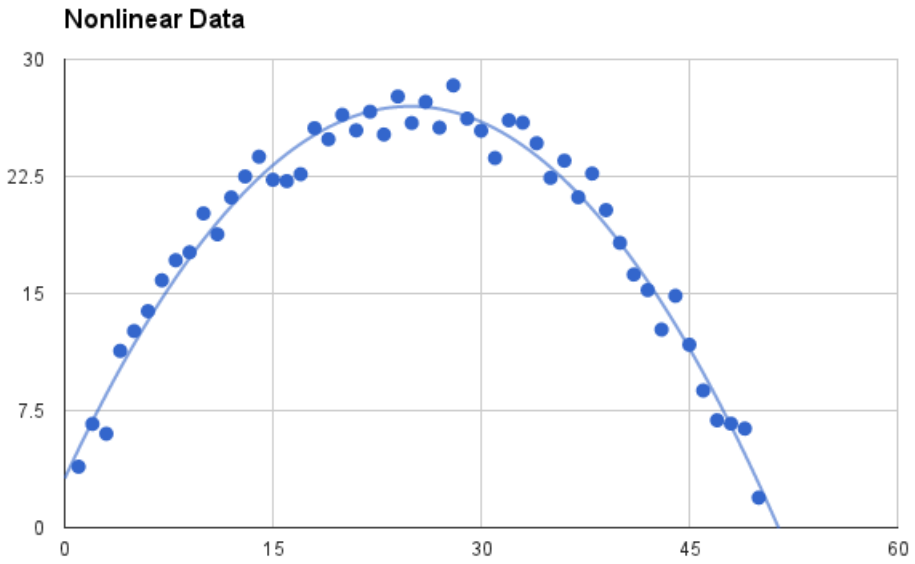
Resulting Logistic Regression Model

- We can think of the logistic regression as the second simplest neural network possible.
- It is pretty much the same as the linear regression, but with a sigmoid applied to the results of the output layer (z).



Notes on Uses of Activation Functions

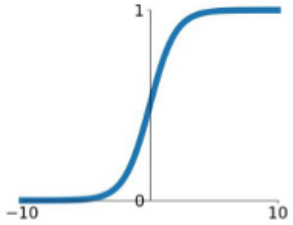
- It makes it easy for the model to generalize or adapt with variety of data
- **Help the network learn complex patterns in the data**
- **Different for hidden and output layer**
- Sigmoid, tanh, Softmax, ReLU etc



Notes on Uses of Activation Functions

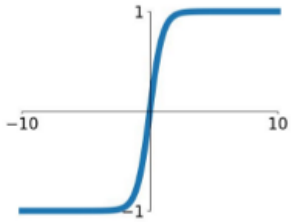
Sigmoid

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



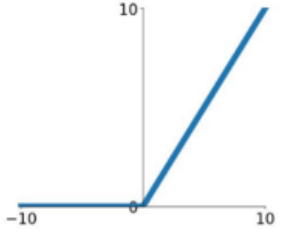
tanh

$$\tanh(x)$$



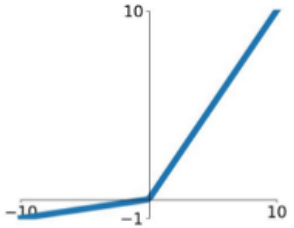
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

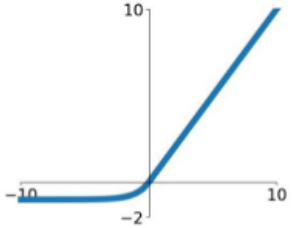


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

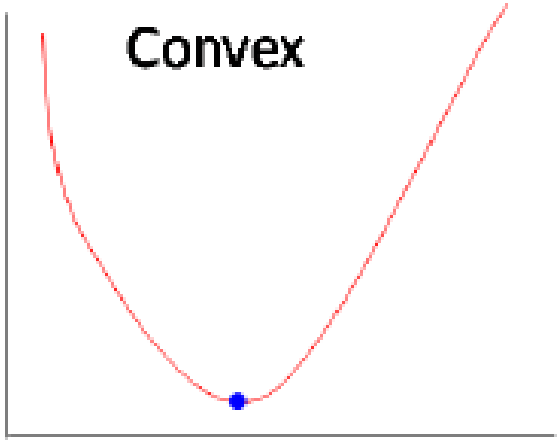
ELU

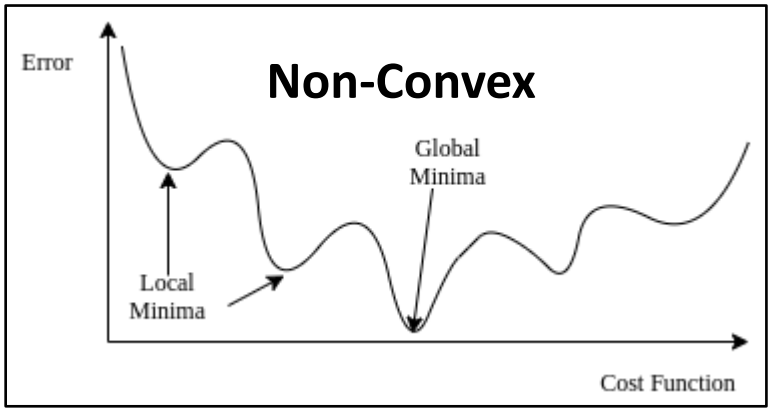
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Binary Classification with MSE

- With sigmoid MSE loss function becomes Non-convex
- If the loss function is not convex, it is not guaranteed that we will always reach the global minima, rather we might get stuck at local minima.

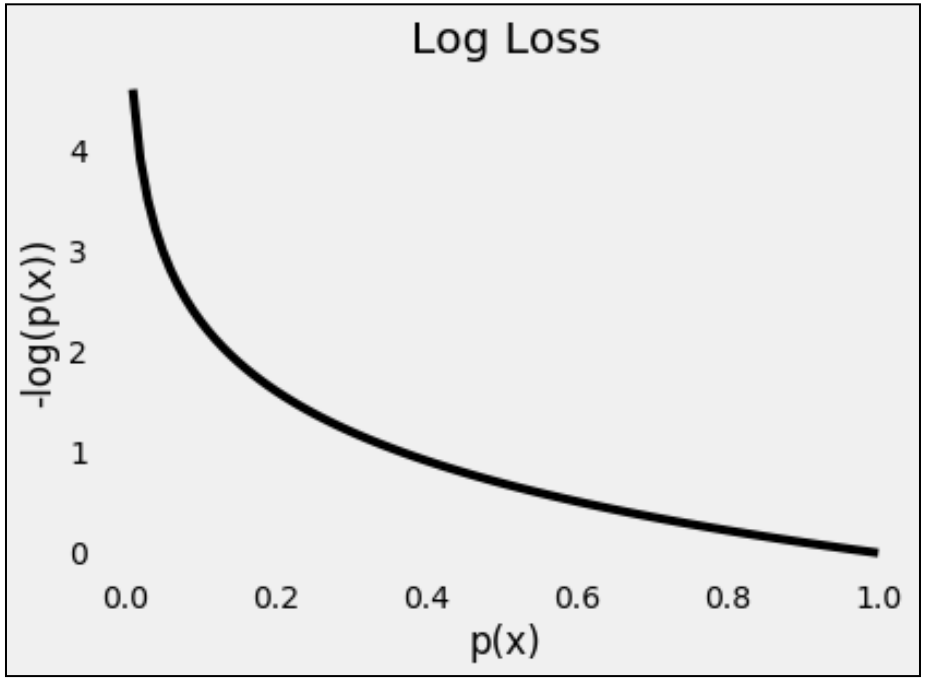




Log loss or Logistic loss

- Also called Binary Cross-Entropy
- $\sigma(Z_i) = p(y_i = 1) = \hat{y}_i$
- **Note I am still calling the final output of network after sigmoid as $\hat{y}_i$**
- $1 - \sigma(Z_i) = p(y_i = 0) = 1 - \hat{y}_i$
- Log Loss for positive class= $-\log(\hat{y}_i)$
- Log Loss for negative class= $-\log(1 - \hat{y}_i)$

- **More loss if predicted probability is away from true class**



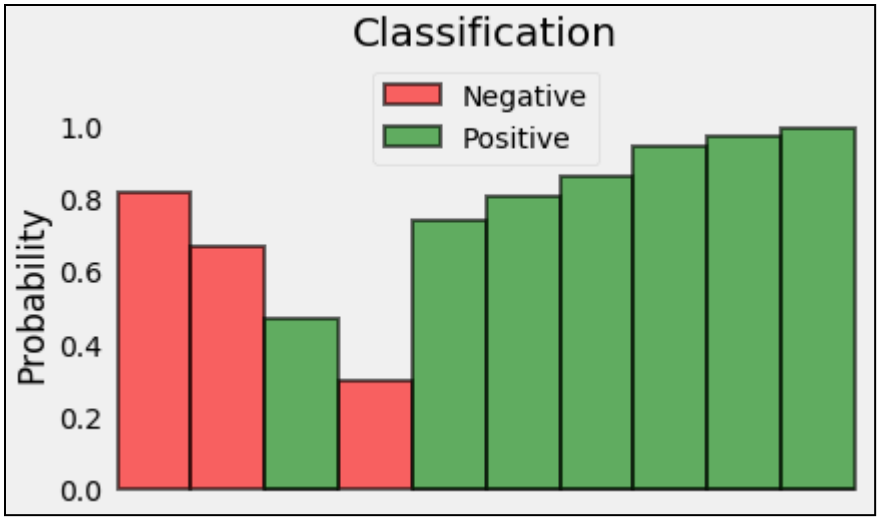
Log loss or Logistic loss

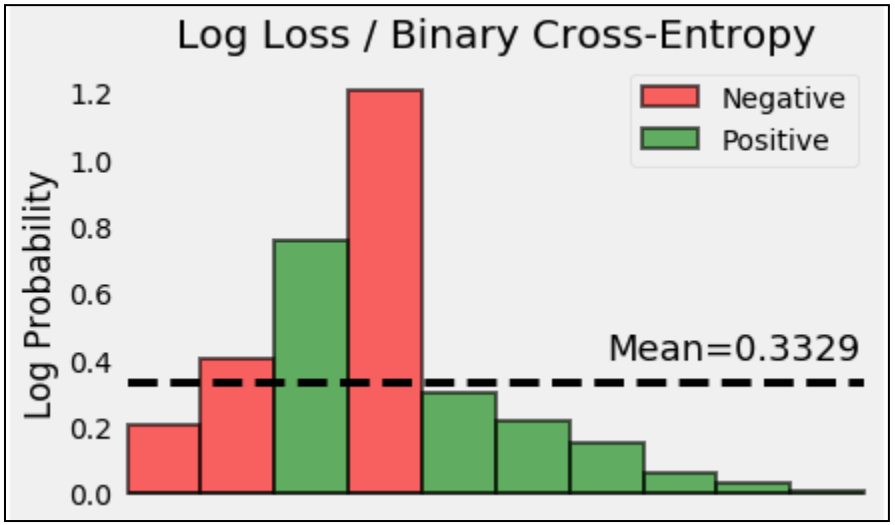
- Combining loss for both classes

  Log Loss = $-y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)$

  Mean Log Loss =

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$





Gradient Descent for logistic regression

- $\mathbf{L(W,X)} = -\frac{1}{N}\sum_{i=1}^{N} y^i \, log(\hat{y}_i) + (1 - y^i) log(1 - \hat{y}_i)$
- $\hat{y}_i = \frac{1}{1+e^{-z}}$
- $\mathbf{Z}=W^T x^i$
- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W}$
- $\frac{\partial z}{\partial W} = x^i$
- $\frac{\partial \hat{y}}{\partial z} = \hat{y}_i(1 - \hat{y}_i)$ ***derivative of sigmoid***
- $\frac{\partial L}{\partial \hat{y}} = -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y^i}{\hat{y}_i} - \frac{(1-y^i)}{1-\hat{y}_i}\right)$
- $= -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y^i(1-\hat{y}_i)-\hat{y}_i(1-y^i)}{\hat{y}(1-\hat{y}_i)}\right)$
- $= -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y^i-\hat{y}_i}{\hat{y}(1-\hat{y})}\right)$
- $\frac{\partial L}{\partial W} = -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y^i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)}\right) * \hat{y}_i(1 - \hat{y}_i) * x^i$
- $\frac{\partial L}{\partial W} = dW = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y^i) * x^i$ **Any surprise?**
- $W_{new} = W_{old} - \alpha * dW$

$$dW = \frac{\partial \mathcal{L}(X,W)}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}(x^{(i)}W - y^{(i)})\,x^{(i)}$$

GD for linear regression

A SIMPLE BINARY CLASSIFICATION PROBLEM (PYTORCH)

The moons dataset

- A simple toy dataset to visualize clustering and classification algorithm
- **scikit-learn** comes with a few small standard datasets that do not require to download any file from some external website
- **make_moons** generate 2d binary classification datasets that are challenging to certain algorithms

Other Datasets

load_boston(*[, return_X_y])	DEPRECATED: load_boston is deprecated in 1.0 and will be removed in 1.2.
load_iris(*[, return_X_y, as_frame])	Load and return the iris dataset (classification).
load_diabetes(*[, return_X_y, as_frame, scaled])	Load and return the diabetes dataset (regression).
load_digits(*[, n_class, return_X_y, as_frame])	Load and return the digits dataset (classification).
load_linnerud(*[, return_X_y, as_frame])	Load and return the physical exercise Linnerud dataset.
load_wine(*[, return_X_y, as_frame])	Load and return the wine dataset (classification).
load_breast_cancer(*[, return_X_y, as_frame])	Load and return the breast cancer wisconsin dataset (classification).

fetch_olivetti_faces(*[, data_home, ...])	Load the Olivetti faces data-set from AT&T (classification).
fetch_20newsgroups(*[, data_home, subset, ...])	Load the filenames and data from the 20 newsgroups dataset (classification).
fetch_20newsgroups_vectorized(*[, subset, ...])	Load and vectorize the 20 newsgroups dataset (classification).
fetch_lfw_people(*[, data_home, funneled, ...])	Load the Labeled Faces in the Wild (LFW) people dataset (classification).
fetch_lfw_pairs(*[, subset, data_home, ...])	Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).
fetch_covtype(*[, data_home, ...])	Load the covertype dataset (classification).
fetch_rcv1(*[, data_home, subset, ...])	Load the RCV1 multilabel dataset (classification).
fetch_kddcup99(*[, subset, data_home, ...])	Load the kddcup99 dataset (classification).
fetch_california_housing(*[, data_home, ...])	Load the California housing dataset (regression).

TORCHAUDIO,TORCHVISION, TORCHTEXT Datasets

- CelebA
- CIFAR
- Cityscapes
- COCO
  - Captions
  - Detection
- DatasetFolder
- EMNIST
- FakeData
- Fashion-MNIST
- Flickr
- HMDB51
- ImageFolder
- ImageNet
- Kinetics-400
- KMNIST
- LSUN
- MNIST

Built-in datasets

Image classification

Image detection or segmentation

Optical Flow

Image pairs

Image captioning

Video classification

torchaudio.datasets

CMUARCTIC

CMUDict

COMMONVOICE

GTZAN

LibriMix

LIBRISPEECH

LibriLightLimited

LIBRITTS

LJSPEECH

SPEECHCOMMANDS

TEDLIUM

VCTK_092

DR_VCTK

YESNO

QUESST14

References

- Text Classification
  - AG_NEWS
  - AmazonReviewFull
  - AmazonReviewPolarity
  - CoLA
  - DBpedia
  - IMDb
  - MNLI
  - MRPC
  - QNLI
  - QQP
  - RTE
  - SogouNews
  - SST2
  - STSB
  - WNLI
  - YahooAnswers
  - YelpReviewFull
  - YelpReviewPolarity

- Language Modeling
  - PennTreebank
  - WikiText-2
  - WikiText103
- Machine Translation
  - IWSLT2016
  - IWSLT2017
  - Multi30k
- Sequence Tagging
  - CoNLL2000Chunking
  - UDPOS
- Question Answer
  - SQuAD 1.0
  - SQuAD 2.0
- Unsupervised Learning
  - CC100
  - EnWik9

make_moons

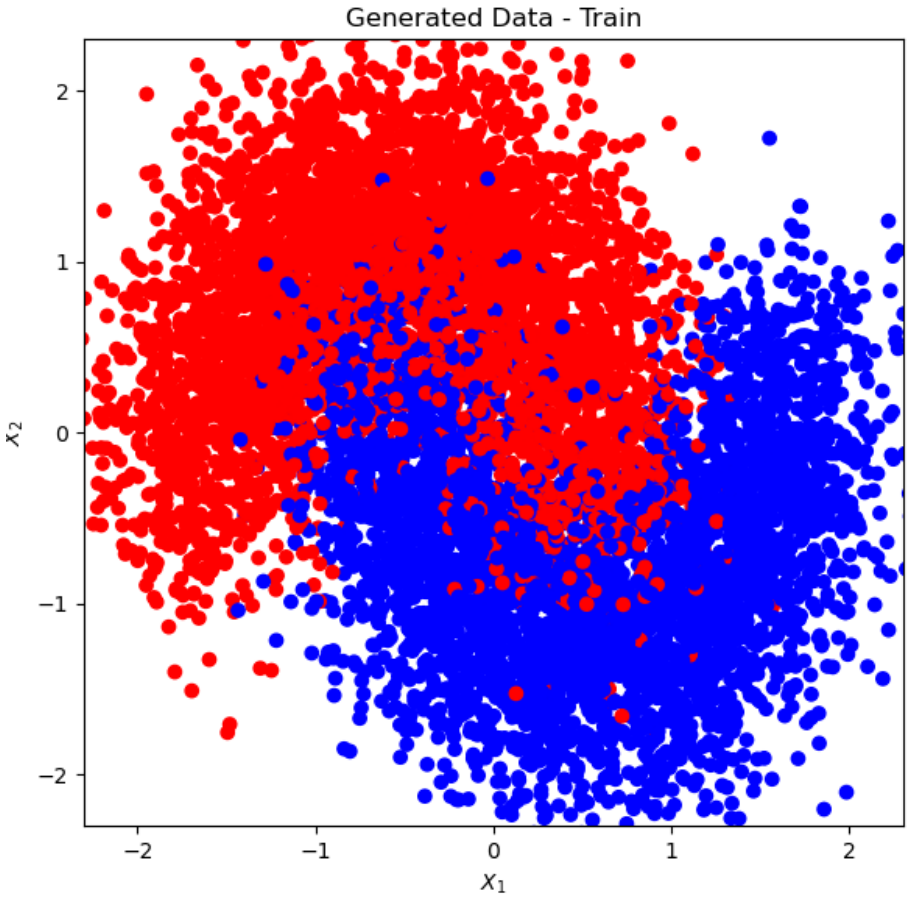
```
from sklearn.datasets import make_moons
```

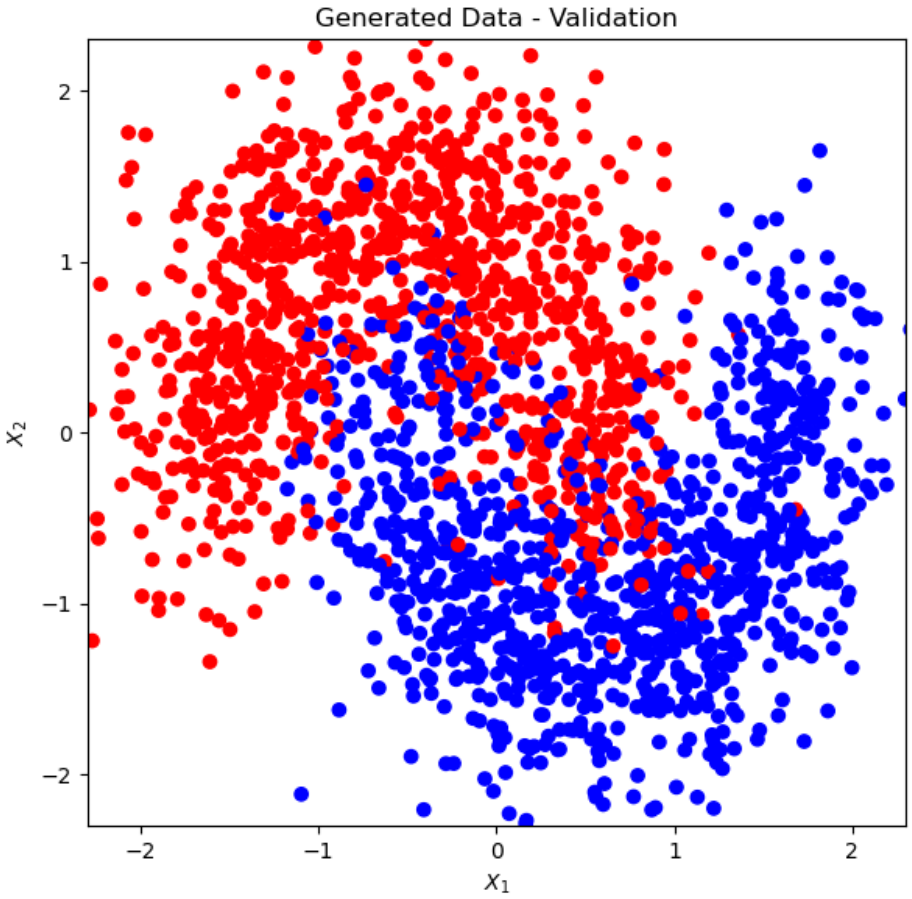
```
#Getting a toy dataset from scikit learn library
X, y = make_moons(n_samples=10000, noise=0.3, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=.2, random_state=13)
```

```
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

ax[0].scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)#, edgecolors='k')
ax[0].set_xlabel(r'$X_1$')
ax[0].set_ylabel(r'$X_2$')
ax[0].set_xlim([-2.3, 2.3])
ax[0].set_ylim([-2.3, 2.3])
ax[0].set_title('Generated Data - Train')
```

make_moons


Generated Data - Train
2
1
0
−1
−2
x_2
−2
−1
0
1
2
x_1


Generated Data - Validation
2
1
0
−1
−2
x_2
−2
−1
0
1
2
x_1

Feature/Data Scaling

- Machine learning algorithms perform better when numerical input variables are scaled to a standard range
- Differences in the scales across input variables may increase the difficulty of the problem being modeled
- For example, algorithms that fit a model that use a weighted sum of input variables are affected, such as linear regression, logistic regression, and artificial neural networks (deep learning).
- Main Types:
  - Normalization (*MinMaxScaler*)
    - scales each input variable separately to the range 0-1
    - newX = (x – min) / (max – min)
  - Standardization (StandardScaler)

Feature/Data Scaling

- Standard Scaler
  - normalize the features i.e. each column of X, INDIVIDUALLY, so that each column/feature/variable will have **μ = 0 and σ = 1**

Standardization:

$$z = \frac{x - \mu}{\sigma}$$

with mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N}(x_i)$$

and standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

```
from sklearn.preprocessing import StandardScaler
```

Feature/Data Scaling

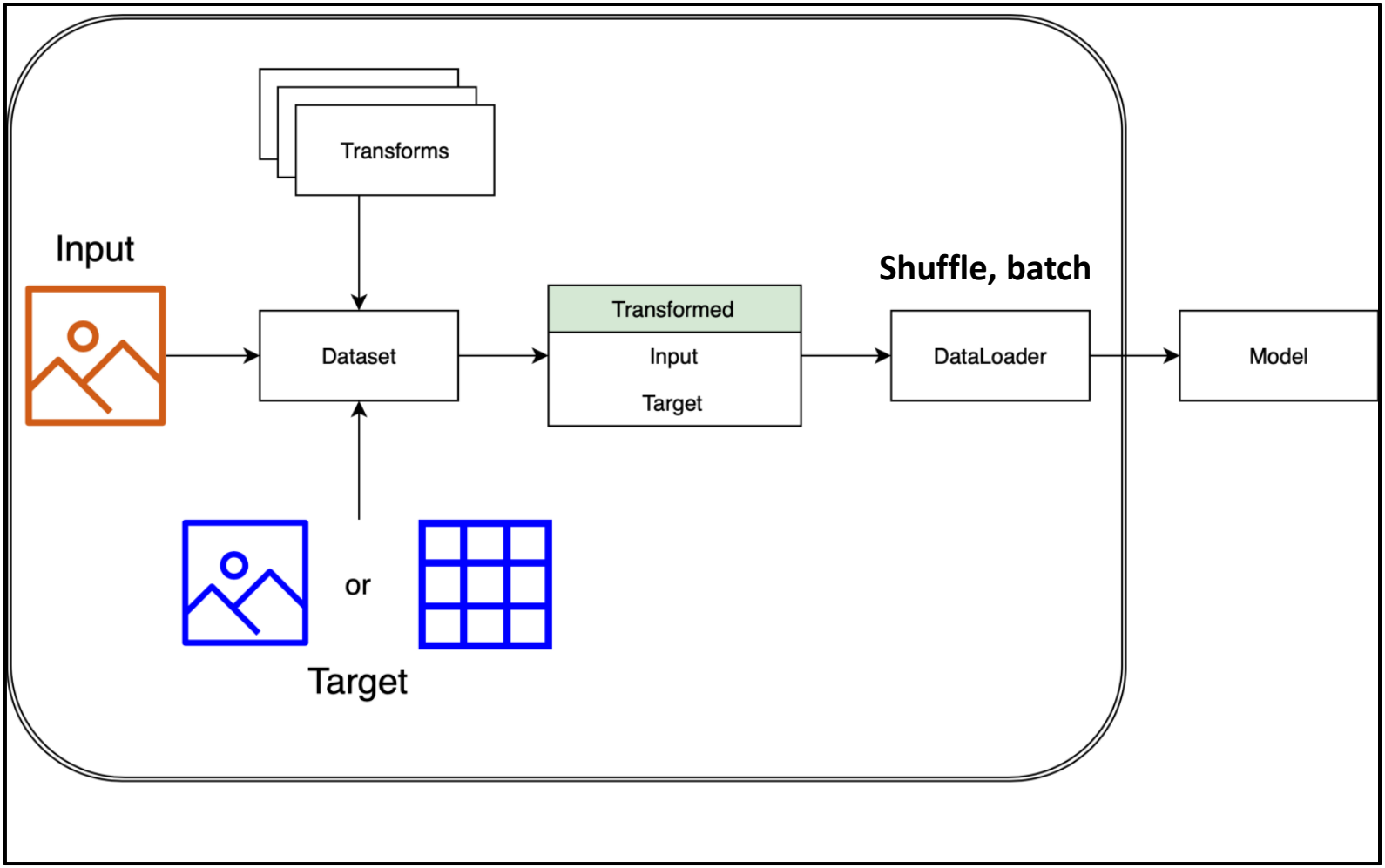
```
from sklearn.preprocessing import StandardScaler
```

```
sc = StandardScaler()
sc.fit(X_train)  #note only from training data

X_train = sc.transform(X_train)
X_val = sc.transform(X_val)
```

- All preprocessing transforms in deep learning are trained (parameter estimation) only using training data
  - We don’t know test data yet
  - Hide validation data as much as we can
- Here Mean, variance for standard scaling is computed from training data only
- Soon we will shift to **PyTorch Transforms**

Datasets and Dataloaders



Datasets

- All PyTorch Datasets (Toy, Synthetic, Pubic) are objects of classes derviced from **Dataset class**
- Built-in PyTorch Dataset classes
  - TensorDataset, ImageFolder, datasets.FashionMNIST etc
- Create your own sub-class for your dataset
  - Load Data
  - Apply Transofrmations
- We will create datasets for numeric, image, text and audio data

Dataset for moons

```
from torch.utils.data import DataLoader, TensorDataset,Dataset
```

```
class MoonsDataSet(Dataset):
    def __init__(self,x_tensor, y_tensor):
        super().__init__()
        self.X=x_tensor
        self.Y=y_tensor


    def __getitem__(self, index):
        return (self.X[index],self.Y[index])

    def __len__(self):
        return len(self.X)
```

Dataset for moons

```
#Getting a toy dataset from scikit learn library
X, y = make_moons(n_samples=10000, noise=0.3, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=.2, random_state=13)
```

```
sc = StandardScaler()
sc.fit(X_train)  #note only from training data

X_train = sc.transform(X_train)
X_val = sc.transform(X_val)
```

```
# Builds tensors from numpy arrays
x_train_tensor = torch.as_tensor(X_train).float()
y_train_tensor = torch.as_tensor(y_train.reshape(-1, 1)).float()
x_val_tensor = torch.as_tensor(X_val).float()
y_val_tensor = torch.as_tensor(y_val.reshape(-1, 1)).float()
# Builds dataset containing ALL data points
train_dataset = MoonsDataSet(x_train_tensor, y_train_tensor)
val_dataset = MoonsDataSet(x_val_tensor, y_val_tensor)
```

DataLoader

- Combines a dataset and a sampler, and provides an iterable over the given dataset
- DataLoader in action (Why shuffle for trainloader only?)

```
# Builds a loader of each set
train_loader = DataLoader(dataset=train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
test_batch=next(iter(train_loader))
total_batches_one_epoch = len(iter(train_loader))
```

```
for X_train, Y_train in train_loader:
```

Size of X_train?

Size of Y_train?

```
for X_val, Y_val in val_loader:
```

Model Class

```
class SimpleClassificationNet(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.linearLayer1 = nn.Linear(2,1000) #hidden layer
        self.linearLayer2 = nn.Linear(1000,1)
        self.sigmoidLayer=nn.Sigmoid()

    def forward(self,x):
        u=self.linearLayer1(x)
        z=self.linearLayer2(u)
        yhat=self.sigmoidLayer(z)
        return yhat
```

#of parameters for each layer?
Total # of parameters?

TorchSummary Package

```
from torchsummary import summary
```

```
model = SimpleClassificationNet().to(device)
stateDict=model.state_dict()
print(stateDict)
print(model)
summary(model,(1,2))
```

```
SimpleClassificationNet(
  (linearLayer1): Linear(in_features=2, out_features=1000, bias=True)
  (linearLayer2): Linear(in_features=1000, out_features=1, bias=True)
  (sigmoidLayer): Sigmoid()
)
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Linear: 1-1                            [-1, 1, 1000]             3,000
├─Linear: 1-2                            [-1, 1, 1]                1,001
├─Sigmoid: 1-3                           [-1, 1, 1]                --
==========================================================================================
Total params: 4,001
Trainable params: 4,001
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.02
Estimated Total Size (MB): 0.02
```

Optimizer & Loss

- PyTorch implements the binary cross-entropy loss in **nn.BCELoss**
- **BCEWithLogitsLoss (be careful)**
  - The former loss function took probabilities as an argument (together with the labels, obviously). This loss function takes **logits** as an argument, instead of probabilities.

```
lr = 0.001
optimizer = optim.SGD(model.parameters(), lr=lr)

loss_fn = nn.BCELoss()
```

Batch Training Loop
Thanks to DataLoader

```
#batch wise training loop
epochs = 1000
train_losses = []
val_losses = []
best_accuracy=0
for epoch in range(epochs):  #epochs loop

    all_Y_train_epoch=np.array([]).reshape(0,1)
    all_Yhat_train_epoch=np.array([]).reshape(0,1)
    all_train_losses_epoch=np.array([])

    for X_train, Y_train in train_loader:        #batch wise  training on train set
        model.train()
        X_train = X_train.to(device)
        Y_train = Y_train.to(device)
        y_hat = model(X_train)

        loss = loss_fn(y_hat, Y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        #store metrics for all batches of current epoch
        all_Y_train_epoch=np.vstack((all_Y_train_epoch,Y_train.detach().cpu().numpy()))
        all_Yhat_train_epoch=np.vstack((all_Yhat_train_epoch,y_hat.detach().cpu().numpy()))
        all_train_losses_epoch=np.append(all_train_losses_epoch,loss.item())
```

What DataLoader will do if batch_size is not divisible by number of training samples?

Why we are saving losses for each batch along with y and yhat?

Batch Training Loop Computing Metrics

- $Accuracy = \frac{\#\ of\ correct\ predictions}{total\ number\ of\ predictions}$
- **Is it a good measure?**
- **We will study more metrics next week**

	Total	Correct Prediction	Accuracy
Cancer=Yes	300	90	30%
Cancer= No	9700	9560	98.5%

Overall Accuracy: 96.5% Error: 3.5%

```
#computing metrics for current epoch
train_losses.append(all_train_losses_epoch.mean()) #mean loss for all batches
preidctions=(all_Yhat_train_epoch>=0.5) #from probabilities to predictions
acTrain=accuracy_score(all_Y_train_epoch, preidctions)
```

Batch Training Loop Computing Metrics

- $Accuracy = \frac{\#\ of\ correct\ predictions}{total\ number\ of\ predictions}$
- **Is it a good measure?**
- **We will study more metrics next week**

	Total	Correct Prediction	Accuracy
Cancer=Yes	300	90	30%
Cancer= No	9700	9560	98.5%

Overall Accuracy: 96.5% Error: 3.5%

Computing Accuracy

```
from sklearn.metrics import accuracy_score,
```

```
#computing metrics for current epoch
train_losses.append(all_train_losses_epoch.mean()) #mean loss for all batches
preidctions=(all_Yhat_train_epoch>=0.5) #from probabilities to predictions
acTrain=accuracy_score(all_Y_train_epoch, preidctions)
```

Is 0.5 a good choice for threshold?

Validation Also Batch Wise

```
#validation loop also bacth wise
all_Y_val_epoch=np.array([]).reshape(0,1)
all_Yhat_val_epoch=np.array([]).reshape(0,1)
all_val_losses_epoch=np.array([])
for X_val, Y_val in val_loader:  #batch wise validation set predictions only
    model.eval()

    X_val = X_val.to(device)
    Y_val = Y_val.to(device)

    with torch.no_grad():
        y_hat_val = model(X_val)
        loss = loss_fn(y_hat_val, Y_val)

    #store metrics for all batches of current epoch
    all_Y_val_epoch=np.vstack((all_Y_val_epoch,Y_val.detach().cpu().numpy()))
    all_Yhat_val_epoch=np.vstack((all_Yhat_val_epoch,y_hat_val.detach().cpu().numpy()))
    all_val_losses_epoch=np.append(all_val_losses_epoch,loss.item())


#computing metrics for current epoch
val_losses.append(all_val_losses_epoch.mean()) #mean loss for all batches
preidctions=(all_Yhat_val_epoch>=0.5) #from probabilities to predictions
acVal=accuracy_score(all_Y_val_epoch, preidctions)
```

Model CheckPointing

```
#checkpointing training
if(acVal>best_accuracy):
    checkpoint = {'epoch': epoch,'model_state_dict': model.state_dict(),
                  'optimizer_state_dict': optimizer.state_dict(),'loss': train_losses,
                  'val_loss': val_losses}
    torch.save(checkpoint,'best.pth')
```

```
#loading best model
checkpoint = torch.load('best.pth')
# Restore state for model and optimizer
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
total_epochs = checkpoint['epoch']
losses = checkpoint['loss']
val_losses = checkpoint['val_loss']
```

- **Can we save the whole model object instead?**
- **Any other Advantage of checkpointing?**
- **resuming training**

WHERE ARE THE CURVES?

TensorBoard (Why Tensorflow here?)

- TensorBoard is TensorFlow's visualization toolkit
- PyTorch provides classes and methods for us to integrate it with our model.
- Running on colab and standalone system

```
conda install -c conda-forge tensorboard
```

```
# Load the TensorBoard notebook extension
%load_ext tensorboard

%tensorboard --logdir runs
```

TensorBoard

- It all starts with the creation of a **SummaryWriter** object

```
#tensorboard
tboardWriter=SummaryWriter('runs/simpleClassification')
```

add graph	add scalars	add scalar
add histogram	add images	add image
add figure	add video	add audio

add text	add embedding	add pr curve
add custom scalars	add mesh	add hparams

Plotting Loss and Accuracy Curves using TensorBoard

```
from torch.utils.tensorboard import SummaryWriter
```

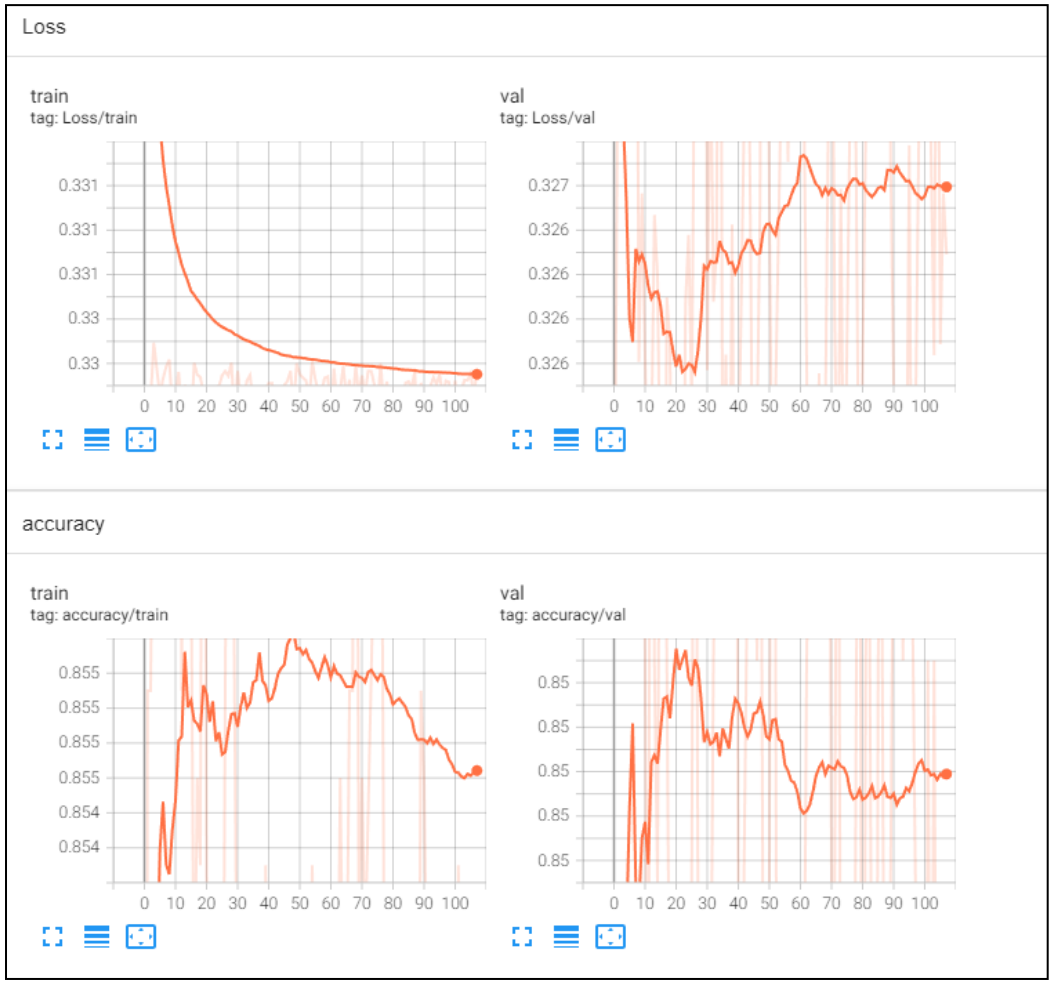
```
#tensorboard
tboardWriter=SummaryWriter('runs/simpleClassification')
```

```
tboardWriter.add_scalar("Loss/train", train_losses[epoch], epoch)
tboardWriter.add_scalar("Loss/val", val_losses[epoch], epoch)
tboardWriter.add_scalar("accuracy/train", acTrain, epoch)
tboardWriter.add_scalar("accuracy/val", acVal, epoch)
```

```
(envpt) C:\WINDOWS\system32>tensorboard --logdir=F:\adeel\DLCourse\week4\runs\simpleClassification
```

TensorBoard Output

```
TensorBoard 2.6.0 at http://localhost:6006/ (Press CTRL+C to quit)
```



Summary

- New Concepts Learned
  - Role of Activation Functions
    - Sigmoid Activation
  - New Loss Function
    - Log loss or logistic loss or BCE
  - Logistic Regression GD
  - Datasets and Dataloaders
  - Feature/Data Scaling
  - Batch wise training
  - Mode Visualization using TorchSummary
  - Evaluation Metrics
  - Model Checkpointing
  - TensorBoard
  - scikit-learn library
- Prepare well by coding each concept yourself
- Practice with provided code with each lecture
  - PyTorch Autograd
  - Binary Classification of moons_dataset

Graded Home Task-2

- Create a binary classification model for sklearn's digits dataset
  - DataSet (X,y) with size (1797,64)
  - Each 64 features are pixels of an 8×8 digit image
  - Output Y vector size is 1797
  - Modify the output vector such that all values with Y>1 becomes 1 (to make it binary classification problem) digit 0 vs all other digits
- Train the model using your own logistic regression model discussed in this lecture
- Take help from this lecture code
- Check in your code + tensorboard graphs