Google Classroom Code: mhxgl24

Classification Metrics, Multiclass Classification, Softmax Activation, Cross-Entropy Loss, Adam

Deep Learning (DS-5006)
Dr. Adeel Mumtaz

Lecture 6

Fall, 2022

National University
Of Computer and Emerging Sciences

Contents

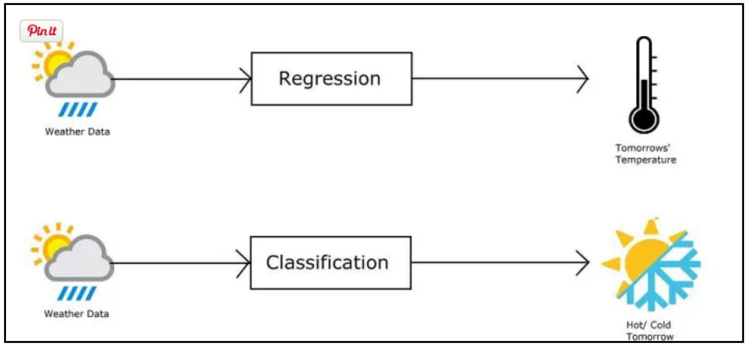
BINARY CLASSIFICATION

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

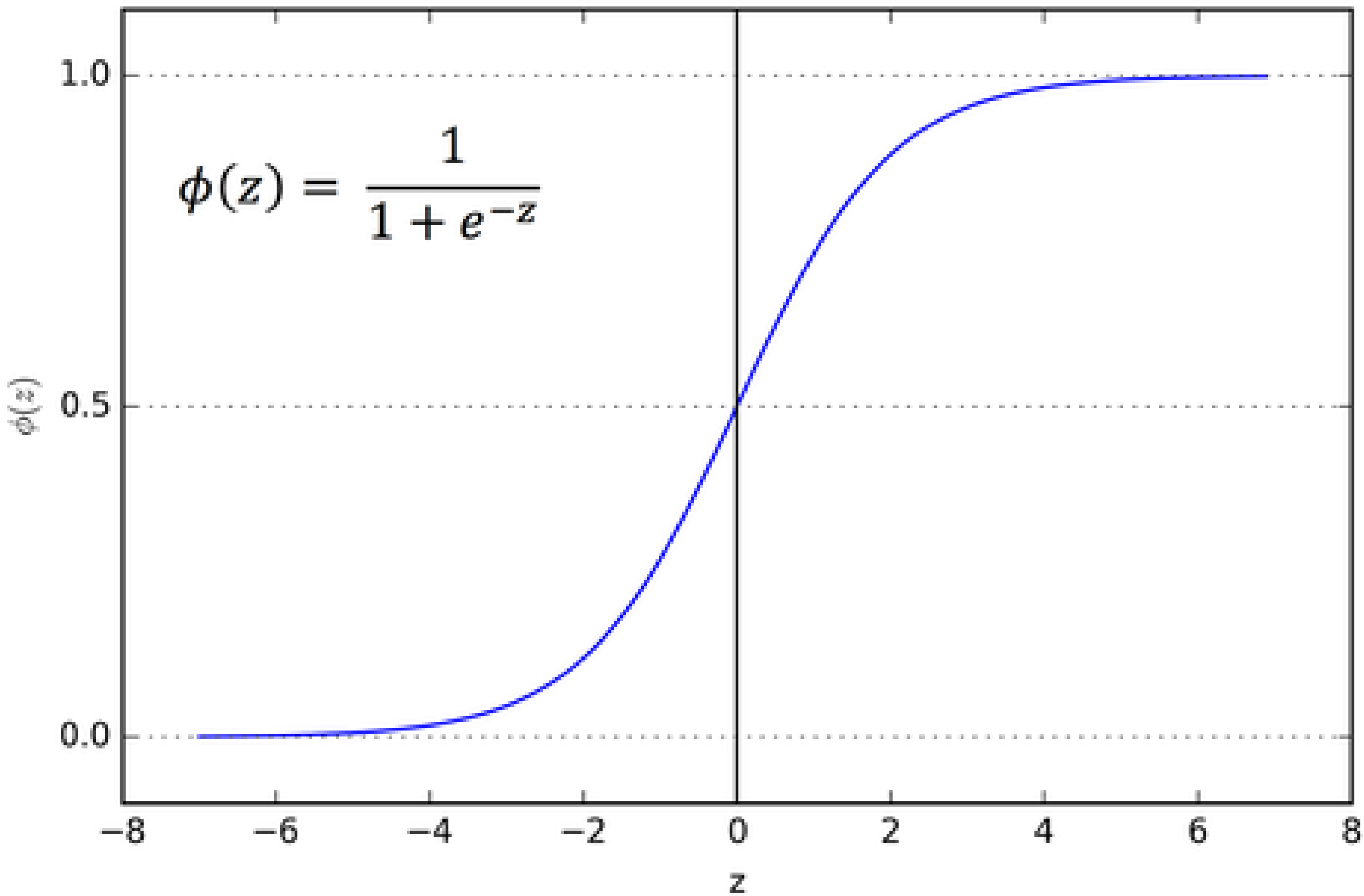
- Logistic Regression
- k-Nearest Neighbors
- Decision Trees
- Support Vector Machine
- Naive Bayes



Binary Classification

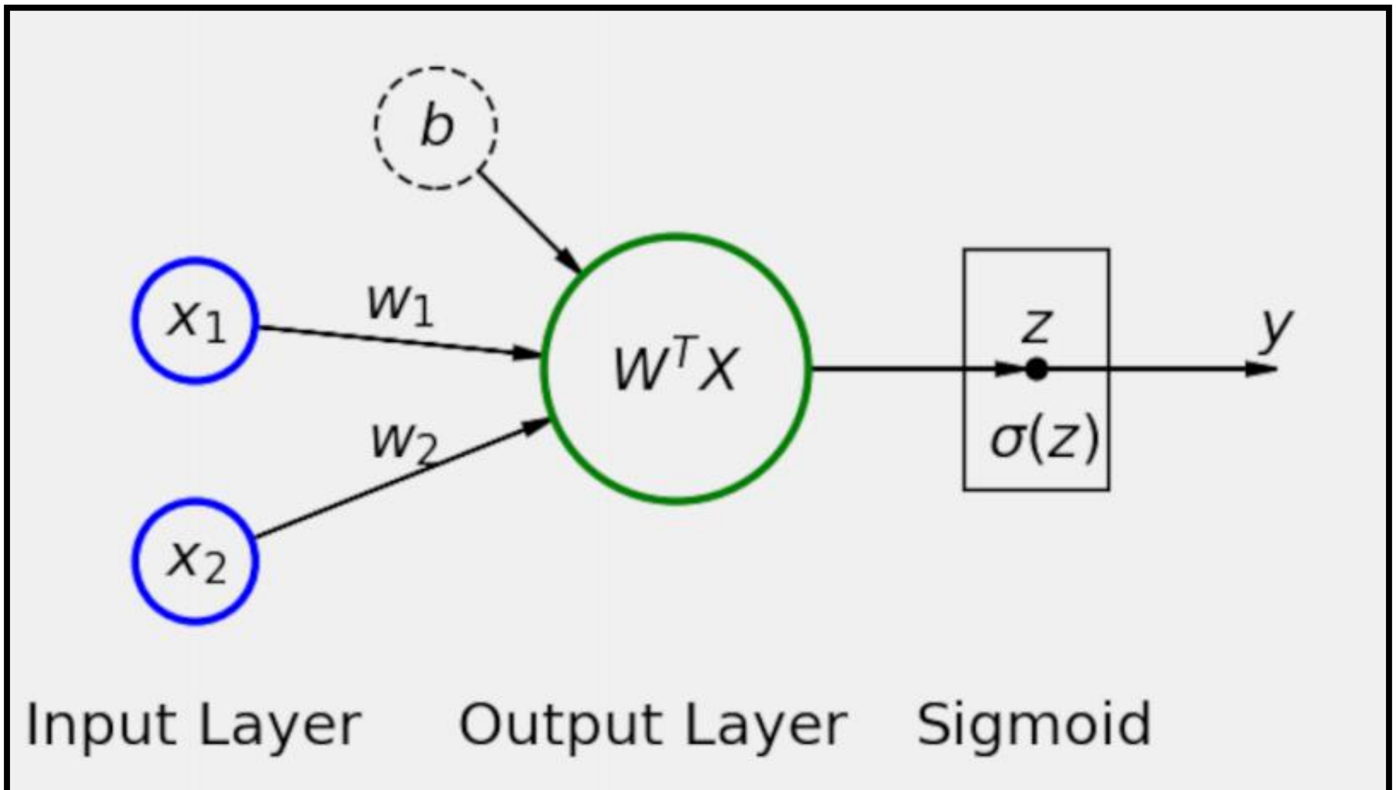
- Why Linear Regression is not suitable for classification?

Solution Sigmoid Activation



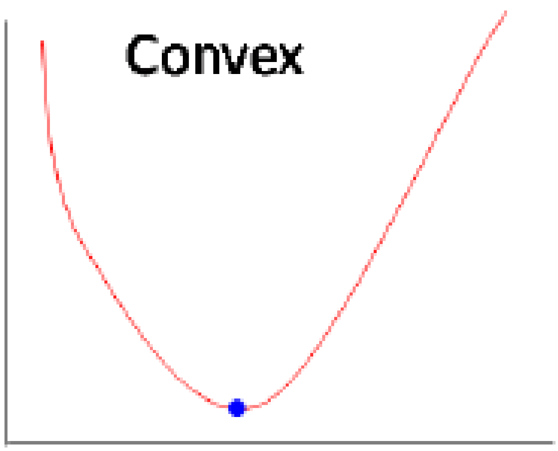
Resulting Logistic Regression Model

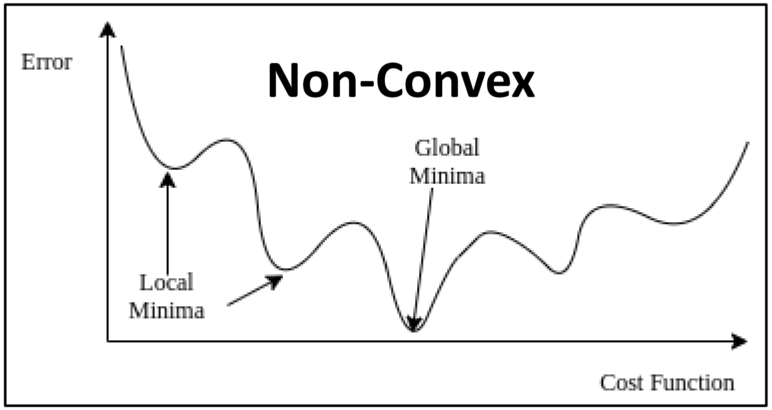
- We can think of the logistic regression as the second simplest neural network possible.
- It is pretty much the same as the linear regression, but with a sigmoid applied to the results of the output layer (z).



Binary Classification with MSE

- Why MSE Loss can't be used for logistic regression
  - 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.





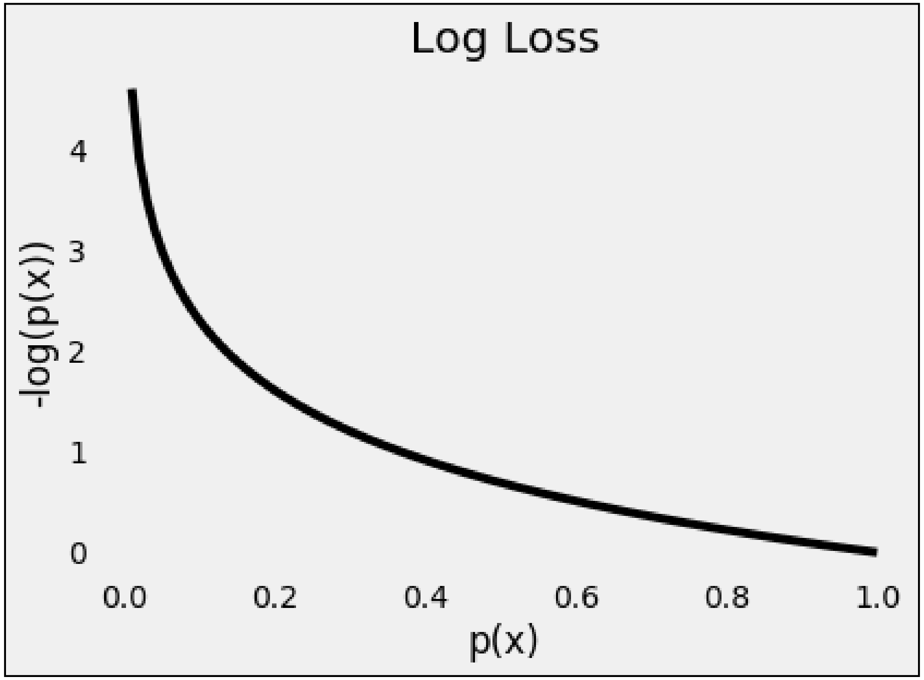
Solution Log loss or Logistic loss

- Combining loss for both classes

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

  Mean Log Loss =

$$-\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{N}} \boldsymbol{y_i}\,\boldsymbol{log}(\hat{\boldsymbol{y}}_{\boldsymbol{i}}) + (\mathbf{1}-\boldsymbol{y_i})\boldsymbol{log}(\mathbf{1}-\hat{\boldsymbol{y}}_{\boldsymbol{i}})$$



Gradient Descent for logistic regression

- $\mathbf{L(W,X)} = -\frac{1}{N}\sum_{i=1}^{N} y^i \, log(\hat{y}_i) + (1-y^i) log(1-\hat{y}_i)$
- $\hat{y}_i = \frac{1}{1+e^{-z}}$
- $Z = W^T x^i$
- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W}$
- $\frac{\partial z}{\partial W} = x^i$
- $\frac{\partial \hat{y}}{\partial z} = \hat{y}_i(1-\hat{y}_i)$ *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

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 **$\mu = 0$ and $\sigma = 1$**

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
```

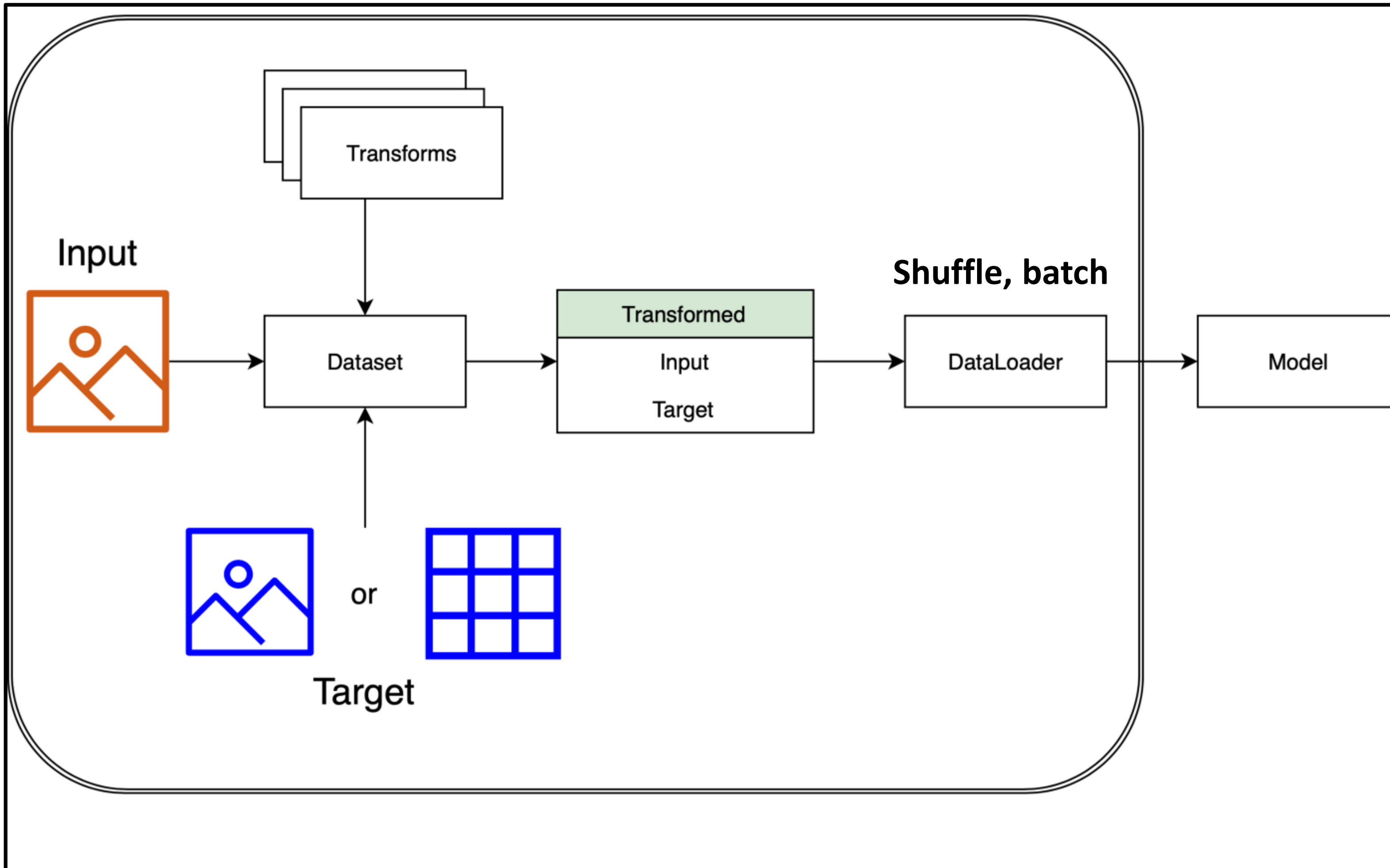
Feature/Data Scaling

```
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



Dataset Example

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

```python
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 Example

```python
#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)
```

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

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

```python
# 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:
```

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
```

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

- $\color{blue}{\boldsymbol{Accuracy}} = \dfrac{\boldsymbol{\#\ of\ correct\ predictions}}{\boldsymbol{total\ number\ of\ predictions}}$
- **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?

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**

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

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

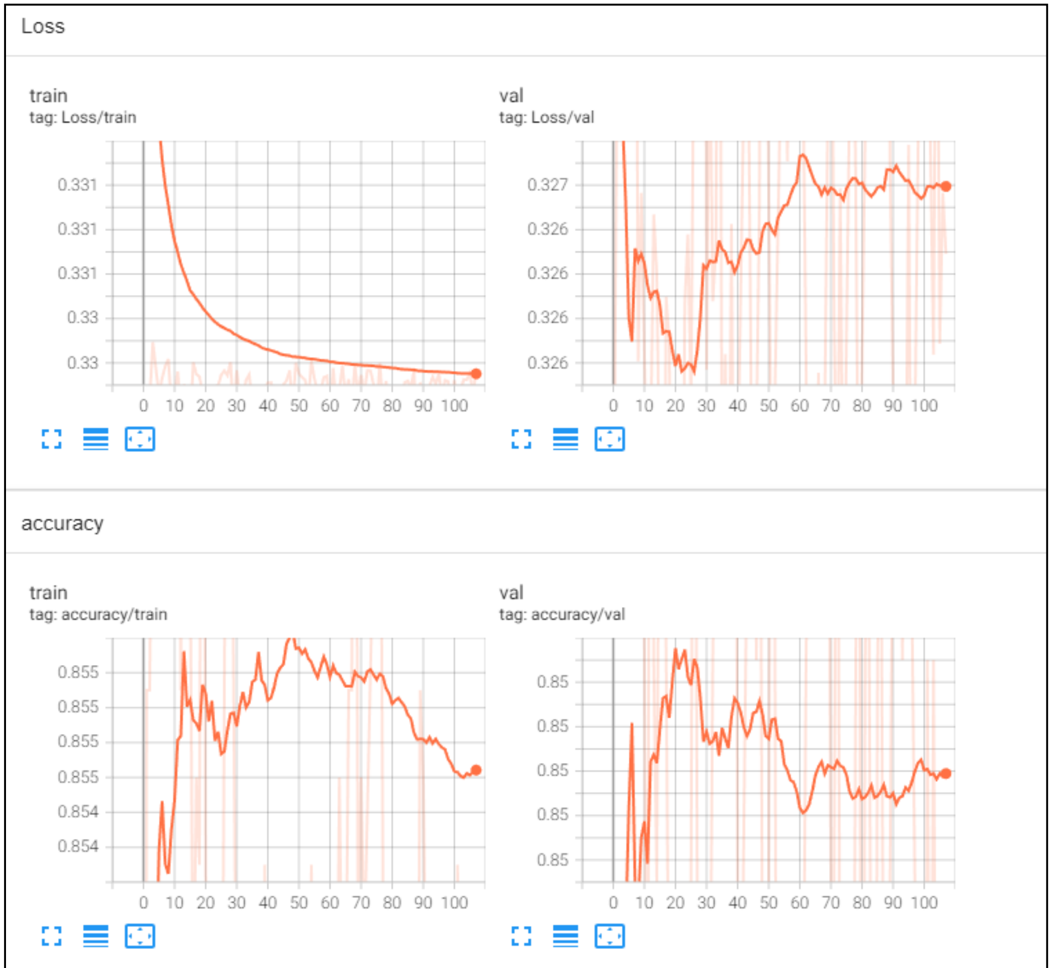
```python
#tensorboard
tboardWriter=SummaryWriter('runs/simpleClassification')
```

```python
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)
```



QUIZ-3

- X$= [-2,\ \ 0.7,\ \ 3]$
- Find $\mathrm{Y} = \sigma(X), where\ \sigma$ is sigmoid function
- Find $\boldsymbol{\frac{\partial Y}{\partial X}}$
- Assuming X as a single feature
  – Standardize X using Standard Scalar
  – Normalize X using MinMax normalization

MORE METRICS

Confusion Matrix

- True Positive
  - The model predicted true and it is true.
- True Negative
  - The model predicted false and it is false.
- False Positive
  - The model predicted True and it is false.
- False Negative
  - The model predicted false and it is true.

Different Versions

A)

		Actual Label	
		1	0
Predicted Label	1	TP	FP
	0	FN	TN

B)

		Actual Label	
		0	1
Predicted Label	0	TN	FN
	1	FP	TP

C)

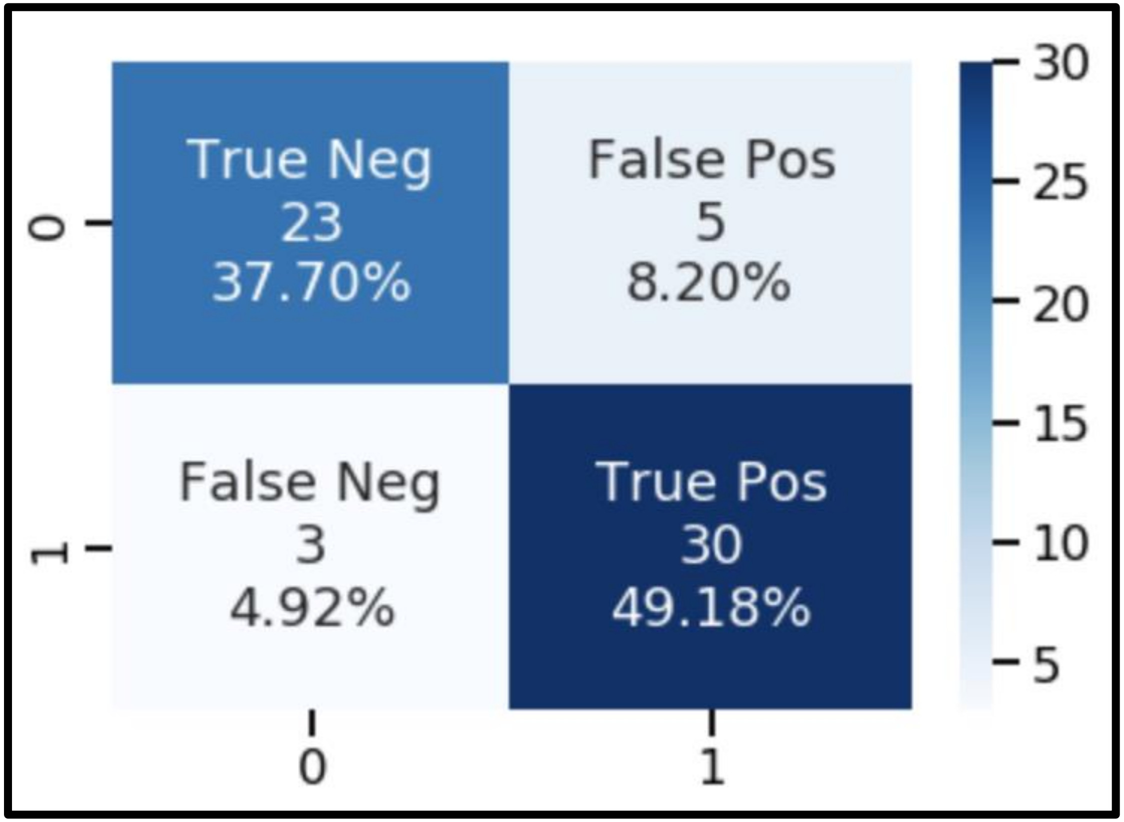
		Predicted Label	
		1	0
Actual Label	1	TP	FN
	0	FP	TN

Scikit Learn

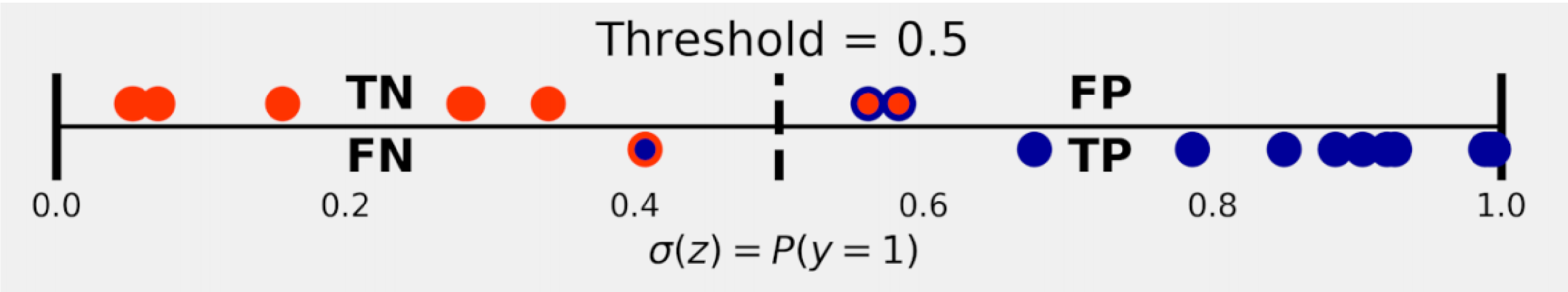
D)

		Predicted Label	
		0	1
Actual Label	0	TN	FP
	1	FN	TP

Example


True Neg
23
37.70%
False Pos
5
8.20%
False Neg
3
4.92%
True Pos
30
49.18%
0
1
0
1
30
25
20
15
10
5

Confusion Matrix Code




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

```
cm50=confusion_matrix(all_Y_val_epoch, all_Yhat_val_epoch>=0.5)
TN, FP, FN, TP = cm50.ravel()
```

True and False Positive Rates

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN}$$

- The **true positive rate** tells you, from all points you know to be positive, how many your model got right
  - If false negatives are bad for your application, you need to focus on **improving** the TPR/recall metric of your model.
  - Take airport security screening, for example, where **positive means the existence of a threat**
    - False positives are common (extra inspection)
    - A false negative means that the machine **failed to detect an actual threat**

True and False Positive Rates

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN}$$

- The **false positive rate** tells you, from all points you know to be negative, how many your model got wrong
  - If false positives are bad for your application, you need to focus on **reducing** FPR metric of your model.
  - Example: Investment decision
    - Positive means a profitable investment.
    - False negatives are missed opportunities: they seemed like bad investments, but they weren't. You did not make a profit, but you didn't sustain any losses either.
    - A false positive means that you chose to invest but ended up losing your money.

Precision and Recall

$$Recall = \frac{TP}{TP + FN} \quad Precision = \frac{TP}{TP + FP}$$

- Recall
  - Same as?
- Precision
  - from all points classified as positive by your model, how many your model got right
    - If false positives are bad for your application , you need to focus on improving the precision metric of your model or ?
    - Target recognition system must have high precision

Accuracy

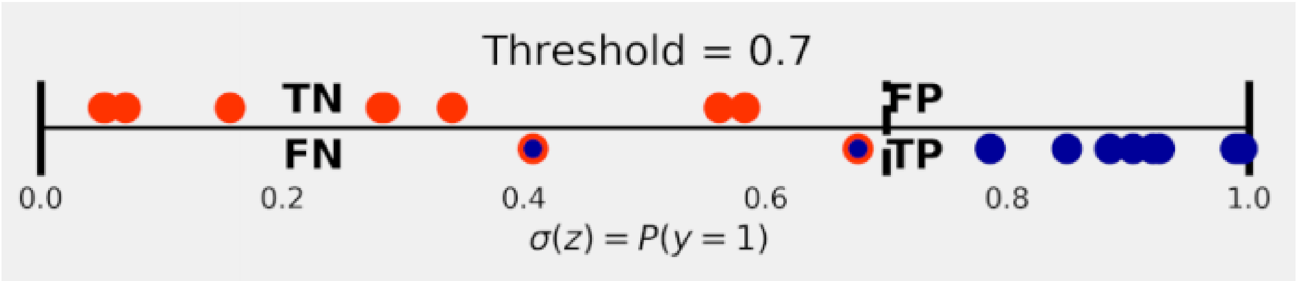
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- How many times your model got it right, considering all data points
- If you have an imbalanced dataset, relying on accuracy can be misleading.

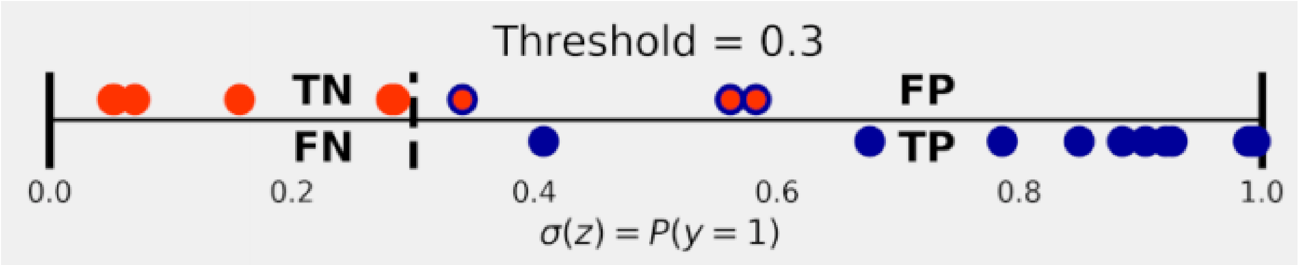
Code

```
cm50=confusion_matrix(all_Y_val_epoch, all_Yhat_val_epoch>=0.5)
TN, FP, FN, TP = cm50.ravel()
TPR=TP/(TP+FN)
FPR=FP/(FP+TN)
precision = TP / (TP + FP)
recall = TP / (TP + FN)
Accuracy = (TP+TN)/(TP+TN+FP+FN)
```

FPR/TPR Tradeoff

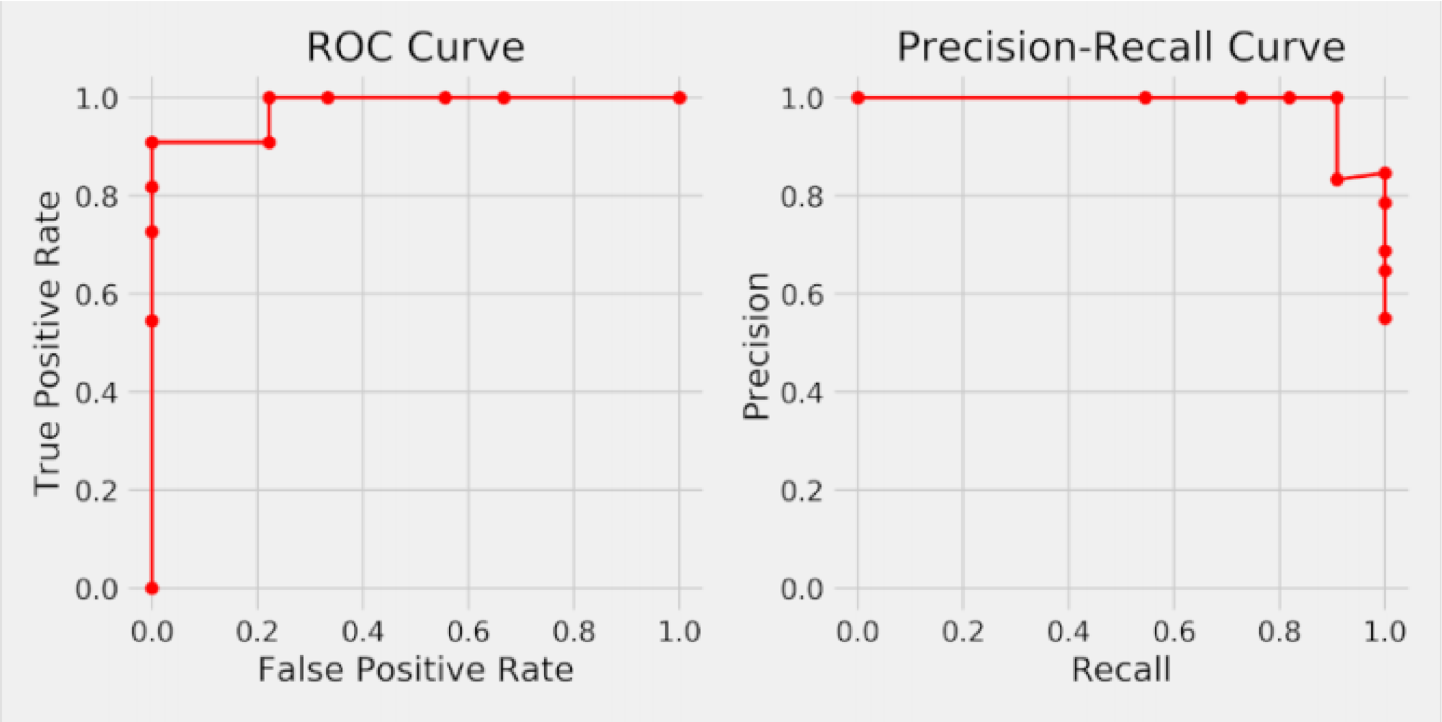


Less false positives, more false negatives



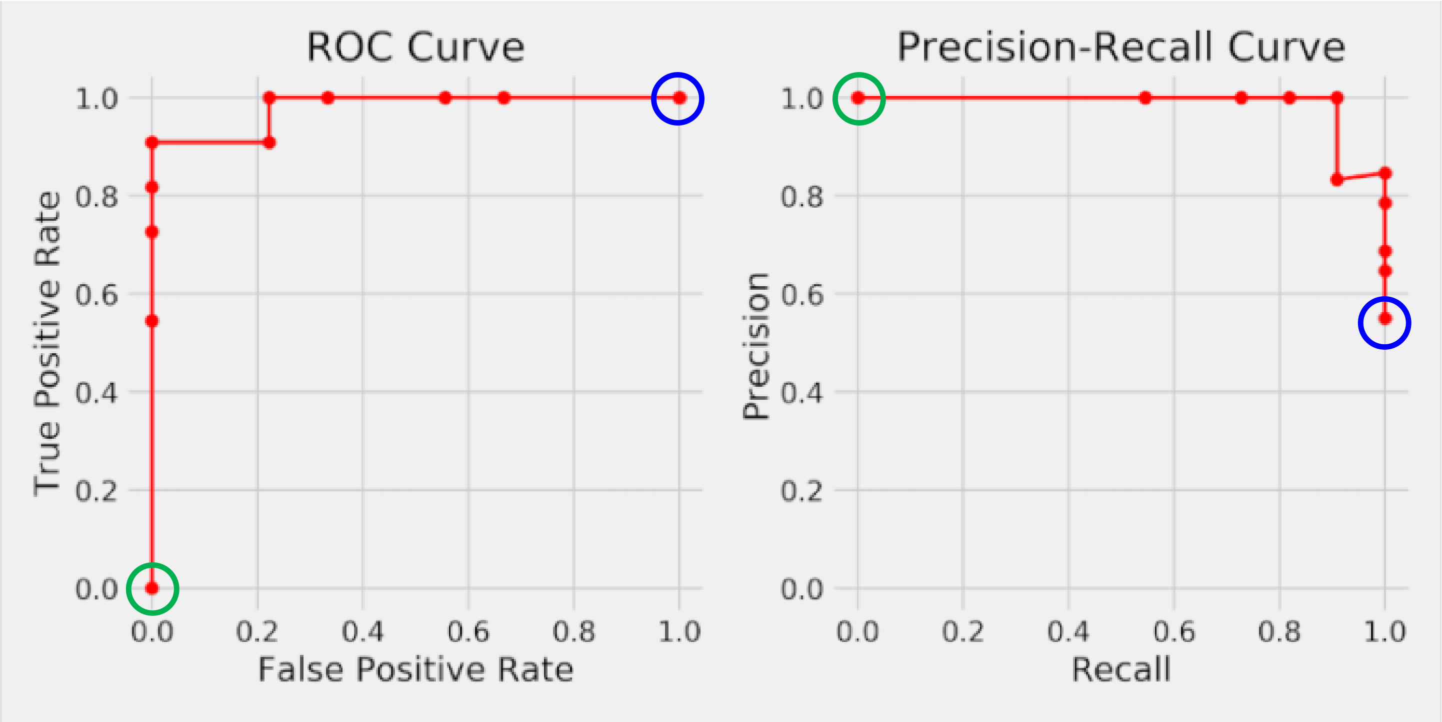
More false positives, less false negatives

ROC & PR Curves (by varying threshold)



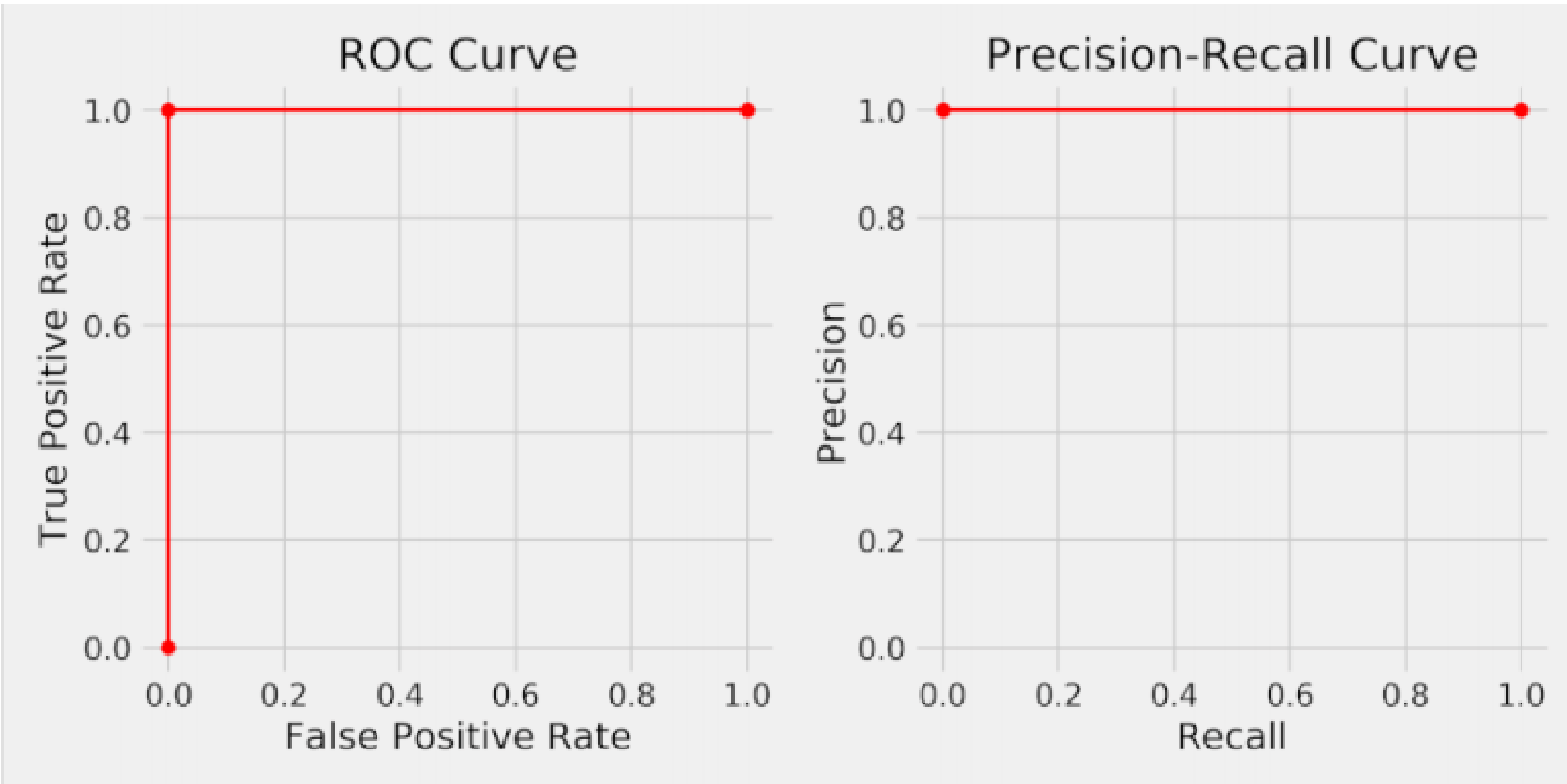
- **Which point corresponds to a threshold of zero (every prediction is positive)?**
- **Which point corresponds to a threshold of one (every prediction is negative)?**
- **What does the right-most point in the PR curve represent?**
- **If I raise the threshold, how do I move along the curve?**

ROC & PR Curves (by varying threshold)

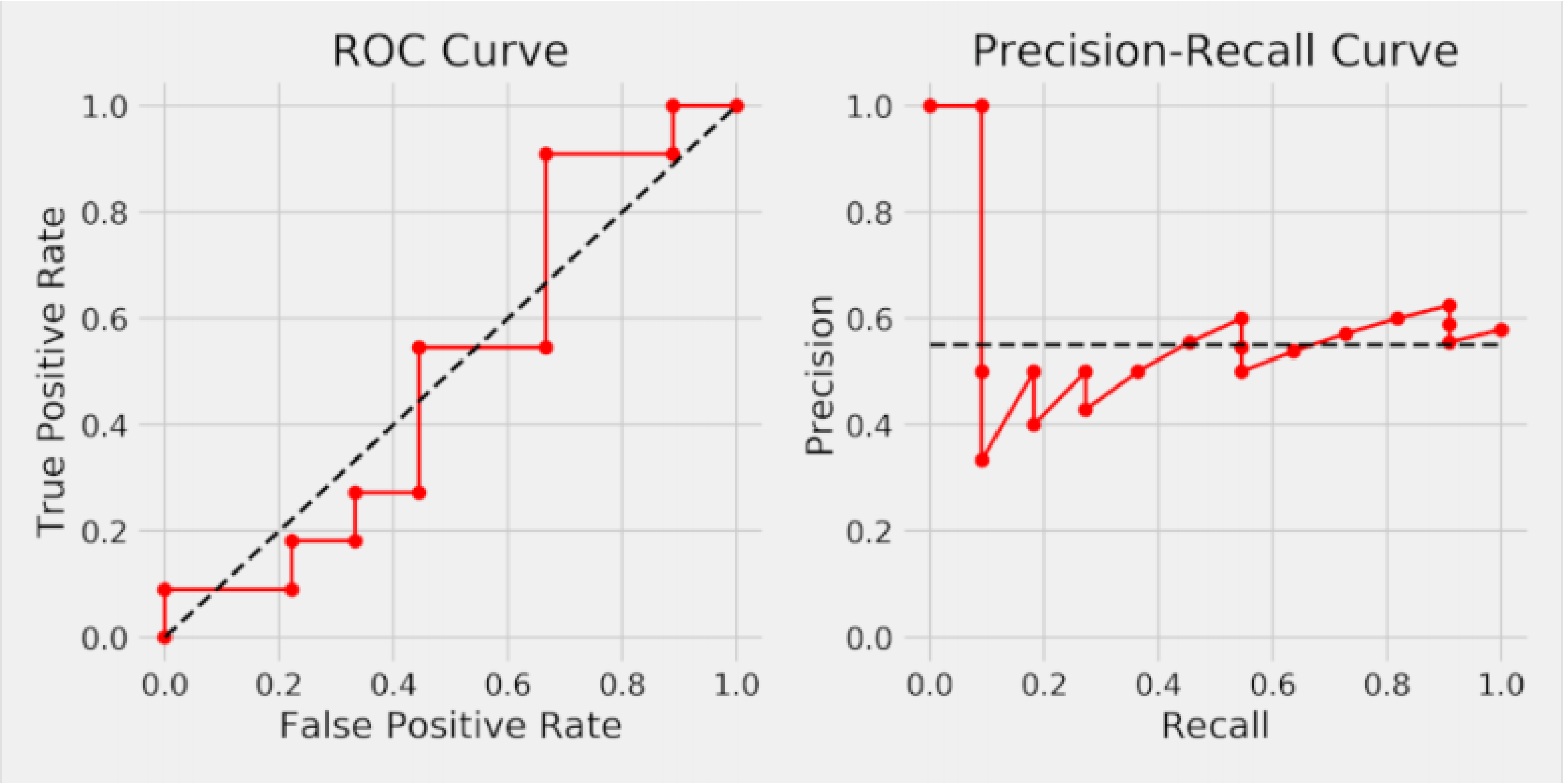


- **Threshold of zero corresponds to the right-most point in both curves**
- **The threshold of one corresponds to the left-most point in both curves**
- **Precision value of the Right-most point in the PR curve represents the proportion of positive examples**
- **if I raise the threshold, I am moving to the left along both curves**

Best Curves (Maximum AUC)



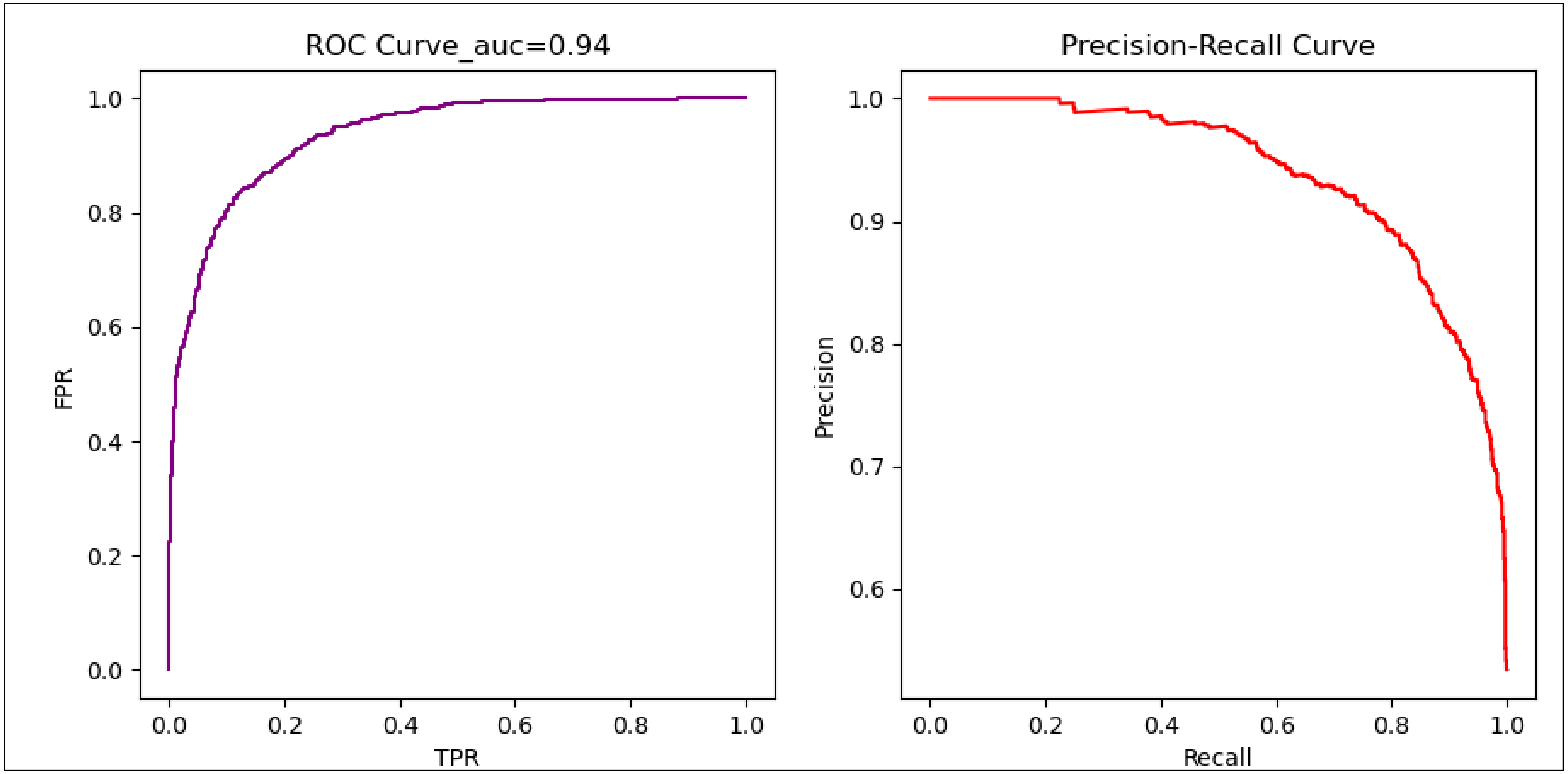
Bad Curves


ROC Curve
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
Precision-Recall Curve
1.0
0.8
0.6
0.4
0.2
0.0
Precision
0.0
0.2
0.4
0.6
0.8
1.0
Recall

Code

```
fpr, tpr, rocthresholds = roc_curve(all_Y_val_epoch, all_Yhat_val_epoch)
precision,recall,prThreshold=precision_recall_curve(all_Y_val_epoch, all_Yhat_val_epoch)
auc = roc_auc_score(all_Y_val_epoch, all_Yhat_val_epoch)

plt.subplot(1,2,1)
plt.plot(fpr, tpr, color='purple')
plt.title('ROC Curve'+'_auc='+str("{:.2f}".format(auc)))
plt.ylabel('FPR')
plt.xlabel('TPR')

plt.subplot(1,2,2)
plt.plot(recall, precision, color='red')
plt.title('Precision-Recall Curve')
plt.ylabel('Precision')
plt.xlabel('Recall')
plt.show()
plt.pause(0.1)
```

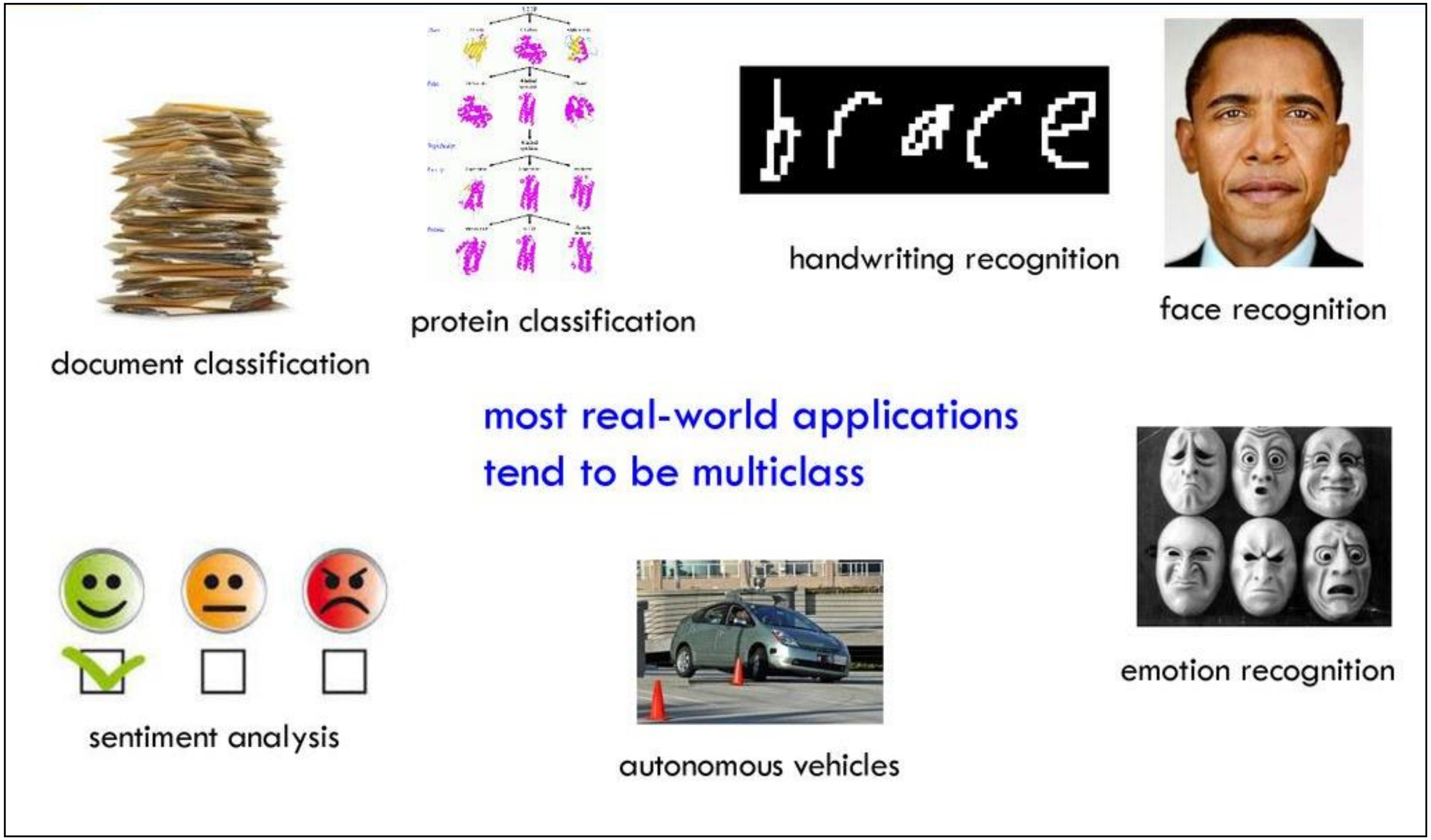
Results (moons data)



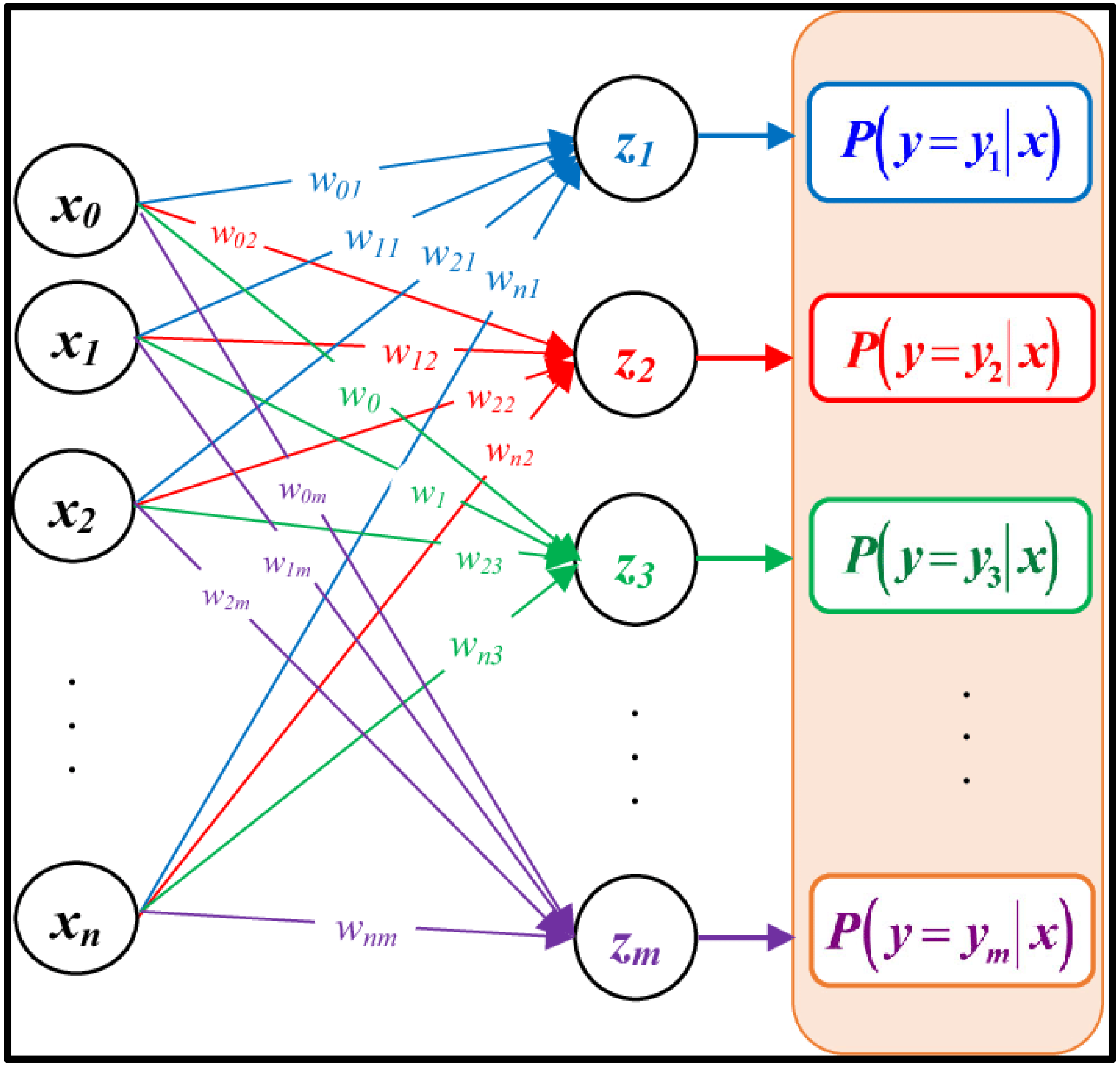
MULTICLASS CLASSIFICATION (THEORY)

Multiclass Classification

- A problem is considered a multiclass classification problem if there are more than two classes



Architecture

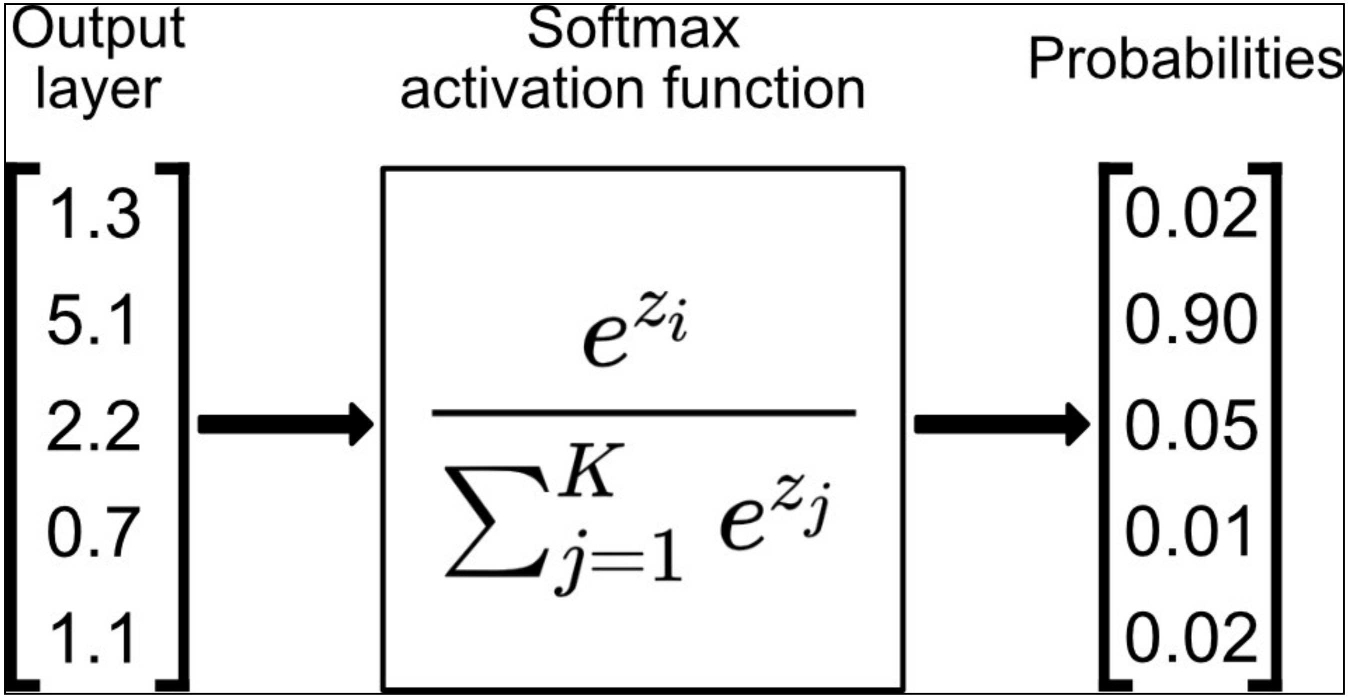


Differences with Binary Classification:

1. Output layer has **m neurons** each having its own logit
2. Output of network has **m probability** values
3. **Can we use Sigmoid?**
4. **Can we use BCELoss?**

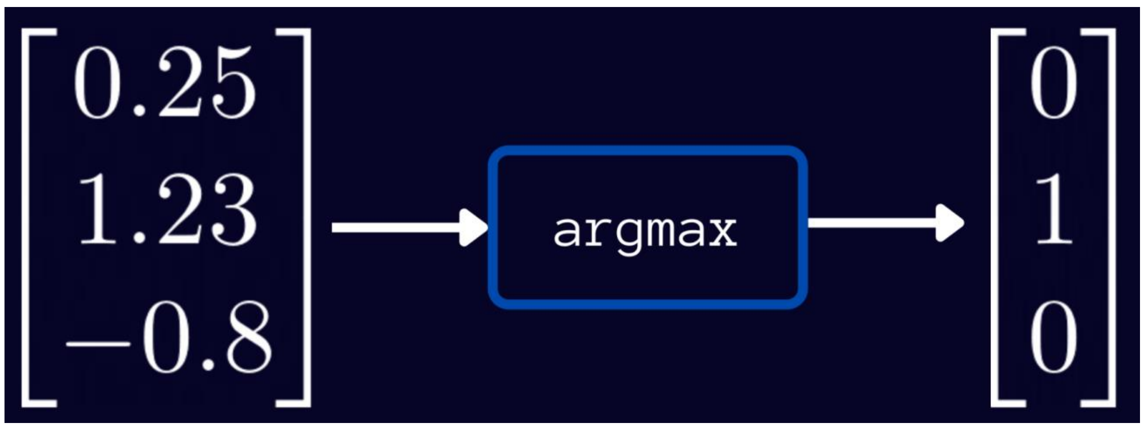
Softmax Activation

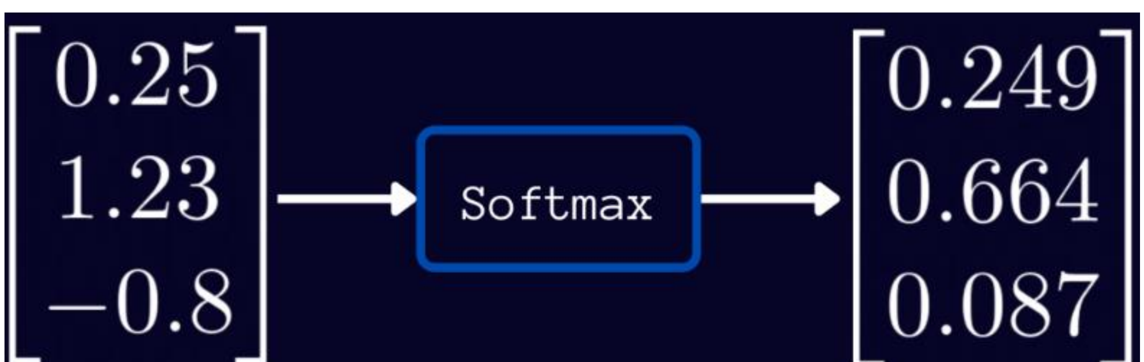
- Softmax is a mathematical function that converts a vector of numbers into a **vector of probabilities**



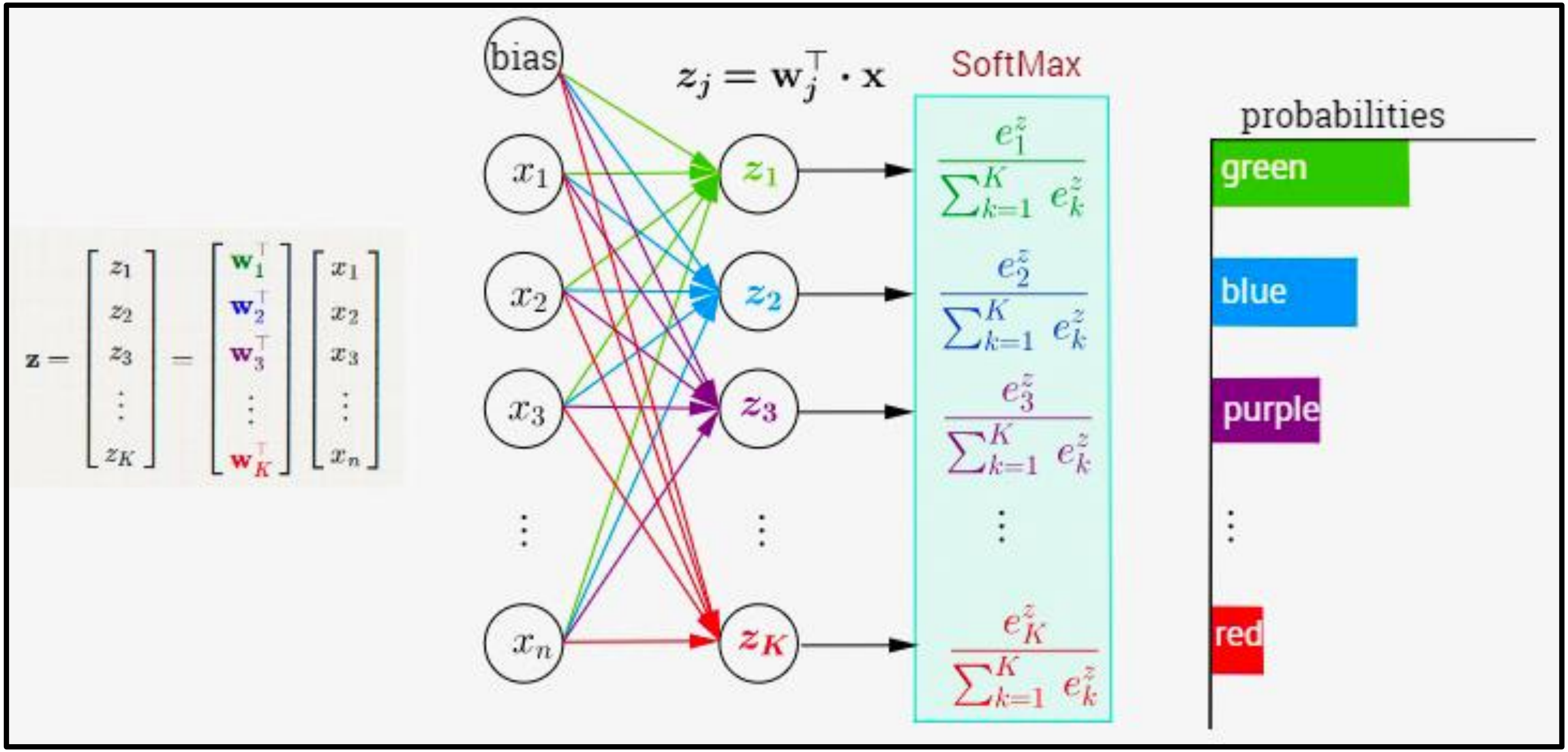
Softmax Vs Argmax

- Why argmax is not a good candidate
  - Recall gradients





Architecture with Softmax



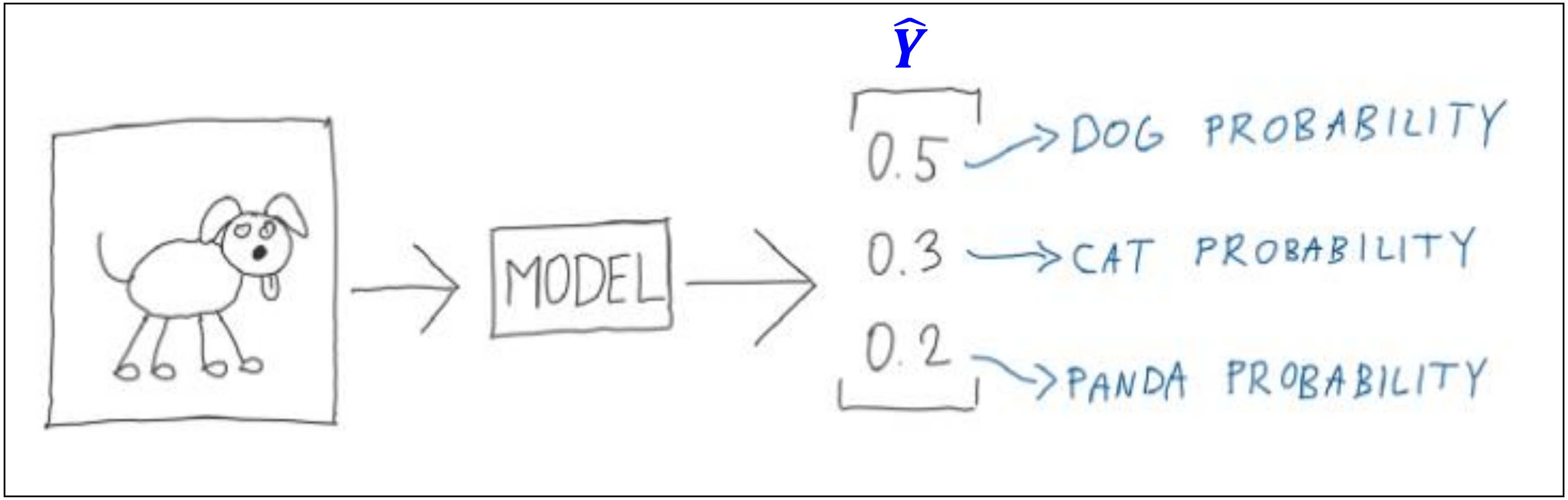
PyTorch Implementation of Softmax

- **torch.nn.Softmax** as a layer
- **torch.nn.functional.softmax** as a function

```python
import torch
import torch.nn.functional as F
import torch. nn as nn

dummy_logits_for_batch = torch.tensor([[21,-5,0.5],[7,8,-10]]) #shape (2,3)

#Softmax as a layer
softmaxLayer = nn.Softmax(dim=-1)
probabilities_yhat = softmaxLayer(dummy_logits_for_batch) #shape (2,3)
print(probabilities_yhat)
test_sum_each_row = torch.sum(probabilities_yhat,dim=1)  #shape (2)

#softmax as a function
probabilities_yhat=F.softmax(dummy_logits_for_batch,dim=-1) #shape (2,3)
test_sum_each_row = torch.sum(probabilities_yhat,dim=1)  #shape (2)
print(probabilities_yhat)
```

Loss Function for Multiclass Classification

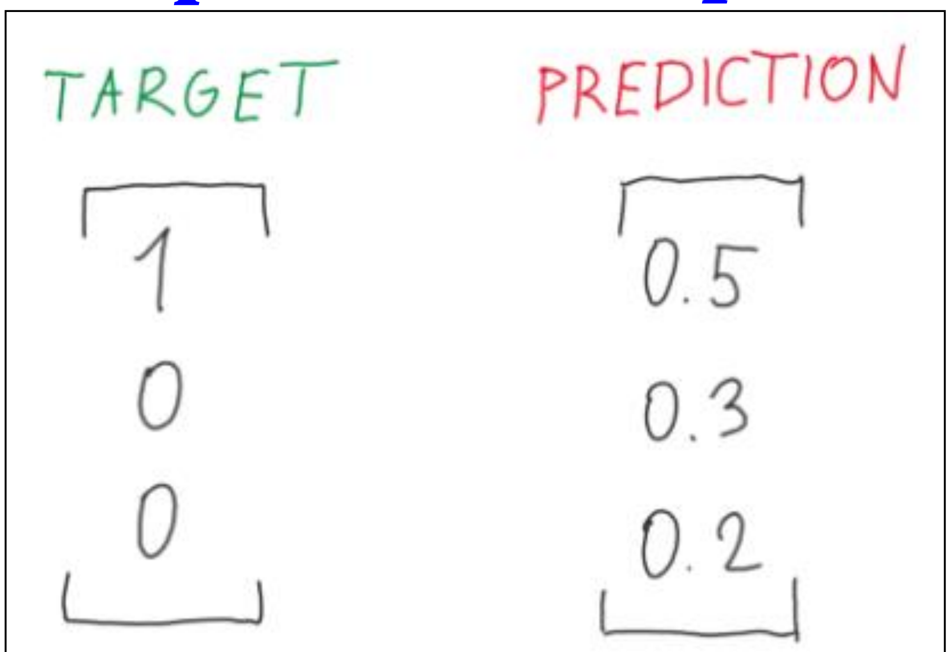
- How does the target (Y) and predicted ($\hat{Y}$) looks now?



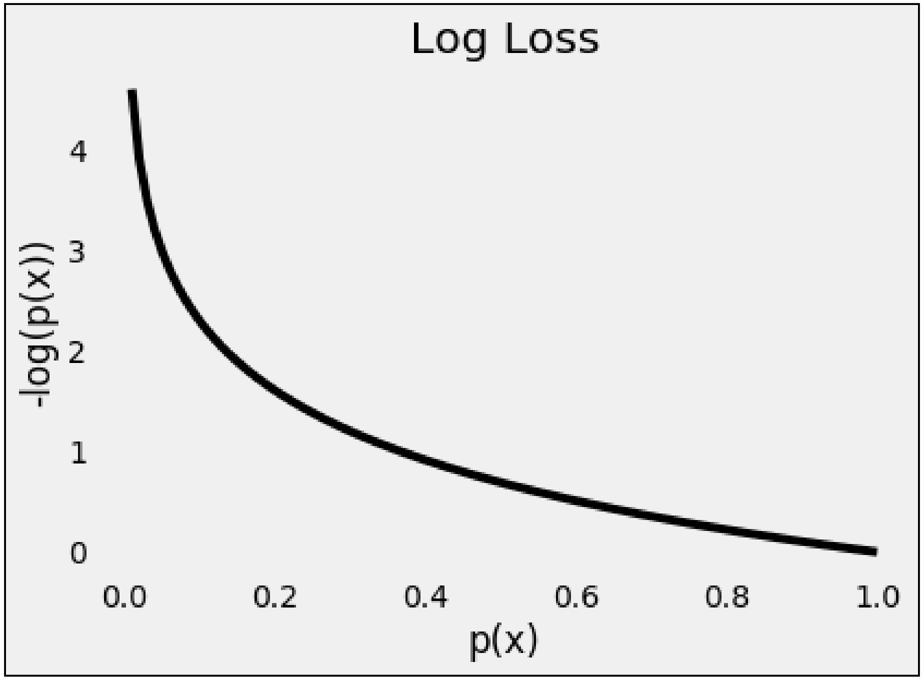
Loss Function for Multiclass Classification

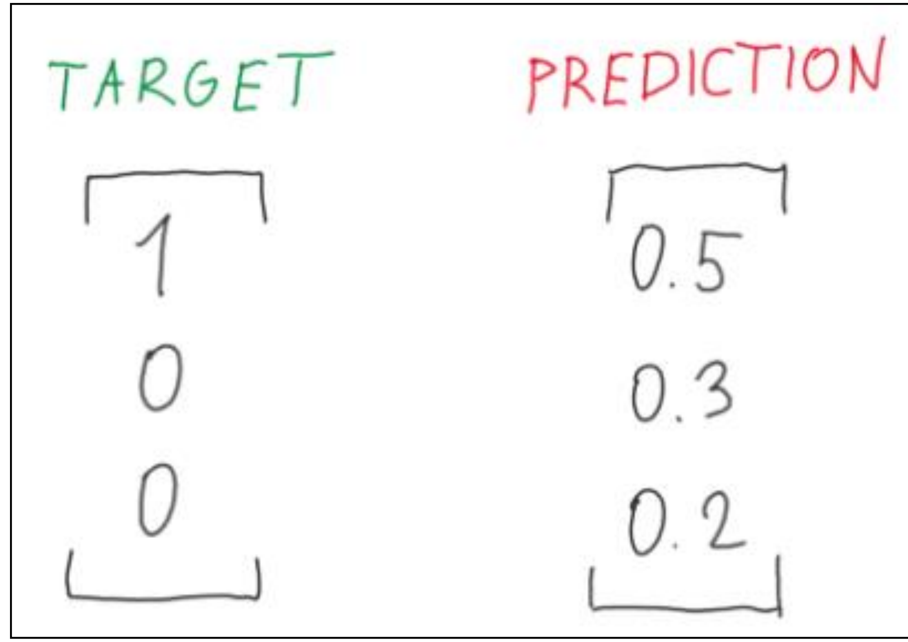
- How does the target (Y) and predicted ($\hat{Y}$) looks now?

$\boldsymbol{Y^i}$ $\qquad\qquad$ $\hat{\boldsymbol{Y}}^{\boldsymbol{i}}$



Remember $-\log(\hat{Y})$





Loss for class X = $-Y log(\hat{Y})$

Loss for class Dog = $-1 * log(0.5) = 0.69$

Loss for class Cat= ?

Loss for class Panda =?

Cross-Entropy (CE) or negative log-likelihood (NLL) Loss

Three Classes

$$NLLLoss(y) = -\frac{1}{(N_0 + N_1 + N_2)}\left[\sum_{i=1}^{N_0} log(P(y_i = 0)) + \sum_{i=1}^{N_1} log(P(y_i = 1)) \sum_{i=1}^{N_2} log(P(y_i = 2))\right]$$

$$\boxed{\text{cross-entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} t_{i,j}\log(p_{i,j})}$$

Note:

$t_{i,j}$ is the target value (Y) from the one hot vector of ith training example and jth class

$p_{i,j}$ is the predicted value ($\hat{Y}$) from the logit vector of ith training example and jth class

Cross-Entropy Loss in PyTorch

- Option-1: **nn.CrossEntropyLoss**
- Option-2: **nn.LogSoftmax + nn.NLLLoss**

```
import torch
import torch.nn.functional as F
import torch. nn as nn

torch.manual_seed(11)
dummy__batch_of_logits_Z = torch.randn((5, 3))
dummy_labels_Y = torch.tensor([0, 0, 1, 2, 1])


#option-1 logits to Loss (no need for softmax layer)
loss_fn = nn.CrossEntropyLoss()
loss=loss_fn(dummy__batch_of_logits_Z, dummy_labels_Y)
print(loss)


#option-2 log(Yhat) to Loss (last layer of network as LogSoftMax)
logSoftmaxLayer=nn.LogSoftmax(dim=-1)
log__batch_Yhat=logSoftmaxLayer(dummy__batch_of_logits_Z)
loss_fn = nn.NLLLoss()
loss=loss_fn(log__batch_Yhat, dummy_labels_Y)
print(loss)
```

Summary (Activation + Loss)

	Activation Function	Loss Function
Binary Classification	Sigmoid	BCELoss
	No activation (logits only)	BCEWithLogitsLoss
Multiclass Classification	LogSoftmax	NLLLoss
	No activation (logits only)	CrossEntropyLoss

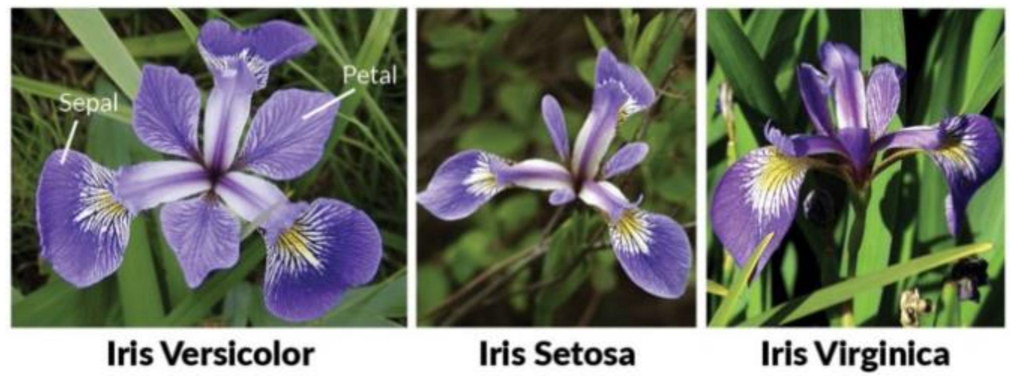
Be careful and choose right activation for a selected Loss function

Putting it All Together

MULTICLASS CLASSIFICATION (CODE)

Iris Flowers Classification Toy Dataset

- Multi-class classification problem based on Iris data set
  - Features
    - sepal length (cm)
    - sepal width (cm)
    - petal length (cm)
    - petal width (cm)
  - Classes
    - Iris-setosa
    - Iris-versicolour
    - Iris-virginica
  - 150 examples (50 for each class)



Loading Dataset

```
from sklearn.datasets import load_iris
class IRISDataset(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)

iris = load_iris()
X = iris['data']
y = iris['target']
names = iris['target_names']
feature_names = iris['feature_names']

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=.2, random_state=13)
#preprocessing normalizing the features (mean=0, var=1)
sc = StandardScaler()
sc.fit(X_train)  #note only from training data
```

Transfoms,Dataset, DataLoaders

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

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 = IRISDataset(x_train_tensor, y_train_tensor)
val_dataset = IRISDataset(x_val_tensor, y_val_tensor)
# Builds a loader of each set
train_loader = DataLoader(dataset=train_dataset, batch_size=10, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=10)
test_batch=next(iter(train_loader))
total_batches_one_epoch = len(iter(train_loader))
```

Model,Loss,Optimizer

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

    def __init__(self):
        super().__init__()
        self.linearLayer1 = nn.Linear(4,50) #hidden layer
        self.linearLayer2 = nn.Linear(50,100) #hidden layer
        self.relu = nn.ReLU()
        self.linearLayer3 = nn.Linear(100,3) #hidden layer

    def forward(self,x):
        u=self.linearLayer1(x)
        v=self.relu(u)
        w=self.linearLayer2(v)
        m=self.relu(w)
        z=self.linearLayer3(m)
        return z
```

```
lr = 0.1
optimizer = optim.SGD(model.parameters(), lr=lr)
loss_fn = nn.CrossEntropyLoss()
```

Training Loop

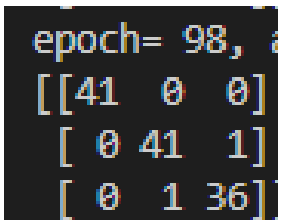
```
#batch wise training loop
epochs = 100
train_losses = []
val_losses = []
best_accuracy=0
for epoch in range(epochs):  #epochs loop

    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)
        logits = model(X_train)

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

        #store metrics for all batches of current epoch
        y_hat=F.softmax(logits,dim=-1)
        y_hat=y_hat.detach().cpu().numpy()
        y_hat=np.argmax(y_hat,axis=1)
        y_hat=y_hat.reshape(-1,1)

        Y_train=Y_train.detach().cpu().numpy()
        Y_train=Y_train.reshape(-1,1)
        all_Y_train_epoch=np.vstack((all_Y_train_epoch,Y_train))
        all_Yhat_train_epoch=np.vstack((all_Yhat_train_epoch,y_hat))
        all_train_losses_epoch=np.append(all_train_losses_epoch,loss.item())



    #computing metrics for current epoch
    train_losses.append(all_train_losses_epoch.mean()) #mean loss for all batches
    acTrain=accuracy_score(all_Y_train_epoch, all_Yhat_train_epoch)
    cmTrain=confusion_matrix(all_Y_train_epoch, all_Yhat_train_epoch)
    print(cmTrain)
```

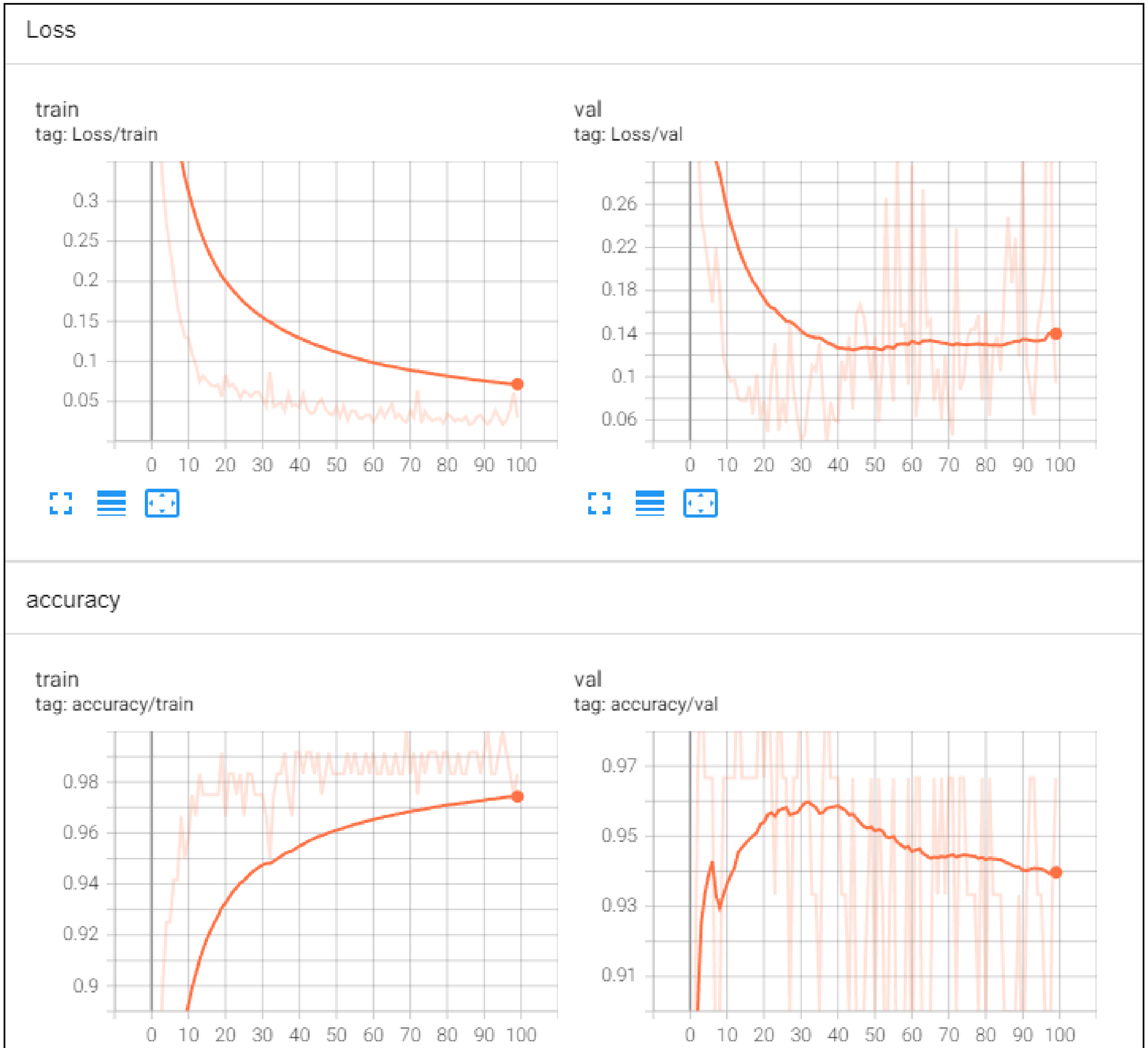
Check:

- **how we got probabilities from logits Using Softmax (which is not part of Model Class (why?))**
- **How we got predictions from probabilities using argmax**

Results

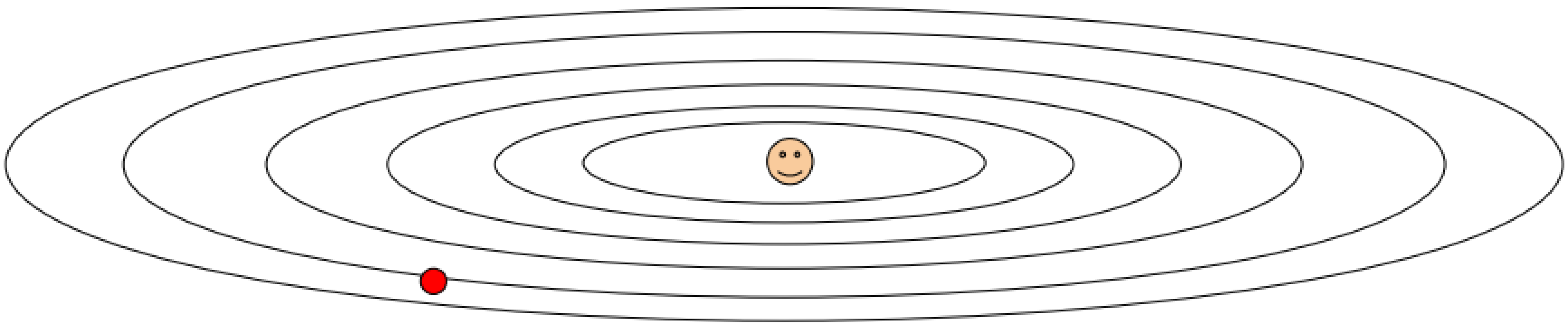
Confusion Matrix





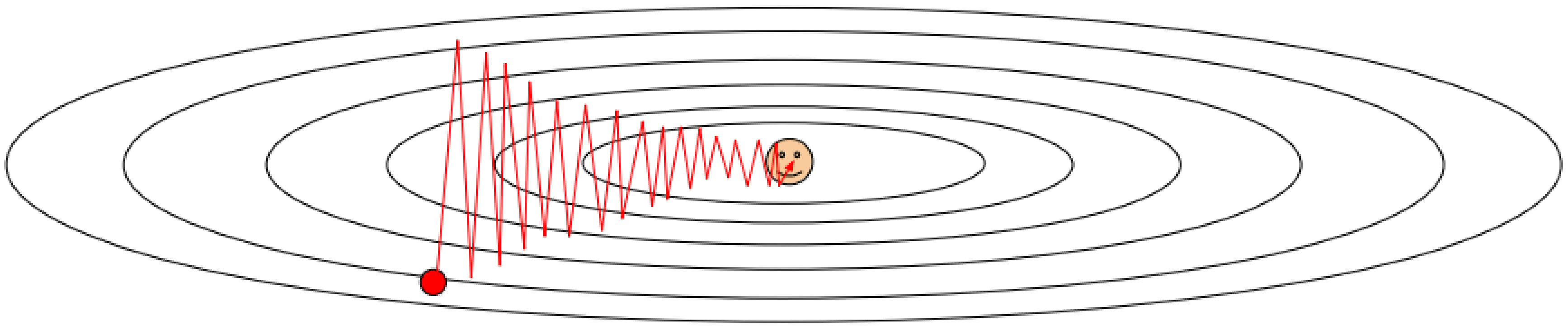
MORE OPTIMIZERS

1. Problems with SGD (When Data and Parameters are large, it can be **slow**)



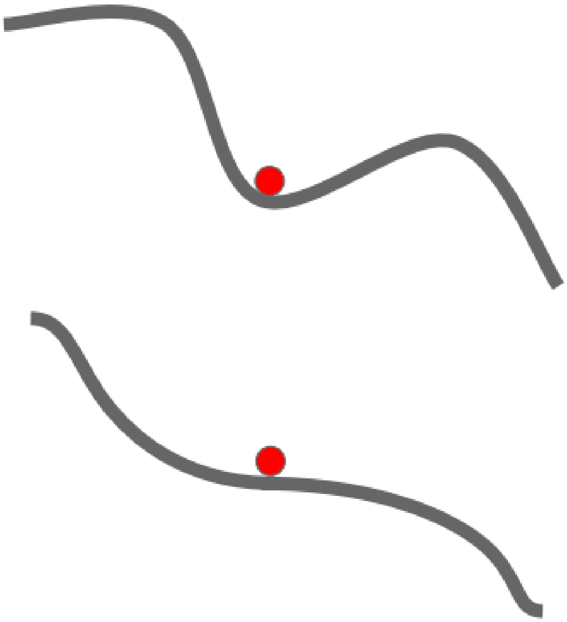
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

1. Problems with SGD (When Data and Parameters are large, it can be **slow**)



What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

2. Problems with SGD (It can **stuck**)



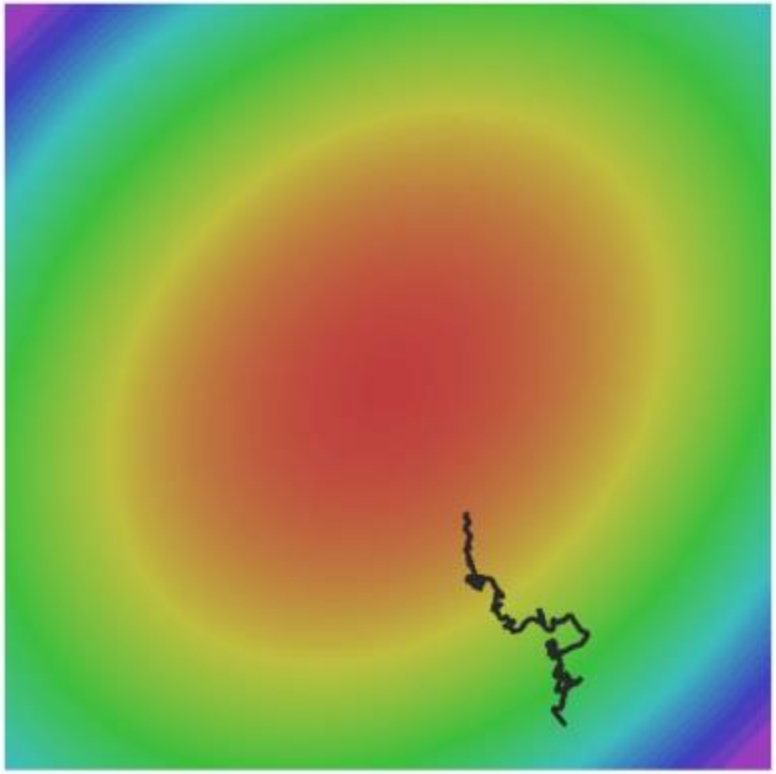
What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck

3. Problems with SGD (It can be **Noisy**)

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Our gradients come from minibatches: **so they can be noisy!**

SGD with Momentum

- Extension of SGD that accelerate the gradient descent algorithm by taking into consideration the **weighted average** of the gradients.

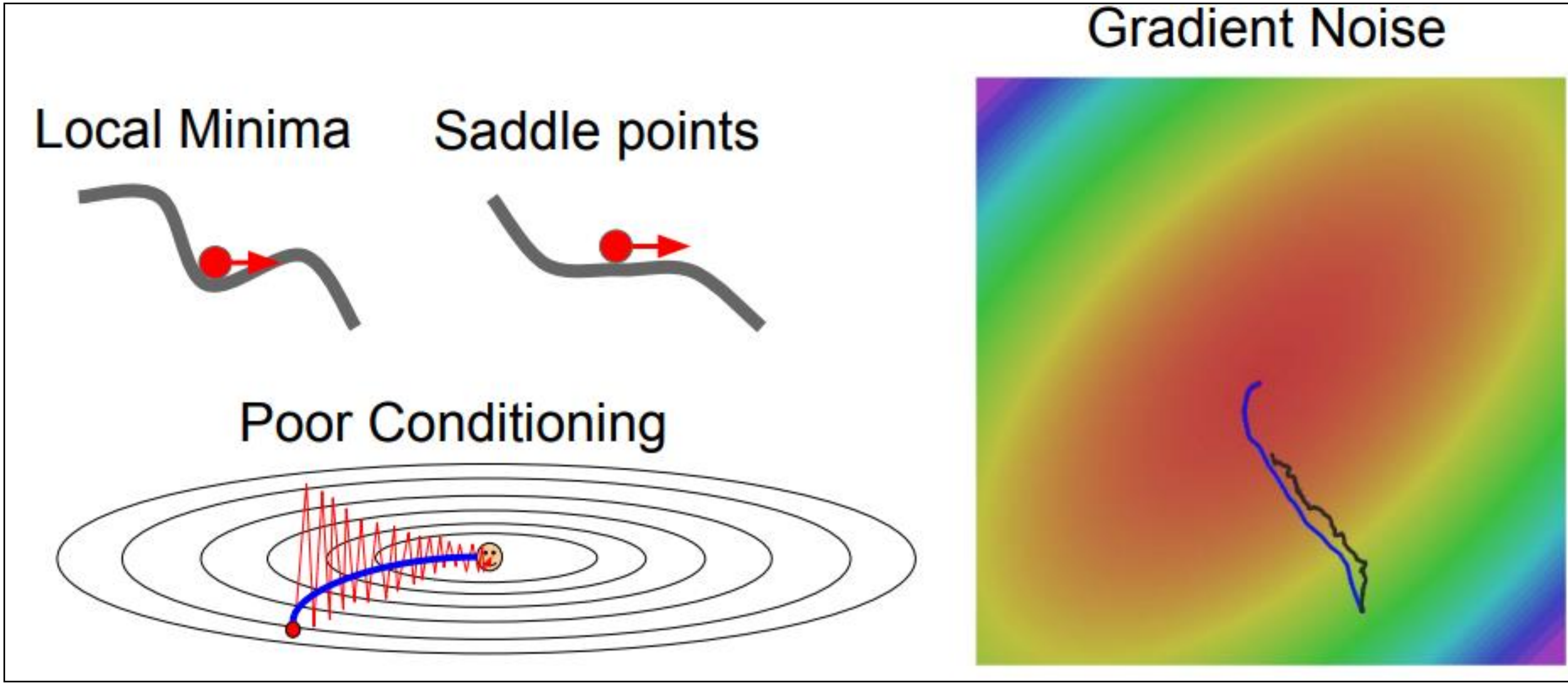
$$w_{t+1} = w_t - \alpha m_t$$

where,

$$m_t = \beta m_{t-1} + (1 - \beta)\left[\frac{\delta L}{\delta w_t}\right]$$

```
mₜ = aggregate of gradients at time t [current] (initially, mₜ = 0)
mₜ₋₁ = aggregate of gradients at time t-1 [previous]
Wₜ = weights at time t
Wₜ₊₁ = weights at time t+1
αₜ = learning rate at time t
∂L = derivative of Loss Function
∂Wₜ = derivative of weights at time t
β = Moving average parameter (const, 0.9)
```

SGD with Momentum



RMSProp

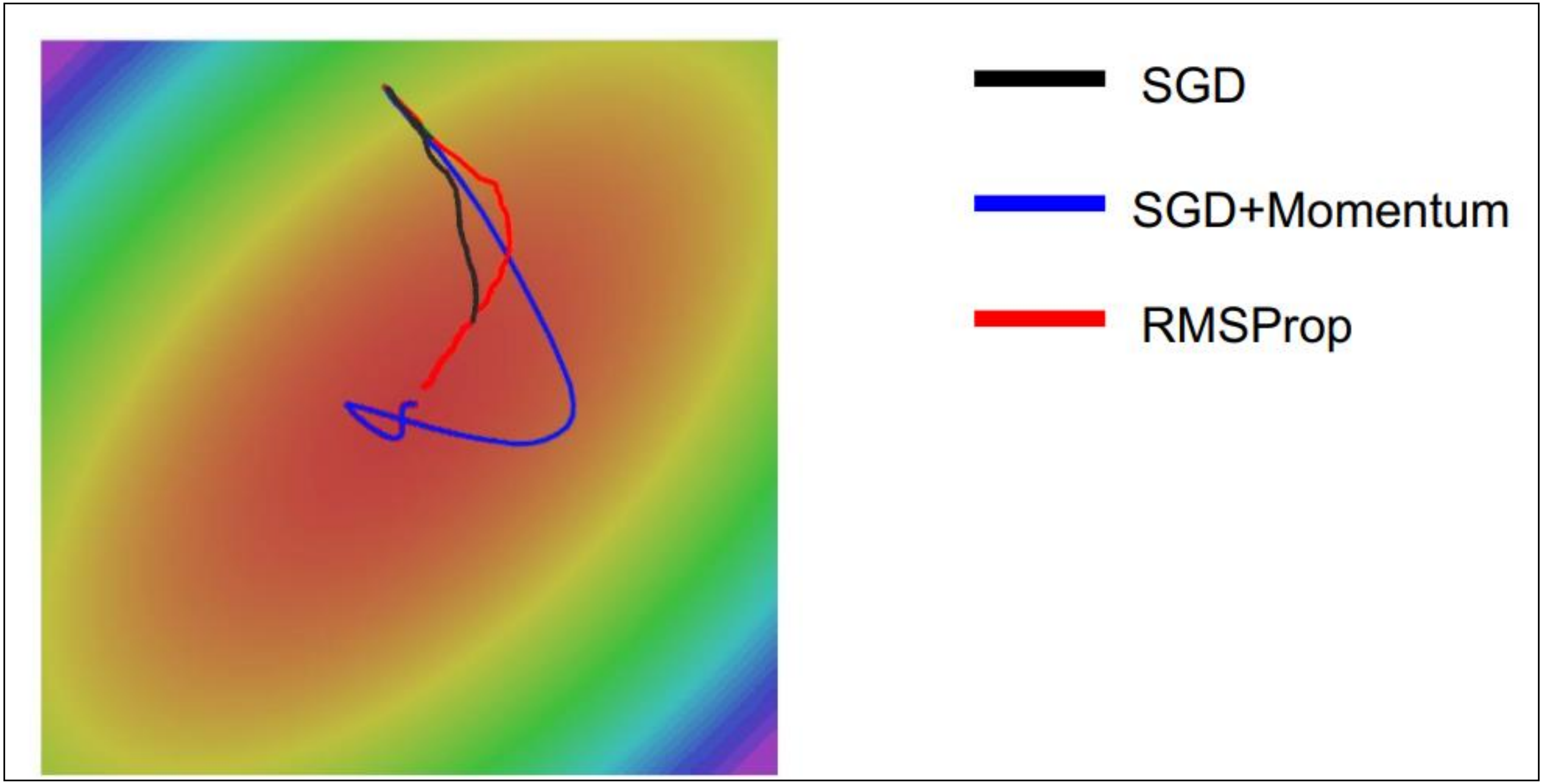
- Momentum has the effect of **dampening down** the change in the gradient and, in turn, the step size with each new point in the search space.
- RMSProp maintains a moving average of the squares of the recent gradients

$$v_t = \beta v_{t-1} + (1-\beta) * \left[\frac{\delta L}{\delta w_t}\right]^2$$

$$w_{t+1} = w_t - \frac{\alpha_t}{(v_t+\varepsilon)^{1/2}} * \left[\frac{\delta L}{\delta w_t}\right]$$

```
Vt = sum of square of past gradients. [i.e sum(∂L/∂Wt-1)] (initially, Vt = 0)
β = Moving average parameter (const, 0.9)
```

RMSProp



Adam (Adaptive moment Estimation)

- Adam Optimizer inherits the strengths or the positive attributes of the above two methods

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\left[\frac{\delta L}{\delta w_t}\right] \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} \qquad \widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$

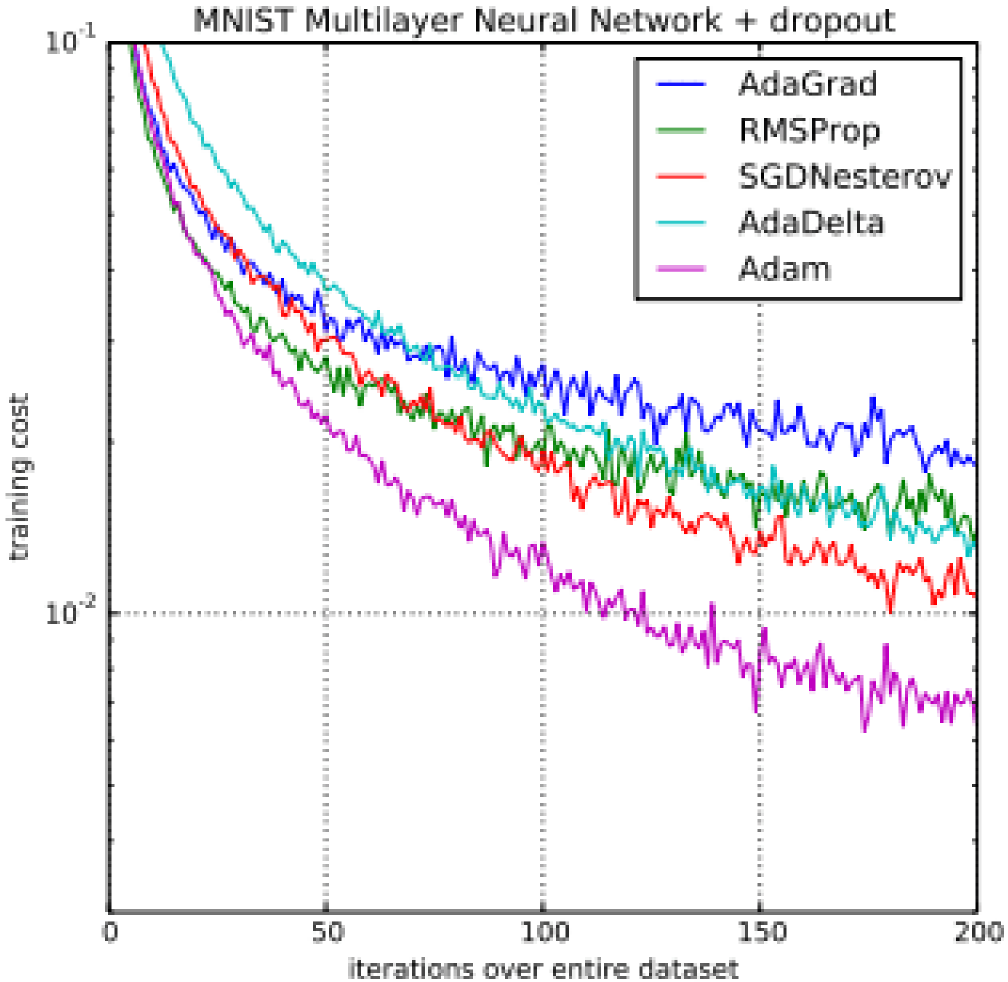
Bias Correction

$$w_{t+1} = w_t - \widehat{m_t}\left(\frac{\alpha}{\sqrt{\widehat{v_t}+\varepsilon}}\right)$$

Parameters Used :

1. **ϵ =** a small +ve constant to avoid 'division by 0' error when ($v_t \to 0$). (10^{-8})
2. **β_1 & β_2 =** decay rates of average of gradients in the above two methods. ($\beta_1 = 0.9$ & $\beta_2 = 0.999$)
3. **α –** Step size parameter / learning rate (0.001)

Adam


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
10^{-1}
10^{-2}
0
50
100
150
200
iterations over entire dataset

PyTorch for Adam

```
#model, optimizer and loss
model = SimpleMultiClassificationNet().to(device)
stateDict=model.state_dict()
print(stateDict)
print(model)
summary(model,(10,4))

lr = 0.1
#optimizer = optim.SGD(model.parameters(), lr=lr)
optimizer = optim.Adam(model.parameters(), lr=lr,betas=(0.9,0.999),eps=1e-08)
loss_fn = nn.CrossEntropyLoss()
```

Summary

- New Concepts
  - Metrics (CM, TPR, FPR, PR, ROC)
  - What is Multiclass Classification
  - Softmax Activation
  - Cross-Entropy Loss
  - Advanced Optimizers
    - SGD with Momentum
    - RMSProp
    - Adam

Graded Home Task 3

- Create a multiclass 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
- Do training with different optimizers (SGD+ADAM) and show their comparison (Loss vs Iteration)
- Compute Confusion matrix for all 10 classes
- Check in your code and results