Deep Learning with Feedforward Neural Network

Yunfan Li PhD

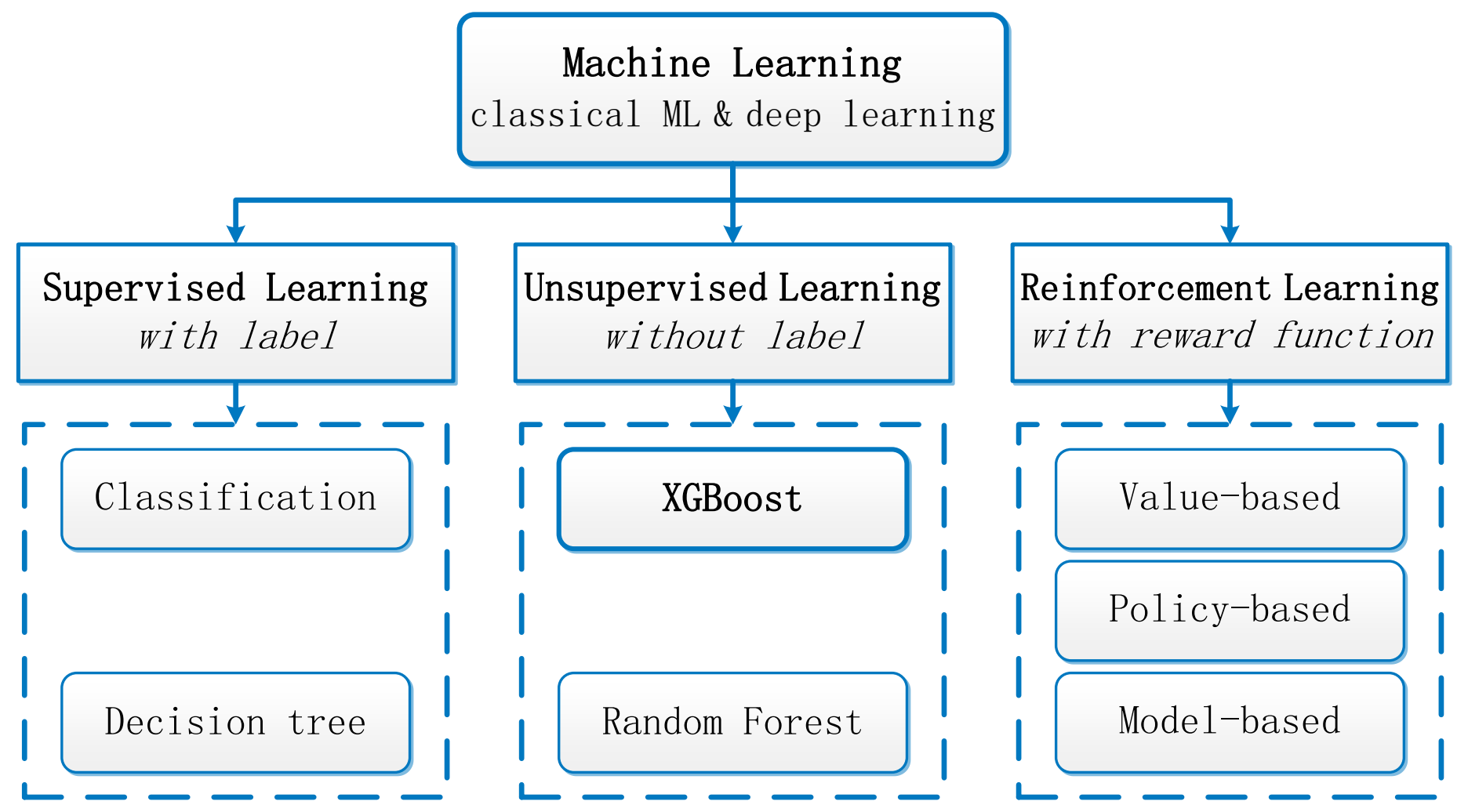
Tsinghua University, University of California at Riverside

About This Poster

Deep learning is a subset of machine learning that uses artificial neural network (ANN). Feedforward neural network (FNN) is one type of ANN. This poster is about FNN-based deep learning model (FNN model).

Background and mathematical formulation of FNN are presented first. Essential mathematics and python implementation of FNN model training are presented next. At the end, more activation functions, a component in model design are shown; different loss function for binary, multiclass classification and regression are discussed.

Background



Machine Learning
classical ML & deep learning

Supervised Learning
with label

Unsupervised Learning
without label

Reinforcement Learning
with reward function

Classification

Decision tree

XGBoost

Random Forest

Value-based

Policy-based

Model-based

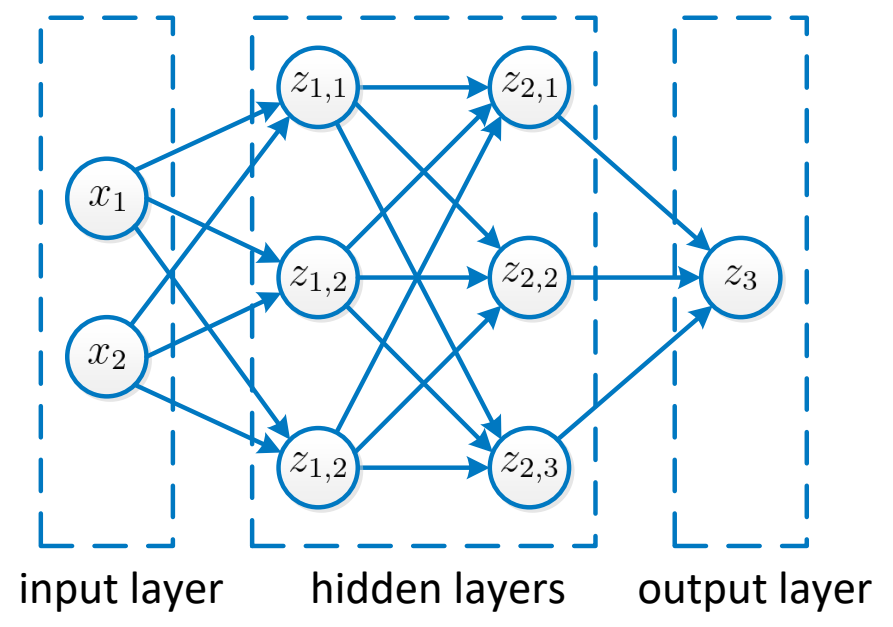
- Machine learning includes supervised, unsupervised, reinforcement.
- Based whether ANN is used to transform features, split machine learning into classical machine learning (no ANN) and deep learning (with ANN).

Mathematical Formulation of FNN

Artifical Neural Network

- ANNs are computation models that transform input via hidden layers to output.
- For a layer with activation function σ , its input is a linear combination of its previous layer with weight W and bias $\boldsymbol{\beta}$, its output is

$$\boldsymbol{z}_k = \sigma_k(W_{k-1}\boldsymbol{z}_{k-1}+\boldsymbol{\beta}_{k-1})$$



Feedforward Neural Network

- A FNN is an ANN in which the node connections does not form a cycle.

FNN-Based Deep Learning Model

Model Design

- FNN architecture: number and size of hidden layers, unknown parameter
$$\boldsymbol{\theta}=[W_0,\boldsymbol{\beta}_0,...,W_{K-1},\boldsymbol{\beta}_{K-1}]$$
- Activation functions $[\sigma_1,...,\sigma_K]$.

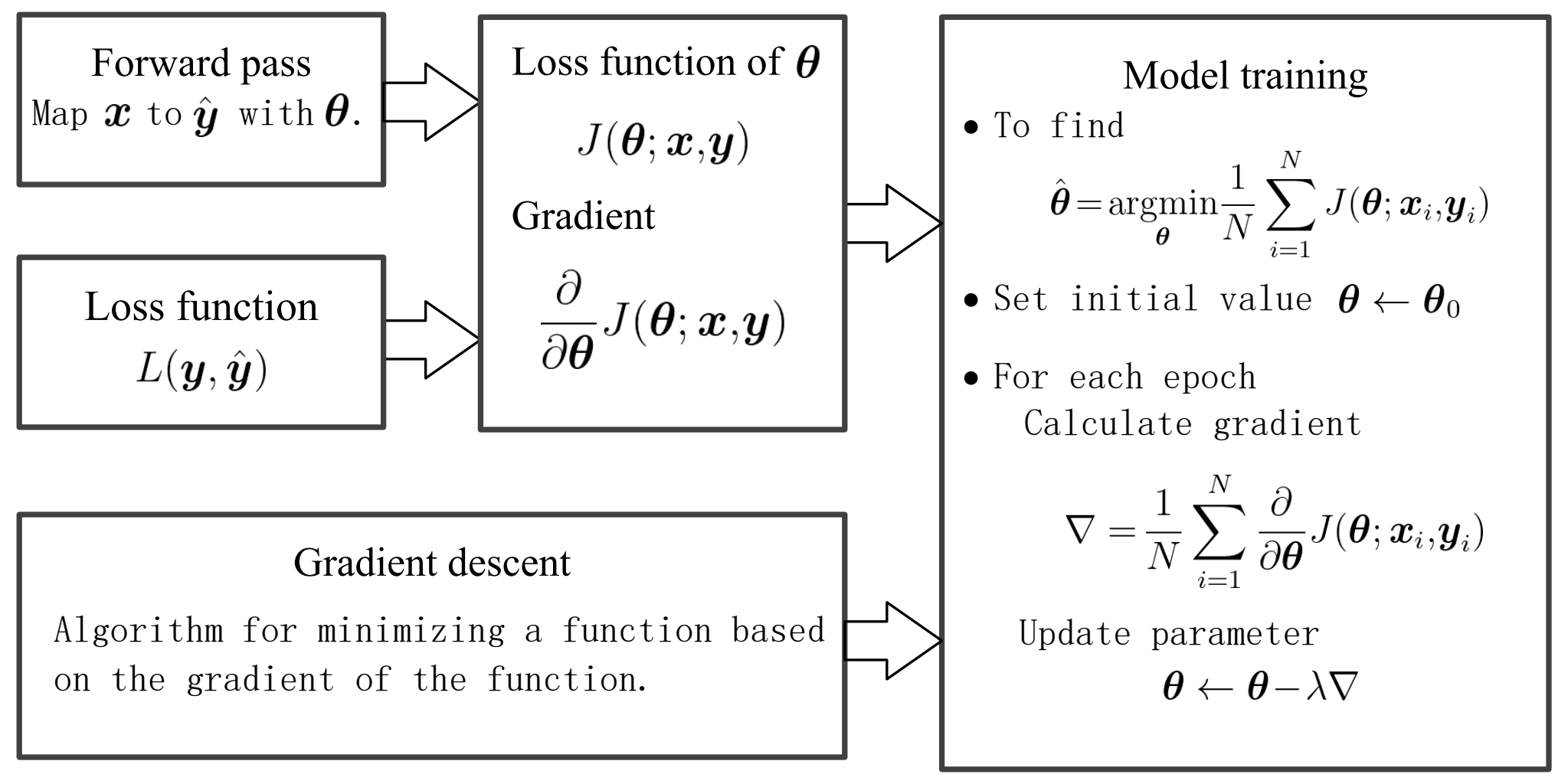
Model output, also known as forward pass $\hat{\boldsymbol{y}}=\boldsymbol{z}_K\overset{\triangle}{=}\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})$.

Loss function

- Measures the distance between label and model output $L(\boldsymbol{y},\hat{\boldsymbol{y}})$.
- Loss as a function of parameter $J(\boldsymbol{\theta};\boldsymbol{x},\boldsymbol{y})=L[\boldsymbol{y},\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})]$.
- Estimate parameter based on N records

$$\hat{\boldsymbol{\theta}}=\underset{\boldsymbol{\theta}}{\text{argmin}}\frac{1}{N}\sum_{i=1}^{N}J(\boldsymbol{\theta};\boldsymbol{x}_i,\boldsymbol{y}_i)$$

Model Training Building Blocks



Forward pass
Map $\boldsymbol{x}$ to $\hat{\boldsymbol{y}}$ with $\boldsymbol{\theta}$.

Loss function
$L(\boldsymbol{y},\hat{\boldsymbol{y}})$

Loss function of $\boldsymbol{\theta}$
$J(\boldsymbol{\theta};\boldsymbol{x},\boldsymbol{y})$
Gradient
$\frac{\partial}{\partial\boldsymbol{\theta}}J(\boldsymbol{\theta};\boldsymbol{x},\boldsymbol{y})$

Gradient descent
Algorithm for minimizing a function based on the gradient of the function.

Model training
- To find
$$\hat{\boldsymbol{\theta}}=\underset{\boldsymbol{\theta}}{\text{argmin}}\frac{1}{N}\sum_{i=1}^{N}J(\boldsymbol{\theta};\boldsymbol{x}_i,\boldsymbol{y}_i)$$
- Set initial value $\boldsymbol{\theta}\leftarrow\boldsymbol{\theta}_0$
- For each epoch
Calculate gradient
$$\nabla=\frac{1}{N}\sum_{i=1}^{N}\frac{\partial}{\partial\boldsymbol{\theta}}J(\boldsymbol{\theta};\boldsymbol{x}_i,\boldsymbol{y}_i)$$
Update parameter
$$\boldsymbol{\theta}\leftarrow\boldsymbol{\theta}-\lambda\nabla$$

Model Training Mathematics and Algorithm

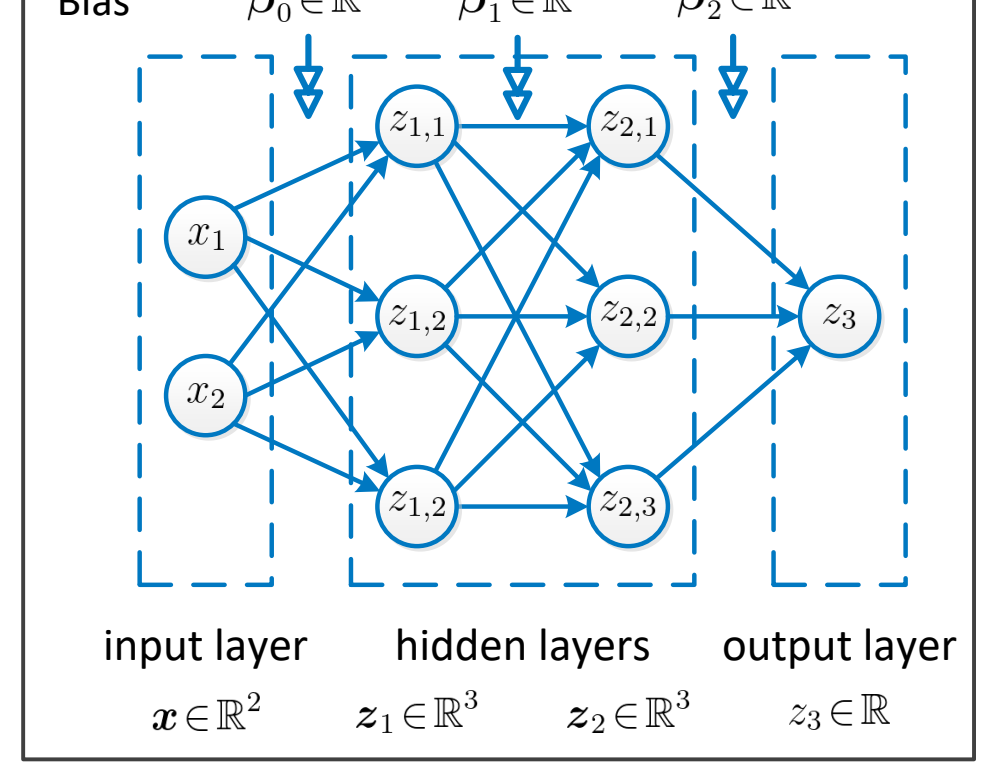
Model Design

Neural Network Architecture

Network parameters $\boldsymbol{\theta}=[W_0,W_1,W_2,\boldsymbol{\beta}_0,\boldsymbol{\beta}_1,\beta_2]$

Weights $W_0\in\mathbb{R}^{3\times2}$ $W_1\in\mathbb{R}^{3\times3}$ $W_2\in\mathbb{R}^{1\times3}$

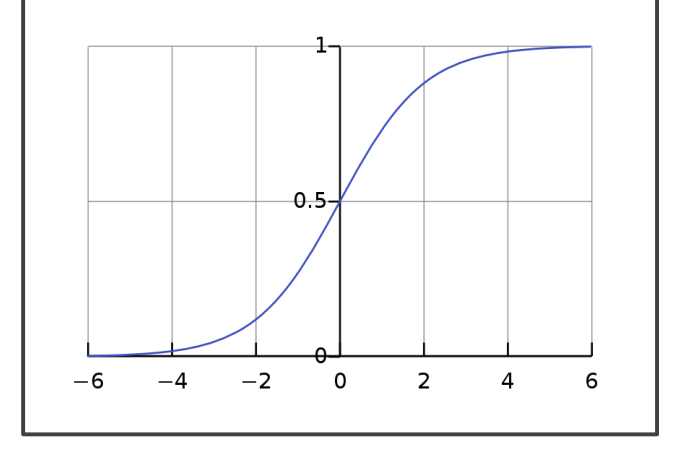
Bias $\boldsymbol{\beta}_0\in\mathbb{R}^3$ $\boldsymbol{\beta}_1\in\mathbb{R}^3$ $\boldsymbol{\beta}_2\in\mathbb{R}$



input layer $\boldsymbol{x}\in\mathbb{R}^2$ hidden layers $\boldsymbol{z}_1\in\mathbb{R}^3$ $\boldsymbol{z}_2\in\mathbb{R}^3$ output layer $z_3\in\mathbb{R}$

Activation Function

$$\sigma(x)=\frac{\exp(x)}{1+\exp(x)} \quad ①$$



Loss Function

Binary cross-entropy
$$H(y,p)=-y\ln p-(1-y)\ln(1-p)$$

Loss function $J(\boldsymbol{\theta};\boldsymbol{x},y)=H[y,z_3(\boldsymbol{x};\boldsymbol{\theta})]$ ②

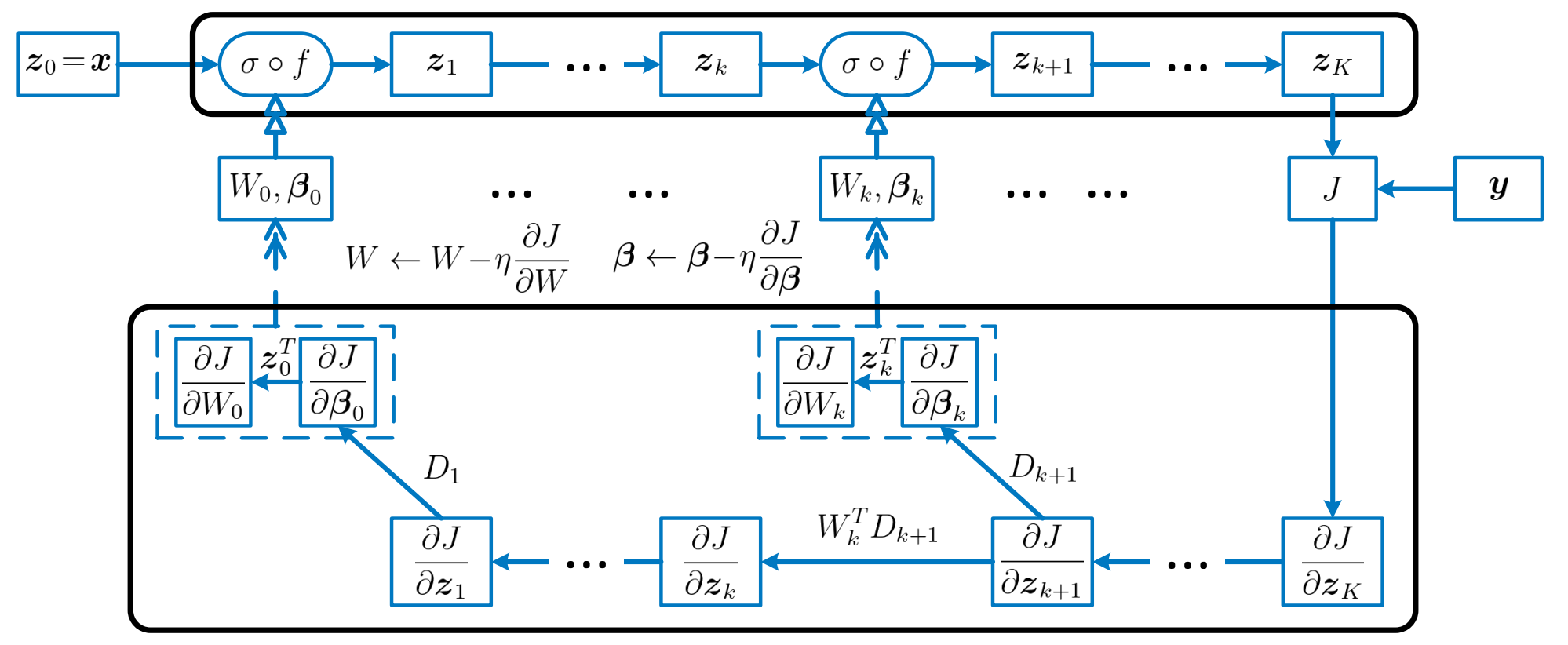
Estimate parameters $\hat{\boldsymbol{\theta}}=\underset{\boldsymbol{\theta}}{\text{argmin}}\sum_i J(\boldsymbol{\theta};\boldsymbol{x}_i,y_i)$

Forward Pass

$$\boldsymbol{z}_1=\sigma(W_0\boldsymbol{x}+\boldsymbol{\beta}_0)=\boldsymbol{z}_1(\boldsymbol{x};\boldsymbol{\theta})$$
$$\boldsymbol{z}_2=\sigma(W_1\boldsymbol{z}_1+\boldsymbol{\beta}_1)=\boldsymbol{z}_2(\boldsymbol{x};\boldsymbol{\theta})$$
$$z_3=\sigma(W_2\boldsymbol{z}_2+\beta_2)=z_3(\boldsymbol{x};\boldsymbol{\theta})$$

Backpropagation Based Calculation of Loss Function Gradient

For generic feedforward neural networks


$\boldsymbol{z}_0=\boldsymbol{x}$
$\sigma\circ f$
$\boldsymbol{z}_1$
$\boldsymbol{z}_k$
$\sigma\circ f$
$\boldsymbol{z}_{k+1}$
$\boldsymbol{z}_K$
$W_0,\boldsymbol{\beta}_0$
$W_k,\boldsymbol{\beta}_k$
J
$\boldsymbol{y}$
$W\leftarrow W-\eta\frac{\partial J}{\partial W}$
$\boldsymbol{\beta}\leftarrow\boldsymbol{\beta}-\eta\frac{\partial J}{\partial\boldsymbol{\beta}}$
$\frac{\partial J}{\partial W_0}$
$\boldsymbol{z}_0^T$
$\frac{\partial J}{\partial\boldsymbol{\beta}_0}$
$\frac{\partial J}{\partial W_k}$
$\boldsymbol{z}_k^T$
$\frac{\partial J}{\partial\boldsymbol{\beta}_k}$
D_1
D_{k+1}
$W_k^TD_{k+1}$
$\frac{\partial J}{\partial\boldsymbol{z}_1}$
$\frac{\partial J}{\partial\boldsymbol{z}_k}$
$\frac{\partial J}{\partial\boldsymbol{z}_{k+1}}$
$\frac{\partial J}{\partial\boldsymbol{z}_K}$

$$f(\boldsymbol{z};W,\boldsymbol{\beta})=W\boldsymbol{z}+\boldsymbol{\beta}$$
$$\boldsymbol{h}_{k+1}\overset{\triangle}{=}f(z_k;W_k,\boldsymbol{\beta}_k)$$
$$D_k\overset{\triangle}{=}\text{diag}[\sigma'(\boldsymbol{h}_k)]$$

$$\boldsymbol{h}_{k+1}\overset{\triangle}{=}W_k\boldsymbol{z}_k+\boldsymbol{\beta}_k$$
$$\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}=\frac{\partial\boldsymbol{z}_{k+1}^T}{\partial\boldsymbol{h}_{k+1}}\frac{\partial J}{\partial\boldsymbol{z}_{k+1}}=D_{k+1}\frac{\partial J}{\partial\boldsymbol{z}_{k+1}}$$
$$\frac{\partial J}{\partial\boldsymbol{z}_k}=\frac{\partial\boldsymbol{h}_{k+1}^T}{\partial\boldsymbol{z}_k}\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}=W_k^T\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}$$
$$\frac{\partial J}{\partial\boldsymbol{\beta}_k}=\frac{\partial\boldsymbol{h}_{k+1}^T}{\partial\boldsymbol{\beta}_k}\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}=\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}$$

$$\frac{\partial J}{\partial W_k}=\frac{\partial\boldsymbol{h}_{k+1}^T}{\partial W_k}\left(\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}\otimes I\right)$$
$$\frac{\partial(AX)}{\partial X^T}=I_m\otimes A$$
$$=(I\otimes\boldsymbol{z}_k^T)\left(\frac{\partial J}{\partial\boldsymbol{h}_{k+1}^T}\otimes I\right)$$
$$=\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}\otimes\boldsymbol{z}_k^T=\frac{\partial J}{\partial\boldsymbol{h}_{k+1}}\boldsymbol{z}_k^T$$
$$(A\otimes B)(C\otimes D)=(AC\otimes BD)$$

Loss Function Gradients

① $D_k=\text{diag}(\boldsymbol{z}_k)[I-\text{diag}(\boldsymbol{z}_k)]$

$$\frac{\partial J}{\partial z_3}\overset{②}{=}\frac{z_3-y}{z_3(1-z_3)}\overset{\triangle}{=}\frac{g}{z_3(1-z_3)}$$
$$\frac{\partial J}{\partial\boldsymbol{z}_2}=W_2^TD_3\frac{\partial J}{\partial z_3}=W_2^Tg$$
$$\frac{\partial J}{\partial\boldsymbol{z}_1}=W_1^TD_2\frac{\partial J}{\partial\boldsymbol{z}_2}$$
$$\cdots=W_1^TD_2W_2^Tg$$

$$\frac{\partial J}{\partial\boldsymbol{\beta}_2}=D_3\frac{\partial J}{\partial z_3}=g \qquad \frac{\partial J}{\partial W_2}=\frac{\partial J}{\partial\boldsymbol{\beta}_2}\boldsymbol{z}_2^T=g\boldsymbol{z}_2^T$$
$$\frac{\partial J}{\partial\boldsymbol{\beta}_1}=D_2\frac{\partial J}{\partial\boldsymbol{z}_2}=D_2W_2^Tg \qquad \frac{\partial J}{\partial W_1}=\frac{\partial J}{\partial\boldsymbol{\beta}_1}\boldsymbol{z}_1^T=D_2W_2^Tg\boldsymbol{z}_1^T$$
$$\frac{\partial J}{\partial\boldsymbol{\beta}_0}=D_1\frac{\partial J}{\partial\boldsymbol{z}_1}=D_1W_1^TD_2W_2^Tg \qquad \frac{\partial J}{\partial W_0}=\frac{\partial J}{\partial\boldsymbol{\beta}_0}\boldsymbol{z}_0^T=D_1W_1^TD_2W_2^Tg\boldsymbol{z}_0^T$$

Gradient Descent Algorithm

For a contour $\mathcal{C}=\{\boldsymbol{\theta}:f(\boldsymbol{\theta})=c\}$ of $f(\boldsymbol{\theta})$, $df=\nabla f\cdot d\boldsymbol{\theta}=0$.

For $\boldsymbol{\theta}_0\in\mathcal{C}$, the gradient $\nabla f(\boldsymbol{\theta}_0)$ is orthogonal to the tangent plane
$$\nabla f(\boldsymbol{\theta}_0)\cdot(\boldsymbol{\theta}-\boldsymbol{\theta}_0)=0$$
$f(\boldsymbol{\theta})$ decreases fastest if $\boldsymbol{\theta}$ goes from $\boldsymbol{\theta}_0$ along $-\nabla f(\boldsymbol{\theta}_0)$.

```
Input : learning rate $\eta$, loss function $J$, $N_{\text{epochs}}$
return : argmin $J(\boldsymbol{\theta})$
    $\boldsymbol{\theta}\leftarrow\boldsymbol{\theta}_{\text{initial}}$
    for $i:=1,\ldots,N_{\text{epochs}}$
        $\boldsymbol{\theta}\leftarrow\boldsymbol{\theta}-\eta\frac{\partial}{\partial\boldsymbol{\theta}}J(\boldsymbol{\theta})$
    return $\boldsymbol{\theta}$
```

Model Training Algorithm

Variations of Gradient Descent

Stochastic Gradient Descent In each training epoch, update model with each record.

Mini-Batch Gradient Descent In each training epoch, split the whole training data into multiple mini-batches, update model with one mini-batch at a time.

```
Initialize parameters                 $\boldsymbol{\theta}\leftarrow[W_0,W_1,W_2,\boldsymbol{\beta}_0,\boldsymbol{\beta}_1,\boldsymbol{\beta}_2]_{\text{init}}$
Multiple training epochs              for epoch $:=1,\ldots,N_{\text{epochs}}$
For all minibatches                       for $\{X,\boldsymbol{y}\}:=\{X,\boldsymbol{y}\}_1,\ldots,\{X,\boldsymbol{y}\}_m$
                                              $W_0,W_1,W_2,\boldsymbol{\beta}_0,\boldsymbol{\beta}_1,\boldsymbol{\beta}_2\leftarrow\boldsymbol{\theta}$
                                              $W_0^*,W_1^*,W_2^*\leftarrow W_0,W_1,W_2$
For all records in a minibatch                for $\boldsymbol{x},y$ in $\{X,\boldsymbol{y}\}$
                                                  $\boldsymbol{z}_1,\boldsymbol{z}_2,z_3\leftarrow\boldsymbol{z}_1(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{z}_2(\boldsymbol{x}_i;\boldsymbol{\theta}),z_3(\boldsymbol{x}_i;\boldsymbol{\theta})$
                                                  $\boldsymbol{g}\leftarrow z_3-y$
Backpropagation                                   for $j:=2,1,0$
                                                      $W_j,\boldsymbol{\beta}_j\leftarrow W_j-\frac{\eta}{N}\boldsymbol{g}\boldsymbol{z}_j^T,\boldsymbol{\beta}_j-\frac{\eta}{N}\boldsymbol{g}$
                                                      $\boldsymbol{g}\leftarrow D(\boldsymbol{z}_j)W_j^{*T}\boldsymbol{g}$      $D(\boldsymbol{z})=\text{diag}[\boldsymbol{z}\odot(\boldsymbol{1}-\boldsymbol{z})]$
Update parameters after                   $\boldsymbol{\theta}\leftarrow[W_0,W_1,W_2,\boldsymbol{\beta}_0,\boldsymbol{\beta}_1,\boldsymbol{\beta}_2]$
training with a minibatch
                                      return $\boldsymbol{\theta}$
```

Model Training in Python

```
#mini-batch of data
batch_size =50
n_batch = int(np.ceil(data.shape[0]/batch_size))
#hyperparameters
learning_rate,epochs = 0.1,400
```

Model Training with PyTorch

```
from torch import nn
class Net(nn.Module):
   def __init__(self,input_shape,n1=3,n2=3):
      super(Net,self).__init__()
      self.fc1 = nn.Linear(input_shape,n1)
      self.fc2 = nn.Linear(n1,n2)
      self.fc3 = nn.Linear(n2,1)
   def forward(self,x):
      x = torch.sigmoid(self.fc1(x))
      x = torch.sigmoid(self.fc1(x))
      x = torch.sigmoid(self.fc1(x))
      return x
```

```
from torch.utils.data import Dataset,
DataLoader
class dataset(Dataset):
   def __init__(self,x,y):
      self.x = torch.tensor(x,dtype=torch.float32)
      self.y = torch.tensor(y,dtype=torch.float32)
      self.length = self.x.shape[0]
   def __getitem__(self,idx):
      return self.x[idx],self.y[idx]
   def __len__(self):
      return self.length
```

```
x,y = data[:,1:2],data[:,0]
trainset = dataset(x,y)
trainloader=DataLoader(trainset,
                       batch_size=batch_size)
```

```
model = Net(input_shape=2)
```

```
optimizer = torch.optim.SDG(
        model.parameters(),lr=learning_rate)
loss_fn = nn.BCELoss()
for i in range (epochs):
   for j,(x,y) in enumerate(trainloader):
      output = model(x)
      loss = loss_fn(output,y.reshape(-1,1))
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()
```

Model Training with Numpy

```
class Model_NP:
   def __init__(self, model_torch):
      self.params = []
      params = [param.data.numpy() for
               param in model_torch.parameters()]
      for i in range(0,len(params_np),2):
         self.params.self.append({'w':params[i],
                                  'b':params[i+1]})

   def forward(self,x):
      self.z_lists = []
      for z in x:
         z_list = [z]
         for param in self.params:
            w,b = param.values()
            z = expit(w@z + b)
            z_list.append(z)
         self.z_lists.append(z_list)
```

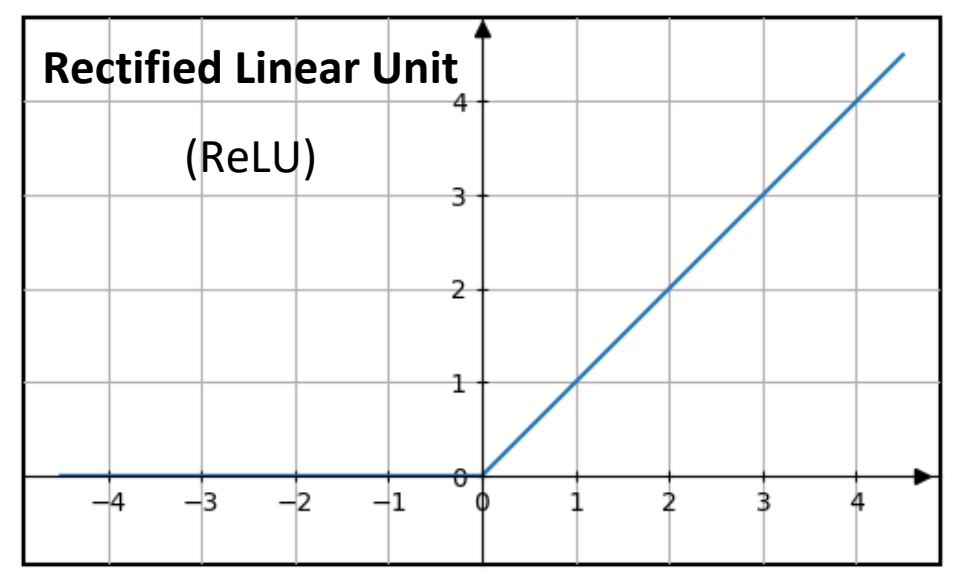
```
model_np = Model_NP(model)
```

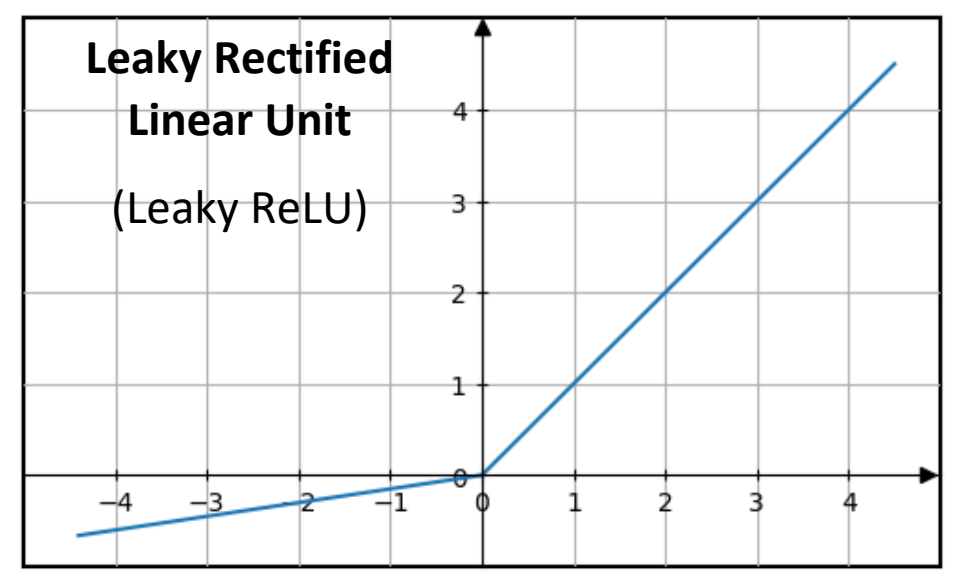
```
for epoch in range(epochs):
   for i in range(n_batch):
      batch = data[i*batch_size:(i+1)*batch_size]
      x,y = batch[:,1:],batch[:,0]
      model_np.forward(x)
      grads = SDG(model_np).backward(y)
      for param,g in zip(model_np.params,grads):
         param['w'] -= learning_rate * g['w']
         param['b'] -= learning_rate * g['b']
```

```
class SDG:
   def __init__(self,model_np):
      self.model = model_np

   @staticmethod
   def _gradients(y,z_list,w_list):
      g = z_list[-1].reshape(-1,1) - y
      grads = []
      for z,w in reversed(list(zip(z_list[:-1],w_list))):
         grads.append({'w':g@z.reshape(-1,1),'b':g})
         g = np.diag(z*(1-z))@w.T@g
      return grads

   @staticmethod
   def avg_grad(grads):
      N = len(grads)
      w = sum([g['w'] for g in grads])/N
      b = ([g['b'] for g in grads])/N
      return {'w':w,'b':b[:,0]}

   def backward(self,y_vec):
      params = self.model.parameter
      w_list = [param['w'] for param in params]
      z_lists = self.model.z_lists
      grads_all = []
      for z_list,y in zip(z_lists,y_vec):
         grads = self._gradients(y,z_list,w_list)
         grads_all.append(grads)
      grads_zip = [list(x) for x in zip(*grads_all)]
      grads_avg=[self.avg_grad(g) for g in grads_zip]
      return reversed(grads_avg)
```

More About Activation Functions

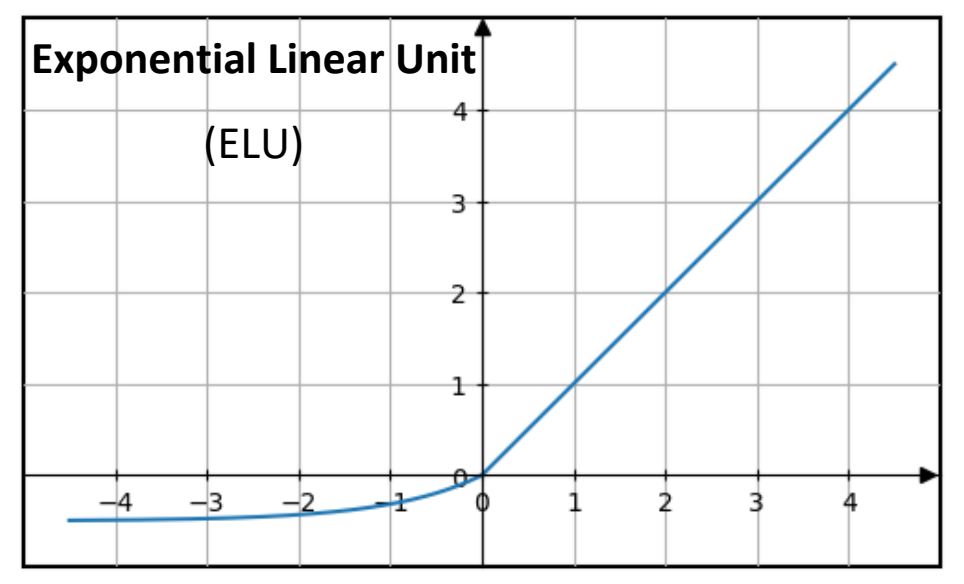
- Other frequently used activation functions

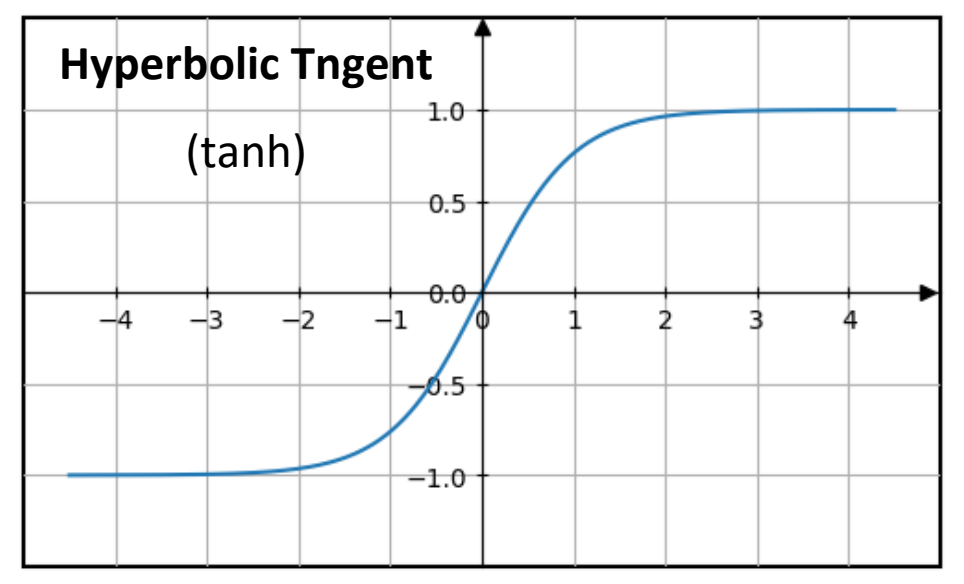




$$f(x)=\max(x)$$

$$f(x;\alpha)=\begin{cases}x, & x\ge0\\ \alpha x, & x<0\end{cases}$$





$$f(x;\alpha)=\begin{cases}x, & x\ge0\\ \alpha(e^x-1), & x<0\end{cases}$$

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

- Different activation functions can be applied to different layers.

More About Loss Function

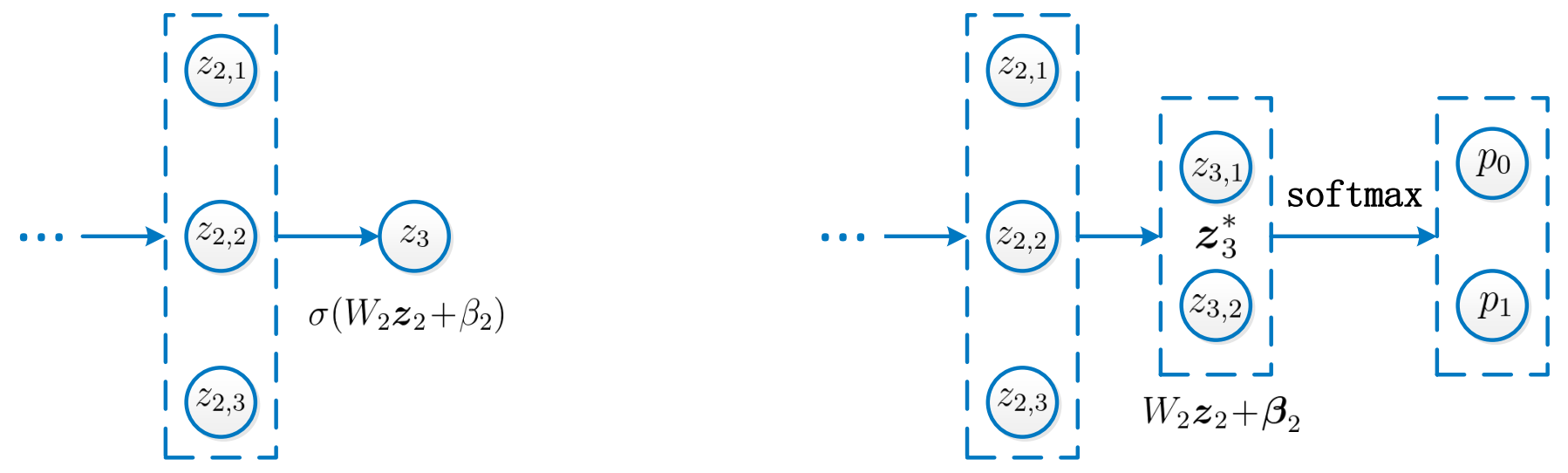
Binary classification

Label $y=0,1$, model output $p\in(0,1)$. Alternatively

- One-hot encode label $y\to Y=[1-y,y]$, change output dimension to 2.
- Remove the output layer activation, apply softmax to raw score $\boldsymbol{z}_3^*$ to get probability $\boldsymbol{p}$.
- Cross-Entropy Loss Function

$$H(\boldsymbol{Y},\boldsymbol{p})=-\boldsymbol{Y}^T\ln\boldsymbol{p}=-Y_0\ln p_0-Y_1\ln p_1$$

Softmax
$$\boldsymbol{z}\in\mathbb{R}^k, s(\boldsymbol{z})=\frac{\exp(\boldsymbol{z})}{\boldsymbol{1}\cdot\exp(\boldsymbol{z})}.$$



Note The probability $[p_0,p_1]$ is the same as $[1-z_3,z_3]=s([0,\sigma^{-1}(z_3)])$.

Multiclass classification (*k*-class)

- Same as binary classification, one-hot encode label to a length-k vector.
- Set the output dimension to k.
- Use cross-entropy as loss function.

Note torch.nn.CrossEntropyLoss includes one-hot encoding and softmax.

Numpy

```
def softmax(x):
   return np.exp(x)/np.sum(np.exp(x))

Y_pred = [2,1,0.1]
p = softmax(Y_pred)
```

```
array([0.659 , 0.2424, 0.0986])
```

```
def cross_entropy(actual,Y_pred):
    return -np.sum(actual*np.log(Y_pred))
Y = [1,0,0]
L = cross_entropy(y,p)
```

```
0.4170
```

PyTorch

```
loss = torch.nn.CrossEntropyLoss()
y = torch.tensor([0])
Y_pred = torch.tensor([[2,1,0.1]])
l = loss(Y_pred,y)
```

```
0.4170
```

Regression

- The output layer with dimension 1, without activation.
- Mean Squared Error and Mean Absolute Error loss are frequently used.

$$\text{MSE}=\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$$

```
loss = nn.MSELoss()
l = loss(Y_pred,y)
```

$$\text{MAE}=\frac{1}{n}\sum_{i=1}^{n}|y_i-\hat{y}_i|$$

```
loss = nn.L1Loss()
l = loss(Y_pred,y)
```