Metric

Classi prob pred : Log Loss

Regres : RMSE : $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}$ square to ignore pos, neg term

Bias : inability of model to learn from data.

Variance: the diff in fits between datasets

Low bias : learning well with dataset

High var : high diff of fitting between datasets

Gradient

$w_1 = w_0 - \alpha \nabla \frac{1}{2}\sum_{i=1}^{m}((\beta_0+\beta_1 x^{(i)}_{obs}) - y^{(i)}_{obs})^2$ (α: learning rate)

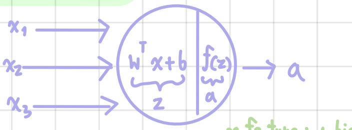
Stochastic

$w_1 = w_0 - \alpha \nabla \frac{1}{2}((\beta_0+\beta_1 x^{(i)}_{obs}) - y^{(i)}_{obs})^2$ (α: learning rate)

Mini batch

$w_1 = w_0 - \alpha \nabla \frac{1}{2}\sum_{i=1}^{n}((\beta_0+\beta_1 x^{(i)}_{obs}) - y^{(i)}_{obs})^2$ (α: learning rate; n: 100 from 1k data)

neural net



m feature + 1 bias

$z = b + x_1 w_1 + x_2 w_2 + \dots + x_m w_m = b + x^T w$

Sigmoid : $f(z) = \frac{1}{1+e^{-z}}$ (prob 0-1)(bin class)

Relu : $Relu(z) = \begin{cases} 0, & z<0 \\ z, & z\geq 0 \end{cases} = max(0,z)$

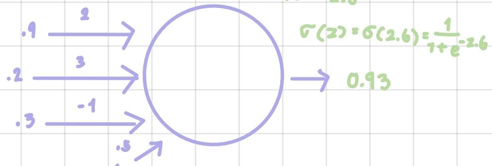
Tanh : $tanh(0)=0$, $tanh(\infty)=1$, $tanh(-\infty)=-1$

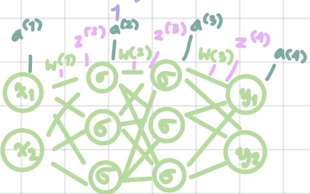
LReLU (Leaky) : $LReLU(z) = \begin{cases} az, & z<0 \\ z, & z\geq 0 \end{cases} = max(az, z)$ [a<1]

$LReLU(0) = 0$, $LReLU(\infty) = z$, $-(-z) = -az$

Softmax : multiclass (prob 0-1)

$z = .18 + .6 - .3 + .5 = 2.6$





$w^{(1)}$: 2x3 matrix

$z^{(2)}$: 3 vector

$a^{(2)}$: 3 vector

step

Input : vector x, $z^{(2)} = xW^{(1)}$, $a^{(2)} = \sigma(z^{(2)})$

$z^{(3)} = a^{(2)}W^{(2)}$, $a^{(3)} = \sigma(z^{(3)})$, $z^{(4)} = a^{(3)}W^{(3)}$

$\hat{y} = softmax(z^{(4)})$

Loss func : 1 iter, Loss cost : whole iter

Hyper para : upper layer para, tunable

Loss func, Learning rate, epoch, batch_size, optimizer

Forward propogate

m: output node (1)

n: input node (2)

W size: m·n, x = n·1, B = m·1

$z = W \cdot x + B$, $z = [w_1, w_2] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [b] = [(w_1x_1 + w_2x_2) + b]$

Backpropogation

Error at $w_1 = \frac{\partial L}{\partial w_1}$ ↑ in w_1, ↑ in L

$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1}$ Cal each term

CNN Steps

1) init : weight, bias randomly
2) forward : • pass input through network layer by layer • compute weighted sum of inputs & apply activation func • Continue to final layer & gen pred output
3) Calculate Error : cal error (RMSE), – regression; Cross-entropy loss – classify
4) Back : • Cal gradient of the loss with respect to weights & biases • update in the opposite direction to minimize loss (optimizer algo)
5) Repeat front/back step for epoch, adjust hyperparams

Vanishing Gradient

$\frac{\partial J}{\partial W^{(1)}} = (\hat{y}-y) \cdot W^{(3)} \cdot \sigma'(z^{(3)}) \cdot W^{(2)} \cdot \sigma'(z^{(2)}) \cdot x$

- $\sigma'(z) = \sigma(z)(1-\sigma(z)) \leq .25$
- More layer, smaller gradient on early layers
- Solve w/ ReLU

Keras

- Sequential Model : linear stack of layers, simpler, more convenient if model've this form

Functional API : more detailed, complex allows more complicated architectures

```
from keras.models import Sequential
model = Sequential()
from keras.layers import Dense, Activation
```

first

```
model.add (Dense (units=4, input_shape=
x_train.shape[1:], activation='relu'))
```

subsequent layer

```
Model.add (Dense (units=4, activation='relu'))
Model.add (Dense (units=1, "...        "='sigmoid'))
```

Multiclass

- one hot encoding for categories
- vector's length = categories's num

$\begin{bmatrix}1\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\1\end{bmatrix}$
cat dog toaster

- Final layer's length = pos class's num
- softmax = [0-1] prob
- loss : categorical cross entropy

"Log Loss" $C.E. = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$

$\frac{\partial C.E.}{\partial softmax} \cdot \frac{\partial softmax}{\partial z_i} = \hat{y}_i - y_i$

input scale

Linear Scaling to [0,1]

$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$

" ——— " [-1,1]

$x_i = 2\left(\frac{x_i - x_{min}}{x_{max} - x_{min}}\right) - 1$

Standardization var approx. std. normal

$x_i = \frac{x_i - \bar{x}}{\sigma}$; $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2}$

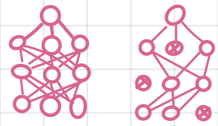
Regularizing

- Penalized cost func

add penalty to loss func for having high weights

$J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\sum_{j=1}^{m} W_j^2$

- Dropout
  - randomly remove subset of neurons at each batch
  - prevent NN from rely too much on individual path way



traing time . present w/prob p

test time : always present

- Early stopping
  - make a rule to be called to stop the traing at some condition.
  - Example: check validation log-loss every 10 epochs, if it is higher than it was last time, stop and use previous model

Optimizer

- Momentum : change direction a little each time, keeps a "running average" of step directions, smoothing out the variation of individual points.

average value over time – $V_t = \beta \cdot V_{t-1} + (1-\beta) \cdot \nabla J$ (β: momen; ∇J: gradient)

$W = W - \alpha \cdot V_t$ (α: learning rate)

$(\beta < 1)$

gradient update'll be bit smoother & more able to travel down shallow directions.

- Nesterov Momentum
  - Control "overshooting" by looking ahead
  - Apply gradient only to "non-momentum" component.

$V_t = \beta \cdot V_{t-1} + \alpha \nabla J(W - \beta \cdot V_{t-1})$ (lookahead)

$W = W - V_t$

Adagrad (Adaptive Grad)

- Scale update for each w separately
- Keep running sum of previous updates
- Devide new update with prev sum

$W = W - \frac{\alpha}{\sqrt{G_t} + \epsilon}\nabla J$ dynamic learning rate $G_t = G_{t-1} + (\nabla J)^2$

- learning.. / sqr of sum of each component separately

CNN

- Edges → Shape → relations betw shapes

kernels (filters)

- grid of weights "overlaid" on image, centered on 1 pix
  traditional image processing techniques
  - Blur · Sharpen · Edge detection · Emboss

3	2	1
1	2	3
1	1	1

-1	0	1
-2	0	2
-1	0	1

	2	

$= (3\cdot -1) + (2\cdot 0) + (1\cdot 1) + (1\cdot -2) + (2\cdot 0) + (3\cdot 2) + (1\cdot -1) + (1\cdot 0) + (1\cdot 1) = 2$

local feature detector

-1	1	-1
-1	1	-1
-1	1	-1

-1	-1	-1
1	1	1
-1	-1	-1

-1	-1	-1
-1	1	1
-1	1	1

- Let NN learn which kernel are most useful
- Same set of kernel across entire img (translation invariance)
- Reduce num of params & variance

Grid size (h & width)

- num of pixels a kernel see at once
- Typically odd num so there's "center" pixel

Padding

valid → no padding
same → same as input

- prevent edge effect on kernel
- add extra pixel around frame (0)
- every pix of ori img can be center pix as the kernel move

0	0	0	0	0	0	0
0	1	2	0	3	1	0
0	·	·	·	·	·	0
0	·	·	·		·	0
0	·	·	·	·	·	0
0	·	·	·	·	·	0
0	0	0	0	0	0	0

-1	1	2
1	1	0
-1	-2	0

-1				

input, filter

no-pad-out size : $n-f+1$
pad-" —— " : $n+2p-f+1$

Stride

- Step size of kernel
- stride > 1 → output dimension scales down

-1	2	0	3	1
1	0	0	2	2
2	1	2	1	1
0	0	1	0	0
1	2	1	1	1

-1	1	2
1	1	0
-1	-2	0

-2	

output size $= \left\lfloor \frac{n-f}{s} + 1 \right\rfloor$ nopad 72

with pad $= \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$

Depth (channels)

- จำนวนขึ้นกับการ design และ behavior ของ data
- RGB : 3 chan, CMYK : 4 chan
- kernel's depth = input's depth
- Computation : 5x5 kernel on RGB img
  : 5x5x3 = 75 weights
  output's depth of convo layer = num of kernel it applies to prev layer
  each kernel is trained diffly
- 10 kernels in layer, layer output depth 10

Pooling

Reduce Img size by mapping a patch of pix to single val

- Shrink img's dimensions

2	1	0	-1
-3	8	2	5
1	-1	3	4
0	1	1	-2

→

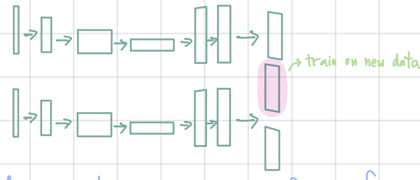
Max pooling / Average Pooling

pooling layer : outsize $= \frac{n - poolsize}{stride} + 1$

Transfer Learning

- Early layer in NN are hard to train due to "vanishing gradient"
- Feature map : out of conv layer represent specific feature in the input image
- later layer are easier to train with more immediate impact on final result
  quicker : back propagating's sigmoid make the earlier weight smaller & smaller
- Famous, competitive model are dif to train from scratch → huge datasets, long num of iteration, very heavy computing machinery, Time experimenting to get hyper-params right

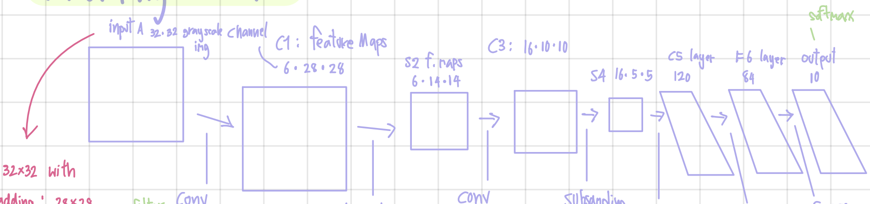
- Idea : keep early layers of pretrained & retrain the later layer for specific UC



Fine tune : train pretrained model for specific use on specific new dataset

Calculating Conv2d

params : weights + biases



Input 32x32 with 2 pixel padding : 28x28

filter Conv
5x5 conv with stride 1 depth 6
output : 6·28·28
total params : ((1·5·5)+1)·6
: 156

Subsampling
2·2 pooling stride 2
output : $\frac{28}{2}\cdot\frac{28}{2}\cdot 6 = 14\cdot 14\cdot 6$
no weight

Conv
5·5 conv stride 1
depth 16
output size :
$\frac{14-5}{1}+1 = 10\cdot 10\cdot 16$
weights : (6·5·5) per kernel
params : (6·5·5)+1 per kernel
total params : ((6·5·5)+1)·16
: 2416

Subsampling
2·2 pooling
output : $\frac{10}{2}\cdot\frac{10}{2}\cdot 16$
5·5·16

Full connection
flatten to length 400 vector

400 { o o o } 120
$w_1x_1 + w_2x_2 + \dots + w_nx_n$
400·120
1 node รับ 400 input
params : (400+1)·120
: 48120

Full
(120+1)·84
: 10164

Gaussian connection
(84+1)·10
: 850