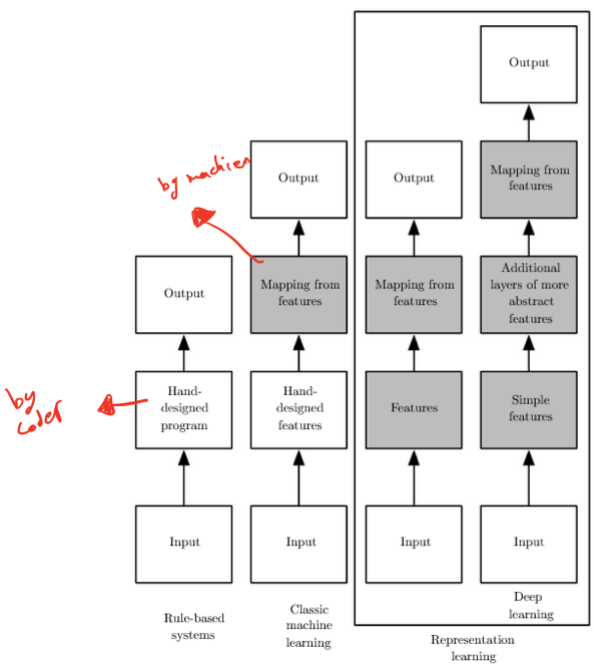
01

- The true challenge proved to be solving tasks that are easy for people to perform but hard for people to describe formally

Application of deep learning :-

1. classification
2. retrieval
3. detection
4. segmentation
5. Image captioning
6. Deep Art: Combining Content and Style from Different Images

- Simple machine learning algorithms depends heavily on the representation of the data they are given



02 - Classification

Images

① binary

② grey

③ color

Problems with Image classification:

① view point

② background clutter

③ illumination

④ occlusion

⑤ deformation

❖ Intraclass Variation (should be low)
between the samples in the same class

* Interclass variation (should be high)
between two classes

Ways to classify Images:

① Knn

- using distance
- no real training
- complexity => train = O(1), test = O(n)
- hyperparameters => K, distance metric (L_1, L_2)
- Set them by: ① train, validation, test
  ② cross-validation

② Linear classification:

- score function: $f(x, W) = Wx + b$

03 - Loss function and Optimization

A **loss function** tells how good our current classifier is

1. SVM loss function
2. Regularization (add L_2 term)
   - to prevent overfitting
3. Softmax classifier

What is the **softmax loss** and the **SVM** loss **if I double the correct class score from 10 -> 20**?

for SVM will be the same

How to reduce the loss?

1. trail and error
2. gradient descent
   - numeric
   - analytic (derivatives)

Multiclass SVM loss function

if the values are differents the loss fun. is good



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	12.9

= max(0, 5.1 - 3.2 + 1)
+max(0, -1.7 - 3.2 + 1)
= max(0, 2.9) + max(0, -3.9)
= 2.9 + 0
= 2.9

= max(0, 1.3 - 4.9 + 1)
+max(0, 2.0 - 4.9 + 1)
= max(0, -2.6) + max(0, -1.9)
= 0 + 0
= 0

= max(0, 2.2 - (-3.1) + 1)
+max(0, 2.5 - (-3.1) + 1)
= max(0, 6.3) + max(0, 6.6)
= 6.3 + 6.6
= 12.9

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Loss over full dataset is average:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

L = (2.9 + 0 + 12.9) / 3

L = **5.27** → we try to minimize it

Softmax Classifier (Multinomial Logistic Regression)

- Want to interpret raw classifier scores as **probabilities**
  - **Softmax**

$$s = f(x_i, W) \qquad P(Y = k|X = x_i) = \frac{e^{s}_{k}}{\sum_j e^{s}_{j}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

cross entropy

$$L_i = -log\ P(Y = k|X = x_i)$$

		exp		normalize	
cat	3.2	$e^{3.2}$	24.5		0.13
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

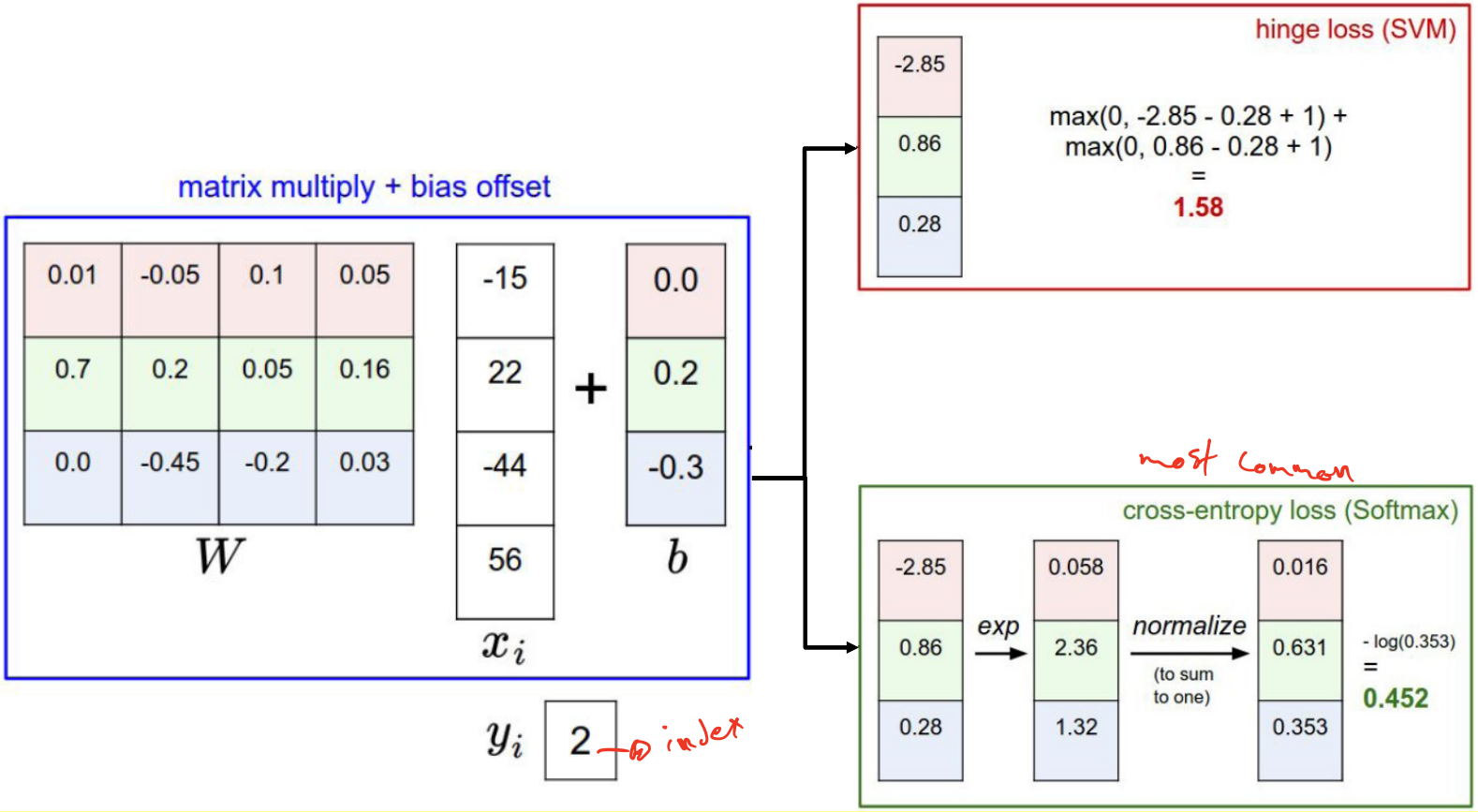
$$L_i = -log\ (0.13) = \mathbf{2.04}$$

unnormalized probabilities

probabilities

Maximum Likelihood Estimation
Choose weights to maximize the likelihood of the observed data

Softmax vs. SVM



Recap

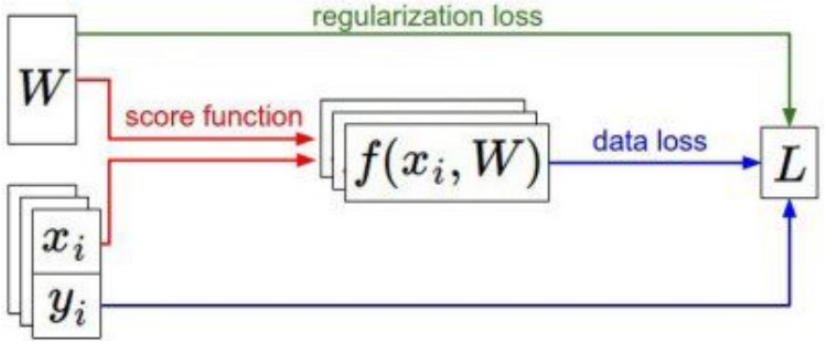
- We have some dataset of (x,y)
- We have a **score function:** $s = f(x_i, W)$
- We have a **loss function**:

Softmax $L_i = -log\ (\frac{e_k^s}{\sum_j e_j^s})$

SVM $L_i = \sum_{j \neq yi} \max(0, s_j - s_{yi} + 1)$

Full loss $L = \frac{1}{N} \sum_{i=1}^{N} Li + R(W)$

How do we find the best W?



04 - NN

* of neurals in the last layer = * of classes

05 - Multilayer Perceptron

Can't handle non-linear inputs

if $y^i(\sum w_j x_j) > 0$

else if $y^i(\sum w_j x_j) \leq 0$

$\vec{w} \leftarrow \vec{w} + y^{(i)} x^{(i)}$

[-1, 1]

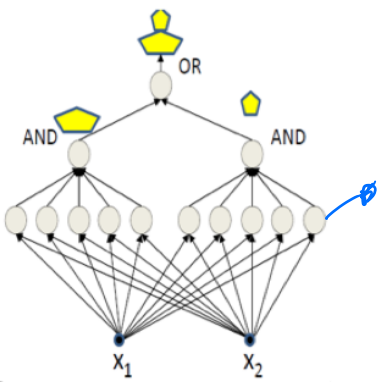
repeat until no classification errors

- The perceptron learning algorithm is guaranteed to converge if classes are linearly separable after $\left(\frac{R}{\gamma}\right)^2$ misclassifications

- Feed-Forward Neural Network
  - Has No loops: Neuron outputs do not feed back to their inputs directly or indirectly

for complex distribution

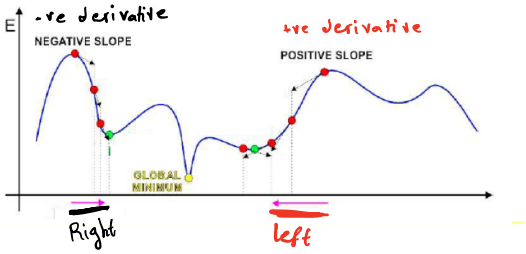
Classification Rule: $Sign(W^T X_i)$



liner classifier

- We don't know the outputs of the individual intermediate neurons in the network for any training input

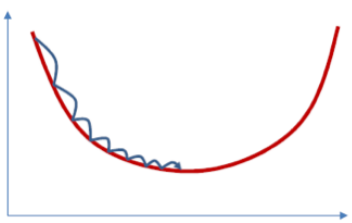
$\frac{df}{d\sigma} = \sigma(s)(1-\sigma(s))$



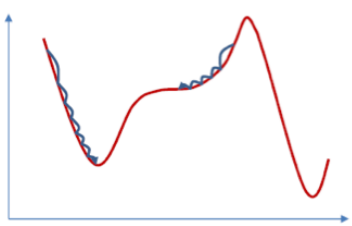
- For multivariate function, to find a minimum, move exactly opposite the direction of the gradient

$w = w - learning\_rate * dw$

- For appropriate step size, for convex (bowlshaped) functions, gradient descent will always find the minimum



- For non-convex functions it will find a local minimum or an inflection point



$\Sigma wx + b$

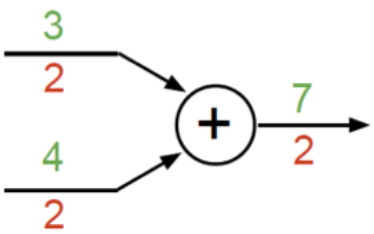
* the input of neurons is => Weighted sum of input + bais

* we can use different activation function in different layers
* we can't use different gradient descent algo. in different layers

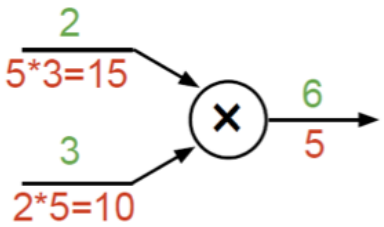
06-1- Backpropagation

Patterns in gradient flow

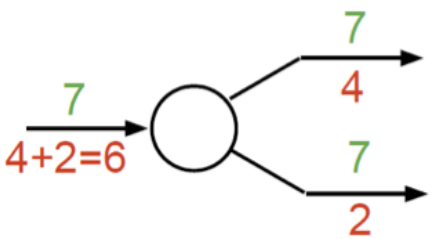
add gate: gradient distributor



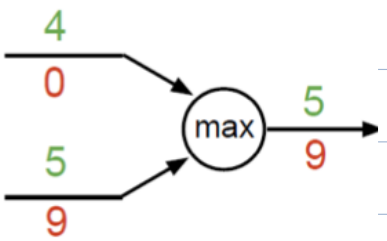
mul gate: "swap multiplier



copy gate: gradient adder



max gate: gradient route



06 - 2 - Backpropagation

Scalar to Scalar ⇒ Regular derivative

Vector to Scalar ⇒ Gradient (n input, 1 output)

$$\frac{\partial f}{\partial x} = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]$$

Vector to Vector ⇒ Jacobian (n input, m output)

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial f_n} \end{bmatrix}$$

Solve Examples on Vector Backpropagation

---

07 - Start with pyTorch

* automatic backpropagation
* Dynamic Computation graphs
* Tensors are single data type

```
# cannot get GPU tensor as
# numpy array directly
```

→ you need to convert it to CPU then to numpy

❖ .backward() doesn't replace, but accumulates gradients → use x.grad.data.zero_()

Losses in PyTorch

- nn.CrossEntropyLoss does both the softmax and the actual cross-entropy.

① forward
② backward
③ optimizing
④ learning (update)

```
x = torch.arange(0.,32)
net = torch.nn.Linear(32,10)
```

input (features)
output (classes)

torch.nn.Linea
to the incomi

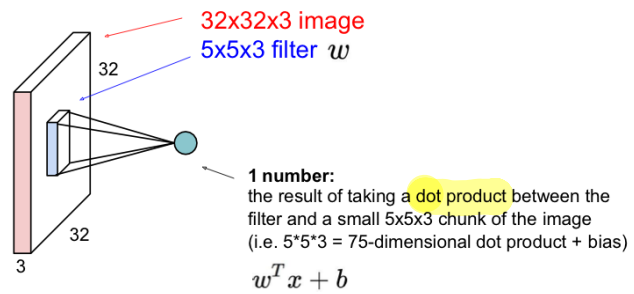
08 - CNN

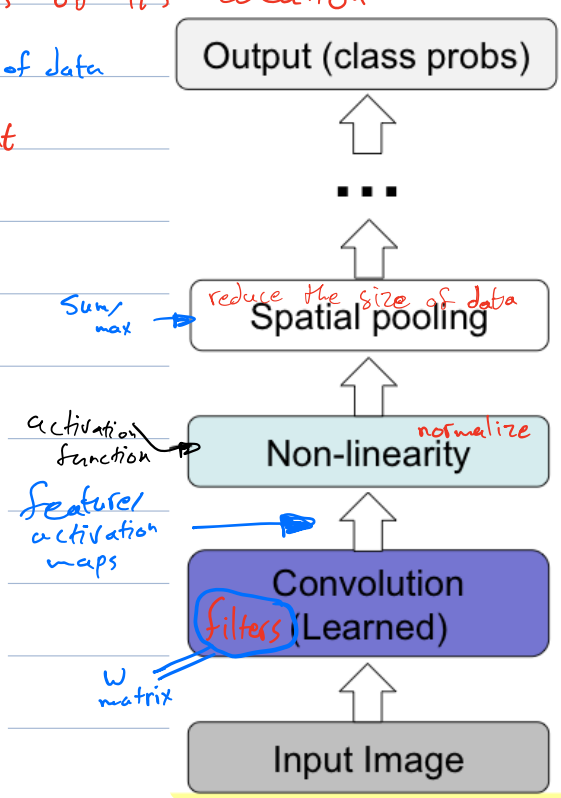
when we need to keep the locations of input

* Goal => detect a pattern regardless of it's location
* normal NN affected by changing the location of data

So we need a new network with shift invariant

filters



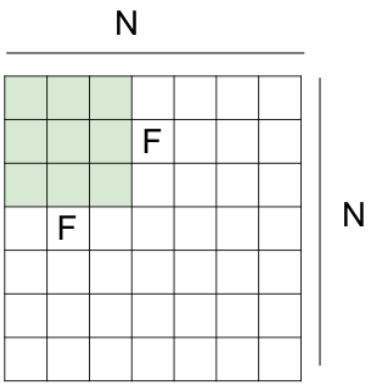


Each filter will look for some feature in the data (learned automatically)

Stride = Shift

for Convolution layers:-

Output size:
(N - F) / stride + 1



to get output = input ⇒ Set padding = (F-1)/2
and Stride = 1

Output size:
(N + 2*padding - F) / stride + 1

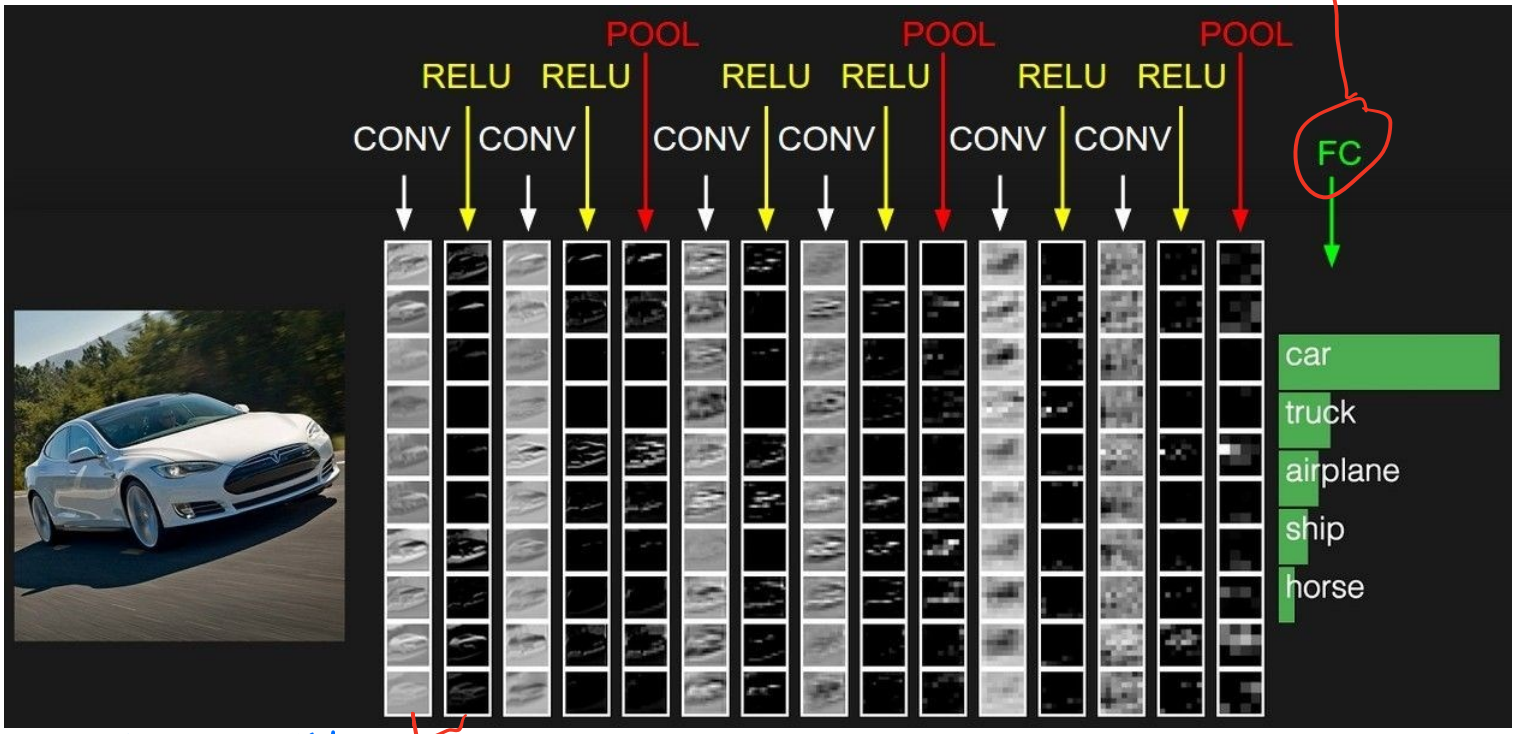
Padding doesn't do anything with resolution of the image

Pooling:- ⇒ helps in reducing overfitting

Max Pooling = take the max of a nxn filter

there is NO learning parameters

Putting it all together



fully connected layer

U_net => it's Fully convNet (without Fc)

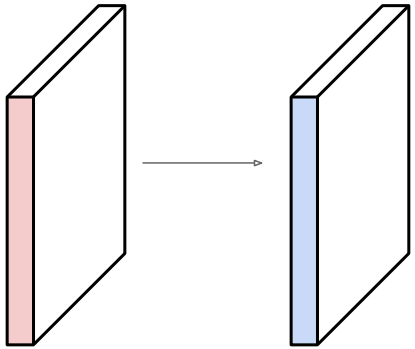
one layer

feature extraction

classification

Examples time:

5 hyperparameters



Input volume: **32x32x3**

10 5x5x3 filters with stride 1, pad 2

(5) and # of layers

Number of parameters in this layer?

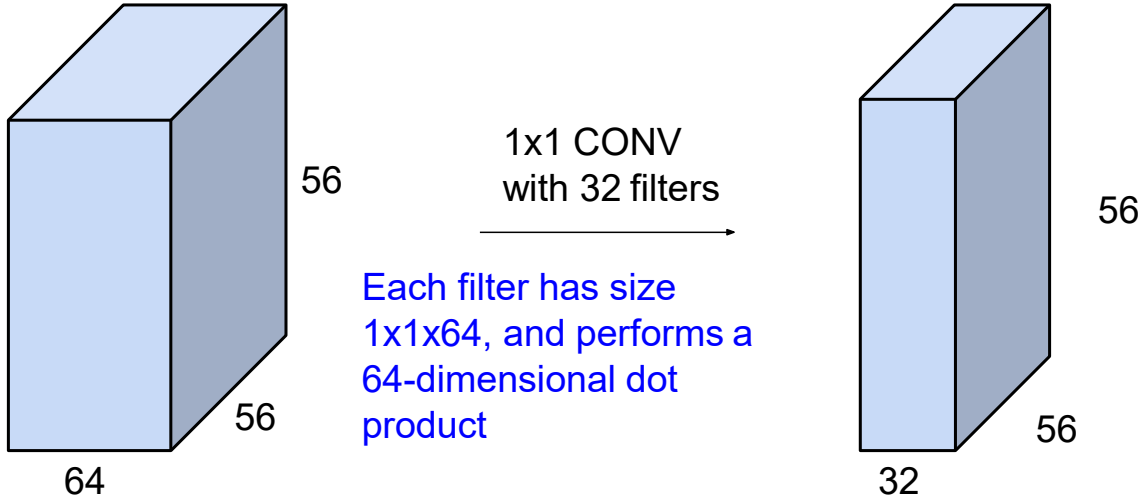
each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

bais

1x1 convolutions

- to reduce the depth of the data (dimensionality reduction)
- used for feature pooling



- because of small kernal size, it doesn't affected by overfitting

09 - Common CNN

① AlexNet

② VGG16 (138M parameters)

③ VGG19

Why use smaller filters?

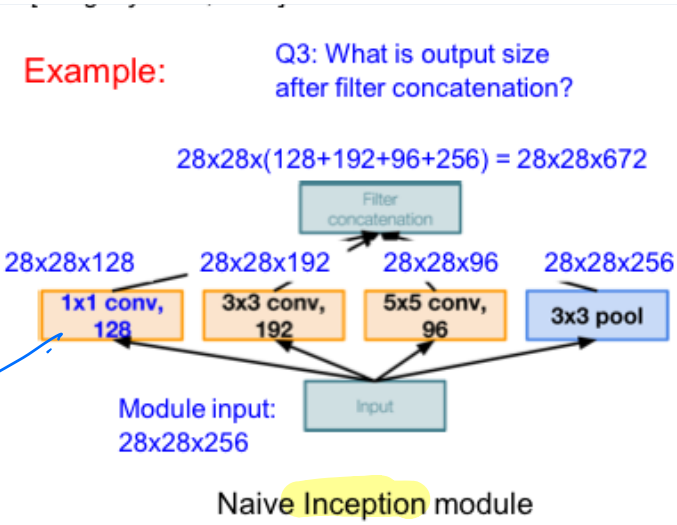
① deeper networks

② more non-linearities

③ fewer parameters

④ GoogleNet

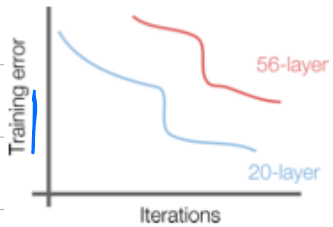
- 22 layers
- No FC layers
- 5M parameters
- 3 outputs

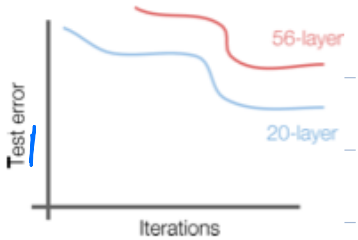


is "bottleneck"

⑤ ResNet

- 152 layers





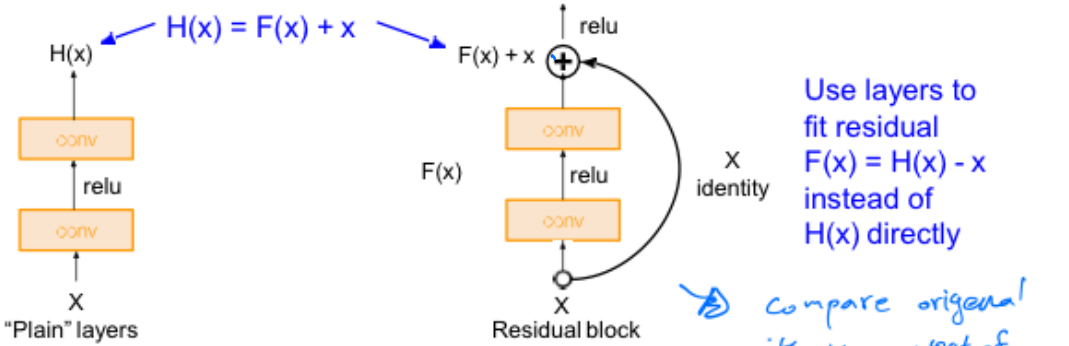
go deeper → overfitting

So

56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it's not caused by overfitting!

I think because of vanishing

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



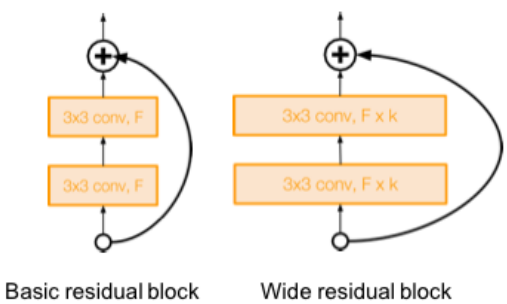
Use layers to fit residual F(x) = H(x) - x instead of H(x) directly

compare original with the output of conv then take the better

So it will give better or same accuracy

⑥ Wide ResNet

- Increasing width instead of depth more computationally efficient (parallelizable)

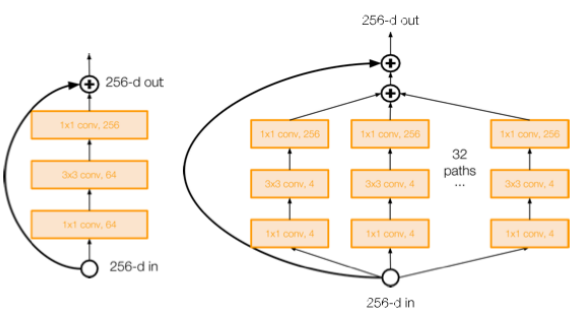


Basic residual block

Wide residual block

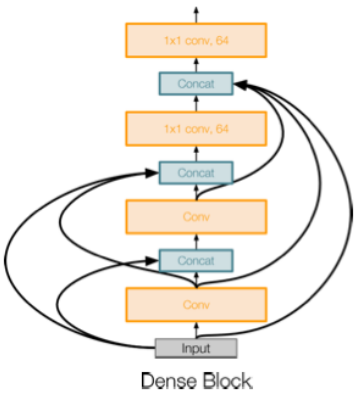
⑦ ResNeXt

apply the Inception module



⑧ DenseNet

- Dense blocks where each layer is connected to every other layer in feedforward fashion



Dense Block

10 - NN Training

1. **One time setup**

   preprocessing, activation functions, weight initialization, regularization, gradient checking

Start with small data to get the perfect parameters

2. **Training dynamics**

   babysitting the learning process, parameter updates, hyperparameter optimization

3. **Evaluation**

   model ensembles

① Data Augmentation (increase # of samples)

- translation shifting and adding padding
- rotation
- stretching
- shearing,
- lens distortions
- ...add noise
  --random crops and scales

Should be reasnable and accepted in the Problem domain

② Preprocessing:-

1- normalize $\frac{X - \mu}{\sigma}$

2- features selection → without modify them

3- features reduction → return new features sorted be important

③ Weight Initialization

❖ Q: what happens when all W=0?

Every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates

Solution

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

vanishing

Solution

weights initialization: "Xavier" initialization ⇒ $W = \frac{\text{rand}(fan\_in, fan\_out)}{\sqrt{fan\_in}}$ → not good with ReLU

"He init" initialization ⇒ $W = \frac{\text{rand}(fan\_in, fan\_out)}{\sqrt{2/fan\_in}}$

high W ⇒ exploding

low W ⇒ vanishing

W and b

Are NOT hyperparameters

④ Activation Functions → to decide whether a neuron will fire or not

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 [redacted]

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

it's not for normalization

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU / PReLU
- Try out tanh but don't expect much
- Don't use sigmoid

- Sigmoid and Tanh doesn't suffer from vanishing and exploding gradient unlike ReLU

- sum of probabilities of all sigmoid unit $\neq 1$

(5) Batch Normalization

what → (preprocessing at every layer of net)

why

- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!

doesn't affect the period of training

How

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

to avoid dividing by zero

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

In testing: we get μ and σ from training

when →
① After FC
② After Conv
③ Before nonlinearity (activation function)

FC

$x: N \times D$

Normalize ↓

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Conv

depth

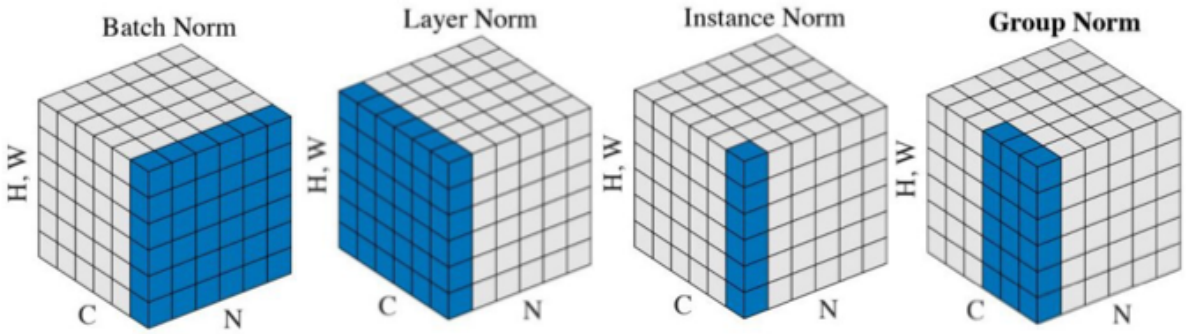
$x: N \times C \times H \times W$

Normalize ↓ ↓ ↓

$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$



6 Babysitting the Learning Process

- Preprocess data
- Choose architecture
- Initialize and check initial loss with no regularization
- Increase regularization, loss should increase
- Then train – try small portion of data, check you can overfit
- Add regularization, and find learning rate that can make the loss go down
- Check learning rates in range [1e-3 … 1e-5]
- Coarse-to-fine search for hyperparameters (e.g. learning rate, regularization)

make it overfit

Do these and monitor:

① loss

② accuracy

if cost = nan ⇒ learning rate is high

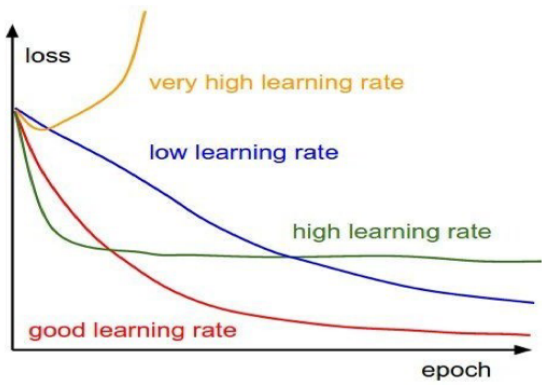
W and b are not hyperparameters

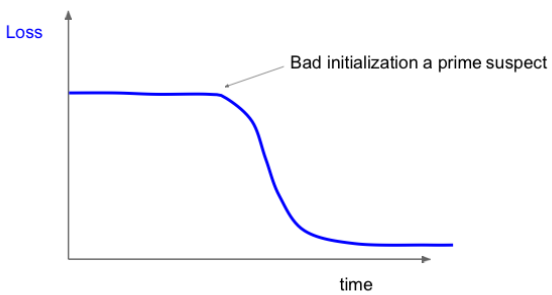
⑦ Hyperparameter Optimization

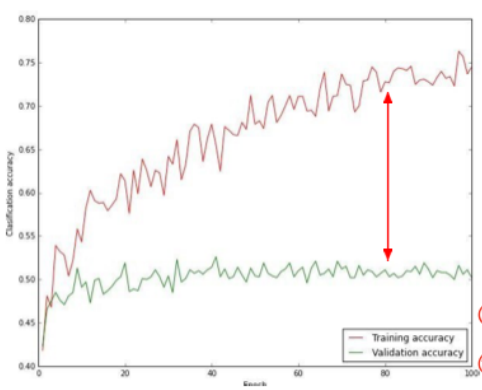
* Start with a few epochs then go longer
* type of search for best hyperparameters: (to get best accuracy)
  - ① grid set values for every hyperparameter → test the combination
  - ② random

✳ **Hyperparameters to play with:**

- ❖ network architecture
- ❖ learning rate, its decay schedule, update type
- ❖ regularization (L2/Dropout strength)







big gap = overfitting
=> increase regularization strength?
or drop out

no gap
① => increase model capacity?
② early stopping (less # of epoch) to avoid overfitting

# of layer
# of neurals
⇓
to improve the accuracy

11 - NN Training

② Optimizers (to optimize gradient) ← learning step

problem with SGD:

① may be very slow

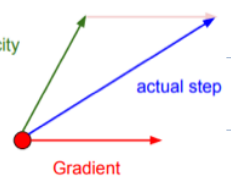
② get stuck with local minimum

③ can be noisy

① SGD + momentum:- (like a ball)

Parameters: rho = 0.9 OR 0.99

Momentum update:



② AdaGrad

Progress along "steep" (انحدار) directions is damped; progress along "flat" (أفقي) directions is accelerated

③ RMSProp (faster)

Parameters: beta1, beta2

④ Adam

AdaGrad / RMSProp

to avoid local minima:-

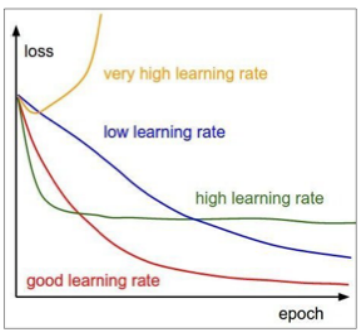
- increase learning rate
- use momentum or adaptive learning
- add some noise while updating weights

In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

(9) Learning rate Schedules

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time

we decay the learning rate at a few fixed point
e.g after every 30 epochs

ways to decay:-

(1) **exponential decay:**

$$\alpha = \alpha_0 e^{-kt}$$

(2) **1/t decay:**

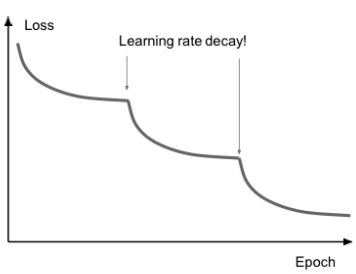
$$\alpha = \alpha_0 / (1 + kt)$$

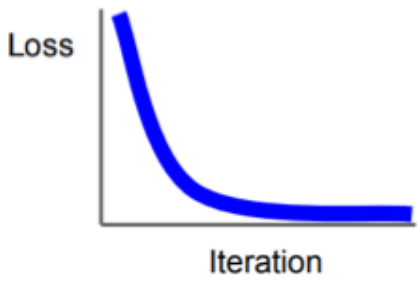
k → constant
t → epoch

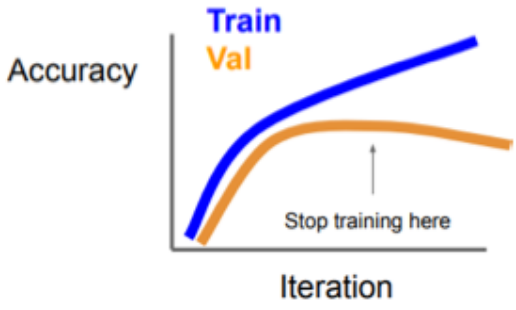
(3) **Cosine:** $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

(4) **Linear:** $\alpha_t = \alpha_0(1 - t/T)$

(5) **Inverse sqrt:** $\alpha_t = \alpha_0/\sqrt{t}$







- Stop training the model when accuracy on the validation set decreases

(10) Choosing hyperparameters:- like babysitting (using training set)

Step 1: Check initial loss → without decay
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer → Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay
Step 6: Look at loss curves

Loss not going down? LR too low, bad initialization
Loss explodes to Inf or NaN? LR too high, bad initialization

Improving single-model performance is easier and more effective than train multible independent models.

(11) Improve single-model performance:-

by Regularization and Dropout

(1) Regularization ⇒ add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$

(2) Dropout ⇒

What → In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common

So → each epoch will have different # of neurals

Why →

- ❖ Dropout forces the neurons to learn "rich" and redundant patterns
- ❖ E.g. without dropout, a noncompressive layer may just "clone" its input to its output
  - ✧ Transferring the task of learning to the rest of the network upstream
- ❖ Dropout forces the neurons to learn denser patterns
  - ✧ With redundancy

We can apply dropout in hidden and input layers

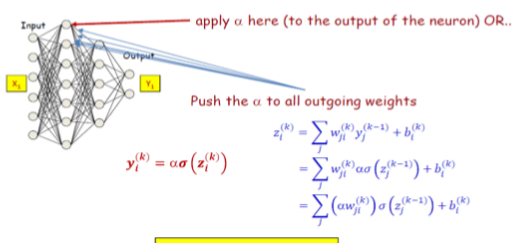
At test

❖ At test time all neurons are active always

⇒ We must scale the activations so that for each neuron:

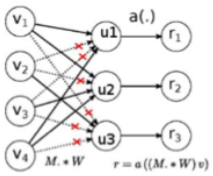
output at test time = expected output at training time

❖ At test time, multiply by dropout probability α



$$W_{test} = \alpha W_{trained}$$

types of Dropout:

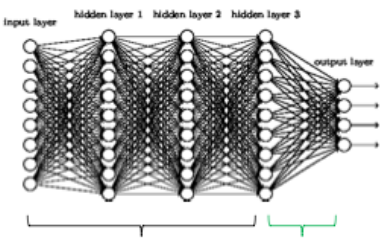


- **Dropconnect** drop some weights
  - Drop individual connections, instead of nodes
- **Shakeout** Scale some weights
  - Scale up the weights of randomly selected weights
    - $|w| \rightarrow \alpha |w| + (1 - \alpha)\, c$
  - Fix remaining weights to a negative constant
    - $w \rightarrow -c$
- **Whiteout** add some noise
  - Add or multiply weight-dependent Gaussian noise to the signal on each connection

Transfer learning

Your data = target

rich data = source



Set these to the already learned weights from another network

Learn these on your own ~~task~~ data

freeze them

More generic

More specific

to what model train on it

my data	Very similar dataset	Very different dataset
Very little data	Use Linear Classifier on top layer (just last layer)	You're in trouble… Try linear classifier from different stages
quite a lot of data	relearn Finetune a few layers	Finetune a larger number of layers

to prevent overfitting:

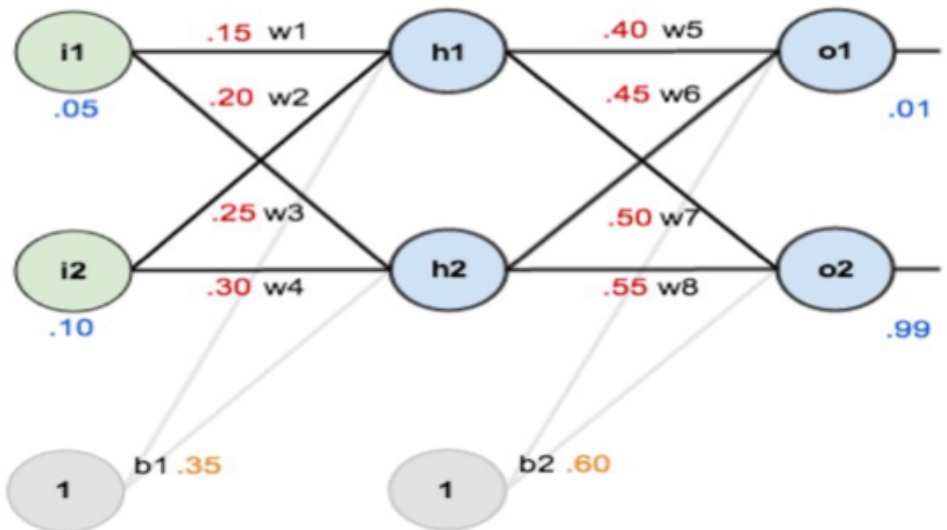
① Data Augmentation

② Dropout

③ Early Stopping

④ Bach normalization

Example to solve



Forward Pass

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

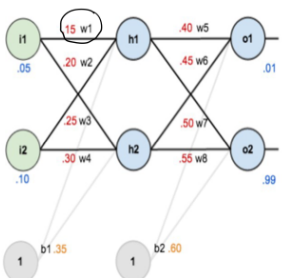
$$out_{h2} = 0.596884378$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$out_{o2} = 0.772928465$$



- Given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

Note\ E is the loss

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

$$w_2^+ = 0.19956143 \qquad w_3^+ = 0.24975114 \qquad w_4^+ = 0.29950229$$

from example 1 find $\frac{\partial E}{\partial w_5}$

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial o_1} \quad \frac{\partial o_1}{y_1} \quad \frac{\partial y_1}{\partial w_5}$$

from example 1 find $\frac{\partial E}{\partial w_1}$

$$\frac{\partial E}{\partial w_1} = \left(\frac{\partial E}{\partial o H_1}\right) \cdot \frac{\partial o H_1}{\partial H_1} \quad \frac{\partial H_1}{\partial w_1}$$

$$\frac{\partial E}{\partial o y_1} \cdot \frac{\partial o y_1}{\partial y_1} \quad \frac{\partial y_1}{\partial o h_1} + \frac{\partial E}{\partial o y_2} \quad \frac{\partial o y_2}{\partial y_2} \quad \frac{\partial y_2}{\partial o h_1}$$

$$\frac{\partial o H_1}{\partial H_1} = \sigma(h_1)\,(1 - \sigma(h_1)) = 0.593 * (1 - 0.593) = 0.2418$$

$$\frac{\partial H_1}{\partial w_1} = i_1 = 0.05$$

$$\frac{\partial E}{\partial o y_1} = -(T_1 - o y_1) = -(0.01 - 0.7513) \quad 0.7413$$

$$\frac{\partial o y_1}{\partial y_1} = \sigma(y_1)\,(1 - \sigma(y_1)) = 0.7513 * (1 - 0.7513) \quad 0.1868$$

$$\frac{\partial y_1}{\partial H_1} = w_5 = 0.4$$

$$\frac{\partial E}{\partial o y_2} = -(T_2 - o y_2) = -(0.99 - 0.773) \quad -0.217$$

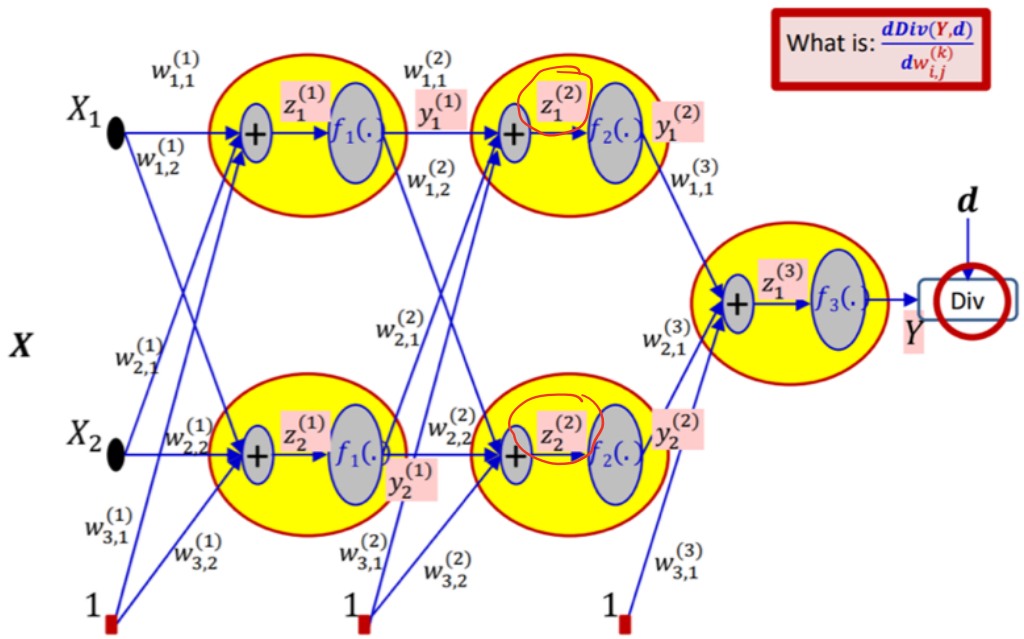
$$\frac{\partial o y_2}{\partial y_2} = \sigma(y_2)\,(1 - \sigma(y_2)) = 0.773 * (1 - 0.773) \quad 0.1755$$

$$\frac{\partial y_2}{\partial H_1} = w_6 = 0.55$$

$0.0554 + (-0.0209)$

0.0345

$$\frac{\partial E}{\partial w_1} = 0.0345 * 0.2414 * 0.05 = 0.000416415$$


What is: $\frac{dDiv(Y,d)}{dw_{i,j}^{(k)}}$
X_1
X_2
X
1
1
1
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{2,1}^{(1)}$
$w_{2,2}^{(1)}$
$w_{3,1}^{(1)}$
$w_{3,2}^{(1)}$
$z_1^{(1)}$
$z_2^{(1)}$
$f_1(.)$
$y_1^{(1)}$
$y_2^{(1)}$
$w_{1,1}^{(2)}$
$w_{1,2}^{(2)}$
$w_{2,1}^{(2)}$
$w_{2,2}^{(2)}$
$w_{3,1}^{(2)}$
$w_{3,2}^{(2)}$
$z_1^{(2)}$
$z_2^{(2)}$
$f_2(.)$
$y_1^{(2)}$
$y_2^{(2)}$
$w_{1,1}^{(3)}$
$w_{2,1}^{(3)}$
$w_{3,1}^{(3)}$
$z_1^{(3)}$
$f_3(.)$
Y
d
Div