keras - DL framework
built on Tensorflow

Numpy - height, width, size

Aspect Ratio

CNN
- pixel, NOT feature extraction

Training
- train, test, validation
- same aspect ratio
- tune params
- ~1000 - 5000 example images/class
- class imbalances

Process
- gather dataset
- preprocess
- split the dataset
- train the classifier
- evaluate

KNN

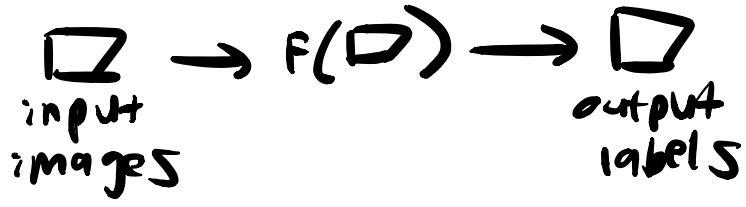
- easy to implement
- not actually "learning"
- requires comparison to ALL data
  - ↳ ALOT of memory → not scalable
- ANN
  - ↳ make estimates
    - → sacrifice "correctness"

Parametrized Learning

- learning the necessary parameters of a given model

Linear classifier

- N images in RGB, sized 32x32 pixels

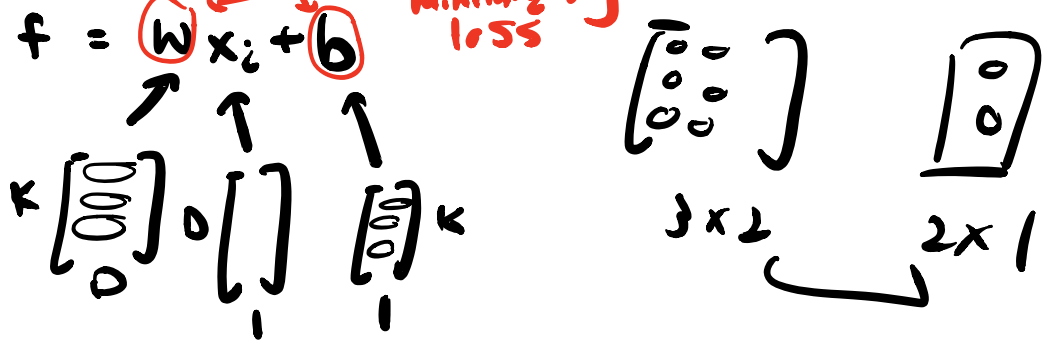


- loss ← minimize

- $D = 32 \cdot 32 \cdot 3 = 3072$ ← image pixels
  (32 with, 32 height, 3 channels)

$L = 3$ ← labels

$f = Wx_i + b$ (optimize by minimizing loss)



$3 \times 3072 \quad 3072 \times 1 \quad 3 \times 1$

Loss

- Hinge loss ← calculates margin (SVMs)

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

j^{th} class ↑ score of correct i^{th} data point

$$s_j = f(x_i, W)_j$$ loss score

- Squared Hinge loss

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

- Avg loss

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

- Cross-entry loss (softmax classifiers)
  - probabilities for each class label

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$ ← bigger the better
- correct value guessed

lower the better

$$= -\log\left[P(Y = y_i \mid X = x_i)\right]$$

negative log likelihood

Optimization

- gradient descent ← updates EACH iteration — slow
  - finds a good enough solution
    even if it only finds a local minima
  - assure it's a convex problem
    (aka that we will find the global minima)

→ combine W and b into one matrix
→ find optimal α learning rate

SGD

```
loop until 1
- epoch done
- loss low
- loss not improved
[ batch = next_training_batch(data, 256)
  W_gradient = evaluate_gradient(loss, data, W)
  W += -α * W_gradient
```

batch

→ better to train with MORE EPOCHS and SMALLER α

- stochastic gradient descent — more noisy, faster convergence, no negative effects to loss/accuracy
  - updates weight via small batches of training data
  - batch size:
    32
    64
    128
    256
  - multiple weight updates per epoch
    → gives model more chances to

learn updates table to fit
weight matrix

- Updates

$$W = W - \alpha \nabla_W f(W)$$

- Momentum

$$V = \gamma V_{t-1} + \alpha \nabla_W f(W)$$

$$W = W - V_t$$

always use (incl. convergence)

→ increase strength of updates in the same direction

decrease strength when gradients switch direction

$\gamma = 0.9$ usually

(or 0.5 until stabilizes)

- Acceleration
  - corrects Momentum
  - use for small datasets

  avoid in large datasets

- Regularization
  - helps w/ generalization

  prevents overfitting

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W)$$

$$W = W - \alpha \nabla_W f(W) - \lambda R(W)$$

L2 regularization (weight decay)

$$R(W) = \sum_i \sum_j W_{ij}^2$$

L1 regularization

$$R(W) = \sum_i \sum_j |W_{ij}|$$

Elastic Net

$$R(W) = \sum_i \sum_j \beta W_{ij}^2 + |W_{ij}|$$

Neural Networks

- neuron fires or doesn't fire

input → weighted sum → activation function → output

repeat

- activation functions ← help learn complex data
  linear → nonlinear

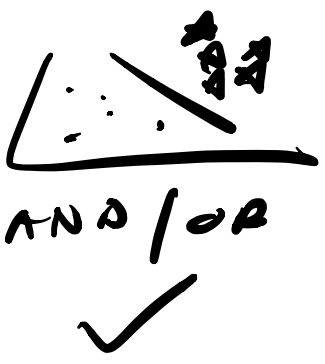
  - step ✗
  - sigmoid ✗
  - tanh ✗
  - reLU ✓
  - leaky reLU ✓
  - ELU ✓

- tips

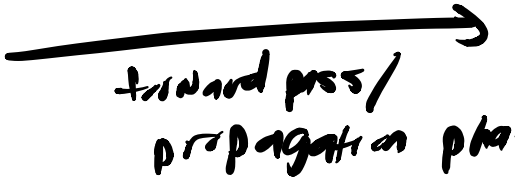
  use ReLU for baseline

→ ReLU
→ ECU

Perceptron
- linear
- update to until get all correct predictions



Backpropagation



$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

w: weight
E: loss
o: node output
net: net output

XOR
- can't be solved w/ one layer (perceptron) looping
  b/c not a linear problem
- requires multiple layers (NN)
  which requires backprop to update weights

(similar to looping in perceptron)

NN w/ backprop allows us to learn patterns in datasets that are nonlinearly separable

Backprop
- generalization of gradient descent algorithms specifically used to train multi-layer feedforward networks
- requires:
  - forward pass
  - backward pass

MNIST vs CIFAR
- basic feedforward network w/ only fully-connected networks not suitable for complex images

⇒ CNN !!

NN System
1. dataset
2. model / architecture
3. loss function
4. optimization method

creating own NN is long/prone to bugs

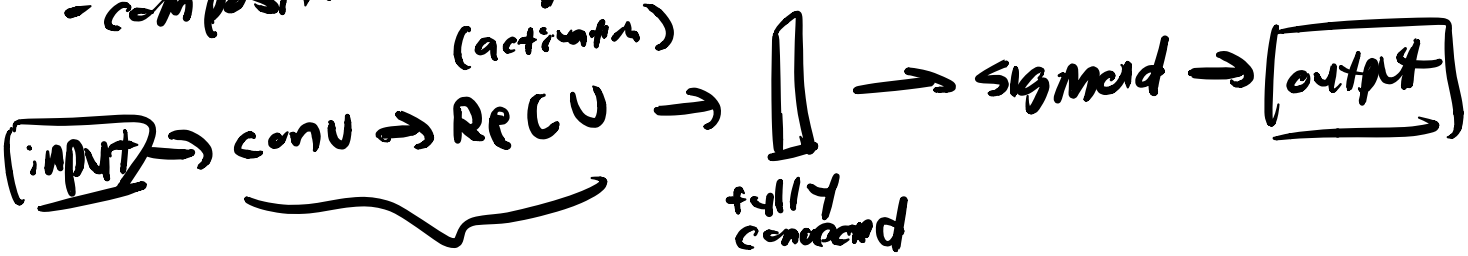
→ use
- keras
- theano
- tensorflow
- pytorch

weight initialization

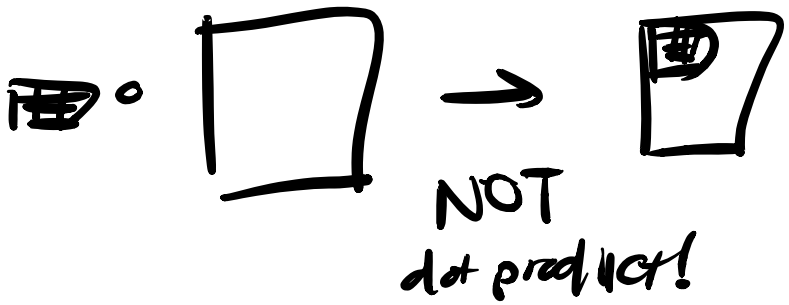
- constant
- uniform / normal distributions
  - LeCun
  - Glorot / Xavier
  - He et al / Kaiming / MSRA

CNN

- local invariance
- compositionality ← using lower-level layers to learn high-level features

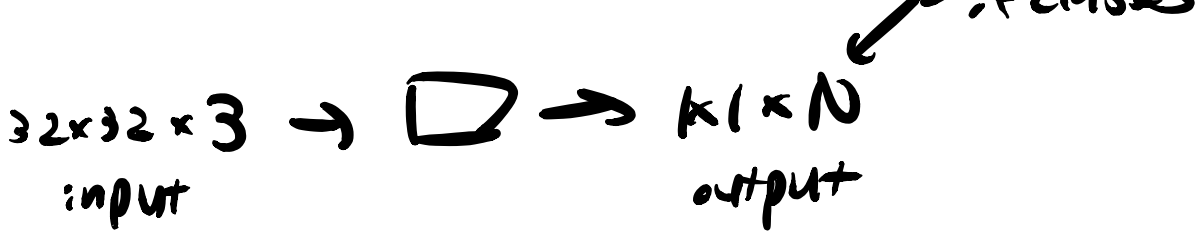


- convolution



- must use odd kernel sizes to ensure valid center coord
- apply rescale to [0, 255] b/c CNN can increase value

- local connectivity
  - CNNs are 3D: width, height, depth/channels
  - subsequent layer neurons only connected to a small region of layer before via conv
  - decreases # of parameters



- layer types
  - conv
  - activation
  - pooling
  - fully-connected
  - batch norm
  - dropout

★ CONV
★ ACT/RELU ← always assumed to be included
★ POOL
★ FC
BN
DO

★ critical to architecture
usually not considered their own "layer"

ie. INPUT → CONV → RELU → FC → SOFTMAX

- local connectivity
  - connect each neuron only to a local region of the input volume (receptive field)

ie. 32x32x(3) input volume

3x3 receptive field / kernel size

(3) channel depth

⇒ 3×3×3 = 27 weights

ie. 16×16×94 input volume

3×3 receptive field / kernel size ← F×F filter

(94) channel depth

⇒ 3×3×94 = 846 weights

- size of output volume
  - depth ← # of filters/channels (K)
    depth column: filters looking at the same region
  - stride
    - for each step, create a new depth column around local region of image where we convolve each of the K filters
    - size of output in 3D volume
      - small stride → larger output volume
      - large stride → smaller output volume
    - reduce spatial dimensions of input
  - zero-padding size
    - output volume matches input volume
    - critical for deep CNN that apply multiple filters on top of each other

- max output size

• compute

$$\left(\frac{W-F+2P}{S}\right)+1$$ ← MUST be integer

W: square input image

F: filter size

S: stride

P: padding

- output of conv layer $W_{out} \times H_{out} \times D_{out}$

$$W_{out} = \frac{W_{in}-F+2P}{S}+1$$

$$H_{out} = \frac{H_{in}-F+2P}{S}+1$$

$$D_{out} = K$$

- output of activation layer ← immediately follows conv

$W_{out} = W_{in}$

$H_{out} = H_{in}$

$D_{out} = D_{in}$

- output of pooling layer ← between layers after activation

• reduce # parameters/computation

• controls overfitting

• max pooling ← middle of NN

- avg pooling ← final layer
- usually
  S=1, S=2
  pool size 2x2 or 3x3 (early in NN)
- $W_{out} = \frac{W_{in}-F}{S} + 1$

  $H_{out} = \frac{H_{in}-F}{S} + 1$

  $D_{out} = D_{in}$
- common to NOT use pooling in middle of NN, and ONLY use at end w/ FC
  ^ avg pooling

Batch Normalization
- reduces # epochs needed to train NN
- "stabilizes" training ← more stable loss curve
- lower final loss
- placed AFTER nonlinear activation

Dropout
- regularization
- for each mini-batch, randomly drop neurons

- reduce overfitting

Architecture

- CONV, FC, BN ← learn params
- POOL, RELU/ACT ← perform operation

INPUT → [[CONV → RELU] · N → POOL?] → [FC → RELU] · K → FC

$0 \leq N \leq 3$

$0 \leq M$

$0 \leq K \leq 2$

Examples

- INPUT → FC
- INPUT → [CONV → RELU → POOL] · 2 → FC → RELU → FC
- INPUT → [CONV → RELU → CONV → RELU → POOL] · 3
  → [FC → RELU] · 2
  → FC
- INPUT → CONV → RELU → FC
- AlexNet
- VGGNet

Stacking multiple CONV layers before POOL

- allows CONV layers to develop more complex

features before destructive pooling op

- POOL vs CONV
  ↑
  Scatters just NO pool,
  and use conv step size to
  reduce volume size

Rules

- SQUARE input images
  - take adv of linear alg opt libraries
  - common sizes:
    32, 64, 96, 224, 227, 229
- input layer should be divisible by 2 many times
  - tweak filter size & stride
  - allows spatial inputs to be sampled
    down via POOL operation
- conv layer sizes
  - general: 3×3, 5×5
  - only in first conv: 7×7, 11×11
  - too large
    - will reduce spatial dim of vol

too quickly

- stride $S = 1$ ← CONV learn features

  use pool to downsample only

- apply zero-padding
  to match input/output dim

- max pooling
  - for downsampling
  - 2x2 receptive field size
    $S = 2$ stride

    $NO \geq 3$

- batch normalization
  - expensive op
    → SIGNIFICANT INCREASE IN COMP TIME
  - use after activation
  - stabilize training to tune hyperparams

- drop-out
  - $p = 0.5$
  - in between FC layers

- (sometimes) btwn pool & conv,
  p=0.1 no.25
- battles overfitting

- pool → conv
  - initially explore/learn w/ pool

  maxpool → conv → avg pooling

- rotation/scale — scale
  - not exactly rotation invariant
    but filters have learned

- translation
  - CNN EXCELS

Keras

- settings
  - epsilon: $1e^{-7}$
  - float32
  - channels_last ← opencv: rows, cols, channels
  - backend: tensorflow

- CIFAR
  - very easy to overfit due to limited # of low res training samples
  - want:
    - boost accuracy
    - reduce overfitting
- shallownet
  - input → conv → relu → fc → output

Model serialization
- saving & loading a trained model

```
model.save(filepath)
load_model(filepath)
```

LeNet + MNIST ← benchmarking

hello world!!! ← gradient based SGD

- recognizing handwritten digits

INPUT → [CONV · TANH · POOL] · 2 (RELU)
→ FC → TANH → FC → SIGMOID (RELU)

"hidden"

4 layers!
- 2 CONV
- 2 FC

→ OUTPUT

mnist.load_data
→ 784-d vector
(28×28 grayscale)

- deep
  - → CNN layers increase
    - spatial input dim decrease

Python Imports

1. Network Architecture
   - ie. ShallowNet, LeNet
2. optimizer
   - ie. SGD
3. preprocess / split test train
4. classification Report

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$32 \times 32 \times 3$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
POOL	$16 \times 16 \times 32$	2×2
DROPOUT	$16 \times 16 \times 32$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
POOL	$8 \times 8 \times 64$	2×2
DROPOUT	$8 \times 8 \times 64$	
FC	512	
ACT	512	
BN	512	
DROPOUT	512	
FC	10	
SOFTMAX	10	

VGGNet

1. only uses 3x3 filter for CONV layers
2. stack MULTIPLE CONV→RELU layers
   - before applying pool

More Tips

- smaller α learning rate
  - → smaller weight updates
  - → reduces overfitting
- USE batch norm!!!

Learning Rate Scheduler

- "learning rate annealing"
  "adaptive learning rates"
- standard weight update formula

$$W \mathrel{+}= -\alpha \cdot \nabla$$

↑
typically
.1 or 0.01

- beneficial to decrease learning rate over time
- learning rate scheduling
  - decrease gradually per epochs
  - drop at specific epochs

- process
  - find good weights early ✗
    use high α
  - tune weights later w/ small α

time-based
- learning rate update formula

$$\alpha = \alpha_0 \left(\frac{1}{1 + decay \cdot iterations} \right)$$

$$decay = \frac{\alpha_0}{\#epochs}$$

drop-based
- drop learning rate usually by:
  - F = 0.5 or order of magnitude
  - every fixed # epochs

$$\alpha_E = \alpha_0 F^{E/D}$$

- E: current epoch
- F: learning rate drop factor
- D: specified drop epochs

faster drop (0.25)
→ can stagnate learning of weights (drop too fast)

BE prepared to spend ALOT of time training/tuning networks!!!

- even simple datasets can take 100s of experiments to get a high accuracy!

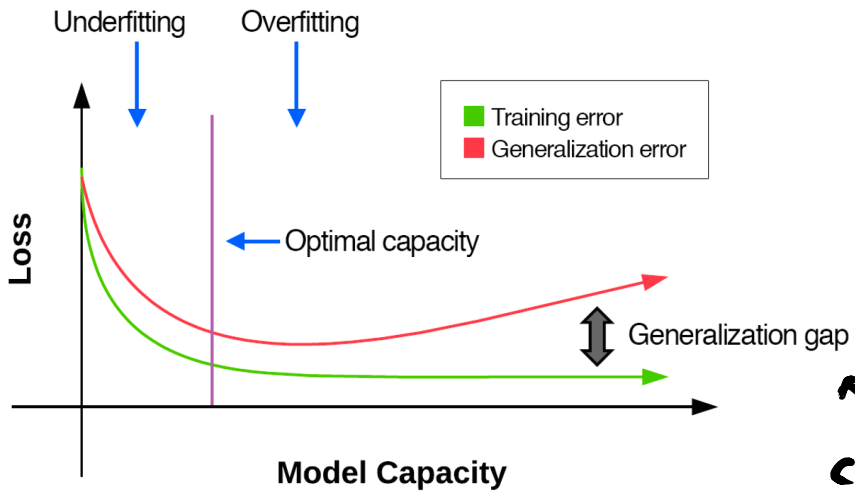
- training deep NNs:
  - part science
  - part art ← intelligent guess-and-check -.-

Spotting Under/over-fitting

- Training Monitor
  - visualize loss graph after every epoch

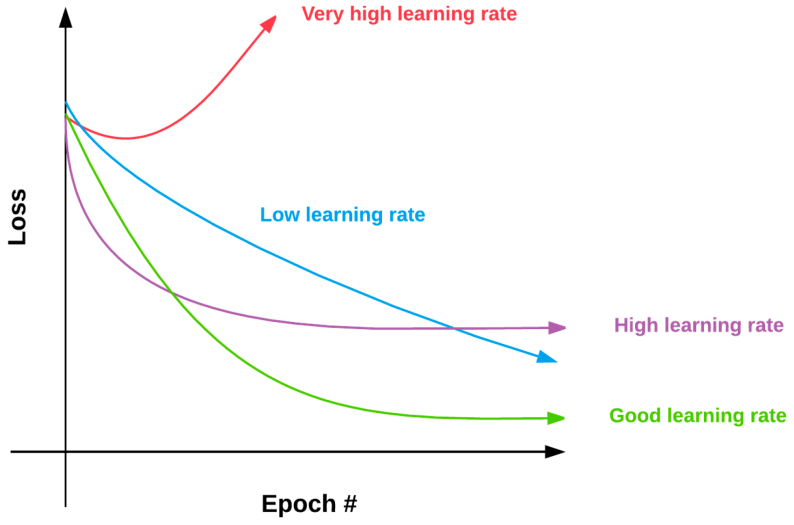
- Training goals:
  - reduce training loss
  - ensure gap btwn traing/testing is reasonably small

- Adjusting capacity of NN
  - increase capacity
    - add more layers + neurons
  - decrease capacity
    - more layers / neurons
    - apply regularization



CONTROL the gap!!!

- Effects of Learning Rates



- Assessing your model

  - are you applying regularization techniques?
  - is the learning rate too high?
  - is your network too deep?

- If validation loss < training loss (AWESOME!)
  - data augmentation during training
    → form of regularization
    enables better generalization
  - not training "hard enough"
    → try increasing α learning rate
    & tweaking regularization strength

- If you think you see signs of underfitting
  → train for another 10-15 epochs to ensure

- If underfitting
  → add more neurons

- If overfitting
  → various regularization techniques

checkpointing

- combines:
  - save/serialize models to disk
  - monitoring under/overfitting

⇒ Keras: ModelCheckpoint (built-in callback)

- 2 methods:
  - checkpoint incremental improvements
  - checkpoint only the best model

Architecture visualization

- used for:
  - debugging ← HIGHLY ENCOURAGED
  - publication

- visualizes
  (batch_size, height, width, depth)

Goal of ImageNet challenge

- classify input image into 1000 categories

Pretrained Networks

- model size based on # parameters

  NOT # train data

- Keras
  - VGG16
  - VGG19
  - ResNet50
  - InceptionV3
  - Xception

  ] works out of the box!

VGGNet

- only 3x3 filters
- very slow to train
- large weights ~500MB

ResNet

- deep networks trained via only SGD
- global avg pooling
- smaller weights than ResNet ~100MB

] 224x224 input

Inception V3

- multi-level feature extractor
- smaller weights than VGGNet and ResNet
- originally called "GoogLeNet"

~90-100MB

Xception

- extension of Inception
- smallest pretrained networks included in the Keras library!

~90-100MB

299 x 299 input

SqueezeNet

- not included in Keras
- extremely small weights

~4.9MB

Typical Image Sizes in ImageNet

- 224
- 227
- 256
- 299

Responsible Disclosure

- how to disclose a vulnerability
- contact stakeholders directly
  to ensure they know there is an issue!
  → as opposed to immediately
  posting vuln. online!

Dataset organization

/root/classname/imagename.jpg

→ /dataset/{1-9}/example.JPG

ROI

"region of interest"

Breaking captchas

- case study
  - download set of images
  - label/annotate images for training
  - training a CNN on your custom dataset
  - evaluating/testing the trained CNN

- obtaining and labeling a dataset can be half (if not more the battle!)

  → try to use traditional CV techniques to speed up labeling process