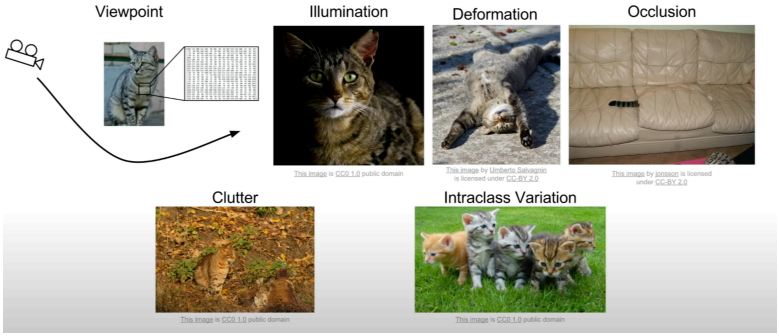
Image representation in memory

- Grayscale (0 - 255)
- RGB : $m \times n \times 3$ → 3 channels RGB
  - ↳ resolution of image
- HSV

Hue - color in the rainbow
Saturation - the 'colorfulness'
Value - Brightness

Factors affecting image processing



https://codewords.recurse.com/issues/six/image-processing-101

↳ Refresher on openCV

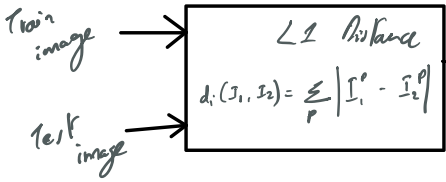
project: counting fingers of a hand

K- Nearest Neighbours

Lecture 2

CIFAR 10
- 10 classes
- 50,000 training images
- 10,000 testing images
- 32 x 32 x 3

- How can two images be compared using KNN?



→ Pixel wise absolute value difference (ex: 374)

- The least difference means the closest neighbours

```
distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
                          train    test images
min_index = np.argmin(distances)
Ypred[i] = self.ytr[min_index]
                    train labels
```

If we consider classes of 'k' neighbors to find the class of test data by majority vote; it is called KNN, if k=1 it becomes Nearest Neighbour

Flexibility

- By modifying the distance measurement KNN can be used very generally in many fields
- eg: If the distance between 2 paragraphs can be taken then KNN can be used to compare two text files
- Similarly KNN can be adopted to compare two things of any types :- text files, images etc
- Simple algorithm

Hyperparameter tuning for KNN

- What value of k to use ?
- What distance metrics to use ?

L_1 or L_2

L_1 ↗ Manhattan distance

L_2 ↖ Euclidean distance

A. Trial & Error since it is very problem dependent

Setting Hyper-parameters

Idea #1 : Choose hyperparameters based on training dataset accuracy alone (No splitting, entire dataset is used for accuracy)

BAD Approach : Leads to overfitting ; not sure if algorithm will generalize in unseen data

Idea #2 : Split data into train & test, choose hyperparameters that work best on test dataset

BAD approach : Test data is fixed, may be choice of hyperparameters only work for that set of test dataset. The choice of test dataset limits the accuracy for unseen data

Idea #3 : Split the data into train, validation & test, choose hyperparameters that work best on val data and evaluate on test dataset

Better !!

- most of the data goes to training set

① Train algorithm on many different choices of hyper-parameters on training set

② Evaluate each of these algorithms on validation set

③ Select the hyperparameter that did best on validation set and → Repeat to find the best set of hyperparameters

④ Run it once on the test set

↳ Find accuracy of the selected hyperparameters

⑤ Note: So typically when writing a paper or creating a report the test dataset is touched only in the last part of the evaluation.

*** Idea #4 : Cross-Validation

Split the data into folds; try each fold as validation & average the results

5 fold validation

- Cross validation is useful for small datasets
- but not too frequently in deep learning
- Because it is very computationally expensive

. Need to calculate L2 distance with all images in dataset

- This means that L2 distance is not capturing the differences properly
- Making small insignificant changes in images belonging to same class may lead to them classified as different classes
- A single image is boiled down to a single distance metric
  - This is over-simplification in case of images.

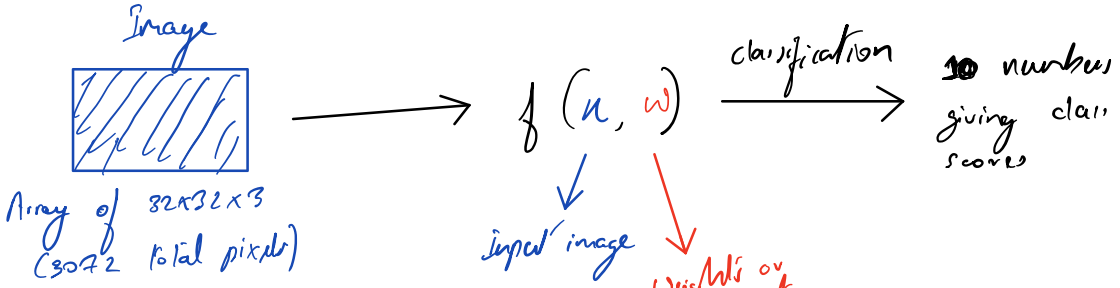
Linear Classifiers

Lecture 2

- Linear classifiers are main building blocks of a neural network
- You can consider a neural network as lego blocks
  - comprising of blocks of different types joining together
- Ex: Image Captioning Model ⇒ CNN → for image recognition + LSTM → for language model

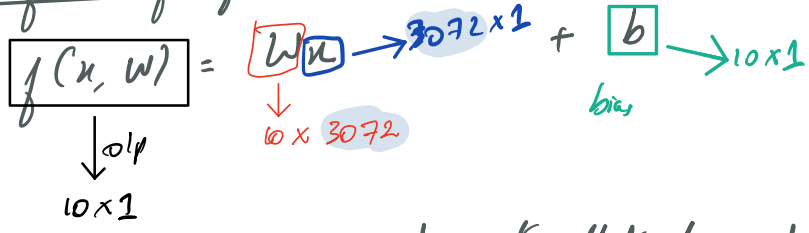
Parametric Approach

- Linear Classifiers are the simplest examples of parametric models
- After training the model the whole intelligence of the model is captured in weights & biases
  - Now we no longer require the training dataset
- KNN on the other hand requires the training dataset to compute the L2 distance with each test image at runtime

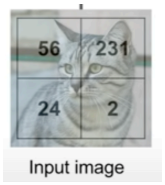


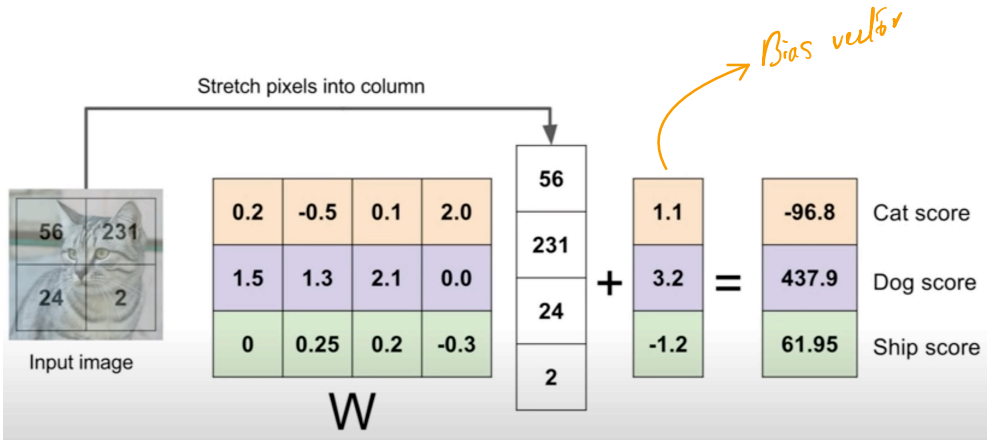
- The whole story of deep learning is to derive the right function 'f' that combines input image with weights
- Simplest form of 'f' is multiplication

ie; $f(x, W) = Wx + b$



- Bias is a constant vector of 10 elements that does not interact with the training data and instead gives us some data independent preferences for some classes over another.


56
231
24
2
Input image

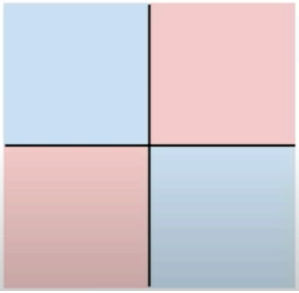

Stretch pixels into column
Bias vector
56
231
24
2
Input image
0.2
-0.5
0.1
2.0
1.5
1.3
2.1
0.0
0
0.25
0.2
-0.3
W
56
231
24
2
+
1.1
3.2
-1.2
=
-96.8
437.9
61.95
Cat score
Dog score
Ship score


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

Hard cases for a linear classifier

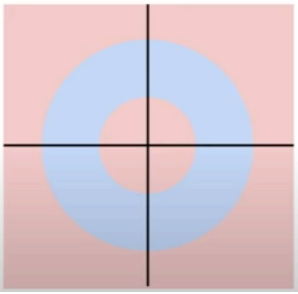
Class 1:
number of pixels > 0 odd

Class 2:
number of pixels > 0 even



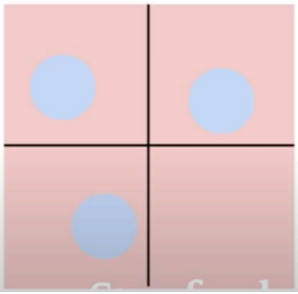
Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



Class 1:
Three modes

Class 2:
Everything else



Lecture 3 : Loss Function & Optimization

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

- N: # of Samples
- L_i: Loss function
- x_i: Image
- y_i: Integer label

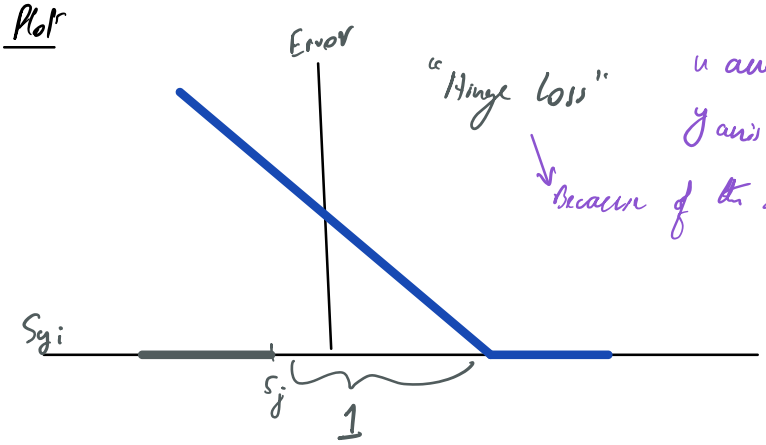
$$\begin{aligned} L_i &= \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} \\ &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \end{aligned}$$

y_i

$s_j - s_{y_i} + 1$

- s_j: Sum of s over all the other classes
- s_{y_i}: Score of actual class
- 1: bias

Plot



"Hinge loss"

x axis: s_{y_i} (score)

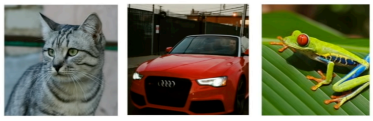
y axis: Error

↳ Because of the shape of the graph

- Error linearly decreases until a threshold where all classes are accurately classified

the loss basically says that we're happy if the score of the true class is much higher than for other classes

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



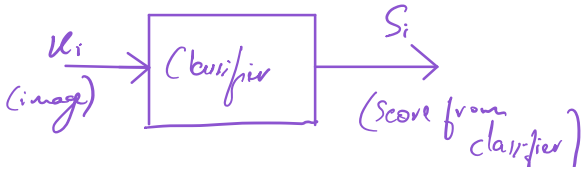
cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9		

the SVM loss has the form:

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

= 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

These are S_i values



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

- choice of +1 is arbitrary, even if we change the value we can observe that the end loss is the same
- We do not care about the absolute value of the scores.
- We only care that the correct score is higher than the relative score for wrong classifications.

Questions on Hinge Loss or Multi-class SVM loss

(1Q) What happens to loss if car scores of the car image a little bit?

(A) - No change, since car is already classified correctly
- Since SVM score only cares of the fact that the correct score is one more than the incorrect scores.
- Car score is already quite a bit larger than the other scores.

(2Q) What is the min/max possible loss for SVM loss

(A) min $\Rightarrow$ 0 (Across all classes, if correct scores are much larger)
max $\Rightarrow -\infty$

(3Q) At time of initialization, W is assigned small random values as a result initially all $s \simeq 0$. What is the loss in this case? That is the loss when all s is approximately 0.

(A) C - 1; where C is the number of classes.
- We are looping over all the incorrect answers; C - 1 iterations
- for each step loss is little over 1 (approximately to 1)

- This is useful for debugging
  - When you start training, after first iteration if loss ≠ C-1 then probably there's a bug in the code

(4a) What if sum was over all the classes? (including $j = y_i$)

(A) - loss increases by 1
- Loss will never be zero, minimum loss will always be 1

5a) What if we use mean instead of sum here?

$$L_i = \frac{\sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}{\text{here we mean instead!}}$$

(A) No change, final loss will be rescaled by a constant since the number of classes is fixed therefore denominator in mean; the number of classes is always a constant

(6a) What if we used squared term

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

Would this end up with the same classifier?

(A) No,
- Loss function is completely different in a non-linear way
- Wrong predictions are penalised more when we use squared loss
- This kind of squaring is used when we want to make sure that wrong classifications are treated as a really bad thing

- When squared is not taken, a small difference in classification and a large error in classification is almost the same thing

* Loss function decides how we weigh off different types of mistakes made by the classifiers.
- if we use squared loss then each bad classification is treated as squared bad value => really really bad
- Hinge loss doesn't care between if we are little bit wrong and a lot wrong.
- Difference btw linear & squared error is how we quantify how much we care about different categories of error.

Sample Code : Multi-Class SVM loss

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

```
def L_i_vectorized(x, y, W):
    scores = W.dot(x)
    margins = np.maximum(0, scores - scores[y] + 1)
    margins[y] = 0   # make y_i zero
    loss_i = np.sum(margins)
    return loss_i
```

- Vectorized trick to zero out the margin corresponding to correct class.
- Iterates over all but one class and then take sum

$$f(x, W) = Wx$$

$$L = \frac{1}{N} L_i$$

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

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

(Q) Suppose that we found a W such that $L = 0$. Is this W unique?

(A) No!! 2 times W also has $L = 0$ (if W has $L = 0$)

Any scaled version of W has $L = 0$.

(Q) If there are multiple values of W for which $L = 0$ then how does classifier find the best value of "W".

Optimize value of 'W'

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

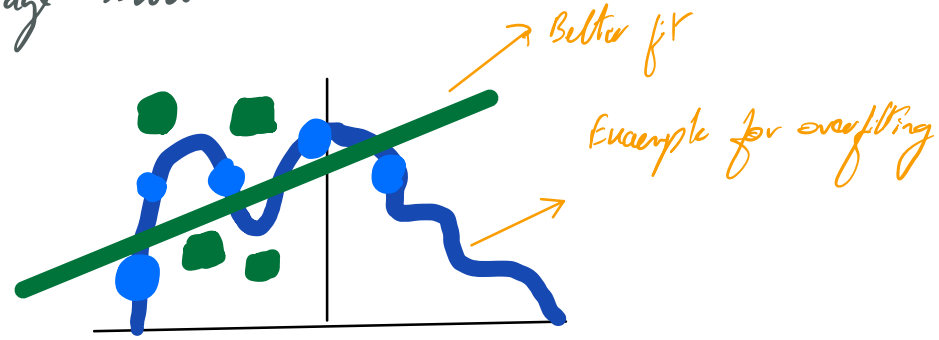
amount of regularization (hyper parameter)

Data loss: Model predictions should match training data

Regularization: Model should be "simple", so it works on test data

- The data loss only means that the fit in train dataset is as perfect as possible. It doesn't consider generalisation

- This leads to tendency of model to overfit as shown in image below



- In the above image box is lead in the blue algorithm.
- It fits all the blue circles perfectly
- But it fails to generalise the field

* Example of regularisation

- For polynomial regression, suppose we get good fit for lower order equation & higher order equation
- In higher order, there is a good chance of over-fitting
- Therefor regularization over penalize equations of higher order so that lower order equation are preferred

> Occam's Razor :
> "Among competing hypothesis, the simplest is the best"

- Regularization parameter 'λ' is one of the hyper-parameters that we arrive at during hyper parameter tuning.

* Regularization

λ = Regularization strength

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

Common Regularization techniques :-

1) L_2 Regularization (or weight decay) : $R(W) = \sum_k \sum_l W_{k,l}^2$ → Common
2) L_1 Regularization : $R(W) = \sum_k \sum_l |W_{k,l}|$
3) Elastic net ($L_1 + L_2$) : $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$
4) Max norm regularization
5) Dropout → Common in deep learning
6) More fancier techniques : Batch normalization, stochastic depth ← Common in deep learning

- The whole idea of regularization is that it somehow penalizes the complexity of the model rather than explicitly trying to fit the training data

* Example of L_2 regularization

- Suppose we have an input vector $x = [1, 1, 1, 1]$.
  For this input vector if $w_1 = [1, 0, 0, 0]$,
  $w_2 = [0.25, 0.25, 0.25, 0.25]$
  In case of both these weights L_2 is the same ie; $w_1^T x = w_2^T x = 1$

$$R(w_1) = \sum_k \sum_l W_{k,l}^2 = w_1^2 + w_2^2 + w_3^2 + \ldots + w_n^2 = 1$$

L_2 Regularization $R(w_2) = 0.25^2 + 0.25^2 + 0.25^2 + 0.25^2 = 0.25$ → Least value

∴ L_2 regularization prefers W_2 over W_1

L_1 Regularization

$R(W_1) = W_1 + W_2 + \dots W_n = 1$

$R(W_2) = 0.25 + 0.25 + 0.25 + 0.25 = 1$

∴ L_1 regularization cannot differentiate between both values.

- The main reason for overfitting is model complexity

  ↑ complex model ⇒ ↑ chance of over-fitting

- Regularization controls model complexity by penalizing higher terms in the model.
- With regularization model tries to minimise both loss and complexity.

Without regularization : minimize ⇒ Loss (model)

With regularization : minimize ⇒ Loss (model) + complexity (model)

* L_1 Regularization

- Also called regularization for sparsity
- Forces the weight of uninformative features to be zero by subtracting a small amount from the weight at each iteration and thus making weight zero eventually.
- Reduces total no of features

* L_2 Regularization

- Also called regularization for simplicity

- forces weights towards zero but it does not exactly make them zero

Note:

Ridge regression uses L_2 regularization

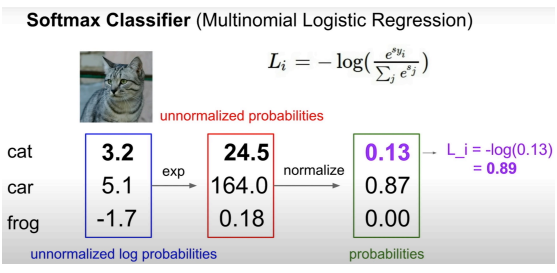
Lasso regression uses L_1 regularization

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{where} \quad s = f(x_i; W)$$

Softmax function

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

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$



L_i

$\nearrow f(x; W)$

$f(x; W) = Wx$ → simplest example of function

Examples of loss functions

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

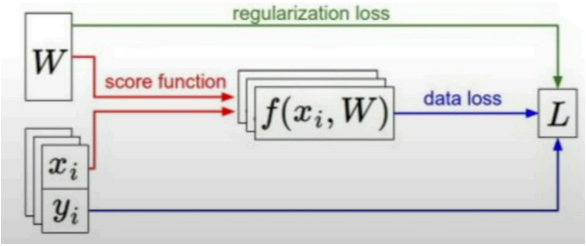
SVM

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Softmax

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

Regularization



This is a pretty generic overview of how any **supervised algorithm works**

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

numeric method for derivatives in 1D

weight vector

current W:	W + h (first dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] loss 1.25347	[0.34 + **0.0001**, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] loss 1.25322	[-2.5, ?, ?, ?, ?,…]

10 elements

h

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

f(x) f(x+h)

Step 1

current W:	W + h (third dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] loss 1.25347	[0.34, -1.11, 0.78 + **0.0001**, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] loss 1.25347	[-2.5, 0.6, 0, ?, ?,…]

(1.25347 - 1.25347)/0.0001
= 0

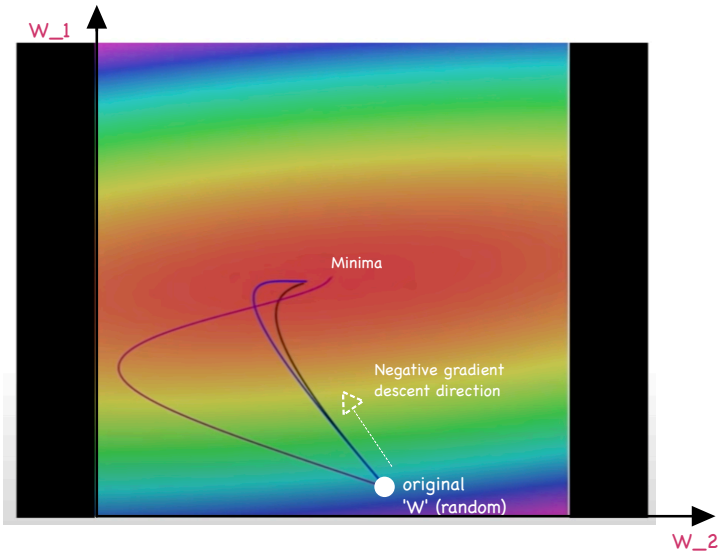
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Step 2

Initial weights are random

also called learning rate (hyper-parameter)

$\nabla_W L$ Mathematical Notation of gradient of Loss with respect to 'W'



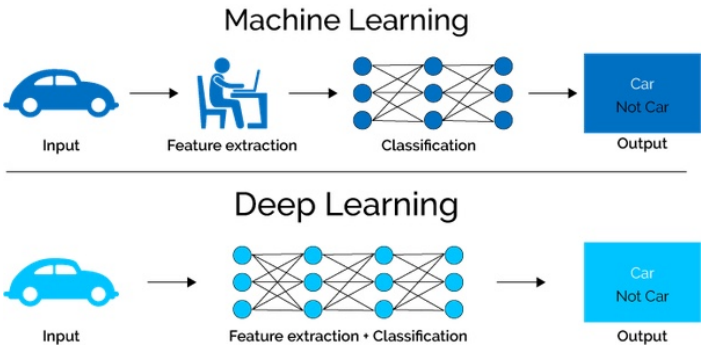
N is very very large

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

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

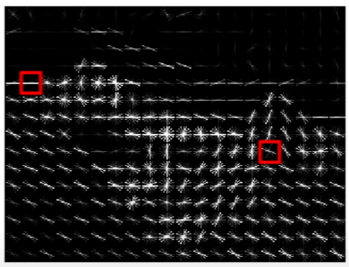
- Taking sum over the whole dataset is computationally expensive
- Calculative derivative is also expensive across the weights and data vector
- There all operations are done approximately on a **minibatch** examples
- ex: 32, 64, 128 samples (power's of 2)

Stochastic Gradient Descent



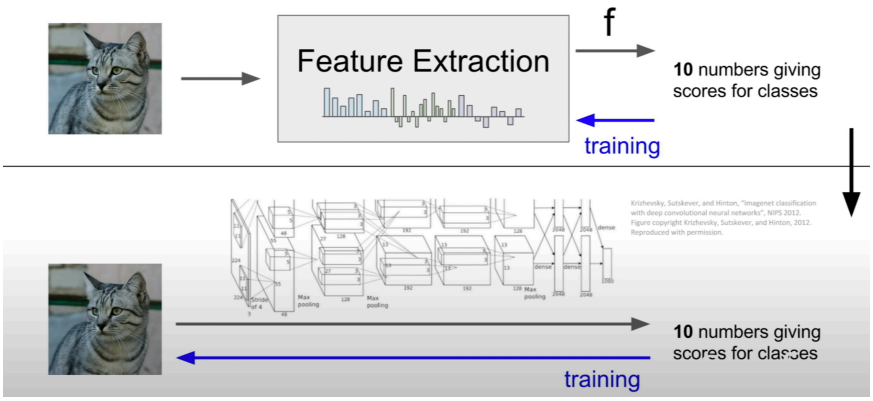
Example: Histogram of Oriented Gradients (HoG)

Divide image into 8x8 pixel regions
Within each region quantize edge direction into 9 bins



Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers

Image features vs ConvNets



- Features extractor is fixed block and is not updated during training stage. Only linear classifier working on top of these features are updated

- Features are not extracted ahead of time
- Feature extractor is learned from data
- Both feature extractor and classifier are updated during training

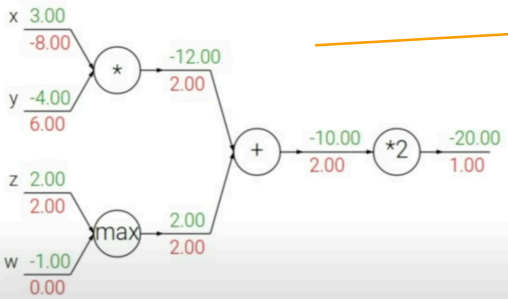
- neural nets will be very large: impractical to write down gradient formula by hand for all parameters
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the **forward**() / **backward**() API
- **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs

Patterns in backward flow

add gate: gradient distributor

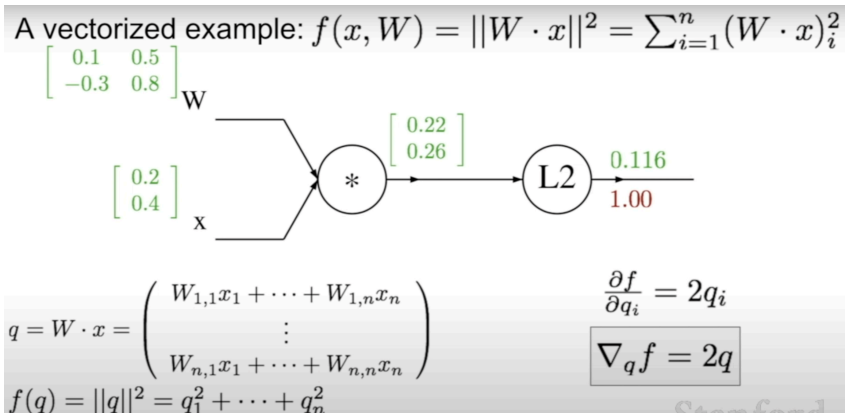
max gate: gradient router

mul gate: gradient switcher



Computational Graph using chain rule

- splits complex derivatives into simpler derivatives
- Green is forward pass
- Red is backward pass

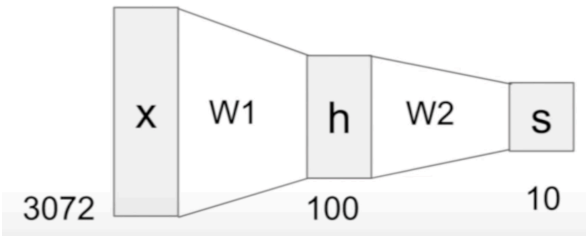


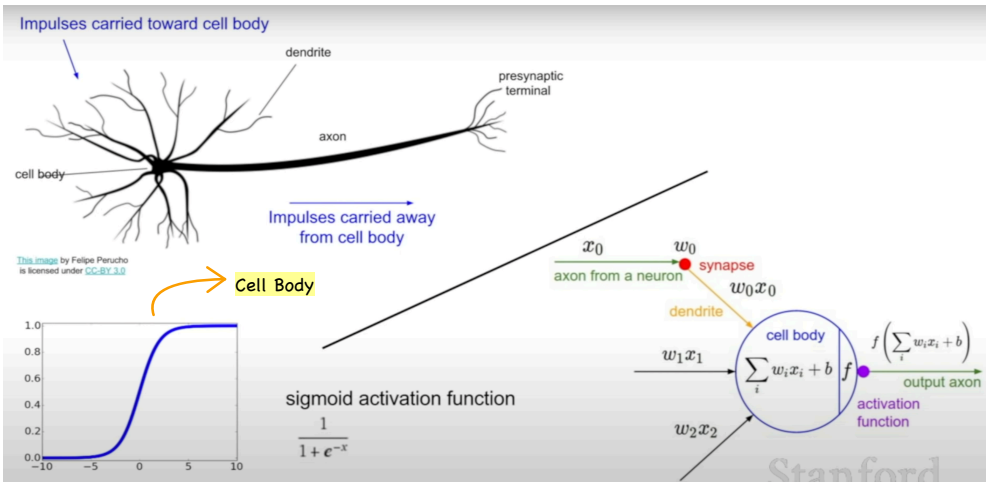
(**Before**) Linear score function: $f = Wx$

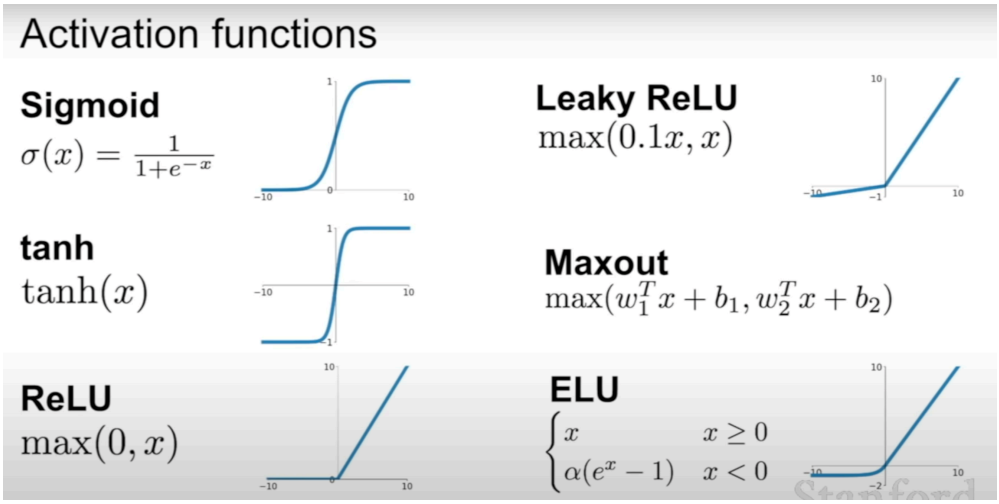
(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

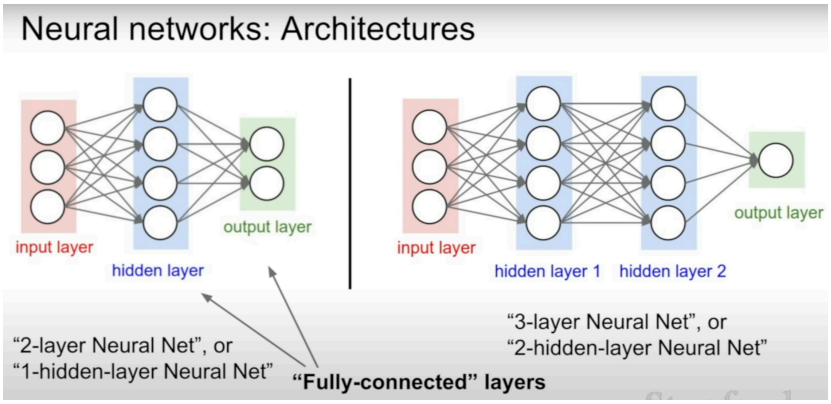
3 layer nerual network => $f = W_3 \max(0, W_2 \max(0, W_1 x))$

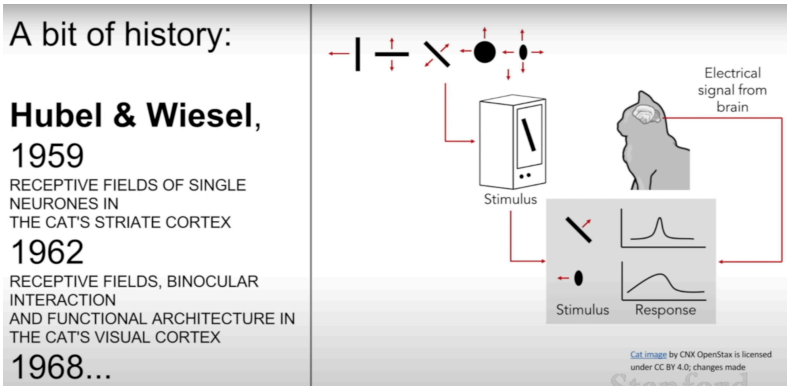
- In linear classifier (img above) the 'Car' is represented as just one template
  - All red
- In non-linear classifier (img right) **W1** could be these individual templates but in **W2** we give weightage to these multiple templates.
  - Here **h** is the value for each template

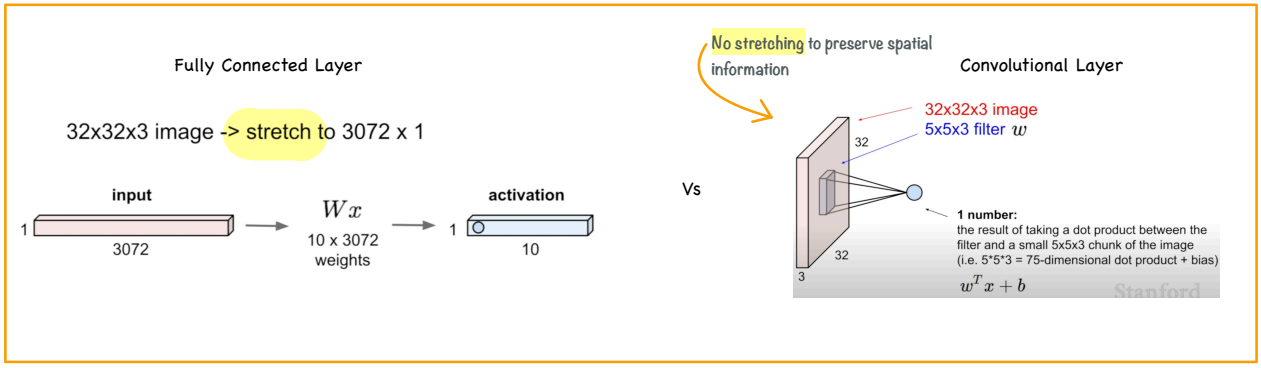


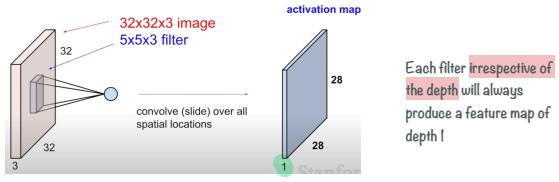

Impulses carried toward cell body
dendrite
presynaptic terminal
axon
cell body
Impulses carried away from cell body
This image by Felipe Perucho is licensed under CC-BY 3.0
Cell Body
sigmoid activation function
$\frac{1}{1+e^{-x}}$
x_0
w_0
axon from a neuron
synapse
w_0x_0
dendrite
cell body
w_1x_1
w_2x_2
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
output axon
activation function
Stanford


Activation functions
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$
Leaky ReLU
$\max(0.1x, x)$
tanh
$\tanh(x)$
Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$
ReLU
$\max(0, x)$
ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
Stanford

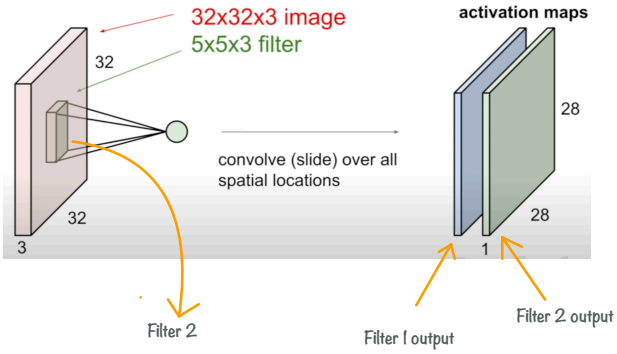

Neural networks: Architectures
input layer
hidden layer
output layer
input layer
hidden layer 1
hidden layer 2
output layer
"2-layer Neural Net", or
"1-hidden-layer Neural Net"
"Fully-connected" layers
"3-layer Neural Net", or
"2-hidden-layer Neural Net"

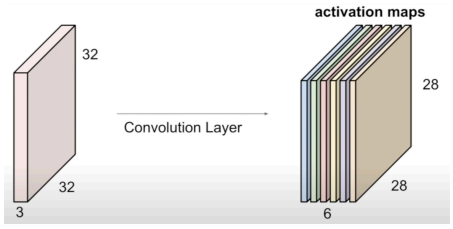






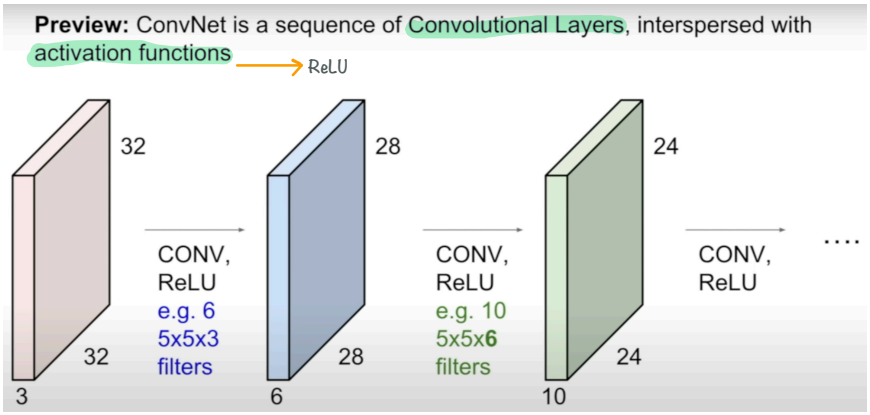
Stacking 6 5 x 5 x 3 filters => "new image" of size 28 x 28 x 6

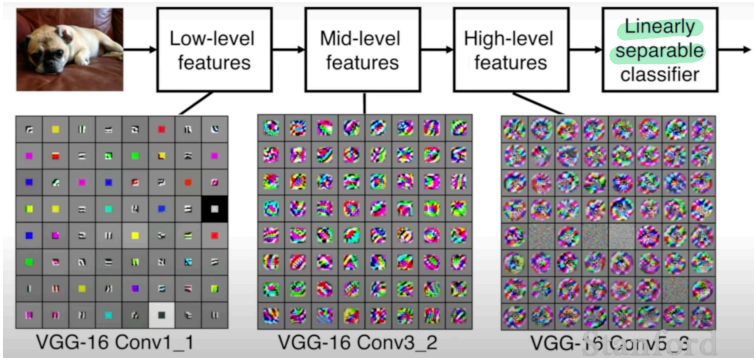


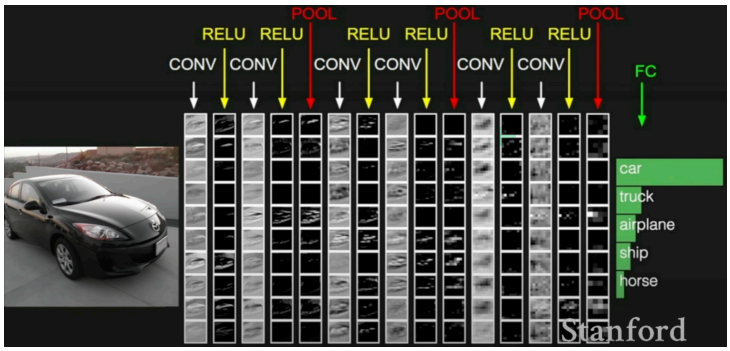


Each filter is:

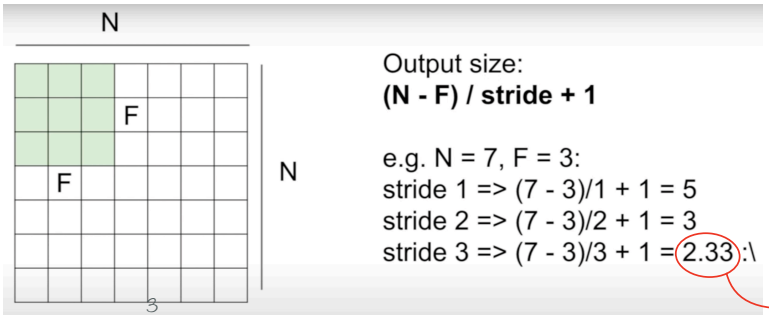
- Looking at same parts of the image
- But for different features







- Pool layer is used to down sample our activation maps
- Last Fully connected Layer gives the final scores from the activation maps



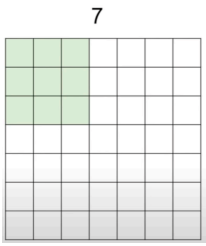
Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

<= Formula to calculate output size of activation map after convolution layer for different values of stride

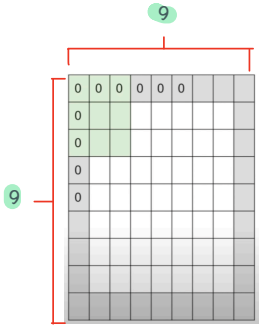
- Filter of size F x F
- Image of size N x N

Wrong, therefore this value of stride is invalid



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on 7x7 input with stride 3.



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

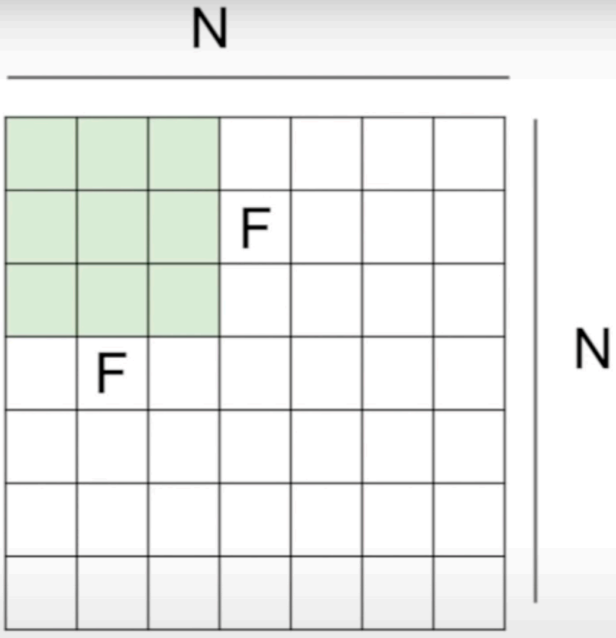
(9-3)/1+1=7

(recall:)
(N - F) / stride + 1

Stanford

in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Stanford



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

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

Example 1

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

Example 2

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params
=> 76*10 = **760**

Each filter has a **Bias** parameter

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$ → Input size
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Example: CONV layer in Torch

SpatialConvolution

```
module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH, [dW], [dH], [padW], [padH])
```

Applies a 2D convolution over an input image composed of several input planes. The `input` tensor in `forward(input)` is expected to be a 3D tensor (`nInputPlane x height x width`).

The parameters are the following:

- `nInputPlane` : The number of expected input planes in the image given into `forward()` .
- `nOutputPlane` : The number of output planes the convolution layer will produce.
- `kW` : The kernel width of the convolution
- `kH` : The kernel height of the convolution
- `dW` : The step of the convolution in the width dimension. Default is `1` .
- `dH` : The step of the convolution in the height dimension. Default is `1` .
- `padW` : The additional zeros added per width to the input planes. Default is `0` , a good number is `(kW-1)/2` .
- `padH` : The additional zeros added per height to the input planes. Default is `padW` , a good number is `(kH-1)/2` .

Note that depending of the size of your kernel, several (of the last) columns or rows of the input image might be lost. It is up to the user to add proper padding in images.

If the input image is a 3D tensor `nInputPlane x height x width` , the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = floor((width  + 2*padW - kW) / dW + 1)
oheight = floor((height + 2*padH - kH) / dH + 1)
```

Summary. To summarize, the Conv Layer:

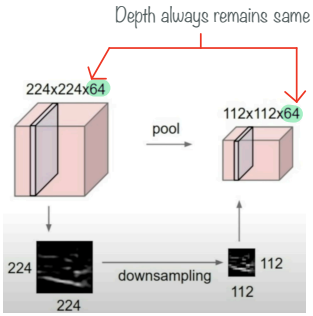
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.

Convolution layer in Caffe

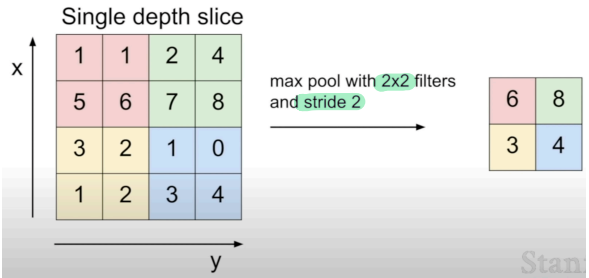
```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96     # learn 96 filters
    kernel_size: 11    # each filter is 11x11
    stride: 4          # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01        # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
  }
}
```

Pooling

- Makes the representations smaller and more manageable
  - The number of parameters required in the final fully connected layer decreases
- Operates over each activation map independently
  - that is; if there are six activation maps; the final depth is also six



Max Pooling



- output size = (N-F)/S +1
- In pooling it is common to use stride so that there is no overlap

Pooling summary

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers