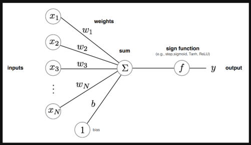
The Perceptron



For a step function,

$$f = \begin{cases} 0 & w.x + b \leqslant 0 \\ 1 & w.x + b > 0 \end{cases}$$

Let z = w.x + b

Activation: a = f(z)

Sigmoid function

$$a = \sigma(z) = \frac{1}{1+e^{-z}}$$

ReLU:
- Adds a non-linearity and is more powerful if you want to stack convolutions.
- Makes gradients sparse. Helps avoid vanishing gradients.
- Really speeds up convergence of gradient descent

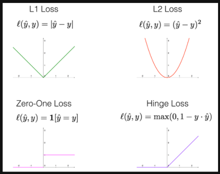
f(x) = max(0, x)
Keeps all of the positive activations and throws away all negative

Softmax function:
converts output values to probabilities

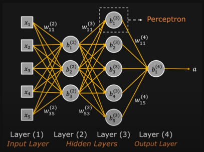
$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

```
def softmax(x):
    """
    x is a numpy array of size [number of examples, number of classes]
    softmax should be done for each row
    """
    res = None

    x -= np.max(x, axis=1, keepdims=True)
    res = np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True)

    return res
```

Loss Function



Multi-Layer Perceptron



Gradient Descent

Change in cost:

From Taylor series,

$$f(x+\Delta x, y+\Delta y) \approx f(x,y) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

$$\Delta f(x,y) \approx \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

Applying to cost function,

$$\Delta C \approx \frac{\partial C}{\partial w_i}\Delta w_i + \frac{\partial C}{\partial b_j}\Delta b_j$$

$$\Delta C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix} \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix}$$

$$\Delta C = \underbrace{\nabla C}_{\text{gradient}} \cdot \underbrace{\Delta v}_{\text{change in weight and bias}}$$

Let $\Delta v = -\eta \nabla C$

$$\Delta C = -\eta \|\nabla C\|^2$$

η = learning rate (how fast to descend)

For each step,

$$w_i \rightarrow w_i' = w_i - \eta \frac{\partial C}{\partial w_i}$$

$$b_j \rightarrow b_j' = b_j - \eta \frac{\partial C}{\partial b_j}$$

Additive update

For general multi-layer neural network,

$$\Delta v = \begin{bmatrix} \Delta w_{jk}^{(l)} \\ \vdots \\ \Delta b_j^{(l)} \\ \vdots \end{bmatrix} \qquad \nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_{jk}^{(l)}} \\ \vdots \\ \frac{\partial C}{\partial b_j^{(l)}} \\ \vdots \end{bmatrix}$$

j: neuron
k: input
l: layer

$$y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$$

$$\frac{\partial y_i}{\partial x_i} = W_{ij} \qquad \frac{\partial y_i}{\partial W_{ij}} = x_i \qquad \frac{\partial y_i}{\partial b_j} = 1$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \cdots & \frac{\partial J}{\partial W_{1k}} \\ \vdots & & \\ \frac{\partial J}{\partial W_{d1}} & \cdots & \frac{\partial J}{\partial W_{dk}} \end{bmatrix} \qquad \frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} \cdot \frac{\partial y_i}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} \cdot x_i$$

$$\Rightarrow \frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial y_1} \cdot x_1 & \cdots & \frac{\partial J}{\partial y_k} \cdot x_1 \\ \vdots & & \\ \frac{\partial J}{\partial y_1} \cdot x_d & \cdots & \frac{\partial J}{\partial y_k} \cdot x_d \end{bmatrix} = \begin{bmatrix} \delta_1 x_1 & \cdots & \delta_k x_1 \\ \vdots & & \\ \delta_1 x_d & \cdots & \delta_k x_d \end{bmatrix}$$

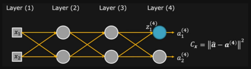
$$\boxed{\frac{\partial J}{\partial W} = X\delta^T \in \mathbb{R}^{d \times k}}$$

$$\frac{\partial J}{\partial x} = \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \vdots \\ \frac{\partial J}{\partial x_d} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{k} \frac{\partial J}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_1} \\ \vdots \\ \sum_{j=1}^{k} \frac{\partial J}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_d} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{k} \delta_j W_{1j} \\ \vdots \\ \sum_{j=1}^{k} \delta_j W_{dj} \end{bmatrix}$$

$$\boxed{\frac{\partial J}{\partial x} = W\delta \in \mathbb{R}^{d \times 1}}$$

$$\frac{\partial J}{\partial b} = \begin{bmatrix} \frac{\partial J}{\partial b_1} \\ \vdots \\ \frac{\partial J}{\partial b_k} \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial y_1} \cdot \frac{\partial y_1}{\partial b_1} \\ \vdots \\ \frac{\partial J}{\partial y_k} \cdot \frac{\partial y_k}{\partial b_k} \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial y_1} \\ \vdots \\ \frac{\partial J}{\partial y_k} \end{bmatrix}$$

Back Propagation



$$C_x \xleftarrow{\|\hat{a}_1 - a_1^{(4)}\|^2} a_1^{(4)} \xleftarrow{\sigma(z_1^{(4)})} z_1^{(4)} \longleftarrow w_{11}^{(4)} \cdot a_1^{(3)} + w_{12}^{(4)} \cdot a_2^{(3)} + b_1^{(4)}$$

$$\frac{\partial C_x}{\partial w_{11}^{(4)}} = \frac{\partial C_x}{\partial a_1^{(4)}} \cdot \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} \cdot \frac{\partial z_1^{(4)}}{\partial w_{11}^{(4)}} = 2\,\|\hat{a}_1 - a_1^{(4)}\| . \sigma'(z_1^{(4)}) . a_1^{(3)}$$

$$\sigma'(z) = \sigma(z)\left[1 - \sigma(z)\right]$$

$$\sigma(z) = a$$

$$\frac{\partial C_x}{\partial w_{11}^{(4)}} = \underbrace{2\,(\hat{a}_1 - a_1^{(4)}) . a_1^{(4)}(1 - a_1^{(4)})}_{\delta_1^{(4)}} . a_1^{(3)}$$

$$\Rightarrow \frac{\partial C_x}{\partial w_{11}^{(4)}} = \delta_1^{(4)} . a_1^{(3)} \qquad \frac{\partial C_x}{\partial w_{12}^{(4)}} = \delta_1^{(4)} . a_2^{(3)} \qquad \frac{\partial C_x}{\partial b_1^{(4)}} = \delta_1^{(4)}$$

For any weight or bias in the network,

$$\frac{\partial C_x}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} . a_k^{(l-1)} \qquad \frac{\partial C_x}{\partial b_j} = \delta_j^{(l)}$$

where δ for a layer is derived from the δ of the next layer

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} a_j^{(l)} (1 - a_j^{(l)})$$

k = number of neurons in the output layer

Summary:

1. Initialize network with random weights and biases

2. For each training image:

    a. Compute activations for the entire network

    b. Forward pass: Compute δ for neurons in the output layer using network activation and desired activation:

    $$\delta_j^{(l)} = 2\left(\hat{a}_j^{(l)} - a_j^{(l)}\right) . a_j^{(l)} . (1 - a_j^{(l)})$$

    c. Compute δ for all neurons in the previous layers:

    $$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} a_j^{(l)} (1 - a_j^{(l)})$$

    d. Compute gradient and cost w.r.t to each weight and bias for the training image using δ :

    $$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} . a_k^{(l-1)} \qquad \frac{\partial C}{\partial b_j} = \delta_j^{(l)}$$

3. Average the gradient w.r.t each weight and bias over the entire training set:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{1}{n} \sum \frac{\partial C_x}{\partial w_{jk}^{(l)}} \qquad \frac{\partial C}{\partial b_j} = \frac{1}{n} \sum \frac{\partial C_x}{\partial b_j^{(l)}}$$

4. Update the weights and biases using gradient descent:

$$w_{jk}^{(l)} \longrightarrow w_{jk}^{(l)} = w_{jk}^{(l)} - \eta \frac{\partial C}{\partial w_{jk}^{(l)}}$$

$$b_j^{(l)} \rightarrow b_j^{(l)} = b_j^{(l)} - \eta \frac{\partial C}{\partial b_j^{(l)}}$$

5. Repeat steps 2-4 till cost reduces below an acceptable level

$$\boxed{\frac{\partial J}{\partial b} = \delta \in \mathbb{R}^{k \times 1}}$$

Confusion Matrix:

```python
confusion_matrix = np.zeros((train_y.shape[1],train_y.shape[1]))

# Forward pass with learned weights
h1 = forward(test_x,params,'layer1',sigmoid)
test_probs = forward(h1,params,'output',softmax)

predicted_labels = np.argmax(test_probs, axis=1)
true_labels = np.argmax(test_y, axis=1)

for true, pred in zip(true_labels, predicted_labels):
  confusion_matrix[true, pred] += 1

# visualize confusion matrix
import string
plt.imshow(confusion_matrix,interpolation='nearest', cmap='Blues')
plt.grid()
plt.xticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in range(10)]))
plt.yticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in range(10)]))
plt.xlabel("predicted label")
plt.ylabel("true label")
plt.colorbar()
plt.show()
```

1. Batch Size
Batch size determines how many samples are passed through the network before updating the weights.

Common sizes: 16, 32, 64, 128, or 256. Use powers of 2 (e.g., 32, 64) for memory optimization on GPUs.

- Small Batch Size (< 32):
Pros:
Leads to noisier updates, which can help escape local minima.
Can generalize better due to increased stochasticity.
Cons:
Slower training as updates happen more frequently.
Less efficient use of hardware (GPU/TPU).

- Large Batch Size (> 64):

Pros:
Faster training due to fewer updates.
More stable gradient updates.
Cons:
Requires more memory (may not fit in GPU memory).
Risk of poorer generalization (may converge to sharp minima).

2. Number of Epochs
One epoch means one complete pass through the entire dataset.

Factors to Consider:

- Overfitting Risk:
Too many epochs can cause the model to overfit (memorize training data instead of generalizing).
Monitor validation accuracy/loss to stop training early (use early stopping).

- Dataset Size:
For large datasets, fewer epochs (10-20) may suffice.
For smaller datasets, you might need 50-100 epochs or more.

- Learning Curve Monitoring:
Plot training/validation loss vs. epochs.
Stop when the validation loss stops improving for several epochs.

3. Learning Rate
The step size for adjusting weights during training.

Typical Values:
Common range: 1e-1 to 1e-6.

Depends on the optimizer:
SGD: 0.01 to 0.1
Adam: 1e-3 to 1e-4

Factors to Consider:

- High Learning Rate:
Pros: Faster convergence.
Cons: Can overshoot the optimal solution or fail to converge.

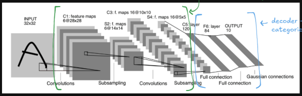
- Low Learning Rate:
Pros: Fine-grained convergence to a better solution.
Cons: Slower training; risk of getting stuck in local minima.

- Learning Rate Scheduling:
Reduce learning rate during training if loss plateaus.
Use schedulers like ReduceLROnPlateau or ExponentialDecay.

LeNet 5

encoder - extracts a representational vector of the image

decoder - maps vector to categorical probabilities



Encoder steps:

1. INPUT: input image size is fixed to 32x32. Image is scaled down if too big, and aspect ratio is changed if necessary.
Can have 3 channels too (rgb).

2. Convolutions C1: Feature map obtained is 28x28. This means the size of the convolution kernel is 5x5, if we don't assume border padding (can't compute convolution for 2 pixels around the border). A small kernel can capture common primitive patterns across a wide range of images.
One convolutional kernel gives one feature map. 6 kernels are used in total to give 6 feature maps.

3. Subsampling S2: Reduces the size of feature maps (average pooling with scale and bias + sigmoid function)

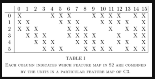
$$h[i, j] = \text{sigmoid}\left[\underbrace{\alpha}_{\text{scale}} . \sum_{u,v} g[u,v]\, f[i-u, j-v] - \underbrace{\beta}_{\text{bias}}\right]$$

Gives 14x14 because of stride (does not look at every pixel but skips a few). Stride/setp size = 2, receptive field = 2x2.
Subsampling acts like a low-pass filter by blurring the image.

4. Convolutions C3: Multi-channel convolution

$$h[i, j] = \sum_k \sum_u \sum_v g[u,v,k]\, f[i-u, j-v, k]$$

This is not a 3D convoltion because the kernel does not slide in the depth dimension.
k was there before too in the input image, but it was k=1.
Kernel size = 5x5x6
5x5 kernel gives 10x10 feature map
Even though it has 6 channels, it does not use all and is given by the table below. This forces the network to learn different features.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

So this gives 16 feature maps.

5. Subsampling S4: Average pooling (scale + bias) + sigmoid. This gives 16 5x5 feature maps.

Decoder steps:

1. Full connection C5: A 5x5x16 convolution filter is used on the 16 5x5 feature maps. All channels of the filter are used. One convolutional filter gives one element. 120 of these are used to give a fully connected layer with 120 elements.

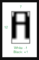
2. Full connection C6: $f(a) = \underbrace{A}_{\text{amplitude}} \cdot \tanh(\underbrace{S}_{\text{Scale}}a)$ for a single perceptron

84 perceptrons

3. Gaussian connections OUTPUT: Uses an Euclidean Radial Basis Function (RBF), which is an unnormalized negative log-likelihood of a gaussian

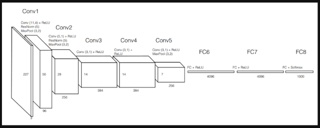
$$\underbrace{y_i}_{\text{single class output}} = \sum_{j}^{84} (\underbrace{x_j}_{\text{FC6}} - \underbrace{w_{ij}}_{\text{parameter set manually } (-1, +1)})^2$$

w =



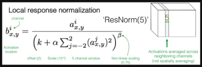
w is set manually to match ASCII character. So LeNet 5 layer F6 is trained to draw ASCII characters

AlexNet



Conv1: Convolution + ReLU + response normalization + max pooling
Convolution with size = 11x11x3
Stride of convolution = 4
96 convolution filters
56x56 is the size of response map after convolution
Shorthand notation = Conv(11, 4)

Regularized Linear Unit (ReLU): f(x) = max(0, x)   Keeps all of the positive activations and throws away all negative.



Local response normalization

'ResNorm(5)'

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(k + \alpha \sum_{j=-2}^{2} (a^j_{x,y})^2\right)^\beta}$$

channel

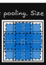
Activation location

offset (2)   Scale (10⁻⁴)   5 channel window   Non-linear scaling (0.75)

Activations averaged across neighboring channels (not spatially averaging)

Response normalization

Overlapped max pooling. Size = 3x3, stride = 2
MaxPool(3,2)



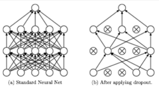
nowadays overlapped pooling is not done

FC8:
FC + softmax is just multi-nomial logistic regression

$$p(y_i|x) = \frac{e^{x^T w_i}}{\sum_j e^{x^T w_j}}$$

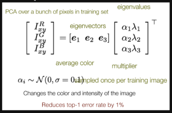
Tricks for training AlexNet:

1. Dropout:
- During training certain nodes are randomly dropped out in the FC
- Forces the network to learn with less parameters
- Reduces overfitting (doubles training time)
- Has the effect of model averaging
- Use all nodes at test time



(a) Standard Neural Net   (b) After applying dropout.

2. Data Augmentation:
- Random translation and horizontal flipping (2048x increase in data). A test time, prediction is averaged over 10 patches (corners, centers and flips). This prevents overfitting.
- Color augmentation



PCA over a bunch of pixels in training set

$$\begin{bmatrix} I^R_{xy} \\ I^G_{xy} \\ I^B_{xy} \end{bmatrix} = [\boldsymbol{e}_1 \;\; \boldsymbol{e}_2 \;\; \boldsymbol{e}_3] \begin{bmatrix} \alpha_1\lambda_1 \\ \alpha_2\lambda_2 \\ \alpha_3\lambda_3 \end{bmatrix}^\top$$

eigenvalues

eigenvectors

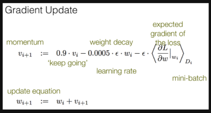
average color

multiplier

$\alpha_i \sim \mathcal{N}(0, \sigma = 0.1)$ sampled once per training image

Changes the color and intensity of the image

Reduces top-1 error rate by 1%



Gradient Update

$$v_{i+1} := 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$$

momentum

'keep going'

weight decay

learning rate

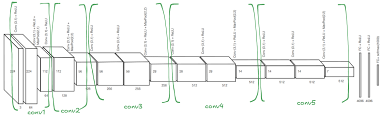
expected gradient of the loss

mini-batch

update equation

$$w_{i+1} := w_i + v_{i+1}$$

VGG-16



Use of stacking convolutions:
- Decision function is more discriminative - can capture more complex boundary decision
- Decreases the number of parameters

- Convolutions all standardized to 3x3
- Convolutions always followed by ReLU
- Stack several convolutions at the same resolution
- Downsample by 2, increase channels by 2
- No local response normalization

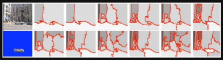
- Initialize bottom layers with smaller network
- Dropout is important
- Data augmentation is important

Batch Normalization:

Segmentation

Superpixels - oversegmentation
- Lets not even try to compute a correct segmentation, because we cant define what is "correct". There is no ground truth.
- Lets be content with an oversegmenatation in which each region is very likely to be uniform

Multiple segmentations
- Generate many segmentations of the same image
- Even though many regions are "wrong", some consensus should emerge



Chicken and egg problem:
- If we knew the regions, we could compute the features and label the right regions
- But to know the right regions, we need to know the labels

Solution
- Generate lots of segmentations
- Combine the classifications to get consensus

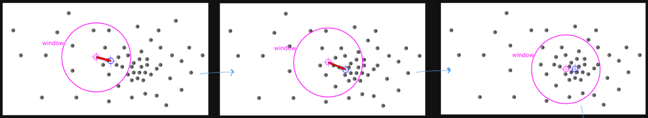
Segmentation as Clustering:
- Given a set of features {xi}
- Given a distance between features d(x,x')
- Find a partition {Cj} such that:
    - d(xi, xj) is small if xi and xj belong to the same group
    - d(xi, xj) is large if xi and xj belong to different groups

Mean-Shift Algorithm

Segmentation as equivalent to finding places with high density in feature space

We want to figure out where there are points of high density. These points are taken as cluster centres.

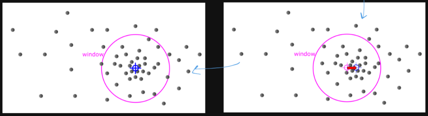
Density = number of points within a region (weighted average)



Mean is shifted based on density

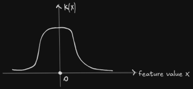
We will eventually reach a point where the density is locally the maximum.
This is gradient ascent.

Gradient is the mean-shift vector



Kernel Density Estimation:

Define a kernel function K(X) with the properties:
- K decays to zero far away from 0
- K is maximum at 0
- K is symmetric

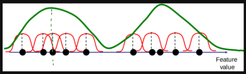


We can define the kernel at each data point and sum up the result into a single function:

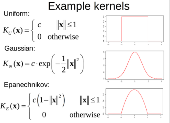
$$f(X) = \frac{1}{N}\sum_{i} K(X - X_i)$$

kernel density estimate (just like probability distribution function)

f(X) approximates the probability that feature X is observed given the data points. Maxima of f correspond to the clusters in the data.



How to choose K(X)? Think of a kernel as measuring how much two data points look alike.
Could be a gaussian function



Example kernels

Uniform:

$$K_U(\mathbf{x}) = \begin{cases} c & \|\mathbf{x}\| \le 1 \\ 0 & \text{otherwise} \end{cases}$$

Gaussian:

$$K_N(\mathbf{x}) = c \cdot \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right)$$

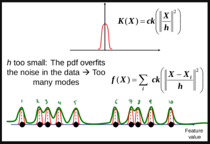
Epanechnikov:

$$K_E(\mathbf{x}) = \begin{cases} c\left(1-\|\mathbf{x}\|^2\right) & \|\mathbf{x}\| \le 1 \\ 0 & \text{otherwise} \end{cases}$$

General algorithm:

- For i = 1, ..., N
  $X \leftarrow X_i$
  - Repeat
    $$X \longleftarrow X_i + \nabla f(X) = X_i + \frac{1}{N}\sum_i \nabla K(X - X_i)$$
  - Until X does not change

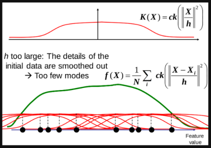
Bandwidth of kernel:
Controls whether kernel is thin tall gaussian or wide short gaussian.
Controls the radius of influence of each data point.
- Too small: Overfits the data points
- Too large: Smoothes out the details of the data
It controls the number of clusters indirectly.



$$K(X) = ck\left(\left\|\frac{X}{h}\right\|^2\right)$$

h too small: The pdf overfits the noise in the data → Too many modes

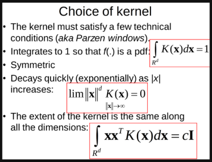
$$f(X) = \sum_i ck\left(\left\|\frac{X - X_i}{h}\right\|^2\right)$$



$$K(X) = ck\left(\left\|\frac{X}{h}\right\|^2\right)$$

h too large: The details of the initial data are smoothed out → Too few modes

$$f(X) = \frac{1}{N}\sum_i ck\left(\left\|\frac{X - X_i}{h}\right\|^2\right)$$



Choice of kernel

- The kernel must satisfy a few technical conditions (*aka Parzen windows*).
- Integrates to 1 so that *f*(.) is a pdf: $\boxed{\int_{R^d} K(\mathbf{x})d\mathbf{x} = 1}$
- Symmetric
- Decays quickly (exponentially) as |*x*| increases: $\boxed{\lim_{\|\mathbf{x}\|\to\infty} \|\mathbf{x}\|^d K(\mathbf{x}) = 0}$
- The extent of the kernel is the same along all the dimensions: $\boxed{\int_{R^d} \mathbf{x}\mathbf{x}^T K(\mathbf{x})d\mathbf{x} = c\mathbf{I}}$

Gradient of Kernel:

$g(t) = -k'(t)$

$$\nabla K(X - X_i) = \nabla\left[ck\left(\frac{\|X - X_i\|^2}{h^2}\right)\right] = \frac{2c}{h^2}(X_i - X)\,g\left[\frac{\|X - X_i\|^2}{h^2}\right]$$

Gradient of entire pdf:

$$\nabla f(X) = \frac{1}{N}\sum_i \nabla K(X - X_i) = \frac{2c}{Nh^2}\sum_i (X_i - X)\,g\left[\frac{\|X - X_i\|^2}{h^2}\right]$$

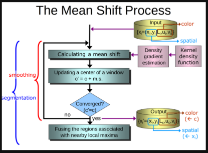
$$\nabla f(X) = \left(\frac{2c}{Nh^2}\sum_i g\left[\frac{\|X - X_i\|^2}{h^2}\right]\right)\underbrace{\left(\frac{\sum_i X_i\, g\left[\frac{\|X - X_i\|^2}{h^2}\right]}{\sum_i g\left[\frac{\|X - X_i\|^2}{h^2}\right]} - X\right)}$$

Mean shift vector
M(X) = difference between X and the mean of the data points weighted by g(.) (points further from X count less)

The mean shift vector points in the same direction as the gradient.
Hence we iteratively move in the direction of the mean shift vector.

Mean-Shift Algorithm:

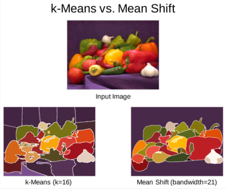
- Initialize: Set X to the value of the point to classify
- Repeat:
  - Move X by the corresponding mean shift vector
    $$X \leftarrow X + M(X) = \frac{\sum_i X_i\, g\left[\frac{\|X - X_i\|^2}{h^2}\right]}{\sum_i g\left[\frac{\|X - X_i\|^2}{h^2}\right]}$$
- Until X converges



The Mean Shift Process

Input
{x_i=(x_i,y_i L_i,u_i,v_i)}
color
spatial
Calculating a mean shift
Density gradient estimation
Kernel density function
Updating a center of a window c' = c + m.s.
smoothing
segmentation
Converged? (c'=c)
no
yes
Output
{x_i'=(x_i,y_i L_c,u_c,v_c)}
color (← c)
spatial (← x_i)
Fusing the regions associated with nearby local maxima

The fundamental operation to compute the kernels is to find the neighbors within some radius (defined by h). This can be very expensive in high dimension with lots of points.



k-Means vs. Mean Shift

Input Image

k-Means (k=16)

Mean Shift (bandwidth=21)

High-Speed Tracking-by-Detection Without Using Image Information

http://elvera.nue.tu-berlin.de/files/1517Bochinski2017.pdf

Tracking-by-detection systems:
1. Object detector is applied to each video frame
2. A tracker is used to associate these detections to tracks

Assumption:
- Detector produces detection per frame for every object to be tracked - this means that there are none or very few gaps in detection.
- Detections of objects in consecutive frames have very high IoU (Intersection over Union), which is usually the case for very high frame rates.

If these two conditions are met, tracking can be done even without using image information

$$IOU(a, b) = \frac{Area(a) \cap Area(b)}{Area(a) \cup Area(b)}$$

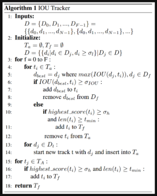
Method:
- Track is continued by associating the detection with highest IOU to the last detection in previous frame if it is above a threshold σ_{IOU}
- All detections not assigned to an existing track will start a new track
- All tracks without assigned detection will end
- Filters out all tracks with a length shorter than t_{min} and the ones without at least one detection with a score above σ_h

Short tracks removed because they usually root in false positives and generally add clutter to the output. Requiring a track to have at least one high scoring detection ensures that the track belongs to a true object of interest

D_f - detections at frame f
d_j - jth detection at that frame

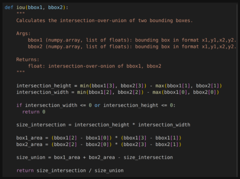
T_a - active tracks
T_f - finished tracks

F - number of frames in the sequence



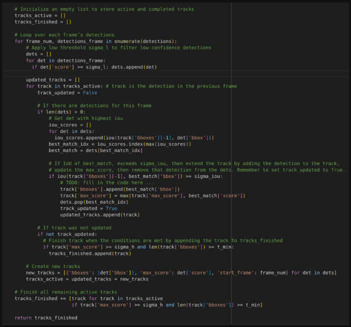
Algorithm 1 IOU Tracker

```
Inputs:
    $D = \{D_0, D_1, ..., D_{F-1}\} =$
    $\{\{d_0, d_1, ..., d_{N-1}\}, \{d_0, d_1, ..., d_{N-1}\}, ...\}$
Initialize:
    $T_a = \emptyset, T_f = \emptyset$
    $D = \{\{d_i | d_i \in D_j, d_i \geq \sigma_l\} | D_j \in D\}$
for f = 0 to F :
    for $t_i \in T_a$ :
        $d_{best} = d_j$ where $max(IOU(d_j, t_i)), d_j \in D_f$
        if $IOU(d_{best}, t_i) \geq \sigma_{IOU}$ :
            add $d_{best}$ to $t_i$
            remove $d_{best}$ from $D_f$
        else
            if $highest\_score(t_i) \geq \sigma_h$
              and $len(t_i) \geq t_{min}$ :
                add $t_i$ to $T_f$
            remove $t_i$ from $T_a$
    for $d_j \in D_t$ :
        start new track t with $d_j$ and insert into $T_a$
for $t_j \in T_A$ :
    if $highest\_score(t_i) \geq \sigma_h$ and $len(t_i) \geq t_{min}$ :
        add $t_i$ to $T_f$
return $T_f$
```



```
def iou(bbox1, bbox2):
    """
    Calculates the intersection-over-union of two bounding boxes.

    Args:
        bbox1 (numpy.array, list of floats): bounding box in format x1,y1,x2,y2.
        bbox2 (numpy.array, list of floats): bounding box in format x1,y1,x2,y2.

    Returns:
        float: intersection-over-onion of bbox1, bbox2
    """

    intersection_height = min(bbox1[3], bbox2[3]) - max(bbox1[1], bbox2[1])
    intersection_width = min(bbox1[2], bbox2[2]) - max(bbox1[0], bbox2[0])

    if intersection_width <= 0 or intersection_height <= 0:
      return 0

    size_intersection = intersection_height * intersection_width

    box1_area = (bbox1[2] - bbox1[0]) * (bbox1[3] - bbox1[1])
    box2_area = (bbox2[2] - bbox2[0]) * (bbox2[3] - bbox2[1])

    size_union = box1_area + box2_area - size_intersection

    return size_intersection / size_union
```

🌐https://www.v7labs.com/blog/intersection-over-union-guide



```
# Initialize an empty list to store active and completed tracks
tracks_active = []
tracks_finished = []

# Loop over each frame's detections
for frame_num, detections_frame in enumerate(detections):
    # Apply low threshold sigma_l to filter low-confidence detections
    dets = []
    for det in detections_frame:
      if det['score'] >= sigma_l: dets.append(det)

    updated_tracks = []
    for track in tracks_active: # track is the detection in the previous frame
        track_updated = False

        # If there are detections for this frame
        if len(dets) > 0:
            # Get det with highest iou
            iou_scores = []
            for det in dets:
              iou_scores.append(iou(track['bboxes'][-1], det['bbox']))
            best_match_idx = iou_scores.index(max(iou_scores))
            best_match = dets[best_match_idx]

            # If IoU of best_match, exceeds sigma_iou, then extend the track by adding the detection to the track,
            # update the max_score, then remove that detection from the dets. Remember to set track_updated to True.
            if iou(track['bboxes'][-1], best_match['bbox']) >= sigma_iou:
                # TODO: fill in the code here ...
                track['bboxes'].append(best_match['bbox'])
                track['max_score'] = max(track['max_score'], best_match['score'])
                dets.pop(best_match_idx)
                track_updated = True
                updated_tracks.append(track)

        # If track was not updated
        if not track_updated:
          # Finish track when the conditions are met by appending the track to tracks_finished
          if track['max_score'] >= sigma_h and len(track['bboxes']) >= t_min:
            tracks_finished.append(track)

    # Create new tracks
    new_tracks = [{'bboxes': [det['bbox']], 'max_score': det['score'], 'start_frame': frame_num} for det in dets]
    tracks_active = updated_tracks + new_tracks

# Finish all remaining active tracks
tracks_finished += [track for track in tracks_active
                    if track['max_score'] >= sigma_h and len(track['bboxes']) >= t_min]

return tracks_finished
```

Spectral Graph Techniques

We convert an image into nodes. Each pixel or feature or superpixel location could be a node.
Edges between nodes tell us the affinity between the nodes (probability that the two nodes stick together).

Affinity: If distance is very small, then affinity is very high

Feature value = position in image p

$$m_{ij} = \exp -\|p_i - p_j\| / \sigma^2$$

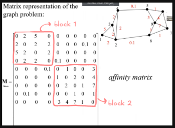
Feature value = intensity I

$$m_{ij} = \exp -\|I_i - I_j\| / \sigma^2$$

Feature value = color c

$$m_{ij} = \exp -dist(c_i, c_j) / \sigma^2$$

Feature value = edge position, orientation, length



Matrix representation of the graph problem:

$$\underline{M} = \begin{bmatrix} 0 & 2 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 2 & 2 & 0 & 0 & 0 & 0.1 & 0 \\ 5 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 0 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 1 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 & 0 & 2 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 1 & 7 \\ 0 & 0.1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 3 & 4 & 7 & 1 & 0 \end{bmatrix}$$

If the nodes were randomly labelled, this affinity matrix can be rearranged such that we can get all the high affinities in a single block.

Graph G = (V,E) consists of:
- V: a set of features $\{x_i\}$ represented as nodes
- E: a set of connections or edges between these features weighted by affinity m_{ij}

$D(C_i, C_j)$: Min edge weight between any node in cluster C and any node in cluster C . This represents the dissimilarity (distance) between the two clusters.

Don't merge if $D(C_i, C_j)$ > threshold