```
import tensorflow as tf
hello_const = tf.constant("hello")
with tf.Session() as sess:
    output = sess.run(hello_const)
    print(output)
```

- TF data isn't stored as integers, floats, or strings. The values are encapsulated in an object called **Tensors**
- hello_const ⟶ 0-dimensional string tensor.
  ⟶ a constant tensor
- A = tf.constant(1234) ⟶ 0-dim int32 tensor
  B = tf.constant([123,456,789]) ⟶ 1D int32 tensor
  C = tf.constant([[123,456,789]
  [2,8,4]]) ⟶ 2D " "

Computational Graph

- tf.Session() an environment for running computational graphs
- tf.placeholder(), feed_dict

```
x = tf.placeholder(tf.string)
with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: "hello"})
```

- tf.add(), tf.subtract, tf.multiply()
- tf.cast() ⟶ no implicit casting!

- Classification ⟶ core building block task in many ML tasks
  - reinforcement Learning
  - detection
  - ranking
  - regression

- Logistic Classifier (Linear Classifier)

$$Wx + b = \hat{y}$$

- Softmax function: $S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$

  y_i, y_j : scores

  Softmax function takes the scores (a.k.a. logits) and turns them to probabilities.

- Tensorflow uses $xW + b$ instead of $Wx + b$
- tf.Variable(), x = tf.Variable(5)
- tf.global_variables_initializer()
- tf.truncated_normal() ⟶ generate random numbers from Normal dist
  (n_features, n_labels)
- tf.zeros()
- tf.matmul()
- tf.nn.softmax( )

Cross Entropy

- Using one-hot encoding of input we can measure how well we are doing by comparing the two vectors (one-hot label , and the output of softmax (probabilities))

S(y) ← probabilities

0.7
0.2
0.1

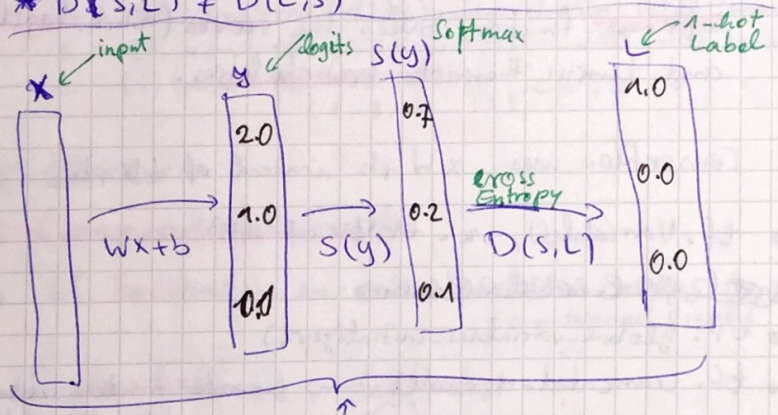
$$D(S,L) = -\sum_i L_i \log(S_i)$$

distance

L ← label (one-hot)

1.0
0.0
0.0

* $D(S,L) \neq D(L,S)$



- Multinomial Logistic Classification

$$D(S(WX+b), L)$$

- ~~Training~~ classification goal (D↓ for correct classification and D↑ for wrong classification)

→

Loss = Average cross-entropy distance

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(WX_i+b), L_i)$$

- Two Questions
  - How do we feed image pixels to the classifier
  - Where do we initialize the optimization.

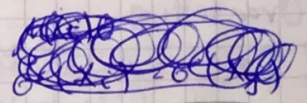
Numerical Stability

- be worried when working with very large and very small numbers
- specially when adding them together.

```
a = 1 000 000 000
for i in rang(1 000 000):
    a = a + 1e-6
print(a - 1000000000)
```

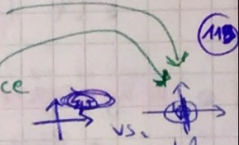
prints 0.95367431640625 instead of 1.0

- we want the values involved in the calculation of this big Loss function to never get too big or too small
  - Guiding principle:



$\mu(X_i) = 0$ zero mean

$\sigma(X_i) = \sigma(X_j)$ equal variance



- Badly conditioned problem vs. well conditioned problem

- For images subtract 128 from every channel (Pixels - 128)
  - doesn't change the content of the image
  - But makes it much more suitable for numerical calculation. (optimization).

- Weight Initialization
  - draw weights randomly from $\mathcal{N}(0, \sigma)$
  - σ large => output of the initial point of optimization high order of magnitude
    => prob. dist. by Softmax too peaky (opinionated)
  - σ small => output small
    => prob. dist. of Softmax uncertain

=> Choose a small: σ

- Training, Validation, Test set
  - measuring performance
- Once you know how to measure performance on a problem, you have already solved half of it

- Overfitting
  - when the classifier memorizes the training set
  - it fails to generalize.

- Why separate validation and test set? !
  - Because every time you tune model parameters as a result of validation set, you actually implicitly leaking information about your validation set to the classifier !

- Rule of 30:
  a change that affects 30 examples in your validation set, is usually statistically significant and can be trusted.

  z.B.
  Validation set size 3000 or 80.5%
  - improvement from 80% to 80.1% are noise, because they are affecting 3 or 15 validation examples
  - improvement from 80% to 81% is significant because it affects 30 validation examples.

- that's why most people hold back a validation set of >30,000 examples
  => changes > 0.1% in accuracy are significant and can be trusted
- If don't have enough data ~> do cross-validation
- But cross-validation can be a slow process.
  => get more data is better.

- Training Logistic regression using gradient descent is greate
  - we directly optimizing the error measure (great idea)
  - That's why in practice a lot of ML research is about designing the right loss function to optimize

- But the biggest problem is scaling →

Problem of scaling GD

rule of thumb !

- if computing the loss $\sum_i D_i$ takes n flops, then computing the gradient of L takes about 3x more flops.

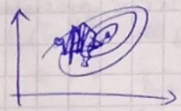
- The L is huge, it depends on every single data point in data set
- But we want to be able to train on lots of data.
- also GD is iterative => we have to go through the entire DS many times 10s or 100 of times.

=> Let's cheat:

- instead of computing the loss we compute an estimate of it (a terrible estimate in fact :))

Stochastic GD (SGD)

- compute the loss for a very small random subset of data. Between 1 to 1000 samples each time
- it is very important that it is random !
- iterate many many times more.



- Vastly more effective than single GD
  - scales well with both data and model

Momentum

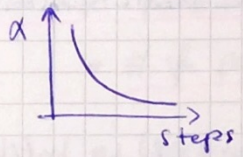
Helping SGD: { mean = 0, equal variance } inputs; { random, equal variance (small) } initial weights

- Momentum:
  - take advantage of the accumulated knowledge from previous step about direction to go
  - e.g. keep a running avg. of the gradient

$$M \leftarrow 0.9\,M + \Delta L$$

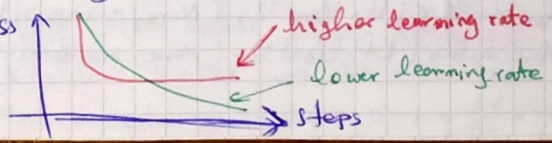
Learning rate decay:

- SGD: take smaller noisier steps towards goal
- How small? It is a whole area of research itself
- One thing however is always beneficial: make that step smaller and smaller as you train !
- e.g. an exponential decay, or reduce it every time loss reaches a platen



Lower it over time!

- Learning faster is not better than learning slower



- SGD ≈ Black Magic (with so many hyper param to tune with)

- If (love to) remember just one thing:

  When things don't work, Keep calm and Lower your Learning rate

---

- ADAGRAD: a modification of SGD that implicitly does momentum and LR decay

  => Less sensitive to hyperparameters.

- Mini-Batching
  - train on a subset of data instead of all
  - train model even if the memory is not enough to contain the entire DS.
  - quite useful with SGD:
    - shuffle the data at the start of each epoch, then create mini batches

- Setting batches in TF
  - the size of data might not be dividable by batch size

  => use tf.placeholder

```
features = tf.placeholder(tf.float32, [None, n_inputs])
labels = tf.placeholder(tf.float32, [None, n_classes])
```

- Epoch: A single forward and backward pass of the whole DS.

TF Lab Notebook:

- Min-Max Scaling $X' = a + \frac{(X - X_{min})(b-a)}{X_{max} - X_{min}}$

  bring the data to a range of $[a, b]$

11.09.
21:30
2h

Lesson 12: Deep Neural Networks

- Linear Model Complexity

  N inputs, K outputs } $(N+1)K$ parameters

- Linear model limitations
  - # of params: too few, we want many many more
  - Linearity:

    $Y = X_1 + X_2$ ✓

    $Y = X_1 * X_2$ ✗

- But Linear models have advantages too:
  - Stable: $Y = WX \rightarrow \Delta Y \sim |W| \Delta X$ (bounded)
  - Derivatives are constant: $Y = WX \rightarrow \frac{dY}{dX} = W^T$, $\frac{dY}{dW} = X^T$

- ⇒ We want to keep our parameters inside big linear functions,

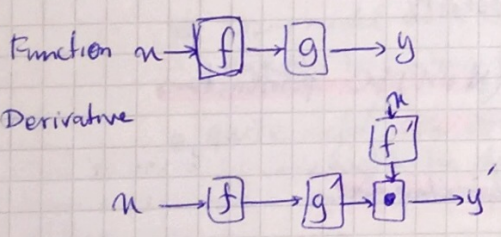
$$Y = W_1 W_2 W_3 X = W X$$

But we would also like our entire model to be non-linear.

⇒ add non-linearities

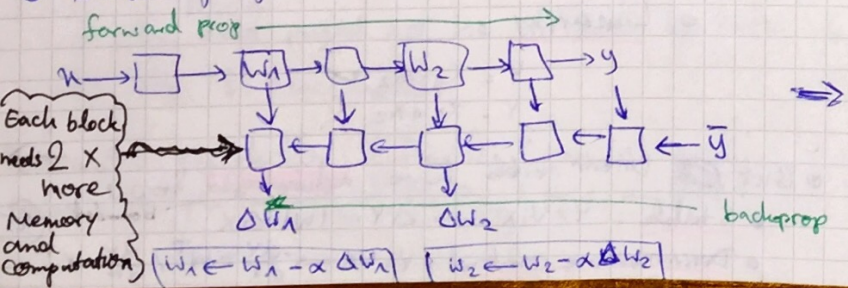
- ReLU (Rectified Linear Unit)
  - Lazy engineer's favorite non-linear function :)
  - $f(x) = \max(0, x)$
  - tf.nn.relu()

- Chain Rule Computational Graph



- Efficient data pipeline
- Lots of data reuse
- Simple data pipeline

- Back-prop



⇒

- tf.train.Saver

- Regularization
  - Why DL now?
    - because large enough DS's available
    - because we know better ways to regularize data.
  - General issue with Numerical Optimization:
    - the skinny jeans problem
    - Traing a network that is just the right size for your network is very very hard to train.
    - ⇒ we most often train Networks that are way too big for our data, and then try our best to prevent them from overfitting

- Early Termination
  - Check performance of validation set and stop traing as soon as performance degrades.

- Regularization

Prevent Overfitting

- Regularization:
  - applying artificial constraints on the network that implicitly reduce the number of free params while not making it more difficult to optimize
  - stretch pants in skinny jeans analogy :)

- L2 Regularization:
  - add another term to Loss functions which penalizes large weights.

$$L' = L + \beta \frac{1}{2} \|w\|_2^2$$

  - adding l_2 norm of weights to loss, multiplied by a small constant.

+ it is very simple and does not need to change the network
+ it derivative is also very simple: w

- Dropout
  - developed recently, works amazingly well
  - looks insane :), but works
  - randomly set half of the activations to zero

=> randomly destroy half of the information that flows through the network. Sounds crazy

=> Network can never rely on any given activation to be present => it is forced to learn a redundant representation for everything!

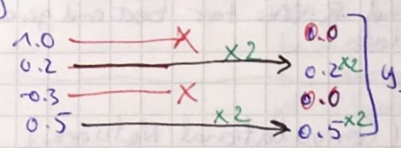
- Forcing the network to learn a redundant representation of every thing
  - makes things more robust
  - prevents over fitting
  - it is as if the resulting network is taking a consensus over an ensemble of networks.

Dropout

- One of the most important techniques to emerge over the last few years.
  - If it doesn't work for you, you should probably be using a bigger network!

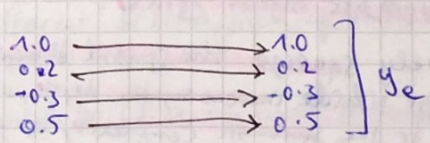
- Evaluating a network trained with dropout:
  - we obviously don't want the randomness when evaluating; we want something deterministic
  - we will evaluate the consensus over these redundant models
  - we average the activations

Training



$$y_e = E(y_t)$$

Evaluation



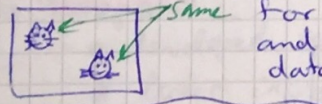
=> During training, scale the remaining activations by factor x2

- tf.nn.dropout()
  - ↳ TF multiplies the units that are kept by $\frac{1}{\text{keep\_prop}}$
  - → a good starting value for keep_prob 0.5
  - → during testing: keep_prob = 1.0

12.04.
20:15
2 h

Lesson 13: Convolutional Neural Networks

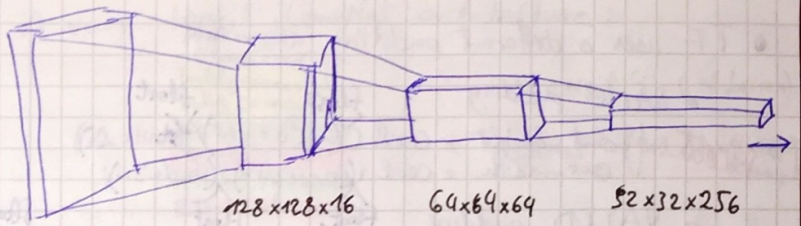
- If your data has some structure, and your network doesn't have to learn that structure from scratch, it performs better.
- Statistical Invariance
  - e.g. translation invariance in image
  - things that does not change over time and space
- Weight Sharing
  - When you know that two inputs can contain the same kind of information, then you share their weights, and train the weights jointly for those inputs



- Weight sharing leads to CNNs for images and to embedings and RNNs for text and general sequential data

- CONVNETS (Convolutional Networks)
  - share weights across space
  - progressively squeeze the spatial dimention of the image while increasing the depth, which corresponds roughly to the semantic complexity of your representation

→



→ Classifier

- We have now a representation where all the spatial information have bee squeezed out and only parameters that map to content of the image (like Cat) remain.
- Patch = Kernel = filter
- Feature map = Channels
- Stride: how much to shift the filter
  - Stride of two ⇒ input size is halfed
- Padding
  - ↳ Valid padding: don't go pass the edge
  - ↳ Zero padding: go off the edge and pad with zeros (same padding) because size of output will be the same as input in case of stride = 1
- filter depth:
  - different filters pick up different quality of a patch.
- Dimentionality (size of model parameters) ?
  - given: input W×H, filter size F, stride S, Padding P, #filters = K
  - W_out = $\lfloor (W-F+2P)/S \rfloor + 1$ | H_out = $\lfloor (H-F+2P)/S \rfloor + 1$
  - D_out = K, output volume = W_out · H_out · D_out

$\lfloor \cdot \rfloor$: floor
⇒ integer

- TF uses a different padding algorithm
  - SAME padding

Float | out_height = ceil(float(in_height) / float(stride_h))
out_width = ceil(float(in_width) / float(stride_w))

  - VALID padding

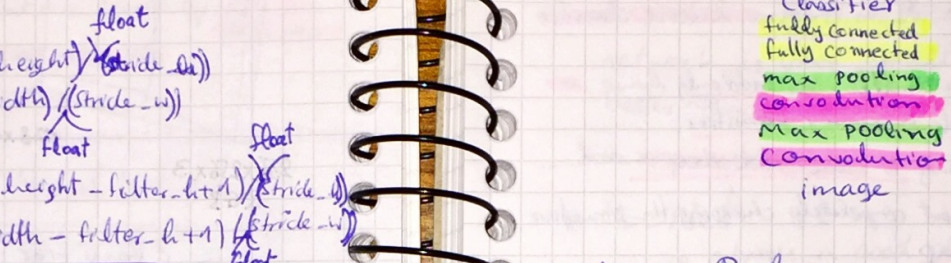
Float | out_height = ceil(float(in_height - filter_h + 1) / float(stride_h))
out_width = ceil(float(in_width - filter_h + 1) / float(stride_w))

- Deep Visualization toolbox

- TF CNNs
  - tf.nn.conv2d()
  - tf.nn.bias_add()

- Pooling
  - Stride of 2 is a very aggressive way to down sample an image. It removes a lot of information.
  - What if instead of skipping one in every two convolutions, we still run with a very small stride, say e.g. 1, but then take all the convolutions in the neighborhood and combine them somehow.
  - Max Pooling
    - ⊕ does not add to the number of parameters ⇒ don't increase the risk of overfitting.
    - ⊕ often yields more accurate models.
    - ⊖ model becomes more expensive to compute
    - ⊖ more hyperparam: pooling-size, pooling-stride

- A very typical ConvNet architecture:



LeNet '5 ('98 Lecun)

AlexNet ('12 Alex Krizhevsky)

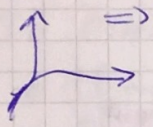
- Average Pooling
  - take avg instead of max
  - like a blurred, low resolution view of the feature map below
- tf.nn.max_pool()
- pooling decreases the size of the output ⇒ prevents overfitting by reducing the number of parameters in future layers.
- Pooling layers have fallen out of favor recently:
  - recent DSs are so big and complex, that we are more concerned with underfitting than overfitting
  - Dropout is a much better regularizer
  - Pooling results in loss of information. e.g. in max pooling on n numbers, we discard information contained in n-1 of them.

- 1x1 convolution
  - why would we ever want a 1x1 convolution? they are not really looking for a patch in image but just a pixel.

- A normal convolution is basically a small classifier for a patch of the image, but it is only a linear classifier
- by adding a 1x1 conv in the middle, we suddenly have a neural network instead of a linear classifier
- it is a very inexpensive way to make models deeper and have more parameters without completely changing the structure
- they are also very cheap.
  it is actually only a matrix multiply with relatively few parameters

---

- Avg pooling, 1x1 convs

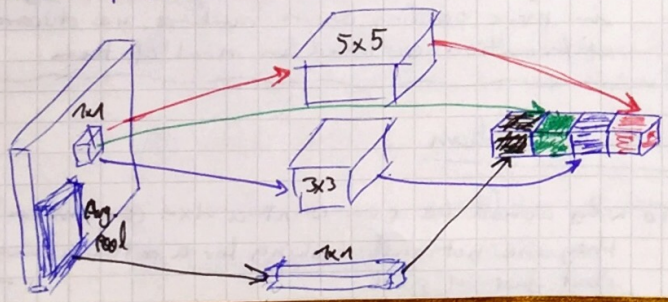


=> a very successfull strategy for building ConvNets that are better and smaller than convnets that simply use a pyramid of convolutions.

Inception Modules

- instead of making a choice at each layer between pooling, convolution, 1x1, 3x3, 5x5, let's have a combination of all of them
- a composition of:
  - Avg Pooling followed by 1x1 conv
  - a 1x1 conv
  - a 1x1 followed by a 3x3
  - a 1x1 followed by a 5x5

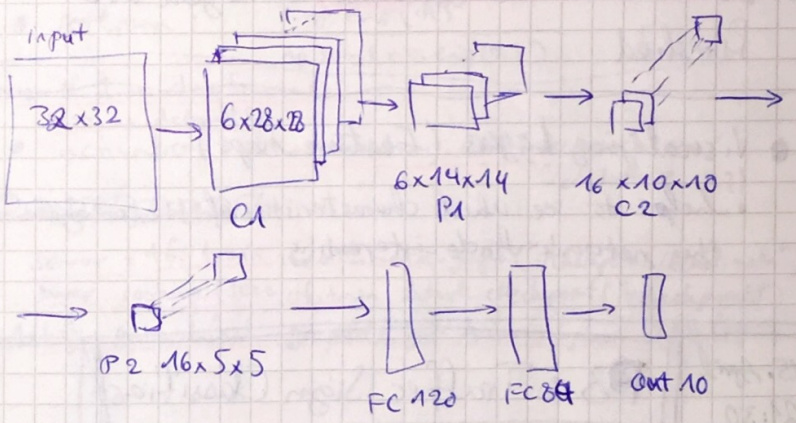
and concatenate the output of each of them



→

- It looks complicated, but we can choose those parameters in such a way that the total number of params in network is very small, but the performance is much better than having individual convolutions only.

- LeNet
  - always a good starting point
- tf.contrib.layers.flatten
- How many parameters LeNet?



- Number of params

$[6 \times (5 \times 5 \times 1) + 6] + [16 \times (5 \times 5 \times 6) + 16] + [(400 \times 120) + 120]$

$+ [(120 \times 84) + 84] + [(84 \times 10) + 10] = 61706$

- Tensor Board for visualizing networks

15. April
17–19
(2h, actually only 30 min)

Lesson 14: LeNet for Traffic Signs

- Always shuffle the data, otherwise the ordering in the data can have a major impact on the training
- training pipeline / evaluation pipeline
- sklearn.train_test_split()
  - 20% is a good rule of thumb for validation set
- Don't run the test set until you are finished

- Visualizing Layers (Feature maps)
  - helps to see which characteristics of the image the network finds interesting.

15. April 21:30
26. April

P3: Traffic Sign Classifier

- data set distribution
- normalization (pixel - 128) / 128 (zero mean)
- shuffle
- Learning rate: 0.1 ✗ 0.01 ✗ 0.001 ✓
- increasing batch size.
- Epochs, perecesion oscilation and stagnation on validation set
  - ↳ early stopping.
  - Dropout
- restore model
  - ↳ don't init variables again
- # construction code (network, layers, variables)

```
saver = tf.train.Saver()
with tf.Session() as sess:
    saver.restor("./checkpoint/...")
    sess.run(logits, feed_dict={...?})
```

- softmax: $\frac{np.exp(x)}{np.sum(np.exp(x))}$

```
sess = tf.InteractiveSession()
sess.as_default()
```

- activation = tf_activation.eval(session=sess, feed_dict={...?})

~~tf.get_default~~

```
saver = tf.train.import_meta_graph("./checkpoints/lenet.meta")
saver.restore(sess, tf.train.latest_checkpoint("./checkpoints/")
conv1 = tf.get_default_graph().get_tensor_by_name("C1:0")
```

26. April 18:30
15
26. April 20:30
17
12

Lesson 16: Keras

2h

Otto, Voyage

Andrew Gray (Uber ATG)

- Behavioral Cloning

End-to-End Learning:

- Network learns the steering wheel angle and speed using only the input from sensors.

- Why DL over Robotic approach?
  - Both approaches are used.
  - DL has the potential to fundamentally change the way SDCs are programmed.
  - Robotic approach involves programming a lot of detailed knowledge about sensors and rules and control into the car.
  - With DL on the other hand, we only feed all the data we have and let the network decide what is important and what is not.
    + with DL we have a feedback loop: The more we drive, the more data we collect, and the better becomes the network.

- Keras:
  - a front-end to TF that makes creating networks much faster and cleaner.
  - sometimes if you need very fine-grained control you need to create the layers directly using TF.

- keras.model.Sequential()
  - a wrapper for NN model
  - fit(), evaluate(), compile()

- two ways to build Keras models:
  - sequential
  - functional

- Sequential(), Flatten(), Activation(), Dense(), Conv2D(), MaxPooling2D(), Dropout(!)
  ↑ dropout prob not keep_prob
- sklearn.preprocessing.LabelBinarizer()
- model.fit() ← X-normalied, y-one-hot, epochs, valid_split
- model.compile("adam", "categorical_cross_entropy", ["accuracy"])
- Lambda layers for preprocessing → Cropping2D()

26. April 20:30
27. April 23:00

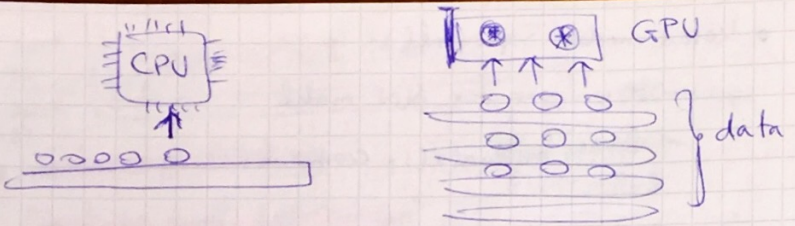
L17: Transfer Learning

1h 30m

(VP. DL) Bryan Catanzaro (NVIDIA): creator of cuDNN

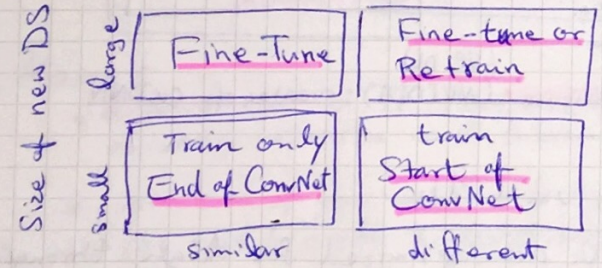
- GPUs vs. CPUs
  - GPUs are optimized for high throughput computation

→ 139

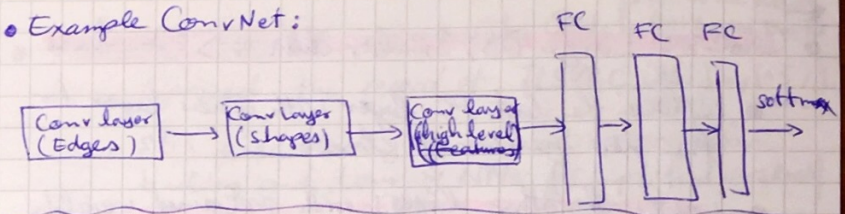


- CPUs are optimized for latency: running a single thread of instructions as quickly as possible
- GPUs are optimized for throughput: running as many simultaneous operations as possible.
- Rule of thumb
  - Training is 5x faster on a GPU.
- Check graph on "CPU vs. GPU GigaFLOPs"

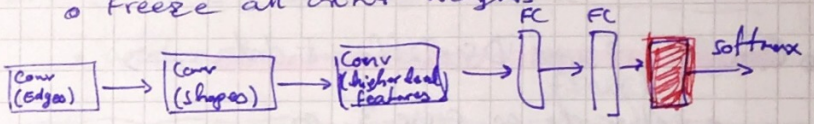
- Transfer Learning
  - Two dimensions for decision:
    - size of the new dataset
    - similarity of the new DS to the original DS



- Example ConvNet:

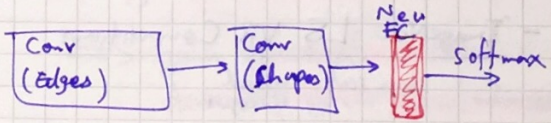


- Case 1: Small DS, Similar Data:
  - Cut-off the last FC layer and retrain it for your problem (with new number of classes)
  - Freeze all other weights

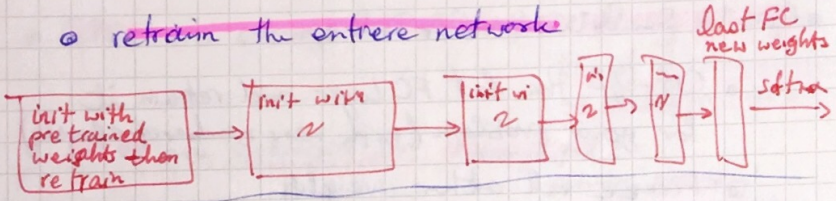


  - both DSs have similar higher level features
    ⇒ keep all the feature detector layers.

- Case 2: Small DS, Different data
  - Keep only few of the Conv layers at the beginning (Feature detectors for general features, like edges and shapes)
    - the new DS has different higher level features than the original DS
      ⇒ The conv layer for higher level features is not useful.
  - Add a new FC layer for the new number of classes and train its weights.



- Case 3: Large DS, Similar data ⇒ Fine-tune
  - replace the last FC layer with a new layer for the new number of classes.
  - init all other layers with pretrained weights
  - retrain the entrere network



- Case 4: Large DS, Different data ⇒
  - either do as case 3, or
  - or retrain entire net with random weights.

- Rise of DL
  → huge labeled DSs because of Internet
  → greater computational powers.

- Keras Applications: import a pre-trained network:
  Xception, VGG16, VGG19, ResNet, ResNetV2, InceptionV3, InceptionResnetV2, MobileNet, MobileNetV2, DenseNet, NASNet

- ImageNet: A huge database of hand-labeled Images.
  - ImageNet LS VR Competition
    (large scale, visual Recognition)

- AlexNet (2012)
  - parallel GPUs
  - first to use ReLU
  - first to use Dropout
  - improved the record of ILSVRC (26% error) by (15% error)
  - still used today as a starting point for building and training NNs. (Simplified versions of it)

- VGG
  - visual geometry group @ oxford / ILSVRC error 7%
  - simple and elegant arch.: great for transfer learning
    16/19
    ↳ a long sequence of 3×3 convolutions broken up by 2×2 pooling layers a trio of FC layers.
  - used a lot as starting point; works really well. flexible

Advice, Wisdom

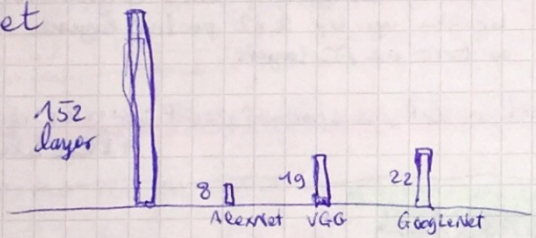
- DL is a hands on emporical field
  - → more experiments → more ideas
  - practice is most of the times ahead of theory
  - ⇒ don't get too attached to theory and network architectures

To be a successful DL Practitioner you need to have a bias towards Action
⇒ Experiment, try

- DL is actually one of the few places in CS that we actually do sience → experiment
- Get your hands dirty; Try to solve real problems
  ⇒ gain practical skills
  develop your confidence

- GoogLeNet / Inception
  - very fast
    almost as fast as AlexNet
  - Inception modules
    => total number of parameters is small.
  - great accuracy
  - great choice if you want to run your network in real-time

- ResNet



- similar to VGG
- same structure, repeated again and again
- short cut connections (skip layers)
- less on ImageNet 3% !
  => better than Human

- Freezing layers
  ⊕ memory usage, training speed
- layer.trainable = False / model.layers
  inception.summary()
  model.pop() / include_top = False / 2 top layers
- Sequential vs. Model API
- CIFAR-10 Dataset

- GlobalAveragePooling2D()
- callbacks
  - ModelCheckpoint()
  - EarlyStopping()
- ImageDataGenerator()
- model.fit_generator() / model.fit()
  model.predict_generator() large DS
  model.predict()
  - whole DS at once.
  - use if whole DS fits into memory

---

28. April
05. May
(7 days)

P4: Behavioral Cloning

- Lambda layers for preprocessing.

```
model.add(Lambda(lambda x: (x/255.0) - 0.5,
                 input_shape=(160,320,3)))
```

- Cropping2D()

```
model.add(Cropping2D(cropping=((75,25),(0,0))))
```

- Simulator: 2 modi: training mode / autonomous mode
- press R to record

- drive on the center
- drive recovery if car drives off the center
- drive counter clock wise to generalize model
- flipping images to augment data
- collect the data in the second track to help the model to generalize
- avoiding over fitting and under fitting
- knowing when to stop collecting data

- Data [-a,1] [0,1] [0,1] [0,50]

center image, left, right , steering , throttle, brake, speed

features labels

- 3 image RGB, 160 x 320
  - may need to crop top and bottom of the image (sky, trees, hills, car's hood). distract the model
- left and right images help the network learn how to recover when off the center
- regression model
  - loss "mse"
- Save model.h5

```
input csv
lines = []
with open("../data/driving_log.csv") as csv_file:
    reader = csv.reader(csv_file)
    for line in reader:
        ...

images = []
measurements = []

X_train = np.array(measurements)

model = Sequential()
model.add(Flatten(input_shape=(160,320,3)))
model.add(Dense(1))

model.compile(loss="mse", optimizer="adam")
model.fit(X_train, y_train, validation_split=0.2,
          shuffle=True)

model.save("model.h5")
```

- drive.py model.h5 / underfitting / overfitting
- improve model:
  - increase epochs
  - add more convolutions
  - use drop out and pooling
  - collect more data
  - fewer conv and FC

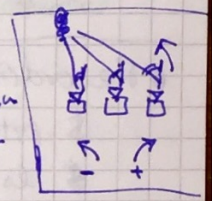
- Lamda layer : Preprocessing

```
Lambda(lambda x:
       (x/255.0)-0.5)
```

  - normalization : Pixel/255
  - mean center : Pixel_normalized - 0.5
- LeNet
- Track and hense the training data is biased towards steering to the left
  - Data augmentation : flip the images (np.fliplr(image))
    - steering = -steering

  => Balanced data set

- use left and right images for learning how to recover from off-center



```
correction = 0.2
steer_left = steer_center + correction
steer_right = steer_center - correction

images.extend(left, right)
steerings.extend(steer_left, steer_right)
```

- Cropping images
  - Cropping2D() layer
- NVIDIA network
- Collecting more data
  - driving counter clockwise } => balanced data => better
  - driving other track } => model generalizes
- How much data?
  - loss -> MSE
  - loss high on training and validation? => underfitting
  - loss low on training but high on validation? => overfitting
  - two or three laps center drivings
  - one lap recovery driving from center, one lap driving smoothly on curves

- Outputting training and validation Loss Metrics.
  - verbose option for fit() and fit_generator()
    - 1 / 2 → loss on training and valid set after each epoch
  - model history object : contains the traing & validation loss for each epoch

---

- Generators : Coroutine
  - largedata : don't store the whole DS at once in memory ;
    - pull chunks of data
  - yield keyword in python

- Recording video

```
$ python drive.py model.h5 run1
$ ls run1
$ python video.py run1 --fps 48
```

05. May
23-00:30

Lesson 20: Sensors

0:30 m

- Michael Maile : Director of Sensor fusion team at Mercedes Bez.
  - Andrei Vatavu
  - Dominik Nuss → Luminar (lidar)
    - ↳ Ulm uni.
- Kalman Filter : Extended Kalman Filter
  - track a pedestrian relative to the car with help of radar an Lidar data
- Sebastian Thrun : AI for Robotics course
- C++ to fuse Radar & lidar data
- Unscented Kalman Filter:
  - non-linear tracking

- Prototype SDC of Mercedes Benz (Choobaka :))
  - 15 sensors :
    - a stereo camera rig (2 cameras)
    - a long range lens camera for detection of traffic lights (1 camera)
    - radars (behind the front bumper)
    - lidar

- Radar uses Doppler effect to measure speed directly. → Important for Sen. Fus. →

← Doppler effect very important for Sens. Fus. because it gives us the velocity as an independent measured parameter

⟹ Fusion Algorithms converge much faster

- Radar also used for localization by generating the radar map of the environment.
  - Because lidar signals waves bounce off hard surfaces they can provide measurements to objects without direct line of sight
    e.g. see underneath other vehicles on spot objects and buildings that would be obscured otherwise.

- Radar:
  - least effective by rain and fog
  - wide FoV about 150°
  - a long range 200+ meters

	Cam.	Lidar	Radar
Resolution	●	◐	○ ← low res. spec. vertical
Noise	◉	○	○
Velocity	○	○	●
All-weather	○	○	◉
size	◉	○	◉

- Low Res. of Radar ⟹ reflection from static objects can cause problems. [Radar clutter]

RADAR: Radio Detection and Ranging
LIDAR: Laser Detection and Ranging [Velodyne] for MB. SDC

- Lidar:
  - Infrared laser beam (900 nm wave length)
    ↳ some use larger WLs ⟹ better in in rain and fog
  - much higher spec. res. than Radar
    ↳ • larger number of scan layers in ver. direct
    • high density of points per layer.
  - Cannot measure velocity directly
    ↳ have to rely on position change between two or more scans.
  - more affected by weather and dirt on sensor
    ⟹ requires keeping them clean.
  - bulkier

06. May, 21:00
08. May, 21:00

Lesson 21: Kalmen Filters

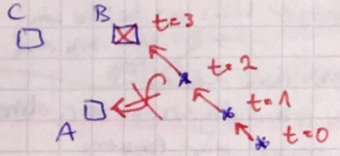
- Stanford automotive research center.
- Junior: SDC of Stanford. child of Stanley, first SDC that is now in National Museum of computing history.

- Lidar and sensor data to track other cars to avoid colliding with them
  - not only know where the other cars are (localization)
  - but also how fast they are moving!

- Avoid collision for future : tracking
- Kalman filters
  - Estimate continous state of a system

Kalman Filter	Monte Carlo
Continous	Discrete
Unimodal	Multimodal

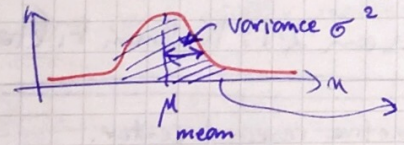
Particle Filters
Continous
Multimodal



- Assuming no drastic change of velocity (A), Kalman filters

Noise → takes observations
→ estimates future locations and velocities

- In Kalman Filters the dist. is given by a Gaussian



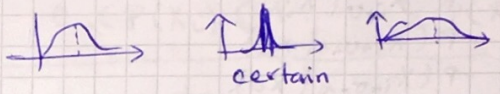
1-D Gaussian (μ, σ^2)

- Task in Kalman Filters:
  to maintain a μ and σ^2 that are best estimate of the location of the object

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}$$

- Unimodal → one peak

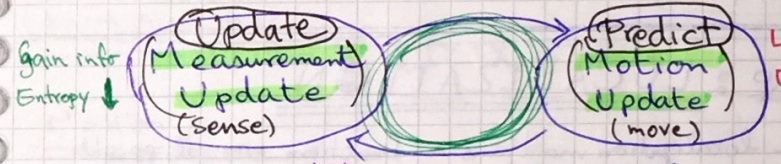
σ^2: certainty. The larger the covariance, the more uncertain we are.



- Gaussian as Belief in Kalman filters.
- $x = \mu$ ⇒ Gaussian value is max: $\frac{1}{\sqrt{2\pi\sigma^2}}$

Kalman Filters: [recap Localization monte carlo (histogram) filters]

- represent the distribution in Gaussians.
- iterate on two main cycles:



- requires product
- uses Bayes Rule
- involves convolution
- uses total probability

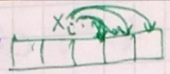
```
world = ['green', 'red', 'red', 'green', 'green']
Z = 'red'  # measurement
pHit = 0.6
pMiss = 0.2   # sensor may not read correctly (sensor prior)

def sense(p, Z):          <- measurement
    q = []                <- prior
    for i in range(len(p)):
        hit = (Z == world[i])
        q.append(p[i] * (hit * pHit + (1-hit) * pMiss))
    return q              <- new belief (posterior)
```

normalization:
```
s = sum(q)
for i in range(len(q)):
    q[i] = q[i] / s
```
(not normalized: the probabilities do not add to one. Needs normalization!)

Measurement Update:
- take measurement and prior
- update the belief
- output posterior

Motion Update (move() function):



```
pExact = 0.8      # P(X_{i+U} | X_i)
pOvershoot = 0.1  # P(X_{i+U+1} | X_i)
pUndershoot = 01  # P(X_{i+U-1} | X_i)

def move(p, U):          # p: prior belief of where the robot is; U: number of steps
    q = []
    for i in range(len(p)):
        q_exact = p[(i-U) % len(p)] * pExact
        q_overshoot = p[(i-U-1) % len(p)] * pOvershoot
        q_undershoot = p[(i-U+1) % len(p)] * pUndershoot
        q.append(q_exact + q_overshoot + q_undershoot)   # Sum
    return q                                             # posterior
```

convolution

Recap : LOCALIZATION

- The localization maintains a function over all possible places a robot might be, where each cell has an associated probability value

BELIEF = PROBABILITY

SENSE = PRODUCT ← measurement update: take those probabilities and multiply them up or down based on the exact measurement.
followed by normalization

MOVE = CONVOLUTION (= ADDITION)

For each possible location after the motion, we reversed engineered the situation, and guessed where the robot might have come from, and then collected (added) the corresponding probabilities

Formal Definition of Localization

$0 \leq P(X_i) \leq 1$ $\quad \sum P(X_i) = 1$ for $X_i \in$ World (a grid cell)

Measurements

Bayes Rule $\quad X$: grid cell $\quad Z$: measurment

- the measurment uptade seeks to calculate a belief over my location after seeing the measurment.

$$P(X_i|Z) = \frac{P(Z|X_i) \cdot P(X_i)}{P(Z)}$$

likelihood; prior multiplied by $P(Z|X_i)$: the chances of seeing a red or green cell for every possible location

probability of seeing a measurment devoid of any location information

our normalization term gives this

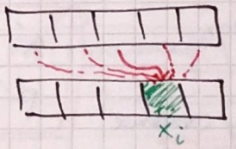
$$P(Z) = \sum_i P(Z|X_i) \cdot P(X_i)$$

Motion

Total Probability

$$P(X_i^t) = \sum_j P(X_j^{t-1}) \cdot P(X_i|X_j)$$

time; grid cell

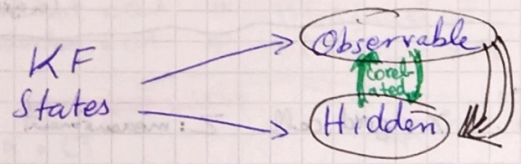


$$P(A) = \sum_B P(A|B) \cdot P(B)$$

Total probability
Convolution (weighted sum)

Check the Kalman Filter notes from Intro to SDC. course.

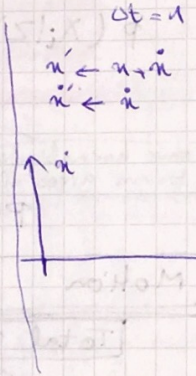
- Kalmon Filters allow infering a hidden variable, like velocity, from measurements, like (x,y) position



allows inferences as a result of this corelation

- For design of a KF we need two things:
  - State transition function

$$F: \begin{pmatrix} x' \\ \dot{x}' \end{pmatrix} \leftarrow \underbrace{\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}}_{F} \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$



  - A measurement function

$$H: z \leftarrow \underbrace{\begin{pmatrix} 1 & 0 \end{pmatrix}}_{H} \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

x: best estimate
F: State transition matrix
u: motion vector
P: process uncertainty covariance (error cov. matrix)
z: measurement
H: measurment function
R: measurmet noise
K: Kalmon Grain

Predict

$$x' = Fx + u$$
$$P' = F \cdot P \cdot F^T$$

Measurement Update

(error) We compare the measur. with our prediction

$$y = z - H \cdot x$$

projecting sys. uncertainty into the measurment space

$$S = H \cdot P \cdot H^T + R$$
$$K = P \cdot H^T \cdot S^{-1}$$
$$x' = x + (Ky)$$
$$P' = (I - K \cdot H) \cdot P$$

Predict: $x' = Fx + Bu$ (sometimes control-input model) $\quad P' = FPF^T + Q$ (Q: process noise cov. matrix)

Update

$$y = z - Hx'$$
$$S = R + HP'H^T$$
$$K = P' \cdot H^T \cdot S^{-1}$$
$$x = x' + Ky$$
$$P = (I - KH)P'$$
$$y = z - Hx$$

May 09. 22:00
May 14. 21:00

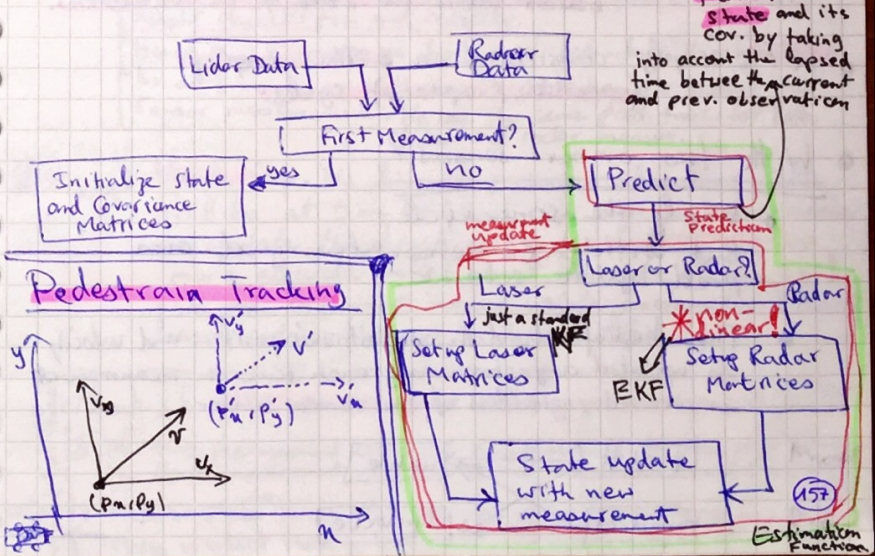
Lesson 24: Extended Kalman Filters

4 hours

- **Extended KF** (EKF): A **non-linear** version of Kalmon Filter which linearizes about an estimate of the current mean and covariance.

capable of handling more complex motion and measurement models

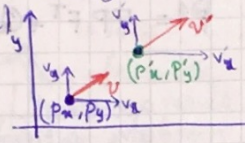
- Structure of Sensor Fusion using KFs:



Pedestrain Tracking

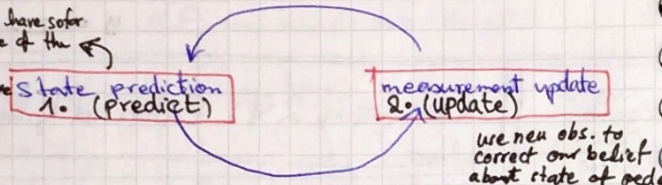
Overview: Sensor Fusion with KF

- We want to track a pedestrian moving in front of our AV (autonomous vehicle)



- The KF uses a two-step Estimation problem

use information we have so far to predict the state of the pedestrian until the next measurement arrives.

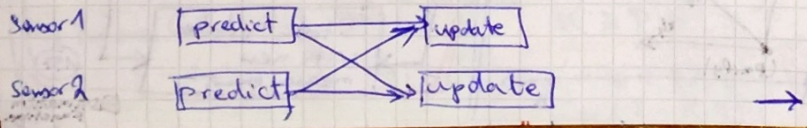


- we always start with what we already know about the pedestrian
- We use that information to predict the state of the pedestrian up to the time when the next measurement arrives. This is called prediction step.

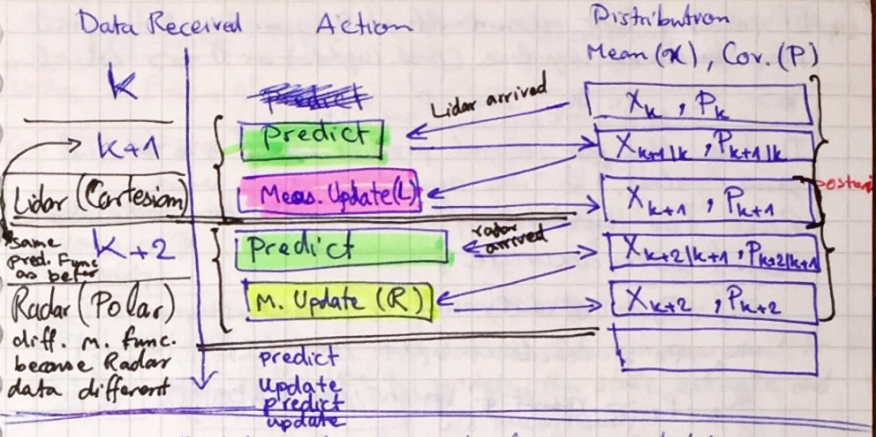
Prediction Step

- **Update Step**: we use the new observation to correct our belief about the state of the pedestrian.
- KF: simply consists of an endless loop of (predict / update) cycles.

- With two or more sensors?
  - Simply the same cycle. But with each senser having it own pred/update scheme.
  - The belief about the pedestrian's position and velocity is updated asynchronously each time the measurement is received regardless of the source sensor.



→



$\mathbf{x}$: "mean" state vector; Pos. and Vel. are presented by a Gaussian distribution with mean $\mathbf{x}$

P: State Cov. matrix; information about the uncertainty of the object's pos. and velocity. Think of it as standard deviation

k: time step; x_k: object's pos. and vel. at time k

$x_{k+1|k}$: prediction step. At time $k+1$ we receive a sensor meas. We predict, where we think the object would be at time $k+1$, base it on its pos. and vel. at time k; Before, we take the sensor meas. to update our belief about object's pos. and velocity.

Hence $x_{k+1|k}$ means we have predicted where the object will be at time $k+1$, but haven't yet taken into account the sensor measurement. We use the same pred. func. for both lidar and radar measur.

x_{k+1}: Update Step. We have now predicted where the object will be at time $k+1$ and then used the sensor measur. to update the objects pos. and vel. (posterior: a more accurate belief about the state at time $k+1$)

(we use different measur. update function for lidar and radar, (because the data is different))

- What if two measurements at the same time? $(k+3)$
  - if two measurement come in simultaneously, the time step between the first measurement and the second measurement would be zero
  - Predict state to $k+3$, the use use either one of the sensors to update
  - Then predict state to $k+3$ again, and update with the other sensor

repeat two times

- If receiving two measurements at the same time the time step between cycles (pred. /update) is 0 $\Rightarrow \Delta t = 0$

$\Rightarrow x'_{t+1} = x_t$ and $P'_{t+1} = P_t$

That is the second predict would give us the same values, so we can discard it and. Run two updates after each other to make our belief more accurate:

x_0, P_0 := result of predict step

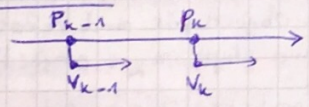
x_1, P_1 := result of update state LIDAR(x_0, P_0)

x_2, P_2 := result of update state RADAR(x_1, P_1)

The final result is (x_2, P_2)

Recap: KF for simple 1D motion:

$x = \begin{pmatrix} p \\ v \end{pmatrix}$ position (p), velocity (v)



- Design of the KF involves designing two linear functions:

models how the state is changed from time k-1 to time k ← **State transition Function**

$x' = f(x) + \nu = Fx + B\vec{u}^{\,0} + \nu \quad \nu \sim N(0, Q)$

$x' = Fx + \nu$

deterministic — stochastic (random noises that affect the prediction and update steps)

models how the measurment is calculated and how it's related to the predicted state ← **Measurement Function**

$z = h(x') + \omega = Hx' + \omega \quad \omega = N(0, R)$

State transition function

- Using a linear motion model with constant velocity:

$p' = p + v\Delta t$

$v' = v$

$\begin{pmatrix} p' \\ v' \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ v \end{pmatrix}$

Measurement Function — we only sense the position

$z = p'$

$z = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} p' \\ v' \end{pmatrix}$

← Prediction Step:

predicted state $x' = Fx + \nu$

Cov. $P' = FPF^T + Q$

predict the state and covariance P (uncertainty)

Predict Step in Lesson map: (see 4 steps before)

compute elapsed time Δt → use Δt to compute new EKF, F, Q → predict x, P

Measurement Update Step: (correction step)

$y = z - Hx'$

$S = HP'H^T + R$

$K = P'H^T S^{-1}$

$x = x' + Ky$

$P = (I - KH)P'$

Use the latest measurement to update the state estimate and its uncertainty

- For 2D:

State Transition Equation

$$\begin{pmatrix} p'_x \\ p_y \\ v'_x \\ v_y \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} + \begin{pmatrix} \nu_{p_x} \\ \nu_{p_y} \\ \nu_{v_x} \\ \nu_{v_y} \end{pmatrix}$$

$x' = F \cdot x + \nu$

- Process Cov. Matrix for 2D:

$P' = FPF^T + Q$

$$Q = \begin{pmatrix} \frac{\Delta t^4}{4}\sigma_{ax}^2 & 0 & \frac{\Delta t^3}{2}\sigma_{ax}^2 & 0 \\ 0 & \frac{\Delta t^4}{4}\sigma_{ay}^2 & 0 & \frac{\Delta t^3}{2}\sigma_{ay}^2 \\ \frac{\Delta t^3}{2}\sigma_{ax}^2 & 0 & \Delta t^2\sigma_{ax}^2 & 0 \\ 0 & \frac{\Delta t^3}{2}\sigma_{ay}^2 & 0 & \Delta t^2\sigma_{ay}^2 \end{pmatrix} \quad (4 \times 4)$$

~~Derivation~~ Calculation of Q: Because our state model only tracks position and velocity, we model acceleration as a random noise.

- the matrix Q includes Δt to account for the fact that as more time passes we become more uncertain about our estimate of position and velocity. So, as Δt increases, we add more uncertainty to the state cov. matrix P.

← our 2D motion model is: $v_x, v_y, v_{x_2}, v_{y_2} \Rightarrow a = \frac{\Delta v}{\Delta t} = \frac{v_{t+1} - v_t}{\Delta t}$

$$\begin{cases} p'_x = p_x + v_x \Delta t + \frac{a_x \Delta t^2}{2} \\ p'_y = p_y + v_y \Delta t + \frac{a_y \Delta t^2}{2} \\ v'_x = v_x + a_x \Delta t \\ v'_y = v_y + a_y \Delta t \end{cases}$$

deterministic part | Stochastic part: acceleration is considered in noise component.

$\vec{\nu} \in \mathcal{N}(0, Q)$, $a_x = \mathcal{N}(0, \sigma_{ax}^2)$, $a_y = \mathcal{N}(0, \sigma_{ay}^2)$

$$\vec{\nu} = \begin{pmatrix} a_x \frac{\Delta t^2}{2} \\ a_y \frac{\Delta t^2}{2} \\ a_x \Delta t \\ a_y \Delta t \end{pmatrix} = \begin{pmatrix} \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t}{2} \\ \Delta t & 0 \\ 0 & \Delta t \end{pmatrix} \begin{pmatrix} a_x \\ a_y \end{pmatrix} = G\,a$$

G: does not contain random variables.

a: contains random acc. components.

process cov. matrix:

$$Q = \mathbb{E}[\vec{\nu}\vec{\nu}^T] = \mathbb{E}[G a a^T G^T]$$

because G does not contain random variables (it is constant)

$$= G\,\mathbb{E}[aa^T]\,G^T = G \begin{pmatrix} \sigma_{ax}^2 & \sigma_{axy} \\ \sigma_{axy} & \sigma_{ay}^2 \end{pmatrix} G^T$$

$$= G Q_v G^T$$

expect. of and (cov.)

$$\Rightarrow Q_v = \begin{pmatrix} \sigma_{ax}^2 & \sigma_{axy} \\ \sigma_{axy} & \sigma_{ay}^2 \end{pmatrix}$$: three statistical moments

expectation of a_y (cov.)

$\sigma_{axy} := 0$ because a_x and a_y are assumed to be uncorrelated noise processes

$$\Rightarrow Q_v = \begin{pmatrix} \sigma_{ax}^2 & 0 \\ 0 & \sigma_{ay}^2 \end{pmatrix} \Rightarrow Q = G Q_v G^T = \begin{pmatrix} 4 \times 4 \end{pmatrix}$$

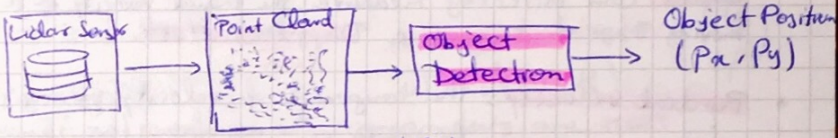
(see prev. page)

• New Measurement Model for 2D

measur. vector z
measur. matrix H
cov. matrix R

important

- The laser sensor output is a point cloud. But here we assume we already analyzed the point cloud to compute the 2D location of the pedestrian.



$$z = \begin{pmatrix} p_x \\ p_y \end{pmatrix}, \quad \text{state } x = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} \qquad z = H \cdot x + w$$

- H: measurement matrix projects our belief about the object's current state into the measurement space of the sensor. For lidar, this means, we discard the velocity information, since the lidar only measures the position: The state vector x contains information about $[p_x, p_y, v_x, v_y]$. The measurement vector z contains only information about $[p_x, p_y]$. Multiplying Hx allows us to compare x, our belief, with z, the sensor measurement.

$$\Rightarrow H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

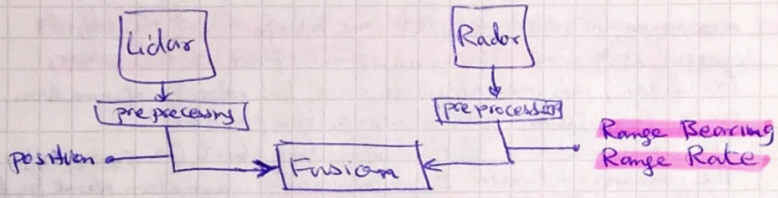
- R: measurement cov. matrix. Uncertainty of sensor about its measurments. Square matrix, with each side the same as the number of measurement parameters.

$$R = \mathbb{E}[ww^T] = \begin{pmatrix} \sigma_{px}^2 & 0 \\ 0 & \sigma_{py}^2 \end{pmatrix}$$

- usually the parameters for the random noise measurement matrix will be provided by the sensor manufacturer.
- off-diagonal 0s in R indicate that the noise processes are uncorrelated.

Radar Measurements:

- LIDAR → preprocessing → Position (but no velocity directly)
  RADAR → preprocessing → radial velocity
- Radar can directly measure the radial velocity of a moving object using the Doppler Effect.
- Radial velocity: the component of velocity moving towards or away from the sensor.
- But Radar has a lower ~~radial~~ spatial resolution, than Lidar. ⇒ combine both sensors together.



- State transition function will be the same as in the lidar case

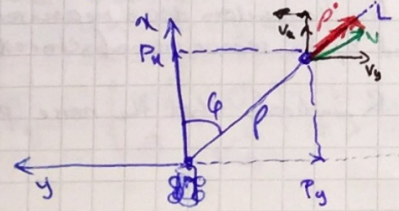
$$x' = f(x) + \nu \qquad x = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix}$$

same state vector)
same linear motion model and process noise

- measurement function is different

$$z = h(x') + \omega$$

- The Radar sees the world differently



- x-axis always points in the vehicles direction of movement
- y-axis points to the left

→

- instead of a 2D pose, the Radar can directly measure

$z = \begin{cases} \text{Range } (\rho): \text{ radial distance from origin.} \\ \text{Bearing } (\varphi): \text{ angle between } \rho \text{ and } x\text{-axis's} \\ \text{Range rate } (\dot{\rho}): \text{ change of } \rho \text{ (radial velocity)} \end{cases}$ normalize $[-\pi, \pi]$

velocity along ρ vector.

$$z = h(x') + \underbrace{\omega}_{\text{noise}}$$

measurement vector $z = \begin{pmatrix} \rho \\ \varphi \\ \dot{\rho} \end{pmatrix}$

(assuming) $\omega \sim \mathcal{N}(0, R)$

- Three measurement components are not cross-correlated

$$\Rightarrow R = \begin{pmatrix} \sigma_\rho^2 & 0 & 0 \\ 0 & \sigma_\varphi^2 & 0 \\ 0 & 0 & \sigma_{\dot{\rho}}^2 \end{pmatrix}$$

Radar measur. cov. matrix

$$\begin{pmatrix} \rho \\ \varphi \\ \dot{\rho} \end{pmatrix} \leftarrow h(x') \leftarrow \begin{pmatrix} p_x' \\ p_y' \\ v_x' \\ v_y' \end{pmatrix}$$

z: measurment vector

measurement Function: maps the predicted state x' into the measurement space.

x: State Vector

$$h(x') = \begin{pmatrix} \sqrt{p_x'^2 + p_y'^2} \\ \arctan\left(p_y'/p_x'\right) \\ \dfrac{p_x' v_x' + p_y' v_y'}{\sqrt{p_x'^2 + p_y'^2}} \end{pmatrix}$$

measurement function

$h(x')$ is not linear

Derivation of $\dot{\rho}(t)$:

$$\rho(t) = \sqrt{p_x(t)^2 + p_y(t)^2}$$

$$\dot{\rho} = \frac{\partial \rho(t)}{\partial t} = \dots = \frac{p_x v_x + p_y v_y}{\sqrt{p_x^2 + p_y^2}}$$

$$\frac{\partial p_x}{\partial t} = v_x, \quad \frac{\partial p_y}{\partial t} = v_y$$

Alternative:

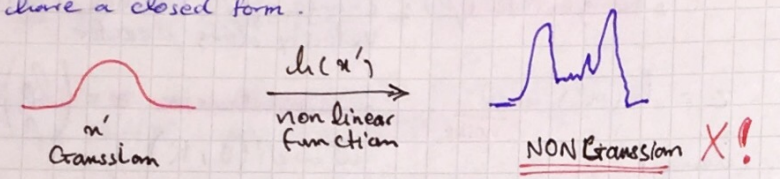
- $\dot{\rho}$ can be seen as the scalar projection of velocity vector $\vec{v}$ onto the $\vec{\rho}$.

$$\vec{\rho} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad \vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

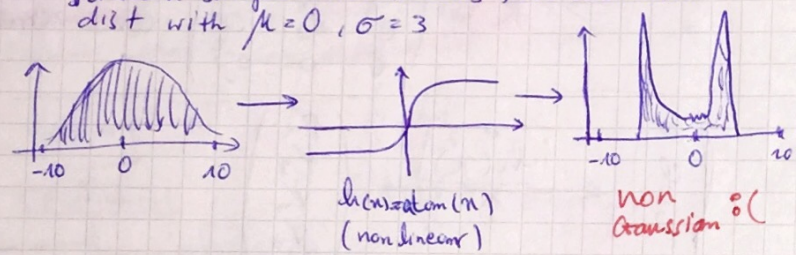
$$\dot{\rho} = \frac{\vec{v}\vec{\rho}}{\|\vec{\rho}\|} = \frac{(v_x, v_y)\begin{pmatrix} p_x \\ p_y \end{pmatrix}}{\sqrt{p_x^2 + p_y^2}}$$

Extended Kalman Filters

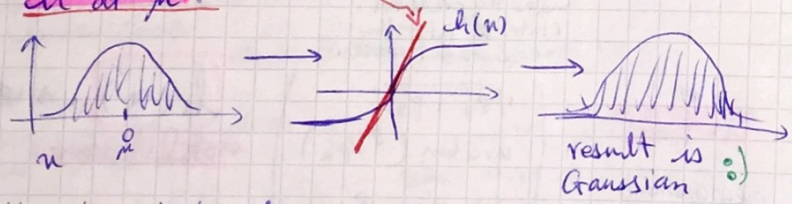
- Say we have our predicted state x' described by a Gaussian dist.
- If we map this dist. by a non linear function, the resulting dist is not a Gaussian. It doesn't also have a closed form.



Example:
Random sample 10 000 numbers from a Gaussian dist with $\mu = 0$, $\sigma = 3$



- Replace $h(x)$ with the linear approximation of h at μ.



- How to get the linear approximation? (How to linearize a non linear function?)
- Taylor (series) Expansion: of $f(x)$ at point μ.

$$f(x) \approx f(\mu) + \frac{\partial f(\mu)}{\partial x}(x-\mu)$$

Multivariate Taylor Series:

- Remeber the measurement function $h(x')$ for Radar measurements was composed of three equations that showed how to map the state $x' = (p_x, p_y, v_x, v_y)^T$ to measurement space $z = (\rho, \varphi, \dot{\rho})^T$

$$h(x') = \begin{pmatrix} \sqrt{p_x'^2 + p_y'^2} \\ \arctan(p_y'/p_x') \\ \frac{p_x' v_x' + p_y' v_y'}{\sqrt{p_x'^2 + p_y'^2}} \end{pmatrix}$$

Kalman (not) Müller (not) Algolux

Multivariate Taylor series at point a

$$T(x) = f(a) + (x-a)^T Df(a) + \frac{1}{2!}(x-a)^T D^2 f(a)(x-a) + \dots$$

$Df(a)$ = Jacobian Matrix (first order derivative)
$D^2 f(a)$ = Hessian matrix (second order derivative)

Linear Approximation of the function

- only keep the Taylor expansion up to the Jacobian term $Df(a)$. We ignore the Hessian term and other higher order terms. Assuming that $(x-a)$ is small, $(x-a)^2$ or the multidimensional equivalent $(x-a)^T(x-a)$ will be even smaller; our extended KF assumes the higher order terms are negligible.

State vector $x = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix}$ measurement function describes three components $z = \begin{pmatrix} \rho \\ \varphi \\ \dot{\rho} \end{pmatrix}$ ← normalize $[-\pi, \pi]$

⇒ Jacobian of $h(x)$ is a 3×4 matrix

$$H_j = \begin{pmatrix} \frac{\partial \rho}{\partial p_x} & \frac{\partial \rho}{\partial p_y} & \frac{\partial \rho}{\partial v_x} & \frac{\partial \rho}{\partial v_y} \\ \frac{\partial \varphi}{\partial p_x} & \frac{\partial \varphi}{\partial p_y} & \frac{\partial \varphi}{\partial v_x} & \frac{\partial \varphi}{\partial v_y} \\ \frac{\partial \dot{\rho}}{\partial p_x} & \frac{\partial \dot{\rho}}{\partial p_y} & \frac{\partial \dot{\rho}}{\partial v_x} & \frac{\partial \dot{\rho}}{\partial v_y} \end{pmatrix} \quad (3 \times 4)$$

$$H_j = \begin{bmatrix} \frac{p_x}{\sqrt{p_x^2+p_y^2}} & \frac{p_y}{\sqrt{p_x^2+p_y^2}} & 0 & 0 & 0 \\ -\frac{p_y}{p_x^2+p_y^2} & \frac{p_x}{p_x^2+p_y^2} & 0 & 0 & 0 \\ \frac{p_y(v_x p_y - v_y p_x)}{(p_x^2+p_y^2)^{3/2}} & \frac{p_x(v_y p_x - v_x p_y)}{(p_x^2+p_y^2)^{3/2}} & \frac{p_x}{\sqrt{p_x^2+p_y^2}} & \frac{p_y}{\sqrt{p_x^2+p_y^2}} \end{bmatrix}$$

$\rho = \sqrt{p_x^2+p_y^2}$ $\quad \varphi = \arctan(p_y/p_x)$ $\quad \dot{\rho} = \frac{p_x v_x + p_y v_y}{\sqrt{p_x^2+p_y^2}}$

- **Extended Kalman Filter:** (state transition)
  - First linearize the nonlinear prediction (F) and measurement (H) function
  - Use the normal KF scheme using the linearized F_j and H_j (= Jacobians)

Kalman Filter

prediction

$x' = Fx + u$

$P' = FPF^T + Q$

measurement update

$y = z - Hx'$

$S = HP'H^T + R$

$K = P'H^T S^{-1}$

$x = x' + Ky$

$P = (I - KH)P'$

Extended Kalman Filter

$x' = f(x, u')$ $\quad u' = 0$

use F_j instead of F

(Why simply F_j and H_j instead of Taylor expansion of F and H at $x = x'$? See the proof.)

$y = z - h_j(x')$

use H_j instead of H

- The mathematical proof is complex, but the equations for KF and EKF are very similar. The differences are:

→

←

- the F matrix will be replaced by F_j when calculating P'
- the H matrix will be replaced by H_j when calculating S, K, and P
- to calculate x', the prediction function f, is used instead of F matrix
- to calculate y, the h function is used instead of H matrix.

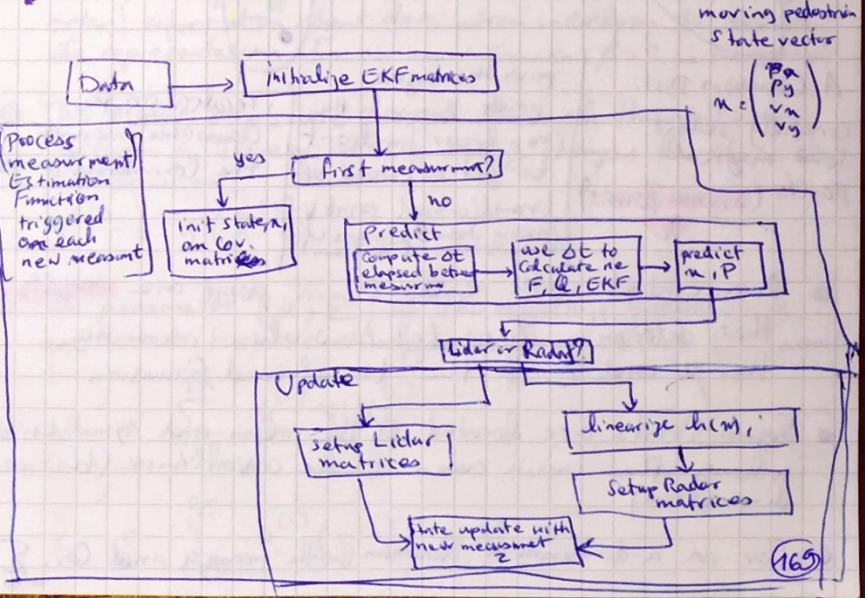
- Evaluating performance of KF: RMSE metric

$$\sqrt{\frac{1}{n}\sum_{t=1}^{n}(x_t^{est} - x_t^{ground\,truth})^2}$$

May 14. 21:00
May 18. 20:00

Project 5: Extended Kalman Filter

- RViz: ROS Visualization tool.
- Sensor Fusion General Flow

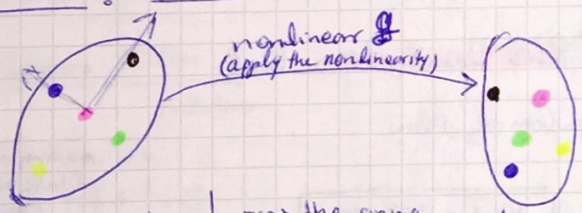


Extra: Unscented KF

May 18, 21:00
May 19, 18:00

- UKF does not use a Taylor series for linearizing $f(x)$ and $h(x)$
- It avoids to linearize around the mean as the EKF does.
- It is based on the assumption/intuition that it is easier to approximate a Guassian distribution than it is to approximate an arbitrary nonlinear function.
- It uses a stochastic linearization through the use of weighted statistical linear regression process.

Idea: Unscented Transform



A Gaussian Dist. Carefully select some magical points (Sigma Points) χ

pass the sigma points through the nonlinear function to get a cloud of transformed points y (probing how g changes the shape of Gaussian)

recalculate the Gaussians around the transformed points

- Associated with each Sigma point are weights that determine its contribution when recovering the μ and Σ of the transformed Gaussian.
- Sigma Points are located at the mean and symmetrically along the main axes of the covariance (two per dimension)
- For an n-dimensional Gaussian with mean μ and Cov. Σ →

←

the $2n+1$ Sigma points are chosen according to the following rule:

$$\chi^{[0]} = \mu$$

$$\chi^{[i]} = \mu + \left(\sqrt{(n+\lambda)\Sigma}\right)_i \quad \text{for } i = 1, \dots, n$$

$$\chi^{[i]} = \mu - \left(\sqrt{(n+\lambda)\Sigma}\right)_i \quad \text{for } i = n+1, \dots, 2n$$

where $\lambda = \alpha^2(n+\kappa) - n$ with α and κ scaling param that determine how far the sigma points are spread from the mean.

- Each Sigma point $\chi^{[i]}$ has two weights associated with it: One weight $w_m^{[i]}$, is used when computing the mean, the other weight, $w_c^{[i]}$, is used when recovering the Cov. of the Gaussian.

$$w_m^{[0]} = \frac{\lambda}{n+\lambda}, \quad w_c^{[0]} = \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta)$$

$$w_m^{[i]} = w_c^{[i]} = \frac{1}{2(n+\lambda)} \quad \text{for } i = 1, \dots, 2n$$

$\sum w_i = 1$

$\mu = \sum_i w_i \chi_i$

$\Sigma = \sum_i w_i (\chi_i - \mu)(\chi_i - \mu)^T$

the param β can be chosen to encode additional (higher order) information about distribution underlying the Gaussian. For an exact Gaussian $\beta = 2$ is the optimal

- The sigma points are passed through the nonlinear function g thereby probing how g changes the shape of the Gaussian

$$y^{[i]} = g(\chi^{[i]})$$

- The parameter (μ', Σ') of the resulting Gaussian are extracted from the mapped sigma points $y^{[i]}$ according to:

$$\mu' = \sum_{i=0}^{2n} w_m^{[i]} y(\chi^{[i]})$$

$$\Sigma' = \sum_{i=0}^{2n} w_c^{[i]} (y^{[i]} - \mu')(y^{[i]} - \mu')^T$$

- Algorithm Unscented KF $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:

prediction step

$\gamma = \sqrt{n+\lambda}$

$$\chi_{t-1} = (\mu_{t-1} \quad \mu_{t-1} + \gamma\sqrt{\Sigma_{t-1}} \quad \mu_{t-1} - \gamma\sqrt{\Sigma_{t-1}})$$

$$\bar{\chi}_t^* = g(u_t, \chi_{t-1})$$

$$\bar{\mu}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\chi}_t^{*[i]}$$

$$\bar{\Sigma}_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\chi}_t^{*[i]} - \bar{\mu}_t)(\bar{\chi}_t^{*[i]} - \bar{\mu}_t)^T + R_t$$

Measurement update step

$$\bar{\chi}_t = (\bar{\mu}_t \quad \bar{\mu}_t + \gamma\sqrt{\bar{\Sigma}_t} \quad \bar{\mu}_t - \gamma\sqrt{\bar{\Sigma}_t})$$

$$\bar{Z}_t = h(\bar{\chi}_t)$$

$$\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{Z}_t^{[i]}$$

$$S_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{Z}_t^{[i]} - \hat{z}_t)(\bar{Z}_t^{[i]} - \hat{z}_t)^T + Q_t$$

$$\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\chi}^{[i]} - \bar{\mu}_t)(\bar{Z}_t^{[i]} - \hat{z}_t)^T$$

- cross-covariance between state and observation, use to compute the Kalmen gain
- corresponds to the term $P'H^T$ in EKF.

Kalman Gain $K_t = \bar{\Sigma}_t^{x,z} S_t^{-1}$

$$\mu_t = \bar{\mu}_t + K(z_t - \hat{z}_t)$$

$$\Sigma_t = \bar{\Sigma}_t - K_t S_t K_t^T$$

returns μ_t, Σ_t

- UKF is almost as fast as EKF.
- For non linear systems UKF is as good or better than EKF with difference being higher with the degree of non linearity. In many practical applications though the difference between UKF and EKF are negligible.
- UKF is derivative-free

Part 2

19. May 18:00

Localization | Path Planning
control | System Integration

19. May 18:00

Localization
Lesson 1: Intro to Localization

- Tiffany Huang: Localization Engineer (Mercedes-Benz) (MB RD North America)
- Maximilian Muffert: " " " " "

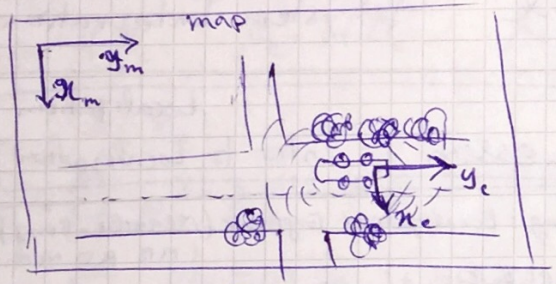
- Localization: A Robot takes the information about the environment and compares that to information it already knows about the world.

- Intuition
- Blind folded in the world ⟹ can be anywhere, no belief uncertainty as big as the whole world
- Open eye and see Eifel tower

- Intuition: Reduce the uncertainty by measuring/sensing the world and comparing that measurement to the pre known model of the world and hence building a belief.

- Answer the question:
  where is the car in a given map with an accuracy of 10 cm or less.

- GPS ⟶ 3 - 50 meters (not usable

- 

- Find the transformation between ~~local~~ local coordinate system of the car and the global coordinate system of the map by finding position of objects for whom we know their position in the map.

- 1D Baysian Filter in C++
- Motion models
- 2D Particle Filter in C++

20. May 12:00
22. May 20:00

Lesson 2: Markov Localization

~~6h~~ 3h

- Localization: Multiplication (Bayes Rule sense/measurement) and Addition (motion (convolution) Total probability)
- Whole underling math behind the General localization Problem
  - ↳ Derivation of the Bayes Localization Filter (Markov Localization) (one of the most common loc. frameworks)
  - ↳ 1D Realization of Markov Loc. Filter in C++
  - ↳ Motion & Observation Model.
- The project: Kidnapped Vehicle uses Particle Filters.
- Markov Localization is a generalized Filter for localization and all other loc. approaches are realizations of this approach.

- By Understanding the Markov Loc. Filter, we develop intuition and methods that help us solve any vehicle loc. tasks, including Particle Filters.
  - ↳ Think of Vehicle position generally as a Prob. dist.
  - ↳ Each time we move ⟶ dist. becomes wider more uncertain
  - ↳ Each time we feed the filter with data (map data, measurement, control) ⟶ the dist. becomes ~~more certain~~ more concentrated more narrow

What do we have?

- Map with Landmark positions in global co.sy.
- Observation from the onboard sensors in the local co.sy.
- information about how the car moves between two time steps.