Lecture 2: Framing the ML problem

2.1 The ML process

ML is the process of training a piece of software, called a model, to make useful predictions using a data set
⇒ the model can then make predictions from new, unforeseen data

1. Hypothesis
2. Data
   - Obtain
   - Explore
   - Clean
3. Model
   - train
   - evaluate
   - repeat
4. Build the data pipeline
5. Run the model
   - Monitor
   - Evaluate
   - Adapt

} Scientific method (compared)

Waterfall (sequential): Requirements → Design → Implementation → Testing → Release
Agile: Requirements → Prototype → Feedback → Repeat until satisfied

The ML mindset is different from traditional software development
↳ - Experiment until you find amazing, useful model
- Produces bugs that are difficult to debug

2.2 The ML Problem Spectrum

- feature: an input variable on the basis of which a prediction can be made, like weight
- label: the output we are predicting

2.2.1 Supervised Learning: The model is provided with labeled training data (in supervised)

data contains features with corresponding labels
relationship between these two is the model

2.2.2 Unsupervised Learning:
- the goal is to identify meaningful patterns in the data
- machine must learn from unlabeled data set
- model has no hints how to categorize each piece of data

↳ e.g. Reinforcement Learning:
- You tell the model (agent) the goal
- During training, the agent receives a reward when reaching the goal (reward function) — difficult to come up with
⇒ - less stable and predictable than supervised models
- You need a way for agent to communicate with environment of interest

2.3 Types of ML models

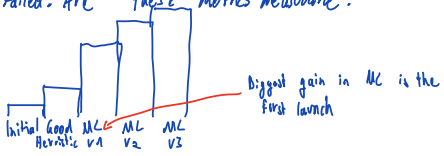
		Example
Classification:	Pick one of N labels	Cat, dog, horse
Regression:	Predict numerical values	
Clustering:	Group similar examples	Most relevant documents (unsupervised)

Association rule learning: Infer likely association patterns in data — If you buy hamburger buns, you're likely to buy patties (unsupervised)
Structured output: Create complex outputs — NLP trees, image recognition bounding boxes
Ranking: Identify position on a scale as a task: — Search result ranking

Hard ones: clustering, anomaly detection, no existing data, causation

2.4 Deciding on ML

- Start clearly and simply: what would you like your ML model to do? What is your ideal outcome?
- Success and failure metrics: How will you know if the system has succeeded or failed? Are these metrics measurable?
- What output would you like the ML model to produce?
- How might you solve your problem without ML? e.g. heuristics



2.5 Formulate your problem as an ML problem

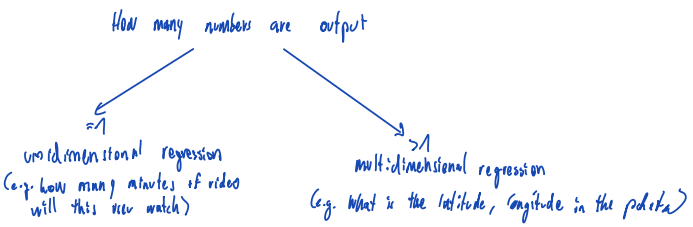
2.5.1 Articulate your problem

- Our problem is best framed as ...
  - Binary classification
  - regression
  - Multi-class single-label classification
  - Multidimensional regression
  - Clustering
  - Other ...
- Which predicts ...

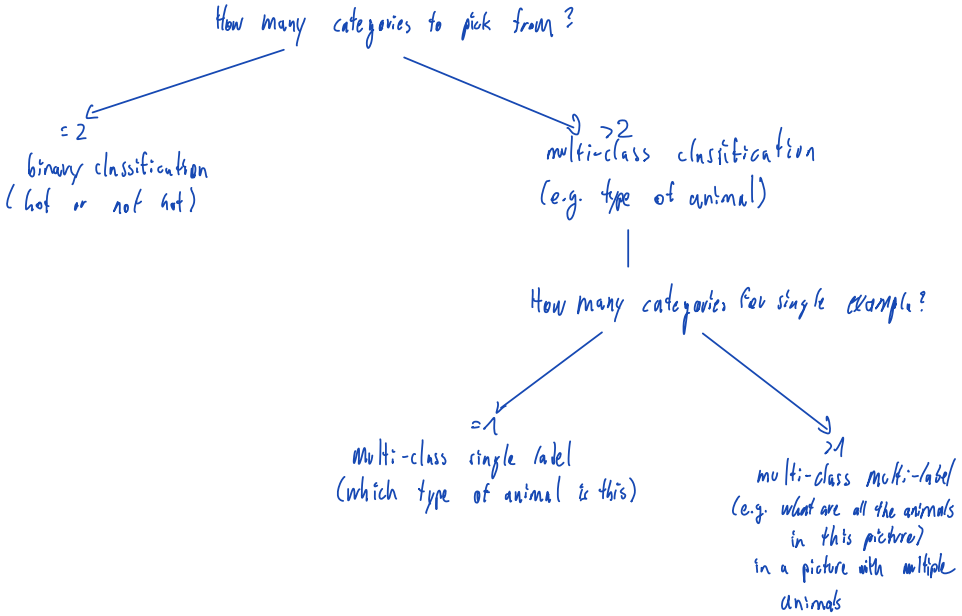
2.5.2 Start simple

- Simplify the model: - Simple model is easier to understand, implement and iterate
  - Full data pipeline for complex models is harder than iterating on simpler models
  - ⇒ Always start with unidimensional regression problem or binary classification first
    - ↪ If neither fits, try other model types
- Once you start with the simplest model, iterate to improve
  - ⇒ Can be your baseline model, can help you decide if a more complex model is needed

Regression flow chart:



Classification flow chart:



2.5.3 Identify your data sources

- How much labeled data do we have
- What is the source of your label
- Is your label closely connected to the decision you will be making?

2.5.4 Design your data for the model

- Identify the data that your ML system should use to make predictions (input → output)
- Provide a sample data table:

- Rows: A feature can be a scalar or 1D list of strings or numeric values
  ↳ If not 1-dimensional, consider splitting into separate inputs

2.5.5 Determine where data comes from

- Assess how much work it would be to develop a data pipeline to construct
- Make sure all inputs are available at prediction time ⇒ if it will be difficult to obtain feature values at prediction time, omit those features from your model

2.5.6 Determine easily obtained inputs

- Pick 1-3 features that would be easy to obtain, that would provide a reasonable, initial outcome
- Which features would be useful for implementing heuristics mentioned previously
- Start from minimal possible infrastructure with simple pipeline
- Consider the engineering cost to develop a data pipeline to prepare the inputs

2.5.7 Ability to learn

- Will the ML model be able to learn
- List aspects of your problem that might cause difficulty learning
- For example: - The data does not contain enough positive labels
  - The training data doesn't contain enough examples
  - The labels are too noisy
  - The system memorizes the training data, but has difficulty generalizing to new cases

2.5.8 Think about potential bias

- Many datasets are biased in some way
  e.g.: - Watch out for "popular": May inforce unfair or unbiased views.
  - training data might not be representative of population

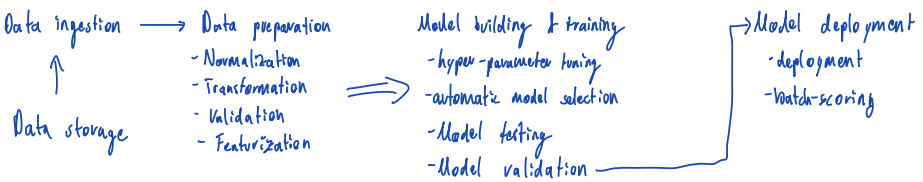
Lecture 3

3.1 Key terminology

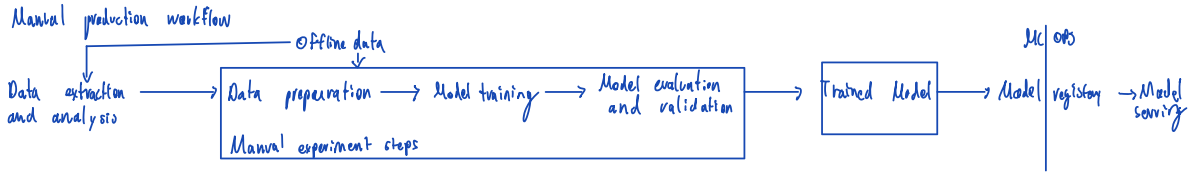
- Labels: the thing we are predicting, the y variable
- Features: the input (variables), the x variable
- Models: define relationship between features and labels
- Regression Model: Continuous prediction values
- Classification: Predicts discrete values

3.2 Workflow

3.2.1 General workflow



3.2.2 Manual production workflow



3.2.3 Automated production workflow

3.3 Data collection methods

3.3.1 CSV-File: Comma separated value file format

+ text file, simplest data format
+ human readable, excel compatible
+ import pandas

```
df = pandas.read_csv('hrdata.csv')
print(df)
```

- difficult to debug: embedded commas in data / missing columns

⇒ - Better suited for "smaller" data sets
- (probably) not good enough for production quality workflow

3.3.2 SQL: standard query language

- used when interacting with relational databases
- read and write into sql database using python, make part of project pipeline

3.3.2 API: Application Programming Interface

- Standard way to communicate with other programs
- Is often published by developers
- Replaces passing files around
- Scales better
- RESTful APIs: see funniaja (will be mostly GET-requests), usually JSON
- Client-Server via HTTP
- Stateless, cacheable

3.3.3 JSON: Javascript Object Notation
- Widely used standard
- Should never write/read JSON directly (use parsing libraries)
- Simple, human-readable
- not optimal

3.3.4 Web Scraping

- To be avoided, last resort
- Python: beautiful soup LINUX/UNIX: curl
- Scrape data directly from html code

3.4 Data preparation

- Data cleaning: Identifying and correcting mistakes and errors in the data
- Feature selection: Identifying those input variables that are most relevant to the task
- Data transforms: Changing the scale or distribution of variables
- Feature engineering: Deriving new variables from available data.
- Dimensionality reduction: Creating compact projections of the data

3.6 Generalization: Simpler models tend to allow for more generalization

- Overfitting: Creating a model that matches the training data so closely that it fails to make correct predictions on new data ⇒ the model does not generalize well to new data; making model more complex than necessary
  Underfitting (vice versa)
  ⇒ Renzos suit example

Big picture: - Goal: predict well on new data drawn from hidden true distribution
- Problem: - new data comes from a population but we only see samples
- If model fits current samples well, how can we trust it will predict well from new samples

⇒ Ockham's Razor: Scientist should prefer simpler formulas or theories over more complex ones
⇒ the less complex the ML model, the more likely that a good empirical result is not just due to peculiarities of the sample.

⇒ fit data well, but also as simply as possible

5.5.2 Test Set Methodology: take another draw from the distribution and see how we do

- Supervised learning assumptions:
  - i.i.d = examples don't influence each other, and are drawn independently, identically, at random
  - Stationarity (see time series course)
  - same distribution: we pull our data from the same distribution

3.6 Splitting data

- two ways to split:
  - Draw twice: your training and test sets from same distribution seperately
  - Draw once: randomly split your data to training and test sets

⇒ Good performance on the test set is a good indicator the model will perform well on new data given that,
- the test set is large enough
- Obviously: Never train on test set (e.g. duplicates in test an training set)

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(scaled_features, labels, test_size=0.1, random_state=1)
```

3.6.2 How big should the test set be?

- the bigger the better e.g. 90/10, 85/15, 80/20 are good heuristics

3.6.3 Data set partition

2 partition:
1. train model on training set
2. Evaluate model on test set
3. Tweak model according to results on test set
4. Repeat
5. Pick model that does best on test set

⇒ the more the workflow is repeated, the greater the chances of overfitting
↳ use 3-partition set

3 partition (introduce validation set)
1. train model on test set
2. Evaluate model on validation set
3. Tweak model based on results on validation set
4. Pick model that does best on validation set
5. Confirm results on test set

⇒ better workflow, because it creates fewer exposures to the test set.

Validation set because:

feature ↑ label ↑

1. Train on TrainX and Y
2. Test on TestX and Y
3. Update the model based on results

↓

this will fit the data to this specific test set, the model should only see the test set once use validation set for that purpose

Lecture 4

4.1 Feature Engineering: extracting features from raw data (in the form of logs, databases, images, etc...)

e.g.:

```
0: {
    house_info: {
        roomcount: 6
        bedrooms: 4
        bathrooms: 3
    }
}
```

Feature engineering →

Features: [6.0, 4.0, 3.0]

⇒ Transform: - numerical data to floats
⇒ Some numerical is not ought to be converted / to be multiplied by weights (e.g. zipcode)
- strings: hot-encoding

One-hot encoding:

	Sun	Moon	Earth
0 Sun	1	0	0
1 Sun	1	0	0
2 Moon	0	1	0
3 Earth	0	0	1

← Multi hot encoding: same thing, but less size since multiple 1's can exist

#categories ↓ $\log_2(x)$

the one that has a true value is one, the others are zero (row-wise)

Hot encoding:

1. Define vocabulary for strings (set of strings, e.g. categorical features) ↗ cannot just assign a number to each string due to weight multiplication
   ⇒ Any string not in the set is out-of-vocabulary
2. One-hot encoding: Binary vector for each string in the set with one element=1 all others 0
3. Multi-hot encoding: A binary vector for each string, where multiple elements can be set to 1

pd.get_dummies()

green	blue	red
1	0	0
0	1	0
0	0	1

} inefficient to store for large string sets

4.2 Qualities of good features

- Avoid rarely used discrete feature values
  ⇒ Feature values rarely used should appear at least 5 times, so the model can see it with different labels

?

- Use clear and obvious feature names
- Do not mix "magic" values with actual data (e.g. -1 for age)
- The feature definition should not change over time (dependencies with other systems)

4.3 Cleaning the data (Bad apples analogy: even a few bad apples can spoil the entire analysis

1. Know your data: visualize (plotting, frequency diagram)
2. Debug: Duplicate examples, missing values, training/validation leakage
3. Monitor
4. Check for quality: E.g. use_age_years ASO

4.4 Scaling feature values = scaling values from their natural range into a standard range

Manual:

```
y = pd.DataFrame()
for col in df.columns:
    y[col] = (df[col] - df[col].mean()) / df[col].std()
```

⇒ helps the model converge faster
⇒ helps avoid NAN number trap
⇒ helps the model learn appopiate weights for each feature (without scaling, the model will add more weight to features with a wider range)
⇒ Don't need same scale for all features

or:

```
scaler = StandardScaler()
scaler.fit(data) ⇒ different values
scaled_features = scaler.transform(data)
```

⇒ e.g. Linear Map Min Max [0,1], Z-Score (scaled = (value - mean) / stdev) (-3, and 3), Logarithmic (take log of every value)

4.5 Handling outliers (minimize the influence on extreme outliers

⇒ Not just chop them off ⇒ lose some truth
- Scaling is not always sufficient
- Clipping: All values greater than X become X ⇒ preserves the truth to some degree

- Binning: Converting continuous feature into discrete one
  - equal width: bins have the same width
  - equal frequency
  - rank
  - Quantiles: so that all buckets contain the same number of examples
  - Entropy-based
  - Other math functions

bad features: e.g. thermometer was left out in sun

bad labels: person mislabeled oak tree as maple

- Scrubbing: Removing bad individual examples
  - ↳ can also detect bad data in aggregate (if aggregates (max and min, ...) match your expectations)

4.6 Feature crossing: For solving non-linear problems
(comes from cross product)

- Create a feature cross (product): A synthetic feature, that encodes nonlinearity in the feature space by multiplying two or more input features together
  - e.g.: $X_3 = X_1 X_2$ treat this like any other feature
  - e.g.: $b + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2$ (treated like any other weight) Nonlinearity in linear model

- linear learners use linear models => scale well to massive data
- Feature crosses + massive data = efficient strategy for highly complex models (Neural nets provide another strategy)

4.6.2 Crossing one-hot vectors

e.g.: predict city-specific housing prices between rooms per person and housing price
[binned latitude X binned longitude X binned rooms per person]

- e.g. binned_latitude [0, 0, 0, 1, 0] : [$0_1, \ldots, 0_{24}, 1$] (One hot encoding) => Model can then learn particular associations about that conjunction
  binned_longitude [0, 1, 0, 0, 0]

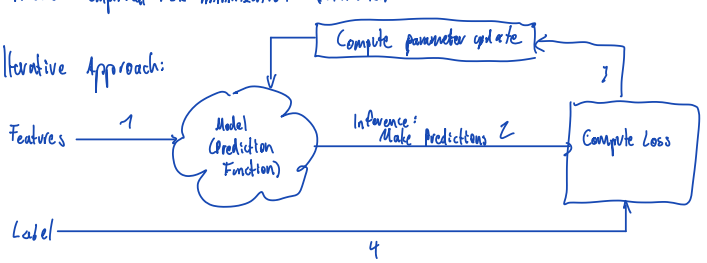
- Neural networks are a more sophisticated version of feature crosses => In essence, neural networks learn the appropriate feature crosses for you

4.6.3 Reducing Loss

- To build an ML model we need:
  - A problem T
  - A performance measure P
  - An experience E

- Training = learning (determining good values on average for all the weights and the bias from many labeled examples.
  (for supervised learning)

- good values => minimize loss
  - Loss = number indicating how bad a single prediction (measure of how "bad" the model is) (how far a model's predictions are from it's labels)
  - Process = empirical risk minimization (statistics)
  - ↳ Iterative Approach:



→ Make a prediction
→ Compare against label => discover new parameter with lowest possible loss => until loss stops changing or extremely slowly changes
→ Update parameters → model has converged
→ Make prediction

4.6.4 Linear Regression Loss

- Curve fitting is not ML : ML is looking for generalisation that can be applied to new data
- $y = mx + b$
- In ML: $y' = b + w_1 x_1$ (y': predicted label; b: bias / y-intercept; x_1: feature; w_1: weight)

```
model = LinearRegression
model.fit(x, y)
r_sq = model.score(x, y)
b = model.coeff_
y = model.intercept
```

- More features: $y' = b + w_1 x_1 + w_2 x_2 + w_3 x_3 + \dots + w_i x_i$

Loss functions: - MSE: Mean squared error: average squared loss per example over entire dataset

$$\frac{1}{N} \sum_{(x,y) \in D} (y - \text{prediction}(x))^2$$

- Log loss: (Binary Logistic Regression): $-[\, y_i \ln p_i + (1 - y_i) \ln (1 - p_i) \,]$
- MAE = mean absolute error: $\frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$
- MBE = mean bias error: $\frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)}{n}$

4.6.5 How to efficiently minimize loss:

$y' = b + w_1 x_1 + \dots + w_i x_i$

⇒ Not efficient to calculate the loss for all values of w_i, BUT loss function is convex w.r.t w_i

4.7 Gradient descent:

- Minimize loss by computing the gradients of loss w.r.t. to the models parameter (e.g. weights and biases)
- When performing gradient descent, we tune all the model parameters simultaneously

→ Calculate gradient of loss with values for both w_n and bias b
→ Modify hyperparameters based on this gradient
→ Repeat these steps until we reach minimum loss

E.g.: 1. Model: $y = mX + C$ 2. Loss function: $MSE = \frac{1}{n} \sum_{i=0}^{n} (y_i - \bar{y}_i)^2$ 3. $\frac{1}{n} \sum_{i=0}^{n} (y_i - (mX_i + C))^2 = D$

Learning rate / step size ↓

4. Gradient: $\frac{D}{\partial M} = \frac{1}{n} \sum_{i=0}^{n} 2 (y_i - (mX_i + C)) \cdot (-X_i)$

$\frac{D}{\partial M^2} = \frac{-2}{n} \sum_{i=0}^{n} X_i (y_i - \hat{y}_i)$

$\frac{D}{\partial M} = \frac{-2}{n} \sum_{i=0}^{n} (y_i - \bar{y}_i)$

6. $m = m - L \times D_m$
$c = c - L \times D_c$

⇒

partial derivative → gives direction (sign) and magnitude for minimization

4.8 Learning rate L

- The gradient has direction and magnitude
- influence magnitude with scalar L ⇒ can neither be to small (takes long) or too large (might step over)

=> In one dimension: $\frac{1}{f''(x)}$

=> In multiple dimensions: Inverse of Hessian matrix: H_f : $\begin{pmatrix} \frac{\delta^2 f}{\delta x_1^2} & \frac{\delta^2}{\delta x_1 \delta x_2} & \cdots & \frac{\delta^2}{\delta x_1 x_n} \\ \frac{\delta^2 f}{\delta x_2 x_1} & \ddots & & \\ \vdots & & \ddots & \\ \frac{\delta^2 f}{\delta x_n x_1} & & & \frac{\delta^2 f}{\delta x_n^2} \end{pmatrix}$

Gradient: Vector of first order derivatives of a scalar field (direction of fastest increase)

Jacobian: A matrix of partial derivatives for a set of functions. (The same as the gradient if its just one function)

Hessian: Matrix of second order mixed partial derivatives of a scalar field.

5. Regularization and Logistic Regression

5.1 Risk Minimization

- Loss: Measure of how "bad" a model is
- Empirical risk minimization: Minimize loss
- Structural risk minimization (SRM): - Minimize loss and complexity
  - minimize (loss function + λ(regularization function))
    - regularization rate
      - if too high, model will be simple, but underfitting is likely
      - if too low, model will be complex, won't generalize well to new data
    - ↳ help prevent overfitting

⇒ builds the most predictive model
⇒ keeps the model as simple as possible

5.2 Regularization: penalty on a models complexity

Complexity: - total number of features with nonzero weights
- weights of all the features

5.2.1 L_p Regularization

- L_1 penalizes |weight| ⇒ the derivative is a constant, subtracts a constant every time ⇒ could go to 0
- L_2 penalizes weight^2 ⇒ the derivative is 2·weight, subtracts x% of the weight every time ⇒ cannot go to 0
  - penalizes weight in proportion of the sum of absolut values of the weights
  - ↳ penalize weights in proportion to the sum of squares of the weights

L_1 (Lasso regression): helps drive the weights of irrelevant features to exactly 0 and remove them from the model
L_2 (ridge regression) helps drive outlier weights closer to 0, but not quite to 0 ⇒ always improves generalization in linear models

5.2.2 Updated Math

- L_1-Regularization Error function = $\sum_{i=0}^{n} (y_i - \sum_{j=0}^{n} x_{ij} w_j)^2 + \lambda \sum_{j=0}^{n} |w_j|$
- L_2-Regularization Error function = $\sum_{i=0}^{n} (y_i - \sum_{j=0}^{n} x_{ij} w_j)^2 + \lambda \sum_{j=0}^{n} w_j^2$

regularization term
prevents weights from becoming too high

L0 norm = total number of non-zero vector elements
L1 norm = sum of the magnitudes of vectors in a space
L2 norm = sum of shortest distances $\|x\|_2 = \sqrt{(|3|^2 + |4|^2)} = \sqrt{9+16} = 5$

⇒ Regularization will save RAM and may reduce noise due to the possible dropout of complexity
⇒ L_1 Regularization for maximum sparsity, since it encourages features weights to drop to exactly 0.0

5.3 Dropout Regularization: used in neural networks

- removing a random selection of a fixed number of the units in a network layer for a single gradient step.
  ↳ the more units dropped out the stronger the regularization (don't make the model depend too much on a single unit)

5.4 Early stopping regularization

- ending model training before training loss finishes decreasing
  ⇒ end the training when the loss on validation dataset increases

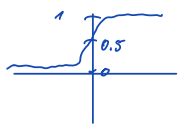
5.5 Learning Rate and lambda (regularization rate)

↗ if λ is too high = underfit
↘ if λ is too small = overfit

- tweaking learning rate and λ may have confounding effects
- strong L_2 regularization tends to drive values closer to 0
- Low learning rates often produce the same effect because the steps away from 0 aren't as large
  ⇒ tweaking learning rate and lambda simultaneously may have confounding effects

5.6 Logistic Regression

$\frac{e^x}{e^x+1}$



⇒ returns a probability between $[0,1]$ by plugging the result in the sigmoid function
⇒ result can be used as is or can be converted to binary category
⇒ is simple and scales well to large data sets

1. Calculate output of logistic regression model = $z = b + w_1x_1 + w_2x_2 + \dots + w_nx_n$
2. Plug into sigmoid. $\frac{1}{1+e^{-z}}$

Loss function is log loss: $\sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$
(data set (feature, label); y' = predicted value)

⇒ Logistic regression needs regularization (asymptotic nature of logistic regression drives loss to zero) ⇒ early stopping, L_2 regularization

5.7 Classification: distinguish between two or more discrete classes

- Can use, for example the probability output from Logistic regression for classification
  ↳ Set a classification threshold (all values above indicate one class, all values below indicate one class) ⇒ ROC curve

Confusion matrix:

Actual class \ Predicted class	+	−
+	TP: True Positive (correctly predict positive)	FN: False Negative (wrongly predict negative)
−	FP: False Positive (wrongly predict positive)	TN: True Negative: (correctly predict negative)

```
from sklearn.metrics import confusion_matrix
matrix = confusion_matrix(testlabels, predictions)
classification_report(testlabels, prediction of testtrain)
```

Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$ — how many are correct. Performance of model (is not enough alone?)

Precision: $\frac{TP}{TP+FP}$ — How accurate are positives. (if we said positive how often were we correct) how many of our positives are actually correct

Recall / Sensitivity / True positive rate: $\frac{TP}{TP+FN}$ — How many of all possible did we recall

Tug of war

Specificity: $\frac{TN}{TN+FP}$ — How many negatives did we recall

F1-Score: $\frac{2TP}{2TP+FP+FN}$: Hybrid metric for unbalanced classes (The higher the better, conveys balance between precision and recall)

Recall/TPR

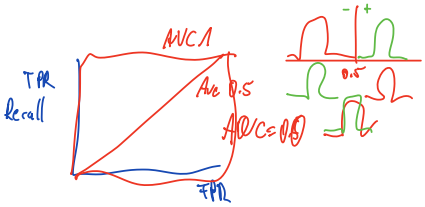
$2 \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$ if either one is low, it will result in low f-score

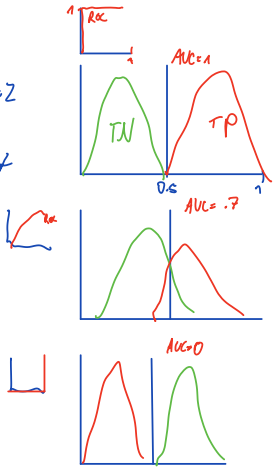
5.7.2 Receiver Operating Curve (ROC)

- used to find the threshold for classification
- plots recall vs. False positive rate (1 - specificity)
  "false alarm rate" (False positives / all negatives) how often false alarm

5.7.2 AUC (Area under Curve)

- probability that a model ranks a positive examples more highly than negative example (0.0 ← → 1.0)
- measures seperability: the higher the AUC the better the model is at predicting 1=1 and 2=2
  - AUC = 0.5 ⇒ model has no seperability
  - AUC = 0 the model is inverting the results ⇒ predictions are 100% incorrect





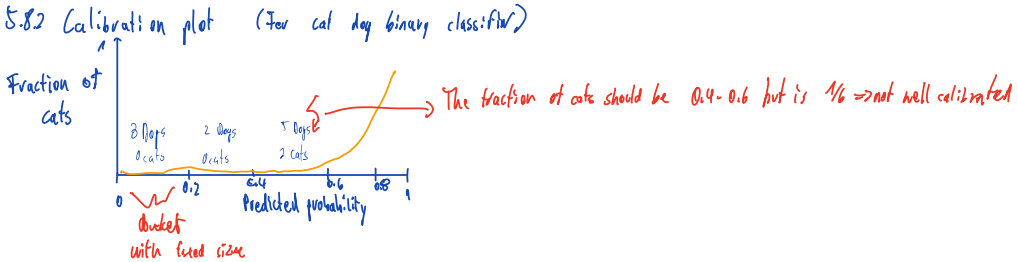
5.8 Prediction bias

(the inability for a ML method to capture the true relationship is bias)

- Logistic regressions should be unbiased. Average of predictions = Average of observations
- Prediction bias is a quantity that measures how far apart those to averages are
  prediction bias = average of predictions - average of labels

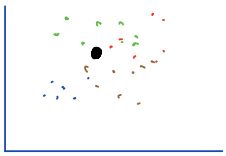
→ significant nonzero prediction bias tells you there is a bug somewhere in your model
⇒ zero prediction bias does not imply great model (Model that predicts mean value for all predictions has zero bias)

5.8.2 Calibration plot (For cat dog binary classifier)



6. Classification (algorithms)

⇒ more common than regression problems, since they make a decision
- logistic regression, KNN, SVM,



6.1 K-Nearest Neighbor (KNN)

- output is a class membership based on k-nearest neighbors
- we consider the k-nearest neighbors, they all get a vote towards the class (based on clustered data) (Lp norms for "nearness"

1. Load the data and cluster it
2. Initialise value of k
3. - Calculate distance between test data and each row of training data
   - Sort calculated distances in ascending order based on distance values
   - Get top k rows from sorted array
   - Get most frequent class of these rows
   - Return predicted class

manhattan
euclidean
chebyshev
cosine ⇒ similar

uncorrelated ⇒ opposite ⇒ similarity

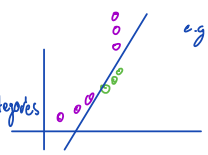
Python experiment with k:

```
error_rate = []
for i in range(1,40):
    knn = KNeighbours Classifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    pred = knn.predict(X_test)
    error_rate.append(np.mean(pred != y_test))
error_rate
```

6.2 Support vector machines

- Binary classifier
-

⇒ figure out maximal margin ⇒ builds a model with the wide as-possible gap between two categories



- if data is so that maximum margin cannot be used: ⇒ move data to higher dimension

a subspace with a dimension 1 less than its ambient space

⇒ the optimal hyperplane maximizes the distance between it and the closest data points (margin)
⇒ the points closest to the hyperplane are called support vectors. ⇒ max margin classifier
⇒ Hyperplane has dimension #features -1
⇒ large margin makes no less certainty decisions, slight error in measurement or document variation will not cause misclassification

Maximize margin, since the flatter the separator, the fewer choices there are can be put.

6.3 Linear discriminant analysis

- is intended for two-class problems
- classable will well separated or with few examples

- Addresses limitations of logistic regressions:

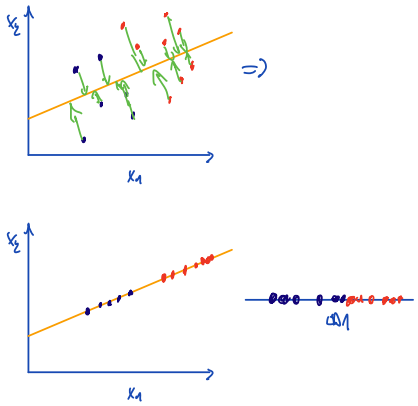
- goal remove dimensionality through linear transformation ⇒ superior for multi-class, but can use both in combination
- PCA can be described as an unsupervised algorithm, while LDA is supervised and computes the directions ("linear discriminants") that will represent the axes that maximize separation between multiple classes.

- Remove redundant and dependent features by changing the dataset onto a lower-dimensional space (reduce variables/features that have correlation with one another)

- Curse of dimensionality: - As you add more features, the model's performance improves on the training set until an optimal number of features is reached, after that it doesn't decline anymore
  => Because of overfitting

  - Refers to various phenomena that arise when analyzing data in high-dimensional spaces

  - In ML, the difficulties related to training models due to high dimensional dat is referred to as the curse of dimensionality



- Assumes categorical classification
- Assumes Gaussian
- Remove outliers
- Same variance across all input variables

8. Natural language processing

8.1 Motivation: Apart from time series data, most finance data is unstructured, in text format and language specific
→ Financial reports, earnings calls, news stories, analyst reports, excel spreadsheets

8.2 Difficulties:

- Many languages with different rules
- Unstructured data
- Context
- Written language has mistakes
- abbreviations
- colloquilisms

8.3 Definition

Natural language processing refers to the study and development of computer systems that can interpret speech and text as humans naturally speak and type it. (⇒ human communication is extremely vague at times with colloquialisms, abbreviations, etc)

8.4 Three aspects

- Semantic information: the specific meaning of an individual word (e.g. bat)
- Syntax information: sentence or phrase structure (subject, object word ...), the type
- Context information: context a word/phrase or sentence appears is in e.g. sick = cool ≠ unhealthy

} Need to combine all three at once to understand

⇒ Rules based : takes a lot of time, hard to generalize
ML : not a lot of labeled data
⇒ Machine learning + some rules

8.5 NLP use cases

- Language translation, - Sentiment analysis (the most sophisticated, varies wildly with context) (classify emotions and opinions as positive, negative or neutral)
- Text extraction: enables you to pull out pre-defined information from text
- Topics classification: organize unstructured texts into topics
- chatbots

In finance:
- Automation: e.g. automating capture of earnings calls, presentations, announcements
- Data enrichment: flag documents (like earnings call) with metadata to make stuff easier to find (e.g. where do they talk about environment?)
- Risk assessment: Assess credit risk by extracting loan data
- Financial sentiment: See if/how the market will react to news e.g. FinBERT (Hayn)
  ↳ Train machine learning model on pre-scored data to understand what a phrase/word means in each context
- Portfolio Selection
- Stock behavior predictions

8.6 Techniques

1. Syntactic analysis: identify sentence structure using basic grammar rules, see how words relate to each other
2. Semantic analysis: First, learn what every word means (lexical semantics), then, look at combination of words and what they mean in context

1.1 Tokenization: Break text document into pieces that machine can understand (= words). English is easy, just break up in white space, but for language without whitespace, for example, we use machine learning for tokenization.

1.2 PoS (Part of speech) Tagging: label tokens as verb, adverb, noun, etc. ⇒ helps infer meaning, since e.g. "book" means different things as noun or verb.

lemma = dictionary form
stem = unchanged word, but cut off ending

1.3 Lemmatization & stemming: consist of reducing inflected words to their base form to make them easier to analyse (remove end, aiming to arrive at base or dictionary form of word = lemma)

1.4 Stop word removal: remove frequently occuring words that don't add any semantic value such the, a, an, so, what

2.1 Words sense disambiguation: identify in which sense a word is being used in a given context.
2.2 Relationship extraction: understand how entities relate to each other in a text.
2.3 Named entity recognition: Identify people, places, things (products) mentioned in a text document
2.4 Sparsity Challenge: Text data (the vocabulary can have hundreds of thousands of dimensions but tends to be very sparse, every tweet has just a few dozen)

- Each text is transformed into a numerical representation as form of a vector.
- One of the most common is bag of words where a vector represents the frequency of a word in a predefined dictionary of words.

Then, a machine learning algorithm is fed with training data that consits of pairs of features sets (vectors for each each text) and tags (e.g. sports, politics) to produce a classification model)
→ Naive Bayes, SVM, Deep learning

2.7 Unsupervised ML for NLP

- Clustering (grouping similar documents together into groups, which are then sorted by relevance)
- Latent semantic indexing: identify words and phrases that frequently occur with each other
- Matrix Factorization: Break large matrix down into combination of two smaller matrices using latent factors

} tricky, but far less labor and data intensive than supervised

3.7 Categorization & Classification: sort contents in buckets to get a quick, high-level overview of what's in the data.
e.g. articles can be organized by topics, support tickets by urgency
assign tags to a documents, assign a set of pre-defined categories to open-ended texts

⇒ Since 80 % of information is unstructured, companies can automatically structure texts fast and cleanly

Why use ML? - Consistent criteria (humans are subjective, and get bored, tired etc.) → manual classification with human annotator ⇒ good results but time consuming and expensive
- Scalability: manually annotating is slow, less accurate and always "behind"
- Real time analysis

Automatic (as opposed to manual = human annotator):
- Rule based
- Machine learning based
- hybrid

- Rule based: classify text into organized groups by using a set of handcrafted linguistic rules. these rules instruct the system to use semantically relevant elements of a text to identify relevant categories based on its content ⇒ is human comprehensible and can be improved over time, BUT requires deep knowledge of domain, time-consuming to build rules, difficult to maintain

e.g. Want to classify into two groups: sports and politics ⇒ Have two big dictionaries of words for those topics.
⇒ Count number of words in text in the dictionaries and then classify.

-Machine learning based:

-Instead of relying on manually crafted rules, machine learning text classification learns to make classifications based on past observations
-By using pre-labeled examples as training data, machine learning algorithms can learn the different associations between pieces of text, and that a particular output (the tags?) is expected for a particular input (text)
→ The tag is the pre-determined classification or category that any given text could fall into.

Text feature extraction: transform text into numerical representation in the form of a vector.

-This is often done using bag of words technique: a vector represents the frequency of a word in a predefined dictionary of words.

→ Then, the model is fed with training data, that consists of pairs of feature sets (vectors for each text example) and tags (e.g. sport, politics) to produce a classification model.

-Algorithms used: Naive Bayes, Support Vector machines, Deep learning

FinBERT Sentiment Analysis

-biggest challenge in NLP is the shortage of training data
⇒ researchers have found pre-training using enormous amount of unannotated text on the web
This pre-trained model can then be fine-tuned on small data NLP tasks like question answering sentiment.

BERT: Bidirectional Encoder Representations from Transformers = BERT

it is deeply bidirectional, pre-trained using a plain text corpus (Wikipedia)

-Pre-trained representations can either be context-free or contextual, and contextual representations can further be unidirectional or bidirectional

- Context-free models generate a single word embedding representation for each word in the vocabulary.
⇒ bank has the same context-free representation in bank account and bank of river

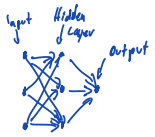
-Contextual models instead generate a representation of each word that is based on the other words in the sentence. For example: In the sentence "I accessed the bank account", a unidirectional contextual model would represent "bank" based on "accessed" but not "account".

⇒ BERT is bidirectional and represents "bank" using both its previous and next context - "I accessed the... account" starting from the bottom of a deep neural network, making it deeply bidirectional.

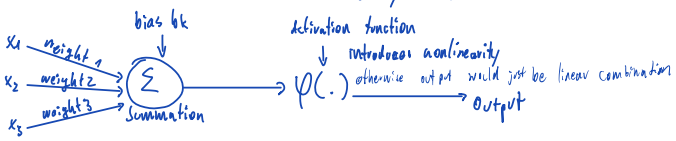
finBERT = pre trained NLP model based on BERT trained for financial data
(BERT is not really trained with finance specific jargon)

3 Artificial Neural Networks (ANN)

Batch size: How many features are processed at once?
Epoch: When all features are processed once = N features / batch-size
Iterations: A forward-pass and a backpropagation is completed for every batch



Neuron:



3.1 Activation Functions

- To model a nonlinear problem, we can directly introduce a nonlinearity
- We can pipe each hidden layer node through a nonlinear function (a.k.a. activation function that determines the weight of the vote from the neuron)
  Common activation functions: Sigmoid, ReLU, Tanh

Deep Learning: mimics the neurons of the human brain with multiple hidden layers

3.2 Terminology:

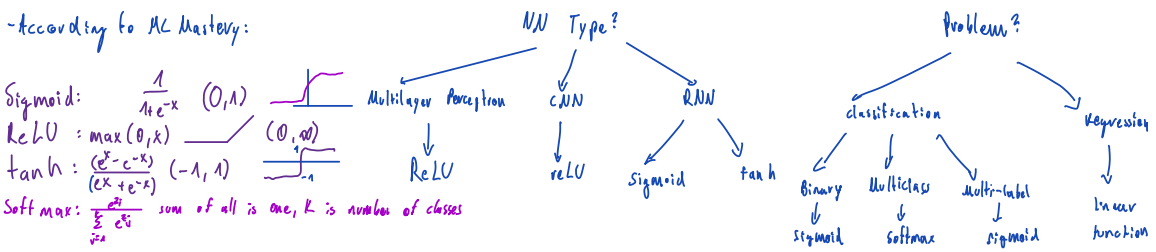
- A set of nodes, analogous to neurons organized in layers
- A set of weights, representing the connections between each neural network layer and the layer beneath it
- A set of biases for each node, similar to constant in a linear function
- An activation function that transforms the output of each node in a layer. Different layers may have different activation functions
- Input: the set of features that are fed into the model for the learning process (e.g. pixel values for an image)
- Weight: gives importance to those features that contribute more towards learning. It does so by introducing scalar multiplication between the input value and the weight matrix (decides how much influence input has on output)
- Bias: The role of the bias is to shift the value produced by the activation function. (=constant in linear function)
- Transfer function: Combine multiple inputs into one output value, so that activation function can be applied, (summation)
- Activation Function: It introduces non-linearity in the working of perceptrons to consider varying of all vals
  e.g. sigmoid, tanh... → $\sigma(\underbrace{w}_{\text{weight}} x + \underbrace{b}_{\text{bias}})$ linearity with the inputs. Without this, the output would just be a linear combination of input values and would not be able to introduce non-linearity to the network

- Hidden Layers: Enable deep learning, take the input from the previous layer to detect more specific features.
- Output Layer: Takes the input from the preceding hidden layers and comes to a final prediction based on the model's learnings.
- Input Layer: The data that need feed into the model (the features) is loaded into the input layer, it is the only visible layer that passes the complete information from the outside world without any computation

3.3 Choosing the right activation function:

- Usually the activation function is the same throughout a network model
- Until 1990 the sigmoid was popular, then till 2010 the tanh, then the reLU now (modern)
- Trial and error
- According to ML Mastery:

Sigmoid: $\frac{1}{1+e^{-x}}$ $(0,1)$
ReLU: $\max(0,x)$ $(0,\infty)$
tanh: $\frac{(e^x - e^{-x})}{(e^x + e^{-x})}$ $(-1,1)$
Soft max: $\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ sum of all is one, K is number of classes



⇒ If problem is a regression problem, then you should use a linear activation function
⇒ If classification/predicting a probability: Binary: Sigmoid Multi-class: Softmax Multi-Label: Sigmoid
↓ rounding ↓ argmax

3.4 Back propagation

- The primary algorithm for performing gradient descent on neural networks
- Adjust each weight in the network in proportion to how much it contributes to the error
- First, the output values of each node a calculated and cached in a forward pass.
- Then, the partial derivative of the error with respect to each parameter is calculated in a backward pass through the graph

Failure Scenarios:

- If the weighted sum for a ReLU unit falls below 0, the unit can get stuck since it outputs 0 activation, contributing nothing to the network's output, and gradients cannot improve it during backpropagation
  ⇒ Lowering learning rate helps

Dropout
- Randomly drops out unit activations in a network for a single gradient step. The higher the drop out, the stronger the regularization (0.0 dropout = no regularization, 1.0 dropout = drop out everything)

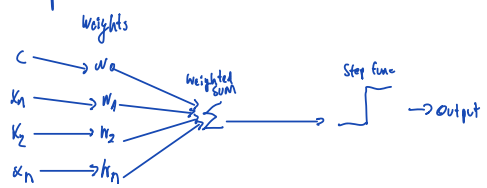
- Vanishing gradient: Due to chaining, the gradients for the lower layers can become very small (gradient gets smaller and smaller as we iterate backwards due to product)
  ⇒ low layers train very slowly, or not at all
  ⇒ ReLU function helps to prevent vanishing gradients

- Exploding gradients: If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case, you can have exploding gradients: gradients that get too large to converge.
  ⇒ Fix: Batch Normalization, lowering learning rate

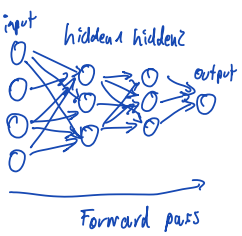
3.5 Types of Neural Networks

- Standard: Perceptron, Feed-Forward Network, Residual Network Comonly used for simple structured regression and classification
- RNN: Recurrent RNN, Long short-term memory (LSTM), Echo State Networks (sequential data)
- CNN: Convolutional neural networks (For classification of images and videos)
- GAN: Generative Adversarial Network (generate data)
- Transformer neural networks (Sequential data, better than RNN) ⇒ parallelize training using encoder-decoder structure ⇒ feed complete sentence together
  ⇒ series of perceptrons, stacked in a row and piled in different layers

Perceptron:



Feed-Forward NN



3.6 Recurrent Neural Networks

- Recurrent neural networks have the power to remember what it has learned in the past and apply it in future predictions
- The input is in the form of sequential data that is fed into the RNN, which has a hidden internal state that gets updated every time it reads the following sequence of data in the input.
- This internal hidden state will be fed back to the model.

⇒ solve NLP problems, time series

- They have loops in them, allowing information to persist
- Can be thought of, as multiple copies of the same network, each passing a message to the successor

3.7 LSTM

RNN Problem: Sometimes we only need to look at recent information to perform the present task. Then, if the gap between the relevant information and the place that it's needed is small, RNNs can learn to use past information.

However, if the gap is longer and we need more context (or just have more stuff in between) RNNs are unable to learn and connect the information. (In theory they are, in practice it doesn't work)

- Training RNNs
- The vanishing or exploding gradient problem
- RNNs cannot be stacked up
- Slow and complex learning procedures
- Difficult to process longer sequences

⇒ LSTMs

Long short-term memory networks are able to learn long term dependencies (special type of RNN with similar base (chain))

- In vanilla RNNs the repeating module will have a very simple structure such as a single tanh layer
- The repeating module in LSTM contains four interacting layers, interacting in a very special way

- The key to LSTMs is the cell state, the LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates

1. Forget gate

the forget gate looks at h_{t-1} (previous hidden state) and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 therefore represents completely keep this, completely get rid of this.

→ we get the next value (e.g. a word) and check which elements in the long term state are still relevant, if the values (or "words") in the long-term "cell" state are not relevant anymore given the next value x_t, $[h_{t-1}, x_t]$ is 0 and therefore it will get rid of this.

2. Remembering

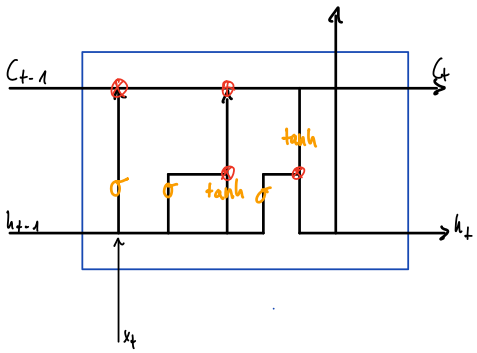
- First, a sigmoid-layer called the input gate layer decides which values to update
- Next, a tanh layer creates a vector of new candidate values $\tilde{C}_t$ that could be added to the next state
- Then these two are combined and added to the cell state.

which values we update ↓ , ↓ new candidate values

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

3. Outputting

- Output will be based on cell state, but a filtered version
- First, we run a sigmoid layer, that decides what parts of the cell state we are going to output
- Then we push the values through tanh (-1,1) and multiply with output from sigmoid, so we only output what we want to as a hidden state



9.8 Variants

1. Adding "peephole connections". This means that we let the gate layers look at the cell state.
2. Use coupled forget and input gates: Make forget and cell input decision together, we only forget if we want to put something in its place
3. GRU (Gated Recurrent Unit): Combine forget an input into single update gate and merges cell state and hidden state ⇒ less complex

3.10 Convolutional neural networks (CNNs)

1. Convolution
2. Non Linearity (RELU)
3. Pooling or Sub Sampling
4. Classification

1. Convolution (Element wise matrix addition and multiplication)

- Main purpose: Extract features from input image (from e.g. RGB values)
  Sliding filter/kernel moves over the matrix and multiplies its fixed values with the image.
  Resulting in a matrix reduced to the filter size. (= Feature Map/Convolved Feature)

- Different fixed values of the filter matrix will produce different feature maps for the same input image
  ⇒ different filters detect different features from an image (e.g. edges, curves, blur, identity)

⇒ These filters are not predefined. The values of each filter is learned during training process, randomly initialized at start, so that each filter learns to identify different features.
- The number of filters, the filter size etc. has to be specified
- The more filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

Depth: The number of filters we use for the convolution operation
Stride: Stride is the number of pixels by which we slide our filter matrix over the input matrix
⇒ The larger the stride, the smaller the feature map
Zero padding: pad input matrix with zeros around the border, to apply the filter to bordering elements of our input matrix. (wide ll narrow convolution)

2. ReLU step

Apply the ReLU function onto all feature maps (=rectified feature map) to introduce non-linearity and replace all negative values by 0.

3. Pooling

Spatial pooling reduces the dimensionality of the kernel matrix while retaining the most important information
Overlay a sliding window (e.g.) 2x2 over feature map and take the largest/average/sum of all elements within that window. Usually we use max-pooling

- makes the input representations (feature dimension) smaller and more manageable
- reduces the number of parameters and computations in the network, and therefore controlling overfitting
- makes the network invariant to small transformations, distortions and translations to the input image
  (a small distortion in input will not change the output of pooling, since we take the maximum/average value in a local neighborhood)

- helps us to arrive at an almost scale invariant representation of our image (the exact term is equivariant). This is very powerful, since we can detect objects in an image no matter where they are located.

4. Fully Connected Layer

- The fully connected layer is a traditional Multi Layer Perceptron that uses a softmax activation function in the output layer (each neuron in the previous layer is connected to each neuron in the next layer)
- The output from the convolutional and pooling layers represent high-level features of the input image, the fully connected layer uses these features for classification
- Also adds non-linear combinations of features
- Sum of the output from softmax in last layer is 1

Steps: 1. Initialize all filters and parameters/weights with random values

2. The network takes a training image as input, goes through convolution, ReLU, pooling, forward pass and finds probabilities for each class

   - In the first iteration, all these output values are random, since all weights randomly assigned for the first example are also random

3. Calculate the total error of the output layer $\sum \frac{1}{2}(\text{target probability} - \text{output probability})^2$

4. Use backpropagation to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values/weights and parameter values to minimize the output error

5. Repeat

6. When a new, unseen image is now presented as input to the CNN, the network goes to the entire process once and output a probability for each class

- It is usual to multiple convolution and pooling layers
- In general, the more convolution steps we have, the more complicated features our network will be able to recognise

11. GANs : Generative Adversarial Networks

- GANs create new data instances that resemble your training data.
- they achieve this by pairing two Neural Nets, a generator which learns to produce the target output, and a discriminator, which learns to distinguish true data from the output of the generator

⇒ the generator tries to fool the discriminator and the discriminator tries to keep from being fooled

- Generative Models can generate new data instances (like photos of animals/people that look like real animals)
- Discriminative Models discriminate between different data instances (could tell a dog from a cat)
- Given a data set of features X and labels Y: - Generative models capture the joint probability $p(X, Y)$
  - Discriminative models capture the conditional probability $P(Y|X)$

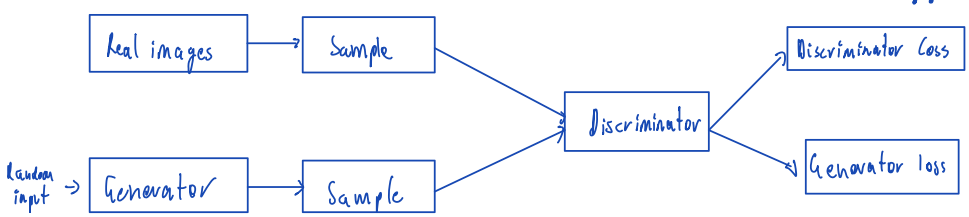
- Neither the discriminator, nor the generator have to return a numerical probability.

- Generative Models are Harder: they tackle more difficult tasks than the discriminator since they have to model more, they have to capture correlations like "things that look like boats are probably going to appear near things that look like water" and "eyes are unlikely to appear on foreheads"

⇒ Discriminative models draw boundaries in the data space, generative models try to model how data is placed throughout the space. They have to model the distribution throughout the space and try to fall close to the real counterparts in the data space

Discriminative: 

Generative 



- The generator learns to generate plausible data. The generated instances become negative training examples for the discriminator.

- The discriminator learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.
- Both are neural networks, the generators output is connected directly to the discriminator input

11.2 Training the discriminator

- the training data comes from two instances: Real data instances as positive examples, and fake data from the generator as negative examples.

⇒ during discriminator training the generator does not train and vice versa
- the discriminator classifies both real, and fake data from the generator
- the discriminator loss penalizes the discriminator for misclassifying a real instance as fake or fake as real
- the discriminator updates its weights through back propagation from the discriminator loss

11.3 Generator Training

The generator part of GAN learns to create fake data by incorporating feedback from the discriminator, it learns to make the discriminator classify its output as real.

penalizes the generator for failing to fool the discriminator

Random input → Generator → Sample → Discriminator → Generator Loss

11.3.1 Random input: In the easiest form, GANs just use random noise as its input. The generator then transforms this noise into a meaningful output. We sample from different places in the target distribution. Distribution of the noise doesn't matter too much, but we can choose something that is easy to sample from, like a uniform distribution.

11.3.2 Training

Usually, we alter the net's weights to reduce the error or loss of its output.
In our GAN, however, the generator is not directly connected to the loss we are trying to affect
⇒ Back propagation starts at the output, and flows back through the discriminator into the generator.
⇒ We do not want the discriminator to change during generator training, hard to hit a moving target

1. Sample random noise
2. produce generator output from sampled random noise
3. Get discriminator "real" or "fake" classification for generator output.
4. Calculate loss from discriminator classification
5. Backpropagate through both the discriminator and generator to obtain gradients
6. Use gradients to change only the generator weights

In conclusion, 2 complications
- GANs must juggle 2 different kinds of training (discriminator and generator)
- GAN convergence is hard to identify.

⇒ Alternate training:

How to train the GAN as a whole: keep the generator constant during the discriminator training phase AND THEN, keep the discriminator constant during the generator training phase

⇒ GAN training proceeds in 2 alternating periods:

1. The discriminator trains for one or more epochs
2. The generator trains for one or more epochs
3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks

11.3.3 Fleeting Convergence

As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake ⇒ if the generator succeeds, then the discriminator has an accuracy of 50%
⇒ the discriminator gets less meaningful over time, if the GAN continues training past the point where the discriminator is giving random feedback, the generator starts to train on junk feedback and its quality may collapse
⇒ For GANs, convergence is often a fleeting, rather than stable, state.

11.3.4 Loss functions

Both derive from 1 loss function

A GAN can have 2 loss functions: one for generator training, and one for discriminator training
⇒ Often the generator and discriminator losses derive from a single measure of distance between probability distributions
However, the generator can only affect one term in the distance measure: the one that reflects the fake data
⇒ during generator training we drop the other term, which reflects the distribution of the real data

GANs try to replicate a probability distribution: They should therefore use loss functions that reflect the difference between the distribution of the data generated by GAN and the distribution of real data.
⇒ MiniMax Loss, Wasserstein Loss

discriminators estimate that real is real

D's estimate that fake is real

11.3.5 Minimax Loss: $E_x(\log(D(x))) - E_z(\log(1 - D(G(z))))$ ⇒ can get stuck in the beginning, therefore the paper suggests modifying generator loss to maximize $\log(D(G(z)))$

11.3.6 Wasserstein Loss: depends on modification of GAN scheme ("called WGAN or Wasserstein GAN) in which the discriminator does not classify instances
- The discriminator just outputs a number, which it tries to make bigger for real instances than for fake instances.
- Because it cannot really discriminate between real and fake instances, the discriminator is called a critic

Critic Loss: $D(x) - D(G(z))$: The critic tries to maximize the difference between its output on fake and real instances

Generator Loss: $D(G(z))$: the generator tries to maximize the discriminator's output for the fake instances

11.3.7 Common Problems of GAN:

Vanishing Gradients: Research has suggested that if your discriminator is too good, the generator training can fail due to vanishing gradients. An optimal discriminator doesn't provide enough information for the model to make progress

Mode Collapse:
- Usually you want the GAN to produce a wide variety of outputs
- However, if the generator produces an especially plausible output, the generator may learn to produce only that output. In fact, the generator is always trying to find the one output that is most plausible to the discriminator.

- If the generator starts to produce the same output over and over again, the discriminators best strategy is to reject that outputs. But if the next generation discriminator gets stuck in a local minimum and doesn't find the optimal strategy, its easy for the next generator iteration to find the most plausible output for the current discriminator

⇒ each iteration of generators over optimize for a particular discriminator, and the discriminator never manages to learn its way out of the trap ⇒ As a result, the generators rotate through a small set of output type

12. Clustering

- Unsupervised learning is a machine learning approach in which models do not have any supervisor to guide them.
- Models themselves find the hidden patterns and insights from the provided data
- similar to how a human learns
- We have a set of features for each observed examples, but no labels

Basic idea of clustering: - find groups of data in dataset that are similar to one another - what we call clusters
- it is the process of dividing the entire data into groups (also known as clusters) based on patterns in the data
=> clustering is an unsupervised learning problem, labeled data is not available or needed

12.1 Use cases:

- Customer segmentation
- Document clustering: helps group similar documents
- Image segmentation: club similar pixels in the image together
- Recommendation Engines: Clustering to find similar songs like friends and then recommend

12.2 Bank Credit Card Offers Example: Instead of looking at each data point individually, segment customers into different groups (e.g. high income, average income, low income)

12.3 Clusters

- Properties: All data points in a cluster should be similar to each other
All dat points in different clusters should be as different as possible

- Evaluation Metrics for Clusters:

1. Inertia: - tells us how far the points within a cluster are by calculating the sum of distances of all points within a cluster from the centroid of that cluster.
- gives us the sum of intracluster differences
- the lower the better

2. Dunn index: - takes into account the distance between two clusters
- inter cluster distance: distance between two clusters

$$= \frac{\min(\text{inter cluster distance})}{\min(\text{intra cluster difference}} \Rightarrow \text{the higher the better}$$

11.22 Clustering Algorithms:

K-means, Mean Shift, Agglomerative Hierarchical, DBSCAN Algorithm, Expectation Maximization Clustering using Gaussian Mixture Models

⇒ can also be used in supervised learning by clustering the data into similar groups and using these cluster labels as independent variables in the supervised machine learning algorithm.

K-means: Minimize the distance of the points within a cluster, each cluster is associated with a centroid

1. Choose the number of clusters k, e.g. k=2
2. Randomly select data point as centroid for each cluster
3. Assign all the points to their closest centroid
4. Recompute the centroids of newly formed clusters
6. Repeat 3. and 4.

Stopping criteria:
- Centroids of newly formed clusters do not change
- Points remain in the same clusters
- Maximum number of iterations reached

Challenges of K-means:
- If the size of the clusters is different
- if the densities of the original points are different

⇒ Possible fix: Perhaps using more clusters can help, but computationally expensive

⇒ K-Means++: It is problematic, that we assign random centroids in the beginning because we might get different clusters each time.
⇒ K-Means++ can be use to choose the initial values, or the initial cluster centroids

1. Choose the first cluster at random
2. Compute the distance of each datapoint x D(x) from the cluster center that has been chosen in 1.
3. Then, choose the next centroid so it will be the one whose squared distance $D(x)^2$ is the farthest
4. Repeat two and 3 until k-clusters have been chosen

⇒ Continue with k-means (is shown to converge more rapidly)

Choose the right number of clusters:

Plot a graph, x-axis presents the number of clusters and y-axis is an evaluation metric, like Inertia or Dunn index

13. Reinforcement Learning

13.1 Reinforcement learning is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones. In general, a reinforcement learning agent is able to perceive and interpret its environment, take actions and learn through trial and error.

13.2 Markov Decision Process (MDP)

- Gives us a way to formalize sequential decision making. Is the basis for structuring problems that are solved with reinforcement learning.

- Components:

  - Agent: Decision Maker
  - Environment
  - State
  - Action
  - Reward

AESAR, CÄSAR without C

At each time step, the agent receives a representation of the environment's state. Based on this state, the agent selects an action. (S_t, A_t)

$\Rightarrow$ Trajectory: $S_0 A_0 R_1 S_1 A_1 R_2 S_2$
SAR - SAR

Next time step, the agent receives a numerical reward and the environment transitions to a new state.

13.3 Transition probabilities:

For all $s' \in S$, $s \in S$, $r \in R$, $a \in A$, we define the probability of the transition to the state s' with reward r from taking action a in state s as:

$$p(s', r \mid s, a) = \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

$\Rightarrow$ It is the agents goal to maximize the expected return of rewards

The expected return is the sum of future rewards (here, without discounting): $G_t = R_{t+1} + R_{t+2} + \dots + R_T$

13.4 Episodic Tasks

- Intuitively: An episode is until "the game" is over
- The agent-environment interaction breaks up into subsequences, called episodes
- Each episode ends in an terminal state, at time T, which is followed by resetting the environment to some standard starting state
- $\Rightarrow$ Tasks with episodes are called episodic tasks

13.5 Continuous Tasks

- If the agent-environment interactions don't break up naturally into episodes, but continue without limit, these tasks are called continuing tasks

⇒ Now the return has to be defined differently because at our final time step $T \to \infty$, the reward could be infinite due to infinite sum

⇒ Discount future rewards: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ (γ: discount factor, number between 0 and 1. Also makes us value today's return more.)

$= R_{t+1} + \gamma G_{t+1}$

If reward constant 1 and $\gamma < 1$ then $G_t = \frac{1}{1-\gamma}$ = Taylor expansion

13.6 Policies and Value Functions:

How probable is it for an agent to select any given action from a given state ⇒ Policies
How good is any given action or any given state for an agent ⇒ value function

A policy π is a function that maps a given state to probabilities of selecting each possible action from that state.

States → state-value function

(state, action) → action-value function

- The state-value function returns the value of a state under π. The value of state s under policy π is the expected return from starting from state s at time t and following policy π thereafter.

$$v_\pi(s) = E_\pi(G_t \mid S_t = s)$$
$$= E_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right)$$

- The value of action a in state s under policy π is the expected return from starting from state s at time t, taking action a and following policy π thereafter

Q-function

$$q_\pi(s,a) = E_\pi(G_t \mid S_t = s, A_t = a)$$
$$= E_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right)$$ ← Q-value

Optimal policy: $\pi \geq \pi'$ iff $v_\pi(s) \geq v_{\pi'}(s)$ for all s (expected return for starting in state s and then following π)
the best policy is the optimal policy

Optimal state value function: $v_*(s) = \max_\pi v_\pi(s)$

Optimal action value function: $q_*(s,a) = \max_\pi q_\pi(s,a)$

13.7 Bellman Optimality Equation

$$q_*(s,a) = E\left(R_{t+1} + \gamma \max_{a'} q_*(s',a')\right)$$

the reward we get from taking action a in state s, plus the maximum discounted return that can be achieved from any possible next state action pair

- max ((left, empty), (right, empty), ...)
  any possible state action of next state

13.8 Q-Learning:

Q-table: Stores the Q-value for each state action pair

First actions: trade off between exploration and exploitation

- Exploration is the act of exploring the environment to find out information about it
- Exploitation is the act of exploiting the information that is already known about the environment in order to maximize the return

=> explore so we do not miss out on possibly better strategies
=> use epsilon greedy strategy

Exploration rate is the probability that our agent will explore the environment rather than exploit it.
It is initially set to 1, certain that the agent will explore the environment.
As the agent learns more about the environment, at the start of each episode, ε will decay by some rate so that the likelihood of exploration becomes less and less probable.
To determine whether the agent chooses exploration or exploitation at each time step, we generate a random number between 0 and 1.

```
if random_number > epsilon ε
        choose action via exploitation
} else {
    choose action via exploration (also randomly)
}
```

Loss: $q_*(s,a) - q(s,a) = \text{loss}$

$$E\left(R_{t+1} + \gamma \max_{a'} q_*(s',a')\right) - E\left(\sum_{t=0}^{\infty} \gamma^t R_{t+t+1}\right)$$

The learning rate is a number between 0 and 1, which can be thought of as how quickly the agent abandons the previous Q-value in the Q-table for a given state-action pair for the new Q-value.

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\underbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}_{\text{learned value}}$$

Deep Q-Learning:

0. Initialize memory replay capacity
1. Initialize network with random weights
2. Clone the policy network and call it target network
3. For each episode:
    1. Initialize the starting state
    2. For each time step:
        1. Select an action
            Via exploration or exploitation
        2. Execute selected action in an emulator
        3. Observe reward and next state
        4. Store experience in replay memory
        5. Sample random batch from replay memory
        6. Preprocess states from batch
        7. Pass batch of preprocessed states to policy network
        8. Calculate loss between output Q-values and target q-values: Requires a pass to the target network for the next state
        9. Gradient descent updates weights in the policy network to minimize loss.
            ⇒ After x time steps, the weights in the target network are updated to the weights in the policy network

We store the agents experiences at each time step in a data set called the replay memory.
The agents experience at time t is defined as $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$

We then sample a minibatch from this experience replay memory and train on it.

⇒ By using a target network, we prevent the training process from spiraling around because we are fixing the the targets for multiple time steps, thus allowing the network weights to move consistently towards the target.