

Machine Learning Engineering



## Why Machine Learning?

- Search engines (e.g. Google)
- Recommender systems (e.g. Netflix)
- Automatic translation (e.g. Google Translate)
- Speech understanding (e.g. Siri, Alexa)
- Game playing (e.g. AlphaGo)
- Self-driving cars
- Personalized medicine
- Progress in all sciences: Genetics, astronomy, chemistry, neurology, physics,...

# Lecture 1: Introduction

A few useful things to know about machine learning

Joaquin Vanschoren

## What is Machine Learning?

- Learn to perform a task, based on experience (examples)  $X$ , minimizing error  $\mathcal{E}$ 
    - E.g. recognizing a person in an image as accurately as possible
  - Often, we want to learn a function (model)  $f$  with some model parameters  $\theta$  that produces the right output  $y$
- $$f_\theta(X) = y$$
- $$\underset{\theta}{\operatorname{argmin}} \mathcal{E}(f_\theta(X))$$
- Usually part of a *much* larger system that provides the data  $X$  in the right form
    - Data needs to be collected, cleaned, normalized, checked for data biases,...

## Inductive bias

- In practice, we have to put assumptions into the model: *inductive bias b*
  - What should the model look like?
    - Mimic human brain: Neural Networks
    - Logical combination of inputs: Decision trees, Linear models
    - Remember similar examples: Nearest Neighbors, SVMs
    - Probability distribution: Bayesian models
  - User-defined settings (hyperparameters)
    - E.g. depth of tree, network architecture
  - Assumptions about the data distribution, e.g.  $X \sim N(\mu, \sigma)$
- We can *transfer knowledge* from previous tasks:  $f_1, f_2, f_3, \dots \Rightarrow f_{new}$ 
  - Choose the right model, hyperparameters
  - Reuse previously learned values for model parameters  $\theta$
- In short:

$$\underset{\theta, b}{\operatorname{argmin}} \mathcal{E}(f_{\theta, b}(X))$$

## Machine learning vs Statistics

- See Breiman (2001): Statistical modelling: The two cultures

- Both aim to make predictions of natural phenomena:



- Statistics:

- Help humans understand the world

- Assume data is generated according to an understandable model

- Machine learning:

- Automate a task entirely (partially replace the human)

- Assume that the data generation process is unknown

- Engineering-oriented, less (too little?) mathematical theory



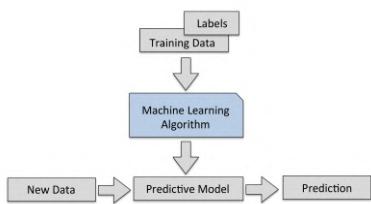
## Types of machine learning

- **Supervised Learning:** learn a model  $f$  from *labeled data*  $(X, y)$  (ground truth)
  - Given a new input  $X$ , predict the right output  $y$
  - Given examples of stars and galaxies, identify new objects in the sky
- **Unsupervised Learning:** explore the structure of the data ( $X$ ) to extract meaningful information
  - Given inputs  $X$ , find which ones are special, similar, anomalous, ...
- **Semi-Supervised Learning:** learn a model from (few) labeled and (many) unlabeled examples
  - Unlabeled examples add information about which new examples are likely to occur
- **Reinforcement Learning:** develop an agent that improves its performance based on interactions with the environment

Note: Practical ML systems can combine many types in one system.

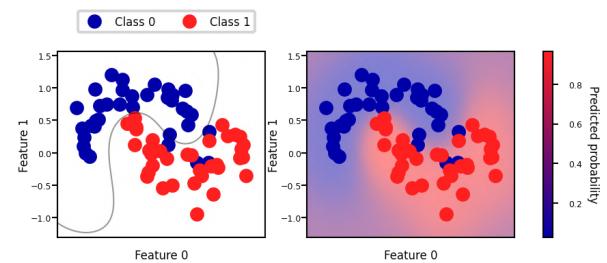
## Supervised Machine Learning

- Learn a model from labeled training data, then make predictions
- Supervised: we know the correct/desired outcome (label)
- Subtypes: *classification* (predict a class) and *regression* (predict a numeric value)
- Most supervised algorithms that we will see can do both



### Classification

- Predict a *class label* (category), discrete and unordered
  - Can be *binary* (e.g. spam/not spam) or *multi-class* (e.g. letter recognition)
  - Many classifiers can return a *confidence* per class
- The predictions of the model yield a *decision boundary* separating the classes



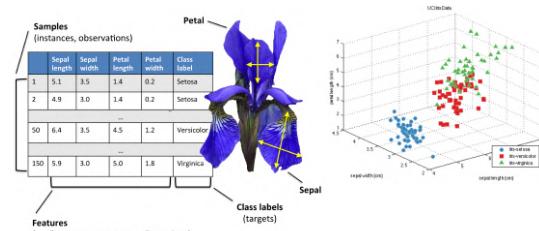
Example: Flower classification

Classify types of Iris flowers (setosa, versicolor, or virginica). How would you do it?



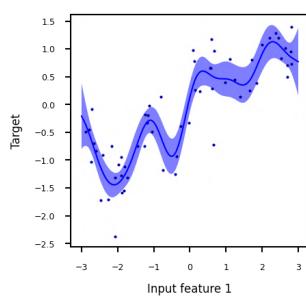
### Representation: input features and labels

- We could take pictures and use them (pixel values) as inputs (-> Deep Learning)
- We can manually define a number of input features (variables), e.g. length and width of leaves
- Every 'example' is a point in a (possibly high-dimensional) space



## Regression

- Predict a continuous value, e.g. temperature
  - Target variable is numeric
  - Some algorithms can return a *confidence interval*
- Find the relationship between predictors and the target.

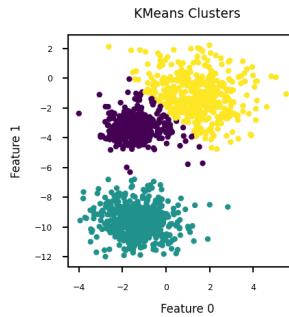


## Unsupervised Machine Learning

- Unlabeled data, or data with unknown structure
- Explore the structure of the data to extract information
- Many types, we'll just discuss two.

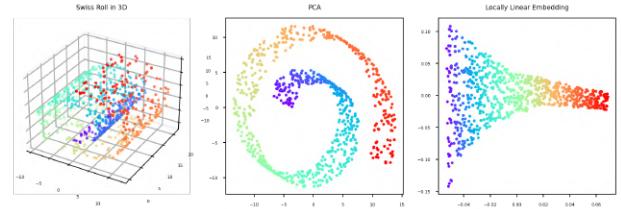
## Clustering

- Organize information into meaningful subgroups (clusters)
- Objects in cluster share certain degree of similarity (and dissimilarity to other clusters)
- Example: distinguish different types of customers



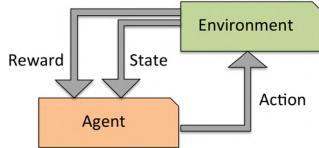
## Dimensionality reduction

- Data can be very high-dimensional and difficult to understand, learn from, store,...
- Dimensionality reduction can compress the data into fewer dimensions, while retaining most of the information
- Contrary to feature selection, the new features lose their (original) meaning
- The new representation can be a lot easier to model (and visualize)



## Reinforcement learning

- Develop an agent that improves its performance based on interactions with the environment
  - Example: games like Chess, Go,...
- Search a (large) space of actions and states
- Reward function defines how well a (series of) actions works
- Learn a series of actions (policy) that maximizes reward through exploration



## Learning = Representation + evaluation + optimization

All machine learning algorithms consist of 3 components:

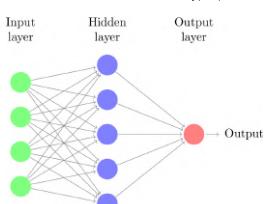
- Representation:** A model  $f_\theta$  must be represented in a formal language that the computer can handle
  - Defines the 'concepts' it can learn, the *hypothesis space*
  - E.g. a decision tree, neural network, set of annotated data points
- Evaluation:** An *internal* way to choose one hypothesis over the other
  - Objective function, scoring function, loss function  $\mathcal{L}(f_\theta)$
  - E.g. Difference between correct output and predictions
- Optimization:** An *efficient* way to search the hypothesis space
  - Start from simple hypothesis, extend (relax) if it doesn't fit the data
  - Start with initial set of model parameters, gradually refine them
  - Many methods, differing in speed of learning, number of optima,...

A powerful/flexible model is only useful if it can also be optimized efficiently

## Neural networks: representation

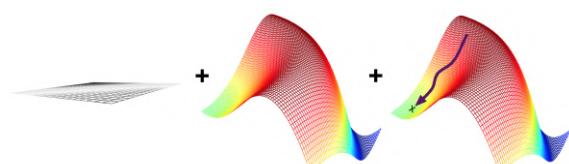
Let's take neural networks as an example

- Representation: (layered) neural network
  - Each connection has a *weight*  $\theta_i$  (a.k.a. model parameters)
  - Each node receives weighted inputs, emits new value
  - Model  $f$  returns the output of the last layer
- The architecture, number/type of neurons, etc. are fixed
  - We call these *hyperparameters* (set by user, fixed during training)

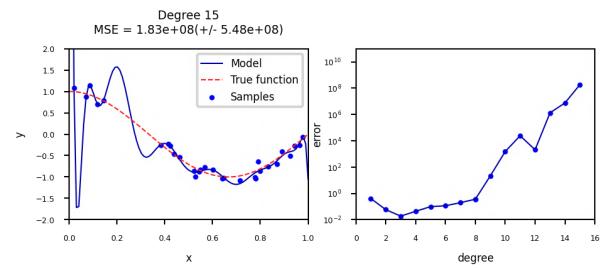


## Neural networks: evaluation and optimization

- Representation: Given the structure, the model is represented by its parameters
  - Imagine a mini-net with two weights  $(\theta_0, \theta_1)$ : a 2-dimensional search space
- Evaluation: A *loss function*  $\mathcal{L}(f_\theta)$  computes how good the predictions are
  - Estimated on a set of training data with the 'correct' predictions
  - We can't see the full surface, only evaluate specific sets of parameters
- Optimization: Find the optimal set of parameters
  - Usually a type of search in the hypothesis space
  - E.g. Gradient descent:  $\theta_i^{new} = \theta_i + \frac{\partial \mathcal{L}(f_\theta)}{\partial \theta_i}$



- There is often a sweet spot that you need to find by optimizing the choice of algorithms and hyperparameters, or using more data.
- Example: regression using polynomial functions



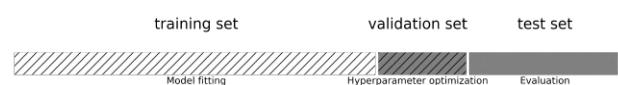
## Overfitting and Underfitting

- It's easy to build a complex model that is 100% accurate on the training data, but very bad on new data
- Overfitting: building a model that is too complex for the amount of data you have
  - You model peculiarities in your training data (noise, biases,...)
  - Solve by making model simpler (regularization), or getting more data
  - Most algorithms have hyperparameters that allow regularization
- Underfitting: building a model that is too simple given the complexity of the data
  - Use a more complex model
- There are techniques for detecting overfitting (e.g. bias-variance analysis). More about that later
- You can build ensembles of many models to overcome both underfitting and overfitting

## Model selection

- Next to the (internal) loss function, we need an (external) evaluation function
  - Feedback signal: are we actually learning the right thing?
    - Are we under/overfitting?
  - Carefully choose to fit the application.
  - Needed to select between models (and hyperparameter settings)

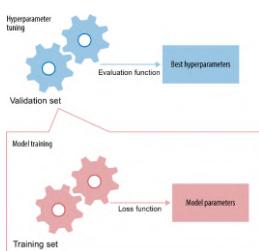
- Data needs to be split into training and test sets
  - Optimize model parameters on the training set, evaluate on independent test set
- Avoid data leakage:
  - Never optimize hyperparameter settings on the test data
  - Never choose preprocessing techniques based on the test data
- To optimize hyperparameters and preprocessing as well, set aside part of training set as a validation set
  - Keep test set hidden during all training



- For a given hyperparameter setting, learn the model parameters on training set
  - Minimize the loss
- Evaluate the trained model on the validation set
  - Tune the hyperparameters to maximize a certain metric (e.g. accuracy)

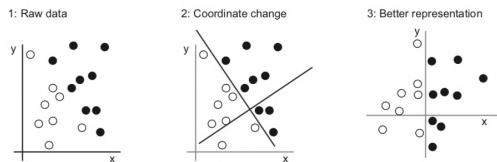
Only generalization counts!

- Never evaluate your final models on the training data, except for:
  - Tracking whether the optimizer converges (learning curves)
  - Diagnosing under/overfitting:
    - Low training and test score: underfitting
    - High training score, low test score: overfitting
- Always keep a completely independent test set
- On small datasets, use multiple train-test splits to avoid sampling bias
  - You could sample an 'easy' test set by accident
  - E.g. Use cross-validation (see later)



## Better data representations, better models

- Algorithm needs to correctly transform the inputs to the right outputs
- A lot depends on how we present the data to the algorithm
  - Transform data to better representation (a.k.a. *encoding* or *embedding*)
  - Can be done end-to-end (e.g. deep learning) or by first 'preprocessing' the data (e.g. feature selection/generation)



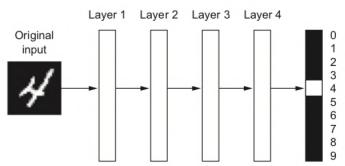
## Feature engineering

- Most machine learning techniques require humans to build a good representation of the data
- Especially when data is naturally structured (e.g. table with meaningful columns)
- Feature engineering is often still necessary to get the best results
  - Feature selection, dimensionality reduction, scaling, ...
  - Applied machine learning is basically feature engineering (Andrew Ng)*
- Nothing beats domain knowledge (when available) to get a good representation
  - E.g. Iris data: leaf length/width separate the classes well

Build prototypes early-on

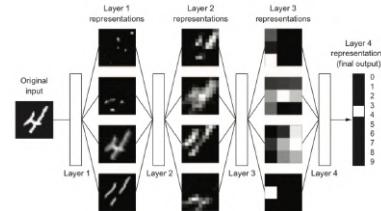
## Learning data transformations end-to-end

- For unstructured data (e.g. images, text), it's hard to extract good features
- Deep learning: learn your own representation (embedding) of the data
  - Through multiple layers of representation (e.g. layers of neurons)
  - Each layer transforms the data a bit, based on what reduces the error



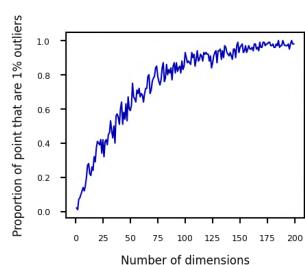
## Example: digit classification

- Input pixels go in, each layer transforms them to an increasingly informative representation for the given task
- Often less intuitive for humans



## Curse of dimensionality

- Just adding lots of features and letting the model figure it out doesn't work
- Our assumptions (inductive biases) often fail in high dimensions:
  - Randomly sample points in an n-dimensional space (e.g. a unit hypercube)
    - Almost all points become outliers at the edge of the space
    - Distances between any two points will become almost identical



## Practical consequences

- For every dimension (feature) you add, you need exponentially more data to avoid sparseness
- Affects any algorithm that is based on distances (e.g. KNN, SVM, kernel-based methods, tree-based methods,...)
- Blessing of non-uniformity: on many applications, the data lives in a very small subspace
  - You can drastically improve performance by selecting features or using lower-dimensional data representations

## "More data can beat a cleverer algorithm"

(but you need both)

- More data reduces the chance of overfitting
- Less sparse data reduces the curse of dimensionality
- *Non-parametric* models: number of model parameters grows with amount of data
  - Tree-based techniques, k-Nearest neighbors, SVM,...
  - They can learn any model given sufficient data (but can get stuck in local minima)
- *Parametric* (fixed size) models: fixed number of model parameters
  - Linear models, Neural networks,...
  - Can be given a huge number of parameters to benefit from more data
  - Deep learning models can have millions of weights, learn almost any function.
- The bottleneck is moving from data to compute/scalability

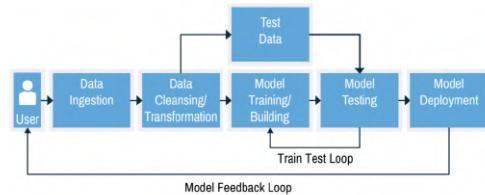
## Building machine learning systems

A typical machine learning system has multiple components, which we will cover in upcoming lectures:

- Preprocessing: Raw data is rarely ideal for learning
  - Feature scaling: bring values in same range
  - Encoding: make categorical features numeric
  - Discretization: make numeric features categorical
  - Label imbalance correction (e.g. downsampling)
  - Feature selection: remove uninteresting/correlated features
  - Dimensionality reduction can also make data easier to learn
  - Using pre-learned embeddings (e.g. word-to-vector, image-to-vector)

- Learning and evaluation
  - Every algorithm has its own biases
  - No single algorithm is always best
  - *Model selection* compares and selects the best models
    - Different algorithms, different hyperparameter settings
  - Split data in training, validation, and test sets
- Prediction
  - Final optimized model can be used for prediction
  - Expected performance is performance measured on *independent* test set

- Together they form a *workflow* of *pipeline*
- There exist machine learning methods to automatically build and tune these pipelines
- You need to optimize pipelines continuously
  - *Concept drift*: the phenomenon you are modelling can change over time
  - *Feedback*: your model's predictions may change future data



## Summary

- Learning algorithms contain 3 components:
  - Representation: a model  $f$  that maps input data  $X$  to desired output  $y$ 
    - Contains model parameters  $\theta$  that can be made to fit the data  $X$
  - Loss function  $\mathcal{L}(f_\theta(X))$ : measures how well the model fits the data
  - Optimization technique to find the optimal  $\theta$ :  $\operatorname{argmin}_\theta \mathcal{L}(f_\theta(X))$
- Select the right model, then fit it to the data to minimize a task-specific error  $\mathcal{E}$ 
  - Inductive bias  $b$ : assumptions about model and hyperparameters
 
$$\operatorname{argmin}_{\theta,b} \mathcal{E}(f_{\theta,b}(X))$$
- Overfitting: model fits the training data well but not new (test) data
  - Split the data into (multiple) train-validation-test splits
  - Regularization: tune hyperparameters (on validation set) to simplify model
  - Gather more data, or build ensembles of models
- Machine learning *pipelines*: preprocessing + learning + deployment

## Lecture 2: Linear models

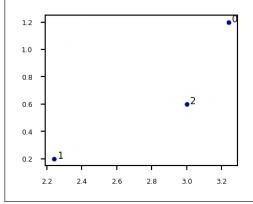
### Basics of modeling, optimization, and regularization

Joaquin Vanschoren

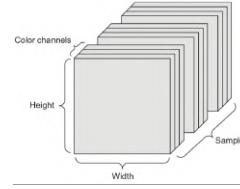
### Notation and Definitions

- A **scalar** is a simple numeric value, denoted by an italic letter:  $x = 3.24$
- A **vector** is a 1D ordered array of  $n$  scalars, denoted by a bold letter:  $\mathbf{x} = [3.24, 1.2]$ 
  - $x_i$  denotes the  $i$ th element of a vector, thus  $x_0 = 3.24$ .
  - Note: some other courses use  $x^{(i)}$  notation
- A **set** is an *unordered* collection of unique elements, denote by calligraphic capital:  $\mathcal{S} = \{3.24, 1.2\}$
- A **matrix** is a 2D array of scalars, denoted by bold capital:  $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \end{bmatrix}$ 
  - $\mathbf{X}_i$  denotes the  $i$ th row of the matrix
  - $\mathbf{X}_{:,j}$  denotes the  $j$ th column
  - $\mathbf{X}_{i,j}$  denotes the element in the  $i$ th row,  $j$ th column, thus  $\mathbf{X}_{1,0} = 2.24$

- $\mathbf{X}^{n \times p}$ , an  $n \times p$  matrix, can represent  $n$  data points in a  $p$ -dimensional space
  - Every row is a vector that can represent a *point* in an  $n$ -dimensional space, given a *basis*.
  - The *standard basis* for a Euclidean space is the set of unit vectors
  - E.g. if  $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \\ 3.0 & 0.6 \end{bmatrix}$



- A **tensor** is an  $k$ -dimensional array of data, denoted by an italic capital:  $T$ 
  - $k$  is also called the order, degree, or rank
  - $T_{i,j,k,\dots}$  denotes the element or sub-tensor in the corresponding position
  - A set of color images can be represented by:
    - a 4D tensor (sample x height x width x color channel)
    - a 2D tensor (sample x flattened vector of pixel values)



### Basic operations

- Sums and products are denoted by capital Sigma and capital Pi:

$$\sum_{i=0}^p = x_0 + x_1 + \dots + x_p \quad \prod_{i=0}^p = x_0 \cdot x_1 \cdot \dots \cdot x_p$$

- Operations on vectors are **element-wise**: e.g.  $\mathbf{x} + \mathbf{z} = [x_0 + z_0, x_1 + z_1, \dots, x_p + z_p]$

- Dot product  $\mathbf{w}\mathbf{x} = \mathbf{w} \cdot \mathbf{x} = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^p w_i \cdot x_i = w_0 \cdot x_0 + w_1 \cdot x_1 + \dots + w_p \cdot x_p$

- Matrix product  $\mathbf{W}\mathbf{x} = \begin{bmatrix} \mathbf{w}_0 \cdot \mathbf{x} \\ \dots \\ \mathbf{w}_p \cdot \mathbf{x} \end{bmatrix}$

- A function  $f(x) = y$  relates an input element  $x$  to an output  $y$

- It has a *local minimum* at  $x = c$  if  $f(x) \geq f(c)$  in interval  $(c - \epsilon, c + \epsilon)$
- It has a *global minimum* at  $x = c$  if  $f(x) \geq f(c)$  for any value for  $x$

- A vector function consumes an input and produces a vector:  $\mathbf{f}(\mathbf{x}) = \mathbf{y}$

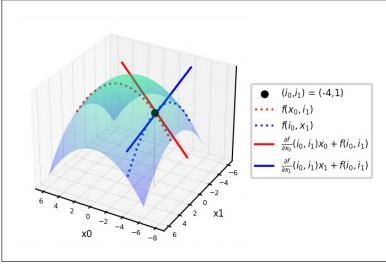
- $\max_{x \in X} f(x)$  returns the highest value  $f(x)$  for any  $x$

- $\operatorname{argmax}_{x \in X} f(x)$  returns the element  $x$  that maximizes  $f(x)$

### Gradients

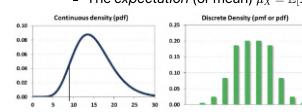
- A **derivative**  $f'$  of a function  $f$  describes how fast  $f$  grows or decreases
- The process of finding a derivative is called **differentiation**
  - Derivatives for basic functions are known
  - For non-basic functions we use the chain rule:  $F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$
- A function is **differentiable** if it has a derivate in any point of its domain
  - It's *continuously differentiable* if  $f'$  is itself a function
  - It's *smooth* if  $f', f'', f''', \dots$  all exist
- A **gradient**  $\nabla f$  is the derivate of a function in multiple dimensions
  - It is a vector of partial derivatives:  $\nabla f = \left[ \frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \dots \right]$
  - E.g.  $f = 2x_0 + 3x_1^2 - \sin(x_2) \rightarrow \nabla f = [2, 6x_1, -\cos(x_2)]$

- Example:  $f = -(x_0^2 + x_1^2)$ 
  - $\nabla f = \left[ \frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1} \right] = [-2x_0, -2x_1]$
  - Evaluated at point (-4,1):  $\nabla f(-4,1) = [8, -2]$ 
    - These are the slopes at point (-4,1) in the direction of  $x_0$  and  $x_1$  respectively



## Distributions and Probabilities

- The normal (Gaussian) distribution with mean  $\mu$  and standard deviation  $\sigma$  is noted as  $N(\mu, \sigma)$
- A random variable  $X$  can be continuous or discrete
- A probability distribution  $f_X$  of a continuous variable  $X$ : **probability density function (pdf)**
  - The expectation is given by  $E[X] = \int x f_X(x) dx$
- A probability distribution of a discrete variable: **probability mass function (pmf)**
  - The expectation (or mean)  $\mu_X = E[X] = \sum_{i=1}^k x_i \cdot P(X=x_i)$



## Linear models

Linear models make a prediction using a linear function of the input features  $X$

$$f_w(\mathbf{x}) = \sum_{i=1}^p w_i \cdot x_i + w_0$$

Learn  $w$  from  $X$ , given a loss function  $\mathcal{L}$ :

$$\underset{w}{\operatorname{argmin}} \mathcal{L}(f_w(X))$$

- Many algorithms with different  $\mathcal{L}$ : Least squares, Ridge, Lasso, Logistic Regression, Linear SVMs,...
- Can be very powerful (and fast), especially for large datasets with many features.
- Can be generalized to learn non-linear patterns: *Generalized Linear Models*
  - Features can be augmented with polynomials of the original features
  - Features can be transformed according to a distribution (Poisson, Tweedie, Gamma,...)
  - Some linear models (e.g. SVMs) can be *kernelized* to learn non-linear functions

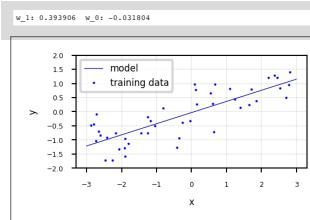
## Linear models for regression

- Prediction formula for input features  $x$ :

- $w_1 \dots w_p$  usually called *weights* or *coefficients*,  $w_0$  the *bias* or *intercept*

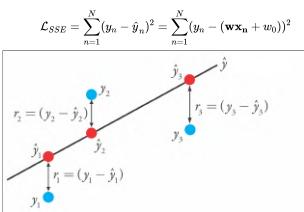
- Assumes that errors are  $N(0, \sigma)$

$$\hat{y} = \mathbf{w}\mathbf{x} + w_0 = \sum_{i=1}^p w_i \cdot x_i + w_0 = w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_p \cdot x_p + w_0$$



## Linear Regression (aka Ordinary Least Squares)

- Loss function is the sum of squared errors (SSE) (or residuals) between predictions  $\hat{y}_i$  (red) and the true regression targets  $y_i$  (blue) on the training set.



## SOLVING ORDINARY LEAST SQUARES

- Convex optimization problem with unique closed-form solution:

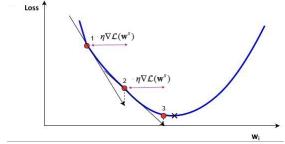
$$w^* = (X^T X)^{-1} X^T Y$$

- Add a column of 1's to the front of  $X$  to get  $w_0$
- Slow. Time complexity is quadratic in number of features:  $O(p^2 n)$ 
  - $X$  has  $n$  rows,  $p$  features, hence  $X^T X$  has dimensionality  $p \cdot p$
  - Only works if  $n > p$
- Gradient Descent*
  - Faster for large and/or high-dimensional datasets
  - When  $X^T X$  cannot be computed or takes too long ( $p$  or  $n$  is too large)
- Very easily overfits.**
  - coefficients  $w$  become very large (steep incline/decline)
  - small change in the input  $x$  results in a very different output  $y$
  - No hyperparameters that control model complexity

## GRADIENT DESCENT

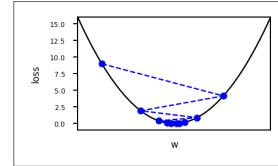
- Start with an initial, random set of weights:  $\mathbf{w}^0$
- Given a differentiable loss function  $\mathcal{L}$  (e.g.  $\mathcal{L}_{SSE}$ ), compute  $\nabla \mathcal{L}$
- For least squares:  $\frac{\partial \mathcal{L}_{SSE}}{\partial w_i}(\mathbf{w}) = -2 \sum_{n=1}^N (y_n - \hat{y}_n)x_{n,i}$ 
  - If feature  $X_{:,i}$  is associated with big errors, the gradient wrt  $w_i$  will be large
- Update all weights slightly (by step size or learning rate  $\eta$ ) in 'downhill' direction.
- Basic update rule (step s):

$$\mathbf{w}^{s+1} = \mathbf{w}^s - \eta \nabla \mathcal{L}(\mathbf{w}^s)$$

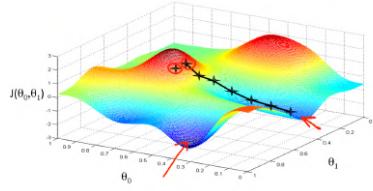


### Important hyperparameters

- Learning rate
  - Too small: slow convergence. Too large: possible divergence
- Maximum number of iterations
  - Too small: no convergence. Too large: wastes resources
- Learning rate decay with decay rate  $k$ 
  - E.g. exponential ( $\eta^{s+1} = \eta^0 e^{-ks}$ ), inverse-time ( $\eta^{s+1} = \frac{\eta^0}{1+ks}$ )...
- Many more advanced ways to control learning rate (see later)
  - Adaptive techniques: depend on how much loss improved in previous step



In two dimensions:



- You can get stuck in local minima (if the loss is not fully convex)
  - If you have many model parameters, this is less likely
  - You always find a way down in some direction
  - Models with many parameters typically find good local minima

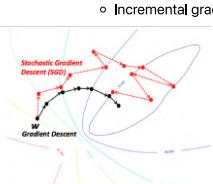
### Intuition: walking downhill using only the slope you "feel" nearby



(Image by A. Karpathy)

## STOCHASTIC GRADIENT DESCENT (SGD)

- Compute gradients not on the entire dataset, but on a single data point  $i$  at a time
  - Gradient descent:  $\mathbf{w}^{s+1} = \mathbf{w}^s - \eta \nabla \mathcal{L}(\mathbf{w}^s) = \mathbf{w}^s - \frac{\eta}{n} \sum_{i=1}^n \nabla \mathcal{L}_i(\mathbf{w}^s)$
  - Stochastic Gradient Descent:  $\mathbf{w}^{s+1} = \mathbf{w}^s - \eta \nabla \mathcal{L}_i(\mathbf{w}^s)$
- Many smoother variants, e.g.
  - Minibatch SGD: compute gradient on batches of data:  $\mathbf{w}^{s+1} = \mathbf{w}^s - \frac{\eta}{B} \sum_{i=1}^B \nabla \mathcal{L}_i(\mathbf{w}^s)$
  - Stochastic Average Gradient Descent (SAG, SAGA). With  $i_s \in [1, n]$  randomly chosen per iteration:
    - Incremental gradient:  $\mathbf{w}^{s+1} = \mathbf{w}^s - \frac{\eta}{n} \sum_{i=1}^n v_i^s$  with  $v_i^s = \begin{cases} \nabla \mathcal{L}_i(\mathbf{w}^s) & i = i_s \\ v_i^{s-1} & \text{otherwise} \end{cases}$



## IN PRACTICE

- Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.

- `LinearRegression` uses closed form solution, `SGDRegressor` with `loss='squared_loss'` uses Stochastic Gradient Descent
- Large coefficients signal overfitting
- Test score is much lower than training score

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression().fit(X_train, y_train)
```

```
Weights (coefficients): [-412.711 -52.243 -131.899 -12.004 -15.511 28.716 54.704 2980.775 26.582 150.008 114.187 -16.971 40.961 -24.264 57.616 1278.121 -2239.869 222.825 -2.182 42.996 -13.398 -19.389 -2.375 -89.013 9.66 4.914 -0.012 -7.647 33.784 -1.145 89.135 -17.005 48.813 1.14 ]
```

```
Bias (intercept): 30.93456367364078
```

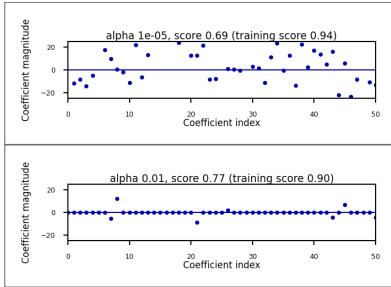
```
Training set score (R^2): 0.95
```

```
Test set score (R^2): 0.61
```



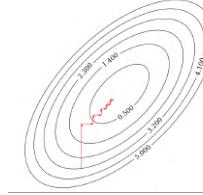
Analyze what happens to the weights:

- L1 prefers coefficients to be exactly zero (sparse models)
- Some features are ignored entirely: automatic feature selection
- How can we explain this?



#### COORDINATE DESCENT

- Alternative for gradient descent, supports non-differentiable convex loss functions (e.g.  $\mathcal{L}_{Lasso}$ )
- In every iteration, optimize a single coordinate  $w_i$  (find minimum in direction of  $x_i$ )
  - Continue with another coordinate, using a selection rule (e.g. round robin)
- Faster iterations. No need to choose a step size (learning rate).
- May converge more slowly. Can't be parallelized.



#### COORDINATE DESCENT WITH LASSO

- Remember that  $\mathcal{L}_{Lasso} = \mathcal{L}_{SSE} + \alpha \sum_{i=1}^p |w_i|$
- For one  $w_i$ :  $\mathcal{L}_{Lasso}(w_i) = \mathcal{L}_{SSE}(w_i) + \alpha |w_i|$
- The L1 term is not differentiable but convex: we can compute the **subgradient**
  - Unique at points where  $\mathcal{L}$  is differentiable, a range of all possible slopes [a,b] where it is not
  - For  $|w_i|$ , the subgradient  $\partial_{w_i}|w_i| = \begin{cases} -1 & w_i < 0 \\ [-1, 1] & w_i = 0 \\ 1 & w_i > 0 \end{cases}$
  - Subdifferential  $\partial(f+g) = \partial f + \partial g$  if  $f$  and  $g$  are both convex
- To find the optimum for Lasso  $w_i^*$ , solve
 
$$\begin{aligned} \partial_{w_i} \mathcal{L}_{Lasso}(w_i) &= \partial_{w_i} \mathcal{L}_{SSE}(w_i) + \partial_{w_i} \alpha |w_i| \\ 0 &= (w_i - \rho_i) + \alpha \cdot \partial_{w_i} |w_i| \\ w_i &= \rho_i - \alpha \cdot \partial_{w_i} |w_i| \end{aligned}$$
  - In which  $\rho_i$  is the solution for  $\mathcal{L}_{SSE}(w_i)$

- We found:  $w_i = \rho_i - \alpha \cdot \partial_{w_i} |w_i|$

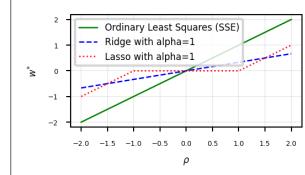
- Lasso solution has the form of a **soft thresholding function**

$$w_i^* = S(\rho_i, \alpha) = \begin{cases} \rho_i + \alpha, & \rho_i < -\alpha \\ 0, & -\alpha < \rho_i < \alpha \\ \rho_i - \alpha, & \rho_i > \alpha \end{cases}$$

- Small weights become 0: sparseness!

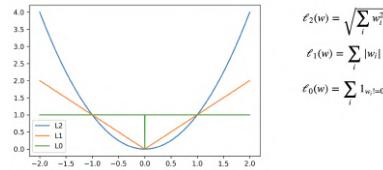
- If the data is not normalized,  $w_i^* = \frac{1}{z_i} S(\rho_i, \alpha)$  with  $z_i$  a normalizing constant

- Ridge solution:  $w_i = \rho_i - \alpha \cdot \partial_{w_i} w_i^2 = \rho_i - 2\alpha \cdot w_i$ , thus  $w_i^* = \frac{\rho_i}{1+2\alpha}$



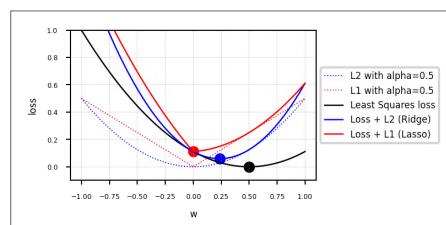
#### Interpreting L1 and L2 loss

- L1 and L2 in function of the weights

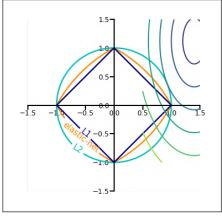


#### Least Squares Loss + L1 or L2

- Lasso is not differentiable at point 0
- For any minimum of least squares, L2 will be smaller, and L1 is more likely to be 0



- In 2D (for 2 model weights  $w_1$  and  $w_2$ )
  - The least squared loss is a 2D convex function in this space
  - For illustration, assume that L1 loss = L2 loss = 1
    - L1 loss ( $\sum|w_i|$ ): every  $\{w_1, w_2\}$  falls on the diamond
    - L2 loss ( $\sum w_i^2$ ): every  $\{w_1, w_2\}$  falls on the circle
  - For L1, the loss is minimized if  $w_1$  or  $w_2$  is 0 (rarely so for L2)

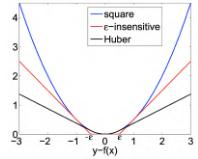


### Elastic-Net

- Adds both L1 and L2 regularization:
$$\mathcal{L}_{Elastic} = \sum_{i=1}^N (y_i - (\mathbf{x}_i \cdot \mathbf{w} + w_0))^2 + \alpha\rho \sum_{i=1}^p |w_i| + \alpha(1-\rho) \sum_{i=1}^p w_i^2$$
- $\rho$  is the L1 ratio
  - With  $\rho = 1$ ,  $\mathcal{L}_{Elastic} = \mathcal{L}_{Lasso}$
  - With  $\rho = 0$ ,  $\mathcal{L}_{Elastic} = \mathcal{L}_{Ridge}$
  - $0 < \rho < 1$  sets a trade-off between L1 and L2.
- Allows learning sparse models (like Lasso) while maintaining L2 regularization benefits
  - E.g. if 2 features are correlated, Lasso likely picks one randomly, Elastic-Net keeps both
- Weights can be optimized using coordinate descent (similar to Lasso)

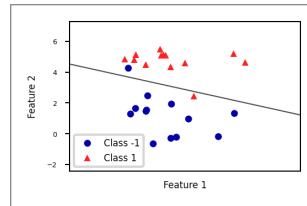
### Other loss functions for regression

- Huber loss: switches from squared loss to linear loss past a value  $\epsilon$ 
  - More robust against outliers
- Epsilon insensitive: ignores errors smaller than  $\epsilon$ , and linear past that
  - Aims to fit function so that residuals are at most  $\epsilon$
  - Also known as Support Vector Regression (SVR in sklearn)
- Squared Epsilon insensitive: ignores errors smaller than  $\epsilon$ , and squared past that
- These can all be solved with stochastic gradient descent
  - SGDRegressor in sklearn



### Linear models for Classification

Aims to find a hyperplane that separates the examples of each class. For binary classification (2 classes), we aim to fit the following function:  
 $\hat{y} = w_1 * x_1 + w_2 * x_2 + \dots + w_p * x_p + w_0 > 0$   
When  $\hat{y} < 0$ , predict class -1, otherwise predict class +1



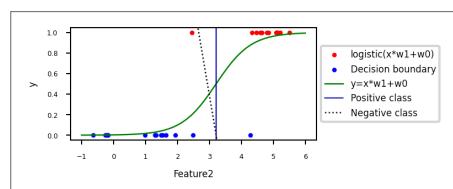
- There are many algorithms for linear classification, differing in loss function, regularization techniques, and optimization method
- Most common techniques:
  - Convert target classes {neg, pos} to {0, 1} and treat as a regression task
    - Logistic regression (Log loss)
    - Ridge Classification (Least Squares + L2 loss)
  - Find hyperplane that maximizes the margin between classes
    - Linear Support Vector Machines (Hinge loss)
  - Neural networks without activation functions
    - Perceptron (Perceptron loss)
  - SGDClassifier: can act like any of these by choosing loss function
    - Hinge, Log, Modified\_hubert, Squared\_hinge, Perceptron

### Logistic regression

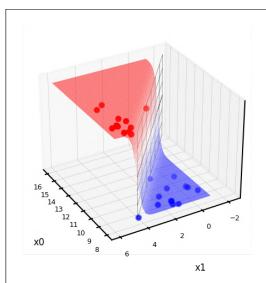
- Aims to predict the probability that a point belongs to the positive class
- Converts target values {negative (blue), positive (red)} to {0, 1}
- Fits a logistic (or sigmoid or S curve) function through these points
  - Maps  $(-\infty, \infty)$  to a probability [0, 1]

$$\hat{y} = \text{logistic}(f_\theta(\mathbf{x})) = \frac{1}{1 + e^{-f_\theta(\mathbf{x})}}$$

- E.g. in 1D:  $\text{logistic}(x_1 w_1 + w_0) = \frac{1}{1 + e^{-(x_1 w_1 + w_0)}}$



- Fitted solution to our 2D example:
  - To get a binary prediction, choose a probability threshold (e.g. 0.5)

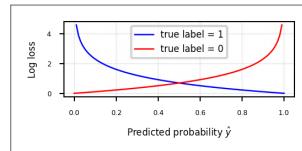


#### LOSS FUNCTION: CROSS-ENTROPY

- Models that return class probabilities can use cross-entropy loss

$$\mathcal{L}_{\text{log}}(\mathbf{w}) = - \sum_{n=1}^N H(p_n, q_n) = - \sum_{n=1}^N \sum_{c=1}^C p_{n,c} \log(q_{n,c})$$

- Also known as log loss, logistic loss, or maximum likelihood
- Based on true probabilities  $p$  (0 or 1) and predicted probabilities  $q$  over  $N$  instances and  $C$  classes
  - Binary case ( $C=2$ ):  $\mathcal{L}_{\text{log}}(\mathbf{w}) = - \sum_{n=1}^N [y_n \log(\hat{y}_n) + (1 - y_n) \log(1 - \hat{y}_n)]$
  - Penalty (or surprise) grows exponentially as difference between  $p$  and  $q$  increases
  - Often used together with L2 (or L1) loss:  $\mathcal{L}_{\text{log}}'(\mathbf{w}) = \mathcal{L}_{\text{log}}(\mathbf{w}) + \alpha \sum_i w_i^2$



#### OPTIMIZATION METHODS (SOLVERS) FOR CROSS-ENTROPY LOSS

- Gradient descent (only supports L2 regularization)
  - Log loss is differentiable, so we can use (stochastic) gradient descent
  - Variants thereof, e.g. Stochastic Average Gradient (SAG, SAGA)
- Coordinate descent (supports both L1 and L2 regularization)
  - Faster iteration, but may converge more slowly, has issues with saddlepoints
  - Called `liblinear` in sklearn. Can't run in parallel.
- Newton-Raphson or Newton Conjugate Gradient (only L2):
  - Uses the Hessian  $H = [\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_i \partial \mathbf{w}_j}]$ :  $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta H^{-1}(\mathbf{w}^t) \nabla \mathcal{L}(\mathbf{w}^t)$
  - Slow for large datasets. Works well if solution space is (near) convex
- Quasi-Newton methods (only L2)
  - Approximate, faster to compute
  - E.g. Limited-memory Broyden–Fletcher–Goldfarb–Shanno (`lbfgs`)
    - Default in sklearn for Logistic Regression
- [Some hints on choosing solvers](#)
  - Data scaling helps convergence, minimizes differences between solvers

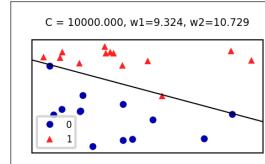
#### IN PRACTICE

- Logistic regression can also be found in `sklearn.linear_model`.

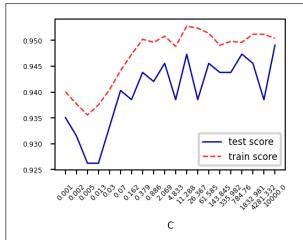
- `C` hyperparameter is the inverse regularization strength:  $C = \alpha^{-1}$
- `penalty`: type of regularization: L1, L2 (default), Elastic-Net, or None
- `solver`: `newton-cg`, `lbfgs` (default), `liblinear`, `sag`, `saga`

- Increasing `C`: less regularization, tries to overfit individual points

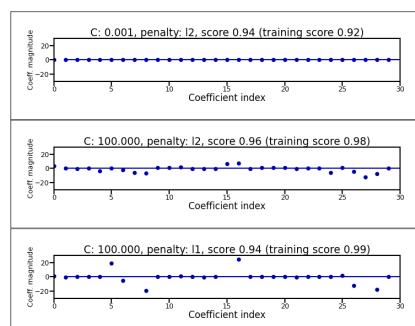
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1).fit(X_train, y_train)
```



- Analyze behavior on the breast cancer dataset
  - Underfitting if  $C$  is too small, some overfitting if  $C$  is too large
  - We use cross-validation because the dataset is small



- Again, choose between L1 or L2 regularization (or elastic-net)
- Small  $C$  overfits, L1 leads to sparse models



## Ridge Classification

- Instead of log loss, we can also use ridge loss:

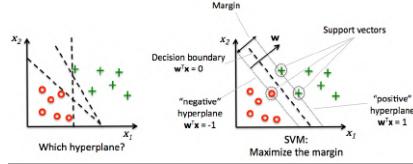
$$\mathcal{L}_{Ridge} = \sum_{n=1}^N (y_n - (\mathbf{w} \cdot \mathbf{x}_n + w_0))^2 + \alpha \sum_{i=0}^p w_i^2$$

- In this case, target values {negative, positive} are converted to {-1, 1}
- Can be solved similarly to Ridge regression:

- Closed form solution (a.k.a. Cholesky)
- Gradient descent and variants
  - E.g. Conjugate Gradient (CG) or Stochastic Average Gradient (SAG,SAGA)
- Use Cholesky for smaller datasets, Gradient descent for larger ones

## Support vector machines

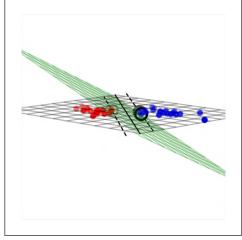
- Decision boundaries close to training points may generalize badly
  - Very similar (nearby) test point are classified as the other class
- Choose a boundary that is as far away from training points as possible
- The **support vectors** are the training samples closest to the hyperplane
- The **margin** is the distance between the separating hyperplane and the support vectors
- Hence, our objective is to *maximize the margin*



## SOLVING SVMs WITH LAGRANGE MULTIPLIERS

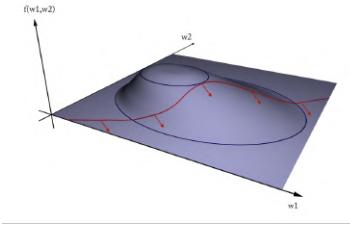
- Imagine a hyperplane (green)  $y = \sum_i^p w_i \cdot x_i + w_0$  that has slope  $w$ , value '+1' for the positive (red) support vectors, and '-1' for the negative (blue) ones
  - Margin between the boundary and support vectors is  $\frac{2w_0}{\|w\|^2}$ , with  $\|w\| = \sqrt{\sum_i^p w_i^2}$
  - We want to find the weights that maximize  $\frac{1}{\|w\|^2}$ . We can also do that by maximizing

$$\frac{1}{\|w\|^2}$$



## Geometric interpretation

- We want to maximize  $f = \frac{1}{\|w\|^2}$  (blue contours)
- The hyperplane (red) must be  $> 1$  for all positive examples:
 
$$g(\mathbf{w}) = \mathbf{w} \cdot \mathbf{x}_i + w_0 > 1 \quad \forall i, y(i) = 1$$
- Find the weights  $\mathbf{w}$  that satisfy  $g$  but maximize  $f$



## Solution

- A quadratic loss function with linear constraints can be solved with **Lagrangian multipliers**
- This works by assigning a weight  $a_i$  (called a dual coefficient) to every data point  $x_i$ 
  - They reflect how much individual points influence the weights  $w$
  - The points with non-zero  $a_i$  are the **support vectors**
- Next, solve the following **Primal** objective:
  - $y_i = \pm 1$  is the correct class for example  $x_i$

$$\mathcal{L}_{Primal} = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n a_i y_i (\mathbf{w} \cdot \mathbf{x}_i + w_0) + \sum_{i=1}^n a_i$$

so that

$$\mathbf{w} = \sum_{i=1}^n a_i y_i \mathbf{x}_i$$

$$a_i \geq 0 \quad \text{and} \quad \sum_{i=1}^n a_i y_i = 0$$

- It has a **Dual** formulation as well (See 'Elements of Statistical Learning' for the derivation):

$$\mathcal{L}_{Dual} = \sum_{i=1}^l a_i - \frac{1}{2} \sum_{i,j=1}^l a_i a_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

so that

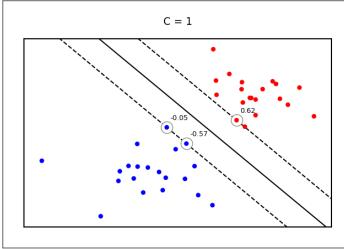
$$a_i \geq 0 \quad \text{and} \quad \sum_{i=1}^l a_i y_i = 0$$

- Computes the dual coefficients directly. A number  $l$  of these are non-zero (sparseness).

- Dot product  $\mathbf{x}_i \cdot \mathbf{x}_j$  can be interpreted as the closeness between points  $\mathbf{x}_i$  and  $\mathbf{x}_j$
- $\mathcal{L}_{Dual}$  increases if nearby support vectors  $\mathbf{x}_i$  with high weights  $a_i$  have different class  $y_i$
- $\mathcal{L}_{Dual}$  also increases with the number of support vectors  $l$  and their weights  $a_i$

- Can be solved with quadratic programming, e.g. Sequential Minimal Optimization (SMO)

Example result. The circled samples are support vectors, together with their coefficients.



#### MAKING PREDICTIONS

- $a_i$  will be 0 if the training point lies on the right side of the decision boundary and outside the margin
- The training samples for which  $a_i$  is not 0 are the *support vectors*
- Hence, the SVM model is completely defined by the support vectors and their dual coefficients (weights)
- Knowing the dual coefficients  $a_i$ , we can find the weights  $w$  for the maximal margin separating hyperplane:

$$w = \sum_{i=1}^l a_i y_i x_i$$

- Hence, we can classify a new sample  $u$  by looking at the sign of  $wu + w_0$

#### SVMs and kNN

- Remember, we will classify a new point  $u$  by looking at the sign of:

$$f(x) = wu + w_0 = \sum_{i=1}^k a_i y_i x_i u + w_0$$

- Weighted k-nearest neighbor is a generalization of the k-nearest neighbor classifier. It classifies points by evaluating:

$$f(x) = \sum_{i=1}^k a_i y_i \text{dist}(x_i, u)^{-1}$$

- Hence: SVM's predict much the same way as k-NN, only:

- They only consider the truly important points (the support vectors): *much* faster
- The number of neighbors is the number of support vectors
- The distance function is an *inner product of the inputs*

#### REGULARIZED (SOFT MARGIN) SVMs

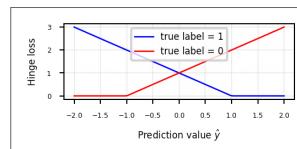
- If the data is not linearly separable, (hard) margin maximization becomes meaningless
- Relax the constraint by allowing an error  $\xi_i$ :  $y_i(wx_i + w_0) \geq 1 - \xi_i$
- Or (since  $\xi_i \geq 0$ ):

$$\xi_i = \max(0, 1 - y_i \cdot (wx_i + w_0))$$

- The sum over all points is called *hinge loss*:  $\sum_i \xi_i$
- Attenuating the error component with a hyperparameter  $C$ , we get the objective

$$\mathcal{L}(w) = \|w\|^2 + C \sum_i \xi_i$$

- Can still be solved with quadratic programming

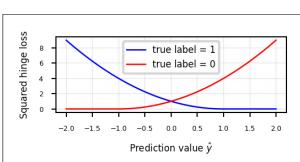


#### LEAST SQUARES SVMs

- We can also use the squares of all the errors, or squared hinge loss:  $\sum_i \xi_i^2$
- This yields the Least Squares SVM objective

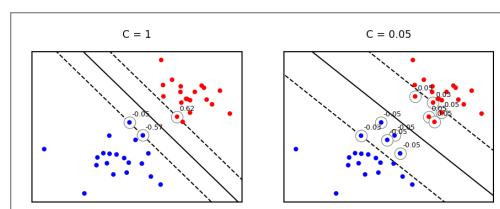
$$\mathcal{L}(w) = \|w\|^2 + C \sum_i \xi_i^2$$

- Can be solved with Lagrangian Multipliers and a set of linear equations
  - Still yields support vectors and still allows kernelization
  - Support vectors are not sparse, but pruning techniques exist

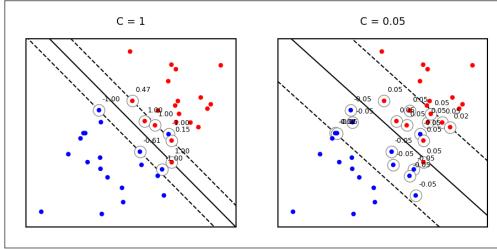


#### EFFECT OF REGULARIZATION ON MARGIN AND SUPPORT VECTORS

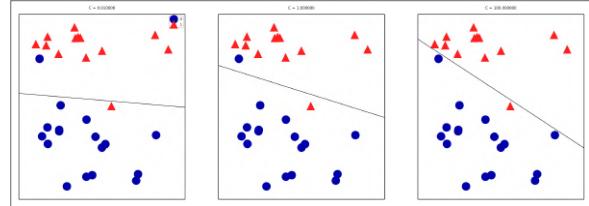
- SVM's Hinge loss acts like L1 regularization, yields sparse models
- $C$  is the *inverse* regularization strength (inverse of  $\alpha$  in Lasso)
  - Larger  $C$ : fewer support vectors, smaller margin, more overfitting
  - Smaller  $C$ : more support vectors, wider margin, less overfitting
- Needs to be tuned carefully to the data



Same for non-linearly separable data



Large C values can lead to overfitting (e.g. fitting noise), small values can lead to underfitting



#### SVMs in scikit-learn

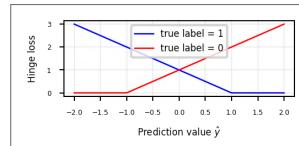
- `svm.LinearSVC`: faster for large datasets
  - Allows choosing between the primal or dual. Primal recommended when  $n \gg p$
  - Returns `coef_` ( $w$ ) and `intercept_` ( $w_0$ )
- `svm.SVC` with `kernel='linear'`: allows kernelization (see later)
  - Also returns `support_vectors_` (the support vectors) and the `dual_coef_` ( $a_i$ )
  - Scales at least quadratically with the number of samples  $n$
- `svm.LinearSVR` and `svm.SVR` are variants for regression

```
clf = svm.SVC(kernel='linear')
clf.fit(X, Y)
print("Support vectors:", clf.support_vectors_)
print("Coefficients:", clf.dual_coef_[0])
```

```
Support vectors:
[[-0.001 -0.241]
 [-0.467 -0.531]
 [ 0.951  0.58 ]]
Coefficients:
[[-0.048 -0.569  0.617]]
```

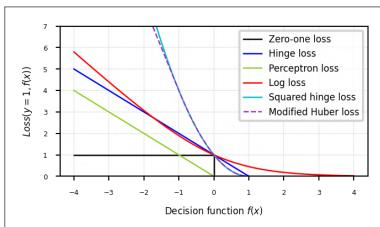
#### Solving SVMs with Gradient Descent

- Soft-margin SVMs can, alternatively, be solved using gradient descent
    - Good for large datasets, but does not yield support vectors or kernelization
  - Squared Hinge is differentiable
  - Hinge is not differentiable but convex, and has a subgradient:
- $$\mathcal{L}_{\text{Hinge}}(\mathbf{w}) = \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0))$$
- $$\frac{\partial \mathcal{L}_{\text{Hinge}}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) < 1 \\ 0 & \text{otherwise} \end{cases}$$
- Can be solved with (stochastic) gradient descent



#### Generalized SVMs

- Because the derivative of hinge loss is undefined at  $y=1$ , smoothed versions are often used:
  - Squared hinge loss: yields *least squares SVM*
    - Equivalent to Ridge classification (with different solver)
  - Modified Huber loss: squared hinge, but linear after  $-1$ . Robust against outliers
- Log loss can also be used (equivalent to logistic regression)
- In scikit-learn, `SGDClassifier` can be used with any of these. Good for large datasets.



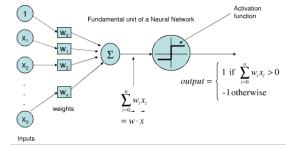
#### Perceptron

- Represents a single neuron (node) with inputs  $x_i$ , a bias  $w_0$ , and output  $y$
- Each connection has a (synaptic) weight  $w_i$ . The node outputs  $\hat{y} = \sum_i^n x_i w_i + w_0$
- The *activation function* predicts 1 if  $\hat{y} > 0$ , -1 otherwise
- Weights can be learned with (stochastic) gradient descent and Hinge(0) loss

- Updated only on misclassification, corrects output by  $\pm 1$

$$\mathcal{L}_{\text{Perceptron}} = \max(0, -y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0))$$

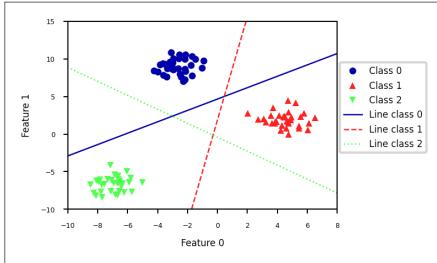
$$\frac{\partial \mathcal{L}_{\text{Perceptron}}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) < 0 \\ 0 & \text{otherwise} \end{cases}$$



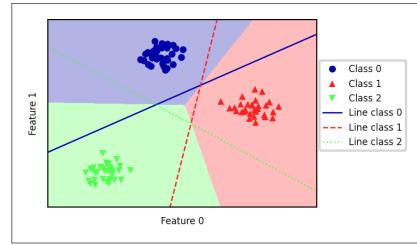
## Linear Models for multiclass classification

### one-vs-rest (aka one vs all)

- Learn a binary model for each class vs. all other classes
- Create as many binary models as there are classes



- Every binary classifiers makes a prediction, the one with the highest score ( $>0$ ) wins



### one-vs-one

- An alternative is to learn a binary model for every combination of two classes
  - For  $C$  classes, this results in  $\frac{C(C-1)}{2}$  binary models
  - Each point is classified according to a majority vote amongst all models
  - Can also be a 'soft vote': sum up the probabilities (or decision values) for all models. The class with the highest sum wins.
- Requires more models than one-vs-rest, but training each one is faster
  - Only the examples of 2 classes are included in the training data
- Recommended for algorithms that learn well on small datasets
  - Especially SVMs and Gaussian Processes

## Linear models overview

Name	Representation	Loss function	Optimization	Regularization
Least squares	Linear function (R)	SSE	CFS or SGD	None
Ridge	Linear function (R)	SSE + L2	CFS or SGD	L2 strength ( $\alpha$ )
Lasso	Linear function (R)	SSE + L1	Coordinate descent	L1 strength ( $\alpha$ )
Elastic-Net	Linear function (R)	SSE + L1 + L2	Coordinate descent	$\alpha$ , L1 ratio ( $\rho$ )
SGDRRegressor	Linear function (R)	SSE, Huber, $\epsilon$ -ins,... + L1/L2	SGD	L1/L2, $\alpha$
Logistic regression	Linear function (C)	Log + L1/L2	SGD, coordinate descent,...	L1/L2, $\alpha$
Ridge classification	Linear function (C)	SSE + L2	CFS or SGD	L2 strength ( $\alpha$ )
Linear SVM	Support Vectors	Hinge(1)	Quadratic programming or SGD	Cost (C)

Name	Representation	Loss function	Optimization	Regularization
Least Squares SVM	Support Vectors	Squared Hinge	Linear equations or SGD	Cost (C)
Perceptron	Linear function (C)	Hinge(0)	SGD	None
SGDClassifier	Linear function (C)	Log, (Sq.) Hinge, Mod. Huber,...	SGD	L1/L2, $\alpha$

- SSE: Sum of Squared Errors
- CFS: Closed-form solution
- SGD: (Stochastic) Gradient Descent and variants
- (R)egression, (C)lassification

## Summary

- Linear models
  - Good for very large datasets (scalable)
  - Good for very high-dimensional data (not for low-dimensional data)
- Can be used to fit non-linear or low-dim patterns as well (see later)
  - Preprocessing: e.g. Polynomial or Poisson transformations
  - Generalized linear models (kernelization)
- Regularization is important. Tune the regularization strength ( $\alpha$ )
  - Ridge (L2): Good fit, sometimes sensitive to outliers
  - Lasso (L1): Sparse models: fewer features, more interpretable, faster
  - Elastic-Net: Trade-off between both, e.g. for correlated features
- Most can be solved by different optimizers (solvers)
  - Closed form solutions or quadratic/linear solvers for smaller datasets
  - Gradient descent variants (SGD, CD, SAG, CG, ...) for larger ones
- Multi-class classification can be done using a one-vs-all approach



## Feature Maps

- Linear models:  $\hat{y} = \mathbf{w}\mathbf{x} + w_0 = \sum_{i=1}^p w_i x_i + w_0 = w_0 + w_1 x_1 + \dots + w_p x_p$

- When we cannot fit the data well, we can add non-linear transformations of the features

- Feature map (or basis expansion)  $\phi: X \rightarrow \mathbb{R}^d$

$$y = \mathbf{w}^T \mathbf{x} \rightarrow y = \mathbf{w}^T \phi(\mathbf{x})$$

- E.g. Polynomial feature map: all polynomials up to degree  $d$  and all products

$$[1, x_1, \dots, x_p] \xrightarrow{\phi} [1, x_1, \dots, x_p, x_1^2, \dots, x_p^2, \dots, x_p^d, x_1 x_2, \dots, x_{p-1} x_p]$$

- Example with  $p = 1, d = 3$ :

$$y = w_0 + w_1 x_1 \xrightarrow{\phi} y = w_0 + w_1 x_1 + w_2 x_1^2 + w_3 x_1^3$$

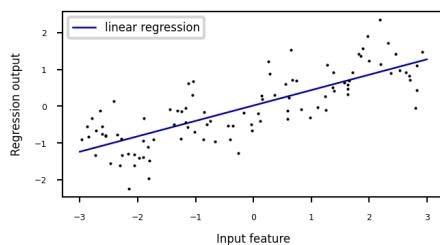
## Lecture 3: Kernelization

### Making linear models non-linear

Joaquin Vanschoren

### Ridge regression example

Weights: [0.418]

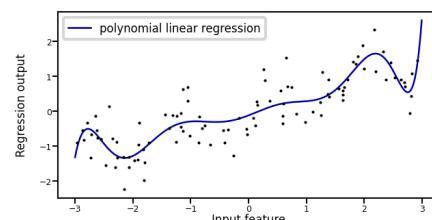


- Add all polynomials  $x^d$  up to degree 10 and fit again:

- e.g. use sklearn PolynomialFeatures

	$x0$	$x0^2$	$x0^3$	$x0^4$	$x0^5$	$x0^6$	$x0^7$	$x0^8$
0	-0.752759	0.566647	-0.426548	0.321088	-0.241702	0.181944	-0.136960	0.103098
1	2.704286	7.313162	19.776880	53.482337	144.631526	391.124988	1057.713767	2860.360362
2	1.391964	1.937563	2.697017	3.754150	5.225640	7.273901	10.125005	14.093639
3	0.591951	0.350406	0.207423	0.122784	0.072682	0.043024	0.025468	0.015076
4	-2.063888	4.259634	-8.791409	18.144485	-37.448187	77.288869	-159.515582	329.222321

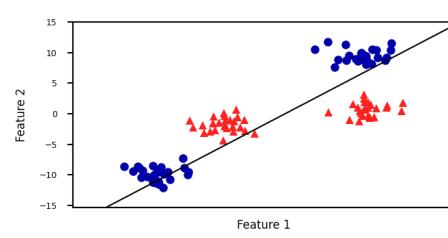
Weights: [ 0.643 0.297 -0.69 -0.264 0.41 0.096 -0.076 -0.014 0.004 0.001 ]



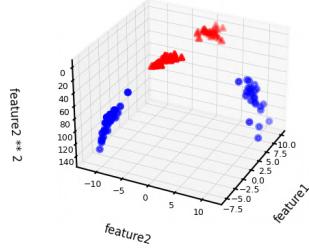
How expensive is this?

- You may need MANY dimensions to fit the data
  - Memory and computational cost
  - More weights to learn, more likely overfitting
- Ridge has a closed-form solution which we can compute with linear algebra:
 
$$w^* = (X^T X + \alpha I)^{-1} X^T Y$$
- Since  $X$  has  $n$  rows (examples), and  $d$  columns (features),  $X^T X$  has dimensionality  $d \times d$
- Hence Ridge is quadratic in the number of features,  $\mathcal{O}(d^2 n)$
- After the feature map  $\Phi$ , we get
 
$$w^* = (\Phi(X)^T \Phi(X) + \alpha I)^{-1} \Phi(X)^T Y$$
- Since  $\Phi$  increases  $d$  a lot,  $\Phi(X)^T \Phi(X)$  becomes huge

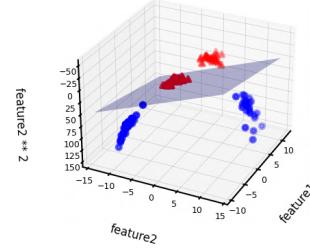
### Linear SVM example (classification)



We can add a new feature by taking the squares of feature1 values

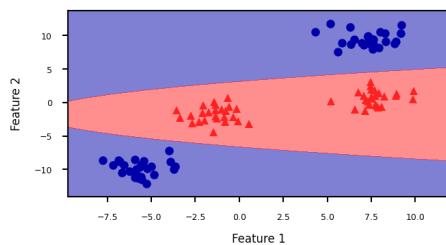


Now we can fit a linear model



As a function of the original features, the decision boundary is now a polynomial as well

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_2^2 > 0$$



## The kernel trick

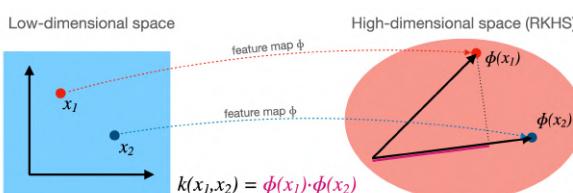
- Computations in explicit, high-dimensional feature maps are expensive
- For some feature maps, we can, however, compute *distances between points* cheaply
  - Without explicitly constructing the high-dimensional space at all
- Example: *quadratic* feature map for  $\mathbf{x} = (x_1, \dots, x_p)$ :  

$$\Phi(\mathbf{x}) = (x_1, \dots, x_p, x_1^2, \dots, x_p^2, \sqrt{2}x_1x_2, \dots, \sqrt{2}x_{p-1}x_p)$$
- A *kernel function* exists for this feature map to compute dot products  

$$k_{quad}(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j + (\mathbf{x}_i \cdot \mathbf{x}_j)^2$$
- Skip computation of  $\Phi(\mathbf{x}_i)$  and  $\Phi(\mathbf{x}_j)$  and compute  $k(\mathbf{x}_i, \mathbf{x}_j)$  directly

## Kernelization

- Kernel  $k$  corresponding to a feature map  $\Phi: k(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$
- Computes dot product between  $x_i, x_j$  in a high-dimensional space  $\mathcal{H}$ 
  - Kernels are sometimes called *generalized dot products*
  - $\mathcal{H}$  is called the *reproducing kernel Hilbert space* (RKHS)
- The dot product is a measure of the *similarity* between  $x_i, x_j$ 
  - Hence, a kernel can be seen as a similarity measure for high-dimensional spaces
- If we have a loss function based on dot products  $x_i \cdot x_j$  it can be *kernelized*
  - Simply replace the dot products with  $k(\mathbf{x}_i, \mathbf{x}_j)$

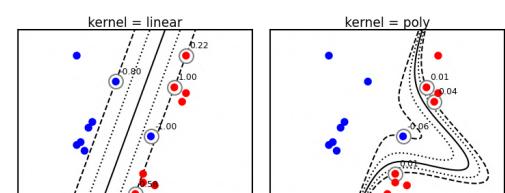


## Example: SVMs

- Linear SVMs (dual form, for  $l$  support vectors with dual coefficients  $a_i$  and classes  $y_i$ ):  

$$\mathcal{L}_{Dual}(a_i) = \sum_{i=1}^l a_i - \frac{1}{2} \sum_{i,j=1}^l a_i a_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$
- Kernelized SVM, using any existing kernel  $k$  we want:  

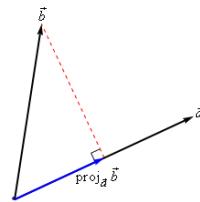
$$\mathcal{L}_{Dual}(a_i, k) = \sum_{i=1}^l a_i - \frac{1}{2} \sum_{i,j=1}^l a_i a_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$



## Linear kernel

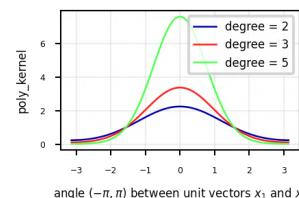
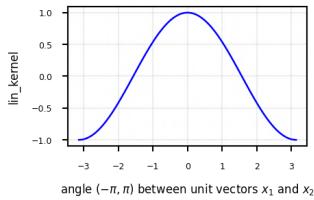
Which kernels exist?

- A (Mercer) kernel is any function  $k : X \times X \rightarrow \mathbb{R}$  with these properties:
    - Symmetry:  $k(\mathbf{x}_1, \mathbf{x}_2) = k(\mathbf{x}_2, \mathbf{x}_1) \quad \forall \mathbf{x}_1, \mathbf{x}_2 \in X$
    - Positive definite: the kernel matrix  $K$  is positive semi-definite
      - Intuitively,  $k(\mathbf{x}_1, \mathbf{x}_2) \geq 0$
  - The kernel matrix (or Gram matrix) for  $n$  points of  $\mathbf{x}_1, \dots, \mathbf{x}_n \in X$  is defined as:
- $$K = \mathbf{X}\mathbf{X}^T = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & \dots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}$$
- Once computed ( $\mathcal{O}(n^2)$ ), simply lookup  $k(\mathbf{x}_1, \mathbf{x}_2)$  for any two points
  - In practice, you can either supply a kernel function or precompute the kernel matrix



## Polynomial kernel

- Linear kernel between point  $(0,1)$  and another unit vector at angle  $\alpha$  (in radians)
  - Points with similar angles are deemed similar



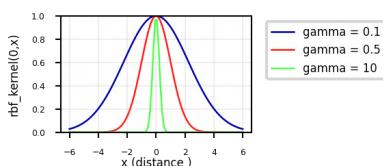
## RBF (Gaussian) kernel

- The Radial Basis Function (RBF) feature map builds the Taylor series expansion of  $e^x$

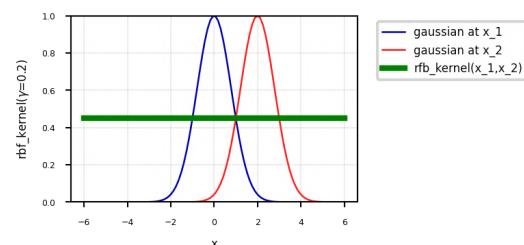
$$\Phi(x) = e^{-x^2/2\gamma^2} \left[ 1, \sqrt{\frac{1}{1!\gamma^2}}x, \sqrt{\frac{1}{2!\gamma^4}}x^2, \sqrt{\frac{1}{3!\gamma^6}}x^3, \dots \right]^T$$

- RBF (or Gaussian) kernel with kernel width  $\gamma \geq 0$ :

$$k_{RBF}(\mathbf{x}_1, \mathbf{x}_2) = \exp(-\gamma \|\mathbf{x}_1 - \mathbf{x}_2\|^2)$$

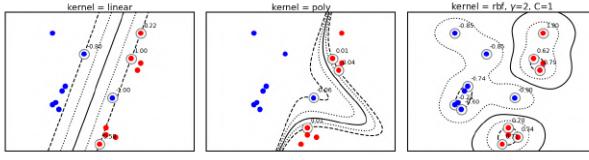


- The RBF kernel  $k_{RBF}(\mathbf{x}_1, \mathbf{x}_2) = \exp(-\gamma \|\mathbf{x}_1 - \mathbf{x}_2\|^2)$  does not use a dot product
  - It only considers the distance between  $\mathbf{x}_1$  and  $\mathbf{x}_2$
  - It's a *local kernel*: every data point only influences data points nearby
    - Linear and polynomial kernels are *global*: every point affects the whole space
  - Similarity depends on closeness of points and kernel width
    - value goes up for closer points and wider kernels (larger overlap)



## Kernelized SVMs in practice

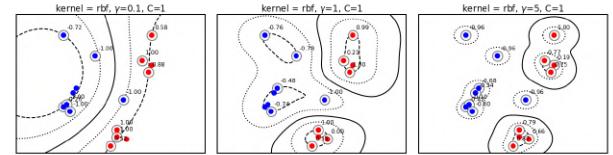
- You can use SVMs with any kernel to learn non-linear decision boundaries



## SVM with RBF kernel

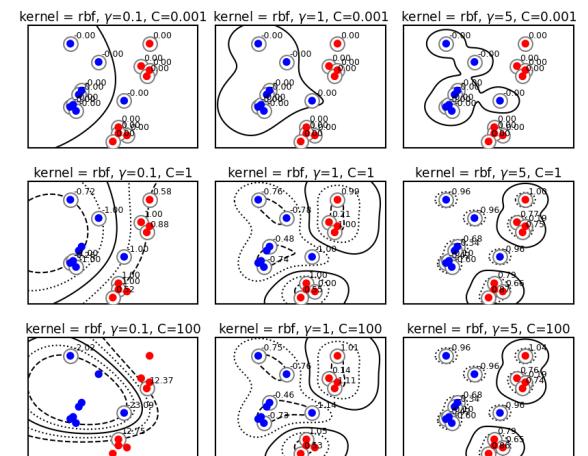
- Every support vector *locally* influences predictions, according to kernel width ( $\gamma$ )
- The prediction for test point  $\mathbf{u}$ : sum of the remaining influence of each support vector  

$$f(\mathbf{x}) = \sum_{i=1}^l a_i y_i k(\mathbf{x}_i, \mathbf{u})$$

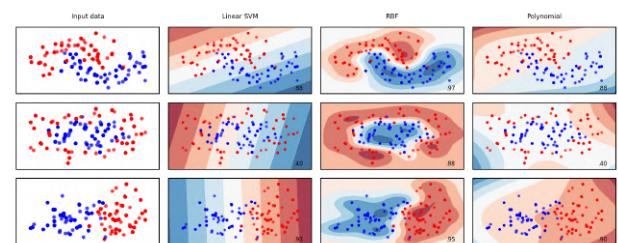


## Tuning RBF SVMs

- gamma (kernel width)
  - high values cause narrow Gaussians, more support vectors, overfitting
  - low values cause wide Gaussians, underfitting
- C (cost of margin violations)
  - high values punish margin violations, cause narrow margins, overfitting
  - low values cause wider margins, more support vectors, underfitting



## Kernel overview



## SVMs in practice

- C and gamma *always* need to be tuned
  - Interacting regularizers. Find a good C, then finetune gamma
- SVMs expect **all** features to be approximately on the same scale
  - Data needs to be scaled beforehand
- Allow to learn complex decision boundaries, even with few features
  - Work well on both low- and high-dimensional data
  - Especially good at small, high-dimensional data
- Hard to inspect, although support vectors can be inspected
- In sklearn, you can use `SVC` for classification with a range of kernels
  - `SVR` for regression

## Other kernels

- There are many more possible kernels
- If no kernel function exists, we can still *precompute* the kernel matrix
  - All you need is some similarity measure, and you can use SVMs
- Text kernels:
  - Word kernels: build a bag-of-words representation of the text (e.g. TFIDF)
    - Kernel is the inner product between these vectors
  - Subsequence kernels: sequences are similar if they share many sub-sequences
    - Build a kernel matrix based on pairwise similarities
- Graph kernels: Same idea (e.g. find common subgraphs to measure similarity)
- These days, deep learning embeddings are more frequently used

## The Representer Theorem

- We can kernelize many other loss functions as well
- The *Representer Theorem* states that if we have a loss function  $\mathcal{L}'$  with
  - $\mathcal{L}'$  an arbitrary loss function using some function  $f$  of the inputs  $\mathbf{x}$
  - $\mathcal{R}$  a (non-decreasing) regularization score (e.g. L1 or L2) and constant  $\lambda$
$$\mathcal{L}'(\mathbf{w}) = \mathcal{L}(y, f(\mathbf{x})) + \lambda \mathcal{R}(\|\mathbf{w}\|)$$
- Then the weights  $\mathbf{w}$  can be described as a linear combination of the training samples:
 
$$\mathbf{w} = \sum_{i=1}^n a_i y_i f(\mathbf{x}_i)$$
- Note that this is exactly what we found for SVMs:  $\mathbf{w} = \sum_{i=1}^l a_i y_i \mathbf{x}_i$
- Hence, we can also kernelize Ridge regression, Logistic regression, Perceptrons, Support Vector Regression, ...

## Kernelized Ridge regression

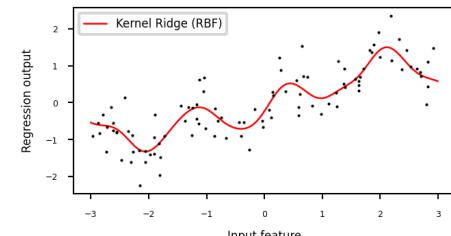
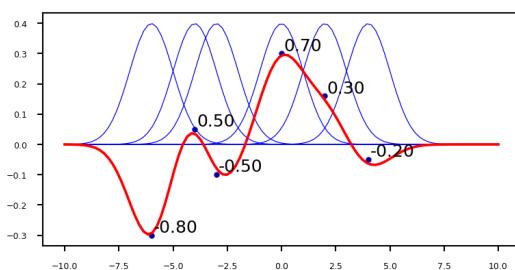
- The linear Ridge regression loss (with  $\mathbf{x}_0 = 1$ ):
 
$$\mathcal{L}_{Ridge}(\mathbf{w}) = \sum_{i=1}^n (y_i - \mathbf{w} \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|^2$$
- Filling in  $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$  yields the dual formulation:
 
$$\mathcal{L}_{Ridge}(\mathbf{w}) = \sum_{i=1}^n (y_i - \sum_{j=1}^n \alpha_j y_j \mathbf{x}_i \mathbf{x}_j)^2 + \lambda \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j$$
- Generalize  $\mathbf{x}_i \cdot \mathbf{x}_j$  to  $k(\mathbf{x}_i, \mathbf{x}_j)$ 

$$\mathcal{L}_{KernelRidge}(\alpha, k) = \sum_{i=1}^n (y_i - \sum_{j=1}^n \alpha_j y_j k(\mathbf{x}_i, \mathbf{x}_j))^2 + \lambda \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$

## Example of kernelized Ridge

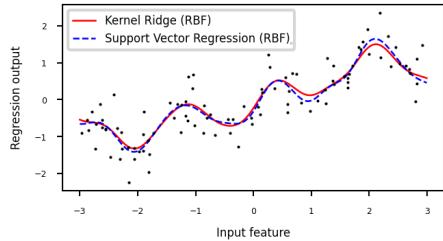
- Prediction (red) is now a linear combination of kernels (blue):  $y = \sum_{j=1}^n \alpha_j y_j k(\mathbf{x}, \mathbf{x}_j)$
- We learn a dual coefficient for each point

- Fitting our regression data with `KernelRidge`



## Other kernelized methods

- Same procedure can be done for logistic regression
- For perceptrons,  $\alpha \rightarrow \alpha + 1$  after every misclassification
- $$\mathcal{L}_{DualPerceptron}(x_i, k) = \max(0, y_i \sum_{j=1}^n \alpha_j y_j k(x_j, x_i))$$
- Support Vector Regression behaves similarly to Kernel Ridge



## Summary

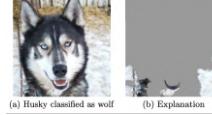
- Feature maps  $\Phi(x)$  transform features to create a higher-dimensional space
  - Allows learning non-linear functions or boundaries, but very expensive/slow
- For some  $\Phi(x)$ , we can compute dot products without constructing this space
  - Kernel trick:  $k(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$
  - Kernel  $k$  (generalized dot product) is a measure of similarity between  $x_i$  and  $x_j$
- There are many such kernels
  - Polynomial kernel:  $k_{poly}(x_1, x_2) = (\gamma(x_1 \cdot x_2) + c_0)^d$
  - RBF (Gaussian) kernel:  $k_{RBF}(x_1, x_2) = \exp(-\gamma||x_1 - x_2||^2)$
  - A kernel matrix can be precomputed using any similarity measure (e.g. for text, graphs,...)
- Any loss function where inputs appear only as dot products can be kernelized
  - E.g. Linear SVMs: simply replace the dot product with a kernel of choice
- The Representer theorem states which other loss functions can also be kernelized and how
  - Ridge regression, Logistic regression, Perceptrons,...

## Lecture 4: Model Selection

**Can I trust you?**  
Joaquin Vanschoren

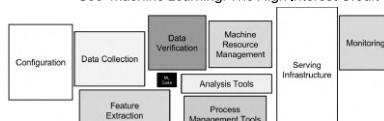
### Evaluation

- To know whether we can *trust* our method or system, we need to evaluate it.
- Model selection: choose between different models in a data-driven way.
  - If you cannot measure it, you cannot improve it.
- Convince others that your work is meaningful
  - Peers, leadership, clients, yourself(!)
- When possible, try to *interpret* what your model has learned
  - The signal your model found may just be an artifact of your biased data
  - See 'Why Should I Trust You?' by Marco Ribeiro et al.



## Designing Machine Learning systems

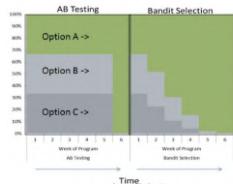
- Just running your favourite algorithm is usually not a great way to start
- Consider the problem: How to measure success? Are there costs involved?
  - Do you want to understand phenomena or do black box modelling?
- Analyze your model's mistakes. Don't just finetune endlessly.
  - Build early prototypes. Should you collect more, or additional data?
  - Should the task be reformulated?
- Overly complex machine learning systems are hard to maintain
  - See 'Machine Learning: The High Interest Credit Card of Technical Debt'



Only a small fraction of real-world ML systems is composed of the ML code

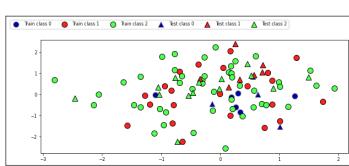
### Real world evaluations

- Evaluate predictions, but also how outcomes improve *because of them*
- Beware of feedback loops: predictions can influence future input data
  - Medical recommendations, spam filtering, trading algorithms,...
- Evaluate algorithms *in the wild*.
  - A/B testing: split users in groups, test different models in parallel
  - Bandit testing: gradually direct more users to the winning system



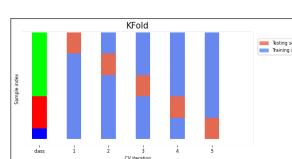
## Performance estimation techniques

- Always evaluate models as *if they are predicting future data*
- We do not have access to future data, so we pretend that some data is hidden
- Simplest way: the *holdout* (simple train-test split)
  - Randomly split data (and corresponding labels) into training and test set (e.g. 75%-25%)
  - Train (fit) a model on the training data, score on the test data



### K-fold Cross-validation

- Each random split can yield very different models (and scores)
  - e.g. all easy (or hard) examples could end up in the test set
- Split data into  $k$  equal-sized parts, called *folds*
  - Create  $k$  splits, each time using a different fold as the test set
- Compute  $k$  evaluation scores, aggregate afterwards (e.g. take the mean)
- Examine the score variance to see how *sensitive* (unstable) models are
- Large  $k$  gives better estimates (more training data), but is expensive



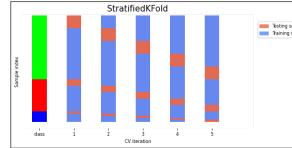
Can you explain this result?

```
kfold = KFold(n_splits=3)
cross_val_score(logistic_regression, iris.data, iris.target, cv=kfold)
```



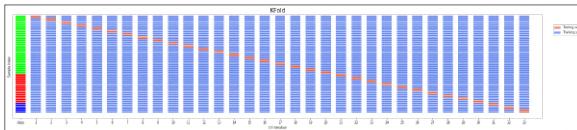
#### STRATIFIED K-FOLD CROSS-VALIDATION

- If the data is unbalanced, some classes have only few samples
- Likely that some classes are not present in the test set
- Stratification: proportions between classes are conserved in each fold
  - Order examples per class
  - Separate the samples of each class in  $k$  sets (strata)
  - Combine corresponding strata into folds



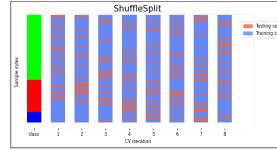
#### LEAVE-ONE-OUT CROSS-VALIDATION

- $k$  fold cross-validation with  $k$  equal to the number of samples
- Completely unbiased (in terms of data splits), but computationally expensive
- Actually generalizes less well towards unseen data
  - The training sets are correlated (overlap heavily)
  - Overfits on the data used for (the entire) evaluation
  - A different sample of the data can yield different results
- Recommended only for small datasets



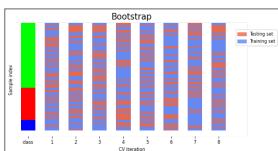
#### SHUFFLE-SPLIT CROSS-VALIDATION

- Shuffles the data, samples (`train_size`) points randomly as the training set
- Can also use a smaller (`test_size`), handy with very large datasets
- Never use if the data is ordered (e.g. time series)



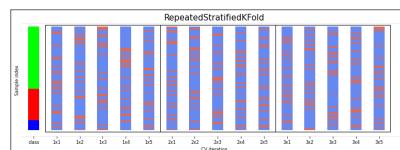
#### The Bootstrap

- Sample  $n$  (dataset size) data points, with replacement, as training set (the bootstrap)
  - On average, bootstraps include 66% of all data points (some are duplicates)
- Use the unsampled (out-of-bootstrap) samples as the test set
- Repeat  $k$  times to obtain  $k$  scores
- Similar to Shuffle-Split with `train_size=0.66, test_size=0.34` but without duplicates



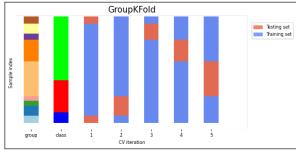
#### Repeated cross-validation

- Cross-validation is still biased in that the initial split can be made in many ways
- Repeated, or  $n$ -times- $k$ -fold cross-validation:
  - Shuffle data randomly, do  $k$ -fold cross-validation
  - Repeat  $n$  times, yields  $n$  times  $k$  scores
- Unbiased, very robust, but  $n$  times more expensive



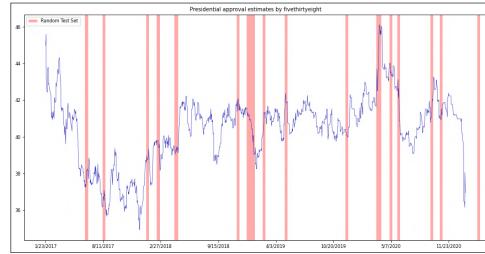
## Cross-validation with groups

- Sometimes the data contains inherent groups:
  - Multiple samples from same patient, images from same person,...
- Data from the same person may end up in the training *and* test set
- We want to measure how well the model generalizes to *other* people
- Make sure that data from one person are in *either* the train or test set
  - This is called *grouping* or *blocking*
  - Leave-one-subject-out cross-validation: test set for each subject/group



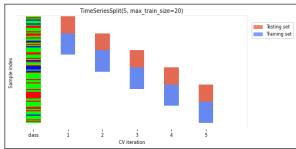
## Time series

When the data is ordered, random test sets are not a good idea



## TEST-THEN-TRAIN (PREquential EVALUATION)

- Every new sample is evaluated only once, then added to the training set
  - Can also be done in batches (of  $n$  samples at a time)
- `TimeSeriesSplit`
  - In the  $k$ th split, the first  $k$  folds are the train set and the  $(k+1)$ th fold as the test set
  - Often, a maximum training set size (or window) is used
    - more robust against concept drift (change in data over time)



## Choosing a performance estimation procedure

- No strict rules, only guidelines:
- Always use stratification for classification (sklearn does this by default)
  - Use holdout for very large datasets (e.g. >1.000.000 examples)
    - Or when learners don't always converge (e.g. deep learning)
  - Choose  $k$  depending on dataset size and resources
    - Use leave-one-out for very small datasets (e.g. <100 examples)
    - Use cross-validation otherwise
      - Most popular (and theoretically sound): 10-fold CV
      - Literature suggests 5x2-fold CV is better
  - Use grouping or leave-one-subject-out for grouped data
  - Use train-then-test for time series

## Evaluation Metrics for Classification

### Evaluation vs Optimization

- Each algorithm optimizes a given objective function (on the training data)
  - E.g. remember L2 loss in Ridge regression
- The choice of function is limited by what can be efficiently optimized
- However, we evaluate the resulting model with a score that makes sense in the real world
  - Percentage of correct predictions (on a test set)
  - The actual cost of mistakes (e.g. in money, time, lives,...)
- We also tune the algorithm's hyperparameters to maximize that score

### Binary classification

- We have a positive and a negative class
- 2 different kind of errors:
  - False Positive (type I error): model predicts positive while true label is negative
  - False Negative (type II error): model predicts negative while true label is positive
- They are not always equally important
  - Which side do you want to err on for a medical test?



## CONFUSION MATRICES

- We can represent all predictions (correct and incorrect) in a confusion matrix
  - $n \times n$  array ( $n$  is the number of classes)
  - Rows correspond to true classes, columns to predicted classes
  - Count how often samples belonging to a class  $C$  are classified as  $C$  or any other class.
  - For binary classification, we label these true negative (TN), true positive (TP), false negative (FN), false positive (FP)

	Predicted Neg	Predicted Pos
Actual Neg	TN	FP
Actual Pos	FN	TP

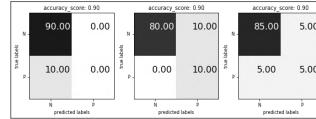
```
confusion_matrix(y_test, y_pred):
[[48 1]
 [5 85]]
```

## PREDICTIVE ACCURACY

- Accuracy can be computed based on the confusion matrix
- Not useful if the dataset is very imbalanced
- E.g. credit card fraud: is 99.99% accuracy good enough?

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (1)$$

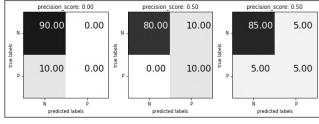
- 3 models: very different predictions, same accuracy:



## PRECISION

- Use when the goal is to limit FPs
  - Clinical trials: you only want to test drugs that really work
  - Search engines: you want to avoid bad search results

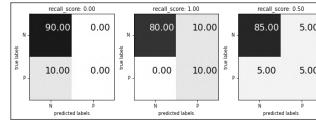
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2)$$



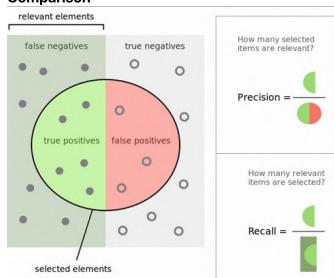
## RECALL

- Use when the goal is to limit FNs
  - Cancer diagnosis: you don't want to miss a serious disease
  - Search engines: You don't want to omit important hits
- Also known as sensitivity, hit rate, true positive rate (TPR)

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (3)$$



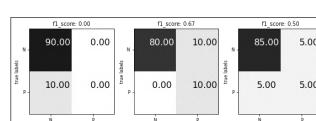
## Comparison



## F1-SCORE

- Trades off precision and recall:

$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \quad (4)$$



## Classification measure Zoo

		True condition			
		Condition positive	Condition negative	Precision = $\frac{\text{True positive}}{\text{True positive} + \text{False positive}}$	Accuracy (ACC) = $\frac{\text{True positive} + \text{True negative}}{\text{Total population}}$
Predicted condition	Total	True positive, Power	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\text{True positive}}{\text{True positive} + \text{False positive}}$	False discovery rate (FDR) = $\frac{\text{False positive}}{\text{True positive} + \text{False positive}}$
	Positive	False negative, Type II error	True negative	False omission rate (FOR), Type II error rate = $\frac{\text{False negative}}{\text{True negative}}$	Negative predictive value (NPV) = $\frac{\text{True negative}}{\text{True negative} + \text{False positive}}$
		True positive rate (TPR), Recall, Sensitivity, Probability of detection = $\frac{\text{True positive}}{\text{True positive} + \text{False negative}}$	False negative rate (FNR), Fall-out, Probability of false alarm = $\frac{\text{False positive}}{\text{True negative} + \text{False positive}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FNR}}$	Diagnostic ratio = $\text{LR}^+ / \text{LR}^-$
		False positive rate (FPR), Miss rate = $\frac{\text{False positive}}{\text{True negative}}$	True negative rate (TNR) = $\frac{\text{True negative}}{\text{True negative} + \text{False positive}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TPR}}$	F <sub>1</sub> score = $\frac{2}{\text{Precision} + \text{Recall}}$

[https://en.wikipedia.org/wiki/Precision\\_and\\_recall](https://en.wikipedia.org/wiki/Precision_and_recall)

## Multi-class classification

- Train models per class : one class viewed as positive, other(s) als negative, then average

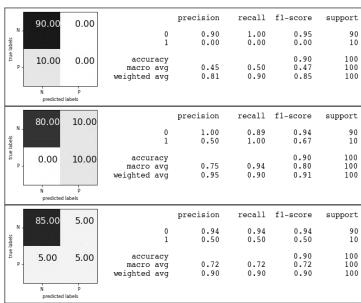
- micro-averaging: count total TP, FP, TN, FN (every sample equally important)
  - micro-precision, micro-recall, micro-F1, accuracy are all the same

$$\text{Precision: } \frac{\sum_{c=1}^C \text{TP}_c}{\sum_{c=1}^C \text{TP}_c + \sum_{c=1}^C \text{FP}_c} \xrightarrow{c=2} \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- macro-averaging: average of scores  $R(y_c, \hat{y}_c)$  obtained on each class
  - Preferable for imbalanced classes (if all classes are equally important)
  - macro-averaged recall is also called *balanced accuracy*

$$\frac{1}{C} \sum_{c=1}^C R(y_c, \hat{y}_c)$$

- weighted averaging ( $w_c$ : ratio of examples of class  $c$ , aka support):
 
$$\frac{1}{\sum_{c=1}^C w_c} \sum_{c=1}^C w_c R(y_c, \hat{y}_c)$$



## Other useful classification metrics

### Cohen's Kappa

- Measures 'agreement' between different models (aka inter-rater agreement)
- To evaluate a single model, compare it against a model that does random guessing
  - Similar to accuracy, but taking into account the possibility of predicting the right class by chance
- Can be weighted: different misclassifications given different weights
- 1: perfect prediction, 0: random prediction, -1: worse than random
- With  $p_0$  = accuracy, and  $p_e$  = accuracy of random classifier:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

### Matthews correlation coefficient

- Corrects for imbalanced data, alternative for balanced accuracy or AUROC
- 1: perfect prediction, 0: random prediction, -1: inverse prediction

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}$$

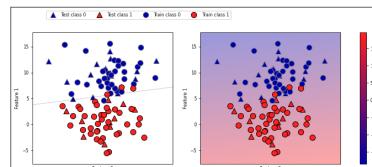
## Probabilistic evaluation

- Classifiers can often provide uncertainty estimates of predictions.
- Remember that linear models actually return a numeric value.
  - When  $\hat{y} < 0$ , predict class -1, otherwise predict class +1
  - $\hat{y} = w_0 * x_0 + w_1 * x_1 + \dots + w_p * x_p + b$
- In practice, you are often interested in how certain a classifier is about each class prediction (e.g. cancer treatments).
- Most learning methods can return at least one measure of confidence in their predictions.
  - Decision function: floating point value for each sample (higher: more confident)
  - Probability: estimated probability for each class

## The decision function

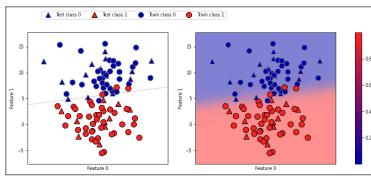
In the binary classification case, the return value of the decision function encodes how strongly the model believes a data point belongs to the "positive" class.

- Positive values indicate preference for the positive class.
- The range can be arbitrary, and can be affected by hyperparameters. Hard to interpret.



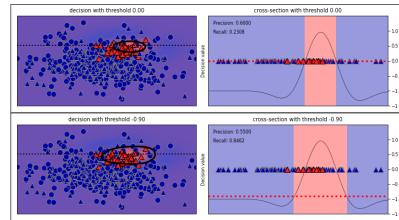
## Predicting probabilities

Some models can also return a *probability* for each class with every prediction. These sum up to 1. We can visualize them again. Note that the gradient looks different now.



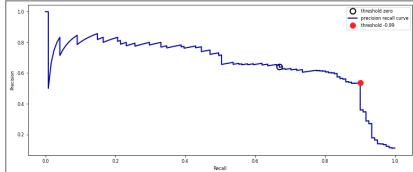
## Threshold calibration

- By default, we threshold at 0 for `decision_function` and 0.5 for `predict_proba`
- Depending on the application, you may want to threshold differently
  - Lower threshold yields fewer FN (better recall), more FP (worse precision), and vice-versa



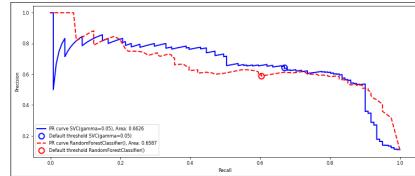
## Precision-Recall curve

- The best trade-off between precision and recall depends on your application
  - You can have arbitrary high recall, but you often want reasonable precision, too.
- Plotting precision against recall for *all possible thresholds* yields a **precision-recall curve**
  - Change the threshold until you find a sweet spot in the precision-recall trade-off
  - Often jagged at high thresholds, when there are few positive examples left



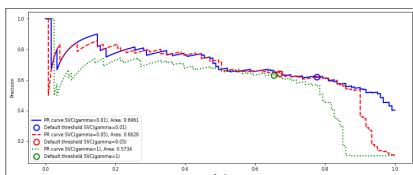
## MODEL SELECTION

- Some models can achieve trade-offs that others can't
- Your application may require very high recall (or very high precision)
  - Choose the model that offers the best trade-off, given your application
- The area under the PR curve (AUPRC) gives the **best overall model**



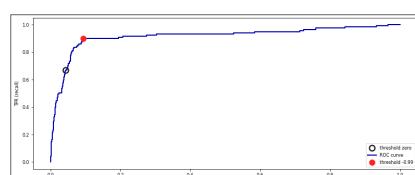
## HYPERPARAMETER EFFECTS

Of course, hyperparameters affect predictions and hence also the shape of the curve



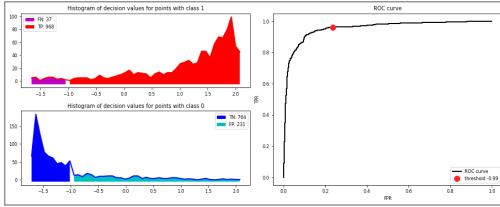
## Receiver Operating Characteristics (ROC)

- Trade off true positive rate  $TPR = \frac{TP}{TP+FN}$  with false positive rate  $FPR = \frac{FP}{FP+TN}$
- Plotting TPR against FPR for *all possible thresholds* yields a **Receiver Operating Characteristics curve**
  - Change the threshold until you find a sweet spot in the TPR-FPR trade-off
  - Lower thresholds yield higher TPR (recall), higher FPR, and vice versa



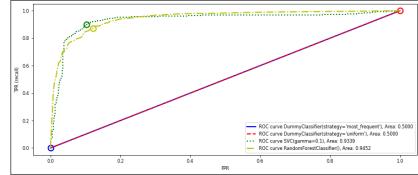
## VISUALIZATION

- Histograms show the amount of points with a certain decision value (for each class)
- $TPR = \frac{TP}{TP+FN}$  can be seen from the positive predictions (top histogram)
- $FPR = \frac{FP}{FP+TN}$  can be seen from the negative predictions (bottom histogram)



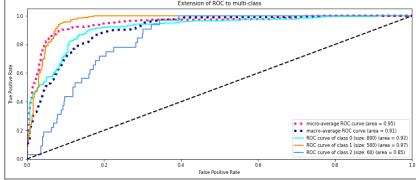
## MODEL SELECTION

- Again, some models can achieve trade-offs that others can't
- Your application may require minimizing FPR (low FP), or maximizing TPR (low FN)
- The area under the ROC curve (AUROC or AUC) gives the best overall model
  - Frequently used for evaluating models on imbalanced data
  - Random guessing ( $TPR=FPR$ ) or predicting majority class ( $TPR=FPR=1$ ): 0.5 AUC



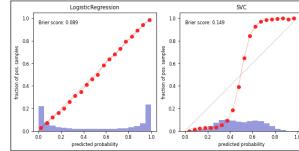
## MULTI-CLASS AUROC (OR AUPRC)

- We again need to choose between micro- or macro averaging TPR and FPR.
  - Micro-average if every sample is equally important (irrespective of class)
  - Macro-average if every class is equally important, especially for imbalanced data



## Model calibration

- For some models, the predicted uncertainty does not reflect the actual uncertainty
  - If a model is 90% sure that samples are positive, is it also 90% accurate on these?
- A model is called *calibrated* if the reported uncertainty actually matches how correct it is
  - Overfitted models also tend to be over-confident
  - LogisticRegression models are well calibrated since they learn probabilities
  - SVMs are not well calibrated. Biased towards points close to the decision boundary.



## BRIER SCORE

- You may want to select models based on how accurate the class confidences are.
- The **Brier score loss**: squared loss between predicted probability  $\hat{p}$  and actual outcome  $y$ 
  - Lower is better

$$\mathcal{L}_{Brier} = \frac{1}{n} \sum_{i=1}^n (\hat{p}_i - y_i)^2$$

```
Logistic Regression Brier score loss: 0.0322
SVM Brier score loss: 0.0795
```

## MODEL CALIBRATION TECHNIQUES

- We can post-process trained models to make them more calibrated.
- Fit a regression model (a calibrator) to map the model's outcomes  $f(x)$  to a calibrated probability in  $[0,1]$ 
  - $f(x)$  returns the decision values or probability estimates
  - $f_{calib}$  is fitted on the training data to map these to the correct outcome
    - Often an internal cross-validation with few folds is used
  - Multi-class models require one calibrator per class

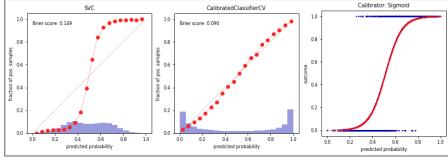
$$f_{calib}(f(x)) \approx p(y)$$

## Platt Scaling

- Calibrator is a logistic (sigmoid) function:

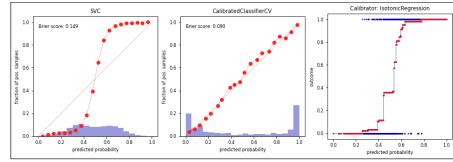
- Learn the weight  $w_1$  and bias  $w_0$  from data

$$f_{platt} = \frac{1}{1 + \exp(-w_1 f(x) - w_0)}$$



## Isotonic regression

- Maps input  $x_i$  to an output  $\hat{y}_i$  so that  $\hat{y}_i$  increases monotonically with  $x_i$  and minimizes  $\sum_i^n (y_i - \hat{y}_i)$
- Predictions are made by interpolating the predicted  $\hat{y}_i$
- Fit to minimize the loss between the uncalibrated predictions  $f(x)$  and the actual labels
- Corrects any monotonic distortion, but tends to overfit on small samples



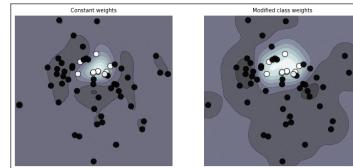
## Cost-sensitive classification (dealing with imbalance)

- In the real worlds, different kinds of misclassification can have different costs
  - Misclassifying certain classes can be more costly than others
  - Misclassifying certain samples can be more costly than others
- Cost-sensitive resampling: resample (or reweight) the data to represent real-world expectations
  - oversample minority classes (or undersample majority) to 'correct' imbalance
  - increase weight of misclassified samples (e.g. in boosting)
  - decrease weight of misclassified (noisy) samples (e.g. in model compression)

## Class weighting

- If some classes are more important than others, we can give them more weight
  - E.g. for imbalanced data, we can give more weight to minority classes
- Most classification models can include it in their loss function and optimize for it
  - E.g. Logistic regression: add a class weight  $w_c$  in the log loss function

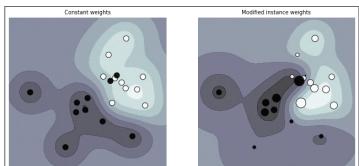
$$\mathcal{L}_{log}(\mathbf{w}) = - \sum_{c=1}^C \mathbf{w}_c \sum_{n=1}^N p_{n,c} \log(q_{n,c})$$



## Instance weighting

- If some training instances are important to get right, we can give them more weight
  - E.g. when some examples are from groups underrepresented in the data
- These are passed during training (fit), and included in the loss function
  - E.g. Logistic regression: add a instance weight  $w_n$  in the log loss function

$$\mathcal{L}_{log}(\mathbf{w}) = - \sum_{c=1}^C \sum_{n=1}^N w_n p_{n,c} \log(q_{n,c})$$

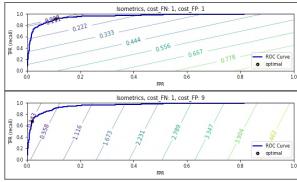


## Cost-sensitive algorithms

- Cost-sensitive algorithms
  - If misclassification cost of some classes is higher, we can give them higher weights
  - Some support cost matrix  $C$ : costs  $c_{i,j}$  for every possible type of error
- Cost-sensitive ensembles: convert cost-insensitive classifiers into cost-sensitive ones
  - MetaCost: Build a model (ensemble) to learn the class probabilities  $P(j|x)$ 
    - Relabel training data to minimize expected cost:  $\operatorname{argmin}_i \sum_j P_j(x)c_{i,j}$
    - Accuracy may decrease but cost decreases as well.
  - AdaCost: Boosting with reweighting instances to reduce costs

### Tuning the decision threshold

- If every FP or FN has a certain cost, we can compute the total cost for a given model:  
 $\text{total cost} = \text{FPR} * \text{cost}_{FP} * \text{ratio}_{pos} + (1 - \text{TPR}) * \text{cost}_{FN} * (1 - \text{ratio}_{pos})$
- This yields different *isometrics* (lines of equal cost) in ROC space
- Optimal threshold is the point on the ROC curve where cost is minimal (line search)



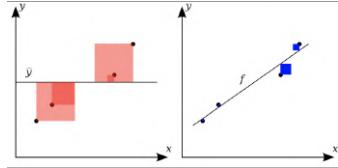
### Regression metrics

- Most commonly used are
- mean squared error:  $\frac{\sum_i (y_{\text{pred},i} - y_{\text{actual},i})^2}{n}$ 
    - root mean squared error (RMSE) often used as well
  - mean absolute error:  $\frac{\sum_i |y_{\text{pred},i} - y_{\text{actual},i}|}{n}$ 
    - Less sensitive to outliers and large errors



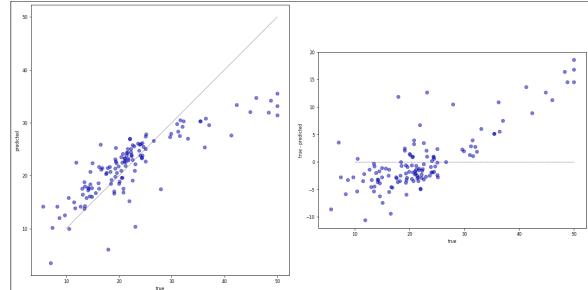
### R squared

- $R^2 = 1 - \frac{\sum_i (y_{\text{pred},i} - y_{\text{actual},i})^2}{\sum_i (y_{\text{mean}} - y_{\text{actual},i})^2}$
- Ratio of variation explained by the model / total variation
- Between 0 and 1, but *negative* if the model is worse than just predicting the mean
- Easier to interpret (higher is better).



### Visualizing regression errors

- Prediction plot (left): predicted vs actual target values
- Residual plot (right): residuals vs actual target values
  - Over- and underpredictions can be given different costs



### Bias-Variance decomposition

- Evaluate the same algorithm multiple times on different random samples of the data
- Two types of errors can be observed:
  - Bias error: systematic error, independent of the training sample
    - These points are predicted (*equally*) wrong every time
  - Variance error: error due to variability of the model w.r.t. the training sample
    - These points are sometimes predicted accurately, sometimes inaccurately

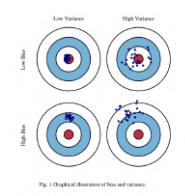


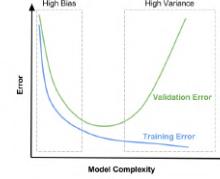
Fig. 1 Graphical illustration of bias and variance.

## Computing bias and variance error

- Take 100 or more bootstraps (or shuffle-splits)
- Regression: for each data point  $x$ :
  - $bias(x)^2 = (x_{true} - mean(x_{predicted}))^2$
  - $variance(x) = var(x_{predicted})$
- Classification: for each data point  $x$ :
  - $bias(x) = \text{misclassification ratio}$
  - $variance(x) = (1 - (P(class_1)^2 + P(class_2)^2))/2$
  - $P(class_i)$  is ratio of class  $i$  predictions
- Total bias:  $\sum_x bias(x)^2 * w_x$ : the percentage of times  $x$  occurs in the test sets
- Total variance:  $\sum_x variance(x) * w_x$

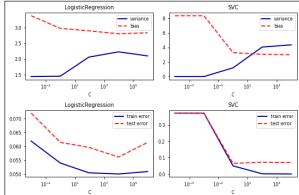
## Bias and variance, underfitting and overfitting

- High variance means that you are likely overfitting
  - Use more regularization or use a simpler model
- High bias means that you are likely underfitting
  - Do less regularization or use a more flexible/complex model
- Ensembling techniques (see later) reduce bias or variance directly
  - Bagging (e.g. RandomForests) reduces variance, Boosting reduces bias

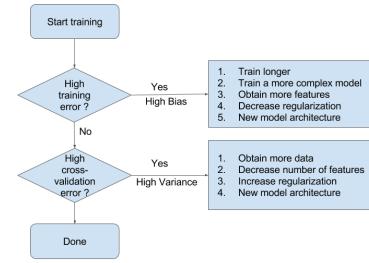


## Understanding under- and overfitting

- Regularization reduces variance error (increases stability of predictions)
  - But too much increases bias error (inability to learn 'harder' points)
- High regularization (left side): Underfitting, high bias error, low variance error
  - High training error and high test error
- Low regularization (right side): Overfitting, low bias error, high variance error
  - Low training error and higher test error

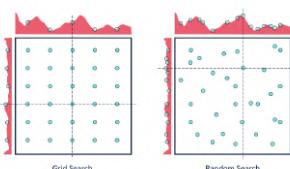


## Summary Flowchart (by Andrew Ng)

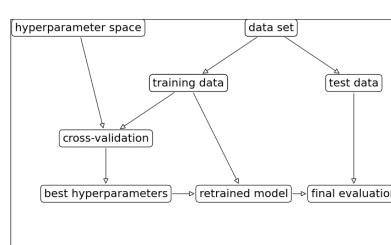


## Hyperparameter tuning

- There exists a huge range of techniques to tune hyperparameters. The simplest:
  - Grid search: Choose a range of values for every hyperparameter, try every combination
    - Doesn't scale to many hyperparameters (combinatorial explosion)
  - Random search: Choose random values for all hyperparameters, iterate  $n$  times
    - Better, especially when some hyperparameters are less important
- Many more advanced techniques exist, see lecture on Automated Machine Learning



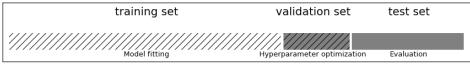
- First, split the data in training and test sets (outer split)
- Split up the training data again (inner cross-validation)
  - Generate hyperparameter configurations (e.g. random/grid search)
  - Evaluate all configurations on all inner splits, select the best one (on average)
- Retrain best configurations on full training set, evaluate on held-out test data



### Nested cross-validation

- Simplest approach: single outer split and single inner split (shown below)
- Risk of over-tuning hyperparameters on specific train-test split
  - Only recommended for very large datasets
- Nested cross-validation:
  - Outer loop: split full dataset in  $k_1$  training and test splits
  - Inner loop: split training data into  $k_2$  train and validation sets
- This yields  $k_1$  scores for  $k_1$  possibly different hyperparameter settings
  - Average score is the expected performance of the tuned model
- To use the model in practice, retune on the **entire** dataset

```
hps = {'C': expon(scale=100), 'gamma': expon(scale=.1)}
scores = cross_val_score(RandomizedSearchCV(SVC()), hps, cv=3), X, y, cv=5)
```



### Summary

- Split the data into training and test sets according to the application
  - Holdout only for large datasets, cross-validation for smaller ones
  - For classification, always use stratification
  - Grouped or ordered data requires special splitting
- Choose a metric that fits your application
  - E.g. precision to avoid false positives, recall to avoid false negatives
- Calibrate the decision threshold to fit your application
  - ROC curves or Precision-Recall curves can help to find a good tradeoff
- If possible, include the actual or relative costs of misclassifications
  - Class weighting, instance weighting, ROC isometrics can help
  - Be careful with imbalanced or unrepresentative datasets
- When using the predicted probabilities in applications, calibrate the models
- Always tune the most important hyperparameters
  - Manual tuning: Use insight and train-test scores for guidance
  - Hyperparameter optimization: be careful not to over-tune



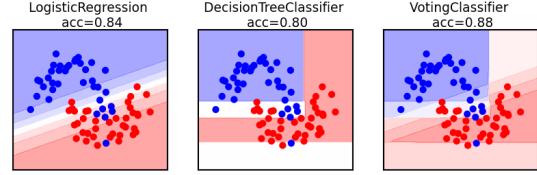
## Ensemble learning

- If different models make different mistakes, can we simply average the predictions?
- Voting Classifier: gives every model a vote on the class label
  - Hard vote: majority class wins (class order breaks ties)
  - Soft vote: sum class probabilities  $p_{m,c}$  over  $M$  models:  $\operatorname{argmax}_c \sum_{m=1}^M w_m p_{m,c}$
  - Classes can get different weights  $w_c$  (default:  $w_c = 1$ )

# Lecture 5. Ensemble Learning

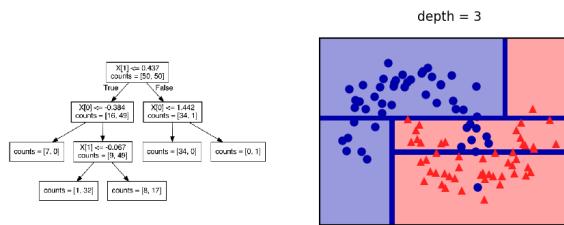
Crowd intelligence

Joaquin Vanschoren



## Decision trees (recap)

- Why does this work?
  - Different models may be good at different 'parts' of data (even if they underfit)
    - Individual mistakes can be 'averaged out' (especially if models overfit)
- Which models should be combined?
- Bias-variance analysis teaches us that we have two options:
  - If model underfits (high bias, low variance): combine with other low-variance models
    - Need to be different: 'experts' on different parts of the data
  - Bias reduction. Can be done with **Boosting**
- If model overfits (low bias, high variance): combine with other low-bias models
  - Need to be different: individual mistakes must be different
  - Variance reduction. Can be done with **Bagging**
- Models must be uncorrelated but good enough (otherwise the ensemble is worse)
- We can also learn how to combine the predictions of different models: **Stacking**

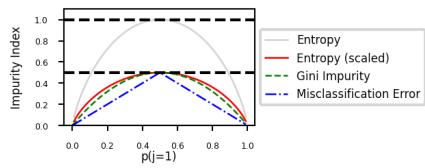


## Evaluation (loss function for classification)

- Every leaf predicts a class probability  $\hat{p}_c$  = the relative frequency of class  $c$
- Leaf impurity measures (splitting criteria) for  $L$  leafs, leaf  $l$  has data  $X_l$ :

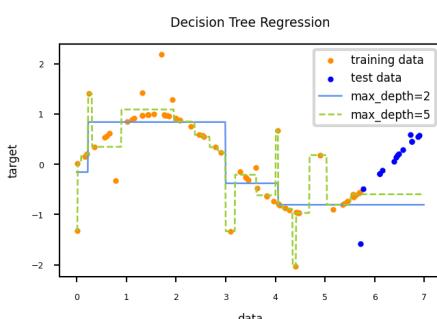
- Gini-Index:  $Gini(X_l) = \sum_{c \neq c'} \hat{p}_c \hat{p}_{c'}$
- Entropy (more expensive):  $E(X_l) = -\sum_{c \neq c'} \hat{p}_c \log_2 \hat{p}_c$
- Best split maximizes information gain (idem for Gini index)

$$Gain(X, X_i) = E(X) - \sum_{l=1}^L \frac{|X_{i=l}|}{|X_i|} E(X_{i=l})$$



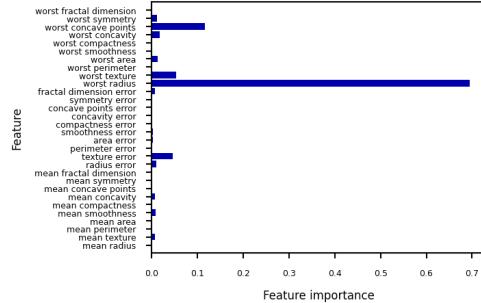
## Regression trees

- Every leaf predicts the mean target value  $\mu$  of all points in that leaf
- Choose the split that minimizes squared error of the leaves:  $\sum_{x_i \in L} (y_i - \mu)^2$
- Yields non-smooth step-wise predictions, cannot extrapolate



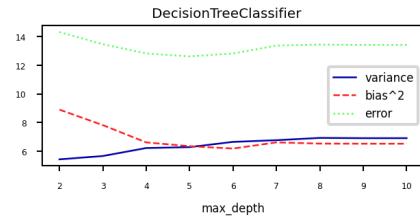
## Impurity/Entropy-based feature importance

- We can measure the importance of features (to the model) based on
  - Which features we split on
  - How high up in the tree we split on them (first splits are more important)



## Under- and overfitting

- We can easily control the (maximum) depth of the trees as a hyperparameter
- Bias-variance analysis:
  - Shallow trees have high bias but very low variance (underfitting)
  - Deep trees have high variance but low bias (overfitting)
- Because we can easily control their complexity, they are ideal for ensembling
  - Deep trees: keep low bias, reduce variance with **Bagging**
  - Shallow trees: keep low variance, reduce bias with **Boosting**

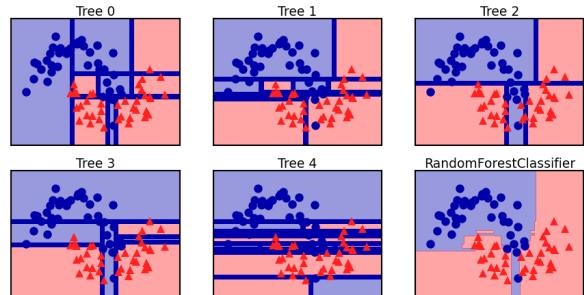


## Bagging (Bootstrap Aggregating)

- Obtain different models by training the *same* model on *different training samples*
  - Reduce overfitting by averaging out individual predictions (variance reduction)
- In practice: take  $I$  bootstrap samples of your data, train a model on each bootstrap
  - Higher  $I$ : more models, more smoothing (but slower training and prediction)
- Base models should be unstable: different training samples yield different models
  - E.g. very deep decision trees, or even randomized decision trees
  - Deep Neural Networks can also benefit from bagging (deep ensembles)
- Prediction by averaging predictions of base models
  - Soft voting for classification (possibly weighted)
  - Mean value for regression
- Can produce uncertainty estimates as well
  - By combining class probabilities of individual models (or variances for regression)

## Random Forests

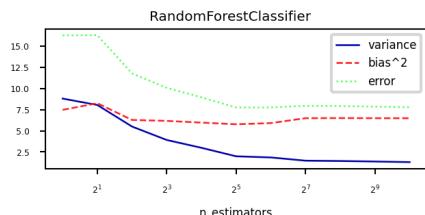
- Uses *randomized trees* to make models even less correlated (more unstable)
  - At every split, only consider `max_features` features, randomly selected
- Extremely randomized trees: considers 1 random threshold for random set of features (faster)



## In practice

### Effect on bias and variance

- Increasing the number of models (trees) decreases variance (less overfitting)
- Bias is mostly unaffected, but will increase if the forest becomes too large (oversmoothing)



- Different implementations can be used. E.g. in scikit-learn:

- `BaggingClassifier`: Choose your own base model and sampling procedure
- `RandomForestClassifier`: Default implementation, many options
- `ExtraTreesClassifier`: Uses extremely randomized trees

- Most important parameters:

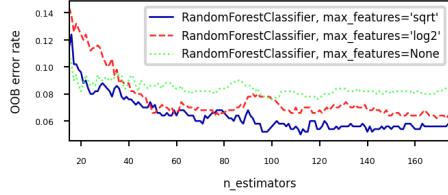
- `n_estimators` (>100, higher is better, but diminishing returns)
  - Will start to underfit (bias error component increases slightly)
- `max_features`
  - Defaults:  $\sqrt{p}$  for classification,  $\log_2(p)$  for regression
  - Set smaller to reduce space/time requirements
- parameters of trees, e.g. `max_depth`, `min_samples_split`, ...
  - Prepruning useful to reduce model size, but don't overdo it

- Easy to parallelize (set `n_jobs` to -1)

- Fix `random_state` (bootstrap samples) for reproducibility

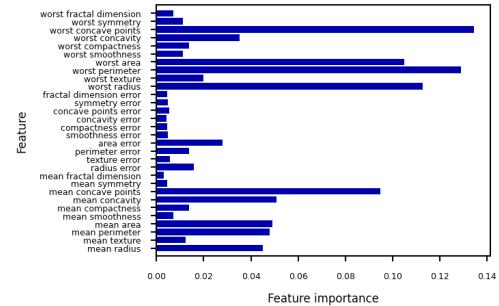
## Out-of-bag error

- RandomForests don't need cross-validation: you can use the out-of-bag (OOB) error
- For each tree grown, about 33% of samples are out-of-bag (OOB)
  - Remember which are OOB samples for every model, do voting over these
- OOB error estimates are great to speed up model selection
  - As good as CV estimates, although slightly pessimistic
- In scikit-learn: `oob_error = 1 - clf.oob_score_`



## Feature importance

- RandomForests provide more reliable feature importances, based on many alternative hypotheses (trees)



## Other tips

- Model calibration
  - RandomForests are poorly calibrated.
  - Calibrate afterwards (e.g. isotonic regression) if you aim to use probabilities
- Warm starting
  - Given an ensemble trained for  $J$  iterations, you can simply add more models later
  - You *warm start* from the existing model instead of re-starting from scratch
  - Can be useful to train models on new, closely related data
    - Not ideal if the data batches change over time (concept drift)
    - Boosting is more robust against this (see later)

## Strength and weaknesses

- RandomForest are among most widely used algorithms:
  - Don't require a lot of tuning
  - Typically very accurate
  - Handles heterogeneous features well (trees)
  - Implicitly selects most relevant features
- Downsides:
  - less interpretable, slower to train (but parallelizable)
  - don't work well on high dimensional sparse data (e.g. text)

## Adaptive Boosting (AdaBoost)

- Obtain different models by *reweighting* the training data every iteration
  - Reduce underfitting by focusing on the 'hard' training examples
- Increase weights of instances misclassified by the ensemble, and vice versa
- Base models should be simple so that different instance weights lead to different models
  - Underfitting models: decision stumps (or very shallow trees)
  - Each is an 'expert' on some parts of the data
- Additive model: Predictions at iteration  $I$  are sum of base model predictions
  - In AdaBoost, also the models each get a unique weight  $w_i$
$$f_I(\mathbf{x}) = \sum_{i=1}^I w_i g_i(\mathbf{x})$$
- AdaBoost minimizes exponential loss. For instance-weighted error  $\varepsilon$ :
 
$$\mathcal{L}_{Exp} = \sum_{n=1}^N e^{\varepsilon(f_I(\mathbf{x}))}$$
- By deriving  $\frac{\partial \mathcal{L}}{\partial w_i}$  you can find that optimal  $w_i = \frac{1}{2} \log\left(\frac{1-\varepsilon}{\varepsilon}\right)$

## AdaBoost algorithm

- Initialize sample weights:  $s_{n,0} = \frac{1}{N}$
- Build a model (e.g. decision stumps) using these sample weights
- Give the model a weight  $w_i$  related to its weighted error rate  $\varepsilon$ 

$$w_i = \lambda \log\left(\frac{1-\varepsilon}{\varepsilon}\right)$$
  - Good trees get more weight than bad trees
  - Logit function maps error  $\varepsilon$  from  $[0,1]$  to weight in  $[-\infty, \infty]$  (use small minimum error)
  - Learning rate  $\lambda$  (shrinkage) decreases impact of individual classifiers
    - Small updates are often better but requires more iterations
- Update the sample weights
  - Increase weight of incorrectly predicted samples:  $s_{n,i+1} = s_{n,i} e^{w_i}$
  - Decrease weight of correctly predicted samples:  $s_{n,i+1} = s_{n,i} e^{-w_i}$
  - Normalize weights to add up to 1
- Repeat for  $I$  iterations

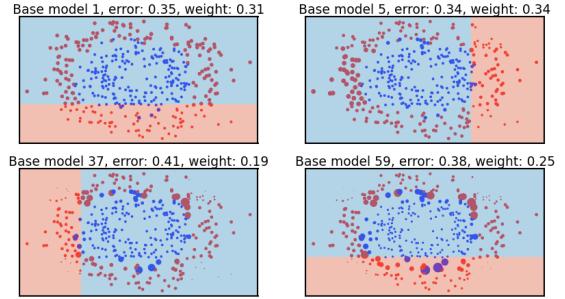
## AdaBoost in action

- Size of the samples represents sample weight
- Background shows the latest tree's predictions

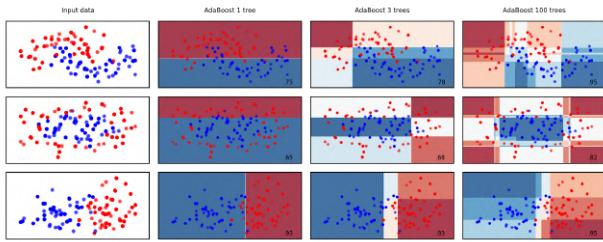
### AdaBoost variants

- Discrete AdaBoost: error rate  $\varepsilon$  is simply the error rate (1-Accuracy)
- Real AdaBoost:  $\varepsilon$  is based on predicted class probabilities  $\hat{p}_c$  (better)
- AdaBoost for regression:  $\varepsilon$  is either linear ( $|y_i - \hat{y}_i|$ ), squared ( $(y_i - \hat{y}_i)^2$ ), or exponential loss
- GentleBoost: adds a bound on model weights  $w_i$
- LogitBoost: Minimizes logistic loss instead of exponential loss

$$\mathcal{L}_{\text{Logistic}} = \sum_{n=1}^N \log(1 + e^{\varepsilon(f_n(\mathbf{x}))})$$

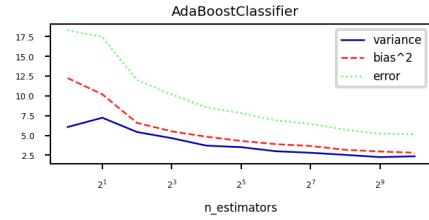


### Examples



### Bias-Variance analysis

- AdaBoost reduces bias (and a little variance)
  - Boosting is a *bias reduction* technique
- Boosting too much will eventually increase variance



## Gradient Boosting

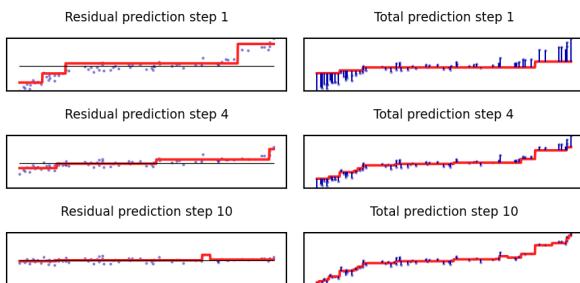
- Ensemble of models, each fixing the remaining mistakes of the previous ones
  - Each iteration, the task is to predict the *residual error* of the ensemble
- Additive model: Predictions at iteration  $I$  are sum of base model predictions
  - Learning rate (or *shrinkage*)  $\eta$ : small updates work better (reduces variance)
- The *pseudo-residuals*  $r_i$  are computed according to differentiable loss function
  - E.g. least squares loss for regression and log loss for classification
  - Gradient descent: *predictions* get updated step by step until convergence
- Base models  $g_i$  should be low variance, but flexible enough to predict residuals accurately
  - E.g. decision trees of depth 2-5

### Gradient Boosting Trees (Regression)

- Base models are regression trees, loss function is square loss:  $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$
- The pseudo-residuals are simply the prediction errors for every sample:
 
$$r_i = -\frac{\partial \mathcal{L}}{\partial \hat{y}} = -2 * \frac{1}{2}(y_i - \hat{y}_i) * (-1) = y_i - \hat{y}_i$$
- Initial model  $g_0$  simply predicts the mean of  $y$
- For iteration  $m = 1..M$ :
  - For all samples  $i=1..n$ , compute pseudo-residuals  $r_i = y_i - \hat{y}_i$
  - Fit a new regression tree model  $g_m(\mathbf{x})$  to  $r_i$ 
    - In  $g_m(\mathbf{x})$ , each leaf predicts the mean of all its values
  - Update ensemble predictions  $\hat{y} = g_0(\mathbf{x}) + \sum_{m=1}^M \eta \cdot g_m(\mathbf{x})$
- Early stopping (optional): stop when performance on validation set does not improve for  $nr$  iterations

### Gradient Boosting Regression in action

- Residuals quickly drop to (near) zero



### GradientBoosting Algorithm (Classification)

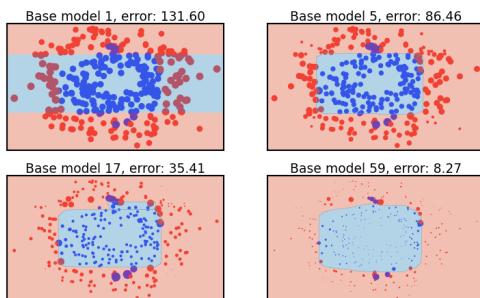
- Base models are *regression trees*, predict probability of positive class  $p$ 
  - For multi-class problems, train one tree per class
- Use (binary) log loss, with true class  $y_i \in 0, 1$ :  

$$\mathcal{L}_{\text{log}} = -\sum_{i=1}^N [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$
- The pseudo-residuals are simply the difference between true class and predicted  $p$ :  

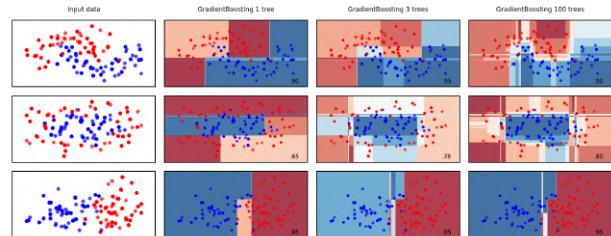
$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial \mathcal{L}}{\partial \log(p_i)} = y_i - p_i$$
- Initial model  $g_0$  predicts  $p = \log(\frac{\# \text{positives}}{\# \text{negatives}})$
- For iteration  $m = 1..M$ :
  - For all samples  $i=1..n$ , compute pseudo-residuals  $r_i = y_i - p_i$
  - Fit a new regression tree model  $g_m(\mathbf{x})$  to  $r_i$ 
    - In  $g_m(\mathbf{x})$ , each leaf predicts  $\frac{\sum r_i}{\sum_i p_i (1-p_i)}$
  - Update ensemble predictions  $\hat{y} = g_0(\mathbf{x}) + \sum_{m=1}^M \eta \cdot g_m(\mathbf{x})$
- Early stopping (optional): stop when performance on validation set does not improve for  $nr$  iterations

### Gradient Boosting Classification in action

- Size of the samples represents the residual weights: most quickly drop to (near) zero

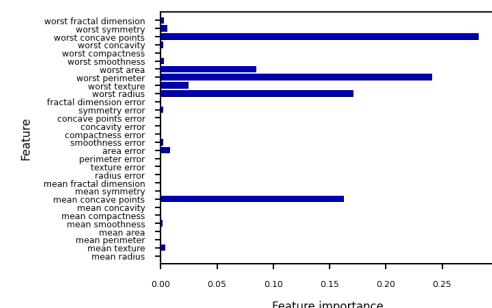


### Examples



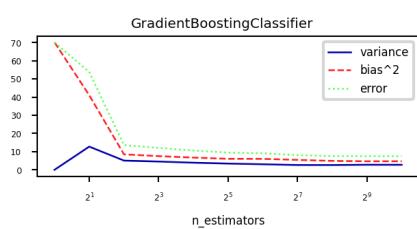
### Feature importance

- Gradient Boosting also provide feature importances, based on many trees
- Compared to RandomForests, the trees are smaller, hence more features have zero importance



### Bias-variance analysis

- Gradient Boosting is very effective at reducing bias error
- Boosting too much will eventually increase variance



## Gradient Boosting: strengths and weaknesses

- Among the most powerful and widely used models
- Work well on heterogeneous features and different scales
- Typically better than random forests, but requires more tuning, longer training
- Does not work well on high-dimensional sparse data

## Main hyperparameters:

- `n_estimators`: Higher is better, but will start to overfit
- `learning_rate`: Lower rates mean more trees are needed to get more complex models
  - Set `n_estimators` as high as possible, then tune `learning_rate`
  - Or, choose a `learning_rate` and use early stopping to avoid overfitting
- `max_depth`: typically kept low (<5), reduce when overfitting
- `max_features`: can also be tuned, similar to random forests
- `n_iter_no_change`: early stopping: algorithm stops if improvement is less than a certain tolerance `tol` for more than `n_iter_no_change` iterations.

## Extreme Gradient Boosting (XGBoost)

- Faster version of gradient boosting: allows more iterations on larger datasets
- Normal regression trees: split to minimize squared loss of leaf predictions
  - XGBoost trees only fit residuals: split so that residuals in leaf are more similar
- Don't evaluate every split point, only  $q$  quantiles per feature (binning)
  - $q$  is hyperparameter (`sketch_eps`, default 0.03)
- For large datasets, XGBoost uses approximate quantiles
  - Can be parallelized (multicore) by chunking the data and combining histograms of data
  - For classification, the quantiles are weighted by  $p(1 - p)$
- Gradient descent sped up by using the second derivative of the loss function
- Strong regularization by pre-pruning the trees
- Column and row are randomly subsampled when computing splits
- Support for out-of-core computation (data compression in RAM, sharding...)

## XGBoost in practice

- Not part of scikit-learn, but `HistGradientBoostingClassifier` is similar
  - binning, multicore,...
- The `xgboost` python package is sklearn-compatible
  - Install separately, `conda install -c conda-forge xgboost`
  - Allows learning curve plotting and warm-starting
- Further reading:
  - [XGBoost Documentation](#)
  - [Paper](#)
  - [Video](#)

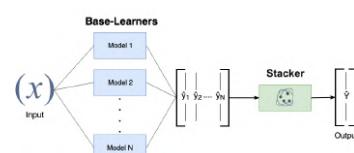
## LightGBM

Another fast boosting technique

- Uses *gradient-based sampling*
  - use all instances with large gradients/residuals (e.g. 10% largest)
  - randomly sample instances with small gradients, ignore the rest
  - intuition: samples with small gradients are already well-trained.
  - requires adapted information gain criterion
- Does smarter encoding of categorical features

## Stacking

- Choose  $M$  different base-models, generate predictions
- Stacker (meta-model) learns mapping between predictions and correct label
  - Can also be repeated: multi-level stacking
  - Popular stackers: linear models (fast) and gradient boosting (accurate)
- Cascade stacking: adds base-model predictions as extra features
- Models need to be sufficiently different, be experts at different parts of the data
- Can be very accurate, but also very slow to predict



## CatBoost

Another fast boosting technique

- Optimized for categorical variables
  - Uses bagged and smoothed version of target encoding
- Uses symmetric trees: same split for all nodes on a given level aka
  - Can be much faster
- Allows monotonicity constraints for numeric features
  - Model must be a non-decreasing function of these features
- Lots of tooling (e.g. GPU training)

## Algorithm overview

Name	Representation	Loss function	Optimization	Regularization
Classification trees	Decision tree	Entropy / Gini index	Hunt's algorithm	Tree depth,...
Regression trees	Decision tree	Square loss	Hunt's algorithm	Tree depth,...
RandomForest	Ensemble of randomized trees	Entropy / Gini / Square	(Bagging)	Number/depth of trees,...
AdaBoost	Ensemble of stumps	Exponential loss	Greedy search	Number/depth of trees,...
GradientBoostingRegression	Ensemble of regression trees	Square loss	Gradient descent	Number/depth of trees,...
GradientBoostingClassification	Ensemble of regression trees	Log loss	Gradient descent	Number/depth of trees,...
XGBoost, LightGBM, CatBoost	Ensemble of XGBoost trees	Square/log loss	2nd order gradients	Number/depth of trees,...
Stacking	Ensemble of heterogeneous models	/	/	Number of models,...

## Other ensembling techniques

- Hyper-ensembles: same basic model but with different hyperparameter settings
  - Can combine overfitted and underfitted models
- Deep ensembles: ensembles of deep learning models
- Bayes optimal classifier: ensemble of all possible models (largely theoretic)
- Bayesian model averaging: weighted average of probabilistic models, weighted by their posterior probabilities
- Cross-validation selection: does internal cross-validation to select best of  $M$  models
- Any combination of different ensembling techniques

## Summary

- Ensembles of voting classifiers improve performance
  - Which models to choose? Consider bias-variance tradeoffs!
- Bagging / RandomForest is a variance-reduction technique
  - Build many high-variance (overfitting) models on random data samples
    - The more different the models, the better
  - Aggregation (soft voting) over many models reduces variance
    - Diminishing returns, over-smoothing may increase bias error
  - Parallelizes easily, doesn't require much tuning
- Boosting is a bias-reduction technique
  - Build low-variance models that correct each other's mistakes
    - By reweighting misclassified samples: AdaBoost
    - By predicting the residual error: Gradient Boosting
  - Additive models: predictions are sum of base-model predictions
    - Can drive the error to zero, but risk overfitting
  - Doesn't parallelize easily. Slower to train, much faster to predict.
    - XGBoost, LightGBM, ... are fast and offer some parallelization
- Stacking: learn how to combine base-model predictions
  - Base-models still have to be sufficiently different



# Lecture 6. Data preprocessing

Real-world machine learning pipelines

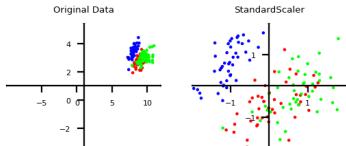
Joaquin Vanschoren

## Data transformations

- Machine learning models make a lot of assumptions about the data
- In reality, these assumptions are often violated
- We build *pipelines* that transform the data before feeding it to the learners
  - Scaling (or other numeric transformations)
  - Encoding (convert categorical features into numerical ones)
  - Automatic feature selection
  - Feature engineering (e.g. binning, polynomial features,...)
  - Handling missing data
  - Handling imbalanced data
  - Dimensionality reduction (e.g. PCA)
  - Learned embeddings (e.g. for text)
- Seek the best combinations of transformations and learning methods
  - Often done empirically, using cross-validation
  - Make sure that there is no data leakage during this process!

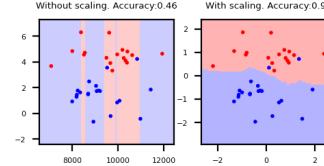
## Scaling

- Use when different numeric features have different scales (different range of values)
  - Features with much higher values may overpower the others
- Goal: bring them all within the same range
- Different methods exist



## Why do we need scaling?

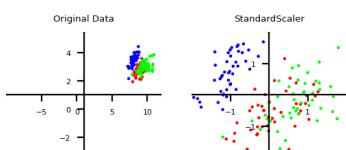
- KNN: Distances depend mainly on feature with larger values
- SVMs: (kernelized) dot products are also based on distances
- Linear model: Feature scale affects regularization
  - Weights have similar scales, more interpretable



## Standard scaling (standardization)

- Generally most useful, assumes data is more or less normally distributed
- Per feature, subtract the mean value  $\mu$ , scale by standard deviation  $\sigma$
- New feature has  $\mu = 0$  and  $\sigma = 1$ , values can still be arbitrarily large

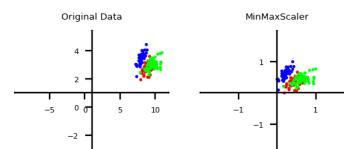
$$\mathbf{x}_{new} = \frac{\mathbf{x} - \mu}{\sigma}$$



## Min-max scaling

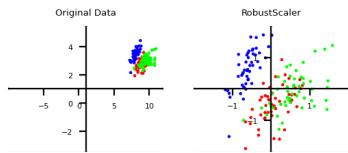
- Scales all features between a given  $min$  and  $max$  value (e.g. 0 and 1)
- Makes sense if min/max values have meaning in your data
- Sensitive to outliers

$$\mathbf{x}_{new} = \frac{\mathbf{x} - x_{min}}{x_{max} - x_{min}} \cdot (max - min) + min$$



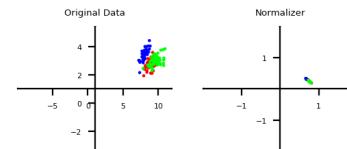
## Robust scaling

- Subtracts the median, scales between quantiles  $q_{25}$  and  $q_{75}$
- New feature has median 0,  $q_{25} = -1$  and  $q_{75} = 1$
- Similar to standard scaler, but ignores outliers



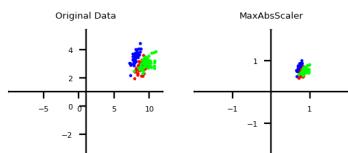
## Normalization

- Makes sure that feature values of each point (each row) sum up to 1 (L1 norm)
  - Useful for count data (e.g. word counts in documents)
- Can also be used with L2 norm (sum of squares is 1)
  - Useful when computing distances in high dimensions
  - Normalized Euclidean distance is equivalent to cosine similarity



## Maximum Absolute scaler

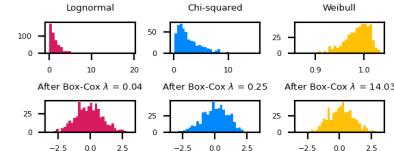
- For sparse data (many features, but few are non-zero)
  - Maintain sparseness (efficient storage)
- Scales all values so that maximum absolute value is 1
- Similar to Min-Max scaling without changing 0 values



## Power transformations

- Some features follow certain distributions
  - E.g. number of twitter followers is log-normal distributed
- Box-Cox transformations transform these to normal distributions ( $\lambda$  is fitted)
  - Only works for positive values, use Yeo-Johnson otherwise

$$bc_\lambda(x) = \begin{cases} \log(x) & \lambda = 0 \\ \frac{x^\lambda - 1}{\lambda} & \lambda \neq 0 \end{cases}$$



## Categorical feature encoding

- Many algorithms can only handle numeric features, so we need to encode the categorical ones

	boro	salary	vegan
0	Manhattan	103	0
1	Queens	89	0
2	Manhattan	142	0
3	Brooklyn	54	1
4	Brooklyn	63	1
5	Bronx	219	0

## Ordinal encoding

- Simply assigns an integer value to each category in the order they are encountered
- Only really useful if there exist a natural order in categories
  - Model will consider one category to be 'higher' or 'closer' to another

	boro	boro_ordinal	salary
0	Manhattan	2	103
1	Queens	3	89
2	Manhattan	2	142
3	Brooklyn	1	54
4	Brooklyn	1	63
5	Bronx	0	219

## Target encoding

### One-hot encoding (dummy encoding)

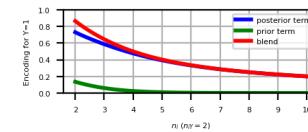
- Simply adds a new 0/1 feature for every category, having 1 (hot) if the sample has that category
- Can explode if a feature has lots of values, causing issues with high dimensionality
- What if test set contains a new category not seen in training data?
  - Either ignore it (just use all 0's in row), or handle manually (e.g. resample)

	boro_Bronx	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens	salary
0 Manhattan	0	0	0	1	0	652
1 Queens	0	0	0	1	0	89
2 Manhattan	0	0	1	0	0	142
3 Brooklyn	0	1	0	0	0	54
4 Brooklyn	0	1	0	0	0	65
5 Bronx	1	0	0	0	0	219

- $n_{iY}$ : nr of samples with category i and class Y=1,  $n_i$ : nr of samples with category i
- Preferred when you have lots of category values. It only creates one new feature per class
- Blends posterior probability of the target  $\frac{n_{iY}}{n_i}$  and prior probability  $\frac{n_Y}{n}$ .
  - Blending: gradually decrease as you get more examples of category i and class Y=0

$$Enc(i) = \frac{1}{1 + e^{-(n_i - 1)}} \frac{n_{iY}}{n_i} + (1 - \frac{1}{1 + e^{-(n_i - 1)}}) \frac{n_Y}{n}$$

Same for regression, using  $\frac{n_{iY}}{n_i}$ : average target value with category i,  $\frac{n_Y}{n}$ : overall mean



## In practice (scikit-learn)

### Example

- For Brooklyn,  $n_{iY} = 2$ ,  $n_i = 2$ ,  $n_Y = 2$ ,  $n = 6$
- Would be closer to 1 if there were more examples, all with label 1

$$Enc(Brooklyn) = \frac{1}{1 + e^{-1}} \frac{2}{2} + (1 - \frac{1}{1 + e^{-1}}) \frac{2}{6} = 0,82$$

Note: the implementation used here sets  $Enc(i) = \frac{n_Y}{n}$  when  $n_{iY} = 1$

	boro	boro_encoded	salary	vegan
0 Manhattan	0.09647	103	0	
1 Queens	0.33333	89	0	
2 Manhattan	0.09647	142	0	
3 Brooklyn	0.820706	54	1	
4 Brooklyn	0.820706	63	1	
5 Bronx	0.33333	219	0	

```
ordinal_encoder = OrdinalEncoder(dtype=int)
one_hot_encoder = OneHotEncoder(dtype=int)
```

- Target encoding is available in `category_encoders`
  - scikit-learn compatible
  - Also includes other, very specific encoders

```
target_encoder = TargetEncoder(return_df=True)
```

- All encoders (and scalers) follow the `fit-transform` paradigm
  - `fit` prepares the encoder, `transform` actually encodes the features
  - We'll discuss this next

```
encoder.fit(X, y)
X_encoded = encoder.transform(X, y)
```

## Applying data transformations

- Data transformations should always follow a fit-predict paradigm
  - Fit the transformer on the training data only
    - E.g. for a standard scaler: record the mean and standard deviation
    - Transform (e.g. scale) the training data, then train the learning model
    - Transform (e.g. scale) the test data, then evaluate the model
- Only scale the input features (X), not the targets (y)
- If you fit and transform the whole dataset before splitting, you get data leakage
  - You have looked at the test data before training the model
  - Model evaluations will be misleading
- If you fit and transform the training and test data separately, you distort the data
  - E.g. training and test points are scaled differently

## In practice (scikit-learn)

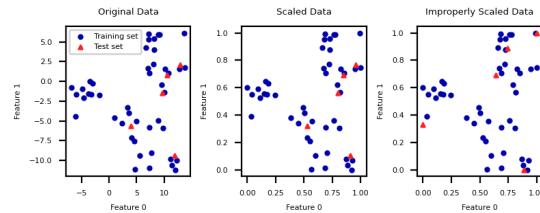
```
# choose scaling method and fit on training data
scaler = StandardScaler()
scaler.fit(X_train)
```

```
# transform training and test data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# calling fit and transform in sequence
X_train_scaled = scaler.fit(X_train).transform(X_train)
# same result, but more efficient computation
X_train_scaled = scaler.fit_transform(X_train)
```

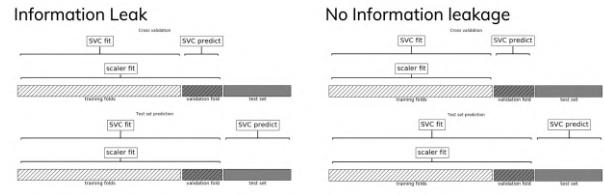
## Test set distortion

- Properly scaled: `fit` on training set, `transform` on training and test set
- Improperly scaled: `fit` and `transform` on the training and test data separately
  - Test data points nowhere near same training data points



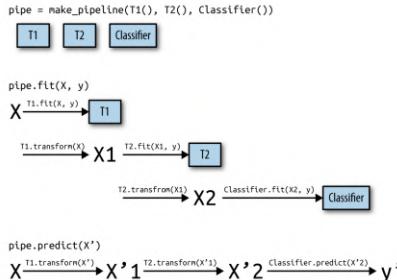
## Data leakage

- Cross-validation: training set is split into training and validation sets for model selection
- Incorrect: Scaler is fit on whole training set before doing cross-validation
  - Data leaks from validation folds into training folds, selected model may be optimistic
- Right: Scaler is fit on training folds only



## Pipelines

- A pipeline is a combination of data transformation and learning algorithms
- It has a `fit`, `predict`, and `score` method, just like any other learning algorithm
  - Ensures that data transformations are applied correctly



### In practice (scikit-learn)

- A `pipeline` combines multiple processing steps in a single estimator
- All but the last step should be data transformer (have a `transform` method)

```

# Make pipeline, step names will be 'minmaxscaler' and 'linearsvc'
pipe = make_pipeline(StandardScaler(), LinearSVC())
# Build pipeline with named steps
pipe = Pipeline([("scaler", StandardScaler()), ("svm", LinearSVC())])

# Correct fit and score
score = pipe.fit(X_train, y_train).score(X_test, y_test)
# Retrieve trained model by name
svm = pipe.named_steps['svm']

# Correct cross-validation
scores = cross_val_score(pipe, X, y)
  
```

- If you want to apply different preprocessors to different columns, use `ColumnTransformer`
- If you want to merge pipelines, you can use `FeatureUnion` to concatenate columns

```

# 2 sub-pipelines, one for numeric features, other for categorical ones
numeric_pipe = make_pipeline(SimpleImputer(), StandardScaler())
categorical_pipe = make_pipeline(SimpleImputer(), OneHotEncoder())

# Using categorical pipe for features A,B,C, numeric pipe otherwise
preprocessor = make_column_transformer((categorical_pipe,
                                       ["A", "B", "C"]),
                                       remainder=numeric_pipe)

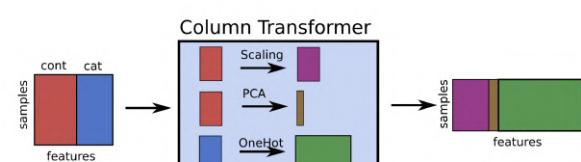
# Combine with learning algorithm in another pipeline
pipe = make_pipeline(preprocessor, LinearSVC())

# Feature union of PCA features and selected features
union = FeatureUnion([('pca', PCA()), ('selected', SelectKBest())])
pipe = make_pipeline(union, LinearSVC())
  
```

- `ColumnTransformer` concatenates features in order

```

pipe = make_column_transformer((StandardScaler(), numeric_features),
                               (PCA(), numeric_features),
                               (OneHotEncoder(), categorical_features))
  
```



## Pipeline selection

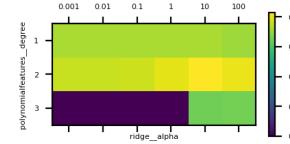
- We can safely use pipelines in model selection (e.g. grid search)
- Use `'__'` to refer to the hyperparameters of a step, e.g. `svm__C`

```
# Correct grid search (can have hyperparameters of any step)
param_grid = {'svm__C': [0.001, 0.01],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid).fit(X,y)
# Best estimator is now the best pipeline
best_pipe = grid.best_estimator_

# Tune pipeline and evaluate on held-out test set
grid = GridSearchCV(pipe,
param_grid=param_grid).fit(X_train,y_train)
grid.score(X_test,y_test)
```

## Example: Tune multiple steps at once

```
pipe = make_pipeline(StandardScaler(), PolynomialFeatures(), Ridge())
param_grid = {'polynomialfeatures_degree': [1, 2, 3],
              'ridge_alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid).fit(X_train, y_train)
```



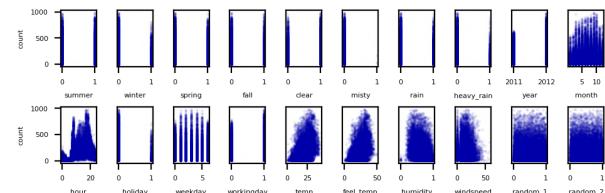
## Automatic Feature Selection

It can be a good idea to reduce the number of features to **only** the most useful ones

- Simpler models that generalize better (less overfitting)
  - Curse of dimensionality (e.g. KNN)
  - Even models such as RandomForest can benefit from this
  - Sometimes it is one of the main methods to improve models (e.g. gene expression data)
- Faster prediction and training
  - Training time can be quadratic (or cubic) in number of features
- Easier data collection, smaller models (less storage)
- More interpretable models: fewer features to look at

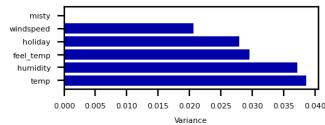
## Example: bike sharing

- The Bike Sharing Demand dataset shows the amount of bikes rented in Washington DC
- Some features are clearly more informative than others (e.g. temp, hour)
- Some are correlated (e.g. temp and feel\_temp)
- We add two random features at the end



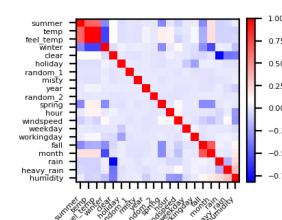
## Unsupervised feature selection

- Variance-based
  - Remove (near) constant features
    - Choose a **small** variance threshold
  - Scale features before computing variance!
  - Infrequent values may still be important
- Covariance-based
  - Remove **correlated** features
  - The **small** differences may actually be important
    - You don't know because you don't consider the target



## Covariance based feature selection

- Remove features  $X_i$  ( $= \mathbf{X}_{:,i}$ ) that are highly correlated (have high correlation coefficient  $\rho$ )
 
$$\rho(X_1, X_2) = \frac{\text{cov}(X_1, X_2)}{\sigma(X_1)\sigma(X_2)} = \frac{\frac{1}{N-1} \sum_i (X_{i,1} - \bar{X}_1)(X_{i,2} - \bar{X}_2)}{\sigma(X_1)\sigma(X_2)}$$
- Should we remove `feel_temp`? Or `temp`? Maybe one correlates more with the target?

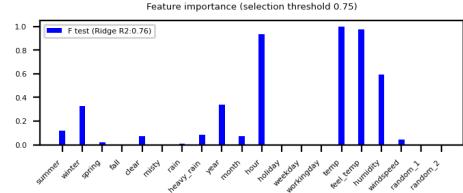


## Univariate statistics (F-test)

- Consider each feature individually (univariate), independent of the model that you aim to apply
- Use a statistical test: is there a *linear statistically significant relationship* with the target?
- Use F-statistic (or corresponding p value) to rank all features, then select features using a threshold
  - Best  $k$ , best  $k\%$ , probability of removing useful features (FPR), ...
- Cannot detect correlations (e.g. temp and feel\_temp) or interactions (e.g. binary features)

### Supervised feature selection: overview

- Univariate: F-test and Mutual Information
- Model-based: Random Forests, Linear models, kNN
- Wrapping techniques (black-box search)
- Permutation importance

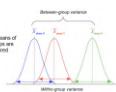
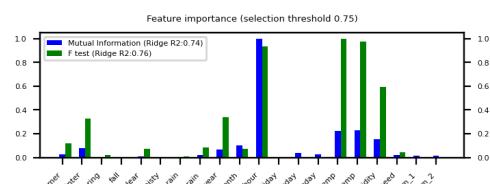


## Mutual information

- Measures how much information  $X_i$  gives about the target  $Y$ . In terms of entropy  $H$ :

$$MI(X, Y) = H(X) + H(Y) - H(X, Y)$$

- Idea: estimate  $H(X)$  as the average distance between a data point and its  $k$  Nearest Neighbors
  - You need to choose  $k$  and say which features are categorical
- Captures complex dependencies (e.g. hour, month), but requires more samples to be accurate



### F-statistic

- For regression: does feature  $X_i$  correlate (positively or negatively) with the target  $y$ ?

$$\text{F-statistic} = \frac{\rho(X_i, y)^2}{1 - \rho(X_i, y)^2} \cdot (N - 1)$$

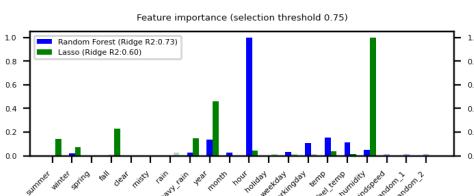
- For classification: uses ANOVA: does  $X_i$  explain the between-class variance?

- Alternatively, use the  $\chi^2$  test (only for categorical features)

$$\text{F-statistic} = \frac{\text{within-class variance}}{\text{between-class variance}} = \frac{\text{var}(X_i)}{\text{var}(X_i)}$$

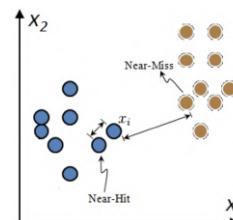
### Model-based Feature Selection

- Use a tuned(!) supervised model to judge the importance of each feature
  - Linear models (Ridge, Lasso, LinearSVM,...): features with highest weights (coefficients)
  - Tree-based models: features used in first nodes (high information gain)
- Selection model can be different from the one you use for final modelling
- Captures interactions: features are more/less informative in combination (e.g. winter, temp)
- RandomForests: learns complex interactions (e.g. hour), but biased to high cardinality features



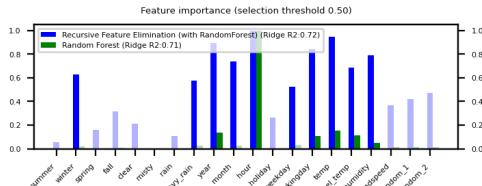
## Relief: Model-based selection with kNN

- For  $I$  iterations, choose a random point  $x_i$  and find  $k$  nearest neighbors  $x_k$
- Increase feature weights if  $x_i$  and  $x_k$  have different class (near miss), else decrease
  - $w_i = w_{i-1} + (x_i - \text{nearMiss}_i)^2 - (x_i - \text{nearHit}_i)^2$
- Many variants: ReliefF (uses L1 norm, faster), RReliefF (for regression), ...



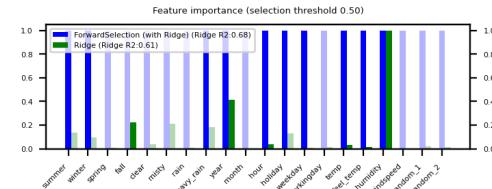
## Iterative Model-based Feature Selection

- Dropping many features at once is not ideal: feature importance may change in subset
- Recursive Feature Elimination (RFE)
  - Remove  $s$  least important feature(s), recompute remaining importances, repeat
- Can be rather slow



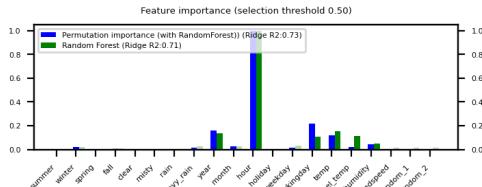
## Sequential feature selection (Wrapping)

- Evaluate your model with different sets of features, find best subset based on performance
- Greedy black-box search (can end up in local minima)
  - Backward selection: remove least important feature, recompute importances, repeat
  - Forward selection: set aside most important feature, recompute importances, repeat
  - Floating: add best new feature, remove worst one, repeat (forward or backward)
- Stochastic search: use random mutations in candidate subset (e.g. simulated annealing)



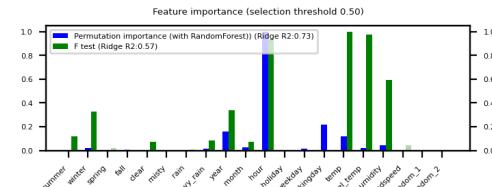
## Permutation feature importance

- Defined as the decrease in model performance when a single feature value is randomly shuffled
  - This breaks the relationship between the feature and the target
- Model agnostic, metric agnostic, and can be calculated many times with different permutations
- Can be applied to unseen data (not possible with model-based techniques)
- Less biased towards high-cardinality features (compared with RandomForests)



## Comparison

- Feature importances (scaled) and cross-validated  $R^2$  score of pipeline
  - Pipeline contains features selection + Ridge
- Selection threshold value ranges from 25% to 100% of all features
- Best method ultimately depends on the problem and dataset at hand



## In practice (scikit-learn)

- Unsupervised: VarianceThreshold
 

```
selector = VarianceThreshold(threshold=0.01)
X_selected = selector.fit_transform(X)
variances = selector.variances_
```
- Univariate:
  - For regression: f\_regression, mutual\_info\_regression
  - For classification: f\_classification, chi2, mutual\_info\_classification
  - Selecting: SelectKBest, SelectPercentile, SelectFpr, ...

```
selector = SelectPercentile(score_func=f_regression, percentile=50)
X_selected = selector.fit_transform(X, y)
selected_features = selector.get_support()
f_values, p_values = f_regression(X, y)
mi_values = mutual_info_regression(X, y, discrete_features=[:])
```

- Model-based:
  - SelectFromModel : requires a model and a selection threshold
  - RFE, RFECV (recursive feature elimination): requires model and final nr features

```
selector = SelectFromModel(RandomForestRegressor(), threshold='mean')
rfe_selector = RFE(RidgeCV(), n_features_to_select=20)
X_selected = selector.fit_transform(X)
rfe_importances = RandomForest().fit(X, y).feature_importances_
```
- Sequential feature selection (from mlxtend, sklearn-compatible)
 

```
selector = SequentialFeatureSelector(RidgeCV(), k_features=20,
                                         forward=True,
                                         floating=True)
X_selected = selector.fit_transform(X)
```
- Permutation Importance (in sklearn.inspection), no fit-transform interface
 

```
importances =
permutation_importance(RandomForestRegressor().fit(X, y),
X, y,
n_repeats=10).importances_mean
feature_ids = (-importances).argsort()[:n]
```

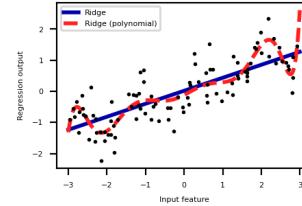
## Feature Engineering

- Create new features based on existing ones
  - Polynomial features
  - Interaction features
  - Binning
- Mainly useful for simple models (e.g. linear models)
  - Other models can learn interactions themselves
  - But may be slower, less robust than linear models

### Polynomials

- Add all polynomials up to degree  $d$  and all products
  - Equivalent to polynomial basis expansions

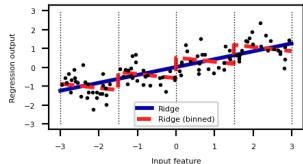
$$[1, x_1, \dots, x_p] \rightarrow [1, x_1, \dots, x_p, x_1^2, \dots, x_p^2, \dots, x_p^d, x_1x_2, \dots, x_{p-1}x_p]$$



### Binning

- Partition numeric feature values into  $n$  intervals (bins)
- Create  $n$  new one-hot features, 1 if original value falls in corresponding bin
- Models different intervals differently (e.g. different age groups)

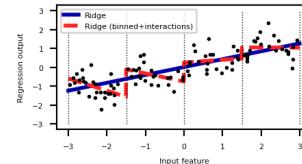
	orig [-3.0,-1.5]	[ -1.5,0.0 ]	[ 0.0,1.5 ]	[ 1.5,3.0 ]
0	-0.752759	0.000000	1.000000	0.000000
1	2.704286	0.000000	0.000000	1.000000
2	1.391964	0.000000	0.000000	1.000000



### Binning + interaction features

- Add interaction features (or product features )
  - Product of the bin encoding and the original feature value
  - Learn different weights per bin

	orig	b0	b1	b2	b3	X*b0	X*b1	X*b2	X*b3
0	-0.752759	0.000000	1.000000	0.000000	0.000000	-0.752759	-0.000000	-0.000000	-0.000000
1	2.704286	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	2.704286
2	1.391964	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	1.391964



### Categorical feature interactions

- One-hot-encode categorical feature
- Multiply every one-hot-encoded column with every numeric feature
- Allows to build different submodels for different categories

	gender	age	pageviews	time
0	M	14	70	269
1	F	16	12	1522
2	M	12	42	235
3	F	25	64	63
4	F	22	93	21

	age_M	pageviews_M	time_M	gender_M_M	age_F	pageviews_F	time_F	gender_F_F
0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	12	1522
2	1	0	0	1	0	0	0	0
3	0	0	0	0	0	25	64	63
4	0	0	0	0	22	93	21	1

### Missing value imputation

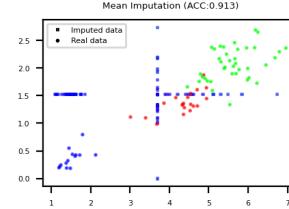
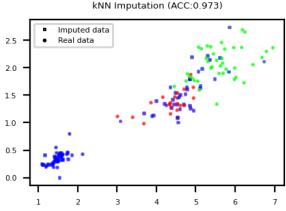
- Data can be missing in different ways:
  - Missing Completely at Random (MCAR): purely random points are missing
  - Missing at Random (MAR): something affects missingness, but no relation with the value
    - E.g. faulty sensors, some people don't fill out forms correctly
  - Missing Not At Random (MNAR): systematic missingness linked to the value
    - Has to be modelled or resolved (e.g. sensor decay, sick people leaving study)
- Missingness can be encoded in different ways: '?' , '-1' , 'unknown' , 'NA' , ...
- Also labels can be missing (remove example or use semi-supervised learning)

## Overview

- Mean/constant imputation
- kNN-based imputation
- Iterative (model-based) imputation
- Matrix Factorization techniques

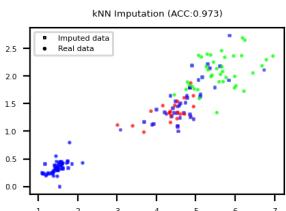
## Mean imputation

- Replace all missing values of a feature by the same value
  - Numerical features: mean or median
  - Categorical features: most frequent category
  - Constant value, e.g. 0 or 'missing' for text features
- Optional: add an indicator column for missingness
- Example: Iris dataset (randomly removed values in 3rd and 4th column)



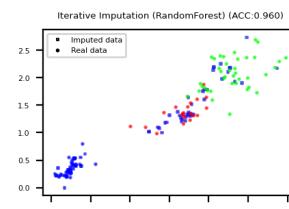
## kNN imputation

- Use special version of kNN to predict value of missing points
- Uses only non-missing data when computing distances



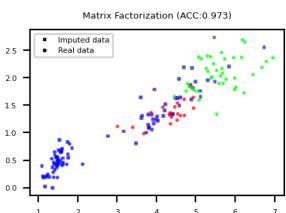
## Iterative (model-based) Imputation

- Better known as Multiple Imputation by Chained Equations (MICE)
- Iterative approach
  - Do first imputation (e.g. mean imputation)
  - Train model (e.g. RandomForest) to predict missing values of a given feature
  - Train new model on imputed data to predict missing values of the next feature
    - Repeat  $m$  times in round-robin fashion, leave one feature out at a time



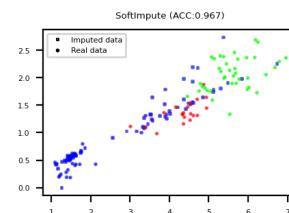
## Matrix Factorization

- Basic idea: low-rank approximation
  - Replace missing values by 0
  - Factorize  $\mathbf{X}$  with rank  $r$ :  $\mathbf{X}^{n \times p} = \mathbf{U}^{n \times r} \mathbf{V}^{r \times p}$ 
    - With  $n$  data points and  $p$  features
    - Solved using gradient descent
  - Recompute  $\mathbf{X}$ : now complete



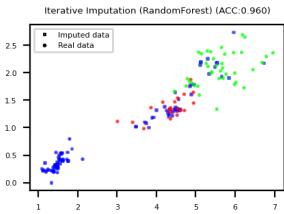
## Soft-thresholded Singular Value Decomposition (SVD)

- Same basic idea, but smoother
  - Replace missing values by 0, compute SVD:  $\mathbf{X} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T$ 
    - Solved with gradient descent
  - Reduce eigenvalues by shrinkage factor:  $\lambda_i = s \cdot \lambda_i$
  - Recompute  $\mathbf{X}$ : now complete
  - Repeat for  $m$  iterations



## Comparison

- Best method depends on the problem and dataset at hand. Use cross-validation.
- Iterative Imputation (MICE) generally works well for missing (completely) at random data
  - Can be slow if the prediction model is slow
- Low-rank approximation techniques scale well to large datasets



## In practice (scikit-learn)

- Simple replacement: `SimpleImputer`
    - Strategies: `mean` (numeric), `median`, `most_frequent` (categorical)
    - Choose whether to add indicator columns, and how missing values are encoded
- ```
imp = SimpleImputer(strategy='mean', missing_values=np.nan,
add_indicator=False)
X_complete = imp.fit_transform(X_train)
```
- KNN Imputation: `KNNImputer`
- ```
imp = KNNImputer(n_neighbors=5)
X_complete = imp.fit_transform(X_train)
```
- Multiple Imputation (MICE): `IterativeImputer`
    - Choose estimator (default: `BayesianRidge`) and number of iterations (default 10)
- ```
imp = IterativeImputer(estimator=RandomForestClassifier(),
max_iter=10)
X_complete = imp.fit_transform(X_train)
```

## In practice (fancyimpute)

- Cannot be used in CV pipelines (has `fit_transform` but no `transform`)
- Soft-Thresholded SVD: `SoftImpute`
  - Choose max number of gradient descent iterations
  - Choose shrinkage value for eigenvectors (default:  $\frac{1}{N}$ )

```
imp = SoftImpute(max_iter=10, shrinkage_value=None)
X_complete = imp.fit_transform(X)
```

- Low-rank imputation: `MatrixFactorization`
  - Choose rank of the low-rank approximation
  - Gradient descent hyperparameters: learning rate, epochs,...
  - Several variants exist

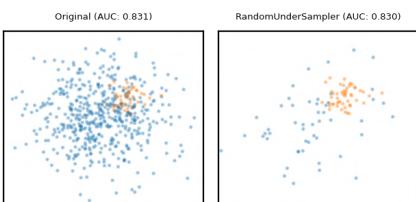
```
imp = MatrixFactorization(rank=10, learning_rate=0.001,
epoches=1000)
X_complete = imp.fit_transform(X)
```

## Handling imbalanced data

- Problem:
  - You have a majority class with many times the number of examples as the minority class
  - Or: classes are balanced, but associated costs are not (e.g. FN are worse than FP)
- We already covered some ways to resolve this:
  - Add class weights to the loss function: give the minority class more weight
    - In practice: set `class_weight='balanced'`
  - Change the prediction threshold to minimize false negatives or false positives
- There are also things we can do by preprocessing the data
  - Resample the data to correct the imbalance
    - Random or model-based
  - Generate synthetic samples for the minority class
  - Build ensembles over different resampled datasets
  - Combinations of these

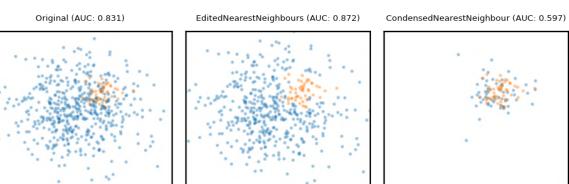
## Random Undersampling

- Copy the points from the minority class
- Randomly sample from the majority class (with or without replacement) until balanced
  - Optionally, sample until a certain imbalance ratio (e.g. 1/5) is reached
  - Multi-class: repeat with every other class
- Preferred for large datasets, often yields smaller/faster models with similar performance



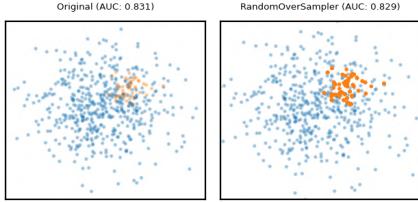
## Model-based Undersampling

- Edited Nearest Neighbors
  - Remove all majority samples that are misclassified by kNN (mode) or that have a neighbor from the other class (all).
  - Remove their influence on the minority samples
- Condensed Nearest Neighbors
  - Remove all majority samples that are *not* misclassified by kNN
  - Focus on only the hard samples



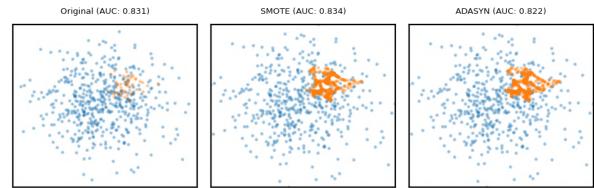
## Random Oversampling

- Copy the points from the majority class
- Randomly sample from the minority class, with replacement, until balanced
  - Optionally, sample until a certain imbalance ratio (e.g. 1/5) is reached
- Makes models more expensive to train, doesn't always improve performance
- Similar to giving minority class(es) a higher weight (and more expensive)



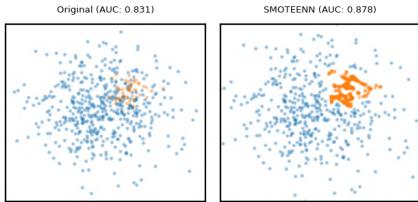
## Synthetic Minority Oversampling Technique (SMOTE)

- Repeatedly choose a random minority point and a neighboring minority point
  - Pick a new, artificial point on the line between them (uniformly)
- May bias the data. Be careful never to create artificial points in the test set.
- ADASYN (Adaptive Synthetic)
  - Similar, but starts from 'hard' minority points (misclassified by KNN)



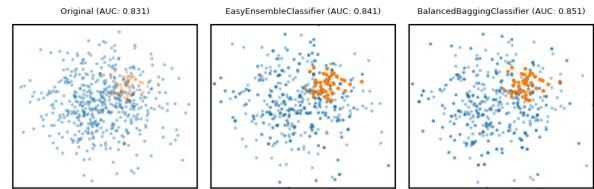
## Combined techniques

- Combines over- and under-sampling
- E.g. oversampling with SMOTE, undersampling with Edited Nearest Neighbors (ENN)
  - SMOTE can generate 'noisy' point, close to majority class points
  - ENN will remove up these majority points to 'clean up' the space



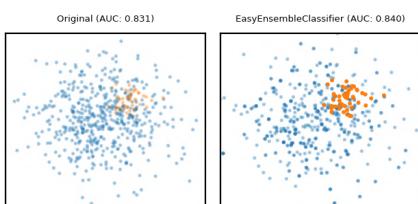
## Ensemble Resampling

- Bagged ensemble of balanced base learners. Acts as a learner, not a preprocessor
- BalancedBagging: take bootstraps, randomly undersample each, train models (e.g. trees)
  - Benefits of random undersampling without throwing out so much data
- Easy Ensemble: take multiple random undersamplings directly, train models
  - Traditionally uses AdaBoost as base learner, but can be replaced



## Comparison

- The best method depends on the data (amount of data, imbalance,...)
  - For a very large dataset, random undersampling may be fine
- You still need to choose the appropriate learning algorithms
- Don't forget about class weighting and prediction thresholding
  - Some combinations are useful, e.g. SMOTE + class weighting + thresholding



## In practice (`imblearn`)

- Follows fit-sample paradigm (equivalent of fit-transform, but also affects y)
- Undersampling: RandomUnderSampler, EditedNearestNeighbours,...
- (Synthetic) Oversampling: RandomOverSampler, SMOTE, ADASYN,...
- Combinations: SMOTEENN,...

```
x_resampled, y_resampled = SMOTE(k_neighbors=5).fit_sample(x, y)
```

- Can be used in imblearn pipelines (not sklearn pipelines)
  - imblearn pipelines are compatible with GridSearchCV,...
  - Sampling is only done in `fit` (not in `predict`)

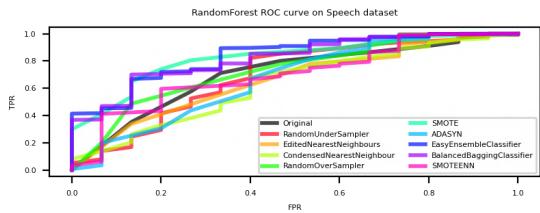
```
smote_pipe = make_pipeline(SMOTE(), LogisticRegression())
scores = cross_validate(smote_pipe, X_train, y_train)
param_grid = {"k_neighbors": [3, 5, 7]}
grid = GridSearchCV(smote_pipe, param_grid=param_grid, X, y)
```

- The ensembling techniques should be used as wrappers

```
cif = EasyEnsembleClassifier(base_estimator=SVC()).fit(X_train, y_train)
```

## Real-world data

- The effect of sampling procedures can be unpredictable
- Best method can depend on the data and FP/FN trade-offs
- SMOTE and ensembling techniques often work well



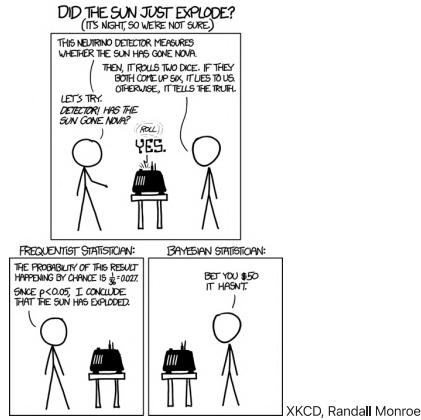
## Summary

- Data preprocessing is a crucial part of machine learning
  - Scaling is important for many distance-based methods (e.g. kNN, SVM, Neural Nets)
  - Categorical encoding is necessary for numeric methods (or implementations)
  - Selecting features can speed up models and reduce overfitting
  - Feature engineering is often useful for linear models
  - It is often better to impute missing data than to remove data
  - Imbalanced datasets require extra care to build useful models
- Pipelines allow us to encapsulate multiple steps in a convenient way
  - Avoids data leakage, crucial for proper evaluation
- Choose the right preprocessing steps and models in your pipeline
  - Cross-validation helps, but the search space is huge
  - Smarter techniques exist to automate this process (AutoML)

# Lecture 7. Bayesian Learning

Learning in an uncertain world

Joaquin Vanschoren



## Bayes' rule

Rule for updating the probability of a hypothesis  $c$  given data  $x$

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Likelihood                      Class Prior Probability  
↓                                  ↑  
Posterior Probability            Predictor Prior Probability

$P(c|x)$  is the posterior probability of class  $c$  given data  $x$ .

$P(c)$  is the prior probability of class  $c$ : what you believed before you saw the data  $x$ .

$P(x|c)$  is the likelihood of data point  $x$  given that the class is  $c$  (computed from your dataset)

$P(x)$  is the prior probability of the data (marginal likelihood): the likelihood of the data  $x$  under any circumstance (no matter what the class is)

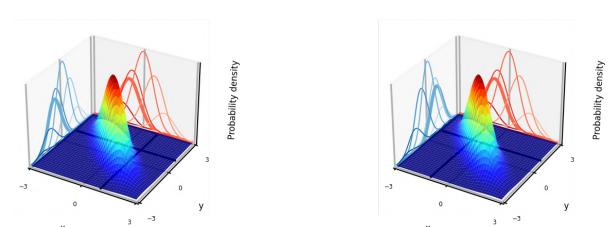
## Example: exploding sun

- Let's compute the probability that the sun has exploded
  - Prior  $P(\text{exploded})$ : the sun has an estimated lifespan of 10 billion years,  $P(\text{exploded}) = \frac{1}{4.38 \times 10^{10}}$
  - Likelihood that detector lies:  $P(\text{lie}) = \frac{1}{36}$
- $$\begin{aligned} P(\text{exploded}|yes) &= \frac{P(yes|\text{exploded})P(\text{exploded})}{P(yes)} \\ &= \frac{(1 - P(\text{lie}))P(\text{exploded})}{P(\text{exploded})(1 - P(\text{lie})) + P(\text{lie})(1 - P(\text{exploded}))} \\ &= \frac{1}{1.25226 \times 10^{-12}} \end{aligned}$$

- The one positive observation of the detector increases the probability

## Bayesian models

- Learn the joint distribution  $P(x, y) = P(x|y)P(y)$ .
  - Assumes that the data is Gaussian distributed (!)
  - With every input  $x$  you get  $P(y|x)$ , hence a mean and standard deviation for  $y$  (blue)
  - For every desired output  $y$  you get  $P(x|y)$ , hence you can sample new points  $x$  (red)
- Easily updatable with new data using Bayes' rule ('turning the crank')
  - Previous posterior  $P(y|x)$  becomes new prior  $P(y)$



## Example: COVID test

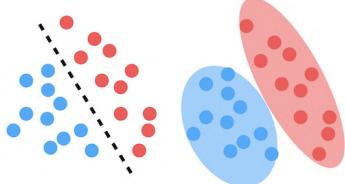
- What is the probability of having COVID-19 if a 96% accurate test returns positive? Assume a false positive rate of 4%
- Prior  $P(C) : 0.015$  (117M cases, 7.9B people)
- $P(TP) = P(\text{pos}|C) = 0.96$ , and  $P(FP) = P(\text{pos}|notC) = 0.04$
- If test is positive, prior becomes  $P(C) = 0.268$ . 2nd positive test:  $P(C|pos) = 0.9$

$$\begin{aligned} P(C|pos) &= \frac{P(pos|C)P(C)}{P(pos)} \\ &= \frac{P(pos|C)P(C)}{P(pos|C)P(C) + P(pos|notC)(1 - P(C))} \\ &= \frac{0.96 * 0.015}{0.96 * 0.015 + 0.04 * 0.985} \\ &= 0.268 \end{aligned}$$

## Generative models

- The joint distribution represents the training data for a particular output (e.g. a class)
- You can sample a new point  $\mathbf{x}$  with high predicted likelihood  $P(\mathbf{x}, c)$ : that new point will be very similar to the training points
- Generate new (likely) points according to the same distribution: *generative model*
  - Generate examples that are *false* but corresponding to a desired output
  - Generative neural networks (e.g. GANs) can do this very accurately for text, images, ...

## Discriminative      Generative



## Naive Bayes

- Predict the probability that a point belongs to a certain class, using Bayes' Theorem
$$P(c|\mathbf{x}) = \frac{P(\mathbf{x}|c)P(c)}{P(\mathbf{x})}$$
- Problem: since  $\mathbf{x}$  is a vector, computing  $P(\mathbf{x}|c)$  can be very complex
- Naively assume that all features are conditionally independent from each other, in which case:  $P(\mathbf{x}|c) = P(x_1|c) \times P(x_2|c) \times \dots \times P(x_n|c)$
- Very fast: only needs to extract statistics from each feature.

## On numeric data

On categorical data

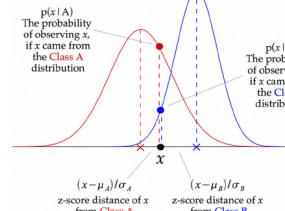
What's the probability that your friend will play golf if the weather is sunny?

$$P(x|c) = P(\text{Sunny} | \text{Yes}) = 3/9 = 0.33$$

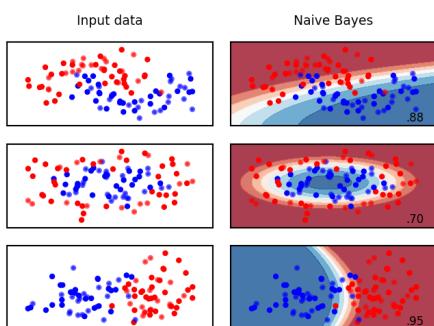
|         |          | Play Golf                                                                                              |      |      |
|---------|----------|--------------------------------------------------------------------------------------------------------|------|------|
|         |          | Yes                                                                                                    | No   |      |
| Outlook | Sunny    | 3/9                                                                                                    | 2/5  | 5/14 |
|         | Overcast | 4/9                                                                                                    | 0/5  | 4/14 |
|         | Rainy    | 2/9                                                                                                    | 3/5  | 5/14 |
|         |          | 9/14                                                                                                   | 5/14 |      |
|         |          | $P(x) = P(\text{Sunny}) = 5/14 = 0.36$                                                                 |      |      |
|         |          | $P(c) = P(\text{Yes}) = 9/14 = 0.64$                                                                   |      |      |
|         |          | <b>Posterior Probability:</b> $P(c x) = P(\text{Yes}   \text{Sunny}) = 0.33 \times 0.64 / 0.36 = 0.60$ |      |      |

$$p(x|v|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}}$$

$$p(x|c) = \frac{p(x|c)p(c)}{p(x)}$$



- What do the predictions of Gaussian Naive Bayes look like?

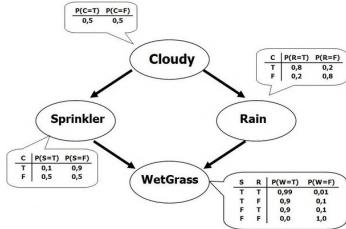


## Other Naive Bayes classifiers:

- BernoulliNB
  - Assumes binary data
  - Feature statistics: Number of non-zero entries per class
- MultinomialNB
  - Assumes count data
  - Feature statistics: Average value per class
  - Mostly used for text classification (bag-of-words data)

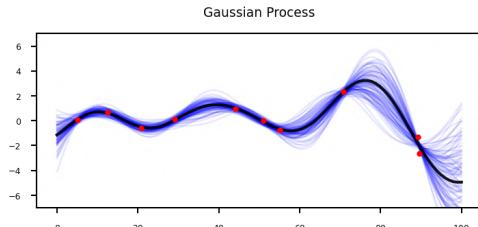
## Bayesian Networks

- What if we know that some variables are not independent?
- A Bayesian Network is a directed acyclic graph representing variables as nodes and conditional dependencies as edges.
- If an edge  $(A, B)$  connects random variables A and B, then  $P(B|A)$  is a factor in the joint probability distribution. We must know  $P(B|A)$  for all values of B and A
- The graph structure can be designed manually or learned (hard!)



## Gaussian processes

- Model the data as a Gaussian distribution, conditioned on the training points



## Probabilistic interpretation of regression

Linear regression (recap):

$$y = f(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w} + b$$

For one input feature:

$$y = w_1 \cdot x_1 + b \cdot 1$$

We can solve this via linear algebra (closed form solution):  $w^* = (X^T X)^{-1} X^T Y$

```
w = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))
```

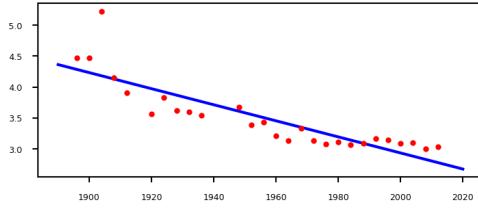
$\mathbf{X}$  is our data matrix with a  $x_0 = 1$  column to represent the bias  $b$ :

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_N^\top \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}$$

Example: Olympic marathon data

We learned:  $y = w_1 x + w_0 = -0.013x + 28.895$

Linear regression ( $w: [[28.895 -0.013]]$ )

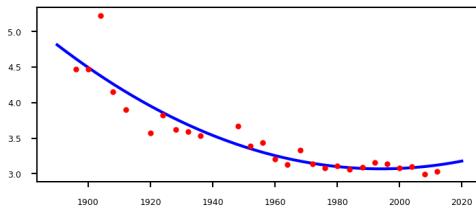


## Polynomial regression (recap)

We can fit a 2nd degree polynomial by using a basis expansion (adding more basis functions):

$$\Phi = [1 \quad x \quad x^2]$$

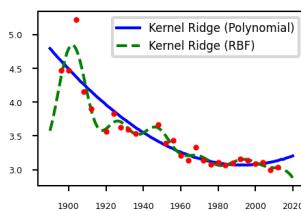
Polynomial regression ( $w: [[643.642 -0.643 0. ]]$ )



## Kernelized regression (recap)

We can also kernelize the model and learn a dual coefficient per data point

Kernel Ridge

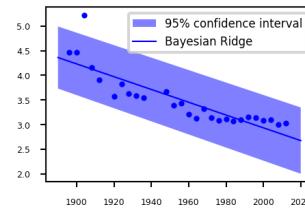


## Probabilistic interpretation

- These models do not give us any indication of the (un)certainty of the predictions
- Assume that the data is inherently uncertain. This can be modeled explicitly by introducing a **slack variable**,  $\epsilon_i$ , known as noise.  
 $y_i = w_1 x_i + w_0 + \epsilon_i$ .
- Assume that the noise is distributed according to a Gaussian distribution with zero mean and variance  $\sigma^2$ .  
 $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
- That means that  $y(x)$  is now a Gaussian distribution with mean  $\mathbf{w}x$  and variance  $\sigma^2$   
 $y = \mathcal{N}(\mathbf{w}x, \sigma^2)$

We have an uncertainty predictions, but it is the same for **all** predictions

- You would expect to be more certain nearby your training points



## How to learn probabilities?

- Maximum Likelihood Estimation (MLE): Maximize  $P(\mathbf{X}|\mathbf{w})$ 
    - Corresponds to optimizing  $\mathbf{w}$ , using (log) likelihood as the loss function
    - Every prediction has a mean defined by  $\mathbf{w}$  and Gaussian noise
- $$P(\mathbf{X}|\mathbf{w}) = \prod_{i=0}^n P(\mathbf{y}_i|\mathbf{x}_i; \mathbf{w}) = \prod_{i=0}^n \mathcal{N}(\mathbf{w}\mathbf{x}, \sigma^2 \mathbf{I})$$

- Maximum A Posteriori estimation (MAP): Maximize the posterior  $P(\mathbf{w}|\mathbf{X})$ 
    - This can be done using Bayes' rule after we choose a (Gaussian) prior  $P(\mathbf{w})$ :  

$$P(\mathbf{w}|\mathbf{X}) = \frac{P(\mathbf{X}|\mathbf{w})P(\mathbf{w})}{P(\mathbf{X})}$$
  - Bayesian approach: model the prediction  $P(y|x_{test}, \mathbf{X})$  directly
    - Marginalize  $w$  out: consider all possible models (some are more likely)
    - If prior  $P(\mathbf{w})$  is Gaussian, then  $P(y|x_{test}, \mathbf{X})$  is also Gaussian!
      - A multivariate Gaussian with mean  $\mu$  and covariance matrix  $\Sigma$
- $$P(y|x_{test}, \mathbf{X}) = \int_w P(y|x_{test}, \mathbf{w})P(\mathbf{w}|\mathbf{X})dw = \mathcal{N}(\mu, \Sigma)$$

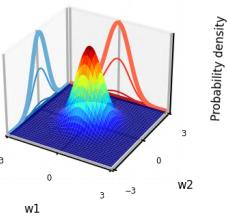
## Gaussian prior $P(w)$

In the Bayesian approach, we assume a *prior (Gaussian) distribution* for the parameters,  $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha \mathbf{I})$ :

- With zero mean ( $\mu=0$ ) and covariance matrix  $\alpha \mathbf{I}$ . For 2D:  $\alpha \mathbf{I} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix}$

I.e.,  $w_i$  is drawn from a Gaussian density with variance  $\alpha$

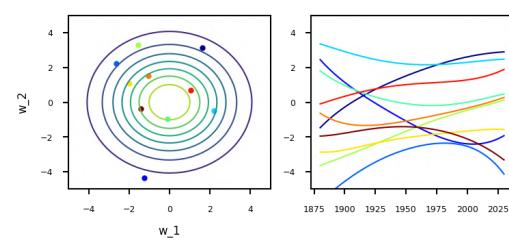
$$w_i \sim \mathcal{N}(0, \alpha)$$



## Sampling from the prior (weight space)

We can sample from the prior distribution to see what form we are imposing on the functions *a priori* (before seeing any data).

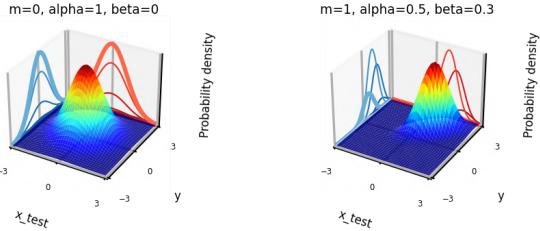
- Draw  $w$  (left) independently from a Gaussian density  $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha \mathbf{I})$ 
  - Use any normally distributed sampling technique, e.g. Box-Muller transform
- Every sample yields a polynomial function  $f(\mathbf{x})$  (right):  $f(\mathbf{x}) = \mathbf{w}\phi(\mathbf{x})$ .
  - For example, with  $\phi(\mathbf{x})$  being a polynomial:



## Learning Gaussian distributions

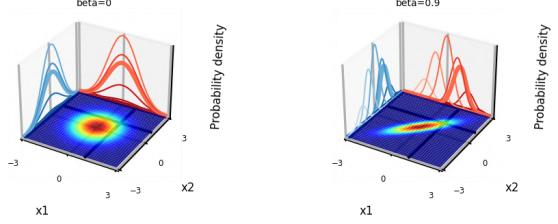
- We assume that our data is Gaussian distributed:  $P(y|x_{test}, \mathbf{X}) = \mathcal{N}(\mu, \Sigma)$

- Example with learned mean  $[m, m]$  and covariance  $\begin{bmatrix} \alpha & \beta \\ \beta & \alpha \end{bmatrix}$
- The blue curve is the predicted  $P(y|x_{test}, \mathbf{X})$



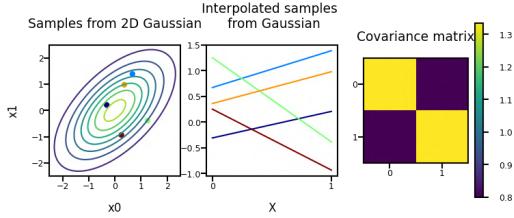
## Understanding covariances

- If two variables  $x_1$  covariate strongly, knowing about  $x_1$  tells us a lot about  $x_2$
- If covariance is 0, knowing  $x_1$  tells us nothing about  $x_2$  (the conditional and marginal distributions are the same)
- For covariance matrix  $\begin{bmatrix} 1 & \beta \\ \beta & 1 \end{bmatrix}$ :

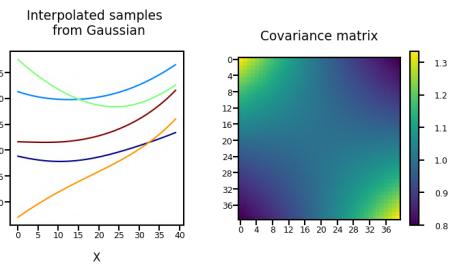


## Sampling from higher-dimensional distributions

- Instead of sampling  $\mathbf{w}$  and then multiplying by  $\Phi$ , we can also generate examples of  $f(x)$  directly.
- $\mathbf{f}$  with  $n$  values can be sampled from an  $n$ -dimensional Gaussian distribution with zero mean and covariance matrix  $\mathbf{K} = \alpha\Phi\Phi^\top$ :
- $\mathbf{f}$  is a stochastic process: series of normally distributed variables (interpolated in the plot)



Repeat for 40 dimensions, with  $\Phi$  the polynomial transform:

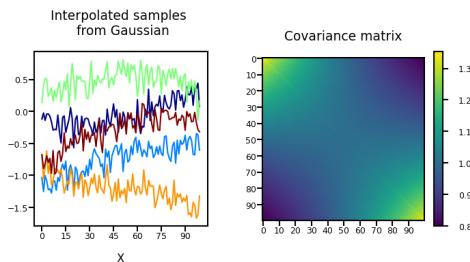


[More examples of covariances](#)

## Noisy functions

We normally add Gaussian noise to obtain our observations:

$$\mathbf{y} = \mathbf{f} + \boldsymbol{\epsilon}$$



## Gaussian Process

- Usually, we want our functions to be smooth: if two points are similar/nearby, the predictions should be similar.
  - Hence, we need a similarity measure (a kernel)
- In a Gaussian process we can do this by specifying the covariance function directly (not as  $\mathbf{K} = \alpha\Phi\Phi^\top$ )
  - The covariance matrix is simply the kernel matrix:  $\mathbf{f} \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$
- The RBF (Gaussian) covariance function (or kernel) is specified by

$$k(\mathbf{x}, \mathbf{x}') = \alpha \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right).$$

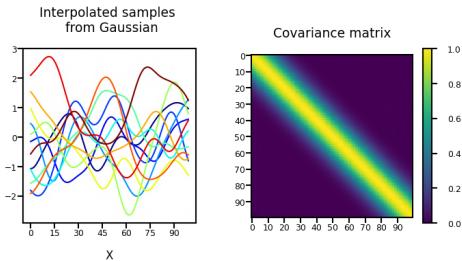
where  $\|\mathbf{x} - \mathbf{x}'\|^2$  is the squared distance between the two input vectors

$$\|\mathbf{x} - \mathbf{x}'\|^2 = (\mathbf{x} - \mathbf{x}')^\top (\mathbf{x} - \mathbf{x}')$$

and the length parameter  $\ell$  controls the smoothness of the function and  $\alpha$  the vertical variation.

Now the influence of a point decreases smoothly but exponentially

- These are our priors  $P(\mathbf{y}) = \mathcal{N}(\mathbf{0}, \mathbf{K})$ , with mean 0
- We now want to condition it on our training data:  $P(\mathbf{y}|\mathbf{x}_{test}, \mathbf{X}) = \mathcal{N}(\mu, \Sigma)$



## Computing the posterior $P(\mathbf{y}|\mathbf{X})$

- Assuming that  $P(\mathbf{X})$  is a Gaussian density with a covariance given by kernel matrix  $\mathbf{K}$ , the model likelihood becomes:

$$P(\mathbf{y}|\mathbf{X}) = \frac{P(\mathbf{y}) P(\mathbf{X}|\mathbf{y})}{P(\mathbf{X})} = \frac{1}{(2\pi)^{\frac{n}{2}} |\mathbf{K}|^{\frac{1}{2}}} \exp\left(-\frac{1}{2} \mathbf{y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{y}\right)$$

- Hence, the negative log likelihood (the objective function) is given by:

$$E(\theta) = \frac{1}{2} \log |\mathbf{K}| + \frac{1}{2} \mathbf{y}^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{y}$$

- The model parameters (e.g. noise variance  $\sigma^2$ ) and the kernel parameters (e.g. lengthscales, variance) can be embedded in the covariance function and learned from data.
- Good news: This loss function can be optimized using linear algebra (Cholesky Decomposition)
- Bad news: This is cubic in the number of data points AND the number of features:  $\mathcal{O}(n^3 d^3)$

## Conditional Density $P(\mathbf{y}|\mathbf{x}_{test}, \mathbf{X})$

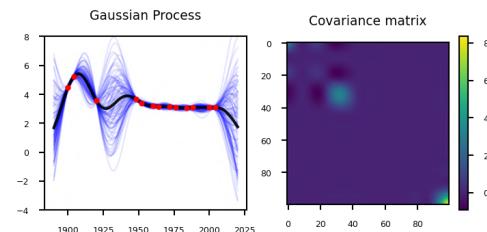
Finally, we need to define *conditional* distributions to answer particular questions of interest

We will need the *conditional density* for making predictions,

$$\mathbf{f}^* | \mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_f, \mathbf{C}_f)$$

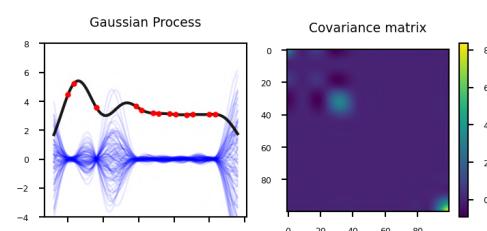
with a mean given by  $\boldsymbol{\mu}_f = \mathbf{K}_*^\top [\mathbf{K} + \sigma^2 \mathbf{I}]^{-1} \mathbf{y}$

and a covariance given by  $\mathbf{C}_f = \mathbf{K}_{*,*} - \mathbf{K}_*^\top [\mathbf{K} + \sigma^2 \mathbf{I}]^{-1} \mathbf{K}_*$ .



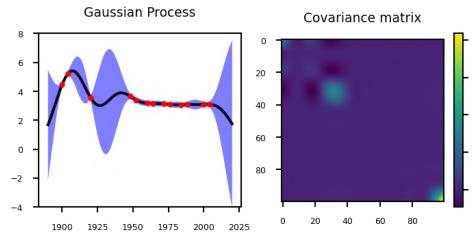
Remember that our prediction is the sum of the mean and the variance:  $P(\mathbf{y}|\mathbf{x}_{test}, \mathbf{X}) = \mathcal{N}(\mu, \Sigma)$

- The mean is the same as the one computed with kernel ridge (if given the same kernel and hyperparameters)
- The Gaussian process learned the covariance and the hyperparameters



## Gaussian Processes in practice (with GPy)

The values on the diagonal of the covariance matrix give us the variance, so we can simply plot the mean and 95% confidence interval

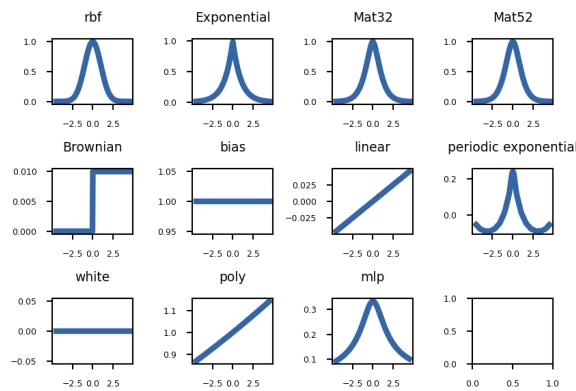


- `GPyRegression`
- Generate a kernel first
  - State the dimensionality of your input data
  - Variance and lengthscale are optional, default = 1
 

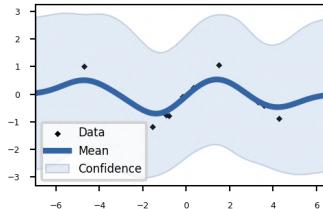
```
kernel = GPy.kern.RBF(input_dim=1, variance=1.,
lengthscale=1.)
```
  - Other kernels:
 

```
GPy.kern.BasisFuncKernel?
```
- Build model:
 

```
m = GPy.models.GPRegression(X,Y,kernel)
```

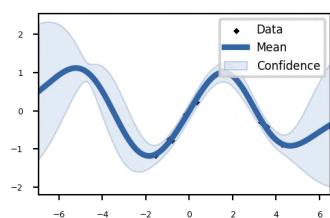


Build the untrained GP. The shaded region corresponds to ~95% confidence intervals (i.e. +/- 2 standard deviation)

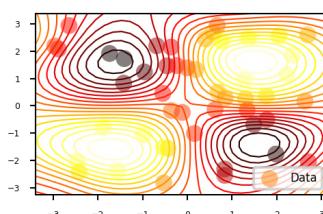


`Matern` is a generalized RBF kernel that can scale between RBF and Exponential

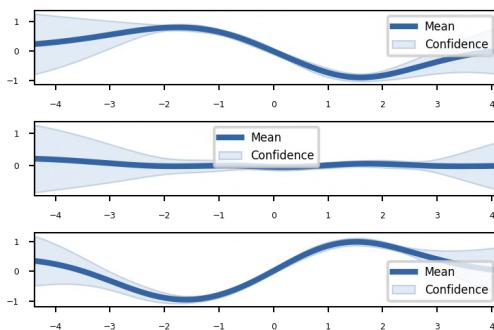
Train the model (optimize the parameters): maximize the likelihood of the data.  
Best to optimize with a few restarts: the optimizer may converge to the high-noise solution. The optimizer is then restarted with a few random initialization of the parameter values.



You can also show results in 2D



We can plot 2D slices using the `fixed_inputs` argument to the `plot` function.  
`fixed_inputs` is a list of tuples containing which of the inputs to fix, and to which value.

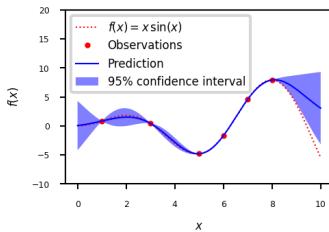


### Gaussian Processes with scikit-learn

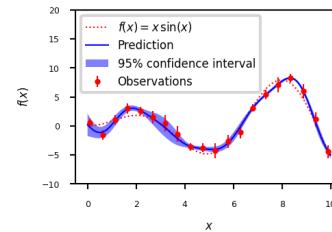
- `GaussianProcessRegressor`
- Hyperparameters:
  - `kernel`: kernel specifying the covariance function of the GP
    - Default: "1.0 \* RBF(1.0)"
    - Typically leave at default. Will be optimized during fitting
  - `alpha`: regularization parameter
    - Tikhonov regularization of covariance between the training points.
    - Adds a (small) value to diagonal of the kernel matrix during fitting.
    - Larger values:
      - correspond to increased noise level in the observations
      - also reduce potential numerical issues during fitting
    - Default: 1e-10
  - `n_restarts_optimizer`: number of restarts of the optimizer
    - Default: 0. Best to do at least a few iterations.
    - Optimizer finds kernel parameters maximizing log-marginal likelihood
- Retrieve predictions and confidence interval after fitting:

```
y_pred, sigma = gp.predict(x, return_std=True)
```

### Example



### Example with noisy data



### Gaussian processes: Conclusions

#### Advantages:

- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals.
- The prediction interpolates the observations (at least for regular kernels).
- Versatile: different kernels can be specified.

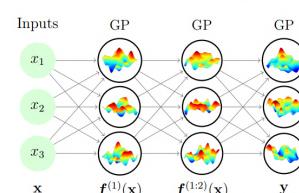
#### Disadvantages:

- They are typically not sparse, i.e., they use the whole sample/feature information to perform the prediction.
  - Sparse GPs also exist: they remember only the most important points
- They lose efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens.

### Gaussian processes and neural networks

- You can prove that a Gaussian process is equivalent to a neural network with one layer and an infinite number of nodes
- You can build *deep Gaussian Processes* by constructing layers of GPs

A net with nonparametric activation functions corresponding to a 3-layer deep GP

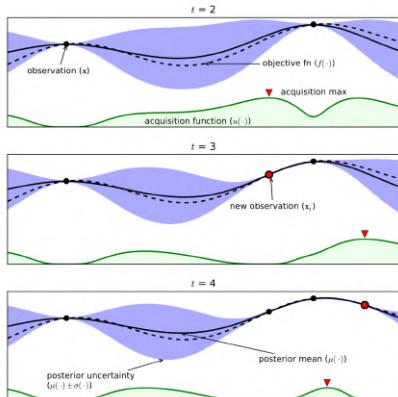


## Bayesian optimization

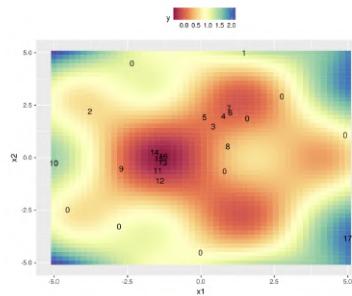
- The incremental updates you can do with Bayesian models allow a more effective way to optimize functions
  - E.g. to optimize the hyperparameter settings of a machine learning algorithm/pipeline
- After a number of random search iterations we know more about the performance of hyperparameter settings on the given dataset
- We can use this data to train a model, and predict which other hyperparameter values might be useful
  - More generally, this is called **model-based optimization**
  - This model is called a **surrogate model**
- This is often a probabilistic (e.g. Bayesian) model that predicts confidence intervals for all hyperparameter settings
- We use the predictions of this model to choose the next point to evaluate
- With every new evaluation, we update the surrogate model and repeat

## Example (see figure):

- Consider only 1 continuous hyperparameter (X-axis)
  - You can also do this for many more hyperparameters
- Y-axis shows cross-validation performance
- Evaluate a number of random hyperparameter settings (black dots)
- Sometimes an initialization design is used
- Train a model, and predict the expected performance of other (unseen) hyperparameter values
  - Mean value (black line) and distribution (blue band)
- An acquisition function (green line) trades off maximal expected performance and maximal uncertainty
- Exploitation vs exploration
- Optimal value of the acquisition function is the next hyperparameter setting to be evaluated
- Repeat a fixed number of times, or until time budget runs out



In 2 dimensions:



Shahriari et al. Taking the Human Out of the Loop: A Review of Bayesian Optimization

## Acquisition Functions

- When we have trained the surrogate model, we ask it to predict a number of samples
  - Can be simply random sampling
  - Better: *Thompson sampling*
    - fit a Gaussian distribution (a mixture of Gaussians) over the sampled points
    - sample new points close to the means of the fitted Gaussians
- Typical acquisition function: *Expected Improvement*
  - Models the predicted performance as a Gaussian distribution with the predicted mean and standard deviation
  - Computes the *expected* performance improvement over the previous best configuration  $\mathbf{X}^+$ :
$$EI(\mathbf{X}) := \mathbb{E} [\max\{0, f(\mathbf{X}^+) - f_{t+1}(\mathbf{X})\}]$$
- Computing the expected performance requires an integration over the posterior distribution, but has a *closed form solution*.

## Surrogate models

- Surrogate model can be anything as long as it can do regression and is probabilistic
- Gaussian Processes are commonly used
  - Smooth, good extrapolation, but don't scale well to many hyperparameters (cubic)
  - Sparse GPs: select 'inducing points' that minimize info loss, more scalable
  - Multi-task GPs: transfer surrogate models from other tasks
- Random Forests
  - A lot more scalable, but don't extrapolate well
  - Often an interpolation between predictions is used instead of the raw (step-wise) predictions
- Bayesian Neural Networks:
  - Expensive, sensitive to hyperparameters

## Bayesian Optimization: conclusions

- More efficient way to optimize hyperparameters
- More similar to what humans would do
- Harder to parallelize
- Choice of surrogate model depends on your search space
  - Very active research area
  - For very high-dimensional search spaces, random forests are popular

# Lecture 8. Neural Networks

How to train your neurons

Joaquin Vanschoren

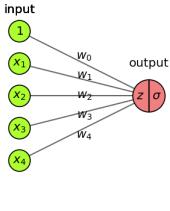
## Overview

- Neural architectures
- Training neural nets
  - Forward pass: Tensor operations
  - Backward pass: Backpropagation
- Neural network design:
  - Activation functions
  - Weight initialization
  - Optimizers
- Neural networks in practice
  - Early stopping
  - Memorization capacity and information bottleneck
  - L1/L2 regularization
  - Dropout
  - Batch normalization
- Model selection

## Linear models as a building block

- Logistic regression, drawn in a different, neuro-inspired, way
  - Linear model: inner product ( $z$ ) of input vector  $\mathbf{x}$  and weight vector  $\mathbf{w}$ , plus bias  $w_0$
  - Logistic (or sigmoid) function maps the output to a probability in [0,1]
  - Uses log loss (cross-entropy) and gradient descent to learn the weights

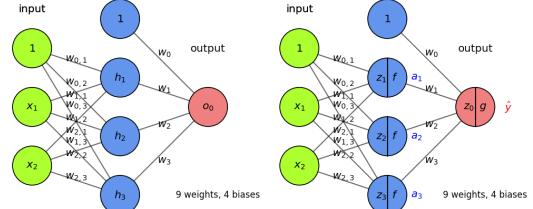
$$\hat{y}(\mathbf{x}) = \text{sigmoid}(z) = \text{sigmoid}(w_0 + \mathbf{w}\mathbf{x}) = \text{sigmoid}(w_0 + w_1 * x_1 + w_2 * x_2 + \dots + w_p * x_p)$$



## Basic Architecture

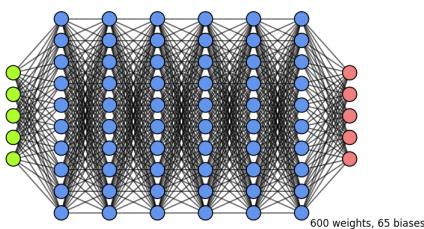
- Add one (or more) **hidden layers**  $h$  with  $k$  nodes (or units, cells, neurons)
- Every 'neuron' is a tiny function, the network is an arbitrarily complex function
- Weights  $w_{i,j}$  between node  $i$  and node  $j$  form a weight matrix  $\mathbf{W}^{(l)}$  per layer  $l$
- Every neuron weights the inputs  $\mathbf{x}$  and passes it through a non-linear activation function
- Activation functions ( $f, g$ ) can be different per layer, output  $\mathbf{a}$  is called activation

$$h(\mathbf{x}) = \mathbf{a} = f(\mathbf{z}) = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{w}_0^{(1)}) \quad o(\mathbf{x}) = g(\mathbf{W}^{(2)}\mathbf{a} + \mathbf{w}_0^{(2)})$$



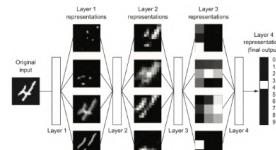
## More layers

- Add more layers, and more nodes per layer, to make the model more complex
  - For simplicity, we don't draw the biases (but remember that they are there)
- In **dense** (fully-connected) layers, every previous layer node is connected to all nodes
- The output layer can also have multiple nodes (e.g. 1 per class in multi-class classification)



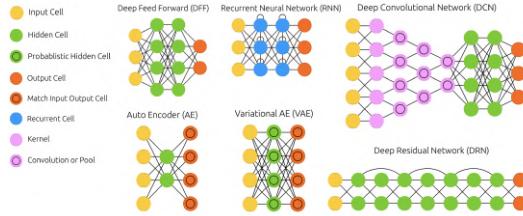
## Why layers?

- Each layer acts as a **filter** and learns a new **representation** of the data
  - Subsequent layers can learn iterative refinements
  - Easier than learning a complex relationship in one go
- Example: for image input, each layer yields new (filtered) images
  - Can learn multiple mappings at once: weight tensor  $W$  yields activation tensor  $A$
  - From low-level patterns (edges, end-points, ...) to combinations thereof
  - Each neuron 'lights up' if certain patterns occur in the input



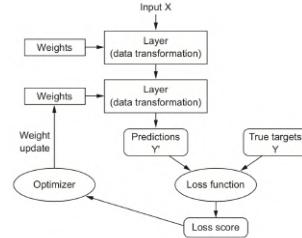
## Other architectures

- There exist MANY types of networks for many different tasks
- Convolutional nets for image data, Recurrent nets for sequential data,...
- Also used to learn representations (embeddings), generate new images, text,...



## Training Neural Nets

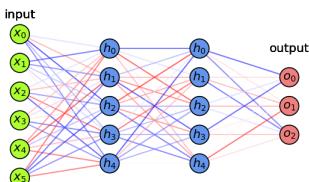
- Design the architecture, choose activation functions (e.g. sigmoids)
- Choose a way to initialize the weights (e.g. random initialization)
- Choose a loss function (e.g. log loss) to measure how well the model fits training data
- Choose an optimizer (typically an SGD variant) to update the weights



## Mini-batch Stochastic Gradient Descent (recap)

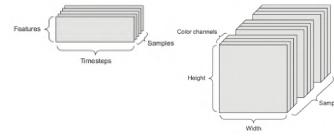
1. Draw a batch of  $batch\_size$  training data  $\mathbf{X}$  and  $y$
2. Forward pass : pass  $\mathbf{X}$  though the network to yield predictions  $\hat{y}$
3. Compute the loss  $\mathcal{L}$  (mismatch between  $\hat{y}$  and  $y$ )
4. Backward pass : Compute the gradient of the loss with regard to every weight
  - Backpropagate the gradients through all the layers
5. Update  $W$ :  $W_{(i+1)} = W_{(i)} - \frac{\partial L(x, W_{(i)})}{\partial W} * \eta$

Repeat until  $n$  passes (epochs) are made through the entire training set



## Forward pass

- We can naturally represent the data as tensors
  - Numerical n-dimensional array (with n axes)
  - 2D tensor: matrix (samples, features)
  - 3D tensor: time series (samples, timesteps, features)
  - 4D tensor: color images (samples, height, width, channels)
  - 5D tensor: video (samples, frames, height, width, channels)

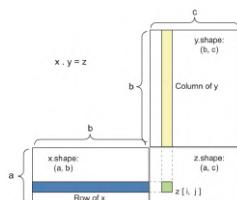


## Tensor operations

- The operations that the network performs on the data can be reduced to a series of tensor operations
  - These are also much easier to run on GPUs
- A dense layer with sigmoid activation, input tensor  $\mathbf{X}$ , weight tensor  $\mathbf{W}$ , bias  $\mathbf{b}$ :

```
y = sigmoid(np.dot(X, W) + b)
```

- Tensor dot product for 2D inputs ( $a$  samples,  $b$  features,  $c$  hidden nodes)



## Element-wise operations

- Activation functions and addition are element-wise operations:

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def add(x, y):
    return x + y
```

- Note: if  $y$  has a lower dimension than  $x$ , it will be broadcasted: axes are added to match the dimensionality, and  $y$  is repeated along the new axes

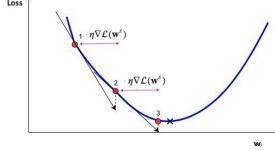
```
>>> np.array([[1,2],[3,4]]) + np.array([10,20])
array([[11, 22],
       [13, 24]])
```

## Backward pass (backpropagation)

- For last layer, compute gradient of the loss function  $\mathcal{L}$  w.r.t all weights of layer  $l$

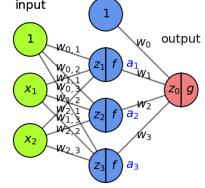
$$\nabla \mathcal{L} = \frac{\partial \mathcal{L}}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{0,0}} & \dots & \frac{\partial \mathcal{L}}{\partial w_{0,l}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{k,0}} & \dots & \frac{\partial \mathcal{L}}{\partial w_{k,l}} \end{bmatrix}$$

- Sum up the gradients for all  $\mathbf{x}_j$  in minibatch:  $\sum_j \frac{\partial \mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}}$
- Update all weights in a layer at once (with learning rate  $\eta$ ):  $W_{(i+1)}^{(l)} = W_{(i)}^{(l)} - \eta \sum_j \frac{\partial \mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}}$
- Repeat for next layer, iterating backwards (most efficient, avoids redundant calculations)



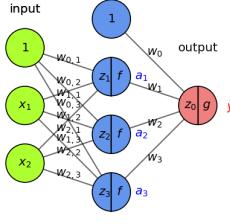
## Backpropagation (example)

- Imagine feeding a single data point, output is  $\hat{y} = g(z) = g(w_0 + w_1 * a_1 + w_2 * a_2 + \dots + w_p * a_p)$
- Decrease loss by updating weights:
  - Update the weights of last layer to maximize improvement:  $w_{i,(new)} = w_i - \frac{\partial \mathcal{L}}{\partial w_i} * \eta$
  - To compute gradient  $\frac{\partial \mathcal{L}}{\partial w_i}$  we need the chain rule:  $f(g(x)) = f'(g(x)) * g'(x)$
- E.g., with  $\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$  and sigmoid  $g: \frac{\partial \mathcal{L}}{\partial w_i} = (\mathbf{y} - \hat{y}) * \sigma'(z_0) * a_i$



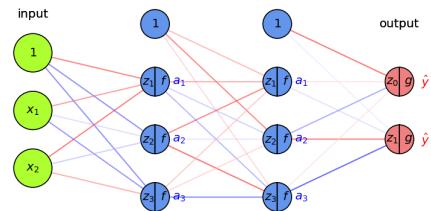
## Backpropagation (2)

- Another way to decrease the loss  $\mathcal{L}$  is to update the activations  $a_i$ 
  - To update  $a_i = f(z_i)$ , we need to update the weights of the previous layer
  - We want to nudge  $a_i$  in the right direction by updating  $w_{i,j}$ :
- $\frac{\partial \mathcal{L}}{\partial w_{i,j}} = \frac{\partial \mathcal{L}}{\partial a_i} \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial w_{i,j}} = \left( \frac{\partial \mathcal{L}}{\partial g} \frac{\partial g}{\partial z_0} \frac{\partial z_0}{\partial a_i} \right) \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial w_{i,j}}$
- We know  $\frac{\partial \mathcal{L}}{\partial g}$  and  $\frac{\partial g}{\partial z_0}$  from the previous step,  $\frac{\partial z_0}{\partial a_i} = w_i$ ,  $\frac{\partial a_i}{\partial z_i} = f'$  and  $\frac{\partial z_i}{\partial w_{i,j}} = x_j$



## Backpropagation (3)

- With multiple output nodes,  $\mathcal{L}$  is the sum of all per-output (per-class) losses
- $\frac{\partial \mathcal{L}}{\partial a_i}$  is sum of the gradients for every output
- Per layer, sum up gradients for every point  $\mathbf{x}$  in the batch:  $\sum_j \frac{\partial \mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}}$
- Update all weights of every layer  $l$
- $W_{(i+1)}^{(l)} = W_{(i)}^{(l)} - \eta \sum_j \frac{\partial \mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}}$
- Repeat with a new batch of data until loss converges
- Nice animation of the entire process

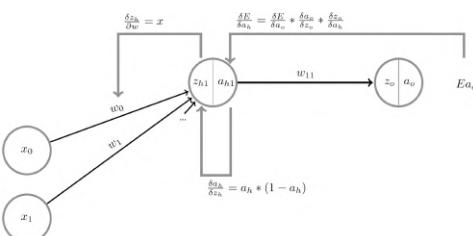


## Backpropagation (summary)

- The network output  $a_o$  is defined by the weights  $W^{(o)}$  and biases  $b^{(o)}$  of the output layer, and
- The activations of a hidden layer  $h_1$  with activation function  $a_{h_1}$ , weights  $W^{(1)}$  and biases  $b^{(1)}$ :  $a_o(\mathbf{x}) = a_o(\mathbf{z}_0) = a_o(W^{(o)} a_{h_1}(z_{h_1}) + b^{(o)}) = a_o(W^{(o)} a_{h_1}(W^{(1)} \mathbf{x} + b^{(1)}) + b^{(o)})$
- Minimize the loss by SGD. For layer  $l$ , compute  $\frac{\partial \mathcal{L}(a_o, x)}{\partial W^{(l)}}$  and  $\frac{\partial \mathcal{L}(a_o, x)}{\partial b^{(l)}}$  using the chain rule

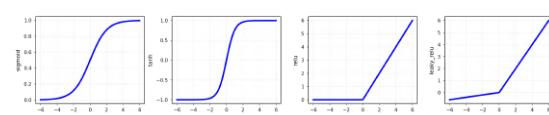
- Decomposes into gradient of layer above, gradient of activation function, gradient of layer input:

$$\frac{\partial \mathcal{L}(a_o)}{\partial W^{(l)}} = \frac{\partial \mathcal{L}(a_o)}{\partial a_{h_1}} \frac{\partial a_{h_1}}{\partial z_{h_1}} \frac{\partial z_{h_1}}{\partial W^{(l)}} = \left( \frac{\partial \mathcal{L}(a_o)}{\partial a_{h_1}} \frac{\partial a_{h_1}}{\partial z_{h_1}} \frac{\partial z_{h_1}}{\partial a_{h_1}} \right) \frac{\partial a_{h_1}}{\partial z_{h_1}} \frac{\partial z_{h_1}}{\partial W^{(l)}}$$



## Activation functions for hidden layers

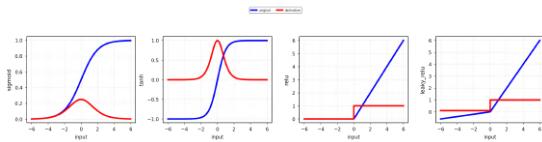
- Sigmoid:  $f(z) = \frac{1}{1+e^{-z}}$
- Tanh:  $f(z) = \frac{2}{1+e^{-2z}} - 1$ 
  - Activations around 0 are better for gradient descent convergence
- Rectified Linear (ReLU):  $f(z) = \max(0, z)$ 
  - Less smooth, but much faster (note: not differentiable at 0)
- Leaky ReLU:  $f(z) = \begin{cases} 0.01z & z < 0 \\ z & \text{otherwise} \end{cases}$



### Effect of activation functions on the gradient

- During gradient descent, the gradient depends on the activation function  $a_h$ :  

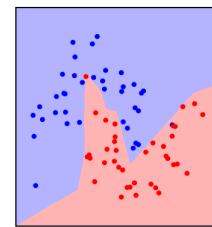
$$\frac{\partial \mathcal{L}(a_h)}{\partial W^{(l)}} = \frac{\partial \mathcal{L}(a_h)}{\partial a_h} \frac{\partial a_h}{\partial z_h} \frac{\partial z_h}{\partial W^{(l)}}$$
- If derivative of the activation function  $\frac{\partial a_h}{\partial z_h}$  is 0, the weights  $w_l$  are not updated
  - Moreover, the gradients of previous layers will be reduced (vanishing gradient)
- sigmoid, tanh: gradient is very small for large inputs: slow updates
- With ReLU,  $\frac{\partial a_h}{\partial z_h} = 1$  if  $z > 0$ , hence better against vanishing gradients
  - Problem: for very negative inputs, the gradient is 0 and may never recover (dying ReLU)
  - Leaky ReLU has a small (0.01) gradient there to allow recovery



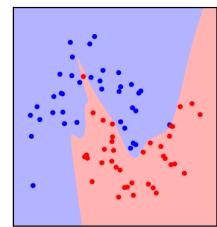
### ReLU vs Tanh

- What is the effect of using non-smooth activation functions?
  - ReLU produces piecewise-linear boundaries, but allows deeper networks
  - Tanh produces smoother decision boundaries, but is slower

ReLU, acc: 0.84, time: 0.03 sec

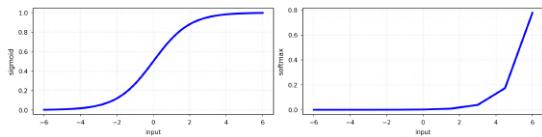


tanh, acc: 0.84, time: 0.03 sec



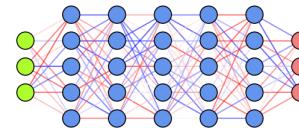
### Activation functions for output layer

- sigmoid converts output to probability in [0,1]
  - For binary classification
- softmax converts all outputs (aka 'logits') to probabilities that sum up to 1
  - For multi-class classification ( $k$  classes)
  - Can cause over-confident models. If so, smooth the labels:  $y_{smooth} = (1 - \alpha)y + \frac{\alpha}{k}$
- softmax( $x, i$ ) = 
$$\frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$
- For regression, don't use any activation function, let the model learn the exact target



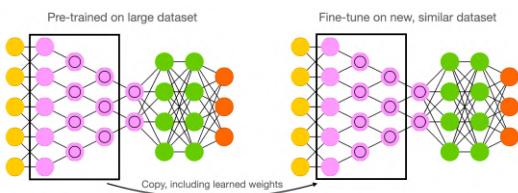
### Weight initialization

- Initializing weights to 0 is bad: all gradients in layer will be identical (symmetry)
- Too small random weights shrink activations to 0 along the layers (vanishing gradient)
- Too large random weights multiply along layers (exploding gradient, zig-zagging)
- Ideal: small random weights + variance of input and output gradients remains the same
  - Glorot/Xavier initialization (for tanh): randomly sample from  $N(0, \sigma)$ ,  $\sigma = \sqrt{\frac{2}{fan\_in + fan\_out}}$ 
    - fan\_in: number of input units, fan\_out: number of output units
  - He initialization (for ReLU): randomly sample from  $N(0, \sigma)$ ,  $\sigma = \sqrt{\frac{2}{fan\_in}}$
  - Uniform sampling (instead of  $N(0, \sigma)$ ) for deeper networks (w.r.t. vanishing gradients)



### Weight initialization: transfer learning

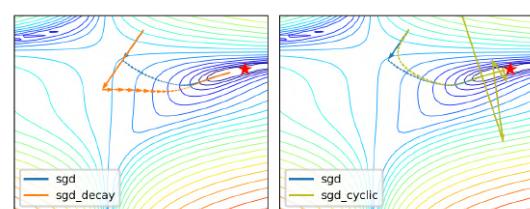
- Instead of starting from scratch, start from weights previously learned from similar tasks
  - This is, to a big extent, how humans learn so fast
- Transfer learning: learn weights on task T, transfer them to new network
  - Weights can be frozen, or finetuned to the new data
- Only works if the previous task is 'similar' enough
  - Meta-learning: learn a good initialization across many related tasks



### Optimizers

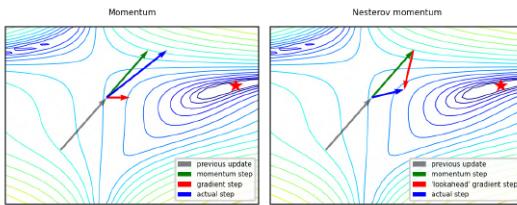
#### SGD with learning rate schedules

- Using a constant learning  $\eta$  rate for weight updates  $\mathbf{w}_{(s+1)} = \mathbf{w}_s - \eta \nabla \mathcal{L}(\mathbf{w}_s)$  is not ideal
- Learning rate decay/annealing with decay rate  $k$ 
  - E.g. exponential ( $\eta_{s+1} = \eta_s e^{-ks}$ ), inverse-time ( $\eta_{s+1} = \frac{\eta_0}{1+ks}$ )...
- Cyclic learning rates
  - Change from small to large: hopefully in 'good' region long enough before diverging
  - Warm restarts: aggressive decay + reset to initial learning rate

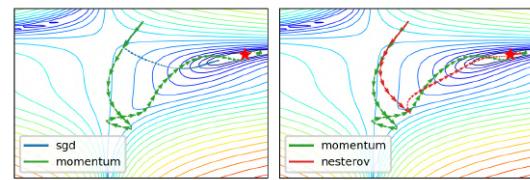


## Momentum

- Imagine a ball rolling downhill: accumulates momentum, doesn't exactly follow steepest descent
  - Reduces oscillation, follows larger (consistent) gradient of the loss surface
- Adds a velocity vector  $\mathbf{v}$  with momentum  $\gamma$  (e.g. 0.9, or increase from  $\gamma = 0.5$  to  $\gamma = 0.99$ )
 
$$\mathbf{w}_{(s+1)} = \mathbf{w}_s + \mathbf{v}_s \quad \text{with} \quad \mathbf{v}_{(s)} = \gamma \mathbf{v}_{(s-1)} - \eta \nabla \mathcal{L}(\mathbf{w}_{(s)})$$
- Nesterov momentum: Look where momentum step would bring you, compute gradient there
  - Responds faster (and reduces momentum) when the gradient changes
 
$$\mathbf{v}_{(s)} = \gamma \mathbf{v}_{(s-1)} - \eta \nabla \mathcal{L}(\mathbf{w}_{(s)} + \gamma \mathbf{v}_{(s-1)})$$



Momentum in practice

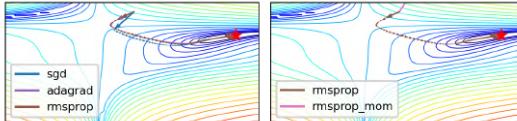


## Adaptive gradients

- 'Correct' the learning rate for each  $w_i$  based on specific local conditions (layer depth, fan-in,...)
  - Adagrad: scale  $\eta$  according to squared sum of previous gradients  $G_{i,(s)} = \sum_{t=1}^s \mathcal{L}(w_{i,(t)})^2$ 
    - Update rule for  $w_i$ . Usually  $\epsilon = 10^{-7}$  (avoids division by 0),  $\eta = 0.001$ 

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{\sqrt{G_{i,(s)} + \epsilon}} \nabla \mathcal{L}(w_{i,(s)})$$
  - RMSProp: use moving average of squared gradients  $m_{i,(s)} = \gamma m_{i,(s-1)} + (1 - \gamma) \nabla \mathcal{L}(w_{i,(s)})^2$ 
    - Avoids that gradients dwindle to 0 as  $G_{i,(s)}$  grows. Usually  $\gamma = 0.9$ ,  $\eta = 0.001$ 

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{\sqrt{m_{i,(s)} + \epsilon}} \nabla \mathcal{L}(w_{i,(s)})$$



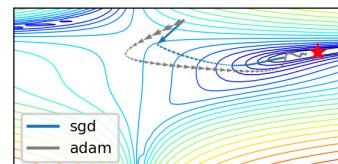
## Adam (Adaptive moment estimation)

- Adam: RMSProp + momentum. Adds moving average for gradients as well ( $\gamma_2 = \text{momentum}$ ):
  - Adds a bias correction to avoid small initial gradients:  $\hat{m}_{i,(s)} = \frac{m_{i,(s)}}{1-\gamma_1}$  and  $\hat{g}_{i,(s)} = \frac{g_{i,(s)}}{1-\gamma_2}$ 

$$g_{i,(s)} = \gamma_2 g_{i,(s-1)} + (1 - \gamma_2) \nabla \mathcal{L}(w_{i,(s)})$$

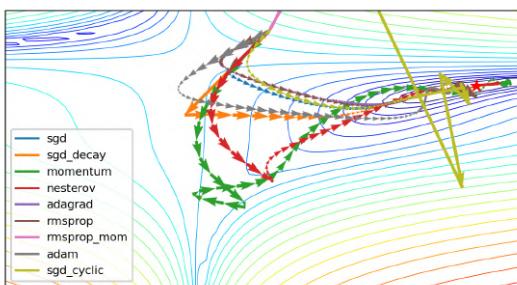
$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{\sqrt{\hat{m}_{i,(s)} + \epsilon}} \hat{g}_{i,(s)}$$
- Adamax: Idem, but use max() instead of moving average:  $u_{i,(s)} = \max(\gamma u_{i,(s-1)}, |\mathcal{L}(w_{i,(s)})|)$ 

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{u_{i,(s)}} \hat{g}_{i,(s)}$$



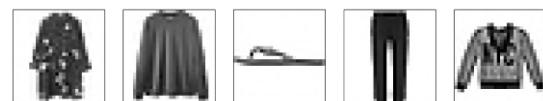
## SGD Optimizer Zoo

- RMSProp often works well, but do try alternatives. For even more optimizers, see here.



## Neural networks in practice

- There are many practical courses on training neural nets. E.g.:
  - With TensorFlow: <https://www.tensorflow.org/resources/learn-ml>
  - With PyTorch: [fast.ai course, https://pytorch.org/tutorials/](https://pytorch.org/tutorials/)
- Here, we'll use Keras, a general API for building neural networks
  - Default API for TensorFlow, also has backends for CNTK, Theano
  - Focus on key design decisions, evaluation, and regularization
  - Running example: Fashion-MNIST
    - 28x28 pixel images of 10 classes of fashion items



## Building the network

- We first build a simple sequential model (no branches)
- Input layer ('input\_shape'): a flat vector of 28\*28=784 nodes
  - We'll see how to properly deal with images later
- Two dense hidden layers: 512 nodes each, ReLU activation
  - Glorot weight initialization is applied by default
- Output layer: 10 nodes (for 10 classes) and softmax activation

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
kernel_initializer='he_normal', input_shape=(28 * 28,)))
network.add(layers.Dense(512, activation='relu',
kernel_initializer='he_normal'))
network.add(layers.Dense(10, activation='softmax'))
```

## Model summary

- Lots of parameters (weights and biases) to learn!
  - hidden layer 1 :  $(28 \cdot 28 + 1) \cdot 512 = 401920$
  - hidden layer 2 :  $(512 + 1) \cdot 512 = 262656$
  - output layer:  $(512 + 1) \cdot 10 = 5130$

```
network.summary()
```

| Model: "sequential" |              |         |
|---------------------|--------------|---------|
| Layer (type)        | Output Shape | Param # |
| dense (Dense)       | (None, 512)  | 401920  |
| dense_1 (Dense)     | (None, 512)  | 262656  |
| dense_2 (Dense)     | (None, 10)   | 5130    |

Total params: 669,706  
Trainable params: 669,706  
Non-trainable params: 0

## Choosing loss, optimizer, metrics

- Loss function**
  - Cross-entropy (log loss) for multi-class classification ( $y_{true}$  is one-hot encoded)
  - Use binary crossentropy for binary problems (single output node)
  - Use sparse categorical crossentropy if  $y_{true}$  is label-encoded (1,2,3,...)
- Optimizer**
  - Any of the optimizers we discussed before. RMSprop usually works well.
- Metrics**
  - To monitor performance during training and testing, e.g. accuracy

```
# Shorthand
network.compile(loss='categorical_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])

# Detailed
network.compile(loss=CategoricalCrossentropy(label_smoothing=0.01),
optimizer=RMSprop(learning_rate=0.001, momentum=0.0),
metrics=[Accuracy()])
```

## Preprocessing: Normalization, Reshaping, Encoding

- Always normalize (standardize or min-max) the inputs. Mean should be close to 0.
  - Avoid that some inputs overpower others
  - Speed up convergence
    - Gradients of activation functions  $\frac{\partial u_i}{\partial z_k}$  are (near) 0 for large inputs
    - If some gradients become much larger than others, SGD will start zig-zagging
- Reshape the data to fit the shape of the input layer, e.g. (n, 28\*28) or (n, 28, 28)
  - Tensor with instances in first dimension, rest must match the input layer
- In multi-class classification, every class is an output node, so one-hot-encode the labels
  - e.g. class '4' becomes [0,0,0,1,0,0,0,0]

```
x = X.astype('float32') / 255
X = X.reshape((60000, 28 * 28))
y = to_categorical(y)
```

## Choosing training hyperparameters

- Number of epochs: enough to allow convergence
  - Too much: model starts overfitting (or just wastes time)
- Batch size: small batches (e.g. 32, 64,... samples) often preferred
  - 'Noisy' training data makes overfitting less likely
    - Larger batches generalize less well ('generalization gap')
  - Requires less memory (especially in GPUs)
  - Large batches do speed up training, may converge in fewer epochs
- Batch size interacts with learning rate
  - Instead of shrinking the learning rate you can increase batch size

```
history = network.fit(X_train, y_train, epochs=3, batch_size=32);
```

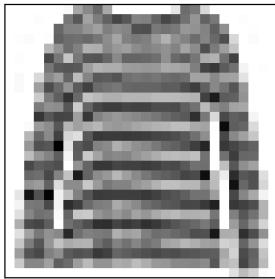
```
Epoch 1/3
1875/1875 [=====] - 24s 13ms/step - loss: 0.4331 - accuracy: 0.8529
Epoch 2/3
1875/1875 [=====] - 25s 13ms/step - loss: 0.4242 - accuracy: 0.8568
Epoch 3/3
1875/1875 [=====] - 26s 14ms/step - loss: 0.4183 - accuracy: 0.8573
```

## Predictions and evaluations

We can now call `predict` to generate predictions, and evaluate the trained model on the entire test set

```
network.predict(X_test)
test_loss, test_acc = network.evaluate(X_test, y_test)
```

```
[0.0240177 0.0001167 0.4472437 0.0056629 0.057807 0.000094 0.4632739
0.0000267 0.0017463 0.0000112]
```

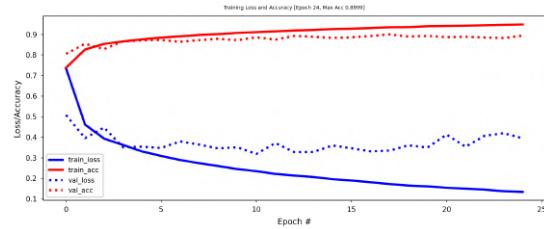


True label: [0. 0. 1. 0. 0. 0. 0. 0. 0.]

```
313/313 [=====] - 2s 7ms/step - loss: 0.3845 - accuracy: 0.8636
Test accuracy: 0.8636000156402588
```

## Model selection

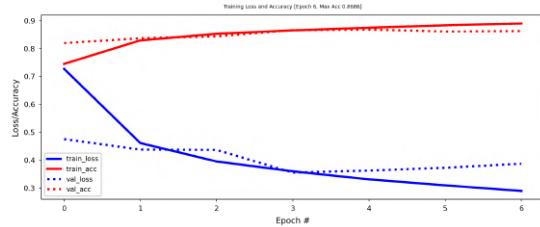
- How many epochs do we need for training?
- Train the neural net and track the loss after every iteration on a validation set
  - You can add a callback to the fit version to get info on every epoch
- Best model after a few epochs, then starts overfitting



## Early stopping

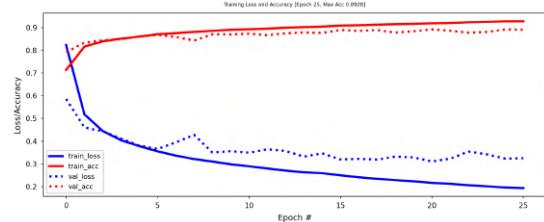
- Stop training when the validation loss (or validation accuracy) no longer improves
- Loss can be bumpy: use a moving average or wait for k steps without improvement

```
earlystop = callbacks.EarlyStopping(monitor='val_loss', patience=3)
model.fit(x_train, y_train, epochs=25, batch_size=512, callbacks=[earlystop])
```



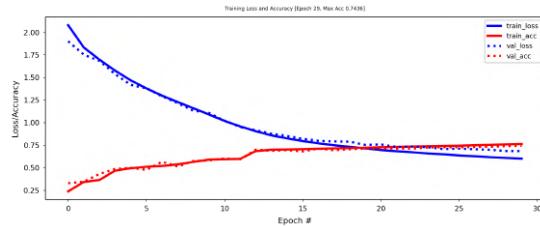
## Regularization and memorization capacity

- The number of learnable parameters is called the **model capacity**
- A model with more parameters has a higher **memorization capacity**
  - Too high capacity causes overfitting, too low causes underfitting
  - In the extreme, the training set can be 'memorized' in the weights
- Smaller models are forced to learn a compressed representation that generalizes better
  - Find the sweet spot: e.g. start with few parameters, increase until overfitting starts.
- Example: 256 nodes in first layer, 32 nodes in second layer, similar performance



## Information bottleneck

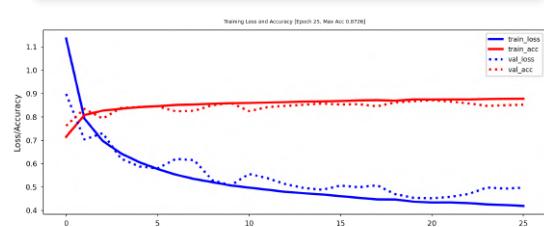
- If a layer is too narrow, it will lose information that can never be recovered by subsequent layers
- **Information bottleneck** theory defines a bound on the capacity of the network
- Imagine that you need to learn 10 outputs (e.g. classes) and your hidden layer has 2 nodes
  - This is like trying to learn 10 hyperplanes from a 2-dimensional representation
- Example: bottleneck of 2 nodes, no overfitting, much higher training loss



## Weight regularization (weight decay)

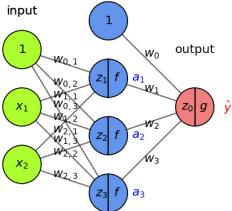
- As we did many times before, we can also add weight regularization to our loss function
- L1 regularization: leads to sparse networks with many weights that are 0
- L2 regularization: leads to many very small weights

```
network = models.Sequential()
network.add(layers.Dense(256, activation='relu',
kernel_regularizer=regularizers.l2(0.001), input_shape=(28 * 28,)))
network.add(layers.Dense(128, activation='relu',
kernel_regularizer=regularizers.l2(0.001)))
```



## Dropout

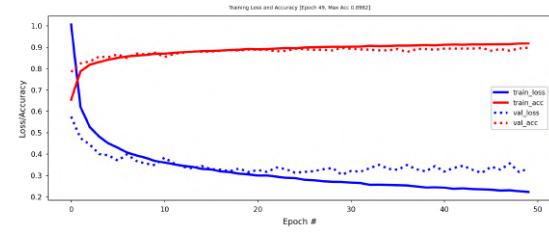
- Every iteration, randomly set a number of activations  $a_i$  to 0
- Dropout rate*: fraction of the outputs that are zeroed-out (e.g. 0.1 - 0.5)
- Idea: break up accidental non-significant learned patterns
- At test time, nothing is dropped out, but the output values are scaled down by the dropout rate
  - Balances out that more units are active than during training



## Dropout layers

- Dropout is usually implemented as a special layer

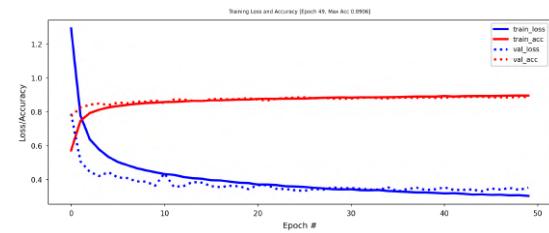
```
network = models.Sequential()
network.add(layers.Dense(256, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dropout(0.5))
network.add(layers.Dense(32, activation='relu'))
network.add(layers.Dropout(0.5))
network.add(layers.Dense(10, activation='softmax'))
```



## Batch Normalization

- We've seen that scaling the input is important, but what if layer activations become very large?
  - Same problems, starting deeper in the network
- Batch normalization: normalize the activations of the previous layer within each batch
  - Within a batch, set the mean activation close to 0 and the standard deviation close to 1
    - Across batches, use exponential moving average of batch-wise mean and variance
  - Allows deeper networks less prone to vanishing or exploding gradients

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(256, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(64, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(32, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
```



## Tuning multiple hyperparameters

- You can wrap Keras models as scikit-learn models and use any tuning technique
- Keras also has built-in RandomSearch (and HyperBand and BayesianOptimization - see later)

```
def make_model(hp):
    m.add(Dense(units=hp.Int('units', min_value=32, max_value=512,
                           step=32)))
    m.compile(optimizer=Adam(hp.Choice('learning_rate', [1e-2, 1e-3,
                           1e-4])))
    return model

from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
clf = KerasClassifier(make_model)
grid = GridSearchCV(clf, param_grid=param_grid, cv=3)

from kerastuner.tuners import RandomSearch
tuner = keras.RandomSearch(build_model, max_trials=5)
```

## Summary

- Neural architectures
- Training neural nets
  - Forward pass: Tensor operations
  - Backward pass: Backpropagation
- Neural network design:
  - Activation functions
  - Weight initialization
  - Optimizers
- Neural networks in practice
- Model selection
  - Early stopping
  - Memorization capacity and information bottleneck
  - L1/L2 regularization
  - Dropout
  - Batch normalization

# Lecture 9: Convolutional Neural Networks

## Handling image data

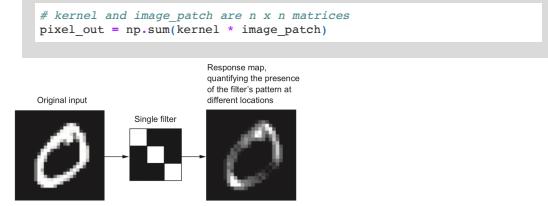
Joaquin Vanschoren, Eindhoven University of Technology

## Overview

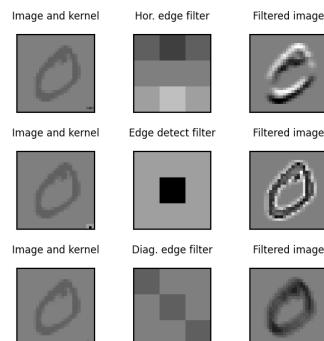
- Image convolution
- Convolutional neural networks
- Data augmentation
- Model interpretation
- Using pre-trained networks (transfer learning)

## Convolution

- Operation that transforms an image by sliding a smaller image (called a *filter* or *kernel*) over the image and multiplying the pixel values
  - Slide an  $n \times n$  filter over  $n \times n$  patches of the original image
  - Every pixel is replaced by the sum of the element-wise products of the values of the image patch around that pixel and the kernel



- Different kernels can detect different types of patterns in the image

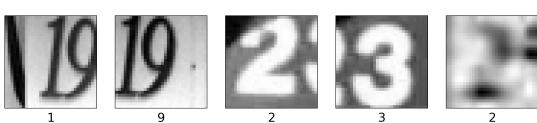


## Demonstration on Google streetview data

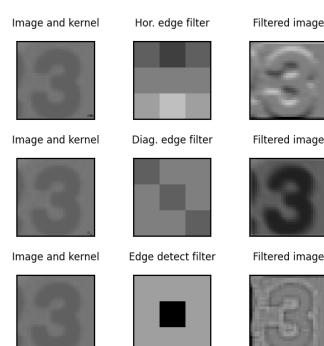
House numbers photographed from Google streetview imagery, cropped and centered around digits, but with neighboring numbers or other edge artifacts.



For recognizing digits, color is not important, so we grayscale the images



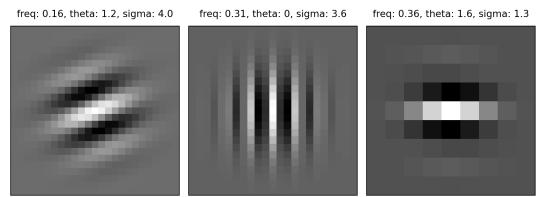
## Demonstration



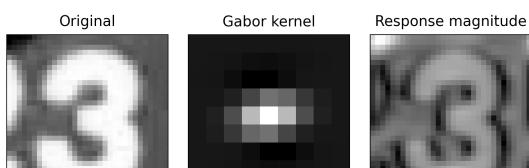
## Image convolution in practice

- How do we know which filters are best for a given image?
- Families of kernels (or *filter banks*) can be run on every image
  - Gabor, Sobel, Haar Wavelets,...
- Gabor filters: Wave patterns generated by changing:
  - Frequency: narrow or wide oscillations
  - Theta: angle (direction) of the wave
  - Sigma: resolution (size of the filter)

## Demonstration

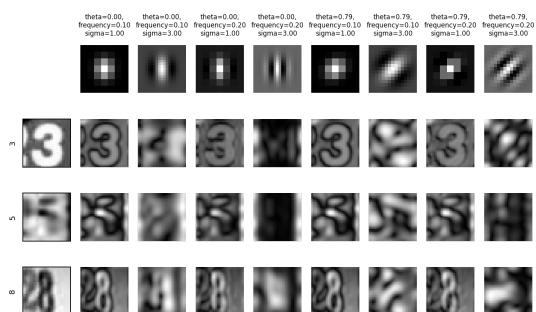


## Demonstration on the streetview data



## Filter banks

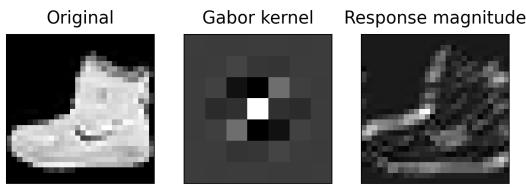
- Different filters detect different edges, shapes,...
- Not all seem useful



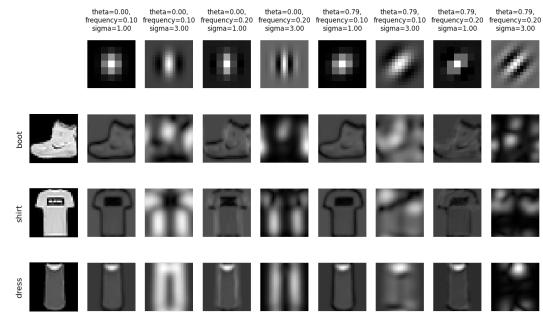
## Another example: Fashion MNIST



### Demonstration

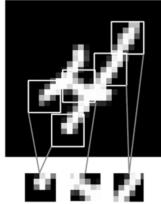


Fashion MNIST with multiple filters (filter bank)



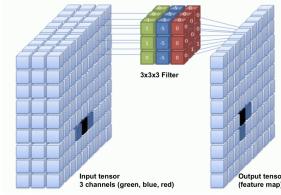
## Convolutional neural nets

- Finding relationships between individual pixels and the correct class is hard
- We want to discover 'local' patterns (edges, lines, endpoints)
- Representing such local patterns as features makes it easier to learn from them
- We could use convolutions, but how to choose the filters?



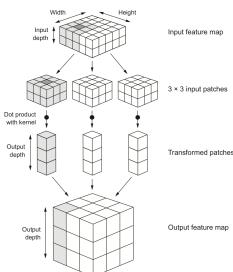
## Convolutional Neural Networks (ConvNets)

- Instead of manually designing the filters, we can also *learn* them based on data
  - Choose filter sizes (manually), initialize with small random weights
- Forward pass: Convolutional layer slides the filter over the input, generates the output
- Backward pass: Update the filter weights according to the loss gradient
- Illustration for 1 filter:



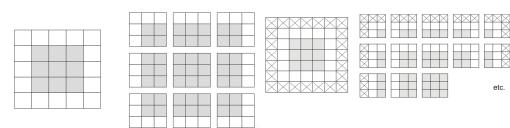
## Convolutional layers: Feature maps

- One filter is not sufficient to detect all relevant patterns in an image
- A convolutional layer applies and learns  $d$  filter in parallel
- Slide  $d$  filters across the input image (in parallel)  $\rightarrow$  a  $(1 \times 1 \times d)$  output per patch
- Reassemble into a *feature map* with  $d$  'channels', a  $(\text{width} \times \text{height} \times d)$  tensor.



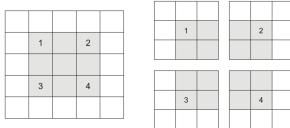
## Border effects (zero padding)

- Consider a  $5 \times 5$  image and a  $3 \times 3$  filter: there are only 9 possible locations, hence the output is a  $3 \times 3$  feature map
- If we want to maintain the image size, we use zero-padding, adding 0's all around the input tensor.



## Undersampling (striding)

- Sometimes, we want to downsample a high-resolution image
  - Faster processing, less noisy (hence less overfitting)
- One approach is to skip values during the convolution
  - Distance between 2 windows: stride length
- Example with stride length 2 (without padding):



## Max-pooling

- Another approach to shrink the input tensors is max-pooling:
  - Run a filter with a fixed stride length over the image
    - Usually 2x2 filters and stride length 2
    - The filter simply returns the max (or avg) of all values
  - Agressively reduces the number of weights (less overfitting)
  - Information from every input node spreads more quickly to output nodes
    - In pure convnets, one input value spreads to 3x3 nodes of the first layer, 5x5 nodes of the second, etc.
    - Without maxpooling, you need much deeper networks, harder to train
  - Increases translation invariance: patterns can affect the predictions no matter where they occur in the image

## Convolutional nets in practice

- Use multiple convolutional layers to learn patterns at different levels of abstraction
  - Find local patterns first (e.g. edges), then patterns across those patterns
- Use MaxPooling layers to reduce resolution, increase translation invariance
- Use sufficient filters in the first layer (otherwise information gets lost)
- In deeper layers, use increasingly more filters
  - Preserve information about the input as resolution decreases
  - Avoid decreasing the number of activations (resolution x nr of filters)
- For very deep nets, add skip connections to preserve information (and gradients)
  - Sums up outputs of earlier layers to those of later layers (with same dimensions)

### Example with Keras:

- Conv2D for 2D convolutional layers
  - 32 filters (default), randomly initialized (from uniform distribution)
  - Deeper layers use 64 filters
  - Filter size is 3x3
  - ReLU activation to simplify training of deeper networks
- MaxPooling2D for max-pooling
  - 2x2 pooling reduces the number of inputs by a factor 4

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Observe how the input image on 28x28x1 is transformed to a 3x3x64 feature map

- Convolutional layer:
  - No zero-padding: every output 2 pixels less in every dimension
  - 320 weights: (3x3 filter weights + 1 bias) \* 32 filters
- After every MaxPooling, resolution halved in every dimension

```
Model: "sequential"
Layer (type)      Output Shape         Param #
=====
conv2d (Conv2D)   (None, 26, 26, 32)    320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)  0
conv2d_1 (Conv2D) (None, 11, 11, 64)    18496
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 64)  0
conv2d_2 (Conv2D) (None, 3, 3, 64)    36928
=====
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

### Completing the network

- To classify the images, we still need a Dense and Softmax layer.
- We need to flatten the 3x3x64 feature map to a vector of size 576

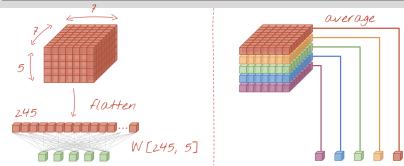
```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

### Complete network

```
Model: "sequential"
Layer (type)          Output Shape         Param #
===
conv2d (Conv2D)        (None, 26, 26, 32)      320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)      0
)
conv2d_1 (Conv2D)      (None, 11, 11, 64)     18496
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 64)      0
2D)
conv2d_2 (Conv2D)      (None, 3, 3, 64)      36928
flatten (Flatten)      (None, 576)            0
dense (Dense)          (None, 64)             36928
dense_1 (Dense)        (None, 10)             650
=====
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

- Flatten adds a lot of weights
- Instead, we can do GlobalAveragePooling : returns average of each activation map
- This sometimes works even without adding a dense layer (the number of outputs is similar to the number of classes)

```
model.add(layers.GlobalAveragePooling2D())
model.add(layers.Dense(10, activation='softmax'))
```



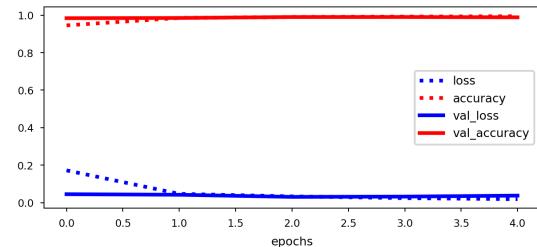
- With GlobalAveragePooling: much fewer weights to learn
- Use with caution: this destroys the location information learned by the CNN
- Not ideal for tasks such as object localization

```
Model: "sequential_1"
Layer (type)          Output Shape         Param #
===
conv2d_3 (Conv2D)      (None, 26, 26, 32)      320
max_pooling2d_2 (MaxPooling2D) (None, 13, 13, 32)      0
)
conv2d_4 (Conv2D)      (None, 11, 11, 64)     18496
max_pooling2d_3 (MaxPooling2D) (None, 5, 5, 64)      0
2D)
conv2d_5 (Conv2D)      (None, 3, 3, 64)      36928
global_average_pooling2d (GlobalAveragePooling2D) (None, 64)      0
dense_2 (Dense)        (None, 10)             650
=====
Total params: 56,394
Trainable params: 56,394
```

### Run the model on MNIST dataset

- Train and test as usual: 99% accuracy
- Compared to 97.8% accuracy with the dense architecture

Accuracy: 0.98880048828125



### Tip:

- Training ConvNets can take a lot of time
- Save the trained model (and history) to disk so that you can reload it later

```
model.save(os.path.join(model_dir, 'mnist.h5'))
with open(os.path.join(model_dir, 'mnist_history.p'), 'wb') as file_pi:
    pickle.dump(history.history, file_pi)
```

### Cats vs Dogs

- A more realistic dataset: Cats vs Dogs
  - Colored JPEG images, different sizes
  - Not nicely centered, translation invariance is important
- Preprocessing
  - Create balanced subsample of 4000 colored images
    - 3000 for training, 1000 validation
  - Decode JPEG images to floating-point tensors
  - Rescale pixel values to [0,1]
  - Resize images to 150x150 pixels

## Data generators

- `ImageDataGenerator`: allows to encode, resize, and rescale JPEG images
- Returns a Python generator we can endlessly query for batches of images
- Separately for training, validation, and test set

```
train_generator =  
ImageDataGenerator(rescale=1./255).flow_from_directory(  
    train_dir,  
    target_size=(150, 150), # Resize images  
    batch_size=20,  
    class_mode='binary')
```

Since the images are larger and more complex, we add another convolutional layer and increase the number of filters to 128.

```
model = models.Sequential()  
model.add(layers.Conv2D(32, (3, 3), activation='relu',  
    input_shape=(150, 150, 3)))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(64, (3, 3), activation='relu'))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(128, (3, 3), activation='relu'))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Conv2D(128, (3, 3), activation='relu'))  
model.add(layers.MaxPooling2D((2, 2)))  
model.add(layers.Flatten())  
model.add(layers.Dense(512, activation='relu'))  
model.add(layers.Dense(1, activation='sigmoid'))
```

```
Model: "sequential_1"  
Layer (type) Output Shape Param #  
===== ====== ====== ======  
conv2d_3 (Conv2D) (None, 148, 148, 32) 896  
max_pooling2d_2 (MaxPooling 2D) (None, 74, 74, 32) 0  
conv2d_4 (Conv2D) (None, 72, 72, 64) 18496  
max_pooling2d_3 (MaxPooling 2D) (None, 36, 36, 64) 0  
conv2d_5 (Conv2D) (None, 34, 34, 128) 73856  
max_pooling2d_4 (MaxPooling 2D) (None, 17, 17, 128) 0  
conv2d_6 (Conv2D) (None, 15, 15, 128) 147584  
max_pooling2d_5 (MaxPooling 2D) (None, 7, 7, 128) 0  
flatten_1 (Flatten) (None, 6272) 0  
dense_2 (Dense) (None, 512) 3211776  
dense_3 (Dense) (None, 1) 513  
===== ====== ====== ======  
Total params: 3,453,121  
Trainable params: 3,453,121  
Non-trainable params: 0
```

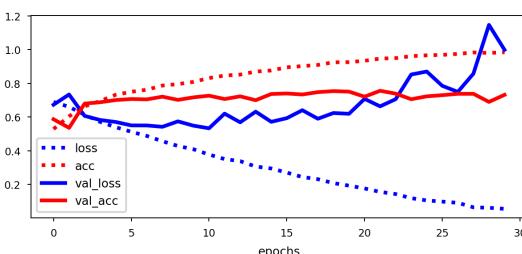
## Training

- The `fit` function also supports generators
  - 100 steps per epoch (batch size: 20 images per step), for 30 epochs
  - Provide a separate generator for the validation data

```
model.compile(loss='binary_crossentropy',  
    optimizer=optimizers.RMSprop(lr=1e-4),  
    metrics=['acc'])  
history = model.fit(  
    train_generator, steps_per_epoch=100,  
    epochs=30, verbose=0,  
    validation_data=validation_generator,  
    validation_steps=50)
```

## Results

- The network seems to be overfitting. Validation accuracy is stuck at 75% while the training accuracy reaches 100%
- There are many things we can do:
  - Regularization (e.g. Dropout, L1/L2, Batch Normalization,...)
  - Generating more training data
  - Meta-learning: Use pretrained rather than randomly initialized filters



## Data augmentation

- Generate new images via image transformations
  - Images will be randomly transformed every epoch
- We can again use a data generator to do this

```
datagen = ImageDataGenerator(  
    rotation_range=40, # Rotate image up to 40 degrees  
    width_shift_range=0.2, # Shift image left-right up to 20% of  
    image width  
    height_shift_range=0.2, # Shift image up-down up to 20% of  
    image height  
    shear_range=0.2, # Shear (slant) the image up to 0.2  
    degrees  
    zoom_range=0.2, # Zoom in up to 20%  
    horizontal_flip=True, # Horizontally flip the image  
    fill_mode='nearest')
```

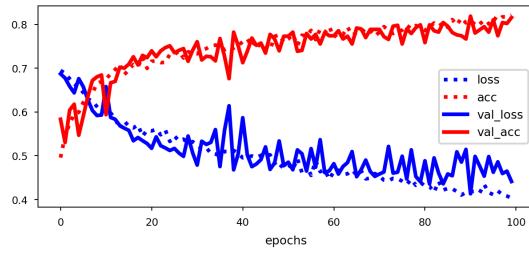
### Example



We also add Dropout before the Dense layer

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

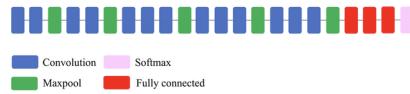
(Almost) no more overfitting!



### Real-world CNNs

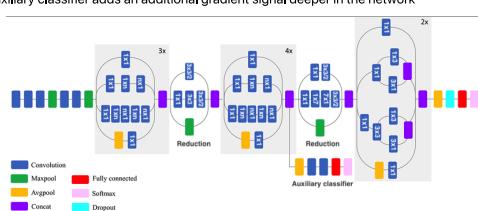
#### VGG16

- Deeper architecture (16 layers): allows it to learn more complex high-level features
  - Textures, patterns, shapes,...
- Small filters (3x3) work better: capture spatial information while reducing number of parameters
- Max-pooling (2x2): reduces spatial dimension, improves translation invariance
  - Lower resolution forces model to learn robust features (less sensitive to small input changes)
  - Only after every 2 layers, otherwise dimensions reduce too fast
- Downside: too many parameters, expensive to train



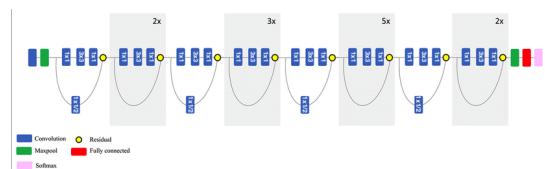
### Inceptionv3

- Inception modules: parallel branches learn features of different sizes and scales (3x3, 5x5, 7x7,...)
  - Add reduction blocks that reduce dimensionality via convolutions with stride 2
- Factorized convolutions: a 3x3 conv. can be replaced by combining 1x3 and 3x1, and is 33% cheaper
  - A 5x5 can be replaced by combining 3x3 and 3x1, which can in turn be factorized as above
- 1x1 convolutions, or Network-In-Network (NIN) layers help reduce the number of channels: cheaper
- An auxiliary classifier adds an additional gradient signal deeper in the network



### ResNet50

- Residual (skip) connections: add earlier feature map to a later one (dimensions must match)
  - Information can bypass layers, reduces vanishing gradients, allows much deeper nets
- Residual blocks: skip small number of layers and repeat many times
  - Match dimensions through padding and 1x1 convolutions
  - When resolution drops, add 1x1 convolutions with stride 2
- Can be combined with Inception blocks



## Interpreting the model

- Let's see what the convnet is learning exactly by observing the intermediate feature maps
  - A layer's output is also called its **activation**
- We can choose a specific test image, and observe the outputs
- We can retrieve and visualize the activation for every filter for every layer

- Layer 0: has activations of resolution 148x148 for each of its 32 filters
- Layer 2: has activations of resolution 72x72 for each of its 64 filters
- Layer 4: has activations of resolution 34x34 for each of its 128 filters
- Layer 6: has activations of resolution 15x15 for each of its 128 filters

| Model: "sequential_3"            |                      |         |
|----------------------------------|----------------------|---------|
| Layer (type)                     | Output Shape         | Param # |
| conv2d_10 (Conv2D)               | (None, 148, 148, 32) | 896     |
| max_pooling2d_8 (MaxPooling 2D)  | (None, 74, 74, 32)   | 0       |
| conv2d_11 (Conv2D)               | (None, 72, 72, 64)   | 18496   |
| max_pooling2d_9 (MaxPooling 2D)  | (None, 36, 36, 64)   | 0       |
| conv2d_12 (Conv2D)               | (None, 34, 34, 128)  | 73856   |
| max_pooling2d_10 (MaxPooling 2D) | (None, 17, 17, 128)  | 0       |
| conv2d_13 (Conv2D)               | (None, 15, 15, 128)  | 147584  |
| max_pooling2d_11 (MaxPooling 2D) | (None, 7, 7, 128)    | 0       |
| flatten_2 (Flatten)              | (None, 6272)         | 0       |
| dropout (Dropout)                | (None, 6272)         | 0       |
| dense_4 (Dense)                  | (None, 512)          | 3211776 |
| dense_5 (Dense)                  | (None, 1)            | 513     |

Total params: 3,453,121  
Trainable params: 3,453,121  
Non-trainable params: 0

Output of the first Conv2D layer, 3rd channel (filter):

- Similar to a diagonal edge detector
- Your own channels may look different

- To extract the activations, we create a new model that outputs the trained layers
  - 8 output layers in total (only the convolutional part)
- We input a test image for prediction and then read the relevant outputs

```
layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs=model.input,
outputs=layer_outputs)
activations = activation_model.predict(img_tensor)
```



Input image



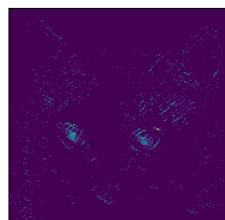
Activation of filter 6

Output of filter 16:

- Cat eye detector?



Input image

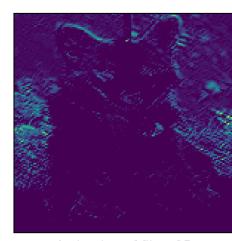


Activation of filter 25

The same filter responds quite differently for other inputs (green detector?).

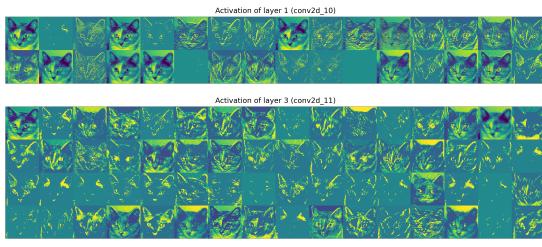


Input image

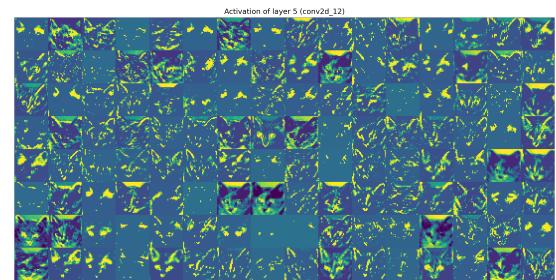


Activation of filter 25

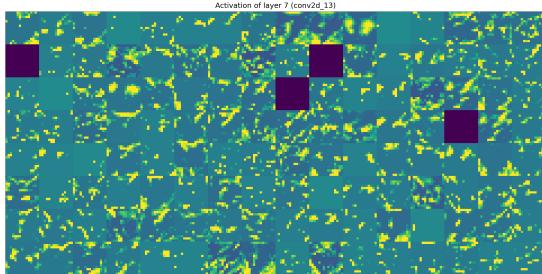
- First 2 convolutional layers: various edge detectors



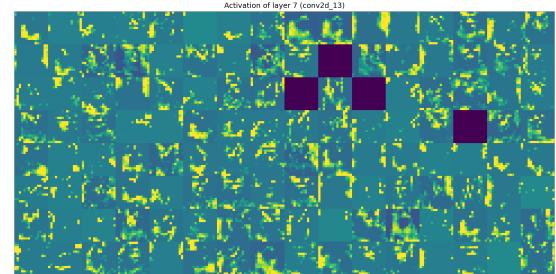
- 3rd convolutional layer: increasingly abstract: ears, eyes



- Last convolutional layer: more abstract patterns
- Empty filter activations: input image does not have the information that the filter was interested in

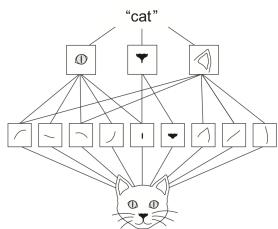


- Same layer, with dog image input
- Very different activations



## Spatial hierarchies

- Deep convnets can learn *spatial hierarchies* of patterns
  - First layer can learn very local patterns (e.g. edges)
  - Second layer can learn specific combinations of patterns
  - Every layer can learn increasingly complex abstractions

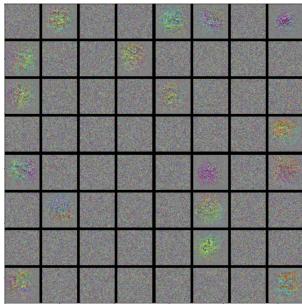


## Visualizing the learned filters

- Visualize filters by finding the input image that they are maximally responsive to
- *gradient ascent in input space*: start from a random image  $x$ , use loss to update the **input values** to values that the filter responds to more strongly (keep weights fixed)
  - $X_{(t+1)} = X_{(t)} + \frac{\partial \text{loss}(X_t)}{\partial X} * \eta$

```
from keras import backend as K
input_img = np.random.random((1, size, size, 3)) * 20 + 128.
loss = K.mean(layer_output[:, :, :, filter_index])
grads = K.gradients(loss, input_img)[0] # Compute gradient
for i in range(40): # Run gradient ascent for 40 steps
    loss_v, grads_v = K.function([input_img], [loss, grads])
    input_img_data += grads_v * step
```

- Learned filters of last convolutional layer
- More focused on center, some vague cat/dog head shapes



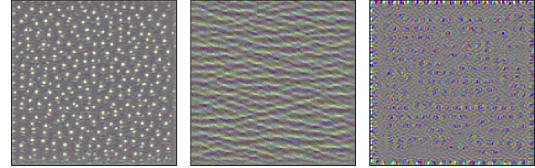
Let's do this again for the `VGG16` network pretrained on [ImageNet](#) (much larger)

```
model = VGG16(weights='imagenet', include_top=False)
```

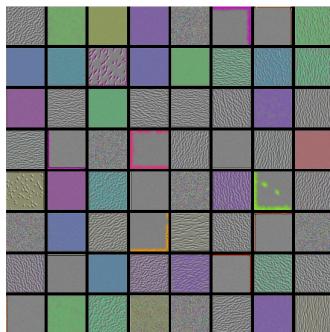
| Layer (type)               | Output Shape            | Param # |
|----------------------------|-------------------------|---------|
| input_1 (InputLayer)       | {(None, None, None, 3)} | 0       |
| block1_conv1 (Conv2D)      | (None, None, None, 64)  | 1792    |
| block1_pool (MaxPooling2D) | (None, None, None, 64)  | 0       |
| block2_conv1 (Conv2D)      | (None, None, None, 128) | 73856   |
| block2_pool (MaxPooling2D) | (None, None, None, 128) | 147584  |
| block3_conv1 (Conv2D)      | (None, None, None, 256) | 0       |
| block3_conv2 (Conv2D)      | (None, None, None, 256) | 295168  |
| block3_pool (MaxPooling2D) | (None, None, None, 256) | 590080  |
| block4_conv1 (Conv2D)      | (None, None, None, 512) | 590080  |
| block4_pool (MaxPooling2D) | (None, None, None, 512) | 0       |
| block5_conv1 (Conv2D)      | (None, None, None, 512) | 1180160 |
| block5_conv2 (Conv2D)      | (None, None, None, 512) | 2359808 |
| block5_conv3 (Conv2D)      | (None, None, None, 512) | 2359808 |
| block5_pool (MaxPooling2D) | (None, None, None, 512) | 0       |

```
block5_pool (MaxPooling2D) (None, None, None, 512) 0
=====
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

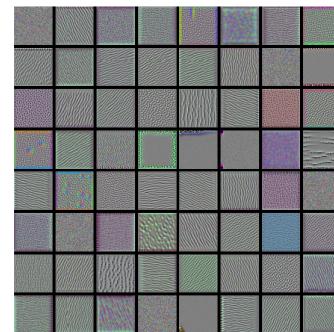
- Visualize convolution filters 0-2 from layer 5 of the VGG network trained on ImageNet
- Some respond to dots or waves in the image



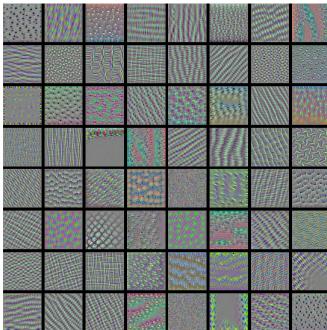
First 64 filters for 1st convolutional layer in block 1: simple edges and colors



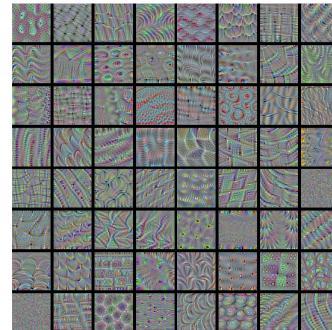
Filters in 2nd block of convolution layers: simple textures (combined edges and colors)



Filters in 3rd block of convolution layers: more natural textures

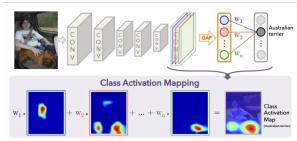


Filters in 4th block of convolution layers: feathers, eyes, leaves,...



#### Visualizing class activation

- We can also visualize which part of the input image had the greatest influence on the final classification. Helps to interpret what the model is paying attention to.
- Class activation maps* : produces a heatmap over the input image
  - Choose a convolution layer, do Global Average Pooling (GAP) to get one output per filter
  - Get the weights between those outputs and the class of interest
  - Compute the weighted sum of all filter activations: combines what each filter is responding to and how much this affects the class prediction



#### Example on VGG with a specific input image

- Take the last convolutional layer of VGG pretrained on ImageNet

```
model = VGG16(weights='imagenet')
last_conv_layer = model.get_layer('block5_conv3')
```

Last conv layer shape: (None, 14, 14, 512)

- Choose an input image and preprocess it so we can feed it to the model

```
img = image.load_img(img_path, target_size=(224, 224))
```

- Find the output node for its class ('african elephant', class 386)

```
african_elephant_output = model.output[:, 386]
```

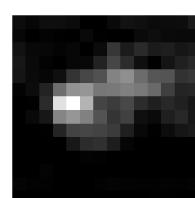


- VGG doesn't use GAP. Compute the average gradient from the output node to the conv layer

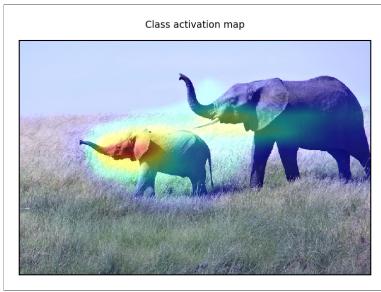
- Multiply (channel-wise) with the activations of the conv layer

```
grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]
pooled_grads = K.mean(grads, axis=(0, 1, 2))
for i in range(512):
    pooled_grads_value[i] = pooled_grads_value[i] * conv_layer.output[:, :, i]
    heatmap = np.mean(conv_layer.output[:, :, i], axis=-1)
```

- Visualize heatmap. It's 14x14 since that's the output dimension of the conv layer

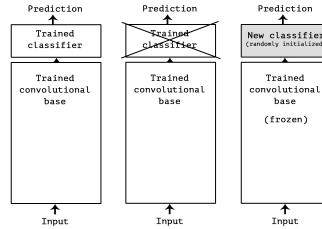


- Upscaled and superimposed on the original image
- The model looked at the face of the baby elephant and the trunk of the large elephant



## Using pretrained networks

- We can re-use pretrained networks instead of training from scratch
- Learned features can be a generic model of the visual world
- Use convolutional base to extract features, then train any classifier on new data
- Also called *transfer learning*, which is a kind of *meta-learning*



- Let's instantiate the VGG16 model (without the dense layers)
- Final feature map has shape (4, 4, 512)

```
conv_base = VGG16(weights='imagenet', include_top=False,
                   input_shape=(150, 150, 3))
```

```
block5_conv2 (Conv2D)      (None, 9, 9, 512)      2359808
block5_conv3 (Conv2D)      (None, 9, 9, 512)      2359808
block5_pool (MaxPooling2D) (None, 4, 4, 512)       0
=====
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

```
Model: "vgg16"
Layer (type)          Output Shape         Param #
=====
input_2 (InputLayer)  [(None, 150, 150, 3)]  0
block1_conv1 (Conv2D)  (None, 150, 150, 64)   1792
block1_conv2 (Conv2D)  (None, 150, 150, 64)   36928
block1_pool (MaxPooling2D) (None, 75, 75, 64)   0
block2_conv1 (Conv2D)  (None, 75, 75, 128)   73856
block2_conv2 (Conv2D)  (None, 75, 75, 128)   147584
block2_pool (MaxPooling2D) (None, 37, 37, 128) 0
block3_conv1 (Conv2D)  (None, 37, 37, 256)   295168
block3_conv2 (Conv2D)  (None, 37, 37, 256)   590080
block3_conv3 (Conv2D)  (None, 37, 37, 256)   590080
block3_pool (MaxPooling2D) (None, 18, 18, 256) 0
block4_conv1 (Conv2D)  (None, 18, 18, 512)   1180160
block4_conv2 (Conv2D)  (None, 18, 18, 512)   2359808
block4_conv3 (Conv2D)  (None, 18, 18, 512)   2359808
block4_pool (MaxPooling2D) (None, 9, 9, 512)   0
block5_conv1 (Conv2D)  (None, 9, 9, 512)   2359808
```

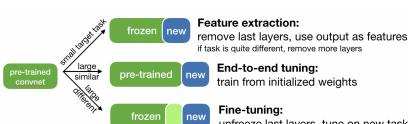
## Using pre-trained networks: 3 ways

- Fast feature extraction (similar task, little data)
  - Call `predict` from the convolutional base to build new features
  - Use outputs as input to a new neural net (or other algorithm)
- End-to-end tuning (similar task, lots of data + data augmentation)
  - Extend the convolutional base model with a new dense layer
  - Train it end to end on the new data (expensive!)
- Fine-tuning (somewhat different task)
  - Unfreeze a few of the top convolutional layers, and retrain
    - Update only the more abstract representations

### Fast feature extraction (without data augmentation)

- Run every batch through the pre-trained convolutional base

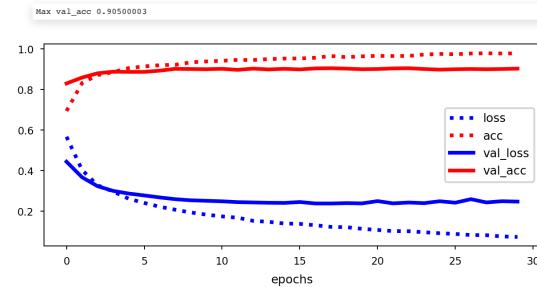
```
generator = datagen.flow_from_directory(dir, target_size=(150, 150),
   batch_size=batch_size, class_mode='binary')
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
```



- Build Dense neural net (with Dropout)
- Train and evaluate with the transformed examples

```
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 *
512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

- Validation accuracy around 90%, much better!
- Still overfitting, despite the Dropout: not enough training data

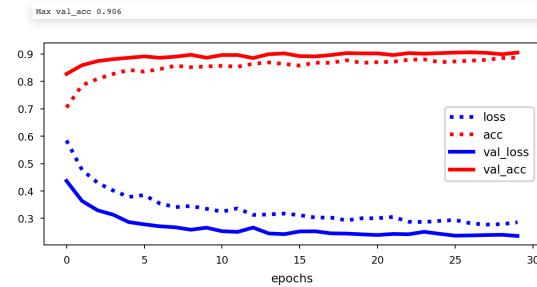


### Fast feature extraction (with data augmentation)

- Simply add the Dense layers to the convolutional base
- Freeze the convolutional base (before you compile)
  - Without freezing, you train it end-to-end (expensive)

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
conv_base.trainable = False
```

We now get about 90% accuracy again, and very little overfitting

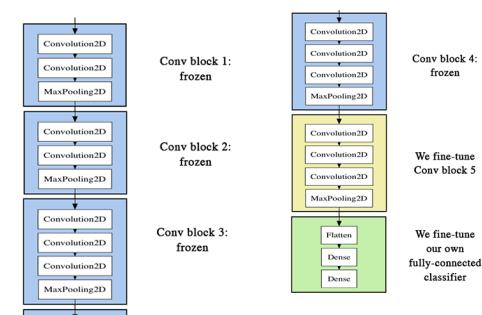


### Fine-tuning

- Add your custom network on top of an already trained base network.
- Freeze the base network, but unfreeze the last block of conv layers.

```
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        layer.trainable = True
    else:
        layer.trainable = False
```

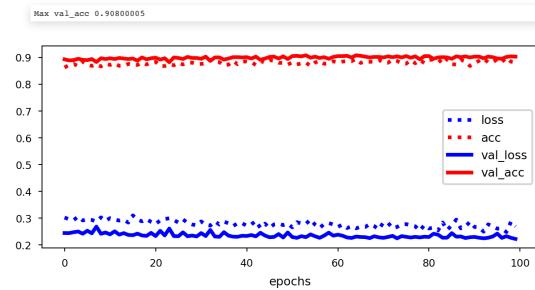
Visualized



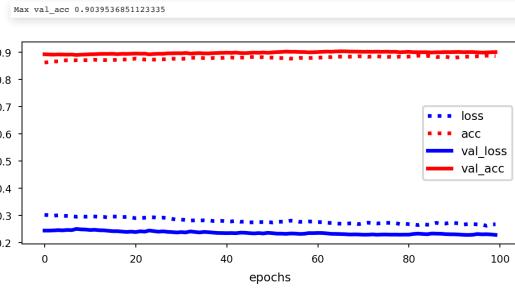
- Load trained network, finetune
  - Use a small learning rate, large number of epochs
  - You don't want to unlearn too much: *catastrophic forgetting*

```
model = load_model(os.path.join(model_dir,
'cats_and_dogs_small_3b.h5'))
model.compile(loss='binary_crossentropy',
optimizer=optimizers.RMSprop(lr=1e-5),
metrics=['acc'])
history = model.fit(
  train_generator, steps_per_epoch=100, epochs=100,
  validation_data=validation_generator,
  validation_steps=50)
```

Almost 95% accuracy. The curves are quite noisy, though.



- We can smooth the learning curves using a running average



## Take-aways

- Convnets are ideal for attacking visual-classification problems.
- They learn a hierarchy of modular patterns and concepts to represent the visual world.
- Representations are easy to inspect
- Data augmentation helps fight overfitting
- You can use a pretrained convnet to build better models via transfer learning

## Lecture 10. Neural Networks for text

## Turning text into numbers

- ## Turning text into nu

## Overview

- Word embeddings
    - Word2Vec, FastText, GloVe
    - Neural networks on word embeddings
  - Sequence-to-sequence models
    - Self-attention
    - Transformer models

## Bag of word representation

- First, build a **vocabulary** of all occurring words. Maps every word to an index.
  - Represent each document as an  $N$  dimensional vector (top- $N$  most frequent words)
    - One-hot (sparse) encoding: 1 if the word occurs in the document
  - Destroys the order of the words in the text (hence, a 'bad' of words)

Example: IMBD review database

- 50,000 reviews, labeled positive (1) or negative (0)
    - Every row (document) is one review, no other input features
    - Already tokenized. All markup punctuation removed.

Review 0: the this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert redford's is a n amazing actor and now the same being director norman's father came from the same scottish isl and as myself as i loved...

Review 5: the begins better than it ends funny that the russian submarine crew outperforms all other actors it's like those scenes where documentary shots br br spoiler part the message dechiffrer was contrary to the whole story it just does not mesh br br

Review 10: the french horror cinema has seen something of a revival over the last couple of years with great films such as *Inside* and *Switchblade Romance* bursting on to the scene maistique p receding the revival just slightly but stands head and shoulders over most modern horror titles and is surely one of the best.

```
1.  
1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.  
1. 0. 1. 1. 1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 0. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 0.  
1. 0. 0. 0. 0. 1. 0. 1.]
```

### Word counts

- Count the number of times each word appears in the document
  - Example using `sklearn CountVectorizer` (on 2 documents)
  - In practice, we also:
    - Preprocess the text (tokenization, stemming, remove stopwords, ...)
    - Use *n*-grams ("not terrible", "terrible acting", ...), character *n*-grams ('ter', 'err', 'eri', ...)
    - Scale the word-counts (e.g. L2 normalization or TF-IDF)

```

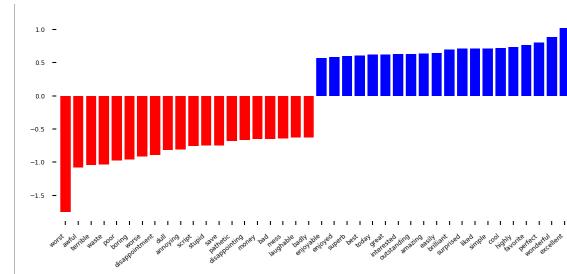
Vocabularies (feature names) after fit: ['actor' 'amazing' 'an' 'and' 'are' 'as' 'bad' 'be' 'been'
g' 'beet' 'big'
'boobs' 'brilliant' 'but' 'came' 'casting' 'cheesy' 'could' 'describe'
'direction' 'director' 'ever' 'everyone' 'father' 'film' 'from' 'giant'
'got' 'had' 'hair' 'horror' 'hundreds' 'imagine' 'is' 'island' 'just'
'location' 'love' 'loved' 'made' 'movie' 'movies' 'music' 'myself'
'norman' 'now' 'of' 'on' 'paper' 'part' 'pin' 'played' 'plot' 'really'
'refined' 'ridiculous' 'robert' 'safety' 'same' 'scenery' 'scottish'
'seem' 'seemed' 'suited' 'terrible' 'the' 'then' 'these' 'they'
'thin' 'this' 'to' 've' 'was' 'words' 'worst' 'you']

Count encoding doc 1: [1 1 2 2 0 1 0 2 0 0 0 0 1 0 1 0 1 0 1 1 1 0 0 0 0 0 0 0 1 1 2
1
0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 0 2 1 0 1 1 0 1 4 0 1 0 1 0 0
1 0 0 1]
Count encoding doc 2: [0 0 0 3 1 0 1 0 1 2 1 0 1 0 0 1 0 1 0 0 1 0 0 0 0 1 1 1 1 1 0 1 0 0
1
0 1 1 1 0 0 0 1 1 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 4 0 1 0 1 2 2 1
0 1 1 0 1]
```

## Classification

- With this tabular representation, we can fit any model (e.g. Logistic regression)
- Visualize coefficients: which words are indicative for positive/negative reviews?

Logistic regression accuracy: 0.8538



## Neural networks on bag of words

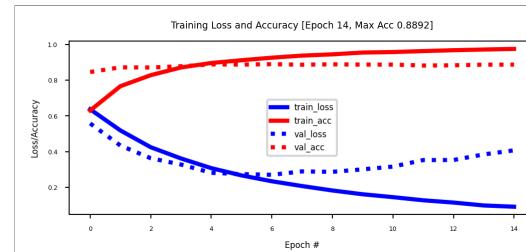
- We can also build neural networks on bag-of-word vectors
  - E.g. Use one-hot-encoding with 10000 most frequent words
- Simple model with 2 dense layers, ReLU activation, dropout

Binary classification: single output node, sigmoid activation, binary cross-entropy loss

```
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

## Evaluation

- Take a validation set of 10,000 samples from the training set
- The validation loss peaks after a few epochs, after which the model starts to overfit
  - Performance is better than Logistic regression



## PREDICTIONS

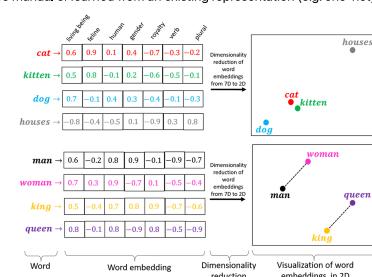
Let's look at a few predictions:

```
782/782 [=====] - 2s 2ms/step
Review 1: ? please give this one a miss like ? ? and the rest of the cast rendered terrible performances the show is flat flat flat br br i don't know how michael madison could have allowed this one on his plate he almost seemed to know this wasn't going to work out and his performance was quite ? so all you madison fans give this a miss
Predicted positiveness: [0.046]

Review 16: ? from 1995 first i watched this movie i feel never reach the end of my satisfaction i feel that i want to watch more and more until now my god i don't believe it was ten years ago and i can believe that i almost remember every word of the dialogues i love this movie and i love this novel absolutely perfection i love willem ? he has a strange voice to spell the words black night and i always say like for many times never being bored i love the music of it's so much made me come in another world deep in my heart anyone can feel what i feel and anyone could make the movie like this i don't believe so thanks thanks
Predicted positiveness: [0.956]
```

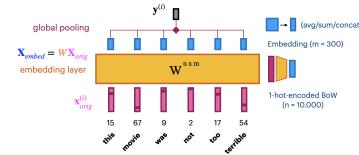
## Word Embeddings

- A word embedding is a numeric vector representation of a word
  - Can be manual or learned from an existing representation (e.g. one-hot)



### Learning embeddings from scratch

- Input layer uses fixed length documents (with 0-padding). 2D tensor (samples, max\_length)
- Add an *embedding layer* to learn the embedding
  - Create  $n$ -dimensional one-hot encoding. Yields a 3D tensor (samples, max\_length,  $n$ )
  - To learn an  $m$ -dimensional embedding, use  $m$  hidden nodes. Weight matrix  $W^{n \times m}$
  - Linear activation function:  $X_{\text{embed}} = W X_{\text{orig}}$ , 3D tensor (samples, max\_length,  $m$ )
- Combine all word embeddings into a document embedding (e.g. global pooling).
- Add (optional) layers to map word embeddings to the output. Learn embedding weights from data.

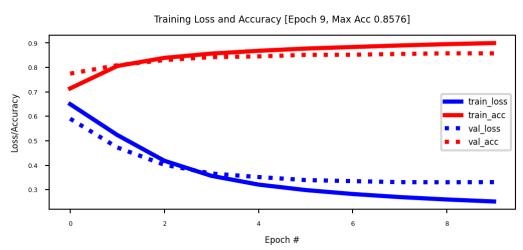


Let's try this:

```
max_length = 100 # pad documents to a maximum number of words
vocab_size = 10000 # vocabulary size
embedding_length = 20 # embedding length (more would be better)

model = models.Sequential()
model.add(layers.Embedding(vocab_size, embedding_length, input_length=max_length))
model.add(layers.GlobalAveragePooling1D())
model.add(layers.Dense(1, activation='sigmoid'))
```

- Training on the IMDB dataset: slightly worse than using bag-of-words?
  - Embedding of dim 20 is very small, should be closer to 100 (or 300)
  - We don't have enough data to learn a really good embedding from scratch

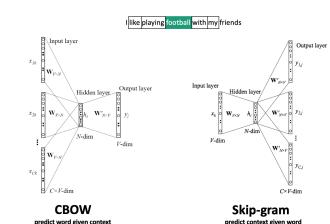


### Pre-trained embeddings

- With more data we can build better embeddings, but we also need more labels
- Solution: learn embedding on auxiliary task that doesn't require labels
  - E.g. given a word, predict the surrounding words.
  - Also called self-supervised learning. Supervision is provided by data itself
- Freeze embedding weights to produce simple word embeddings, or finetune to a new tasks
- Most common approaches:
  - Word2Vec: Learn neural embedding for a word based on surrounding words
  - FastText: learns embedding for character n-grams
    - Can also produce embeddings for new, unseen words
  - GloVe (Global Vector): Count co-occurrences of words in a matrix
    - Use a low-rank approximation to get a latent vector representation

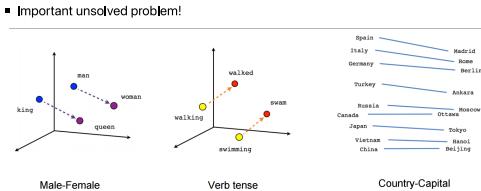
### WORD2VEC

- Move a window over text to get  $C$  context words ( $V$ -dim one-hot encoded)
- Add embedding layer with  $N$  linear nodes, global average pooling, and softmax layer(s)
- CBOW: predict word given context, use weights of last layer  $W_{N \times V}$  as embedding
- Skip-Gram: predict context given word, use weights of first layer  $W_{V \times N}^T$  as embedding
  - Scales to larger text corpora, learns relationships between words better



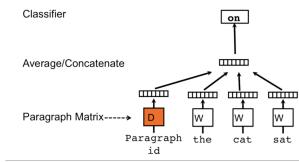
### WORD2VEC PROPERTIES

- Word2Vec happens to learn **interesting relationships** between words
  - Simple vector arithmetic can map words to plurals, conjugations, gender analogies,...
  - e.g. Gender relationships:  $\vec{veCling} - \vec{vecMan} + \vec{vecWoman} \approx \vec{veQueen}$
  - PCA applied to embeddings shows Country - Capital relationship
- Careful: embeddings can capture **gender and other biases** present in the data.
  - Important unsolved problem!



## Doc2Vec

- Alternative way to combine word embeddings (instead of global pooling)
- Adds a paragraph (or document) embedding: learns how paragraphs (or docs) relate to each other
  - Captures document-level semantics: context and meaning of entire document
- Can be used to determine semantic similarity between documents.

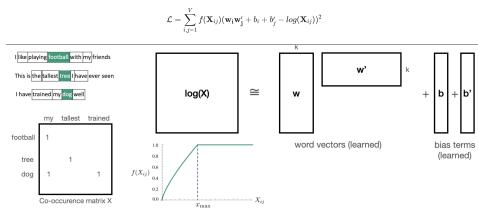


## FastText

- Limitations of Word2Vec:**
  - Cannot represent new (out-of-vocabulary) words
  - Similar words are learned independently: less efficient (no parameter sharing)
    - E.g. 'meet' and 'meeting'
- FastText:** same model, but uses *character n-grams*
  - Words are represented by all character n-grams of length 3 to 6
    - "football" 3-grams: <fo, foo, oot, tba, ba, ll>
  - Because there are so many n-grams, they are hashed (dimensionality = bin size)
  - Representation of word "football" is sum of its n-gram embeddings
- Negative sampling:** also trains on random negative examples (out-of-context words)
  - Weights are updated so that they are *less likely* to be predicted

## GLOBAL VECTOR MODEL (GloVe)

- Builds a co-occurrence matrix  $X$ : counts how often 2 words occur in the same context
- Learn a  $k$ -dimensional embedding  $W$  through matrix factorization with rank  $k$ 
  - Actually learns 2 embeddings  $W$  and  $W'$  (differ in random initialization)
- Minimizes loss  $\mathcal{L}$ , where  $b_i$  and  $b'_i$  are bias terms and  $f$  is a weighting function



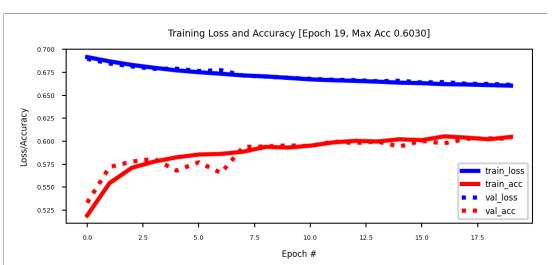
## Let's try this

- Download the [GloVe embeddings trained on Wikipedia](#)
- We can now get embeddings for 400,000 English words
- E.g. 'queen' (in 100-dim):

```
array([-0.5 , -0.708,  0.554,  0.673,  0.225,  0.603, -0.262,  0.739,
       -0.654, -0.216, -0.338,  0.245,  0.857, -0.372, -0.588,
       0.306, -0.307, -0.219,  0.784, -0.619, -0.549,  0.431, -0.027,
       0.976,  0.462,  0.115, -0.998, -0.066, -0.208,  0.532,  0.409,
      1.001,  0.161,  0.115, -0.015, -0.049, -0.049,  0.049, -0.049,
      -0.146, -0.201, -0.171, -0.558,  0.719,  0.07, -0.236, -0.495,
      1.159, -0.051,  0.257, -0.091,  1.266,  1.105, -0.516, -2.003,
      -0.648,  0.164,  0.329,  0.048,  0.19,  0.661,  0.081,  0.336,
      0.228,  0.146, -0.51,  0.638,  0.473, -0.328,  0.084, -0.785,
      0.001, -0.039,  0.497,  0.117, -0.779, -0.011, -0.16, -0.353,
      -0.049, -0.325,  1.498, -0.589, -1.132, -0.607, -0.375, -1.181,
      0.801, -0.5 , -0.166, -0.706,  0.4 ,  0.511, -0.803, -0.696,
      -0.637, -0.36 ,  0.133, -0.561], dtype=float32)
```

## Same simple model, but with frozen GloVe embeddings: much worse!

```
embedding_layer = layers.Embedding(input_dim=10000, output_dim=100,
                                    input_length=max_length, trainable=False)
embedding_layer.set_weights([weights]) # set pre-trained weights
model = models.Sequential([
    embedding_layer, layers.GlobalAveragePooling1D(),
    layers.Dense(1, activation='sigmoid')])
```

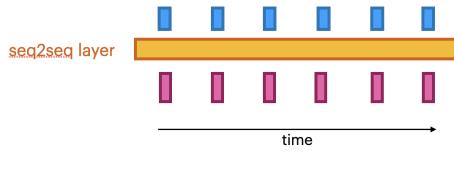


## Sequence-to-sequence (seq2seq) models

- Global average pooling or flattening destroys the word order
- We need to model sequences explicitly, e.g.:
  - 1D convolutional models: run a 1D filter over the input data
    - Fast, but can only look at small part of the sentence
  - Recurrent neural networks (RNNs)
    - Can look back at the entire previous sequence
    - Much slower to train, have limited memory in practice
  - Attention-based networks (Transformers)
    - Best of both worlds: fast and very long memory

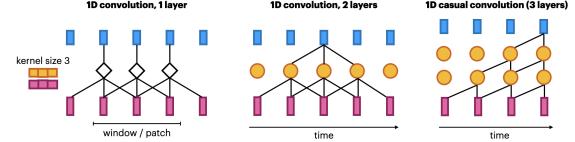
## seq2seq models

- Produce a series of output given a series of inputs over time
- Can handle sequences of different lengths
  - Label-to-sequence, Sequence-to-label, seq2seq,...
  - Autoregressive models (e.g. predict the next character, unsupervised)



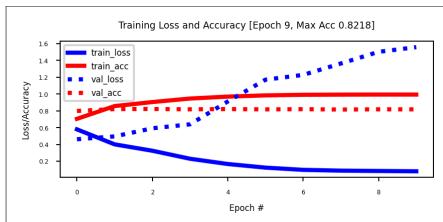
## 1D convolutional networks

- Similar to 2D convnets, but moves only in 1 direction (time)
  - Extract local 1D patch, apply filter (kernel) to every patch
  - Pattern learned can later be recognized elsewhere (translation invariance)
- Limited memory: only sees a small part of the sequence (receptive field)
  - You can use multiple layers, dilations,... but becomes expensive
- Looks at 'future' parts of the series, but can be made to look only at the past
  - Known as 'causal' models (not related to causality)



- Same embedding, but add 2 Conv1D layers and MaxPooling1D. Better!

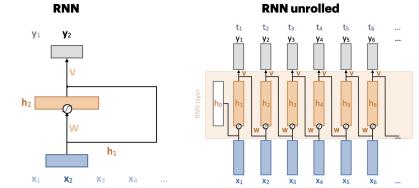
```
model = models.Sequential([
    embedding_layer,
    layers.Conv1D(32, 7, activation='relu'),
    layers.MaxPooling1D(5),
    layers.Conv1D(32, 7, activation='relu'),
    layers.GlobalAveragePooling1D(),
    layers.Dense(1, activation='sigmoid')])
```



```
157/157 [=====] - 3s 18ms/step - loss: 0.0807 - accuracy: 0.9932 - val
loss: 1.5570 - val_accuracy: 0.8172
```

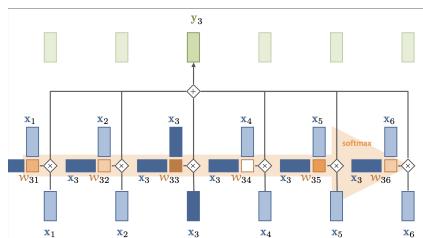
## Recurrent neural networks (RNNs)

- Adds a recurrent connection that concats previous output to next input
  - Hidden layer
 
$$h_t = \sigma(W[h_{t-1}] + b)$$
- Unbounded memory, but training requires *backpropagation through time*
  - Requires storing previous network states (slow + lots of memory)
  - Vanishing gradients strongly limit practical memory
- Improved with *gating*: learn what to input, forget, output (LSTMs, GRUs,...)



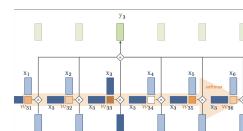
## Simple self-attention

- Compute dot product of input vector  $x_i$  with every  $x_j$  (including itself):  $w_{ij}$
- Compute softmax over all these weights (positive, sum to 1)
- Multiply by each input vector, and sum everything up
- Can be easily vectorized:  $Y^T = W X^T$ ,  $W = \text{softmax}(X^T X)$



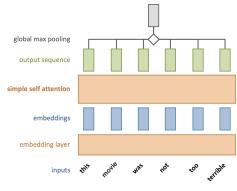
## Simple self-attention (2)

- Output is mostly influenced by the current input ( $w_{ii}$  is largest)
  - Mixes in information from other inputs according to how *similar* they are
- Doesn't learn (no parameters), the embedding of  $x$  defines self-attention
- $Y^T = W X^T$  is *linear*, vanishing gradients only through softmax
- Has no problem looking very far back in the sequence
- Operates on sets (permutation invariant): allows img-to-set, set-to-set,... tasks
  - No access to sequence. For seq tasks we encode sequence in embedding



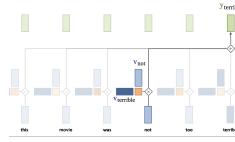
### Simple self-attention layer

- Let's add a simple self-attention layer to our movie sentiment model
- Without self-attention, every word would contribute independently from others
  - Exactly as in a bag-of-words model
  - The word **terrible** will likely result in a negative prediction



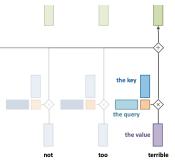
### Simple self-attention layer

- Self-attention can learn that the meaning on the word **terrible** is inverted by the presence of the word **not**, even if it is further away in the sequence.
- In general, each self attention layer can learn specific relationships between words (e.g. inversion). We'll need many of them.



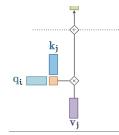
### Standard self-attention

- Inputs occur in one of 3 positions in the self-attention layer:
  - Value  $v$ : input vector that provides the output, weighted by:
  - Query  $q$ : the vector that corresponds to the wanted *output*
  - Key  $k$ : The vector that the query is matched against
- Works as a soft version of a dictionary, in which:
  - Every key matches the query to some extent (w.r.t. its dot product)
  - A weighted mixture of all values (normalized by softmax)



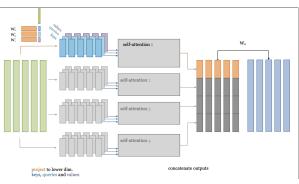
### Standard self-attention

- We want to *learn* how each of these interact by adding learned transformations
  - $k_i = Kx_i + b_k$
  - $q_i = Qx_i + b_q$
  - $v_i = Vx_i + b_v$
- Makes self-attention more flexible, learnable
- Learn what to pay attention to in the input (e.g. sequence, image,...)



### Standard self-attention (2)

- Inputs can have multiple relationships with each other (e.g. negation, strengthening,...)
- To learn these in parallel, we can split the self-attention in multiple heads
- Input vector is embedded (linearly) into lower dimensionality, multiple times



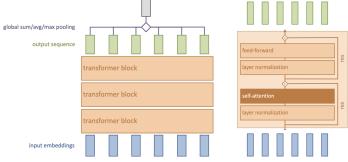
### Standard self-attention (3)

- The softmax operation can still lead to vanishing gradients (unless values are small)
  - We can scale the dot product by the input dimension  $k$ :  $u'_{ij} = \frac{u_i^T v_j}{\sqrt{k}}$

$$u'_{ij} = \frac{u_i^T v_j}{\sqrt{k}}$$

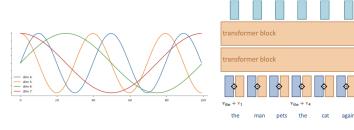
## Transformer model

- Repeat self-attention multiple times in controlled fashion
- Works for sequences, images, graphs,... (learn how sets of objects interact)
- Models consist of multiple transformer blocks, usually:
  - Layer normalization (every input is normalized independently)
  - Self-attention layer (learn interactions)
  - Residual connections (preserve gradients in deep networks)
  - Feed-forward layer (learn mappings)



## Positional encoding

- We need some way to tell the self-attention layer about position in sequence!
- Represent position by vectors, using some easy-to-learn predictable pattern
  - Add these encodings to vector embeddings
  - Give information on how far one input is from the others



## Summary

- Bag of words representations
  - Useful, but limited, since they destroy the order of the words in text
- Word embeddings
  - Learning word embeddings from labeled data is hard, you may need a lot of data
  - Pretrained word embeddings
    - Word2Vec: learns good embeddings and interesting relationships
    - FastText: can also compute embeddings for entirely new words
    - GloVe: also takes the global context of words into account
- Sequence-to-sequence models
  - 1D convolutional nets (fast, limited memory)
  - RNNs (slow, also quite limited memory)
  - Self-attention (allows very large memory)
- Transformers

## Acknowledgement

Several figures came from the excellent [VU Deep Learning course](#)