Lecture 5 - Ensemble methods, neural nets and deep learning

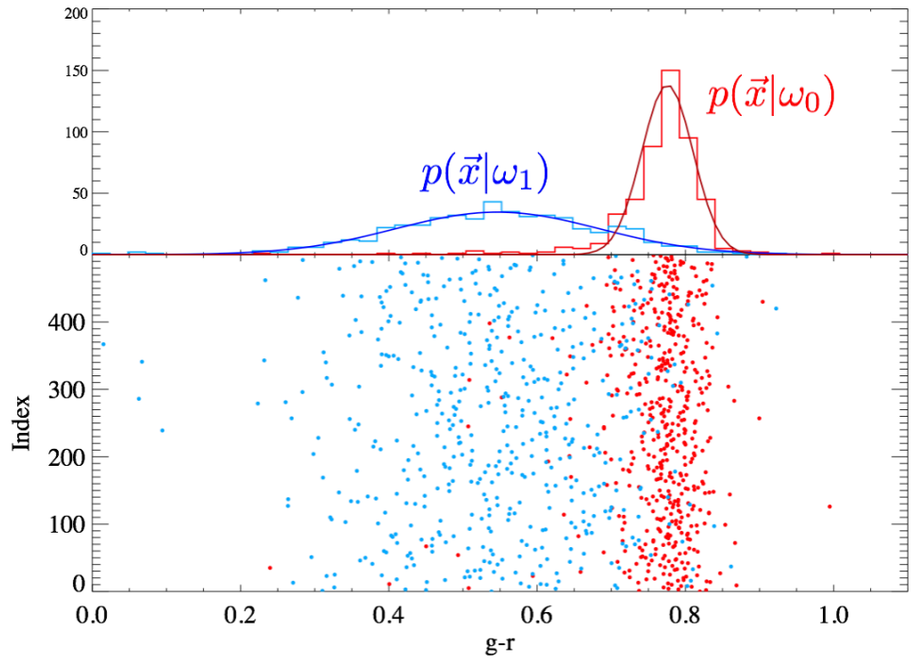
See https://github.com/jbrinchmann/MLD2023 as usual

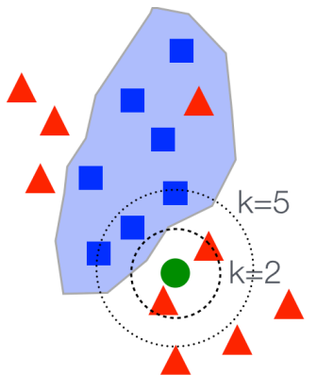
Lectures/Lecture 5 has the PDF for today

Last lecture

Bayesian classification (naïve Bayes):

$$p(\omega_0|\vec{x}) > p(\omega_1|\vec{x}) \Rightarrow \text{Class 0 (elliptical)}$$
$$p(\omega_0|\vec{x}) < p(\omega_1|\vec{x}) \Rightarrow \text{Class 1 (spiral)}$$





k-nearest neighbours (classification or regression)

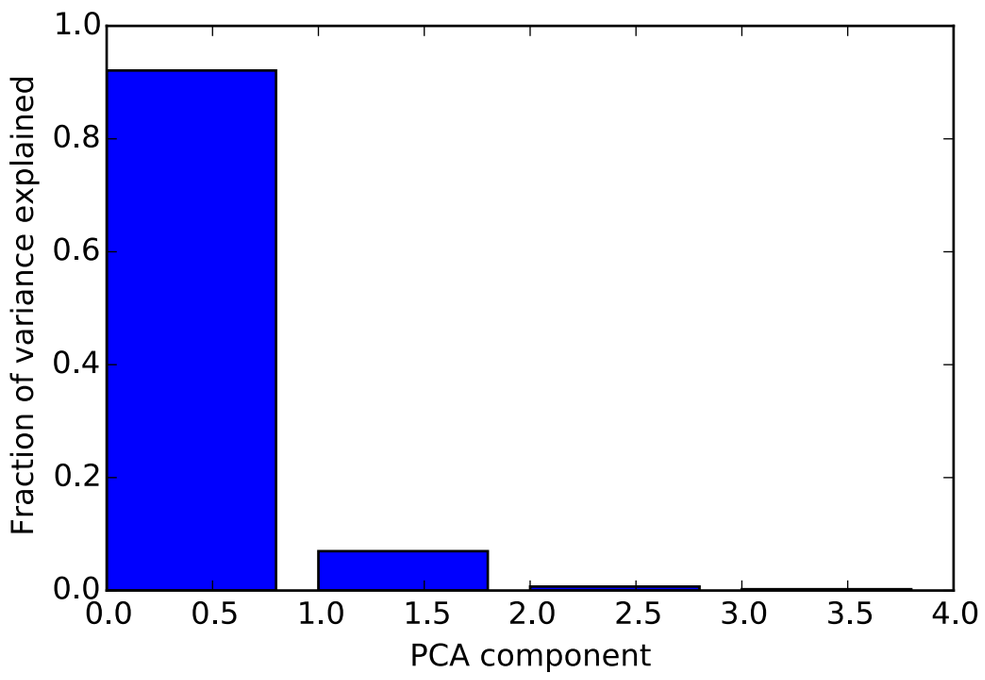
Standardizing data:

$$x_i \rightarrow \frac{x_i - \bar{x}}{\sigma_x}$$

Last lecture

Principal Component Analysis

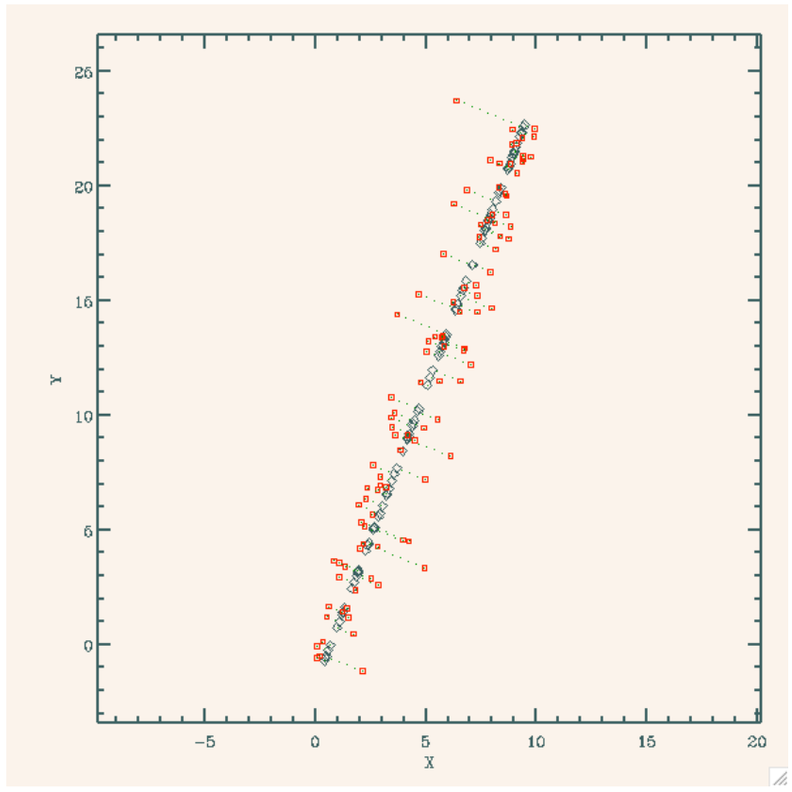
Finds directions where data vary the most, can be very useful for feature selection by looking at which components explain most of the variance

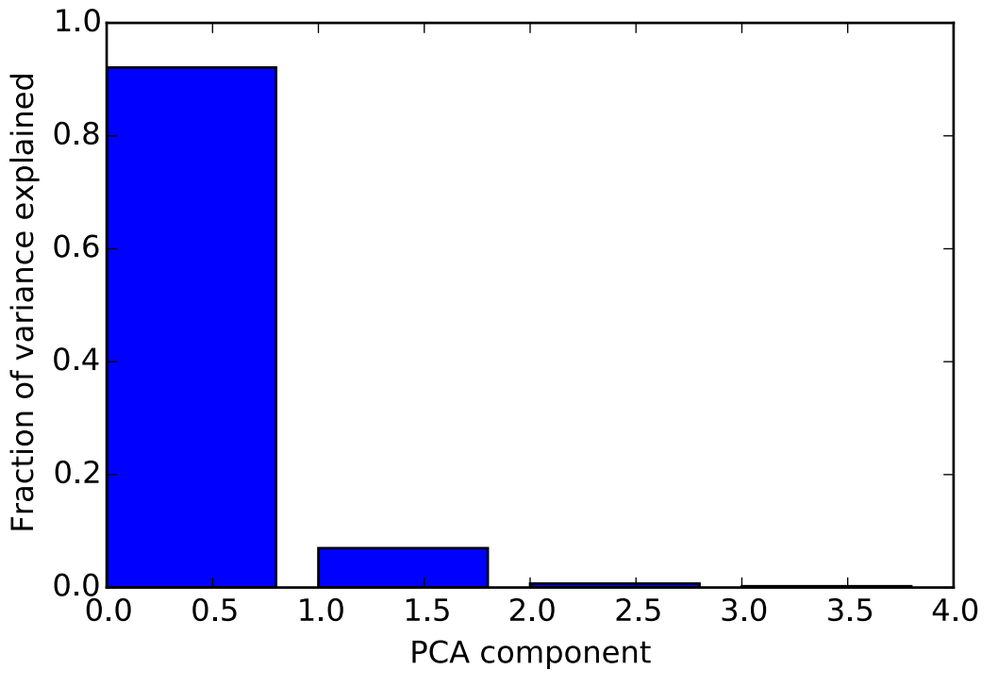


Last lecture

Principal Component Analysis

Finds directions where data vary the most, can be very useful for feature selection by looking at which components explain most of the variance





Manifold learning

Linear vs non-linear models

PCA is a linear method - it can find combinations of data vectors that make the resulting principal components linearly independent. Does that mean that PCs have to be independent?

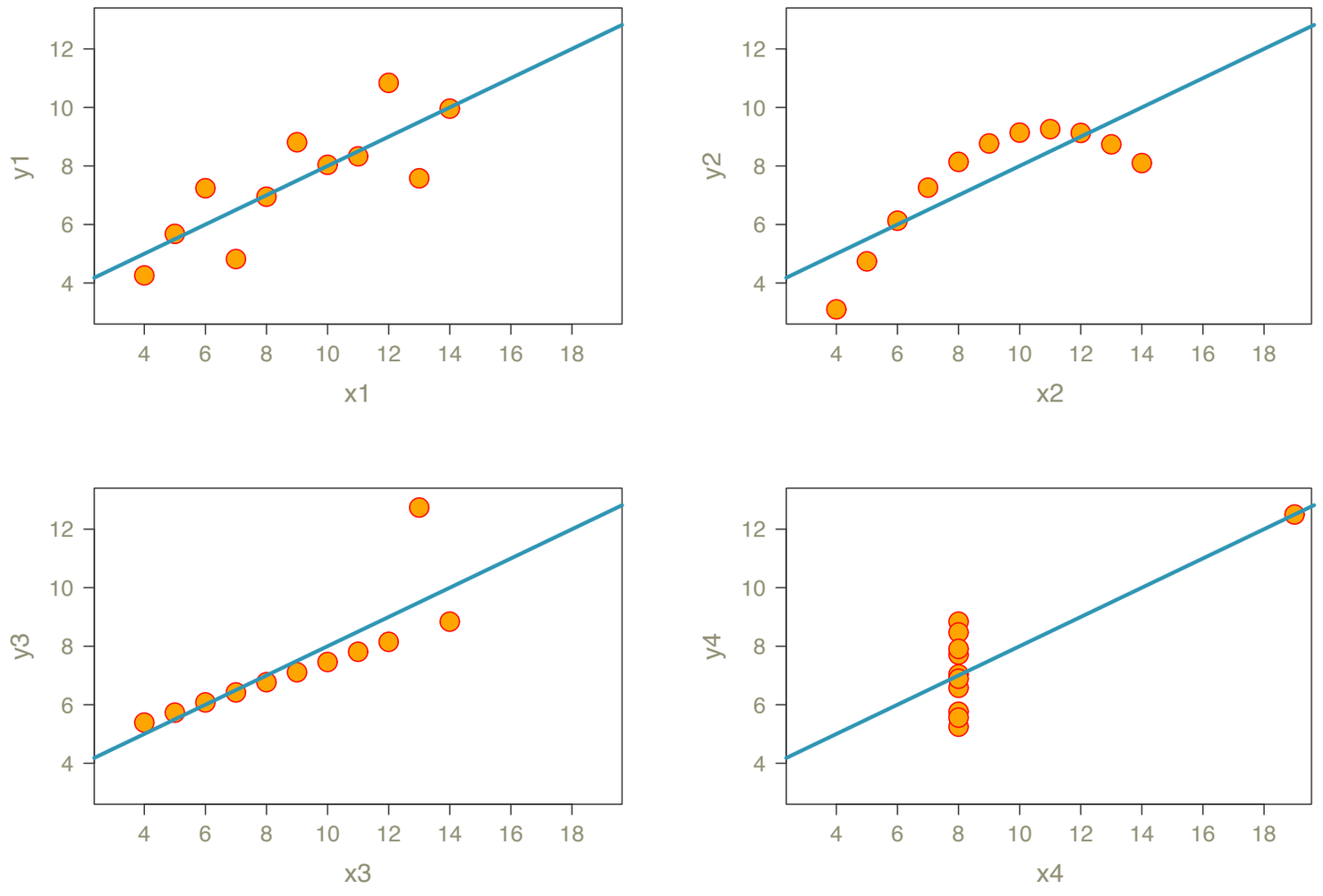
No! (despite what people often write in their papers)

Linear vs non-linear models

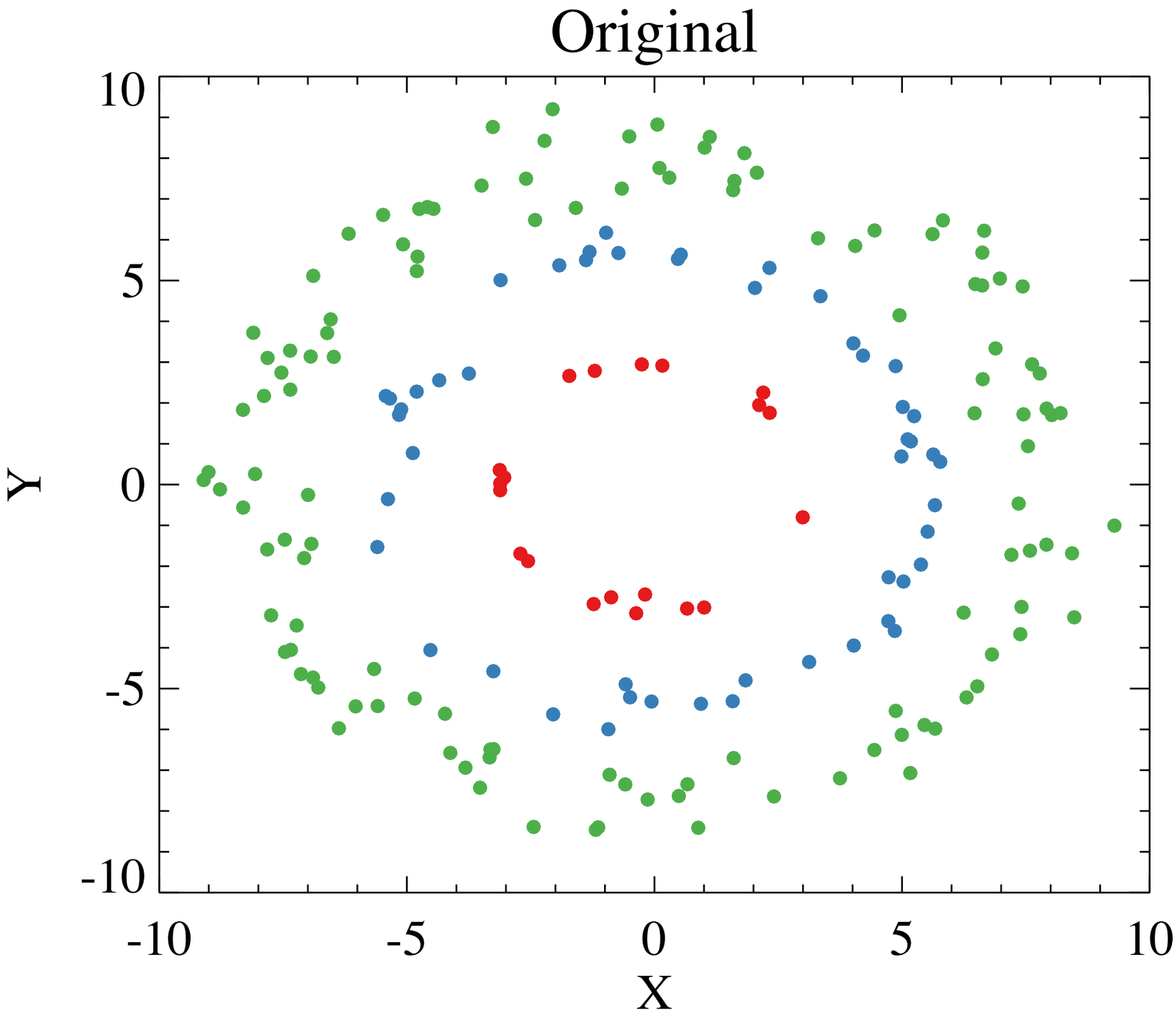
PCA is a linear method - it can find combinations of data vectors that make the resulting principal components linearly independent. Does that mean that PCs have to be independent?

No! (despite what people often write in their papers)

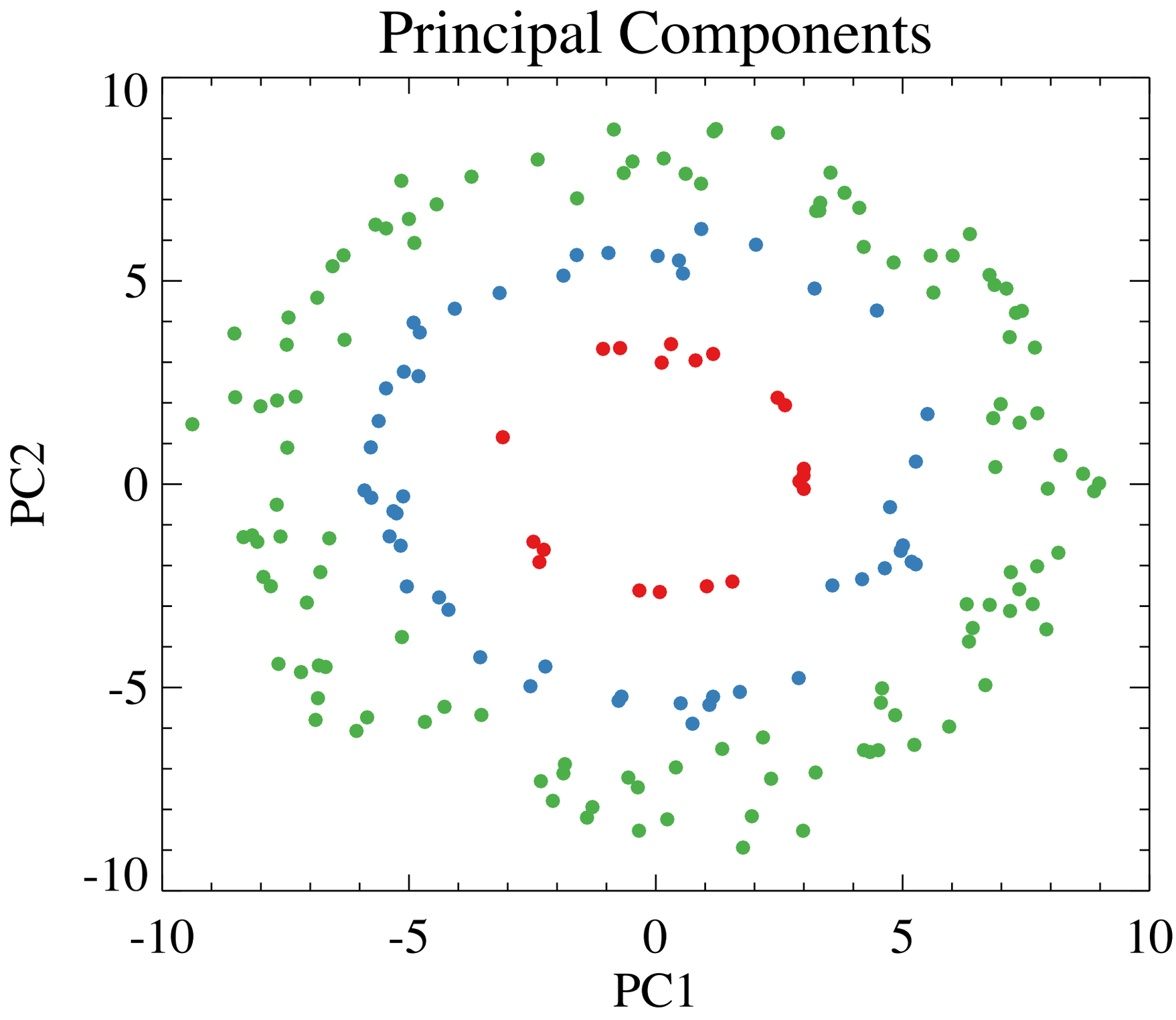
The **correlation coefficient** measures how closely a dataset can be fit by a **straight line** and the *correlation matrix* generalises this to higher dimensions.



Linear vs non-linear



Linear vs non-linear

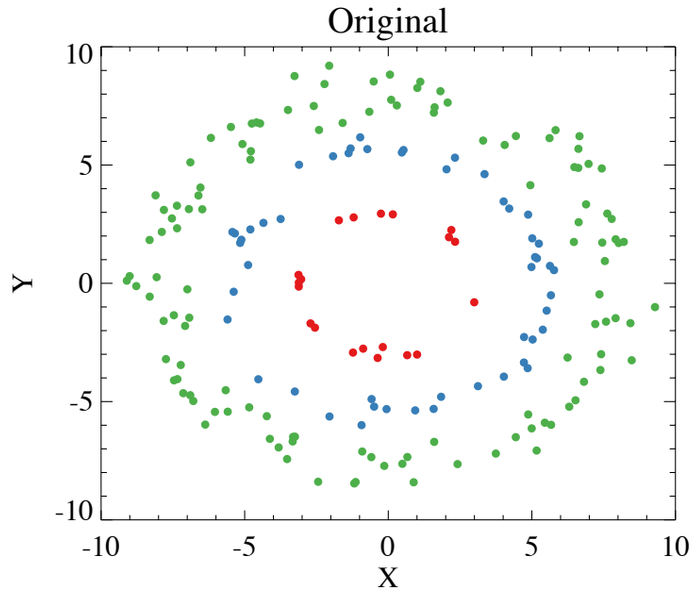


Linear vs non-linear

In this case we know what we should do: We should convert to polar coordinates:

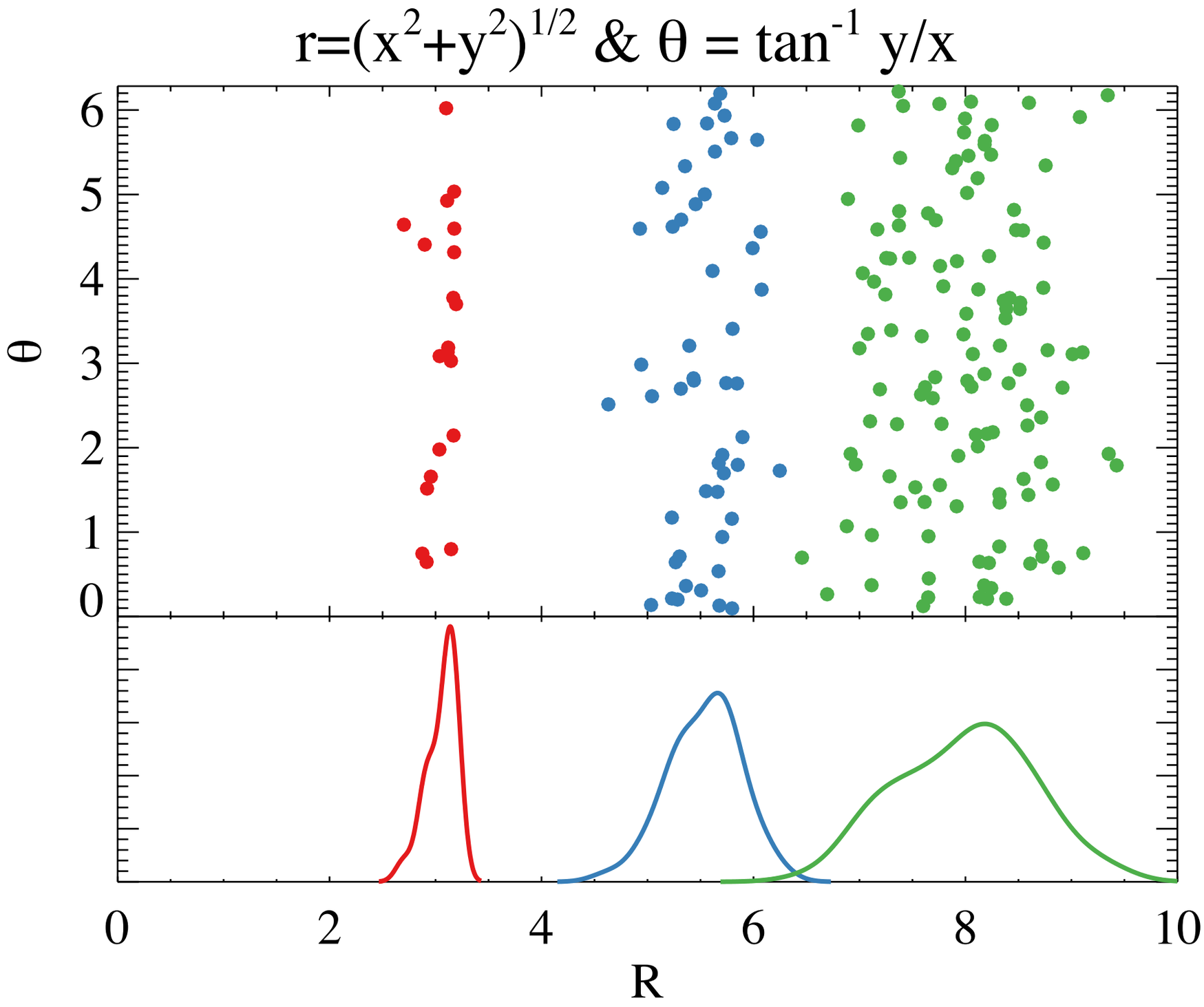
$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1} \frac{y}{x}$$



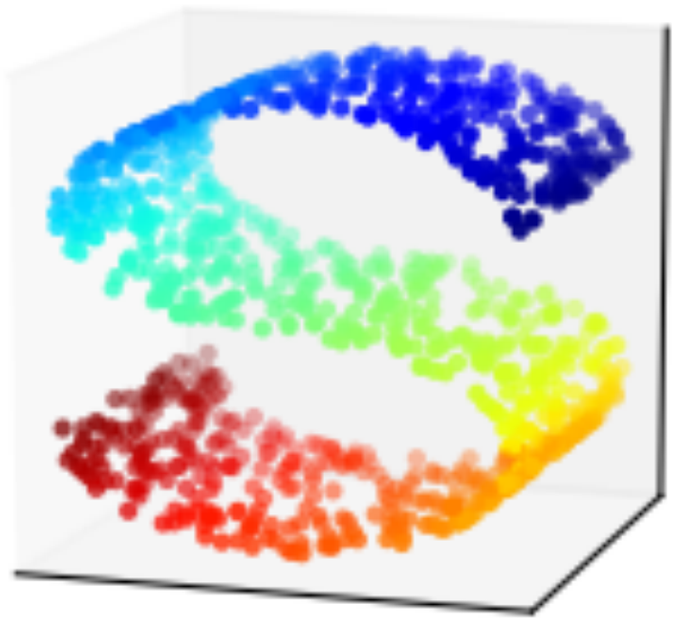
But this is a **non-linear** coordinate transformation so cannot be found by doing **PCA** which only can provide you with **linear transformations** of the input coordinates.

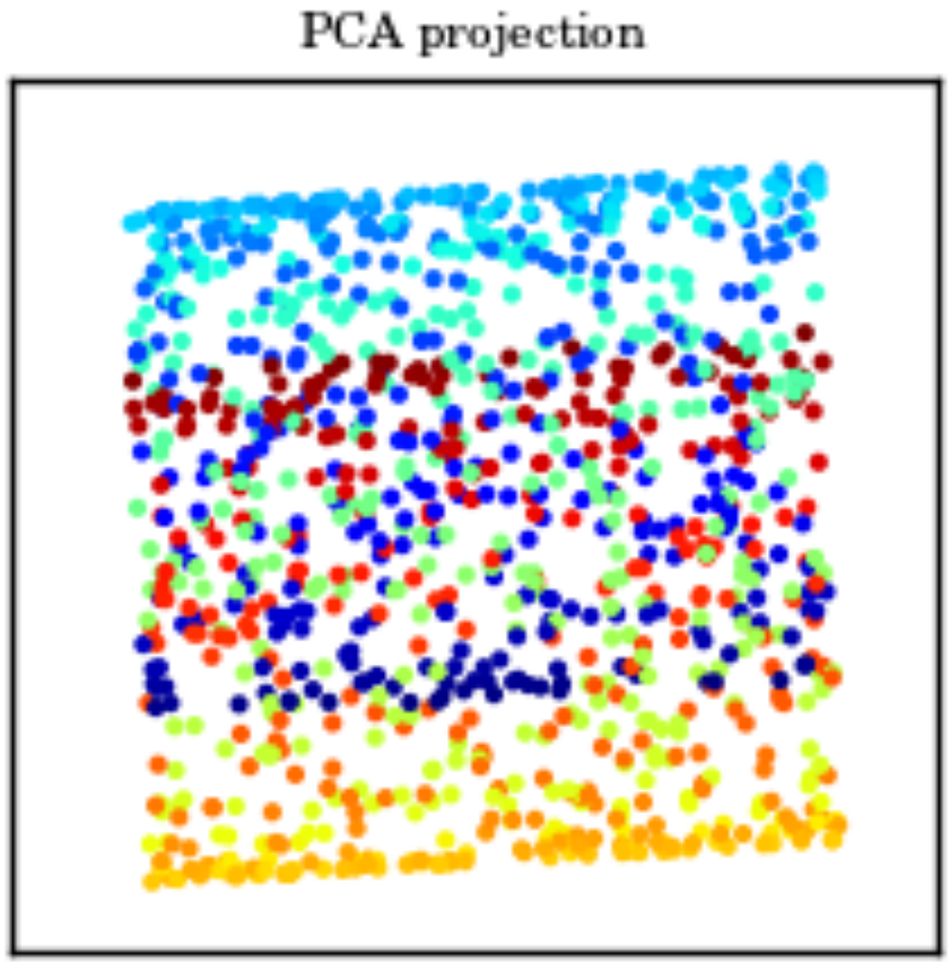
Linear vs non-linear



Kernel densities

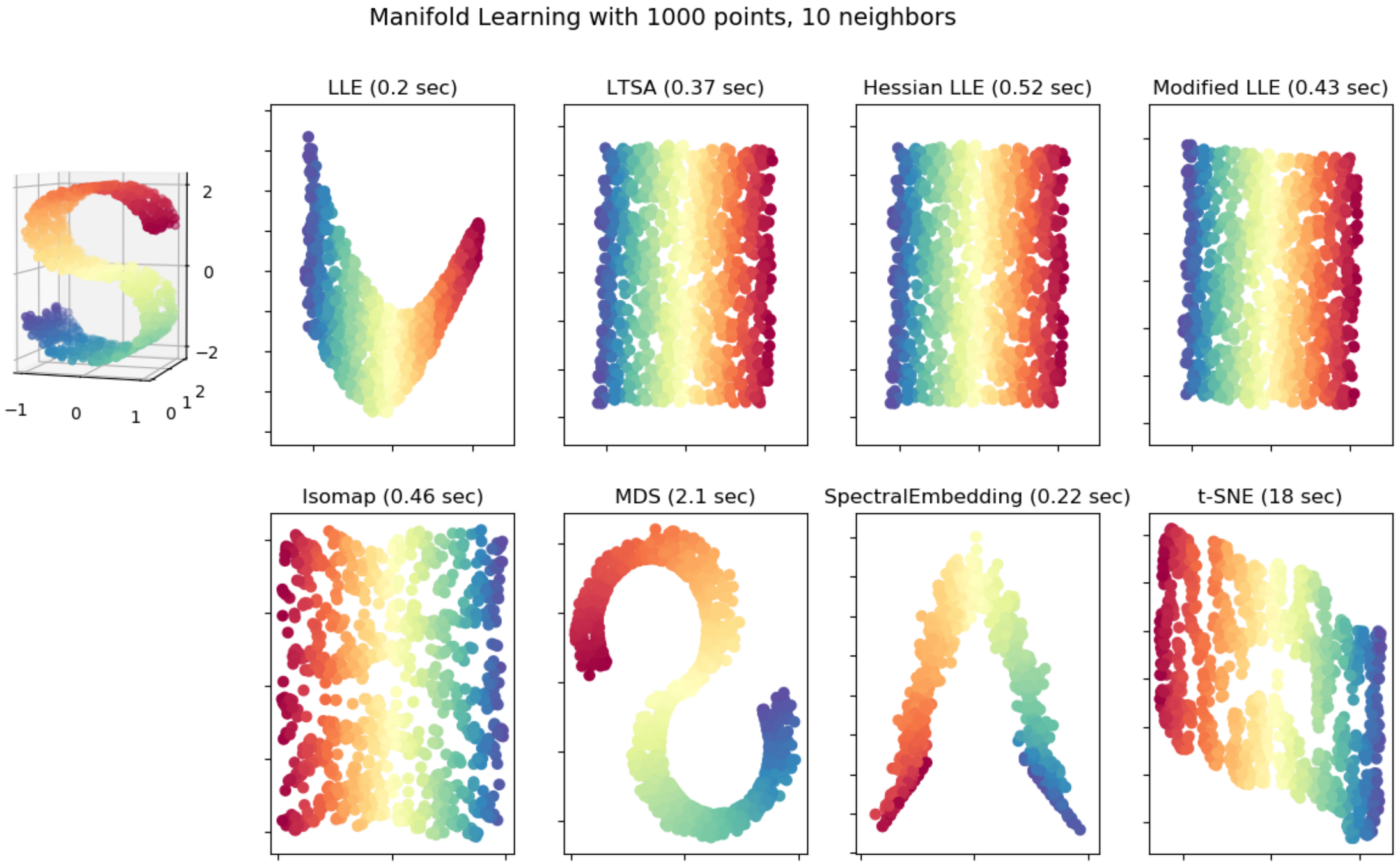
Linear versus non-linear





Enter manifold learning (or nonlinear dimensionality reduction)…

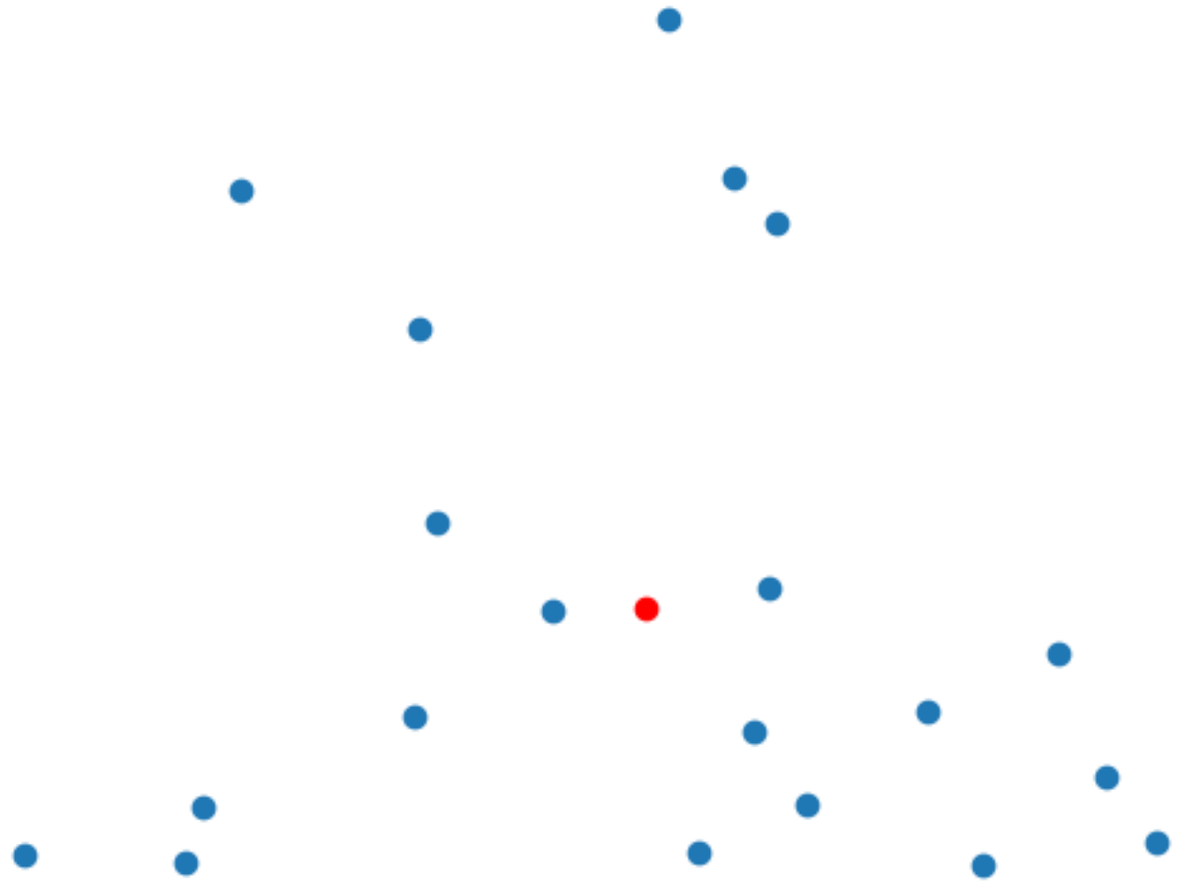
There is a zoo of these - here are a few:



See http://scikit-learn.org/stable/auto_examples/manifold/plot_compare_methods.html

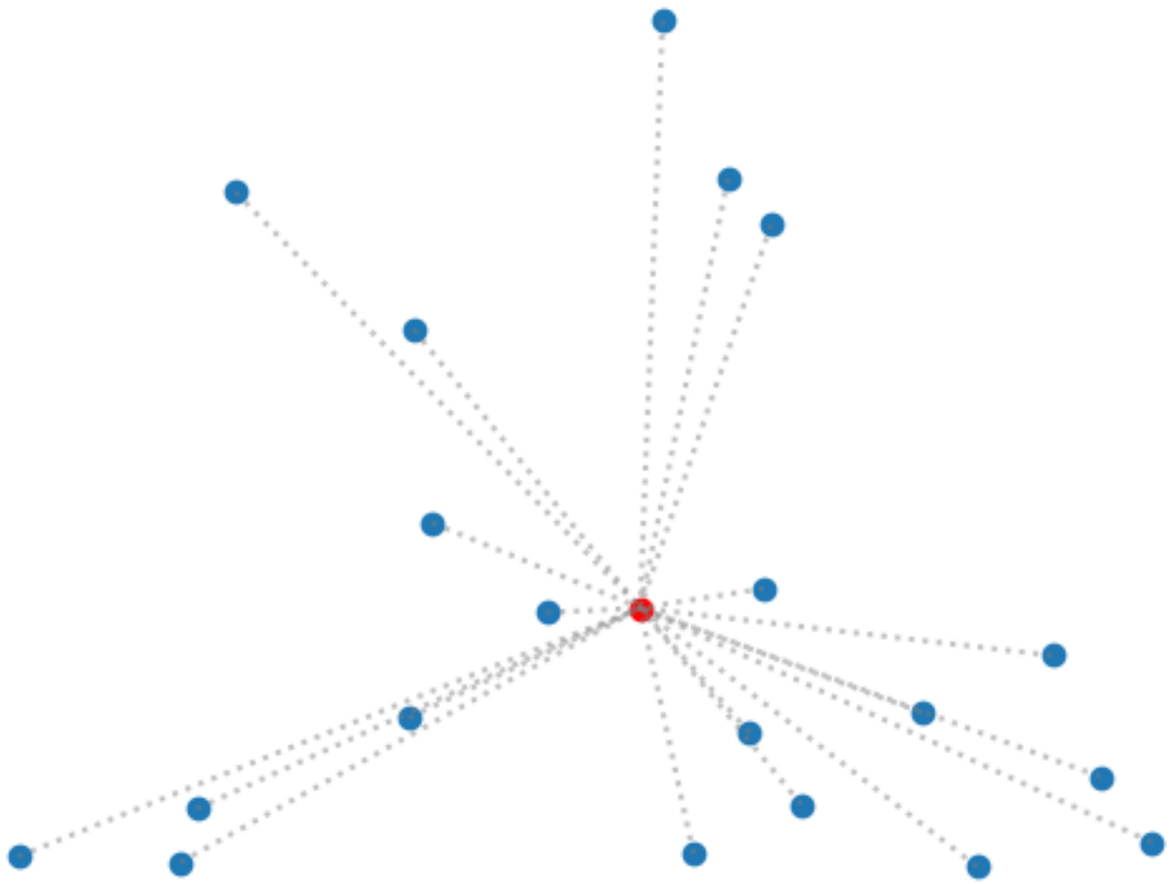
General features

As a rule, the various manifold learning methods use local structure to find good projections.



General features

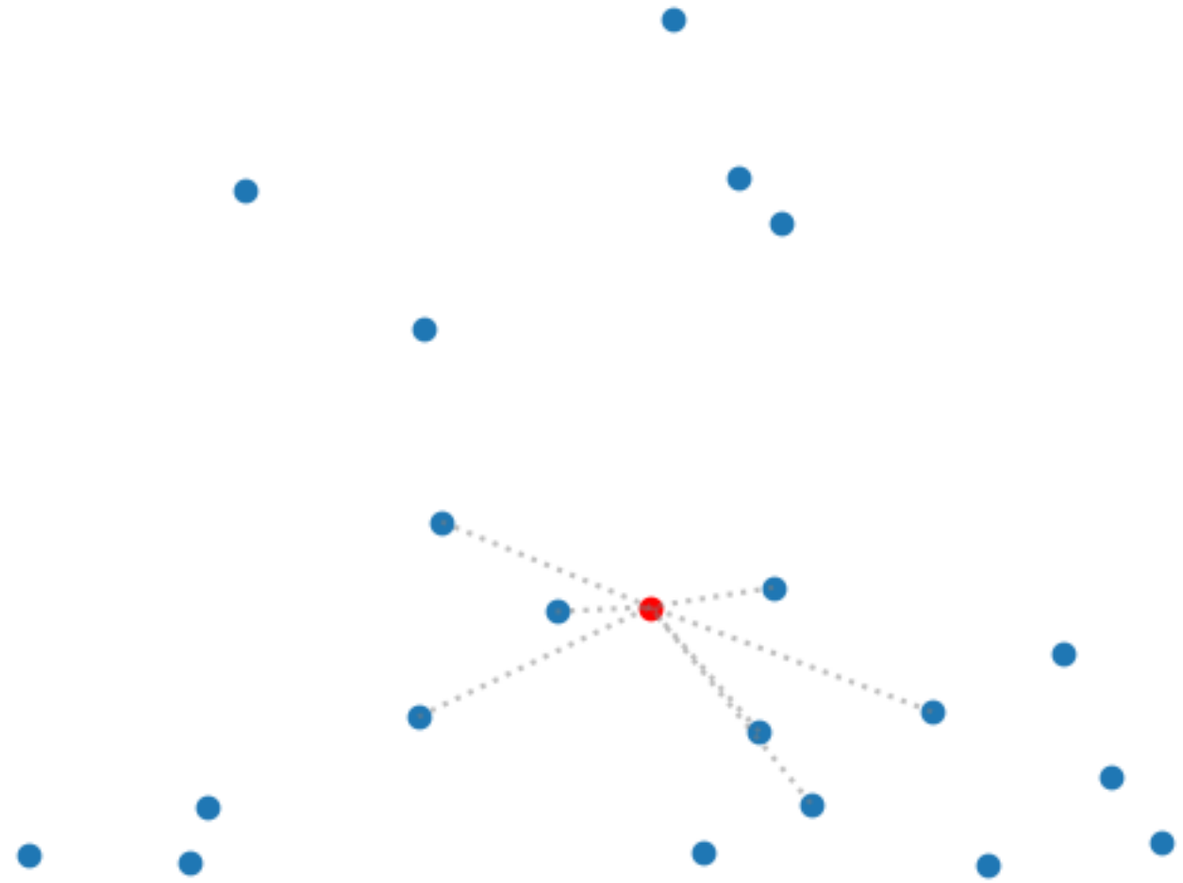
As a rule, the various manifold learning methods use local structure to find good projections.



Using global information to predict

General features

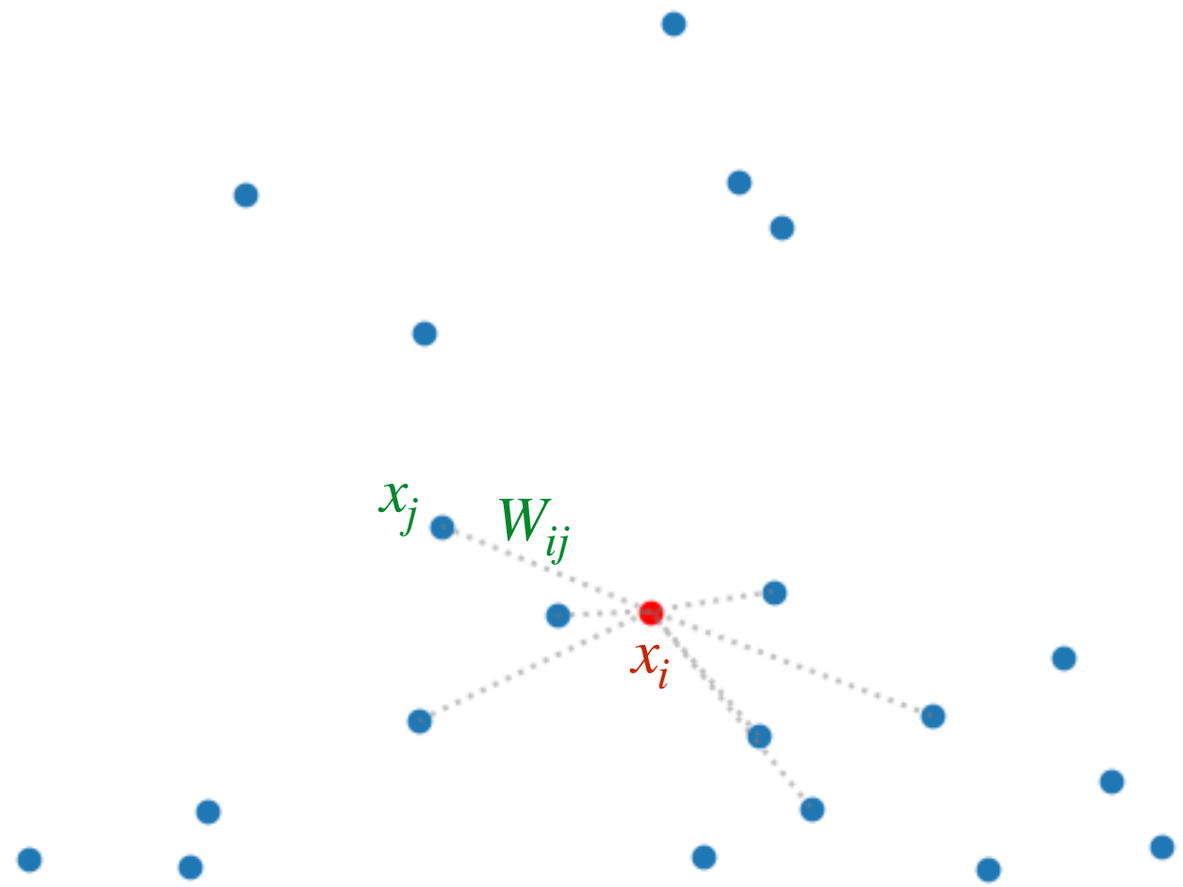
As a rule, the various manifold learning methods use local structure to find good projections.



Using local information to predict

General features

As a rule, the various manifold learning methods use local structure to find good projections.



Using local information to predict

General features

As a rule, the various manifold learning methods use local structure to find good projections.

Example: Locally Linear Embedding (LLE)

Here you try to fit small planes to the neighbourhood of each point and then match these together to project to lower dimensions.

So basically we want to minimise the reconstruction error:

$$\mathrm{Err} = |X - W X|^2$$

Locally Linear Embedding

So basically we want to minimise the reconstruction error of X based on its neighbours:

$$\mathrm{Err} = |X - W X|^2$$

Writing this out we have

$$\mathrm{Err}_x = \sum_{i=1}^{N} \left| x_i - \sum_{j=1}^{N} W_{ij} x_j \right|^2$$

In LLE the trick is to set $W_{ii}=0$ and W_{ij} to zero for all but the k nearest neighbours. Then we minimise the error function to find W_{ij}

Locally Linear Embedding

That sorts things out in the high dimensional space, but now we want to project onto a lower dimensional space. To do this we minimise:

$$\mathrm{Err}_y = \sum_{i=1}^{N} \left| y_i - \sum_{j=1}^{N} W_{ij} y_j \right|^2$$

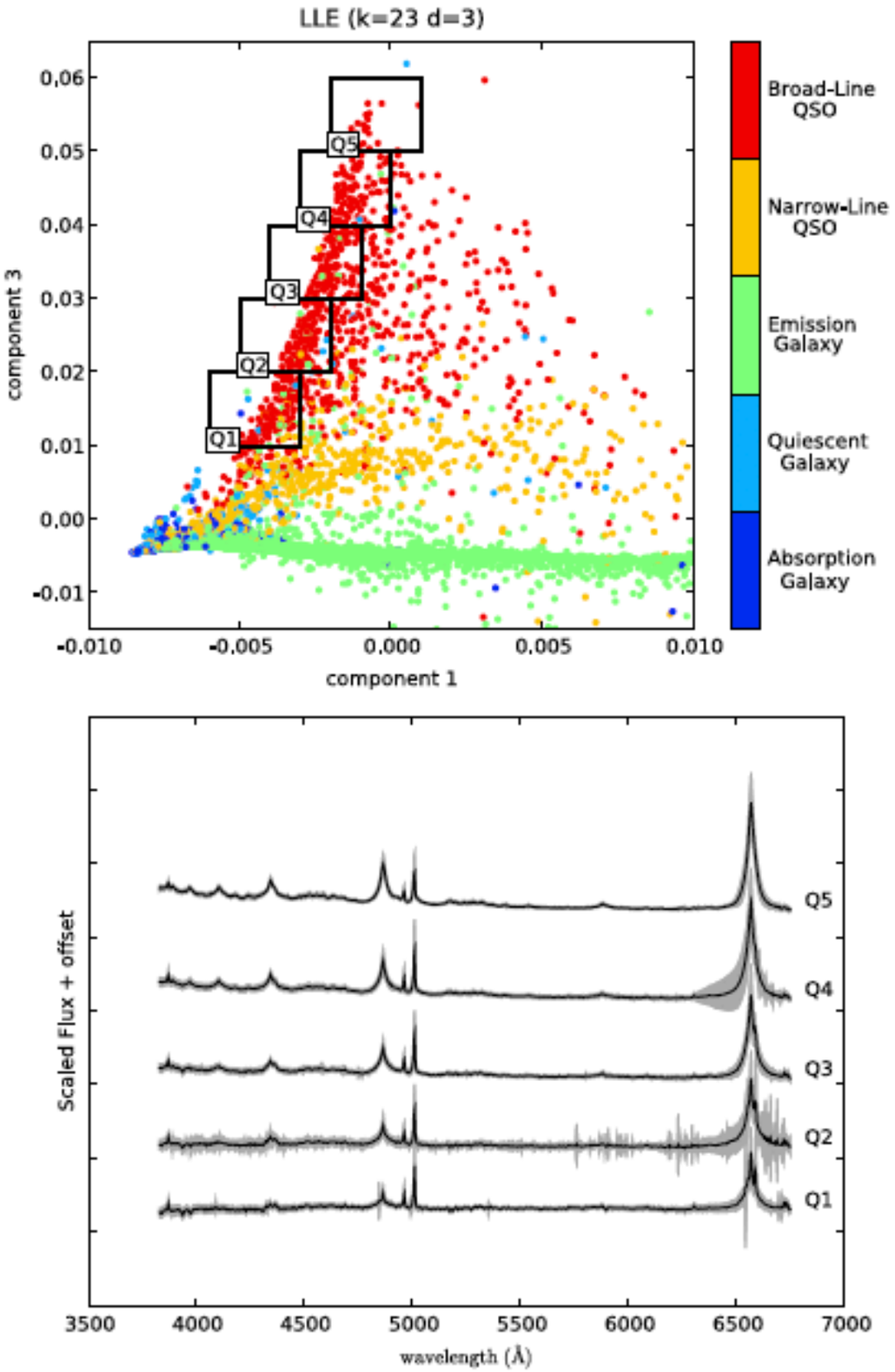
Here W_{ij} is fixed and the positions, y_i, are modified. This is an eigenvector problem and you get some eigenvectors as results.

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(k, n)
lle.fit(X)
proj = lle.transform(X)
```

Locally Linear Embedding in astronomy

Vanderplas & Connolly (2009):

LLEs can be used to classify galaxy spectra from the Sloan Digital Sky Survey. The data is here entire spectra and you look at some components of this.



t-Stochastic Neighbour Embedding (t-SNE)

This is an example of a method that is really aimed to project to 2-3 dimensions.

1. Measure the similarities (“closeness”) of samples in the original high-D space

$$p_{j|i} = \frac{\exp\left(-\left|x_i - x_j\right|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\left|x_i - x_k\right|^2 / 2\sigma_i^2\right)} \quad \Rightarrow \quad p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

2. Create an error function based on the separation in the low-dimensional space (coordinates y) and optimise this:

$$q_{ij} = \frac{f\left(\|y_i - y_j\|\right)}{\sum_{k \neq l} f\left(\|y_k - y_l\|\right)}$$

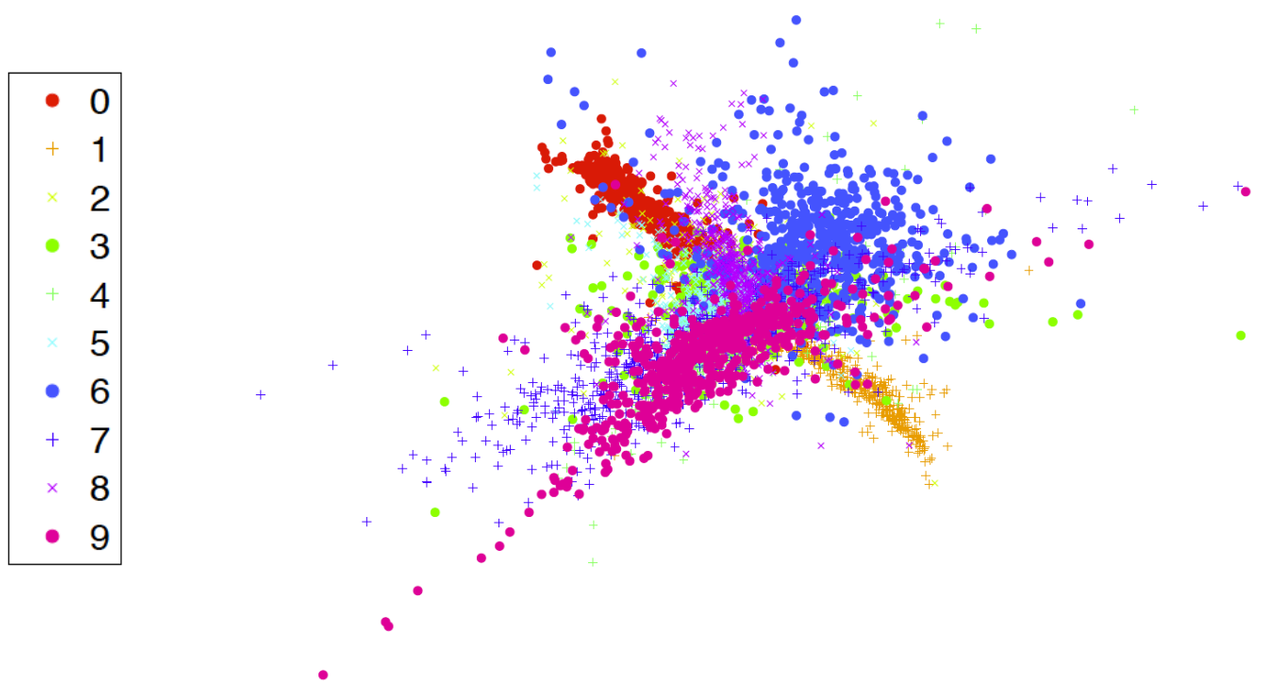
$$f(x) = \frac{1}{1 + x^2}$$

A Cauchy distribution

t-Stochastic Neighbour Embedding (t-SNE)

The MNIST hand-written digits dataset is a classic dataset for testing.

LLE:

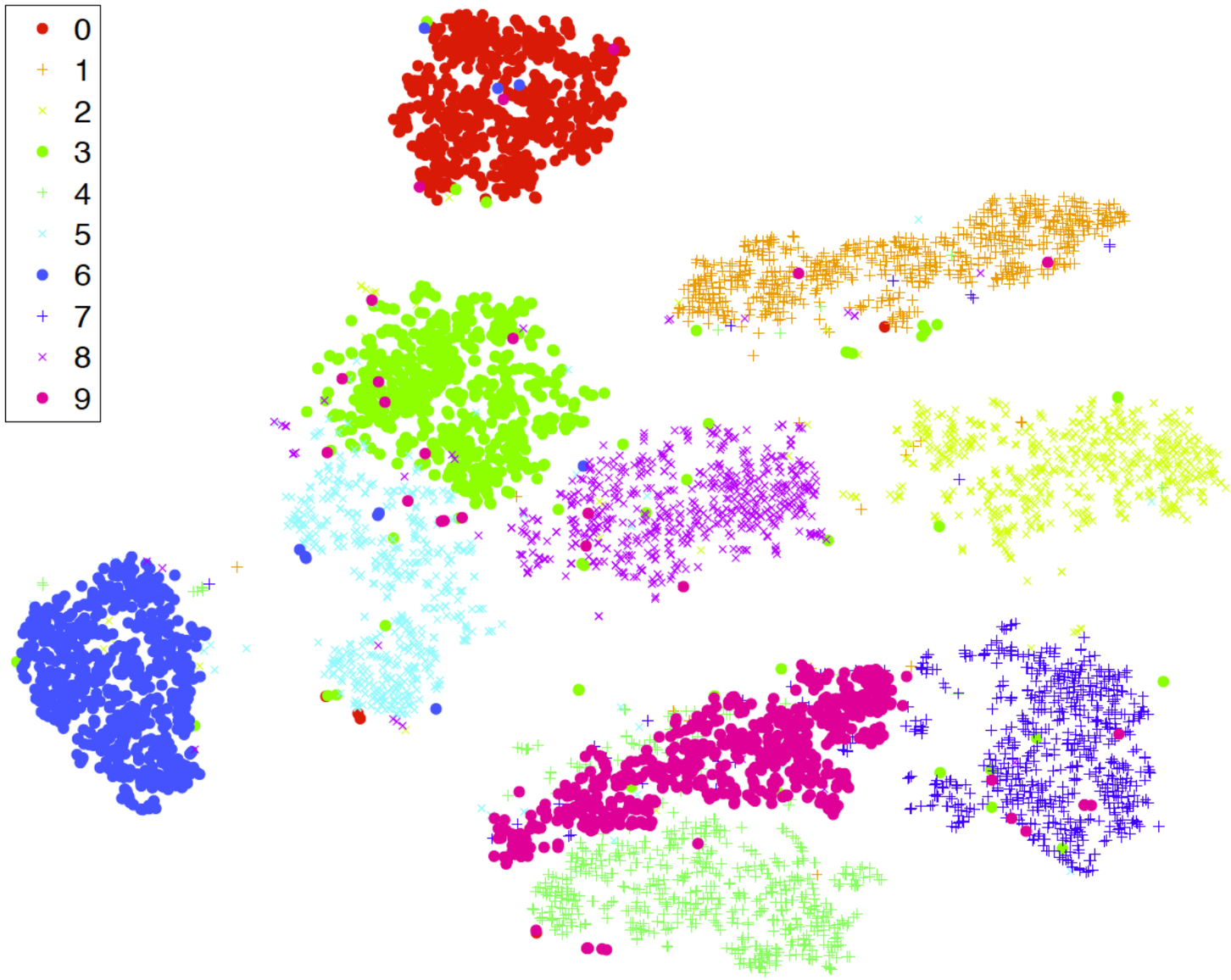


From van Maaten & Hinton (2008), Journal of Machine Learning Research 9 (2008) 2579-2605

t-Stochastic Neighbour Embedding (t-SNE)

The MNIST hand-written digits dataset is a classic dataset for testing.

t-SNE:



From van Maaten & Hinton (2008), Journal of Machine Learning Research 9 (2008) 2579-2605

t-Stochastic Neighbour Embedding (t-SNE)

9

see: https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm

t-Stochastic Neighbour Embedding (t-SNE)

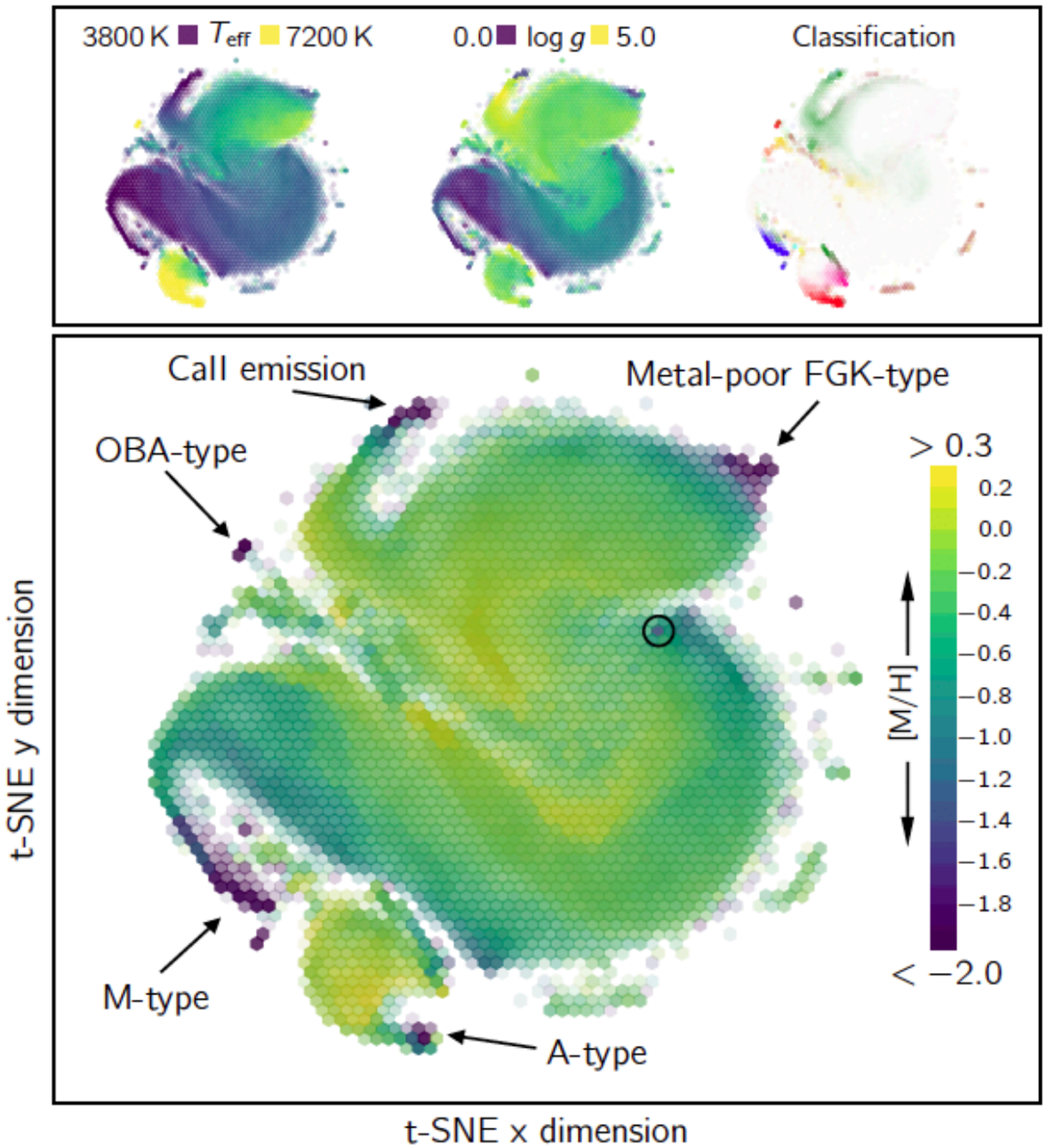
9

see: https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm

t-SNE and others and astronomy

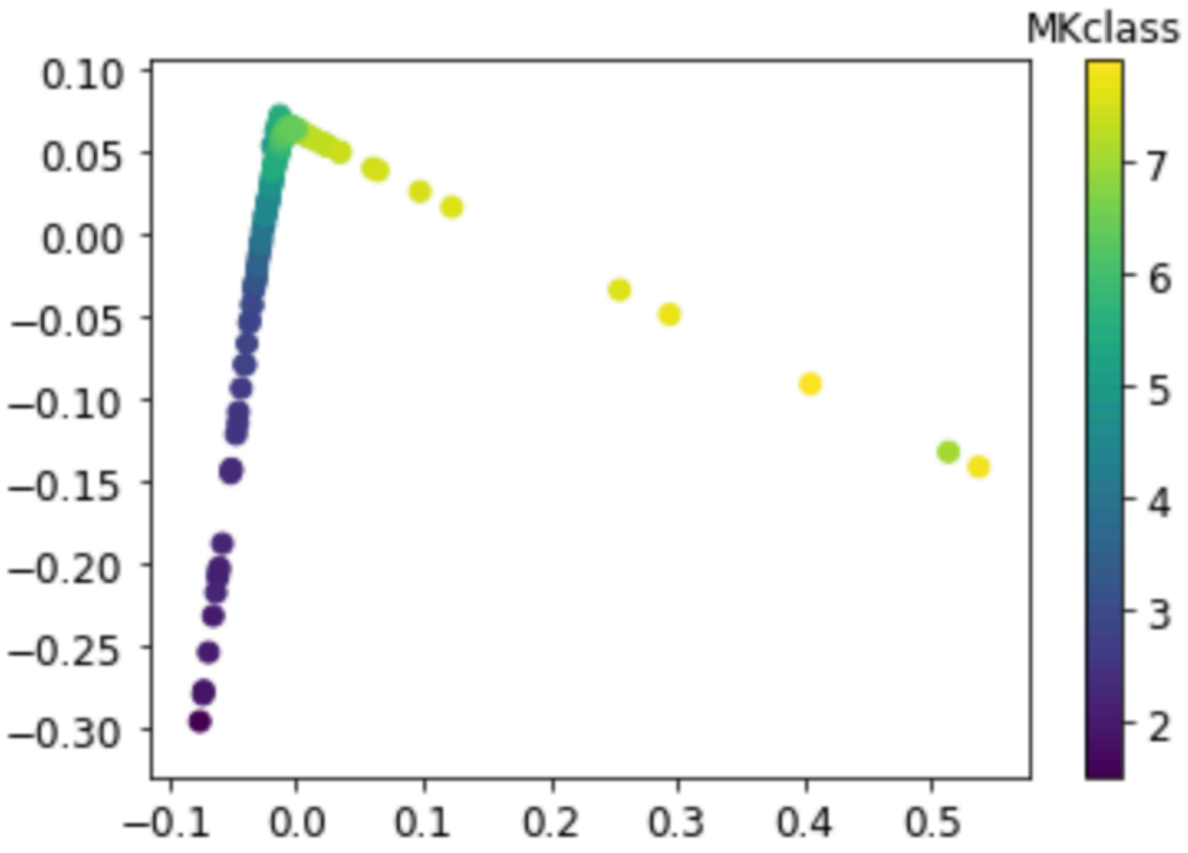
t-SNE is fairly slow but with an update using the Barnes-Hut N-body algorithm it is now useable for large-ish datasets.

Matijevic et al (2017):
Use t-SNE to analyse spectra of stars from the RAVE survey. This is used as a way to identify candidate metal-poor stars.



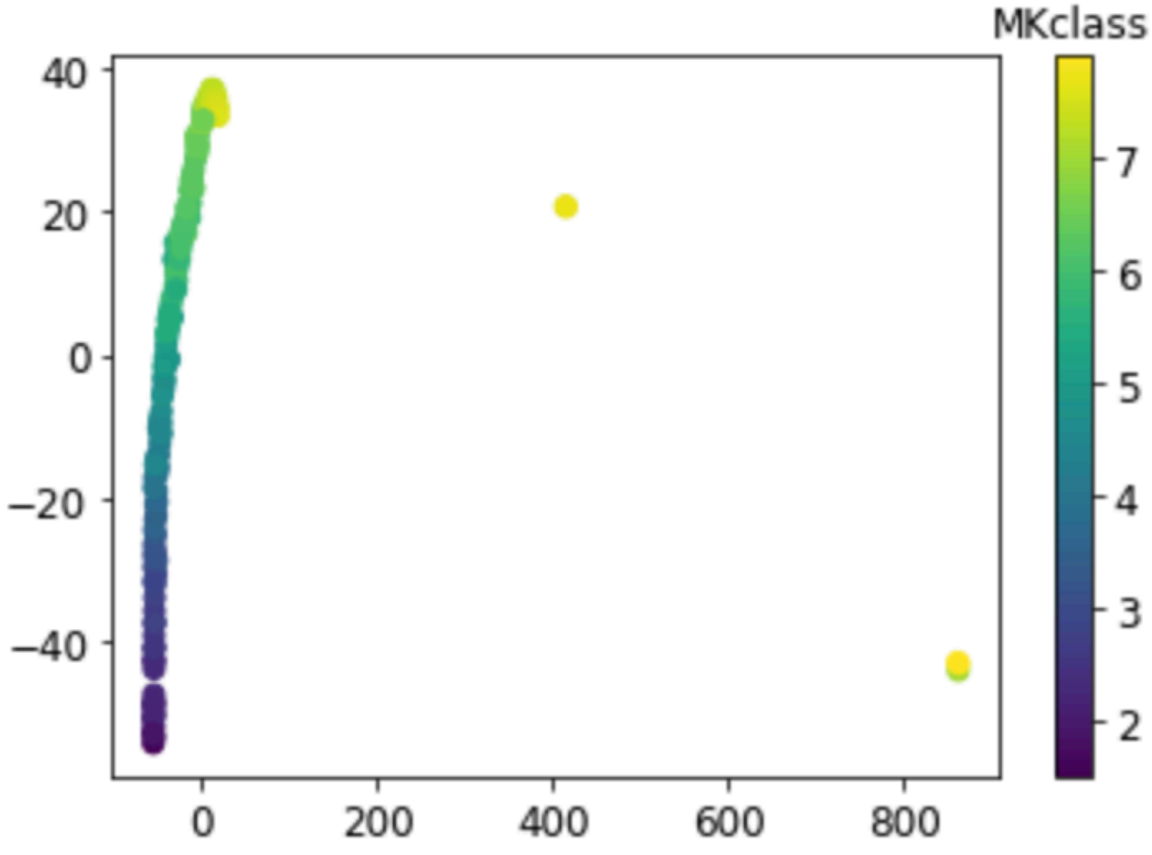
t-SNE & LLE on the stellar spectra

LLE




```
lle = LocallyLinearEmbedding(10,
n_components=2)
proj = lle.fit_transform(X)
```

t-SNE




```
sne = TSNE(n_components=2, init='pca',
random_state=0, perplexity=10)
proj = sne.fit_transform(X)
```

The pros and cons of manifold learners

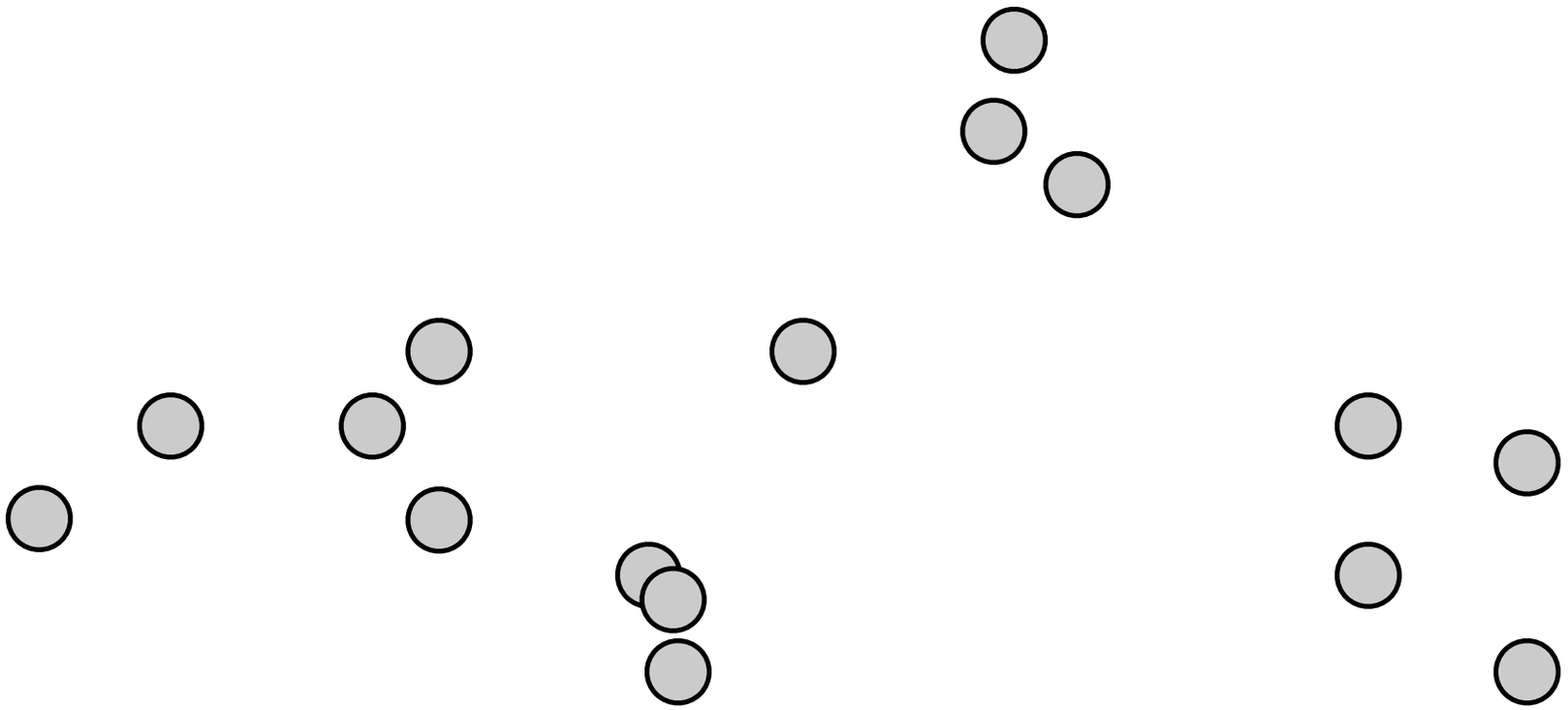
- Non-linear dimension reduction can extract structure out of data with non-linearities.
- Can lead to substantial dimensional reduction.
- They deal badly with noisy/gappy data and are (usually) sensitive to outliers.
- All have tuning parameters that you need to decide on.
- To project to lower dimensions you need the full data.
- Memory/CPU requirements can be substantial.

Bottom line: Try PCA first, or possibly NMF.

Regression - keeping track of local properties - see mostly lecture 1

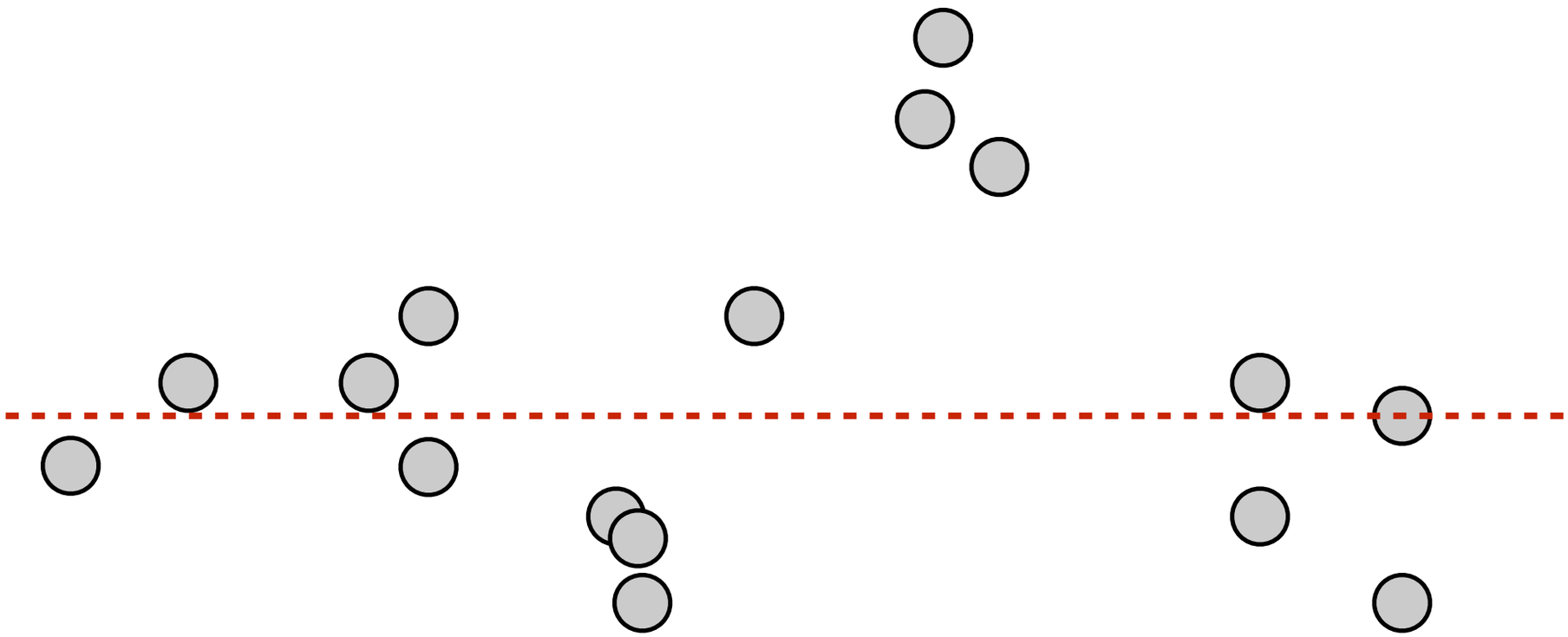
The broad idea

Faced with data where we do not have a clear idea how they are related a simple regression might be unsuitable:



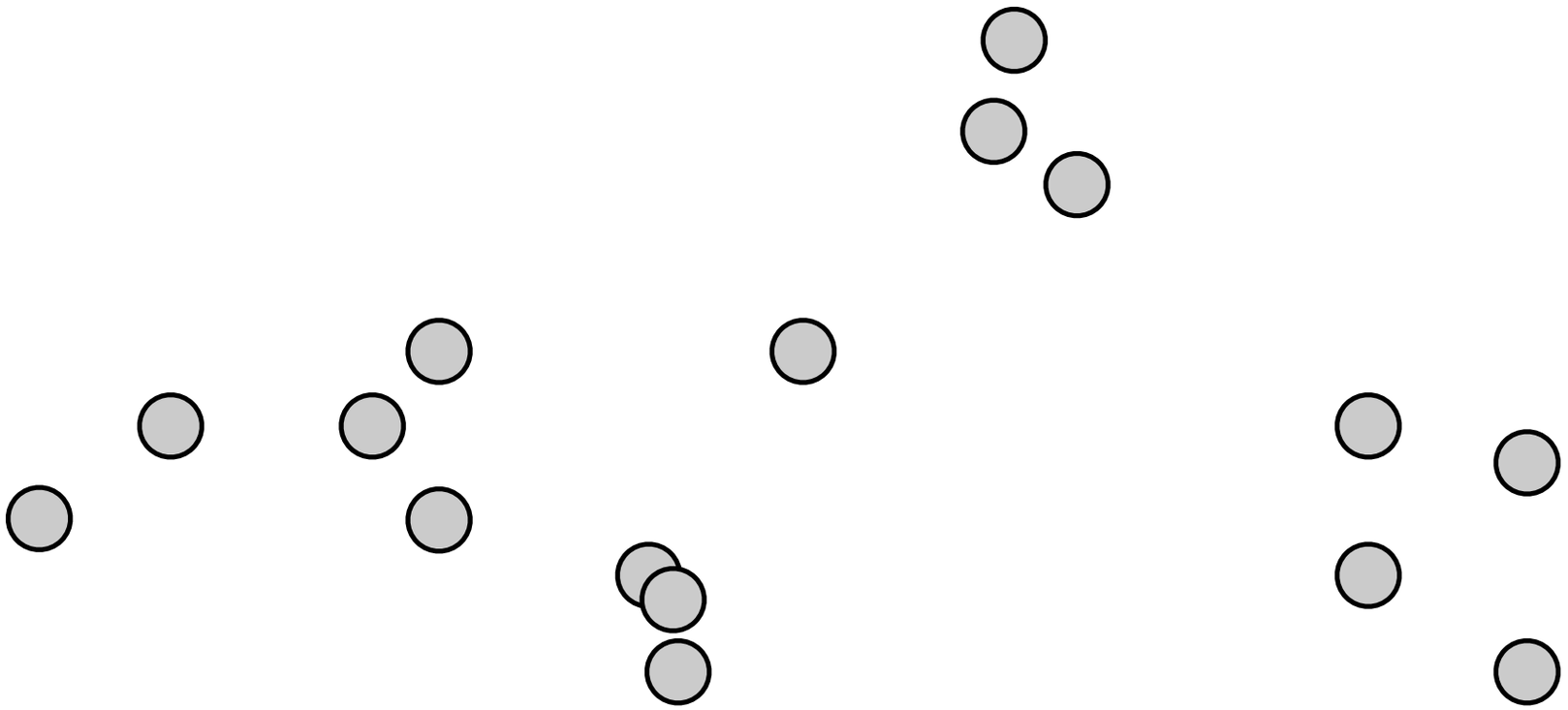
The broad idea

Faced with data where we do not have a clear idea how they are related a simple regression might be unsuitable:



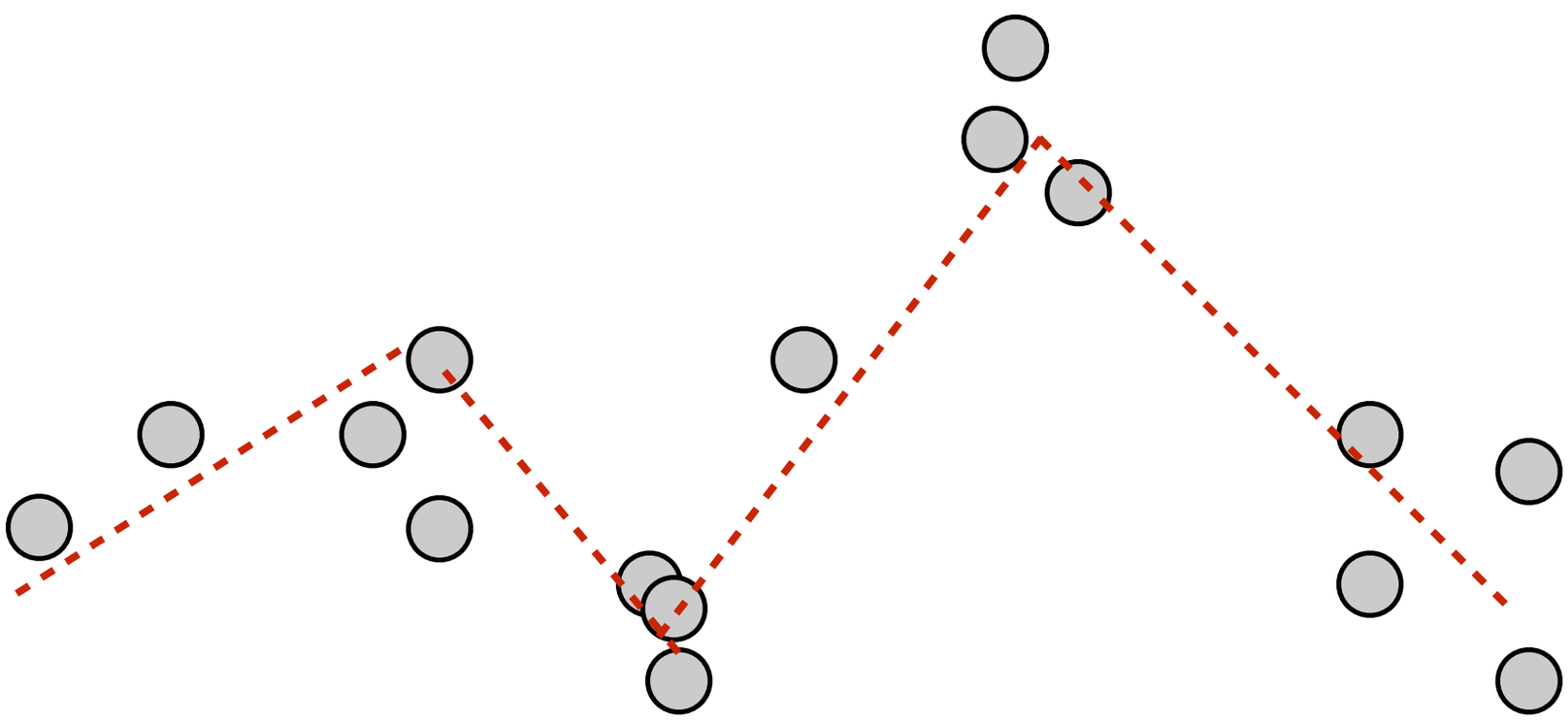
The broad idea

Keeping track of local trends might be a better idea:



The broad idea

Keeping track of local trends might be a better idea:



The main techniques

- Nearest neighbour regression
  - Take the mean of the k nearest points
- Kernel regression
  - Calculate the weighted mean of training points
- Locally linear regression
  - Calculate a weighted linear regression at each point
- Gaussian process regression
  - Drop fixed functions and try to fit in the space of “all” functions

The main techniques

- Nearest neighbour regression
  - Take the mean of the k nearest points
- Kernel regression
  - Calculate the weighted mean of training points
- Locally linear regression
  - Calculate a weighted linear regression at each point
- Gaussian process regression
  - Drop fixed functions and try to fit in the space of “all” functions

Gaussian process regression

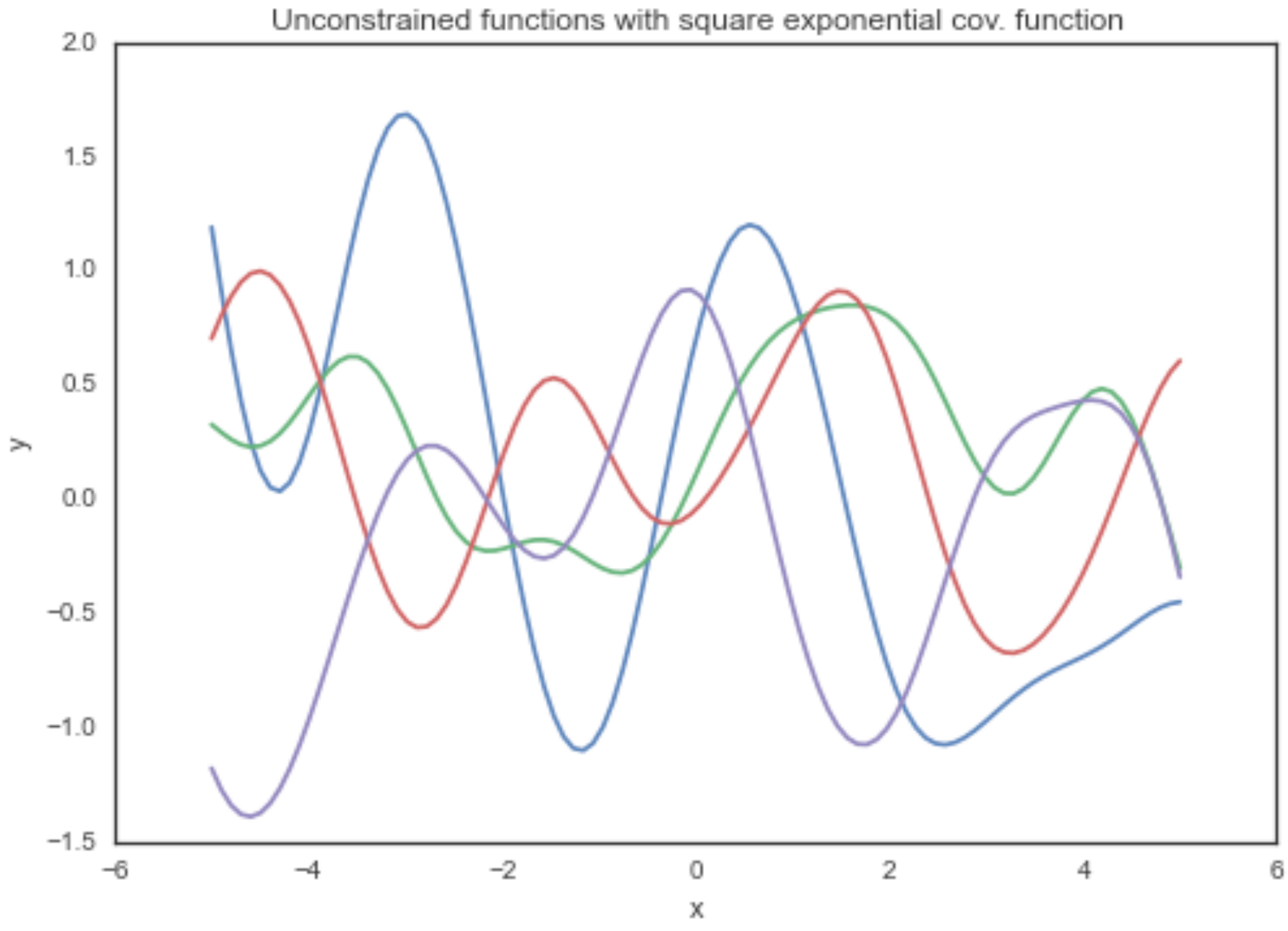
In this case we apply a prior in function space - this prior is specified using a mean & covariance function (since that is all we need for a Gaussian). The most common is:

$$\mathrm{Cov}\,(x, x') = K(x, x') = \exp\left(-\frac{|x - x'|}{2h}\right)$$

If we set the mean to zero, we can then draw random functions because at each x we know what the covariance matrix should be and that is all we need.

Gaussian process regression

Random functions - h=1

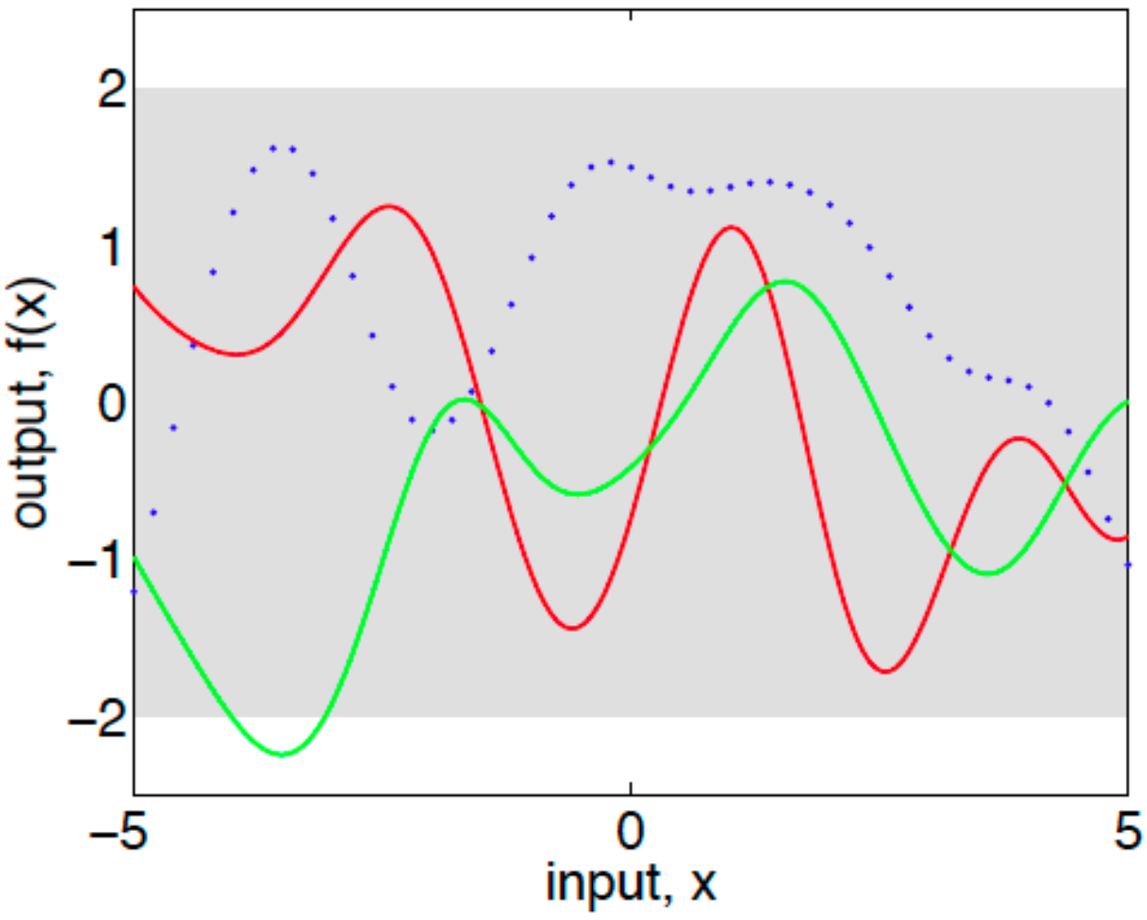


See the Jupyter notebook on the Github site:

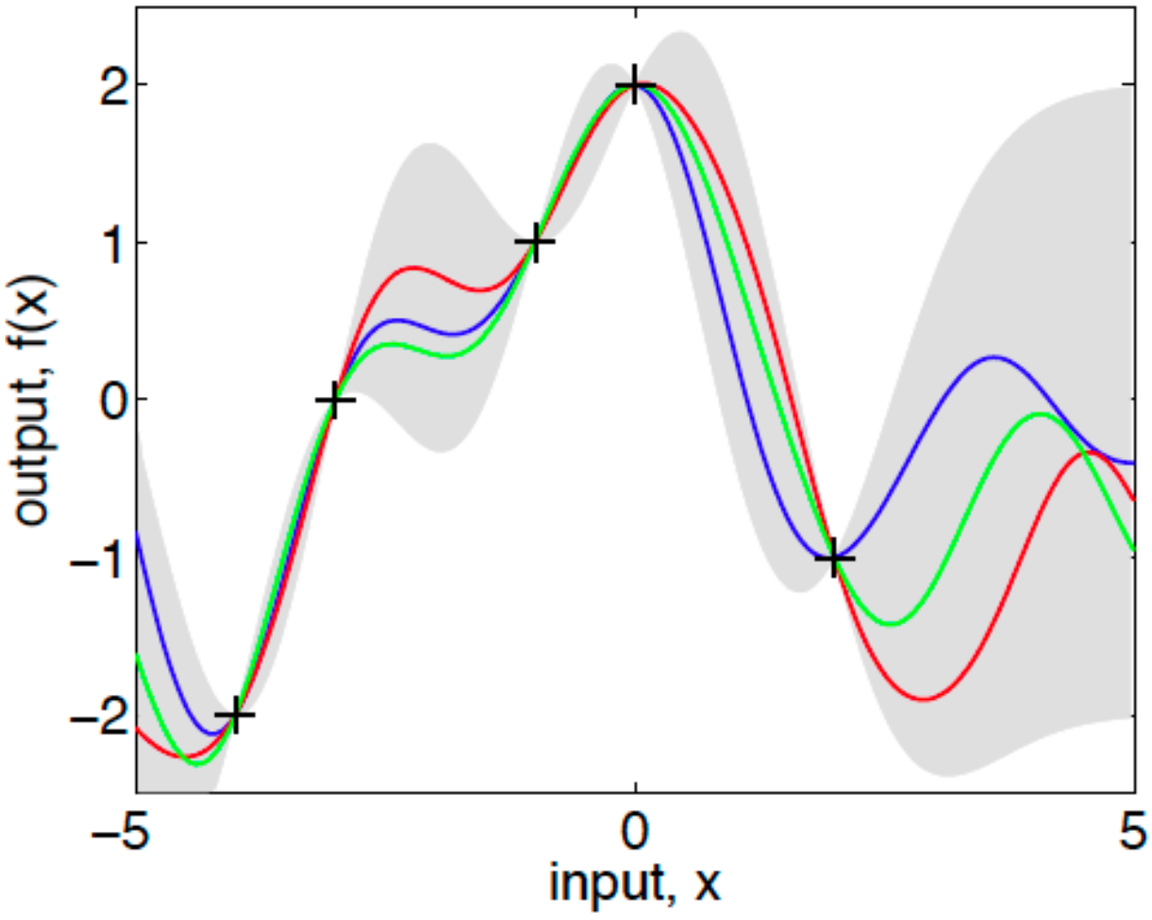
Lectures/Lecture 5/Notebooks/Gaussian process regression.ipynb

Gaussian process regression

We apply constraints by multiplying the prior with the likelihood:



(a), prior



(b), posterior

Taken from Rasmussen & Williams, “Gaussian processes for Machine Learning”, 2006, Figure 2.2
http://www.gaussianprocess.org/

Gaussian Process Regression - features

\+ Very flexible

\+ Provides covariance estimates on predictions

\- Fairly slow

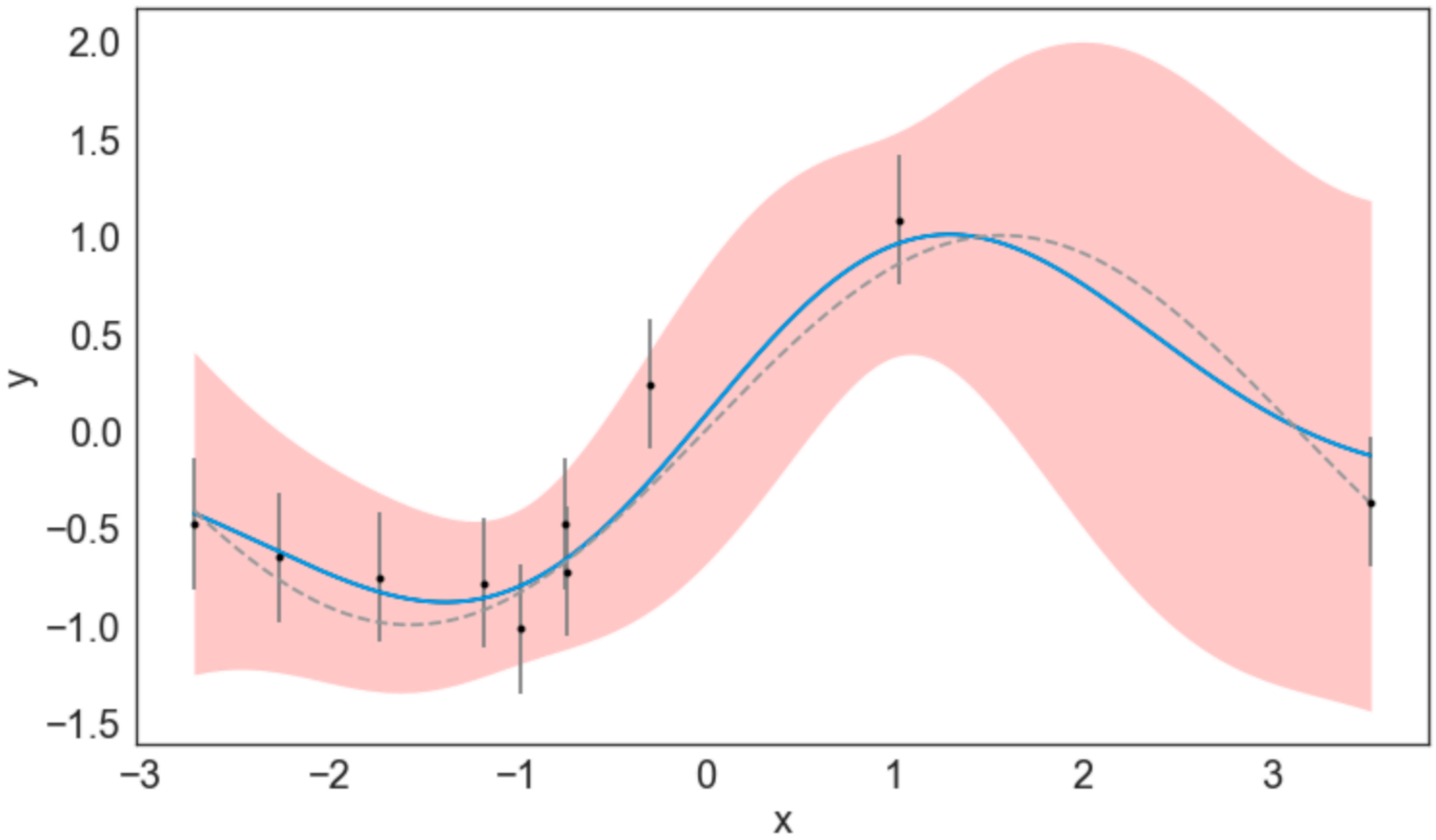
Python usage:

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel

gp = GaussianProcessRegressor(kernel=RBF(0.1),
                              alpha=(dy/y)**2)
```

Example use:

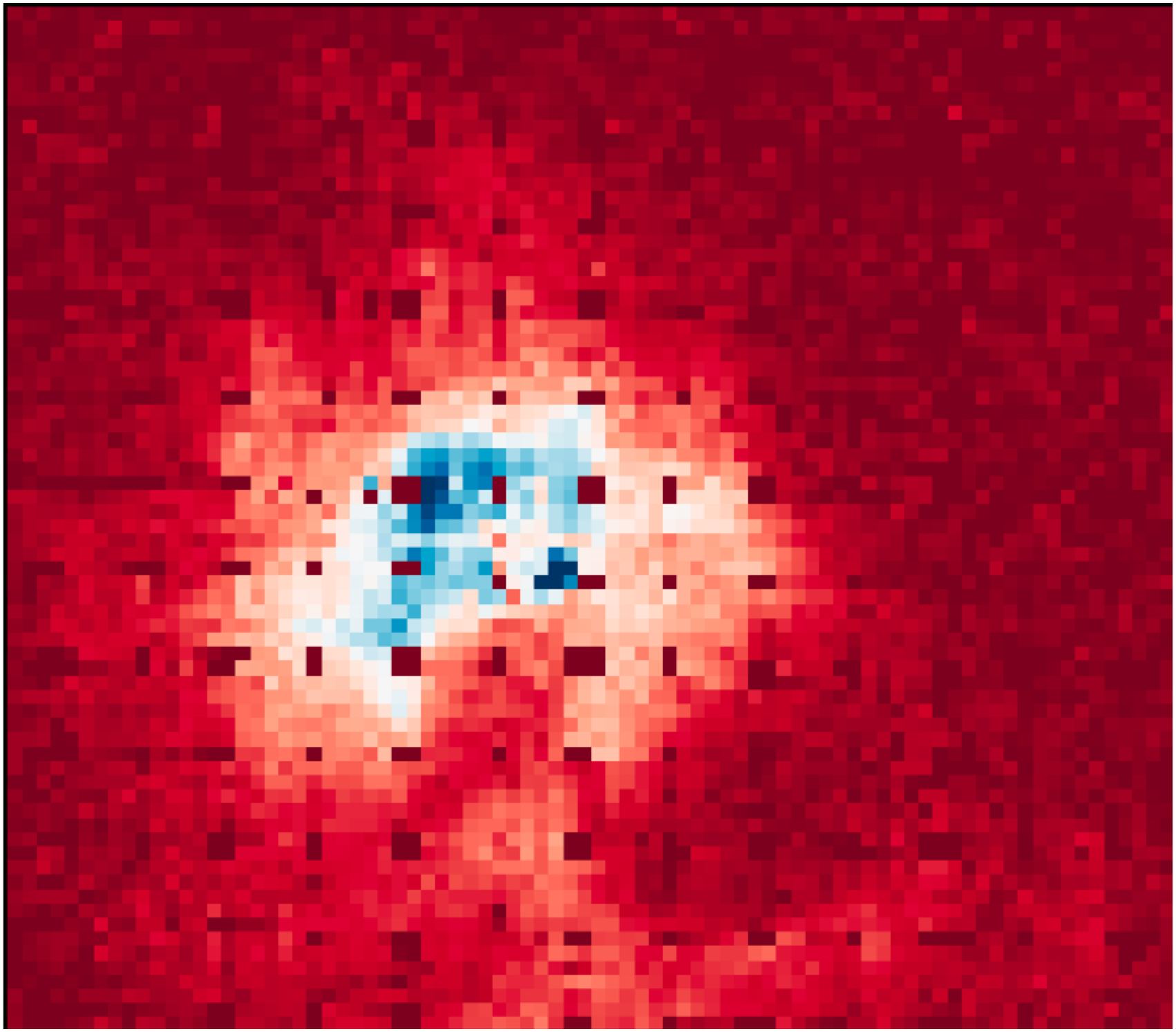
```
g = gp.fit(x[:, np.newaxis], y)
y_pred, sigma = gp.predict(xplot[:, np.newaxis], return_std=True)
plot_a_fit(x, y, dy, xplot, y_pred, sigma, include_true=True)
```



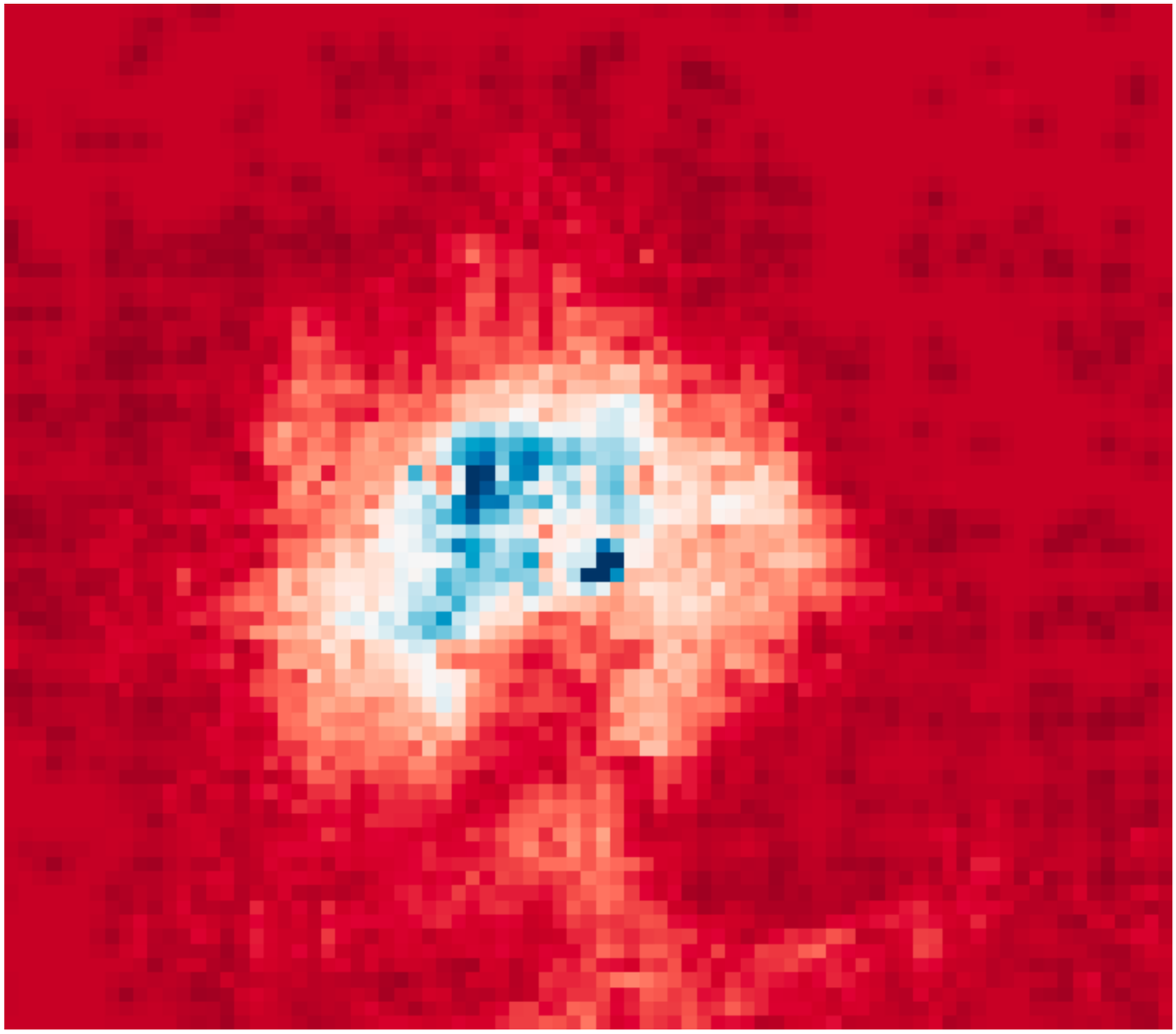
See the Jupyter notebook on the Github site:

Lectures/Lecture 5/Notebooks/Gaussian process regression.ipynb

Example for an image



Example for an image



Some useful resources:

Very nice interactive examples + written explanation:

https://distill.pub/2019/visual-exploration-gaussian-processes/

A tidy and simple introduction with a pedagogic set of Python examples - and not just scikit-learn:

https://www.dominodatalab.com/blog/fitting-gaussian-process-models-python

The main textbook on Gaussian Processes (Rasmussen & Williams 2006) can be found here (with other resources):

http://gaussianprocess.org/

Usefulness in astronomy

- Light-curve modelling: stars (Brewer & Stello 2009), AGNs (Kelly et al 2014), X-ray binaries (Uttley et al 2005).
- Gaussian random field models for cosmic microwave background and large scale structure (e.g. Bond & Efstathiou 1987) are also possible to cast as a Gaussian Process.
- Quasar time-delay modelling (Hojjati et al 2013).
- Spectroscopic calibration (Czekala et al 2017).
- Widely used also as part of a larger model.

Usefulness in astronomy

- Widely used - also as part of a larger model.

Example: Model emulation to create galaxy formation models -
Rodrigues, Vernon & Bower (2016, MNRAS, **466**:2, 2418)

Semi-analytic models of galaxy formation. Many parameters, expensive model to calculate.

The 20 parameters considered in Rodrigues et al:

Process modelled	Section	Parameter name [units]	Range		GP14	Scaling
Star formation (quiescent)	§2.2.1	ν_{sf} [Gyr^{-1}]	0.025	1.0	0.5	lin
		P_{sf}/k_B [cm^{-3}K]	1×10^4	5×10^4	1.7×10^4	log
		β_{sf}	0.65	1.10	0.8	lin
Star formation (bursts)	§2.2.2	f_{dyn}	1.0	100.0	10	log
		$\tau_{min,burst}$ [Gyr]	10^{-3}	1	0.05	log
SNe feedback	§2.2.3	α_{hot}	1.0	3.7	3.2	lin
		$\beta_{0,burst}$	0.5	40.0	11.16	lin
		$\beta_{0,disc}$	0.5	40.0	11.16	lin
		α_{reheat}	0.15	1.5	1.26027	lin
AGN feedback	§2.2.4	α_{cool}	0.1	2.0	0.6	log
		ε_{edd}	0.004	0.1	0.03979	log
		f_{smbh}	0.001	0.01	0.005	lin
Galaxy mergers		f_{burst}	0.01	0.5	0.1	log
		f_{ellip}	0.01	0.5	0.3	log
Disk stability	§2.2.5	f_{stab}	0.61	1.1	0.8	lin
Reionization		V_{cut} [km s^{-1}]	20	60	30	lin
		z_{cut}	5	15	10	lin
Metal enrichment		p_{yield}	0.02	0.05	0.021	lin
Ram pressure stripping		ε_{strip}	0.01	0.99	n/a	lin
		α_{rp}	1.0	3.0	n/a	lin

Rodrigues, Vernon & Bower (2016, MNRAS, **466**:2, 2418)

Fitting data to model

The model is prohibitively expensive to run millions/billions of times, so how do you fit this in a Bayesian framework?

Fitting data to model

The model is prohibitively expensive to run millions/billions of times, so how do you fit this in a Bayesian framework?

Write the vector of model outputs as:

$$\mathbf{M}\left(\nu_{0,\mathrm{sf}}, P_{\mathrm{sf}}/k_{\mathrm{B}}, \ldots, \alpha_{\mathrm{rp}}\right) = \mathbf{M}(\theta)$$

Fitting data to model

The model is prohibitively expensive to run millions/billions of times, so how do you fit this in a Bayesian framework?

Write the vector of model outputs as:

$$\mathbf{M}\left(\nu_{0,\mathrm{sf}}, P_{\mathrm{sf}}/k_{\mathrm{B}}, \ldots, \alpha_{\mathrm{rp}}\right) = \mathbf{M}(\theta)$$

Then write this as a regression problem:

$$M_i(\theta) = \sum_j \beta_{ij} g_{ij}(\theta) + u_i(\theta) + \nu_i(\theta)$$

Fitting data to model

The model is prohibitively expensive to run millions/billions of times, so how do you fit this in a Bayesian framework?

Write the vector of model outputs as:

$$\mathbf{M}\left(\nu_{0,\mathrm{sf}}, P_{\mathrm{sf}}/k_{\mathrm{B}}, \ldots, \alpha_{\mathrm{rp}}\right) = \mathbf{M}(\theta)$$

Then write this as a regression problem:

$$M_i(\theta) = \sum_j \beta_{ij} g_{ij}(\theta) + u_i(\theta) + \nu_i(\theta)$$

Polynomials

Fitting data to model

The model is prohibitively expensive to run millions/billions of times, so how do you fit this in a Bayesian framework?

Write the vector of model outputs as:

$$\mathbf{M}\left(\nu_{0,\mathrm{sf}}, P_{\mathrm{sf}}/k_{\mathrm{B}}, \ldots, \alpha_{\mathrm{rp}}\right) = \mathbf{M}(\theta)$$

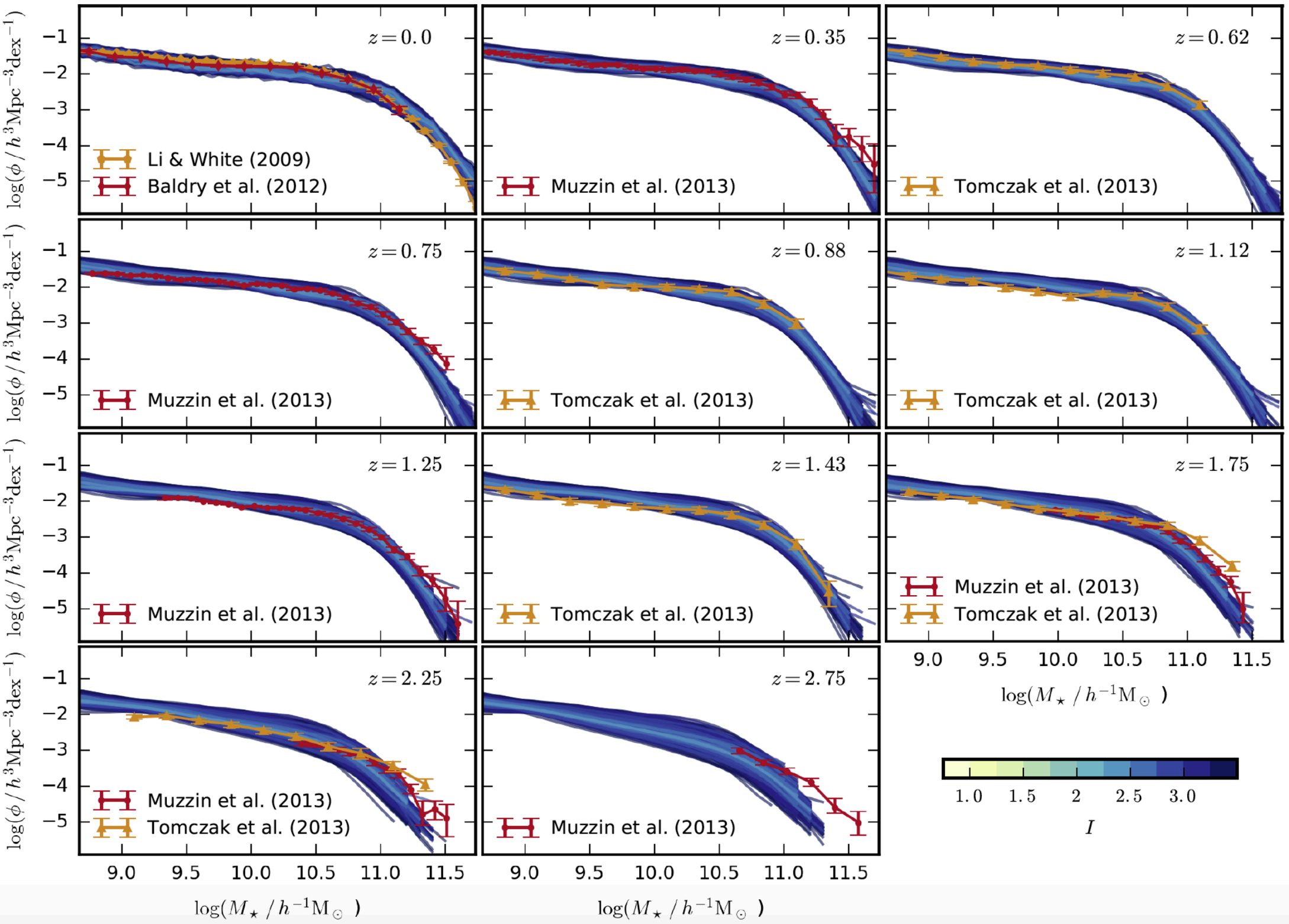
Then write this as a regression problem:

$$M_i(\theta) = \sum_j \beta_{ij} g_{ij}(\theta) + u_i(\theta) + \nu_i(\theta)$$

Polynomials

Gaussian Process

Fitting data to model - galaxy mass functions



Rodrigues, Vernon & Bower (2016, MNRAS, **466**:2, 2418)

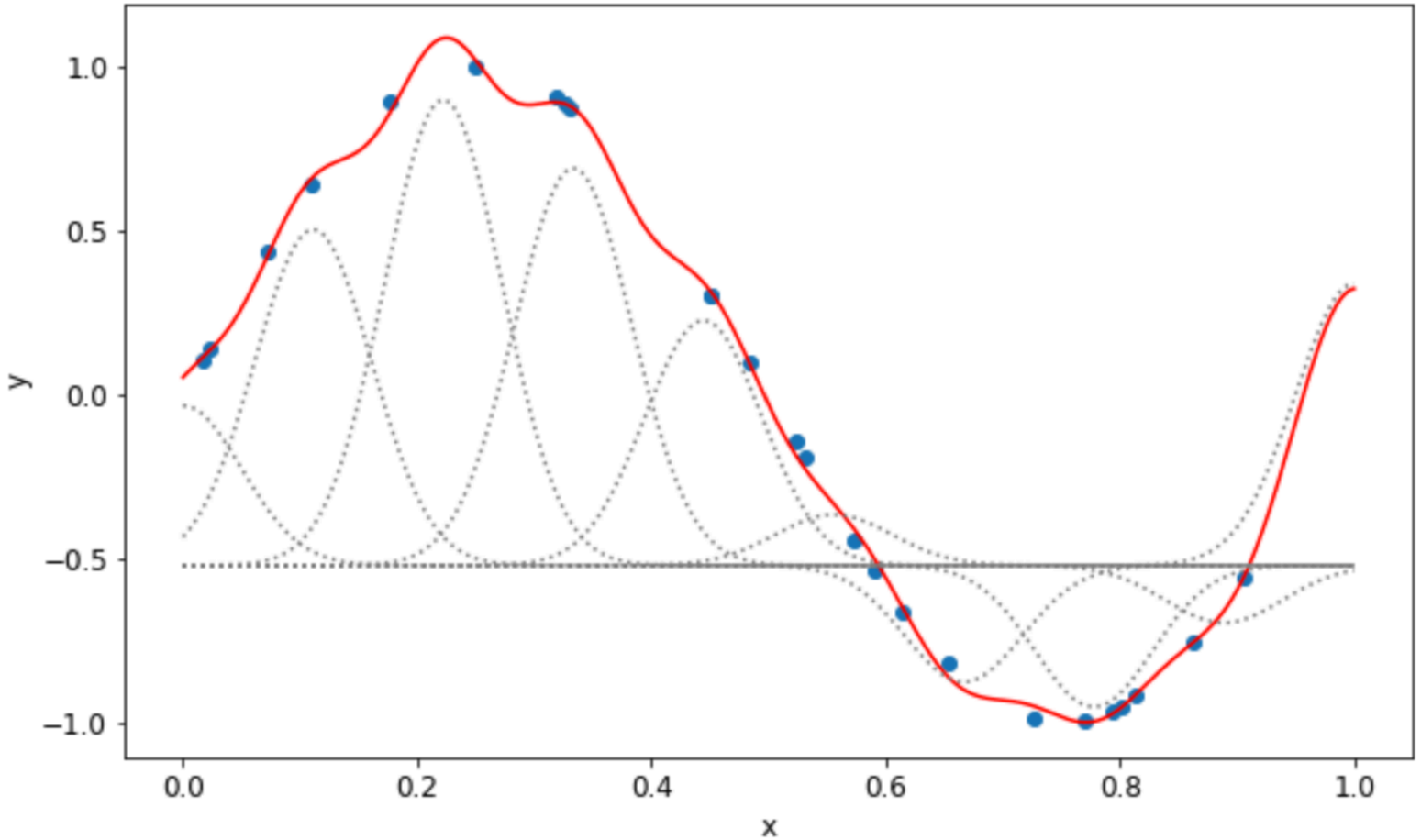
Basis function regression

Finally, let us return to an earlier topic - basis function regression.

$$f(x) = \sum_i w_i \phi_i(x)$$

This is linear regression, but transforming the x values.

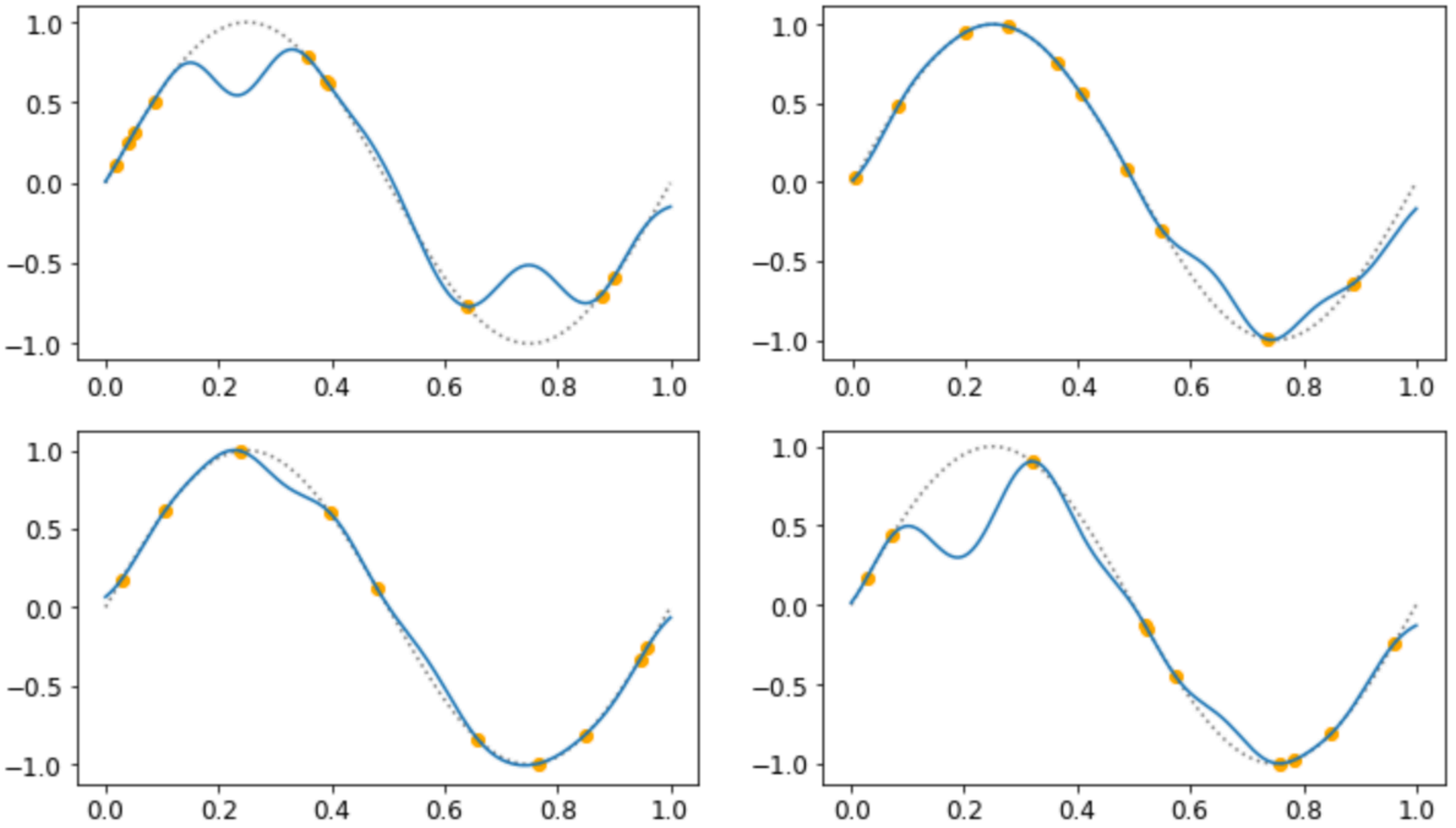
e.g. Gaussian basis:



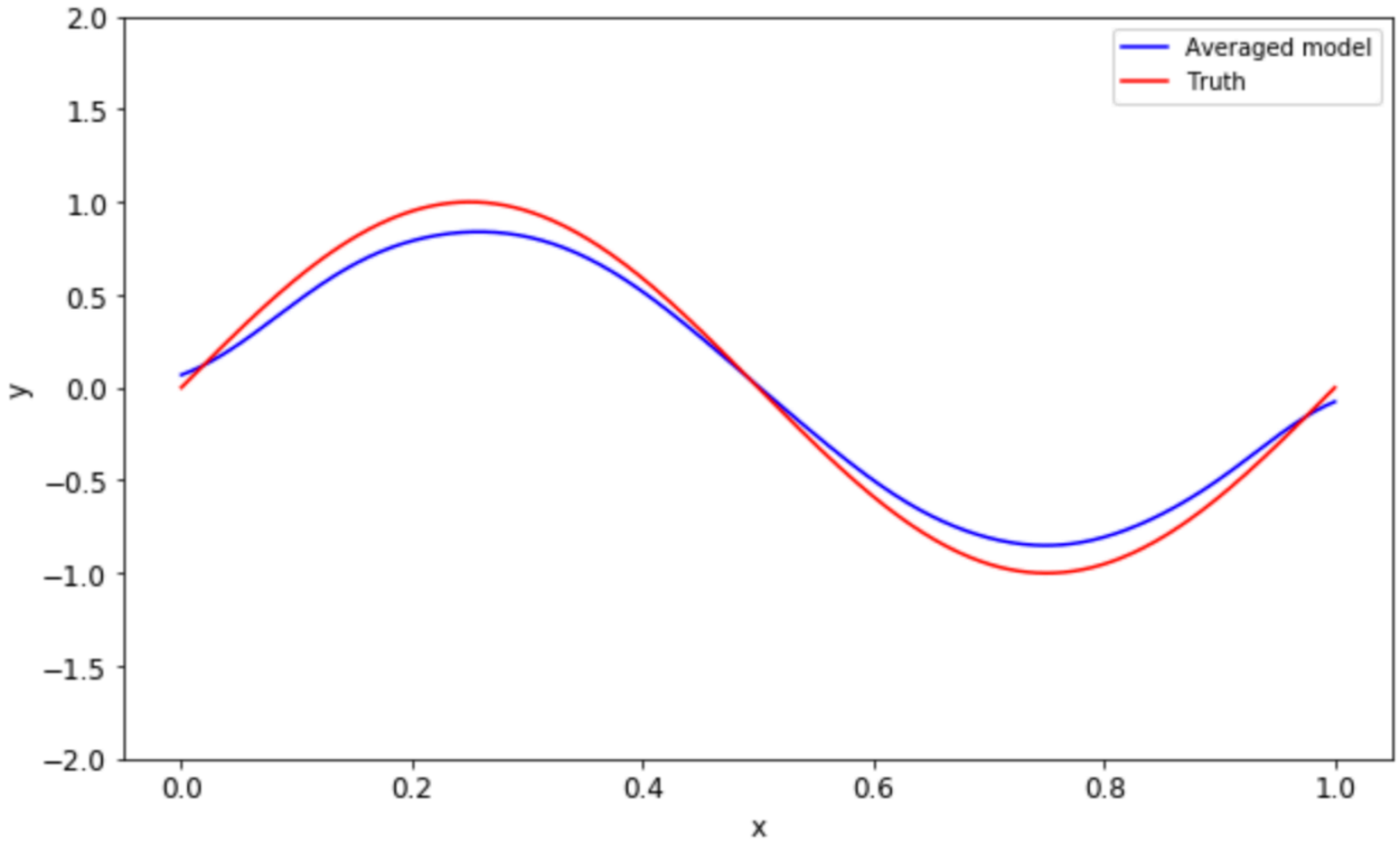
Gaussian-basis function regression

(see notebook under Lectures/Lecture 4/Notebooks/Gaussian basis function regression.ipynb)

Create 100 datasets with 10 points each, fit these with 10 Gaussians with fixed width. Thus easily affected by overfitting, but low bias.



Combining it all



So averaging the results of the fits gives a better final result, less sensitive to overfitting.

Can this be generalised? We usually do not have N indepedent samples…

Taking a step - back, a summary

Taking a step - back, a summary

I have a good idea where my data come from

Taking a step - back, a summary

I have a good idea where my data come from

non-linear fits (e.g. lmfit from last Programmers Club)
Bayesian models

Taking a step - back, a summary

I have a good idea where my data come from

non-linear fits (e.g. lmfit from last Programmers Club)
Bayesian models

My data seems to have a lot of collinearity & I have a lot of data.

Taking a step - back, a summary

I have a good idea where my data come from

non-linear fits (e.g. lmfit from last Programmers Club)
Bayesian models

My data seems to have a lot of collinearity & I have a lot of data.

regular linear regression, basis function regression?

Taking a step - back, a summary

I have a good idea where my data come from

non-linear fits (e.g. lmfit from last Programmers Club)
Bayesian models

My data seems to have a lot of collinearity & I have a lot of data.

regular linear regression, basis function regression?

I haven't got a lot of data and I would like to interpret the fitted model

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression
My data has a lot of local structure, complex shape, and uncertainties…	

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression
My data has a lot of local structure, complex shape, and uncertainties…	local regression, kernel regression, **Gaussian process regression**

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression
My data has a lot of local structure, complex shape, and uncertainties…	local regression, kernel regression, **Gaussian process regression**
I have a lot of outliers in my data…	

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression
My data has a lot of local structure, complex shape, and uncertainties…	local regression, kernel regression, **Gaussian process regression**
I have a lot of outliers in my data…	robust regression, Bayesian outlier detection

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression
My data has a lot of local structure, complex shape, and uncertainties…	local regression, kernel regression, **Gaussian process regression**
I have a lot of outliers in my data…	robust regression, Bayesian outlier detection
I have errors in all variables…	

Taking a step - back, a summary

I have a good idea where my data come from	non-linear fits (e.g. lmfit from last Programmers Club) Bayesian models
My data seems to have a lot of collinearity & I have a lot of data.	regular linear regression, basis function regression?
I haven't got a lot of data and I would like to interpret the fitted model	ridge & LASSO regression, maybe kernel regression
My data has a lot of local structure, complex shape, and uncertainties…	local regression, kernel regression, **Gaussian process regression**
I have a lot of outliers in my data…	robust regression, Bayesian outlier detection
I have errors in all variables…	Bayesian models, hyperfit (http://hyperfit.icrar.org/), linmix in IDL (Kelly 2007, ApJ).

Ensemble methods

Bagging

- Draw bootstrap realisations of your data.
- Fit these data.
- Average the result.

This is also known as Bootstrap aggregating.

It typically can help reduce overfitting problems.

In the case of perfectly uncorrelated errors in prediction, the error goes down as $\frac{\sigma}{\sqrt{N_{\mathrm{models}}}}$

If the errors are completely correlated, the error stays the same - ie. bagging has no effect.

Decision trees - an intermezzo

These are tree structures where at each level you split the data according to some criterion.

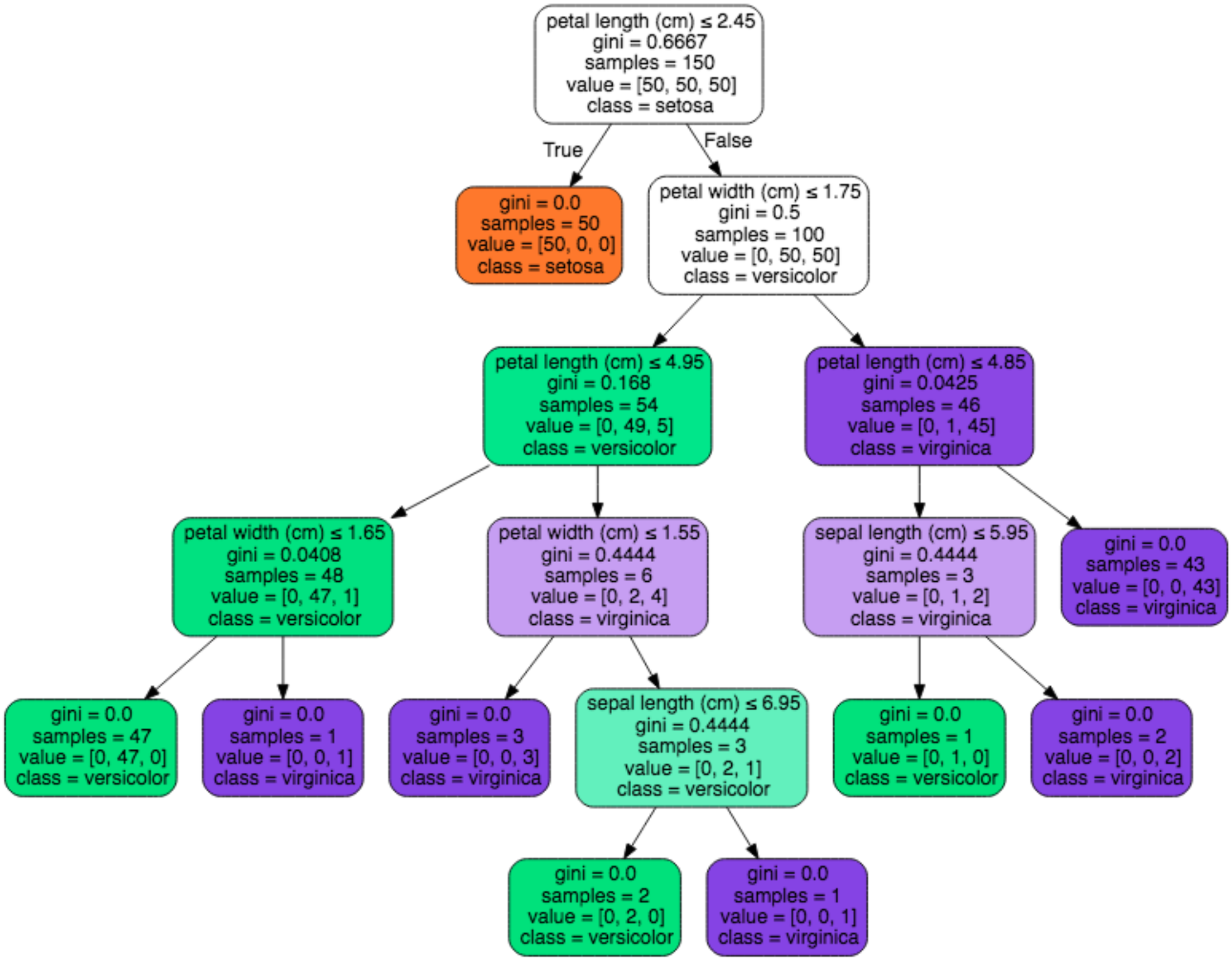
By making the tree deeper you can catch finer details in the data.

These are powerful techniques and can be inspected afterwards (good!) but do not usually have very high accuracy (not so good), and they have high variance (not ideal).

In sklearn:

```
from sklearn import tree
<get your data>
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, Y)
```

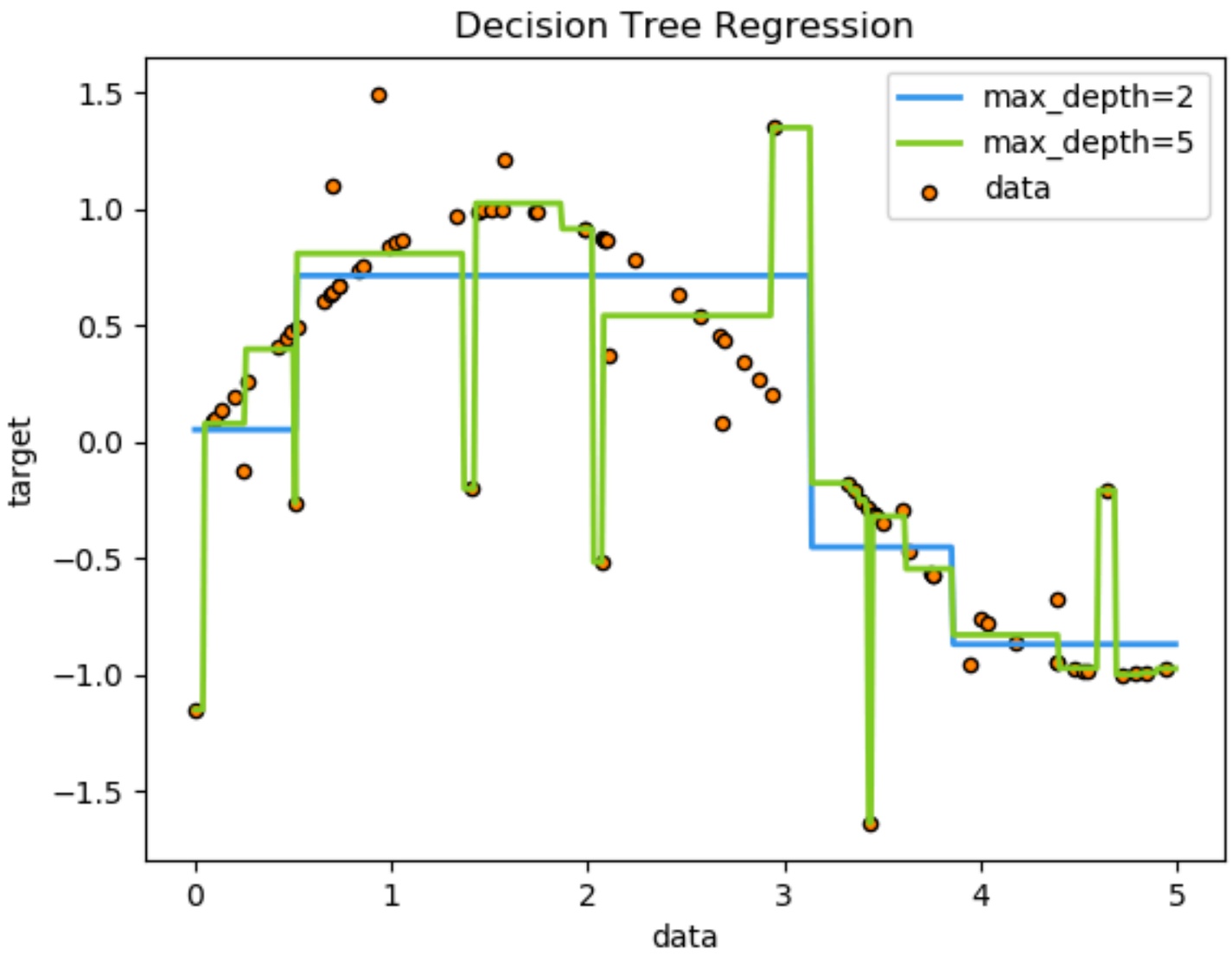
Decision trees - an intermezzo



http://scikit-learn.org/stable/modules/tree.html

Decision trees - an intermezzo

It can also be used for regression:



http://scikit-learn.org/stable/modules/tree.html

Random Forests

The bagging method can be applied to decision trees too. However if you combine bagging with randomly choosing a subset of features at each level, you get **random forests** and these seem to perform particularly well.

Basic algorithm (https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

```
from sklearn.ensemble import RandomForestClassifier
```

For each tree:

1. Sample N cases at random - but with replacement, from the original data
2. At each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.
3. Each tree is grown to the largest extent possible. There is no pruning.

The random subsets is to avoid correlation (recall the bagging error function)

Boosting

These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.

A weak learner typically is chosen to have a low bias, so as a consequence has high variance. The aim of boosting is to combine these to get a low bias, low variance estimator.

Boosting

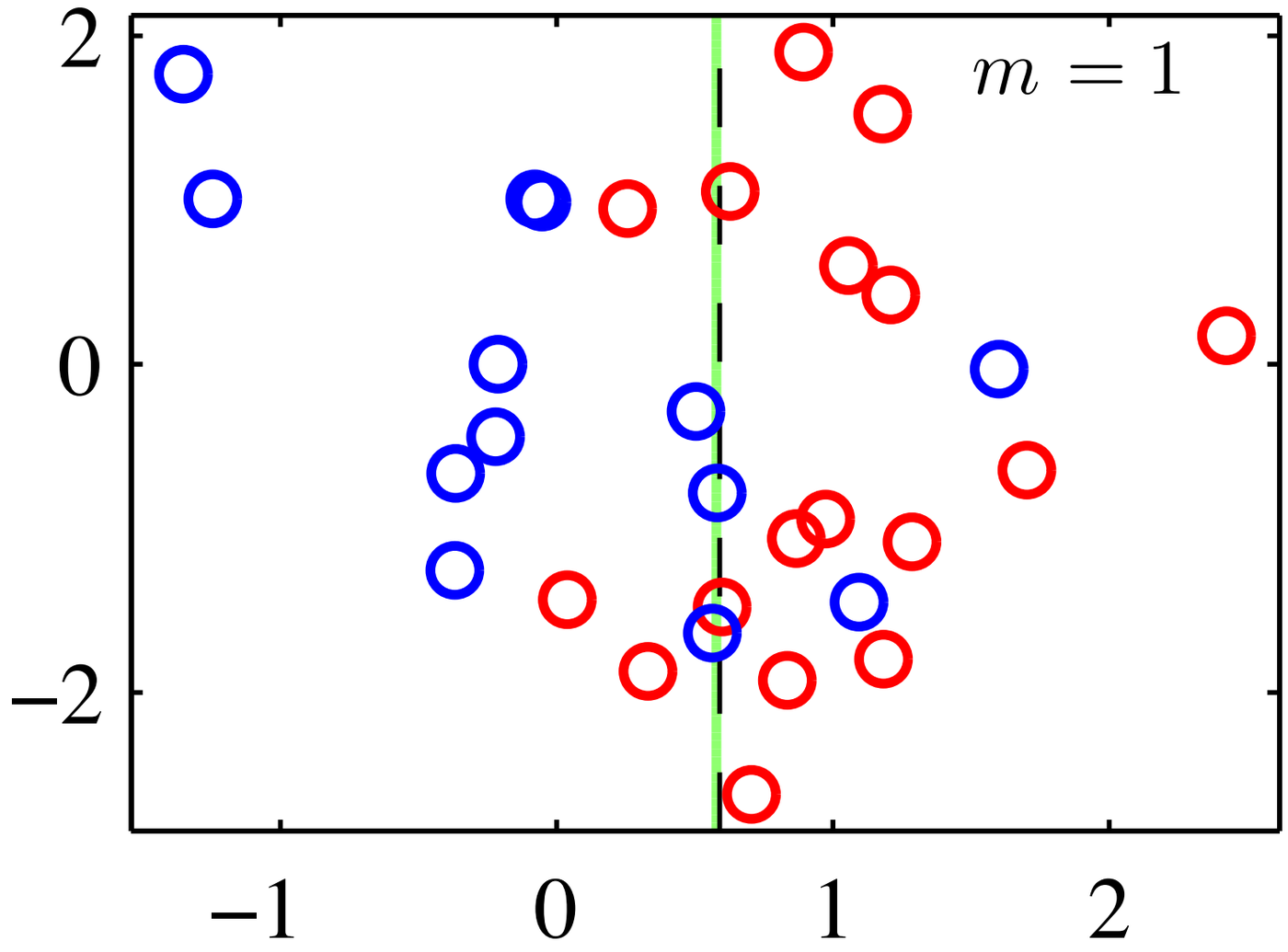
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.

The approach (AdaBoost):

1. Fit a learner (algorithm)
   Find how well this works and give high weight to those examples it did **not** fit.
   Repeat.
2. Average the results using weights estimated during the fitting procedure.

Boosting

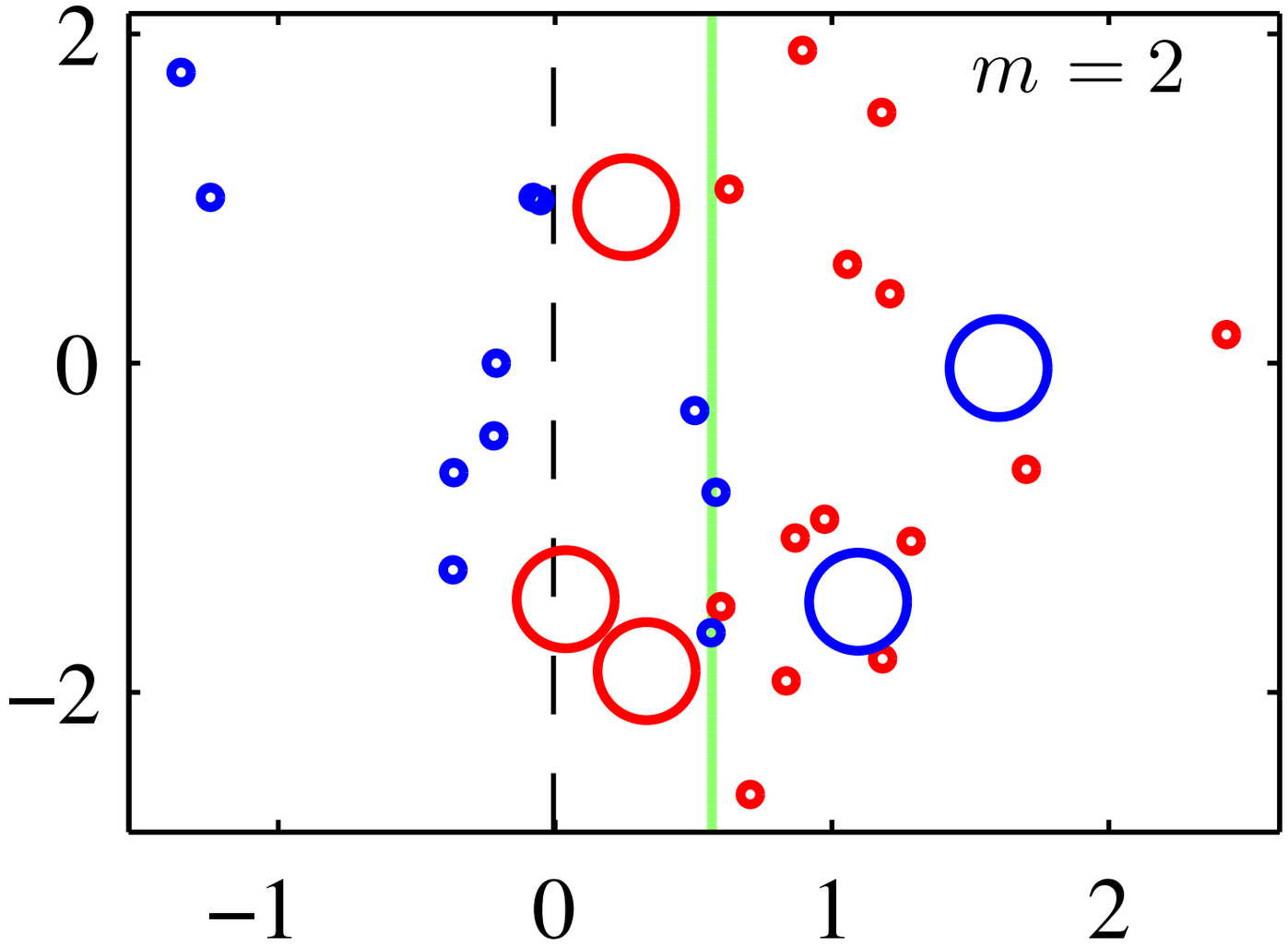
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

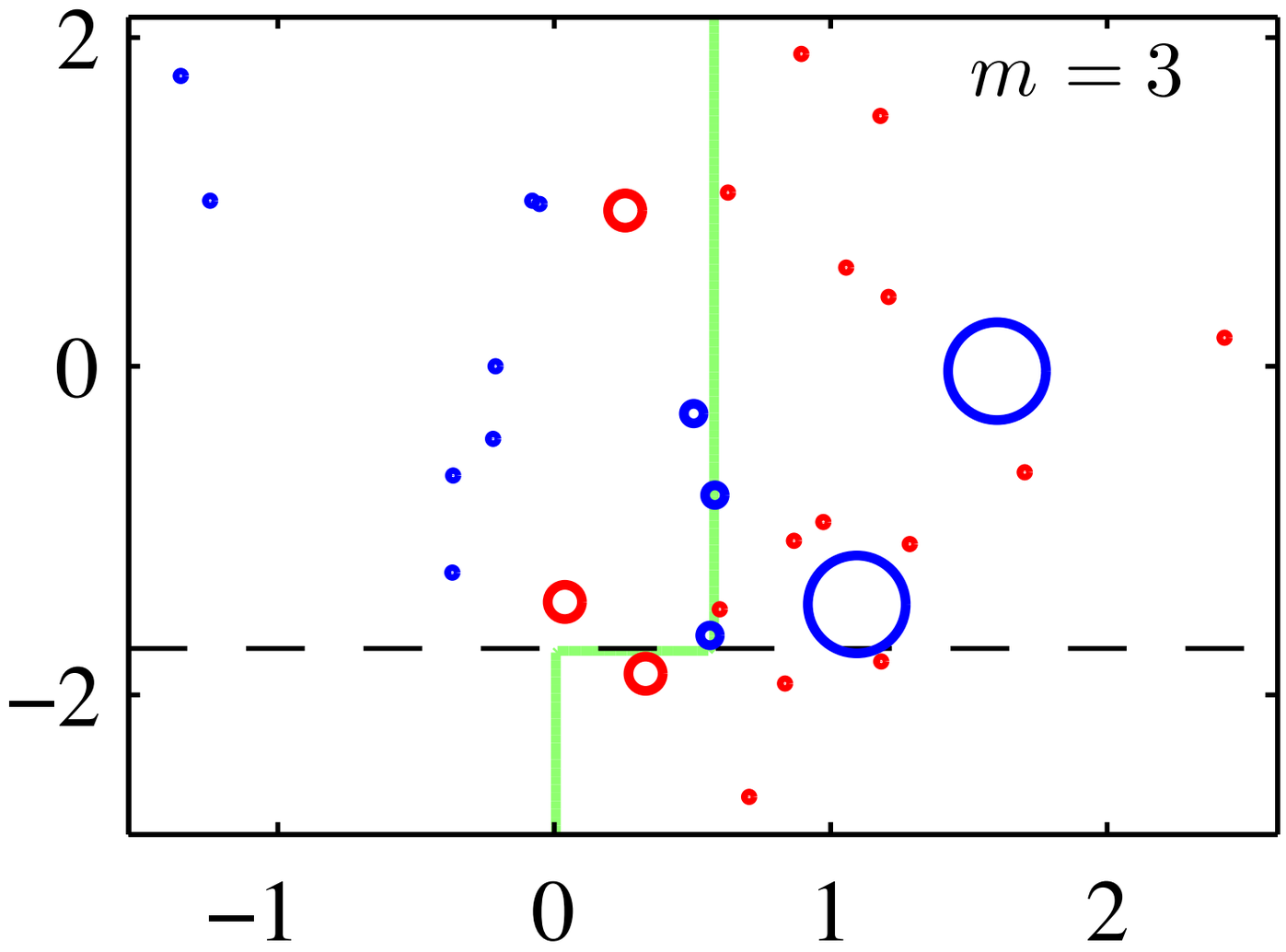
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

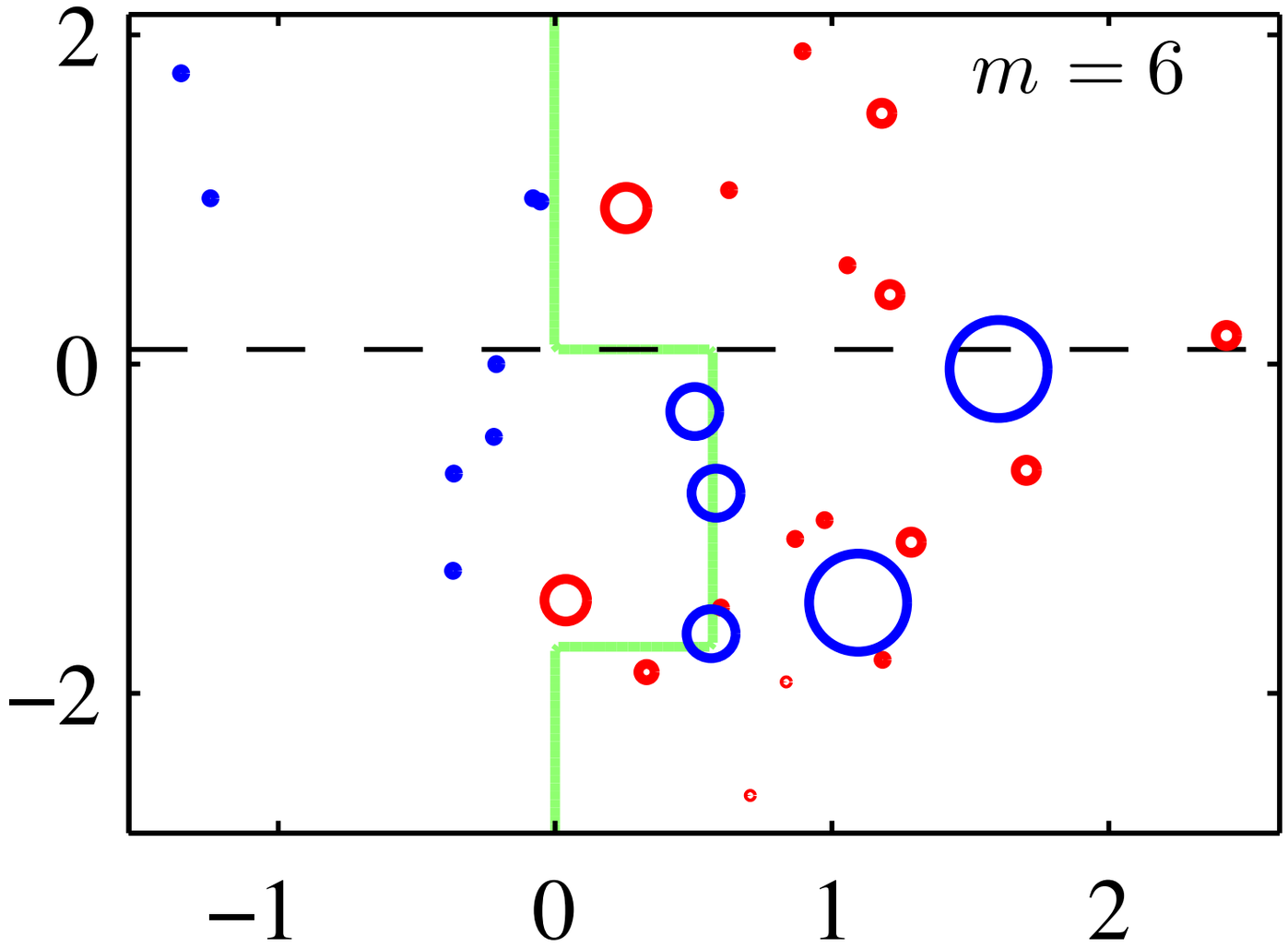
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

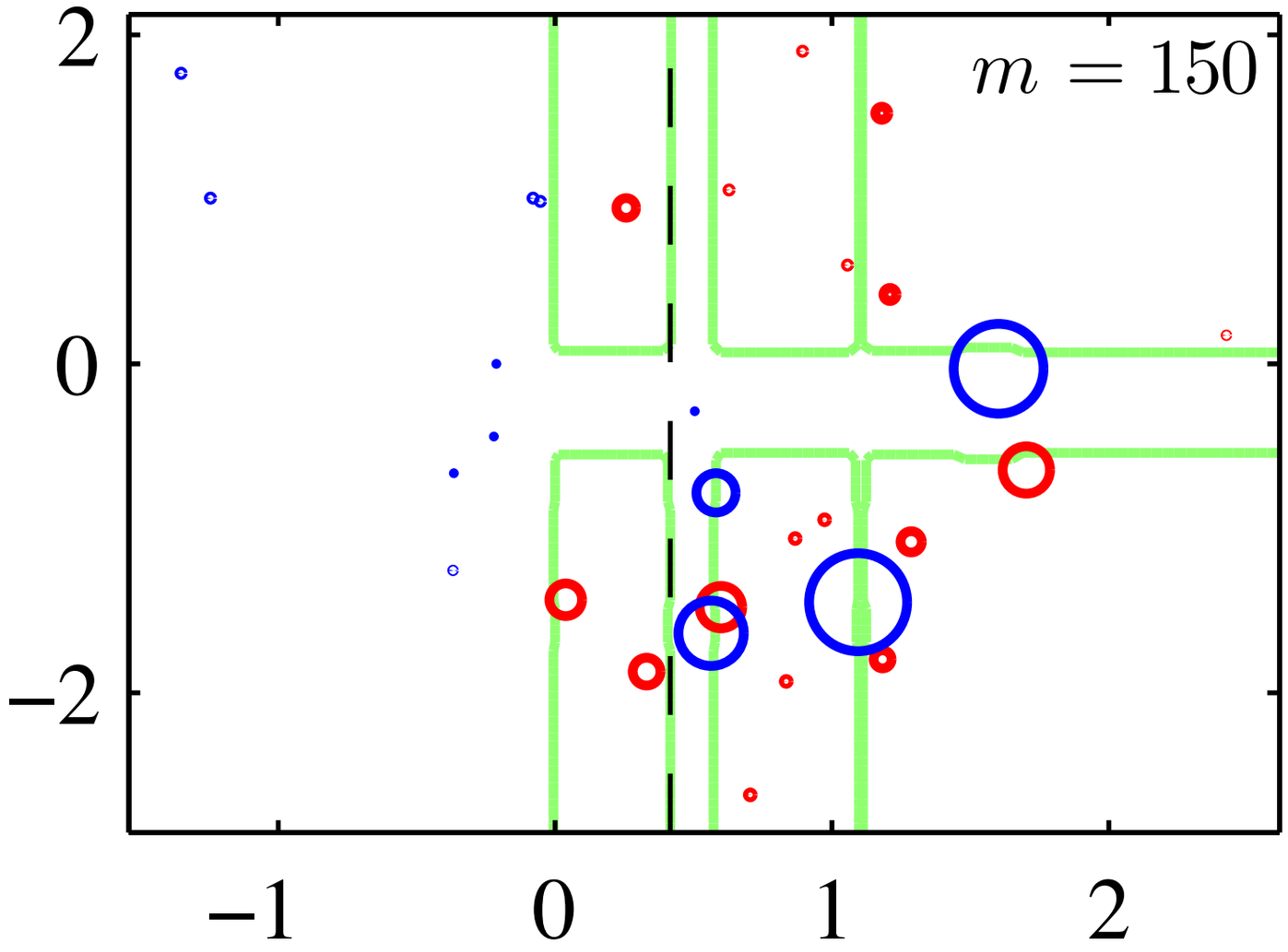
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

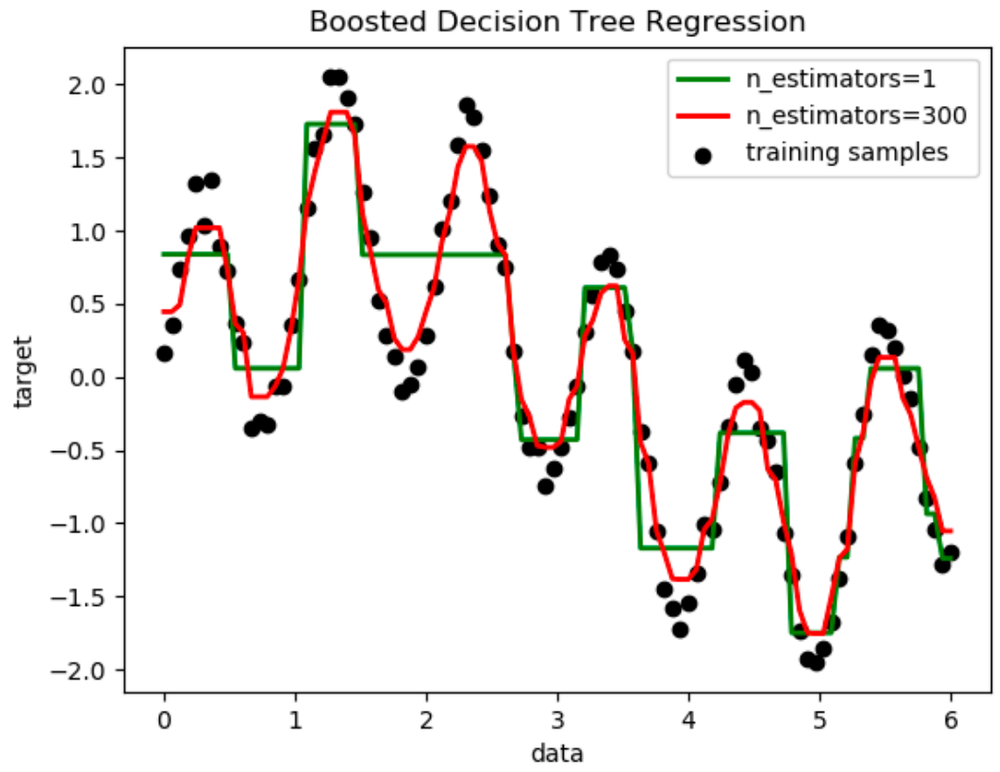
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



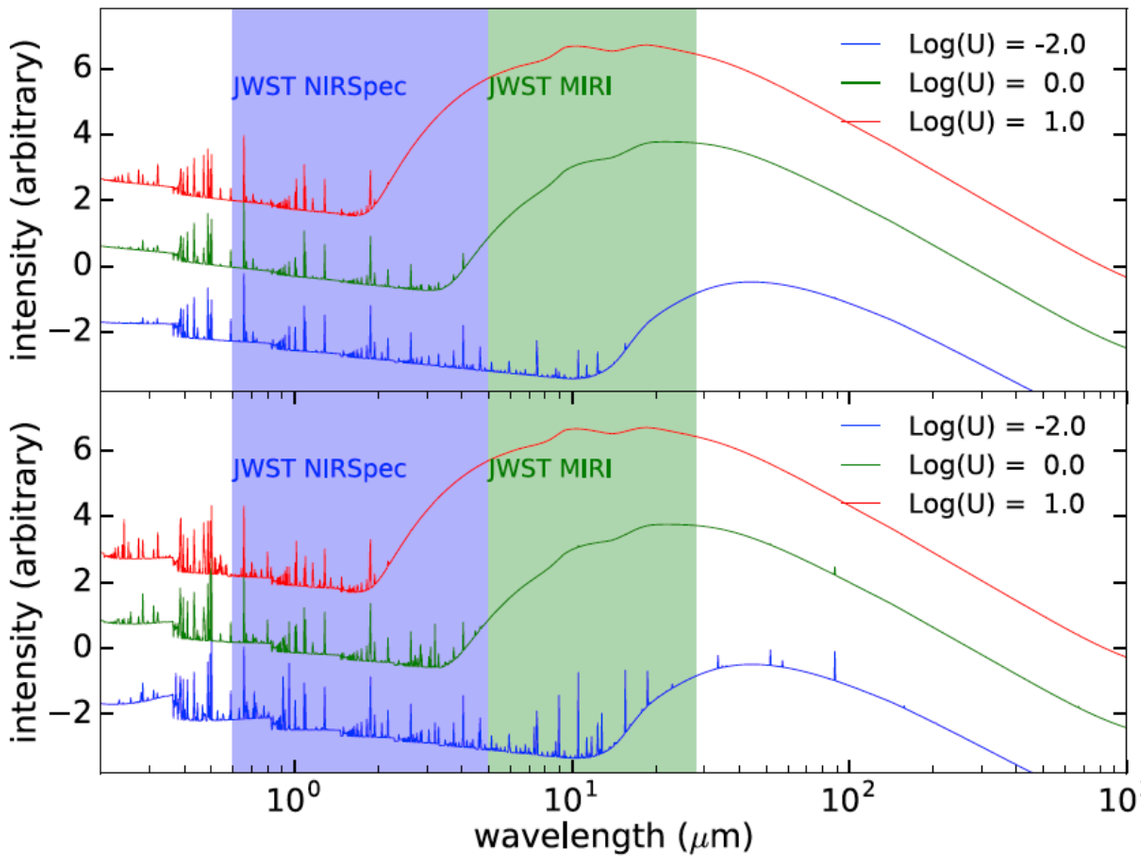

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
<…>
regr = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                         n_estimators=300)
regr.fit(X, y)
y_2 = regr.predict(X)
```

http://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_regression.htm

Use in astronomy

AdaBoost: Ucci et al (2017): Fitting emission line spectra (code GAME)

Input library to compare against:



Use in astronomy

AdaBoost:
Ucci et al (2017): Fitting emission line spectra.
Xin et al (2017): Finding impact craters on Mars.
Zitlau et al (2016): Photometric redshifts for SDSS.
++

Random Forests:
Kuntzer & Courbin (2017): Detecting binary stars.
García-Varela et al (2017): Finding variable stars.
Jouvel et al (2017): Photometric redshifts of galaxies.
Bastien et al (2017): Classification of radio galaxies
Torres et al (2019): White dwarfs in the MW from GAIA
Ulmer-Moll (2019): Exoplanet mass-radius relation
+++

Neural networks

Literature:
Goodfellow et al (2016), “Deep Learning”

MIT course (2023): http://introtodeeplearning.com/

(with excellent notebooks at https://github.com/aamini/introtodeeplearning/)

Types of tasks for a DL algorithm

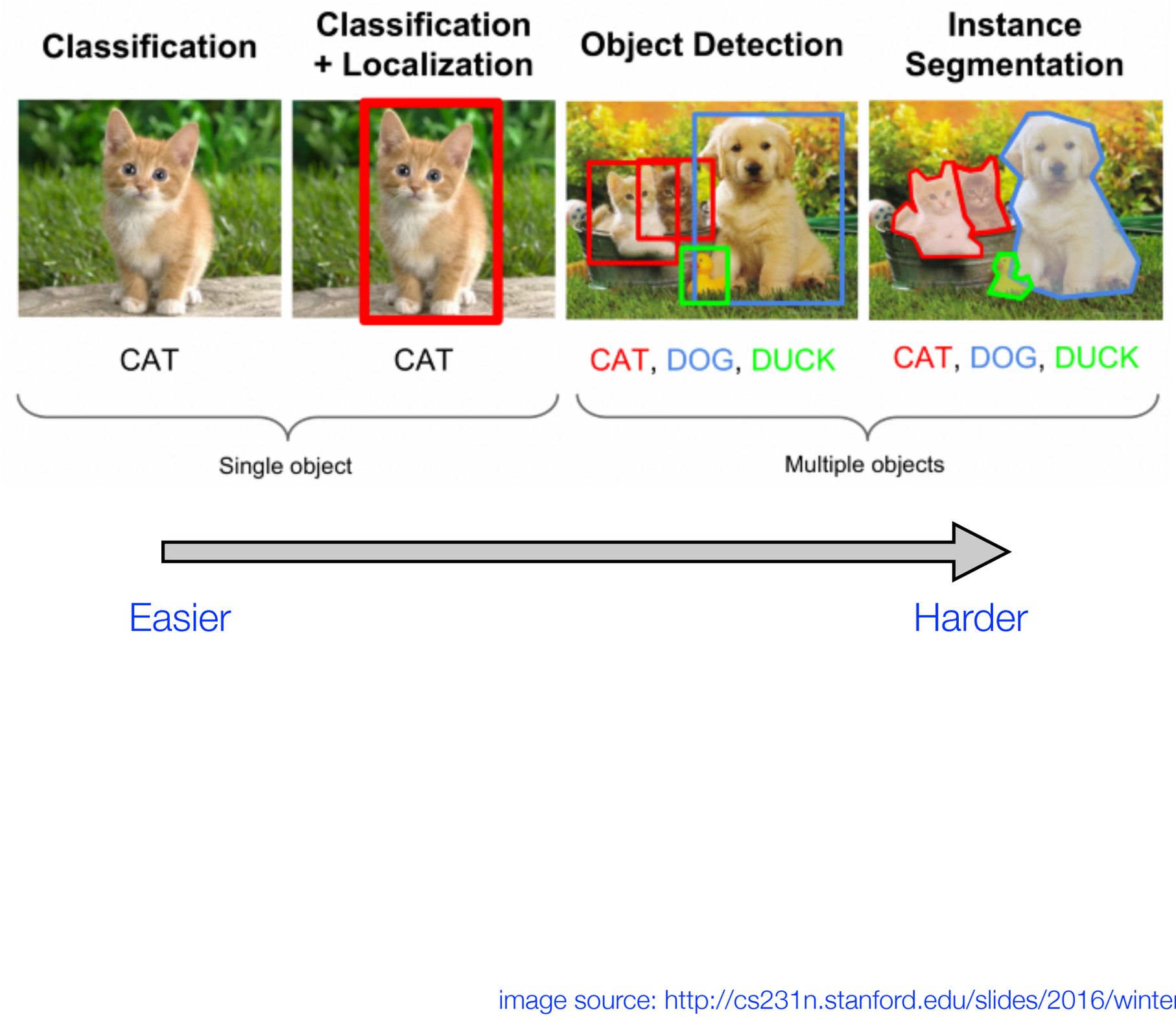
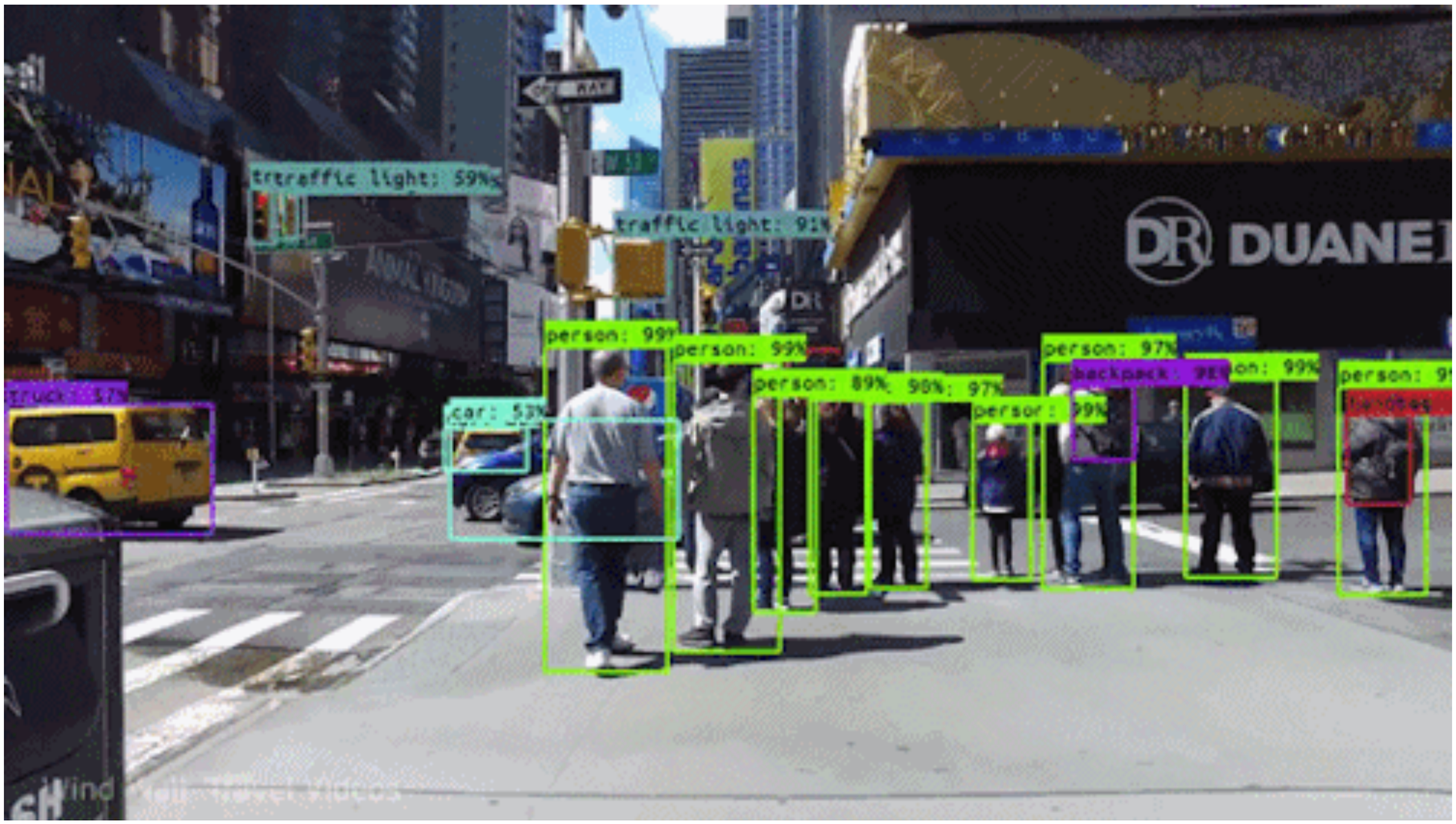


image source: http://cs231n.stanford.edu/slides/2016/winter1516_lecture8.pdf

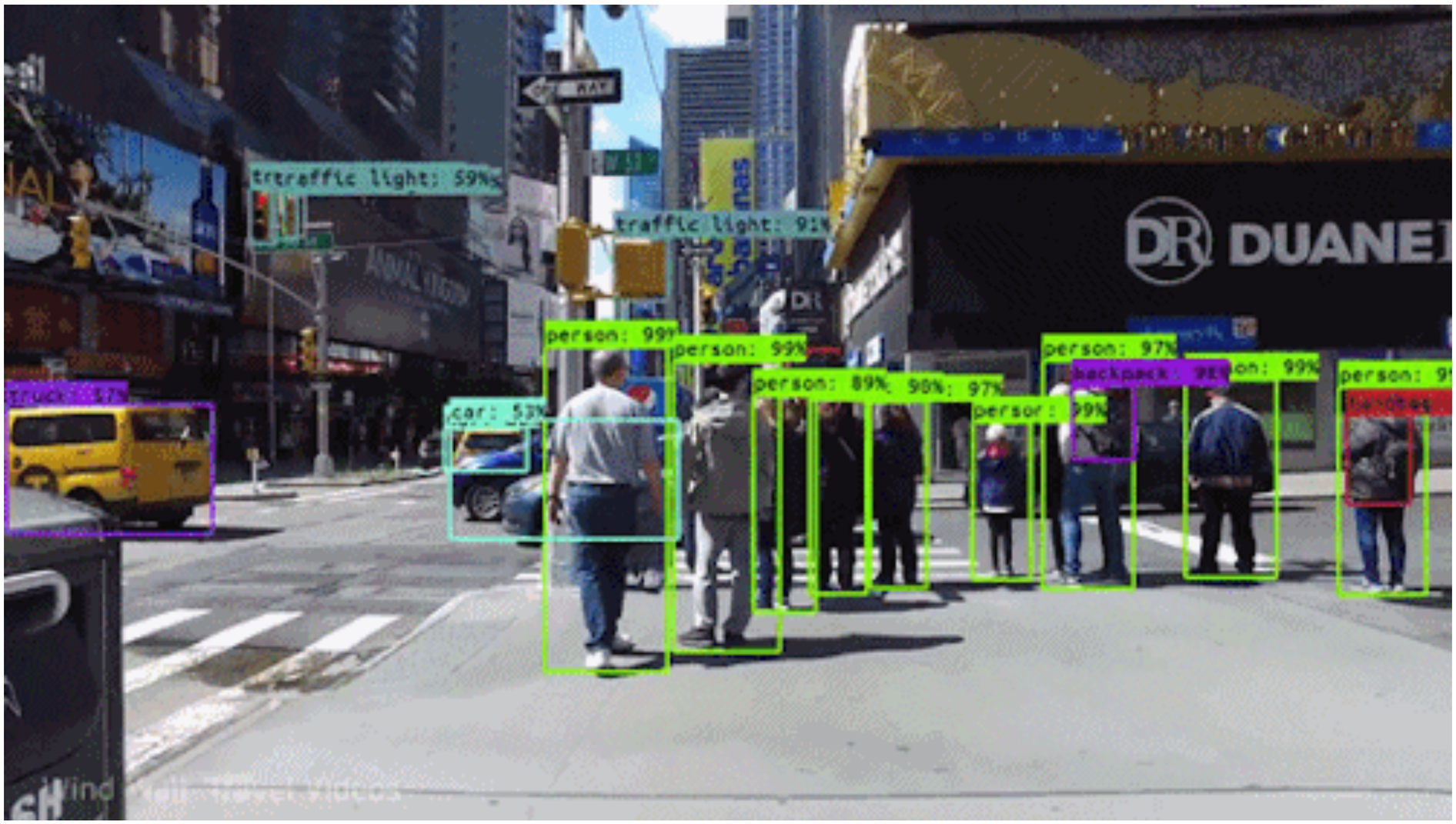
Deep learning - object detection



Accuracy matters… and it needs to be done quickly

https://storage.googleapis.com/openimages/web/index.html

Deep learning - object detection



Accuracy matters… and it needs to be done quickly

https://storage.googleapis.com/openimages/web/index.html

Deep learning - what do we want?

What do we want?

$$P(\text{object} \mid \text{image})$$

The neural network will be our model and we need to train this using N examples. Since the dimension of the data can be large, this can be a very high-D PDF.

The trick then is to efficiently train this.

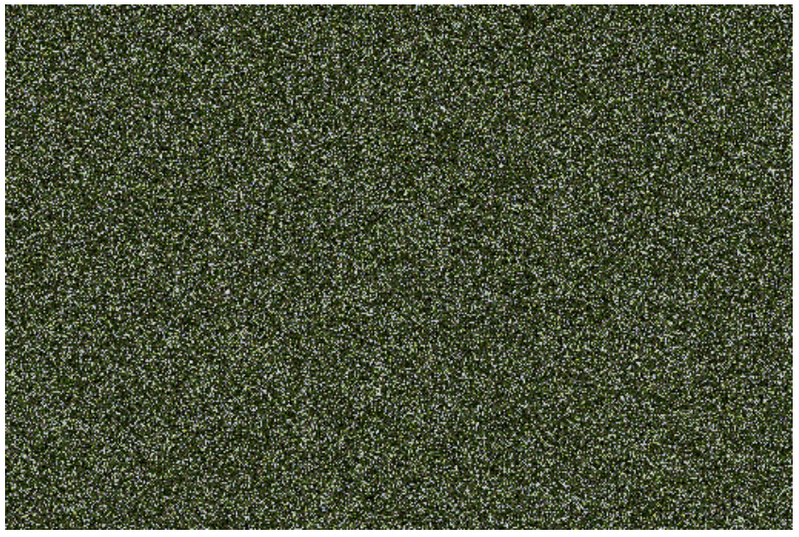
Manifold learning & deep learning

It seems that real images/data do not fully span this high-D space though:

D= 174592

Random resampling

D= 174592



Instead real images span only a small fraction of the full space - typically viewed as a manifold where images change smoothly.

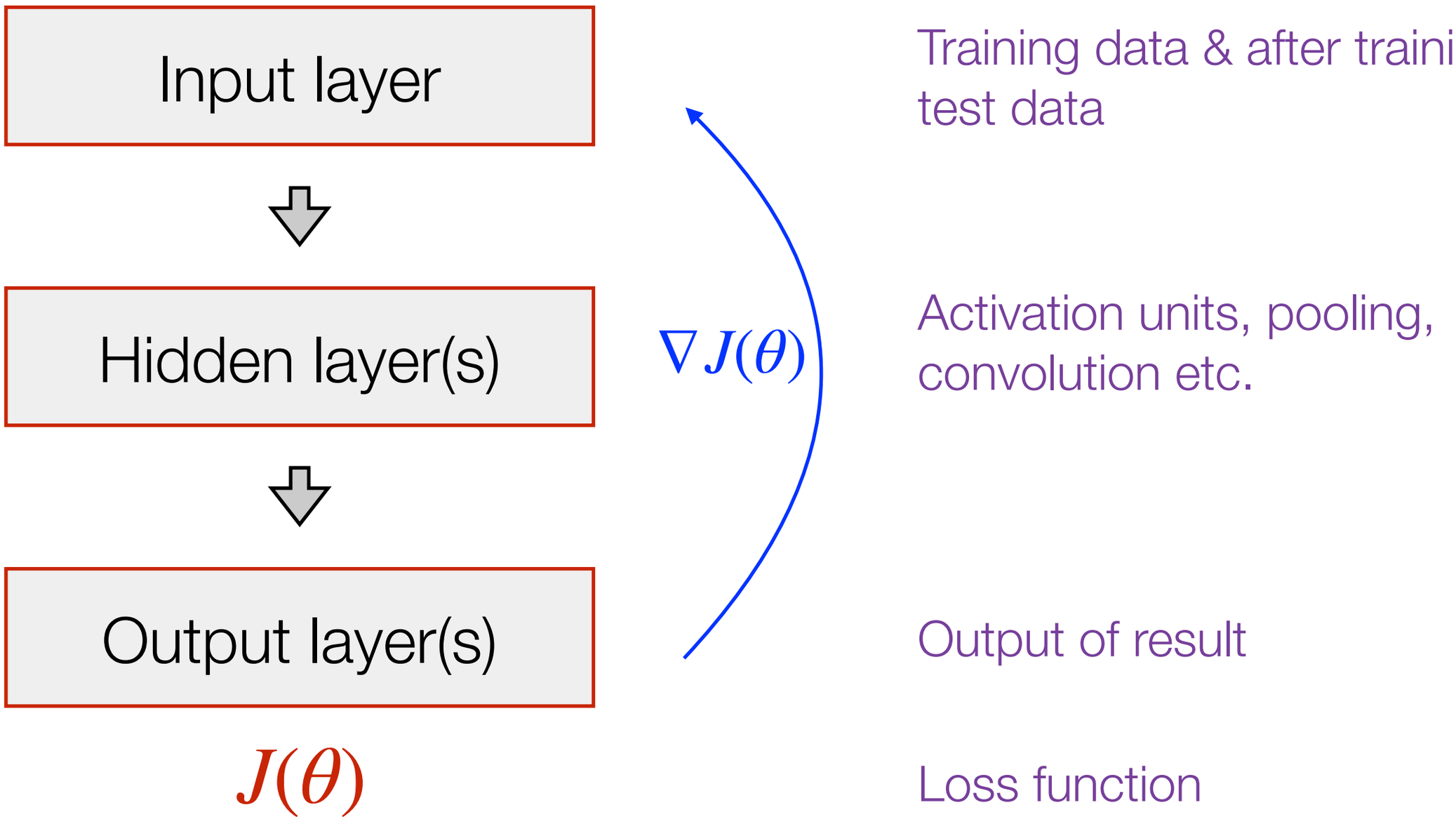
Manifold learning & deep learning

Instead real images span only a small fraction of the full space - typically viewed as a manifold where images change smoothly.

QMUL Multiview Face Dataset - http://www.eecs.qmul.ac.uk/~sgg/QMUL_FaceDataset/

Learning the PDF

The challenge is learning this structure - in deep learning we use a (neural) network (feed-forward network)

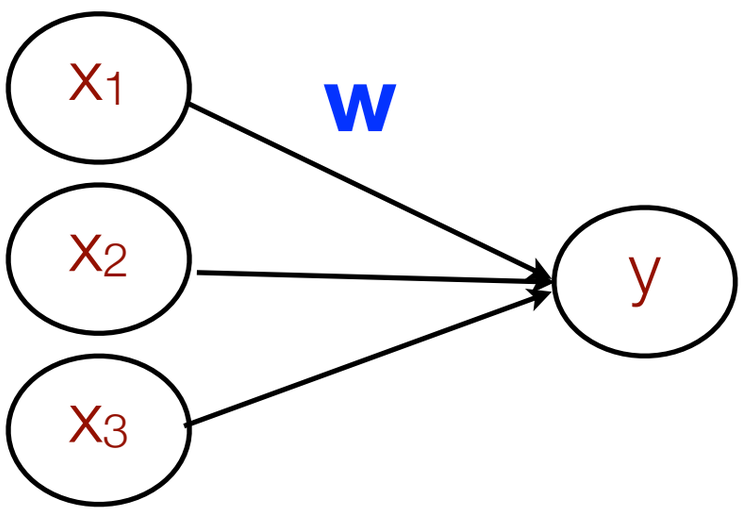


Linear Models - on our way to non-linearity

We can predict a value of a parameter y using a linear combination of an **x** vector & a set of basis functions $\boldsymbol{\phi}_i$:

$$y(\mathbf{x}, \mathbf{w}) = \sum_i w_i \phi_i(\mathbf{x})$$

This then represents a *transformation* of the input data **x**. And we can represent this as:



(if our x has 3 elements)

The simplest Neural Network

Now expand our previous method and create M different linear combinations of **x**, ie. create M different weight vectors $\mathbf{w}_j$:

$$z_j = h\left(\sum_i w_{i,j} x_i\right)$$

The h function is called the **activation function** in neural network contexts.

Activation functions

What function can we use?

Simplest: linear, but then the whole system is linear!

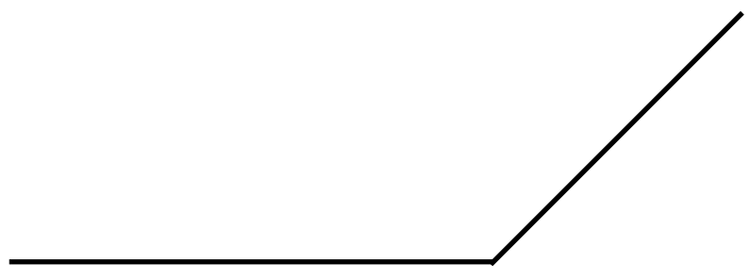
Activation functions

What function can we use?

Simplest: linear, but then the whole system is linear!

So it must be non-linear, but we would like it to behave much like a linear function. The modern choice is the rectified linear unit:

ReLU: $h(z) = \max\{0, z\}$

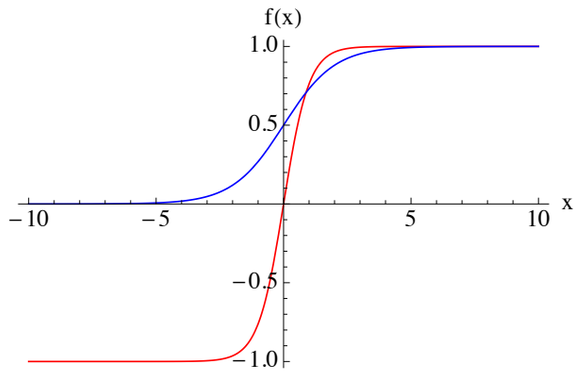


Activation functions

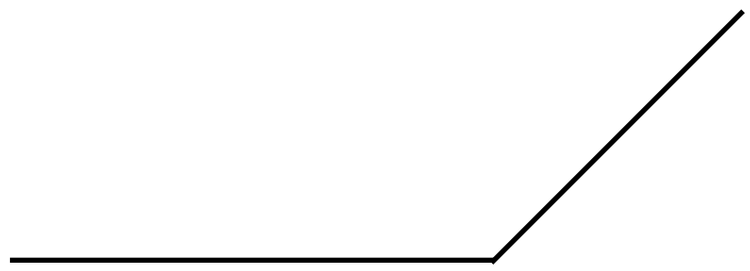
Why this form?

The issue is training: to train we need to calculate gradients

For this to be efficient, we prefer functions whose gradients are well away from zero.



Less good - saturates

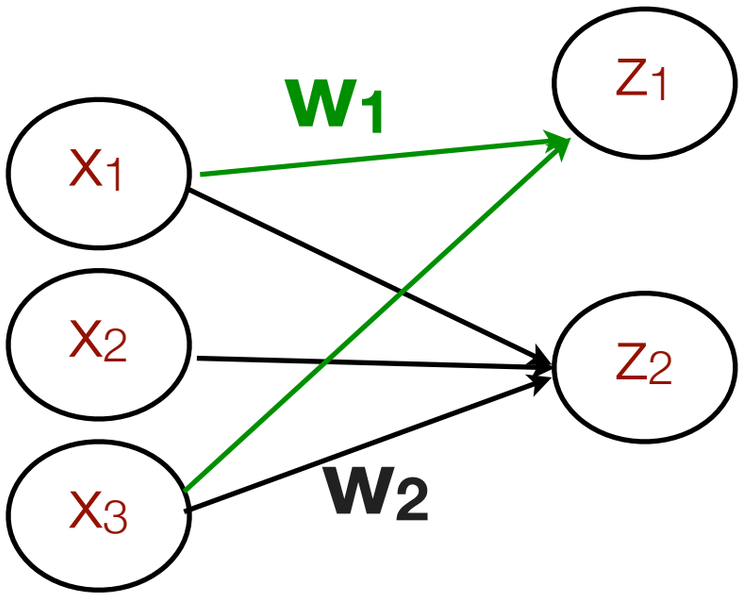


Good with start ≠ 0

The simplest Neural Network

Now expand our previous method and create M different linear combinations of **x**, ie. create M different weight vectors $\mathbf{w}_i$:

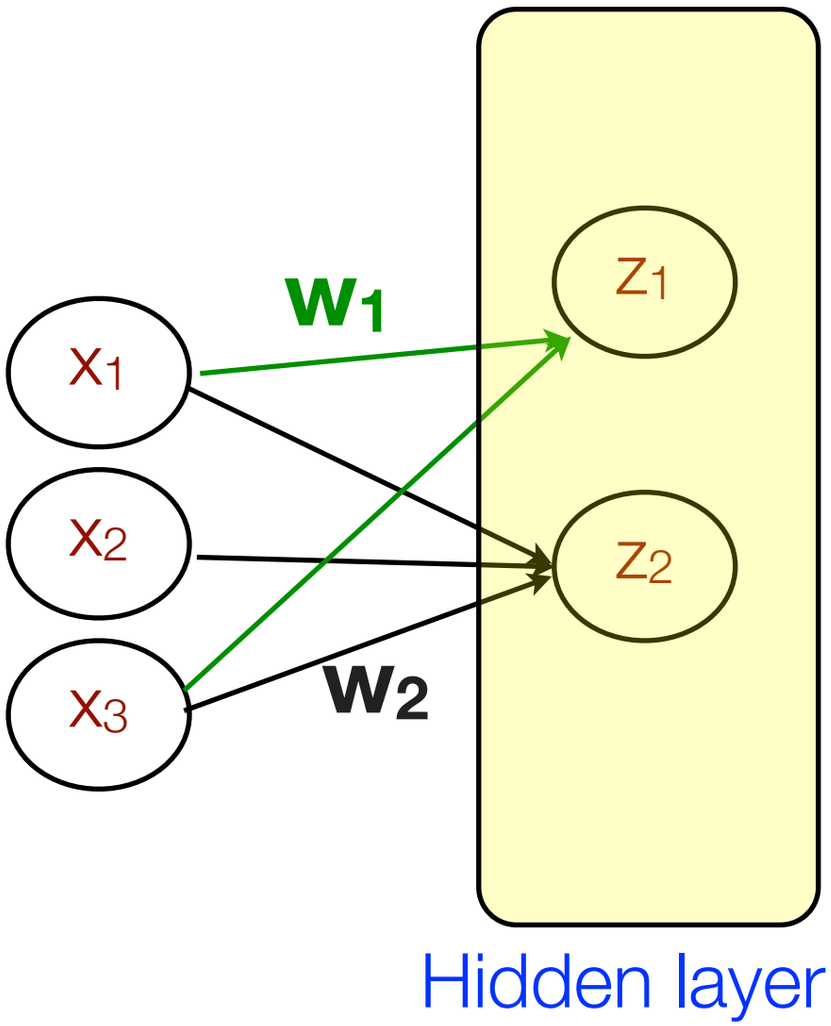
$$z_j = h\left(\sum_i w_{i,j} x_i\right)$$



The simplest Neural Network

Now expand our previous method and create M different linear combinations of **x**, ie. create M different weight vectors $\mathbf{w}_j$:

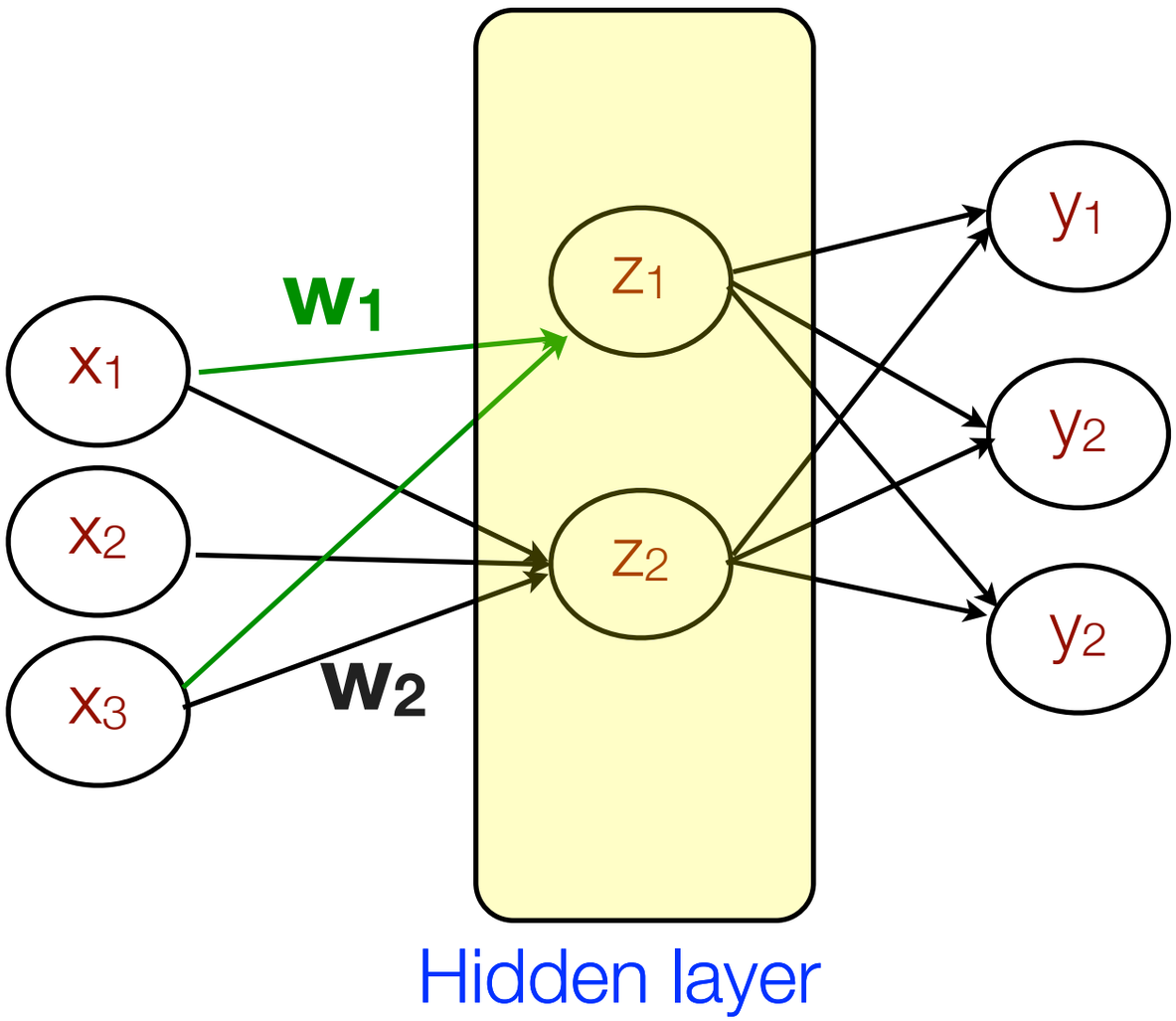
$$z_j = h\left(\sum_i w_{i,j} x_i\right)$$



The simplest Neural Network

Then we want to create one more layer - this is the *output* layer and consists of D outputs.

$$y_k = f\left(\sum_i v_{i,j} z_i\right)$$



The simplest Neural Network

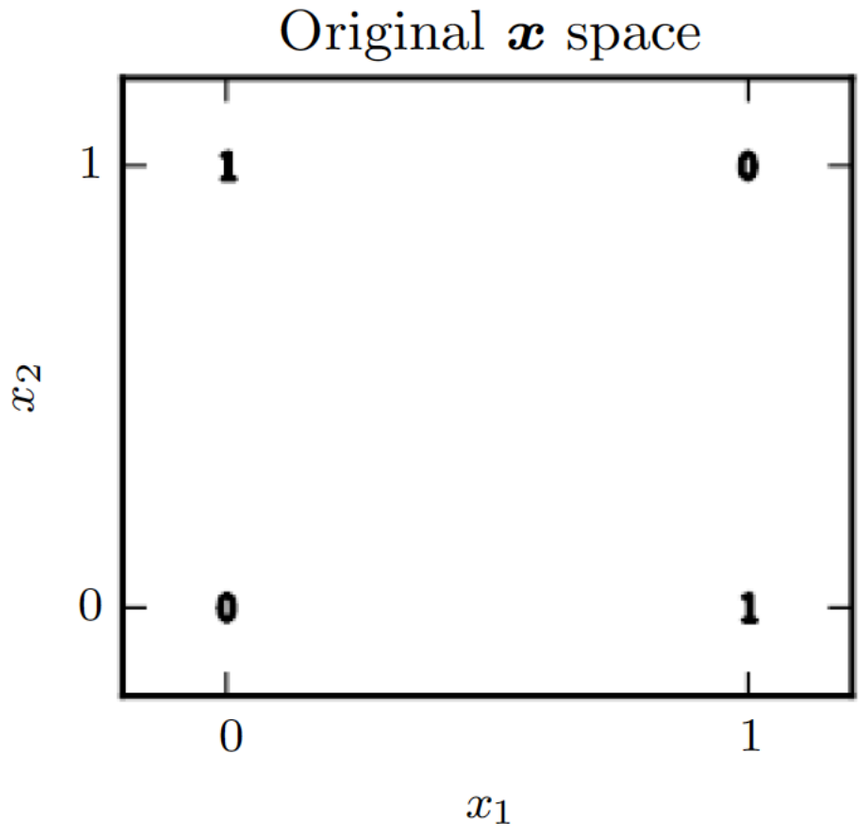
The full equation becomes:

$$y_k(\mathbf{x}, \mathbf{w}, \mathbf{v}) = f\left\{\sum_{j=1}^{M} v_{k,j}\ h\left(\sum_{i=1}^{N} w_{j,i} x_i\right)\right\}$$

For a set of inputs and a choice of weights - this can then be used to calculate y values easily. **The challenge is to determine the right choice of weights.** We will return to this when discussing deep learning.

Learning XOR

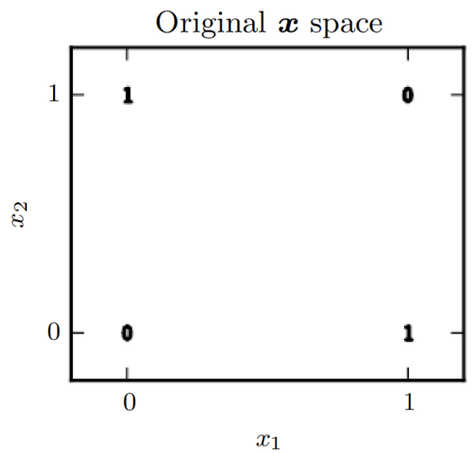
$$XOR(x, y) = \begin{array}{l} 1 \text{ if either } x = 1 \text{ or } y = 1, \text{ but not both} \\ 0 \text{ otherwise} \end{array}$$



We want to learn a function of x to fit this - obviously a linear function won't work!

Following Goodfellow et al, section 6.1

Learning XOR



weights: $\mathbf{w}$

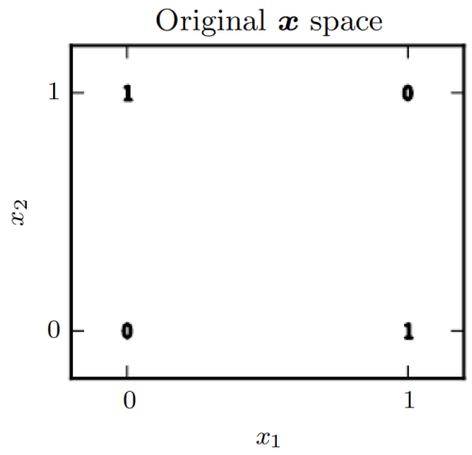
first layer: $\mathbf{h} = g\left(\mathbf{W}^T\mathbf{x} + \mathbf{c}\right)$

Using a ReLU:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\left\{0, \mathbf{W}^T\mathbf{x} + \mathbf{c}\right\} + b$$

Following Goodfellow et al, section 6.1

Learning XOR



$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\left\{0, \mathbf{W}^T\mathbf{x} + \mathbf{c}\right\} + b$$

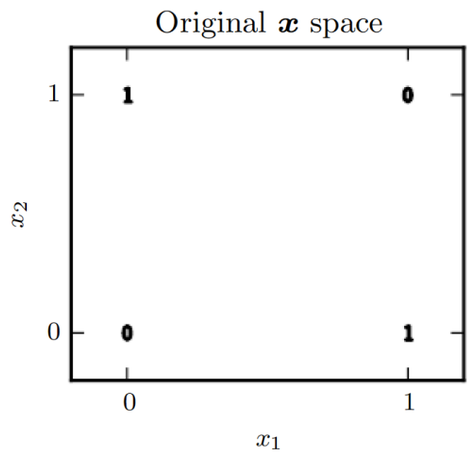
Let: $\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ $\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ $\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$

Data: $\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$ and: $\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

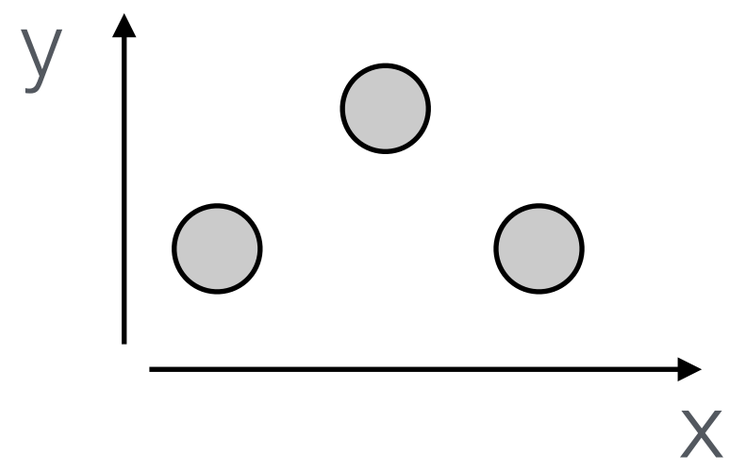
Following Goodfellow et al, section 6.1

Learning XOR

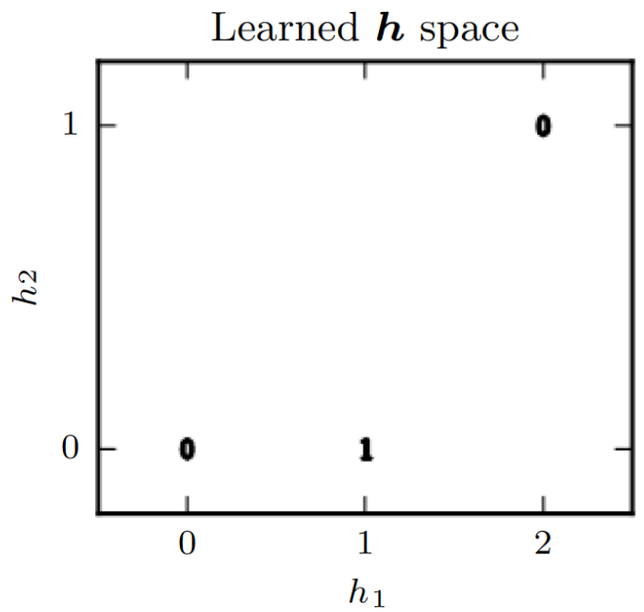


$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\left\{0, \mathbf{W}^T \mathbf{x} + \mathbf{c}\right\} + b$$

$$\mathbf{XW} + \mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



$$\max\{0, \mathbf{XW} + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



Following Goodfellow et al, section 6.1

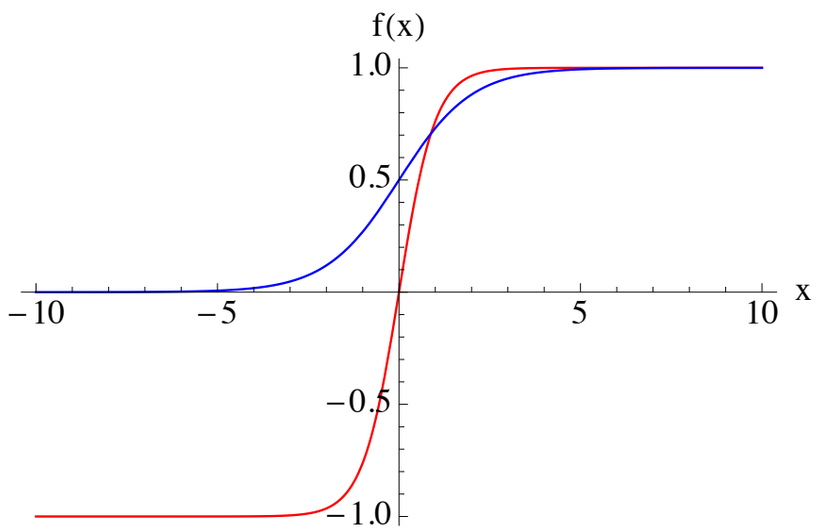
Output units - sigmoids

For classification cases, it is useful to have a function which takes -∞ to +∞ and maps it to -1 to 1 or 0 to 1. These are known as sigmoid functions. They have the advantage that they keep the signals from going off towards infinity and are useful as output units (in the past they were used as activation units too).

There are of course many of these, the most popular in our context are:

$$f(x) = \tanh(x)$$

Logistic: $$f(x) = \frac{1}{1 + e^{-x}}$$



Output units - linear functions

Another case is when you have a regression problem and want to produce a continuous output. In that case we use linear output units

Loss functions

MSE:

$$J(\theta) \propto \sum_{\mathbf{x}} \left(f^{\text{true}}(\mathbf{x}) - f(\mathbf{x}, \theta) \right)^2$$

Common earlier and sometimes used still but for many output units (sigmoids or softmax) this can lead to very bad performance.

Maximum likelihood/ cross-entropy

$$J(\theta) = -\ln \overline{p(\mathbf{y} \mid \mathbf{x}; \theta)}$$

The best general option - leads to MSE for Gaussian PDFs. The main advantage is that it undoes the exponentials in many PDFs.

What do you use as input?

You can give a full spectrum if you wish, or an image, but it could become time-consuming (but is what is typically used in deep learning) - can you do something else?

What do you use as input?

You can give a full spectrum if you wish, or an image, but it could become time-consuming (but is what is typically used in deep learning) - can you do something else?

You can instead provide a (judiciously chosen) set of measurements that summarise a particular image/ spectrum - **features**.

One possibility is to use **PCA** to select features - use these to reduce the dimension of the problem and input the PCA components as input variables.

You can also provide “Observables” - but make sure you don’t provide the same information many times!

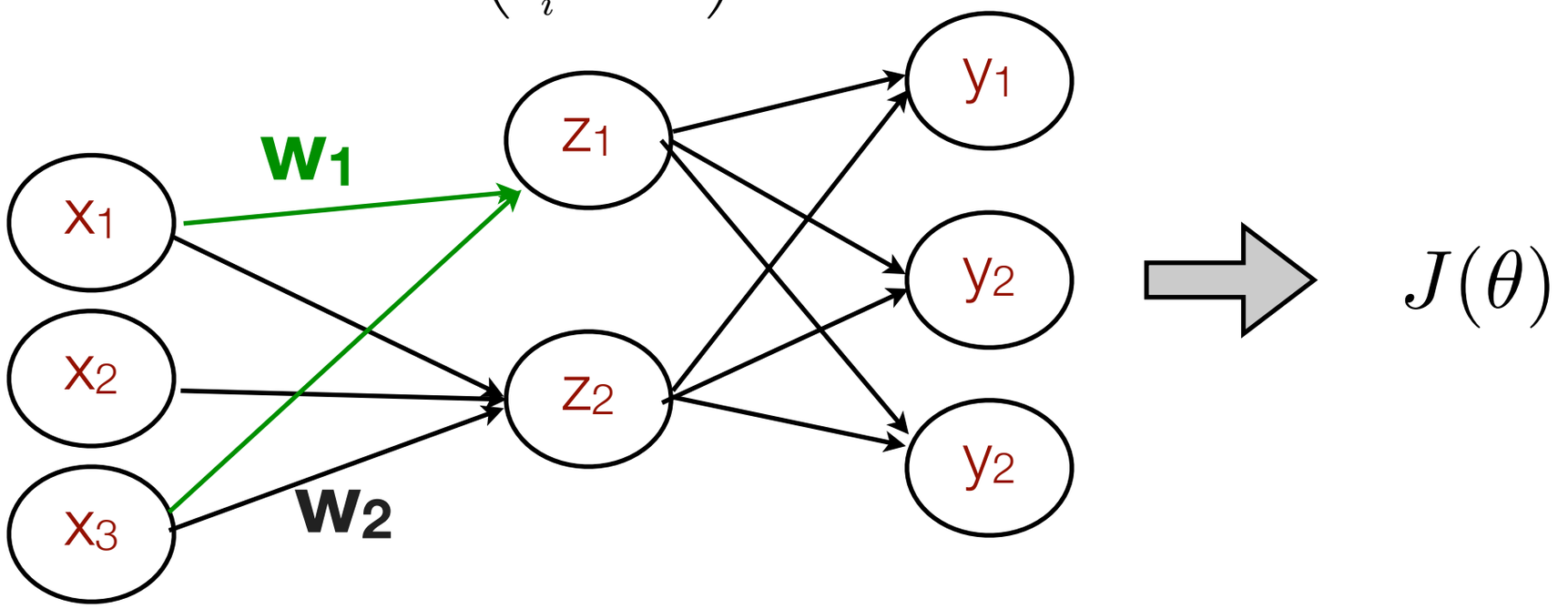
Forward- and back-propagation

Input

$$y_k = f\left(\sum_i v_{i,j} z_i\right)$$

Output

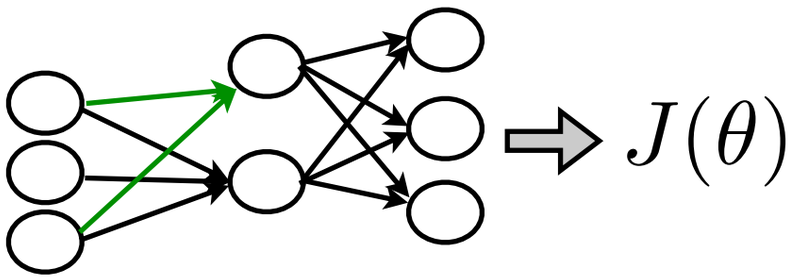
Loss function



Going from left to right is forward propagation

In this case, because there are no loops (cycles), the network is also a feed-forward network.

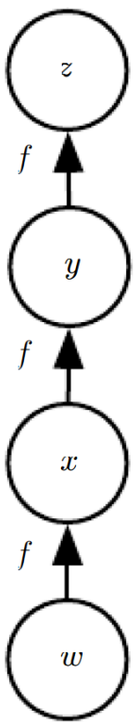
Forward- and back-propagation



But there will be errors made in the estimate.
How do we improve?

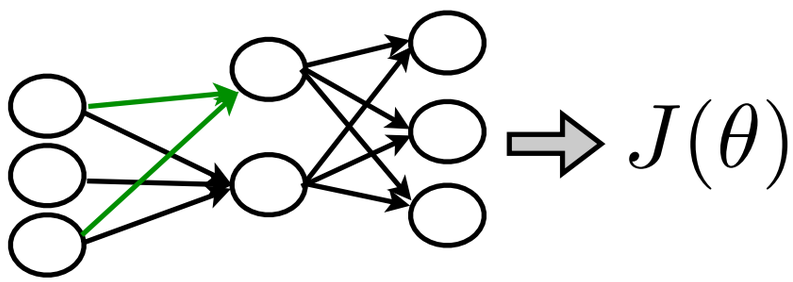
By going backwards!

We want to use gradient descent to update weights,
but for that we need the gradients:



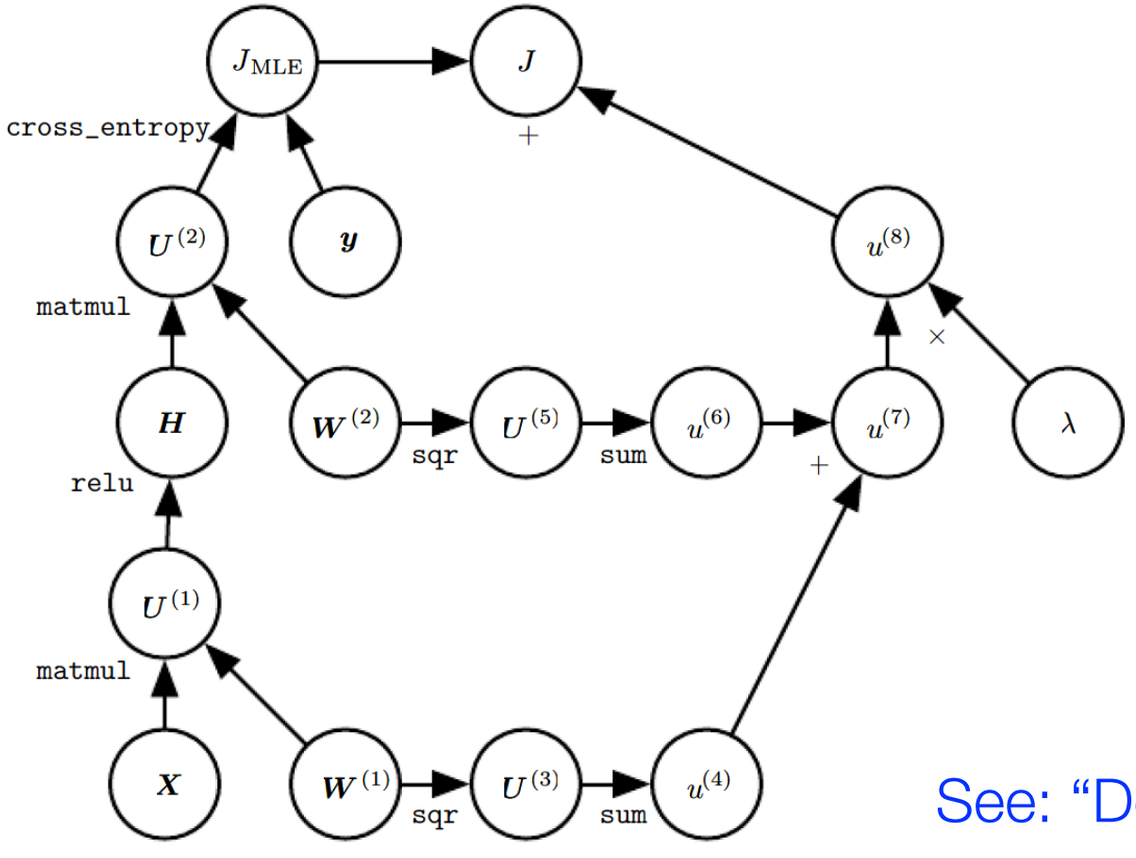
$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

Forward- and back-propagation



But there will be errors made in the estimate.
How do we improve?

The challenge is in slightly more complex situations:



Back-propagation is a clever algorithm to calculate these gradients efficiently and automatically.

See: “Deep learning”, Goodfellow et al (2016) for details.

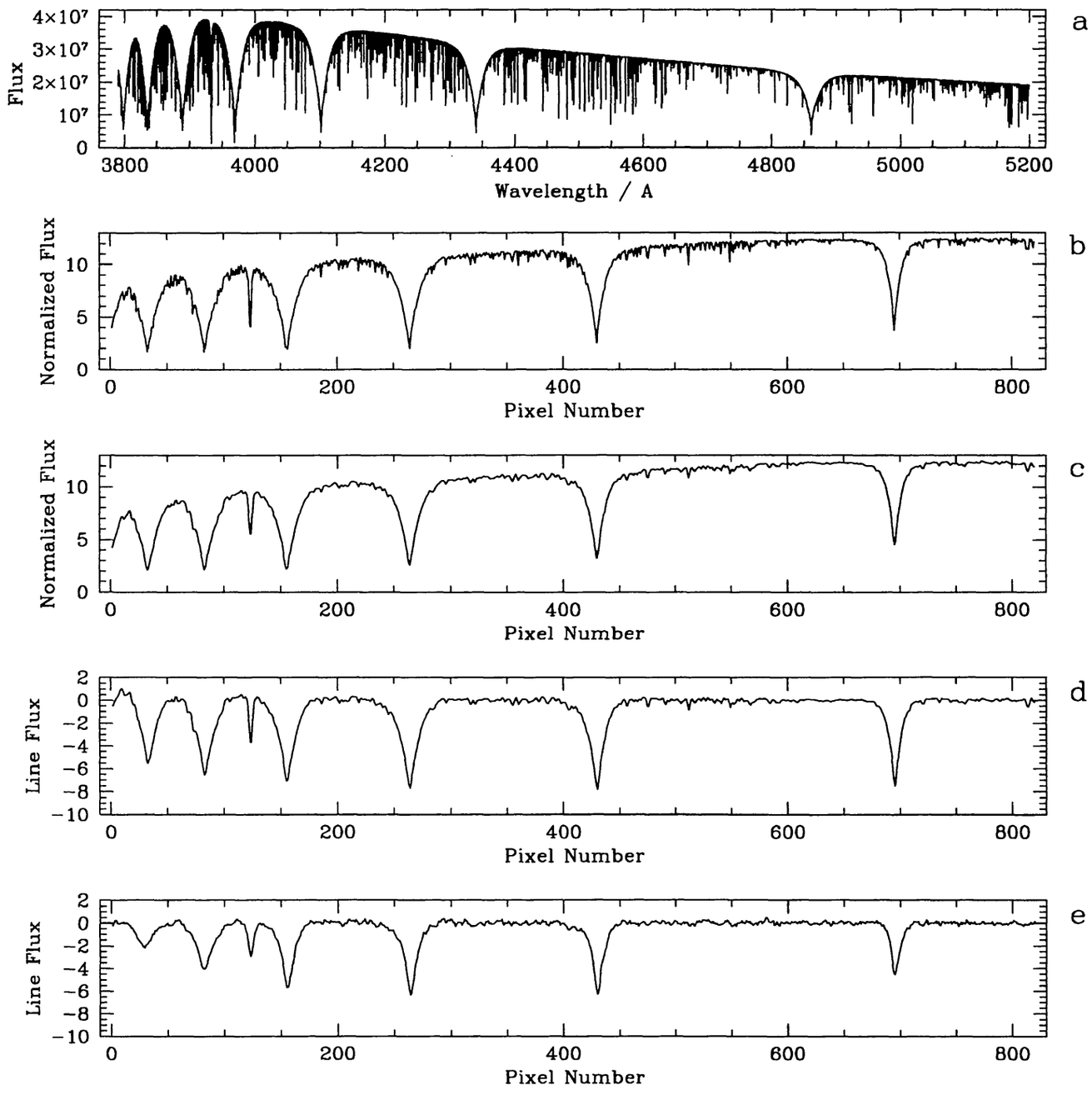
and https://www.ruder.io/optimizing-gradient-descent/ for details on optimization algorithms

Classifying spectra using artificial neural nets

A classical example:

Bailer-Jones et al (1997)

Input:



Classifying spectra using artificial neural nets

Bailer-Jones et al (1997)

Input: 821 fluxes
Neural net: 821 input units, 2 hidden layers with 5 weights each and 1 output unit

Classifying spectra using artificial neural nets

Bailer-Jones et al (1997)

Input: 821 fluxes
Neural net: 821 input units, 2 hidden layers with 5 weights each and 1 output unit

Combined 10 different neural nets (to check results, a kind of simplistic cross-validation).

Classifying spectra using artificial neural nets

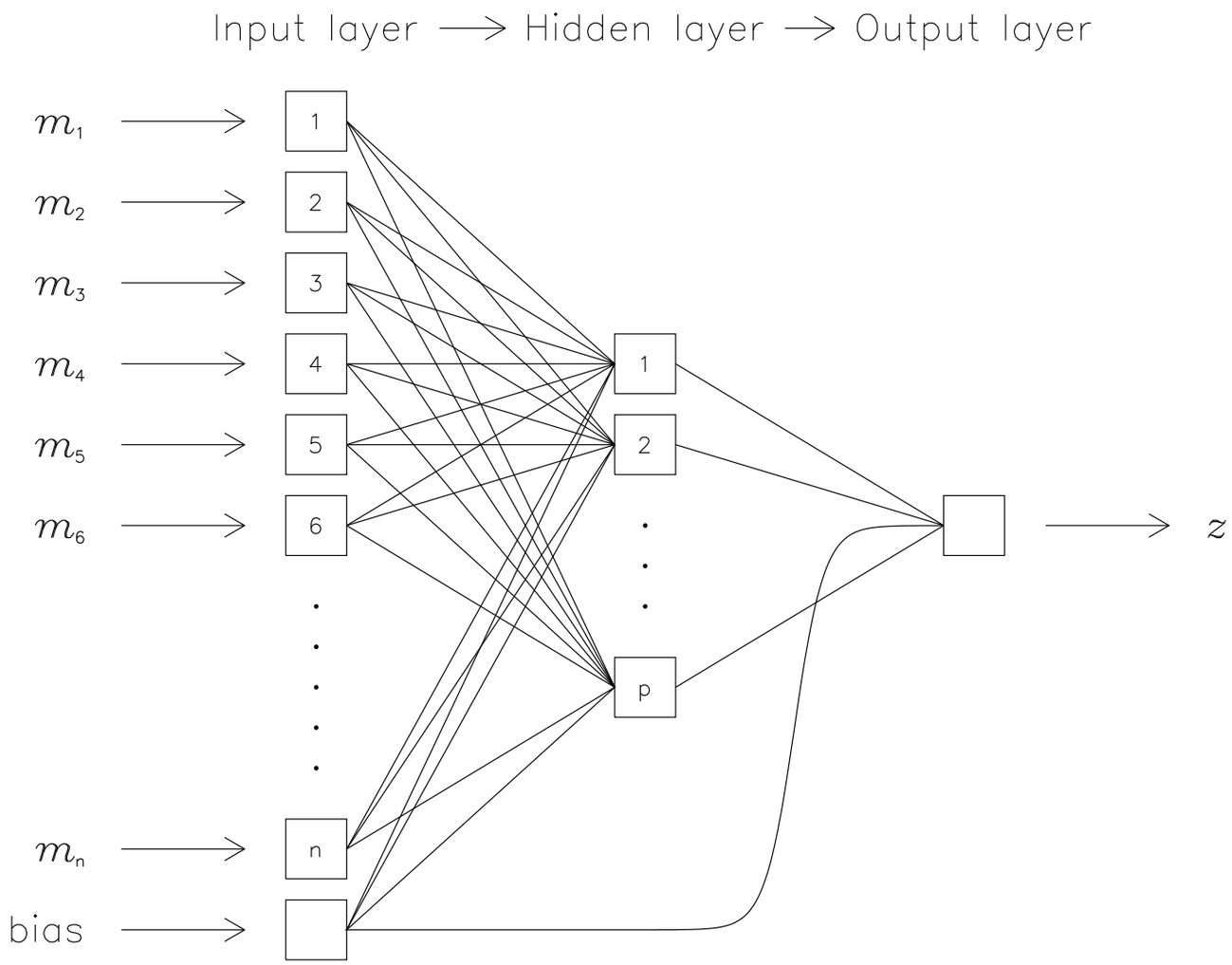
Bailer-Jones et al (1997)

Input: 821 fluxes
Neural net: 821 input units, 2 hidden layers with 5 weights each and 1 output unit

Combined 10 different neural nets (to check results, a kind of simplistic cross-validation).

Found that good estimates of T_{eff} can be found using this simple models but that metallicity introduces some uncertainty.

Redshifts from Neural Networks (ANNz)



ANNz is a freely available code that estimates a galaxy's redshift from input parameters.

Use a group of networks (a committee) and choose the best - needs to train on a comparable sample to what it will be applied to!

http://www.homepages.ucl.ac.uk/~ucapola/annz.html

Is it useful?

Appears to be useful for astronomical problems.
It does have a lot of strong believers and some strong opponents.

Advantages:

Very flexible - can approximate any function using two hidden layers.
Simple in structure and relatively easy to implement.
Widely used so plenty of literature and heuristic information.

Disadvantages:

Can be time-consuming to train and test - cross-validation can be extremely time-consuming.
Very hard to “understand” - do you get any insight from it?

From neural nets to deep learning

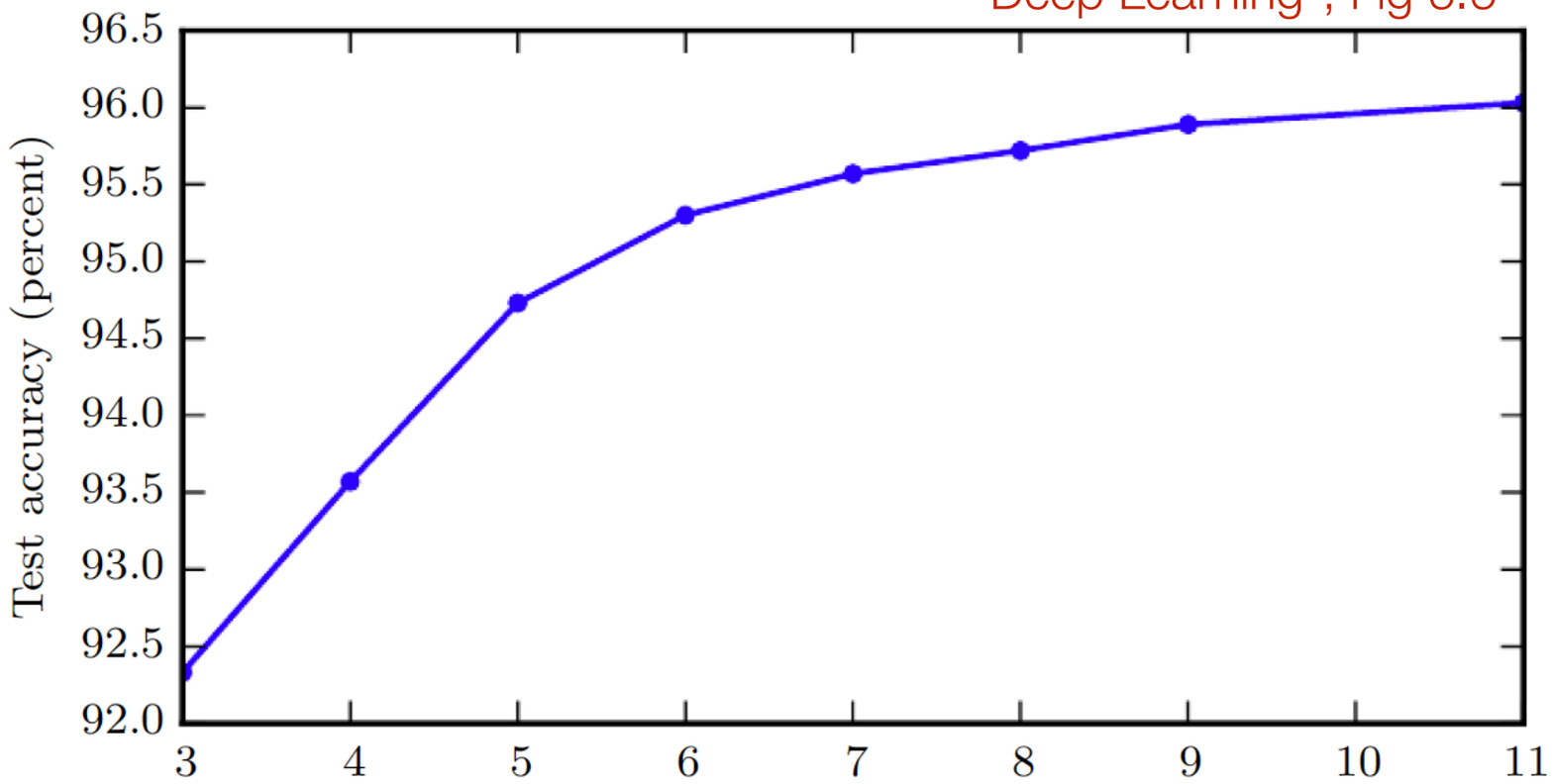
Deep learning is another name for neural networks.
So why the new name?

1. Much larger datasets enable more complex networks to be trained.
2. Improved computer speeds & better algorithms have made it easier to train many-layered networks.

Because these advances allows many more hidden layers to be used, it is called ‘deep’.

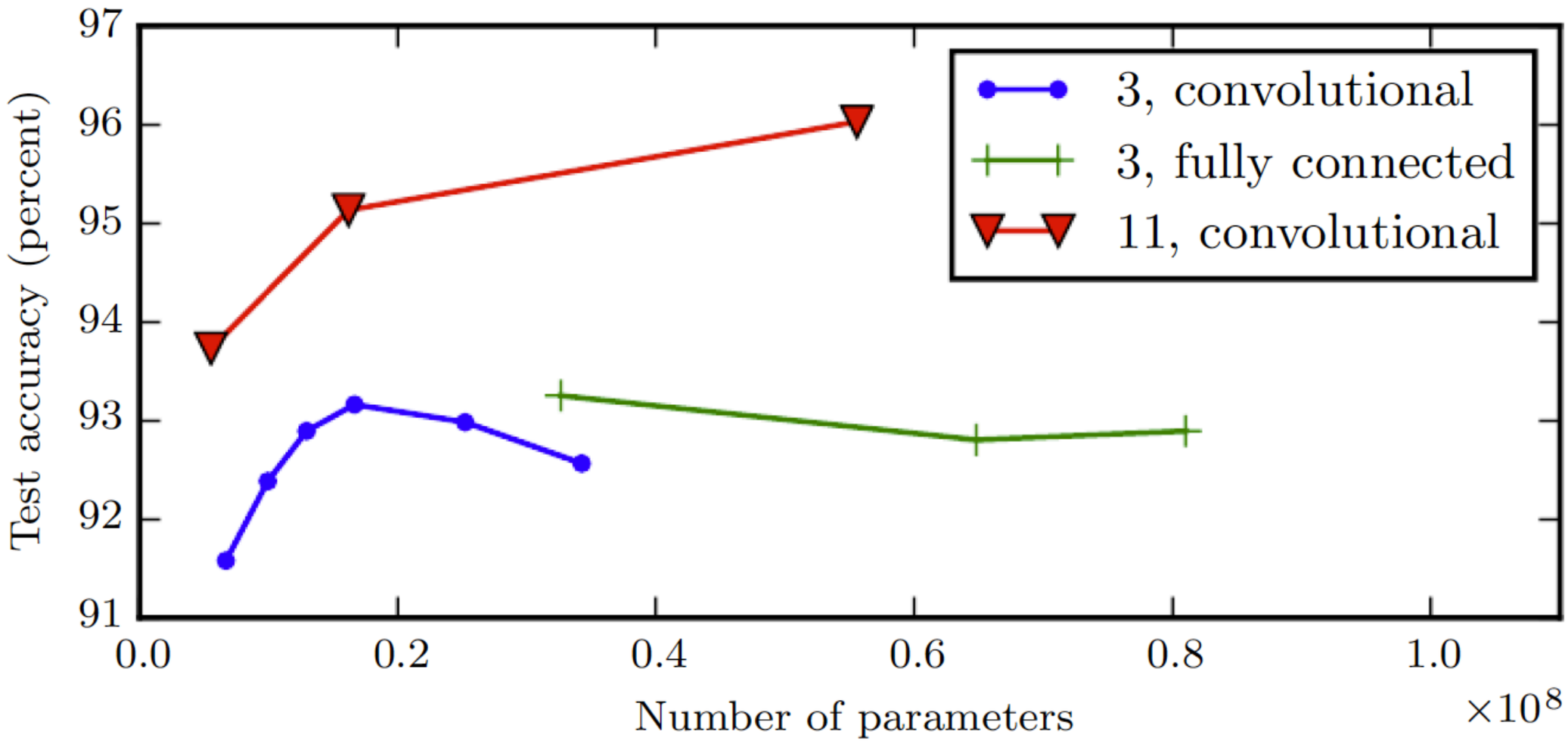
The advantage of depth

"Deep Learning", Fig 6.6



The performance of the neural networks tends to improve with depth.

The (dis?)advantage of depth



10^8 parameters is a lot… You need many training samples to avoid overfitting!

Handling overfitting -

Regularisation (very widely used): $J(\theta; \mathbf{X}, \mathbf{y}) + \lambda\Omega(\theta)$

e.g.:

$$J(\theta) + \lambda \sum \theta_i^2$$

(weight decay)

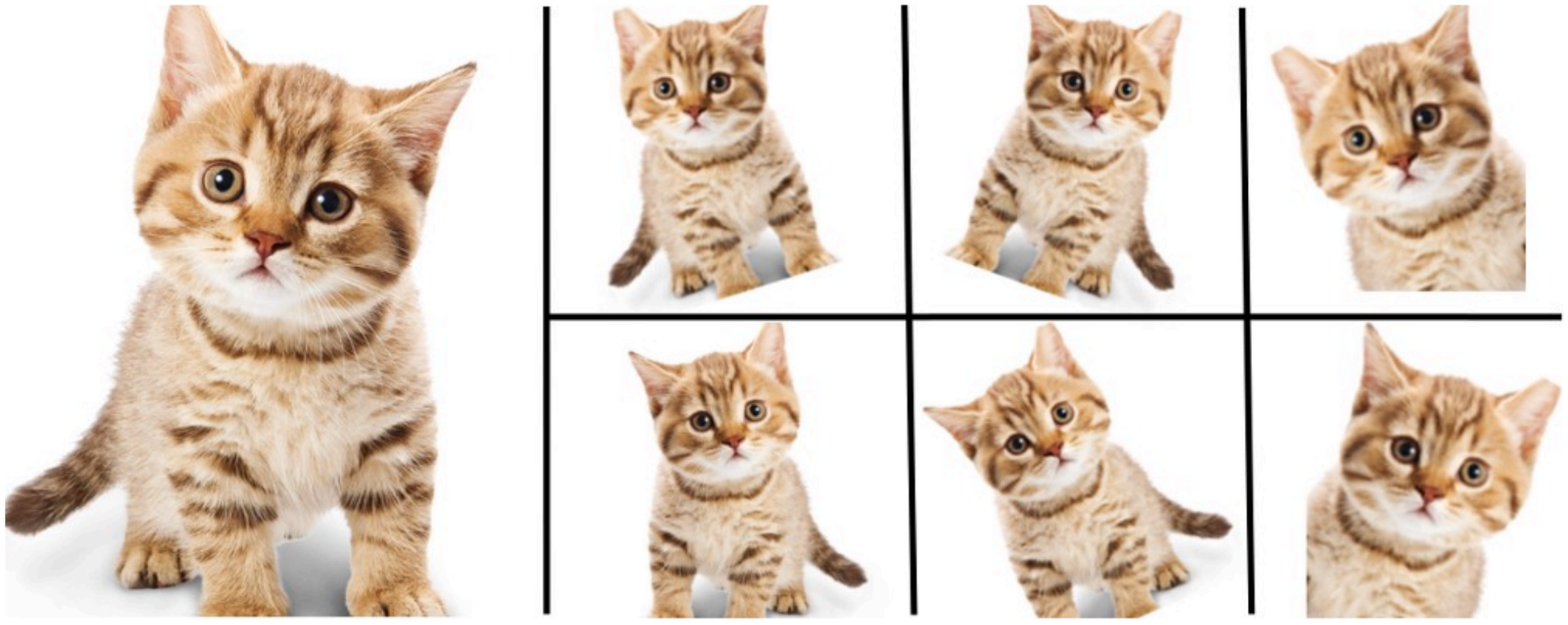
$$J(\theta) + \lambda \sum |\theta_i|$$

Note that the sum is over all i>0, thus the bias term is exempt

Very widely used - similarly to the case for regression, the absolute value regularisation leads to sparse solutions.

Handling overfitting: augmentation

Dataset augmentation: Enlarge your input data with modifications. Add rotated, shifted, noised, scaled examples.

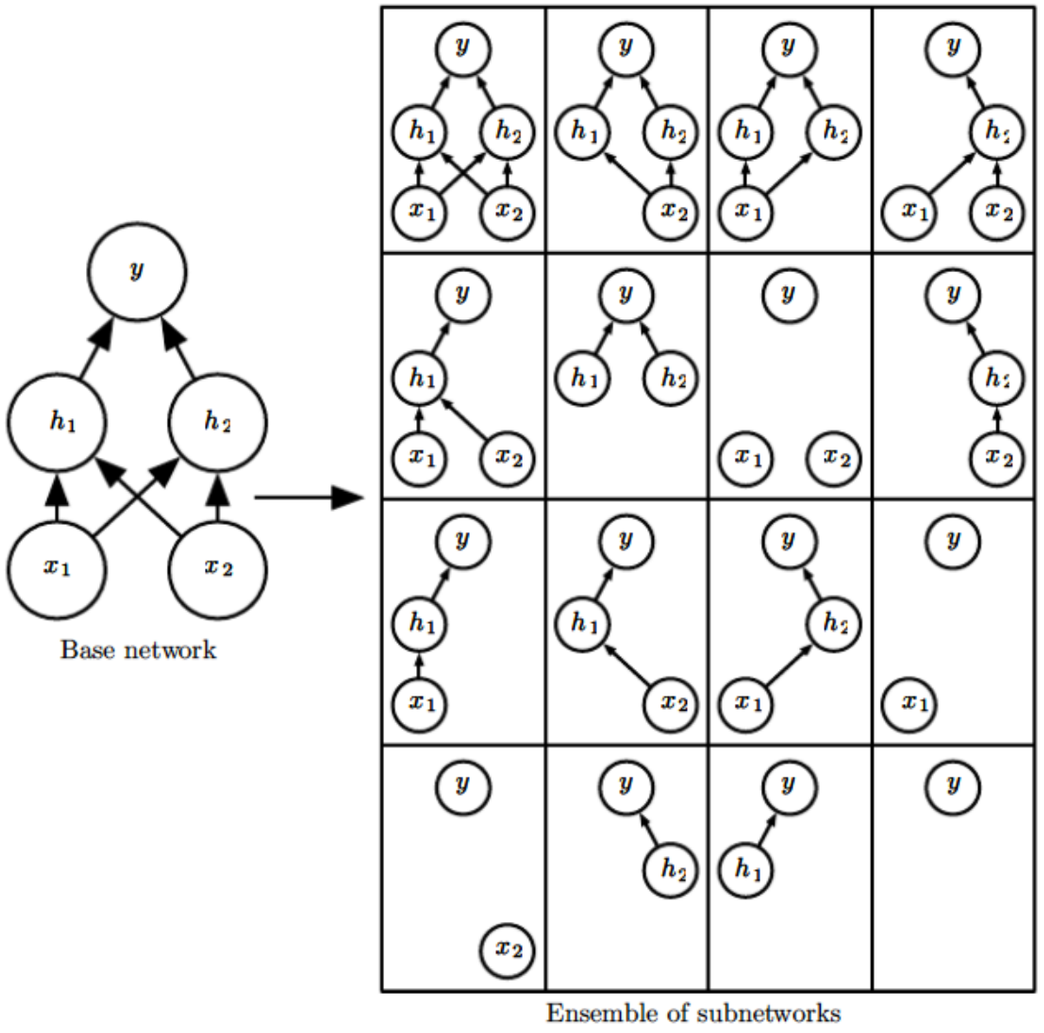


https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

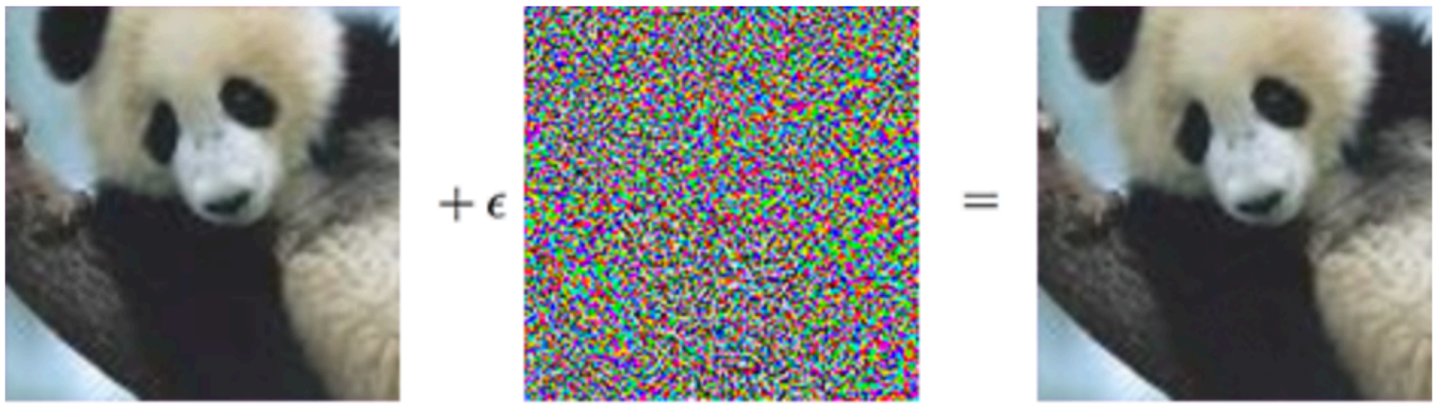
Handling overfitting: ensemble methods

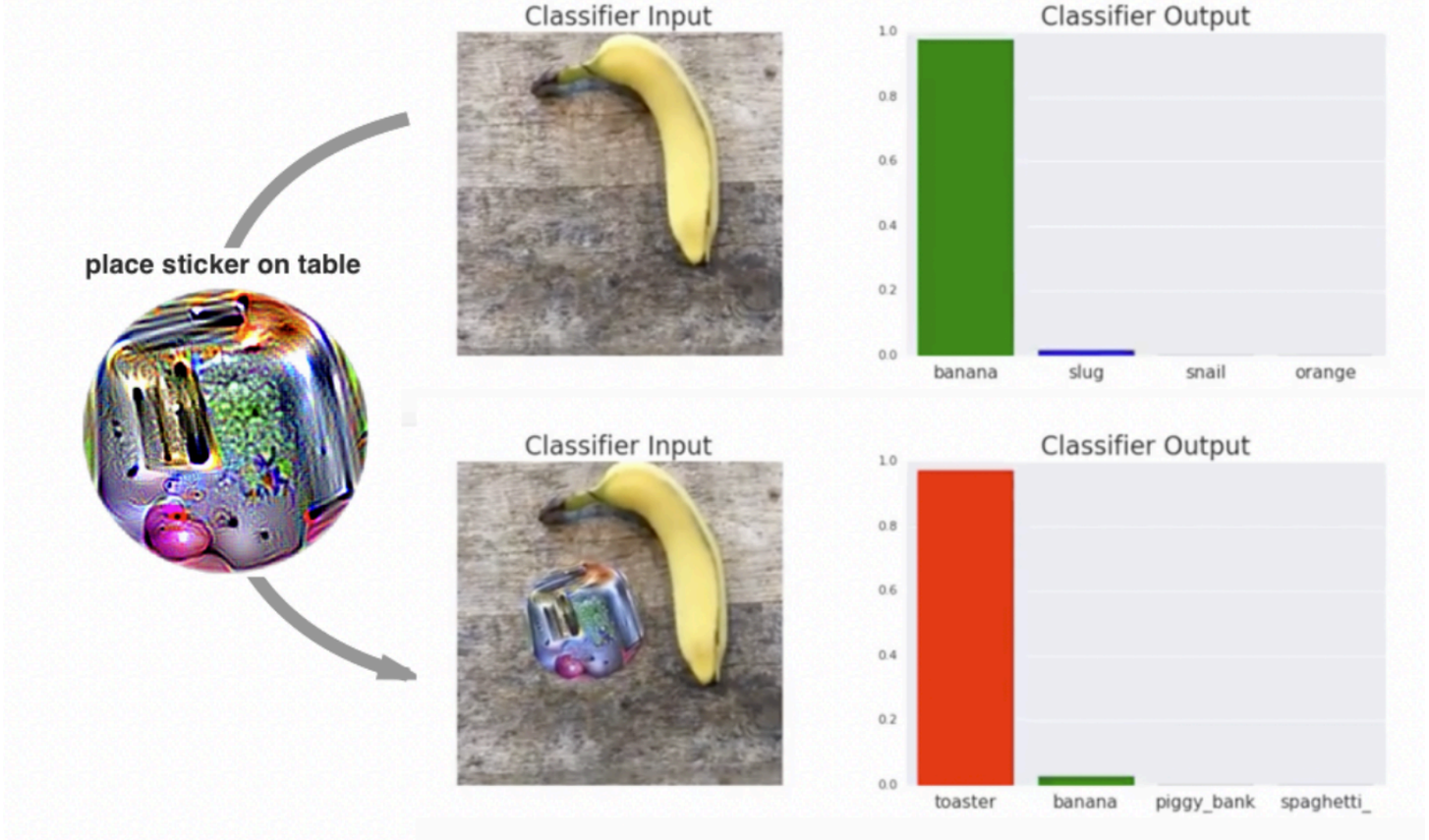
Bagging: averaging multiple neural nets. Works, but can be extremely costly. However winners in competitions often use averaging.

Dropout: in practice consists of randomly dropping connections in the network and is computationally less expensive approach and achieves good performance, so is often used (for large datasets at least).



Sensitivity & adversarial examples





See: https://towardsdatascience.com/breaking-neural-networks-with-adversarial-attacks-f4290a9a45aa for refs

Sensitivity & adversarial examples

Impersonation glasses:

on images:

Reese Witherspoon

100% confidence

Russel Crowe

(using glasses)

Russel Crowe

really.

https://www.cs.cmu.edu/~sbhagava/papers/face-rec-ccs16.pdf

Sensitivity & adversarial examples

Impersonation glasses:

on images:

Reese Witherspoon

Russel Crowe

Russel Crowe

100% confidence (using glasses) really.

in real life:

(a) (b) (c) (d)

https://www.cs.cmu.edu/~sbhagava/papers/face-rec-ccs16.pdf

Sensitivity & adversarial examples

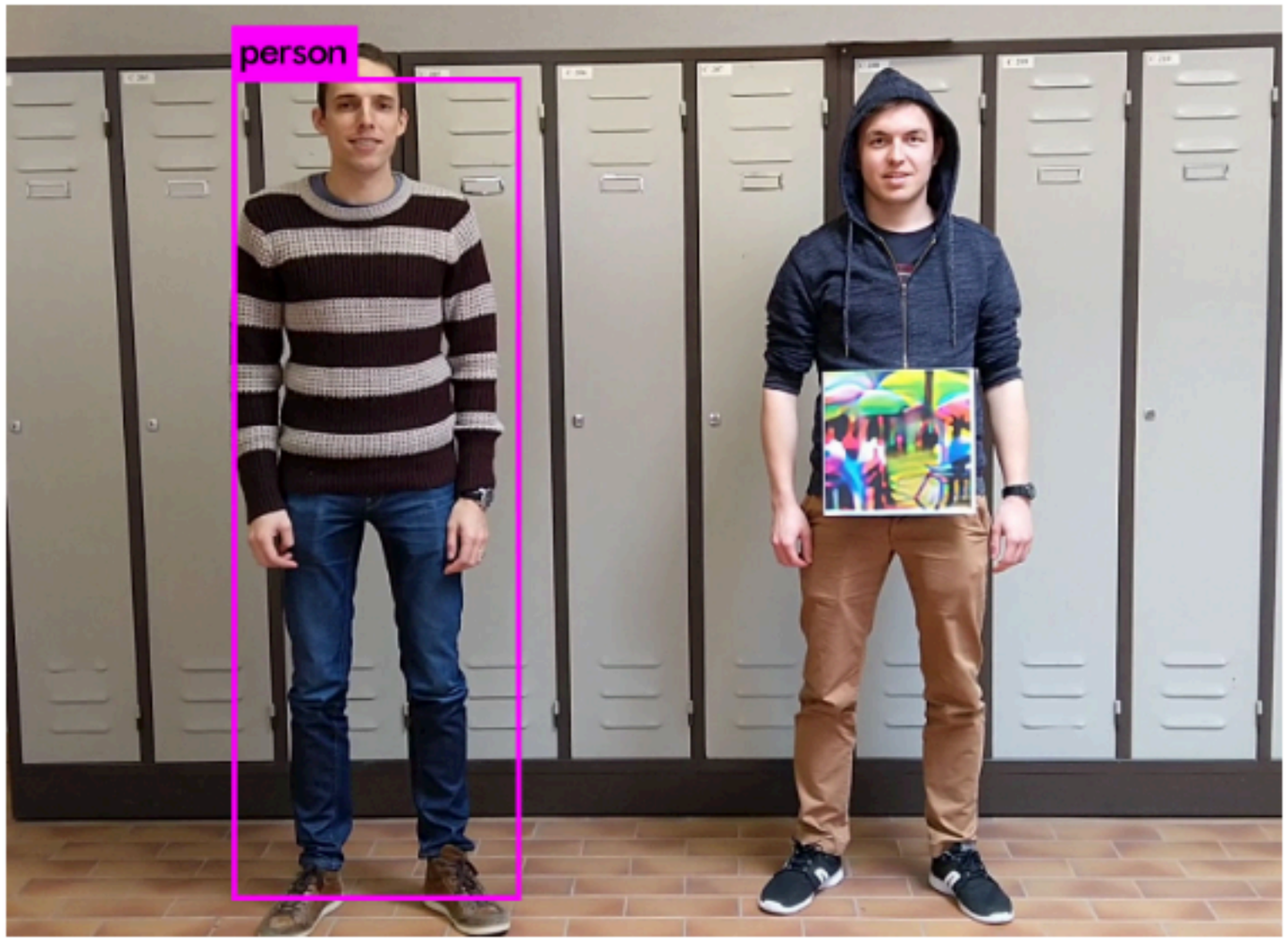
Camouflage Graffiti	Camouflage Art (LISA-CNN)	Camouflage Art (GTSRB-CNN)
		
		
67%	100%	80%

=

Putting small stickers on classifies this all as 45 mph signs

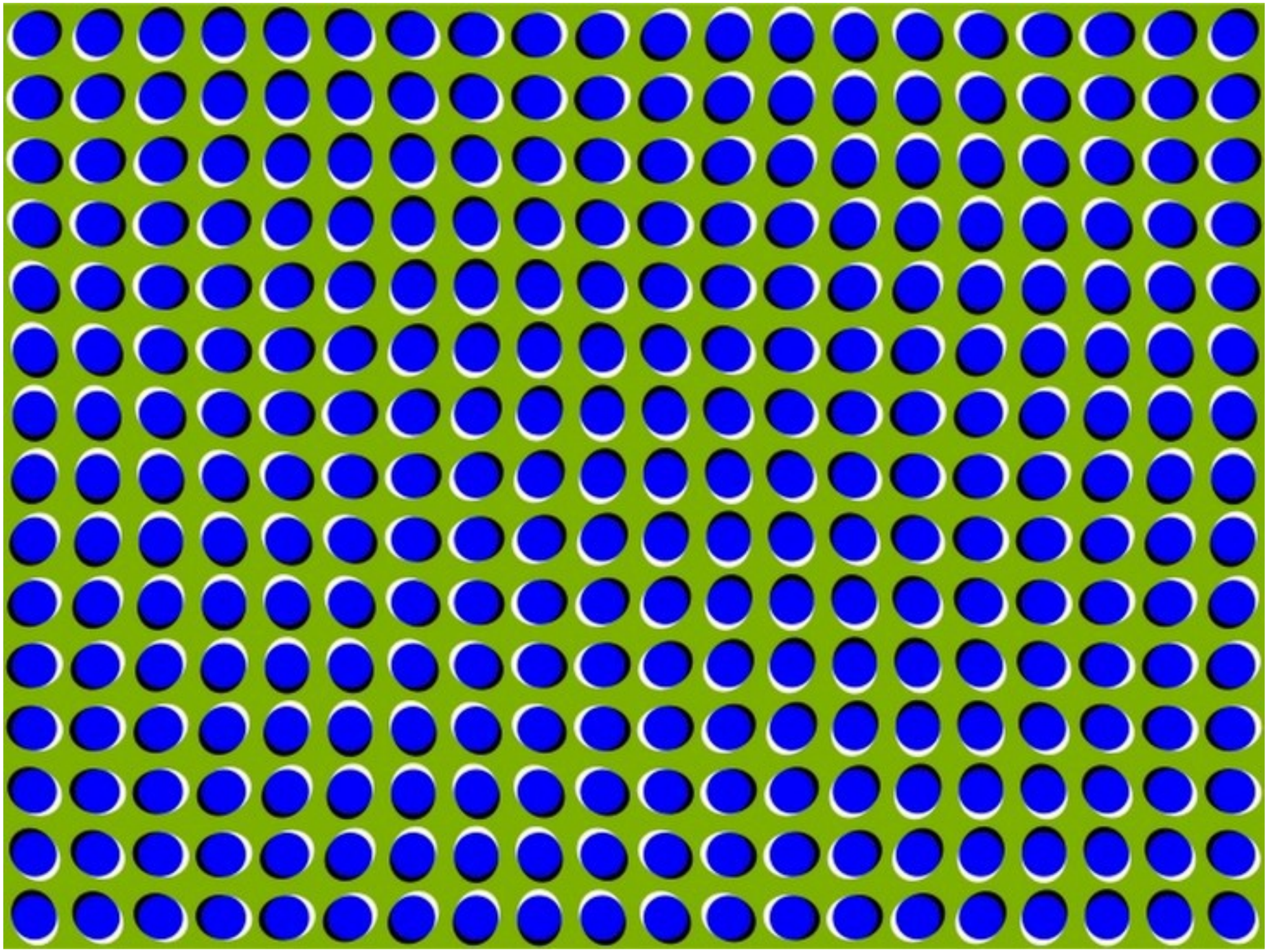
Eykholt et al (2017): https://arxiv.org/abs/1707.08945

“Fooling automated surveillance cameras”



Thys et al (2019, arXiv:1904.08653v1)

Adversarial examples = optical illusions?



How does this work?

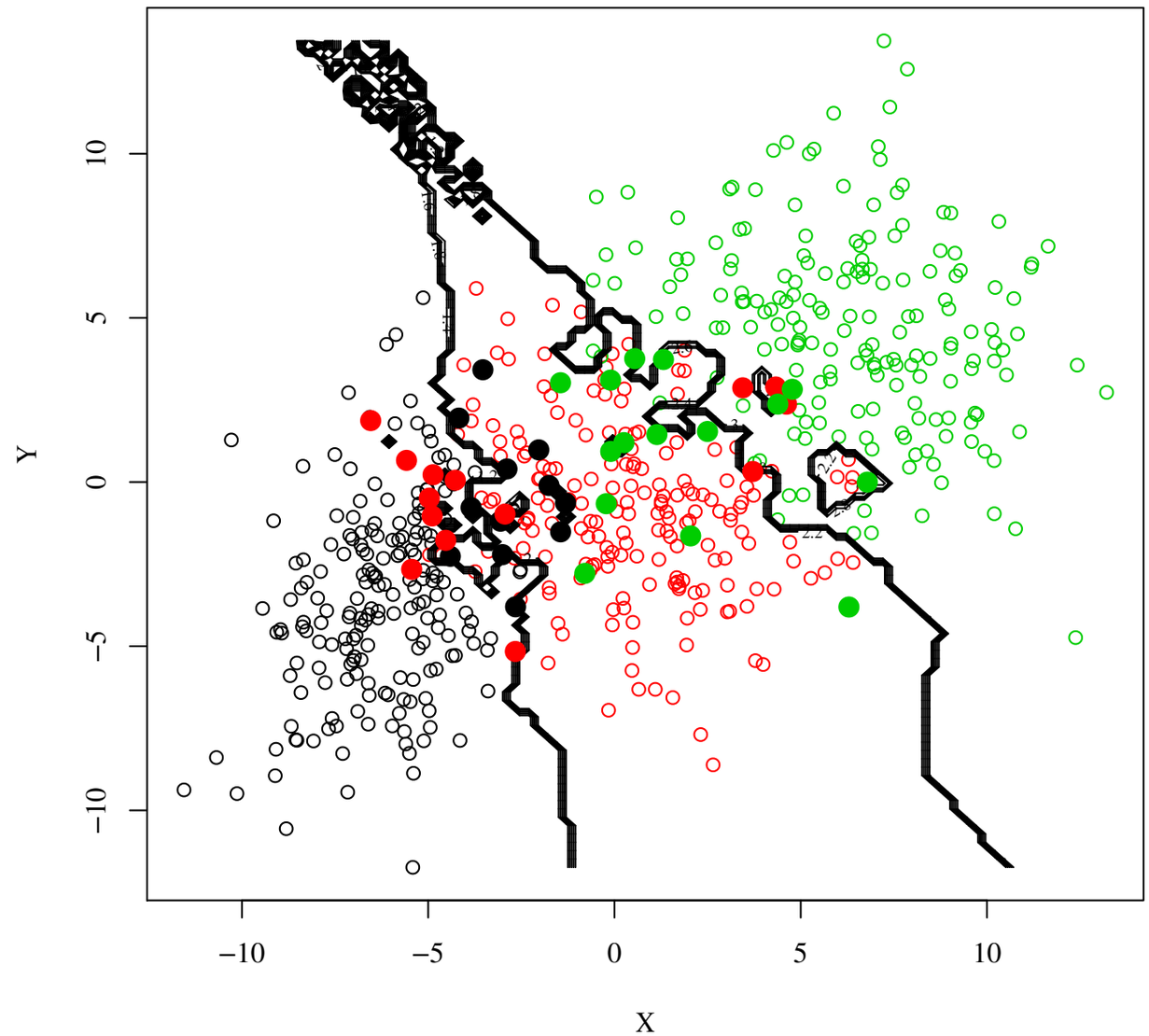
Tailored attacks: look at a part of the PDF for panda, calculate the gradient towards gibbon a place where they are closest and add the gradient:

Imagine something like:

How does this work?

Tailored attacks: look at a part of the PDF for panda, calculate the gradient towards gibbon a place where they are closest and add the gradient:

Imagine something like:

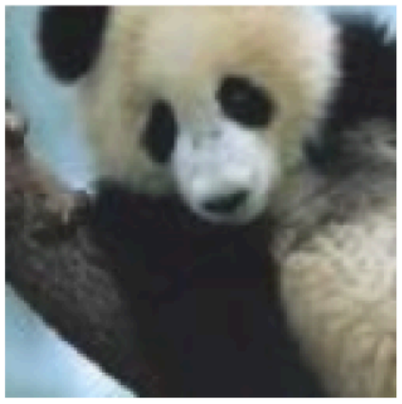


How does this work?

Tailored attacks: look at a part of the PDF for panda, calculate the gradient towards gibbon a place where they are closest and add the gradient:

How does this work?

Tailored attacks: look at a part of the PDF for panda, calculate the gradient towards gibbon a place where they are closest and add the gradient:



$\boldsymbol{x}$

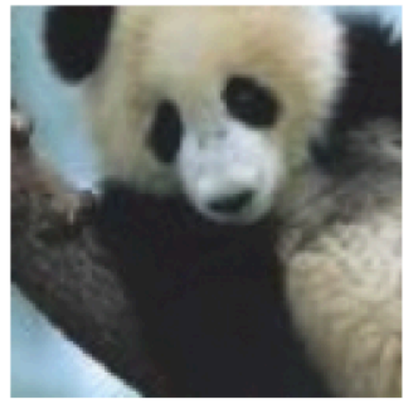
$y =$"panda"
w/ 57.7%
confidence

$+ .007 \times$

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
w/ 8.2%
confidence

$=$



$\boldsymbol{x} +$
$\epsilon \, \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

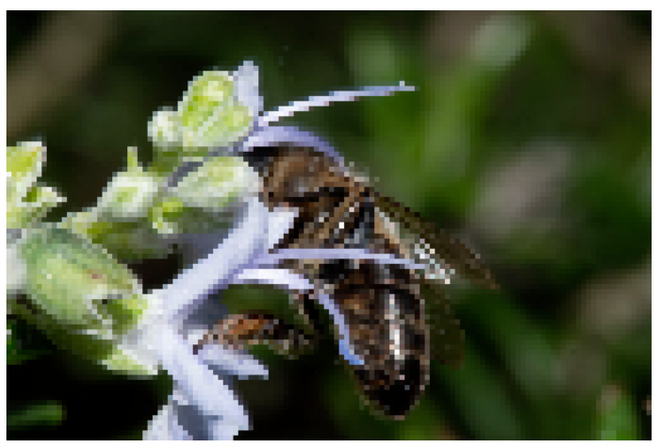
"gibbon"
w/ 99.3 %
confidence

Sharing data - convolutional and recurrent networks

Sometimes you want units to be very similar or identical

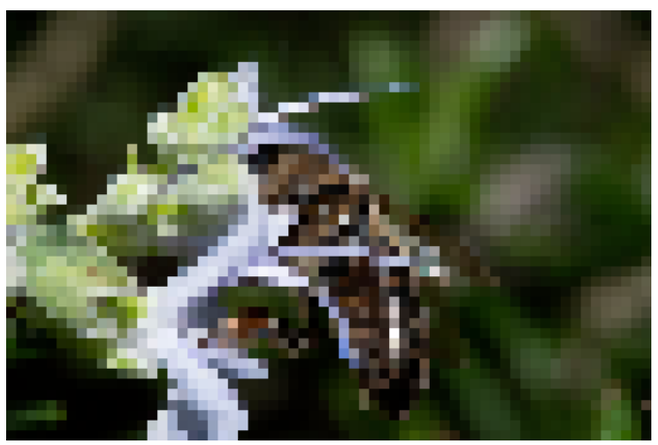
Convolutional layers do this by convolving images with a kernel - this can be used to detect edges, or simply to bin pixels together:

4x



10912 units

8x



2728 units

16x

628 units

Sharing data - convolutional and recurrent networks

Recurrent networks do the same thing in sequences

$$s(t) = f[s(t-1); \theta]$$

the state, s(t), is represented by a hidden layer - and more complex systems can be modelled

Very useful for e.g. speech recognition, handwriting recognition, translation etc.

Modern use of neural networks in astronomy

Write code in TensorFlow to train a neural network for image classification

Sure, here's an example code in TensorFlow to train a neural network for image classification using the CIFAR-10 dataset. This code uses a convolutional neural network (CNN) with three convolutional layers and two fully connected layers.

ork for image classification

rain a neural network for image
code uses a convolutional neural network
ully connected layers.

python Copy code

```python
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert labels to one-hot encoded vectors
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define model architecture
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(32,
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['a

# Train model
model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=1, validation

# Evaluate model on test set
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

In this code, we first load the CIFAR-10 dataset and normalize the pixel values between 0 and 1. We also convert the labels to one-hot encoded vectors.

Next, we define the model architecture using the `**Sequential**` API from Keras. The model has three convolutional layers with increasing number of filters, followed by max pooling and dropout layers to prevent overfitting. The output of the last convolutional layer is flattened and passed through two fully connected layers with ReLU activation and a final softmax layer for classification.

We compile the model with categorical cross-entropy loss and the Adam optimizer. We then train the model on the training set for 10 epochs with a batch size of 128, and validate the model on the test set after each epoch.

Finally, we evaluate the model on the test set and print the test loss and accuracy.

ork for image classification

train a neural network for image
code uses a convolutional neural network
ully connected layers.

```python
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert labels to one-hot encoded vectors
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define model architecture
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(32,
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['a

# Train model
model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=1, validation

# Evaluate model on test set
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

NB: Not tested!

In this code, we first load the CIFAR-10 dataset and normalize the pixel values between 0 and 1. We also convert the labels to one-hot encoded vectors.

Next, we define the model architecture using the `Sequential` API from Keras. The model has three convolutional layers with increasing number of filters, followed by max pooling and dropout layers to prevent overfitting. The output of the last convolutional layer is flattened and passed through two fully connected layers with ReLU activation and a final softmax layer for classification.

We compile the model with categorical cross-entropy loss and the Adam optimizer. We then train the model on the training set for 10 epochs with a batch size of 128, and validate the model on the test set after each epoch.

Finally, we evaluate the model on the test set and print the test loss and accuracy.

Convolutional neural networks

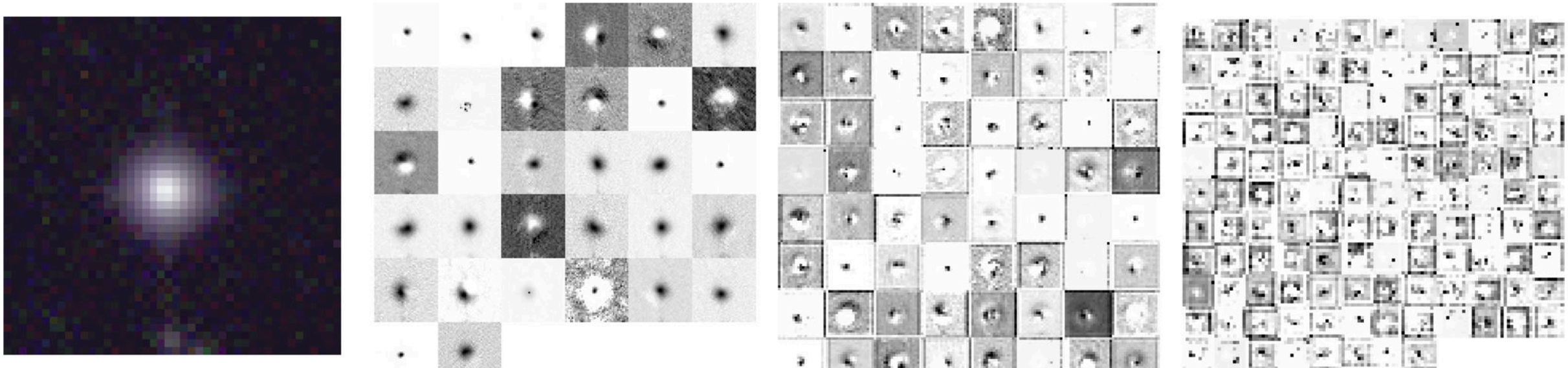
These are neural networks that replace matrix multiplication with convolution in at least one of the layers.

These networks can work on images/spectra/time-series with no need to pre-select features.

For that reason they have quickly become very popular in astronomy.

Star-galaxy separation

Kim & Brunner (2015)



(a) Input (5 bands×44×44) (b) Layer 1 (32 maps×40×40) (c) Layer 3 (64 maps×20×20) (d) Layer 6 (128 maps×10×10)

11 layers in total: 8 convolutional, three fully connected.

Leaky ReLUs as their activation functions: $\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0. \end{cases}$

10^7 parameters, but 4×10^4 images - overfitting!

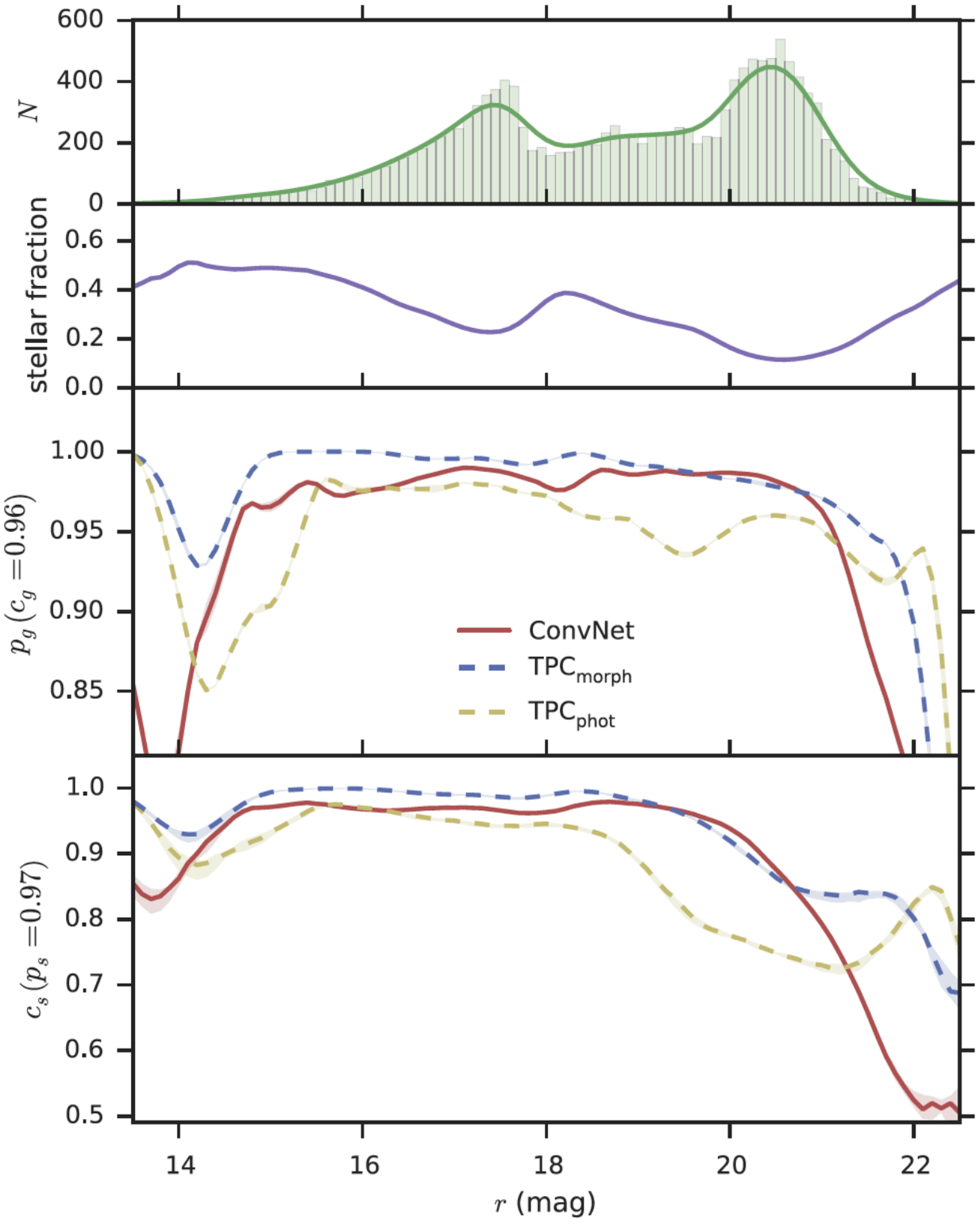
Star-galaxy separation - avoiding overfitting

Kim & Brunner (2015)

a) Data augmentation (also input rotated, reflected, translated, noised images).
b) Dropout (randomly dropping neurons) - this helps the network be more robust.

Grouping pixels together (“pooling”) so that less is kept as you go down in the network.

Star-galaxy separation - results



TPC is a random forest classifier.
As a general rule the convolutional neural network performs almost as well as this.

The advantage(?): TPC requires measurements as input, the ConvNet works straight on the images.

Transient detection

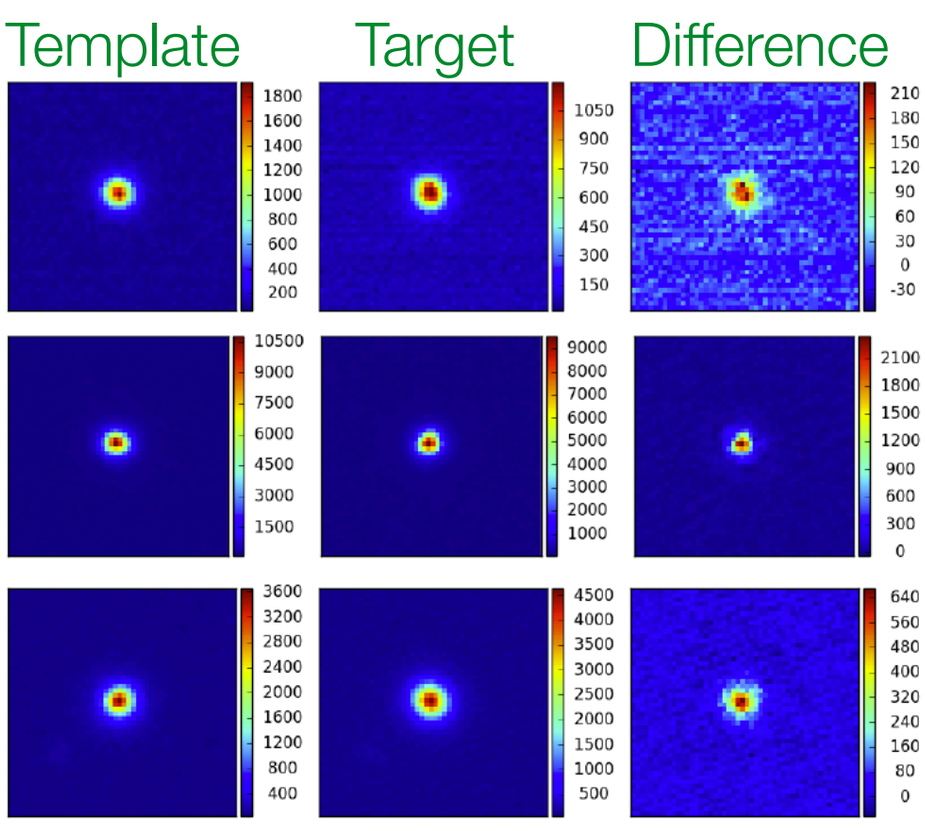
Future surveys like LSST will detect ~1 million transient events per night. Which ones should we follow up?

Analysis typically focuses on difference images & features are analysed using random forests typically.

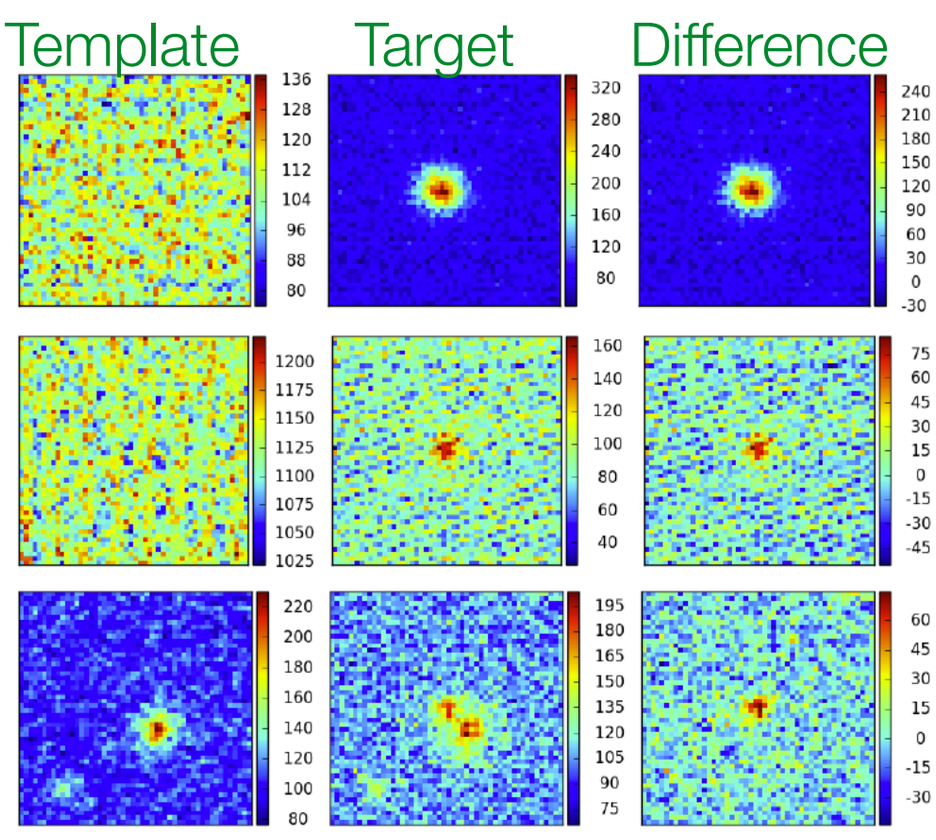
Gieseke et al (2017) uses a convolutional neural network to classify transients.

Transient detection

Gieseke et al (2017) uses a convolutional neural network to classify transients.



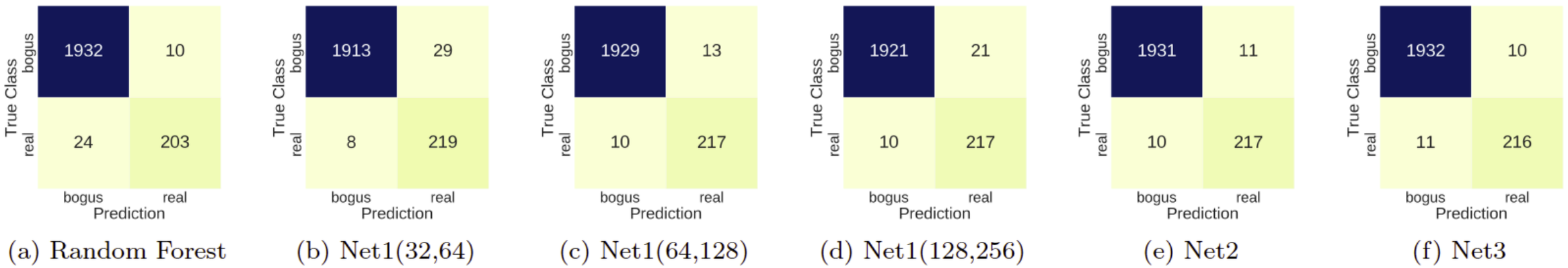
(a) Bogus



(b) Real

Transient detection

Gieseke et al (2017) uses a convolutional neural network to classify transients.

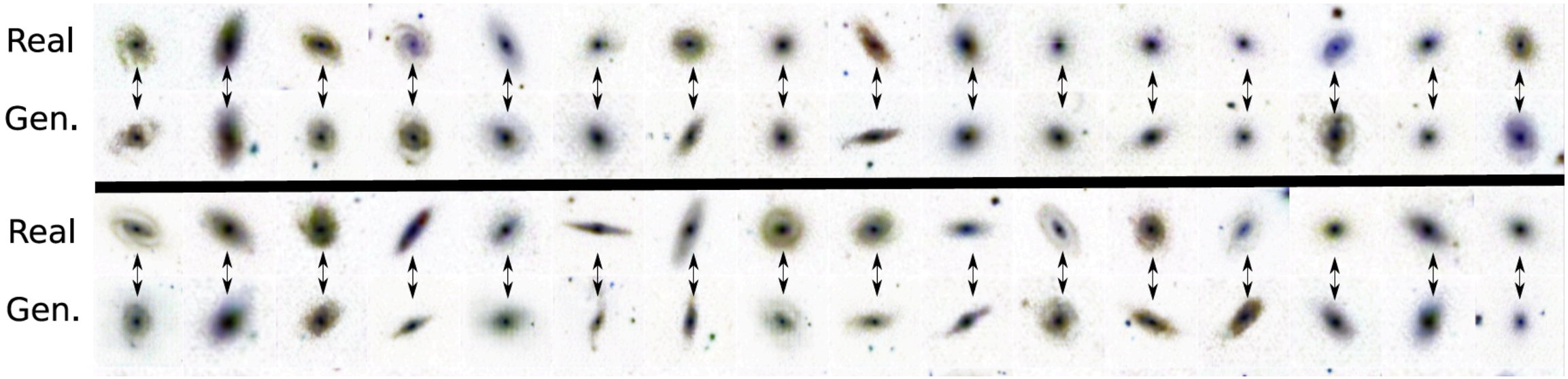


(a) Random Forest (b) Net1(32,64) (c) Net1(64,128) (d) Net1(128,256) (e) Net2 (f) Net3

They find again good performance relative to random forests, but not clearly better.

Generating new images

Ravanbakhsh et al (2016) -

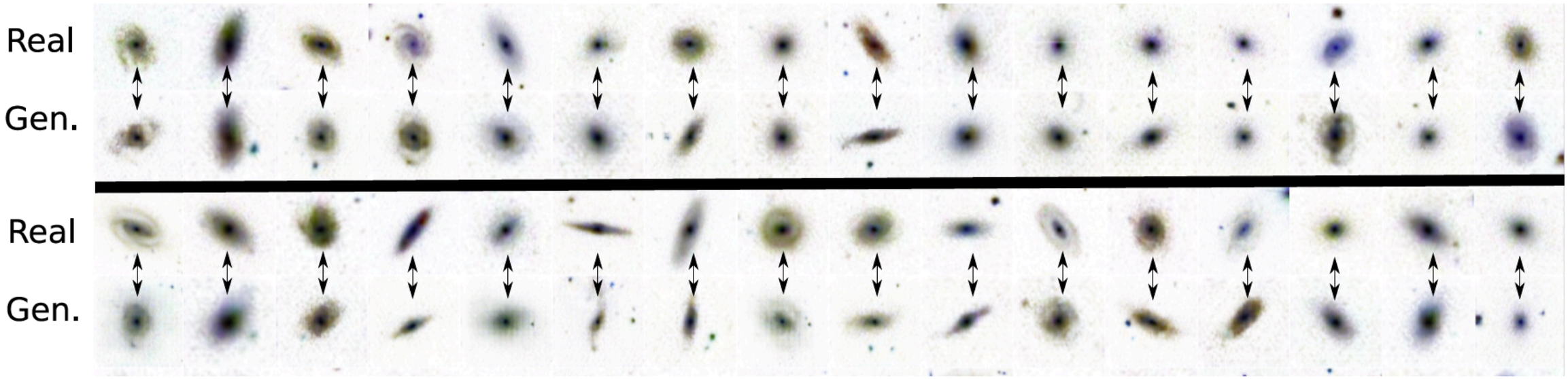


Use neural networks that can generate realistic images:

- Variational Autoencoder
- Generative Adversial Networks

Generating new images

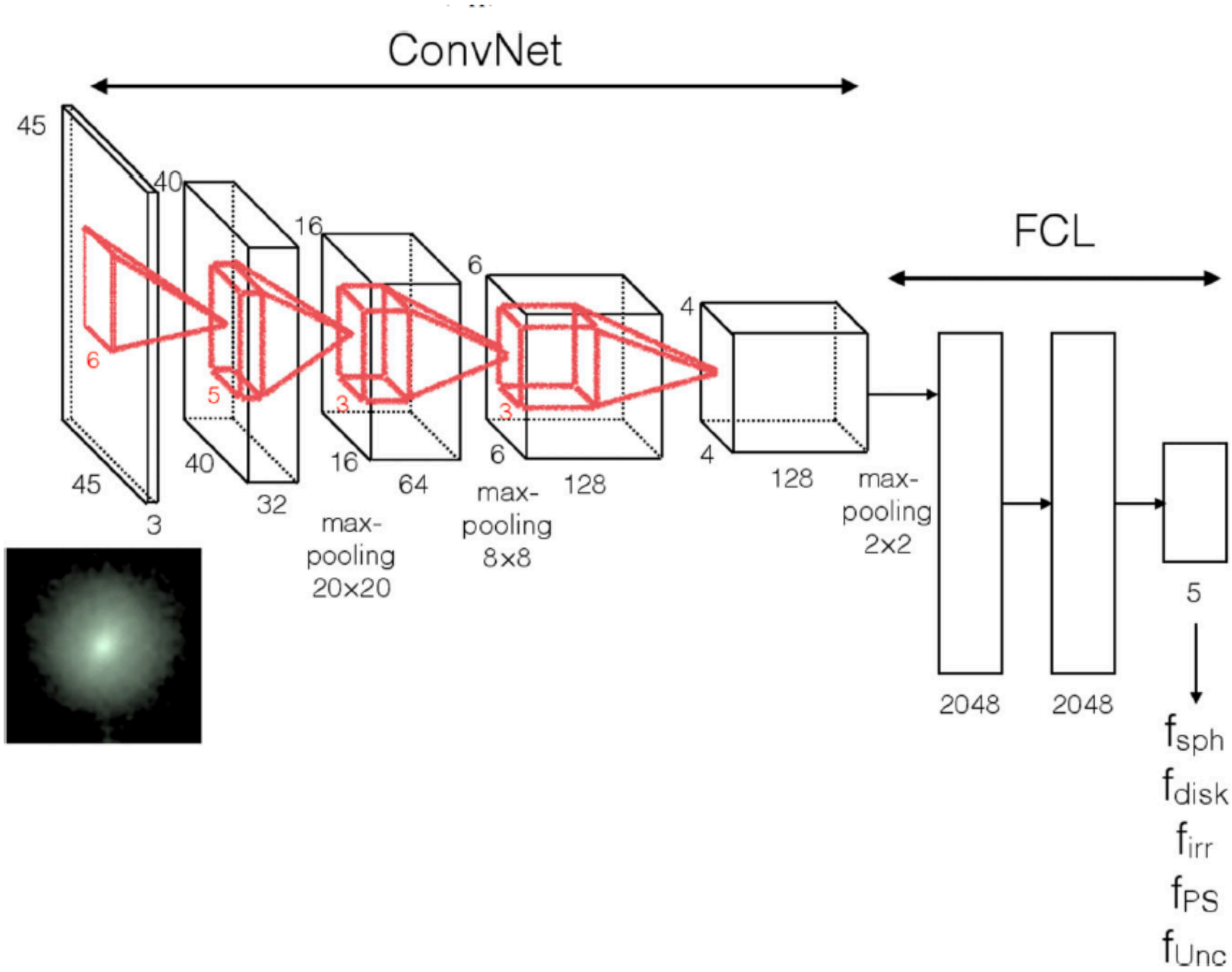
Ravanbakhsh et al (2016) -



aim: Create training samples for gravitation lensing methods etc.

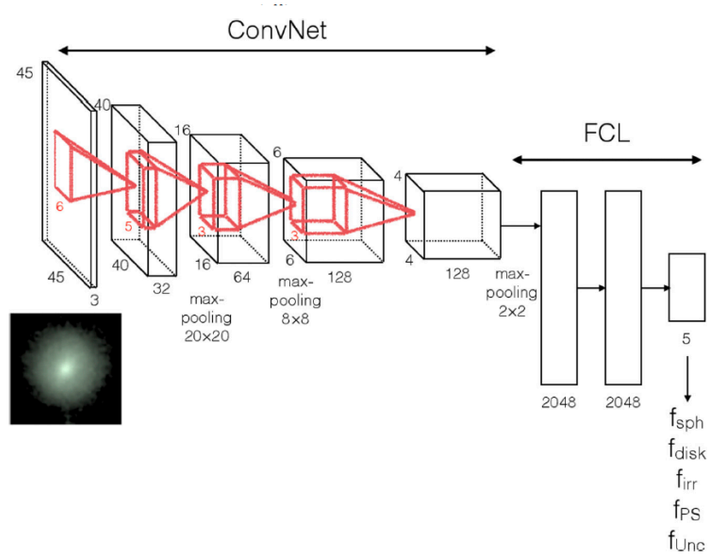
Galaxy morphology

Huertas-Company et al (2015):



Galaxy morphology

Huertas-Company et al (2015):



Convolutional neural network, ~1000 galaxies/hour on a Tesla M2090 GPU

Performance: Less than 10% error, comparable to human classifier.

Metallicity from 3-colour images

Figure 4.

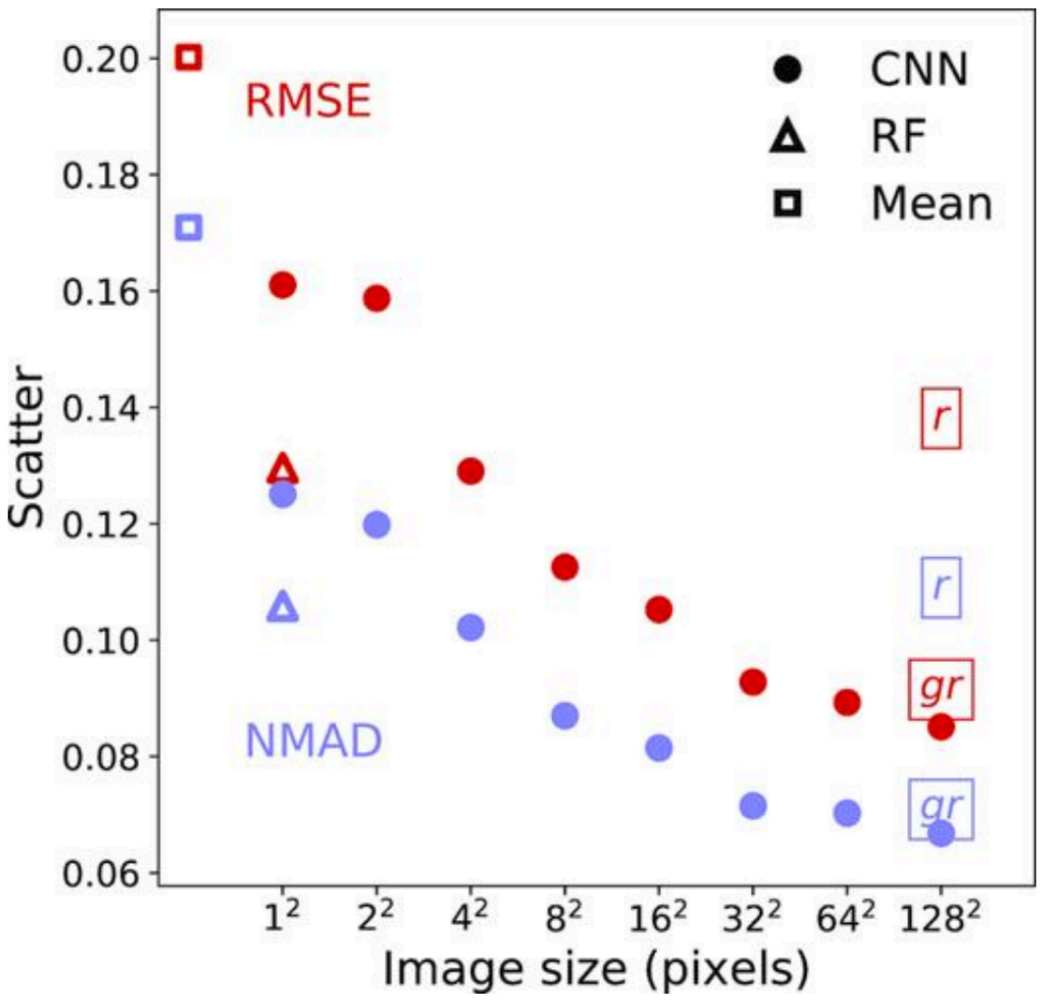
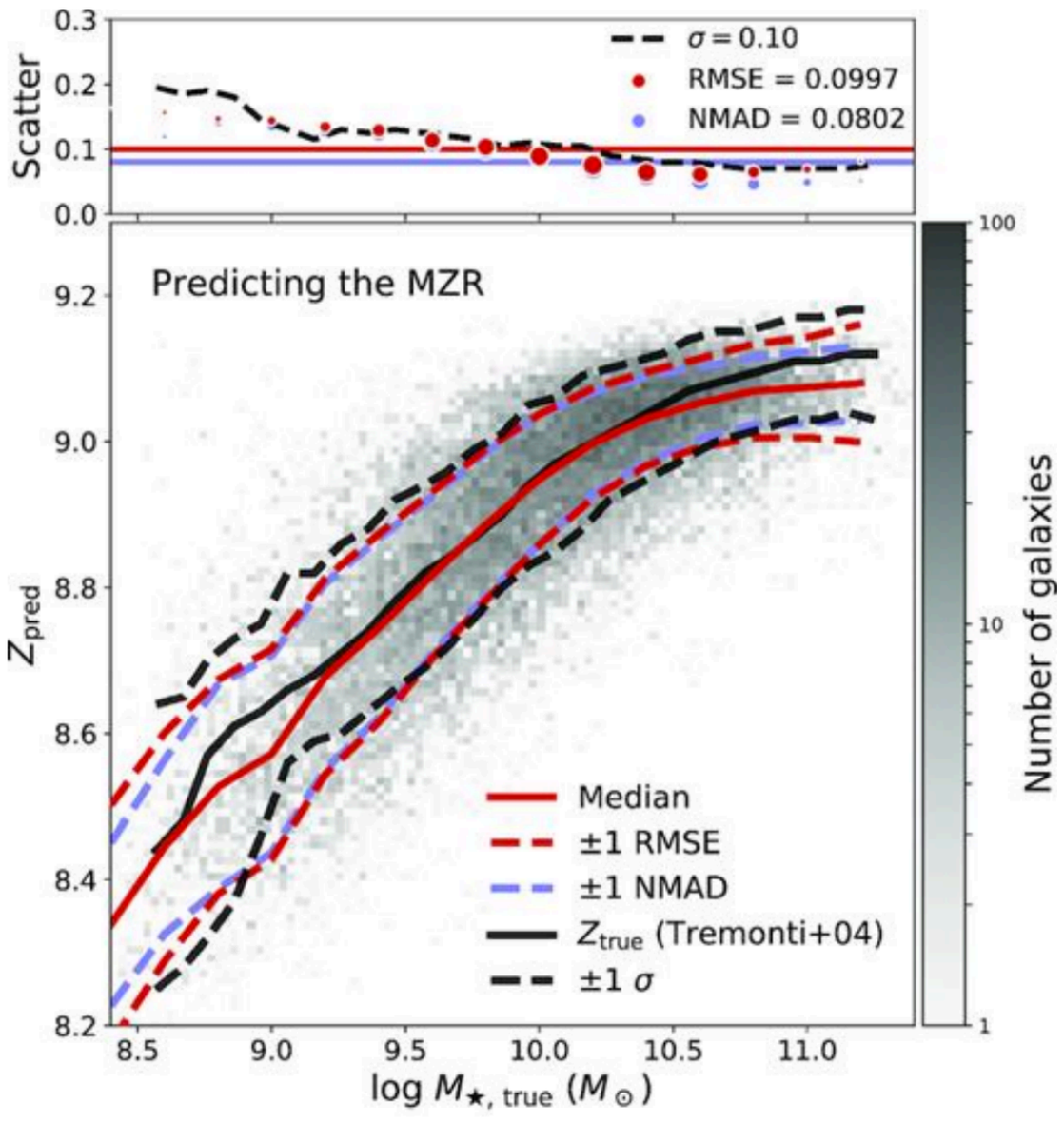


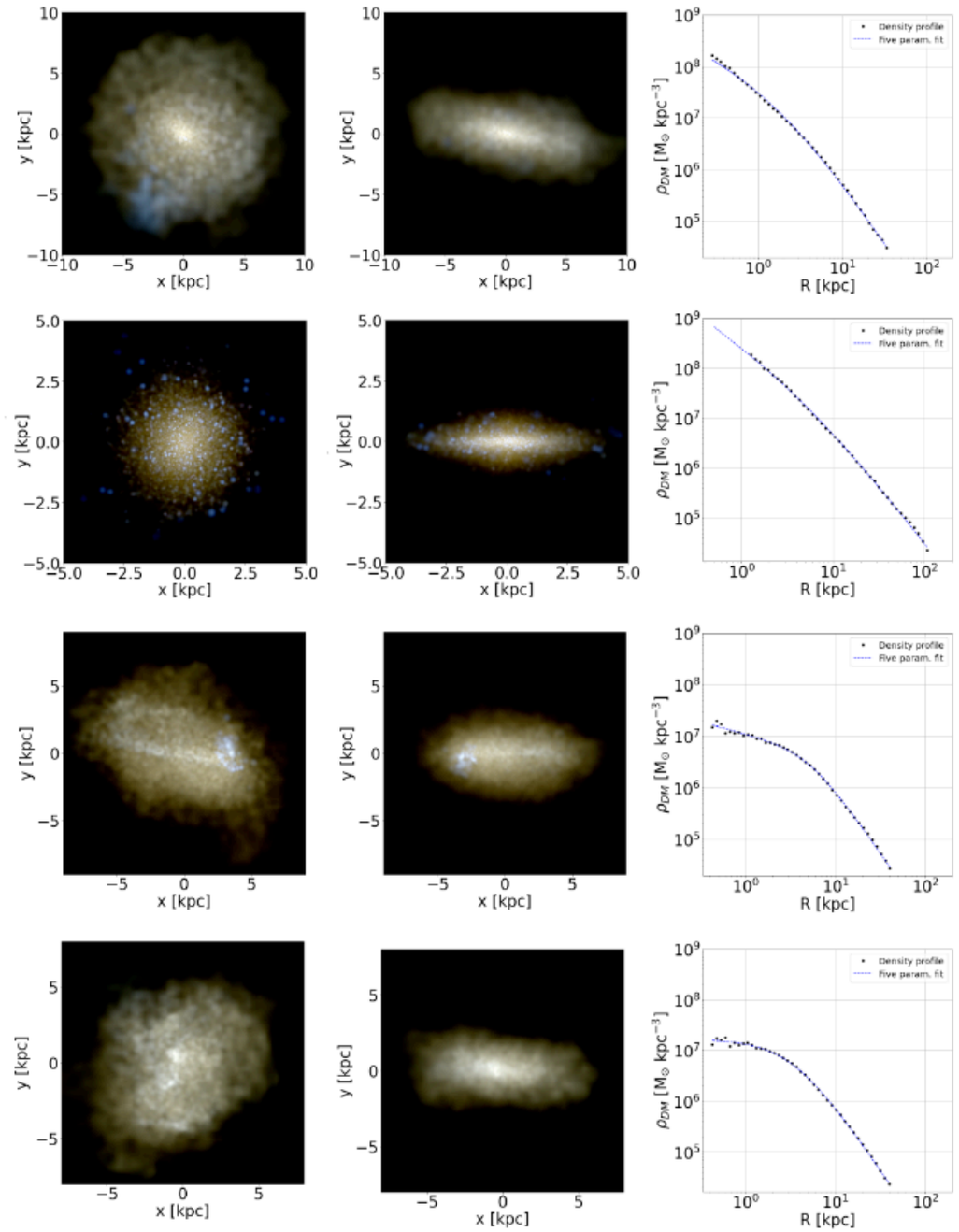
Figure 6.



Wu & Boada (2019)

Convolutional neural network. 34 layers - start with ImageNet.
Using pytorch and also using data augmentation.

Determining mass profiles using deep networks

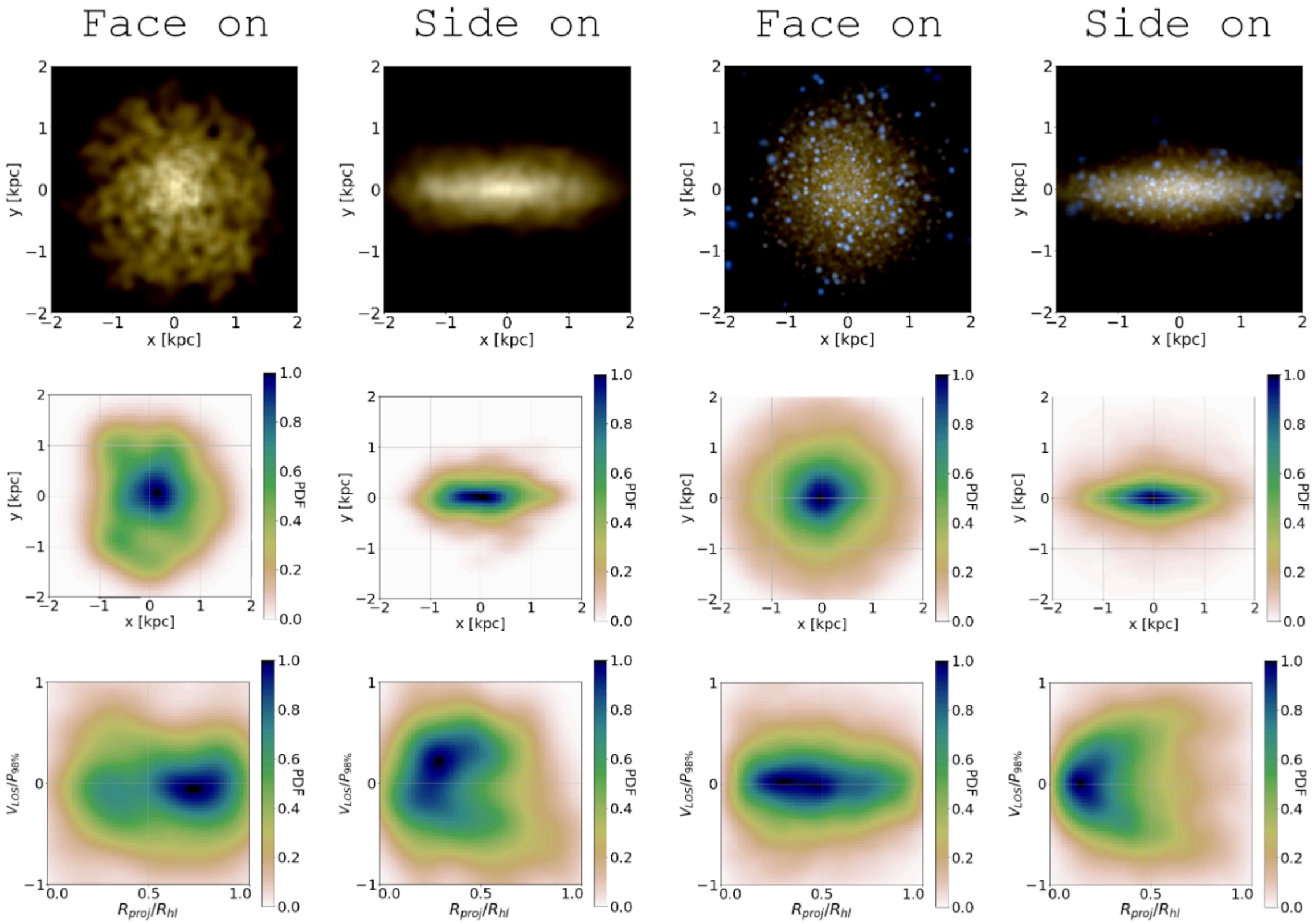


KDE → Input variables

Cored and cuspy galaxies from simulations (NIHAO, AURIGA)

Expósito-Márquez et al (2023)

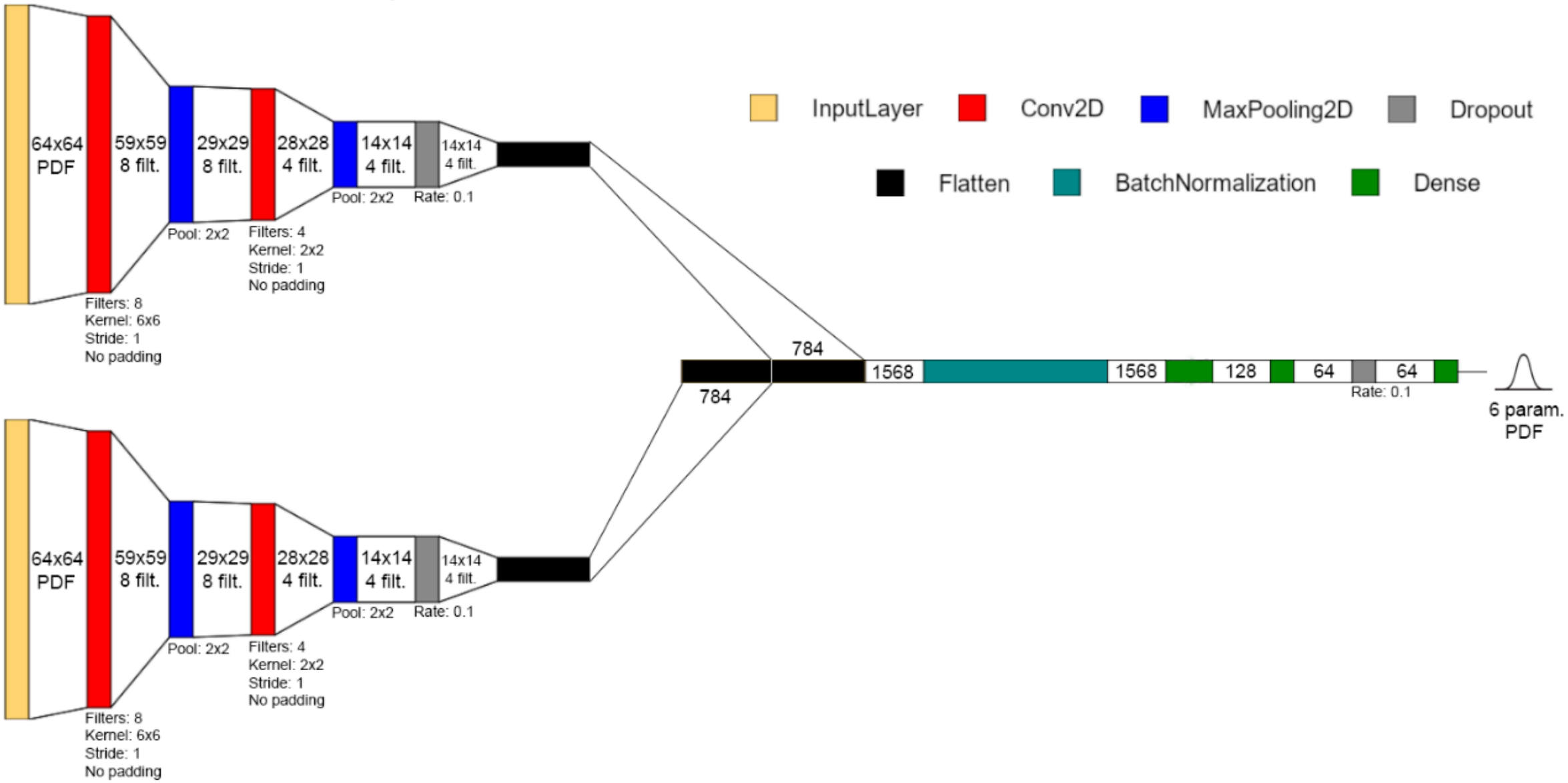
Determining mass profiles using deep networks



Expósito-Márquez et al (2023)

Determining mass profiles using deep networks

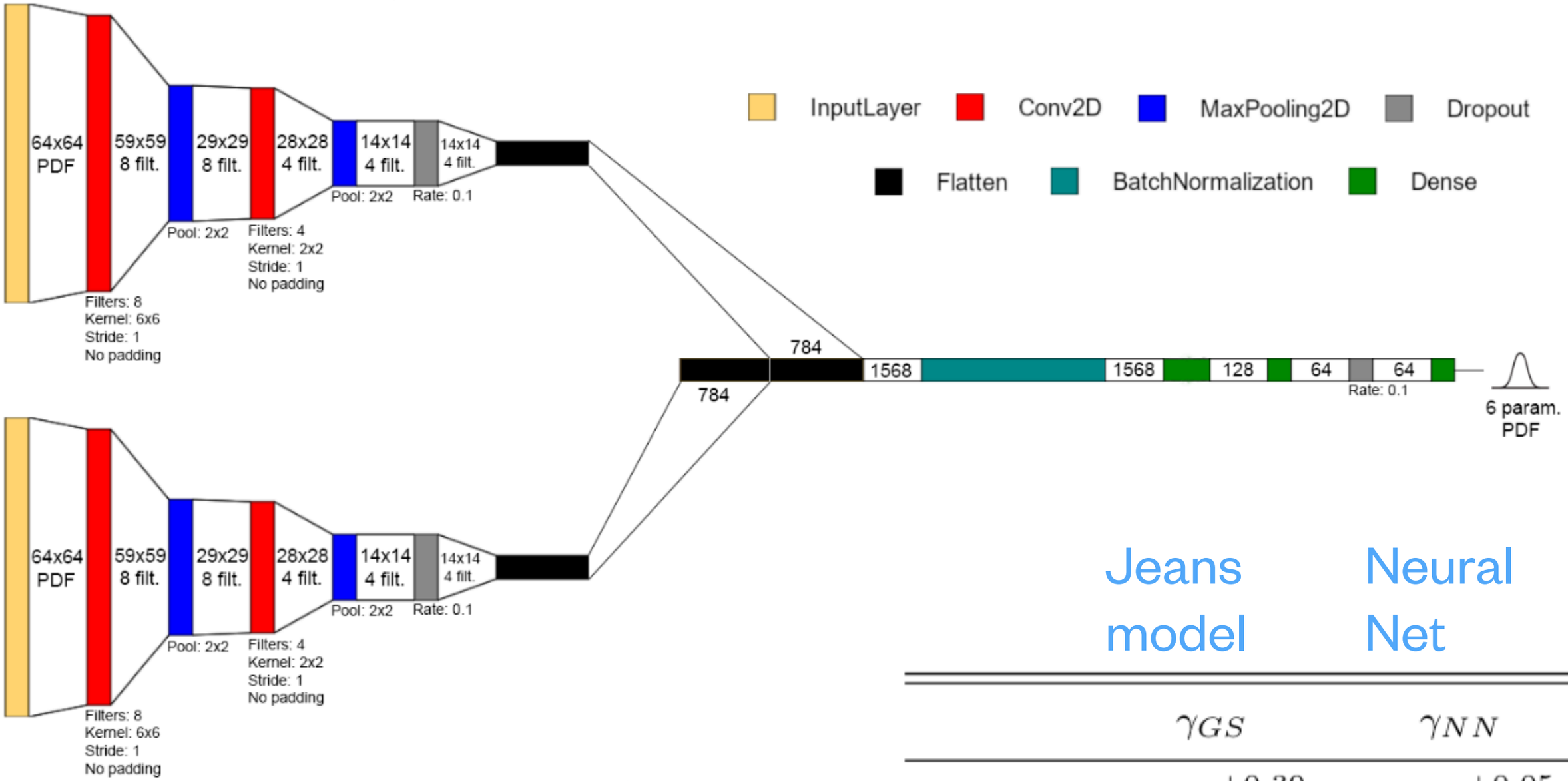
The network layout schematic:



Expósito-Márquez et al (2023)

Determining mass profiles using deep networks

The network layout schematic:



Results:

	Jeans model γ_{GS}	Neural Net γ_{NN}
Carina	$-1.23^{+0.39}_{-0.35}$	$-1.06^{+0.05}_{-0.04}$
Sextans	$-0.95^{+0.25}_{-0.25}$	$-1.25^{+0.25}_{-0.09}$
Fornax	$-0.30^{+0.21}_{-0.28}$	$-0.38^{+0.01}_{-0.02}$
Sculptor	$-0.83^{+0.30}_{-0.25}$	$-1.08^{+0.08}_{-0.04}$

Expósito-Márquez et al (2023)

Tools for deep learning

Toolkits:

Theano (http://deeplearning.net/software/theano/)

Torch (http://torch.ch/) & **pyTorch** (http://pytorch.org/)

TensorFlow (https://pythonhosted.org/nolearn/)

MXNet (http://mxnet.incubator.apache.org/)

Interfaces:

Lasagne (http://lasagne.readthedocs.io/en/latest/) Used by Kim & Brunner (2015)

nolearn (https://pythonhosted.org/nolearn/) Used by Gieseke et al (2017)

Keras (https://keras.io/) Probably the simplest interface today

Example use - Keras

Sequential setup:

Sequential →

Decide on loss function, optimizer, activation functions and various parameters

```
model.compile()
model.fit()
model.predict()
```

Example use - Keras

Sequential setup:

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
```

Example use - Keras

Using the library of stellar spectra from last lecture

```
model = Sequential()

model.add(Dense(n_data, input_dim=n_data,
                activation='relu'))
model.add(Dense(250, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(1, activation='linear'))
```

Example use - Keras

Using the library of stellar spectra from last lecture

```
model = Sequential()

model.add(Dense(n_data, input_dim=n_data,
                activation='relu'))
model.add(Dense(250, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(1, activation='linear'))
```

Predict MK class

Example use - Keras

Using the library of stellar spectra from last lecture

```
model.compile(optimizer='Adam',
              loss='mse')
```

(you can also add ‘metrics’ to get other evaluations of performance that are not used for training)

Example use - Keras

Using the library of stellar spectra from last lecture

```
model.compile(optimizer='Adam',
              loss='mse')
```

(you can also add ‘metrics’ to get other evaluations of performance that are not used for training)

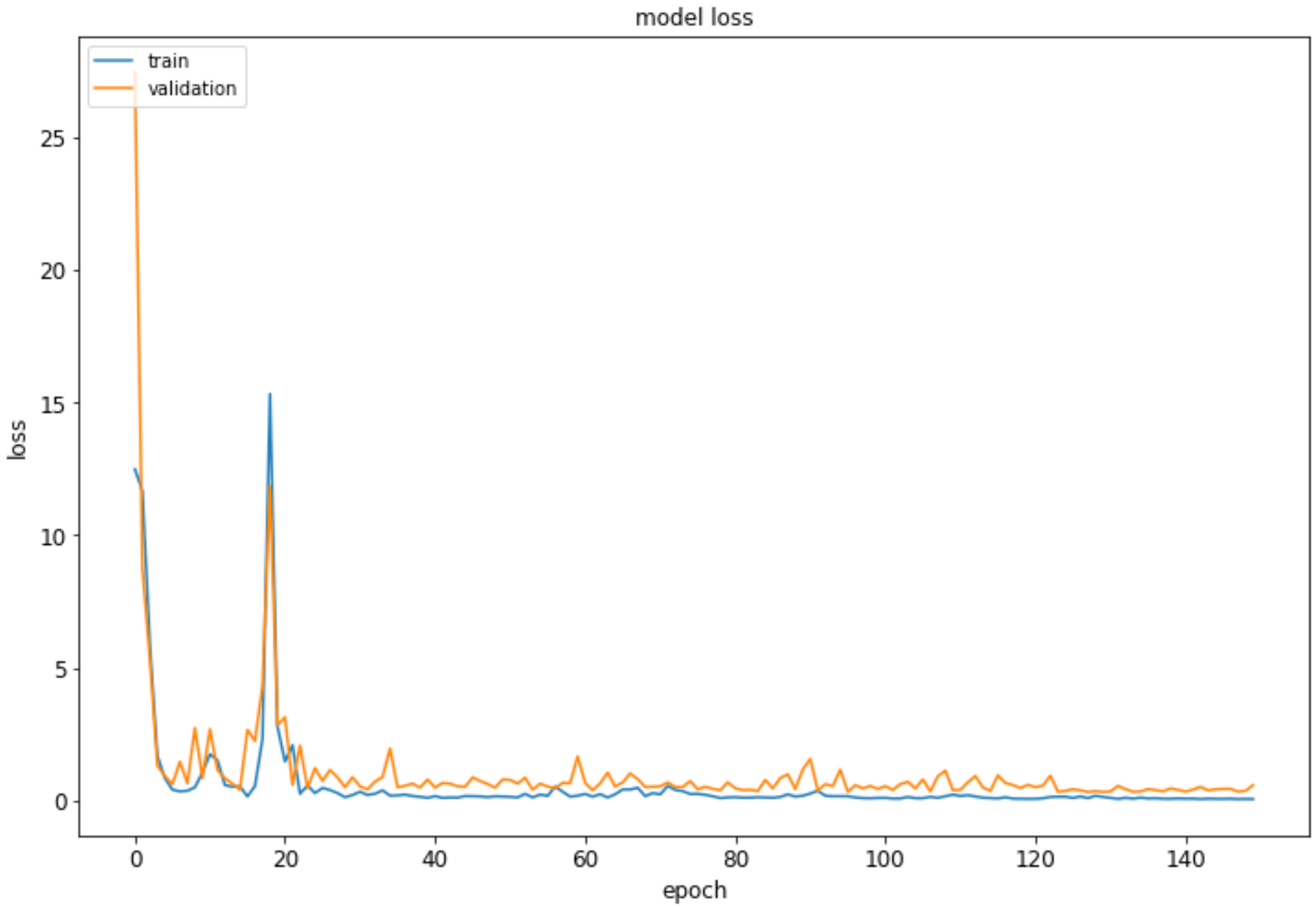
```
history = model.fit(train_X, train_y,
                    validation_data=(test_X, test_y),
                    epochs=150, verbose=0)
```

Example use - Keras

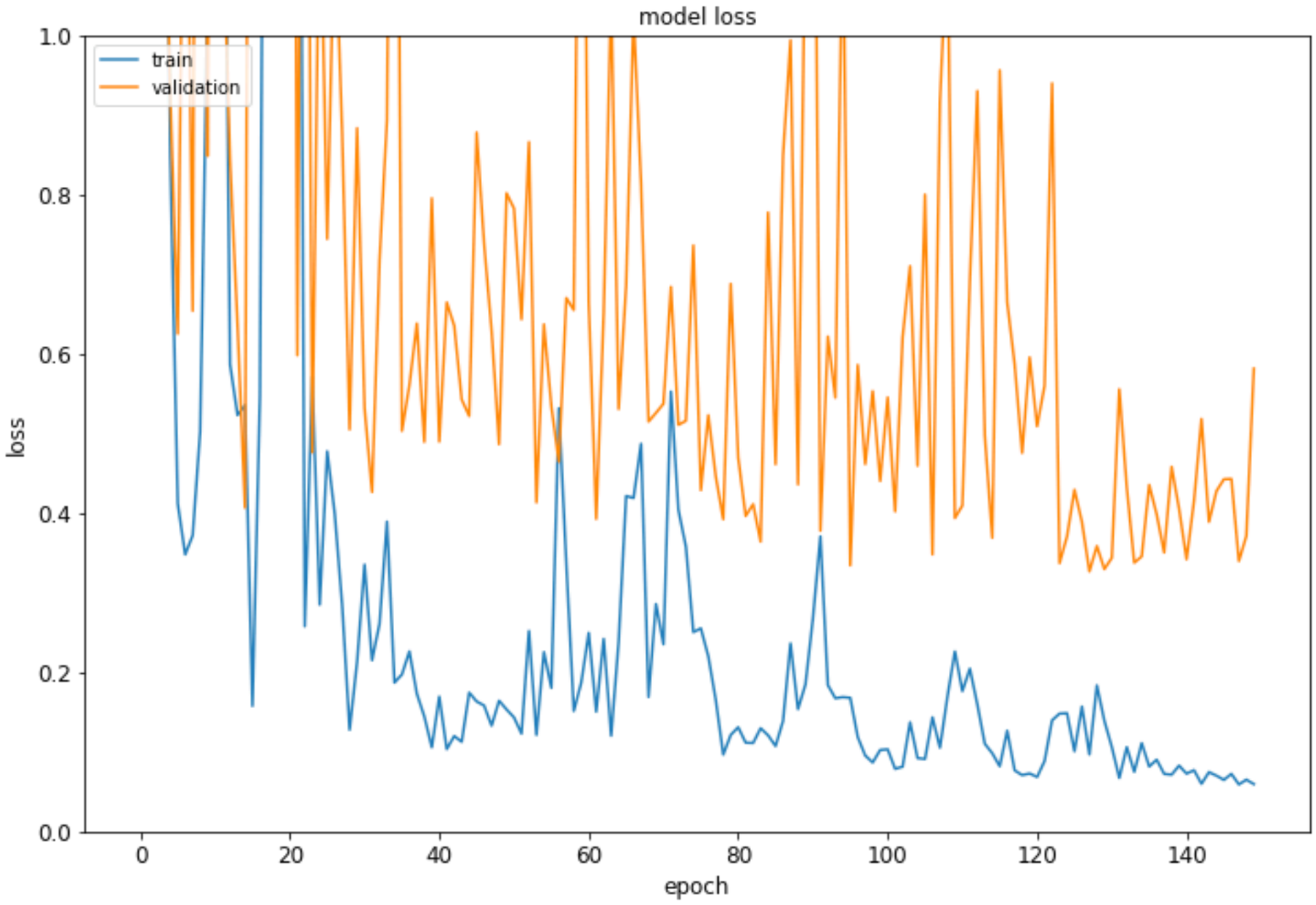
Plotting the training result:

```
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(history.history['loss'])
ax.plot(history.history['val_loss'])
ax.set_title('model loss')
ax.set_ylabel('loss')
ax.set_xlabel('epoch')
ax.legend(['train', 'validation'], loc='upper left')
```

Example use - Keras

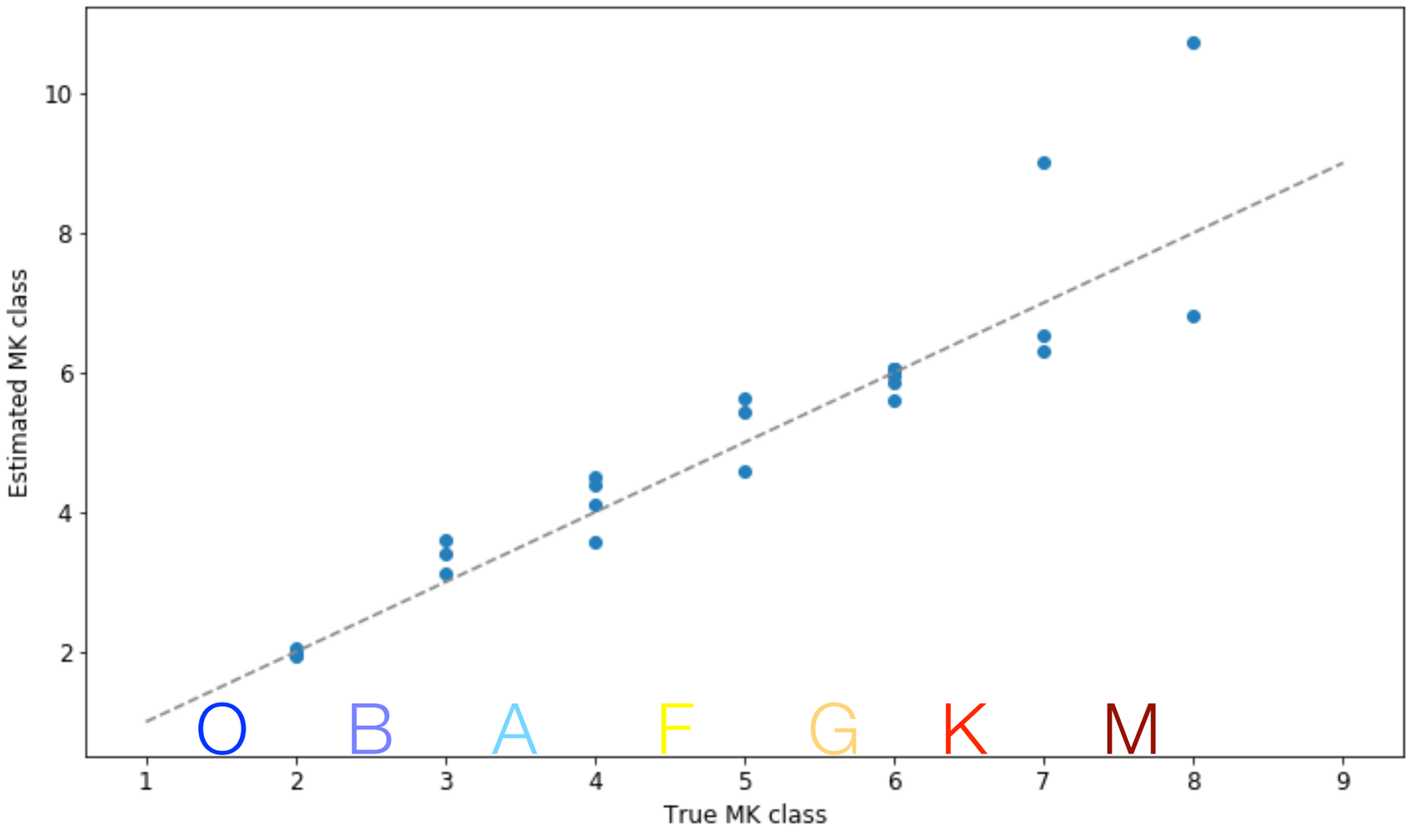


Example use - Keras



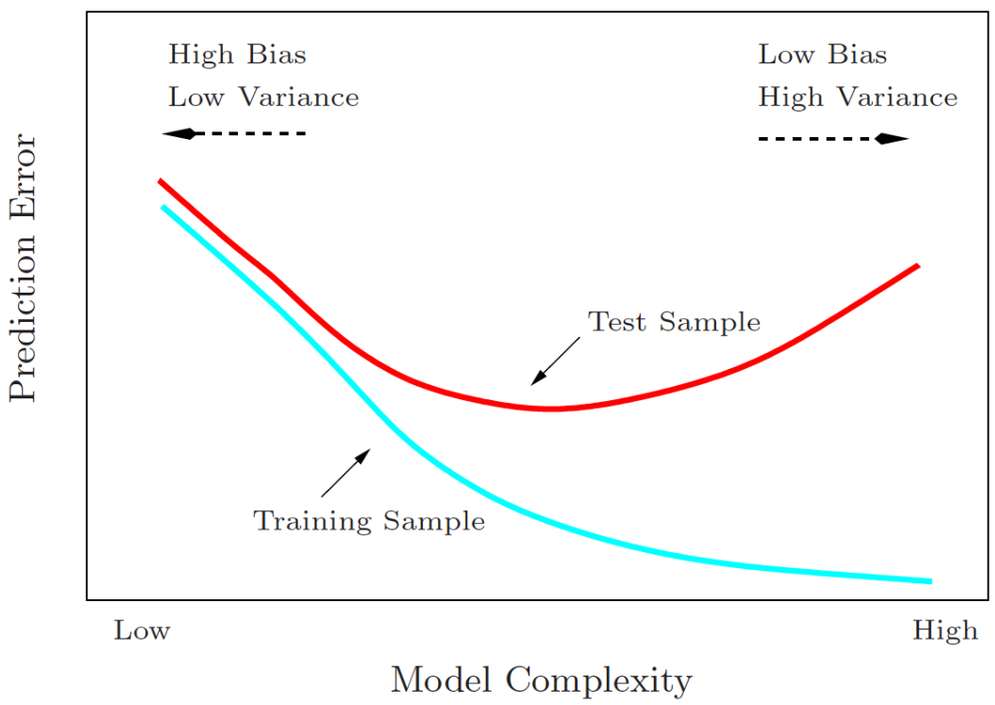
Final result on test set

```
y_pred = model.predict(test_X)
```



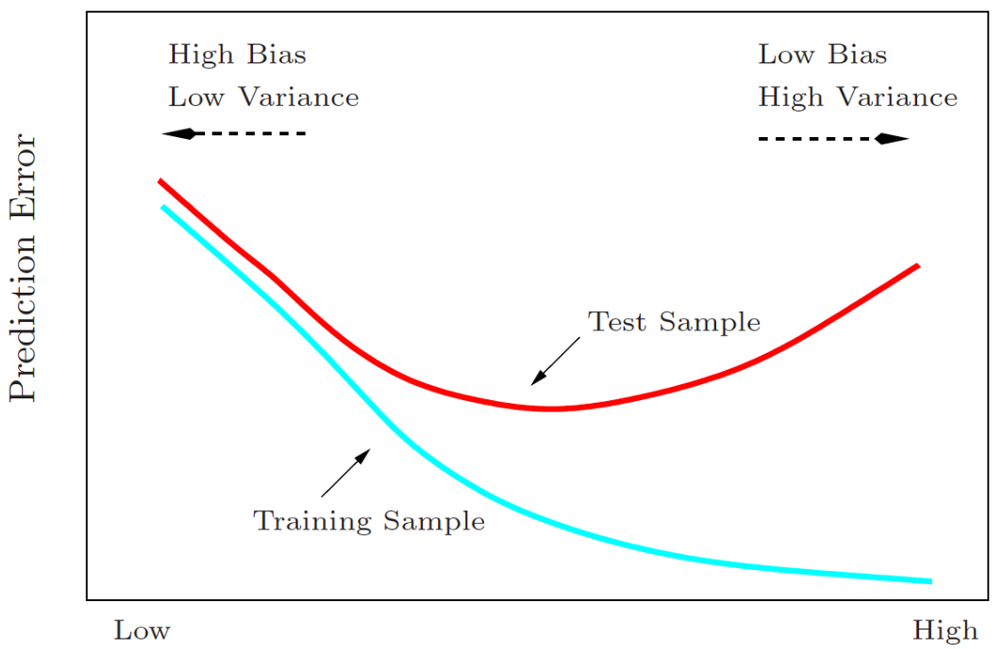
Scatter +/- 0.5 for most classes.

Bias-variance for deep neural networks



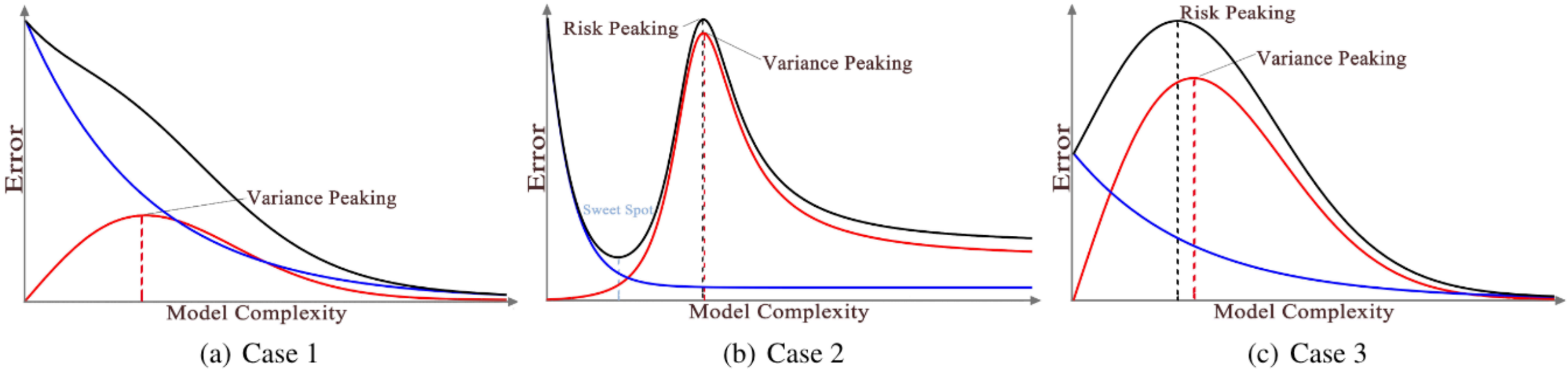
Seemed fine - right?

Bias-variance for deep neural networks



Seemed fine - right?

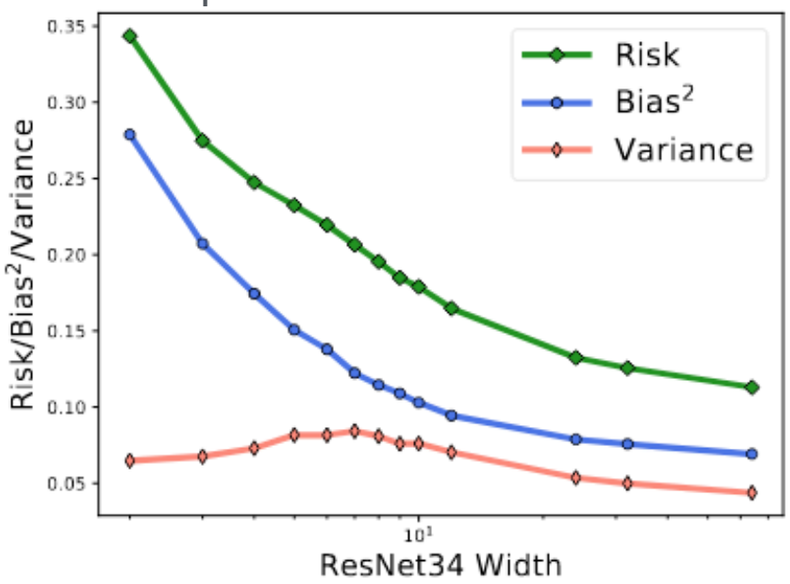
Yang et al (2020, arXiv:2002.11328v3)

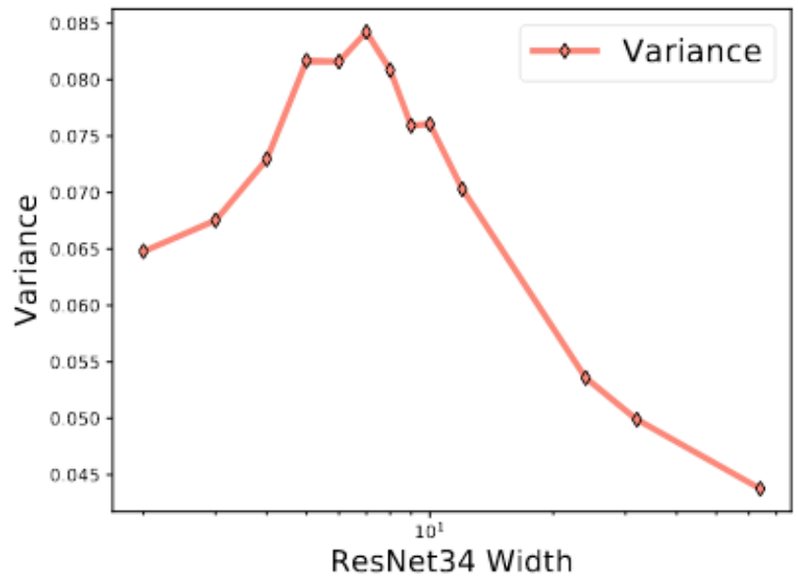


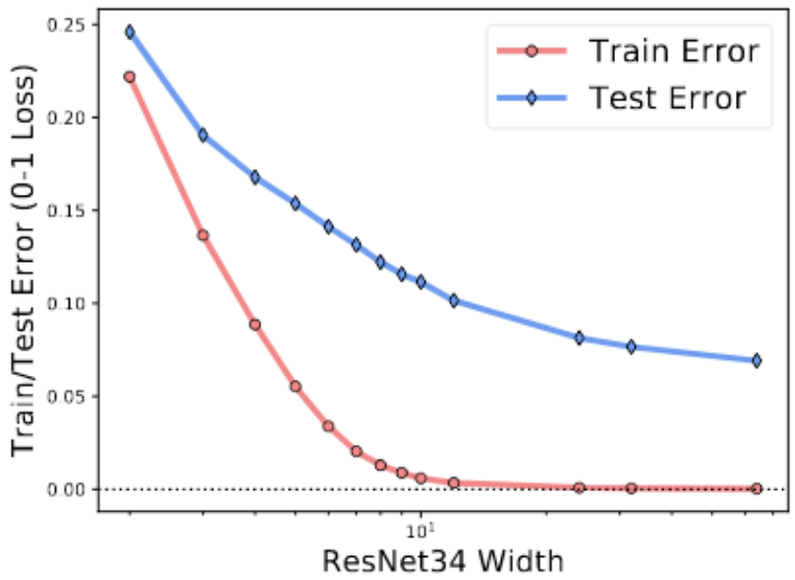
Bias-variance for deep neural networks

Yang et al (2020, arXiv:2002.11328v3)

Empirical runs:







There seems to be fairly broad agreement that deep neural networks have reduced variance when the training sample gets above certain size/network sufficiently complex.

Is this a real, does it signify a change in nature of the learning?
Probably not, but active area of research - likely to be associated to regularisation in the network - see e.g. the Yang et al paper which uses ridge regression to illustrate where this might come from.

Looking back at the course

What should you use?

And when?

Looking back at the course

If you have an idea of a physical model behind, and want to learn something about this:

Regression
Bayesian/frequentist inference
Gaussian mixture models

and other techniques that are easily interpretable

Looking back at the course

If you do not have a (good) model, but plenty of empirical examples, or if you do not care about understanding why something works:

Ensemble methods, in particular random forest
kNNs
Deep Learning networks

among others


That's all Folks!