Computer Science

CSC380: Principles of Data Science

Nonlinear Models

Prof. Jason Pacheco

TA: Enfa Rose George

TA: Saiful Islam Salim

Outline

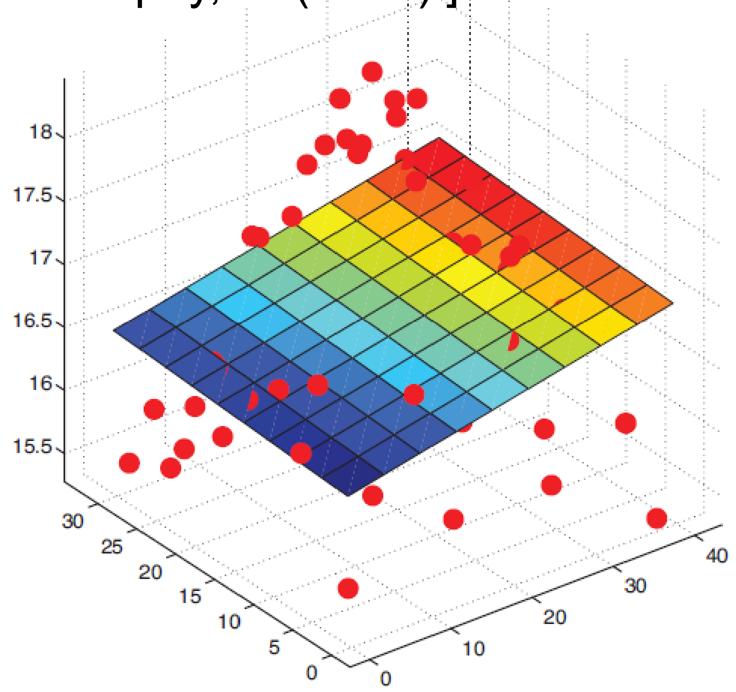
- Basis Functions
- Support Vector Machine Classifier
- Kernels
- Neural Networks

Outline

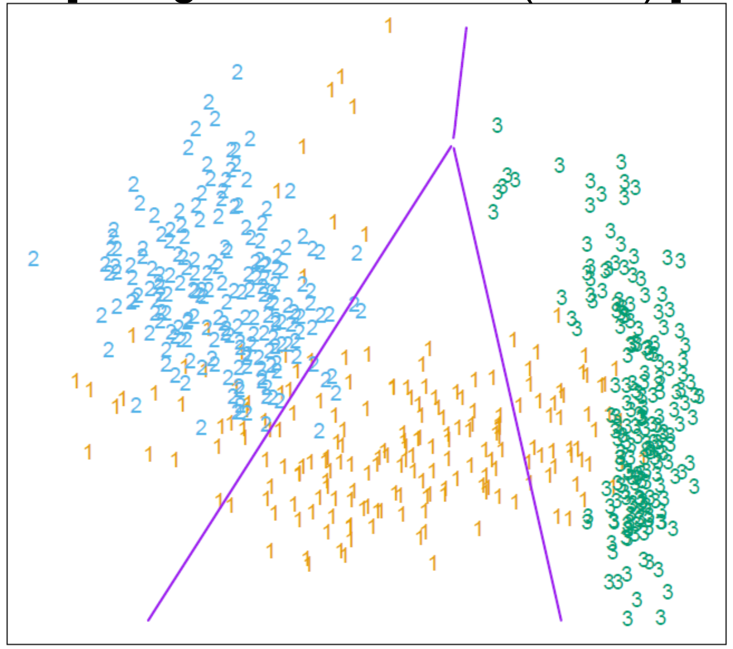
- Basis Functions
- Support Vector Machine Classifier
- Kernels
- Neural Networks

Linear Models

[Image: Murphy, K. (2012)]



[Image: Hastie et al. (2001)]



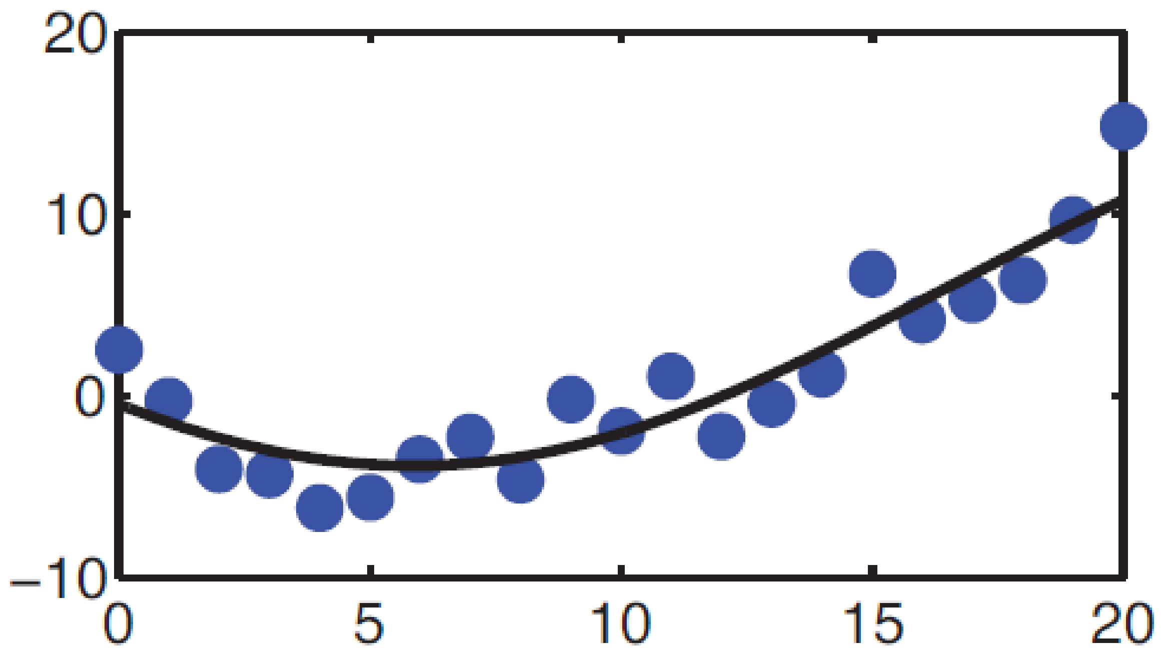
Linear Regression Fit a *linear function* to the data,

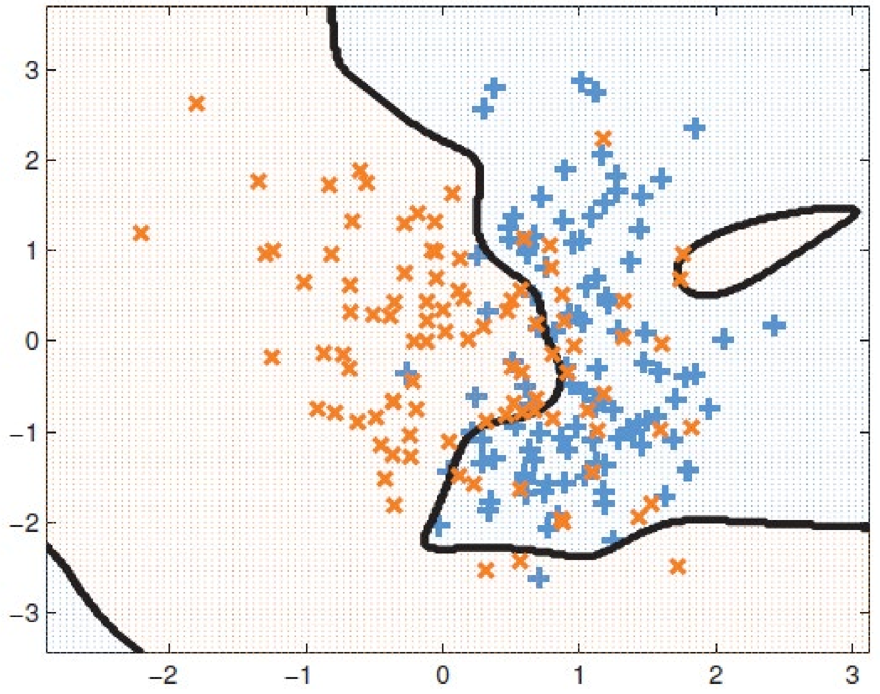
$$y = w^T x + b$$

Logistic Regression Learn a decision boundary that is *linear in the data*,

$$\text{logit}(\sigma(w^T x)) = w^T x$$

Nonlinear Data





What if our data are *not* well-described by a linear function?

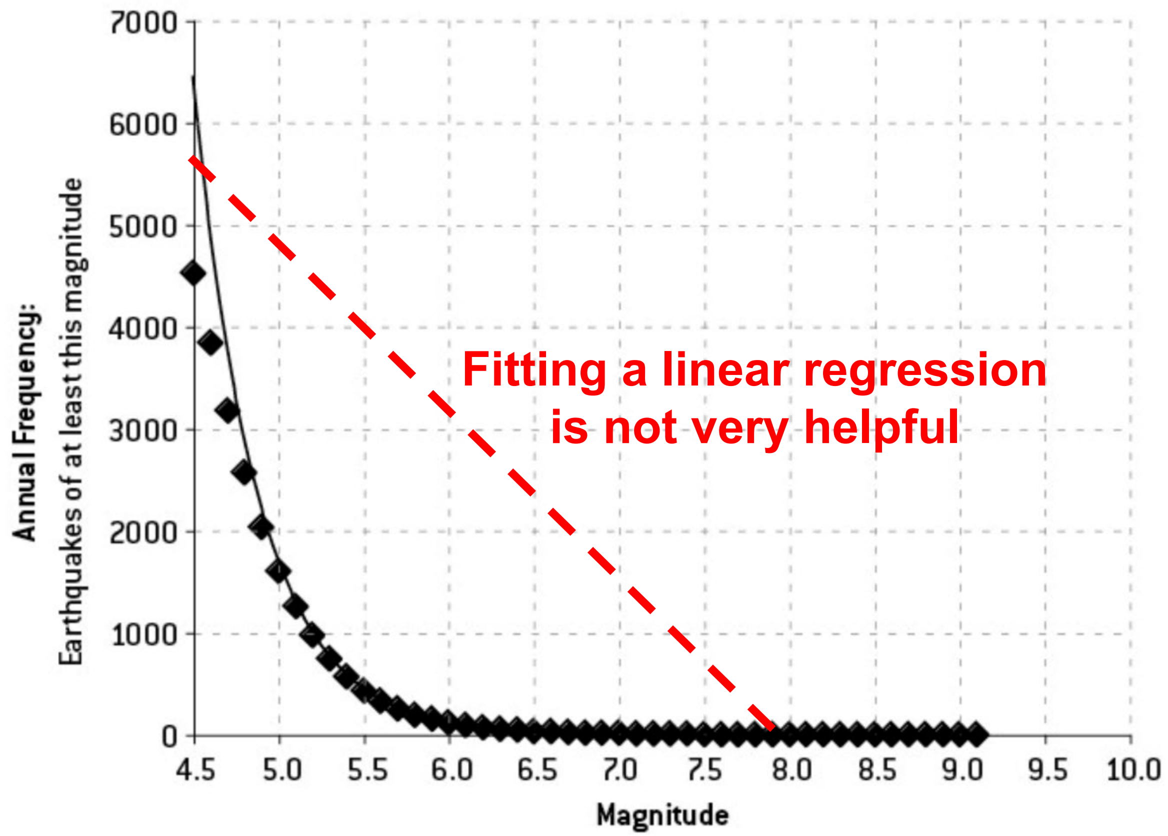
What if classes are *not* *linearly-separable*?

[Source: Murphy, K. (2012)]

Example: Earthquake Prediction

Suppose that we want to predict the number of earthquakes that occur of a certain magnitude. Our data are given by,

FIGURE 5-3A: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964–MARCH 2012

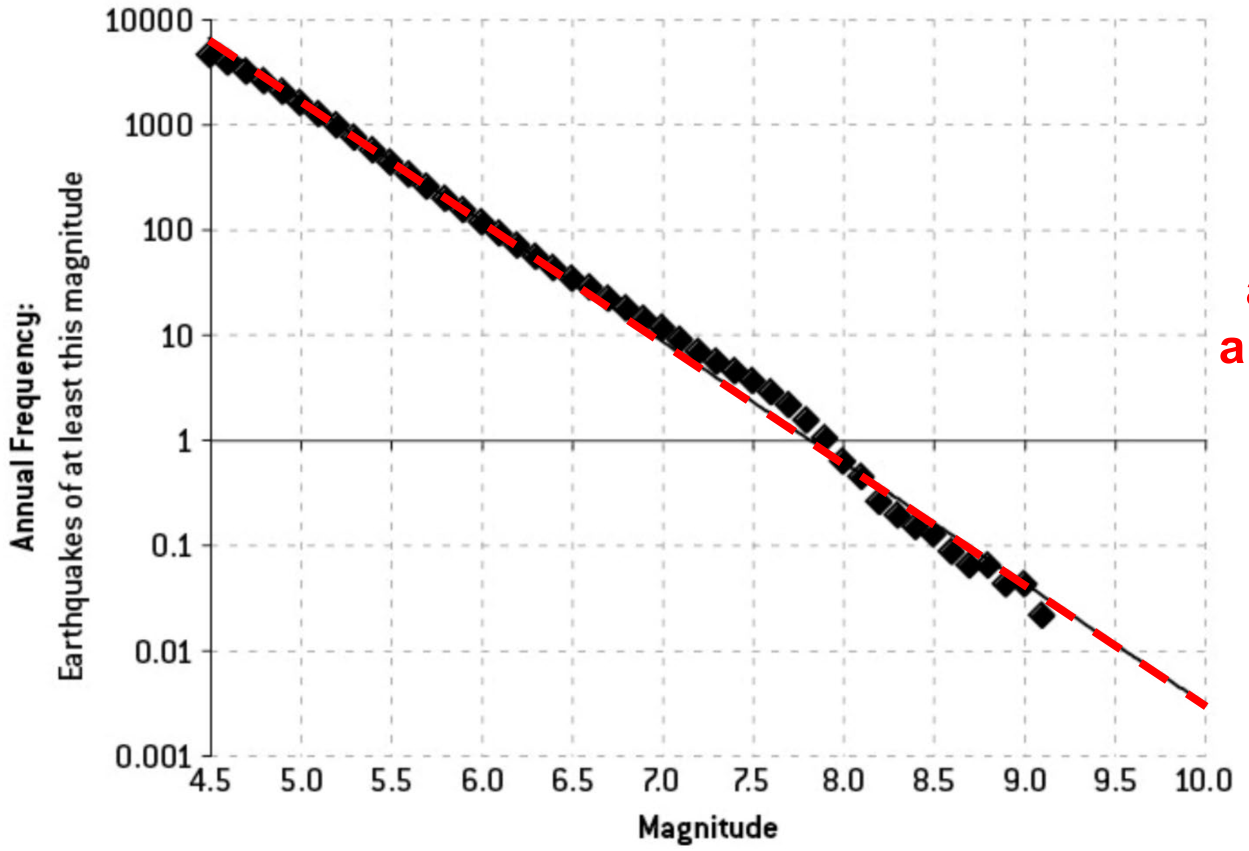


[Source: Silver, N. (2012)]

Example: Earthquake Prediction

Suppose that we want to predict the number of earthquakes that occur of a certain magnitude. Our data are given by,

FIGURE 5-3B: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964—MARCH 2012, LOGARITHMIC SCALE



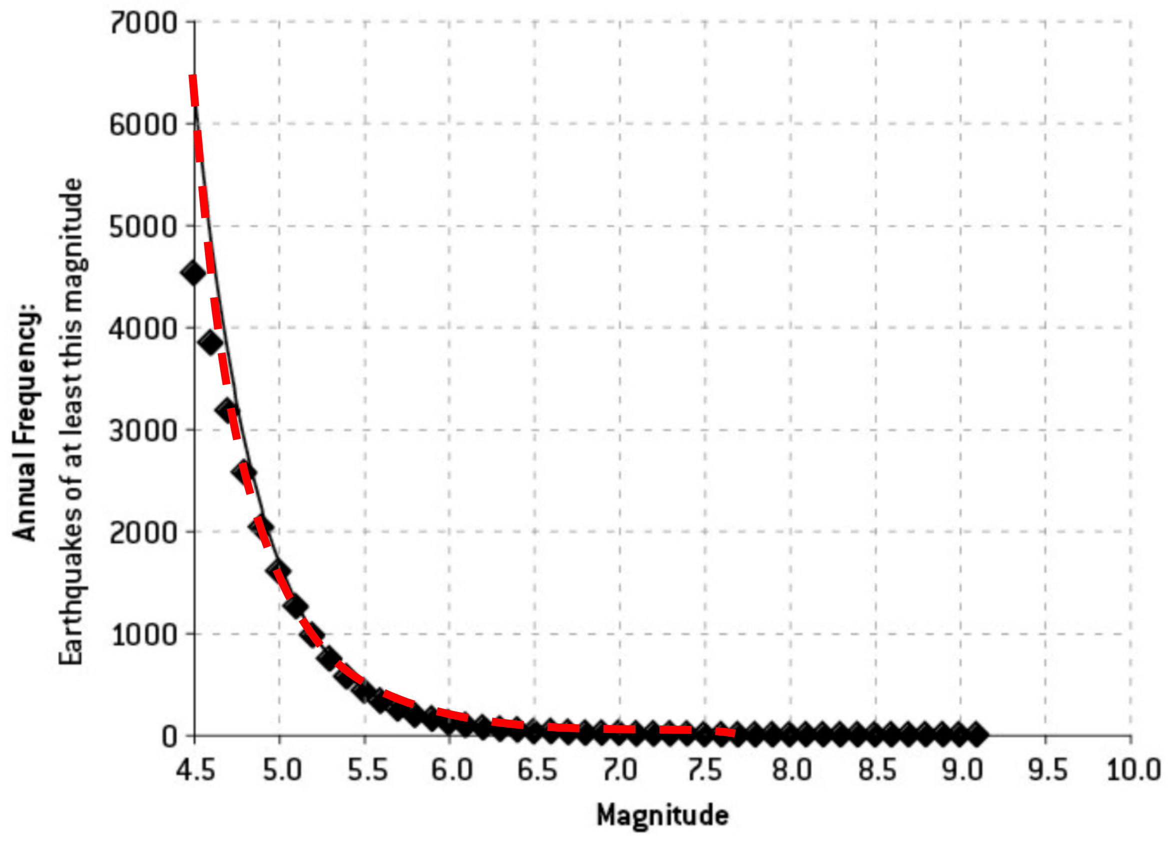
But plotting outputs on a logarithmic scale reveals a strong linear relationship…

[Source: Silver, N. (2012)]

Example: Earthquake Prediction

Suppose that we want to predict the number of earthquakes that occur of a certain magnitude. Our data are given by,

FIGURE 5-3A: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964—MARCH 2012



Idea Instead of fitting ordinary linear regression,

$$y = w^T x$$

First take the logarithm of input values x,

$$y = w^T \log(x)$$

[Source: Silver, N. (2012)]

Basis Functions

- A **basis function** can be any function of the input features X
- Define a set of m basis functions $\phi_1(x), \ldots, \phi_m(x)$
- Fit a linear regression model in terms of basis functions,

$$y = \sum_{i=1}^{m} w_i \phi_i(x) = w^T \phi(x)$$

- Regression model is *linear in the basis transformations*
- Model is *nonlinear in the data X*

Common “All-Purpose” Basis Functions

- Linear basis functions recover the original linear model,

$$\phi_m(x) = x_m$$

Returns mth dimension of X

- Quadratic $\phi_m(x) = x_j^2$ or $\phi_m(x) = x_j x_k$ capture 2nd order interactions
- An order p polynomial $\phi \rightarrow x_d, x_d^2, \ldots, x_d^p$ captures higher-order nonlinearities (but requires *O(d^p)* parameters)
- Nonlinear transformation of single inputs,

$$\phi \rightarrow (\log(x_j), \sqrt{x_j}, \ldots)$$

- An indicator function specifies a region of the input,

$$\phi_m(x) = I(L_m \leq x_k < U_m)$$

`sklearn.preprocessing`.PolynomialFeatures

degree : *int or tuple (min_degree, max_degree), default=2*

If a single int is given, it specifies the maximal degree of the polynomial features. If a tuple `(min_degree, max_degree)` is passed, then `min_degree` is the minimum and `max_degree` is the maximum polynomial degree of the generated features. Note that `min_degree=0` and `min_degree=1` are equivalent as outputting the degree zero term is determined by `include_bias`.

interaction_only : *bool, default=False*

If `True`, only interaction features are produced: features that are products of at most `degree` *distinct* input features, i.e. terms with power of 2 or higher of the same input feature are excluded:

- included: `x[0]`, `x[1]`, `x[0] * x[1]`, etc.
- excluded: `x[0] ** 2`, `x[0] ** 2 * x[1]`, etc.

include_bias : *bool, default=True*

If `True` (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

order : *{'C', 'F'}, default='C'*

Order of output array in the dense case. `'F'` order is faster to compute, but may slow down subsequent estimators.

Example: Polynomial Basis Functions

Create three two-dimensional data points [0,1], [2,3], [4,5]:

```
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
```

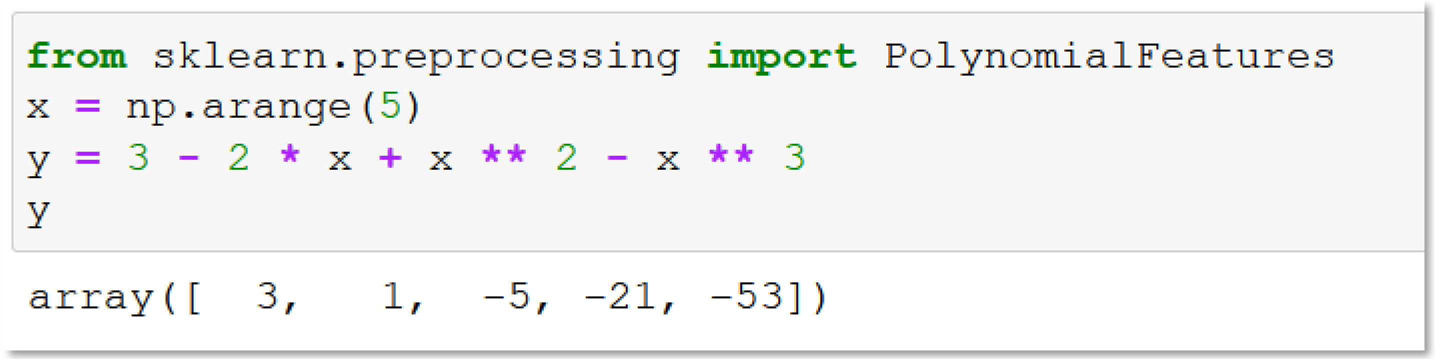
Compute quadratic features $(1, x_1, x_2, x_1^2, x_1 x_2, x_2^2)$,

```
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   0.,   0.,   1.],
       [  1.,   2.,   3.,   4.,   6.,   9.],
       [  1.,   4.,   5.,  16.,  20.,  25.]])
```

These are now our new data and ready to fit a model…

Example: Polynomial Regression

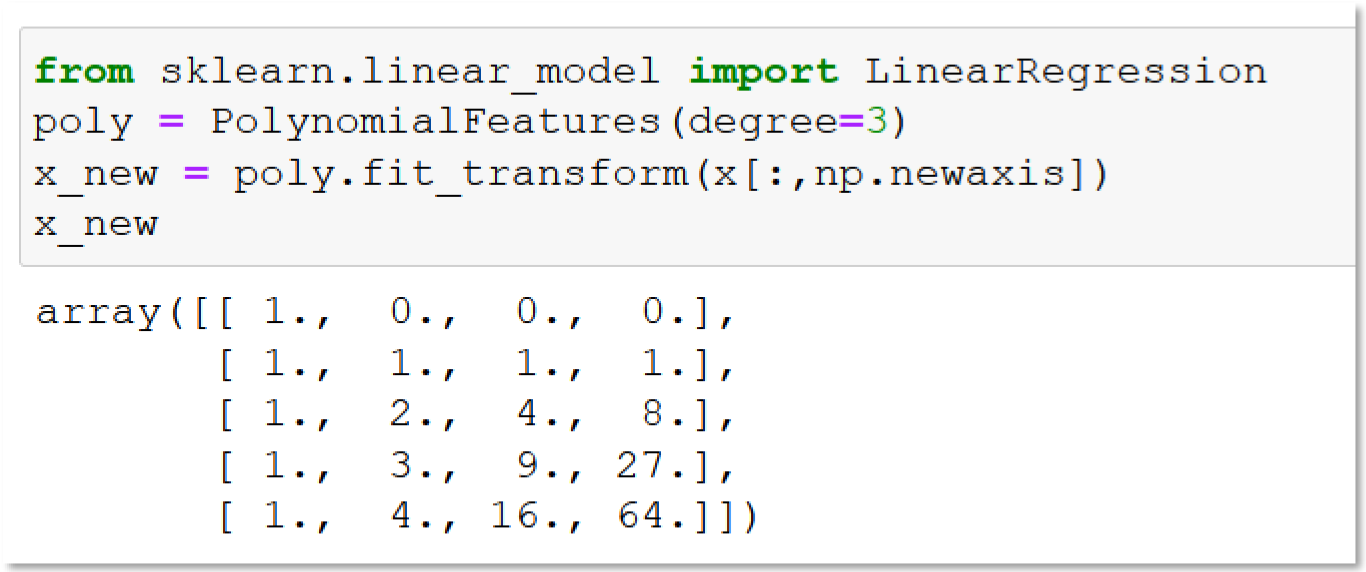
Create a 3-rd order polynomial (cubic) regression,



```
from sklearn.preprocessing import PolynomialFeatures
x = np.arange(5)
y = 3 - 2 * x + x ** 2 - x ** 3
y
```

```
array([  3,   1,  -5, -21, -53])
```

Create cubic features $(1, x, x^2, x^3)$,

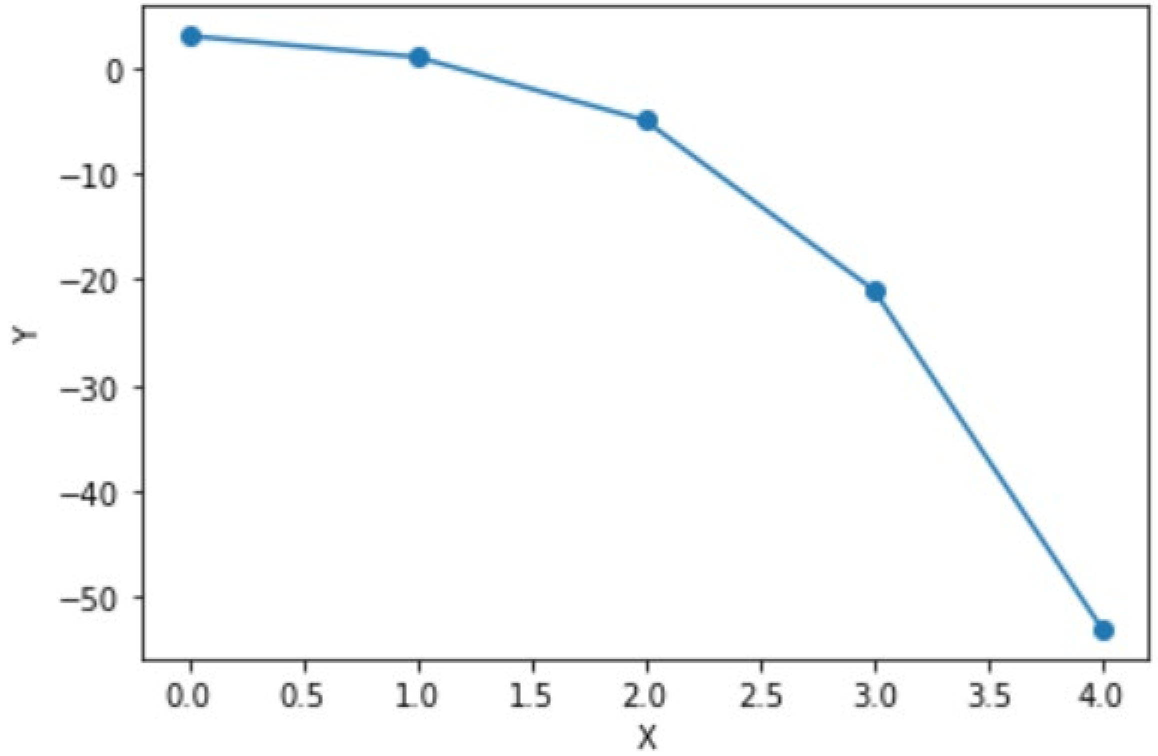


```
from sklearn.linear_model import LinearRegression
poly = PolynomialFeatures(degree=3)
x_new = poly.fit_transform(x[:,np.newaxis])
x_new
```

```
array([[ 1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  2.,  4.,  8.],
       [ 1.,  3.,  9., 27.],
       [ 1.,  4., 16., 64.]])
```

Example: Polynomial Regression

```
model = LinearRegression(fit_intercept=False).fit(x_new, y)
ypred = model.predict(x_new)
plt.scatter(x,y)
plt.plot(x,ypred,'-')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Linear Regression

Recall the ordinary least squares solution is given by,

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1D} \\ 1 & x_{21} & \dots & x_{2D} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{N1} & \dots & x_{ND} \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \qquad w^{\text{OLS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Design Matrix
(each training input on a column)

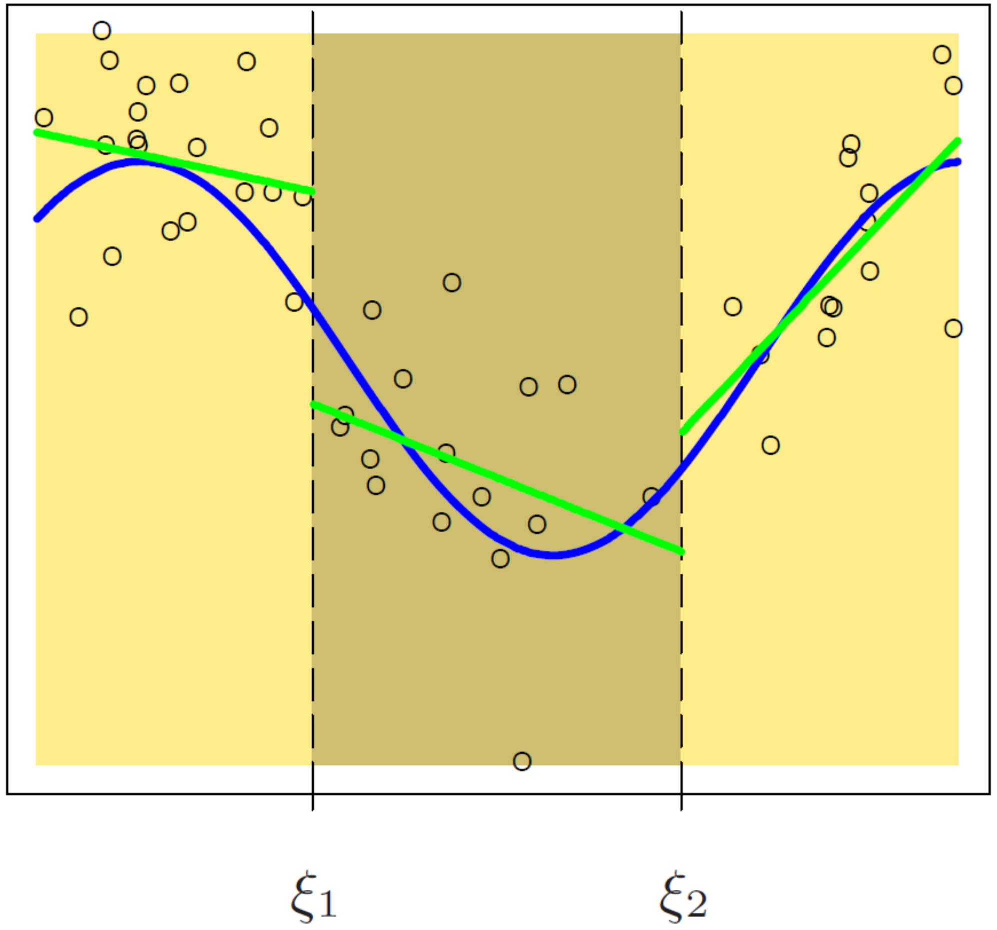
Vector of
Training labels

Can similarly solve in terms of basis functions,

$$\mathbf{\Phi} = \begin{pmatrix} 1 & \phi_1(x_1) & \dots & \phi_M(x_1) \\ 1 & \phi_1(x_2) & \dots & \phi_M(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \phi_1(x_N) & \dots & \phi_M(x_N) \end{pmatrix} \qquad w^{\text{OLS}} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{y}$$

Example: Piecewise Linear Regression

[Source: Hastie et al. (2001)]



Regression lines are discontinuous at boundary points

Decompose the input space into 3 regions with indicator basis functions,

$$\phi_1(x) = I(x < \xi_1)$$
$$\phi_2(x) = I(\xi_1 \leq x < \xi_2)$$
$$\phi_3(x) = I(\xi_2 \leq x)$$

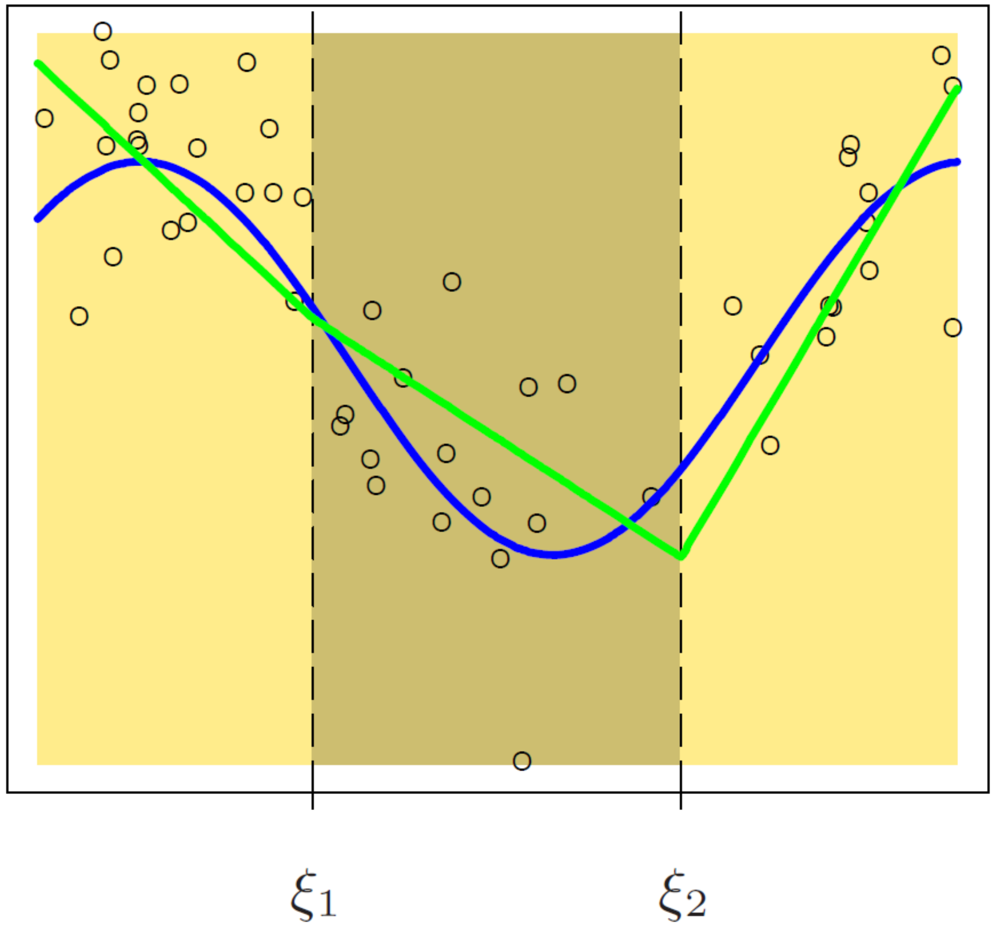
Fit linear regression model,

$$y = w_1\phi_1(x) + w_2\phi_2(x) + w_3\phi_3(x)$$

Effectively fits 3 linear regressions independently to data in each region

Example: Piecewise Linear Regression

[Source: Hastie et al. (2001)]



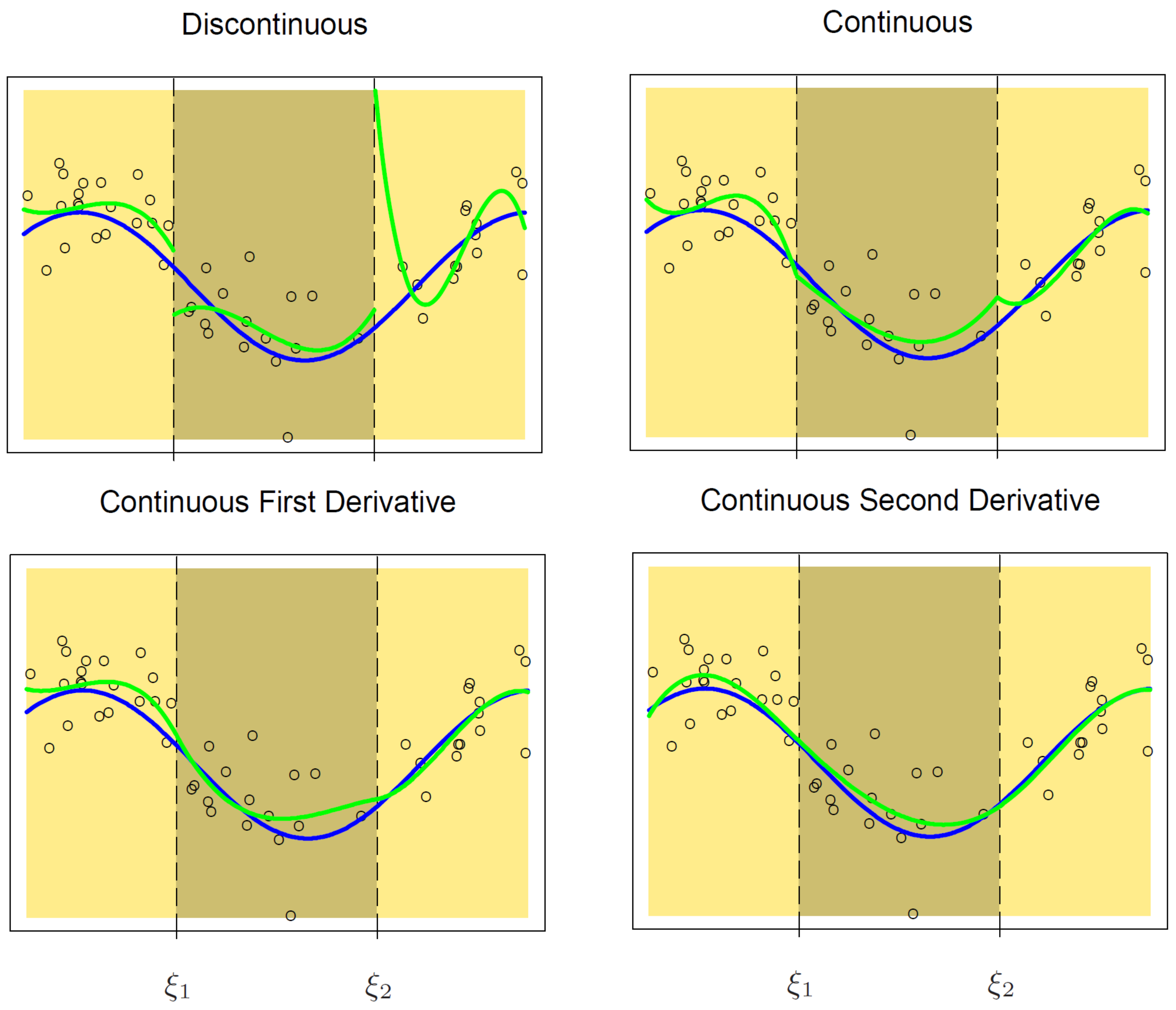
Enforce constraint that lines agree at boundary points,

$$\phi_1(x) = 1$$
$$\phi_2(x) = x$$
$$\phi_3(x) = (x - \xi_1)_+$$
$$\phi_4(x) = (x - \xi_2)_+$$

Where $(\ldots)_+$ means the positive part

An improvement, but generally prefer *smoother* functions…

[Source: Hastie et al. (2001)]



Replace linear basis functions with polynomial,

$$\phi_1(x) = 1 \quad \phi_2(x) = x$$

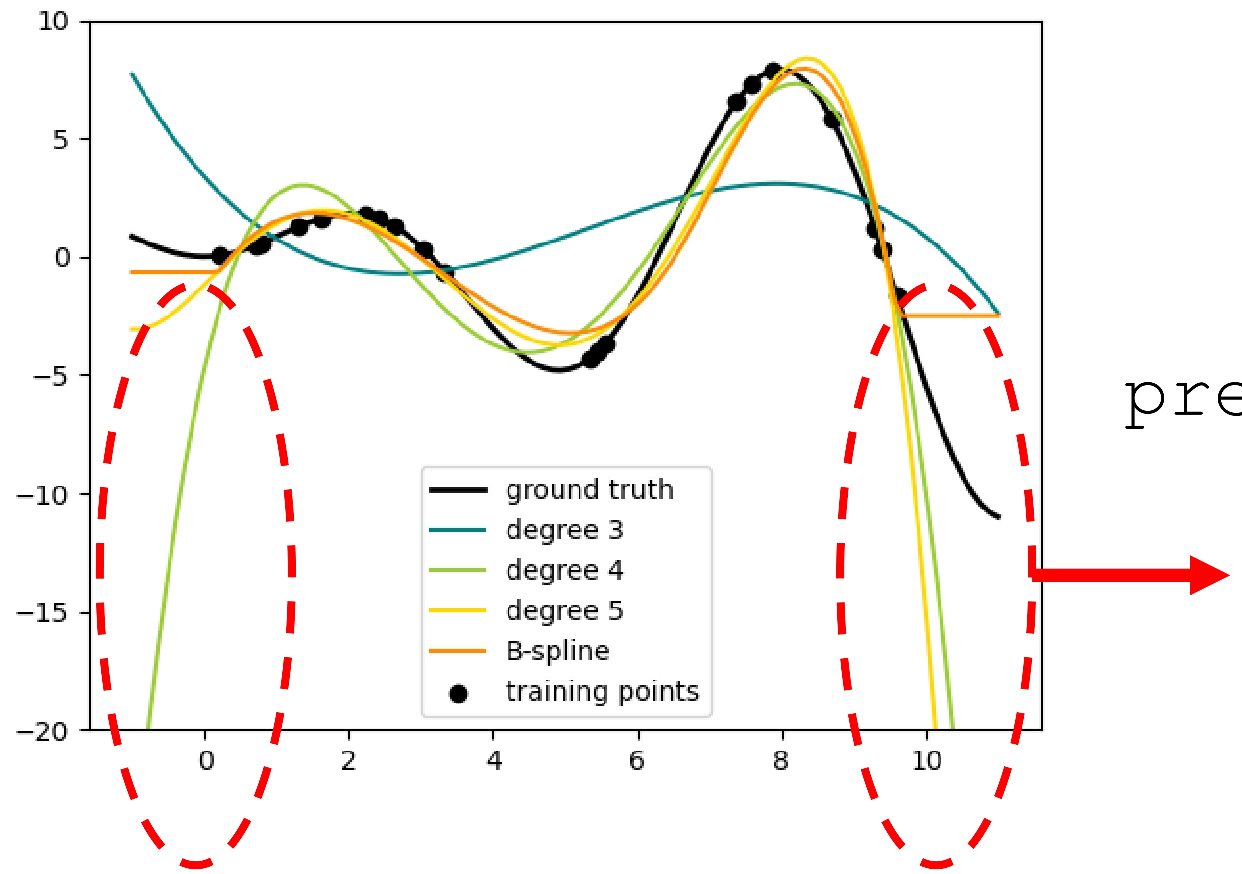
$$\phi_3(x) = x^2 \quad \phi_4(x) = x^3$$

$$\phi_5(x) = (x - \xi_1)_+^3$$

$$\phi_6(x) = (x - \xi_2)_+^3$$

Additional constraints ensure smooth 1st and 2nd derivatives at boundaries

Polynomial Splines



These piecewise regression functions are called *splines*

Supported in Scikit-Learn
`preprocessing.SplineTransformer`

Caution Polynomial basis functions often yield poor out-of-sample predictions with higher order producing more extreme predictions

Data Preprocessing

- Generally the first step in data science involves *preprocessing* or transforming data in some way
  - Filling in missing values (imputation)
  - Centering / normalizing / Z-scoring data
  - Etc.
- We then fit our models to this preprocessed data
- One way to view preprocessing is simply as computing some basis function $\phi(x)$, nothing more

Basis Functions

PROs

- More flexible modeling that is nonlinear in the original data
- Increases model complexity and expressivity

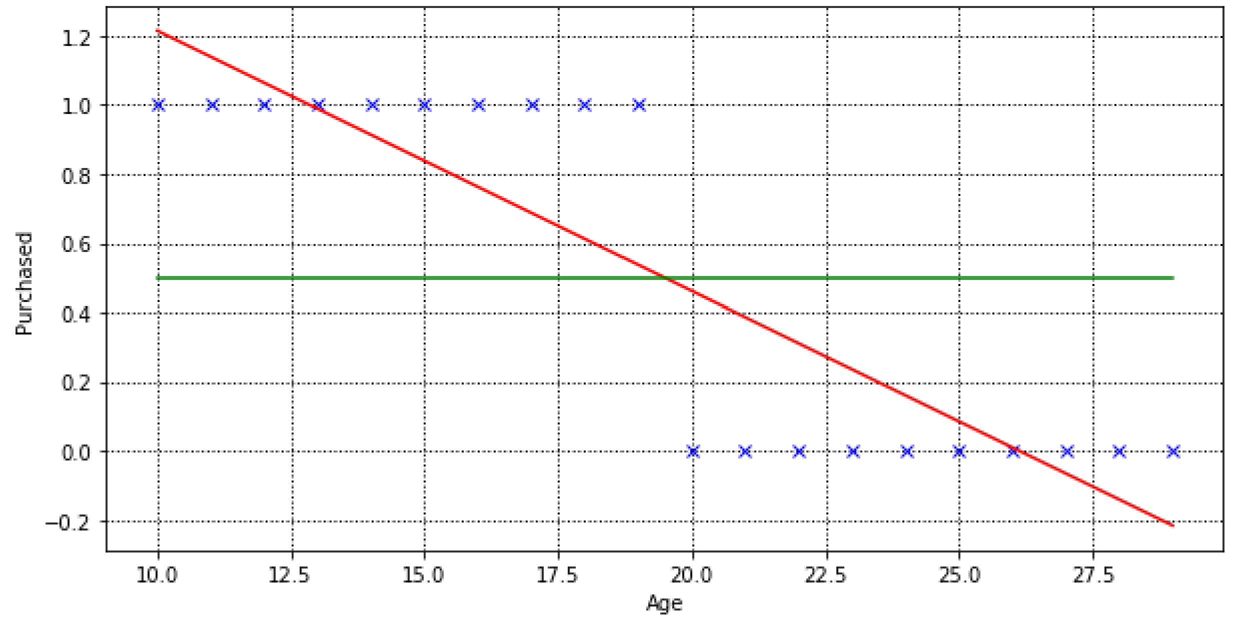
CONs

- Typically requires more parameters to be learned
- More sensitive to overfitting training data
- Requires more regularization to avoid overfitting
- Need to find *good* basis functions (feature engineering)

Outline

- Basis Functions
- Support Vector Machine Classifier
- Kernels
- Neural Networks

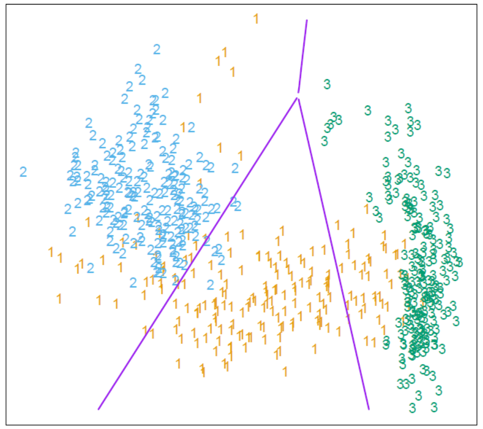
Classification as Regression



Recall our linear regression can be used for classification via the rule,

$$\text{Class} = \begin{cases} 0 & \text{if } w^T x < 0.5 \\ 1 & \text{if } w^T x >= 0.5 \end{cases}$$

- This is a *discriminant* function, since it discriminates between classes
- It is a linear function and so is a *linear discriminant*
- Green line is the *decision boundary* (also linear)

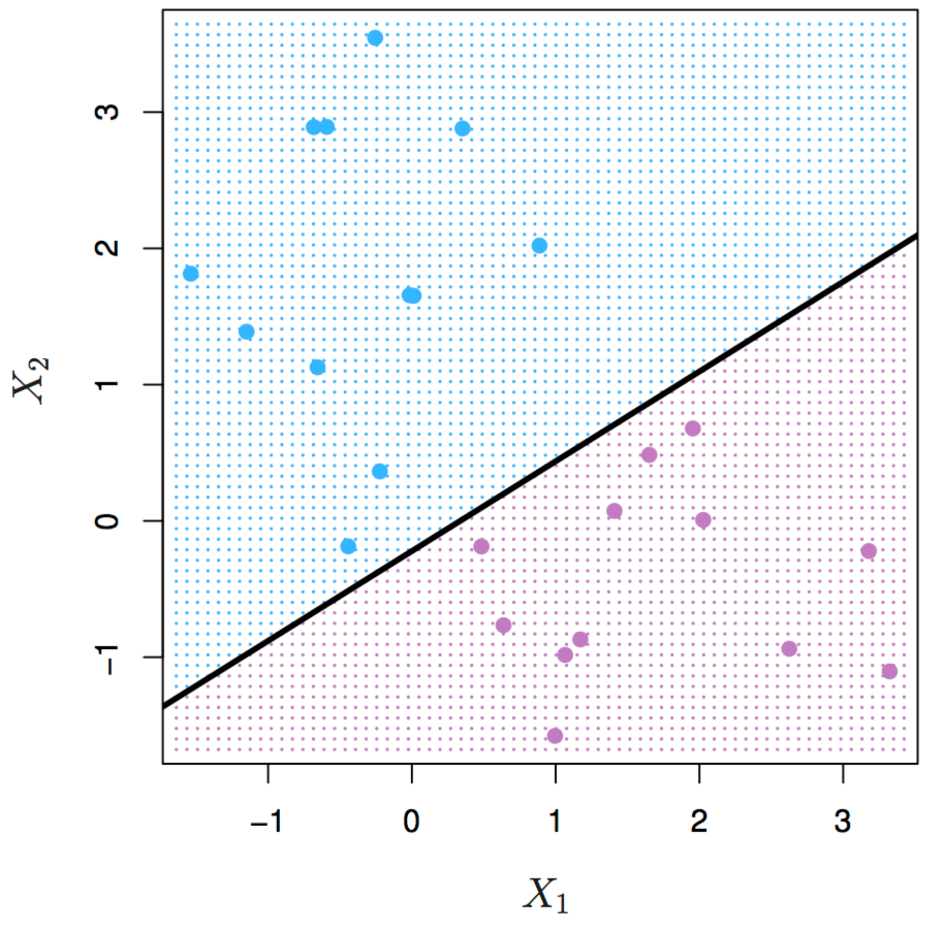


Generalizes to higher-dimensional features

https://towardsdatascience.com/why-linear-regression-is-not-suitable-for-binary-classification-c64457be8e28

Linear Decision Boundary

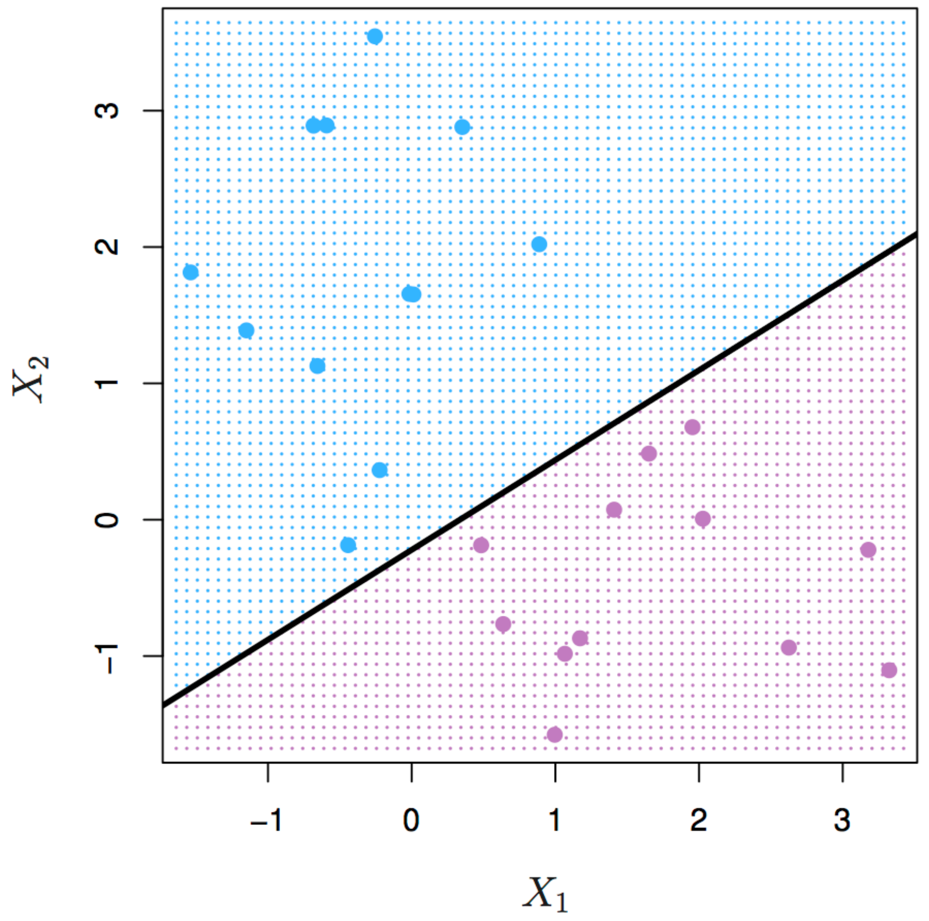
Least squares regression yields decision boundary based on least squares solution…

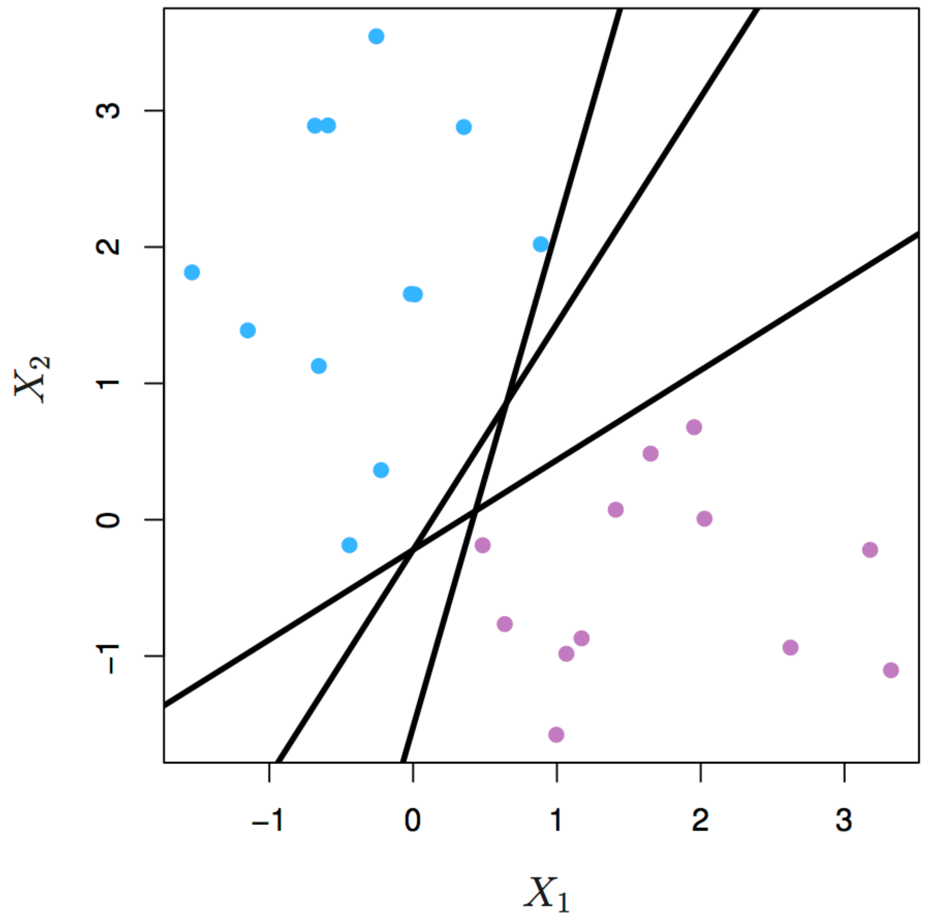


[Source: http://www-bcf.usc.edu/~gareth/ISL/]

Linear Decision Boundary

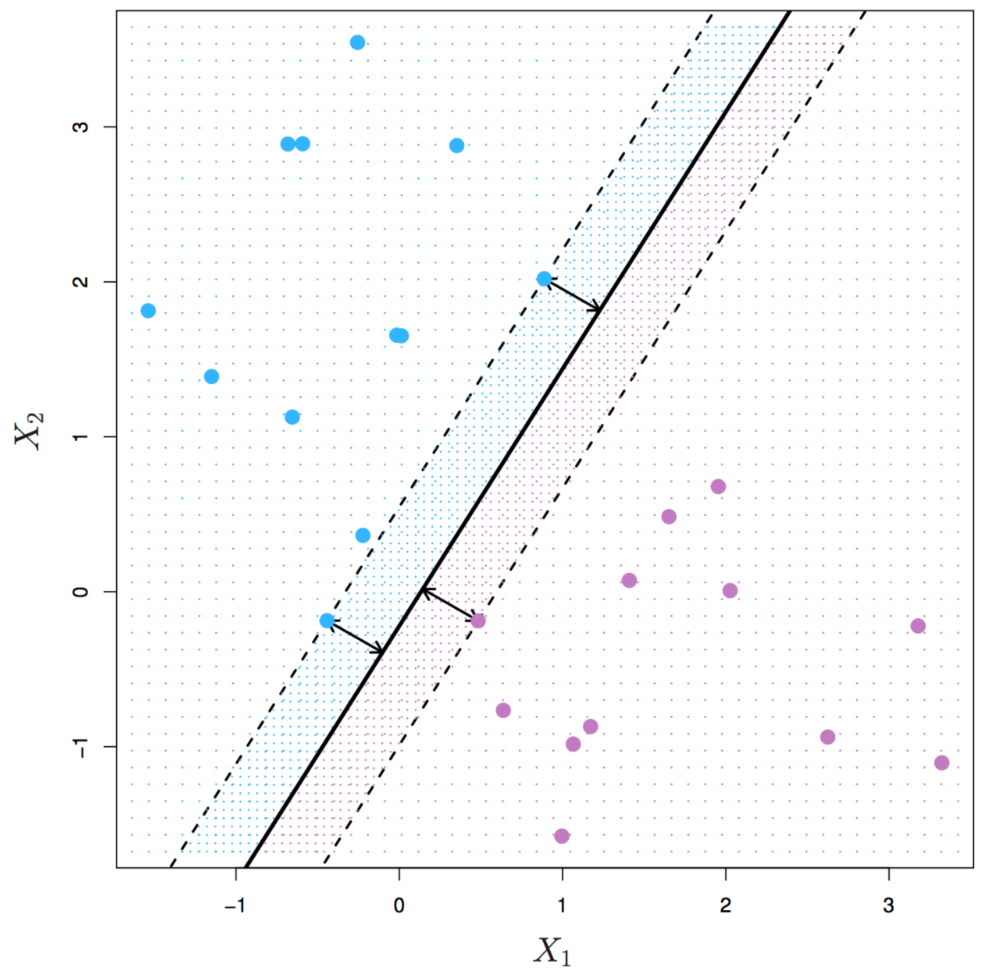
…any boundary that separates classes is equivalently good on training data





[Source: http://www-bcf.usc.edu/~gareth/ISL/]

Classifier Margin



*The **margin** measures minimum distance between each class and the decision boundary*

Observation Decision boundaries with larger margins are more likely to generalize to unseen data

Idea Learn the classifier with the largest margin that still separates the data…

…we call this a *max-margin classifier*

[Source: http://www-bcf.usc.edu/~gareth/ISL/]

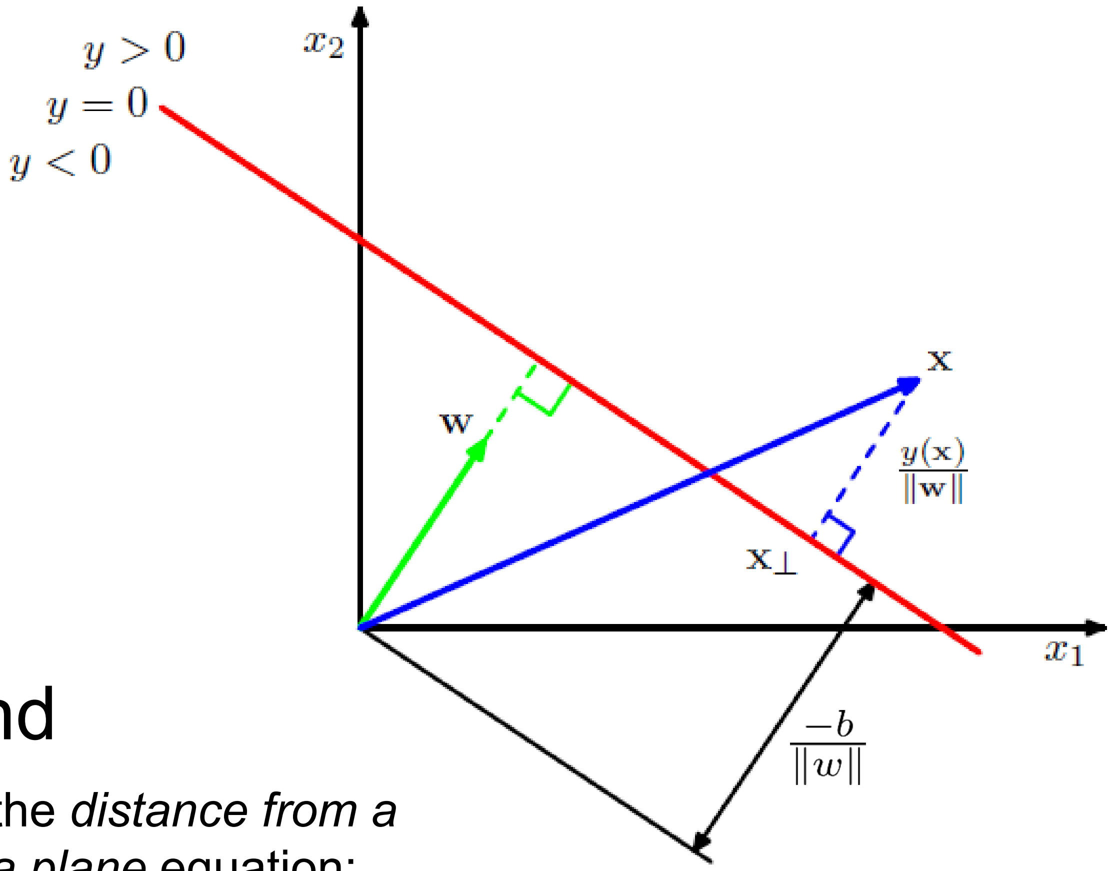
Max-Margin Classifier

Recall that the linear model is given by

$$y(x) = w^T x + b$$

Let classes be $\{-1, 1\}$ so classification rule is,

$$\text{Class} = \begin{cases} -1 & \text{if } y(x) < 0 \\ 1 & \text{if } y(x) >= 0 \end{cases}$$



Decision boundary is now at y(x) = 0 and distance to the margin is,

$$\frac{y(x)}{\|w\|}$$

Known as the *distance from a point to a plane* equation: wiki/Distance_from_a_point_to_a_plane

Where the norm of the weights is $\|w\| = \sqrt{w^T w} = \sqrt{\sum_i w_i^2}$

Max-Margin Classifier

For training data $\{(x_n, y_n)\}$ we only care about the margin for correctly-classified points where,

$$y_n y(x_n) = y_n(w^T x_n + b) > 0$$

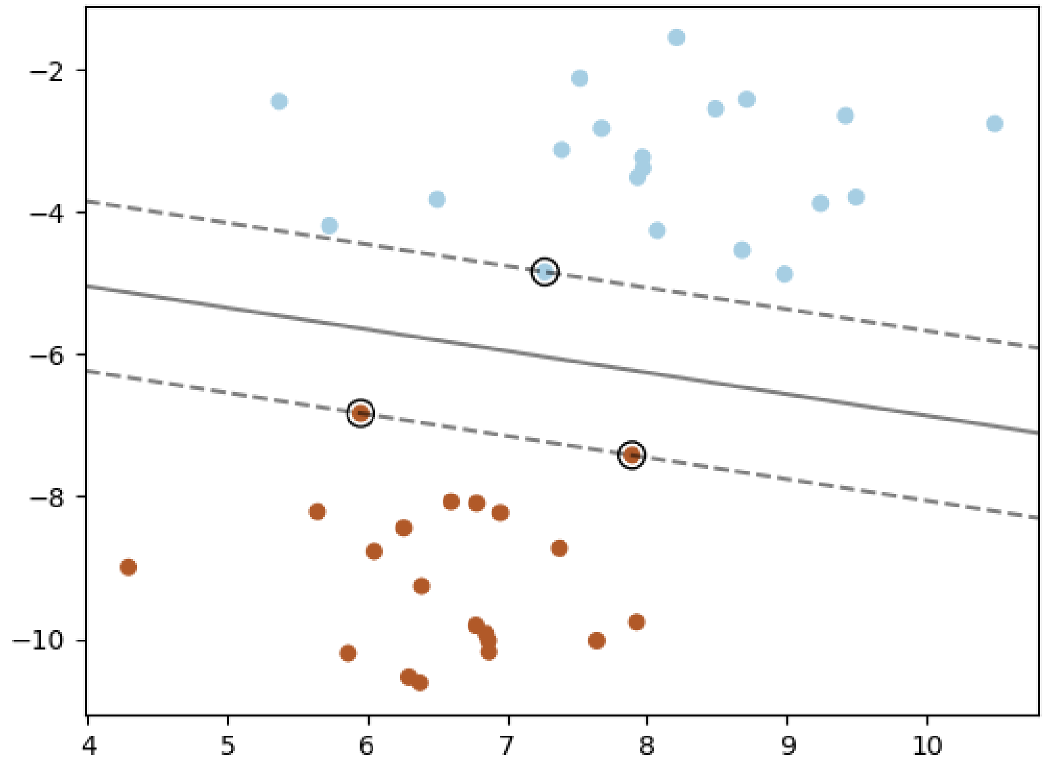
The margin of correctly-classified points is then given by,

$$\frac{y_n y(x_n)}{\|w\|} = \frac{y_n(w^T x_n + b)}{\|w\|}$$

Maximize margin over correctly-classified data points,

$$\arg\max_{w,b} \left\{ \min_n \frac{y_n(w^T x_n + b)}{\|w\|} \right\}$$

Max-Margin Classifier

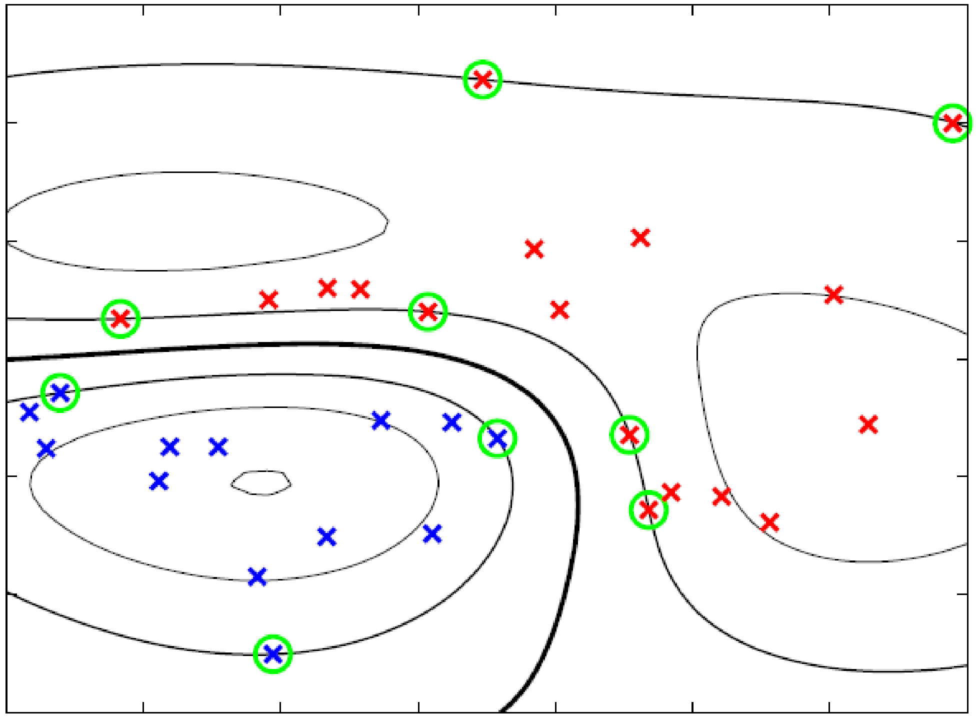


$$\overbrace{\arg\max_{w,b}\left\{\underbrace{\min_n \frac{y_n(w^T x_n + b)}{\|w\|}}_{\text{Minimum margin over all training data}}\right\}}^{\text{Maximize the minimum margin}}$$

Find the parameters (w,b) that **maximize** the **smallest margin** over all the training data

[Source: http://www-bcf.usc.edu/~gareth/ISL/]

Nonlinear Max-Margin Classifier



Just as in the linear models we can introduce basis transformations,

$$y(x) = w^T \phi(x) + b$$

Max-margin learning is similar,

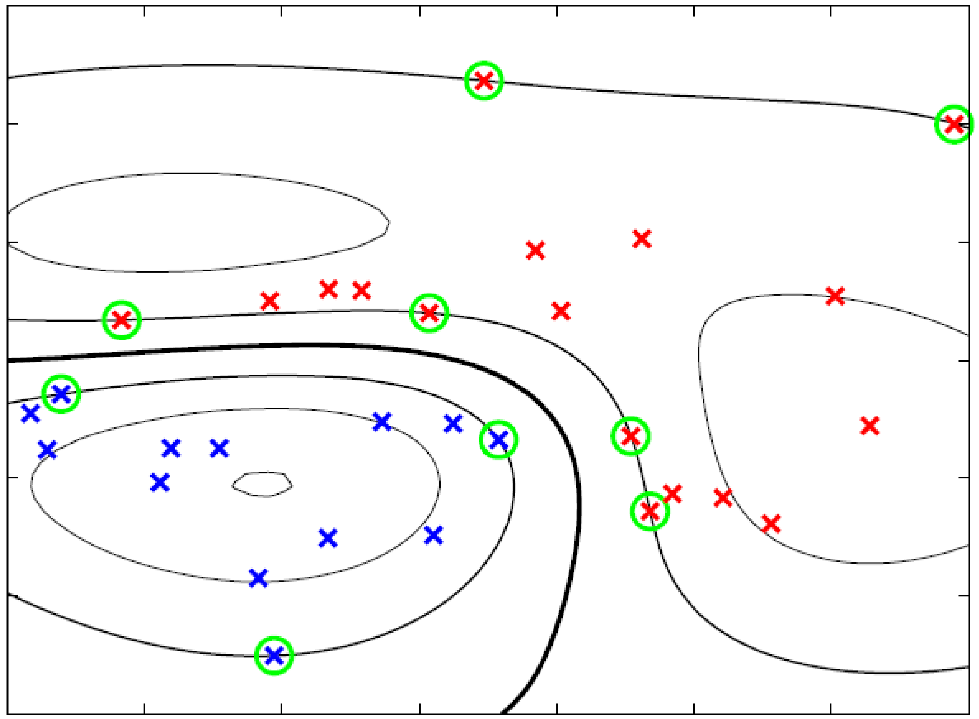
$$\arg\max_{w,b} \left\{ \min_n \frac{y_n(w^T \phi(x_n) + b)}{\|w\|} \right\}$$

Decision boundary is linear in the transformed data, but nonlinear in the original data space

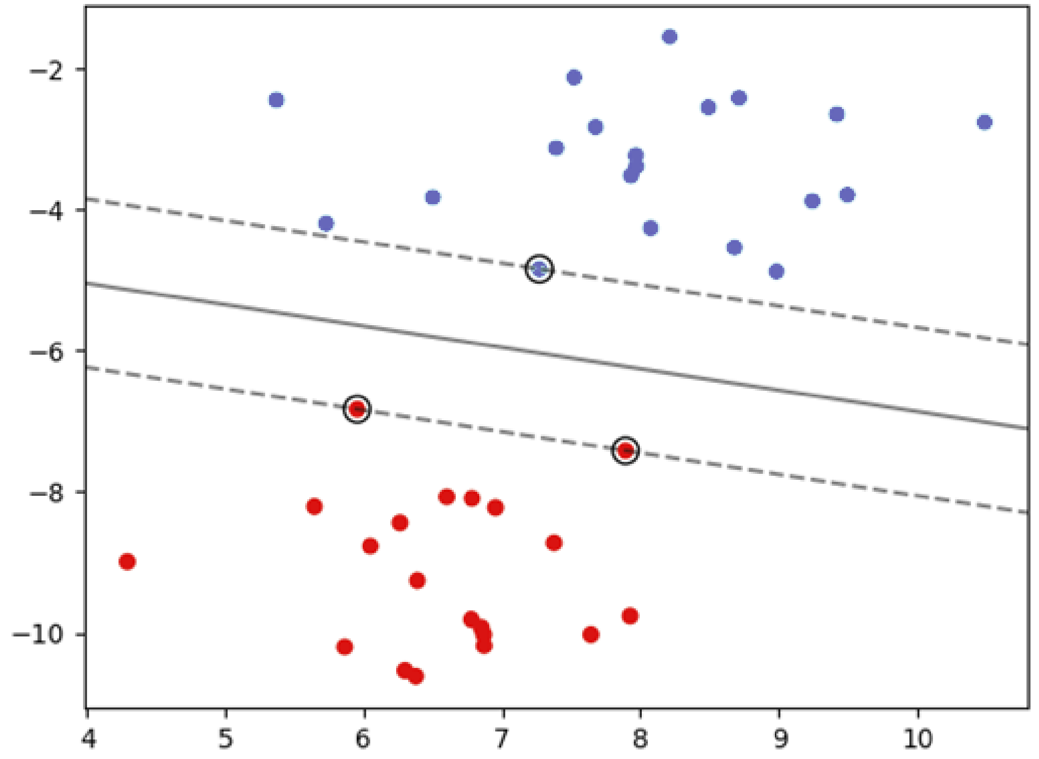
[Source: Bishop, C. PRML]

Nonlinear Max-Margin Classifier

Data Space



Basis Space



Decision boundary is linear in the transformed data, but nonlinear in the original data space

Max-Margin Classifier

Learning objective is hard to solve in this form…

$$\arg\max_{w,b} \left\{ \min_{n} \frac{y_n(w^T\phi(x_n)+b)}{\|w\|} \right\}$$

But we can scale parameters $w \rightarrow \kappa w$ and $b \rightarrow \kappa b$ without changing margin…so we can set the nearest point to the margin so that,

$$y_n(w^T\phi(x_n)+b) = 1$$

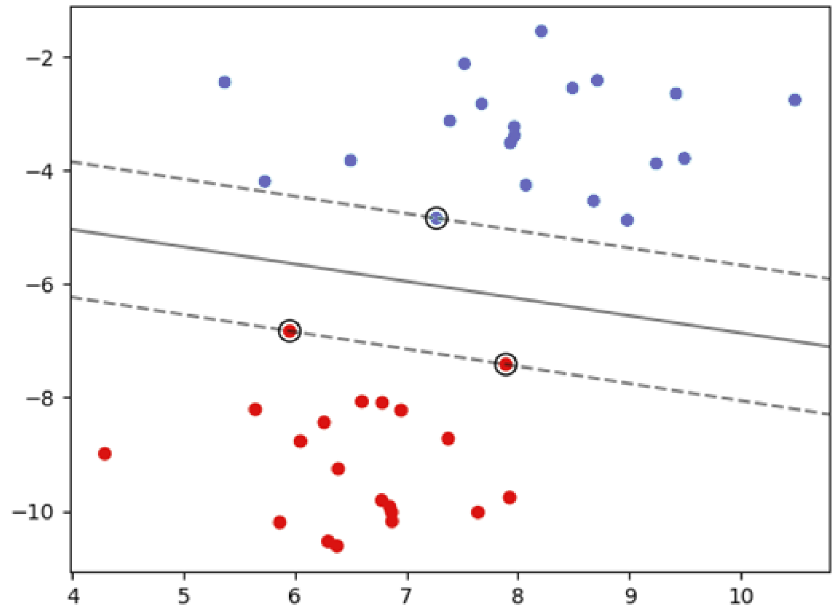
And for all other points not near the margin,

$$y_n(w^T\phi(x_n)+b) \geq 1$$

Now we just have to satisfy these constraints…

Support Vector Machine (Primal)

To learn the classifier, we solve the following *constrained optimization problem…*



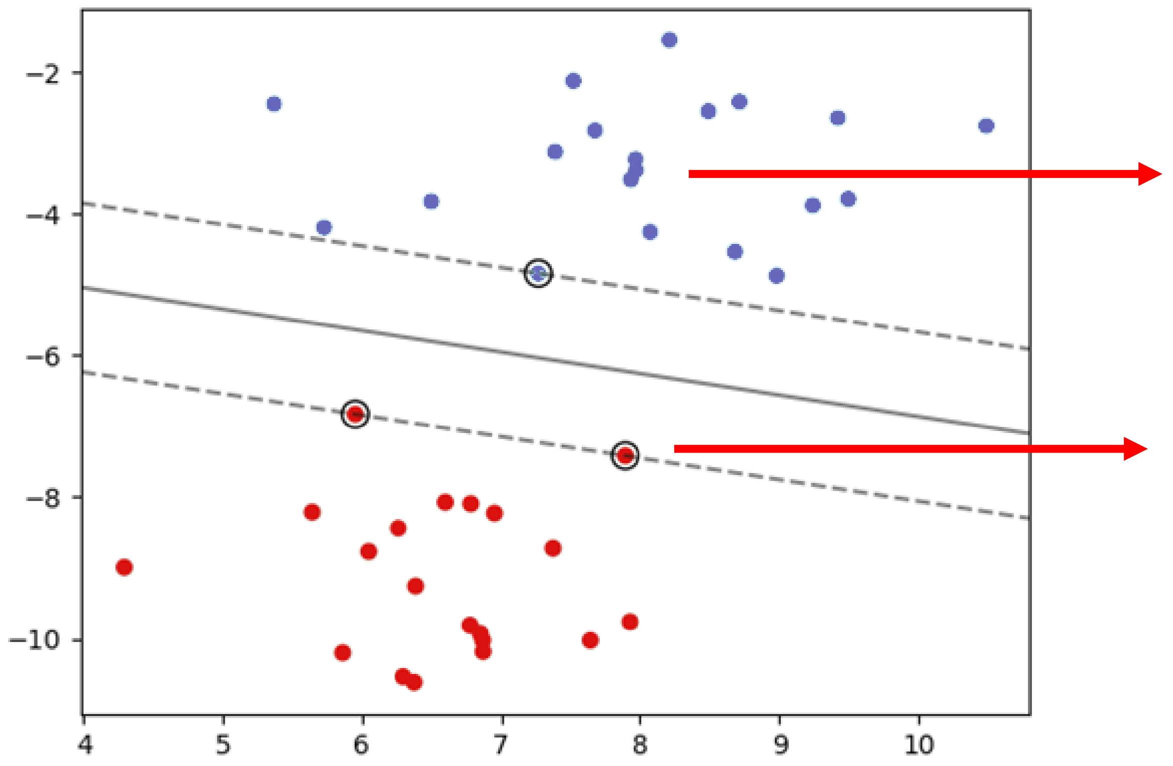
$$\text{minimize } \frac{1}{2}\|w\|^2$$
$$\text{subject to}$$
$$y_n(w^T x_n + b) \geq 1 \qquad \text{for } n = 1, \ldots, N$$

This is known as the *primal* optimization

This is a convex (quadratic) optimization problem that can be solved efficiently

- Data are D-dimensional *vectors*
- Margins determined by nearest data points called *support vectors*
- We call this a *support vector machine* (SVM)

Support Vector Machine (Dual)



All other points are outside the margin and constraints are *loose*:

$$y_n(w^T\phi(x_n)+b) > 1$$

Support vectors are tight to the margin, and satisfy constraints with equality:

$$y_n(w^T\phi(x_n)+b) = 1$$

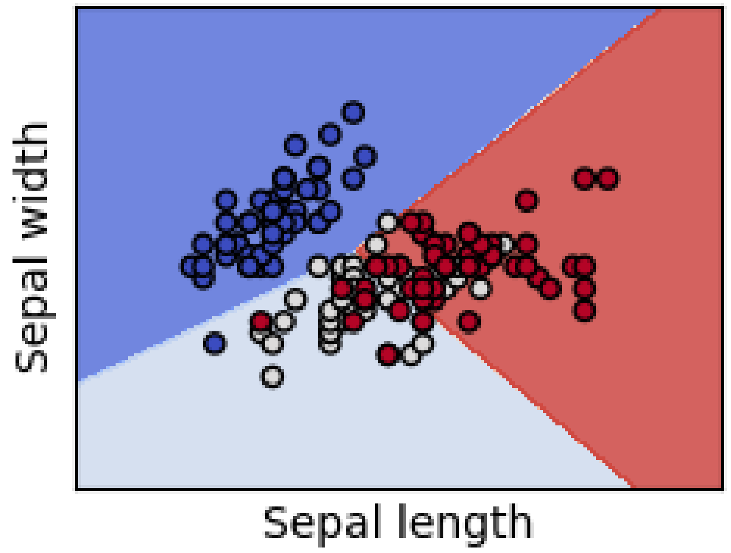
SVM Dual Problem Find the support vectors (set of constraints that hold with equality) that define the largest margin

SVM in Scikit-Learn

SVM with linear decision boundaries,

[sklearn.svm.LinearSVC]

Call options include…



penalty : *{'l1', 'l2'}, default='l2'*

Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

dual : *bool, default=True*

Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

C : *float, default=1.0*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

Only showing linear for a reason that will be clear soon…

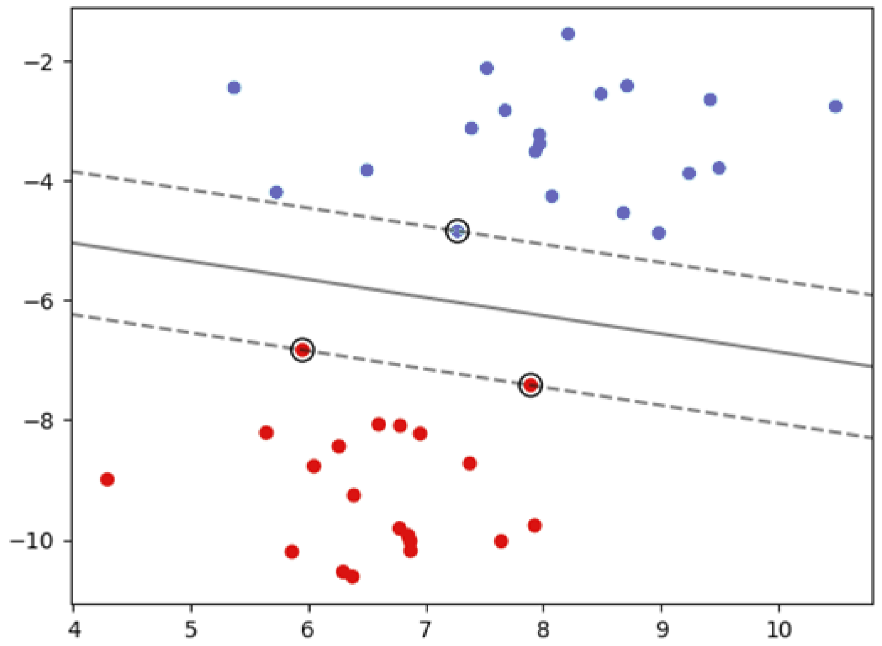
Other options for controlling optimizer (e.g. convergence tolerance 'tol')

Outline

- ➢ Basis Functions
- ➢ Support Vector Machine Classifier
- ➢ Kernels
- ➢ Neural Networks

Support Vector Machine (Dual)

SVM Dual Problem Find the support vectors (set of constraints that hold with equality) that define the largest margin



For each data point, introduce a new optimization variable (dual variable),

$$\lambda_n \geq 0$$

After solving, SVM classifies a new point as:

$$y(x) = \sum_{n=1}^{N} \lambda_n y_n \phi(x)^T \phi(x_n)$$

- Dual variables are nonzero $\lambda_n > 0$ for any support vector
- Exactly zero for non-support vectors $\lambda_n = 0$
- Classifier only needs to store support vectors (sparse representation)

Kernel Functions

Basis transform on new point

Basis transform on training point

$$y(x) = \sum_{n=1}^{N} \lambda_n y_n \underbrace{\phi(x)^T \phi(x_n)}_{\text{Interaction with training points in transformed basis space}}$$

Idea Define a new function as the inner product with basis transforms,

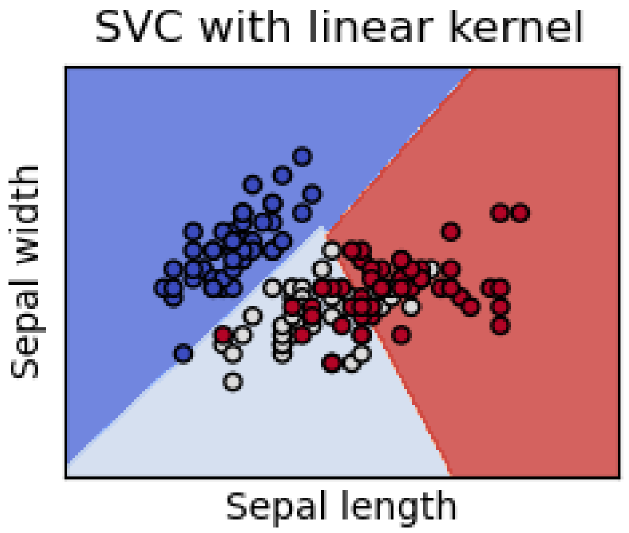
$$\kappa(x, x_n) = \phi(x)^T \phi(x_n)$$

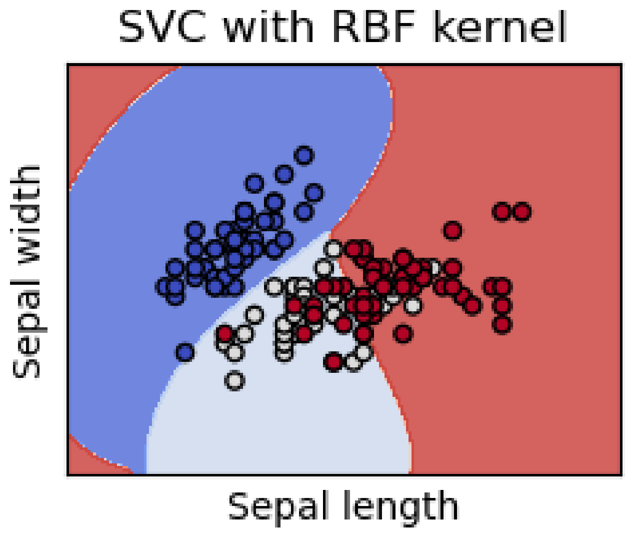
We can now represent the classifier without even knowing the basis,

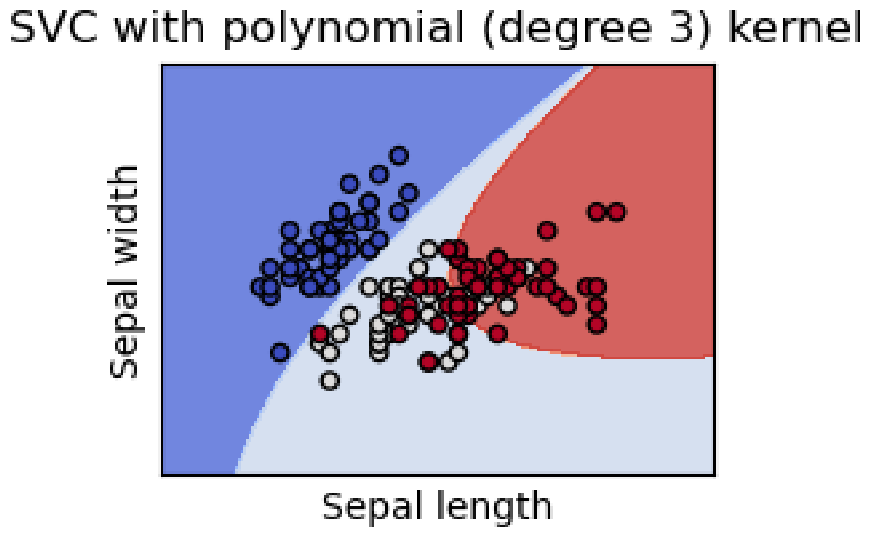
We call this a "kernel function"

$$y(x) = \sum_{n=1}^{N} \lambda_n y_n \kappa(x, x_n)$$

Kernel SVM in Scikit Learn







$$\kappa(x, x') = x^T x'$$

$$\kappa(x, x') = \exp(-\gamma \|x - x'\|^2)$$

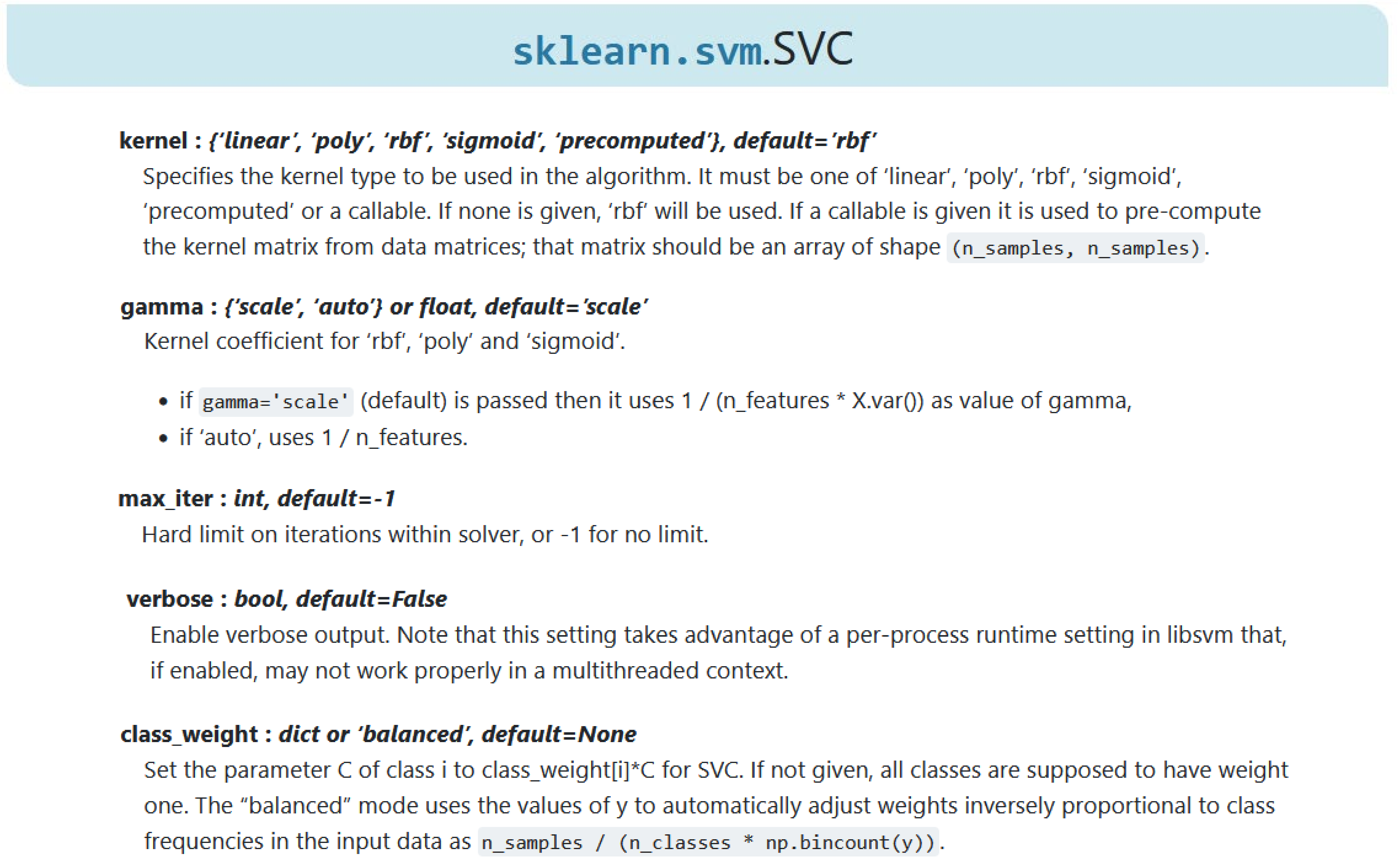
$$\kappa(x, x') = (x^T x' + c)^3$$

Note: No explicit basis function

- General kernel-based SVM lives in:

  sklearn.svm.svc(kernel='kernel name')

- Supports most major kernel types
- Generally use kernel when number of features > number data



sklearn.svm.SVC

kernel : *{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'*

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

gamma : *{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

max_iter : *int, default=-1*

Hard limit on iterations within solver, or -1 for no limit.

verbose : *bool, default=False*

Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

class_weight : *dict or 'balanced', default=None*

Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

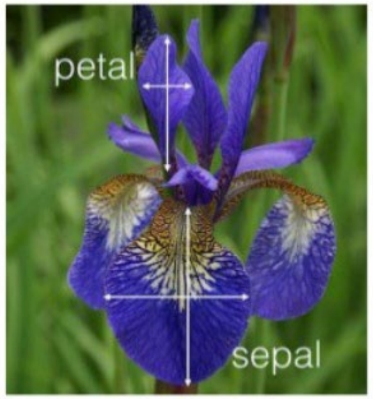
Example: Fisher's Iris Dataset

Classify among 3 species of Iris flowers…

Iris setosa

Iris versicolor

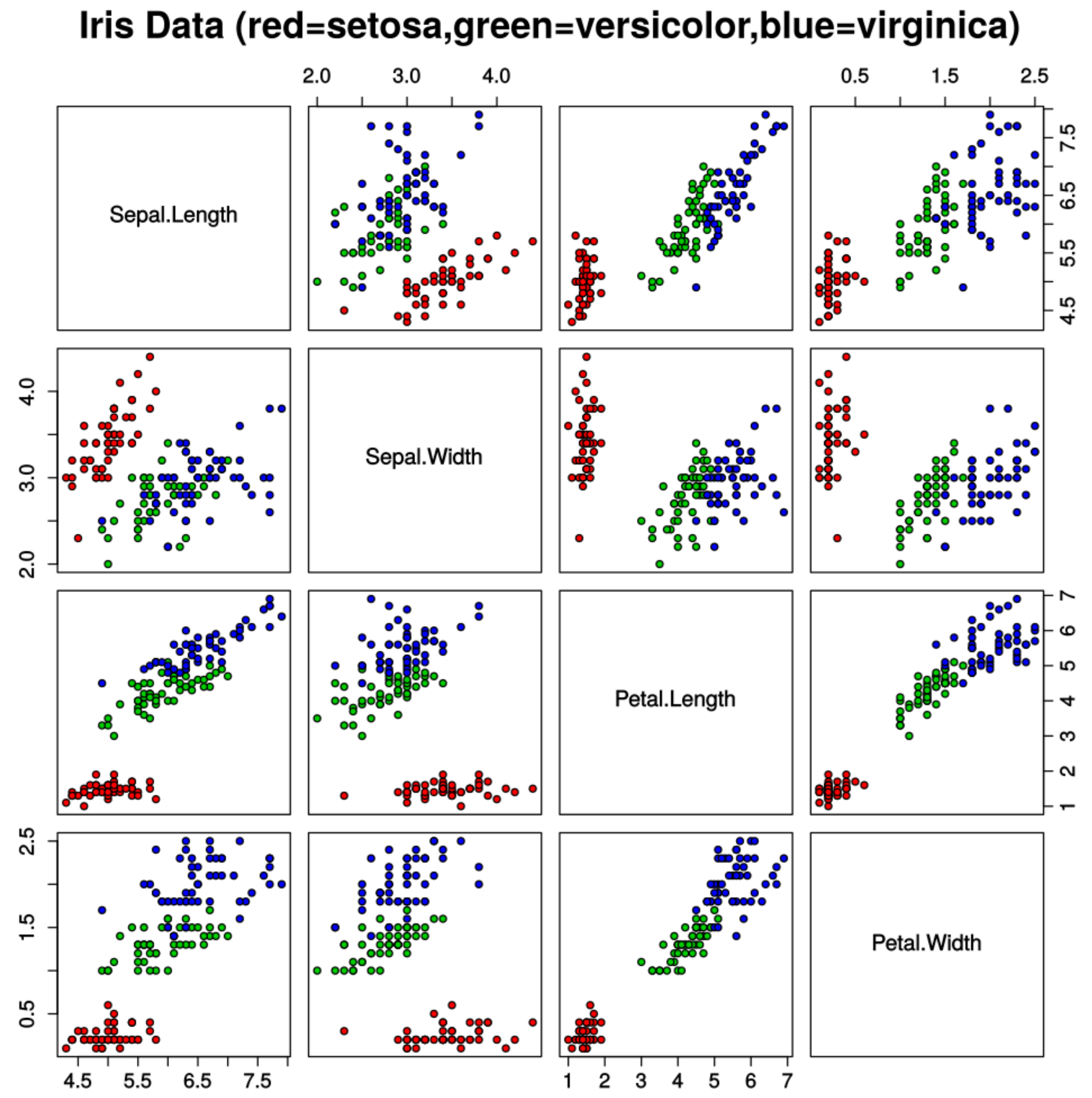
Iris virginica



Four features (in centimeters)

- Petal length / width
- Sepal length / width

Example: Fisher's Iris Dataset



*Fairly easy to separate **setosa** from others using a linear classifier*

Need to use nonlinear basis / kernel representation to better separate other classes

Example: Fisher's Iris Dataset

Train 8-degree polynomial kernel SVM classifier,

```
from sklearn.svm import SVC
svclassifier = SVC(kernel='poly', degree=8)
svclassifier.fit(X_train, y_train)
```

Generate predictions on held-out test data,

```
y_pred = svclassifier.predict(X_test)
```

Show confusion matrix and classification accuracy,

```
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[11  0  0]
 [ 0 12  1]
 [ 0  0  6]]
```

```
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        11
Iris-versicolor       1.00      0.92      0.96        13
 Iris-virginica       0.86      1.00      0.92         6

    avg / total       0.97      0.97      0.97        30
```

[Source: https://stackabuse.com/implementing-svm-and-kernel-svm-with-pythons-scikit-learn/]

Kernel Functions

*A **kernel function** is an inner-product of some basis function computed on two inputs*

$$k(x, x') = \phi(x)^{\mathrm{T}}\phi(x') = \sum_{i=1}^{M} \phi_i(x)\phi_i(x')$$

A consequence is that kernel functions are non-negative real-valued functions over a pair of inputs,

$$\kappa(x, x') \in \mathbb{R} \qquad \kappa(x, x') \geq 0$$

Kernel functions can be interpreted as a measure of distance between two inputs

Kernel Functions

Example The *linear basis* $\phi(x) = x$ produces the kernel,

$$\kappa(x, x') = \phi(x)^T \phi(x') = x^T x'$$

It is often easier to directly specify the kernel rather than the basis function…

Example Gaussian kernel models similarity according to an unnormalized Gaussian distribution,

$$\kappa(x, x') = \exp\left(-\frac{1}{2\sigma^2}(x - x')^2\right)$$

Note Despite the name, this is **not** a Gaussian probability density.

Also called a *radial basis function* (RBF)

Kernel Functions

Given *any* set of data $\{x_i\}_{i=1}^{n}$ a necessary and sufficient condition of a valid kernel function is that the nxn **gram matrix**,

$$\mathbf{K} = \begin{pmatrix} \kappa(x_1, x_1) & \kappa(x_1, x_2) & \dots & \kappa(x_1, x_n) \\ \kappa(x_2, x_1) & \kappa(x_2, x_2) & \dots & \kappa(x_2, x_n) \\ \vdots & \vdots & \vdots & \vdots \\ \kappa(x_n, x_1) & \kappa(x_n, x_2) & \dots & \kappa(x_n, x_n) \end{pmatrix}$$

Is a *symmetric positive semidefinite matrix.*

Techniques for Constructing New Kernels.

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, the following new kernels will also be valid:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= c k_1(\mathbf{x}, \mathbf{x}') && (6.13)\\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x}) k_1(\mathbf{x}, \mathbf{x}') f(\mathbf{x}') && (6.14)\\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.15)\\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.16)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') && (6.17)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') k_2(\mathbf{x}, \mathbf{x}') && (6.18)\\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\right) && (6.19)\\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}} \mathbf{A} \mathbf{x}' && (6.20)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}_a') + k_b(\mathbf{x}_b, \mathbf{x}_b') && (6.21)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}_a') k_b(\mathbf{x}_b, \mathbf{x}_b') && (6.22)
\end{aligned}
$$

[Source: Bishop, C.]

Why Kernel Functions?

At this point you should be slightly confused…

- We learned how to fit linear models
- We learned how to introduce nonlinearities by using basis functions
- Kernels are just inner products of basis functions

…then why do we need Kernels?

Why Kernel Functions?

- Most linear models have an equivalent form in terms of kernels
- Can directly specify kernel function without knowing basis functions
- Kernels can be more intuitive to specify since they capture meaningful distance / difference between two data points
- Kernel-based models can be more flexible than basis functions
- **Example** The RBF (Gaussian) kernel corresponds to infinite-dimensional basis functions. Classifiers based on RBF kernel can perfectly separate any data.

Kernel Ridge Regression

Recall the solution of L2-regularized linear regression (ridge regression),

$$\mathbf{\Phi} = \begin{pmatrix} 1 & \phi_1(x_1) & \dots & \phi_M(x_1) \\ 1 & \phi_1(x_2) & \dots & \phi_M(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \phi_1(x_N) & \dots & \phi_M(x_N) \end{pmatrix} \qquad w^{\text{ridge}} = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda I)^{-1}\mathbf{\Phi}^T\mathbf{y}$$

Define the kernel matrix and vector as,

$$\mathbf{K} = \mathbf{\Phi}^T\mathbf{\Phi} = \begin{pmatrix} \kappa(x_1, x_1) & \kappa(x_1, x_2) & \dots & \kappa(x_1, x_n) \\ \kappa(x_2, x_1) & \kappa(x_2, x_2) & \dots & \kappa(x_2, x_n) \\ \vdots & \vdots & \vdots & \vdots \\ \kappa(x_n, x_1) & \kappa(x_n, x_2) & \dots & \kappa(x_n, x_n) \end{pmatrix}$$

$$\mathbf{k}(\mathbf{x})^T = (\phi(x)^T\phi(x_1), \dots, \phi(x)^T\phi(x_n))$$

Kernel Ridge Regression

The learned regression function (for a new point) is then,

$$y(x) = w^T \phi(x)$$

Solution to ridge regression

$$= \left[(\mathbf{\Phi}^T \mathbf{\Phi} + \lambda I)^{-1} \mathbf{\Phi}^T \mathbf{y} \right]^T \phi(x)$$

$a^T b = b^T a$

$$= \phi(x)^T \left[(\mathbf{\Phi}^T \mathbf{\Phi} + \lambda I)^{-1} \mathbf{\Phi}^T \mathbf{y} \right]$$

Substitute kernel

$$= \mathbf{k}(\mathbf{x})^T (\mathbf{K} + \lambda I)^{-1} \mathbf{y}$$

Also known as the dual formulation of linear regression

Can now express regression without explicitly specifying basis functions

Kernel Ridge Regression

Kernel representation requires inversion of NxN matrix

Primal

$$\mathbf{\Phi} = \begin{pmatrix} 1 & \phi_1(x_1) & \dots & \phi_M(x_1) \\ 1 & \phi_1(x_2) & \dots & \phi_M(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \phi_1(x_N) & \dots & \phi_M(x_N) \end{pmatrix}$$

$$w = \underbrace{(\mathbf{\Phi}^T\mathbf{\Phi} + \lambda I)^{-1}}_{\substack{\textbf{MxM Matrix Inversion} \\ \textbf{O}(\mathbf{M}^3)}}\mathbf{\Phi}^T\mathbf{y}$$

Dual

$$\mathbf{K} = \begin{pmatrix} \kappa(x_1, x_1) & \kappa(x_1, x_2) & \dots & \kappa(x_1, x_n) \\ \kappa(x_2, x_1) & \kappa(x_2, x_2) & \dots & \kappa(x_2, x_n) \\ \vdots & \vdots & \vdots & \vdots \\ \kappa(x_n, x_1) & \kappa(x_n, x_2) & \dots & \kappa(x_n, x_n) \end{pmatrix}$$

$$w = \mathbf{k}(\mathbf{x})^T\underbrace{(\mathbf{K} + \lambda I)^{-1}}_{\substack{\textbf{NxN Matrix Inversion} \\ \textbf{O}(\mathbf{N}^3)}}\mathbf{y}$$

Number of training data N greater than basis functions M

`sklearn.kernel_ridge`.KernelRidge

alpha : *float or array-like of shape (n_targets,), default=1.0*

Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `1 / (2C)` in other linear models such as **`LogisticRegression`** or **`LinearSVC`**. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number. See Ridge regression and classification for formula.

kernel : *str or callable, default="linear"*

Kernel mapping used internally. This parameter is directly passed to `pairwise_kernel`. If `kernel` is a string, it must be one of the metrics in `pairwise.PAIRWISE_KERNEL_FUNCTIONS`. If `kernel` is "precomputed", X is assumed to be a kernel matrix. Alternatively, if `kernel` is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two rows from X as input and return the corresponding kernel value as a single number. This means that callables from **`sklearn.metrics.pairwise`** are not allowed, as they operate on matrices, not single samples. Use the string identifying the kernel instead.

gamma : *float, default=None*

Gamma parameter for the RBF, laplacian, polynomial, exponential chi2 and sigmoid kernels. Interpretation of the default value is left to the kernel; see the documentation for sklearn.metrics.pairwise. Ignored by other kernels.

Example: Kernel Ridge Regression

Generate some sinusoidal (periodic) data,

```
X = 15 * rng.rand(100, 1)
y = np.sin(X).ravel()
y += 3 * (0.5 - rng.rand(X.shape[0]))  # add noise
```

Define an exponentiated sinusoidal kernel,

```
from sklearn.gaussian_process.kernels import ExpSineSquared
kernel = ExpSineSquared(length_scale=4.64, periodicity=12.9)
```

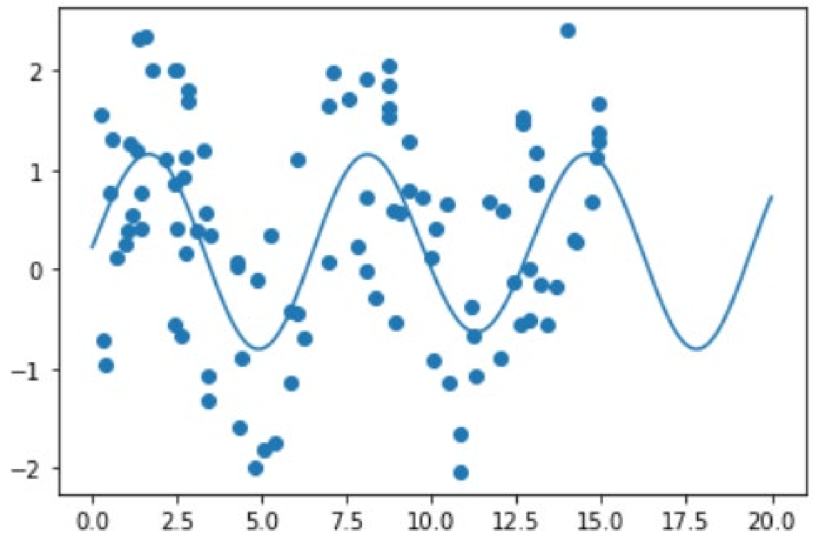
$$\exp\left(-\frac{2\sin^2(\pi d(x_i,x_j)/p)}{l^2}\right)$$

Fit kernel ridge regression,

```
from sklearn.kernel_ridge import KernelRidge
kr = KernelRidge(kernel=kernel, alpha=0.001).fit(X,y)
```

Plot results,

```
X_plot = np.linspace(0, 20, 10000)[:, None]
y_kr = kr.predict(X_plot)
plt.scatter(X,y)
plt.plot(X_plot, y_kr)
plt.show()
```



Outline

- ➢ Basis Functions
- ➢ Support Vector Machine Classifier
- ➢ Kernels
- ➢ Neural Networks

Basis Functions

Basis functions transform linear models into nonlinear ones…

Linear Regression

Classification
(Logistic Regression)

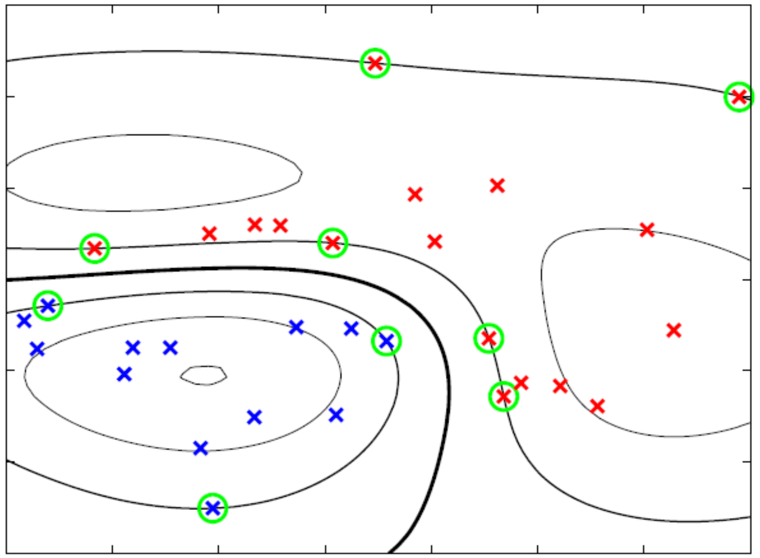
$$\begin{tikzcd}
y = w^T x \arrow[d] & y = \sigma(w^T x) \arrow[d] \\
y = w^T \phi(x) & y = \sigma(w^T \phi(x))
\end{tikzcd}$$

…but it is often difficult to find a good basis transformation

Learning Basis Functions

What if we could learn a basis function so that a simple linear model performs well…

Data Space

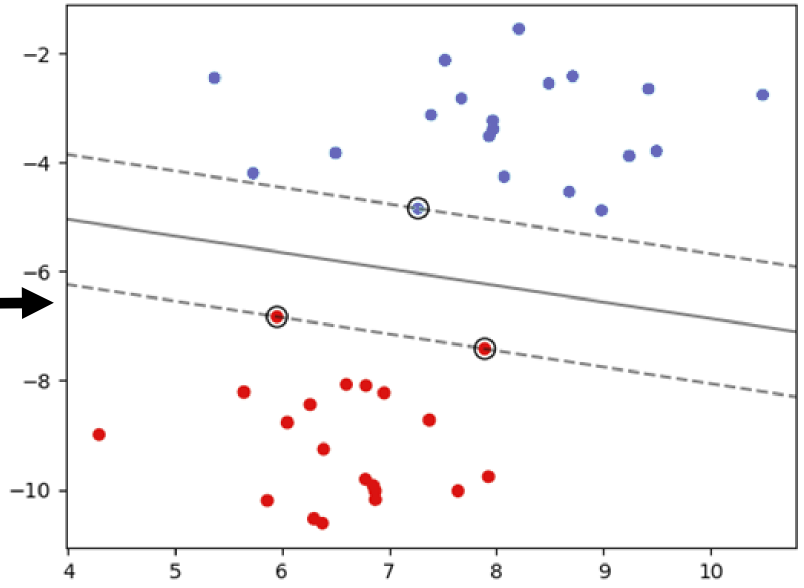


Ignore the circled points…I reused these from the SVM slides

Neural Net

$\phi(x)$

Warped Space



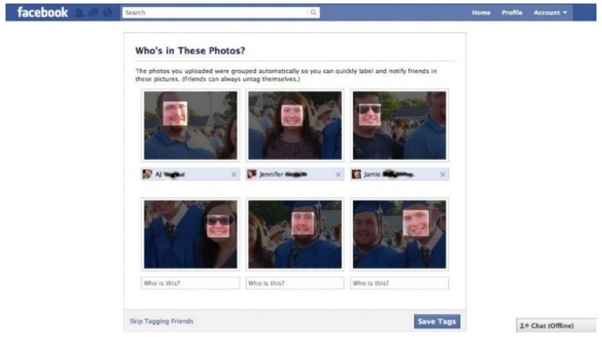
…this is essentially what standard neural networks do…

Neural Networks

- Flexible nonlinear transformations of data
- Resulting transformation is easily fit with a linear model
- Relatively efficient learning procedure scales to massive data
- Apply to many Machine Learning / Data Science problems
  - Regression
  - Classification
  - Dimensionality reduction
  - Function approximation
  - Many application-specific problems

Neural Networks

Forms of NNs are used all over the place nowadays…



Who's in These Photos?

The photos you uploaded were grouped automatically so you can quickly label and notify friends in these pictures. (Friends can always untag themselves.)

AJ [redacted]
Jennifer [redacted]
Jamie [redacted]

Who is this?
Who is this?
Who is this?

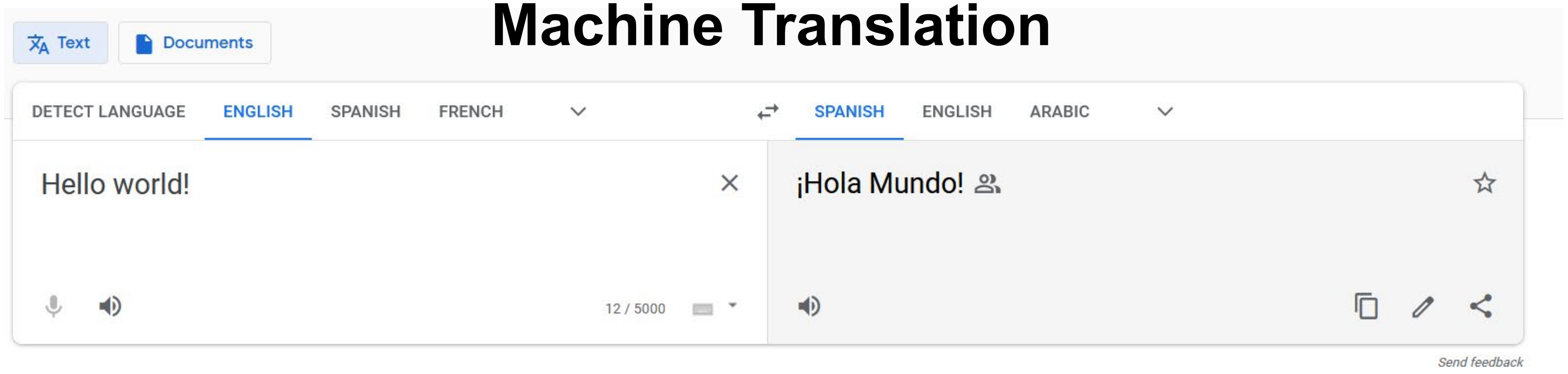
Skip Tagging Friends
Save Tags
Chat (Offline)

FB Auto Tagging

Self-Driving Cars

Creepy Robots

Machine Translation



Hello world!

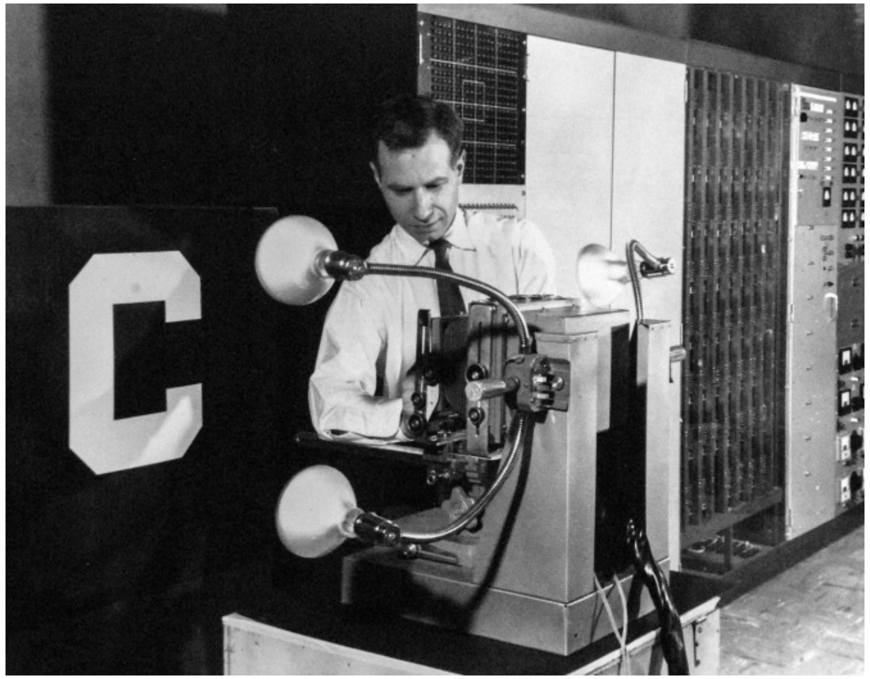
12 / 5000

¡Hola Mundo!

Rosenblatt’s Perceptron

Despite recent attention, neural networks are fairly old

In 1957 Frank Rosenblatt constructed the first (single layer) neural network known as a “perceptron”



He demonstrated that it is capable of recognizing characters projected onto a 20x20 “pixel” array of photosensors

Rosenblatt's Perceptron

FIG. 1 — Organization of a biological brain. (Red areas indicate active cells, responding to the letter X.)

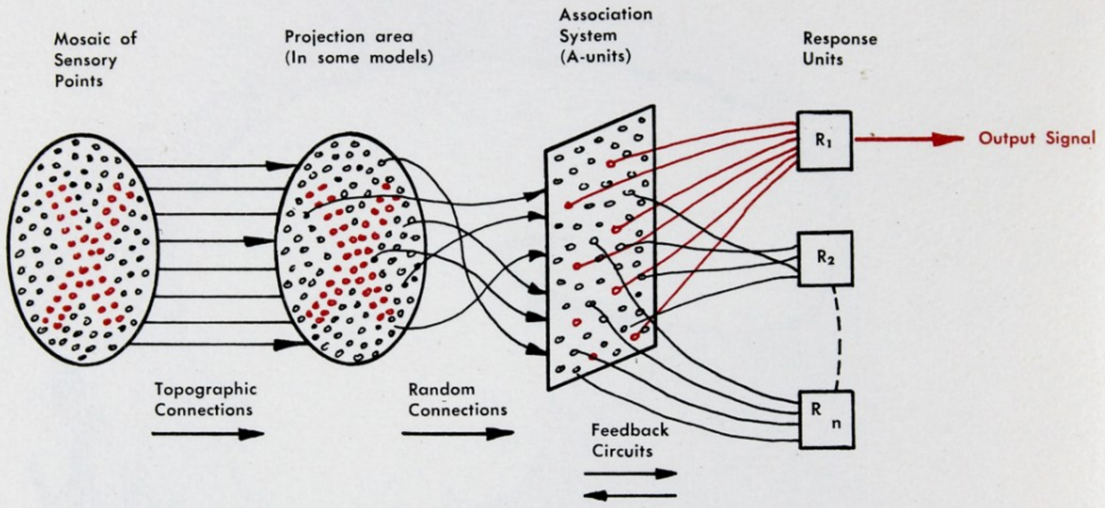
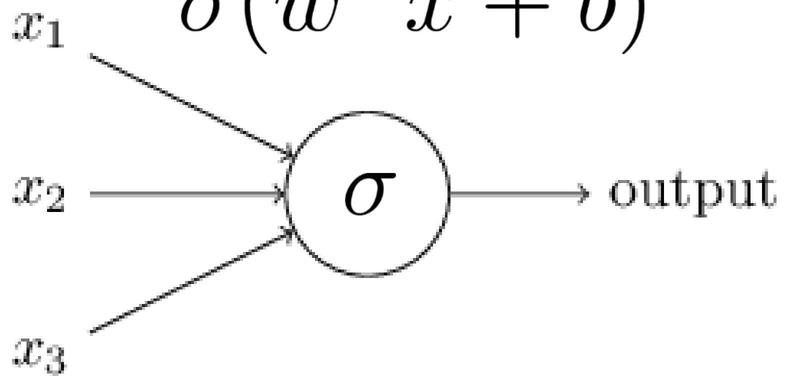


FIG. 2 — Organization of a perceptron.

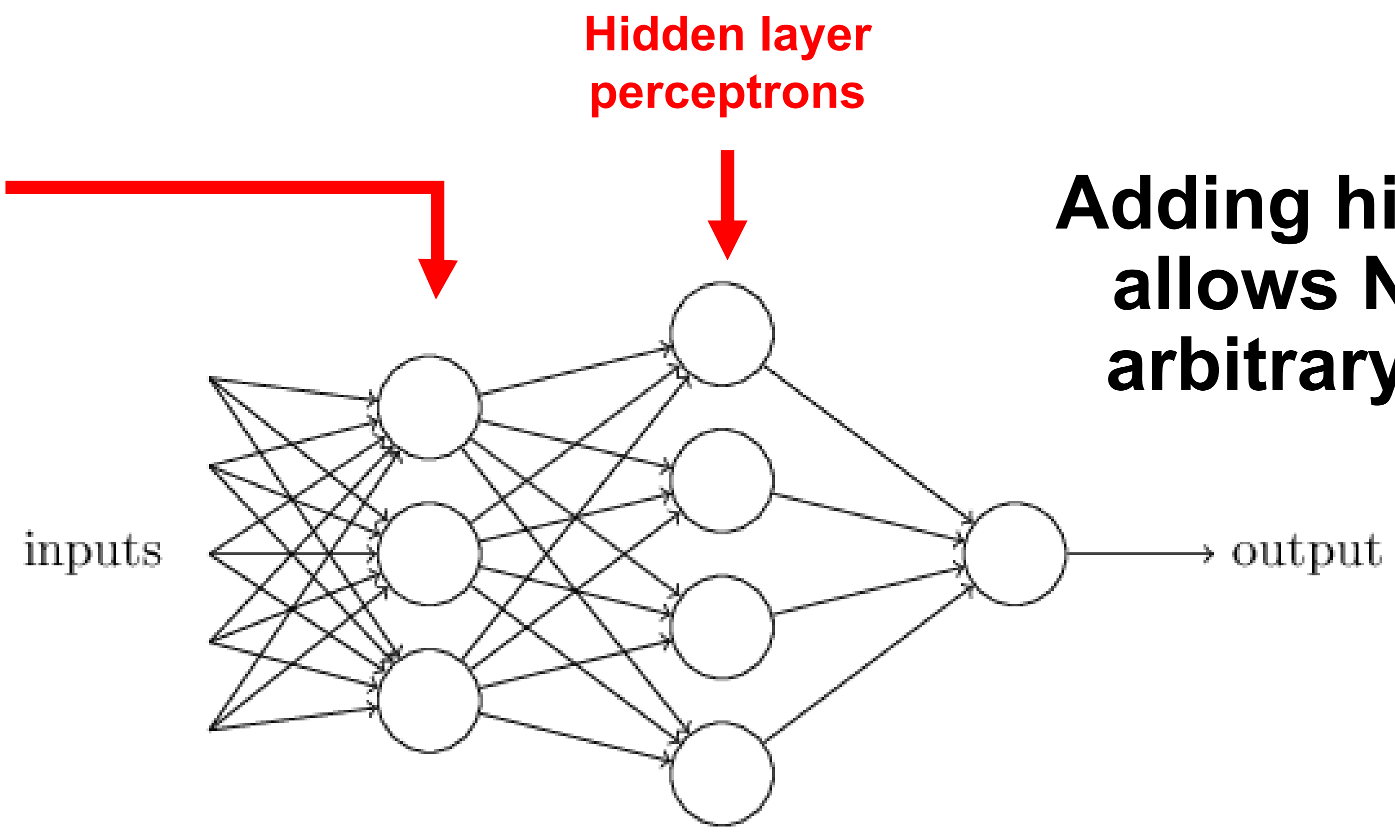
Perceptron

$$\sigma(w^T x + b)$$



- In Rosenblatt's perceptron, the inputs are tied directly to output
- "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanics" (1962)
- Criticized by Marvin Minsky in book "Perceptrons" since can only learn linearly-separable functions
- **The perceptron is just logistic regression in disguise**

Multilayer Perceptron



Adding hidden layers allows NN to learn arbitrary functions

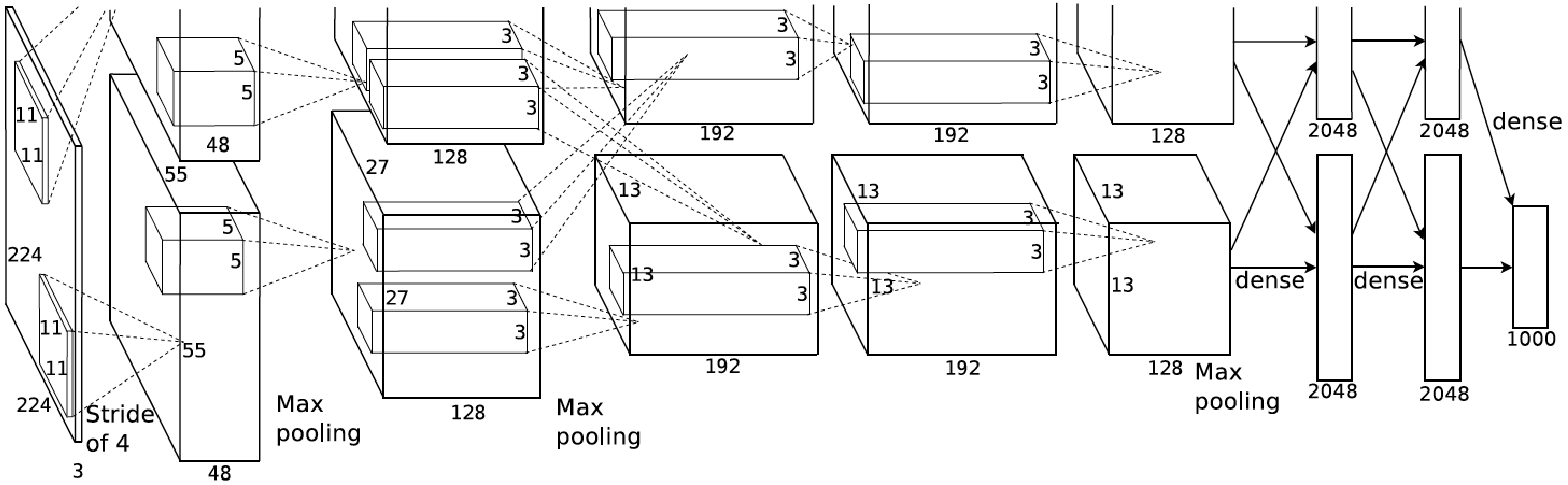
This is the quintessential *Neural Network…*

…also called *Feed Forward Neural Net* or *Artificial Neural Net*

[Source: http://neuralnetworksanddeeplearning.com]

Modern Neural Networks

Modern *Deep Neural networks* add many hidden layers



…and have many millions of parameters to learn

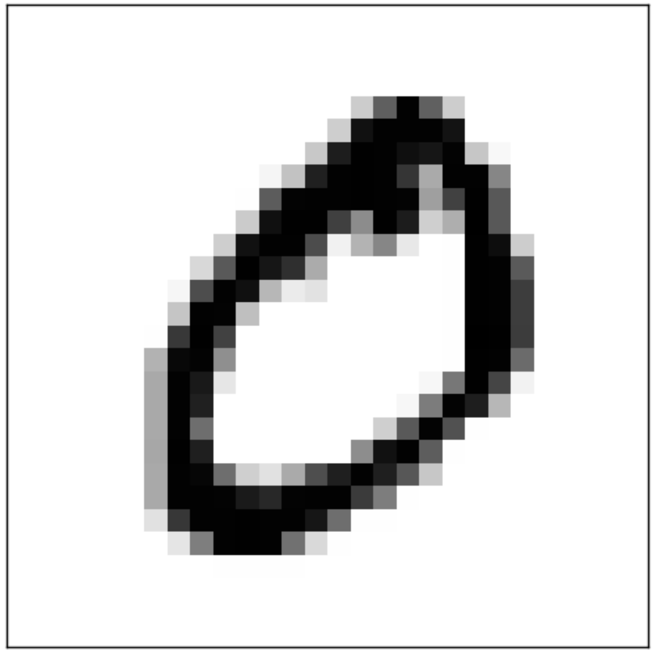
[Source: Krizhevsky et al. (NIPS 2012)]

Handwritten Digit Classification

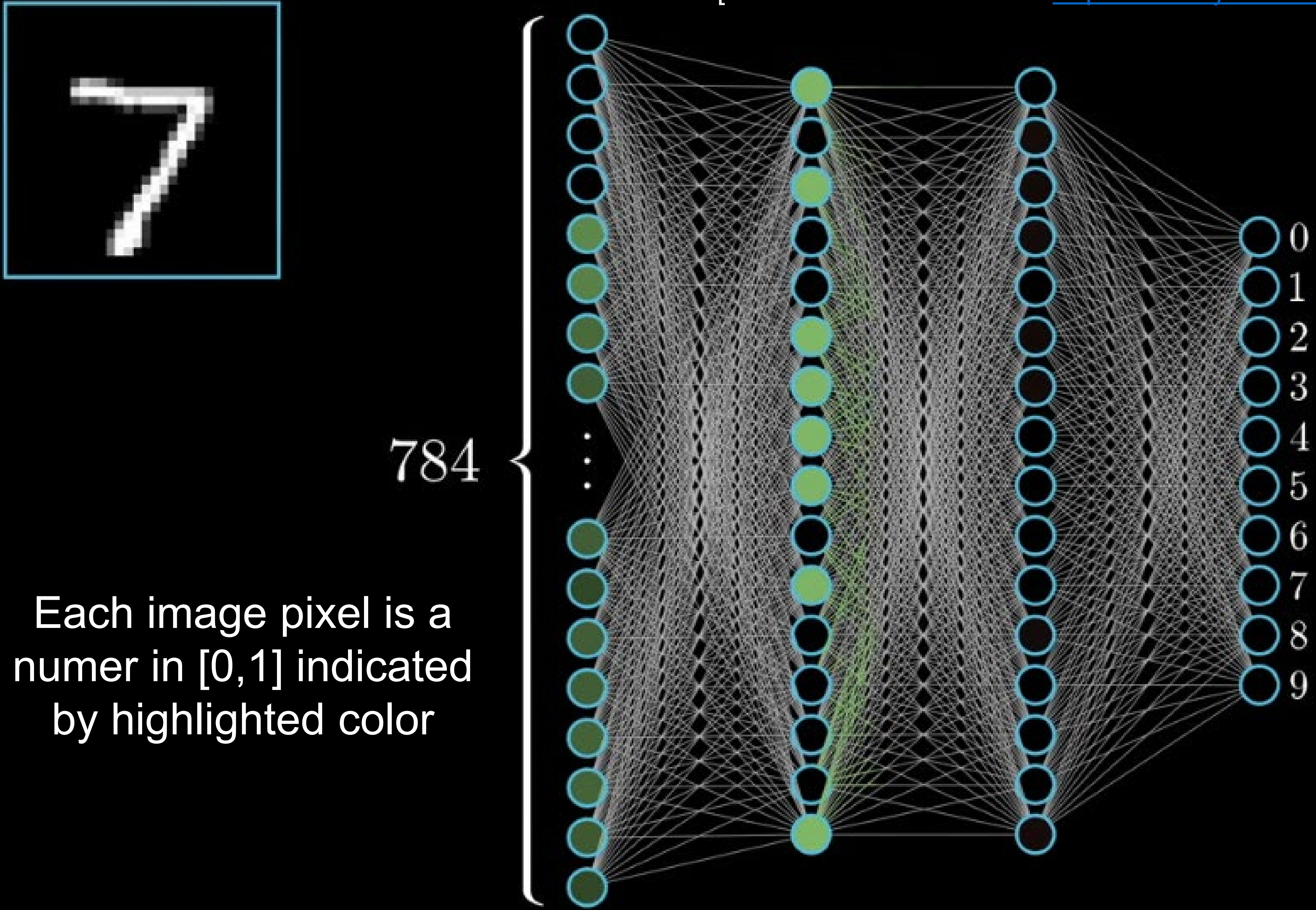
Classifying handwritten digits is the "Hello World" of NNs

Modified National Institute of Standards and Technology (MNIST) database contains 60k training and 10k test images

Each character is centered in a 28x28=784 pixel grayscale image

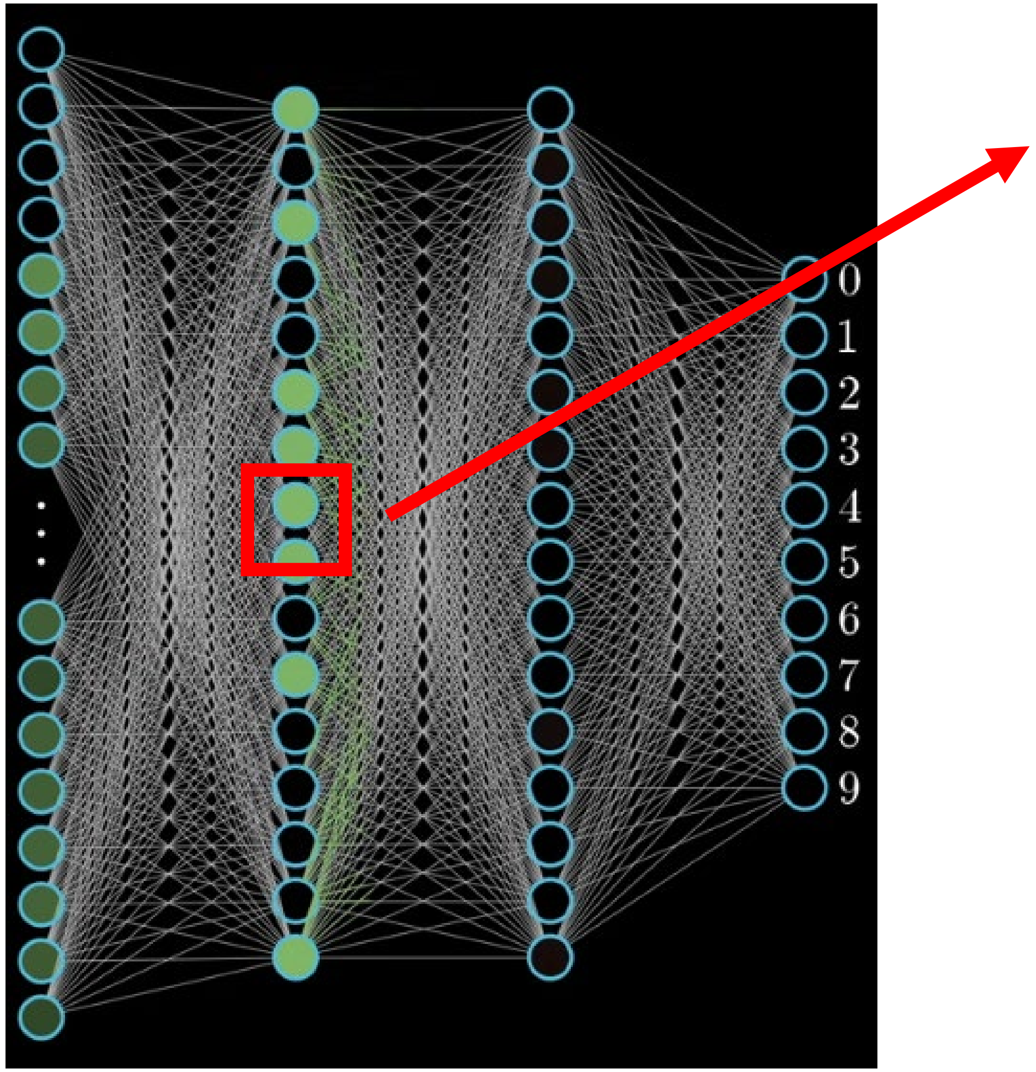


[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]



Each image pixel is a numer in [0,1] indicated by highlighted color

Feedforward Procedure



Each node computes a *weighted combination* of nodes at the previous layer…

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

Then applies a *nonlinear function* to the result

$$\sigma(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$

Often, we also introduce a constant *bias* parameter

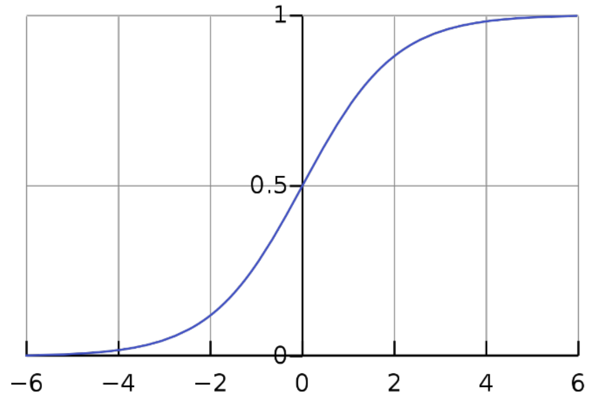
Nonlinear Activation functions

We call this an *activation function* and typically write it in vector form,

$$\sigma(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b) = \sigma(w^T x + b)$$

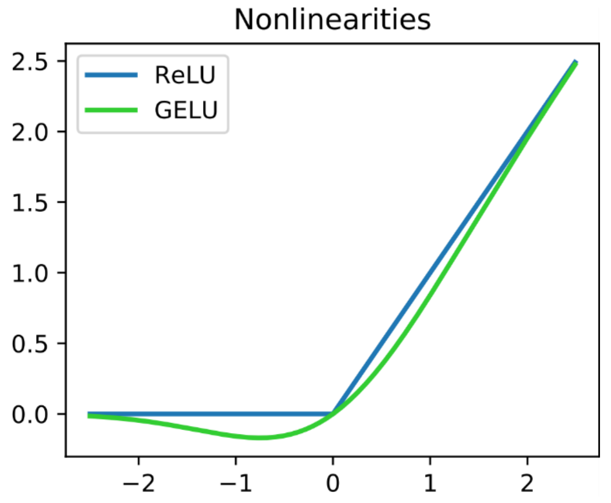
An early choice was the *logistic function*,

$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$



Later found to lead to slow learning and *ridge functions* like the *rectified linear unit (ReLU)*,

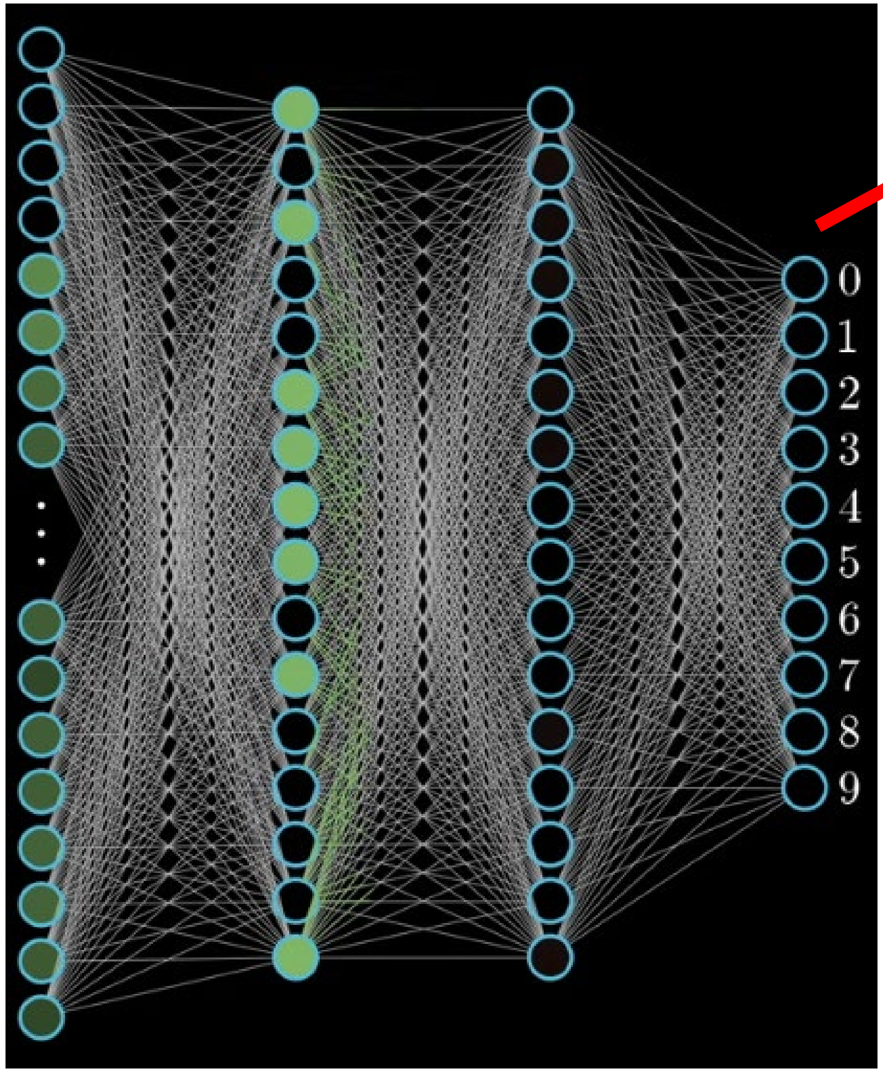
$$\sigma(w^T x + b) = \max(0, w^T x + b)$$



Or the smooth *Gaussian error linear unit (GeLU)*,

$v = w^T x + b$ $\quad\quad \sigma(v) = v\Phi(v) \longleftarrow$ **Gaussian CDF**

Multilayer Perceptron



Final layer is typically a linear model…for classification this is a Logistic Regression

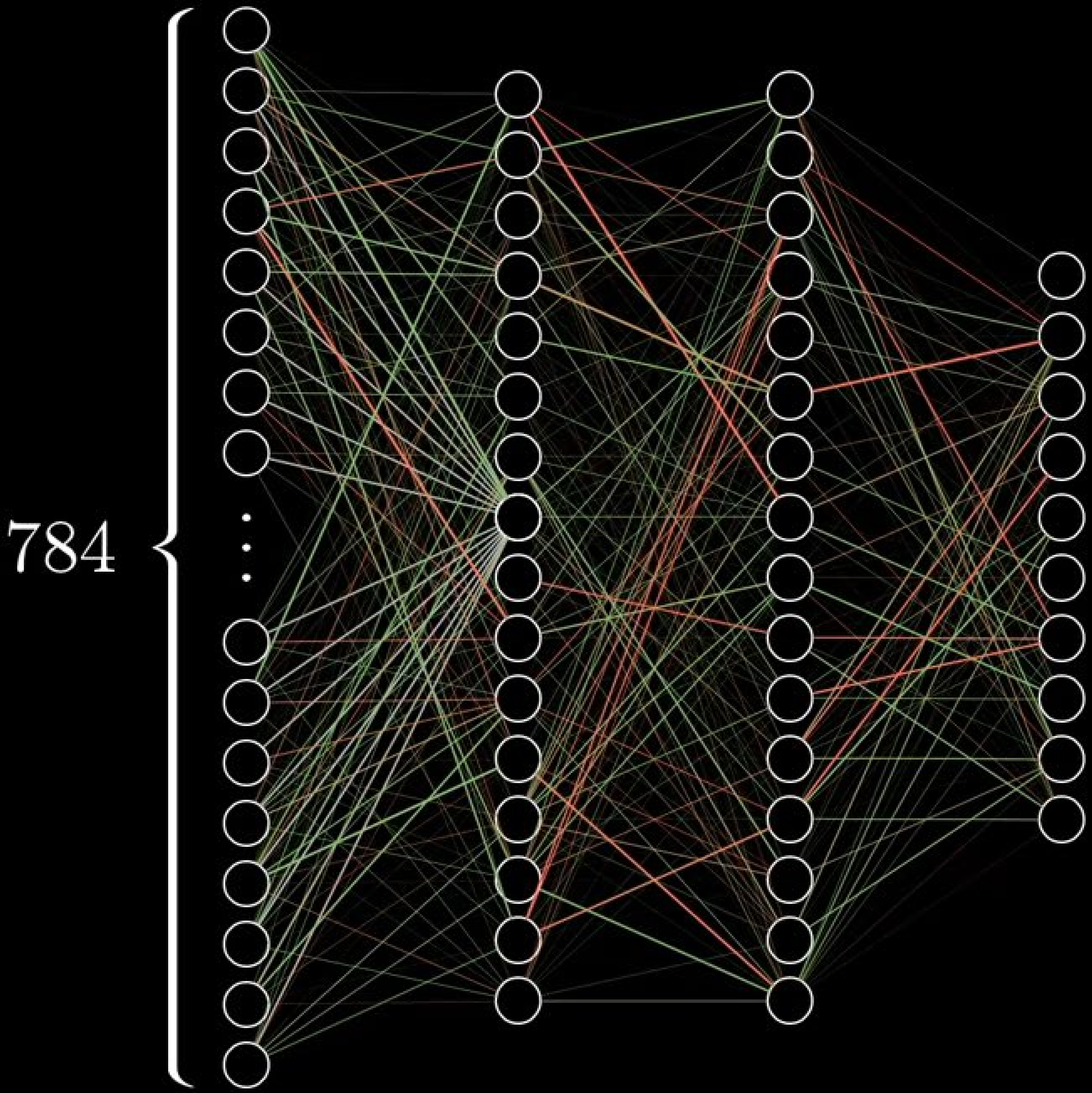
$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Vector of activations from previous layer

Recall that for multiclass logistic regression with K classes,

$$p(\text{Class} = k \mid x) \propto \sigma(w_k^T x + b_k)$$

[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]



$$\boxed{\begin{array}{c} 784\times16+16\times16+16\times10 \\ \text{weights} \\ \\ 16\ +16+10 \\ \text{biases} \end{array}}$$

13,002

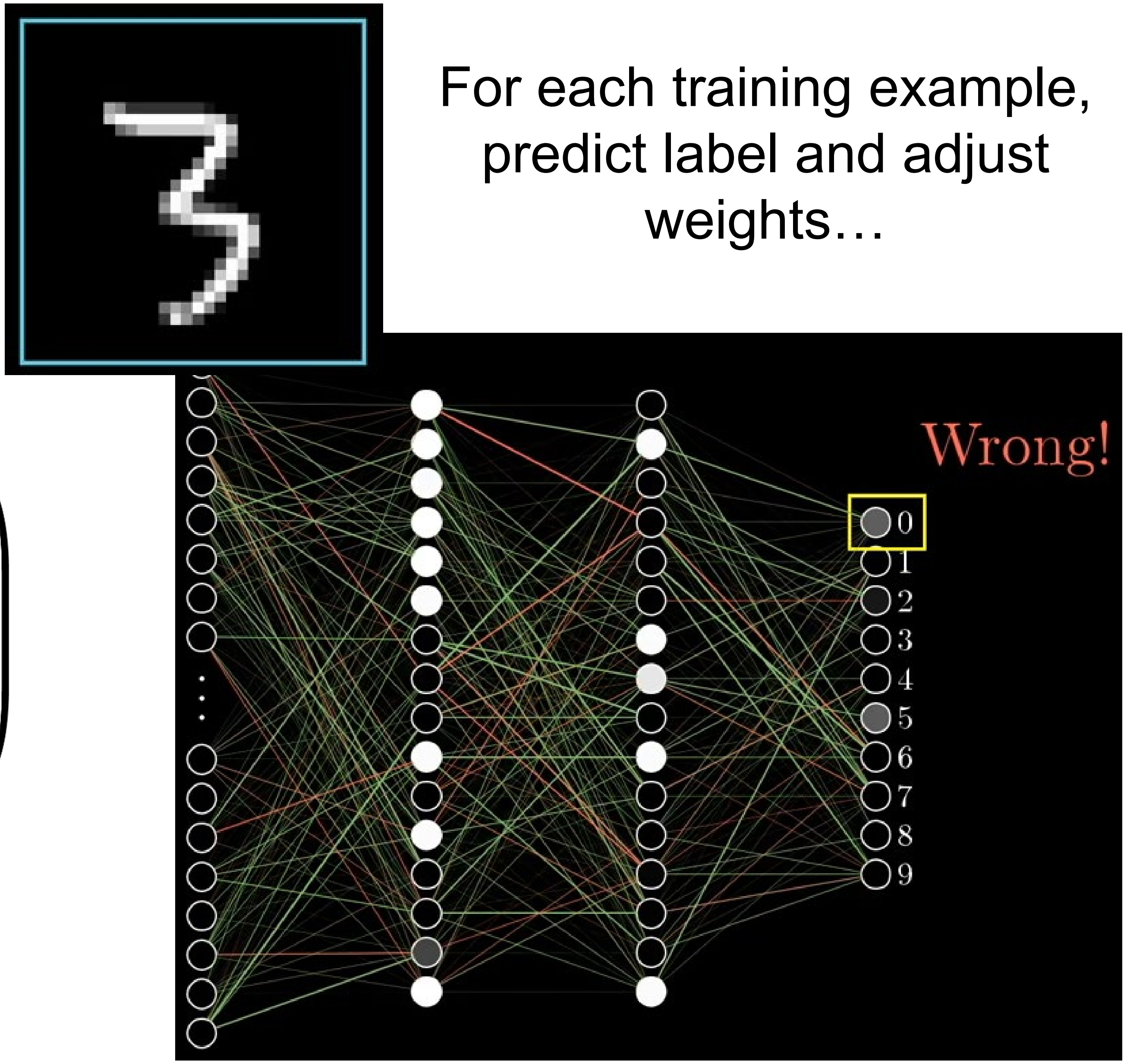
Each parameter has some impact on the output…need to tweak (learn) all parameters simultaneously to improve prediction accuracy

Training Multilayer Perceptron

$X^{\text{Train}} =$

$$Y^{\text{Train}} = \begin{pmatrix} 0 & 4 & 1 & \dots & 3 \\ 5 & 3 & 6 & \dots & 4 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 7 & 4 & 6 & \dots & 5 \end{pmatrix}$$

For each training example, predict label and adjust weights…

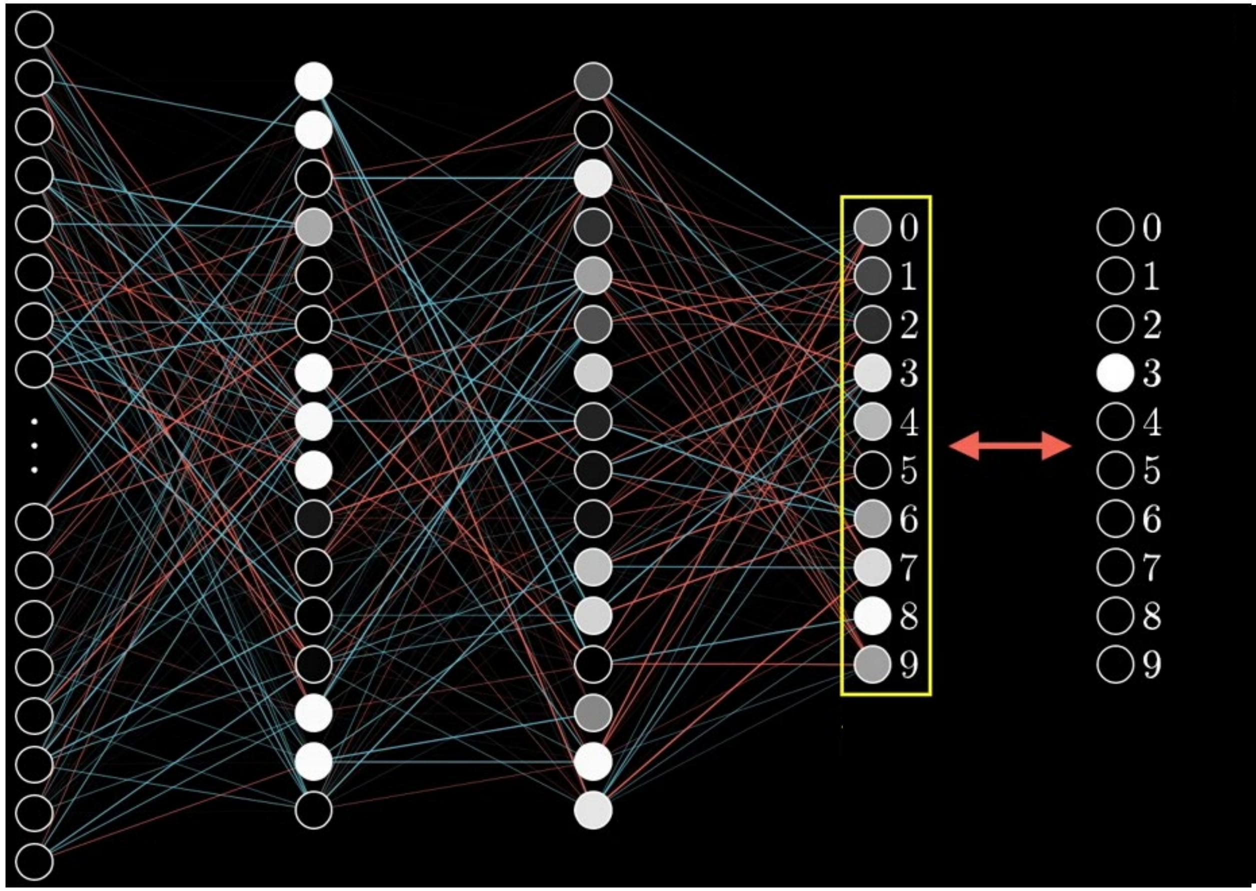


- How to score final layer output?
- How to adjust weights?

Training Multilayer Perceptron

Score based on difference between final layer and one-hot vector of true class…

Input



$$
\begin{aligned}
&(0.43 - 0.00)^2 + \\
&(0.28 - 0.00)^2 + \\
&(0.19 - 0.00)^2 + \\
&(0.88 - 1.00)^2 + \\
&(0.72 - 0.00)^2 + \\
&(0.01 - 0.00)^2 + \\
&(0.64 - 0.00)^2 + \\
&(0.86 - 0.00)^2 + \\
&(0.99 - 0.00)^2 + \\
&(0.63 - 0.00)^2
\end{aligned}
$$

[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]

Training Multilayer Perceptron

Our cost function for ith input is error in terms of weights / biases…

$$\text{Cost}_i(\underbrace{w_1, \ldots, w_n, b_1, \ldots, b_n}_{\textbf{13,002 Parameters in this network}})$$

…minimize cost over all training data…

$$\min_{w,b} \mathcal{L}(w, b) = \sum_i \text{Cost}_i(w_1, \ldots, w_n, b_1, \ldots, b_n)$$

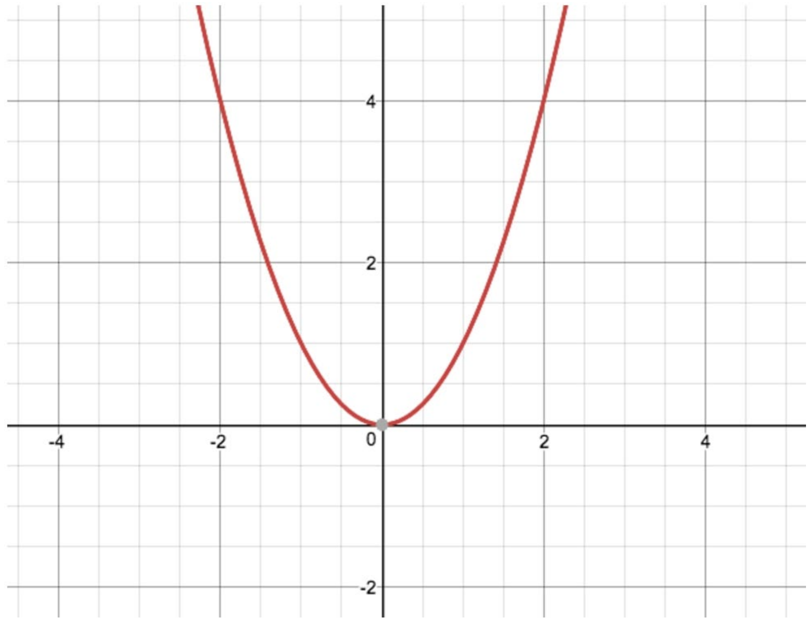
This is a super high-dimensional optimization (13,002 dimensions in this example)…how do we solve it?

Gradient descent!

Training Multilayer Perceptron

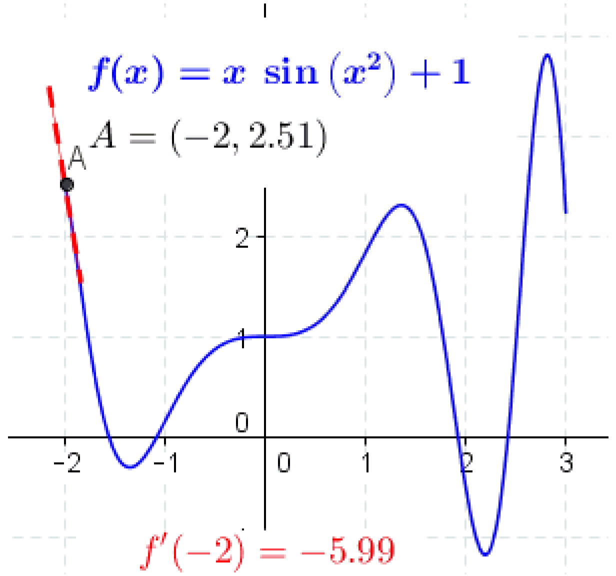
Need to find zero derivative (gradient) solution…

Convex Cost Function



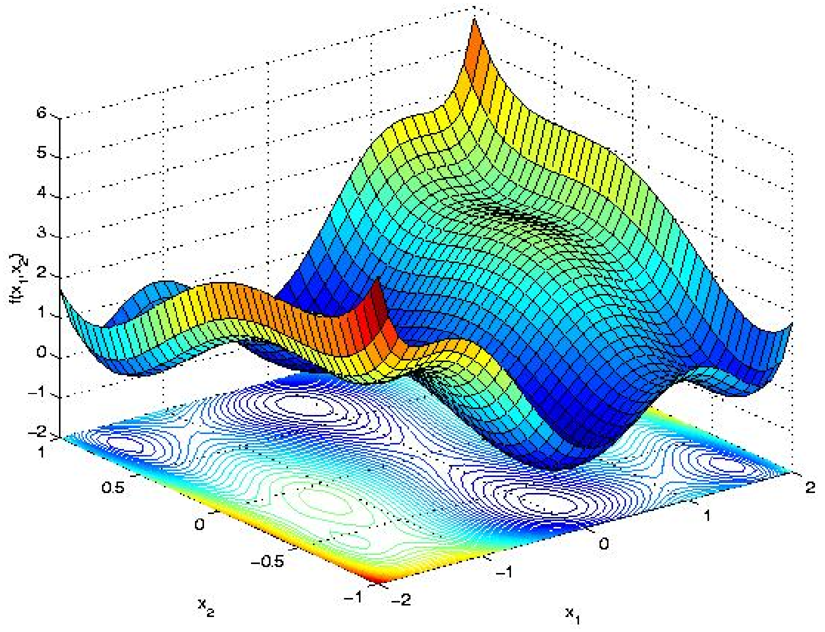
YAY!

Non-convex Cost Function



Boo!

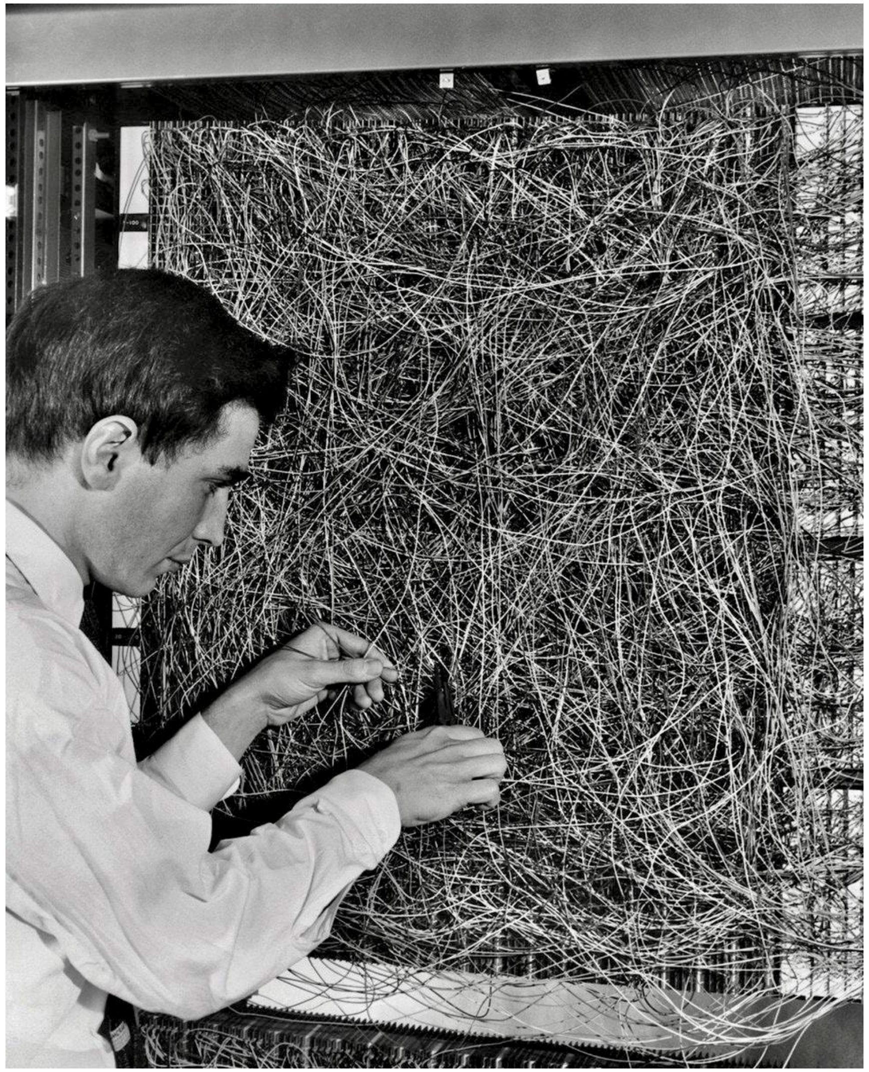
High-Dimensional Non-convex



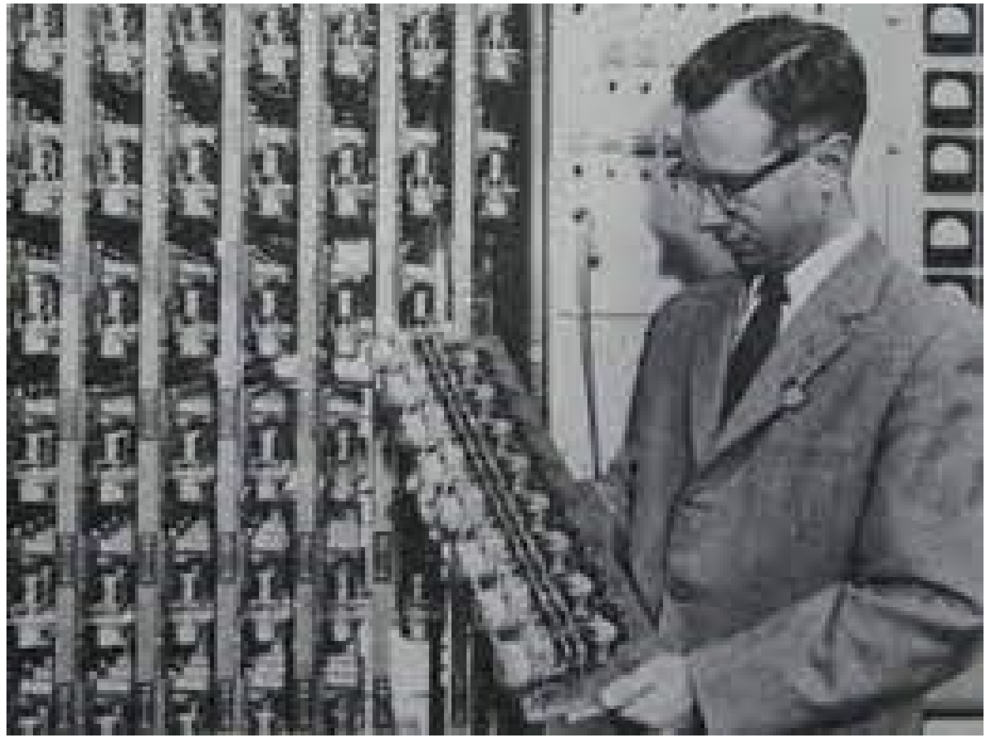
Super Boo!

Actually, the situation is much worse, since the cost is super (13,002) high dimensional…but we proceed as if…

Training the Multilayer Perceptron



Training the MLP is challenging…but it's much easier than how Rosenblatt did it



Example

Play with a small multilayer perceptron on a binary classification task…

https://playground.tensorflow.org/

Computing the Derivative

So we need to compute derivatives of a super complicated function…

$$\frac{d}{dw}\mathcal{L}(w) = \sum_i \frac{d}{dw}\text{Cost}_i(w)$$

Dropped bias terms for simplicity

Recall the **derivative chain rule**

$$\frac{d}{dw}f(g(w)) = \underbrace{\frac{d}{dg(w)}f(g(w))}_{\text{Derivative of f at its argument g(w) e.g. treat g(w) as a variable}} \left(\underbrace{\frac{d}{dw}g(w)}_{\text{Differentiate g with respect to w}}\right)$$

Derivative Chain Rule

Alternatively we can write this as…

$$\frac{d}{dw}f(g(w)) = f'(g(w))g'(w)$$

Example Derivative of the logistic function,

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

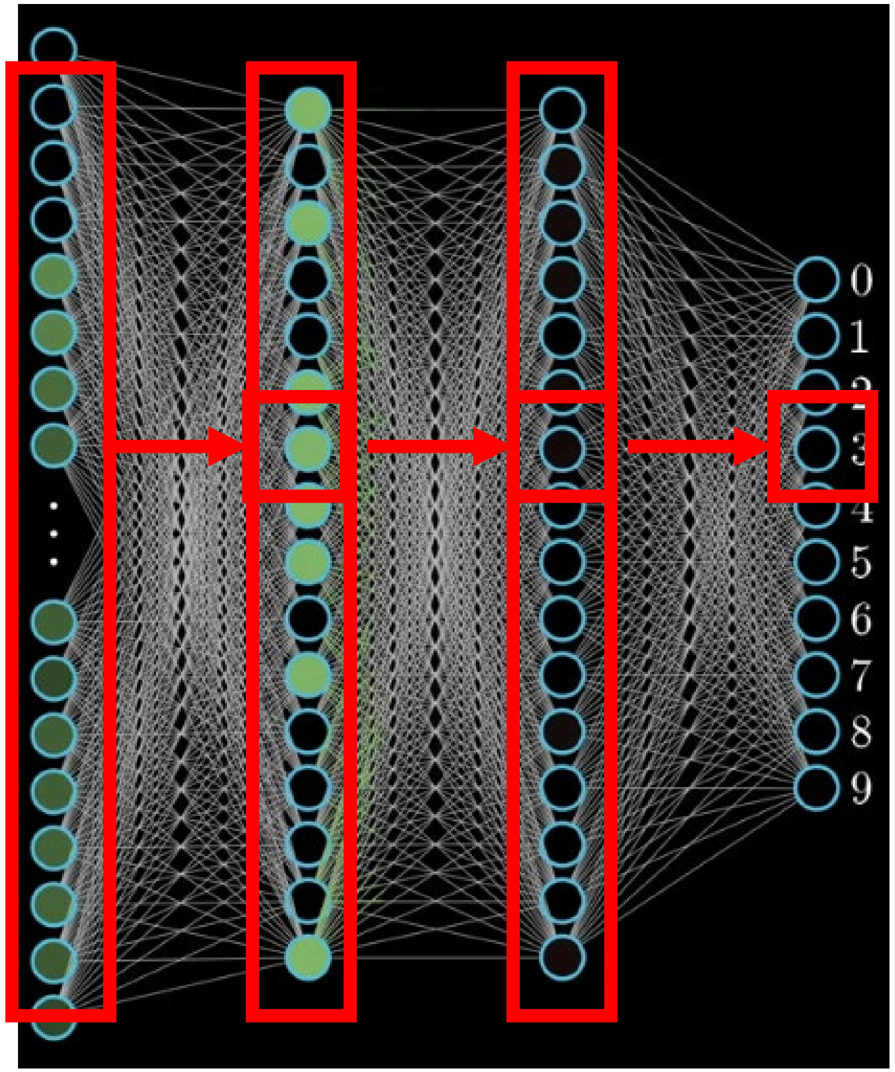
$$f(x) = \frac{1}{x} \qquad g(z) = 1 + e^{-z}$$

$$f'(x) = -\frac{1}{x^2} \qquad g'(z) = -e^{-z}$$

$$\begin{aligned}\sigma'(z) &= f'(g(z))g'(z) \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \sigma(z)(1-\sigma(z))\end{aligned}$$

Backpropagation

[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]



Activation at final layer involves weighted combination of activations at previous layer…

$$\sigma(w^T x)$$

Which involves a weighted combination of the layer before it…

$$\sigma(w_n^T \sigma(w_{n-1}^T x))$$

And so on…

$$\sigma(w_n^T \sigma(w_{n-1}^T \sigma(w_{n-2}^T \sigma(\ldots))))$$

Backpropagation

Backpropagation is the procedure of repeatedly applying the derivative chain rule to compute the full derivative

Example

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$

$$\frac{d}{dz}\sigma(\sigma(z)) = \sigma(\sigma(z))(1 - \sigma(\sigma(z)))\frac{d}{dz}\sigma(z)$$

This is simply the derivative chain rule applied through the entire network, from the output to the input

Backpropagation

- Implementation-wise all we need is a function that computes the derivative of each nonlinear activation

- We can repeatedly call this function, starting at the end of the network and moving backwards

- In practice, neural network implementations use *auto differentiation* to compute the derivative on-the-fly very

- Can do this efficiently on *graphical processing units (GPUs)* on extremely large training datasets

Universal Approximation Theorem

(Informally) For *any* function $f(x)$ there exists a multilayer perceptron that approximates $f(x)$ with arbitrary accuracy.

- Specific cases for arbitrary depth (number of hidden layers) and arbitrary width (number of nodes in a layer)
- Not a constructive proof (doesn't guarantee you can learn parameters)
- Corollary : The multilayer perceptron is a *universal turing machine*
- Also means it can easily overfit training data (regularization is critical)

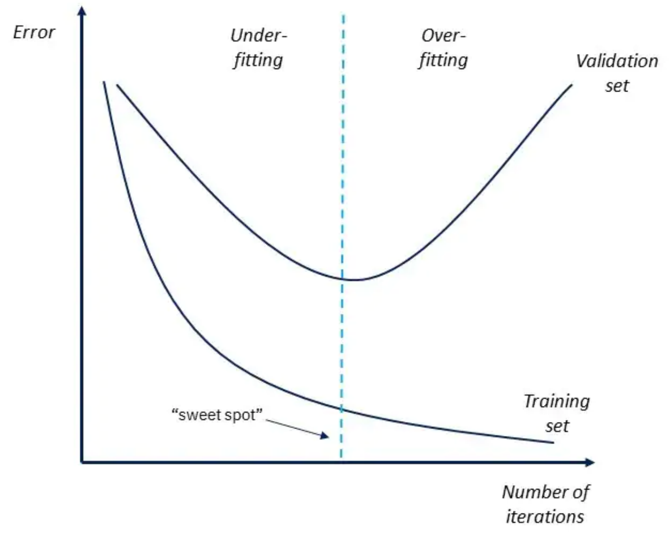
Regularization

With four parameters I can fit an elephant. With five I can make him wiggle his trunk. - John von Neumann

$$w = \arg\min_{w} \mathrm{Cost}(w) + \alpha \cdot \mathrm{Regularizer}(\mathrm{Model})$$

Our example model has 13,002 parameters…that's a lot of elephants! Regularization is critical to avoid overfitting…

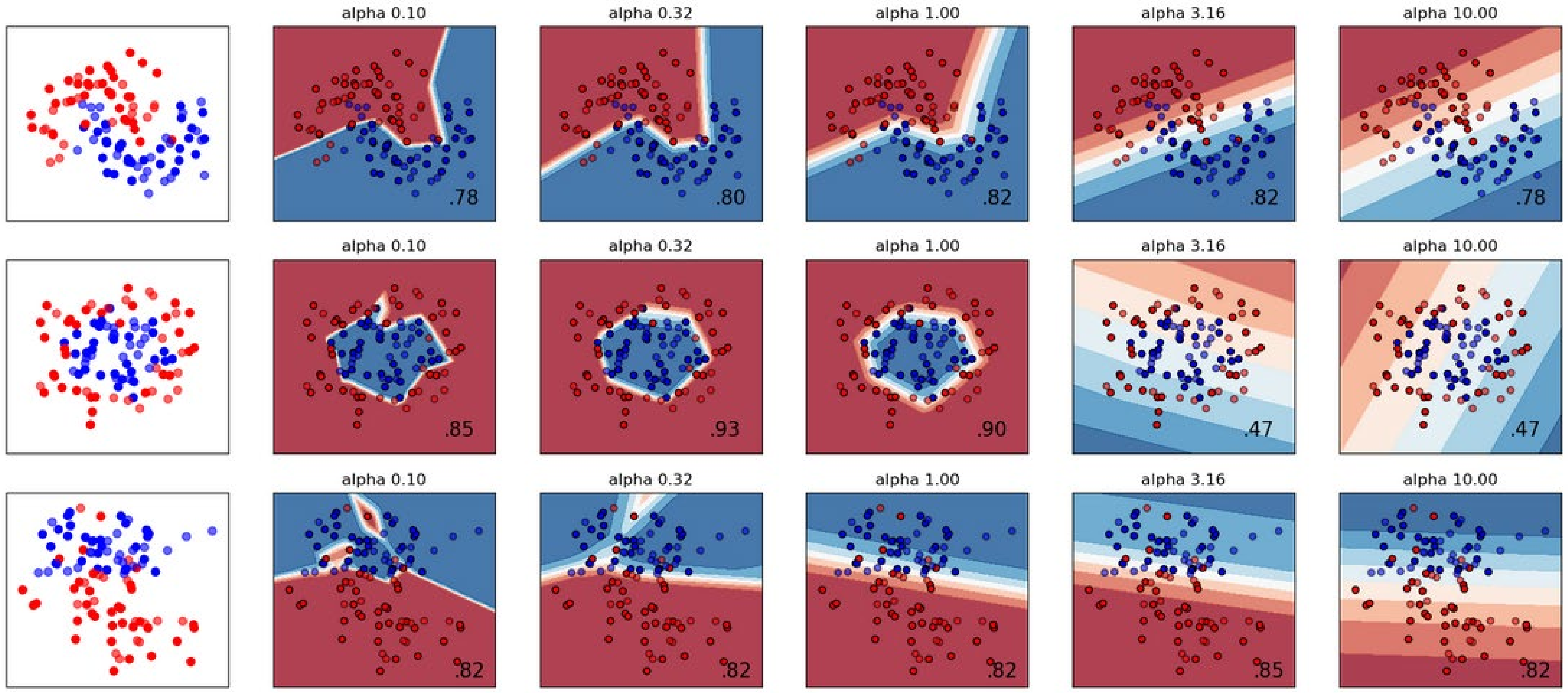
…numerous regularization schemes are used in training neural networks



Regularization : Weight Decay

In neural network speak, adding an L2 penalty is called *weight decay*

$$w = \arg\min_{w} \text{Cost}(w) + \frac{\alpha}{2}\|w\|^2$$



Regularization

- L1 regularization and L1+L2 (elastic net) regularization
- **Dropout** Each iteration randomly selects a small number of edges to temporarily exclude from the network (weights=0)
  - **Intuition** Avoids predictions that are overly sensitive to any small number of edges
- **Early stopping** Just as it sounds…stop the network before reaching a local minimum…dumb-but-effective

sklearn.neural_network.MLPClassifier

hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)*

The ith element represents the number of neurons in the ith hidden layer.

activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

solver : *{'lbfgs', 'sgd', 'adam'}, default='adam'*

The solver for weight optimization.

alpha : *float, default=0.0001*

L2 penalty (regularization term) parameter.

learning_rate : *{'constant', 'invscaling', 'adaptive'}, default='constant'*

Learning rate schedule for weight updates.

early_stopping : *bool, default=False*

Whether to use early stopping to terminate training when validation score is not improving. If set to true,

Scikit-Learn : Multilayer Perceptron

Fetch MNIST data from www.openml.org :

```
X, y = fetch_openml("mnist_784", version=1, return_X_y=True)
X = X / 255.0
```

Train test split (60k / 10k),

```
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]
```

Create MLP classifier instance,

- Single hidden layer (50 nodes)
- Use stochastic gradient descent
- Maximum of 10 learning iterations
- Small L2 regularization alpha=1e-4

```
mlp = MLPClassifier(
    hidden_layer_sizes=(50,),
    max_iter=10,
    alpha=1e-4,
    solver="sgd",
    verbose=10,
    random_state=1,
    learning_rate_init=0.1,
)
```

Scikit-Learn : Multilayer Perceptron

Fit the MLP and print stuff…

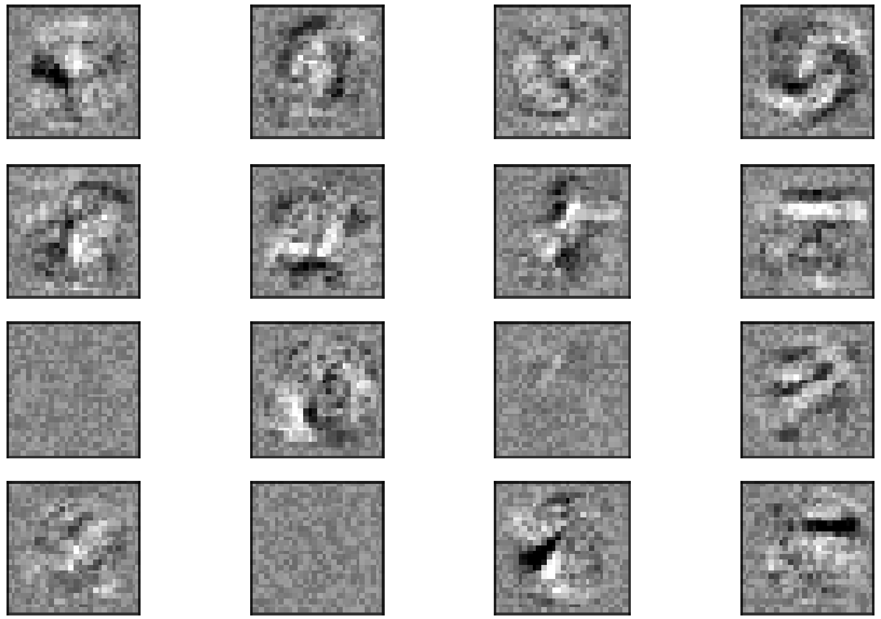
```
mlp.fit(X_train, y_train)
```

```
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

Visualize the weights for each node…

```
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray,
               vmin=0.5 * vmin, vmax=0.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())
```



…magnitude of weights indicates which input features are important in prediction

More Advanced Topics

Many other NN architectures exist beyond MLP

- **Convolutional NN (CNN)** For image processing / computer viz.
- **Recurrent NN (RNN)** For sequence data (e.g. acoustic signals, video, etc.) , long short-term memory (LSTM) is popular
- **Generative Adversarial Nets (GANs)** For generating creepy deepfakes
- **Restricted Boltzmann Machine (RBM)** Another generative model

Many open areas being researched

- More reliable uncertainty estimates
- Robustness to exploits
- Interpretability
- Better scalability

Resources

There are **tons** of excellent resources for learning about neural networks online…here are two quick ones:

3Blue1Brown Youtube channel has a nice four-part intro:

https://www.youtube.com/watch?v=aircAruvnKk

Free book by Michael Nielson uses MNIST example in Python:

http://neuralnetworksanddeeplearning.com/

Prof. Stephen Bethard often teaches an excellent class:

ISTA 457 / INFO 557