



Computer  
Science

# **CSC580: Principles of Data Science**

## **Nonlinear Models**

Jason Pacheco

# Outline

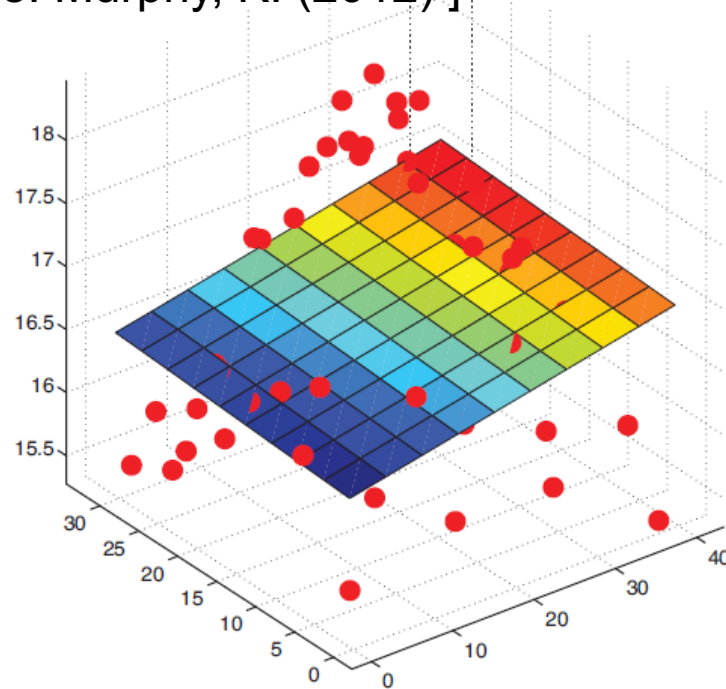
- Basis Functions
- Support Vector Machine Classifier
- Kernels
- Neural Networks

# Outline

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

# Linear Models

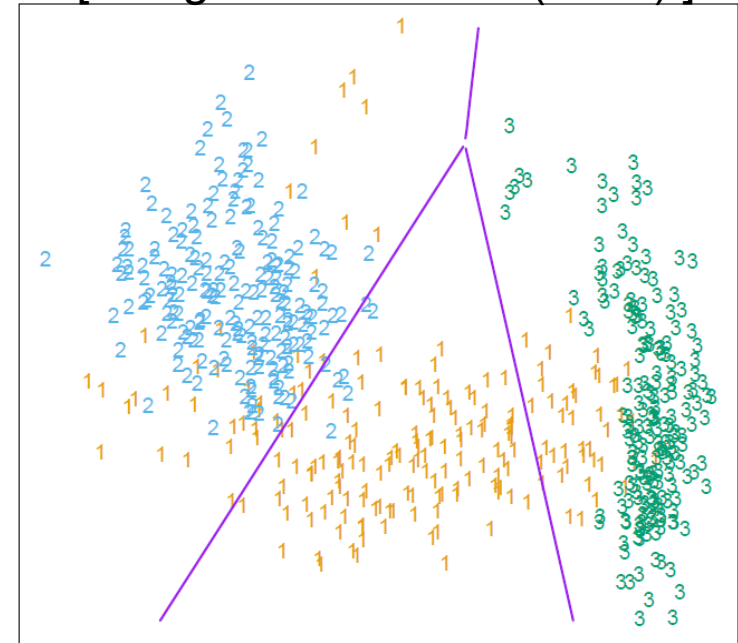
[ Image: Murphy, K. (2012) ]



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

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

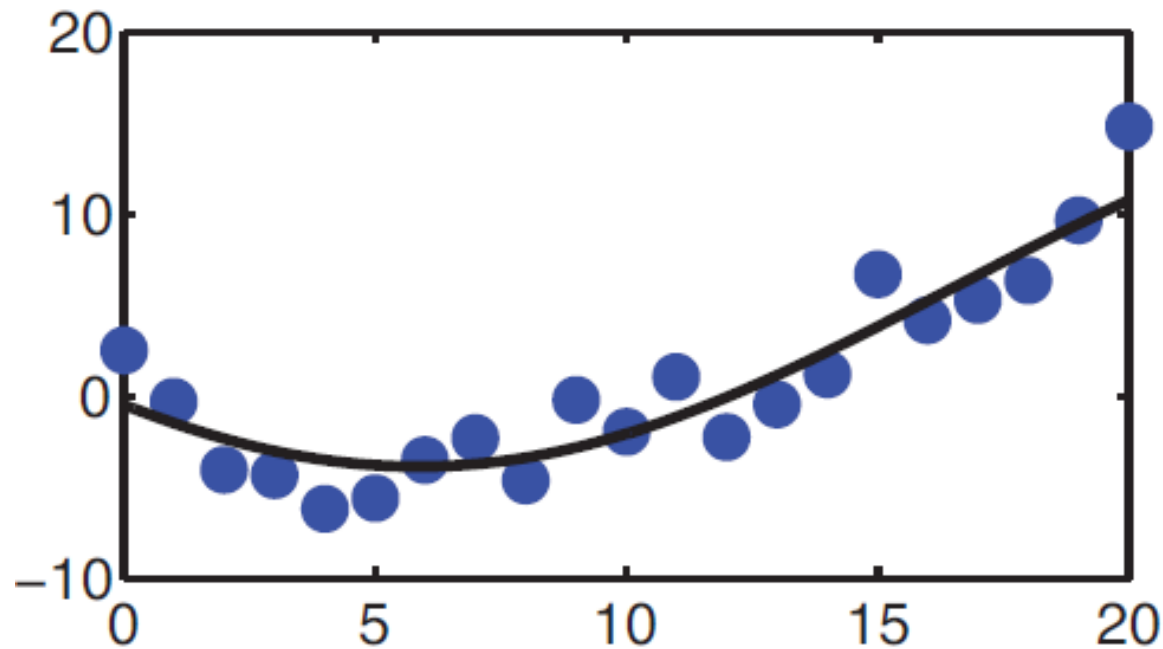
[ Image: Hastie et al. (2001) ]



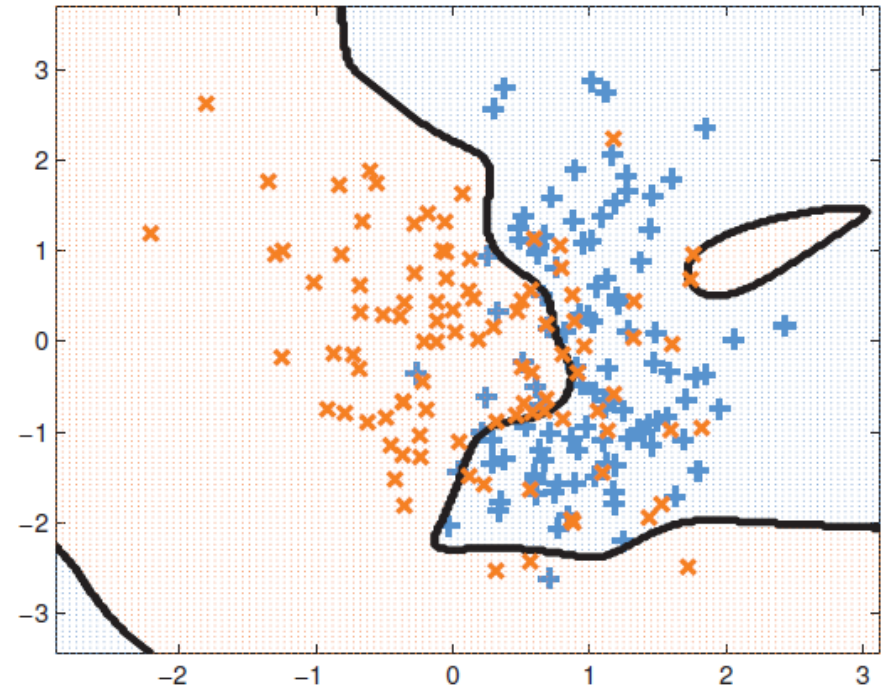
**Logistic Regression** Learn a decision boundary that is *linear in the data*,

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

# Nonlinear Data



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

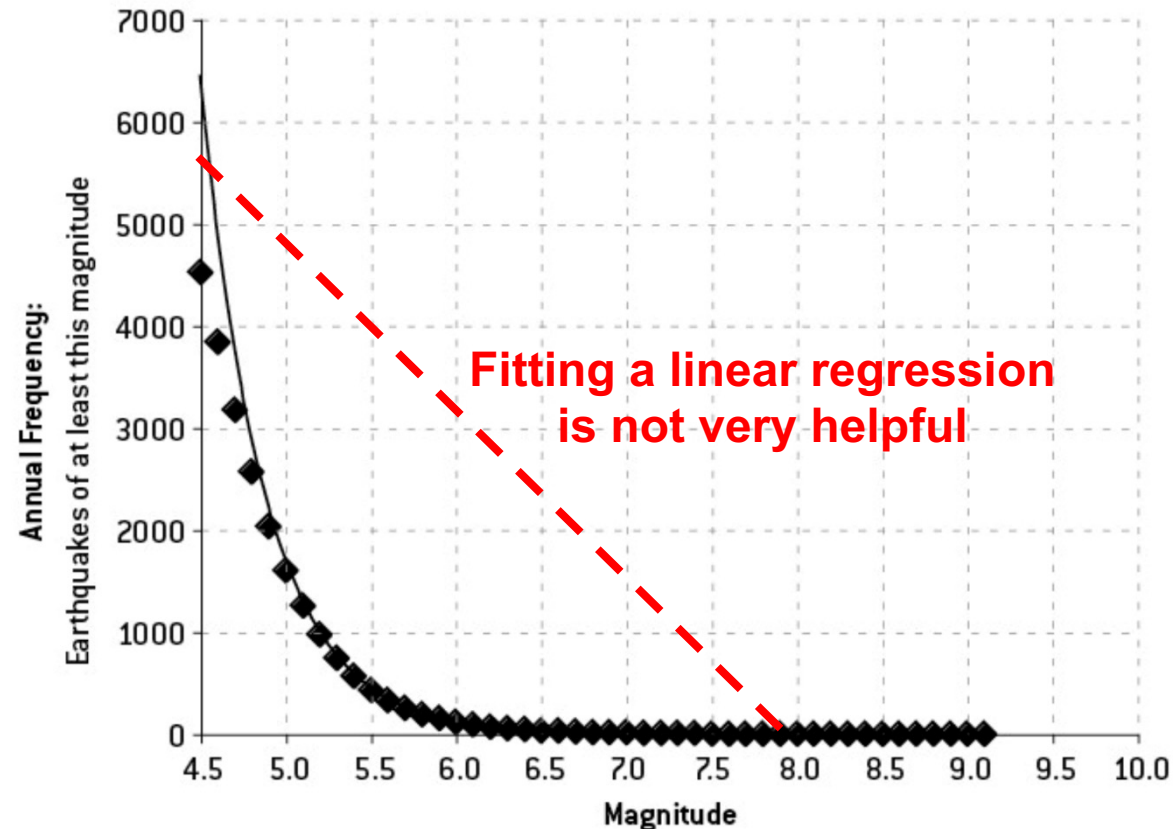


What if classes are *not linearly-separable*?

# Example: Earthquake Prediction

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

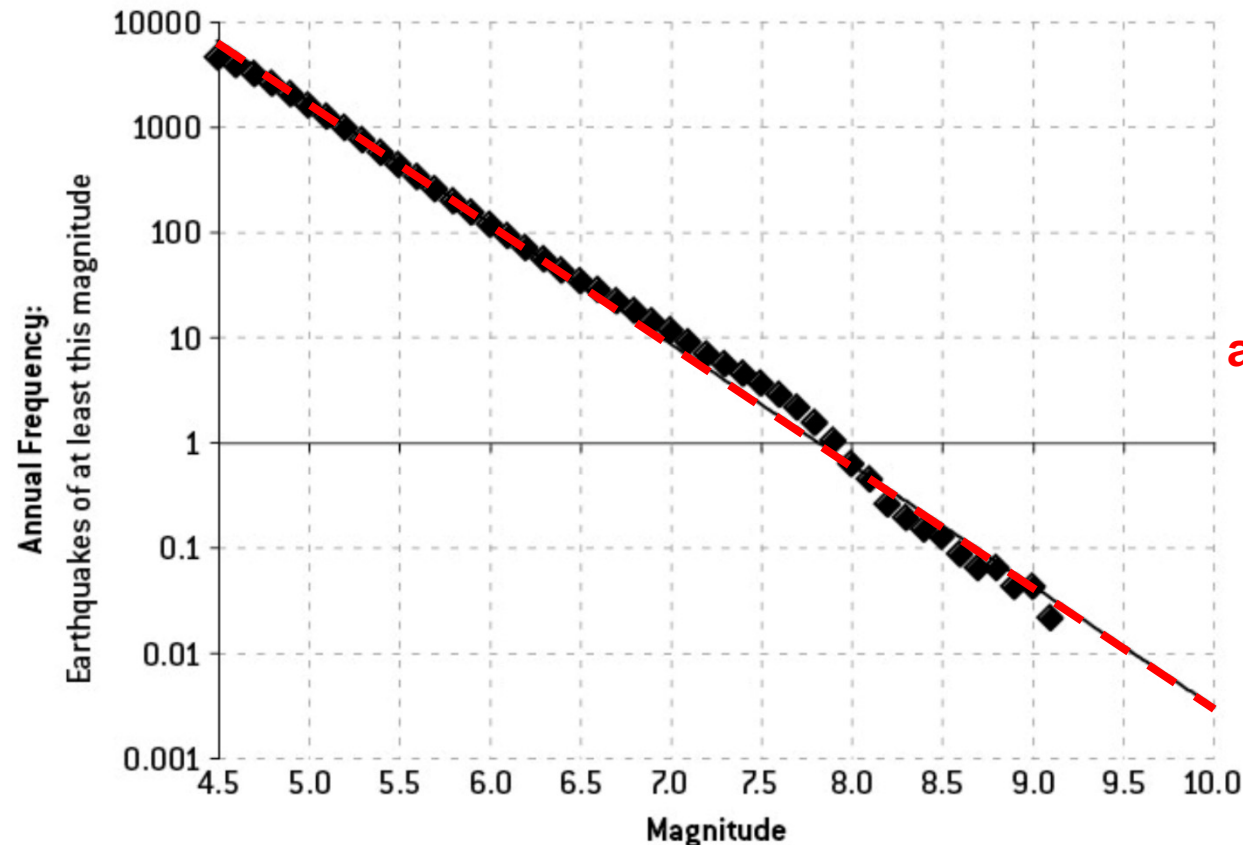
FIGURE 5-3A: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964–MARCH 2012



# 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

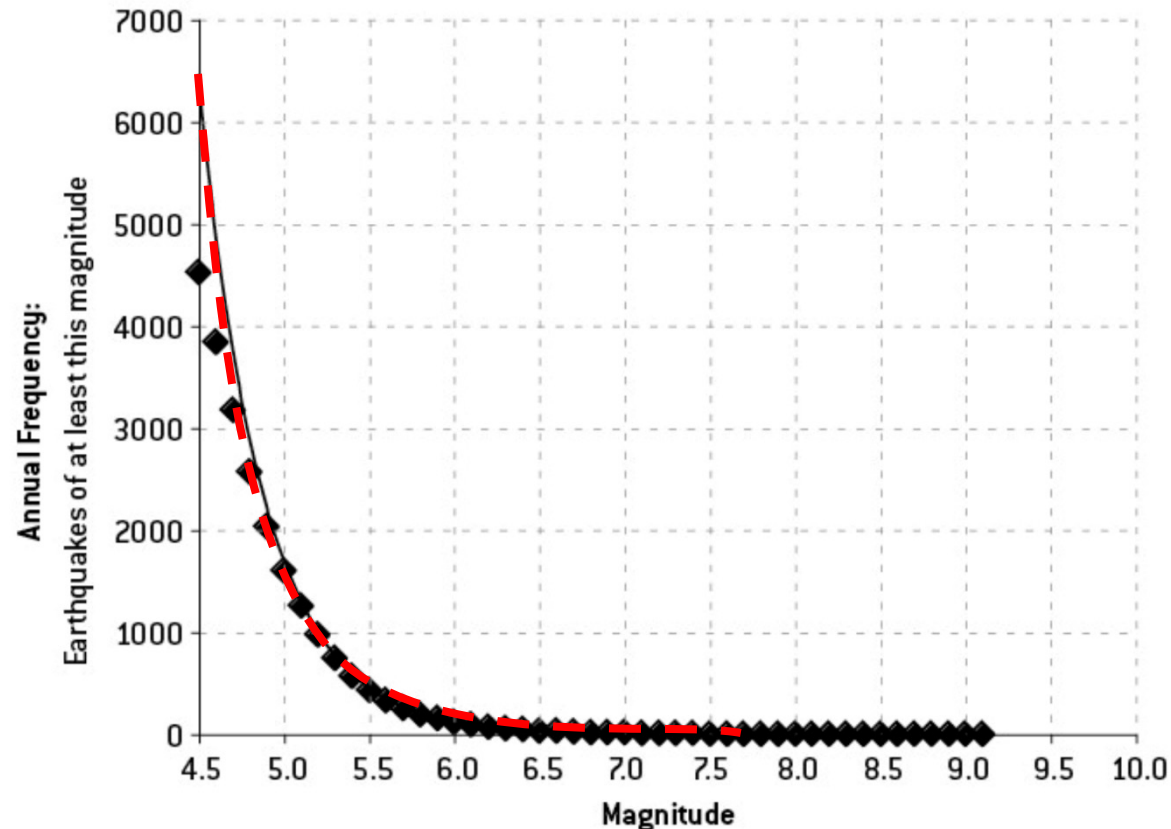


**But plotting outputs on a logarithmic scale reveals a strong linear relationship...**

# 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  $x$ ,

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



# Basis Functions

- A **basis function** can be any function of the input features  $X$
- Define a set of  $m$  basis functions  $\phi_1(x), \dots, \phi_m(x)$
- 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,

$$\phi_m(x) = x_m \quad \text{Returns } m^{\text{th}} \text{ dimension of } X$$

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

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

- 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  $(1, x_1, x_2, x_1^2, x_1x_2, x_2^2)$  ,

```
>>> 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 3rd order polynomial (cubic) regression,

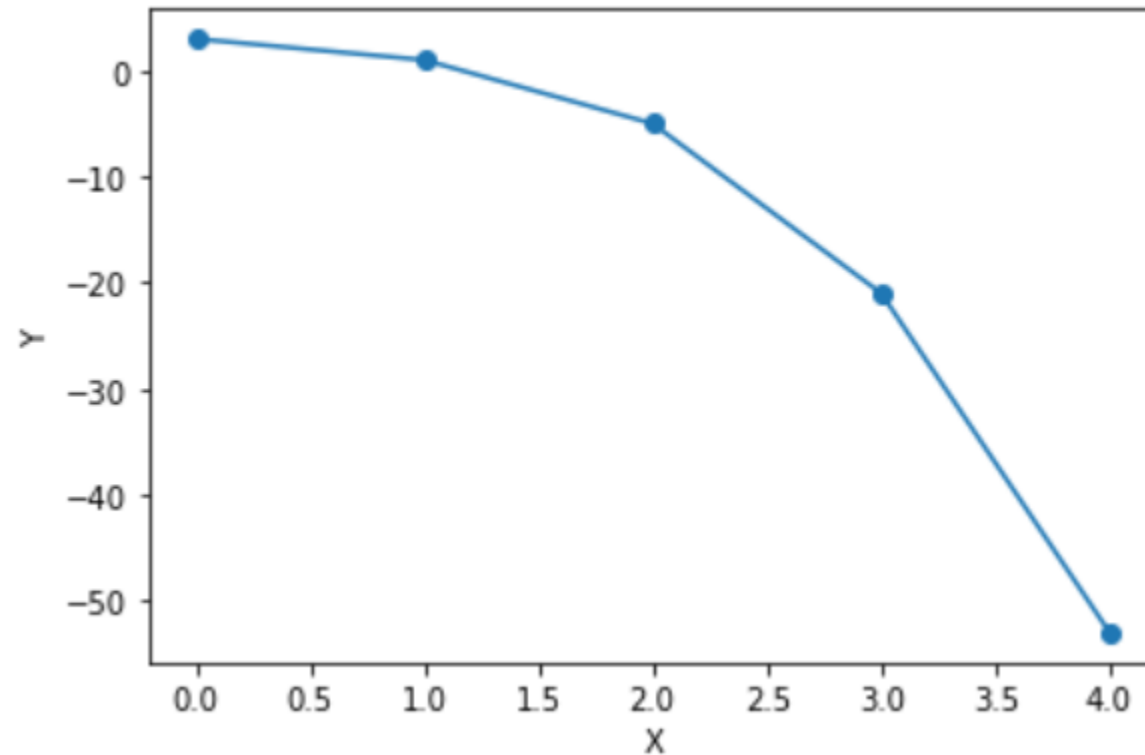
```
from sklearn.preprocessing import PolynomialFeatures
x = np.arange(5)
y = 3 - 2 * x + x ** 2 - x ** 3
y
array([ 3,  1, -5, -21, -53])
```

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,

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

**Design Matrix**  
( each training input on a column )

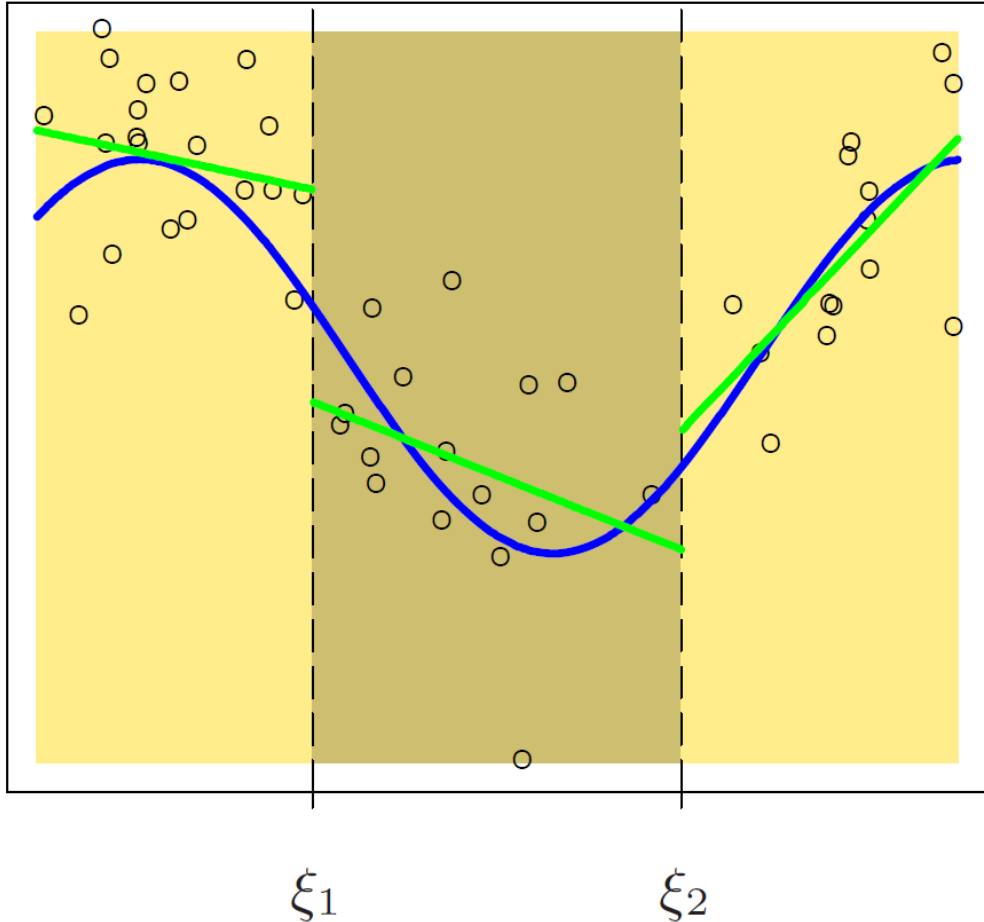
**Vector of**  
**Training labels**

Can similarly solve in terms of basis functions,

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

# 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,

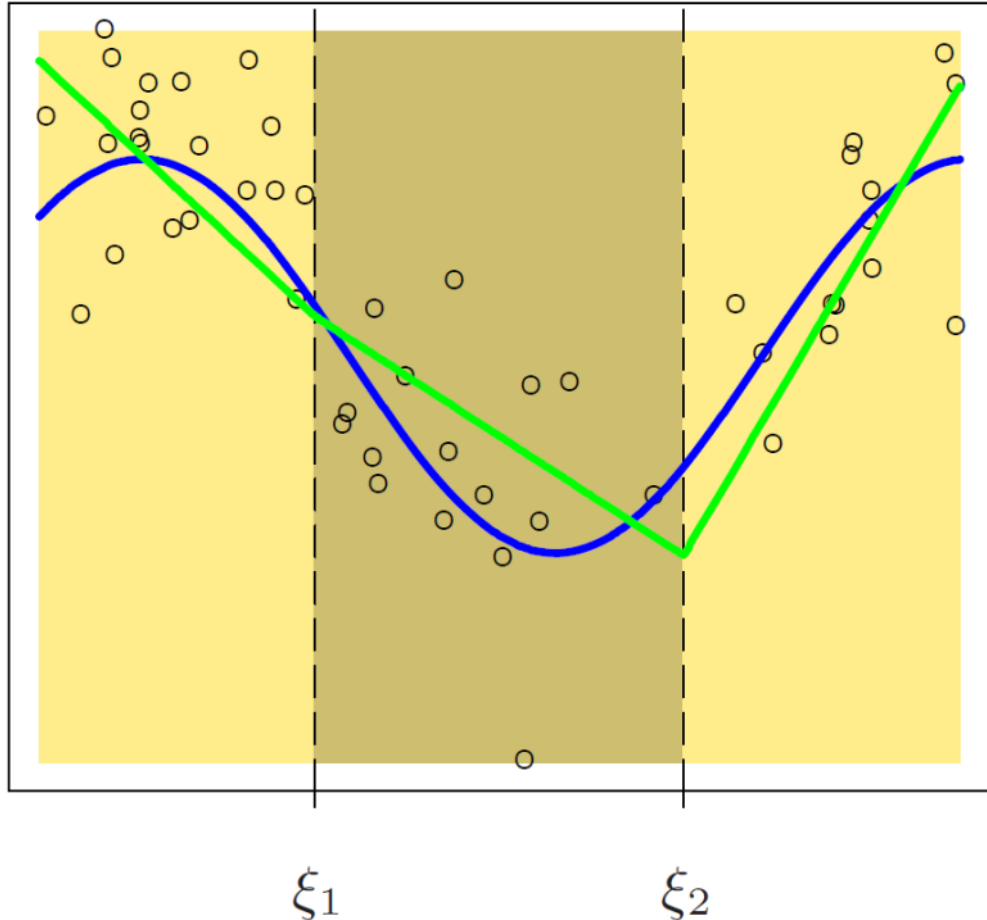
$$y = w_1\phi_1(x) + w_2\phi_2(x) + w_3\phi_3(x)$$

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,

$$\phi_1(x) = 1$$

$$\phi_2(x) = x$$

$$\phi_3(x) = (x - \xi_1)_+$$

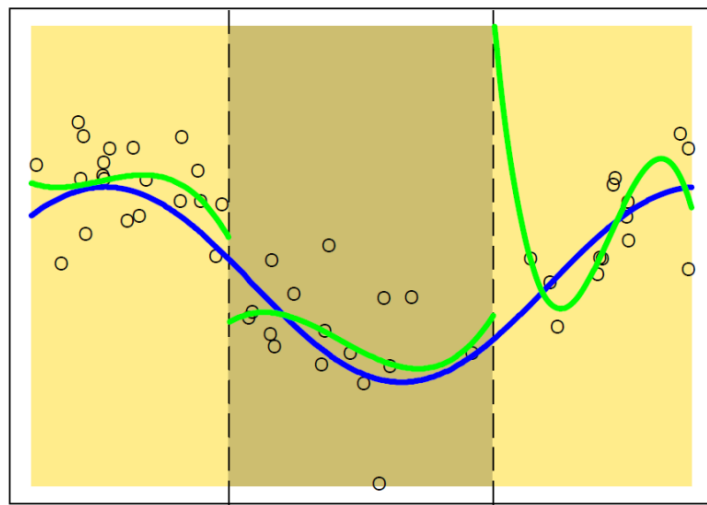
$$\phi_4(x) = (x - \xi_2)_+$$

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

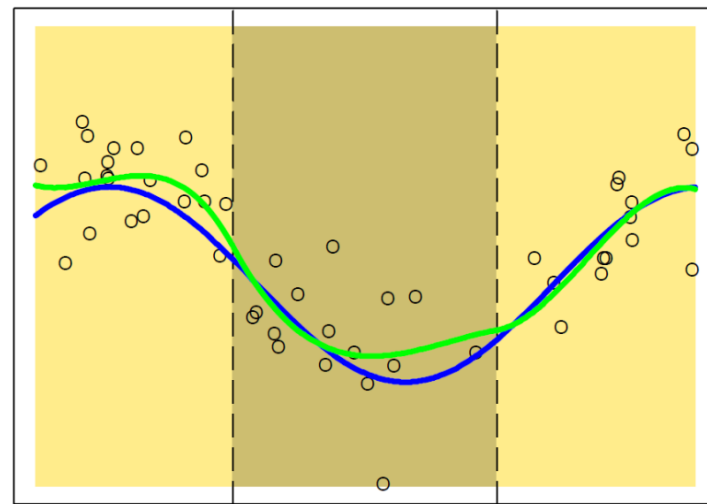
**An improvement, but generally prefer *smoother* functions...**

[Source: Hastie et al. (2001)]

Discontinuous



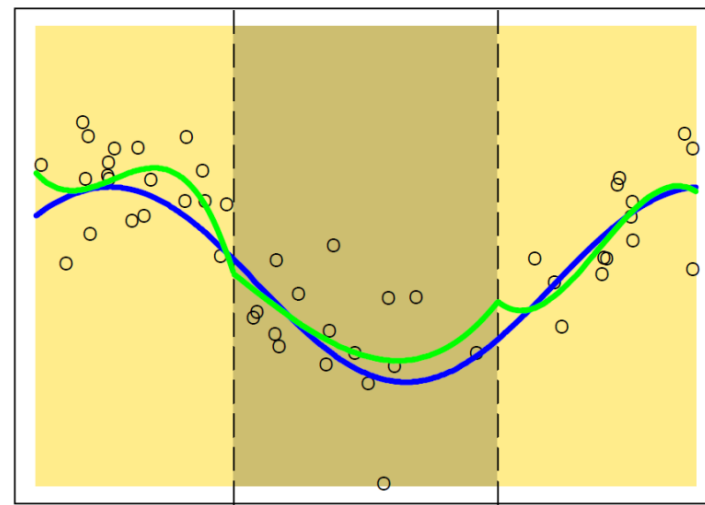
Continuous First Derivative



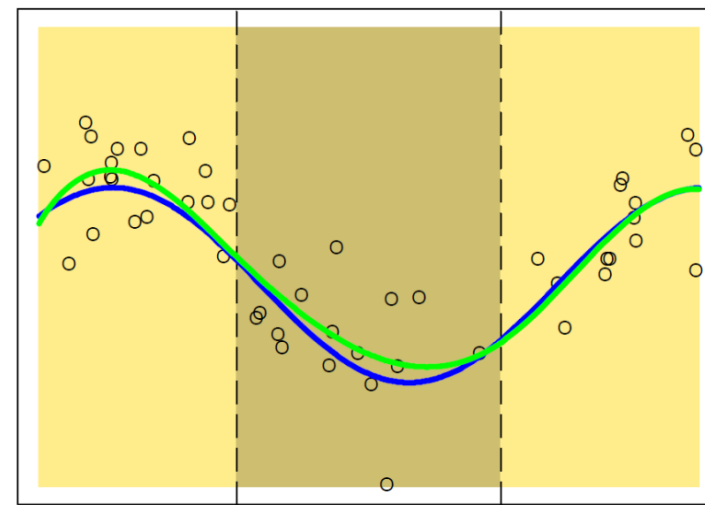
$\xi_1$

$\xi_2$

Continuous



Continuous Second Derivative



$\xi_1$

$\xi_2$

Replace linear basis functions with polynomial,

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

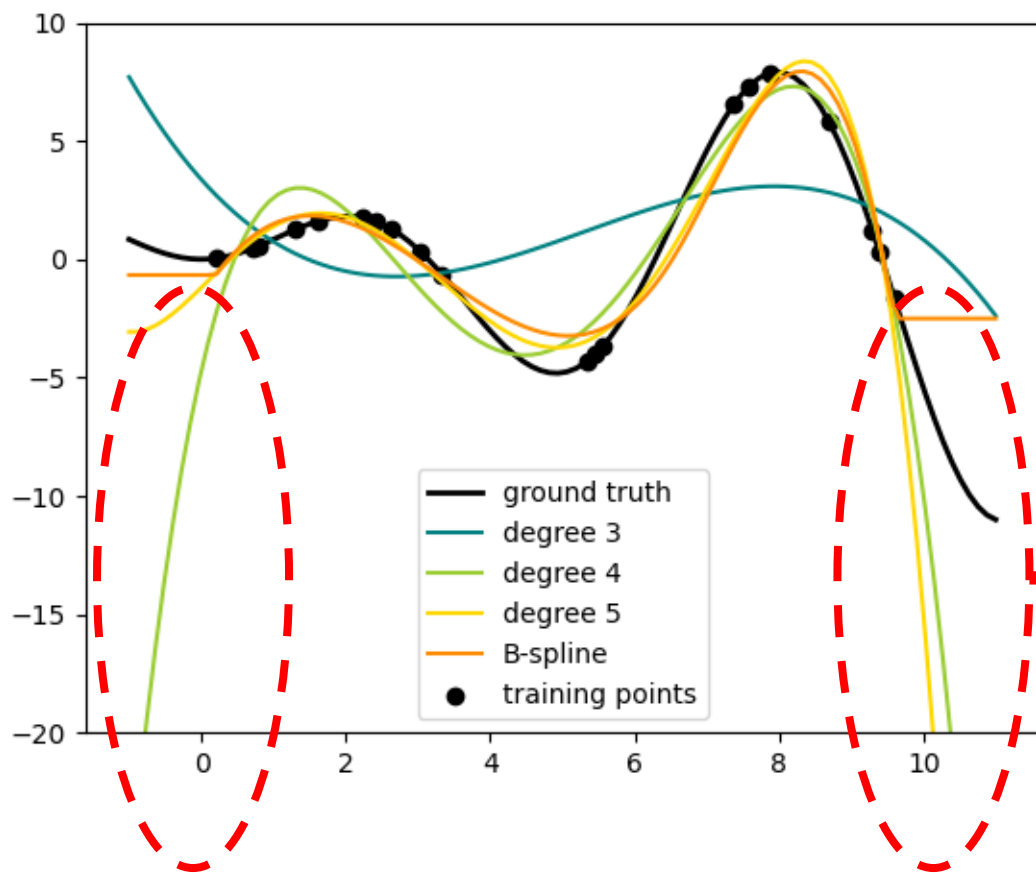
$$\phi_3(x) = x^2 \quad \phi_4(x) = x^3$$

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

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

Additional constraints ensure smooth 1<sup>st</sup> and 2<sup>nd</sup> 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

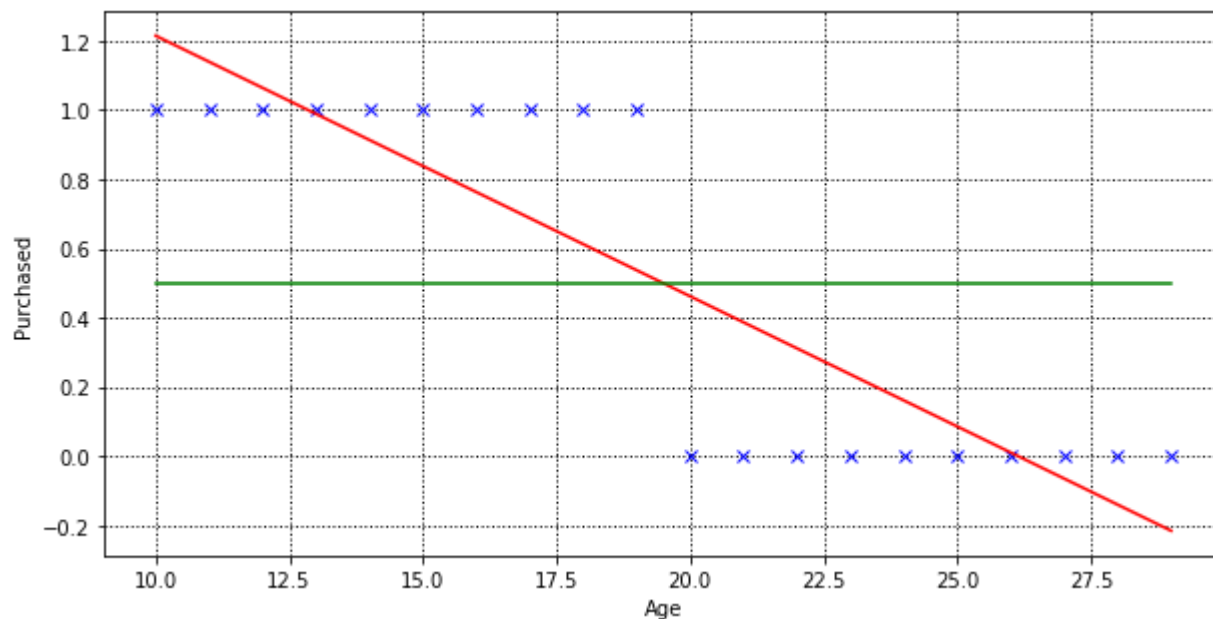
## CONs

- 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

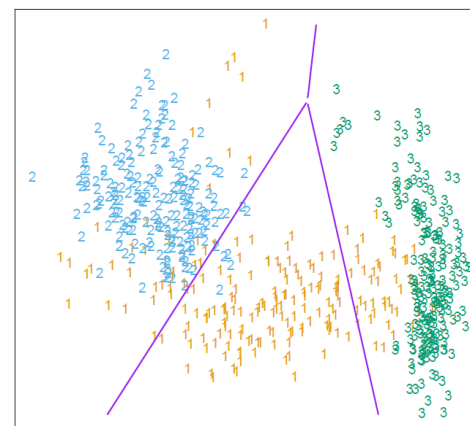
# Classification as Regression



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

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

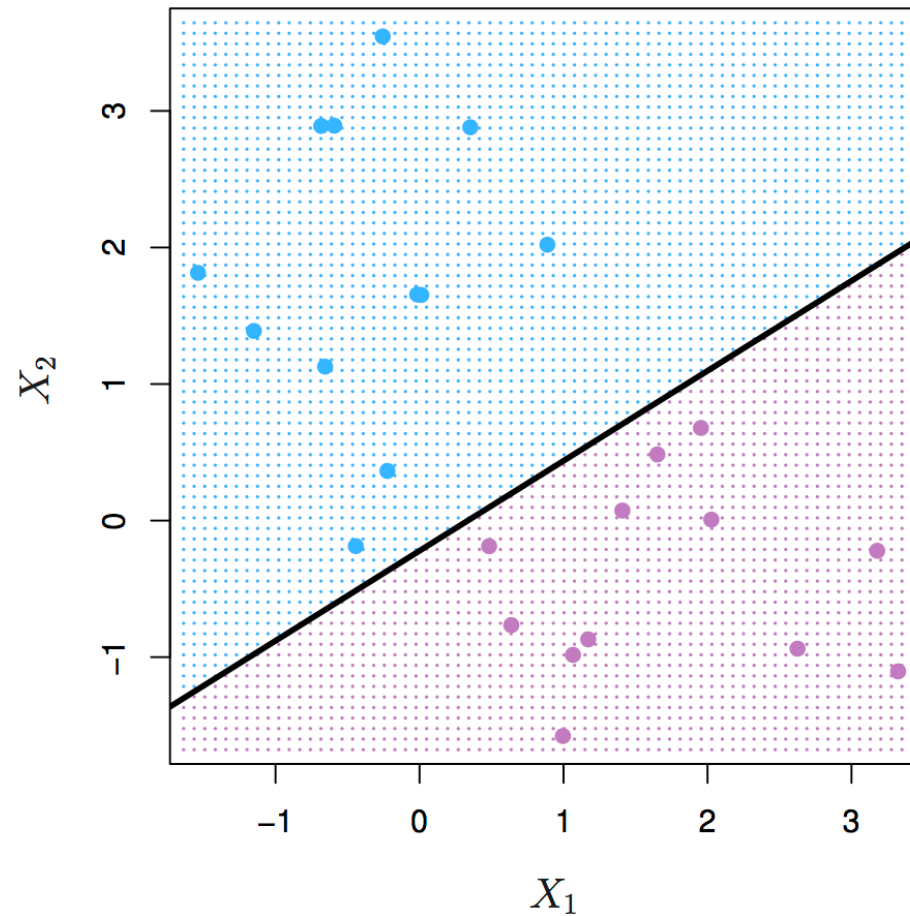
- This is a *discriminant* function, since it discriminates between classes
- It is a linear function and so is a *linear discriminant*
- Green line is the *decision boundary* (also linear)



**Generalizes to  
higher-dimensional  
features**

# Linear Decision Boundary

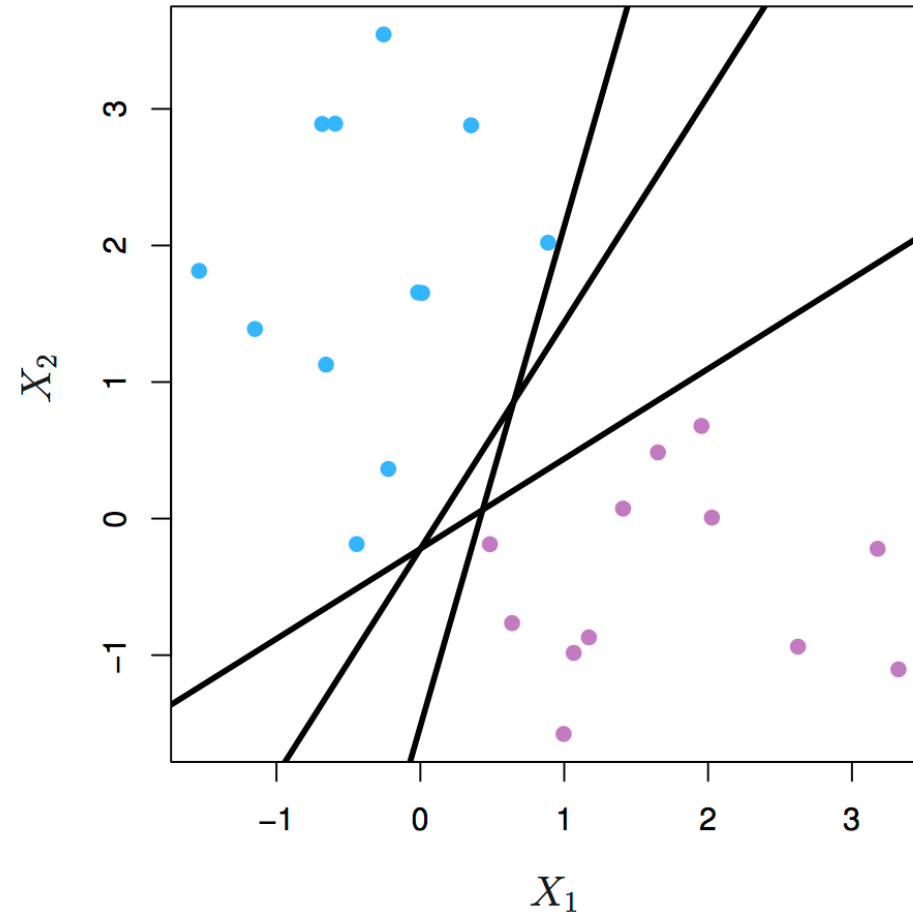
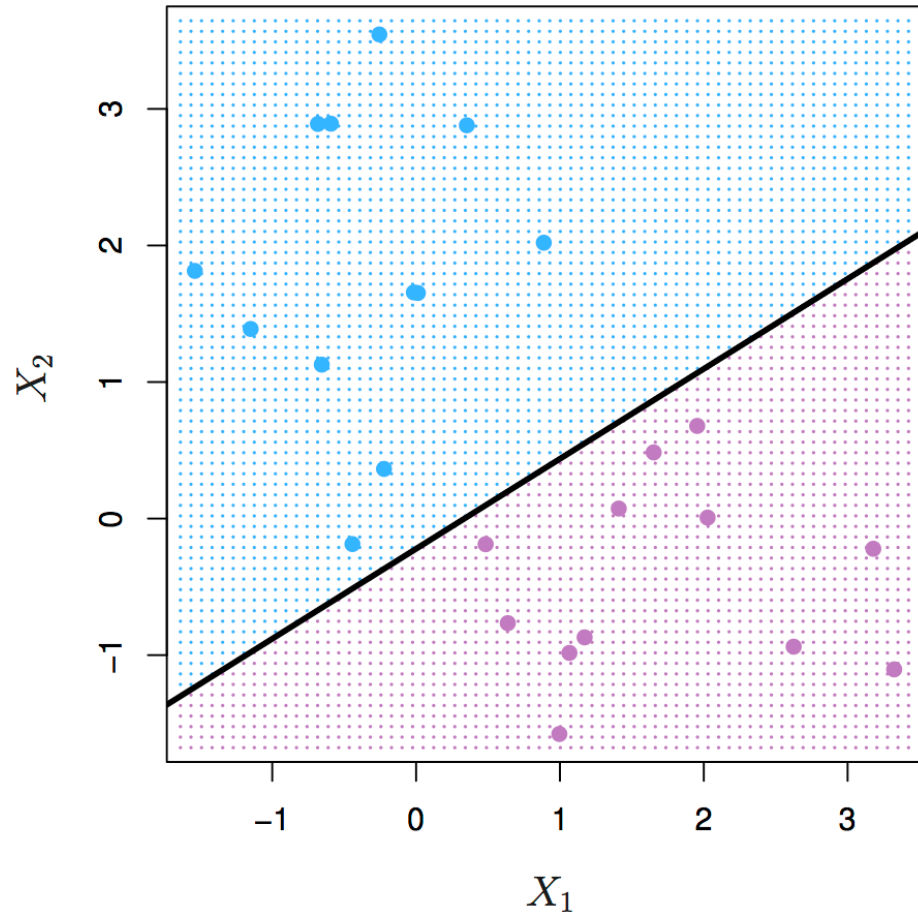
*Least squares regression yields decision boundary based on least squares solution...*



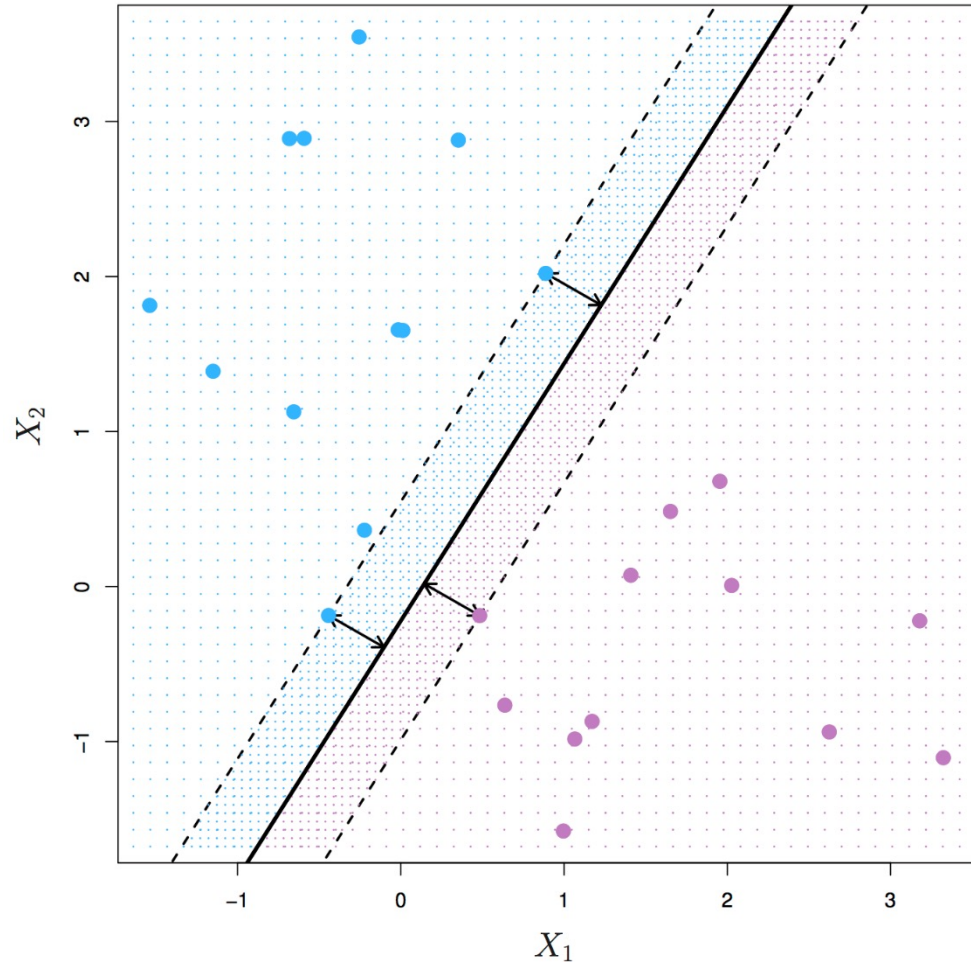


# Linear Decision Boundary

*...any boundary that separates classes is equivalently good on training data*



# 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*

# 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,

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

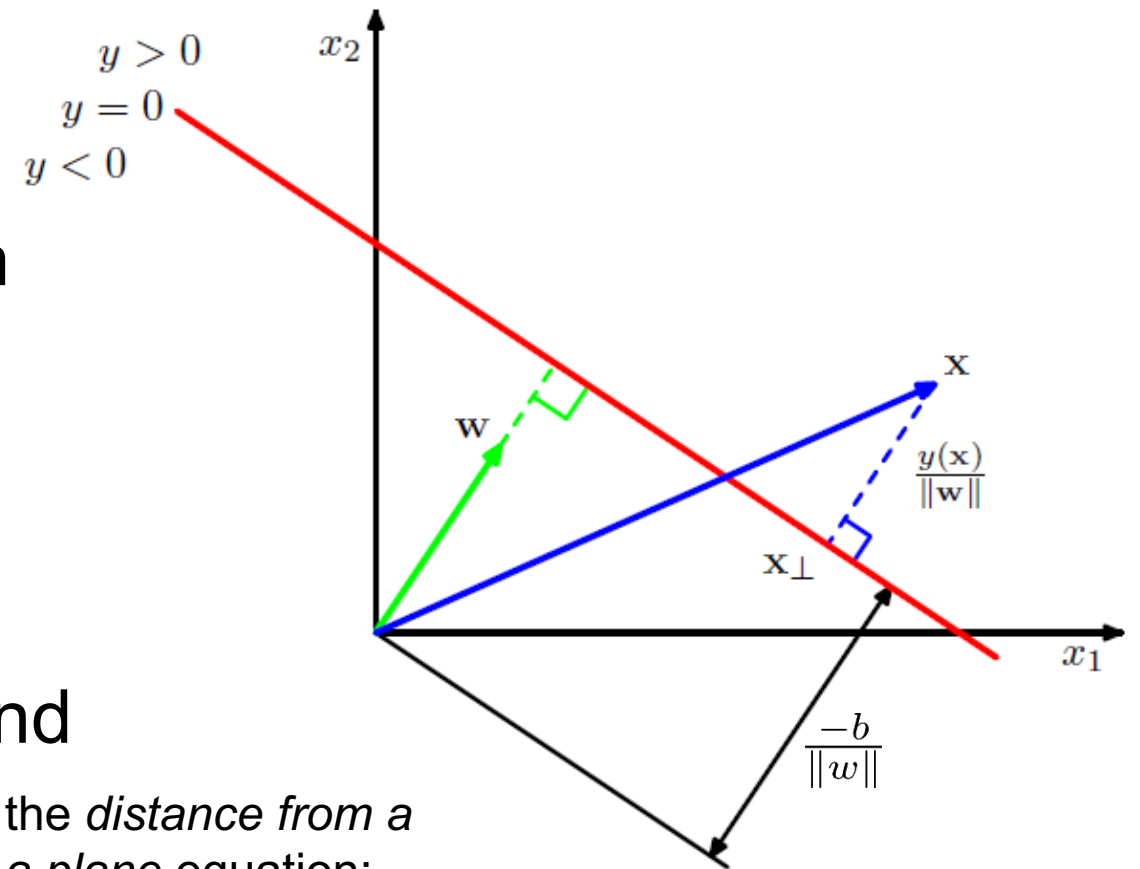
Decision boundary is now at  $y(x) = 0$  and distance to the margin is,

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

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

[wiki/Distance\\_from\\_a\\_point\\_to\\_a\\_plane](https://en.wikipedia.org/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,

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

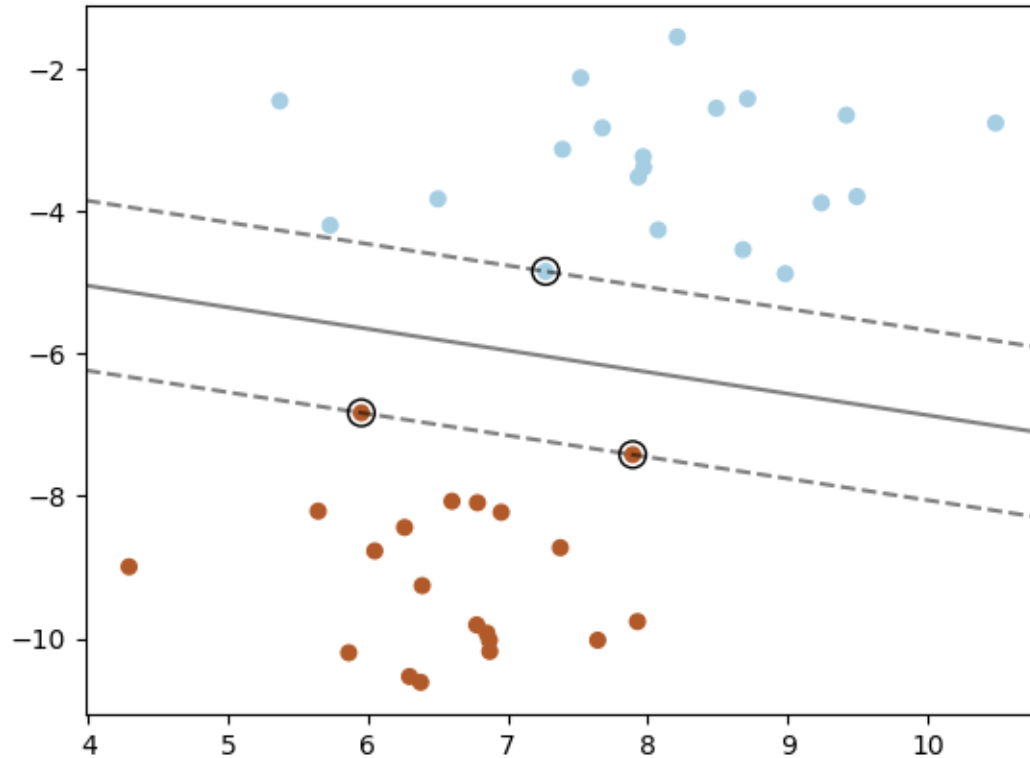
The margin of correctly-classified points is then given by,

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

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



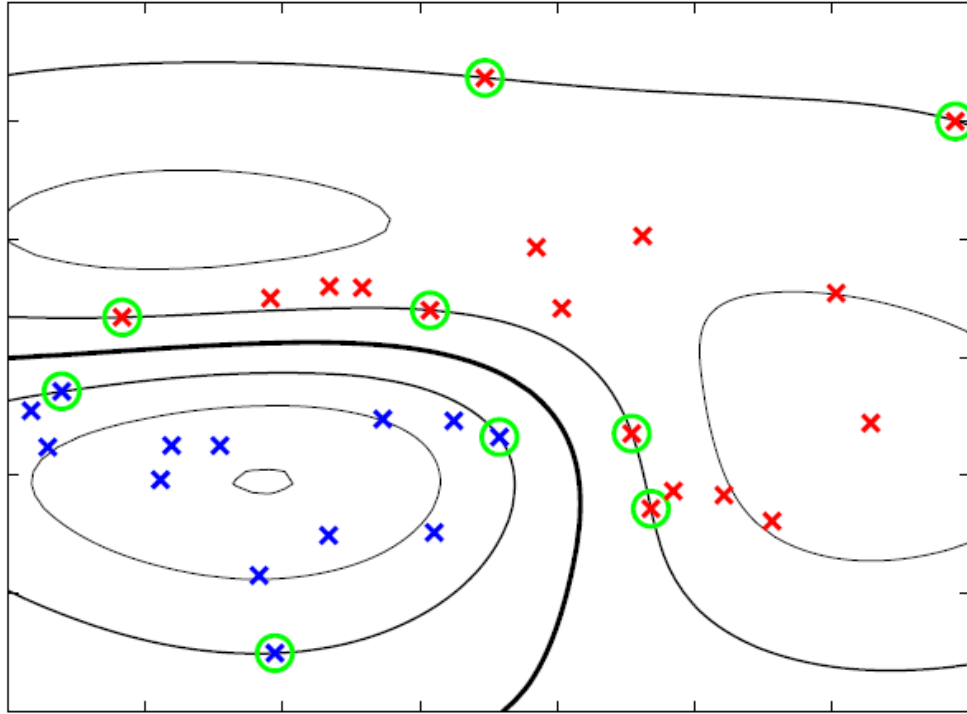
Maximize the  
minimum margin

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

Minimum margin over  
all training data

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

# Nonlinear Max-Margin Classifier



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

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

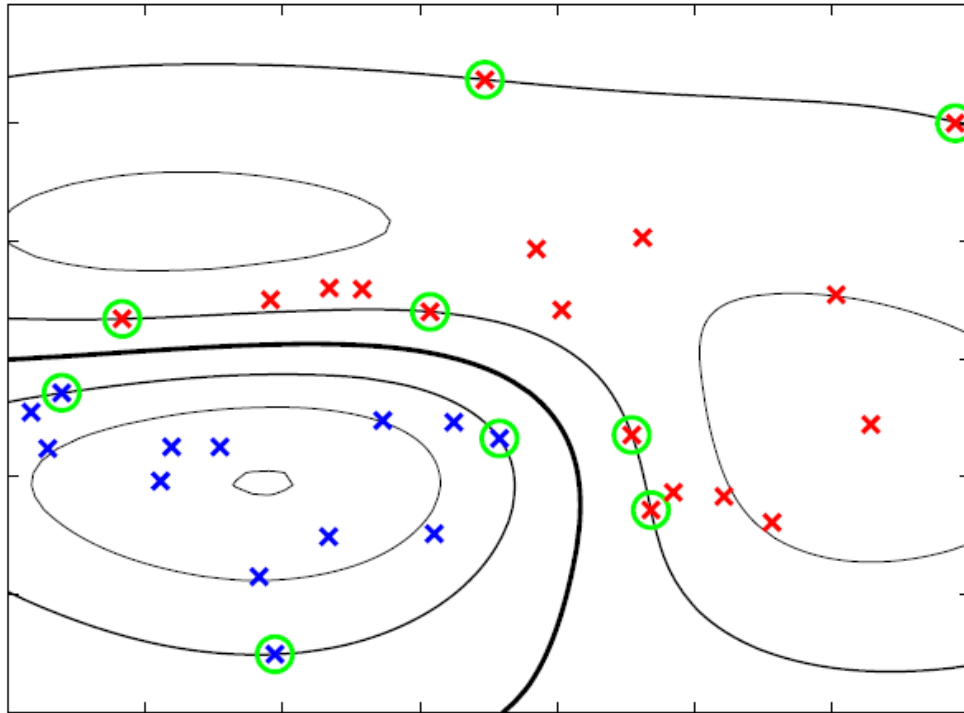
*Max-margin learning is similar,*

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

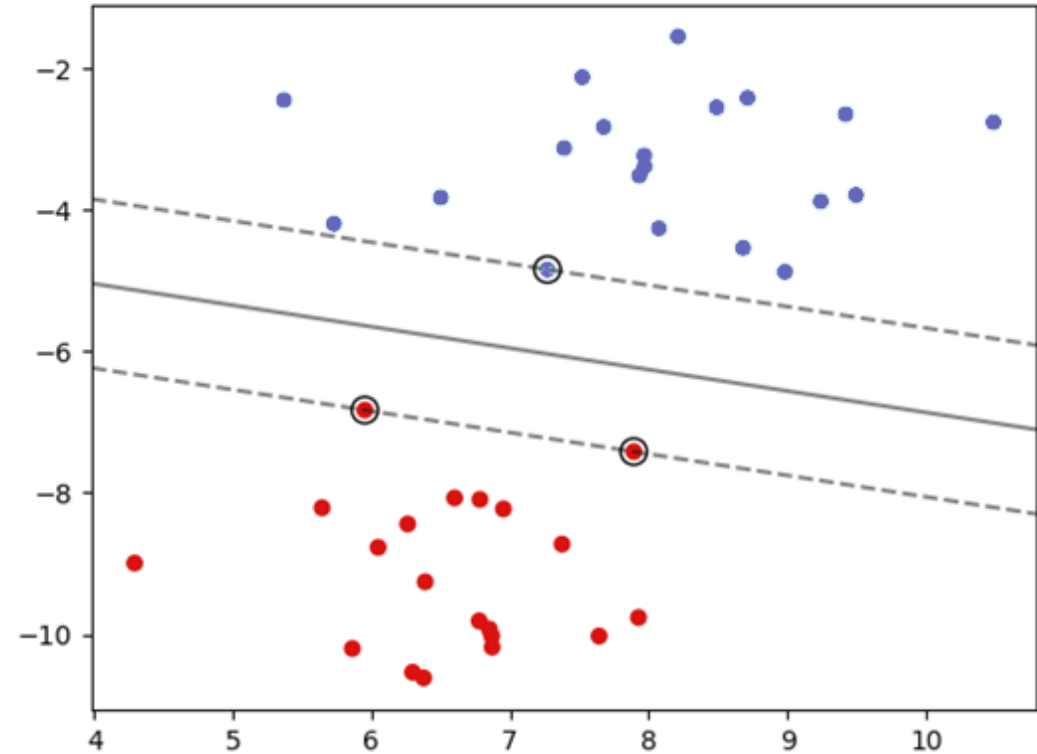
Decision boundary is linear in the transformed data, but nonlinear in the original data space

# 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,

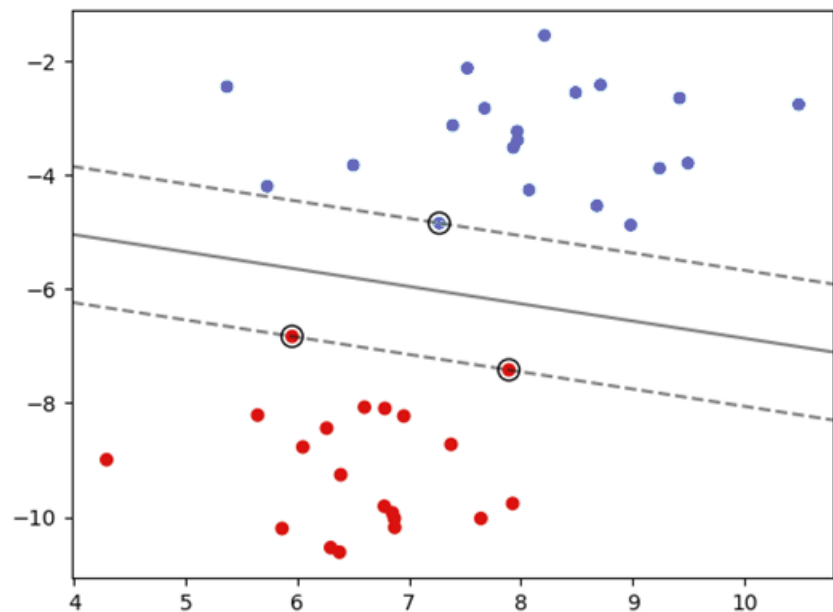
$$y_n (w^T \phi(x_n) + b) \geq 1$$

Now we just have to satisfy these constraints...



# Support Vector Machine (Primal)

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



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

subject to

$$y_n(w^T x_n + b) \geq 1 \quad \text{for } n = 1, \dots, N$$

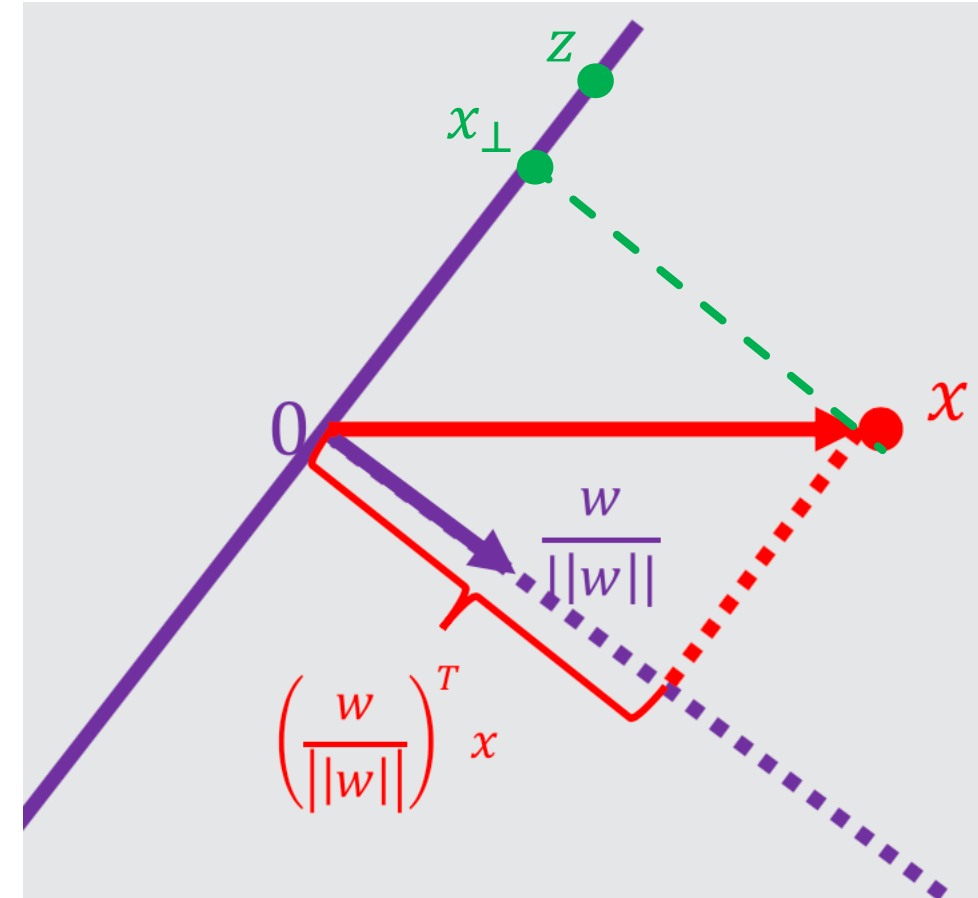
**This is known as the  
primal optimization**

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)

# Facts on vectors

- (Lem 1) a vector  $x$  has distance  $\frac{w^\top x}{\|w\|}$  to the hyperplane  $w^\top x = 0$
- How about with bias?  $w^\top x + b = 0$
- Let us be explicit on the bias:  $f(x; w, b) = w^\top x + b$
- recall:  $w$  is orthogonal to the hyperplane  $w^\top x + b = 0$ 
  - why? (left as exercise)



# Facts on vectors

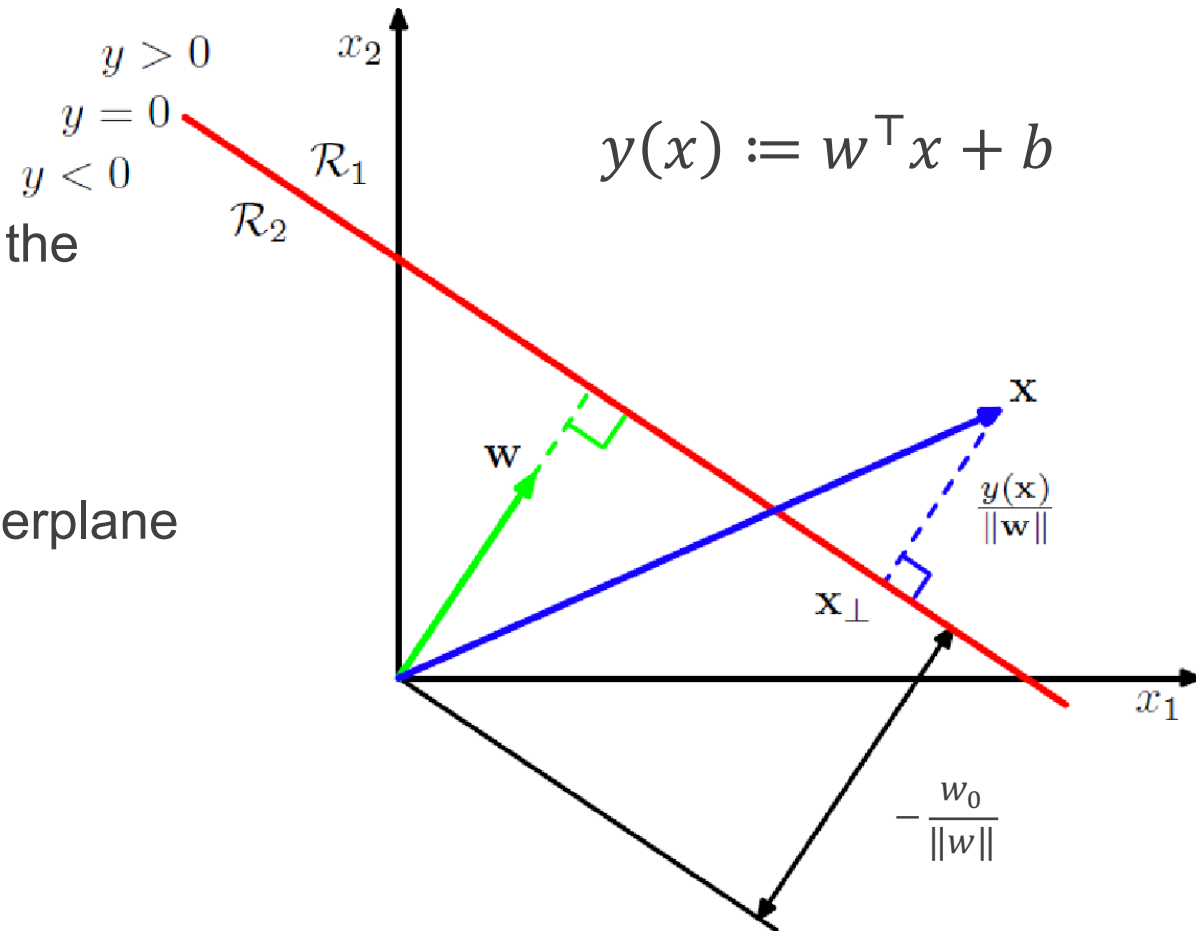
- (Lem 2)  $x$  has distance  $\frac{|w^\top x + b|}{\|w\|}$  to the hyperplane  $w^\top x + b = 0$

claim1 :  $x$  can be written as  $x = x_\perp + r \frac{w}{\|w\|}$  where  $x_\perp$  is the projection of  $x$  onto the hyperplane.

claim2 : then,  $|r|$  is the distance between  $x$  and the hyperplane

Solving for  $r$ :  $w^\top x + b = w^\top x_\perp + r \frac{w^\top w}{\|w\|} + b = r \|w\|$ .

this implies  $|r| = \frac{|w^\top x + b|}{\|w\|}$



# SVM derivation (1)

- Margin of  $(w, b)$  over all training points:  $\gamma'(w, b) = \min_i \frac{|w^\top x_i + b|}{\|w\|}$

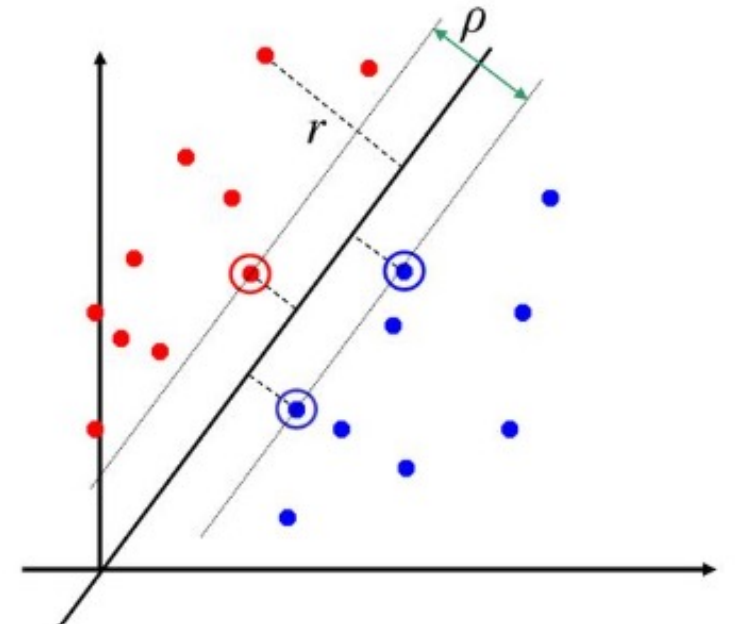
- Choose  $(w, b)$  with the maximum margin? .. wait, we also want it to be a perfect classifier
  - redefine it

$$\gamma(w, b) = \min_i \frac{y_i(w^\top x_i + b)}{\|w\|}$$

- Choose  $w$  with the maximum margin (and perfect classification)

$$(\hat{w}, \hat{b}) = \max_{w, b} \min_{i=1}^n \frac{y_i(w^\top x_i + b)}{\|w\|}$$

- One more issue: still, infinitely many solutions..!



# SVM derivation (2)

- Infinitely many solutions..
- It's actually a matter of removing 'duplicates';  $\exists$  many  $(w,b)$ 's that actually represent the same hyperplane.

$$(\hat{w}, \hat{b}) = \max_{w,b} \min_{i=1}^n \frac{y_i(w^\top x_i + b)}{\|w\|}$$

- Quick solution = achieves the smallest margin
  - For any solution  $(\hat{w}, \hat{b})$ , let  $x_{i^*}$  be the **closest to the hyperplane**  $\hat{w}x_i + \hat{b} = 0$
  - Imagine rescaling  $(\hat{w}, \hat{b})$  so that  $|\hat{w}^\top x_{i^*} + \hat{b}| = 1$
- We can always do that, but can we find a formulation that automatically finds that modified solution?
  - add the constraint  $\min_i y_i(w^\top x_i + b) = 1$

# SVM derivation (3)

$$\begin{aligned} & \max_{w,b} \min_{i=1}^n \frac{y_i(w^\top x_i + b)}{\|w\|} \\ & \text{s.t. } \min_i y_i(w^\top x_i + b) = 1 \end{aligned}$$

- Summary: the constraint encodes (1) correct classification (2) there are no two solutions that represent the same hyperplane!

- Note: If  $(\hat{w}, \hat{b})$  is a solution, then the margin is  $\frac{1}{\|\hat{w}\|}$

$$\begin{aligned} & \max_{w,b} \frac{1}{\|w\|} \\ & \text{s.t. } \min_i y_i(w^\top x_i + b) = 1 \end{aligned}$$

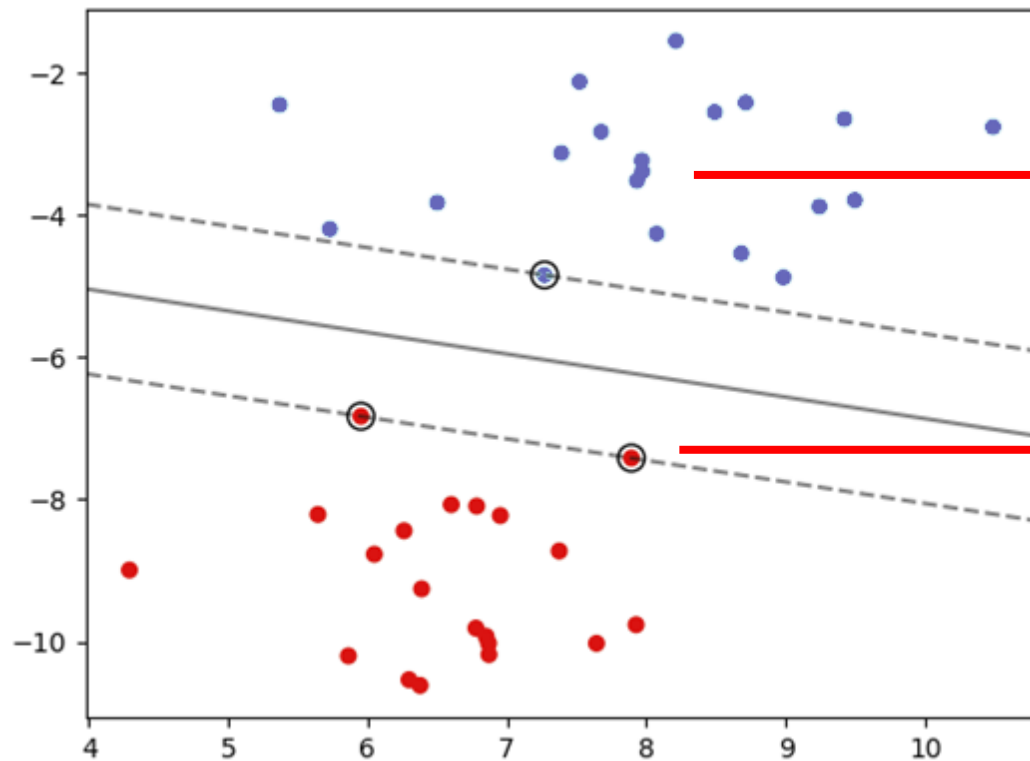
$$\begin{aligned} & \max_{w,b} \frac{1}{\|w\|} \\ & \text{s.t. } \min_i y_i(w^\top x_i + b) \geq 1 \\ & \text{(turns out to be equivalent..)} \end{aligned}$$

$$\begin{aligned} & \max_{w,b} \frac{1}{\|w\|} \\ & \text{s.t. } y_i(w^\top x_i + b) \geq 1, \forall i \end{aligned}$$

**Final formulation in the linearly separable setting:  
(quadratic programming)**

$$\begin{aligned} & \min_{w,b} \|w\|^2 \\ & \text{s.t. } y_i(w^\top x_i + b) \geq 1, \forall i \end{aligned}$$

# 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 the nonseparable setting: Soft-margin

$$\begin{aligned} & \min_{w,b} \|w\|^2 \\ \text{s.t. } & y_i(w^\top x_i + b) \geq 1, \forall i \end{aligned}$$

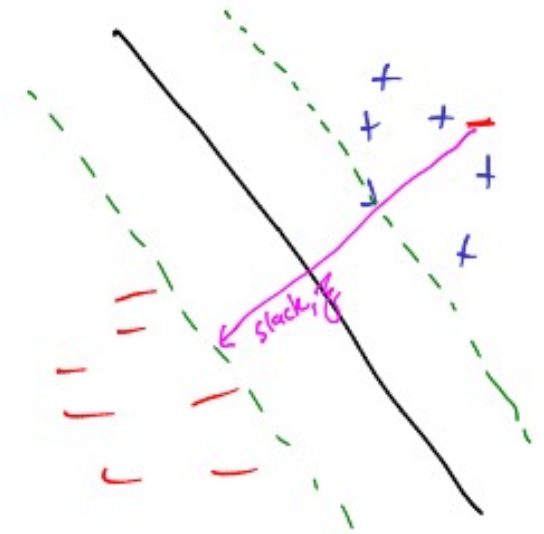
- What if data are **not** linearly separable?
- Introduce 'slack' variables

$$\begin{aligned} & \min_{w,b,\{\xi_i \geq 0\}} \|w\|^2 + C \sum_{i=1}^n \xi_i \quad // C \text{ is a hyper-parameter} \\ \text{s.t. } & y_i(w^\top x_i + b) \geq 1 - \xi_i, \forall i \end{aligned}$$

- Again, a quadratic programming problem
- Fix any  $w, b$ , the optimal  $\xi$ ?

$$\xi_i = 0 \text{ if } y_i(w^\top x_i + b) \geq 1, \text{ and } \xi_i = 1 - y_i(w^\top x_i + b)$$

$$\min_{w,b} \|w\|^2 + C \sum_{i=1}^n (1 - y_i(w^\top x_i + b))_+ \Leftrightarrow \text{Regularized hinge loss minimization } \lambda = \frac{1}{C}$$





# 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.

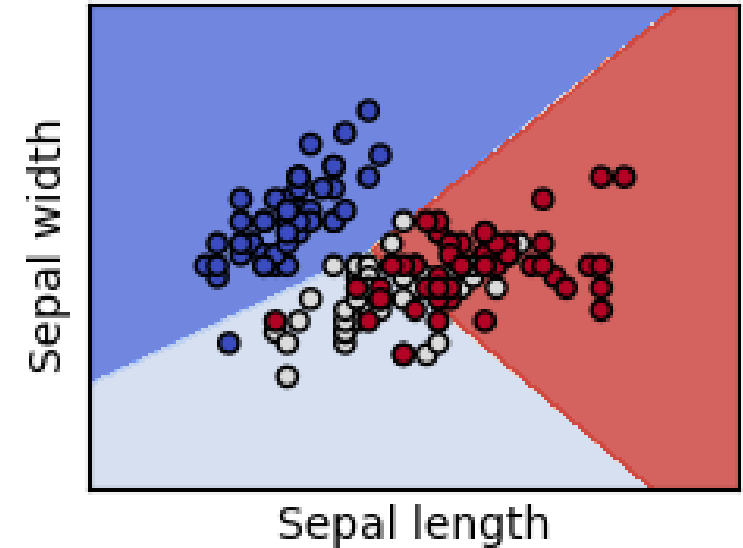
**dual** : bool, default=True

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.

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



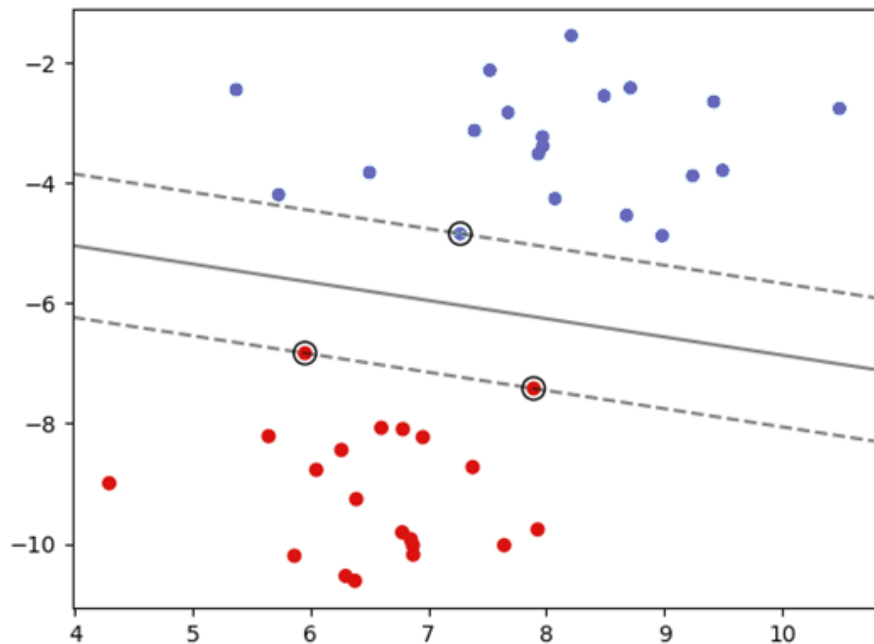
**Only showing linear  
for a reason that will  
be clear soon...**

# 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

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

**Basis transform on new point**      **Basis transform on training point**

**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”**

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

# Example: Fisher's Iris Dataset

*Classify among 3 species of Iris flowers...*



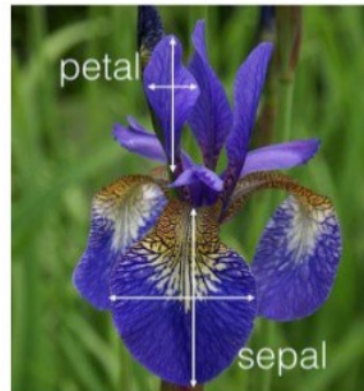
**Iris setosa**



**Iris versicolor**



**Iris virginica**

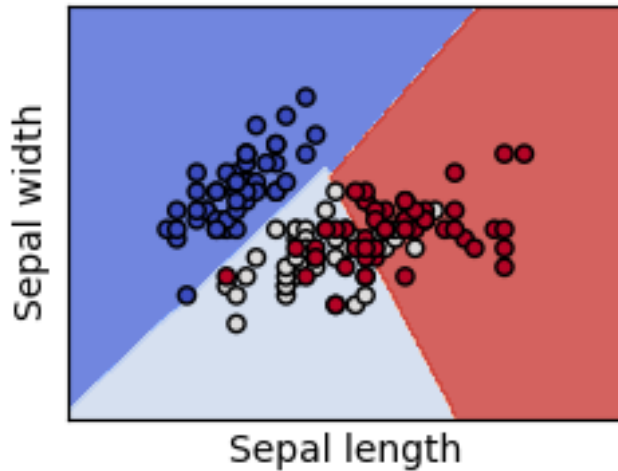


Four features (in centimeters)

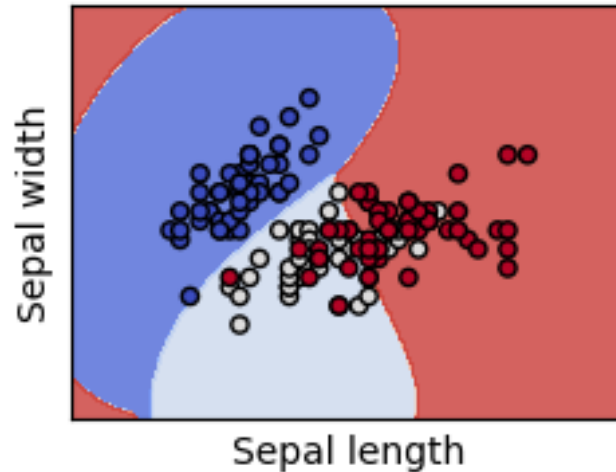
- Petal length / width
- Sepal length / width

# Kernel SVM in Scikit Learn

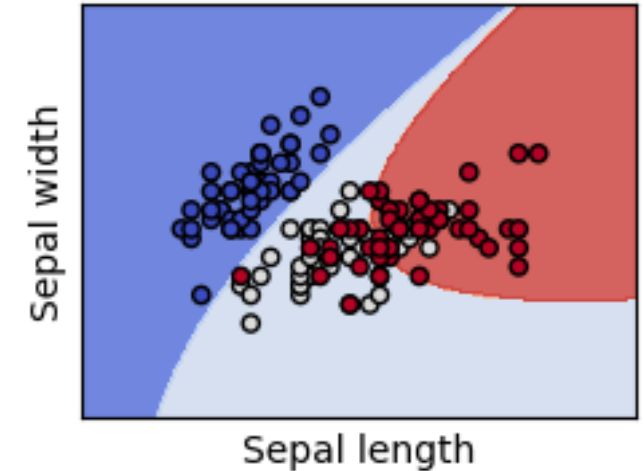
SVC with linear kernel



SVC with RBF kernel



SVC with polynomial (degree 3) kernel



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

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

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

**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.

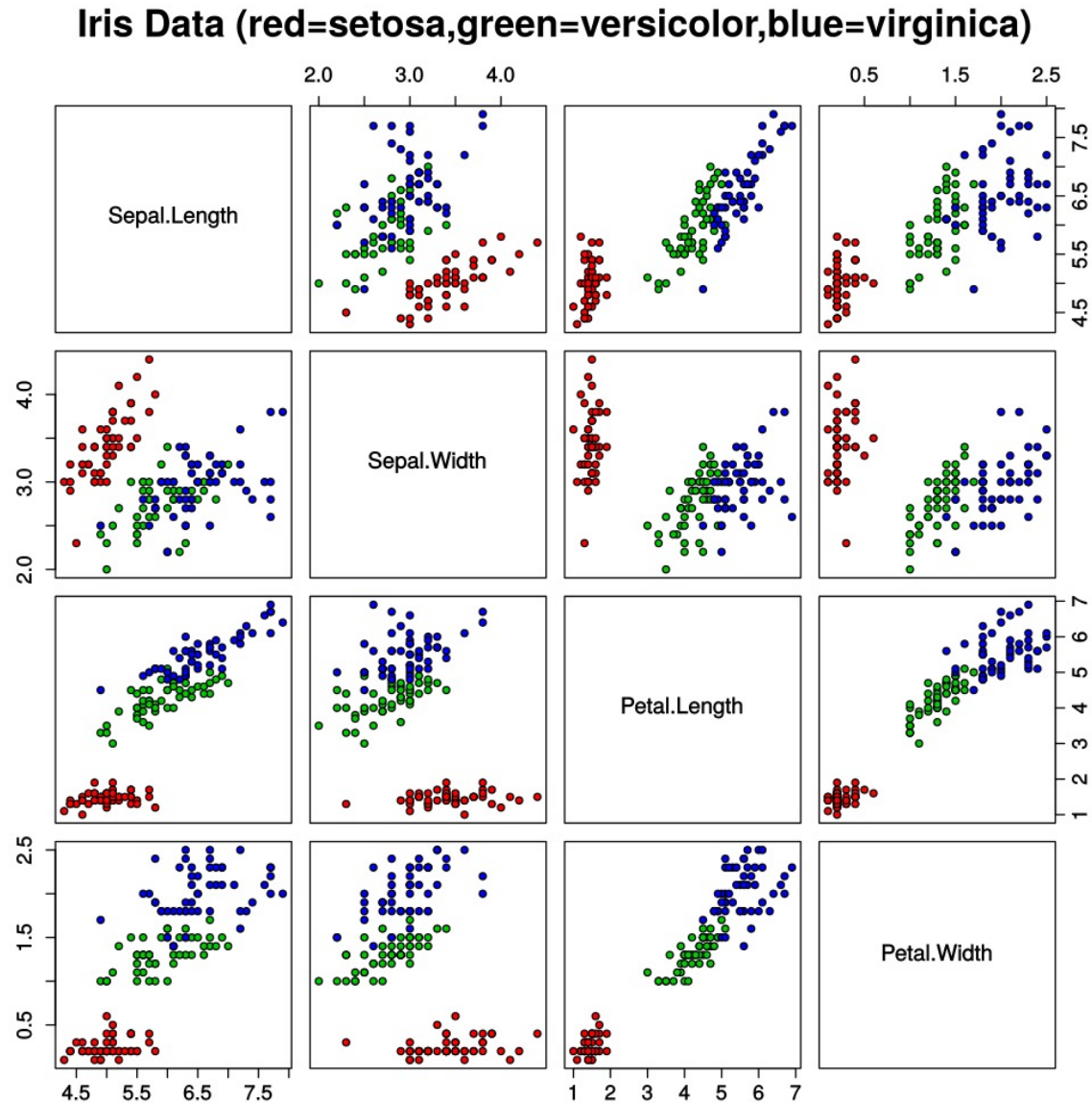
**verbose** : *bool, default=False*

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



*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      |

# Kernel Functions

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

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

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)

# Gaussian/RBF kernels

- $K(x, x') = \exp\left(-\frac{\|x-x'\|^2}{2\sigma^2}\right)$  (often parameterized as  $\exp(-\gamma\|x-x'\|^2)$ )
- How can we show that this is a valid kernel?

Assume  $x \in R^1$  and  $\gamma > 0$ .

$$\begin{aligned} e^{-\gamma\|x_i-x_j\|^2} &= e^{-\gamma(x_i-x_j)^2} = e^{-\gamma x_i^2 + 2\gamma x_i x_j - \gamma x_j^2} \\ &= e^{-\gamma x_i^2 - \gamma x_j^2} \left(1 + \frac{2\gamma x_i x_j}{1!} + \frac{(2\gamma x_i x_j)^2}{2!} + \frac{(2\gamma x_i x_j)^3}{3!} + \dots\right) \\ &= e^{-\gamma x_i^2 - \gamma x_j^2} \left(1 \cdot 1 + \sqrt{\frac{2\gamma}{1!}} x_i \cdot \sqrt{\frac{2\gamma}{1!}} x_j + \sqrt{\frac{(2\gamma)^2}{2!}} x_i^2 \cdot \sqrt{\frac{(2\gamma)^2}{2!}} x_j^2 \right. \\ &\quad \left. + \sqrt{\frac{(2\gamma)^3}{3!}} x_i^3 \cdot \sqrt{\frac{(2\gamma)^3}{3!}} x_j^3 + \dots\right) = \phi(x_i)^T \phi(x_j), \end{aligned}$$

where

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

=> We should find  $\phi(x)$  that results in  $K(x, x') = \langle \phi(x), \phi(x') \rangle$

recall how we make **predictions**:

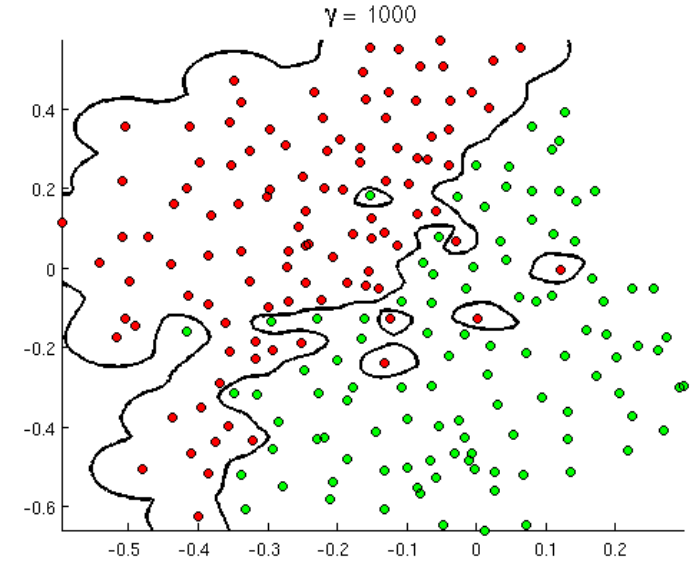
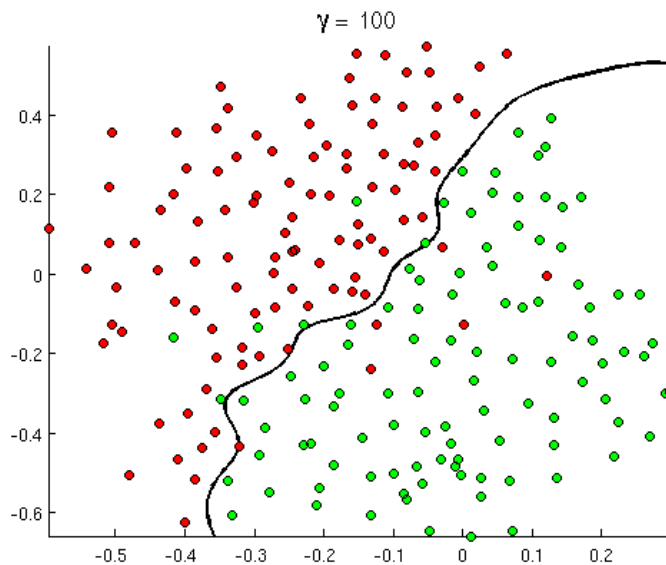
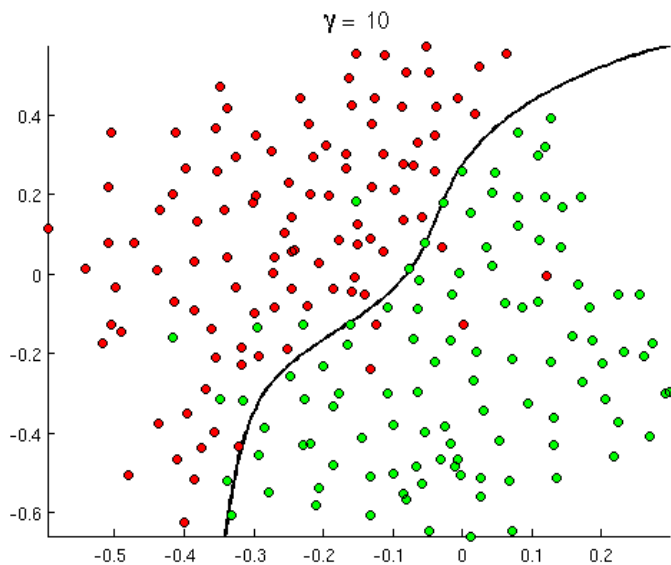
$$w^T x_* = \sum_i \alpha_i y_i x_i^T x_* = \sum_i \alpha_i y_i K(x_i, x_*)$$

**weighted k-NN**

- $\arg \max_y \sum_{i \in N(x_*)} w_i 1\{y_i = y\}$
- e.g.,  $w_i = \exp\left(-\beta \cdot (d(x_i, x_*))^2\right)$

# Gaussian kernel

- $\gamma = \frac{1}{2\sigma^2}$



- Larger  $\gamma \Rightarrow$  smaller  $\sigma^2 \Rightarrow$  more likely to overfit
- A heuristic in practice: choose  $\sigma = \text{median}(\|x_i - x_j\|, i \neq j)$

# 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  $n \times n$  **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:

$$k(\mathbf{x}, \mathbf{x}') = ck_1(\mathbf{x}, \mathbf{x}') \quad (6.13)$$

$$k(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}') \quad (6.14)$$

$$k(\mathbf{x}, \mathbf{x}') = q(k_1(\mathbf{x}, \mathbf{x}')) \quad (6.15)$$

$$k(\mathbf{x}, \mathbf{x}') = \exp(k_1(\mathbf{x}, \mathbf{x}')) \quad (6.16)$$

$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') \quad (6.17)$$

$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}') \quad (6.18)$$

$$k(\mathbf{x}, \mathbf{x}') = k_3(\phi(\mathbf{x}), \phi(\mathbf{x}')) \quad (6.19)$$

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{A} \mathbf{x}' \quad (6.20)$$

$$k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b) \quad (6.21)$$

$$k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a)k_b(\mathbf{x}_b, \mathbf{x}'_b) \quad (6.22)$$

# 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),

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

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

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

**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**

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

$$w = (\underbrace{\Phi^T \Phi + \lambda I}_{\text{MxM Matrix Inversion}})^{-1} \Phi^T y$$

**MxM Matrix Inversion**  
**O(M<sup>3</sup>)**

**Dual**

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

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

**NxN Matrix Inversion**  
**O(N<sup>3</sup>)**

*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)
```

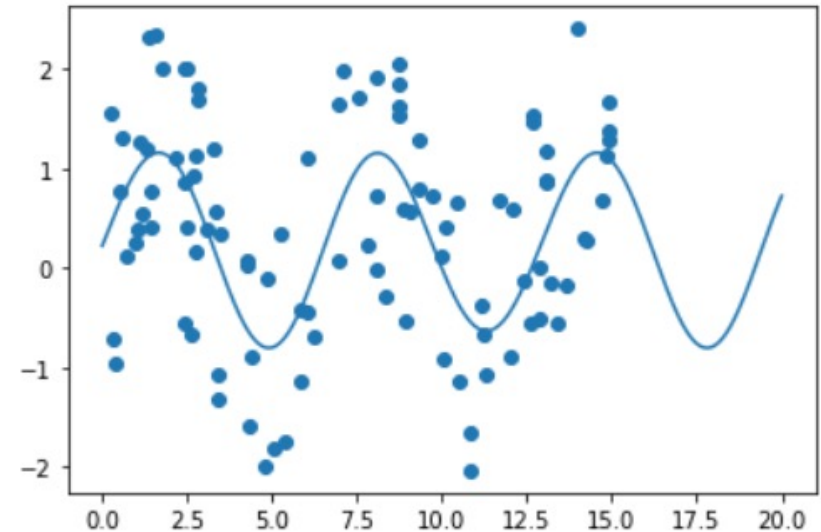
$$\exp\left(-\frac{2 \sin^2(\pi d(x_i, x_j)/p)}{l^2}\right)$$

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()
```



# Kernelized Perceptron algorithm

- How to combine the Perceptron algorithm with a nonlinear feature mapping  $\phi: \mathcal{X} \rightarrow \mathbb{R}^D$ ?
- Recall the Perceptron algorithm:

---

**Algorithm 29** PERCEPTRONTRAIN( $\mathbf{D}$ ,  $MaxIter$ )

---

```
1:  $w \leftarrow 0, b \leftarrow 0$  // initialize weights and bias
2: for  $iter = 1 \dots MaxIter$  do
3:   for all  $(x,y) \in \mathbf{D}$  do
4:      $a \leftarrow w \cdot \phi(x) + b$  // compute activation for this example
5:     if  $ya \leq 0$  then
6:        $w \leftarrow w + y \phi(x)$  // update weights
7:        $b \leftarrow b + y$  // update bias
8:     end if
9:   end for
10: end for
11: return  $w, b$ 
```

---

- Suppose  $\phi$  is associated with a kernel  $K$
- Is it possible to implement this without ever explicitly computing  $\phi$ ?

# Kernelized Perceptron algorithm

- Key observation: throughout the run of the Perceptron algorithm,  $w$  always lies in  $\text{span}(\phi(x_1), \dots, \phi(x_n))$ , i.e.

$w$  always has the form  $\alpha_1 \phi(x_1) + \dots + \alpha_n \phi(x_n)$

- Key algorithmic idea: instead of maintaining  $w \in \mathbb{R}^D$ , we maintain its linear combination coefficient  $(\alpha_1, \dots, \alpha_n) \in \mathbb{R}^n$ !

---

**Algorithm 30** KERNELIZEDPERCEPTRONTRAIN( $\mathbf{D}$ ,  $MaxIter$ )

---

```
1:  $\alpha \leftarrow 0, b \leftarrow 0$  // initialize coefficients and bias
2: for  $iter = 1 \dots MaxIter$  do
3:   for all  $(x_n, y_n) \in \mathbf{D}$  do
4:      $a \leftarrow \sum_m \alpha_m \phi(x_m) \cdot \phi(x_n) + b$  // compute activation for this example
5:     if  $y_n a \leq 0$  then
6:        $\alpha_n \leftarrow \alpha_n + y_n$  // update coefficients
7:        $b \leftarrow b + y$  // update bias
8:     end if
9:   end for
10: end for
11: return  $\alpha, b$ 
```

---



# Outline

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

# Basis Functions

Basis functions transform linear models into nonlinear ones...

**Linear Regression**

$$y = w^T x$$



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

**Classification  
( Logistic Regression )**

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



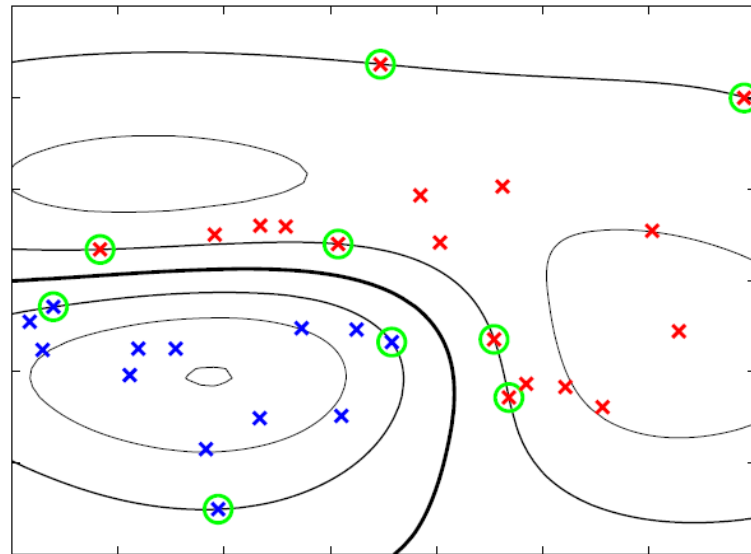
$$y = \sigma(w^T \phi(x))$$

...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**

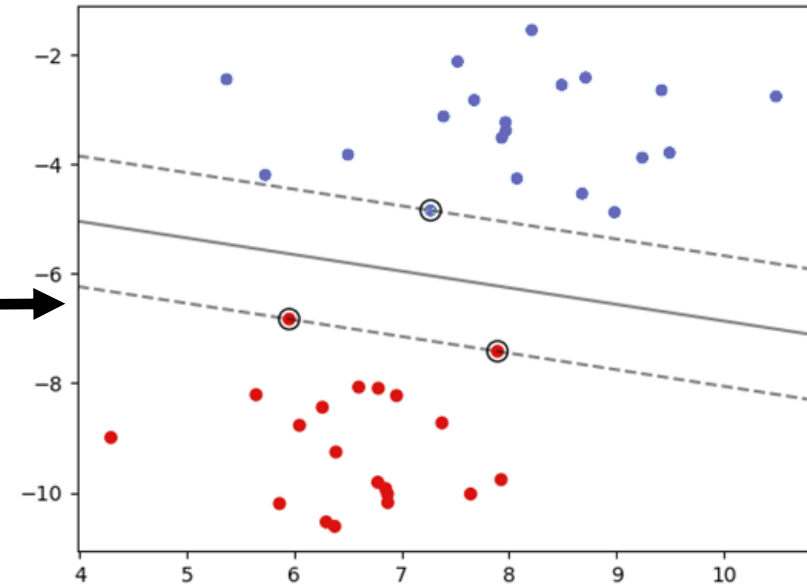


Ignore the circled points...I reused these from the SVM slides

**Neural Net**  
 $\phi(x)$



**Warped Space**



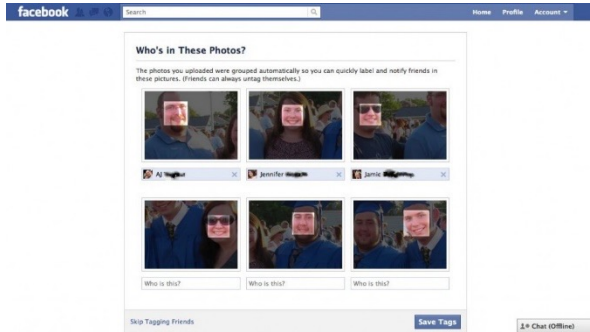
...this is essentially what standard neural networks do...

# 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...



**FB Auto Tagging**

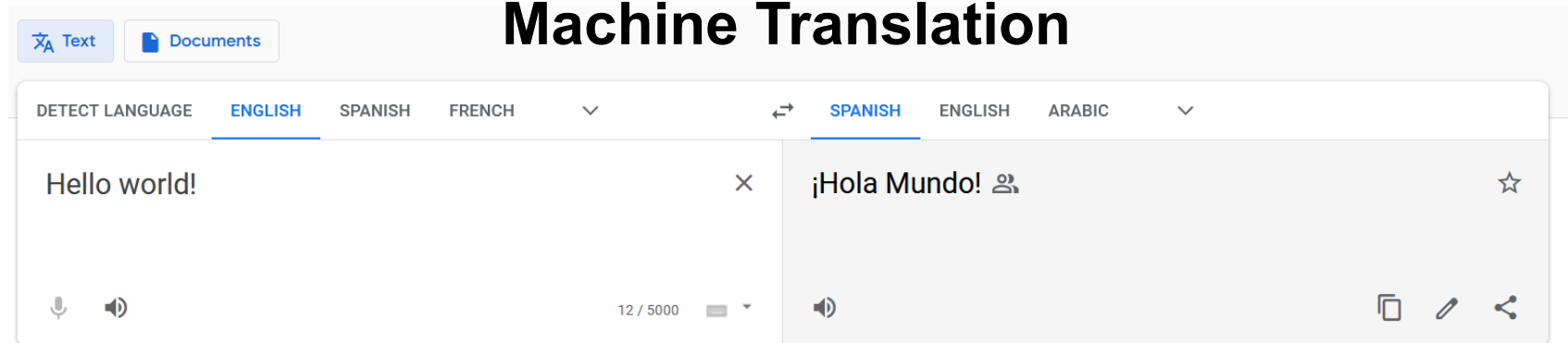


**Self-Driving Cars**



**Creepy Robots**

## Machine Translation

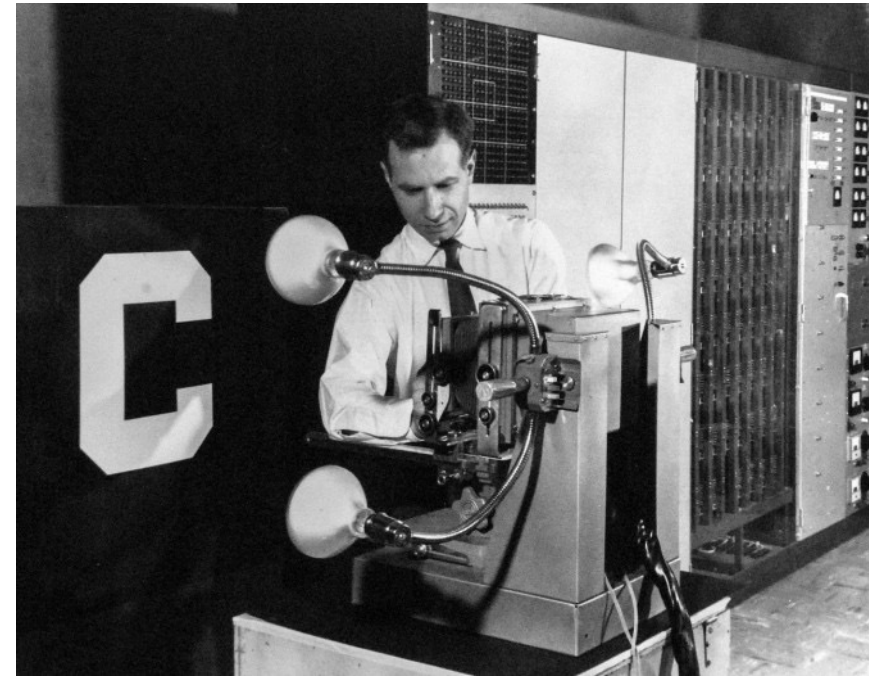


[Send feedback](#)

# Rosenblatt's Perceptron

Despite recent attention, neural networks are fairly old

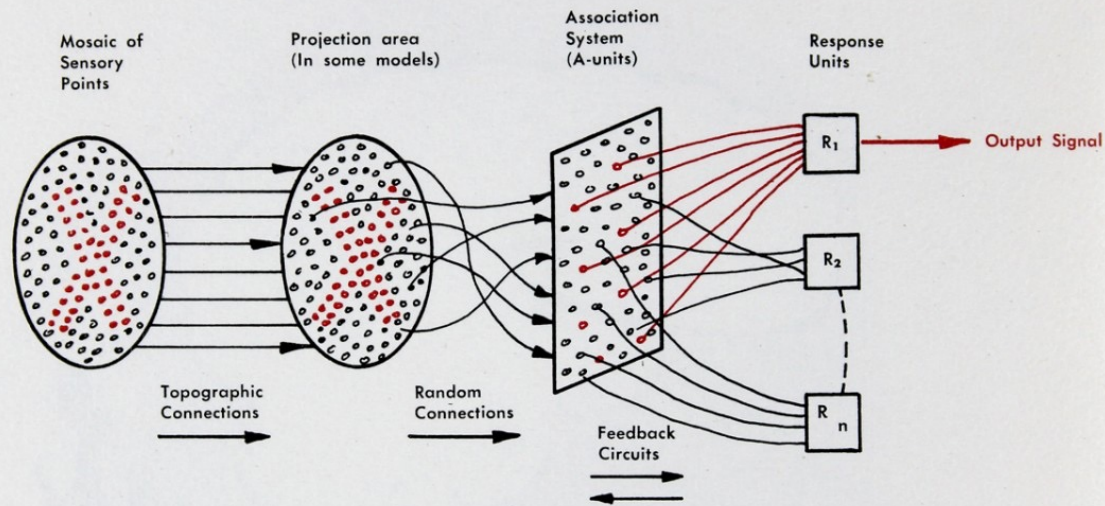
In 1957 Frank Rosenblatt constructed the first (single layer) neural network known as a "perceptron"



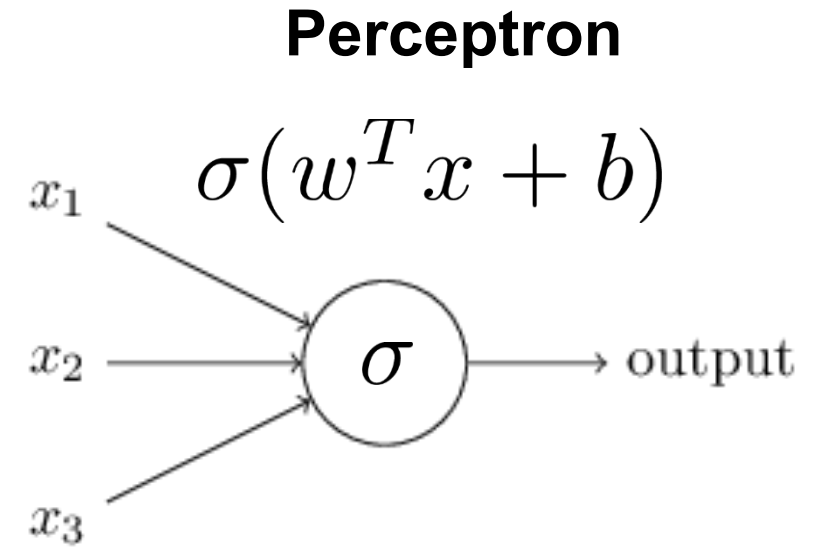
He demonstrated that it is capable of recognizing characters projected onto a 20x20 "pixel" array of photosensors

# Rosenblatt's Perceptron

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

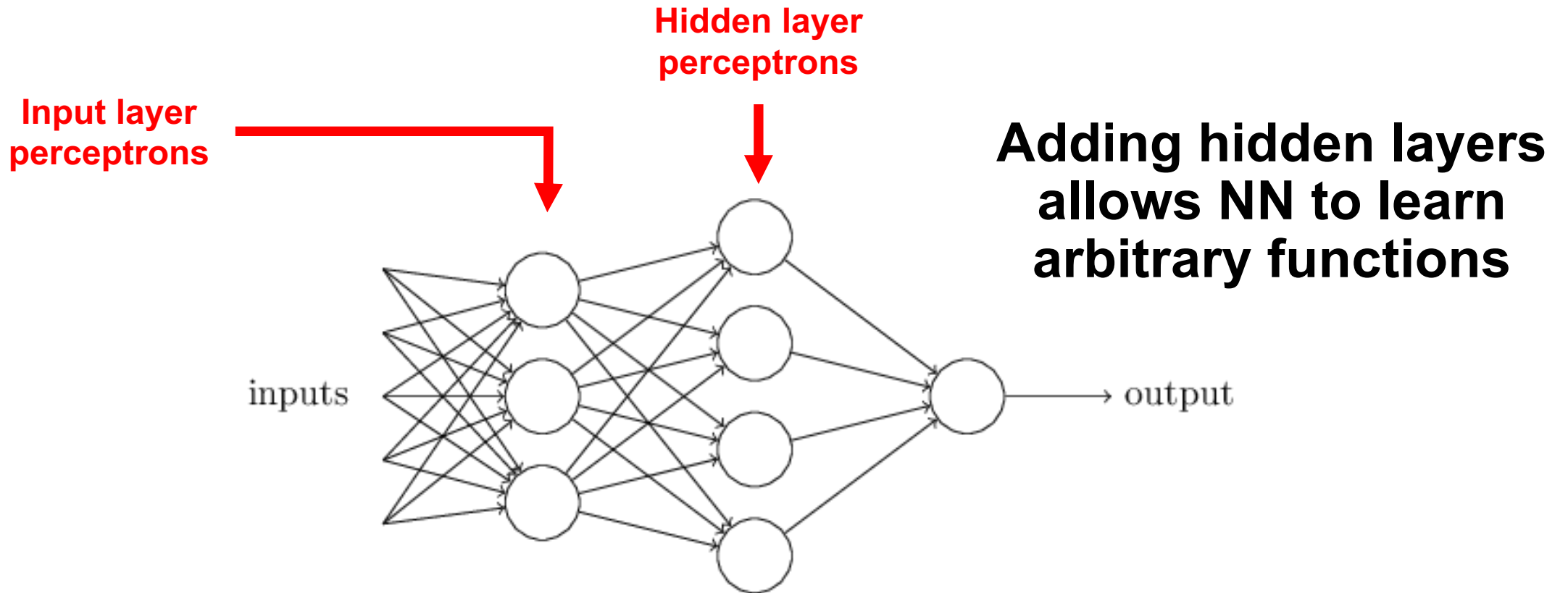


**FIG. 2** — Organization of a perceptron.



- 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



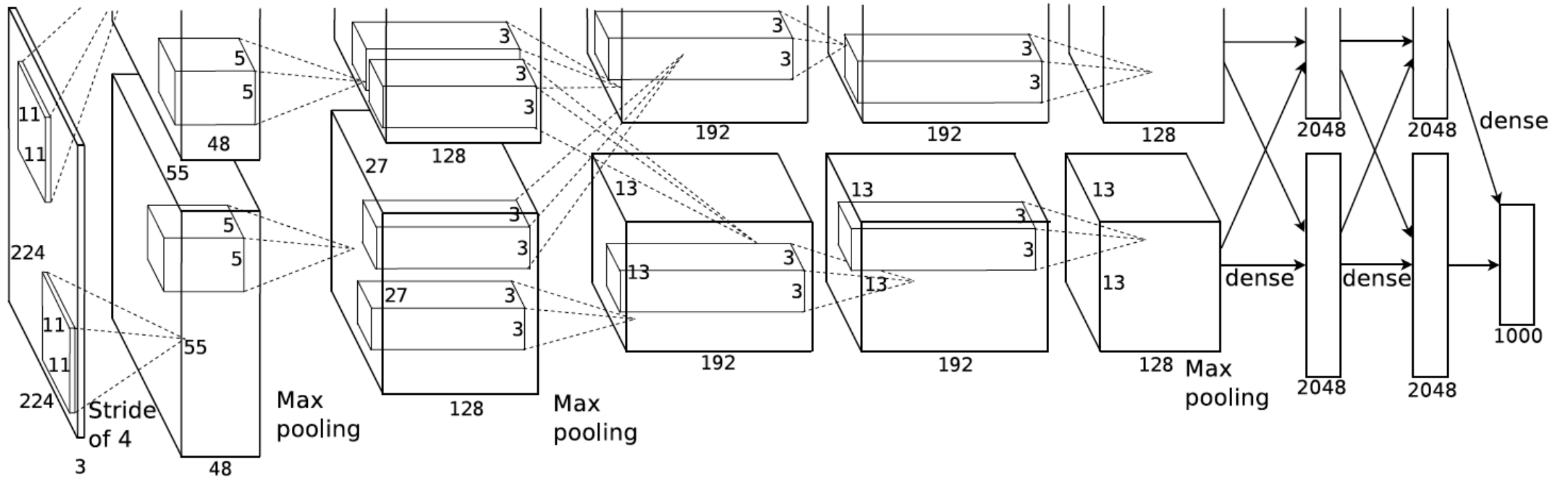
This is the quintessential *Neural Network...*  
...also called *Feed Forward Neural Net* or *Artificial Neural Net*

[ Source: <http://neuralnetworksanddeeplearning.com> ]



# Modern Neural Networks

Modern *Deep Neural networks* add many hidden layers



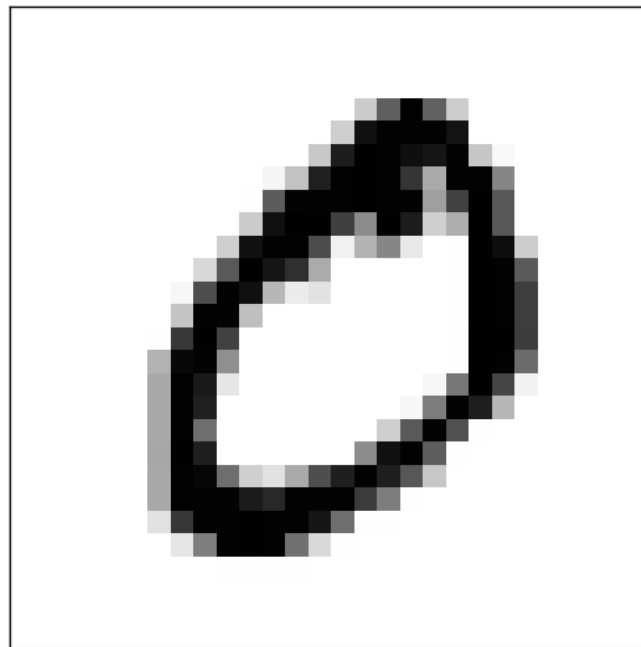
...and have many millions of parameters to learn

# Handwritten Digit Classification

Classifying handwritten digits is the “Hello World” of NNs



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

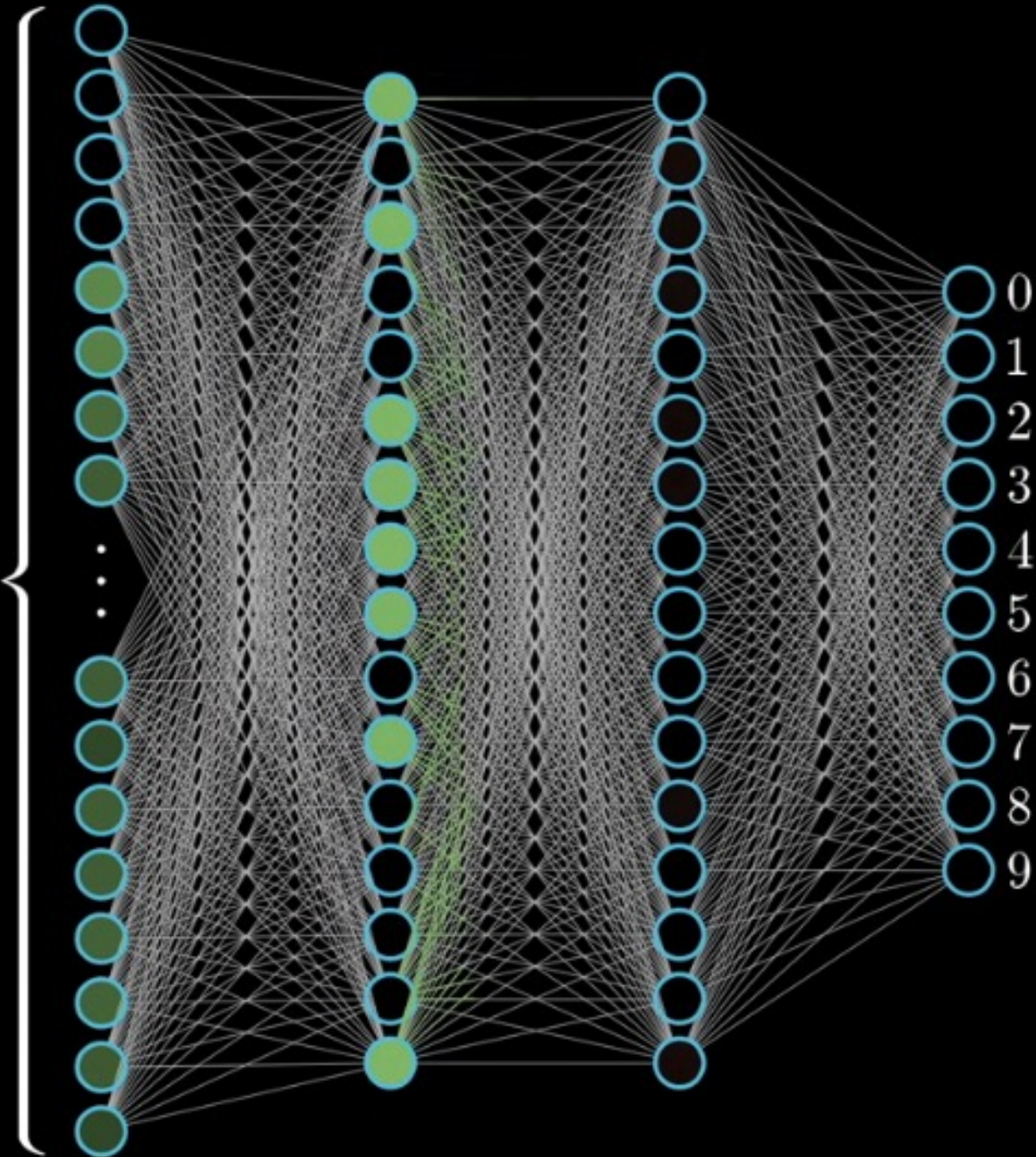


Modified National Institute of  
Standards and Technology  
(MNIST) database contains 60k  
training and 10k test images

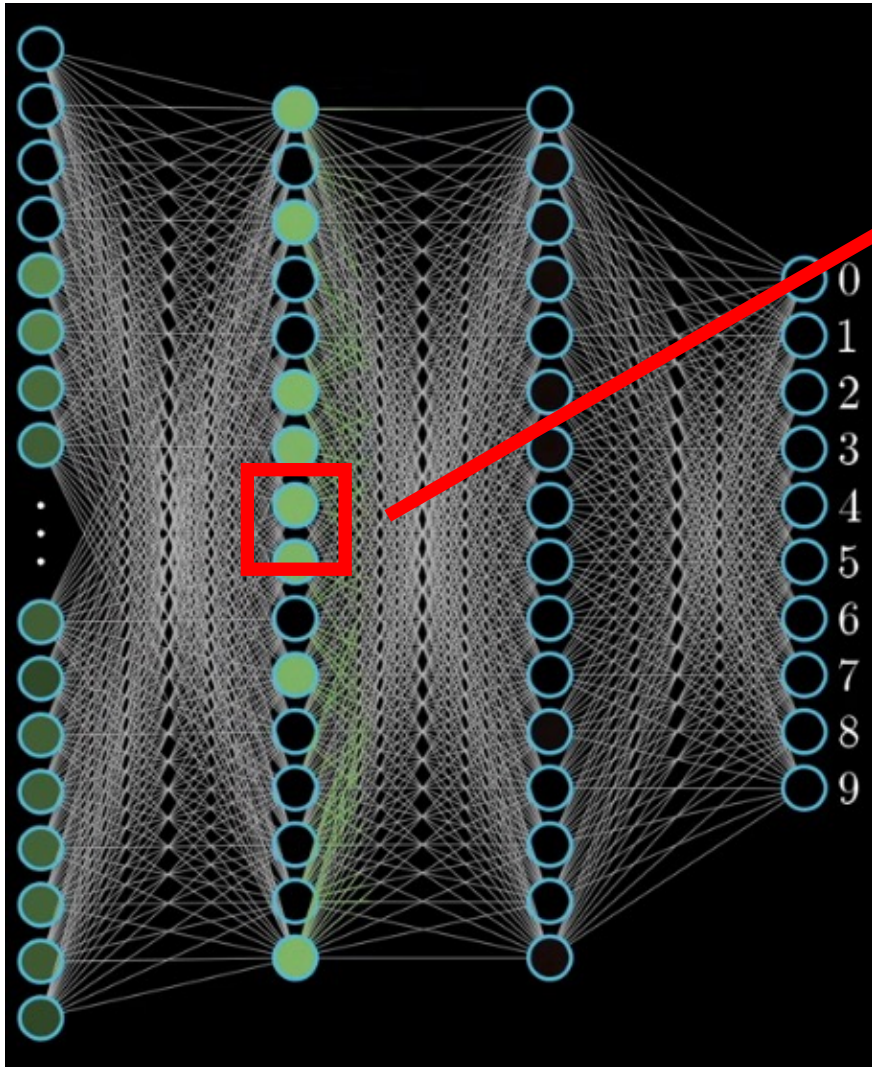


784

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



# Feedforward Procedure



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

$$w_1x_1 + w_2x_2 + \dots + w_nx_n$$

Then applies a *nonlinear function* to the result

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

Often, we also introduce a constant *bias* parameter

# Nonlinear Activation functions

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

$$\sigma(w_1x_1 + w_2x_2 + \dots + w_nx_n + b) = \sigma(w^T x + b)$$

An early choice was the *logistic function*,

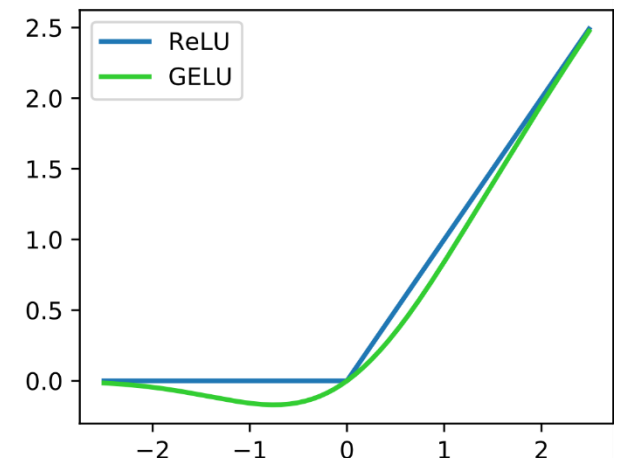
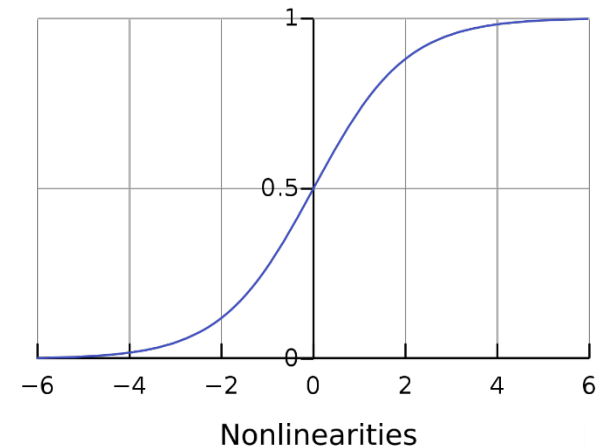
$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

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

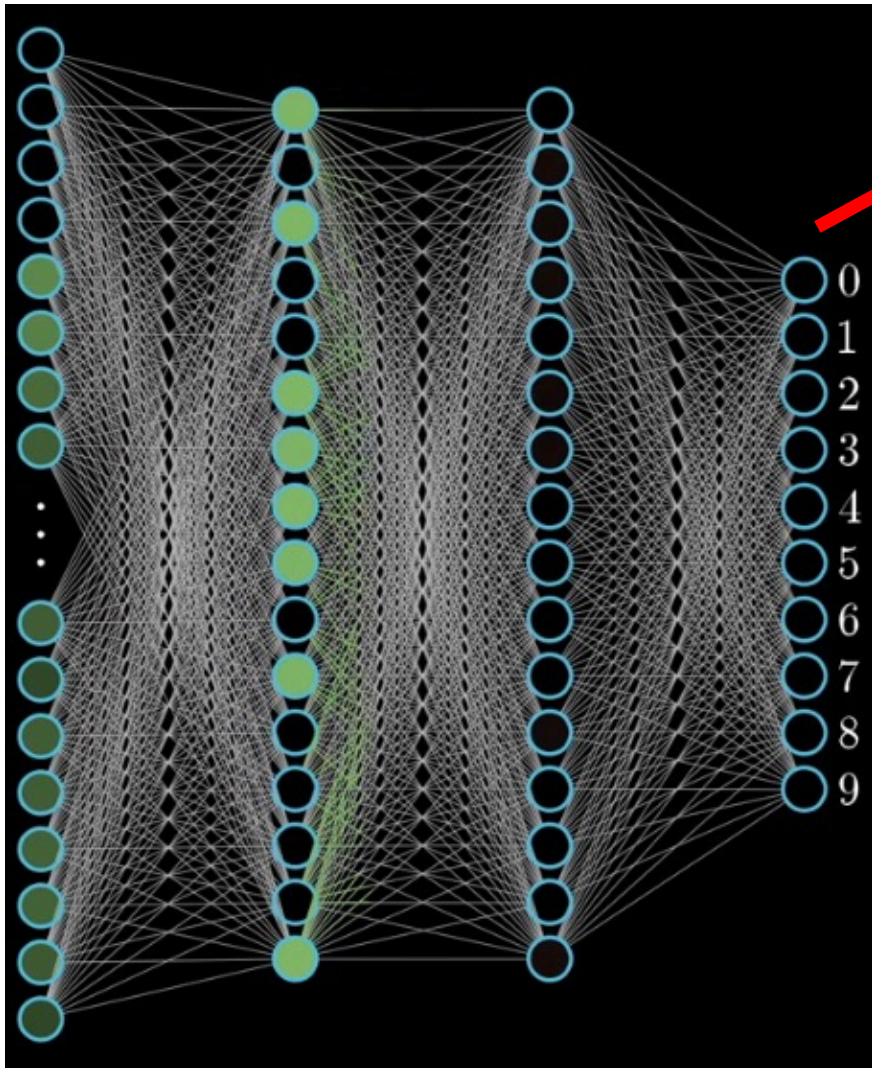
$$\sigma(w^T x + b) = \max(0, w^T x + b)$$

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

$$v = w^T x + b \quad \sigma(v) = v\Phi(v) \quad \leftarrow \text{Gaussian CDF}$$



# Multilayer Perceptron



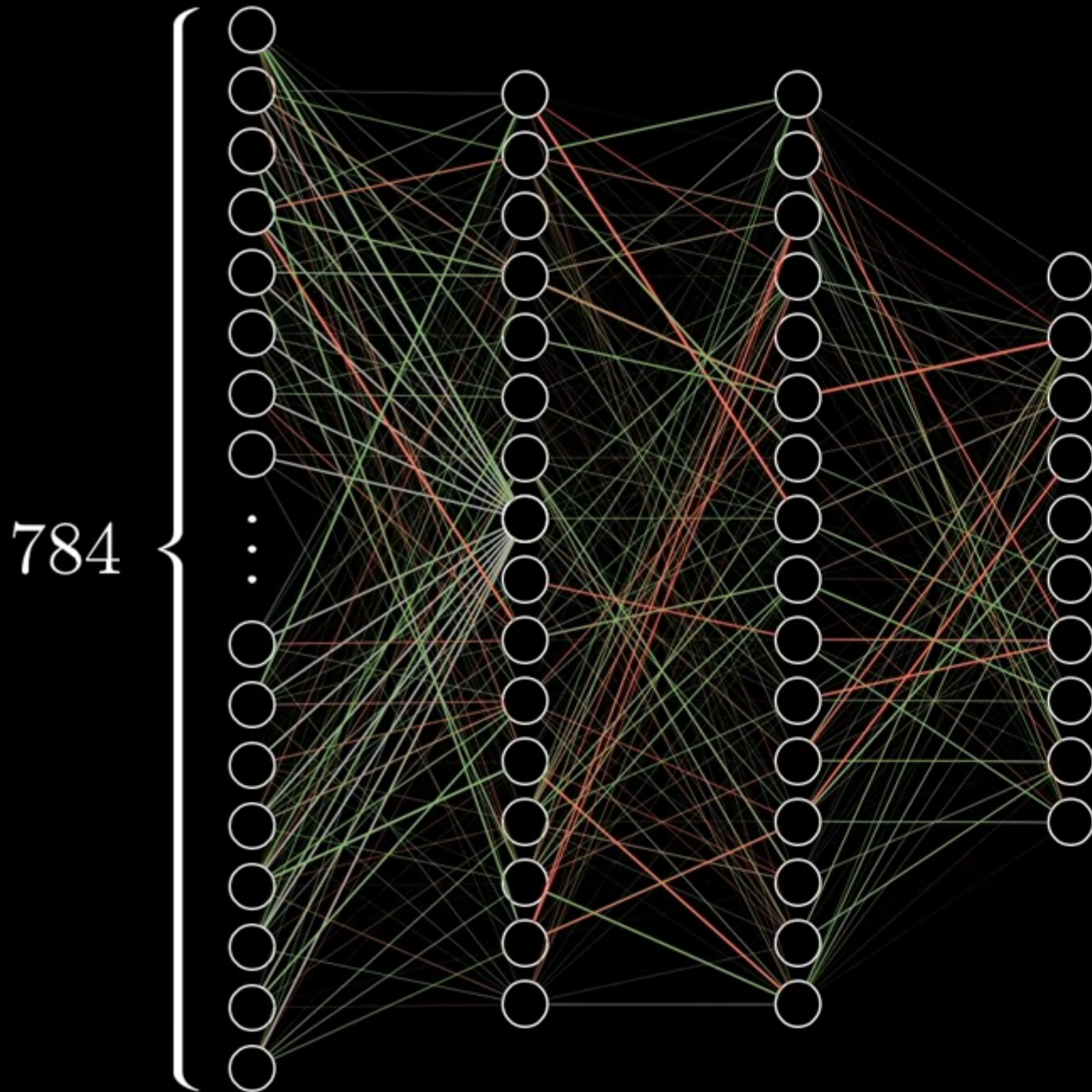
Final layer is typically a linear model...for classification this is a Logistic Regression

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

**Vector of activations from previous layer**

Recall that for multiclass logistic regression with K classes,

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



$$784 \times 16 + 16 \times 16 + 16 \times 10$$

weights

$$16 + 16 + 10$$

biases

**13,002**

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

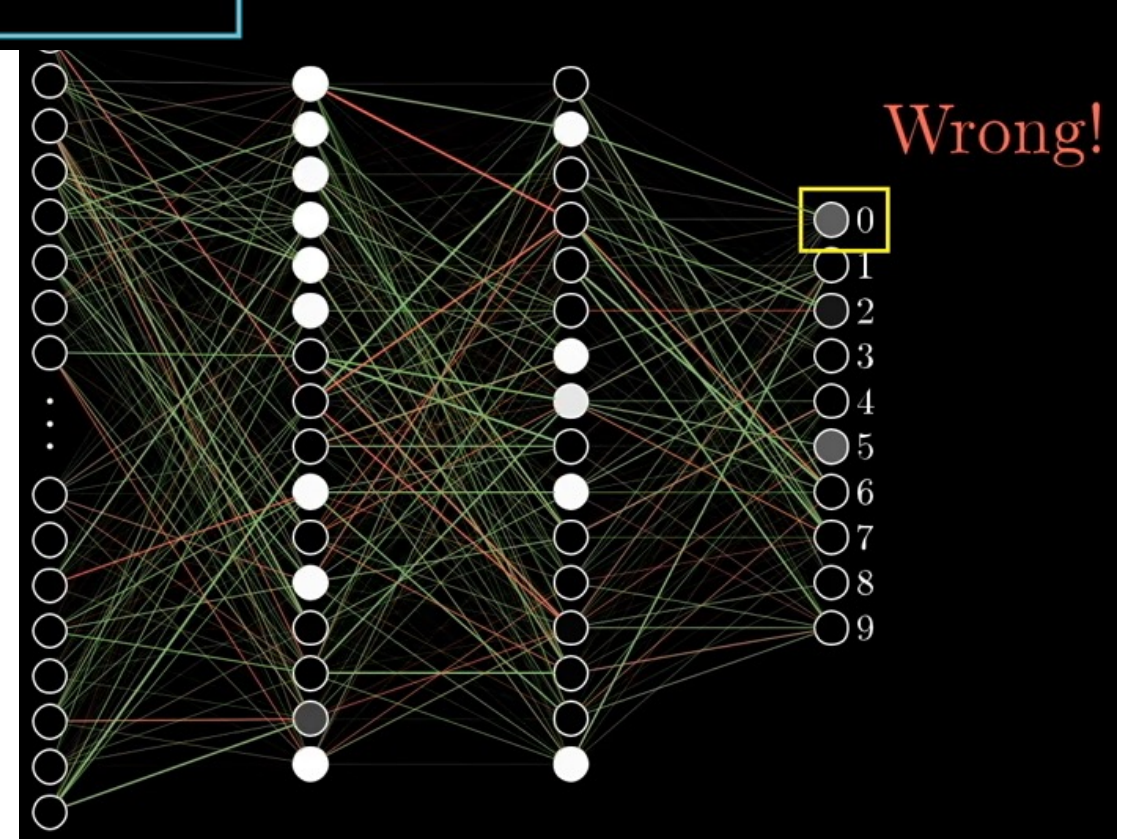
# Training Multilayer Perceptron

$$X^{\text{Train}} = \begin{matrix} \begin{matrix} 0 & 4 & 1 & 9 & 2 & 1 & 3 & 1 & 4 & 3 \\ 5 & 3 & 6 & 1 & 7 & 2 & 8 & 6 & 9 & 4 \\ 0 & 9 & 1 & 1 & 2 & 4 & 3 & 2 & 7 & 3 \\ 8 & 6 & 9 & 0 & 5 & 6 & 0 & 7 & 6 & 1 \\ 8 & 7 & 9 & 3 & 9 & 8 & 5 & 9 & 3 & 3 \\ 0 & 7 & 4 & 9 & 8 & 0 & 9 & 4 & 1 & 4 \\ 4 & 6 & 0 & 4 & 5 & 6 & 1 & 0 & 0 & 1 \\ 7 & 1 & 6 & 3 & 0 & 2 & 1 & 1 & 7 & 9 \\ 0 & 2 & 6 & 7 & 8 & 3 & 9 & 0 & 4 & 6 \\ 7 & 4 & 6 & 8 & 0 & 7 & 8 & 3 & 1 & 5 \end{matrix} \end{matrix}$$

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



For each training example, predict label and adjust weights...

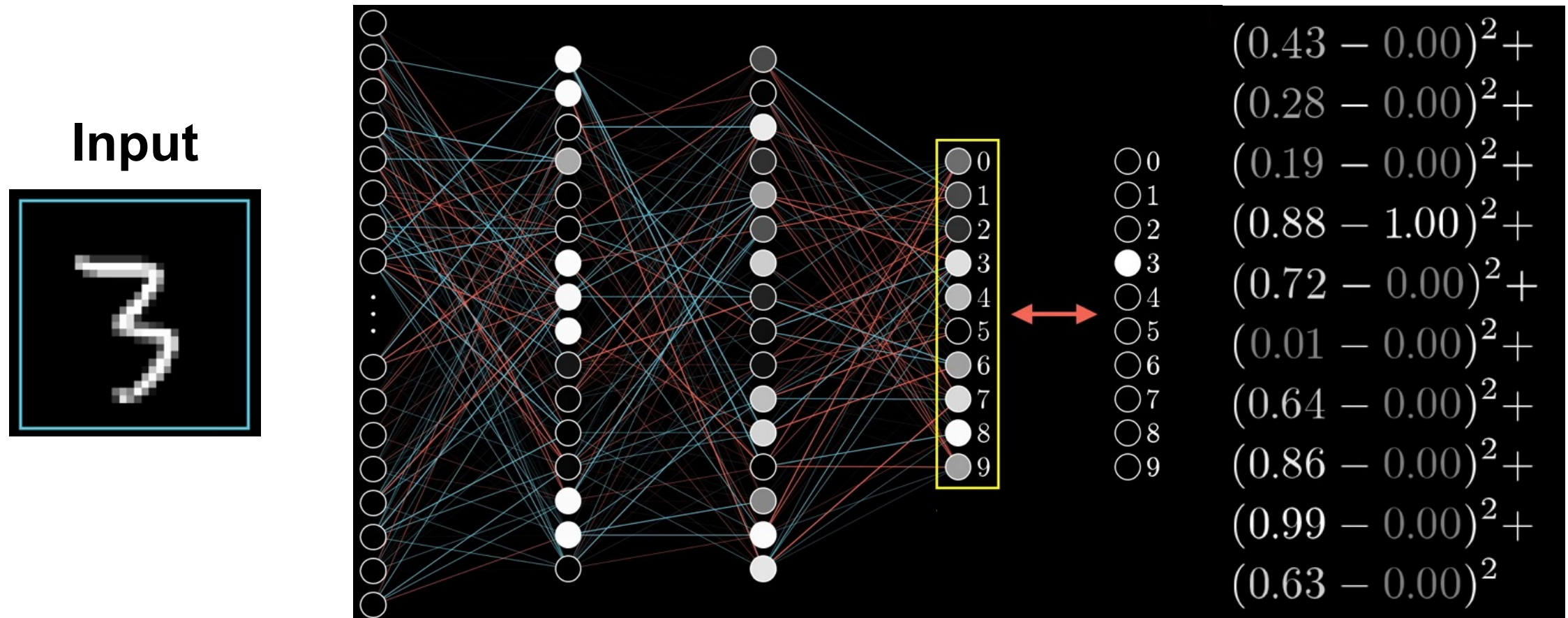


- How to score final layer output?
- How to adjust weights?




# Training Multilayer Perceptron

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



# Training Multilayer Perceptron

Our cost function for  $i^{\text{th}}$  input is error in terms of weights / biases...

$$\text{Cost}_i(w_1, \dots, w_n, b_1, \dots, b_n)$$


**13,002 Parameters  
in this network**

...minimize cost over all training data...

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

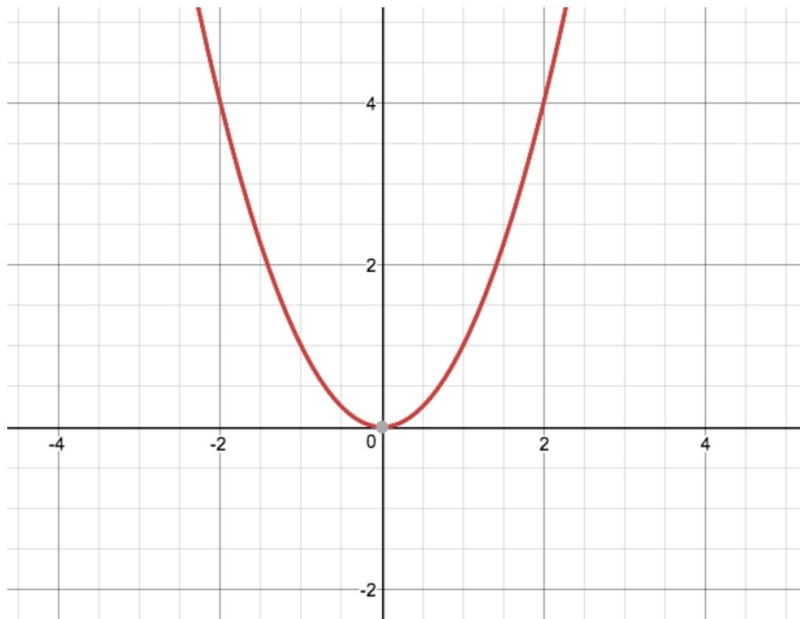
This is a super high-dimensional optimization (13,002 dimensions in this example)...how do we solve it?

**Gradient descent!**

# Training Multilayer Perceptron

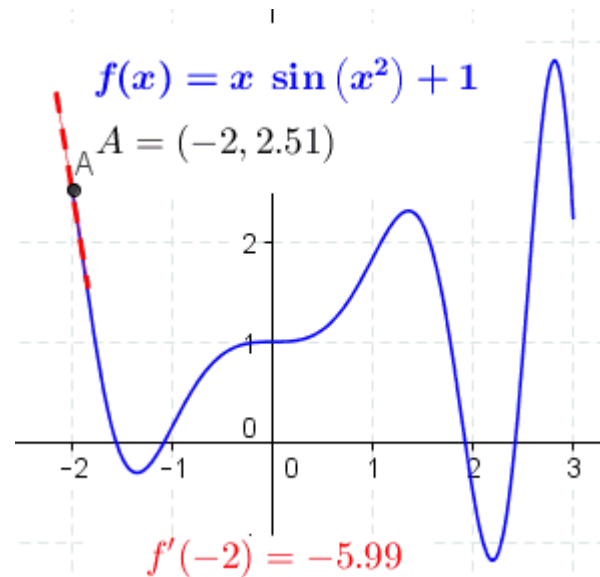
Need to find zero derivative (gradient) solution...

Convex Cost Function



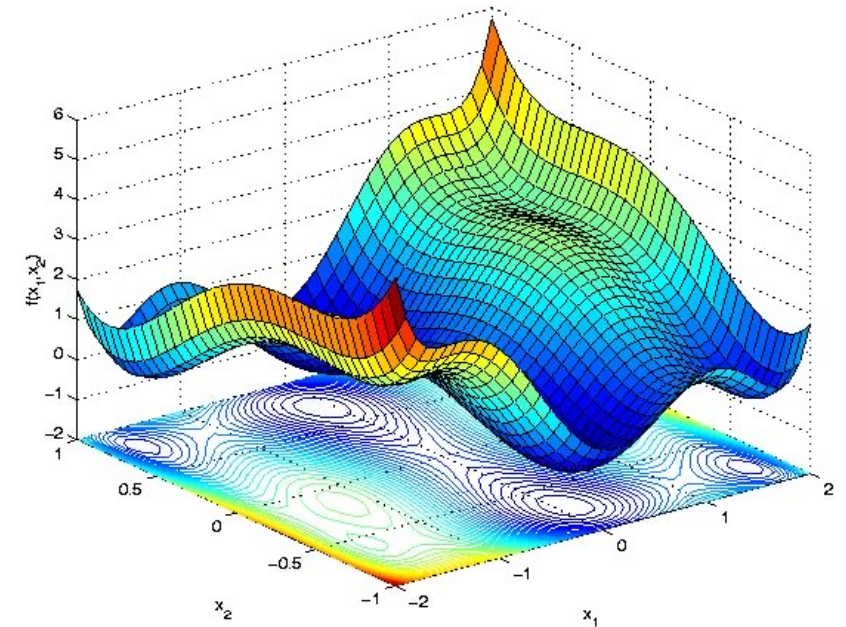
**YAY!**

Non-convex Cost Function



**Boo!**

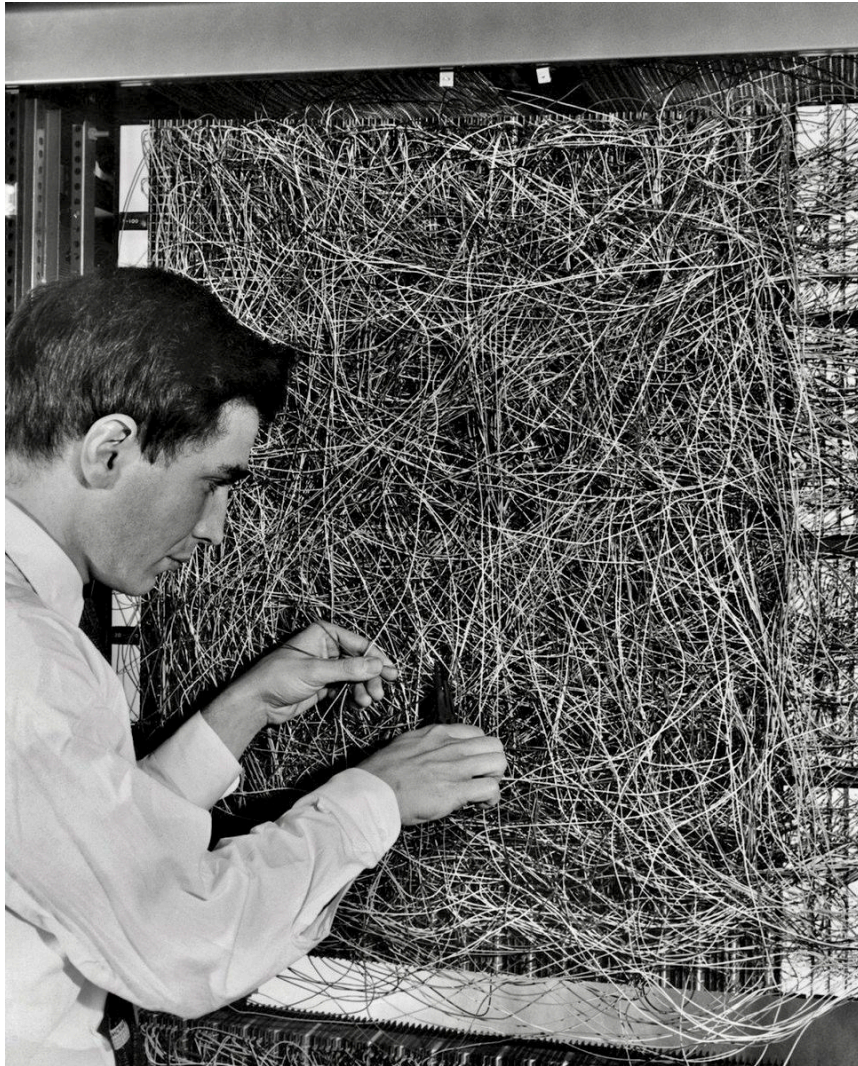
High-Dimensional Non-convex



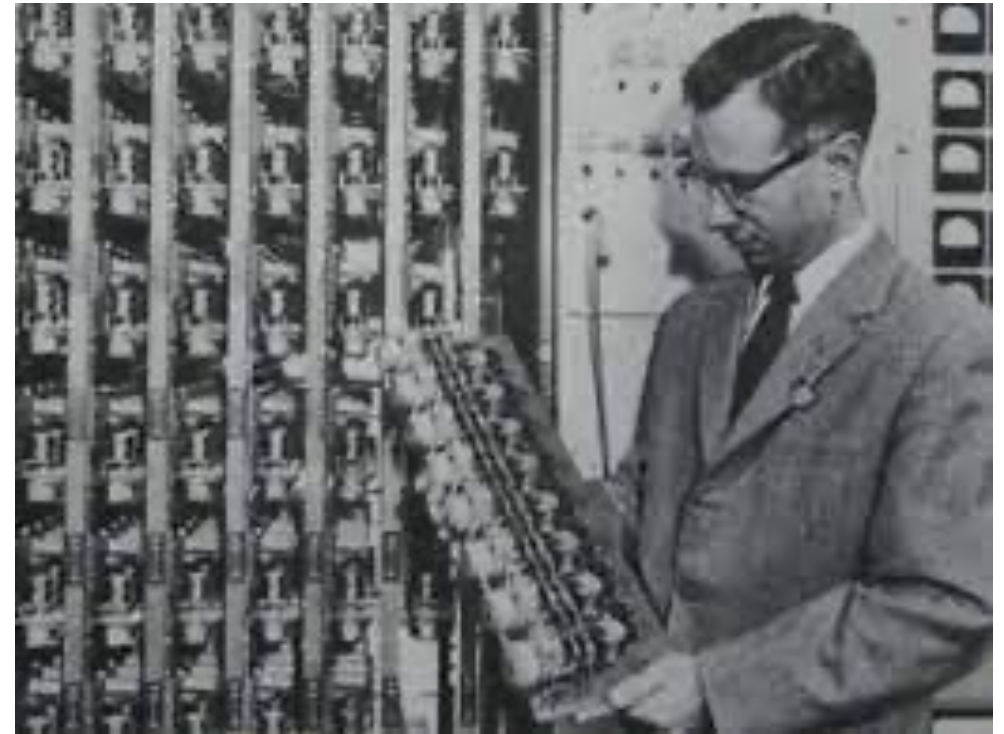
**Super Boo!**

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**

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

Derivative of  $f$  at its argument  $g(w)$   
e.g. treat  $g(w)$  as a variable

Differentiate  $g$  with respect to  $w$

# 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,

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

$$f(x) = \frac{1}{x}$$

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

$$g(z) = 1 + e^{-z}$$

$$g'(z) = -e^{-z}$$

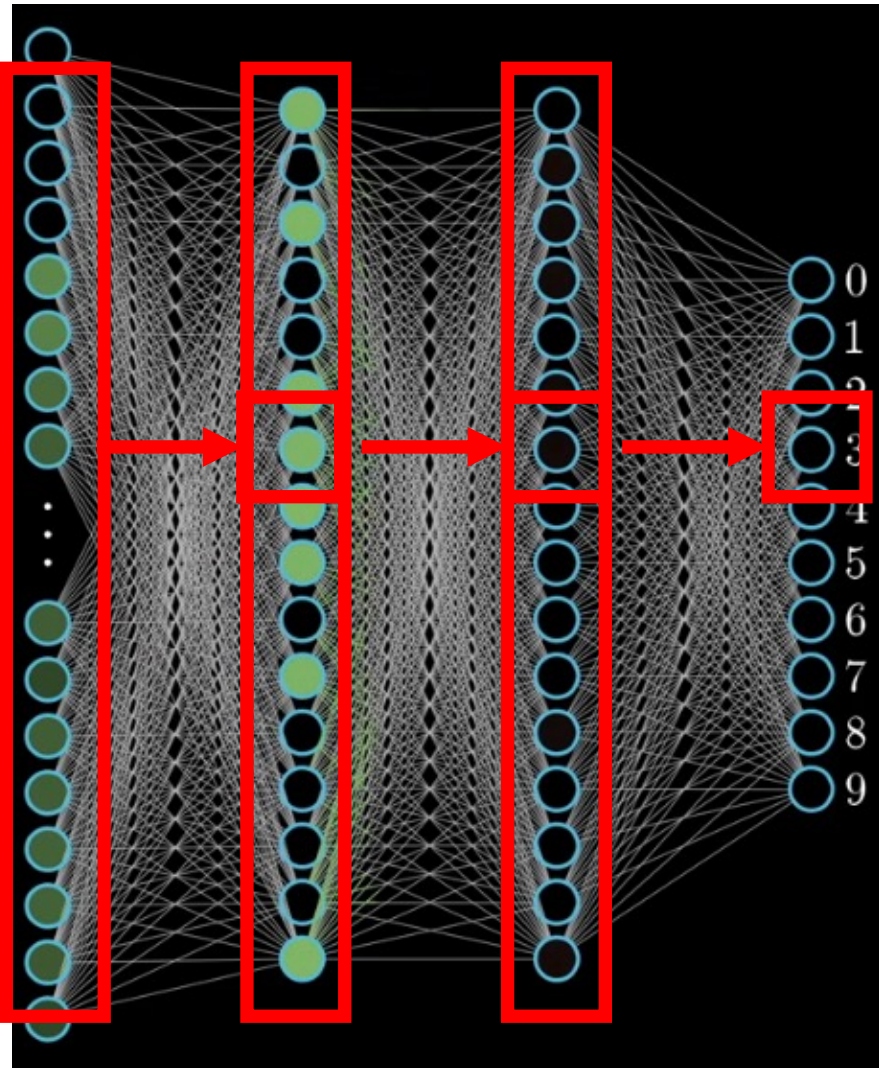
$$\sigma'(z) = f'(g(z))g'(z)$$

$$= \frac{e^{-z}}{(1 + e^{-z})^2}$$

$$= \sigma(z)(1 - \sigma(z))$$

# 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...


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



# 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)

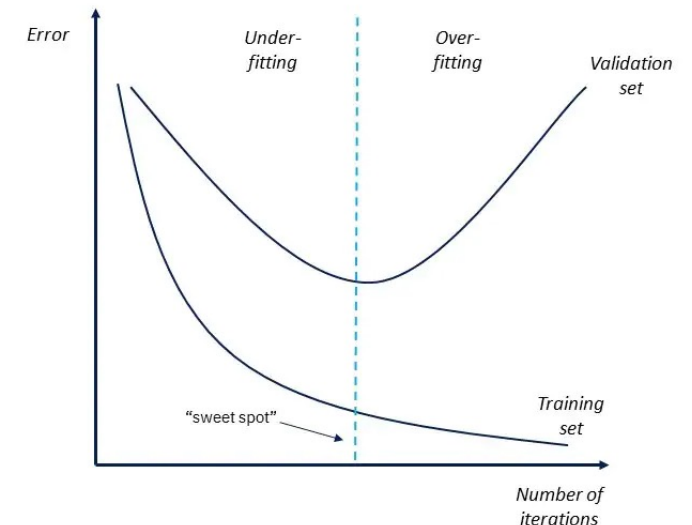
# Regularization

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

$$w = \arg \min_w \text{Cost}(w) + \alpha \cdot \text{Regularizer}(\text{Model})$$

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

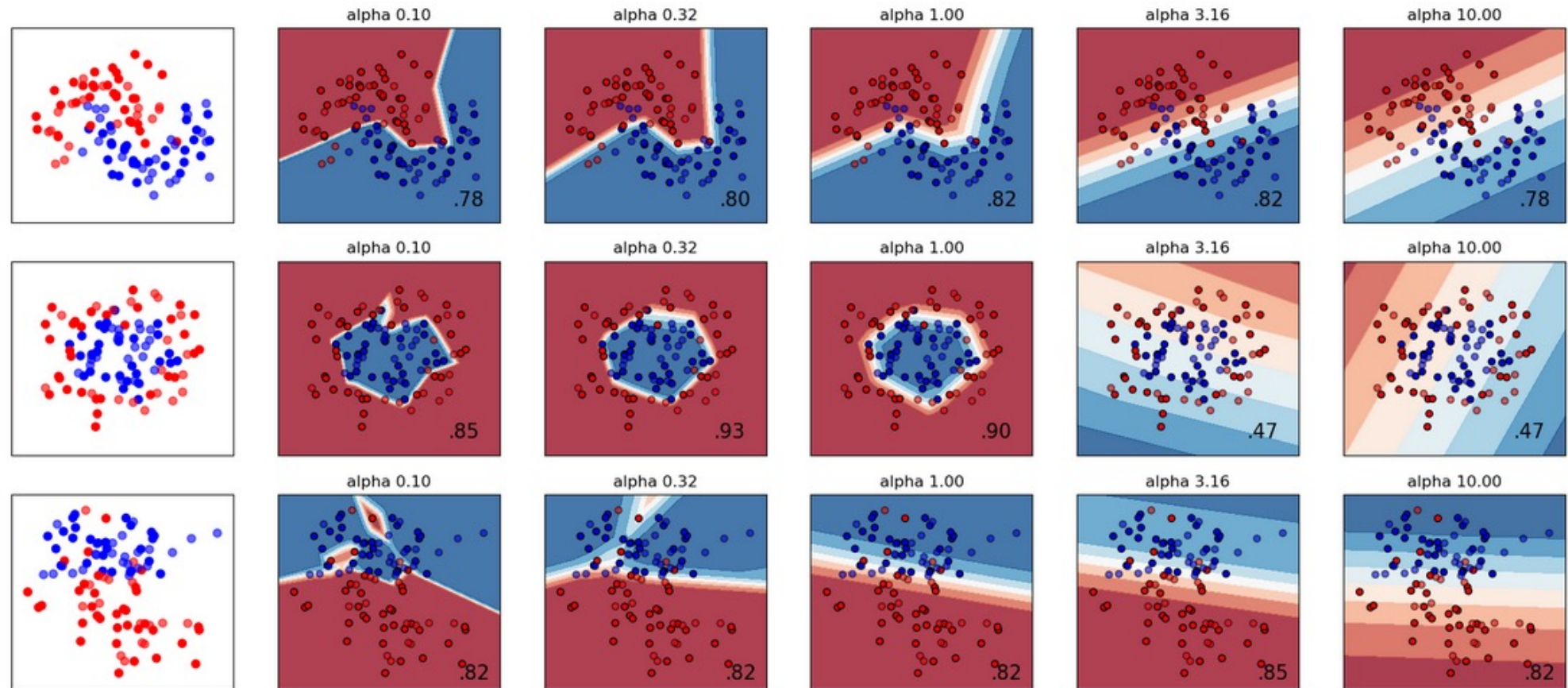
...numerous regularization schemes are used in training neural networks



# 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 *i*th element represents the number of neurons in the *i*th 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](http://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...

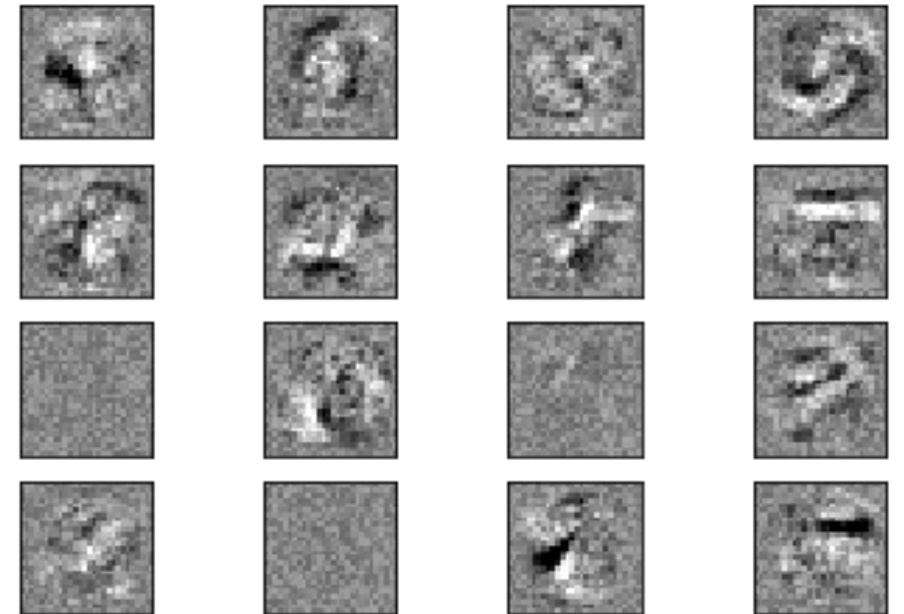
```
mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
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