Supervised Learning – Part 6

ESM3081 Programming for Data Science

Seokho Kang

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

Learning algorithms covered in this course

- **Supervised Learning** (Classification/Regression)
  - K-Nearest Neighbors
  - Linear Models (Logistic/Linear Regression)
  - Decision Trees
  - Random Forests
  - Gradient Boosting Machines
  - Support Vector Machines
  - **Neural Networks**

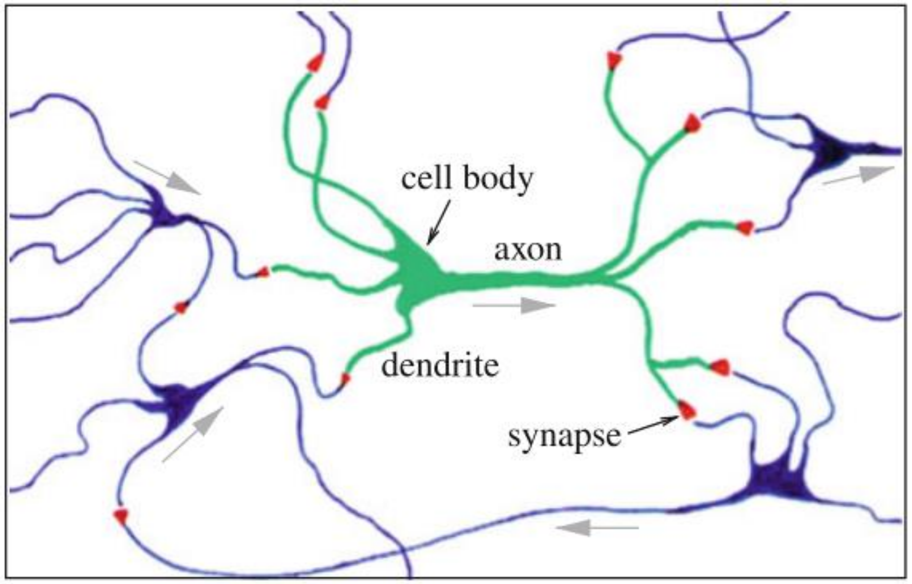
** Many algorithms have a classification and a regression variant, and we will describe both.*

** We will review the most popular machine learning algorithms, explain how they learn from data and how they make predictions, and examine the strengths and weaknesses of each algorithm.*

Neural Networks

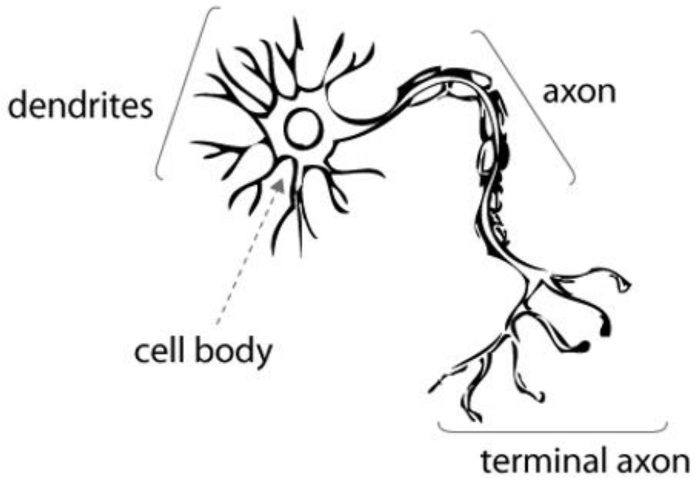
Neural Networks

- **Biological Neural Network** – Network of neurons in the brains of humans and animals
  - The human brain has about 100 billion neurons.
  - For many centuries, biologists, psychologists, and doctors have tried to understand how the brain functions.
  - The neurons and their connections are responsible for awareness, associations, thoughts, consciousness, and the ability to learn.

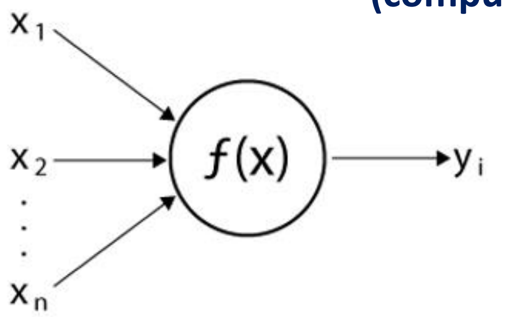


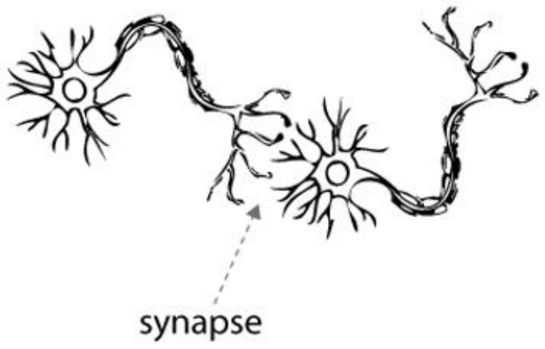
Neural Networks

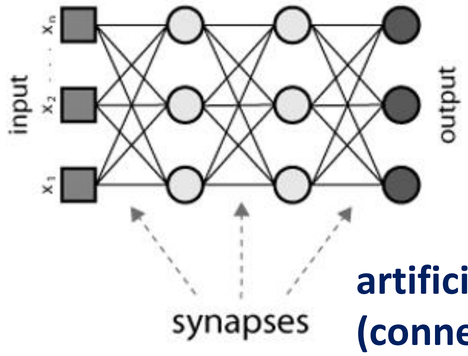
- Brain's architecture (Biological Neural Network) for inspiration on how to build an intelligent machine. → **Artificial Neural Network**



artificial neuron (computing unit)







artificial synapse (connection between two units)

Neural Networks

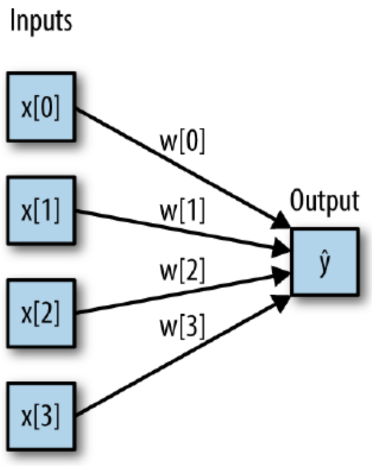
- A family of algorithms known as **neural networks** has recently seen a revival under the name “**deep learning**.”

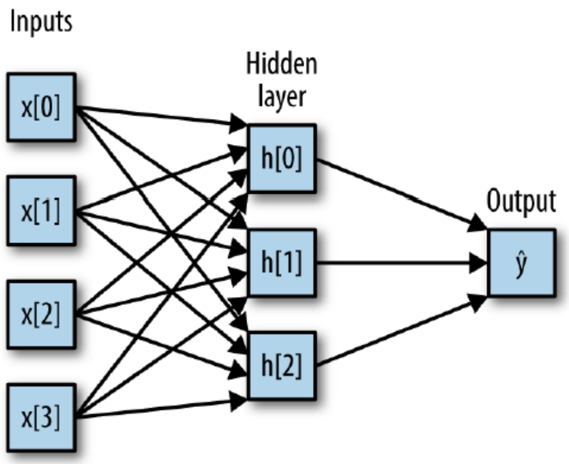
- Here, we will only discuss some relatively simple methods, namely ***multilayer perceptrons*** (MLPs, *a.k.a.*, feed-forward neural networks) for classification and regression, that can serve as a starting point for more involved deep learning methods.

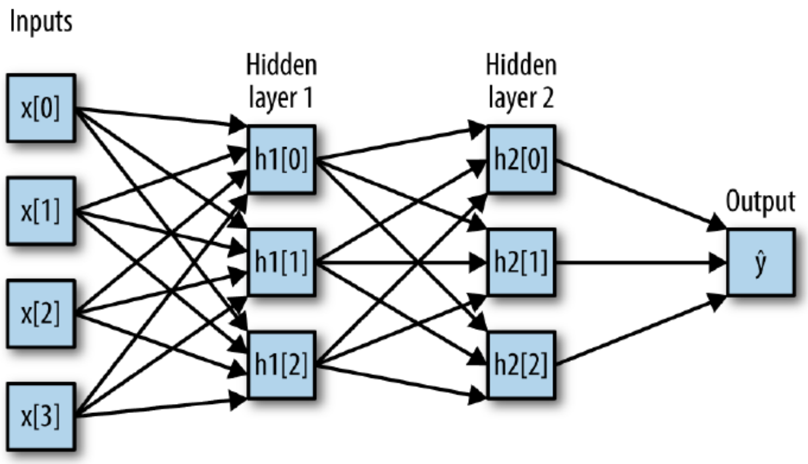
Multi-layer Perceptrons

- **Multi-layer Perceptrons**
  - General structure – input layer, hidden layers (0 to many), and output layer
  - MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision.
  - Multiple layers $f^{(1)}, f^{(2)}, \dots, f^{(l)}$ are connected in a chain to form

$$f(\boldsymbol{x}) = f^{(l)}\left(\dots\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)\right)$$

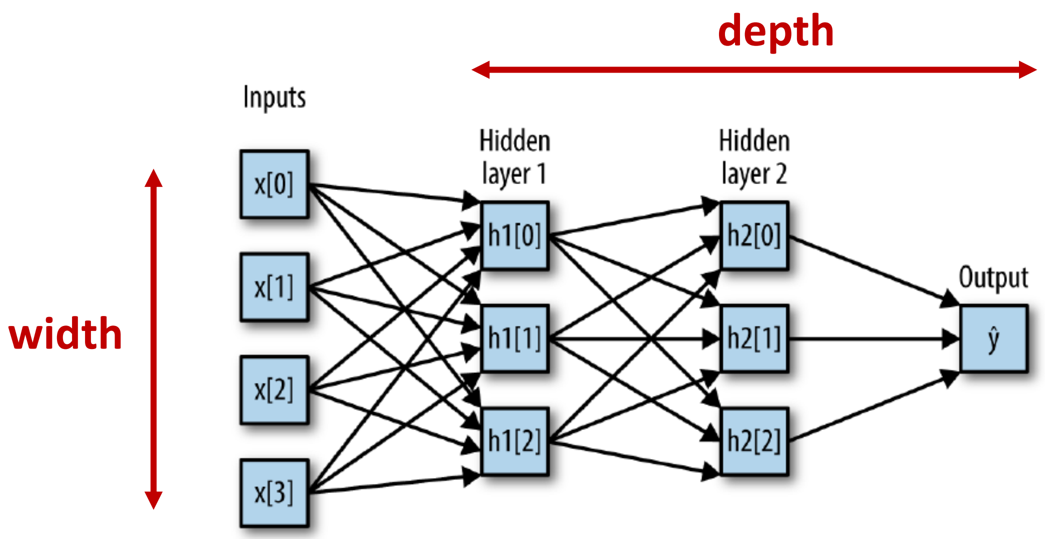






Model Architecture

- **Model Architecture**
  - **Input Layer**: Each *input unit* represents an *input feature*.
  - **Hidden Layer(s)**: Each *hidden unit* represents an intermediate processing step.
  - **Output Layer**: The *output unit* represents the *prediction* of the target label.
  - The connecting lines represent the learnable *parameters*



Model Architecture

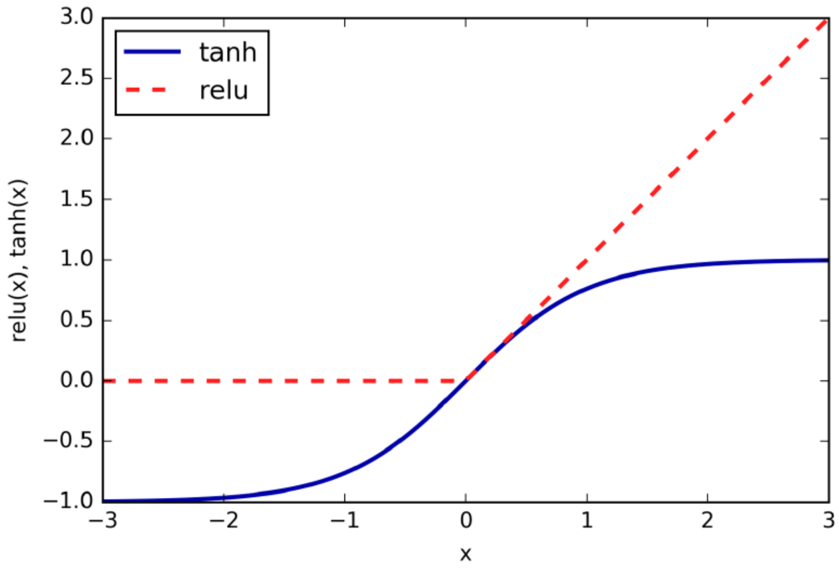
- **Hidden Layers: How do they work?** Chain-based architecture
  - The first hidden layer accepts the vector of the input layer $\boldsymbol{x}$, computing $\boldsymbol{z}^{(1)} = \mathbf{W}^{(1)^T}\boldsymbol{x} + \boldsymbol{b}^{(1)}$, then element-wise non-linear function $\boldsymbol{h}^{(1)} = g\left(\boldsymbol{z}^{(1)}\right)$.
  - The *i* th (*i*>1) hidden layer accepts the vector of the *i*-1 th hidden layer $\boldsymbol{h}^{(i-1)}$, computing $\boldsymbol{z}^{(i)} = \mathbf{W}^{(i)^T}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}$, then $\boldsymbol{h}^{(i)} = g\left(\boldsymbol{z}^{(i)}\right)$.
  - More hidden layers and units result in a more complex model.
- **Non-linear activation function for hidden units**
  - Every unit in a hidden layer computes weighted sum of the outputs from the preceding layer and then applies a *non-linear activation function*.
  - To make the neural network truly more powerful than a linear model, we need to use a nonlinear activation function at hidden units, which allows the neural network to learn much more complicated functions than a linear model could.
  - Computing a series of weighted sums without non-linear activation function is mathematically the same as computing just one weighted sum. → the neural network is then just a linear model.

Model Architecture

- **Hidden Layers: Exploiting non-linearity**

  Example of nonlinear activation functions

  - **rectifying nonlinear unit (*relu*)** $g(z) = \max(0, z)$
    : cuts off values below zero,
  - **hyperbolic tangent (*tanh*)** $g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
    : saturates to –1 for low input values
    and +1 for high input values.



Model Architecture

- **Output Layer: Making predictions for the target task**

  The output of the last hidden layer $\mathbf{h}$ becomes the input for the output layer

  $\hat{y} = f^{(l)}(\mathbf{h})$

  - Linear Unit (for regression, $y \in \mathbb{R}$)
    - $\hat{y} = \mathbf{w}^{\mathrm{T}}\mathbf{h} + b$
  - Sigmoid Unit (for binary classification, $y \in \{0,1\}$)
    - $\hat{y} = P(y = 1|\boldsymbol{x}) = \sigma(z) = \sigma\left(\mathbf{w}^{\mathrm{T}}\mathbf{h} + b\right)$
  - Softmax Units (for multi-class classification, $y \in \{1,2,\ldots,c\}$
    - $\hat{\boldsymbol{y}} = (\hat{y}_1, \ldots, \hat{y}_c)$, where $\hat{y}_k = p(y = k|\boldsymbol{x})$
    - $\boldsymbol{z} = (z_1, \ldots, z_c) = \mathbf{W}^{\mathrm{T}}\mathbf{h} + \boldsymbol{b}$, $\hat{y}_k = \mathrm{softmax}(\boldsymbol{z})_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$ so that $\sum_k \hat{y}_k = 1$

Optimization

- Given a (training) dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of *d* features and y_i is the corresponding target label.

- The model: $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The cost function (to be minimized, usually non-convex)

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} L(y_i, \hat{y}_i)$$

- For training, any gradient-based optimization algorithm can be used.
  - *e.g.*, simple gradient descent $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

$\epsilon > 0$ is the learning rate

Optimization

- Typical choice of the loss function $L(y_i, \hat{y}_i)$
  - For regression ($y_i \in \mathbb{R}$), use squared error

$$L(y_i, \hat{y}_i) = (\hat{y}_i - y_i)^2$$

  - For binary classification ($y_i \in \{0,1\}$), use binary cross-entropy

$$L(y_i, \hat{y}_i) = [-y_i \log \hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)]$$

  - For multi-class classification ($y_i \in \{1,2,\dots,c\}$, $\boldsymbol{y}_i = \mathrm{one\_hot}(y_i) = (y_{i1}, \dots, y_{ic})$), use categorical cross-entropy

$$L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = -\sum_{k=1}^{c} y_{ik} \log \widehat{y_{ik}}$$

Optimization

- $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$? Where does it come from?

- **Let's recall "Taylor series" of calculus**
  - Taylor expansion of a function of $\boldsymbol{\theta}$

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \cdots$$

  - First-order approximation (assume that $\boldsymbol{\theta}$ is very close to $\boldsymbol{\theta}_0$)

$$J(\boldsymbol{\theta}) \simeq J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

  - We want to find a direction $\boldsymbol{\theta}_0 \to \boldsymbol{\theta}$ to make $J(\boldsymbol{\theta}) < J(\boldsymbol{\theta}_0)$

$$J(\boldsymbol{\theta}) - J(\boldsymbol{\theta}_0) \simeq (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) < 0$$

linear function w.r.t. $\boldsymbol{\theta}$

  - The best direction

$$(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \propto -\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

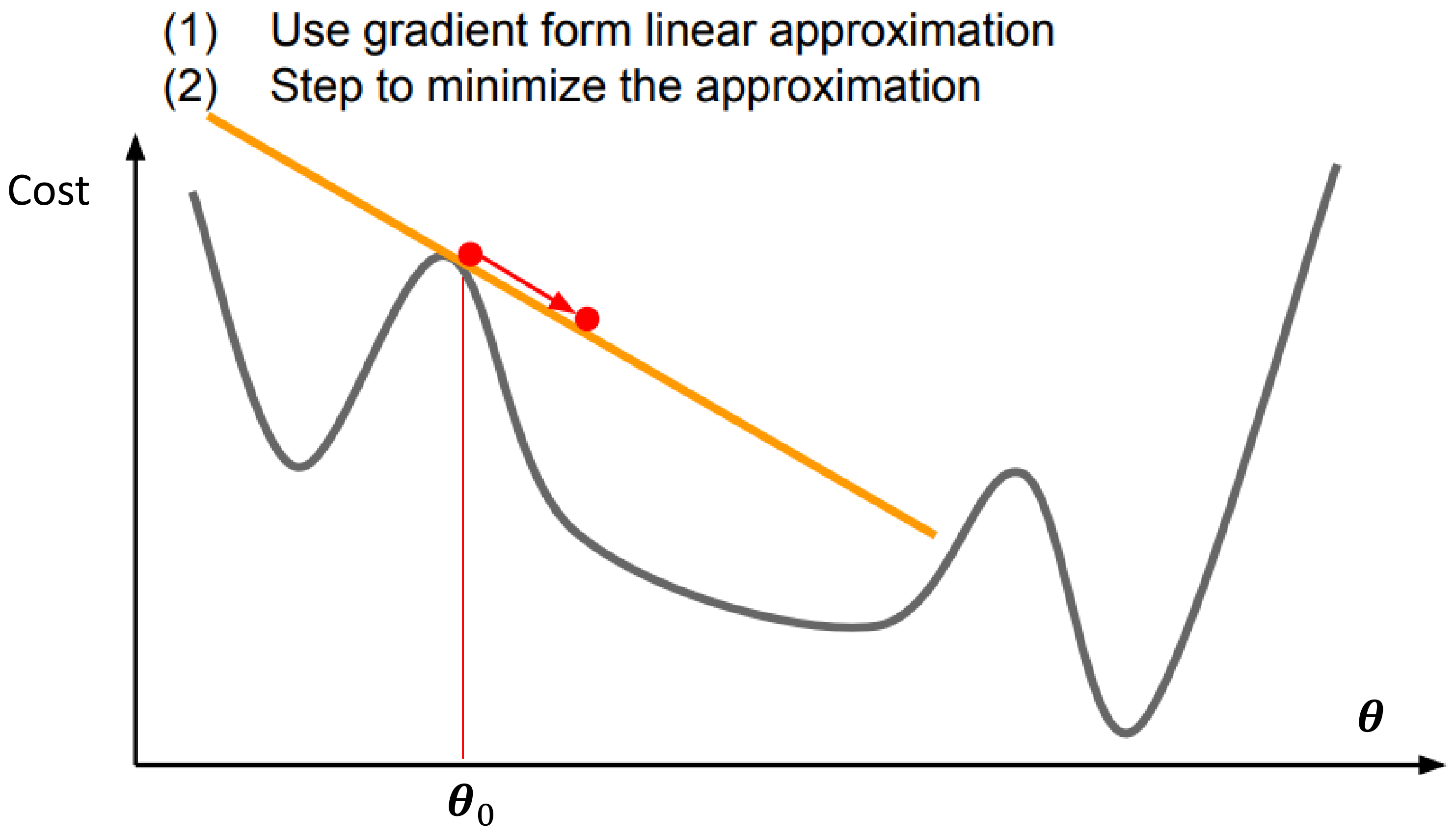
$$(\boldsymbol{\theta} - \boldsymbol{\theta}_0) = -\epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \epsilon > 0$$

$$\boldsymbol{\theta} = \boldsymbol{\theta}_0 - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \epsilon > 0$$

why?

Optimization

- **Illustrative Example**


(1) Use gradient form linear approximation
(2) Step to minimize the approximation
Cost
$\boldsymbol{\theta}$
$\boldsymbol{\theta}_0$

Hyperparameters

- **Model Architecture**
  - *hidden_layer_sizes* : tuple, length = n_layers - 2, default (100,)
  - *activation* : {'identity', 'logistic', 'tanh', 'relu'}, default 'relu'

    when features are too noisy with extremely large values?

    → *tanh and sigmoid can help reduce the impact of large magnitudes*

- **Model Training (Optimization, Regularization)**
  - *alpha* : L2 regularization (by default)
  - *solver* : {'lbfgs', 'sgd', 'adam'}, default 'adam'

    with more advanced options including *batch_size, learning_rate, max_iter, early_stopping, …*

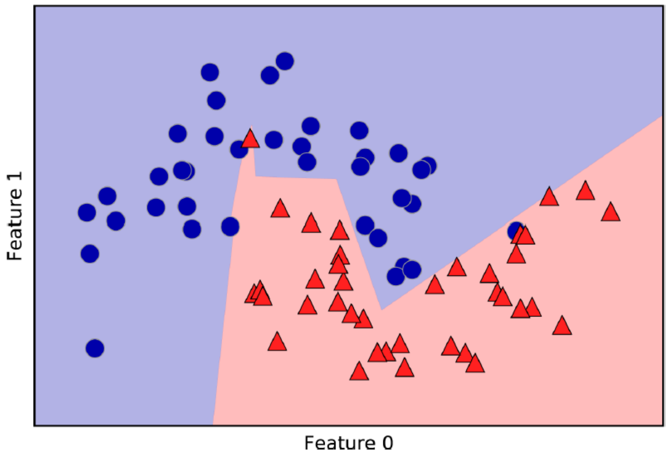
** The default solver 'adam' works pretty well on relatively large datasets (with thousands of training data points or more) in terms of both training time and validation score.*

** For small datasets, however, 'lbfgs' can converge faster and perform better.*

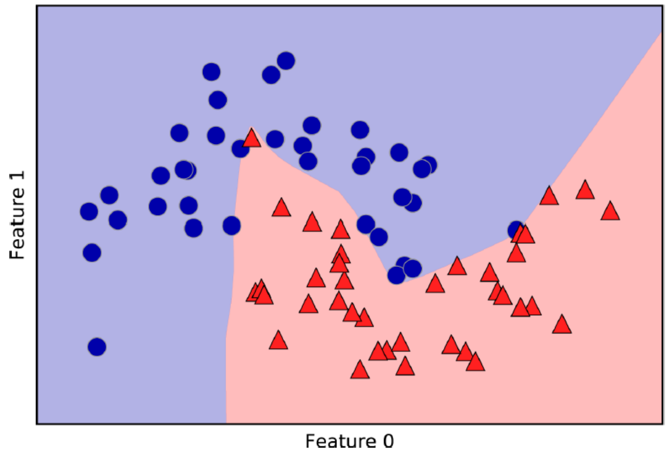
Hyperparameters

- **Example (*two_moon* dataset)**
  - *different numbers of hidden layers and hidden units*

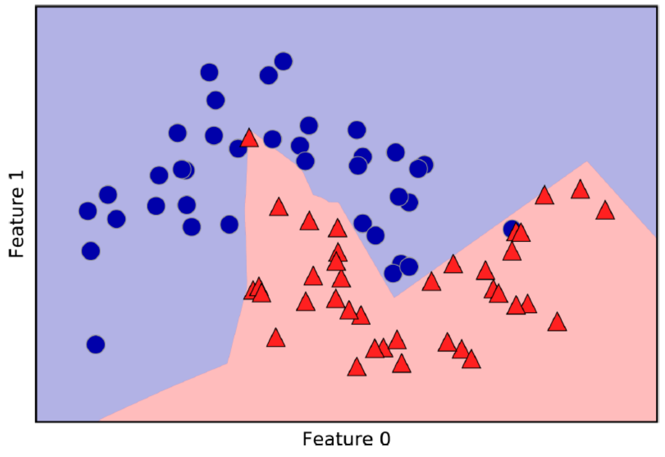
hidden_layer_sizes = (10,)



hidden_layer_sizes = (100,)

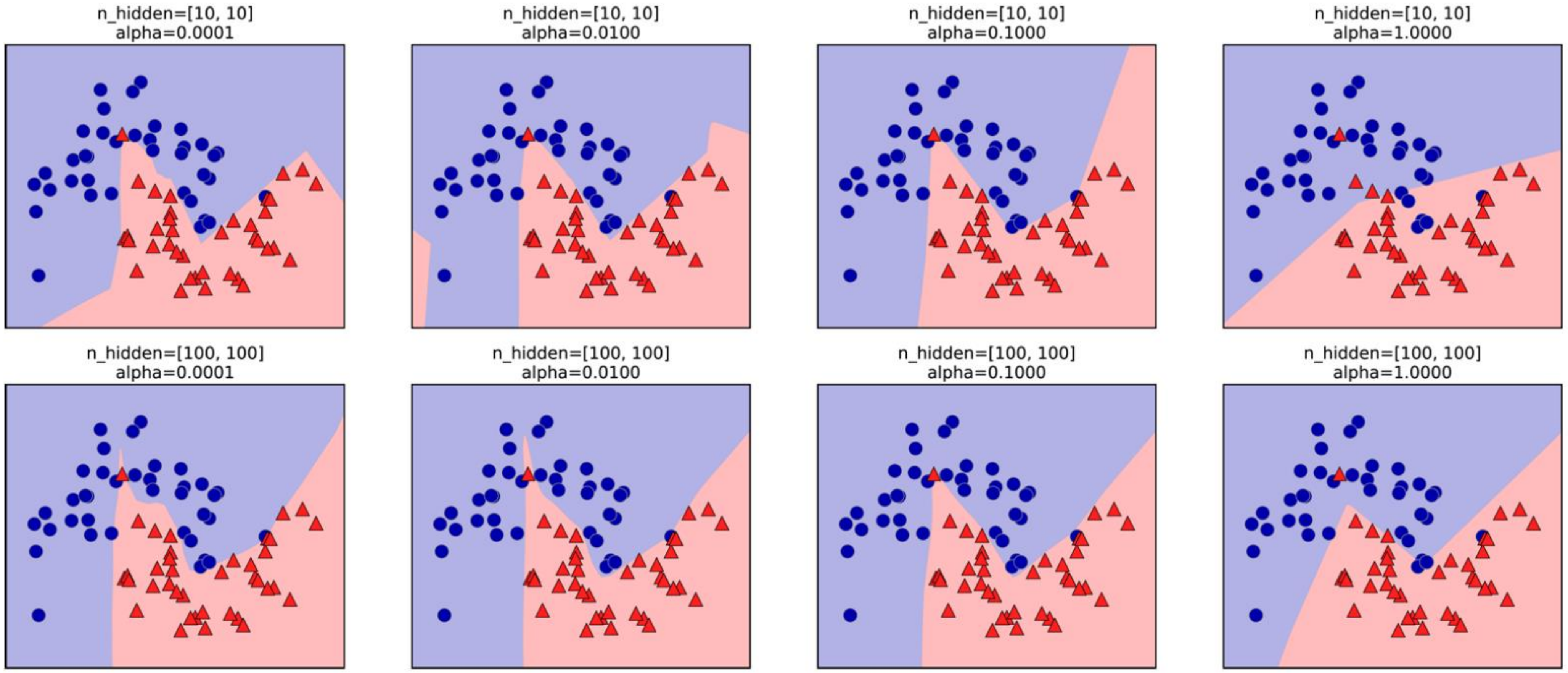


hidden_layer_sizes = (10,10,)



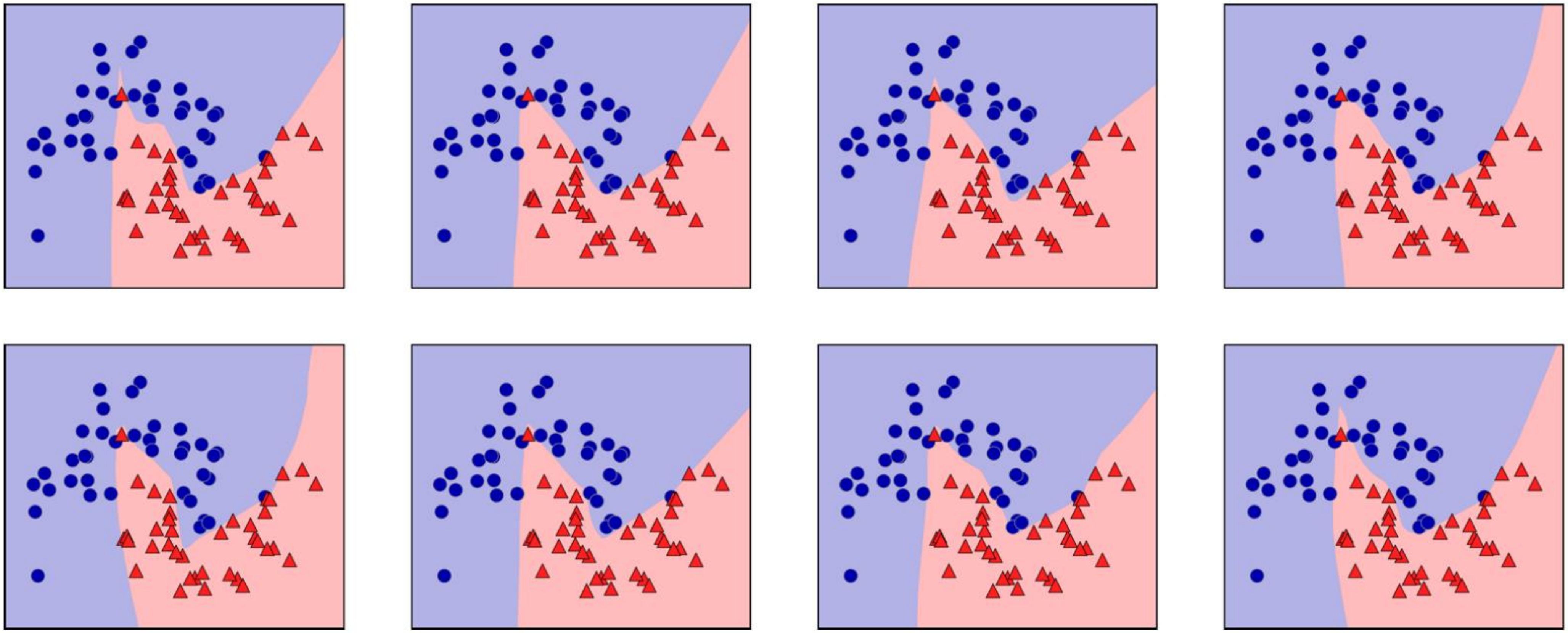
Hyperparameters

- **Example (*two_moon* dataset)**
  - *different numbers of hidden units and different settings of the alpha hyperparameter*



Hyperparameters

- **Example (*two_moon* dataset)**
  - *the same hyperparameters but different random initializations*



scikit-learn Practice: *MLPClassifier*

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

```
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,), activation='relu', *,
solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant’,
learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None,
tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True,
early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
n_iter_no_change=10, max_fun=15000)
```

**** It applies an L2 regularization by default***

Multi-layer Perceptron classifier.

This model optimizes the **log-loss function** using LBFGS or stochastic gradient descent.

hidden_layer_sizes	*array-like of shape(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. • ‘identity’, no-op activation, useful to implement linear bottleneck, returns f(x) = x • ‘logistic’, the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)). • ‘tanh’, the hyperbolic tan function, returns f(x) = tanh(x). • ‘relu’, the rectified linear unit function, returns f(x) = max(0, x)

scikit-learn Practice: *MLPClassifier*

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

solver	*{‘lbfgs’, ‘sgd’, ‘adam’}, default=’adam’* The solver for weight optimization. • ‘lbfgs’ is an optimizer in the family of quasi-Newton methods. • ‘sgd’ refers to stochastic gradient descent. • ‘adam’ refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba For a comparison between Adam optimizer and SGD, see Compare Stochastic learning strategies for MLPClassifier. Note: The default solver ‘adam’ works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, ‘lbfgs’ can converge faster and perform better.
alpha	*float, default=0.0001* Strength of the L2 regularization term. The L2 regularization term is divided by the sample size when added to the loss. For an example usage and visualization of varying regularization, see Varying regularization in Multi-layer Perceptron.

scikit-learn Practice: *MLPClassifier*

- **Example (*breast_cancer* dataset)**

```
[1]: from sklearn.datasets import load_breast_cancer
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.neural_network import MLPClassifier
     from sklearn.metrics import accuracy_score

     cancer = load_breast_cancer()
     X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=42)
```

```
[2]: scaler = StandardScaler()
     scaler.fit(X_train)
     X_train_scaled = scaler.transform(X_train)
     X_test_scaled = scaler.transform(X_test)
```

```
[3]: clf = MLPClassifier(max_iter=1000, random_state=0)
     clf.fit(X_train_scaled, y_train)
```

```
MLPClassifier(max_iter=1000, random_state=0)
```

```
[4]: y_train_hat = clf.predict(X_train)
     print('train accuracy: %.5f'%accuracy_score(y_train, y_train_hat))
     y_test_hat = clf.predict(X_test)
     print('test accuracy: %.5f'%accuracy_score(y_test, y_test_hat))
```

```
train accuracy: 0.99765
test accuracy: 0.96503
```

scikit-learn Practice: *MLPClassifier*

- **Example (*breast_cancer* dataset)**

```
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, stratify=cancer.target,
random_state=42)
```

```
clf = MLPClassifier(max_iter=1000, random_state=0)
clf.fit(X_train_scaled, y_train)
```

`X_train`

`y_train`

`clf`

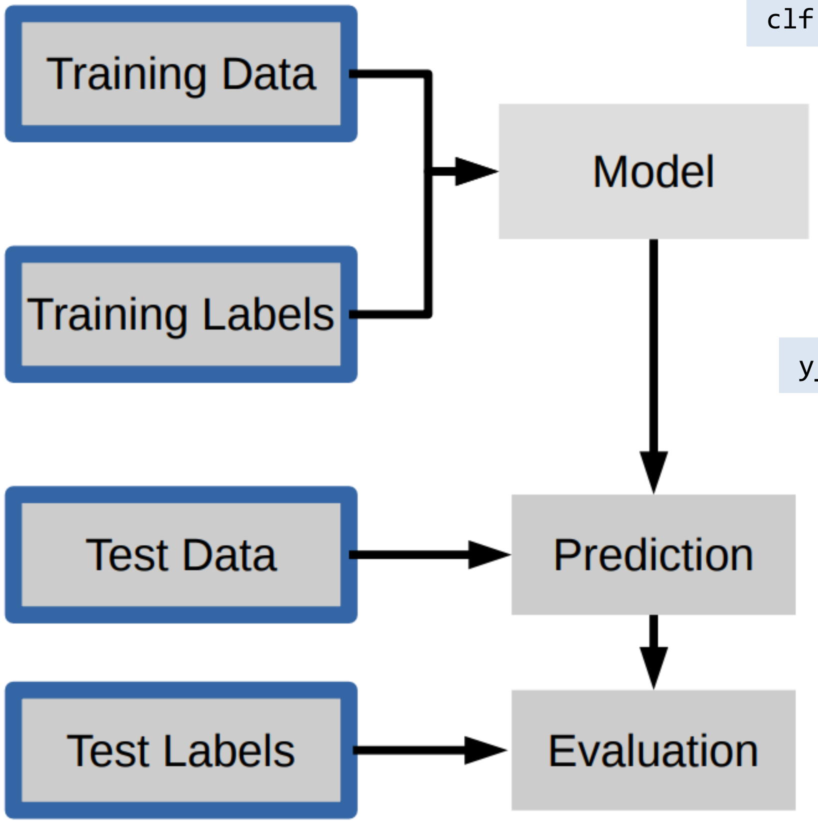
```
y_test_hat = clf.predict(X_test_scaled)
```

`X_test`

`y_test_hat`

`y_test`

`accuracy_score(y_test, y_test_hat)`



scikit-learn Practice: *MLPRegressor*

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

```
class sklearn.neural_network.MLPRegressor(hidden_layer_sizes=(100,), activation='relu', *,
solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant’,
learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None,
tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True,
early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
n_iter_no_change=10, max_fun=15000)
```

**** It applies an L2 regularization by default***

Multi-layer Perceptron regressor.

This model optimizes the **squared error** using LBFGS or stochastic gradient descent.

scikit-learn Practice: *MLPRegressor*

- **Example (*extended_boston* dataset)**

```
[1]: import mglearn
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.neural_network import MLPRegressor
     from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

     X, y = mglearn.datasets.load_extended_boston()
     X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
[2]: scalerX = StandardScaler()
     scalerX.fit(X_train)
     X_train_scaled = scalerX.transform(X_train)
     X_test_scaled = scalerX.transform(X_test)

     scalerY = StandardScaler()
     scalerY.fit(y_train.reshape(-1,1))
     y_train_scaled = scalerY.transform(y_train.reshape(-1,1))
     y_test_scaled = scalerY.transform(y_test.reshape(-1,1))
```

scikit-learn Practice: *MLPRegressor*

- **Example (*extended_boston* dataset)**

```
[3]: reg = MLPRegressor(max_iter=1000, random_state=0)
     reg.fit(X_train_scaled, y_train_scaled)
```

```
MLPRegressor(max_iter=1000, random_state=0)
```

```
[4]: y_train_hat_scaled = reg.predict(X_train_scaled)
     y_train_hat = scalerY.inverse_transform(y_train_hat_scaled.reshape(-1,1))
     print('train MAE: %.5f'%mean_absolute_error(y_train,y_train_hat))
     print('train RMSE: %.5f'% mean_squared_error(y_train,y_train_hat)**0.5)
     print('train R_square: %.5f'%r2_score(y_train,y_train_hat))

     y_test_hat_scaled = reg.predict(X_test_scaled)
     y_test_hat = scalerY.inverse_transform(y_test_hat_scaled.reshape(-1,1))
     print('test MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
     print('test RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
     print('test R_square: %.5f'%r2_score(y_test,y_test_hat))
```

```
train MAE: 0.90392
train RMSE: 1.25077
train R_square: 0.98166
test MAE: 2.47600
test RMSE: 3.80356
test R_square: 0.82292
```

scikit-learn Practice: *MLPRegressor*

- **Example (*extended_boston* dataset)** with tanh activation

```
[3]: reg = MLPRegressor(activation='tanh', max_iter=1000, random_state=0)
     reg.fit(X_train_scaled, y_train_scaled)

     MLPRegressor(max_iter=1000, random_state=0)

[4]: y_train_hat_scaled = reg.predict(X_train_scaled)
     y_train_hat = scalerY.inverse_transform(y_train_hat_scaled.reshape(-1,1))
     print('train MAE: %.5f'%mean_absolute_error(y_train,y_train_hat))
     print('train RMSE: %.5f'% mean_squared_error(y_train,y_train_hat)**0.5)
     print('train R_square: %.5f'%r2_score(y_train,y_train_hat))

     y_test_hat_scaled = reg.predict(X_test_scaled)
     y_test_hat = scalerY.inverse_transform(y_test_hat_scaled.reshape(-1,1))
     print('test MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
     print('test RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
     print('test R_square: %.5f'%r2_score(y_test,y_test_hat))

     train MAE: 0.91912
     train RMSE: 1.26458
     train R_square: 0.98125
     test MAE: 2.43660
     test RMSE: 3.83380
     test R_square: 0.82010
```

Discussion

- **The main hyperparameters of neural networks**
  - **Neural network architecture:** *hidden_layer_sizes, activation*
  - **Training settings:** optimization & regularization

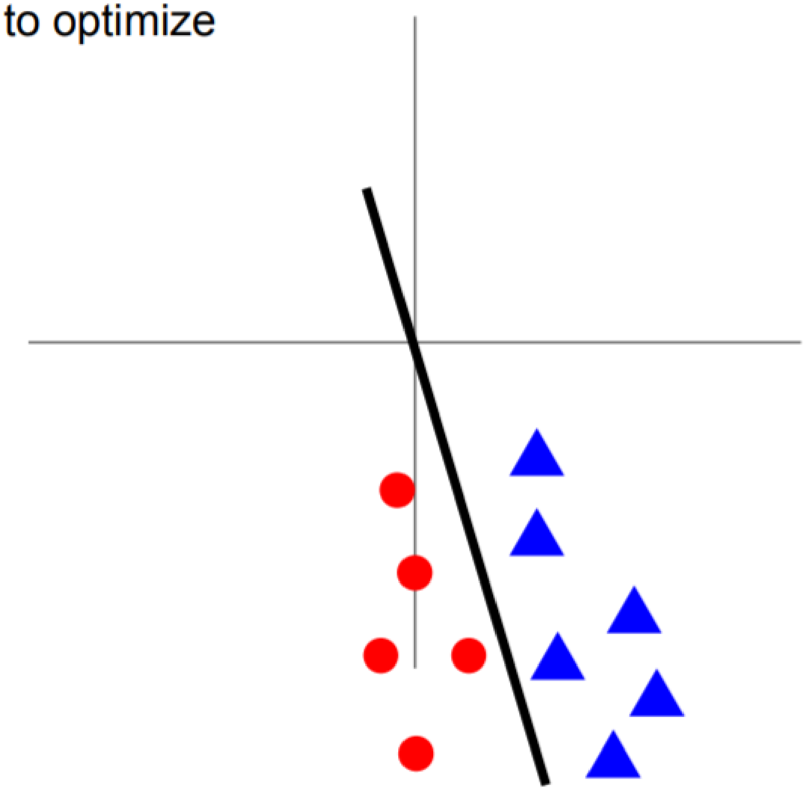
  * Typically chosen to achieve the highest performance on validation data

  * It's important to preprocess your data (including *data scaling* and *one-hot encoding*)
- **Strengths**
  - They are able to capture information contained in large amounts of data and build incredibly complex models, thereby providing good predictive ability.
  - Given enough computation time, data, and careful tuning of the hyperparameters, neural networks often beat other machine learning algorithms.
- **Weaknesses**
  - Large and complex neural networks often take a long time to train.
  - Tuning hyperparameters is also an art unto itself.
  - Considered a "black box" prediction machine, with no insight into relationships between features and target.

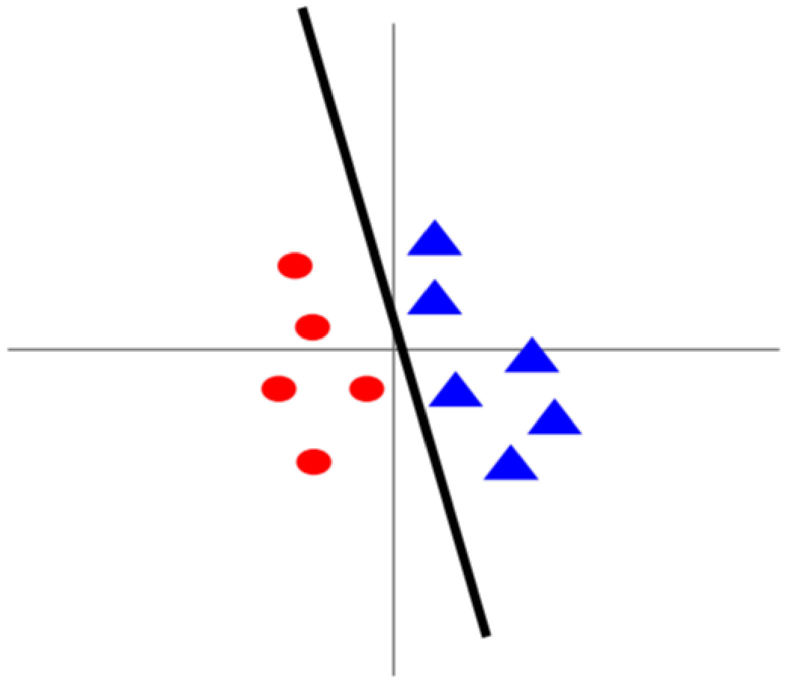
Discussion

- **Data Scaling**

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Practical Guidelines

- A common way to adjust hyperparameters in a neural network
  - [from lower to higher complexity] Start with one or two hidden layers, and possibly expand from there.
  - [from higher to lower complexity] First create a network that is large enough to overfit, making sure that the task can actually be learned by the network. Then, either shrink the network or add regularization

- The number of parameters that are learned is a helpful measure when thinking about the model complexity of a neural network.
  - ***Example:*** If you have a binary classification dataset with 50 input features and the neural network consists of two hidden layers with 10 hidden units,
    - 10 * (50 + 1) = 510 parameters between the input and the first hidden layer.
    - 10 * (10 + 1) = 110 parameters between the first hidden layer and the second hidden layer.
    - 1 * (10 + 1) = 11 parameters between the second hidden layer and the output layer

Moving Forward: Deep Learning

- *MLPClassifier* and *MLPRegressor* only capture a small subset of what is possible with neural networks.
- If you are interested in working with more flexible or larger models, you should look beyond scikit-learn into the following deep learning libraries.

TensorFlow PyTorch

- These deep learning libraries provide a much more flexible interface to build neural networks and track the rapid progress in deep learning research, and allow the use of high performance graphics processing units (GPUs) to accelerate computations.

Uncertainty Estimates from Classifiers

Uncertainty Estimates

- Often, you are interested in how certain a prediction is.
- In practice, different kinds of mistakes lead to very different outcomes in real-world applications.
  - Example: a medical application testing for cancer
    - a false positive prediction might lead to a patient undergoing additional tests.
    - a false negative prediction might lead to a serious disease not being treated.

- Most classifiers provide uncertainty estimates of their predictions.

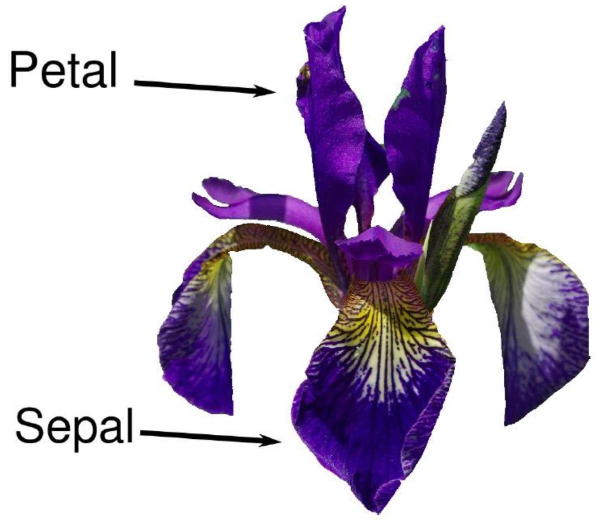
scikit-learn Practice

- **In the case of classification,**
  the output of *predict_proba* **method is a probability estimate for each class.**

Methods	
predict(*X*)	Predict the class labels for the provided data X. Input: **X** *{array-like, sparse matrix} of shape (n_samples, n_features)* The input data. Return: **y** *ndarray, shape (n_samples,) or (n_samples, n_classes)* The predicted classes.
predict_proba(*X*)	Return probability estimates for the provided data X. Input: **X** *{array-like, sparse matrix} of shape (n_samples, n_features)* The input data. Return: **y_prob** *ndarray of shape (n_samples, n_classes)* The predicted probability of the sample for each class in the model, where classes are ordered as they are in self.classes_.

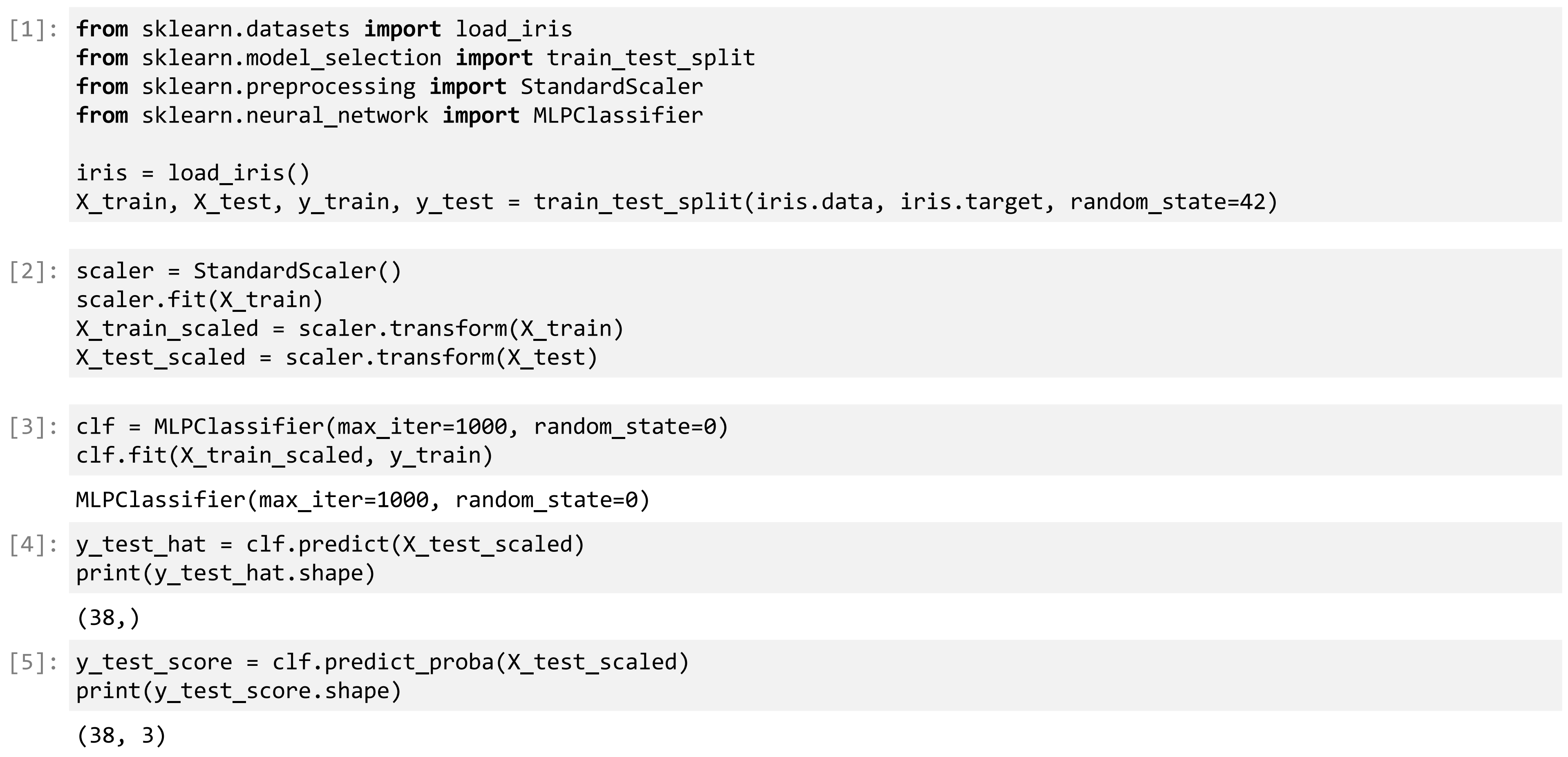
scikit-learn Practice

- **Example with the *iris* dataset**
  - The dataset consists of 150 data points with three classes (50 in each class), described by 4 features (multi-class classification)
  - The features are (1) the length of the petals, (2) the width of the petals, (3) the length of the sepals, and (4) the width of the sepals, all measured in centimeters.
  - Each point belongs to the species *setosa*, *versicolor*, or *virginica.*
  - The goal is to build a machine learning model that can learn from the measurements of these irises whose species is known, so that we can predict the species for a new iris.



scikit-learn Practice

- **Example (*iris* dataset)**



```
[1]: from sklearn.datasets import load_iris
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.neural_network import MLPClassifier

     iris = load_iris()
     X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, random_state=42)
```

```
[2]: scaler = StandardScaler()
     scaler.fit(X_train)
     X_train_scaled = scaler.transform(X_train)
     X_test_scaled = scaler.transform(X_test)
```

```
[3]: clf = MLPClassifier(max_iter=1000, random_state=0)
     clf.fit(X_train_scaled, y_train)

     MLPClassifier(max_iter=1000, random_state=0)
```

```
[4]: y_test_hat = clf.predict(X_test_scaled)
     print(y_test_hat.shape)

     (38,)
```

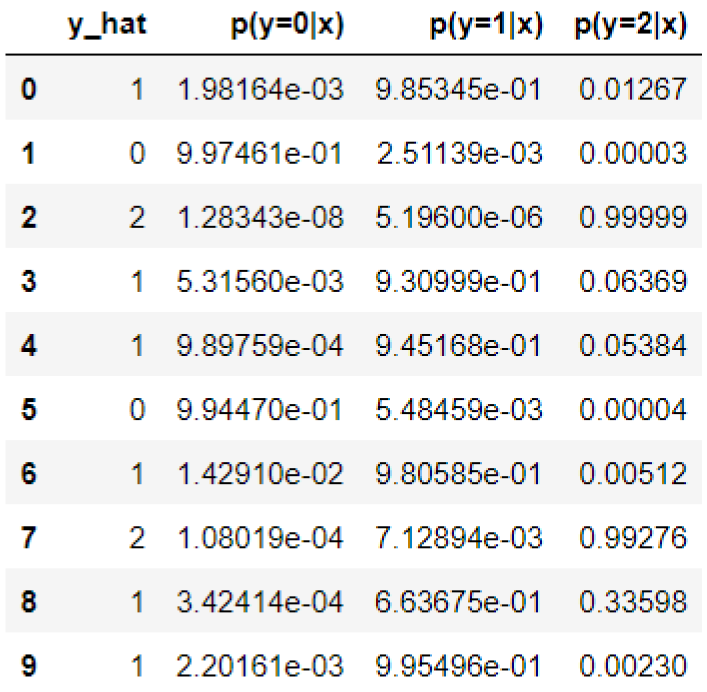
```
[5]: y_test_score = clf.predict_proba(X_test_scaled)
     print(y_test_score.shape)

     (38, 3)
```

scikit-learn Practice

- **Example (*iris* dataset)**

clf.predict(X_test_scaled)

clf.predict_proba(X_test_scaled)



	y_hat	p(y=0|x)	p(y=1|x)	p(y=2|x)
0	1	1.98164e-03	9.85345e-01	0.01267
1	0	9.97461e-01	2.51139e-03	0.00003
2	2	1.28343e-08	5.19600e-06	0.99999
3	1	5.31560e-03	9.30999e-01	0.06369
4	1	9.89759e-04	9.45168e-01	0.05384
5	0	9.94470e-01	5.48459e-03	0.00004
6	1	1.42910e-02	9.80585e-01	0.00512
7	2	1.08019e-04	7.12894e-03	0.99276
8	1	3.42414e-04	6.63675e-01	0.33598
9	1	2.20161e-03	9.95496e-01	0.00230

⋮

Uncertainty Measures

- **Common Uncertainty Measures for Classification**
  - Confidence

    $$U(\boldsymbol{x}) = -\max_j p(y = j|\boldsymbol{x})$$

  - Margin

    $$U(\boldsymbol{x}) = -[p(y = j_1|\boldsymbol{x}) - p(y = j_2|\boldsymbol{x})]$$

    j_1 and j_2 are the most and second-most probable classes.

  - Entropy

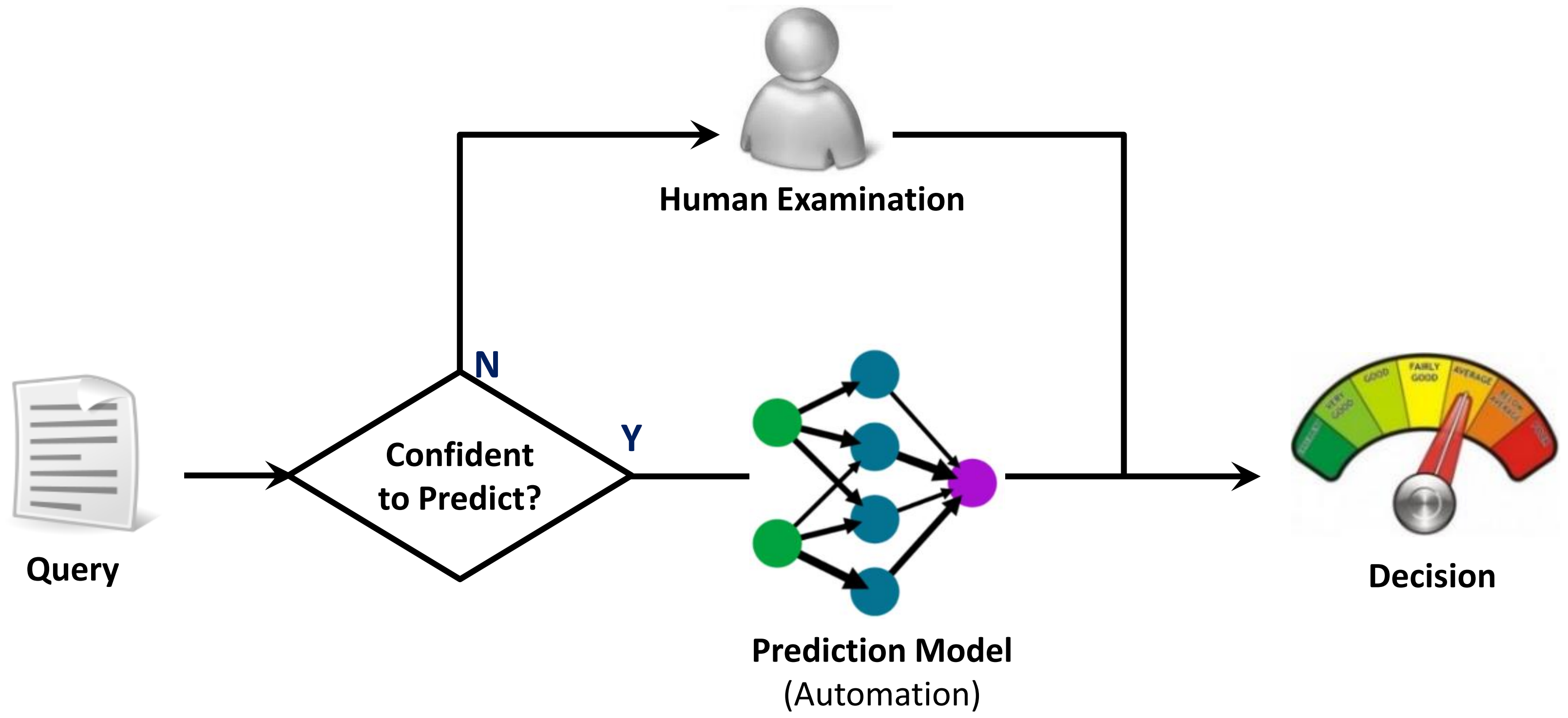
    $$U(\boldsymbol{x}) = \sum_j [-p(y = j|\boldsymbol{x}) \log p(y = j|\boldsymbol{x})]$$

  - Gini Impurity

    $$U(\boldsymbol{x}) = 1 - \sum_j p(y = j|\boldsymbol{x})^2$$

Classification with Reject Option

- **It is often appropriate to report a warning for those data points that are hard to classify, instead of predicting for all data points.**
- **The rejected data points can be conveyed to human experts for careful investigation.**



Summary and Outlook

Takeaway

- **Important Terms**
  - Generalization
  - Model complexity / Overfitting / Underfitting
  - Regularization
  - Parameter / Hyperparameter
  - Training / Validation / Test

- **Things to keep in mind**
  - **Setting the right hyperparameters** is important for good performance.
  - Some of the algorithms are also sensitive to **how we represent the input data.**
  - Blindly applying an algorithm to a dataset without **understanding the assumptions the algorithm makes and the meanings of the hyperparameter settings** will rarely lead to an accurate model.
  - One can usually do much better with **a correct application of a commonplace algorithm** than by sloppily applying an obscure algorithm.

Quick Summary

- **A model is a simplified version of the observations (training data).**
  - The simplifications are meant to discard the superfluous details that are unlikely to generalize to new data.
  - However, to decide what data to discard and what data to keep, you must make *assumptions*.
  - For example, a linear model makes the assumption that the data is fundamentally linear and that the distance between the data points and the straight line is just noise, which can safely be ignored.

Quick Summary

- **When to use each model?**
  - **Nearest neighbors:** For small datasets, good as a baseline, easy to explain.
  - **Linear models:** Go-to as a first algorithm to try, good for very large datasets, good for very high-dimensional data.
  - **Decision trees:** Very fast, don’t need scaling of features, can be visualized and easily explained.
  - **Random forests:** Nearly always perform better than a single decision tree, very robust and powerful. Don’t need scaling of features. Not good for very high-dimensional sparse data.
  - **Support vector machines:** Powerful for medium-sized datasets of features with similar meaning. Require scaling of features, sensitive to hyperparameters.
  - **Neural networks:** Can build very complex models, particularly for large datasets. Sensitive to scaling of features and to the choice of hyperparameters. Large models need a long time to train.

- **General Guideline**
  - When working with a new dataset, it is in general a good idea to start with a simple model, such as linear models or nearest neighbors, and see how far you can get.
  - After understanding more about the data, you can consider moving to an algorithm that can build more complex models, such as random forests or neural networks.

No-Free-Lunch Theorem

- ***No-Free-Lunch*** **theorem** (Wolpert, 1996): if you make absolutely no assumption about the data, then there is no reason to prefer one model over any other.
  - For some datasets the best model is a linear model, while for other datasets it is a neural network.
  - There is no model that is *a priori* guaranteed to work better.
  - **The only way to know for sure which model is best is to evaluate them all.**
  - Since this is not possible, in practice you make some reasonable assumptions about the data and you evaluate only a few reasonable models.

- No machine learning algorithm performs universally better than any other.
- The goal of machine learning research is **not to seek a universal learning algorithm or the absolute best learning algorithm.**
- We must design our machine learning algorithms **to perform well on a specific task**.