CE6146 Introduction to Deep Learning Optimization and Regularization

Chia-Ru Chung

Department of Computer Science and Information Engineering

National Central University

2023/10/5

20230928 Exercise

1. **C**	2. **C**	3. **B**	4. **A**	5. **B**
6. **A**	7. **D**	8. **C**	9. **C**	10. **B**
11. **D**	12. **B**	13. **A**	14. **C**	15. **C**
16. **B**	17. **B**	18. **C**	19. **C**	20. **C**

Outline

- Review
- Optimization and Regularization
- Hand-on Feedforward Networks

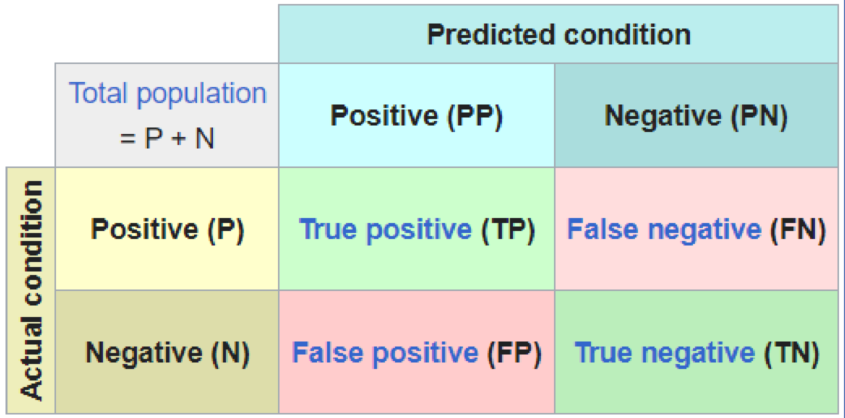
Review

- Evaluation Metrics
- Cross-Validation
- Activation Function
- Forward and Backward Pass

Evaluations for Classification

- Sensitivity (hit rate, true positive rate, or recall) $= \frac{TP}{P} = \frac{TP}{TP+FN}$
- Specificity (true negative rate) $= \frac{TN}{N} = \frac{TN}{TN+FP}$
- Precision (Positive Predictive Value, PPV)$= \frac{TP}{TP+FP}$
- Accuracy (ACC) $= \frac{TP+TN}{P+N} = \frac{TP+TN}{TP+TN+FP+FN}$
- F1 score $= \frac{2TP}{2TP+FP+FN}$
- Matthews correlation coefficient (MCC) $= \frac{TP\times TN-FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

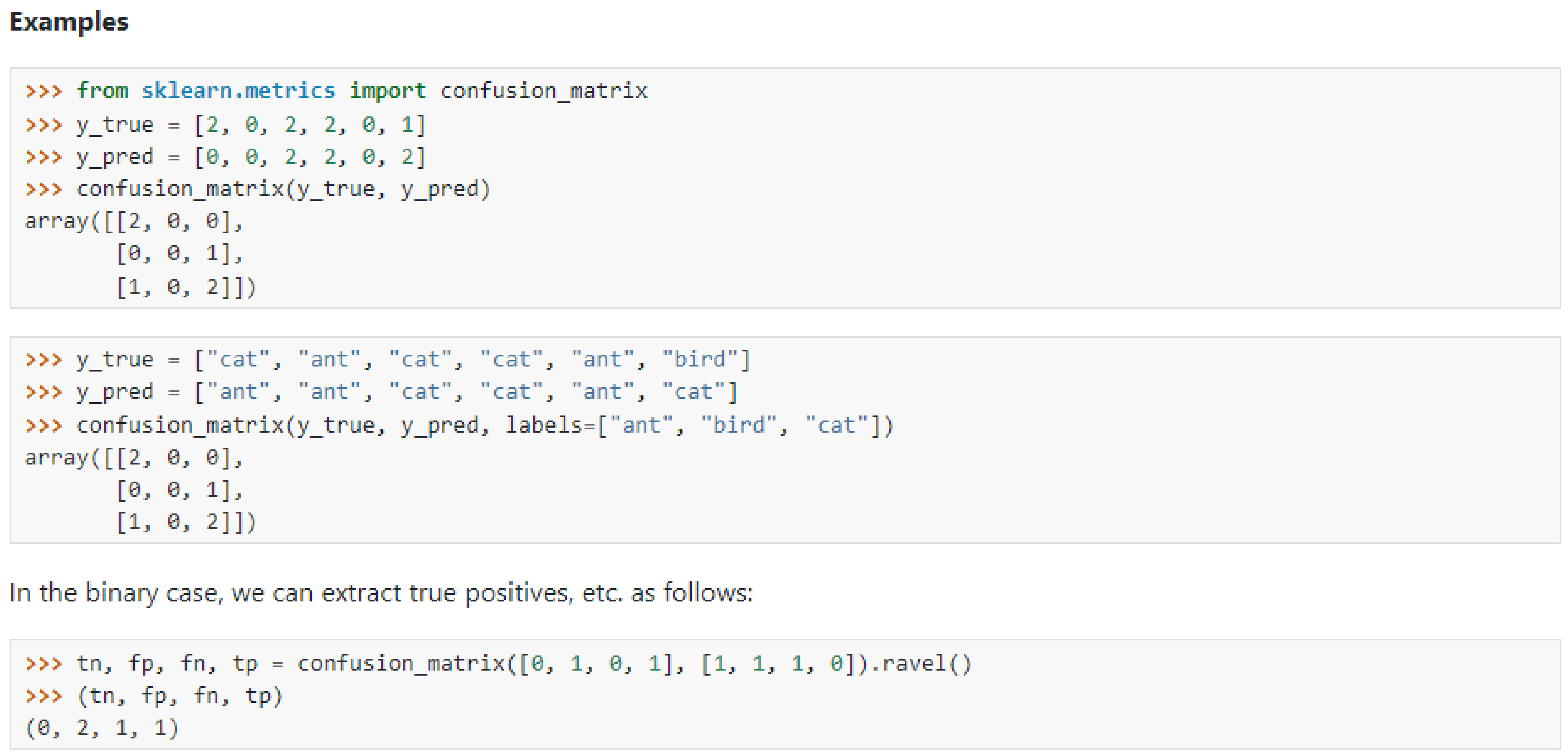
Confusion matrix



		Predicted condition	
	Total population = P + N	Positive (PP)	Negative (PN)
Actual condition	Positive (P)	True positive (TP)	False negative (FN)
	Negative (N)	False positive (FP)	True negative (TN)

Source: https://en.wikipedia.org/wiki/Confusion_matrix

Confusion Matrix in Python



Examples

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

In the binary case, we can extract true positives, etc. as follows:

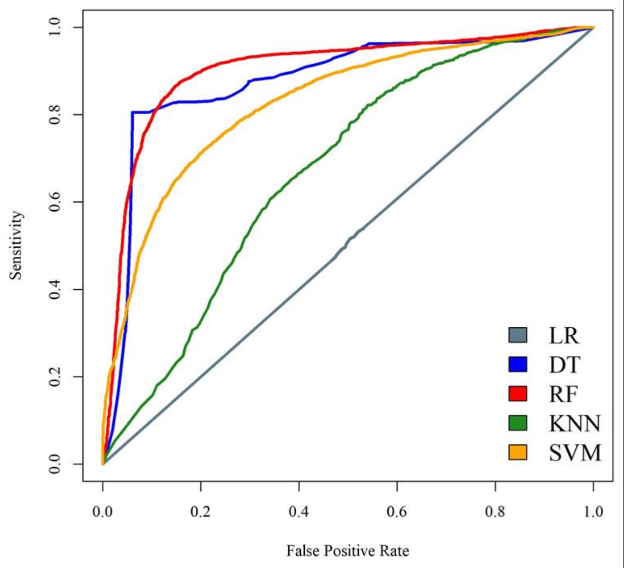
```
>>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
>>> (tn, fp, fn, tp)
(0, 2, 1, 1)
```

Source: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

Receiver Operating Characteristic Curve

- A graphical representation to evaluate the performance of a binary classification model.
- Plots True Positive Rate (TPR) against False Positive Rate (FPR) at various thresholds.
- $\text{TPR} = \frac{TP}{P} = \frac{TP}{TP+FN}$
- $\text{FPR} = \frac{FP}{N} = \frac{FP}{FP+TN} = 1 - \text{Specificity}$

Steps to Create an ROC Curve



Step 1: Sort all predicted probabilities in descending order.

$$p_1 \geq p_2 \geq \cdots \geq p_n$$

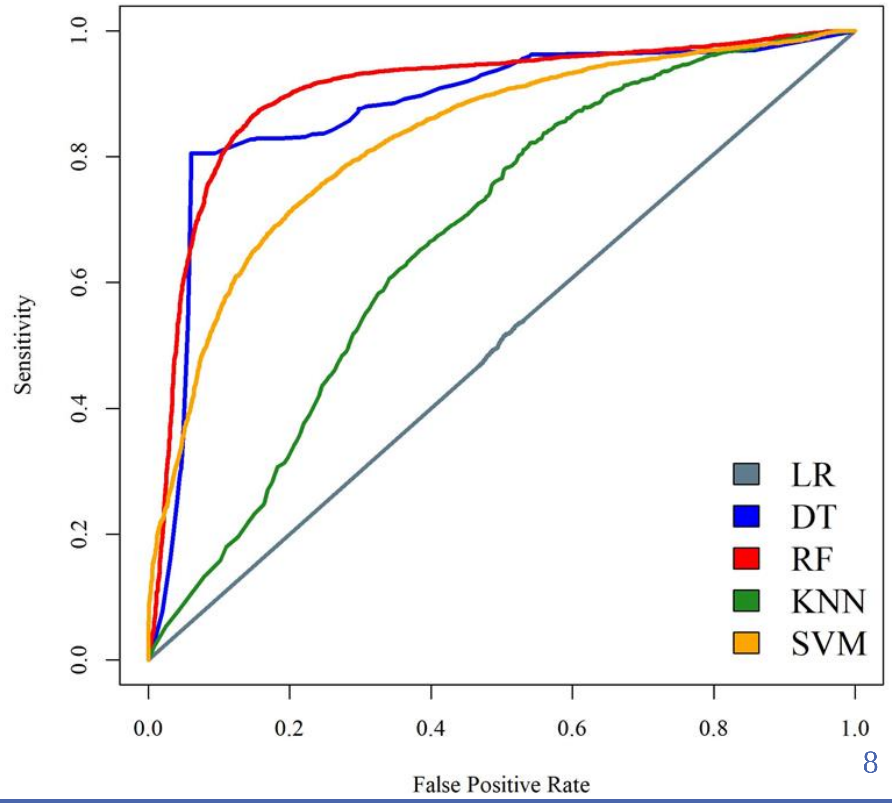
Step 2: Calculate TPR and FPR for each threshold t.

- Classify instances with $p_i \geq t$ as positive and $p_i < t$ as negative
- Calculate TP, FP, TN, and FN for this t
- Calculate TPR(t) and FPR(t)

Step 3: For each t, plot FPR(t) on the X-axis and TPR(t) on the Y-axis.

Interpreting the ROC Curve

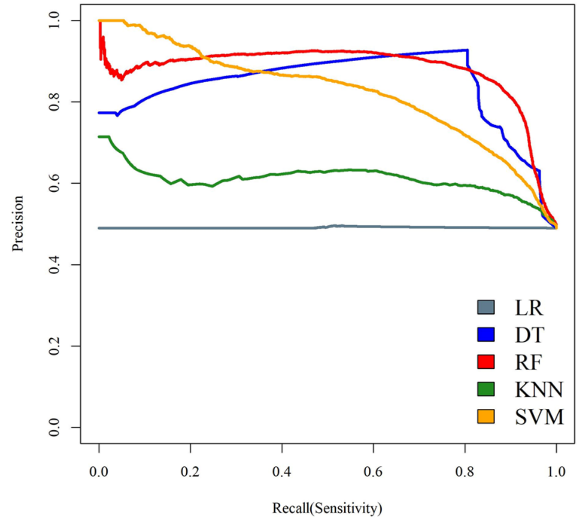
- Ideal Point: The top-left corner of the plot, indicating a FPR of 0 and a TPR of 1.
- AUC: Area Under the Curve
  - Closer to 1 indicates a better model.
  - Perfect classifier: AUC = 1
  - Random classifier: AUC = 0.5



Precision-Recall Curve

- A curve that visualizes the trade-off between Precision and Recall for different threshold values.
- Useful for evaluating the performance of a classification model, especially when classes are imbalanced.
- Precision $= \frac{TP}{TP+FP}$
- Recall $= \frac{TP}{TP+FN}$

Steps to Create a PR Curve



Step 1: Sort all predicted probabilities in descending order.

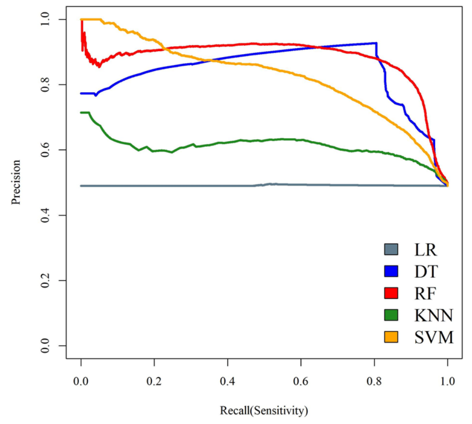
$$p_1 \geq p_2 \geq \cdots \geq p_n$$

Step 2: Calculate Recall and Precision for each threshold t.

- Classify instances with $p_i \geq t$ as positive and $p_i < t$ as negative
- Calculate TP, FP, and FN for this t
- Calculate Precision(t) and Recall(t)

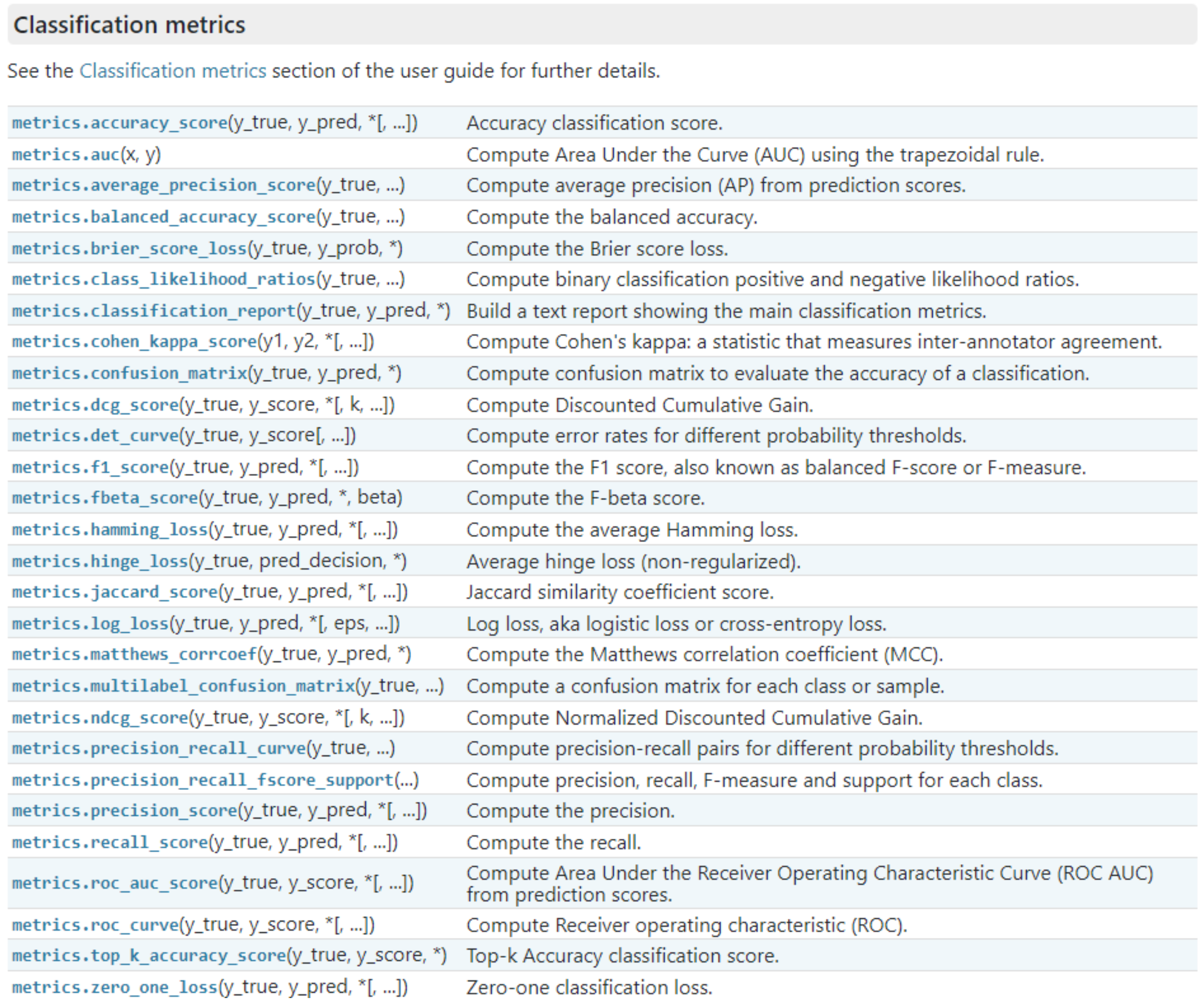
Step 3: For each t, plot Recall(t) on the X-axis and Precision(t) on the Y-axis.

Interpreting the PR Curve



- The "ideal" PR Curve is close to the <u>top right corner</u>, where both Precision and Recall are 1.
- A steeper curve suggests the model has better precision and recall.
  - A steeper curve is an indicator that for a small sacrifice in recall, you can get a significant increase in precision, or vice versa.
  - It suggests that <u>the model is able to achieve high precision without sacrificing much recall</u>, or <u>high recall without sacrificing much precision</u>.

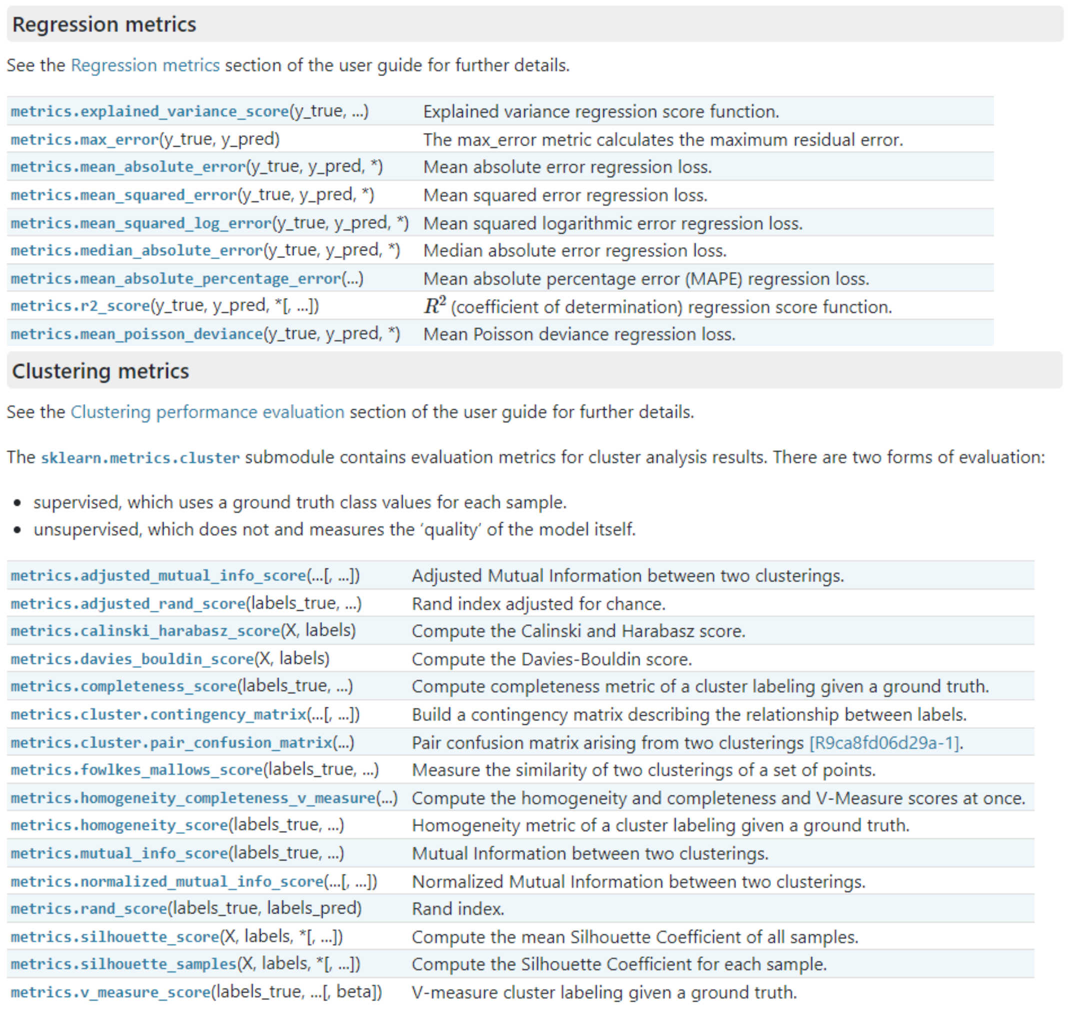
Evaluations Metrics in Python



Classification metrics

See the Classification metrics section of the user guide for further details.

`metrics.accuracy_score`(y_true, y_pred, *[, ...])	Accuracy classification score.
`metrics.auc`(x, y)	Compute Area Under the Curve (AUC) using the trapezoidal rule.
`metrics.average_precision_score`(y_true, ...)	Compute average precision (AP) from prediction scores.
`metrics.balanced_accuracy_score`(y_true, ...)	Compute the balanced accuracy.
`metrics.brier_score_loss`(y_true, y_prob, *)	Compute the Brier score loss.
`metrics.class_likelihood_ratios`(y_true, ...)	Compute binary classification positive and negative likelihood ratios.
`metrics.classification_report`(y_true, y_pred, *)	Build a text report showing the main classification metrics.
`metrics.cohen_kappa_score`(y1, y2, *[, ...])	Compute Cohen's kappa: a statistic that measures inter-annotator agreement.
`metrics.confusion_matrix`(y_true, y_pred, *)	Compute confusion matrix to evaluate the accuracy of a classification.
`metrics.dcg_score`(y_true, y_score, *[, k, ...])	Compute Discounted Cumulative Gain.
`metrics.det_curve`(y_true, y_score[, ...])	Compute error rates for different probability thresholds.
`metrics.f1_score`(y_true, y_pred, *[, ...])	Compute the F1 score, also known as balanced F-score or F-measure.
`metrics.fbeta_score`(y_true, y_pred, *, beta)	Compute the F-beta score.
`metrics.hamming_loss`(y_true, y_pred, *[, ...])	Compute the average Hamming loss.
`metrics.hinge_loss`(y_true, pred_decision, *)	Average hinge loss (non-regularized).
`metrics.jaccard_score`(y_true, y_pred, *[, ...])	Jaccard similarity coefficient score.
`metrics.log_loss`(y_true, y_pred, *[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss.
`metrics.matthews_corrcoef`(y_true, y_pred, *)	Compute the Matthews correlation coefficient (MCC).
`metrics.multilabel_confusion_matrix`(y_true, ...)	Compute a confusion matrix for each class or sample.
`metrics.ndcg_score`(y_true, y_score, *[, k, ...])	Compute Normalized Discounted Cumulative Gain.
`metrics.precision_recall_curve`(y_true, ...)	Compute precision-recall pairs for different probability thresholds.
`metrics.precision_recall_fscore_support`(...)	Compute precision, recall, F-measure and support for each class.
`metrics.precision_score`(y_true, y_pred, *[, ...])	Compute the precision.
`metrics.recall_score`(y_true, y_pred, *[, ...])	Compute the recall.
`metrics.roc_auc_score`(y_true, y_score, *[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
`metrics.roc_curve`(y_true, y_score, *[, ...])	Compute Receiver operating characteristic (ROC).
`metrics.top_k_accuracy_score`(y_true, y_score, *)	Top-k Accuracy classification score.
`metrics.zero_one_loss`(y_true, y_pred, *[, ...])	Zero-one classification loss.



Regression metrics

See the Regression metrics section of the user guide for further details.

`metrics.explained_variance_score`(y_true, ...)	Explained variance regression score function.
`metrics.max_error`(y_true, y_pred)	The max_error metric calculates the maximum residual error.
`metrics.mean_absolute_error`(y_true, y_pred, *)	Mean absolute error regression loss.
`metrics.mean_squared_error`(y_true, y_pred, *)	Mean squared error regression loss.
`metrics.mean_squared_log_error`(y_true, y_pred, *)	Mean squared logarithmic error regression loss.
`metrics.median_absolute_error`(y_true, y_pred, *)	Median absolute error regression loss.
`metrics.mean_absolute_percentage_error`(...)	Mean absolute percentage error (MAPE) regression loss.
`metrics.r2_score`(y_true, y_pred, *[, ...])	R^2 (coefficient of determination) regression score function.
`metrics.mean_poisson_deviance`(y_true, y_pred, *)	Mean Poisson deviance regression loss.

Clustering metrics

See the Clustering performance evaluation section of the user guide for further details.

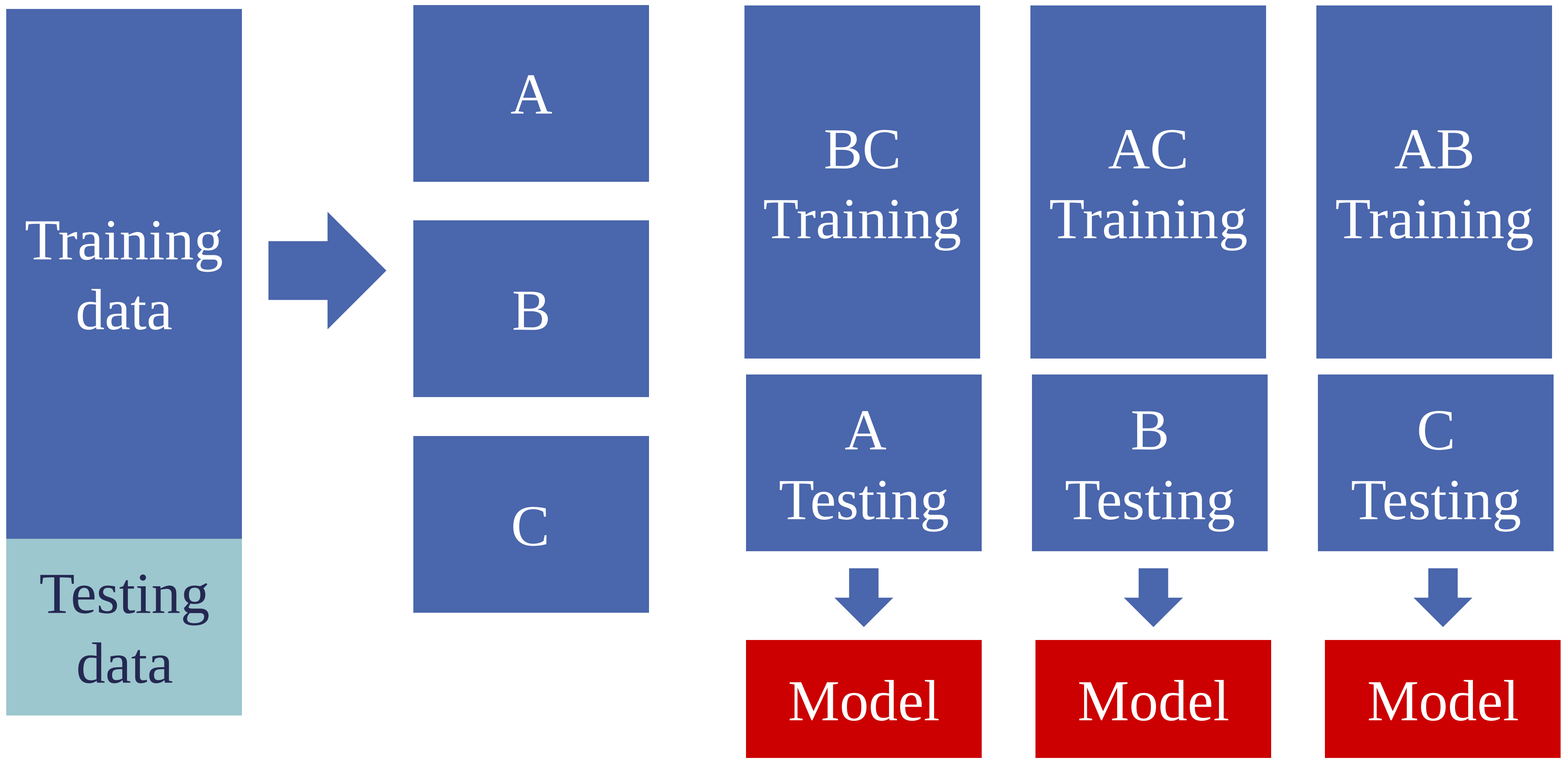
The `sklearn.metrics.cluster` submodule contains evaluation metrics for cluster analysis results. There are two forms of evaluation:

- supervised, which uses a ground truth class values for each sample.
- unsupervised, which does not and measures the 'quality' of the model itself.

`metrics.adjusted_mutual_info_score`(...[, ...])	Adjusted Mutual Information between two clusterings.
`metrics.adjusted_rand_score`(labels_true, ...)	Rand index adjusted for chance.
`metrics.calinski_harabasz_score`(X, labels)	Compute the Calinski and Harabasz score.
`metrics.davies_bouldin_score`(X, labels)	Compute the Davies-Bouldin score.
`metrics.completeness_score`(labels_true, ...)	Compute completeness metric of a cluster labeling given a ground truth.
`metrics.cluster.contingency_matrix`(...[, ...])	Build a contingency matrix describing the relationship between labels.
`metrics.cluster.pair_confusion_matrix`(...)	Pair confusion matrix arising from two clusterings [R9ca8fd06d29a-1].
`metrics.fowlkes_mallows_score`(labels_true, ...)	Measure the similarity of two clusterings of a set of points.
`metrics.homogeneity_completeness_v_measure`(...)	Compute the homogeneity and completeness and V-Measure scores at once.
`metrics.homogeneity_score`(labels_true, ...)	Homogeneity metric of a cluster labeling given a ground truth.
`metrics.mutual_info_score`(labels_true, ...)	Mutual Information between two clusterings.
`metrics.normalized_mutual_info_score`(...[, ...])	Normalized Mutual Information between two clusterings.
`metrics.rand_score`(labels_true, labels_pred)	Rand index.
`metrics.silhouette_score`(X, labels, *[, ...])	Compute the mean Silhouette Coefficient of all samples.
`metrics.silhouette_samples`(X, labels, *[, ...])	Compute the Silhouette Coefficient for each sample.
`metrics.v_measure_score`(labels_true, ...[, beta])	V-measure cluster labeling given a ground truth.

Source: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metricsl

Cross-Validation



Visualization of High-Dimensional Data

- The goal is to reduce the dimensions in a way that retains the most important structures or patterns in the data, allowing us to visualize it in 2D or 3D space.
- Techniques for Visualization:
  - Principal Component Analysis (PCA)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
  - Uniform Manifold Approximation and Projection (UMAP)

Importance of Visualization

- Insight into Data:

  Helps in understanding the underlying structure of the data.

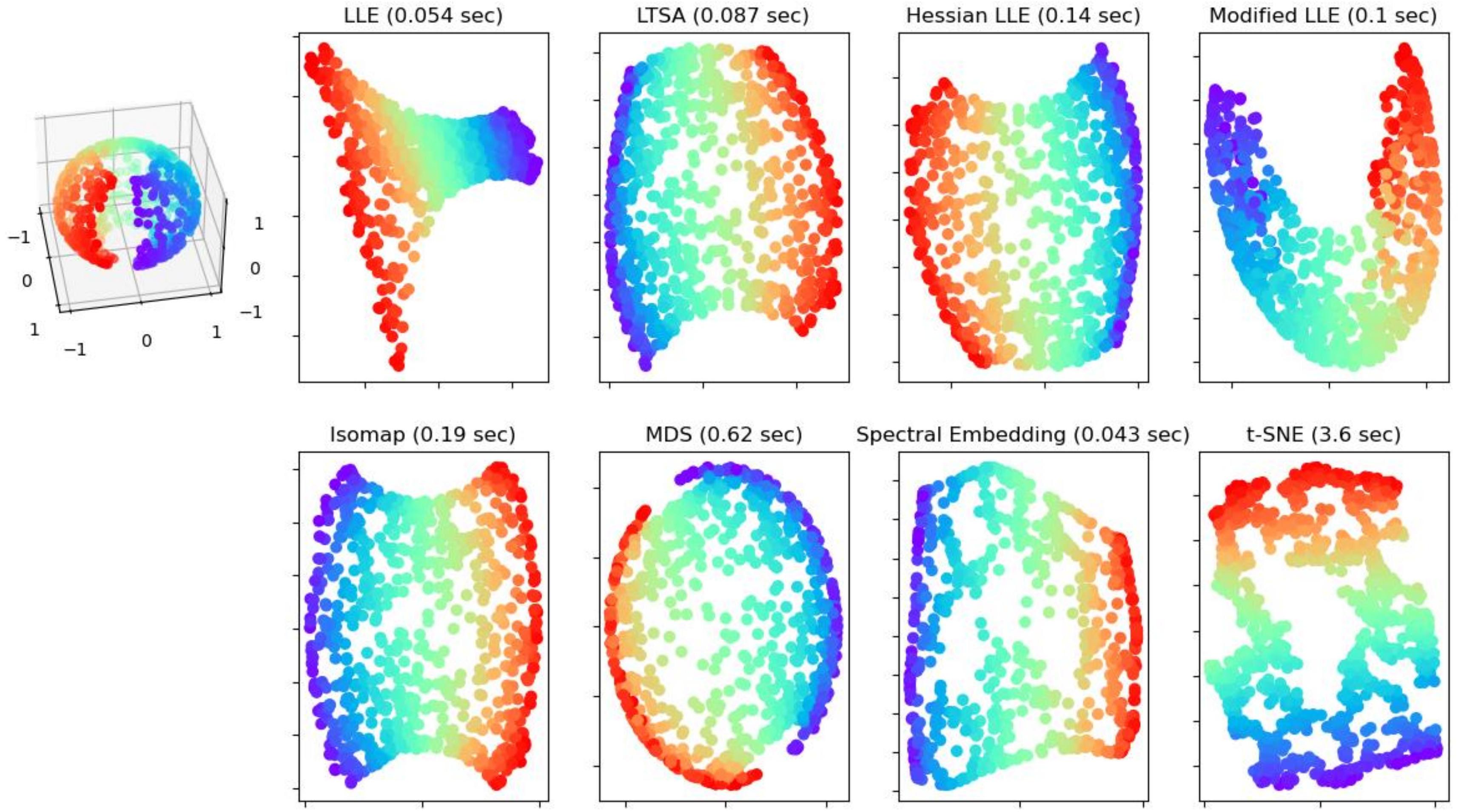
- Cluster Identification:

  Easier to identify clusters or groups within the data.

- Outlier Detection:

  Helps in spotting anomalies or outliers in the dataset.

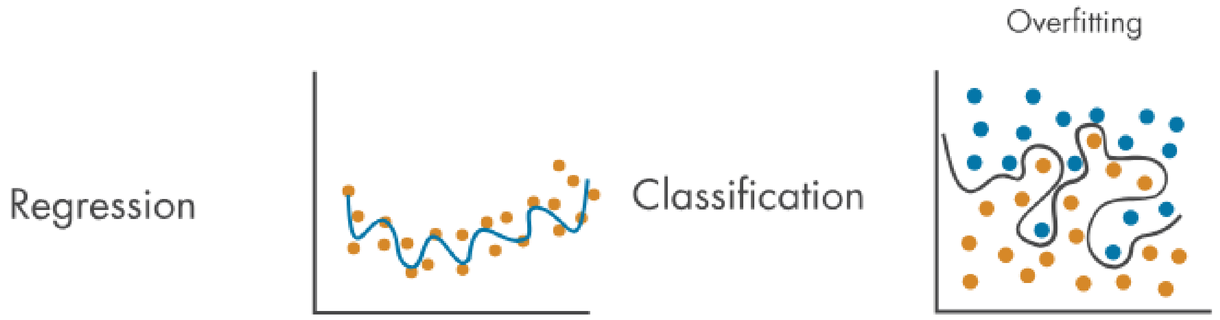
Example of Visualization



Source: https://scikit-learn.org/stable/auto_examples/manifold/plot_manifold_sphere.html#sphx-glr-auto-examples-manifold-plot-manifold-sphere-py

Overfitting

- Overfitting is a common problem where a model learns the training data too well, including its noise and outliers, but performs poorly on unseen or validation data.



Source: https://www.mathworks.com/discovery/overfitting.html

Indicators of Overfitting

- High Training Accuracy, Low Validation/Test Accuracy:

  This is the most straightforward indicator.
- Learning Curves Divergence:

  If training and validation loss diverge significantly during training, it's a sign of overfitting.
- Complexity Analysis:

  A model with excessive complexity relative to the simplicity of the problem may overfit.

Activation Function

- An activation function takes the output signal from a neuron in a neural network and transforms it into a form that can serve as the input to the next layer.
- Mathematically, for a given neuron with input x and output y, the activation function f can be defined as:

$$y = f(w \cdot x + b)$$

where w is the weight and b is the bias associated with the neuron.

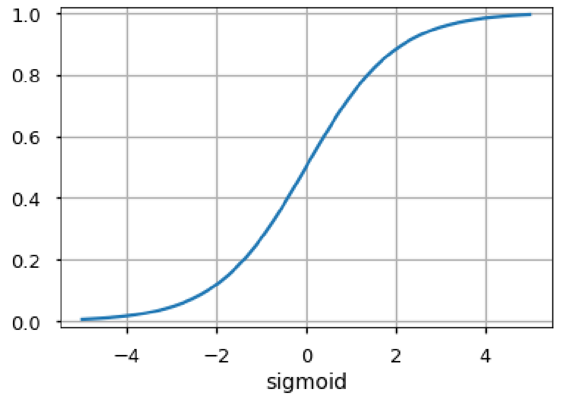
Choice of Activation Function (1/2)

$$\text{Leaky ReLU}(x) = \begin{cases} \frac{x}{a}, if\ x < 0 \\ x, if\ x \geq 0 \end{cases}$$

- Hidden Layers:
  - ReLU is the most commonly used activation function because of its simplicity and efficiency.
  - Leaky ReLU or Parametric ReLU is used to address the “dying ReLU” problem.
  - tanh or sigmoid functions are less commonly used now but are still applicable for certain types of problems.

$$\text{Parametric ReLU}(x) = \begin{cases} ax, if\ x < 0 \\ x, if\ x \geq 0 \end{cases}$$

Choice of Activation Function (2/2)



- Output Layers:
  - For binary classification, a sigmoid function is used.
  - For multi-class classification, the softmax function is suitable.
  - For regression problems, identity activation function is used, or a linear activation function can be applied.

$$softmax(x) = \frac{e^{x}}{\sum_{i=1}^{K} e^{x_i}}$$

Forward and Backward Pass (1/2)

- In the forward pass, the input data passes through the network, layer by layer, until it reaches the output layer.
- The backward pass is the part of backpropagation where gradients are computed, layer by layer, starting from the output layer and going back to the input layer.

Forward and Backward Pass (2/2)

- The forward pass produces an output that is compared to the target to compute a loss.
- The backward pass uses this loss to calculate gradients, which tell us how to adjust the weights to minimize the loss.

Role of Forward Pass

- Primarily used for making predictions.
- Provides the information needed to calculate the loss, which is essential for training the model.

Role of Backward Pass

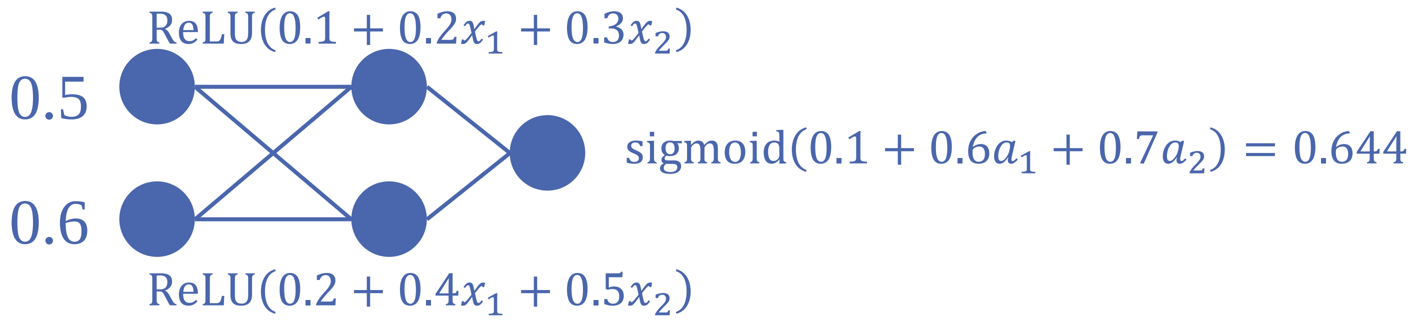
- Computes how much each weight contributed to the error.
- This information (gradients) is used to <u>update the weights </u>in the optimization step.

Example

- Let's consider a simple neural network for binary classification with:
  - Input layer with 2 neurons
  - One hidden layer with 2 neurons (ReLU activation)
  - Output layer with 1 neuron (sigmoid activation)

$$\text{ReLU}(x) = \begin{cases} 0, if\ x < 0 \\ x, if\ x \geq 0 \end{cases}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$


$\text{ReLU}(0.1 + 0.2x_1 + 0.3x_2)$
0.5
0.6
$\text{sigmoid}(0.1 + 0.6a_1 + 0.7a_2) = 0.644$
$\text{ReLU}(0.2 + 0.4x_1 + 0.5x_2)$

Training a Deep Feedforward Neural Network

Step 1 – Data Collection and Preprocessing

Step 2 – Initialize Weights and Biases

Step 3 – Forward Propagation

Step 4 – Compute Loss

Step 5 – Backpropagation

Step 6 – Update Weights and Biases (Optimization)

Optimization and Regularization

- Introduction
- Regularization Techniques
- Practical Guidelines

Optimization

- Optimization is the overarching goal in training a neural network.
- It refers to the process of adjusting the model parameters (usually weights and biases) to minimize the loss function.
- We need optimization to find the “best” set of parameters that make our model as accurate as possible, given the training data.
- The “best” set usually refers to the minimum of the loss function.

Gradient Decent (1/2)

- Gradient Descent is a specific optimization algorithm used for finding the minimum of a function.
- Gradient Descent is an efficient way to navigate the high-dimensional loss landscape and find a local or global minimum.
- Different variants of gradient descent (like Stochastic Gradient Descent, Mini-batch Gradient Descent, and their adaptations like Adam, RMSprop, etc.) offer trade-offs between computational efficiency and the quality of the solution.

Gradient Decent (2/2)

- Step 1. Compute the derivatives of the loss (L) with respect to the parameters ($\phi = [\phi_0, \phi_1, \dots, \phi_N]^T$):

$$\frac{\partial L}{\partial \phi} = [\frac{\partial L}{\partial \phi_0}, \frac{\partial L}{\partial \phi_1}, \dots, \frac{\partial L}{\partial \phi_N}]^T$$

- Step 2. Update the parameters according to the rule:

$$\phi \leftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi}$$

where the positive scalar α determines the magnitude of the change.

Learning Rate

- The learning rate is a hyperparameter in neural networks that controls how much the model’s weights and biases should be updated during training.
- It’s a small positive scalar value used to scale the gradient in the update equation.

Why Use a Learning Rate (1/4)

- Convergence Control:
  - The learning rate helps manage how quickly or slowly a neural network learns.
  - A high learning rate can cause the model to converge too quickly and overshoot the minimum cost, while a low learning rate may cause the model to learn too slowly, consuming more resources.

Why Use a Learning Rate (2/4)

- Stability and Accuracy:
  - A well-tuned learning rate ensures that the model reaches a minimum loss value that is both low and stable, thereby improving the model's performance on unseen data.

Why Use a Learning Rate (3/4)

- Overcoming Optimization Challenges:
  - In non-convex optimization landscapes common in deep learning, a well-set learning rate can help the model escape local minima or saddle points.

Why Use a Learning Rate (4/4)

- Resource Efficiency:
  - An optimal learning rate can make the training process more computationally efficient by requiring fewer epochs to reach the minimum loss.

Gradient Decent – Example (1/2)

- Given a dataset $\{x_i, y_i | i = 1, 2, \dots, I\}$ containing I input/output pairs.
- Build up a linear model $y = f(x, \phi) = \phi_0 + \phi_1 x$
- Loss function

$$\mathrm{L}(\phi) = \sum_{i=1}^{I} (f(x_i, \phi) - y_i)^2 = \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2$$

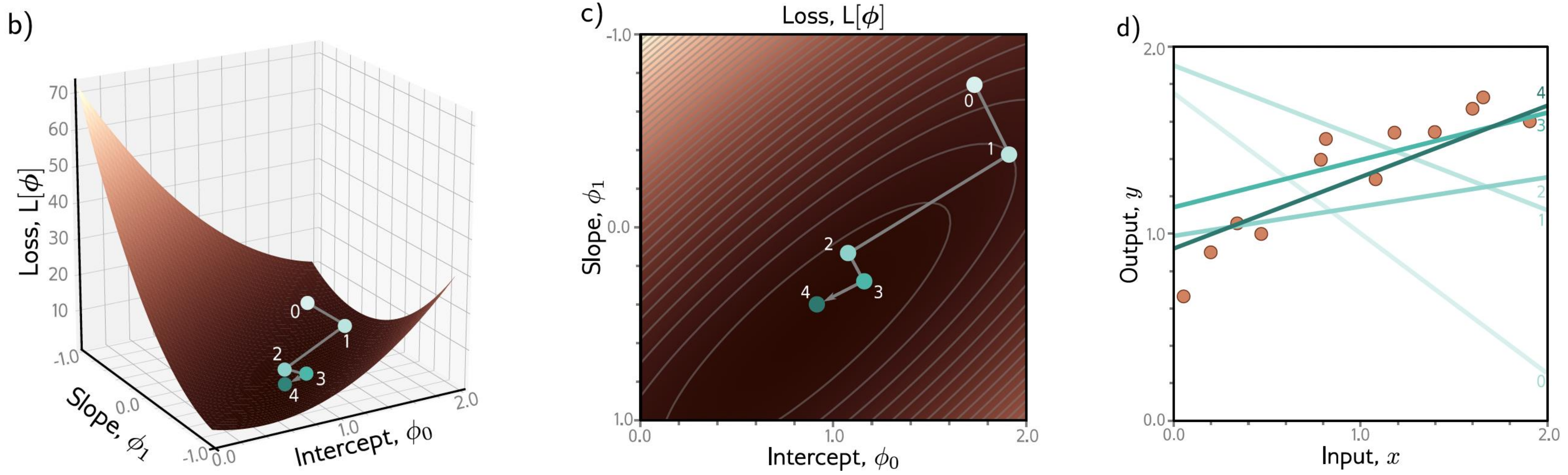
- The derivative of the loss function

$$\frac{\partial L}{\partial \phi} = \frac{\partial}{\partial \phi} \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} \frac{\partial \ell_i}{\partial \phi}$$

$$\frac{\partial \ell_i}{\partial \phi} = [\frac{\partial \ell_i}{\partial \phi_0} \quad \frac{\partial \ell_i}{\partial \phi_1}]^T = [2(\phi_0 + \phi_1 x_i - y_i) \quad 2x_i(\phi_0 + \phi_1 x_i - y_i)]^T$$

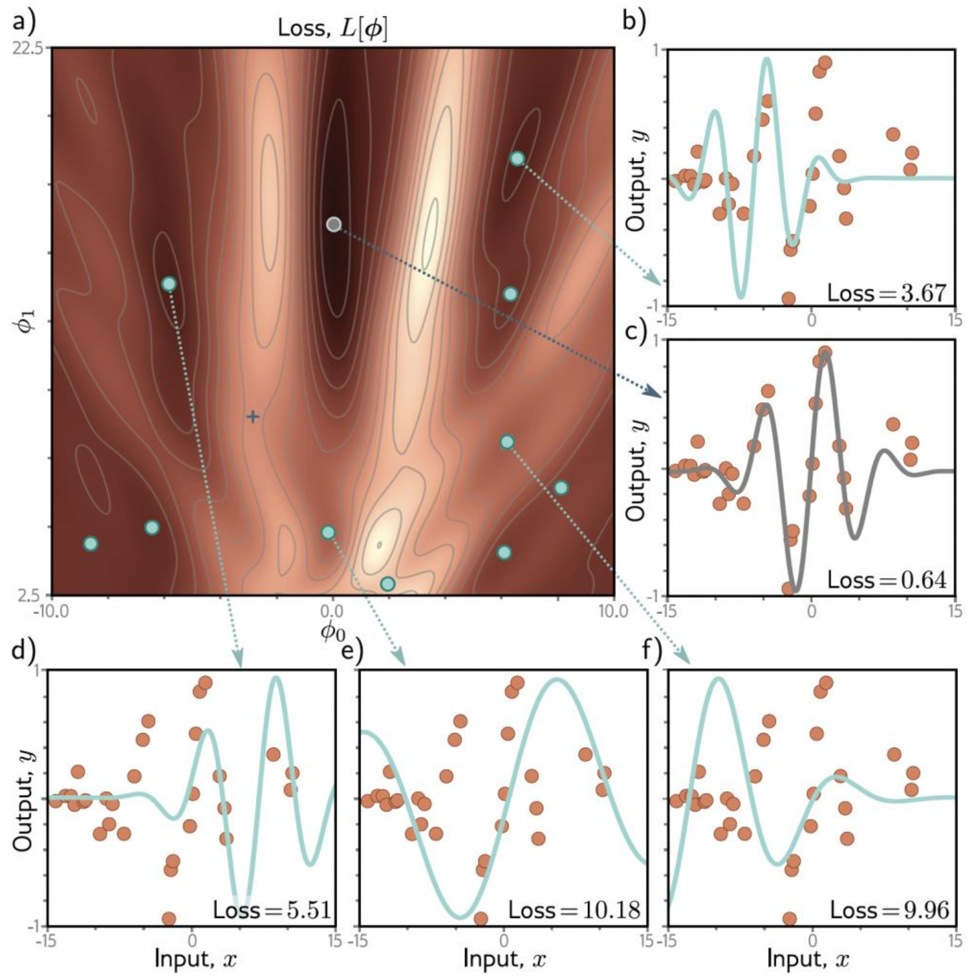
Gradient Decent – Example (2/2)

$$[\phi_0, \phi_1]^T \leftarrow [\phi_0, \phi_1]^T - \alpha \cdot \sum_{i=1}^{I} [2(\phi_0 + \phi_1 x_i - y_i) \quad 2x_i(\phi_0 + \phi_1 x_i - y_i)]^T$$



Source: Figure 6.1 in Understanding Deep Learning by Simon J.D. Prince.

Local Minima and Saddle Points



- a) The loss function is non-convex, with multiple local minima (cyan circles) in addition to the global minimum (gray circle).
- It also contains saddle points where the gradient is locally zero, but the function increases in one direction and decreases in the other.
- The blue cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically.
- b–f) Models associated with the different minima. In each case, there is no small change that decreases the loss.
- Panel (c) shows the global minimum, which has a loss of 0.64.

Source: Figure 6.4 in Understanding Deep Learning by Simon J.D. Prince.

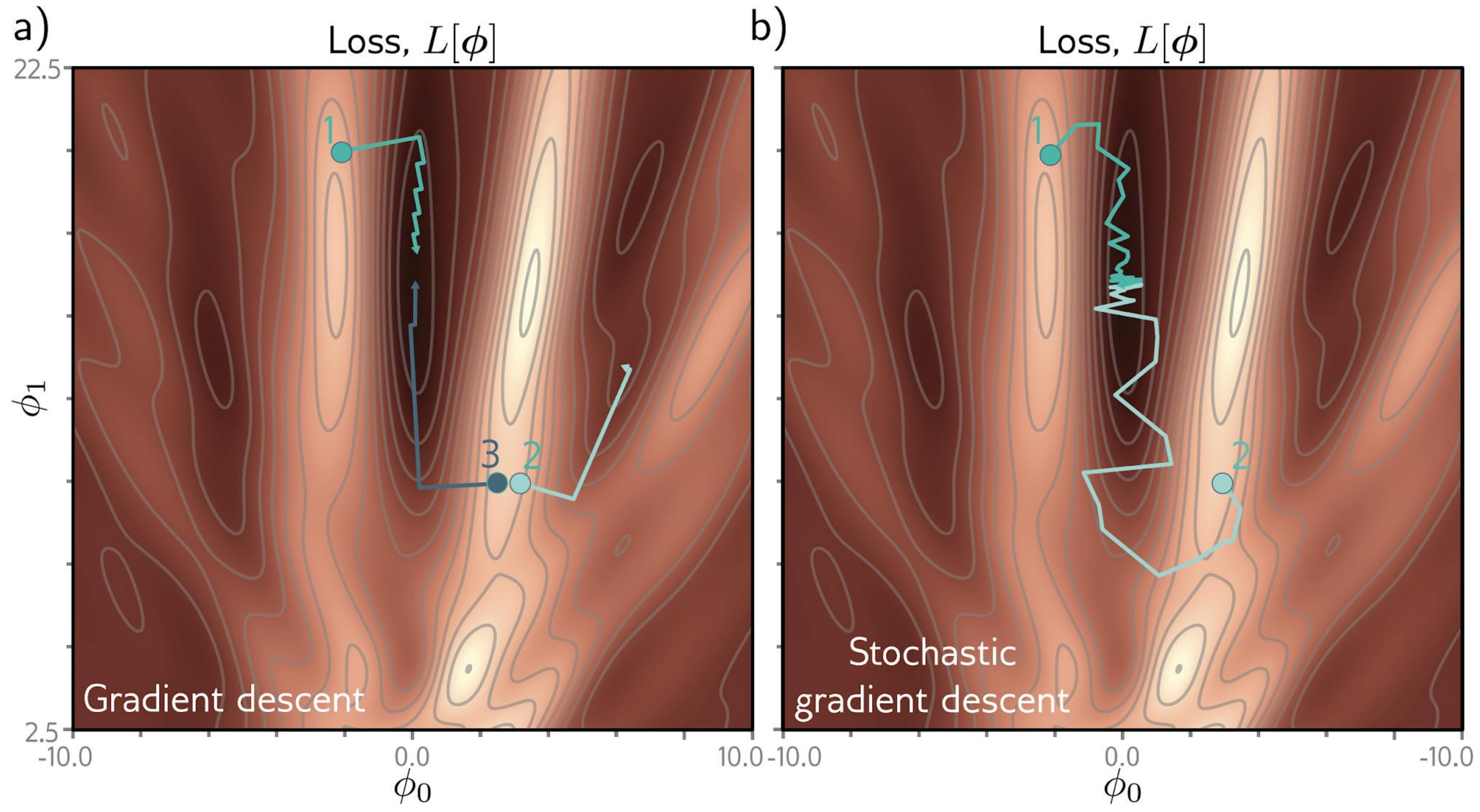
Stochastic Gradient Descent (1/2)

- Stochastic Gradient Descent (SGD) is a variant of the basic Gradient Descent algorithm.
- Instead of using the entire dataset to compute the gradient of the loss function, SGD uses only a single data point chosen randomly at each iteration to update the model parameters (weights and biases).

Stochastic Gradient Descent (2/2)

- For each iteration, a random data point is selected, and the gradient of the loss function with respect to this data point is computed.
- The model parameters are then updated in the opposite direction of this gradient.

Gradient Descent vs. Stochastic Gradient Descent



Source: Figure 6.5 in Understanding Deep Learning by Simon J.D. Prince.

Backpropagation

- Backpropagation is an algorithm used to compute the gradients needed during the gradient descent step.
- Backpropagation provides an efficient way to compute the gradients for a multi-layer neural network by applying the chain rule of calculus.
- Backpropagation calculates the gradients, which are then used by the gradient descent algorithm to update the model parameters, fulfilling the overarching goal of optimization.

Example (1/6)

- Let's consider a simple neural network for binary classification with:
  - Input layer with 2 neurons
  - One hidden layer with 2 neurons (ReLU activation)
  - Output layer with 1 neuron (sigmoid activation)
- Assume our dataset has 4 samples for simplicity:

$$X = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}, y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Example (2/6)

- Initialize weights and bias (intercept):
  - $W_1 = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix}$, $b_1 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
  - $W_2 = \begin{pmatrix} 0.5 \\ 0.6 \end{pmatrix}$, $b_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$
- Hyperparameters:
  - Learning rate (α) = 0.1
  - Batch Size = 2
  - Epochs = 3

Example (3/6)

- Forward Pass (Batch 1: First 2 samples)
  - Input layer: $X_{batch} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$
  - Hidden layer: $z_1 = X_{batch} \times W_1 + b_1$

$$a_1 = ReLU(X_{batch} \times W_1 + b_1) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0.3 & 0.4 \end{pmatrix}$$

  - Output layer: $z_2 = a_1 \times W_2 + b_2$

$$a_2 = sigmoid(a_1 \times W_2 + b_2) = \begin{pmatrix} 0 & 0 \\ 0.3 & 0.4 \end{pmatrix} \times \begin{pmatrix} 0.5 \\ 0.6 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.596 \end{pmatrix}$$

Example (4/6)

- Compute Loss (Batch 1: First 2 samples) $y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
  - $L = -[y log(a_2) + (1 - y) log(1 - a_2)] =$

$$\begin{pmatrix} -[0 \cdot log(0.5) + (1 - 0) \cdot log(1 - 0.5)] \\ -[1 \cdot log(0.596) + (1 - 0.596) \cdot log(1 - 0.596)] \end{pmatrix} = \begin{pmatrix} 0.69 \\ 0.52 \end{pmatrix}$$

$$L = -[y\log(a_2) + (1-y)\log(1-a_2)], \frac{\partial L}{\partial a_2} = -(\frac{y}{a_2} - \frac{1-y}{1-a_2})$$

Example (5/6)

$$a_2 = sigmoid(z_2) = \frac{1}{1+e^{-z_2}}, \frac{\partial a_2}{\partial z_2} = a_2 \times (1-a_2)$$

- Backward Pass

$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} = -\left(\frac{y}{a_2} - \frac{1-y}{1-a_2}\right) \times a_2 \times (1-a_2) = a_2 - y$$

$$\frac{\partial L}{\partial z_2} = \begin{pmatrix} 0.5 \\ 0.596 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.5 \\ -0.404 \end{pmatrix}$$

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} = (a_2 - y) \times W_2^T \times ReLU'(z_1)$$

$$= \begin{pmatrix} 0.5 \\ -0.404 \end{pmatrix} \begin{pmatrix} 0.5 & 0.6 \end{pmatrix} \times ReLU' \begin{pmatrix} 0 & 0 \\ 0.3 & 0.4 \end{pmatrix} = \begin{pmatrix} 0.25 & 0.3 \\ -0.202 & -0.2424 \end{pmatrix}$$

Example (6/6)

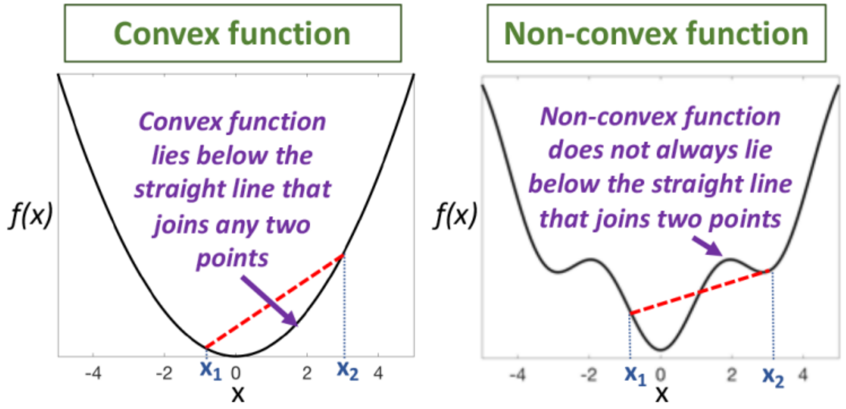
- Weight and Bias Updates (Gradient Descent):

$$W_1^{new} = W_1 - \alpha \times \frac{\partial L}{\partial W_1}, W_2^{new} = W_2 - \alpha \times \frac{\partial L}{\partial W_2}$$

$$b_1^{new} = b_1 - \alpha \times \frac{\partial L}{\partial b_1}, b_2^{new} = b_2 - \alpha \times \frac{\partial L}{\partial b_2}$$

Source: https://qiml.radiology.wisc.edu/wp-content/uploads/sites/760/2021/03/lecture_A1_04_Convexity1.pdf

Challenges in Optimization



- Local Minima:
  - For non-convex cost functions, there's a risk of getting stuck in a local minimum rather than finding the global minimum.
- Overfitting:
  - A very low cost on the training set is not always desirable.
  - It might indicate that the network has memorized the training data and will perform poorly on unseen data.
  - This is where regularization techniques come into play.

Regularization Techniques

- Regularization is a technique used to prevent overfitting by adding an additional term to the loss function.
- This term penalizes certain parameter configurations, effectively limiting the capacity of the model.
- The regularized loss function L' is then:

$$L' = L + \lambda \times \text{Regularization term}$$

Here, λ is the regularization coefficient, and L is the original loss.

Regularization Coefficient

- The regularization coefficient, often denoted by λ, is a hyperparameter that controls the strength of the regularization term in the loss function.
- It balances the trade-off between fitting the training data well and keeping the model parameters small to avoid overfitting.
- A larger λ means stronger regularization, potentially reducing overfitting but at the risk of underfitting if set too high.

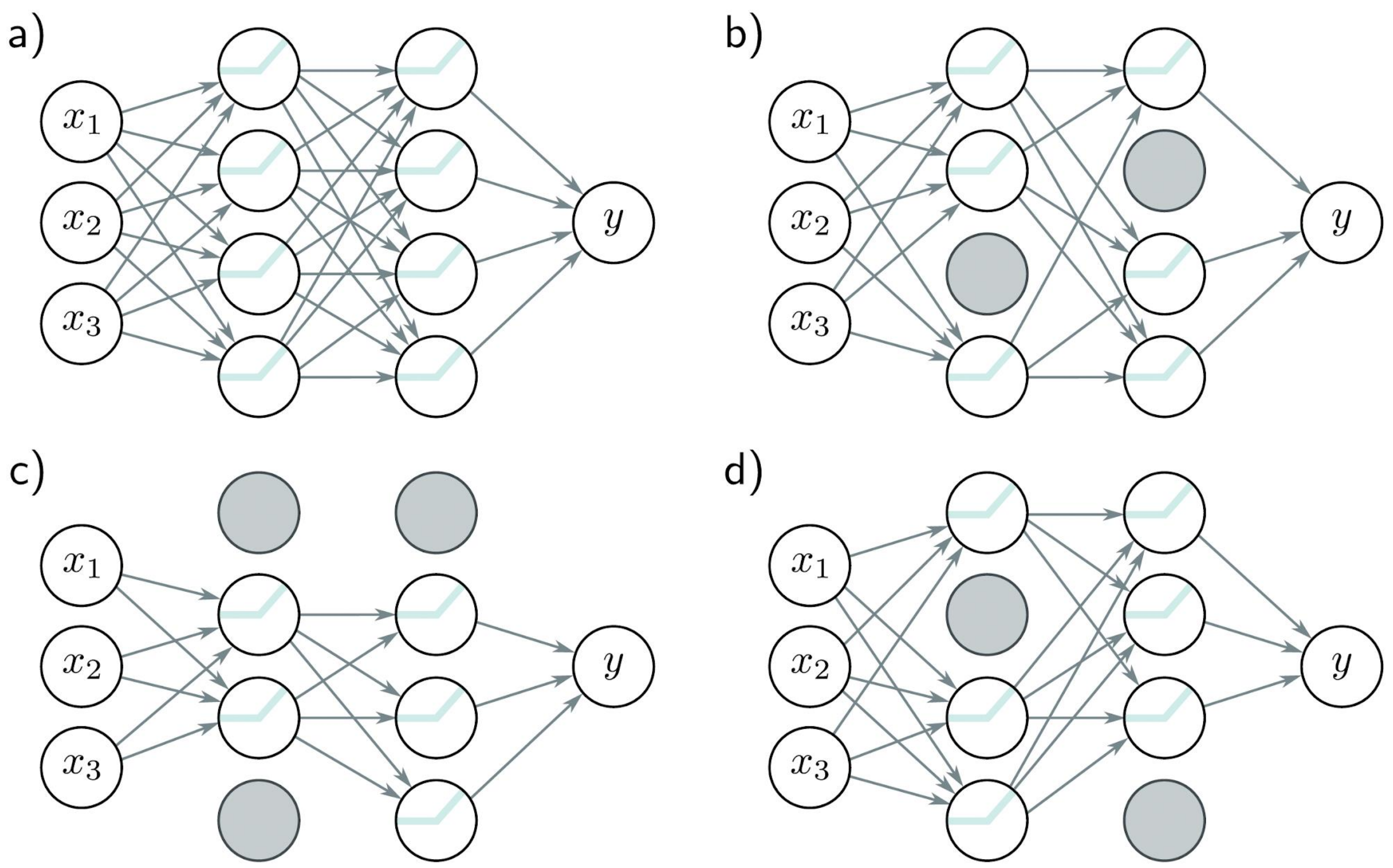
L1 and L2 Regularization

- L1 Regularization:
  - Adds “absolute value of magnitude” of the coefficient as a penalty term to the loss function.
  - $L' = L + \lambda \sum_{i=1}^{n} |w_i|$
- L2 Regularization:
  - Adds “squared magnitude” of the coefficient as a penalty term to the loss function.
  - $L' = L + \lambda \sum_{i=1}^{n} {w_i}^2$

Dropout (1/2)

- Dropout is a technique where randomly selected neurons are ignored during training, effectively dropping out during the forward and backward passes.
- Use dropout when you notice overfitting in your model.

Dropout (2/2)



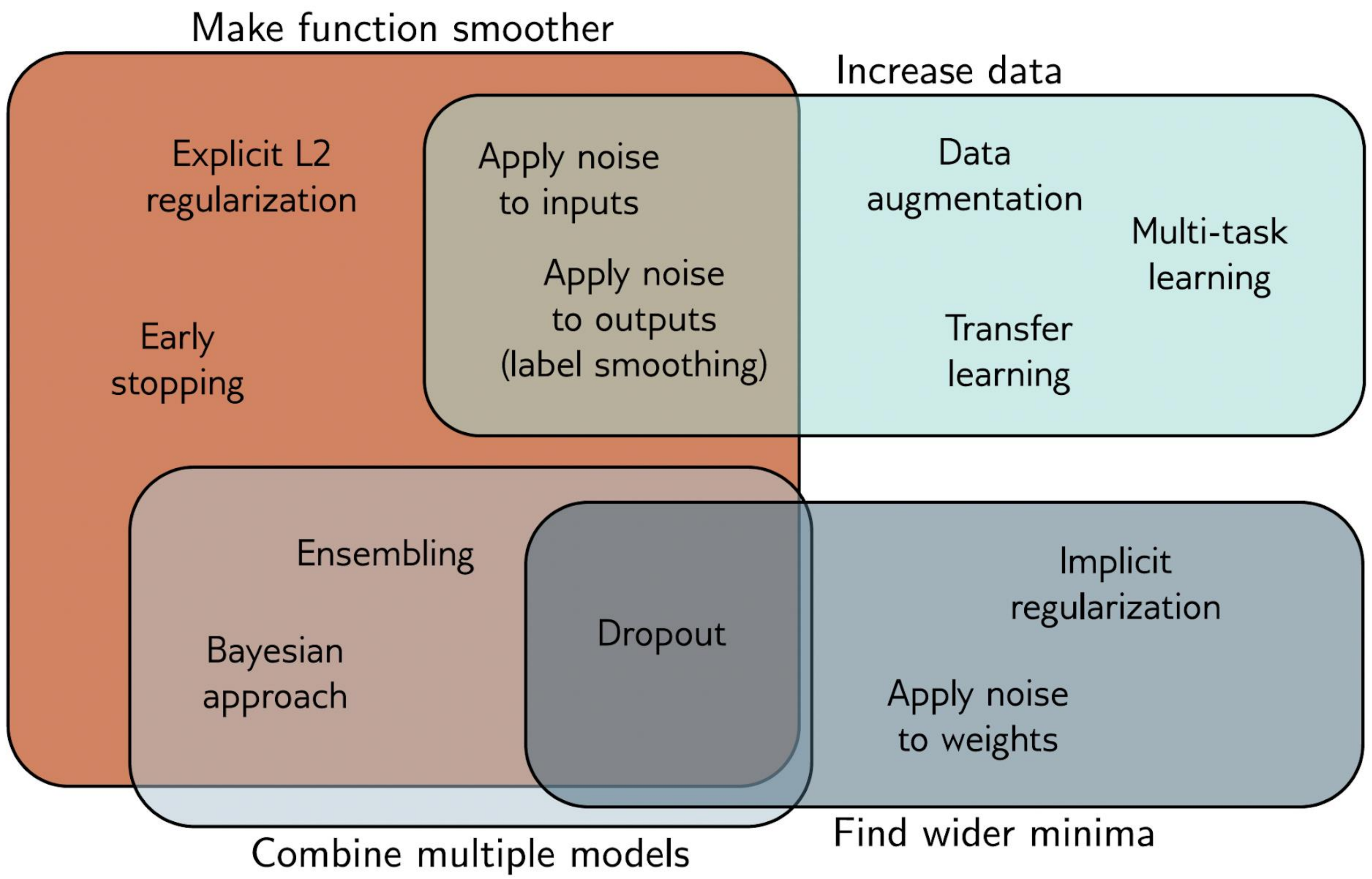
- a) Original network.
- b–d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes).
- The result is that the incoming and outgoing weights from these units have no effect, so we are training with a slightly different network each time

Source: Figure 9.8 in Understanding Deep Learning by Simon J.D. Prince.

Early Stopping

- Early stopping involves halting the training process when the model's performance stops improving on a held-out validation dataset.
- It's a simple and effective technique to prevent overfitting by stopping the training process when the validation error increases, while the training error is still decreasing.

Regularization Methods



Source: Figure 9.14 in Understanding Deep Learning by Simon J.D. Prince.

Hyperparameter Tuning (1/2)

- What are Hyperparameters?
  - Hyperparameters are parameters that are not learned from the data but must be set prior to the learning process.
  - Examples include the learning rate, regularization strength, and architecture settings like the number of layers or hidden units in each layer.

Hyperparameter Tuning (2/2)

- Tuning Methods
  - **Grid Search:** Useful when you have a limited number of hyperparameters and possible values. However, it's computationally expensive as it checks every possible combination.
  - **Random Search:** More efficient than grid search and often as effective. Randomly samples the hyperparameter space a fixed number of times.
  - **Bayesian Optimization:** A probabilistic model-based optimization technique. More efficient than random or grid search but can be more complex to set up.

Guidelines for Hyperparameter Tuning

- Start Small: Begin with a small subset of data and a simpler model to check the code and pipeline.
- Range over Orders of Magnitude: For parameters like learning rate, search in ranges like [0.001, 0.01, 0.1, 1].
- Use Coarse-to-Fine Strategy: Start with a broad range and then narrow it down.
- Parallelize: If possible, run multiple experiments in parallel to save time.
- Track Results: Use tools like TensorBoard, MLflow, or custom logging to track the experiments.
- Revalidate: After finding the optimal hyperparameters, retrain the model on the entire dataset.

Batch vs Mini-batch vs Stochastic

- Batch Gradient Descent:

  Uses all training samples for each update.

- Mini-batch Gradient Descent:

  Uses a subset of training samples for each update.

- Stochastic Gradient Descent:

  Uses a single training sample for each update.

Learning Rate

- Constant Learning Rate

  The learning rate remains the same throughout training.

- Decaying Learning Rate

  The learning rate decreases during training.

- Adaptive Learning Rate

  The learning rate changes based on the performance on the validation set.

Practical Guidelines (1/6)

- Determining the Number of Neurons and Layers
  - The number of neurons in the hidden layer should be between the size of the input layer and the size of the output layer.
  - If your model underfits, consider increasing the number of neurons or adding more layers.

Practical Guidelines (2/6)

- Choosing Activation Functions
  - Use ReLU (Rectified Linear Unit) as the default activation function for hidden layers.
  - For the output layer, use softmax for multi-class classification and sigmoid for binary classification.

Practical Guidelines (3/6)

- Optimization Choices
  - SGD (Stochastic Gradient Descent): Good for large datasets.
  - Adam: Good for quick convergence; generally a safe bet.

Practical Guidelines (4/6)

- Learning Rate
  - Start with 0.01 or 0.001; use learning rate decay or adaptive learning rates for better results.

Practical Guidelines (5/6)

- Smaller batch sizes (32, 64) offer a regularizing effect and lower generalization error.
- Start with a smaller number of epochs, such as 10 or 20, and utilize early stopping to avoid overfitting.
- Use dropout layers in between fully connected layers to prevent overfitting. A dropout rate of 0.5 is a good starting point.

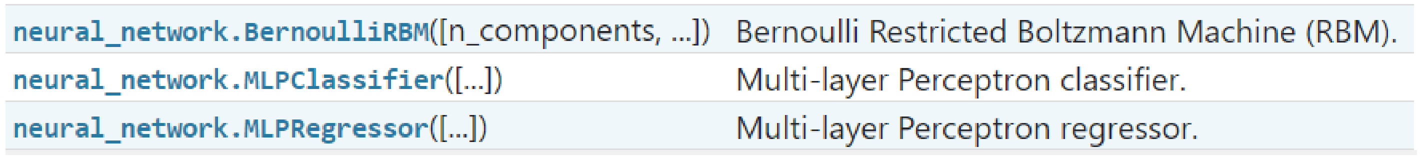
Practical Guidelines (6/6)

- Monitor your validation loss and stop training once the loss starts to increase.
- Use grid search when you have a small set of hyperparameters.
- Use random search when the hyperparameter space is large.
- For automated hyperparameter tuning, consider using tools like AutoML or Hyperopt.

Hand-on Feedforward Networks

- Implementing a Feedforward Network in Sklearn
- Implementing a Feedforward Network in Keras
- Implementing a Feedforward Network in Pytorch

Implementing a Feedforward Network in Sklearn



`neural_network.BernoulliRBM`([n_components, ...])	Bernoulli Restricted Boltzmann Machine (RBM).
`neural_network.MLPClassifier`([...])	Multi-layer Perceptron classifier.
`neural_network.MLPRegressor`([...])	Multi-layer Perceptron regressor.

Source: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.neural_network

- Initialize the Model

```
from sklearn.neural_network import MLPClassifier
mlp_classifier = MLPClassifier(hidden_layer_sizes=(64,
64), max_iter=200, alpha=0.0001, solver='adam',
random_state=42)
```

- Train the Model

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

- Make Predictions

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

Implementing a Feedforward Network in Keras (1/3)

- Initialize the Model

```
from keras.models import Sequential

model = Sequential()
```

- Adding Layers

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))

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

Source: https://keras.io/guides/sequential_model/

Implementing a Feedforward Network in Keras (2/3)

- Compilation

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

- Model Training

```
# x_train and y_train should be numpy arrays
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Source: https://keras.io/guides/sequential_model/

Regularization in Keras (1/2)

- L1 Regularization

```
from keras.regularizers import l1
model.add(Dense(64, activation='relu',
kernel_regularizer=l1(0.01)))
```

- Dropout in Keras

```
from keras.layers import Dropout
model.add(Dropout(0.5))
```

Source: https://keras.io/guides/sequential_model/

Regularization in Keras (1/2)

- L1 Regularization

```
from keras.regularizers import l1
model.add(Dense(64, activation='relu',
kernel_regularizer=l1(0.01)))
```

- L2 Regularization

```
from keras.regularizers import l2
model.add(Dense(50, activation='relu',
input_shape=(10,), kernel_regularizer=l2(0.001)))
```

Source: https://keras.io/api/layers/regularizers/

Regularization in Keras (2/2)

- Dropout

```
from keras.layers import Dropout
model = Sequential()
model.add(Dense(50, activation='relu`,
input_shape=(10,)))
model.add(Dropout(0.5))
model.add(Dense(1, activation='linear'))
```

Implementing a Feedforward Network in Pytorch (1/4)

- Initialize the Model

```
import torch
import torch.nn as nn
import torch.optim as optim
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(100, 64)
        self.fc2 = nn.Linear(64, 10)
```

Source: https://pytorch.org/docs/stable/optim.html

Implementing a Feedforward Network in Pytorch (2/4)

- Compilation

```
criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.01)
```

Source: https://pytorch.org/docs/stable/optim.html

Implementing a Feedforward Network in Pytorch (3/4)

- Model Training

```
for epoch in range(5):  # loop over the dataset multiple times
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Source: https://pytorch.org/docs/stable/optim.html

Implementing a Feedforward Network in Pytorch (4/4)

- Model Evaluation

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

Source: https://pytorch.org/docs/stable/optim.html

Regularization in Pytorch (1/2)

- L2 Regularization in PyTorch

```
import torch.nn as nn
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(100, 64)
        self.fc2 = nn.Linear(64, 10)

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, weight_decay=0.01)  # L2
regularization
```

Source: https://pytorch.org/docs/stable/optim.html

Regularization in Pytorch (1/2)

- Dropout

```
class SimpleModelWithDropout(nn.Module):
    def __init__(self):
        super(SimpleModelWithDropout, self).__init__()
        self.fc1 = nn.Linear(10, 50)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(50, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Source: https://pytorch.org/docs/stable/optim.html

A Neural Network Playground

https://www.youtube.com/watch?v=P1Exq5M22gA

Thank you