CE6146
Introduction to Deep Learning
Feedforward Neural Networks

Chia-Ru Chung
Department of Computer Science and Information Engineering
National Central University

2023/9/28

20230921 Exercise

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

Outline

- Review
- Feedforward Networks
- Hand-on Feedforward Networks

Review

- Types of Machine Learning
- Cross-Validation
- Optimization and Hyperparameter
- Bias-Variance Tradeoff

Criteria	Supervised Learning	Unsupervised Learning	Reinforcement Learning
Data	Labeled	Unlabeled	States, Actions, Rewards
Goal	Predict labels or values	Discover hidden structure	Optimize long-term reward
Examples	Classification, Regression	Clustering, Dimensionality Reduction	Game playing, Robotics
Training Objective	Minimize loss function	Maximize likelihood or other criteria	Maximize expected reward
Feedback	Immediate and direct	None	Delayed
Evaluation Metrics	Accuracy, F1-Score, RMSE	Silhouette Score, Davies-Bouldin Index	Cumulative reward

Criteria	Supervised Learning	Unsupervised Learning	Reinforcement Learning
Real-world Applications	Spam detection, Image recognition	Market segmentation, Anomaly detection	Self-driving cars, Game AI
Algorithms	SVM, Random Forest, Neural Networks	K-means, PCA, t-SNE	Q-Learning, DQN, Policy Gradients
Pros	Direct feedback, easier to measure performance	Works with any kind of data	Can work in complex, poorly-defined environments
Cons	Requires labeled data, can be expensive	Harder to evaluate, may require domain expertise	Requires a lot of data, susceptible to local minima

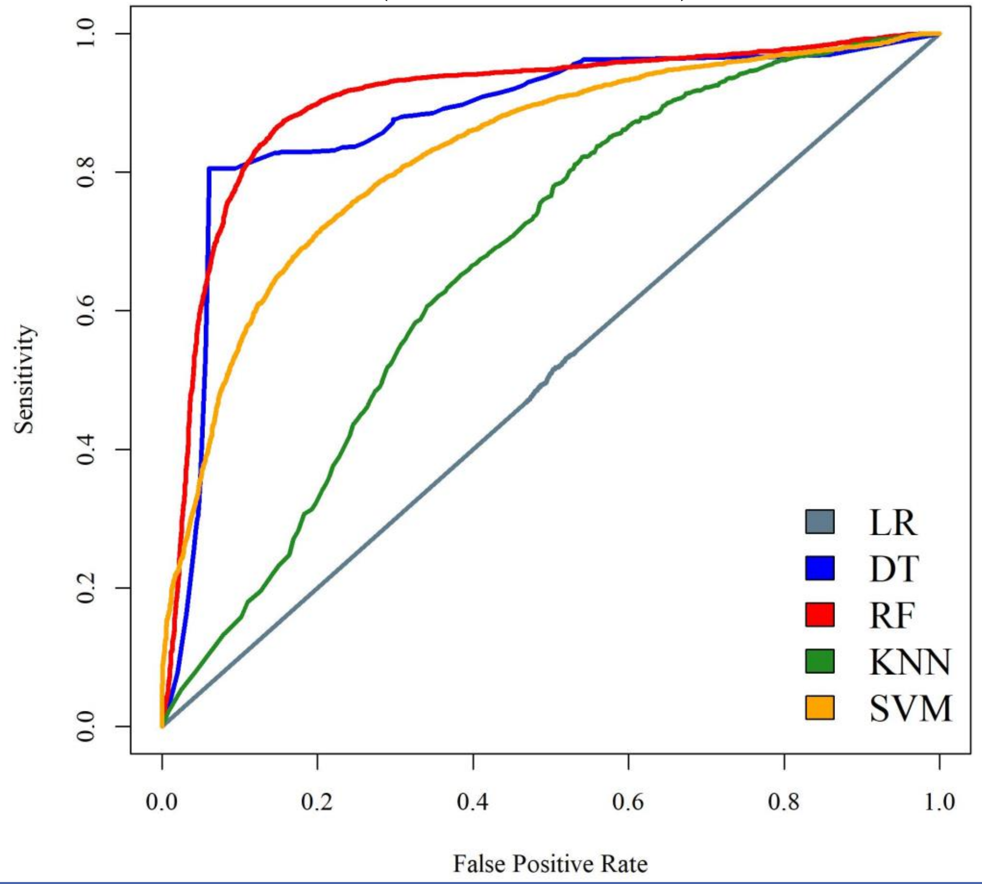
Evaluations for Classification (1/2)

- Sensitivity (hit rate or recall) = $\frac{TP}{P} = \frac{TP}{TP+FN}$
- Specificity = $\frac{TN}{N} = \frac{TN}{TN+FP}$
- 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) = $\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

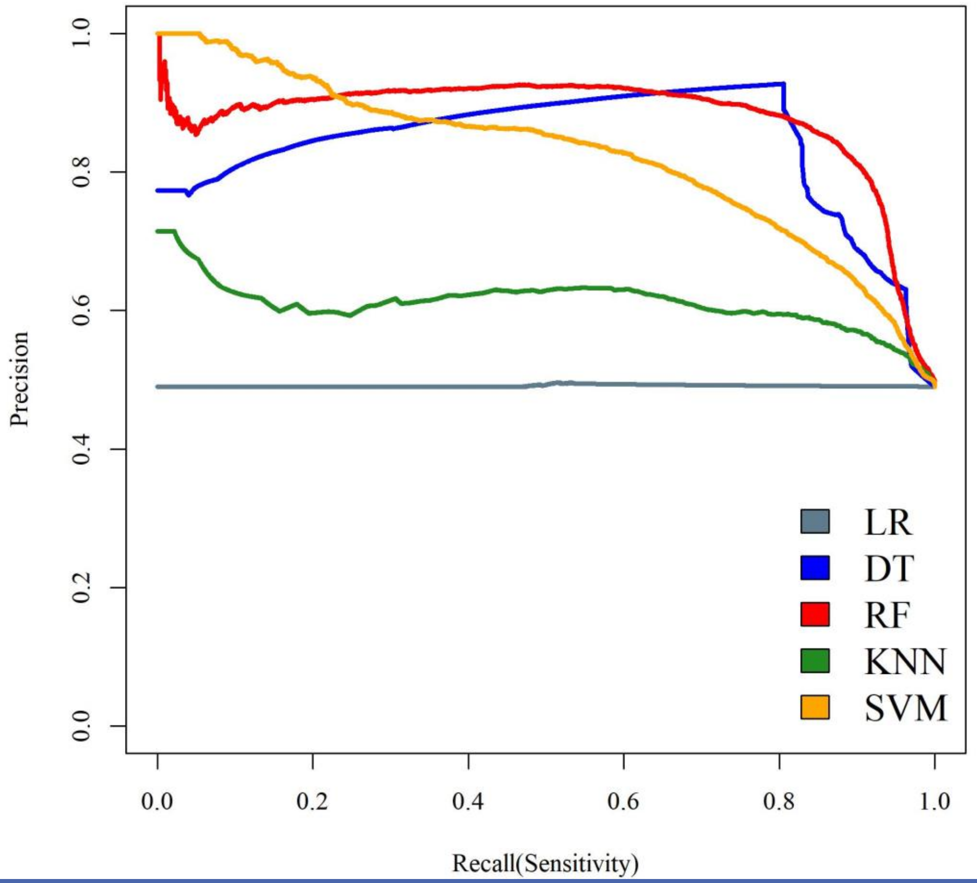
True Condition Status	Test Result		
	Positive	Negative	
Disease	True Positive (TP)	False Negative (FN)	P
Health	False Positive (FP)	True Negative (TN)	N

Evaluations for Classification (2/2)

Receiver operating characteristic curve
(ROC curve)



Precision-Recall curve
(PR curve)



Evaluations for Regression

- Mean Absolute Error (MAE) = $\frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y_i}|$
- Mean Squared Error (MSE) = $\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2$
- Root Mean Squared Error (RMSE) = $\sqrt{MSE}$
- R-Squared (R^2) = $1 - \frac{\sum_{i=1}^{n}(y_i - \widehat{y_i})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$
- Mean Absolute Percentage Error (MAPE) = $\frac{1}{n}\sum_{i=1}^{n}|\frac{y_i - \widehat{y_i}}{y_i}|\times 100$

What is Cross-Validation

- Cross-validation is a technique used in machine learning to assess the generalizability of a model.
- It helps ensure that the performance of a model is not dependent on the way data is split during training and testing.
- It provides a robust way to estimate the performance of a model on an independent dataset and to check for overfitting.

How Does It Work

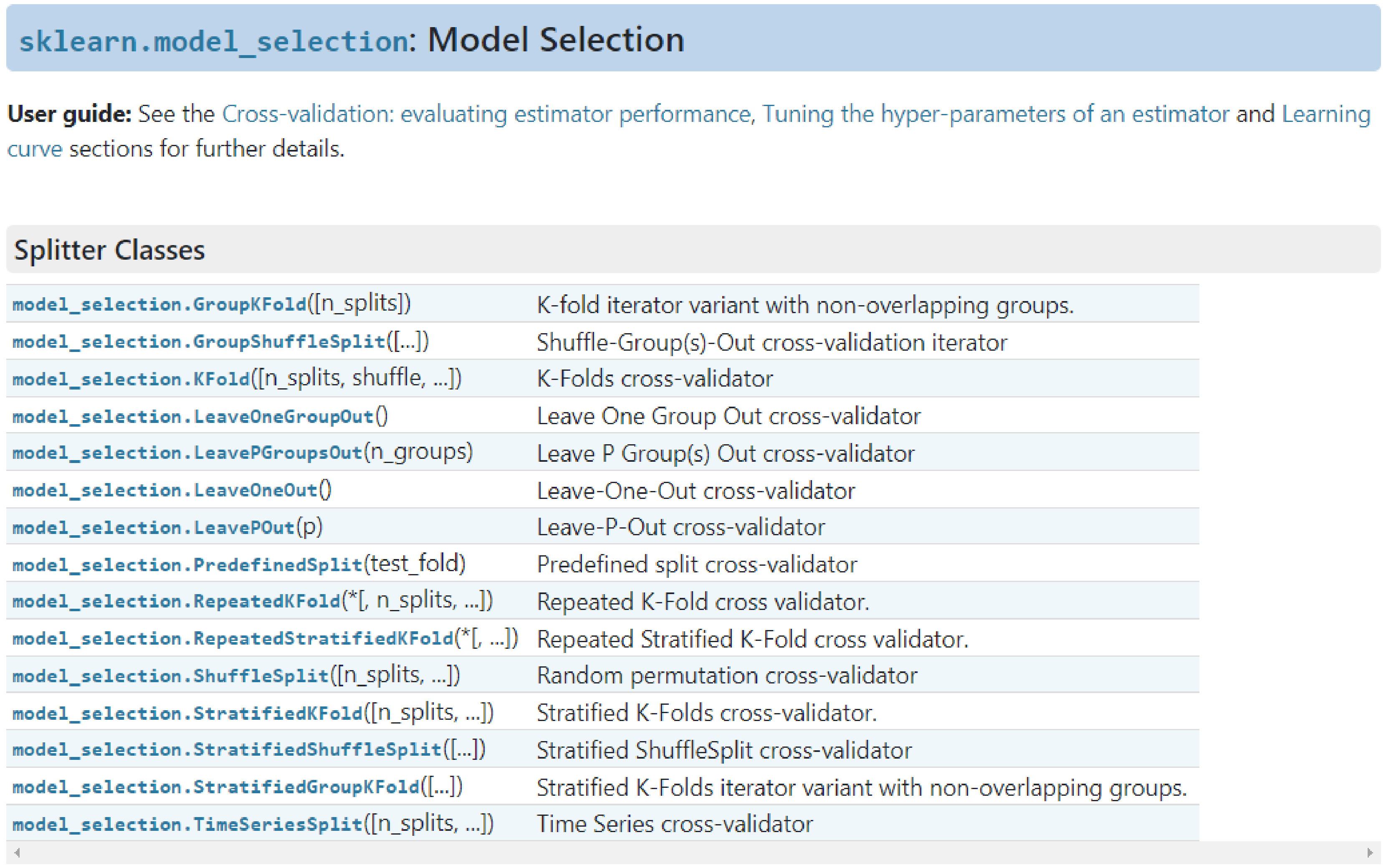
1. Divide the Dataset: The dataset is divided into k subsets, or "folds."
2. Train and Test: The model is trained on $k-1$ of these folds and tested on the remaining one.
3. Repeat: Steps 1 and 2 are repeated k times, each time with a different fold as the test set.
4. Average Score: The k results are averaged to produce a single score.

Types of Cross-Validation (1/2)

- k-Fold Cross-Validation:
  Dataset is divided into k equal-sized folds.
- Stratified k-Fold Cross-Validation:
  Keeps the class distribution in each fold the same as the whole dataset.
- Repeated Cross-Validation:
  Performs k-fold cross-validation multiple times with different random splits and averages the results for a more robust performance estimate.
- Nested Cross-Validation:
  Provides an unbiased performance estimate when hyperparameters also need to be optimized.
- Leave-One-Out Cross-Validation (LOOCV):
  Each data point serves as a single test set, and the rest make up the training set.

  …

Types of Cross-Validation (2/2)



`sklearn.model_selection`: Model Selection

User guide: See the Cross-validation: evaluating estimator performance, Tuning the hyper-parameters of an estimator and Learning curve sections for further details.

Splitter Classes

`model_selection.GroupKFold`([n_splits])	K-fold iterator variant with non-overlapping groups.
`model_selection.GroupShuffleSplit`([...])	Shuffle-Group(s)-Out cross-validation iterator
`model_selection.KFold`([n_splits, shuffle, ...])	K-Folds cross-validator
`model_selection.LeaveOneGroupOut`()	Leave One Group Out cross-validator
`model_selection.LeavePGroupsOut`(n_groups)	Leave P Group(s) Out cross-validator
`model_selection.LeaveOneOut`()	Leave-One-Out cross-validator
`model_selection.LeavePOut`(p)	Leave-P-Out cross-validator
`model_selection.PredefinedSplit`(test_fold)	Predefined split cross-validator
`model_selection.RepeatedKFold`(*[, n_splits, ...])	Repeated K-Fold cross validator.
`model_selection.RepeatedStratifiedKFold`(*[, ...])	Repeated Stratified K-Fold cross validator.
`model_selection.ShuffleSplit`([n_splits, ...])	Random permutation cross-validator
`model_selection.StratifiedKFold`([n_splits, ...])	Stratified K-Folds cross-validator.
`model_selection.StratifiedShuffleSplit`([...])	Stratified ShuffleSplit cross-validator
`model_selection.StratifiedGroupKFold`([...])	Stratified K-Folds iterator variant with non-overlapping groups.
`model_selection.TimeSeriesSplit`([n_splits, ...])	Time Series cross-validator

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

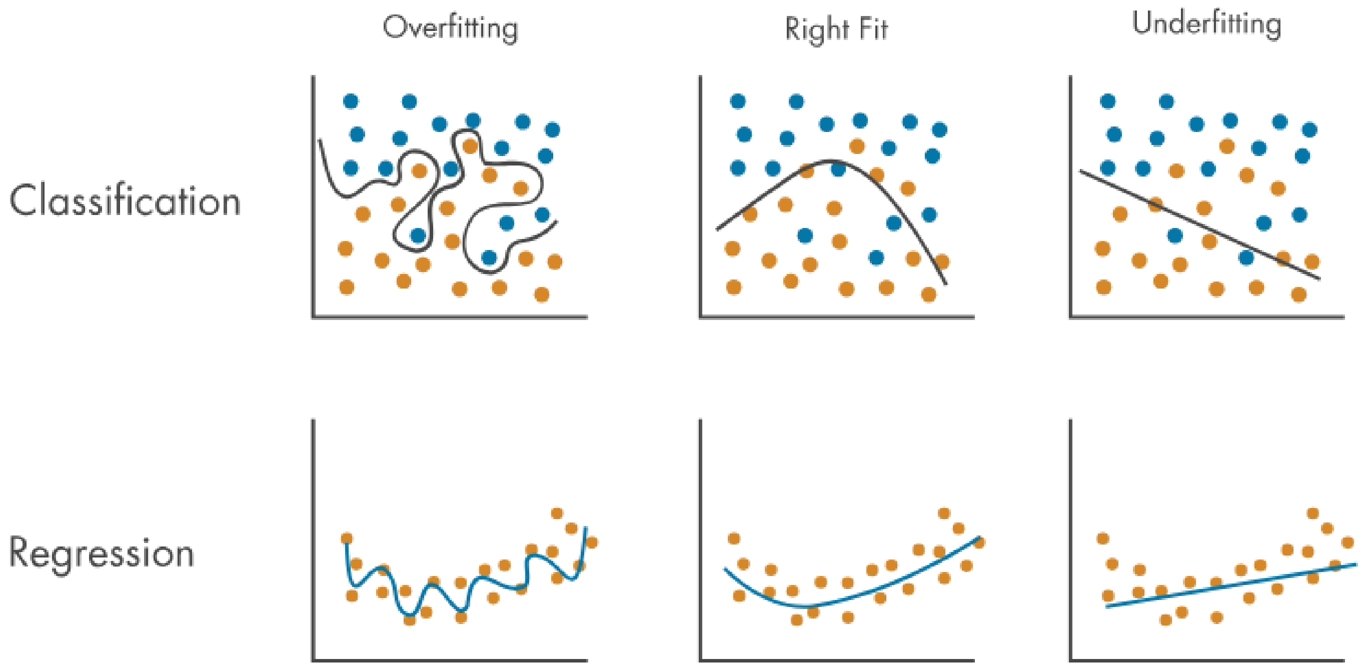
Optimization

- In deep learning, optimization is crucial for training neural networks.
- The objective is to find the set of model parameters that minimize the loss function, which measures how well the neural network performs on the training data.

Hyperparameter

- A hyperparameter is a parameter whose value is set before the learning process begins.
- Examples include the learning rate, the number of hidden layers in a neural network, and the regularization strength.
- They are not learned from the data but are set a priori or tuned using techniques like grid search or random search often with the help of a separate validation set or cross-validation.

Overfitting vs. Underfitting



Error	Overfitting	Right Fit	Underfitting
Training	Low	Low	High
Test	High	Low	High

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

Bias-Variance Tradeoff (1/3)

- The bias-variance tradeoff is the problem of simultaneously minimizing two sources of error that prevent supervised learning algorithms from generalizing beyond their training set.
- Simply put, it's the balance that a model must achieve between fit and flexibility.

Bias-Variance Tradeoff (2/3)

- Low Bias, High Variance:

  A highly complex model (like a deep neural network) will have low bias but high variance. It fits almost all patterns from the data, including noise, which makes it perform poorly on new, unseen data.

- High Bias, Low Variance:

  A less complex model (like linear regression for a non-linear problem) might not capture all the patterns but will be more stable in terms of its prediction on new, unseen data.

Bias-Variance Tradeoff (3/3)

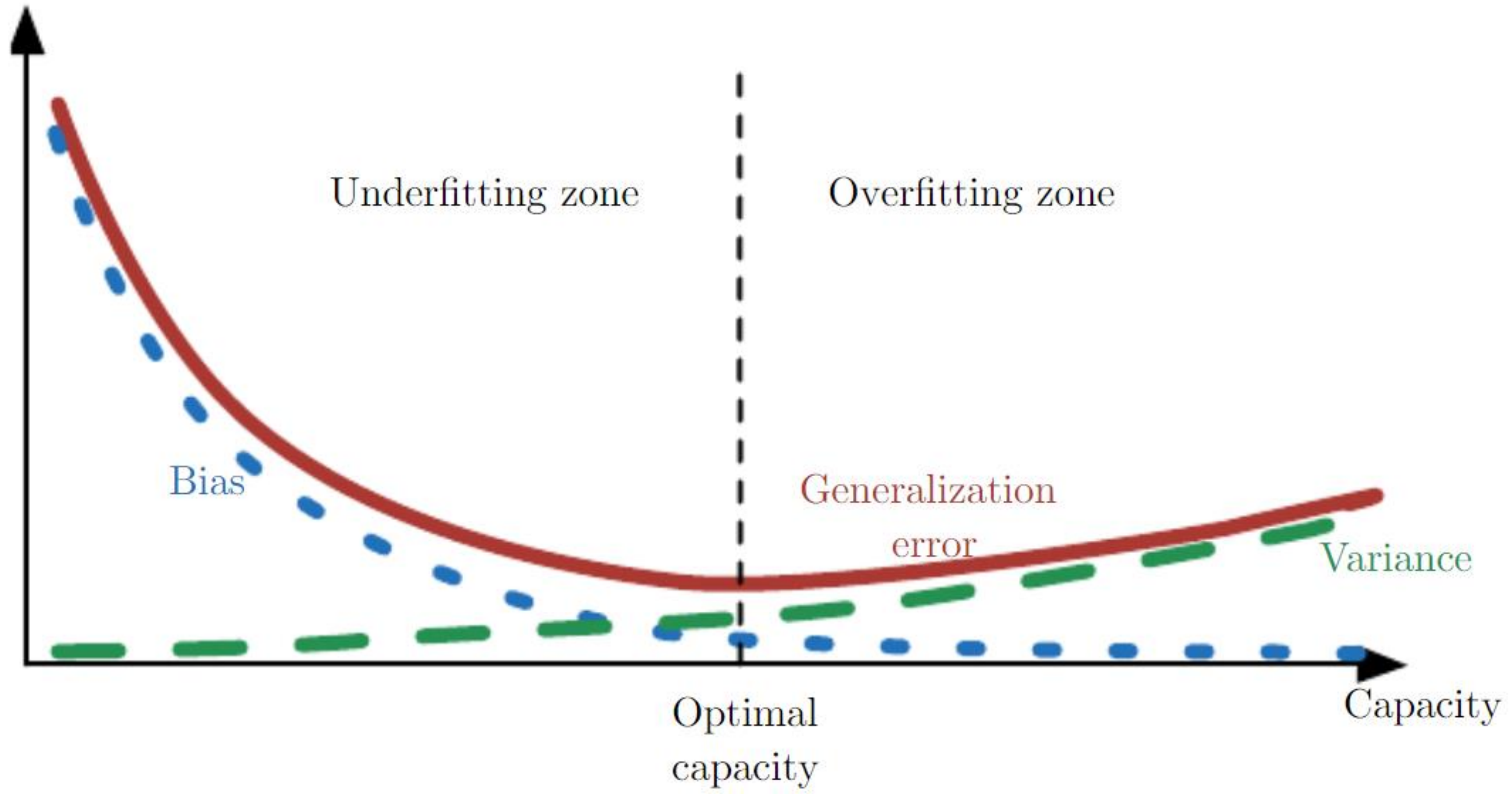


Figure 5.6 in Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

Deep Feedforward Networks

- Introduction
- Terminologies
- Training a Deep Feedforward Neural Network
- Regularization Techniques
- Practical Guidelines

Introduction

- Neural networks are a category of algorithms loosely inspired by the human brain, designed to recognize patterns.

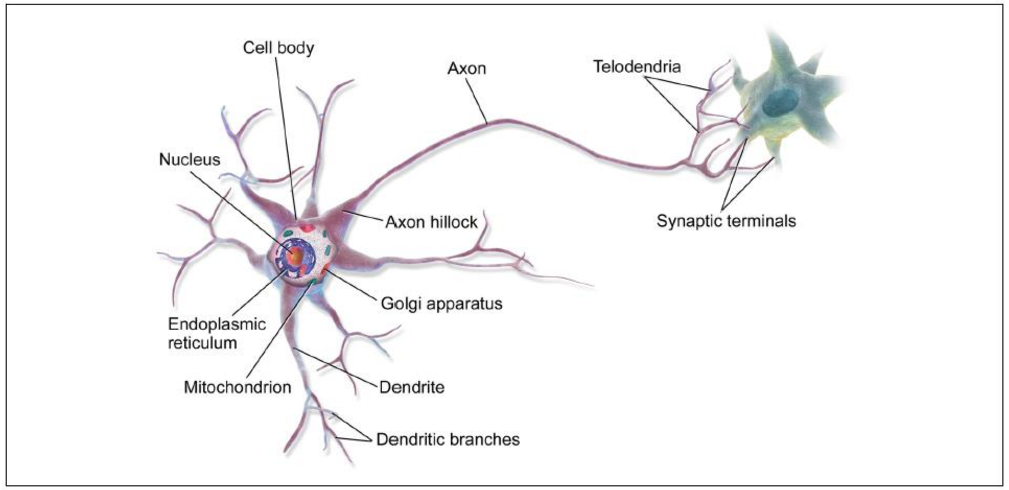


Figure 10-1 in Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow by Aurélien Géron.

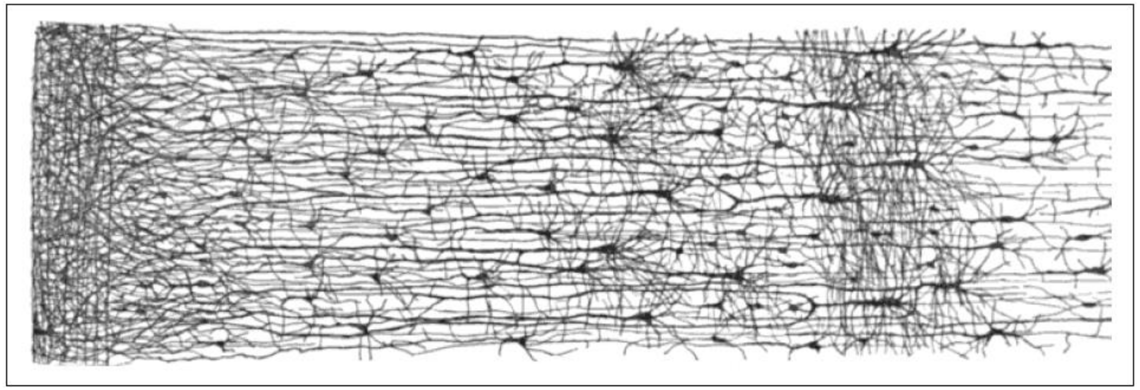
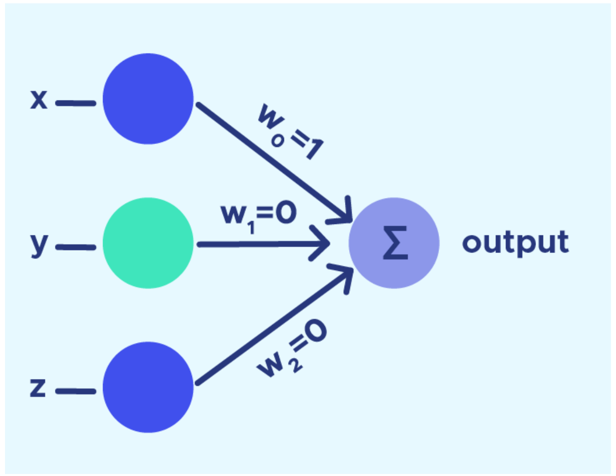


Figure 10-2 in Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow by Aurélien Géron.

What is a Perceptron

- A perceptron is the simplest neural network unit, originally modeled after a biological neuron.
- It takes a set of inputs, applies a linear transformation, and produces an output using a step function.



Source: https://datascientest.com/en/perceptron-definition-and-use-cases

The Importance of Perceptrons

- Perceptrons are foundational to understanding more complex neural networks.
- They were one of the earliest algorithms that allowed computers to learn from data, serving as the stepping stone to multi-layer neural networks and deep learning.

Limitations of Perceptrons

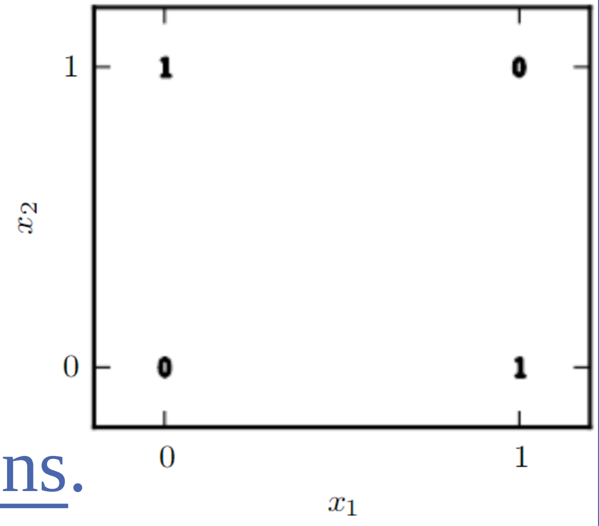


Figure 6.1 in Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

- Perceptrons can only model linearly separable functions.
- They cannot handle problems like XOR, which are not linearly separable.
- This led to the development of multi-layer perceptrons (MLP) and other advanced neural network architectures.

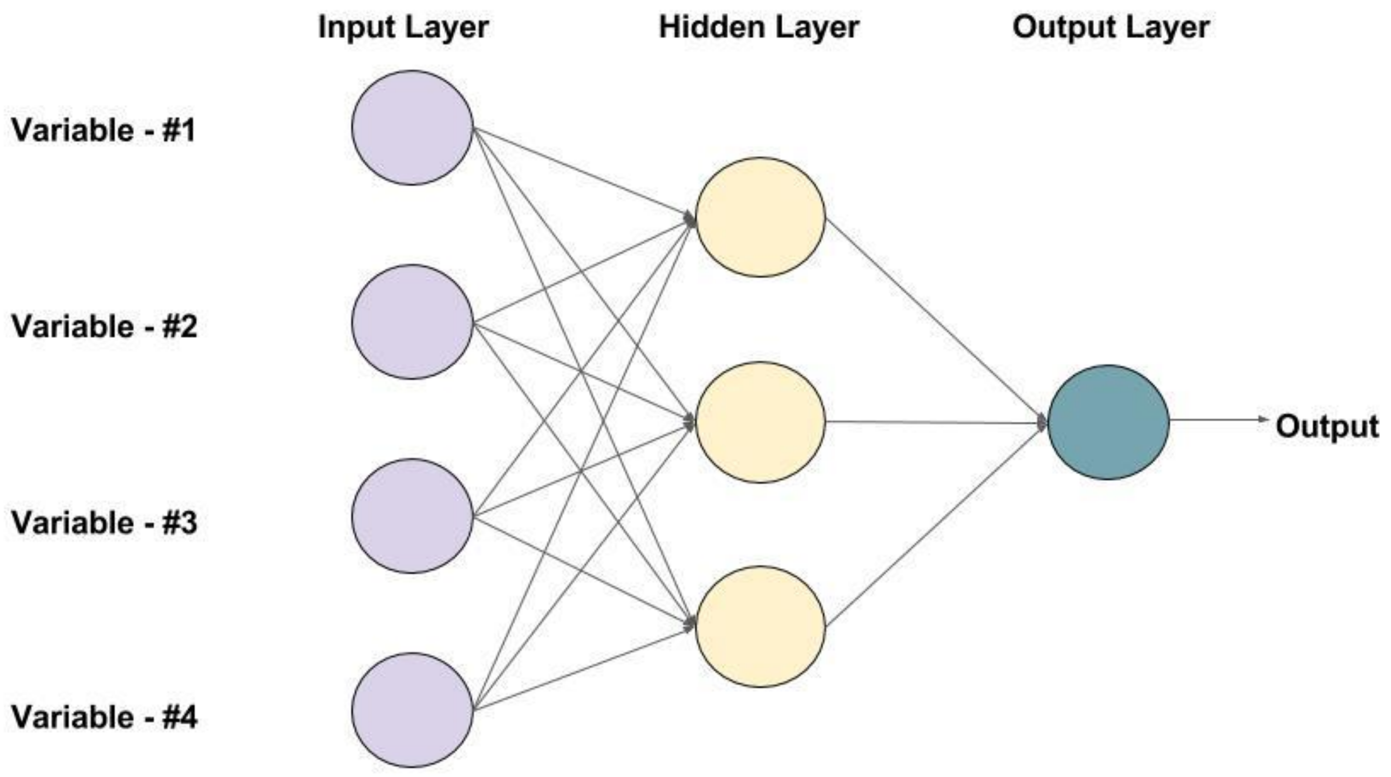
Multi-Layer Perceptrons (MLP)

- A multi-layer perceptron (MLP) consists of multiple layers of perceptrons, often with non-linear activation functions.
- MLPs can model a wider range of functions and are widely used in various deep learning applications.

What is Feedforward

- In a feed-forward neural network, the data flows in one direction, from the input layer to the output layer.
- There are no recurrent or looping connections.
- Each layer only receives information from the previous layer and passes it on to the next.

What is Feedforward



An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)

Source: https://learnopencv.com/understanding-feedforward-neural-networks/

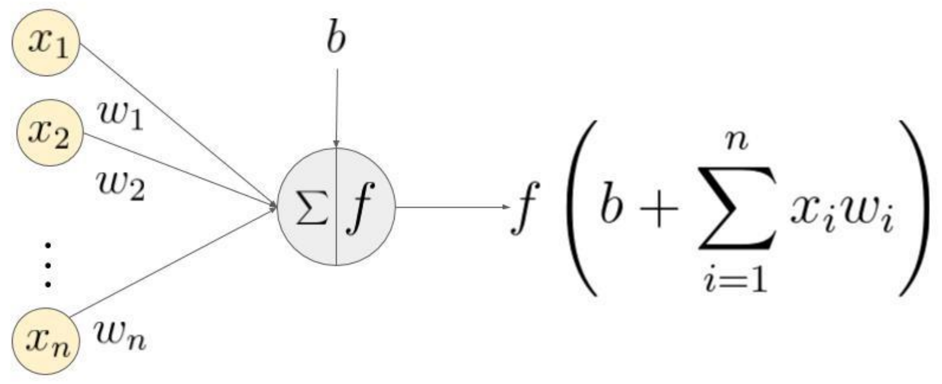
Why Feedforward Networks

- Feed-forward networks are straightforward to understand and implement.
- They are highly effective for various machine learning tasks like classification and regression.
- They serve as the foundation for more complex architectures.

Meaning of Deep

- The "deep" in deep neural networks refers to the architecture's ability to learn complex patterns through multiple layers of abstraction, making them more versatile and powerful for various complex tasks.

Terminologies

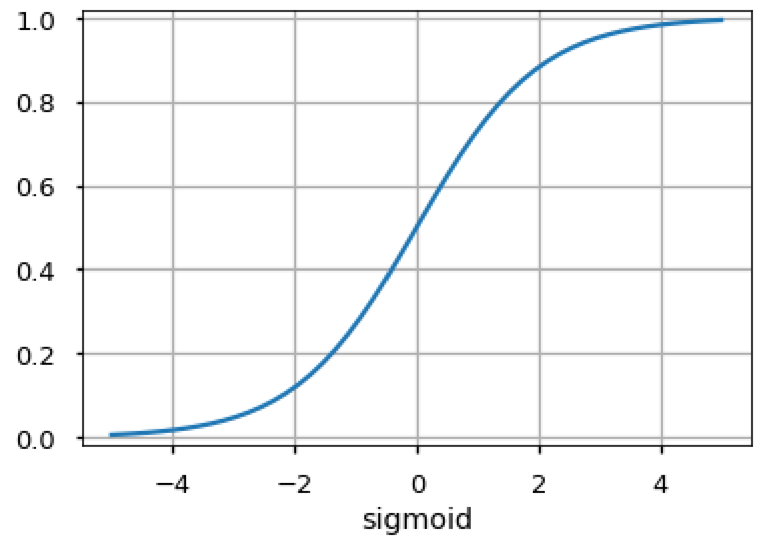


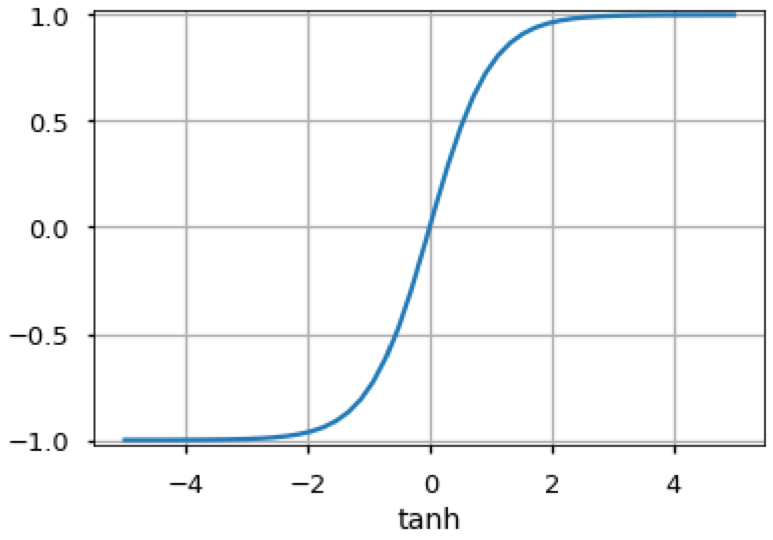
An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

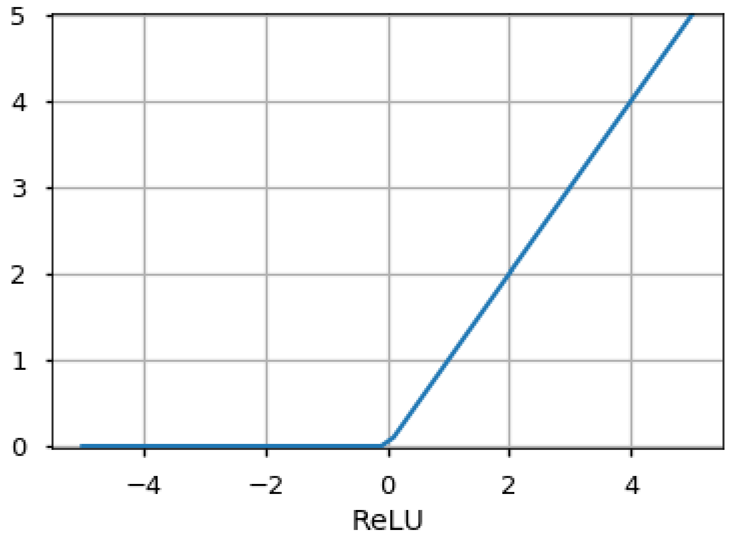
Source: https://learnopencv.com/understanding-feedforward-neural-networks/

- Neuron: Basic unit in a neural netwo
- Layer: A collection of neurons.
- Activation Function: Function applied to neuron's output.
- Weights and Biases: Parameters to be learned.
- Cost Function: Measure of error.
- Batch Size: Number of training samples used per iteration.
- Learning Rate: Step size in optimization.
- Epoch: One pass through the entire dataset.

Activation Function (1/2)







Sigmoid

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

Hyperbolic tangent

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

Rectified Linear Unit

$$f(x) = \max(0, x)$$

Activation Function (2/2)

	Linear	Sigmoid	tanh	ReLU
Formula	$f(x) = x$	$f(x) = \frac{1}{1+e^{-x}}$	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f(x) = \max(0, x)$
Pros	Simple, computationally efficient	Outputs between 0 and 1	Outputs between -1 and 1, zero-centered	Computationally efficient, helps with vanishing gradient problem
Cons	Cannot model complex functions, not zero-centered	Vanishing gradient problem, not zero-centered	Vanishing gradient problem	Dying ReLU problem (some units never activate)
Suitable Problem	Regression problems	Binary Classification, Output layer in some cases	Hidden layers where zero-centered outputs are desired	Most common, especially in CNNs and FNNs

Role of Activation Function

- Introducing Non-Linearity
- Function Approximation
- Decision Making
- Output Range
- Sparsity
- Gradient Descent Optimization

Cost Function

	Mean Squared Error (MSE)	Cross-Entropy	Hinge Loss	Kullback-Leibler Divergence
Formula	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	$-\sum_i [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$	$\max(0,1 - y_i\hat{y}_i)$	$\sum_i P(i) log \frac{P(i)}{Q(i)}$
Pros	Simple, intuitive	Good for classification, not affected by sigmoid saturate	Used for "maximum-margin" classification, such as SVMs	Measures how one probability distribution diverges from a second expected probability distribution.
Cons	Sensitive to outliers	Computational issues for log(0)	Not suitable for probabilistic interpretation	Not symmetric, computationally intensive
Suitable Problem	Regression problems	Binary/Multi-class Classification	Binary Classification	Multi-class Classification

Role of Cost Function in Optimization

- The cost function is the objective function that optimization algorithms like gradient descent try to minimize.
- During the backward pass, the gradient of the cost function is computed for every weight and bias.

Forward Pass (1/2)

- The primary purpose of the forward pass is to compute the network's prediction based on the current weights and biases.
- Input Layer:
  - The input data is fed into the input layer of the neural network.
  - Each neuron in the input layer corresponds to one feature in the dataset.

Forward Pass (2/2)

- Hidden Layers:
  - The data then propagates through one or more hidden layers.
  - Each neuron in a hidden layer computes a weighted sum of its inputs (from neurons in the previous layer), adds a bias, and then applies an activation function.
  - Output = Activation (Weight × Input + Bias)
- Output Layer:
  - Finally, the data reaches the output layer, where a similar computation occurs.
  - The output layer's activation function might differ based on the problem you're solving (e.g., softmax for classification, linear for regression).

Backward Pass (1/2)

- The primary purpose of the backward pass is to adjust the weights and biases to minimize the error. This is how the network learns from the data.
- Compute Error:
  - Once the forward pass is complete, the network calculates the error (or loss) using a loss function.
  - This function measures how well the network performed compared to the actual target.

Backward Pass (1/2)

- The primary purpose of the backward pass is to adjust the weights and biases to minimize the error. This is how the network learns from the data.
- Compute Error:
  - Once the forward pass is complete, the network calculates the error (or loss) using a loss function.
  - This function measures how well the network performed compared to the actual target.

Backward Pass (2/2)

- Backpropagation:
  - The error is then propagated backward through the network.
  - This involves taking the derivative (gradient) of the loss function with respect to each weight by applying the chain rule of calculus.
- Update Weights and Biases:
  - Finally, the weights and biases are updated in the direction that minimally decreases the error.
  - This is typically done using optimization algorithms like Gradient Descent.

Epoch

- An epoch is one complete forward and backward pass of all the training examples.
- In simpler terms, an epoch is one cycle through the full training dataset.
- Usually, training a neural network takes more than a few epochs.

Why is Epoch important

- Convergence:

  Multiple epochs are essential for the model to converge.

- Generalization:

  Running too many epochs can lead to overfitting, where the model performs well on the training data but poorly on unseen data.

- Learning Patterns:

  Running multiple epochs ensures that the model has a chance to see the same data multiple times, thereby learning more complex features.

Batch Size

- Batch size is the number of training examples used in one iteration.
- For instance, let's say we have 1,000 training samples.

  If the batch size is 100, it will take 10 iterations to complete one epoch.

Types of Batches

- Full-batch:
  - The batch size is equal to the size of the training set.
  - This is computationally expensive and not feasible for large datasets.
- Mini-batch:
  - The batch size is a fraction of the dataset, commonly 32, 64, or 128.
  - This is the most commonly used type.

Why is Batch Size important

- Computational Efficiency:

  Mini-batch sizes make the most of modern hardware capabilities and are generally more computationally efficient than other batch types.

- Generalization:

  Smaller batches have a regularizing effect, providing a level of "noise" in the optimization process, reducing the risk of overfitting.

- Memory:

  Large batch sizes can lead to memory constraints. Smaller batch sizes are easier to fit into memory.

Linking Epoch and Batch Size

- More epochs with a smaller batch size might give the model more updates, finer control, but can be computationally expensive.
- Fewer epochs with a larger batch size might make the model converge quickly, but it could overshoot the optimal point.

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)

Step 1 – Data Collection and Preprocessing

- The first step involves collecting the dataset and preprocessing it.
- Preprocessing may include normalization, encoding categorical variables, and splitting the dataset into training, validation, and test sets.
- Normalization is crucial for numerical stability and faster convergence.
- Usually, data is transformed to have zero mean and unit variance.

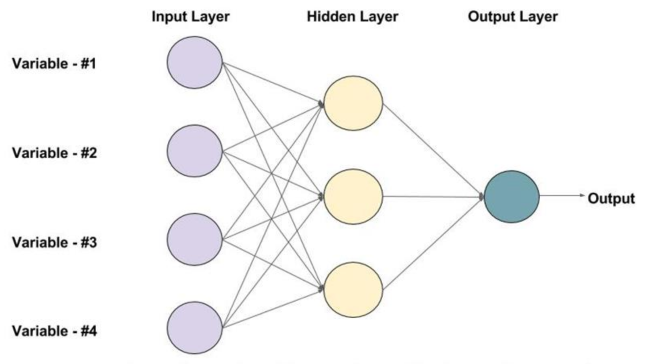
Step 2 – Initialize Weights and Biases

- Randomly initialize the weights ($\mathrm{W}^{(l)}$) and biases ($b^{(l)}$) for each layer in the network.
- Proper initialization is crucial for the network to learn effectively.
- Weights: Initialized using methods like Xavier, He, or others to combat vanishing/exploding gradients.
- Biases: Usually initialized to zero or a small constant.

Xavier and He Initialization Methods

- Xavier initialization
  - More suited for Sigmoid and tanh activation functions.
  - $\mathrm{W}^{(l)} = Random\ values \times \sqrt{\frac{2}{n^{(l-1)} + n^{(l)}}}$
- He initialization
  - More suited for ReLU and its variants.
  - $\mathrm{W}^{(l)} = Random\ values \times \sqrt{\frac{2}{n^{(l-1)}}}$
- $n^{(l-1)}$ is the number of units in the layer preceding the weights and $n^{(l)}$ is the number of units in the layer following the weights.

Step 3 – Forward Propagation



An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)

- Pass the input data through the network to get the output.
- Each layer performs a weighted sum of its input and passes it through an activation function.
- $Z^{(l)} = \mathrm{W}^{(l)} \cdot A^{(l-1)} + b^{(l)}$
- $A^{(l)} = g^{(l)}(Z^{(l)})$
- $g^{(l)}$ is the activation function for layer *l*
- If there are L layers, then the predict value is $A^{(L)} = g^{(L)}(Z^{(L)})$

Step 4 – Compute Loss

- Calculate the loss between the predicted output ($\hat{y}$) and the actual labels (y).
- $J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(y_i, \hat{y}_i)$
- $J(W,b)$ is cost function
- Compute the cost J to check how well the model is doing.
- The cost function could be Mean Squared Error for regression problems or Cross-Entropy for classification.

Step 5 – Backpropagation

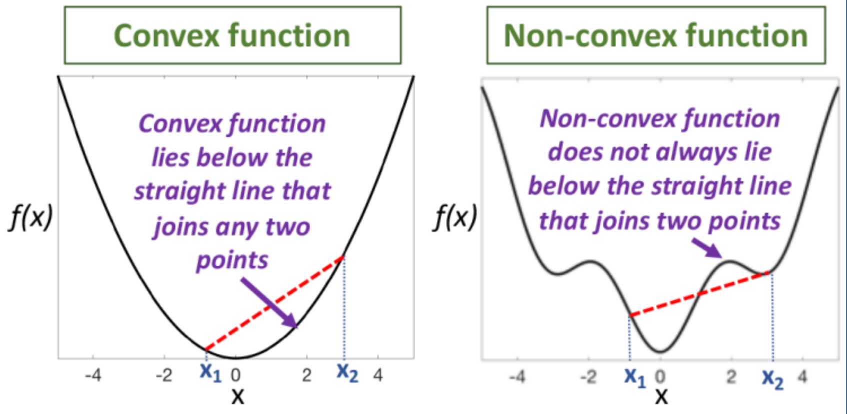
- Compute the gradients of the loss function with respect to each weight and bias by backpropagating from the output layer to the input layer.
- $\frac{\partial J}{\partial \mathrm{W}^{(l)}} = \frac{\partial J}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial \mathrm{W}^{(l)}}$

Step 6 – Update Weights and Biases (Optimization)

- This step involves optimization techniques to update the weights and biases such that the loss is minimized.
- Different optimization algorithms like Gradient Descent, Stochastic Gradient Descent (SGD), Momentum, or Adam can be used to update weights and biases.
- $\mathrm{W}^{(l)} = \mathrm{W}^{(l)} - \alpha \frac{\partial J}{\partial \mathrm{W}^{(l)}}$ and $b^{(l)} = b^{(l)} - \alpha \frac{\partial J}{\partial b^{(l)}}$

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 techniques are used to prevent a model from fitting too closely to the training data, a phenomenon known as overfitting.
- They add constraints to the optimization process, ensuring that the model generalizes well to unseen data.

L1 and L2 Regularization

- L1 Regularization:
  - Adds "absolute value of magnitude" of the coefficient as a penalty term to the loss function.
  - $J(W, b) = J(W, b) + \lambda \sum_{i=1}^{n} |w_i|$
- L2 Regularization:
  - Adds "squared magnitude" of the coefficient as a penalty term to the loss function.
  - $J(W, b) = J(W, b) + \lambda \sum_{i=1}^{n} {w_i}^2$

Dropout

- 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.

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.

Hyperparameter Tuning

- What are Hyperparameters?
  - Parameters in the model that are not learned but set prior to the training process. E.g. Learning rate, batch size, number of layers, etc.
- Common Techniques:
  - Grid Search: Exhaustive search over a specified parameter grid.
  - Random Search: Randomly sampling from a distribution of parameters.
  - Bayesian Optimization: Probabilistic model-based optimization.

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.

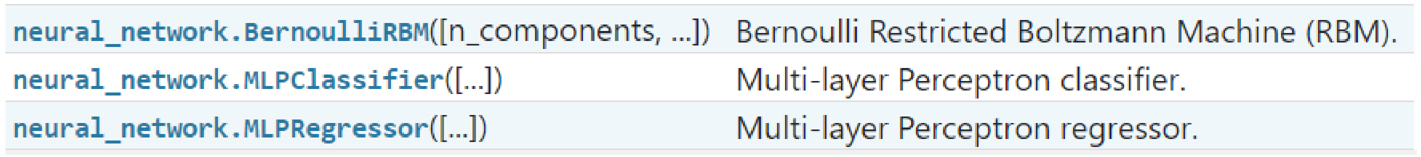
Practical Guidelines (6/6)

- 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)
```

Implementing a Feedforward Network in Pytorch (2/4)

- Compilation

```
criterion = nn.CrossEntropyLoss()

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

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()
```

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
```

A Neural Network Playground

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

Thank you