Deep Learning - In a nutshell

Terms and Definitions

- **Perceptron**: A simple linear classifier, representing a single neuron. It makes predictions based on a weighted sum of inputs and applies a step activation function. Limited to solving linearly separable problems.
- **Neuron**: A fundamental computational unit in neural networks that processes input data by applying weights and an activation function. Neurons can be part of shallow or deep networks and are general units of both simple and complex architectures.
- **Multilayer Perceptron (MLP)**: A type of neural network with multiple layers—consisting of an **input layer**, one or more **hidden layers**, and an **output layer**. MLPs are capable of learning complex, non-linear patterns and are trained using the backpropagation algorithm. They can perform tasks like classification and regression.
- **Deep Neural Network (DNN)**: A specific type of MLP with many hidden layers. The "depth" of the network allows it to model more complex data patterns, making DNNs especially useful for tasks like image recognition, speech processing, and natural language understanding.

Core Concepts

Neural Networks (NNs):

A general term for **computational models** that consist of interconnected **neurons** arranged in layers. NNs can vary in depth (number of layers), with shallow networks typically having only one hidden layer, while deeper networks may have many.

Artificial Neural Networks (ANNs):

A subset of NNs, specifically designed for machine learning and AI tasks. ANNs can be shallow or deep, depending on the number of hidden layers they have. All deep neural networks are ANNs, but not all ANNs are deep.

Multilayer Perceptron (MLP):

- **Structure: Input layer → Hidden layer(s) → Output layer**
- MLPs are a type of feedforward network, where information flows in one direction (from input to output).
- Capable of solving non-linear problems by using non-linear activation functions (e.g., ReLU, Sigmoid).
- Trained using **backpropagation** and **gradient descent** for optimization.

Deep Neural Networks (DNNs):

- A DNN is a type of MLP with more than two hidden layers.
- The additional layers allow DNNs to model more complex patterns and representations in the data.
- Commonly used in high-dimensional tasks like image and speech recognition, where deep architectures can capture intricate relationships in the input data.

Key Operations in Neural Networks

Forward Pass:

The process of passing input data through the network layer by layer to generate an output. Each layer applies transformations to the input using weights, biases, and activation functions, resulting in the network's prediction or classification.

Autograd:

A tool used in machine learning frameworks to automatically compute the gradients of tensors. It is crucial for the **backpropagation** process, allowing efficient calculation of the derivatives needed to update the weights during training.

Backpropagation:

An algorithm used to train neural networks by minimizing the error in predictions. It works by computing the gradient of the loss function with respect to each weight, applying the **chain rule**. The gradients are then used to adjust the weights to reduce the overall error.

Weights and Bias

Weights:

Weights are the parameters in a neural network that define the strength of connections between neurons. Each connection between two neurons has an associated weight, and these weights are adjusted during training to minimize the error in the network's predictions.

How Knowledge is Stored in Weights:

- **Knowledge in a neural network is encoded in its weights**. During training, the network learns by adjusting these weights based on the input data and the error (or loss) of the predictions.
- As the network is trained, the weights capture patterns, features, and relationships from the data. For example, in an image recognition task, certain weights may represent features like edges, textures, or colors.
- Once trained, the final set of weights contains the learned knowledge, allowing the network to generalize and make predictions on new, unseen data.

Bias:

- **Bias** is an additional parameter in a neural network that allows the model to shift the activation function to better fit the data. Unlike weights, which scale the input values, bias shifts the output of the weighted sum before applying the activation function.

- It helps the model handle cases where the input data do not pass through the origin (i.e., when all input values are zero, bias ensures that the network can still produce a meaningful output).
- **Bias allows greater flexibility** by enabling neurons to have a non-zero output even when all inputs are zero, improving the model's ability to fit complex patterns.

How Bias Works:

- For each neuron, the bias term is added to the weighted sum of inputs before the activation function is applied:

$$Output = f(\sum(input \times weight) + bias)$$

- Bias provides an extra degree of freedom that allows the network to adjust the overall output more flexibly. This improves the network's ability to generalize to new data and to learn more complex relationships.

Using DNNs in the real world

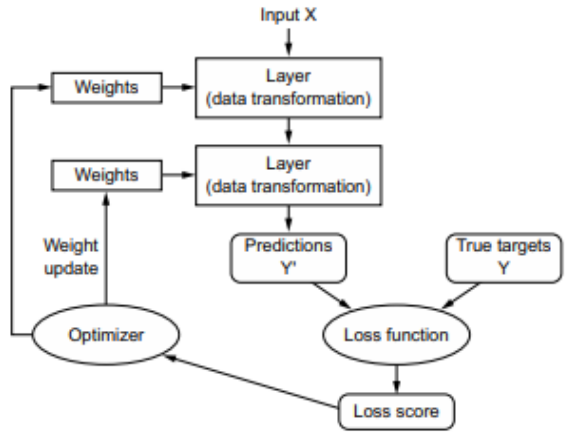


Figure 2.26 Relationship between the network, layers, loss function, and optimizer

Loss

Loss, also known as the cost or error function, is a measure of how well a neural network performs on a given task. It quantifies the difference between the network's predictions and the actual target values. The goal of training is to minimize this loss

function, thereby improving the network's performance. Common loss functions include mean squared error for regression tasks and cross-entropy for classification tasks.

Optimizer

An optimizer is an algorithm used to adjust the parameters (weights and biases) of a neural network during training to minimize the loss function. It determines how the network learns from the computed error and updates its weights. Popular optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop. Each optimizer has its own strategy for updating parameters, often involving techniques like momentum or adaptive learning rates.

Epoch

An epoch in machine learning refers to one complete pass through the entire training dataset. During each epoch, the neural network processes all training examples once, updating its weights after each batch or individual example depending on the training configuration. Multiple epochs are typically required for the network to learn effectively, with the number of epochs being an important hyperparameter that affects training time and model performance.

Batch

A batch is a subset of the training data used in a single iteration of the optimization algorithm. Instead of processing the entire dataset at once, which can be computationally expensive, the data is divided into smaller batches. The network processes one batch at a time, computes the loss, and updates its weights accordingly. Batch size is a crucial hyperparameter that affects training speed, memory usage, and the model's ability to generalize. Common batch sizes range from 32 to 256, but can vary based on the specific problem and available computational resources.

Activation Functions

Activation functions are crucial in neural networks as they introduce non-linear properties to the system, which allows the network to learn complex patterns in the data. They are applied to each neuron in the network, and determine whether it should

be activated or not, based on the weighted sum of its inputs. Here are a few common activation functions:

- **ReLU (Rectified Linear Unit):** This function outputs the input if it is positive; otherwise, it outputs zero. It is the most widely used activation function due to its computational simplicity and effectiveness in mitigating the vanishing gradient problem.
- **Sigmoid:** It transforms its input to a value between 0 and 1, which makes it suitable for binary classification tasks. However, it is less commonly used nowadays because it can cause the vanishing gradient problem during training.
- **Tanh (Hyperbolic Tangent):** Similar to the sigmoid but outputs values between -1 and 1. It is also prone to vanishing gradients but is more symmetric around the origin, which can provide better performance in certain cases.
- **Softmax:** Often used in the output layer of a network for multi-class classification; it converts logits to probabilities by taking the exponentials of each output and then normalizing these values by dividing by the sum of all exponentials.

Linear vs. Non-Linear Activation Functions:

- **Linear Activation Function**: In a linear activation function, the output is simply a linear function of the input. This means that the function does not introduce any non-linearity. Linear activation functions are rarely used in hidden layers because, without non-linearity, the network would effectively become a single-layer model, no matter how many layers are stacked. This limits the network's ability to learn complex patterns.
- **Non-Linear Activation Function**: Most activation functions used in modern neural networks are non-linear (like ReLU, Sigmoid, and Tanh). Non-linearity allows the network to learn and model complex relationships between inputs and outputs, enabling it to make decisions that are not just linear combinations of the input data.

Regularization and Generalization

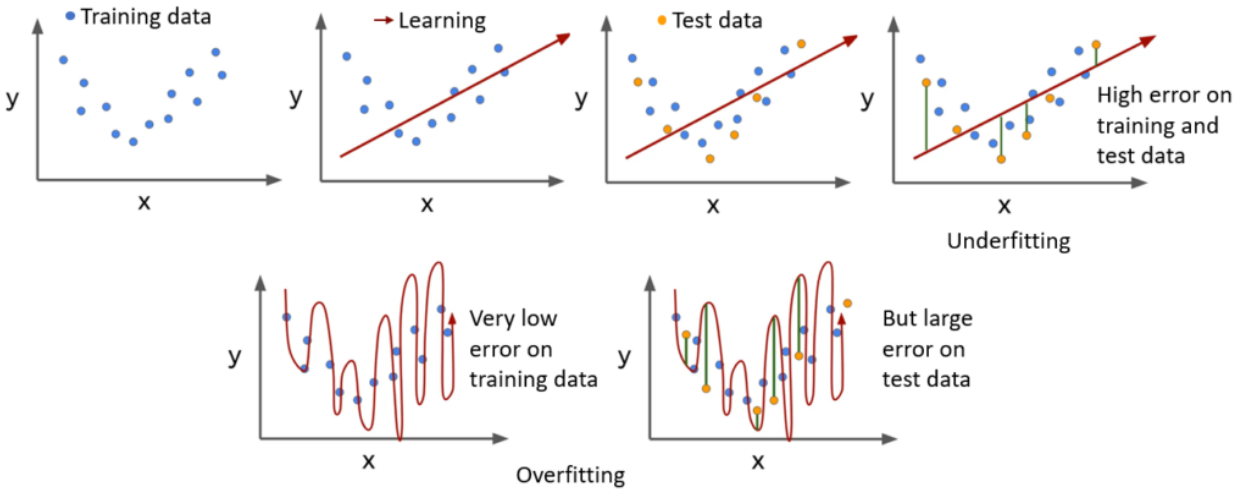
Overfitting:

- **Overfitting** occurs when a model learns too much from the training data, including noise and irrelevant details, which reduces its ability to generalize to new, unseen data.

- An overfitted model performs well on training data but poorly on test or validation data because it has become too complex and tailored to the specific training set.

Underfitting:

- **Underfitting** occurs when a model is too simple and fails to capture the underlying patterns in the training data.
- An underfitted model performs poorly on both training and test data because it cannot model the complexity of the data effectively.



Dropout:

- **Dropout** is a regularization technique used to prevent overfitting. During training, dropout randomly "drops" a percentage of neurons (i.e., sets their output to zero) in the network, forcing the remaining neurons to work more independently.
- By doing this, the network becomes less reliant on specific neurons and generalizes better to new data.
- Dropout is typically used only during training and is turned off during the inference phase (prediction).

L1 Regularization:

- **L1 Regularization** adds a penalty to the loss function based on the sum of the absolute values of the model's weights. This encourages sparsity in the weights, meaning that it can drive some weights to exactly zero.

- L1 regularization is useful when feature selection is desired, as it can lead to sparse models where only the most important features have non-zero weights.

L2 Regularization:

- **L2 Regularization** adds a penalty to the loss function based on the sum of the squared values of the model's weights. This encourages small but non-zero weights across the network.
- L2 regularization helps reduce overfitting by preventing the model from relying too heavily on any single feature and instead distributing the importance across many features.

Metric recap

Accuracy:

- Proportion of correct predictions out of all predictions
- Simple but can be misleading for imbalanced datasets

Precision:

- Proportion of true positive predictions among all positive predictions
- Measures quality of positive predictions

Recall:

- Proportion of true positive predictions among all actual positive instances
- Measures ability to find all positive instances

F1 Score:

- Harmonic mean of precision and recall
- Balances precision and recall into a single metric

AUC-ROC:

- Area under the Receiver Operating Characteristic curve
- Measures model's ability to distinguish between classes

Keras and tensorflow

Relationship between Keras and TensorFlow

In TensorFlow 2.x, Keras has been adopted as the official high-level API of TensorFlow. This integration means that Keras is the recommended way to build neural networks in TensorFlow. It simplifies the process by providing a more user-friendly and modular approach to creating models. TensorFlow handles the lower-level operations, while Keras provides an easy-to-use interface.

Parameters

The number of parameters in a neural network is determined by counting the **connections** (weights) between the layers and adding the **biases**. Here's how to calculate the parameters step-by-step:

- **Connections (weights) between layers**:
  - Between the **input** layer and the **hidden** layer:

    $$input \times hidden = 3 \times 5 = 15$$

  - Between the **hidden** layer and the **output** layer:

    $$hidden \times output = 5 \times 2 = 10$$

- **Biases** in each layer:
  - Biases in the **hidden** layer:

    $$hidden = 5$$

  - Biases in the **output** layer:

    $$output = 2$$

- **Total Parameters**
  - $15 + 10 + 5 + 2 = 32$

Thus, the total number of parameters is **32**.

Core DL toolkit

Regression

- **Loss function:** MSE or MAE
- **Output layer:** Single neuron with a **linear** activation function (aka no activation function)

Binary Classification

- **Loss function:** Binary Crossentropy.
- **Output layer:** Single neuron with **Sigmoid** activation (Probability between 0 and 1)

Multi-class Classification:

- **Loss function:** Categorical crossentropy (or sparse variant)
  - Use `categorical_crossentropy` if your labels are one-hot encoded.
  - Use `sparse_categorical_crossentropy` if your labels are integer-encoded (0..4).
- **Output layer:** Multiple neurons (equal to the number of classes) with **softmax** activation

Multi-Label Classification

- **Loss function:** Binary Crossentropy.
- **Output layer:** 1 output neuron per label using sigmoid

Dimensionality-Reduction (Unsupervised**)**

- Autoencoders

Sequences

- RNNs, LSTM, GRU, Transformers

Why Layers?

In deep learning frameworks like **Keras**, **layers** are the fundamental building blocks of your neural network. They define *how* information flows through the model, how data is transformed at each step, and what parameters (weights, biases) the model learns.

1. **Modularity**
    - Each layer is a **self-contained** module that takes specific inputs and produces specific outputs.
    - Layers maintain their own parameters (e.g., weights and biases) and can perform unique operations. For example, a **Dense** layer does a matrix multiplication followed by an activation, while a **Conv2D** layer performs convolutions on image-like data.
2. **Abstraction**
    - Rather than manually implementing matrix multiplications or activation functions for every neuron, you can simply instantiate layers (e.g., `Dense(64, activation='relu')`).
    - Layers handle the detailed mathematical operations under the hood, letting you focus on *what* you want the network to do rather than *how* it does it.
3. **Composition**
    - Deep learning architectures emerge by **composing** multiple layers. You can stack them sequentially or design more elaborate structures (like skip connections in ResNets or multi-branch flows in inception-like architectures).
    - Keras supports this composition with both the **Sequential API** (simple linear stacking) and the **Functional API** (complex graphs).
4. **Flexibility**
    - By mixing and matching different types of layers (e.g., **Dense**, **Conv**, **LSTM**), you can tailor the network to different data modalities such as images, text, audio, or time series.
    - Additional layers like **Dropout** and **BatchNormalization** help address issues like overfitting or unstable gradients, giving you more fine-grained control over training dynamics.

Key Layer Types to Know

1. Layers for Sequences (NLP pre-transformers)

- **LSTM (Long Short-Term Memory)**, **GRU (Gated Recurrent Unit)**, and **SimpleRNN** handle time-dependent or sequential data by maintaining hidden states over time steps.
- **Conv1D** can also process sequence data by sliding a 1D convolutional filter across the time dimension.

2. layers for NLP

- **Embedding Layer**: Converts discrete tokens (e.g., words, subwords) into continuous vectors.
- **Positional Encoding** (not strictly a “layer” in some frameworks, but conceptually part of the architecture): Encodes each position in the sequence with a sinusoidal or learned embedding to inform the model about the order of tokens.
- **Transformer related layers:** It’s complicated and we’ll cover that bridge once

3. Layers for Images (Deep learning 2)

- **Conv2D** applies 2D convolutional filters, extracting spatial features from images.
- **Pooling layers** (e.g., **MaxPooling2D**) reduce spatial dimensions and help control overfitting.

Utility Layers

1. Batch Normalization

- **What it does**: Normalizes activations across each mini-batch, standardizing the inputs for the next layer.
- **Why it’s used**: Speeds up training, helps stabilize gradients, and often improves model accuracy.

2. Dropout

- **What it does**: Randomly sets a fraction of input units (neurons) to zero at each update during training.

- **Why it’s used**: Helps prevent overfitting by forcing the network to learn more robust features rather than relying on specific neurons.