

1. Apply Hubel and Wiesel's hierarchical model of simple and complex neurons to explain how the human brain would process and recognize a new visual stimulus, such as a triangle. 10M

Evaluation Component	Marks	Key Points
1. Simple neurons	2	Detect straight edges at specific orientations. Each is tuned to one orientation.
2. Collective role of simple neurons	2	Together, they cover all 360° of orientations.
3. Complex neurons	2	Receive input from multiple simple cells; combine information to detect curves/corners.
4. Hierarchical processing	2	Multiple layers → increasing complexity from edges → curves → shapes.
5. Abstract representation	2	Final layers represent complete objects (mouse, cat, bird, dog).

Simple Neurons (Fig. 1.5) – (2 Marks)

- Hubel and Wiesel discovered that certain neurons, called **simple cells**, respond best to **straight edges** at specific orientations.
- Each simple neuron is tuned to one particular orientation (e.g., vertical, horizontal, diagonal).

Collective Role of Simple Neurons (Fig. 1.5) – (2 Marks)

- A large group of simple neurons, each sensitive to different orientations, together can represent **all 360° of edge orientations**.
- This enables the brain to detect edges from any angle.

Complex Neurons (Fig. 1.6) – (2 Marks)

- Complex neurons receive input from several simple cells.

- They **recombine multiple orientations** to recognize more complex patterns like **corners and curves**.

Hierarchical Processing (Fig. 1.6) – (2 Marks)

- Information flows across layers: edges → curves → shapes.
- Each successive layer represents **increasing complexity** of visual features.

Abstract Representation of Objects (Fig. 1.6) – (2 Marks)

- After several hierarchical layers, the brain builds **abstract concepts** of whole objects such as a **mouse, cat, bird, or dog**.
- This shows how visual perception progresses from raw input to high-level understanding.

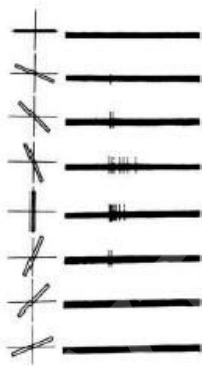


Figure 1.5 A simple cell in the primary visual cortex of a cat fires at different rates, depending on the orientation of a line shown to the cat. The orientation of the line is provided in the left-hand column, while the right-hand column shows the firing (electrical activity) in the cell over time (one second). A vertical line (in the fifth row from the top) causes the most electrical activity for this particular simple cell. Lines slightly off vertical (in the intermediate rows) cause less activity for the cell, while lines approaching horizontal (in the topmost and bottommost rows) cause little to no activity.

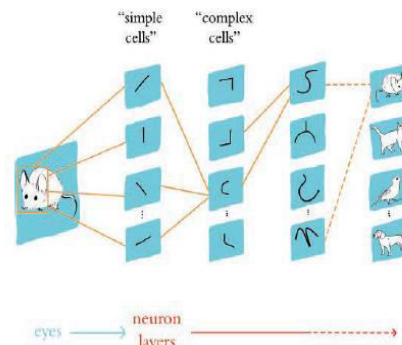


Figure 1.6 A caricature of how consecutive layers of biological neurons represent visual information in the brain of, for example, a cat or a human

2. Construct an explanation of how the neocognitron model, inspired by Hubel and Wiesel's biological findings, demonstrates the progression from edge detection to abstract object recognition in machine vision. 10M

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Link to biological vision (Hubel & Wiesel)	2	Biological simple and complex neurons inspired the model; simple cells detect edges, complex cells combine them.

Evaluation Component	Marks	Key Points (Expected in Answer)
2. Naming and borrowing terminology	2	Fukushima explicitly borrowed “simple” and “complex” cell terminology for artificial layers.
3. Structure of neocognitron	2	Artificial neurons are arranged hierarchically; early layers detect lines and orientations.
4. Increasing abstraction in deeper layers	2	Successive layers represent progressively complex and abstract features.
5. Application to machine vision	2	This layered representation laid the foundation for deep learning in vision, enabling object recognition similar to biological vision.

• **Biological basis: Hubel & Wiesel’s discovery** – (2 Marks)

- Hubel and Wiesel showed that the **primary visual cortex** has *simple cells* that detect edges at specific orientations and *complex cells* that combine these edges into shapes (e.g., curves, corners).
- This hierarchical structure explained how biological vision builds complexity step by step.

• **Direct inspiration and terminology (Fukushima)** – (2 Marks)

- Fukushima (1970s) explicitly cited Hubel and Wiesel’s work.
- He **borrowed their terms “simple” and “complex” cells** for the layers in his artificial vision model.

• **Neocognitron hierarchical design (similar to Fig)** – (2 Marks)

- The **first artificial layers** in the neocognitron act like simple cells, detecting **line orientations** and edges.
- This mimics the way biological neurons close to the retina respond to raw input.

• **Increasing abstraction in deeper layers (Fig.)** – (2 Marks)

- Successive layers recombine information from earlier layers.
- This results in progressively **complex and abstract features**, moving from lines → curves → shapes → objects.

• **Impact on machine vision (Fig. 1.8)** – (2 Marks)

- As shown in the historical timeline (Fig. 1.8), the neocognitron marks a **key step between biology and deep learning-based vision**.
- It laid the **foundation for convolutional neural networks (CNNs)** and showed how machine vision could progress from edge detection to object recognition.

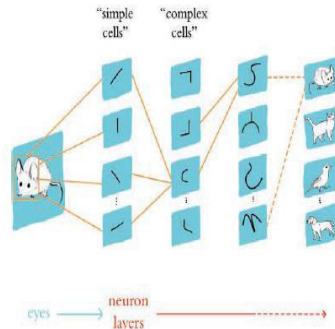


Figure 1.6 A caricature of how consecutive layers of biological neurons represent visual information in the brain of, for example, a cat or a human

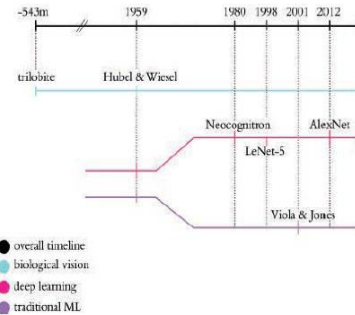


Figure 1.8 Abridged timeline of biological and machine vision, highlighting the key historical moments in the deep learning and traditional machine learning approaches to vision that are covered in this section

3. Apply your understanding of ImageNet, ILSVRC, and AlexNet to explain why AlexNet's 2012 victory in the ILSVRC challenge was a turning point for deep learning in machine vision. 10m

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Role of ImageNet dataset	2	ImageNet provided 14M labeled images across 22,000 categories. ILSVRC subset had 1.4M images in 1,000 categories → large, diverse, high-quality dataset.
2. Context of ILSVRC before 2012 (Fig. 1.15)	2	Until 2012, algorithms were traditional ML with feature engineering → deep learning absent in top models.
3. AlexNet's breakthrough performance (Fig. 1.15 & Fig. 1.17)	2	Krizhevsky, Sutskever, Hinton's team used AlexNet to dramatically outperform benchmarks → deep learning proved superior.
4. Technical factors enabling AlexNet (Fig. 1.17)	2	(i) Large dataset (ImageNet + data augmentation), (ii) GPU acceleration, (iii) Deeper architecture vs LeNet-5.
5. Wider impact	2	Shift from expert-driven feature engineering → automatic feature extraction by deep learning; triggered widespread adoption across vision, NLP, and games.

Answer:

1. Role of ImageNet dataset – 2 Marks

- ImageNet provided about **14 million labeled images** across **22,000 categories**, including objects as diverse as ships, animals, and plants.
- The ILSVRC subset contained **1.4 million images in 1,000 categories**, giving researchers a **large, diverse, and high-quality dataset** to train and evaluate models effectively.

2. Context of ILSVRC before 2012 (Fig. 1.15) – 2 Marks

- Until 2012, all top-performing models in ILSVRC were based on **traditional ML with feature engineering**.

- Deep learning was largely absent from the competition, and year-over-year accuracy improvements were **incremental and small**.

3. AlexNet's breakthrough performance (Fig. 1.15 & Fig. 1.17) – 2 Marks

- In 2012, **Krizhevsky, Sutskever, and Hinton** submitted **AlexNet**, which **dramatically outperformed all benchmarks**.
- The top-5 error rate dropped by nearly **10% compared to traditional ML models**, proving the superiority of deep learning for large-scale vision tasks.

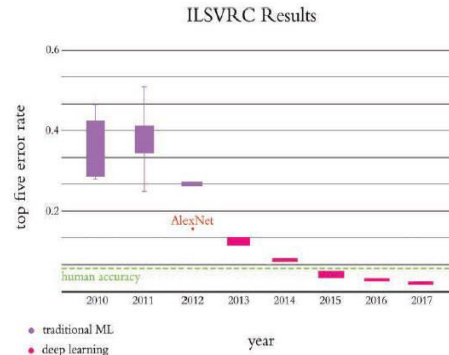


Figure 1.15 Performance of the top entrants to the ILSVRC by year. AlexNet was the victor by a head-and-shoulders (40 percent!) margin in the 2012 iteration. All of the best algorithms since then have been deep learning models. In 2015, machines surpassed human accuracy.

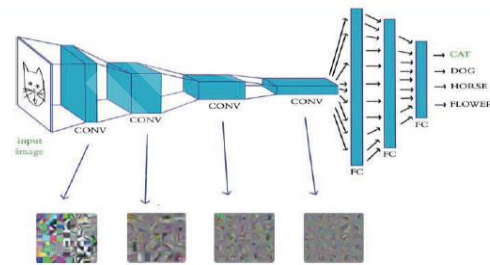


Figure 1.17 AlexNet's hierarchical architecture is reminiscent of LeNet-5, with the first (left-hand) layer representing simple visual features like edges, and deeper layers representing increasingly complex features and abstract concepts. Shown at the bottom are examples of images to which the neurons in that layer maximally respond, recalling the layers of the biological visual system in Figure 1.6 and demonstrating the hierarchical increase in visual feature complexity. In the example shown here, an image of a cat input into LeNet-5 is correctly identified as such (as implied by the green "CAT" output). "CONV" indicates the use of something called a convolutional layer, and "FC" is a fully connected layer; we formally introduce these layer types in Chapters 7 and 10, respectively.

4. Technical factors enabling AlexNet (Fig. 1.17) – 2 Marks

- **Large dataset:** Leveraged ImageNet and applied **data augmentation** (rotations, translations).
- **GPU acceleration:** Parallel processing enabled efficient large-scale training.
- **Architectural depth:** More layers than LeNet-5, with **ReLU activations and dropout** to prevent overfitting.

5. Wider impact – 2 Marks

- Marked a **paradigm shift** from expert-driven feature engineering to **automatic feature extraction** by deep networks.
- Sparked widespread adoption of deep learning across **computer vision, NLP, and game AI**, making it a turning point in AI research.

4. Utilize your understanding of deep learning hierarchies to explain how the TensorFlow Playground uses input, hidden, and output layers to learn and classify. -10M

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Input layer	2	Input layer uses two features: X1 (horizontal position) and X2 (vertical position) of each dot. Data includes orange (negative) and blue (positive) cases.
2. Role of hidden layers	2	Four hidden layers progressively extract features. First hidden layer neurons specialize in detecting edges

Evaluation Component	Marks	Key Points (Expected in Answer)
3. Increasing complexity across layers	2	(straight lines at specific orientations). Deeper layers recombine simple features → form curves, then more abstract shapes. Each layer builds on previous representations.
4. Output layer and classification	2	Output grid (−6 to +6 axes) represents final decision boundary. Neurons can represent intricacies of spiral shapes → classify orange vs. blue dots accurately.
5. Generalization and abstraction (Fig. 1.19)	2	Test data ensures generalization. Hierarchical feature-learning mirrors biological vision (Hubel & Wiesel, LeNet-5, AlexNet). Shows deep learning's ability to automatically build complex feature representations.

1. Input Layer– 2 Marks

- The input layer receives two features per dot:
 - X1 = horizontal position
 - X2 = vertical position.
- These inputs represent the location of dots on the grid.
- The dots are labeled: orange for negative cases and blue for positive cases.
- This simple two-feature input is the raw data from which the network must learn.

2. Role of First Hidden Layer (Fig. 1.19) – 2 Marks

- The first hidden layer neurons begin learning basic features such as edges and straight-line boundaries.
- Each neuron in this layer is specialized for detecting a specific orientation of a line, similar to how Hubel & Wiesel's simple cells in biological vision respond.
- This shows the foundation of hierarchical learning, where simple features are learned first.

3. Increasing Complexity Across Layers – 2 Marks

- As data passes deeper into the second and third hidden layers, neurons recombine these edges to form curves and more complex boundaries.
- By the later hidden layers, neurons represent abstract spiral shapes, which cannot be directly detected by earlier, simpler neurons.
- This demonstrates how complexity and abstraction increase with depth, a core principle of deep learning.

4. Output Layer and Classification – 2 Marks

- The output layer grid (ranging from −6 to +6 on both axes) integrates the features learned by the hidden layers.
- It forms a decision boundary that can accurately separate orange (negative) dots from blue (positive) dots.

- The neurons in the final layer essentially "solve" the spiral classification problem by representing the intricacies of the spiral pattern.

5. Generalization and Biological Analogy (Fig. 1.19) – 2 Marks

- The test data toggle allows evaluation on unseen dots, ensuring the model generalizes well beyond the training data.
- This feature-learning process mirrors the hierarchical structure in biological vision (Hubel & Wiesel's simple/complex cells, LeNet-5, and AlexNet).
- It highlights how deep learning models can automatically construct abstract representations without manual feature engineering.

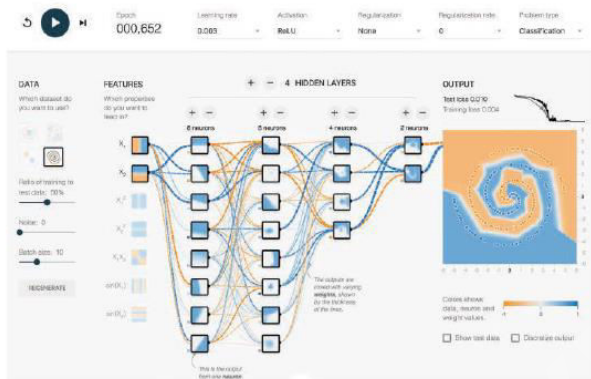


Figure 1.19 The network after training

(WRITE A ROUGH FIGURE FOR THIS)

5. Model the operation of LeNet-5 , showing how its layered architecture and back propagation enable automatic feature learning, and explain how this differs from conventional feature-engineered machine learning. 10M

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Hierarchical architecture of LeNet-5 (Fig. 1.11)	2	Built on neocognitron & biological inspiration (Hubel & Wiesel). Consists of multiple layers of artificial neurons enabling automatic feature extraction.
2. Role of backpropagation	2	Efficient learning through all layers; adjusts weights to minimize error; key to training deep networks like LeNet-5.
3. Advantage over traditional ML (Fig 1.12)	2	Deep learning automatically extracts features from raw data; traditional ML relies heavily on manual feature engineering (regression, SVM, random forest).
4. Data and processing requirements	2	Superior training datasets and faster processing power made LeNet-5 more accurate and efficient than neocognitron or earlier ML models.
5. Practical applications	2	Early commercial deployment: USPS ZIP code recognition; demonstrates real-world utility and shift from handcrafted features to automatic feature learning.

1. Hierarchical Architecture (Fig. 1.11): LeNet-5 built upon the neocognitron and biological vision insights (Hubel & Wiesel). Its multi-layer design allows automatic extraction of increasingly complex features from raw input data, unlike traditional ML which requires manual feature engineering. (2 Marks)
2. Backpropagation: LeNet-5 uses the backpropagation algorithm, which efficiently updates weights across layers to minimize prediction errors. This enables reliable learning throughout the deep network. (2 Marks)
3. Advantage over Traditional ML (Fig. 1.12): Traditional ML depends heavily on feature engineering, where practitioners manually design input variables for models like regression, random forests, or SVMs. LeNet-5, however, learns features automatically from raw data, freeing practitioners from labor-intensive preprocessing. (2 Marks)
4. Data and Processing Requirements: The model benefited from high-quality training data and faster computational resources, which improved both accuracy and efficiency compared to previous models like the neocognitron. (2 Marks)
5. Practical Applications: LeNet-5 was applied commercially by the USPS to automate ZIP code recognition, demonstrating the practical power of deep learning and marking a shift from expert-driven feature engineering to automatic feature learning in real-world tasks. (2 Marks)

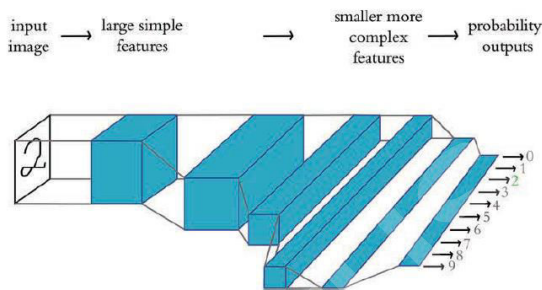


Figure 1.11 LeNet-5 retains the hierarchical architecture uncovered in the primary visual cortex by Hubel and Wiesel and leveraged by Fukushima in his neocognitron. As in those other systems, the leftmost layer represents simple edges, while successive layers represent increasingly complex features. By processing information in this way, a handwritten "2" should, for example, be correctly recognized as the number two (highlighted by the green output shown on the right).

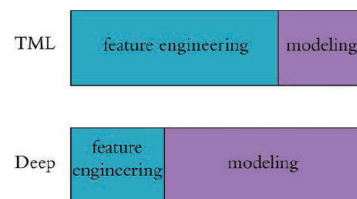


Figure 1.12 Feature engineering—the transformation of raw data into thoughtfully transformed input variables—often predominates the application of traditional machine learning algorithms. In contrast, the application of deep learning often involves little to no feature engineering, with the majority of time spent instead on the design and tuning of model architectures.

6. Make use of the concept of NLP by analyzing how simple, intermediate, and advanced language processing tasks differ in complexity, and evaluate the contribution of deep learning in solving intermediate and complex problems. 10m

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Definition and Scope of NLP (Fig. 2.2)	2	NLP sits at the intersection of computer science , linguistics , and AI . Processes naturally written/spoken human language to automate tasks or assist humans. Excludes programming code or isolated characters.
2. Simple NLP Applications	2	Examples: spell checkers , synonym suggesters , keyword search . Deterministic, rules-based, simple to implement. Deep learning not required.

Evaluation Component	Marks	Key Points (Expected in Answer)
3. Intermediate NLP Applications	2	Examples: assign reading levels, next-word prediction, document classification, named-entity extraction . Deep learning enables automatic feature extraction , handling moderately complex patterns in text.
4. Advanced/Sophisticated NLP Applications	2	Examples: machine translation, chatbots, automated Q&A . Challenges: context across sentences, sequential dependencies, nuance, humor. Requires advanced deep learning architectures (e.g., Transformers, LSTMs). Essential for intermediate and complex NLP tasks .
5. Role of Deep Learning	2	Allows automatic feature extraction, contextual understanding, sequential modeling, and abstraction of patterns . Facilitates scalable, accurate solutions across domains.

□ Definition and Scope of NLP (2 Marks):

Natural Language Processing (NLP) is an interdisciplinary field combining computer science, linguistics, and AI (Fig. 2.2). It involves processing naturally spoken or written human language to automate tasks or assist humans. NLP is distinct from programming languages or isolated character strings in spreadsheets.

□ Simple NLP Applications (2 Marks):

Basic tasks like spell checkers, synonym suggesters, and keyword search are straightforward, relying on rules or reference dictionaries. These tasks are deterministic, computationally lightweight, and do not require deep learning.

□ Intermediate NLP Applications (2 Marks):

Tasks such as assigning reading levels, next-word prediction, document classification, and named-entity extraction involve more complexity. Deep learning models are suitable here because they automatically learn features from raw text, capturing patterns that rules-based systems cannot. For example, predicting sentiment in movie reviews can be efficiently handled using LSTM or Transformer networks.

□ Advanced/Sophisticated NLP Applications (2 Marks):

Complex tasks like machine translation, automated question-answering, and chatbots require handling context across multiple sentences, sequential dependencies, nuance, and implied meaning. These tasks rely on advanced architectures like sequence-to-sequence models or Transformers. Such models allow NLP systems to approximate human-level understanding in specific domains.

□ Role of Deep Learning (2 Marks):

Deep learning automates feature extraction, captures contextual and sequential patterns, and enables modeling of abstract linguistic structures. It is essential for intermediate and complex

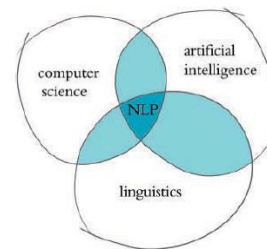


Figure 2.2 NLP sits at the intersection of the fields of computer science, linguistics, and artificial intelligence.

NLP tasks, allowing scalable and accurate NLP applications across domains such as translation, conversational AI, and predictive text generation.

7. Solve the challenge of representing word meaning in NLP by explaining how word vectors are generated, organized in vector space, and manipulated to extract semantic relationships. **10m**

Evaluation Component Marks Key Points (Expected in Answer)

1. Concept and advantage of word vectors	2	Word vectors (embeddings) represent word meaning and context unlike one-hot encodings (location only). Enable deep learning NLP models to automatically learn linguistic features .
2. Creation of word vectors	2	Words initially assigned random positions in vector space. Locations refined based on context words in corpus . Popular methods: word2vec, GloVe . Target word predicted from context; words in similar contexts cluster together.
3. Vector-space representation	2	Words mapped to n-dimensional vector space (3D cartoon example in Fig. 2.6). Coordinates define positions, clusters represent semantic similarity (e.g., days of week, months, programming languages).
4. Word-vector arithmetic for semantic relationships	2	Movements in vector space encode meaning (e.g., gender: king - man + woman $\approx$ queen; capital-country: Paris - France + London $\approx$ England). Preserves quantitative semantic relationships across dimensions.
5. Role in NLP applications	2	Word vectors provide dense, meaningful features for NLP models. Facilitate tasks like word similarity, analogy tasks, and deep learning NLP applications . Foundation for advanced NLP architectures.

Answer :

1. Concept and Advantage (2 Marks)

Word vectors, also called **word embeddings**, are dense vector representations of words that capture both **semantic meaning and context**, unlike one-hot encodings that only indicate word position. This enables deep learning models in NLP to **automatically extract linguistic features**, improving model performance on various NLP tasks.

2. Creation of Word Vectors (2 Marks)

Each word is initially assigned a **random location** in a multidimensional vector space. By analyzing the words that appear near each target word (its **context words**) in a corpus, vectors are gradually adjusted so that words appearing in similar contexts occupy **similar positions**. Popular algorithms include **word2vec** and **GloVe**, where the

model **predicts a target word from its context** across a large corpus.

3. Vector-Space Representation (2 Marks)

Words exist in an **n-dimensional vector space** (Fig. 2.6). In practice, this can be dozens or hundreds of dimensions. Words with similar meanings cluster together—e.g., days of the week (Monday, Tuesday, Wednesday), months, or programming languages form subclusters within broader semantic regions. Coordinates (x, y, z in 3D cartoon) precisely define word positions.

4. Word-Vector Arithmetic (2 Marks)

Word vectors encode semantic relationships.

Movements across vector space capture meaning, e.g.:

- **Gender relationship:** king - man + woman $\approx$ queen

- **Capital-country relationship:** Paris - France + London $\approx$ England

This arithmetic preserves **relative meaning** across dimensions, making it a powerful tool for NLP models (Fig. 2.7).

5. Role in NLP Applications (2 Marks)

Word vectors serve as **dense, meaningful input features** for NLP models. They enable tasks such as **word similarity**, **analogy solving**, **semantic clustering**, and are foundational for **deep learning architectures** in NLP, including sentiment analysis, language translation, and question-answering systems.

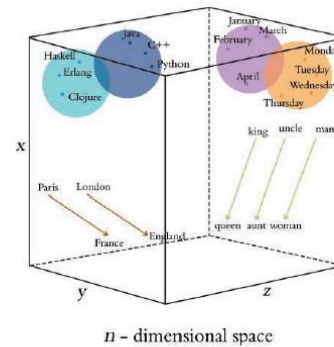


Figure 2.6 Diagram of word meaning as represented by a three-dimensional vector space

$$\begin{aligned} V_{\text{king}} - V_{\text{man}} + V_{\text{woman}} &= V_{\text{queen}} \\ V_{\text{bezos}} - V_{\text{amazon}} + V_{\text{tesla}} &= V_{\text{musk}} \\ V_{\text{windows}} - V_{\text{microsoft}} + V_{\text{google}} &= V_{\text{android}} \end{aligned}$$

Figure 2.7 Examples of word-vector arithmetic

8. Construct a detailed explanation of the hierarchical elements of natural human language—from phonology and morphology to syntax and semantics—and demonstrate how deep learning representations differ from traditional machine learning approaches at each level **10m**

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Figure Reference (Fig. 2.9)	2	Correctly references the schematic showing hierarchical elements: phonemes → morphemes → words → phrases → sentences → semantics. Demonstrates understanding of hierarchy.
2. Table Reference (Table 2.2)	2	Correctly refers to the differences in representations between traditional ML (local, rigid) and deep learning (distributed, flexible) at each element.
3. Phonology	2	Traditional ML: discrete phonemes; Deep learning: feature-learned

Evaluation Component	Marks	Key Points (Expected in Answer)
		phoneme vectors.
4. Morphology	2	Traditional ML: predefined morpheme list; Deep learning: morpheme vectors combined hierarchically to form word vectors.
5. Syntax and Semantics (combined)	2	Syntax: ML uses discrete categories, DL uses vectors to model structure. Semantics: ML limited in meaning; DL uses vector combinations to capture context.

1. Figure Reference (2 Marks)

- Figure 2.9 depicts the **hierarchy of language elements**: phonemes → morphemes → words → phrases → sentences → semantics.
- Shows progression from **basic sounds to abstract meaning**, illustrating the levels at which NLP models operate.

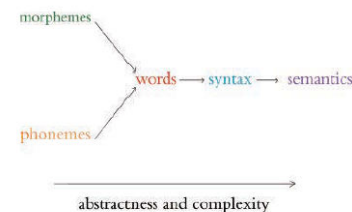


Figure 2.9 Relationships between the elements of natural human language. The leftmost elements are building blocks for further-right elements. As we move to the right, the more abstract the elements become, and therefore the more complex they are to model within an NLP application.

2. Table Reference (2 Marks)

- Table 2.2 contrasts **traditional ML vs deep learning representations**:
 - ML: local, rigid, discrete encoding.
 - Deep learning: distributed, flexible vectors; allows automatic feature learning.
- Highlights the **fundamental**

3. Phonology (2 Marks)

- Traditional ML: encodes discrete phonemes.
- Deep learning: learns features from auditory input and represents phonemes as vectors for flexibility and generalization.

Table 2.2 Traditional machine learning and deep learning representations, by natural language element

Representation	Traditional ML	Deep Learning	Audio-Only
Phonology	All phonemes	Vectors	True
Morphology	All morphemes	Vectors	False
Words	One-hot encoding	Vectors	False
Syntax	Phrase rules	Vectors	False
Semantics	Lambda calculus	Vectors	False

4. Morphology (2 Marks)

- Traditional ML: identifies morphemes from a predefined list.
- Deep learning: predicts morphemes and **combines their vectors hierarchically** to form words (e.g., out + go + ing → outgoing).

5. Syntax and Semantics (2 Marks)

- Syntax**: ML: phrases bucketed into discrete categories; DL: vector representations recombined by neural layers.
- Semantics**: ML limited in capturing abstract meaning, context, or nuance; DL: semantic vectors capture meaning via nonlinear combination of phrase vectors.

9. Apply your understanding of regularization in deep learning to explain why it is essential for controlling model complexity, reducing generalization error, and balancing the trade-offs between bias and variance. –10M

Evaluation Component	Marks	Key Points (Expected in Answer)
1. Definition and necessity of regularization	2	Explain that regularization modifies a learning algorithm to reduce generalization error, ensuring performance on new/unseen data. Mention why overfitting occurs and why regularization is needed.
2. Trade-offs: Bias vs Variance	2	Discuss how regularization increases bias slightly to reduce variance. Explain the context of underfitting, overfitting, and the goal of achieving an optimal model.
3. Regularization in deep learning context	2	Emphasize deep models' complexity; controlling model complexity is crucial because the true data-generating process is unknown and often outside the model family.
4. Common regularization strategies	2	Include: adding constraints on parameters, adding penalty terms in objective function, encoding prior knowledge, ensemble methods. Mention examples where applicable.
5. Practical application and implications	2	Explain that best-fitting deep learning models are often large but regularized; regularization ensures generalization in complex domains like images, audio, or text.

☐ Definition and Necessity (2 Marks)

- Regularization is any modification to a learning algorithm intended to reduce generalization error without necessarily reducing training error.
- It addresses the central challenge in machine learning: performing well not just on training data, but also on unseen inputs.
- Overfitting occurs when a model fits the training data too closely, capturing noise rather than underlying patterns. Regularization helps prevent this.

☐ Trade-offs: Bias vs Variance (2 Marks)

- Regularization trades increased bias for reduced variance.
- Underfitting: model family too simple → bias dominates.
- Overfitting: model family too complex → variance dominates.
- Goal: move the model from an overfitting regime (high variance) to an optimal regime where generalization error is minimized.

☐ Regularization in Deep Learning Context (2 Marks)

- Deep learning models are often large and highly complex, applied to domains like images, text, or audio.
- True data-generating processes are rarely fully known or captured within the model family.
- Therefore, controlling model complexity is not just about reducing parameters but applying strategic regularization.

- Common Regularization Strategies (2 Marks)
 - Parameter constraints: Limit magnitude of weights.
 - Penalty terms: Extra terms in the loss function to discourage over-complexity.
 - Prior knowledge encoding: Encourage the model to prefer simpler or meaningful solutions.
 - Ensemble methods: Combine multiple models to improve generalization.
- Practical Application and Implications (2 Marks)
 - In practice, the best-performing deep learning models are often large but regularized appropriately.
 - Regularization ensures that despite model size, the model generalizes well in complex domains like images, audio sequences, and text.
 - This allows deep learning to handle real-world tasks without overfitting to training data.

10. Identify how regularization addresses under-constrained machine learning problems, and explain its role in iterative optimization and its connection to the Moore-Penrose pseudoinverse. 10m

Evaluation Component	Marks	Expected Answer / Key Points
1. Definition of under-constrained / underdetermined problems	2	- Underdetermined problems occur when there are fewer examples (rows) than features (columns) or the data has zero variance in some direction. - Linear models like linear regression and PCA depend on inverting XX^T , which is singular in under-constrained cases. - Example: Logistic regression with linearly separable classes—multiple weight vectors can perfectly classify data, leading to infinite solutions.
2. Role of regularization in stabilizing under-constrained problems	3	- Regularization adds constraints or penalties to the learning algorithm to make solutions well-defined and numerically stable. - For linear models, regularization modifies XX^T to $XX^T + \alpha I$, ensuring invertibility. - Prevents unbounded growth of weights in iterative optimization (e.g., gradient descent). - Example: Weight decay ensures convergence in logistic regression when classes are linearly separable.
3. Connection to iterative optimization methods	2	- In iterative methods like stochastic gradient descent, under-constrained problems may lead to divergence (weights growing indefinitely). - Regularization provides a stopping mechanism: weight decay or penalties limit weight growth, guaranteeing convergence to a stable solution. - Ensures that iterative optimization produces reliable and generalizable models.
4. Link with Moore-Penrose pseudoinverse	2	- The pseudoinverse provides a stabilized solution for underdetermined linear equations. - Equation: $X^+ = \lim_{\alpha \rightarrow 0} (XX^T + \alpha I)^{-1} X^T = \lim_{\alpha \rightarrow 0} \{ (X^T X + \alpha I)^{-1} \} X^T$ - Interpretation: performing linear

Evaluation Component	Marks	Expected Answer / Key Points
5. Summary / overall significance	1	regression with weight decay; regularization stabilizes solutions even when $X^T X$ is singular. - Connects linear algebra to deep learning regularization techniques. - Regularization is critical in deep learning and machine learning to control model complexity, ensure numerical stability, and guarantee convergence in under-constrained problems. - Enables models to generalize well beyond training data.

For equations please refer to textbook(7.3 Regularization and Under-Constrained Problems)

11. Apply the concept of norm penalties as constrained optimization to explain how parameter norm penalties and explicit constraints (with reprojection) stabilize the training of deep learning models. 10m

1. Norm penalty as constrained optimization (2 Marks)

A parameter-norm penalty augments the original cost with a regularizer:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta).$$

This can be interpreted as the Lagrangian for the constrained problem "minimize $J(\theta)$ subject to $\Omega(\theta) \leq k$ ". Construct the generalized Lagrange function

$$L(\theta, \alpha) = J(\theta) + \alpha(\Omega(\theta) - k),$$

where $\alpha \geq 0$ is a KKT multiplier. The constrained solution is given by the saddle point

$$\theta^* = \arg \min_{\theta} \max_{\alpha \geq 0} L(\theta, \alpha).$$

(2 marks for stating the formulation, Lagrangian and the constrained interpretation.)

2. Equivalence of penalty and explicit constraint; role of α and k (2 Marks)

If we fix an optimal multiplier α^* then minimizing $L(\theta, \alpha^*)$ over θ is identical to minimizing the regularized objective:

$$\theta^* = \arg \min_{\theta} [J(\theta) + \alpha^* \Omega(\theta)].$$

Thus a norm penalty behaves like an implicit constraint: e.g. L2 penalty constrains parameters to lie in an **L2 ball**, L1 penalty constrains them to an **L1 region**. Increasing α tightens (shrinks) the feasible region; decreasing α relaxes it.

However, α does not directly equal k — the mapping $\alpha \leftrightarrow k$ depends on the form of $J(\theta)$ and must generally be determined indirectly (e.g., by cross-validation). (2 marks for explaining equivalence, L2/L1 mapping, and relation of $\alpha \leftrightarrow k$.)

3. Effect on iterative optimization (weight decay; preventing divergence) (2 Marks)

In iterative optimizers (e.g., SGD), a squared-norm penalty (weight decay) adds a term that pulls weights toward zero. Concretely, with weight decay the gradient update includes an extra $-\eta\alpha\theta$ term, which counteracts uncontrolled weight growth. This is essential in under-constrained or separable problems (e.g., logistic regression with linearly separable classes), where without regularization the optimizer can drive $\|\theta\| \rightarrow \infty$ to increase likelihood and never converge. Weight decay prevents that divergence and yields a stable finite solution. (2 marks for describing mechanism and example of divergence without regularization.)

4. Explicit constraints and reprojection: explanation & benefits (2 Marks)

Instead of penalties, explicit constraints force $\Omega(\theta) \leq k$ and then re-project weights back into the feasible region whenever they exceed it.

Key benefits:

- Avoids “dead units”: penalties tend to shrink weights excessively, while reprojection only acts when weights exceed the boundary.
- Provides **numerical stability**: prevents runaway feedback loops of large weights and gradients when training with high learning rates.
- Offers **direct control of k**: the practitioner can enforce a chosen bound rather than relying on an indirect penalty parameter.

(2 marks for explanation and listing benefits clearly.)

5. Column-wise norm constraints (practical trick) (2 Marks)

Hinton et al. recommend **constraining each column** of a weight matrix (incoming weights to a hidden unit) rather than the entire matrix Frobenius norm.

- Prevents any single hidden unit from dominating with very large weights.
- If implemented as a penalty, this would require a separate KKT multiplier per column, but reprojection is simpler and effective.
- This promotes balanced units, improves generalization, and avoids instability.

(2 marks for implementation idea, rationale, and benefit.)

12. Utilize the concept of dataset augmentation to illustrate how various augmentation strategies contribute to improving the robustness and generalization of machine learning models. 10m

1. Definition of Dataset Augmentation (2 marks)

- Dataset augmentation is a strategy to artificially enlarge training datasets by generating synthetic or transformed data points.
- It is particularly useful when real-world labeled data is scarce or costly to obtain.
- By creating new (x, y) pairs, the model is exposed to greater variability, reducing overfitting.

2. Role in Improving Generalization and Robustness (2 marks)

- A model trained on more diverse data becomes less sensitive to noise and variations in unseen inputs.
- For classification, augmentation enables the model to learn invariances (e.g., object position, orientation, small distortions).
- In speech recognition, injecting variations like pitch shifts or background noise makes the system more robust.

3. Illustration of Augmentation Techniques (3 marks)

(a) Geometric Transformations (1 mark)

- Translation (shifting an image a few pixels), rotation, scaling, or cropping are effective for image classification.
- For example, in object recognition, shifting an object slightly in the frame allows the model to recognize it regardless of location.

(b) Noise Injection (1 mark)

- Adding Gaussian noise to inputs or hidden activations simulates real-world imperfections.
- Dropout acts as augmentation at the hidden layer level by randomly “dropping” units, forcing the network to learn redundant, robust features.

(c) Domain-Specific Constraints (1 mark)

- Augmentation must respect the semantics of the data.
- Example: In OCR, flipping a ‘b’ can turn it into a ‘d’; hence, flips are not valid transformations.
- In contrast, for object recognition, flips and rotations are often useful.

4. Challenges and Limitations (2 marks)

- Not all tasks benefit equally:
 - Density estimation is difficult because valid synthetic samples require solving the task itself.
- Some transformations (like out-of-plane rotation in 3D images) are non-trivial to simulate.
- Poorly designed augmentations may harm performance by distorting labels.

5. Benchmarking and Fairness (1 mark)

- Augmentation can artificially inflate performance.
- For fair comparison, algorithms must be benchmarked using the same augmentation scheme.
- Otherwise, improvements may come from augmentation, not from the core algorithm.

1. Apply the concept of L2 Regularization (Weight Decay) to derive the modified weight update rule and closed-form solution for Linear Regression, and explain how the regularization parameter α influences the learned weights. **10m**

Concept / Step	Detailed Explanation, Equations & Interpretation	Marks
Definition of L2 Regularization (Weight Decay)	In machine learning, large model weights can cause overfitting. L2 regularization adds a penalty term proportional to the square of the weight magnitude to the objective function. The modified loss (objective) function is: $\tilde{J}(w) = J(w; X, y) + \frac{\alpha}{2} \ w\ _2^2 = J(w; X, y) + \frac{\alpha}{2} w^T w$ Here, $\alpha > 0$ is the regularization coefficient that controls the strength of the penalty. This term "decays" the weights toward zero, hence the name <i>weight decay</i>.	2
Gradient of the Regularized Objective Function	To minimize the new objective, compute the gradient: $\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \alpha w$ The second term αw is the gradient of the regularization penalty. It adds a <i>pull</i> toward zero for every weight.	1
Weight Update Rule (Gradient Descent)	Using gradient descent, the update step is: $w \leftarrow w - \eta (\nabla_w J(w; X, y) + \alpha w)$ Simplifying: $w \leftarrow (1 - \eta \alpha) w - \eta \nabla_w J(w; X, y)$ where η is the learning rate. → The term $(1 - \eta \alpha)$ <i>shrinks</i> the weights on each iteration before applying the gradient update.	2
Interpretation of Weight Shrinkage	The weight vector is scaled by $(1 - \eta \alpha)$ every step. This means weights that don't significantly contribute to reducing loss are gradually decayed toward zero. → This helps in reducing model variance and improving generalization.	1
Quadratic Approximation around Optimum w^*	Assume the unregularized loss near its minimum w^* is approximately quadratic: $J(w) \approx J(w^*) + \frac{1}{2} (w - w^*)^T H (w - w^*)$ where H is the Hessian matrix of second derivatives of $J(w)$. Since w^* minimizes J, its gradient is zero.	1

Finding the Minimum of the Regularized Objective	To find the minimum $\tilde{w}$ of the regularized function: Set the gradient to zero: $\nabla_w \tilde{J}(w) = 0 \Rightarrow \alpha w + H(w - w^*) = 0$ Rearranging: $(H + \alpha I)w = Hw^*$ Solving for w: $w = (H + \alpha I)^{-1} Hw^*$	1
Eigenvalue Decomposition Insight	If $H = Q\Lambda Q^T$ (where Q is orthogonal and Λ diagonal with eigenvalues λ_i), then: $w = Q(\Lambda + \alpha I)^{-1} \Lambda Q^T w^*$ → Each weight component is scaled by $\frac{\lambda_i}{\lambda_i + \alpha}$. Components corresponding to small eigenvalues (directions of low curvature) shrink more strongly.	1
Application to Linear Regression (Closed-Form Solution)	For linear regression, unregularized loss is: $J(w) = (Xw - y)^T (Xw - y)$ Without regularization, solution is: $w = (X^T X)^{-1} X^T y$ Adding L2 regularization gives: $\tilde{J}(w) = (Xw - y)^T (Xw - y) + \frac{\alpha}{2} w^T w$ Setting gradient to zero: $(X^T X + \alpha I)w = X^T y$ Hence, the regularized (ridge regression) solution is: $w = (X^T X + \alpha I)^{-1} X^T y$	1

2. **Apply the concept of L1 regularization to a linear regression model and explain its effect on weights. Support your answer with relevant equations and comparison with L2 regularization. (10 Marks)**

1. Definition of L1 Regularization (2 marks)

L1 regularization penalizes the sum of absolute values of model parameters:

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|$$

The regularized objective function becomes:

$$\tilde{J}(w; X, y) = J(w; X, y) + \alpha \|w\|_1$$

where $J(w; X, y)$ is the original cost function (e.g., least squares), and $\alpha > 0$ controls the strength of regularization.

2. Gradient / Subgradient of L1 (2 marks)

Unlike L2, the L1 regularization gradient is **element-wise constant**:

$$\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \alpha \text{sign}(w)$$

- $\text{sign}(w_i)$ returns +1, 0, or -1 depending on the value of w_i .
 - This leads to **sparsity**, as the gradient pushes smaller weights exactly to zero.
-

3. Analytical Solution (2 marks)

Assuming a **diagonal Hessian** $H = \text{diag}(H_{1,1}, \dots, H_{n,n})$ for simplification, the approximate solution for each weight dimension is:

$$w_i = \text{sign}(w_i^*) \cdot \max\left(|w_i^*| - \frac{\alpha}{H_{i,i}}, 0\right)$$

- If $|w_i^*| \leq \frac{\alpha}{H_{i,i}} \Rightarrow w_i = 0$ (weight eliminated).
 - If $|w_i^*| > \frac{\alpha}{H_{i,i}} \Rightarrow w_i$ is **shifted towards zero** by $\frac{\alpha}{H_{i,i}}$.
-

4. Comparison with L2 Regularization (2 marks)

For L2 regularization (weight decay) with diagonal Hessian:

$$w_i^{(L2)} = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^*$$

- L2 shrinks all weights but **does not produce exact zeros**.
 - L1 induces **sparse solutions**, making some weights exactly zero.
 - Sparsity allows L1 to act as a **feature selection mechanism**.
-

5. Sparsity & Feature Selection in LASSO (2 marks)

L1 regularization is widely used in **LASSO** (Least Absolute Shrinkage and Selection Operator):

- Zero-valued weights indicate unimportant features that can be discarded.
- Sparsity **simplifies models** and prevents overfitting.
- L1 can be interpreted as **MAP Bayesian inference with a Laplace prior**:

$$\log p(w) = -\alpha \|w\|_1 + n \log \alpha - n \log 2$$

(terms independent of w can be ignored during optimization).

3. Apply noise injection techniques to a neural network model and demonstrate how they improve noise robustness. Include the cases of noise applied to inputs, hidden units, weights, and output targets. Derive the equivalent regularization term for weight noise in the linear regression setting and explain the effect on model stability. 10m

1. Noise Injection at Inputs (2 marks)

- Adding small random noise $\epsilon \sim N(0, \eta I)$ to inputs x is a form of **dataset augmentation**.
- For infinitesimal variance, equivalent to **L2 regularization on weights** (Bishop, 1995).
- Encourages **robustness to small perturbations in input data**.

$$x \rightarrow x + \epsilon, \quad \epsilon \sim N(0, \eta I)$$

2. Noise Injection at Hidden Units (2 marks)

- Adding noise to hidden units is more powerful than input noise.
- Dropout is a special case: randomly sets hidden unit activations to zero during training.
- Encourages **redundancy and robustness in feature representations**.

3. Noise Injection at Weights (2 marks)

- Weight noise $\tilde{W} \sim N(0, \eta I)$ reflects **uncertainty in weights** $\rightarrow$ stochastic Bayesian inference.
- The perturbed objective function:

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\tilde{W})}[(\hat{y}_{\tilde{W}}(x) - y)^2]$$

- For small η , this is equivalent to adding a **stability-inducing regularizer**:

$$\tilde{J}_W \approx J + \eta \mathbb{E}_{p(x,y)} [\|\nabla_W \hat{y}(x)\|^2]$$

- Encourages the model to find **flat minima** where small weight perturbations do not change outputs much.
- **Linear regression case:** $\hat{y}(x) = w^\top x + b$

$$\tilde{J}_W \rightarrow \eta \mathbb{E}_{p(x)} [\|x\|^2]$$

- Independent of weights, **does not alter gradients**, but still encourages stable solutions.

4. Noise Injection at Output Targets (2 marks)

- Labels y can contain errors; noise at outputs prevents overfitting to noisy labels.
- Label smoothing replaces hard targets with **soft probabilities**:

$$y_i = \begin{cases} 1 - \epsilon, & \text{for correct class} \\ \epsilon/(k - 1), & \text{for incorrect classes} \end{cases}$$

- Prevents extreme weights in softmax classifiers and encourages **robust classification**.

5. Effect on Model Robustness (2 marks)

- Noise injection leads to:
 1. **Stability** against input and weight perturbations.
 2. **Improved generalization** by preventing overfitting.
 3. **Sparsity of influence** in hidden layers (dropout).
- Weight noise specifically ensures **flat minima**, meaning small changes in parameters do not drastically change outputs.

4. Construct explanations for Semi-Supervised Learning and Multi-Task Learning in deep learning. For each, describe the key principles, the role of labeled/unlabeled data or shared/task-specific parameters, and explain how they improve generalization. Include relevant equations or representations where applicable. 10m

Definition:

Semi-supervised learning uses **both labeled and unlabeled data** to improve learning performance. In deep learning, this often involves learning a representation $h = f(x)$ such that examples from the same class have similar representations.

Key Principles:

1. Labeled and Unlabeled Data:

- Labeled examples: $(x, y) \sim P(x, y)$
- Unlabeled examples: $x \sim P(x)$

The goal is to estimate $P(y|x)$ more accurately by leveraging $P(x)$.

2. Representation Learning:

- Learn a mapping $h = f(x)$ so that similar inputs cluster in representation space.
- A linear classifier can then operate in the new representation space, often achieving better generalization.

3. Combined Objective Function:

- Trade-off between supervised and unsupervised criteria:

$$J_{\text{total}} = J_{\text{sup}} + \lambda J_{\text{unsup}} = -\log P(y|x) + \lambda(-\log P(x) \text{ or } -\log P(x, y))$$

- Here, λ (or α in some texts) controls the **relative weight of the unsupervised/generative term**.
- **Condition for λ :**

$$0 < \lambda < 1$$

- Too small ($\lambda \rightarrow 0$): mostly supervised, under-utilizes unlabeled data.
- Too large ($\lambda \rightarrow 1$ or higher): unsupervised term dominates, may hurt prediction accuracy.

4. Practical Approach:

- Preprocessing with PCA or other dimensionality reduction can help cluster similar data.
- Generative models of $P(x)$ or $P(x, y)$ can **share parameters** with discriminative models of $P(y|x)$ for improved learning.

Equations Recap:

$$h = f(x), \quad J_{\text{total}} = -\log P(y|x) + \lambda(-\log P(x) \text{ or } -\log P(x, y))$$

- Choosing an appropriate λ balances supervised and unsupervised contributions for **better generalization**.

Definition:

MTL improves generalization by pooling examples from multiple related tasks. This introduces **soft constraints** on shared parameters.

Shared vs Task-Specific Parameters:

- **Shared (generic) parameters:** benefit from examples across all tasks.
- **Task-specific parameters:** benefit only from examples of their own task.

Deep Learning Implementation:

- Shared input x .
- Intermediate shared representation h_{shared} .
- Top-level hidden units $h^{(i)}$ for each task-specific output $y^{(i)}$.

Underlying Principle:

Shared parameters are constrained towards values that generalize across tasks, improving **statistical strength** and reducing overfitting.

Illustration / Explanation:

- Shared representation captures **common factors** across tasks.
- Task-specific units handle **variations unique to individual tasks**.
- No loss function is included in this explanation, focusing on architecture and conceptual understanding.

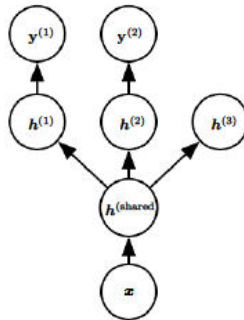


Figure 7.2: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters (associated respectively with the weights into and from $h^{(1)}$ and $h^{(2)}$) can be learned on top of those yielding a shared representation $h^{(shared)}$. The underlying assumption is that there exists a common pool of factors that explain the variations in the input x , while each task is associated with a subset of these factors. In this example, it is additionally assumed that top-level hidden units $h^{(1)}$ and $h^{(2)}$ are specialized to each task (respectively predicting $y^{(1)}$ and $y^{(2)}$) while some intermediate-level representation $h^{(shared)}$ is shared across all tasks. In the unsupervised learning context, it makes sense for some of the top-level factors to be associated with none of the output tasks ($h^{(3)}$): these are the factors that explain some of the input variations but are not relevant for predicting $y^{(1)}$ or $y^{(2)}$.

(refer to textbook)

5. Develop a detailed answer explaining Parameter Tying and Parameter Sharing in deep learning. Emphasize how these techniques help in reducing memory usage and improving model generalization.

10 m

A. Parameter Tying

1. **Definition:** Parameter tying enforces **soft constraints** where corresponding parameters of different models are encouraged to be similar but not strictly equal.
2. **Scenario:** Two models A and B with parameters $w^{(A)}$ and $w^{(B)}$ performing similar tasks.
3. **Regularization:** Using an L2 penalty to tie parameters:

$$\Omega(w^{(A)}, w^{(B)}) = \|w^{(A)} - w^{(B)}\|_2^2$$

This term is added to the loss to encourage corresponding parameters to be close.

4. **Applications:** For example, one model trained supervised and the other unsupervised; tied parameters transfer knowledge between models.

5. Advantages:

- Improves generalization.
- Incorporates domain knowledge about parameter dependencies.

B. Parameter Sharing

1. **Definition:** Parameter sharing enforces **hard equality constraints** among sets of parameters across multiple model components.
2. **Memory Efficiency:** Only the unique set of parameters is stored, significantly reducing memory footprint.
3. **CNN Example:**
 - Translational invariance: a cat detector (set of weights) applies across all positions in an image.
 - Same convolutional filter slides across input regions.
4. **Impact:**
 - Dramatically reduces the number of unique parameters.
 - Allows larger networks without increasing training data proportionally.
5. **Difference from Parameter Tying:**
 - Tying: soft closeness constraint ($\|w^{(A)} - w^{(B)}\|^2$).
 - Sharing: hard equality constraint (same unique parameters applied repeatedly).

explain the equation as well

6. Develop a clear explanation of how machine learning optimization differs from pure optimization. Explain the concept of Empirical Risk Minimization (ERM) and describe why deep learning optimizes surrogate cost functions instead of the true risk. 10m

1. Learning in Machine Learning vs. Pure Optimization:

- **Pure optimization** involves directly minimizing a specific objective function $J(\theta)$. The goal is solely to find θ^* that minimizes $J(\theta)$ without reference to any external performance measure.
- **Learning in machine learning**, on the other hand, usually optimizes a surrogate cost function to improve some performance measure P (e.g., accuracy, F1-score), which is often **intractable or defined on the test set**.
- Formally, in machine learning, we define a model $f(x; \theta)$ that predicts y from input x . We then define a per-example loss function $L(f(x; \theta), y)$ and a cost function over the training set:

$$J(\theta) = \mathbb{E}_{(x,y) \sim \hat{p}_{\text{data}}} [L(f(x; \theta), y)] = \frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)})$$

- Here, $\hat{p}_{\text{data}}$ is the empirical distribution derived from the finite training set of m examples.
- Unlike pure optimization, the goal is **not to minimize the training error directly** but to reduce expected generalization error on unseen data.

2. Expected Risk (True Risk):

- The **expected risk** or true risk represents the quantity we ideally want to minimize:

$$J^*(\theta) = \mathbb{E}_{(x,y) \sim p_{\text{data}}} [L(f(x; \theta), y)]$$

- Here, the expectation is taken over the true underlying distribution $p_{\text{data}}(x, y)$.
- If we had access to p_{data} , minimizing $J^*(\theta)$ would be a **pure optimization problem**.
- However, since p_{data} is unknown, we only have a **finite training set**, which leads to the **empirical risk minimization (ERM)** approach.

3. Empirical Risk Minimization (ERM):

- ERM replaces the unknown true distribution with the empirical distribution from the training set:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)})$$

- Minimizing $J(\theta)$ is intended to reduce the expected generalization error $J^*(\theta)$.
- **Assumption:** The training data is representative of the true distribution.
- **Limitation:** ERM can lead to **overfitting** for models with high capacity, as they may memorize the training set without generalizing well.

4. Indirect Optimization in Deep Learning:

- In deep learning, optimization often occurs **indirectly** because the true performance measure P (like test accuracy) cannot be directly minimized.
- Instead, we minimize a **surrogate cost function** $J(\theta)$ (e.g., cross-entropy loss, mean squared error) that is differentiable and suitable for gradient-based optimization:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$$

- Challenges in deep learning include:
 - **Non-differentiable loss functions** (e.g., 0-1 loss)
 - **Overfitting** to finite data
 - **Indirect relation** between the loss and the true performance measure

5. Key Differences between Learning and Pure Optimization:

Aspect	Pure Optimization	Learning in ML
Objective	Minimize $J(\theta)$ exactly	Optimize a surrogate $J(\theta)$ to improve P
Target	Function value only	Generalization performance on unseen data
Data	May not involve data	Empirical data used to estimate risk
Challenges	Well-defined problem	Indirect optimization, overfitting, non-differentiable losses
Approach	Direct minimization	Empirical Risk Minimization or other surrogate optimization

7. Apply your knowledge of Stochastic Gradient Descent (SGD) to explain its algorithm, learning rate schedule, convergence conditions, differences from batch gradient descent, and practical advantages for large datasets. 10 m

Stochastic Gradient Descent (SGD) is a widely used optimization algorithm in machine learning and deep learning. It is particularly effective for training models on large datasets because it updates parameters incrementally using small subsets (minibatches) of data, rather than the entire dataset at once.

1. Algorithmic Steps of SGD

At iteration k , the SGD algorithm updates the parameters θ as follows:

1. Sample a minibatch of m examples $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
2. Compute an unbiased gradient estimate over the minibatch:

$$\hat{g} = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$$

3. Update the parameters using the learning rate ϵ_k :

$$\theta \leftarrow \theta - \epsilon_k \hat{g}$$

4. Repeat until a stopping criterion is met (e.g., maximum iterations or minimal change in cost).

Key Point: SGD approximates the true gradient of the cost function using a small subset of data, which introduces stochastic noise into the updates.

(refer to textbook for algorithm)

2. Learning Rate (ϵ_k) and Scheduling

- The **learning rate** controls the step size in parameter space.
- Fixed learning rates may cause instability in SGD due to stochastic noise from minibatch gradients.
- To improve stability, the learning rate is gradually decreased over time:

Linear decay schedule:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau, \quad \alpha = \frac{k}{\tau}$$

- After iteration τ , ϵ_k is often kept constant.
- Practical selection:
 - ϵ_0 too large $\rightarrow$ violent oscillations, cost may increase.
 - ϵ_0 too small $\rightarrow$ slow learning, risk of being stuck in high-cost regions.
 - Typically, $\epsilon_\tau \approx 1\%$ of ϵ_0 and τ is set to allow several passes through the dataset.

Observation: Monitoring learning curves helps choose the best ϵ_0 for rapid yet stable convergence.

3. Convergence Conditions

To guarantee convergence of SGD to a minimum (or a neighborhood of it), the learning rates must satisfy:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

Explanation:

- The first condition ensures the algorithm continues to make progress and does not get stuck prematurely.
- The second condition prevents divergence due to the stochastic nature of the minibatch gradients.

4. Comparison with Batch Gradient Descent

Aspect	Batch Gradient Descent	Stochastic Gradient Descent
Gradient computation	Full dataset	Minibatch of examples
Learning rate	Can be fixed	Needs scheduling/decay
Convergence	Smooth, better asymptotic rate	Noisy, slower asymptotic, but faster initial progress
Computational cost	High per update	Low per update, independent of dataset size
Large datasets	May be infeasible	Highly scalable, can converge before full dataset processed

- **Excess error:**
 - Convex problem: $O\left(\frac{1}{\sqrt{k}}\right)$ after k iterations
 - Strongly convex: $O\left(\frac{1}{k}\right)$
- Batch GD enjoys better theoretical convergence but is often less practical for very large datasets.

5. Practical Considerations for Large Datasets

1. **Efficiency:** Computation per update does not grow with dataset size.
2. **Rapid initial progress:** Even evaluating gradients on very few examples can significantly reduce cost in early iterations.
3. **Trade-off with batch GD:** Minibatch size can be gradually increased to balance fast initial learning with stability.
4. **Monitoring:** Adjust learning rate based on early learning curves to avoid instability or slow progress.
5. **Large-scale advantage:** SGD allows training on massive datasets that batch methods cannot handle efficiently.

8. Make use of the momentum method to explain Stochastic Gradient Descent (SGD) with key equations and its practical implications in training large datasets. 10m

1. Introduction to Momentum in SGD:

- Stochastic Gradient Descent (SGD) is widely used in deep learning for optimizing the cost function.
- SGD can be slow, particularly in areas with high curvature or noisy gradients.
- **Momentum** accelerates SGD by accumulating a moving average of past gradients, allowing faster convergence in the relevant directions and reducing oscillations across narrow valleys.

2. Momentum Algorithm:

- **Velocity update equation:**

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$$

- **Parameter update equation:**

$$\theta \leftarrow \theta + \mathbf{v}$$

Where:

- $\alpha \in [0, 1)$ is the momentum hyperparameter controlling the influence of previous gradients.
- ϵ is the learning rate.
- m is the minibatch size.
- L is the loss function.
- $\mathbf{v}$ is the velocity vector representing accumulated momentum.

3. Algorithm Reference (Algorithm 8.2):

Stochastic Gradient Descent with Momentum

1. Initialize parameters θ and velocity $\mathbf{v} = 0$
2. Set learning rate ϵ and momentum α
3. While stopping criterion not met:
 - Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$
 - Compute gradient:

$$g \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$$

- Update velocity: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon g$
 - Update parameters: $\theta \leftarrow \theta + \mathbf{v}$
4. End while

(refer the fig in text book)

4. Physical Analogy (Figure 8.5 Reference):

- Momentum is analogous to a particle moving through a surface under forces:
 - The negative gradient acts as a force pushing the particle downhill.
 - Previous velocity accumulates, helping the particle move faster along consistent directions.
 - Viscous drag (force proportional to $-v$) prevents indefinite oscillations and ensures convergence.
- Figure 8.5 shows how momentum allows the algorithm to traverse a narrow, curved valley more efficiently than plain gradient descent.

5. Practical Implications:

- Momentum reduces oscillations in steep and narrow valleys, speeding up convergence.
- It helps SGD escape shallow local minima by maintaining directional velocity.
- Commonly used hyperparameter values: $\alpha = 0.5, 0.9, 0.99$.
- Works well with large datasets because computation per update depends on minibatch size, not the total dataset.

9. Make use of the concept of convolution to explain its operation in neural networks, including 1-D and 2-D convolution, the role of input, kernel and feature map, and the difference between convolution and cross-correlation 10m

Convolution is a mathematical operation where we combine two functions to produce a third function that shows how one function modifies or “filters” the other.

- In neural networks, we use a **small filter (kernel)** to slide over the input (like an image) and calculate a weighted sum of the overlapping values to create a new output called a **feature map**.
- Essentially, it helps the network **detect patterns or features** from the input.

- **Continuous case:**

For input $x(t)$ and weighting function $w(t)$ (kernel), the smoothed output $s(t)$ is:

$$s(t) = (x * w)(t) = \int x(a)w(t - a) da$$

Here, $x(t)$ is the input (e.g., noisy sensor measurements), $w(t)$ is the weighting function giving more importance to recent measurements, and $s(t)$ is the smoothed output.

- **Discrete case:**

For integer time indices, the convolution becomes:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t - a)$$

2. 2-D Convolution

For 2-D inputs such as images $I(i, j)$ and 2-D kernels $K(m, n)$:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n)$$

- **Input I :** The image or 2-D tensor.
- **Kernel K :** A small 2-D array of parameters learned by the model.
- **Feature map S :** Output representing the result of applying the kernel to the input.

Commutative property:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n)$$

- Flipping the kernel achieves commutativity.

Cross-correlation (commonly used in ML libraries):

$$S(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

- No flipping of the kernel. Libraries often call this convolution.

3. Implementation Considerations

- In practice, input and kernel are finite, so the infinite sums are computed over the stored array elements.
- Convolution is applied over multiple axes for multidimensional data (tensors).
- Discrete convolution can be represented as a sparse matrix multiplication, e.g., Toeplitz matrix in 1-D or doubly block circulant in 2-D.
- Only positions where the kernel fully overlaps the input are considered for "valid" convolution.

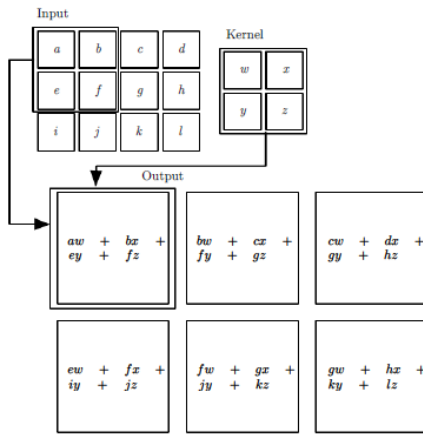


Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called “valid” convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

(refer textbook)

4. Figure Reference

Figure 9.1:

- Shows a 2-D convolution without kernel flipping.
- Each element of the output tensor is obtained by applying the kernel to a corresponding region of the input tensor.
- Arrows indicate contributions from input elements to the output.

10. Build a conceptual understanding of how convolution and pooling operations function as infinitely strong priors that determine local interactions and translation properties in CNNs. 10m

1. Concept of Prior in Learning Models (2 marks)

- A **prior** in machine learning expresses what we already believe about model parameters **before seeing data**.
- It helps guide learning toward certain patterns or relationships.
- **Weak priors** allow parameters to vary freely, while **strong priors** tightly restrict parameter values.
- An **infinitely strong prior** completely forbids some parameter values, assigning them zero probability regardless of data.

2. Infinitely Strong Prior in CNNs (2 marks)

- In **Convolutional Neural Networks (CNNs)**, both **convolution** and **pooling** can be understood as applying **infinitely strong priors** to the model’s weights.
- These priors **restrict how connections and weights are structured**, ensuring only specific, spatially meaningful configurations are allowed.
- This means the model cannot freely learn any relationship—it must follow the rules of locality and translation consistency.

3. Convolution as an Infinitely Strong Prior (3 marks)

- The **convolution operation** assumes that **nearby input features are related**.
- This introduces a prior that says:
 - Each filter (kernel) uses the **same set of weights** across all image locations.
 - Each neuron only looks at a **small local region** of the input (its receptive field).
- This weight-sharing rule acts as an infinitely strong prior because it **forces** the model to learn features that are **local and translation-equivariant** (i.e., shifting the input shifts the output in the same way).
- Thus, convolution limits what the model can learn, ensuring it focuses on local spatial patterns like edges, corners, or textures.

4. Pooling as an Infinitely Strong Prior (2 marks)

- **Pooling operations** (like max or average pooling) introduce a prior that enforces **translation invariance**.
- This means the model becomes **insensitive to small shifts or distortions** in the input image.
- Pooling summarizes information over small neighborhoods, assuming the exact position of a feature is less important than its presence.
- However, this can sometimes cause **underfitting** when tasks require precise spatial detail.

5. Key Insights and Practical Implications (1 mark)

- Convolution and pooling together help CNNs learn **efficient, spatially structured representations**.
- They reduce the number of learnable parameters, improving **generalization** and **training speed**.
- But these priors are only useful when their assumptions (locality and invariance) match the task — otherwise, they may limit performance.
- For example, some modern CNNs use **partial pooling** to retain detailed spatial information.

11. Construct a conceptual explanation of how convolutional networks can be utilized to generate structured outputs such as pixel-wise labeling in images, highlighting the role of recurrent convolutional networks in refining label predictions. 10m

1. Concept of Structured Outputs (2 marks)

- Traditional convolutional networks (CNNs) usually produce a **single class label** (for classification) or a **single numeric value** (for regression).
- In contrast, **structured output models** produce **high-dimensional, spatially organized outputs**, often in the form of a **tensor**.
- For example, a CNN might output a tensor S , where each element $S_{i,j,k}$ represents the **probability that pixel (j, k) belongs to class i**.
- This enables **pixel-wise classification**, allowing precise **image segmentation** or **object masking**.

2. Reduction in Spatial Dimensions (1.5 marks)

- During convolution and pooling operations, the **spatial size** of feature maps can shrink compared to the input image.
- This causes the **output plane** to be smaller than the input plane.
- Solutions to this issue include:
 - **Avoiding pooling** layers altogether.
 - **Using unit-stride pooling operators**.
 - **Generating a lower-resolution grid of labels** (as proposed by Pinheiro & Collobert, 2014, 2015).

3. Refinement through Recurrent Convolutional Networks (3 marks)

- Instead of predicting labels in one pass, the network can **iteratively refine** pixel labels using information from **neighboring pixels**.
- This is achieved through a **Recurrent Convolutional Network (RCNN)** architecture, where:
 - The initial estimate $\hat{Y}^{(1)}$ is produced.
 - Subsequent estimates $\hat{Y}^{(2)}, \hat{Y}^{(3)}, \dots$ refine this prediction.
 - The **same parameters (weights)** are reused at each step, forming a **recurrent loop**.
- This approach allows the model to **progressively improve** segmentation accuracy and **capture spatial dependencies** across pixels.

4. Figure Reference (2 marks)

- **Refer to Figure 9.17** for a schematic of a recurrent convolutional network used for pixel labeling.
- The figure shows:
 - Input image tensor X (rows, columns, and channels).
 - Hidden representations $H(1), H(2), H(3)$ computed using convolution kernels U .
 - Output label tensors $\hat{Y}^{(1)}, \hat{Y}^{(2)}, \hat{Y}^{(3)}$ refined iteratively.
 - Shared kernels V and W used across steps, signifying parameter reuse.
- This demonstrates how CNNs can perform **structured prediction by recurrent refinement**.

5. Post-processing and Segmentation Refinement (1.5 marks)

- Once pixel-wise predictions are made, **post-processing techniques** group contiguous pixels with similar labels.
- **Graphical models** (e.g., Conditional Random Fields) can model the **probabilistic relationships** between neighboring pixels.
- Alternatively, CNNs can be trained to **approximate graphical model objectives**, integrating spatial coherence directly into the learning process.

6. Key Insights and Advantages (1 mark)

- Structured output CNNs provide **fine-grained predictions** (e.g., per-pixel object segmentation).
- The **recurrent refinement mechanism** improves **accuracy and spatial consistency**.
- They bridge the gap between **CNNs for recognition** and **graphical models for structured prediction**.

12. Utilize your understanding of convolutional networks to explain how they handle different data types and variable-sized inputs 10m

1. Introduction to Data Types in CNNs (2 marks)

- Convolutional Neural Networks (CNNs) are designed to process data that contains **spatial or temporal structure**, often organized as **multi-dimensional arrays (tensors)**.
- Each **channel** in the data corresponds to a different observed quantity (e.g., RGB color channels or audio frequencies).
- CNNs can efficiently handle **single-channel** (e.g., grayscale image, audio signal) or **multi-channel** (e.g., RGB image, color video) data.

2. Examples of Data Types (Table 9.1 Reference) (3 marks)

Dimensionality	Single Channel Example	Multi-Channel Example	Explanation	
1-D	Audio waveform (time-series data)	Skeleton animation data	Convolution is applied along time to capture temporal dependencies.	
2-D	2D Fourier-transformed audio data	Color image data (R, G, B channels)	Convolution over height and width provides translation equivariance in both spatial directions.	
3-D	Volumetric data (CT scans)	Color video data (time, height, width)	Convolution is applied across spatial and temporal dimensions for motion and depth information.	

3 Handling Variable-Sized Inputs

- CNNs can process inputs of **different spatial or temporal dimensions**, unlike fully connected networks.
- Convolution kernels are applied **repeatedly**, adapting to input size.
- The **output feature map** scales proportionally to input dimensions.
- Enables CNNs to handle varying image resolutions, audio lengths, or video frames.

4 Strategies for Fixed-Size Outputs (Ref: Figure 9.11)

- When a fixed-size output is needed:
 - **Adaptive pooling layers** adjust pooling region size proportional to input size.
 - Maintains **consistent number of pooled outputs**.
 - Avoiding or modifying pooling can preserve **spatial details** (useful for segmentation).
- Figure 9.11 illustrates how pooling regions scale dynamically for variable-sized inputs.

5 Valid and Invalid Use-Cases of Convolution

- **Valid:** Inputs with variable length but same type of observations (e.g., different image widths, audio sequences).
- **Invalid:** Inputs with variable types (e.g., missing features like test scores in college applications).
- Convolution assumes **same kind of observations across the data**.

13. Build a comprehensive understanding of the historical development of convolutional networks, highlighting their early applications, successes, and contributions to modern deep learning. Discuss their advantages over fully connected networks and their role in processing structured data. 10m

Content Points	Expected Answer / Explanation	Marks
1. Introduction to Convolutional Networks in Deep Learning History	- Convolutional networks (CNNs) were key in early deep learning success. - Inspired by insights from the brain, leading to specialized architectures for grid-structured data (images, etc.).	2
2. Early Applications and Commercial Success	- AT&T in 1990s: CNNs for reading checks. - NEC deployed the system processing >10% of US checks by the end of 1990s. - Microsoft OCR and handwriting recognition systems using CNNs (Simard et al., 2003). - CNNs used in multiple contests before major commercial impact.	2
3. Success in Competitions	- Krizhevsky et al., 2012: CNNs won ImageNet object recognition challenge, sparking current commercial interest in deep learning. - CNNs are computationally efficient: easier to run multiple experiments, tune hyperparameters.	1
4. Advantages over Fully Connected Networks	- Allowed training of larger networks successfully with limited hardware at the time. - Better suited for grid-structured data than fully connected networks.	2
5. Contribution to Modern Deep Learning	- Paved the way for acceptance of neural networks. - Specialized neural networks for images (2D grid topology). - Led to development of other specialized networks (e.g., RNNs for sequential data). - CNNs bridged early neural network research to modern deep learning.	2
6. Summary / Key Insights	- Remain at the forefront of image-based applications. - Played a critical role in shaping deep learning research and deployment.	1

14. Utilize the concepts of sparse interactions, parameter sharing, and equivariant representations to explain how convolutional networks improve computational efficiency and handle variable-sized inputs. 10 m

- **Sparse Interactions (Sparse Connectivity / Sparse Weights)**

- In traditional neural networks, every output unit interacts with every input unit → requires $m \times n$ parameters and $O(m \times n)$ computation.
- Convolution uses **kernels smaller than the input** (e.g., detecting edges in an image) → reduces parameters to $k \times n$ ($k \ll m$).

- Improves **memory efficiency** and **statistical efficiency** while reducing computation.
- **Figures Reference:** Figure 9.2 (viewed from below) and Figure 9.3 (viewed from above) illustrate sparse connectivity.
- Deep layers can indirectly connect to more of the input (Figure 9.4), allowing complex interactions to be constructed from simple sparse blocks.

• Parameter Sharing (Tied Weights)

- Same parameter is used across different positions in the input → reduces storage further.
- Instead of learning separate parameters for each location, **one kernel is shared**.
- Reduces parameters from $k \times n$ to just **k parameters**, while maintaining **$O(k \times n)$ computation**.
- **Figure Reference:** Figure 9.5 shows parameter sharing in a convolutional network versus a fully connected network.

• Equivariant Representations

- A function is **equivariant** if a transformation in input results in the same transformation in output.
- Convolution ensures **translation equivariance**:
 - Shifting an object in the input shifts the feature map by the same amount.
 - Useful for images and time series (e.g., detecting edges, patterns, or events regardless of position).
- **Limitation:** Convolution is not naturally equivariant to scale or rotation → additional mechanisms required for these transformations.

• Handling Variable-Sized Inputs

- Traditional fully connected networks require **fixed-size input**, limiting flexibility.
- Convolutional networks can process inputs of different sizes:
 - Kernel is applied repeatedly across input → output size adjusts accordingly.
 - Can be combined with pooling layers (e.g., adaptive pooling) to produce fixed-size outputs if needed.
- **Figure Reference:** Figure 9.6 illustrates the efficiency improvement using sparse connectivity and parameter sharing.

• Summary / Practical Advantages

- Convolutional networks **reduce computation and memory** compared to fully connected networks.
- **Leverage local structure** in data efficiently.
- **Enable translation equivariance** → detect features regardless of location.
- Can process **variable-sized inputs** → flexible for images, audio, and video data.

1. Utilize the idea of unfolding to show how a recurrent equation becomes an unfolded computational graph. Use equations and Figures to support your answer. (10 Marks)

1. Recurrence in Equation 10.1 (2 Marks)

The system is defined by the recurrent equation:

$$s(t) = f(s(t-1); \theta) \quad (\text{Equation 10.1})$$

- It is *recurrent* because the value at time t depends on the value at $t-1$.
- The same transition function f and parameters θ are reused at every time step.

2. Unfolding for $\tau = 3$ (using Equations 10.2 & 10.3) (2 Marks)

Unfolding for 3 steps removes recurrence by repeatedly applying Equation 10.1:

$$s(3) = f(s(2); \theta) \quad (10.2)$$

$$s(2) = f(s(1); \theta)$$

Substituting:

$$s(3) = f(f(s(1); \theta); \theta) \quad (10.3)$$

- Now the expression contains no **recurrence** and can be drawn as a chain-like computational graph.

3. Unfolded computational graph explanation (Fig. 10.1) (2 Marks)

- Figure 10.1 shows the unfolded graph for Equation 10.1.
- Each node represents $s(t-1)$, $s(t)$, $s(t+1)$.
- Arrows represent repeated applications of function f .
- The same parameters θ are used at *each* time step $\rightarrow$ **parameter sharing**.



Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of θ used to parametrize f) are used for all time steps.

4. RNN recurrence with input-driven dynamics (Eq. 10.5, Fig. 10.2) (2 Marks)

When an external signal is included:

$$s(t) = f(s(t-1), x(t); \theta) \quad (10.4)$$

Using h for hidden state:

$$h(t) = f(h(t-1), x(t); \theta) \quad (10.5)$$

- Figure 10.2 (Left) shows the RNN as a *circuit diagram* with a delay unit (black square).
- Figure 10.2 (Right) shows the *unfolded graph*, where each time step has its own node $h(t)$, $h(t+1)$, etc.

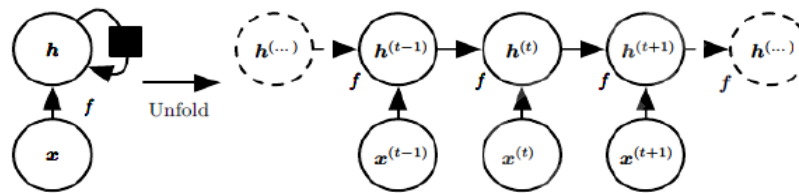


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time. (Left) Circuit diagram. The black square indicates a delay of a single time step. (Right) The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

5. Advantages of unfolding (using Equations 10.6 & 10.7) (2 Marks)

The unfolded recurrence can be expressed as:

$$h(t) = g^{(t)}(x(t), x(t-1), \dots, x(1)) \quad (10.6)$$

where:

$$g^{(t)} = f(f(\dots f(h(0), x(1); \theta), \dots)) \quad (10.7)$$

Unfolding gives two major benefits:

1. Fixed-size model

- Regardless of sequence length, the transition function f always takes a fixed-size input.

2. Parameter sharing

- The *same* parameters θ are reused at every time step $\rightarrow$ enables learning with fewer examples and generalization to longer sequences.

2. Apply the equations to show how an RNN works at each time step. Explain how the hidden state is updated, how the output is made, and how the total loss for the whole sequence is calculated. (10 Marks)

1. RNN architecture & parameters (Figure 10.3) (2 Marks)

Figure 10.3 depicts an RNN with:

- $U \rightarrow$ input-to-hidden weights
- $W \rightarrow$ hidden-to-hidden recurrent connection
- $V \rightarrow$ hidden-to-output weights
- $b, c \rightarrow$ bias vectors for hidden and output layers

The unfolded graph in Fig. 10.3 (right side) shows that **each time step has its own node**, but all share the same parameters (U, V, W).

This is the foundation of all RNN models.

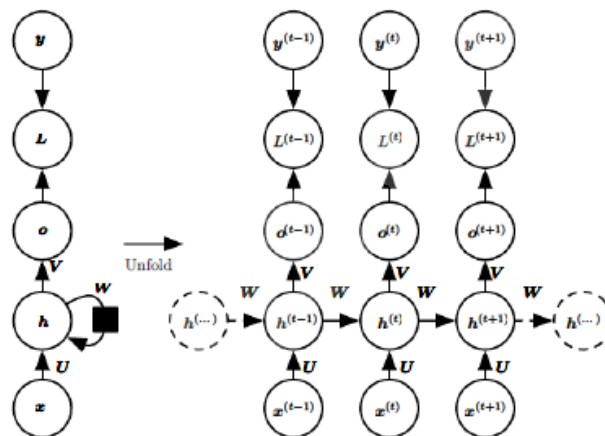


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output o values. A loss L measures how far each o is from the corresponding training target y . When using softmax outputs, we assume o is the unnormalized log probabilities. The loss L internally computes $\hat{y} = \text{softmax}(o)$ and compares this to the target y . The RNN has input to hidden connections parametrized by a weight matrix U , hidden-to-hidden recurrent connections parametrized by a weight matrix W , and hidden-to-output connections parametrized by a weight matrix V . Equation 10.8 defines forward propagation in this model. (Left) The RNN and its loss drawn with recurrent connections. (Right) The same seen as an time-unfolded computational graph, where each node is now associated with one particular time instance.

2. Hidden pre-activation computation (Equation 10.8) (2 Marks)

Forward propagation begins from an initial hidden state $h(0)$.

For each time step:

$$a(t) = b + Wh(t-1) + Ux(t) \quad (10.8)$$

- $h(t-1)$ carries information from the past
- $x(t)$ is the input at the current time
- W enables *recurrence*, making RNNs memory-based models

This equation collects the influence of both previous state and current input.

3. Hidden activation & output computation (Equations 10.9 & 10.10) (2 Marks)

Hidden activation

$$h(t) = \tanh(a(t)) \quad (10.9)$$

The hyperbolic tangent gives each hidden state a nonlinear representation of the time sequence up to t .

Output logits

$$o(t) = c + Vh(t) \quad (10.10)$$

These unnormalized output values are used to compute discrete predictions (e.g., next word or character).

4. Probability distribution using softmax (Equation 10.11) (2 Marks)

$$\hat{y}(t) = \text{softmax}(o(t)) \quad (10.11)$$

- Converts raw scores $o(t)$ into normalized probabilities
- Allows the model to represent discrete categories (e.g., vocabulary tokens)

Figure 10.3 assumes $o(t)$ contains **logits**, and softmax produces the distribution used for prediction.

5. Sequence-level loss & BPTT (Equations 10.12–10.14) (2 Marks)

The RNN outputs a loss **per time step**: $L(t)$.

The total loss of a sequence:

$$L(\{x(1), \dots, x(\tau)\}, \{y(1), \dots, y(\tau)\}) = \sum_t L(t) \quad (10.13)$$

If negative log-likelihood is used:

$$L = - \sum_t \log p_{\text{model}}(y(t)|x(1), \dots, x(t)) \quad (10.14)$$

- p_{model} is obtained from $\hat{y}(t)$
- Training requires **Back-Propagation Through Time (BPTT)**:
 - Forward pass from left→right (unfolded graph)
 - Backward pass right→left
 - Both require $O(\tau)$ time and memory

Thus, RNNs are powerful but computationally expensive.

3. Apply your understanding of teacher forcing and output-recurrence RNNs to explain how the model works during training and testing. Identify why teacher forcing is needed, apply the maximum-likelihood equations, and explain how the absence of hidden-to-hidden recurrence affects learning and prediction. (10marks)

1. Explanation of Output-Recurrence RNN and Absence of Hidden-to-Hidden Connections (2 marks)

Answer (1):

1. The network uses recurrent connections only from the output $o(t)$ to the hidden layer $h(t+1)$ (shown in Figure 10.4).
2. Since hidden-to-hidden recurrence is absent, the network:
 - Cannot store long-term dependencies.
 - Is less powerful and cannot simulate a universal Turing machine.
3. The output units must capture all past information, which is difficult for complex sequences.

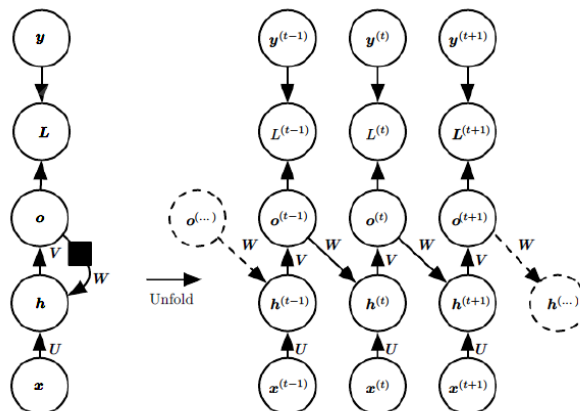


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t , the input is x_t , the hidden layer activations are

2. Teacher Forcing Using Figure 10.6 (2 marks)

Answer (2):

1. **Teacher forcing** feeds the **correct target output $y(t)$** into the network at the next time step during training instead of the model's own output $o(t)$.
2. Figure 10.6 (Left) shows this:
 - Training uses $y(t) \rightarrow h(t+1)$, stabilizing learning.
3. Figure 10.6 (Right) shows test time:
 - The true target is unknown, so $o(t)$ is fed back.

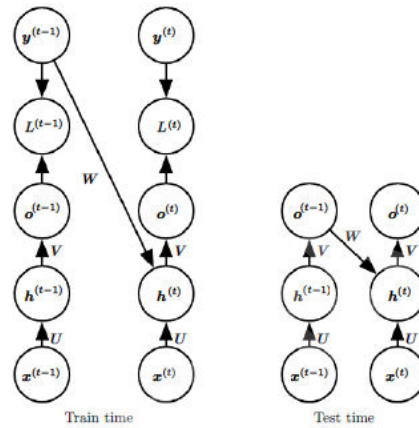


Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. (Left) At train time, we feed the *correct output* $y^{(t)}$ drawn from the train set as input to $h^{(t+1)}$. (Right) When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $y^{(t)}$ with the model's output $o^{(t)}$, and feed the output back into the model.

3. Maximum-Likelihood Decomposition (Equations 10.15–10.16) (2 marks)

Answer (3):

1. For a two-step sequence, maximum likelihood is:

$$\log p(y^{(1)}, y^{(2)} | x^{(1)}, x^{(2)}) \quad (10.15)$$

2. It decomposes as:

$$\log p(y^{(2)} | y^{(1)}, x^{(1)}, x^{(2)}) + \log p(y^{(1)} | x^{(1)}, x^{(2)}) \quad (10.16)$$

3. This shows that $y(1)$ must be provided to predict $y(2)$ during training, which directly supports the use of **teacher forcing**.

4. Parallelization Due to No Hidden-to-Hidden Recurrence (2 marks)

Answer (4):

1. With no hidden-to-hidden connections, **each time step is independent** during training.
2. The loss $L(t)$ depends only on prediction at time t , so:
 - Gradients for each time step can be computed separately.
 - Training becomes **parallelizable**, unlike standard RNNs requiring full BPTT.

5. Train–Test Mismatch and Mitigation Strategies (2 marks)

Answer (5):

1. **Mismatch arises** because:
 - Training uses $y(t)$ (ground truth).
 - Testing uses $\hat{o}(t)$ (model's prediction).
 - Leads to error accumulation and exposure bias.
2. **Mitigation strategies:**
 - Mix teacher-forced and **free-running** predictions during training.
 - Predict future targets using **unfolded output-to-input paths**.
 - Use **scheduled sampling** (Bengio et al., 2015): gradually replace true outputs with generated outputs using curriculum learning.

4. **Apply your understanding of RNNs as directed graphical models to model how an RNN represents the probability of a sequence. Identify how hidden states help reduce parameters, and explain how an RNN decides when to stop generating a sequence.**

1. RNNs as Directed Graphical Models

An RNN can be interpreted as a **directed graphical model** in which each output $y(t)$ depends on all previous outputs $y(1), \dots, y(t-1)$.

- *Figure 10.7* illustrates this by showing a **fully connected graph**, where every past observation may influence future values.
- This allows the RNN to represent long-term dependencies in a sequence.

2. How an RNN Represents the Probability of a Sequence

For a sequence $Y = \{y(1), y(2), \dots, y(\tau)\}$, an RNN models the joint probability using the **chain rule**, as given in Equation (10.31):

$$P(Y) = \prod_{t=1}^{\tau} P(y(t) \mid y(t-1), y(t-2), \dots, y(1)) \quad (10.31)$$

This expresses the sequence as a series of **one-step conditional probabilities**.

Thus, the RNN estimates the probability of each element given all past elements.

3. Negative Log-Likelihood Used for Training

The overall loss for the sequence is:

$$L = \sum_t L(t) \quad (10.32)$$

where each step's contribution is:

$$L(t) = -\log P(y(t) \mid y(t-1), y(t-2), \dots, y(1)) \quad (10.33)$$

This means the RNN is trained to **maximize the conditional probability** of each step given its history.

4. How Hidden States Reduce Parameters (Figure 10.8)

Instead of explicitly modeling all dependencies between past values (which would require $O(k^\tau)$ parameters), an RNN introduces a hidden state $h(t)$.

- *Figure 10.8* shows how $h(t)$ serves as a **compact summary** of all past inputs.
- Because the same function and parameters are reused at every time step, the number of parameters becomes $O(1)$ (constant w.r.t. sequence length).
- This dramatically improves efficiency while still keeping the model expressive.

Thus, hidden states allow the RNN to **capture long-term relationships** without needing a separate set of parameters for each time step.

5. How an RNN Decides When to Stop Generating a Sequence

The RNN must determine the sequence length during sampling. The model can stop in several ways:

1. End-of-sequence symbol (EOS)

- A special symbol is added to the vocabulary.
- When generated, it signals the end of the sequence.
- Common in text generation.

2. Bernoulli "continue/stop" output

- An additional sigmoid output decides whether to continue or halt at each time step.
- Trained using cross-entropy.

3. Predicting τ directly

Using Equation (10.34):

$$P(x(1), \dots, x(\tau)) = P(\tau) P(x(1), \dots, x(\tau) \mid \tau) \quad (10.34)$$

- The model samples a value of τ first.
- Then generates exactly τ time steps.
- Requires giving the RNN extra input so it knows how close it is to the end.

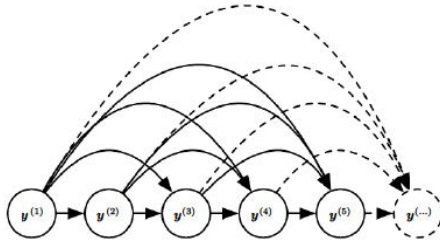


Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (as in equation 10.6) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in figure 10.8.

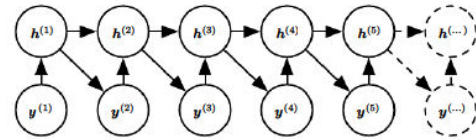


Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $h^{(t)}$ and $y^{(t)}$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

5. Model how a Bidirectional RNN uses both past and future information to make predictions. Explain how this approach can be extended to process 2-D inputs such as images. (10 marks)

1. Explanation of Need for Bidirectional RNNs (2 marks)

Expected Answer:

- Traditional RNNs have a **causal structure**, where the state at time t depends only on past inputs $x(1), \dots, x(t-1)$ and current input $x(t)$.
- However, in tasks like **speech recognition** or **handwriting recognition**, the correct interpretation of the current element $y(t)$ often depends on **future inputs** due to co-articulation or contextual linguistic cues.
- Therefore, a model must access **both past and future context** to resolve ambiguity.

2. Structure of Bidirectional RNNs (with Figure Reference) (2 marks)

Expected Answer:

- A Bidirectional RNN consists of **two sub-RNNs**:
 - A **forward RNN** with hidden state $h(t)$ processing the sequence from start $\rightarrow$ end.
 - A **backward RNN** with hidden state $g(t)$ processing the sequence from end $\rightarrow$ start.
- Figure 10.11 shows this architecture where both $h(t)$ and $g(t)$ provide information to the output unit $o(t)$.
- This allows the model to compute an output that is sensitive to signals around time t from **both directions**.

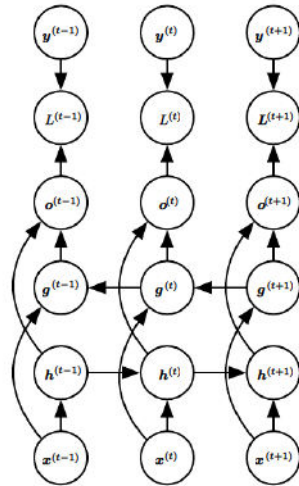


Figure 10.11: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences x to target sequences y , with loss $L^{(t)}$ at each step t .

3. Working Principle: Combining Forward and Backward States (2 marks)

Expected Answer:

- The output unit $o(t)$ uses **both** hidden states:

$$o(t) = f(h(t), g(t))$$

- This combination gives the model access to:
 - Past information** through $h(t)$
 - Future information** through $g(t)$
- The system **does not need a fixed-size window**, unlike CNNs or feed-forward nets.
- The model becomes most sensitive to inputs near time t but can utilize long-range dependencies when needed.

4. Applications in Sequence Tasks (2 marks)

Expected Answer:

- Bidirectional RNNs are very successful in:
 - Speech recognition** (Graves et al., 2013; co-articulation effects)
 - Handwriting recognition** (Graves et al., 2008)
 - Bioinformatics** (Baldi et al., 1999)
- They help disambiguate inputs because the output depends on **complete input sequence**, not just its prefix.

5. Extension to 2-D Data Using Four Directional RNNs (2 marks)

Expected Answer:

- For 2D inputs (e.g., images), the idea generalizes to **four RNNs** moving in:
 - Up, Down, Left, Right directions.
- At position (i, j) , the output $O_{i,j}$ depends on signals from all four directions, enabling **long-range lateral interactions** between features.
- Unlike CNNs (which use fixed receptive fields), RNNs can dynamically propagate information across the entire grid.

6. Construct an Encoder-Decoder (Sequence-to-Sequence) RNN model and explain how it maps a variable-length input sequence to a variable-length output sequence. Apply the architecture to real-world tasks like machine translation or speech recognition, and explain the limitation of fixed-size context vectors and the motivation for attention mechanisms. 10 Marks

1. Purpose of Encoder-Decoder Architecture (2 marks)

Answer (1):

- Maps input sequence $X = (x(1), \dots, x(nx))$ to output sequence $Y = (y(1), \dots, y(ny))$ where lengths nx and ny may differ.
- Useful in applications such as machine translation, speech recognition, and question answering.
- Provides a **flexible framework** compared to earlier RNN architectures where $nx = ny = \tau$.

2. Encoder RNN and Context Vector C (2 marks)

Answer (2):

- The encoder RNN reads the input sequence and produces a **context vector C** , which summarizes the sequence:

$$C = f(h_{nx})$$

where h_{nx} is the **final hidden state** of the encoder.

- *Figure 10.12* illustrates this setup.
- C can be a **fixed-length vector** or **variable-length sequence**.

3. Decoder RNN and Output Generation (2 marks)

Answer (3):

- The **decoder RNN** uses the context C to generate outputs:

$$Y = (y(1), \dots, y(ny)) \sim P(y(1), \dots, y(ny) \mid x(1), \dots, x(nx))$$

- Input from C can be:
 - As the **initial hidden state**
 - Connected to **hidden units at each time step**
 - Or a combination of both
- Produces output **sequentially**, accommodating variable-length sequences.

4. Applications of Encoder-Decoder Architecture (2 marks)

Answer (4):

- **Machine Translation:** input sequence = source sentence, output sequence = target sentence
- **Speech Recognition:** input sequence = audio frames, output sequence = phonemes or text
- **Question Answering:** input sequence = question/context, output sequence = answer
- Trained jointly to **maximize log-likelihood** over all sequence pairs in the training set.

5. Limitation of Fixed-Size Context and Attention Mechanism (2 marks)

Answer (5):

- Limitation: **fixed-size vector C** may be **too small** to summarize long sequences $\rightarrow$ information loss
- Solution: **variable-length context + attention mechanism** (Bahdanau et al., 2015)
- Attention allows the decoder to **focus on relevant parts of the input sequence** for each output element.

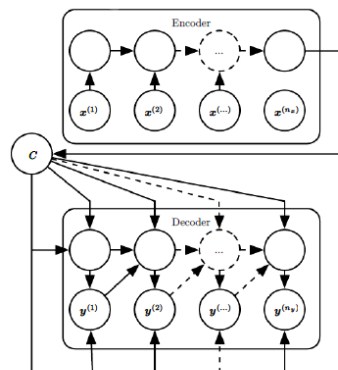


Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture for learning to generate an output sequence $y^{(1)}, \dots, y^{(n_y)}$ given an input sequence $(x^{(1)}, x^{(2)}, \dots, x^{(n_x)})$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

7. Explain how a Recursive Neural Network (RecNN) forms outputs from input sequence, make use of figures to support your answer. Also identify why RecNNs can handle long-term dependencies better than RNNs and select suitable tree structures for different applications. (10 marks)

1. Modeling the Structure of a Recursive Neural Network (2 Marks)

A Recursive Neural Network (RecNN) generalizes an RNN by using a **tree structure** instead of a sequential chain.

Key ideas:

- Inputs begin as **leaf nodes** in a tree.
- Pairs of nodes are **merged** to form parent nodes.
- This merging continues until a **single root node** is formed.
- The tree shape allows RecNNs to work with **variable-length inputs** naturally.

This hierarchical combination helps RecNNs capture **structured relationships** within the data.

2. Using Figure 10.14 to Explain RecNN Computation (2 Marks)

From **Figure 10.14**, the RecNN works as follows:

- Inputs such as $x(1), x(2), x(3), x(4)$ are placed at the **leaves**.
- Each internal node combines its children using the **same shared parameters**, forming a consistent computation pattern.
- These combinations rise upward through the tree, producing a **single final output o** at the root.
- A target output y is associated with this final representation during supervised learning.

The figure highlights:

- A **tree-shaped computational graph**
- **Shared parameters** across all nodes
- A **fixed-size output**, even for long or short inputs

3. Why RecNNs Handle Long-Term Dependencies Better (2 Marks)

RNNs process sequences step-by-step through many layers, making it difficult to remember long-range information.

RecNNs help because:

- They significantly **reduce network depth** from a long chain to a **logarithmic number of steps**.
- Information does not need to flow through every position of the sequence.
- Relationships between distant elements are captured earlier in the tree.

This makes RecNNs more effective for **hierarchical and long-distance dependencies**.

4. Choosing the Right Tree Structure (2 Marks)

The structure of the tree determines how information is combined.

Types of tree structures:

- **Balanced binary trees:** Used when no natural structure exists.
- **Parse trees (NLP):** Provided by a language parser; match sentence grammar.
- **Learned structures:** The network discovers the optimal structure automatically.

Choosing the right tree helps the model capture correct **semantic** or **structural** relationships.

5. Applications and Variants of RecNNs (2 Marks)

RecNNs are widely used in tasks requiring hierarchical or structured input.

Applications:

- **Natural Language Processing:** Sentiment analysis, phrase structure understanding, semantic tasks.
- **Computer Vision:** Hierarchical merging of image regions.
- **Structured data:** Trees, graph-like structures.

Variants:

- Some variants use **tensor operations** or **bilinear forms** to capture richer relationships between node representations.

These features make RecNNs suitable for problems requiring **compositional reasoning**.

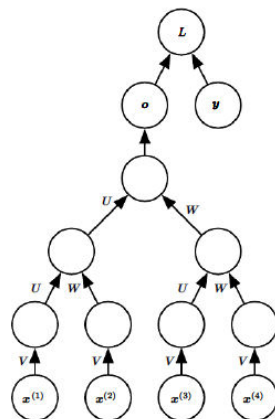


Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $x^{(1)}, x^{(2)}, \dots, x^{(l)}$ can be mapped to a fixed-size representation (the output o), with a fixed set of parameters (the weight matrices U, V, W). The figure illustrates a supervised learning case in which some target y is provided which is associated with the whole sequence.

8. Utilize your understanding of LSTM-based gated RNNs to explain how gating mechanisms (input, forget, and output gates) control information flow through time. Identify the role of each gate using Figure 10.16 and construct the state-update process showing how LSTM prevents vanishing/exploding gradients. (10 Marks)

1. Need for Gated RNNs and Motivation (2 Marks)

Traditional RNNs suffer from **vanishing and exploding gradients**, which makes long-term learning difficult.

- **Leaky units** introduced a slow-decaying self-loop to keep information longer.
- But their forgetting behavior was **manual** or fixed.
- **Gated RNNs**, especially LSTMs, improve this by learning **when to store, when to forget, and when to output information**, giving stable gradient flow through time.

Thus, gated RNNs dynamically control information flow to enable **long-term memory**.

2. LSTM Cell Structure Using Figure 10.16 (2 Marks)

Figure 10.16 shows that an LSTM cell includes:

- An **input unit**
- An **input gate** (σ)
- A **forget gate** (σ)
- An **output gate** (σ)
- A **cell state** with a **self-loop**, controlled by the forget gate
- A **one-step delay** (square block)

These units coordinate to:

- **Decide what information enters**
- **What is kept or erased**
- **What is sent out as output**

The self-loop ensures stable gradient flow.

3. Role of the Input Gate (2 Marks)

The **input gate** controls whether new information should **enter** the cell state.

Concept:

- The input signal is produced by a standard neuron.
- The input gate (sigmoid) scales this information between **0 and 1**.
- When the gate value is high $\rightarrow$ new information is added.
- When the gate value is low $\rightarrow$ new information is blocked.

Thus, the input gate **filters incoming data**, preventing irrelevant information from entering memory.

4. Role of the Forget Gate (2 Marks)

The **forget gate** determines how much of the old state should be kept or erased.

Key idea:

- The cell state has a **linear self-loop**, but the forget gate controls its strength.
- Forget gate output close to 1 → **keep previous memory**.
- Forget gate output close to 0 → **erase previous memory**.

This solves the problem of **manually clearing** the memory in leaky units.

The network *learns* when old information is no longer useful.

5. Role of the Output Gate (2 Marks)

The **output gate** decides how much of the internal cell state is revealed as the LSTM output.

Process:

- The internal state is first squashed (typically by tanh).
- The output gate (sigmoid) scales this activation.
- Only the relevant part of the memory becomes the **hidden output**.

This allows the LSTM to:

- Store information internally
- But reveal only what is needed for the next layer or the next time step

Thus, the output gate gives precise **control over the LSTM's external behavior**.

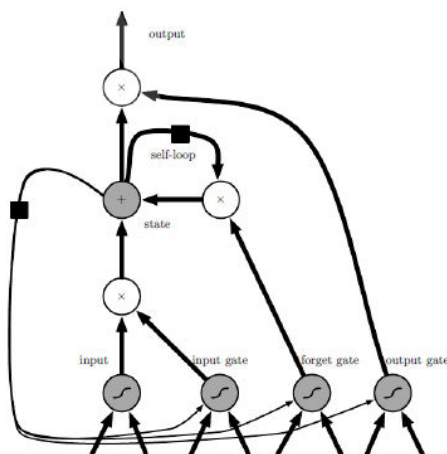


Figure 10.16: Block diagram of the LSTM recurrent network "cell." Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

9. Organize the main steps involved in breaking raw text into sentence-level and word-level tokens, and illustrate each step with simple examples. (10marks)

1. Loading the Text Corpus (2 Marks)

The dataset used is the **Project Gutenberg corpus** available in NLTK.

It contains **18 books**, such as *Emma*, *Alice in Wonderland*, and Shakespeare's plays.

How it is loaded:

```
python

from nltk.corpus import gutenberg
```

Why this matters:

Before tokenizing, we need to access the **raw text** that contains:

- book titles
- chapter names
- paragraphs
- newline characters (`\n`)

Simple example:

Loading *Emma* gives a long block like:

```
"[Emma by Jane Austen 1816]\n\nVOLUME I\nCHAPTER I\n\nEmma Woodhouse..."
```

2. Breaking the Text into Sentences (Sentence Tokenization) (2 Marks)

To convert the large text into individual sentences, we use NLTK's `sent_tokenize()`.

Example from content:

```
python

gberg_sent_tokens = sent_tokenize(gutenberg.raw())
```

This splits the entire book into many sentences.

Result example:

First element mixes title + text (because of newlines):

```
"[Emma by Jane Austen 1816] ... Emma Woodhouse..."
```

Second element is the **first proper sentence**:

```
"She was the youngest of the two daughters..."
```

Simple example:

Input: "Emma is kind. She loves reading."

Output: ["Emma is kind.", "She loves reading."]

3. Checking the Sentence Tokens (2 Marks)

After tokenizing, we inspect them to ensure the separation is correct.

Why needed:

- Some sentences may include titles or chapter headers.
- Raw text often blends together due to newlines.

Content example:

`gberg_sent_tokens[1]` gives a clean sentence:

"She was the youngest of the two daughters..."

This confirms tokenization worked.

4. Breaking Sentences into Words (Word Tokenization) (2 Marks)

Next, we convert each sentence into individual words, removing whitespace and splitting punctuation.

Using:

```
python

word_tokenize(gberg_sent_tokens[1])
```

What happens:

- Words are separated
- Newlines are removed
- Punctuation becomes its own token

Content example:

To get the 15th word in the second sentence:

```
python

word_tokenize(gberg_sent_tokens[1])[14]
```

Output: "father"

Simple example:

Input: "She was kind and honest."

Output: ["She", "was", "kind", "and", "honest", "."]

5. Using `gutenberg.sents()` for Direct Sentence + Word Tokens (2 Marks)

Instead of applying both sentence and word tokenizers, Gutenberg offers a direct shortcut.

Command:

```
python

gberg_sents = gutenberg.sents()
```

What this returns:

- A list of sentences
- Each sentence is already split into words

Example output from content:

```
css

[[[' ', 'Emma', 'by', 'Jane', 'Austen', '1816', ''],
  ['VOLUME', 'I'],
  ['CHAPTER', 'I']]]
```

This separates titles, chapters, and real sentences correctly.

Access example:

To get the 15th word of the second real sentence:

```
python

gberg_sents[4][14]
```

Output: "father"

10. Utilize NLTK to preprocess a given sentence by removing stop words and punctuation, and then apply stemming to the remaining words. Illustrate your steps with a simple example sentence. (10m)

Step 1: Construct the stop words + punctuation list (2 Marks)

- Natural language data contains common words like *the*, *a*, *an* which often do not carry significant meaning in analysis. These are called **stop words**.
- Similarly, punctuation symbols like `.`, `,`, `!` can usually be removed as they don't add semantic meaning in many NLP tasks.
- In Python, we can combine these lists using NLTK and the string library:

```
python

from nltk.corpus import stopwords
import string

stpwrds = stopwords.words('english') + list(string.punctuation)
```

- **Example:** The `stpwrds` list will include: `'the', 'and', 'a', '.', ',', '!',` etc.

Step 2: Convert words to lowercase (2 Marks)

- **Purpose:** To ensure that capitalization does not cause the same word to be treated differently.
- **Example:**
 - Input: `"She Went to the Market."`
 - After lowercase conversion: `["she", "went", "to", "the", "market", "."]`

Step 3: Remove stop words and punctuation (2 Marks)

- **Method:** Use a list comprehension to filter out stop words and punctuation from the sentence.

```
python

filtered_words = [w.lower() for w in sentence if w.lower() not in stpwrds]
```

- **Example:**
 - Input: `["She", "went", "to", "the", "market", "."]`
 - Stop words/punctuation removed: `["went", "market"]`
- **Explanation:** Now, the sentence contains only meaningful words for analysis.

Step 4: Apply stemming using PorterStemmer (2 Marks)

- **Purpose:** To reduce words to their **root form** (stem), which helps pool similar words together and reduce dimensionality.
- **Method:**

```
python

from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()

stemmed_words = [stemmer.stem(w) for w in filtered_words]
```

- **Example:**
 - Words: ["houses", "running", "daughters"]
 - Stemmed: ["hous", "run", "daughter"]
- **Explanation:**
 - *houses* → *hous* (plural treated same as singular)
 - *running* → *run* (different tenses normalized)
 - *daughters* → *daughter*

Step 5: Advantages of these preprocessing steps (2 Marks)

1. **Removes noise:** Stop words and punctuation that do not add meaning are removed.
2. **Reduces vocabulary size:** Stemming pools words together (e.g., *run*, *running*, *ran* → *run*), which helps with smaller datasets.
3. **Improves modeling:** Helps word embeddings (Word2Vec, GloVe) place similar words close in vector space, improving the performance of downstream NLP models.
4. **Task-specific flexibility:** In sentiment analysis, some words like *not* should **not** be removed because they affect meaning.

Illustrative Example Using a Sentence

Original sentence:

"She was the youngest of the two daughters of a most affectionate, indulgent father."

1. **Lowercase:**
["she", "was", "the", "youngest", "of", "the", "two", "daughters", "of", "a", "most", "affectionate", "indulgent", "father"]
 2. **Remove stop words & punctuation:**
["youngest", "two", "daughters", "affectionate", "indulgent", "father"]
 3. **Apply stemming:**
["youngest", "two", "daughter", "affection", "indulg", "father"]
- **Outcome:** Meaningful words are retained in their root forms, ready for use in NLP modeling.

11. Construct a workflow to detect and process n-grams in a text corpus, explaining how bigrams and trigrams are identified and handled with examples. (10 marks)

1. Detecting bigrams using gensim (2 Marks)

- Use `Phrases()` to scan the corpus and identify word pairs that frequently occur together relative to individual word occurrences.
- Use `Phraser()` to create a compact object that efficiently converts these bigram collocations into single tokens.
- **Example from content:**

```
python

phrases = Phrases(gberg_sents)
bigram = Phraser(phrases)
```

- Output is a dictionary of bigram counts and scores.
- **Figure Reference:** See Figure 11.2 – A dictionary of bigrams detected within the corpus.

2. Converting bigrams in sentences (2 Marks)

- Tokenized sentences can be passed through the bigram object to merge frequent word pairs into one token.
- **Example:**

```
python

tokenized_sentence = "Jon lives in New York City".split()
bigram[tokenized_sentence]
# Output: ['Jon', 'lives', 'in', 'New_York', 'City']
```

- "New York" is merged into "New_York", making it a single token.
- **Figure Reference:** Follows the example of tokenized sentence as described in Figure 11.1, showing tokenization at the word level.

3. Converting bigram collocations into single tokens

Once bigrams are detected using `Phrases()` and processed with `Phraser()`, the next step is to convert the detected bigram pairs from two separate tokens into a single token. This ensures that the two words which commonly appear together are treated as one unit in the corpus.

- **Example:**
 - Original tokenized sentence: `["Jon", "lives", "in", "New", "York", "City"]`
 - After passing through the bigram object: `["Jon", "lives", "in", "New_York", "City"]`
 - Here, `New York` has been merged into a single token `New_York`.
- **Purpose:**
 - This is important because it preserves the semantic meaning of the combined words. Treating `New` and `York` separately might dilute context, whereas combining them ensures the model sees it as a single concept.

4. Detecting higher-order n-grams (trigrams, 4-grams, etc.)

After bigrams are handled, you can extend the process to identify trigrams and higher-order n-grams.

- **Process:**
 1. Pass the corpus, which already has bigrams merged, through `Phrases()` and `Phraser()` again.
 2. This identifies frequently occurring sequences of three words (trigrams), e.g., `New York City` → `New_York_City`.
 3. The same approach can theoretically be repeated for 4-grams and 5-grams.
- **Practical note:**
 - There are diminishing returns beyond trigrams. Bigrams and trigrams usually capture the majority of useful multi-word expressions in a corpus.
 - Example: In the Project Gutenberg corpus, `New York City` is unlikely to appear frequently enough to be detected as a trigram because classic literature rarely uses that exact phrase.

5. Handling capitalized words and punctuation

Detected n-grams might still have inconsistencies such as capitalization or punctuation, which can affect downstream modeling.

- **Example issues:**
 - `New York` vs `new york` (capitalization inconsistency)
 - `Mr.` vs `Mr` (punctuation affecting token recognition)
- **Solution:**
 - Preprocess the corpus before or after n-gram detection to:
 - Convert all words to lowercase (e.g., `New_York` → `new_york`)
 - Remove punctuation marks if they do not contribute meaning
 - This ensures consistency in token representation and avoids duplicate tokens representing the same concept.
- **Impact:**
 - Helps the model treat semantically identical n-grams as a single token.
 - Prevents sparsity issues in vector representations, especially important for small corpora.

```
{('two', 'daughters'): (19, 11.966813731181546),
 ('her', 'sister'): (195, 17.7960829227865),
 ('', 's'): (9701, 31.066242737744524),
 ('very', 'early'): (24, 11.01214147275924),
 ('Her', 'mother'): (14, 13.529425062715127),
 ('long', 'ago'): (38, 63.22343628984788),
 ('more', 'than'): (541, 29.023584433996874),
 ('had', 'been'): (1256, 22.306024648925288),
 ('an', 'excellent'): (54, 39.063874851750626),
 ('Miss', 'Taylor'): (48, 453.75918026073305),
 ('very', 'fond'): (28, 24.134280468850747),
 ('passed', 'away'): (25, 12.35053642325912),
 ('too', 'much'): (173, 31.376002029426687),
 ('did', 'not'): (935, 11.728416217142811),
 ('any', 'means'): (27, 14.096964108090186),
 ('wedding', '-'): (15, 17.4695197740113),
 ('Her', 'father'): (18, 13.129571562488772),
 ('after', 'dinner'): (21, 21.5285481168817),
```

Figure 11.2 A dictionary of bigrams detected within our corpus

12. Construct a fully preprocessed corpus from the Project Gutenberg dataset by explaining the key preprocessing steps and bigram detection process. Illustrate your

answer with relevant examples and figures from the content. (10m)

1. Removing capitalization and punctuation (2 Marks)

- Convert all words in the corpus to lowercase to avoid treating the same word differently due to capitalization.
- Remove punctuation marks that do not contribute to semantic meaning.
- **Example (from content):**

```
python

lower_sents = []
for s in gberg_sents:
    lower_sents.append([w.lower() for w in s if w.lower() not in list(string.punctuation)])
```

- **Purpose:** Standardizes tokens, reduces redundancy, and prepares the corpus for consistent n-gram detection.
- **Reference:** Example 11.5

2. Detecting collocated bigrams in the cleaned corpus (2 Marks)

- Use `Phrases()` to detect frequent word pairs (bigram collocations) in the cleaned corpus.
- Use `Phraser()` to create an object that can efficiently convert detected bigrams into single tokens.
- **Example (from content):**

```
python

lower_bigram = Phraser(Phrases(lower_sents))
```

- **Purpose:** Preserves multi-word expressions such as `miss taylor` or `mr woodhouse`.

3. Adjusting thresholds for conservative bigram detection (2 Marks)

- Default `min_count` and `threshold` values may detect too many weak bigrams.
- Increase the minimum count and threshold to focus on meaningful collocations.
- **Example (from content):**

```
python

lower_bigram = Phraser(Phrases(lower_sents, min_count=32, threshold=64))
```

- **Purpose:** Filters out statistically insignificant bigrams like `two daughters` or `her sister`.

4. Creating a clean corpus including bigrams (2 Marks)

- Iterate over each sentence in the lowercase, punctuation-free corpus and apply the `lower_bigram` object to replace bigrams.
- **Example (from content):**

```
python

clean_sents = []
for s in lower_sents:
    clean_sents.append(lower_bigram[s])
```

- **Purpose:** Produces a fully preprocessed corpus suitable for NLP modeling that includes meaningful multi-word tokens.

5. Illustrative example from the clean corpus (2 Marks)

- **Example:** `clean_sents[6]` contains bigrams such as `miss taylor` and `mr woodhouse`.
- **Purpose:** Demonstrates that preprocessing and bigram detection preserved key multi-word expressions while removing capitalization and punctuation.
- **Reference:** Figure 11.5

```
[ 'sixteen',
  'years',
  'had',
  'miss_taylor',
  'been',
  'in',
  'mr_woodhouse',
  's',
  'family',
  'less',
  'as',
  'a',
  'governess',
  'than',
  'a',
  'friend',
  'very',
  'fond',
  'of',
  'both',
  'daughters',
  'but',
  'particularly',
  'of',
  'emma']
```

Figure 11.5 Clean, preprocessed sentence from the Project Gutenberg corpus