

VU LEARNING HUB

vulearninghub.xyz

HIGHLIGHTED HANDOUT

CS631

Deep Learning

CS631 by VU Learning HUB

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

VU LEARNING HUB

vulearninghub.xyz

HANDOUT

CS631

Deep Learning

CS631 by VU Learning HUB

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

VIRTUAL UNIVERSITY OF PAKISTAN

CS631

Deep Learning

STUDY HANDOUT

Deep Learning powers modern AI: image recognition, language models, speech and recommendation systems. This handout builds the subject from the perceptron through backpropagation, optimization, regularization, CNNs, RNNs/LSTMs and generative models, with key terms, worked examples and exam pointers. 

Compiled as a structured study resource for VU students.

Covers the key topics, definitions, concepts and exam pointers of the course.

Table of Contents

1. Introduction to Deep Learning
2. The Perceptron and Artificial Neurons
3. Multilayer Networks and Forward Propagation
4. Backpropagation
5. Optimization: SGD, Momentum, Adam
6. Regularization
7. Convolutional Neural Networks (CNNs)
8. Recurrent Networks: RNN, LSTM, GRU
9. Autoencoders and Generative Models
10. Practical Deep Learning

Unit 1: Introduction to Deep Learning

Learning Objectives

- Define deep learning and contrast it with classical machine learning
- Explain why deep learning succeeded: data, compute (GPUs), algorithms
- Name major application areas of deep learning
- Describe the historical milestones: perceptron, backprop, AlexNet, transformers

Deep learning is machine learning with deep neural networks - networks of many stacked layers that learn hierarchical representations. Classical ML relies on hand-engineered features; deep learning learns the features themselves from raw data (pixels, waveforms, text).

Three forces drove the deep learning revolution: data (massive labeled datasets like ImageNet), compute (GPUs/TPUs making million-parameter training feasible), and algorithms (better initialization, ReLU, dropout, batch normalization, Adam). Around 2012, AlexNet's ImageNet victory made deep learning the default approach in vision, then speech and language.

Applications: computer vision (face recognition, medical imaging, self-driving cars), natural language processing (translation, chatbots, large language models), speech (assistants, transcription), recommendation (feeds, ads), science (protein folding with AlphaFold, weather, materials).

History in brief: 1943 McCulloch-Pitts neuron; 1958 Rosenblatt's perceptron; 1969 Minsky & Papert's critique (XOR problem) caused an AI winter; 1986 backpropagation (Rumelhart, Hinton, Williams) revived training of multilayer nets; 1989 LeNet for digit recognition; 2012 AlexNet; 2014 GANs; 2017 Transformers ('Attention is All You Need') leading to GPT/BERT era.

Deep learning is not magic: it needs large data, heavy compute, and careful tuning; it can be a black box, biased by its training data, and vulnerable to adversarial examples. Knowing the limits is part of knowing the subject.

Key Terms

Deep learning	ML with multi-layer neural networks that learn hierarchical features from raw data.
Representation learning	Learning useful features automatically instead of hand-engineering them.
GPU/TPU	Parallel hardware that made large-scale neural network training practical.
Perceptron	1958 single-neuron classifier; foundation of neural networks.
Backpropagation	1986 algorithm for training multilayer networks via the chain rule.

Example - hierarchy: In image classification, layer 1 learns edges, layer 2 learns textures and corners, layer 3 learns parts (eyes, wheels), layer 4 learns objects (faces, cars) - features built from features.

Example - data hunger: A CNN trained on 100 cat photos overfits badly; the same architecture on 1M ImageNet images generalizes - deep models are data-hungry.

Summary

- ✓ Deep learning = deep neural nets learning features from raw data.
- ✓ Success from data + GPUs + algorithmic advances; AlexNet (2012) was the turning point.
- ✓ Dominates vision, language, speech, recommendations.
- ✓ Limits: data/compute hunger, black-box nature, bias, adversarial vulnerability.

Exam Pointers

- ★ Contrast deep learning with classical machine learning.
- ★ Name the three forces behind deep learning's success and explain each.
- ★ Outline the key historical milestones from perceptron to transformers.

Unit 2: The Perceptron and Artificial Neurons

Learning Objectives

- Describe the McCulloch-Pitts neuron and Rosenblatt's perceptron
- Write the perceptron learning rule and state the convergence theorem
- Explain the XOR limitation and why it matters
- Compare activation functions: step, sigmoid, tanh, ReLU

The artificial neuron computes a weighted sum of inputs plus a bias, then applies an activation function: $y = f(w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b)$. The weights w decide each input's influence; the bias b shifts the decision boundary; f adds non-linearity.

Rosenblatt's perceptron (1958) uses a step activation: output 1 if the weighted sum exceeds 0, else 0. It learns by the perceptron rule: for each misclassified example, $w = w + \eta * (\text{target} - \text{output}) * x$ - nudging weights toward correct classification. The perceptron convergence theorem guarantees it finds a separating hyperplane if the data is linearly separable.

Minsky and Papert (1969) showed a single perceptron cannot learn XOR - no straight line separates the XOR classes. This 'XOR problem' needs a hidden layer: multilayer networks can represent any Boolean function, which is why depth matters.

Activation functions: step (perceptron; not differentiable); sigmoid $\sigma(z) = 1/(1+e^{-z})$, squashes to (0,1) but saturates and causes vanishing gradients; tanh, zero-centered (-1,1) but still saturating; ReLU $\max(0,z)$ - simple, fast, the modern default for hidden layers, though neurons can 'die' (stuck at 0); variants like Leaky ReLU fix this.

Output layers use softmax (multi-class probabilities) or sigmoid (binary).

Key Terms

Neuron	Computes $f(w \cdot x + b)$: weighted sum plus bias through an activation.
Perceptron rule	$w \leftarrow w + \eta \cdot (\text{target} - \text{output}) \cdot x$ on misclassified examples.
Linear separability	Classes separable by a hyperplane; required for perceptron convergence.
XOR problem	Single perceptron cannot learn XOR; motivates hidden layers.
ReLU	$\max(0, z)$; default hidden-layer activation - fast, non-saturating.
Softmax	Turns logits into a probability distribution for multi-class output.

Example - AND with perceptron: $w = (1, 1)$, $b = -1.5$: $(0, 0) \rightarrow -1.5 < 0$; $(1, 0) \rightarrow -0.5 < 0$; $(1, 1) \rightarrow 0.5 > 0$. Correct AND behavior.

Example - XOR needs depth: A 2-2-1 network (2 hidden ReLU neurons) learns XOR easily - the hidden layer creates a new representation where the classes are linearly separable.

Summary

- ✓ Neuron: $y = f(w \cdot x + b)$. Perceptron learns linearly separable functions; guaranteed to converge.
- ✓ XOR proves single-layer limits; hidden layers fix it.
- ✓ ReLU for hidden layers; softmax/sigmoid for outputs.

Exam Pointers

- ★ Write the perceptron learning rule and state the convergence theorem's condition.
- ★ Why can't a single perceptron learn XOR? What fixes it?
- ★ Compare sigmoid, tanh and ReLU as hidden-layer activations.

Unit 3: Multilayer Networks and Forward Propagation

Learning Objectives

- Describe the architecture of a multilayer perceptron (MLP)
- Perform forward propagation through a small network by hand
- Explain the role of hidden layers as learned feature extractors
- State the universal approximation theorem

A multilayer perceptron (MLP) stacks layers: an input layer (the features), one or more hidden layers, and an output layer. It is fully connected (dense): every neuron connects to all neurons in the next layer. 'Deep' simply means many hidden layers.

Forward propagation computes the network's output: for layer l , $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$, $a^{[l]} = f(z^{[l]})$, starting with $a^{[0]} = x$. Each layer transforms the representation; the final layer's activation matches the task (softmax for classification, linear for regression).

Hidden layers are learned feature extractors: early layers detect simple patterns, deeper layers combine them into complex concepts. This hierarchy is why depth beats width for most real tasks.

The universal approximation theorem states that a network with one sufficiently wide hidden layer can approximate any continuous function. In practice, deep (narrower) networks learn such functions far more efficiently - depth gives exponential representational advantages for compositional problems.

Design choices: number of layers/neurons (capacity), activations, and the loss function - mean squared error for regression, cross-entropy for classification. Capacity must match the data: too little underfits, too much overfits (hence regularization, covered later).

Key Terms

MLP	Fully-connected feedforward network: input, hidden, output layers.
Forward propagation	Computing outputs layer by layer: $z = Wx + b$, $a = f(z)$.
Hidden layer	Intermediate layer; learns increasingly abstract features.
Universal approximation theorem	One wide hidden layer can approximate any continuous function.
Loss function	What training minimizes: MSE (regression), cross-entropy (classification).

Example - forward pass: $x = [1, 0]$, $W1 = [[0.5, -0.5], [0.3, 0.8]]$, $b1 = [0, 0]$, ReLU: $z = [0.5, 0.3]$, $a = [0.5, 0.3]$. With $W2 = [1, -1]$, $b2 = 0$: $z = 0.2 \rightarrow$ output 0.2.

Example - loss: Binary classification, true $y=1$, predicted $p=0.8$: cross-entropy loss = $-\log(0.8) = 0.223$. Confident wrong prediction $p=0.1 \rightarrow$ loss 2.3 (much larger).

Summary

- ✓ MLP: stacked dense layers; forward prop computes $z = Wx + b$, $a = f(z)$ per layer.
- ✓ Hidden layers learn hierarchical features; depth is representationally efficient.
- ✓ Choose loss by task: MSE for regression, cross-entropy for classification.

Exam Pointers

- ★ Perform forward propagation on a given 2-2-1 network with ReLU.
- ★ Explain the universal approximation theorem and why depth is still preferred.
- ★ How do you choose the output activation and loss for (a) regression, (b) 3-class classification?

Unit 4: Backpropagation

Learning Objectives

- Explain the chain rule's role in computing gradients
- Derive backpropagation for a small network
- Describe the forward-backward training loop
- Explain vanishing and exploding gradients

Training minimizes the loss by gradient descent: $w = w - \eta * dL/dw$. Backpropagation computes all those gradients efficiently using the chain rule from calculus: $dL/dw = (dL/da)(da/dz)(dz/dw)$, propagating the error signal backward from the output layer to the input.

The algorithm: (1) forward pass - compute all activations; (2) compute loss and output error $\delta^L = dL/dz^L$; (3) backward pass - $\delta^I = (W^{I+1})^T \delta^{I+1} * f'(z^I)$; gradients $dW^I = \delta^I (a^{I-1})^T$, $db^I = \delta^I$; (4) update weights with the optimizer. Frameworks do this automatically via autodiff, but understanding it is essential for debugging. 

Vanishing gradients: with sigmoid/tanh activations, derivatives are < 1 , so gradients shrink exponentially through many layers - early layers barely learn. ReLU (derivative 0 or 1), careful initialization (He/Xavier), batch normalization and residual connections fix it. Exploding gradients (gradients blow up, common in RNNs) are handled by gradient clipping.

Backprop is just the chain rule applied systematically - its brilliance is computational: one forward and one backward pass gives gradients for millions of parameters, instead of finite differences requiring millions of forward passes.

Key Terms

Backpropagation	Chain-rule algorithm computing loss gradients layer by layer, backward.
Chain rule	$dL/dw = dL/da \cdot da/dz \cdot dz/dw$; the mathematical core of backprop.
Autodiff	Automatic differentiation; frameworks compute backprop for you.
Vanishing gradient	Gradients shrink through deep layers; fixed by ReLU, good init, batch norm, residuals.

Exploding gradient

Gradients blow up; fixed by gradient clipping.

Example - chain rule: $L = (y-t)^2$, $y = \text{sigmoid}(z)$, $z = wx$. $dL/dw = 2(y-t) * y(1-y) * x$.
With $y=0.7$, $t=1$, $x=2$: $dL/dw = 2(-0.3)(0.21)(2) = -0.252$.

Example - vanishing: 10 sigmoid layers, each derivative ~ 0.2 : gradient reaching layer 1 $\sim 0.2^{10} \sim 1e-7$ - effectively zero learning. ReLU keeps it alive.

Summary

- ✓ Backprop = chain rule backward through the network; one forward + one backward pass trains everything.
- ✓ Vanishing gradients (sigmoid depth) vs exploding (RNNs); known fixes for each.
- ✓ Autodiff automates it, but you must understand it to debug training.

Exam Pointers

- ★ Derive the backprop update for a single neuron with sigmoid and squared loss.
- ★ Explain vanishing gradients: cause, symptom, and three fixes.
- ★ Why is backpropagation computationally superior to finite-difference gradients?

Unit 5: Optimization: SGD, Momentum, Adam

Learning Objectives

- Describe batch, stochastic and mini-batch gradient descent
- Explain learning rate, momentum and their tuning
- Describe Adam and why it is the default optimizer
- Diagnose training problems from loss curves

Gradient descent variants differ in how much data each update uses: batch GD uses the whole dataset (exact gradient, slow, memory-heavy); SGD uses one example (noisy, fast updates, can escape shallow minima); mini-batch GD (e.g. 32-256 examples) is the practical standard - efficient on GPUs with manageable noise.

The learning rate η is the most important hyperparameter: too large diverges or oscillates; too small crawls. Learning rate schedules (decay, step decay, cosine annealing) start high and reduce. Momentum adds a fraction of the previous update - accelerating through consistent directions and damping oscillation in ravines.

Adam (Adaptive Moment Estimation) combines momentum with per-parameter adaptive learning rates (from RMSprop): it keeps running averages of gradients and squared gradients, correcting bias. Default settings ($\text{lr}=0.001$, $\text{beta1}=0.9$, $\text{beta2}=0.999$) work well out of the box - which is why Adam is the default choice for most deep learning.

Reading loss curves: training loss not falling - learning rate too low, bug, or bad init; training falls but validation rises - overfitting (regularize, more data); both stuck high - underfitting (bigger model, train longer); wildly oscillating - learning rate too high.

Always monitor a validation set.

Key Terms

Mini-batch GD	Updates on small batches (32-256); the standard training loop.
Learning rate	Step size η ; most important hyperparameter. 
Momentum	Adds fraction of past update; speeds consistent directions, damps oscillation.
Adam	Adaptive optimizer combining momentum + per-parameter rates; default choice.
Overfitting / underfitting	Memorizing noise vs failing to capture pattern; diagnosed via validation loss.

Example - schedule: Train 100 epochs: lr 0.01 for epochs 1-30, 0.001 for 31-70, 0.0001 after - fast early progress, fine convergence late.

Example - diagnosis: Train loss 0.05, validation loss 0.9 and rising after epoch 20 -> overfitting: add dropout, augment data, or early-stop at epoch 20.

Summary

- ✓ Mini-batch GD is standard; tune the learning rate first.
- ✓ Momentum accelerates; Adam adapts per parameter and is the default.
- ✓ Diagnose with train vs validation curves: overfit, underfit, or bad lr.

Exam Pointers

- ★ Compare batch, stochastic and mini-batch gradient descent.
- ★ Explain how Adam differs from plain SGD with momentum.
- ★ Given train/validation loss curves, diagnose overfitting vs underfitting and suggest fixes.

Unit 6: Regularization

Learning Objectives

- Explain overfitting and the bias-variance tradeoff
- Describe L1/L2 regularization, dropout and early stopping
- Explain data augmentation and batch normalization
- Choose regularization strategies for a given problem

Overfitting: the model memorizes training noise and fails on new data (low training error, high validation error). The bias-variance tradeoff: simple models underfit (high bias), complex models overfit (high variance); the goal is the sweet spot, controlled by regularization.

L2 regularization (weight decay) adds $\lambda * \sum(w^2)$ to the loss, shrinking weights toward zero - simpler, smoother functions. L1 adds $\lambda * \sum|w|$, driving some weights exactly to zero (feature selection). Dropout randomly zeroes a fraction (e.g. 0.5) of neurons each training step, forcing redundant representations - extremely effective, disabled at test time. Early stopping halts training when validation loss starts rising - free and effective.

Data augmentation artificially expands the dataset (rotate/crop/flip images, paraphrase text) - the single best regularizer when data is scarce. Batch normalization normalizes each layer's inputs per mini-batch, stabilizing training, permitting higher learning rates, and adding mild regularization as a side effect.

Strategy: start with more data/augmentation, then dropout + weight decay, early stopping always, batch norm in deep nets. Regularization strength (λ , dropout rate) is tuned on the validation set - too much causes underfitting.

Key Terms

Overfitting	Memorizing training data; gap between train and validation error.
L2 / weight decay	Penalty $\lambda * \sum(w^2)$; shrinks weights, smoother models.
Dropout	Randomly zeroes neurons during training (e.g. $p=0.5$); off at test time.
Early stopping	Stop when validation loss rises; simplest effective regularizer.
Data augmentation	Synthetic variations of training data; best fix for small datasets.
Batch normalization	Per-batch layer normalization; stabilizes and speeds training.

Example - dropout: A 0.5-dropout MLP on MNIST: each training step uses a different random sub-network; at test time the full network (scaled) generalizes far better than without dropout.

Example - augmentation: 5,000 X-ray images -> 50,000 via rotations, flips, brightness shifts; validation accuracy jumps from 78% to 89%.

Summary

- ✓ Overfitting = high variance; fight it with regularization.
- ✓ Toolkit: L1/L2, dropout, early stopping, augmentation, batch norm.
- ✓ More data/augmentation first; tune regularization strength on validation.

Exam Pointers

- ★ Explain the bias-variance tradeoff and how regularization addresses it.
- ★ Compare L1 vs L2 regularization and dropout - when would you use each?
- ★ Why is data augmentation especially valuable, and what are its limits?

Unit 7: Convolutional Neural Networks (CNNs)

Learning Objectives

- Explain convolution and why it suits image data
- Describe pooling layers and their purpose
- Trace classic architectures: LeNet, AlexNet, VGG, ResNet
- Explain transfer learning with pretrained CNNs

Convolution slides small learnable filters (kernels, e.g. 3x3) across the image, computing dot products - detecting local patterns like edges wherever they appear. Key ideas: local connectivity (each neuron sees a small patch), parameter sharing (same filter everywhere - far fewer weights than dense layers), and translation invariance (a cat's ear is detected anywhere in the image).

Pooling (usually max-pooling 2x2) downsamples feature maps - halving spatial size, reducing computation, and adding robustness to small shifts. A typical CNN stacks convolution -> ReLU -> pooling blocks, ending with dense layers and softmax.

Architectures: LeNet-5 (1998, digit recognition, the template); AlexNet (2012, 8 layers, ReLU+dropout+GPUs, ImageNet breakthrough); VGG (deep stacks of 3x3 convs); ResNet (2015, residual/skip connections letting gradients flow through 100+ layers - solved the degradation problem of very deep nets).

Transfer learning: take a CNN pretrained on ImageNet, replace the final layer, and fine-tune on your (small) dataset - the early layers already know edges and textures. This is the standard practical approach whenever data is limited.

CNNs dominate any grid-like data: images, video, spectrograms, even board games (AlphaGo).

Key Terms

Convolution / kernel	Sliding filter detecting local patterns; shared weights.
Parameter sharing	Same filter reused across the image; huge parameter savings.
Max-pooling	Downsampling by taking local maxima; robustness + efficiency.
ResNet / skip connection	Identity shortcuts enabling 100+ layer training.
Transfer learning	Fine-tuning a pretrained network on a new, smaller task.

Example - parameter saving: 224x224x3 image to 64 features: dense needs ~9.6M weights; a 3x3 conv layer needs $3*3*3*64 = 1,728$. Convolution wins by ~5000x.

Example - transfer: Only 2,000 labeled skin-lesion images: fine-tune ResNet-50 pretrained on ImageNet -> 91% accuracy vs 74% training from scratch.

Summary

- ✓ Conv layers: local filters + weight sharing = efficient, translation-invariant feature learning.
- ✓ Pool to downsample; stack conv blocks, finish with dense + softmax.
- ✓ ResNet's skip connections enabled very deep networks; transfer learning solves small-data problems.

Exam Pointers

- ★ Why is convolution better suited to images than fully-connected layers? Quantify the parameter saving.
- ★ Explain max-pooling and residual connections.
- ★ Describe how you would use transfer learning for a 3,000-image classification task.

Unit 8: Recurrent Networks: RNN, LSTM, GRU

Learning Objectives

- Explain recurrence and why sequences need memory
- Describe the vanishing gradient problem in RNNs
- Explain LSTM gates and GRU as a simplification
- Name applications of sequence models

Sequences (text, speech, time series) need memory: the meaning of a word depends on earlier words. A recurrent neural network (RNN) processes one element at a time, maintaining a hidden state: $h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b)$ - the same weights reused each step, carrying information forward.

Training RNNs by backpropagation through time suffers severe vanishing gradients: the repeated multiplication by W_{hh} shrinks gradients over long sequences, so plain RNNs **effectively remember only ~10 steps back.**

The LSTM (Long Short-Term Memory) fixes this with a cell state - a protected conveyor belt - controlled by three gates: the forget gate (what to discard), the input gate (what new information to store), the output gate (what to reveal). Gradients flow through the cell state nearly unchanged, enabling memory over hundreds of steps. The GRU merges the gates into two (reset, update) - simpler, similar performance.

Applications: language modeling, machine translation (the classic seq2seq encoder-decoder), speech recognition, time-series forecasting, music generation. (Note: Transformers with self-attention have largely replaced RNNs in NLP since 2017, processing whole sequences in parallel - but LSTMs remain important for streaming and small-data sequence tasks.)

Key Terms

RNN	Network with recurrent hidden state; processes sequences step by step.
Backpropagation through time	Unrolling the RNN and applying backprop over steps.
LSTM	Gated cell (forget/input/output gates) preserving long-term gradients.
GRU	Simplified LSTM with reset and update gates.
Seq2seq	Encoder-decoder architecture for sequence-to-sequence tasks like translation.

Example - language: Predicting the next word in 'The keys to the cabinet ____': an LSTM's cell state carries the plural 'keys' across the intervening words to predict 'are' - a plain RNN forgets.

Example - gates: Forget gate ~ 0 on irrelevant old context, ~ 1 on the subject's number; input gate admits the new verb's information.

Summary

- ✓ RNNs add memory via recurrent hidden states; plain RNNs forget long contexts (vanishing gradients).
- ✓ LSTM gates protect a cell state for long-term memory; GRU is the compact variant.
- ✓ Transformers now dominate NLP, but gated RNNs remain useful for sequences.

Exam Pointers

- ★ Write the RNN hidden-state update and explain backpropagation through time.
- ★ How do LSTM gates solve the vanishing gradient problem?
- ★ Compare LSTM and GRU; name two applications of sequence models.

Unit 9: Autoencoders and Generative Models

Learning Objectives

- Explain autoencoders and their uses
- Describe GANs: generator, discriminator, training dynamics
- Describe VAEs at a conceptual level
- Discuss applications and risks of generative models

An autoencoder compresses input x into a low-dimensional latent code z (encoder), then reconstructs $\hat{x}$ from z (decoder), trained to minimize reconstruction error. Uses: dimensionality reduction, denoising (train on noisy $\rightarrow$ clean), anomaly detection (high reconstruction error = anomaly), and pretraining. Variational autoencoders (VAEs) make the latent space probabilistic, so sampling z generates new data.

GANs (Generative Adversarial Networks) pit two networks against each other: the generator creates fake samples from noise, the discriminator tries to distinguish real from fake. They train alternately - the generator improving until its fakes fool the discriminator. Result: stunningly realistic images, but training is unstable (mode collapse: generator produces limited variety).

Modern generative AI is dominated by diffusion models (iteratively denoise pure noise into images - DALL-E, Stable Diffusion) and large language models (autoregressive transformers - GPT), but GANs/VAEs remain the conceptual foundation and exam staple. 

Applications: art and design, photo restoration, synthetic training data, drug discovery.
Risks: deepfakes, misinformation, copyright questions, and bias amplification - generative models deserve ethical scrutiny alongside technical study.

Key Terms

Autoencoder	Encoder-decoder learning compressed latent codes via reconstruction.
Latent space	The compressed representation; sampling it generates new data (VAEs).
GAN	Generator vs discriminator adversarial training for realistic synthesis.
Mode collapse	GAN failure: generator outputs limited variety.
Diffusion model	Generates data by iterative denoising; basis of modern image generators.

Example - denoising: Train an autoencoder on noisy MNIST digits -> clean digits; the bottleneck forces it to learn the true digit structure, ignoring noise.

Example - anomaly: Autoencoder trained on normal network traffic: a cyberattack's traffic reconstructs poorly -> high error flags the intrusion.

Summary

- ✓ Autoencoders compress via reconstruction; VAEs add probabilistic latent spaces.
- ✓ GANs: generator fools discriminator; powerful but unstable.
- ✓ Diffusion models and LLMs are the current state of the art; watch the ethical risks.

Exam Pointers

- ★ Explain the autoencoder architecture and three of its uses.
- ★ Describe GAN training dynamics and mode collapse.
- ★ What ethical concerns do generative models raise?

Unit 10: Practical Deep Learning

Learning Objectives

- Outline a standard deep learning project workflow
- Explain hyperparameter tuning strategies
- Describe debugging techniques for neural networks
- Discuss deployment and ethical considerations

Workflow: (1) define the problem and metric; (2) establish a baseline (simple model or published result); (3) prepare data (clean, split train/val/test, augment); (4) start with a standard architecture and Adam defaults; (5) monitor train/val curves; (6) tune systematically - learning rate first, then architecture/regularization; (7) evaluate once on the test set.

Hyperparameter tuning: grid search (exhaustive, expensive), random search (surprisingly effective), Bayesian optimization (model-guided). Tune in order of impact: learning rate, batch size, architecture depth/width, regularization strength. Never tune on the test set.

Debugging: overfit a single batch first (if the model can't memorize 10 examples, something is broken); check data pipeline (visualize inputs); gradient checking on tiny models; watch for dead ReLUs, exploding loss (NaN - lower lr, clip gradients), and data leakage inflating validation scores.

Deployment: consider model size/latency (quantization, pruning, distillation), monitoring for data drift, and retraining pipelines. Ethics: audit training data for bias, respect privacy, document limitations (model cards), and remember adversarial vulnerability - a model is only as trustworthy as its evaluation.

Key Terms

Baseline	Simple reference result; every project needs one before deep learning.
Random search	Sampling hyperparameters randomly; beats grid search per unit compute.
Data leakage	Test information leaking into training; inflates scores falsely.
Quantization / pruning	Compressing models for deployment (smaller, faster).
Model card	Documentation of a model's intended use, data, and limitations.

Example - single-batch test: New CNN gets 15% on 10 training images after 50 epochs -> bug (probably wrong labels or broken loss), not a tuning problem.

Example - leakage: Including the patient's future lab result as a feature gives 99% 'accuracy' - and a useless model in the clinic.

Summary

- ✓ Baseline first; data and lr tuning beat architecture fiddling.

- ✓ Debug by overfitting a single batch; never tune on test.
- ✓ Deploy with compression and drift monitoring; document ethics and limits.

Exam Pointers

- ★ Outline the standard deep learning project workflow.
- ★ How do you debug a network that won't train?
- ★ Name two deployment and two ethical considerations for a production model.