EE 105: **Neural Networks**
Resources and Worksheets

Prof. Yasser Khan and Prof. Rehan Kapadia

W. V. Unglaub
unglaub@usc.edu

April 29, 2025

Contents

1 Online resources

Below is a curated list of quality online resources that you may find helpful for learning, exploring, and practicing a variety of concepts related to circuits, electronics, and semiconductor devices.

1.1 Tutorials and courses

- Neural Networks and Deep Learning (Michael Nielsen): A free online book that is widely regarded as one of the clearest conceptual introductions to neural networks.
- DeepLearning.AI: Neural Networks and Deep Learning (Coursera): Offered by Andrew Ng, this is the first course in the Deep Learning Specialization. Highly recommended for beginners.
- CS231n: Convolutional Neural Networks for Visual Recognition (Stanford): Though slightly more advanced, its early chapters are accessible and packed with clear illustrations and explanations.
- Fast.ai Practical Deep Learning for Coders: Great for learners who want to build real applications quickly without diving too deep into theory initially.
- 3Blue1Brown: Neural Networks Series (YouTube): Visual, intuitive explanation of how neural networks work. Excellent for visual learners.

1.2 Reference websites

- DeepAI: Offers explanations, tools, and glossaries for AI concepts, including neural networks.
- Distill.pub: Interactive, visually intuitive explanations for deep learning concepts. Particularly good for understanding backpropagation and gradients.
- Keras Documentation: Official docs for Keras, a beginner-friendly deep learning API running on top of TensorFlow.
- TensorFlow Tutorials: Step-by-step tutorials using TensorFlow 2.x, ideal for getting hands-on with Python-based deep learning.

1.3 Online simulation tools

- TensorFlow Playground: Interactive NN simulator in your browser. Helps visualize how changing architecture affects performance.
- ConvNetJS: JavaScript-based neural network framework that runs entirely in the browser.
- NN Sandbox by R2D3: Visual introduction to decision trees and classification, pairs well with neural network learning.
- ml4a Demos: Machine learning tools for artists — offers creative and visual introductions to neural networks.
- Google Colab: Free cloud-based Jupyter notebooks with GPU support. Useful for running your own Python-based NN code.

1.4 Discussion forums and communities

- Stack Overflow: Best for specific technical questions. Use tags like `neural-network`, `tensorflow`, or `keras`.
- r/MachineLearning (Reddit): Active Reddit community focused on all aspects of ML, including beginner questions.
- r/learnmachinelearning (Reddit): Geared toward learners; great for advice, curriculum suggestions, and concept clarifications.
- AI Stack Exchange: A dedicated Q&A platform for AI topics, including neural networks.
- Deep Learning Specialization Forum (Coursera): Course-specific forums for learners following Andrew Ng's curriculum (must first enroll).

2 Overview of neural networks

2.1 Historical Development of Neural Networks

The history of neural networks (NNs) can be traced back to 1943, when McCulloch and Pitts introduced a simple model of an artificial neuron, laying the groundwork for the field. In 1962, Rosenblatt developed the **perceptron**, a neural model capable of adjusting its own weights and with a convergence proof for linearly separable problems.

Despite early optimism, Minsky and Papert's 1969 book exposed critical limitations of single-layer perceptrons, notably their inability to solve problems like XOR. This led to a period of reduced interest. In 1982, Hopfield proposed **Hopfield networks**, introducing the concept of energy functions and stable convergence.

The breakthrough for multilayer networks came in 1986 with the rediscovery of the **backpropagation algorithm**, allowing effective training by minimizing errors across multiple layers. Today, modern NNs incorporate probabilistic models, Bayesian approaches, and biologically inspired **spiking neural networks**.

2.2 Why Deep Neural Networks?

Deep Neural Networks (DNNs) are neural networks with multiple hidden layers between input and output. DNNs are crucial because they enable the automatic extraction of increasingly abstract features from data. Applications include:

- Image and speech recognition
- Natural language processing
- Game playing (e.g., AlphaGo)

A key advantage of DNNs is their scalability: as the volume of training data increases, performance generally improves, which is not always true for traditional machine learning models.

2.3 Anatomy of a Neural Network

Each unit (**neuron**) in a neural network computes an output based on a weighted sum of inputs plus a bias, followed by a nonlinear **activation function**:

$$y = f\left(\sum_{i}^{N} w_i x_i + b\right) = f(w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_N x_N + b), \tag{1}$$

where:

- x_i are the inputs
- w_i are the weights
- b is the bias
- f is the activation function

Common activation functions include:

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- ReLU: $f(x) = \max(0, x)$
- Tanh: $\tanh(x) = 2\sigma(2x) - 1$

NNs are typically organized into:

- **Input Layer**: Receives raw data.
- **Hidden Layers**: Intermediate computations.
- **Output Layer**: Final decision or regression output.

2.4 Training Neural Networks

Training involves optimizing the network parameters (weights and biases) to minimize the difference between predicted outputs and the correct labels.

2.4.1 Forward Pass

Inputs propagate through the network to produce outputs.

2.4.2 Error Calculation

An error is defined using a **cost function**. For regression or classification, a common choice is the **mean squared error (MSE)**:

$$C(w,b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{2}$$

where n is the number of training samples, a is the network output, and $y(x)$ is the true output.

2.4.3 Backpropagation

Using calculus (chain rule), the error is propagated backward to compute gradients with respect to weights and biases.

2.4.4 Update Rule

Parameters are updated using **gradient descent**:

$$w \leftarrow w - \gamma \nabla_w C b \leftarrow b - \gamma \nabla_b C \tag{3}$$

where γ is the learning rate.

2.5 Perceptrons and Logical Functions

The **perceptron** is a fundamental building block of neural networks. It outputs 1 if the weighted sum of its inputs exceeds a threshold and 0 otherwise:

$$\text{Output} = \begin{cases} 1, & \text{if } x \cdot w + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

Perceptrons can implement simple logical functions such as AND, OR, and NAND. Since NAND is functionally complete, any logical function can be built using combinations of perceptrons.

However, perceptrons cannot model non-linearly separable functions like XOR, motivating the use of multilayer networks.

2.6 Neural Network Architecture and Topology

Designing an NN requires careful specification of:

- **Input neurons**: Correspond to input features (e.g., pixels).
- **Output neurons**: Correspond to output classes (e.g., digits 0–9).
- **Hidden layers and neurons**: No fixed rule; depends on problem complexity.

For example, for handwritten digit recognition (MNIST dataset):

- Inputs: $28 \times 28 = 784$ neurons (greyscale pixel intensities)
- Hidden layer: Example, 15 neurons
- Outputs: 10 neurons (one for each digit)

Topology can be feedforward or recurrent. **Feedforward networks** move data strictly forward, while **recurrent networks (RNNs)** incorporate feedback loops and temporal dependencies.

2.7 Cost Function Minimization and Gradient Descent

Gradient descent is the method used to minimize the cost function by updating parameters along the negative gradient direction.

The update rule in general form:

$$v \to v' = v - \gamma \nabla C \tag{5}$$

where v represents weights and biases collectively.

Challenges arise due to the high dimensionality of parameter space. Exact gradient calculation can be computationally expensive.

2.7.1 Stochastic Gradient Descent (SGD)

Rather than using the entire dataset, **SGD** estimates the gradient using a **mini-batch** of randomly selected examples, greatly speeding up training and adding stochasticity that can help escape local minima.

Formally, for a mini-batch of size m:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{x_j} \tag{6}$$

where x_j are the samples in the mini-batch.

This approximation allows rapid iterations while maintaining a reliable direction of descent.

2.8 Conclusion

Neural networks have developed from simple models of binary threshold units to complex architectures capable of solving some of the most challenging problems in modern computing. Understanding the anatomy, training mechanisms, mathematical principles behind gradient descent, and architectural design choices is crucial for designing powerful neural systems that generalize well to unseen data.

3 Hardware implementation of neural networks

3.1 Introduction and Motivation

Neural networks (NNs) offer a flexible, data-driven paradigm for modeling complex, non-linear relationships that are difficult or impossible to capture with traditional rule-based algorithms. While hard-coded mappings suffice for systems governed by well-characterized, linear relationships—such as converting a thermistor's voltage to temperature—many emerging applications exhibit high variability and dimensionality that resist exhaustive rule specification. In speech recognition, for example, differences in accent, intonation, and background noise require a model capable of learning robust, invariant features directly from raw audio. Likewise, autonomous vehicles must interpret a fusion of camera images, LiDAR point clouds, radar returns, and inertial measurements in real time, adapting to changing lighting, weather, and traffic conditions without manual intervention.

Modern NNs achieve this adaptability by extracting hierarchical representations from data, enabling them to generalize to new scenarios and accommodate sensor drift or noise without explicit reprogramming. Crucially, recent advances in low-power hardware and algorithmic optimizations have made on-device inference feasible in resource-constrained environments. From microcontroller-based TinyML frameworks that run quantized networks within tens of kilobytes of memory, to analog and neuromorphic accelerators operating at sub-milliwatt power budgets, these platforms enable always-on, on-edge intelligence. Applications span human activity monitoring in wearables, vibration-based anomaly detection for structural health monitoring, and real-time environmental sensing—all benefiting from the combination of NN adaptability and hardware efficiency.

3.2 Hardware Architectures for Low-Power NN Inference

Modern embedded platforms offer several approaches for on-device NN inference:

- **Digital Microcontrollers (TinyML):** ARM Cortex-M4/M7 MCUs (e.g., nRF52840) feature FPUs and DSP extensions, achieving on the order of 10^8 MAC/s at 64 MHz. Frameworks like TensorFlow Lite for Microcontrollers support quantized 8-bit models with footprints under 100 kB of RAM.
- **Neuromorphic and Analog Accelerators:**
  - *Spiking hardware:* Event-driven computation triggers only on input changes, reducing dynamic power.
  - *Analog MAC arrays:* Perform multiply-accumulate in the charge or current domain (sub-nJ per operation) but require calibration to mitigate device variability.
- **Key Constraints:**
  - Memory: Typical SRAM budgets are 16–256 kB.
  - Latency: Real-time tasks often demand inference within 5–10 ms.

3.3 On-Device Applications & Sub-mW Computing

Edge intelligence in the sub-milliwatt regime enables:

- **Wearables:** Activity and posture monitoring, fall detection.
- **Structural Health Monitoring:** Vibration anomaly detection in remote sensors.
- **Agricultural IoT:** Pest or disease recognition on foliage.

Power-saving strategies:

1. *Duty-cycling:* Wake the MCU only on relevant interrupts (e.g., accelerometer thresholds).
2. *Near-threshold operation:* Lower supply voltage reduces energy per operation at slower clock speeds.
3. *Model compression:* Pruning, weight sharing, and Huffman coding reduce memory-access energy.

3.4 Feature Extraction for Efficient Inference

Instead of raw high-dimensional data, extract compact descriptors:

3.4.1 Time-Domain Features

- Statistical: mean, variance, root-mean-square (RMS), zero-crossing rate.
- Envelope: peak-to-peak amplitude, crest factor.

3.4.2 Frequency-Domain Features

- Short-time Fourier transform (STFT) bins or Goertzel filters.
- Spectral energy ratios (e.g., energy in 0–5 Hz vs. 5–10 Hz bands to distinguish walking vs. jogging).

3.4.3 Hybrid & Learned Features

- Mel-frequency cepstral coefficients (MFCCs) for audio.
- Autoencoder bottleneck embeddings pre-trained off-line.

Implementation Notes: Use fixed-point arithmetic for filter operations to avoid FPU overhead. Balance buffer length for frequency resolution against latency.

3.5 Model Validation, Confusion Matrices & Metrics

A confusion matrix summarizes classifier outcomes:

	Predicted +	**Predicted −**
Actual +	True Positives (TP)	False Negatives (FN)
Actual −	False Positives (FP)	True Negatives (TN)

Table 1: Confusion Matrix

Derived metrics:

$$\begin{aligned}
\text{Accuracy} &= \frac{TP + TN}{TP + FP + FN + TN}, \\
\text{Sensitivity (Recall)} &= \frac{TP}{TP + FN}, \\
\text{Specificity} &= \frac{TN}{TN + FP}, \\
\text{Precision} &= \frac{TP}{TP + FP}, \\
F_1 &= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.
\end{aligned}$$

3.5.1 Imbalanced Data Strategies

- Resampling: oversample minority or undersample majority classes.
- Class weighting: higher loss penalty for misclassifying minority.
- Threshold tuning: shift decision boundary to favor recall or precision.

3.5.2 Training Monitoring

Plot training vs. validation:

- *Underfitting:* Both accuracy and loss remain poor on training and validation.
- *Overfitting:* Training loss decreases while validation loss increases.

Logging periodic confusion matrices in deployment can detect model drift.

3.6 Problem Worksheets

Click here to access a first set of problems on the basics of artificial neural networks. These problems are mostly analytic and conceptual in nature. Click here to access a second set of problems on the basics of artificial neural networks, which are numeric and computational in nature. Click here to access a third set of problems centered on hardware implementation of neural networks. Click here to access a fourth set of exercises centered on hardware implementation of neural networks, featuring numeric and computational problems.

4 Problems

4.1 Neural networks: Problem Set 1 (Analytic/Conceptual)

[return to TOC]

4.1.1 Problem 1.1: Perceptron Output

A single-neuron perceptron has weights

$$w = [2, -1, 0.5], \quad b = -0.5$$

and uses the step activation

$$\text{step}(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0. \end{cases}$$

Compute its output for input $x = [1, 2, -1]$.

4.1.2 Problem 1.2: Designing an AND Perceptron

Find weights w_1, w_2 and bias b such that the perceptron implements the logical AND of two binary inputs x_1, x_2.

4.1.3 Problem 1.3: The XOR Limitation

Show that no single-layer perceptron can implement XOR:

$$\text{XOR}(x_1, x_2) = \begin{cases} 1, & x_1 \neq x_2, \\ 0, & x_1 = x_2. \end{cases}$$

4.1.4 Problem 1.4: Counting Parameters

A feedforward network has 3 inputs, one hidden layer of 4 neurons, and 2 outputs. How many weights and biases does it have in total?

4.1.5 Problem 1.5: Gradient of a Single Neuron

For a neuron $a = f(z)$ with $z = w^T x + b$ and cost $C = \frac{1}{2}(y - a)^2$, derive $\partial C / \partial w$.

4.1.6 Problem 1.6: Universal Approximation Theorem

State the Universal Approximation Theorem and discuss its implication for why even a single hidden layer with enough neurons can approximate any continuous function on a compact domain.

4.1.7 Problem 1.7: Batch vs. Stochastic Gradient Descent

Explain why Stochastic Gradient Descent (SGD) can help escape shallow local minima better than batch gradient descent.

4.1.8 Problem 1.8: Learning-Rate Effects

Describe qualitatively how too large or too small a learning rate γ affects gradient-descent convergence.

4.1.9 Problem 1.9: One-Step GD Update

For cost $C(w) = w^2 + 4w + 4$, perform one gradient-descent update from $w_0 = 0$ with $\gamma = 0.1$.

4.1.10 Problem 1.10: Parameter Count for Arbitrary Topology

A network has layer sizes $[d, H_1, H_2, \ldots, H_L, K]$. Express the total number of weights and biases.

4.1.11 Problem 1.11: Role of Nonlinearity

Explain why nonlinear activation functions in hidden layers are critical for the representational power of neural networks.

4.1.12 Problem 1.12: Steps of Backpropagation

Outline the main steps of the backpropagation algorithm for a two-layer network.

4.1.13 Problem 1.13: Cost Function Choice for Classification

Why is mean squared error (MSE) suboptimal for classification tasks? Which cost is preferred, and why?

4.1.14 Problem 1.14: Designing Deep Topologies

Briefly discuss how depth (number of hidden layers) affects feature abstraction and network capacity.

4.2 Neural networks: Problem Set 2 (Numeric/Computational) [return to TOC]

4.2.1 Problem 2.1: Activation-Function Plots

Plot the sigmoid, tanh, and ReLU activations on the range $[-5, 5]$.

4.2.2 Problem 2.2: Linear Regression by Gradient Descent

Generate 50 points (x, y) with $y = 2x - 1 + \epsilon$ (Gaussian noise). Use batch gradient descent to fit $y = wx + b$. Plot the MSE cost vs iteration.

4.2.3 Problem 2.3: Perceptron Learning Algorithm

Implement the perceptron learning algorithm to learn the AND function. Plot the evolving decision boundary over the data.

4.2.4 Problem 2.4: Single-Example Backprop

For a 2-layer network with one input x, hidden neuron with weight w_1, output neuron with weight w_2, both using sigmoid activations, compute the gradients $\partial C/\partial w_1$ and $\partial C/\partial w_2$ for a single training example (x, y).

4.2.5 Problem 2.5: Cost Surface Contours

For the linear model $y = wx + b$ and one data point $(x, y) = (1, 2)$, plot the cost $C(w, b) = \frac{1}{2}(y - (wx + b))^2$ as a contour over $w \in [-1, 3]$, $b \in [-3, 3]$.

4.2.6 Problem 2.6: Function Approximation with a 1-Hidden-Layer NN

Use a small NN with one input, a hidden layer of 5 sigmoid neurons, and one linear output to approximate $y = \sin(x)$ on $[0, 2\pi]$. Plot true vs. predicted.

4.3 Neural networks: Problem Set 3 (Analytic/Conceptual) [return to TOC]

4.3.1 Problem 3.1: Why Use Neural Networks?

Explain why neural networks (NNs) often outperform hard-coded rule-based systems in domains such as voice recognition and autonomous driving.

4.3.2 Problem 3.2: Digital vs. Analog Accelerators

Compare digital microcontroller implementations of NNs (TinyML) with analog/neuromorphic accelerators in terms of power, precision, and calibration requirements.

4.3.3 Problem 3.3: TinyML Constraints

Describe the key memory and latency constraints for deploying a neural network on a Cortex-M4 microcontroller.

4.3.4 Problem 3.4: Sub-mW Computing Strategies

List and explain three strategies used in sub-milliwatt edge devices to minimize average power consumption.

4.3.5 Problem 3.5: Feature Extraction Rationale

Why extract features like RMS or spectral energy instead of feeding raw accelerometer data directly into a network?

4.3.6 Problem 3.6: Event-Driven Spiking Hardware

Explain how event-driven (spiking) neuromorphic hardware reduces power consumption compared to clocked digital designs.

4.3.7 Problem 3.7: Deployment Workflow

Outline the steps to deploy a pre-trained CNN to an nRF52840 using TensorFlow Lite for Microcontrollers.

4.3.8 Problem 3.8: Model Compression Effects

Discuss how weight pruning and quantization impact both model size and energy consumption in embedded inference.

4.3.9 Problem 3.9: Cross-Validation Techniques

Compare hold-out validation with k-fold cross-validation, and state when each is preferred.

4.3.10 Problem 3.10: Confusion Matrix Interpretation

Given an imbalanced binary classification problem, explain how precision, recall, and F1 score guide model selection beyond overall accuracy.

4.4 Neural networks: Problem Set 4 (Numeric/Computational) [return to TOC]

4.4.1 Problem 4.1: Energy per MAC Comparison

Given a digital multiply–accumulate (MAC) implemented with an effective capacitance $C_{\text{digital}} = 1$ pF and an analog MAC with $C_{\text{analog}} = 0.1$ pF, both operating at $V = 1$ V, compute the energy per MAC for each implementation.

4.4.2 Problem 4.2: RMS Acceleration from Sensor Data

You record 1 second of tri-axial accelerometer data at 1 kHz:

$$a_x(t) = \sin(2\pi 5t) + \text{noise}, \quad a_y(t) = 0.8\sin(2\pi 3t) + \text{noise}, \quad a_z(t) = 0.5\sin(2\pi 1t) + \text{noise}.$$

Compute the root-mean-square (RMS) acceleration on each axis and plot a histogram of the a_x samples.

4.4.3 Problem 4.3: Spectral Energy in Frequency Bands

Generate a time-series signal $s(t) = \sin(2\pi 2t) + 0.5\sin(2\pi 7t)+$ Gaussian noise over 1 second at 1 kHz. (a) Compute and compare the total spectral energy in the 0–5 Hz band vs. the 5–10 Hz band. (b) Plot the power spectrum $|\text{FFT}(s(t))|^2$ versus frequency.

4.4.4 Problem 4.4: Confusion-Matrix Metrics

For a binary classifier you observe $\text{TP} = 50,\ \text{FP} = 10,\ \text{FN} = 5,\ \text{TN} = 35$. Compute the overall accuracy, precision, recall (TPR), and F_1 score.

4.4.5 Problem 4.5: Training and Validation Curves

Given training accuracy increasing linearly from 0.52 to 0.90 over 20 epochs and validation accuracy defined as $\text{val\_acc} = \text{train\_acc} - 0.05\ln(\text{epoch})$, plot both curves versus epoch and comment on under- or over-fitting.

4.4.6 Problem 4.6: MAC Throughput & Inference Latency

A Cortex-M4 runs at 64 MHz and requires 2 clock cycles per MAC. (a) Calculate the maximum MAC/s throughput. (b) If your model requires 10 000 MACs per inference, compute the latency in milliseconds.

4.4.7 Problem 4.7: Average Power with Duty-Cycling

An edge sensor wakes for 5 ms every 100 ms period. During the active window it consumes 20 mW; in sleep it draws 0.1 mW. Compute its average power consumption.

4.4.8 Problem 4.8: 8-bit Quantization of Weights

Given a weight vector $\mathbf{w} = [0.10, -0.50, 0.75]$, quantize it to signed 8-bit integers (range –128 to +127) and then dequantize back to floating point. Report both the integer values and the recovered floats.

4.4.9 Problem 4.9: Delta MFCC Shape

You extract 13 MFCC coefficients for each of 100 successive audio frames, forming a 13×100 matrix. Compute the first-order delta coefficients (frame-to-frame differences) and state the resulting matrix dimensions.

4.4.10 Problem 4.10: Pruning Savings

A network originally has 100,000 parameters; after pruning it has 25,000 parameters. Calculate the percentage reduction in model size (memory) and, assuming energy scales linearly with parameter count, the estimated energy savings.

5 Solutions to problems

5.1 Neural networks: Solution Set 1 (Analytic/Conceptual)

5.1.1 Solution 1.1: Perceptron Output

Compute the net input

$$z = w \cdot x + b = 2 \cdot 1 + (-1) \cdot 2 + 0.5 \cdot (-1) - 0.5 = 2 - 2 - 0.5 - 0.5 = -1.$$

Since $z \leq 0$, the step output is 0.

5.1.2 Solution 1.2: Designing an AND Perceptron

We need

$$x_1 = x_2 = 1 \implies w_1 + w_2 + b > 0,$$

and for any other combination, $w_1 x_1 + w_2 x_2 + b \leq 0$. A standard choice is

$$w_1 = w_2 = 1, \quad b = -1.5.$$

Check: - $(1,1)$: $1 + 1 - 1.5 = 0.5 > 0 \Rightarrow 1$. - $(1,0)$ or $(0,1)$: sum $= -0.5 \leq 0 \Rightarrow 0$. - $(0,0)$: sum $= -1.5 \leq 0 \Rightarrow 0$.

5.1.3 Solution 1.3: The XOR Limitation

The four input points $(0,0), (0,1), (1,0), (1,1)$ cannot be separated into two classes by one linear boundary: the positive outputs $(0,1)$ and $(1,0)$ lie on opposite corners, so no line can separate them from the negatives $(0,0)$, $(1,1)$. Hence no single linear threshold unit exists.

5.1.4 Solution 1.4: Counting Parameters

- Between input and hidden: $3 \times 4 = 12$ weights, plus 4 biases.
- Between hidden and output: $4 \times 2 = 8$ weights, plus 2 biases.
- **Total**: $12 + 4 + 8 + 2 = 26$ parameters.

5.1.5 Solution 1.5: Gradient of a Single Neuron

By the chain rule:

$$\frac{\partial C}{\partial w} = (a - y)\, \frac{\partial a}{\partial z}\, \frac{\partial z}{\partial w} = (a - y)\, f'(z)\, x.$$

5.1.6 Solution 1.6: Universal Approximation Theorem

Universal Approximation Theorem: A feedforward neural network with a single hidden layer containing a finite number of neurons (with non-constant, bounded, and continuous activation functions) can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to any desired accuracy.

Implication: Depth is not strictly necessary for expressivity—width can suffice. However, deeper networks can achieve the same accuracy with far fewer units, improving parameter efficiency and often training time.

5.1.7 Solution 1.7: Batch vs. Stochastic Gradient Descent

SGD updates parameters using noisy estimates of the true gradient (computed on mini-batches). This noise can kick the optimizer out of small local minima or flat regions, enabling exploration of the cost surface. In contrast, full-batch updates follow a smooth, deterministic path that may get "stuck."

5.1.8 Solution 1.8: Learning-Rate Effects

- **Too large** γ: Steps overshoot the minimum, causing divergence or oscillation.
- **Too small** γ: Convergence is extremely slow, requiring many iterations to approach the minimum.
- **Ideal** γ: balances step size and stable descent.

5.1.9 Solution 1.9: One-Step GD Update

$$\nabla C = 2w + 4, \quad \nabla C|_{w=0} = 4.$$

Update:

$$w_1 = w_0 - \gamma \nabla C = 0 - 0.1 \times 4 = -0.4.$$

5.1.10 Solution 1.10: Parameter Count for Arbitrary Topology

- Weights: $\sum_{i=0}^{L} H_i \times H_{i+1}$, where $H_0 = d$, $H_{L+1} = K$.
- Biases: $\sum_{i=1}^{L+1} H_i$.
- **Total**: $\sum_{i=0}^{L} H_i H_{i+1} + \sum_{i=1}^{L+1} H_i$.

5.1.11 Solution 1.11: Role of Nonlinearity

Without nonlinearity, a stack of linear layers collapses to a single linear transformation. Nonlinear activations allow networks to approximate complex, non-linear mappings by composing simple non-linearities.

5.1.12 Solution 1.12: Steps of Backpropagation

1. **Forward pass**: Compute all activations layer by layer.
2. **Compute output error**: $\delta^L = \nabla_a C \odot f'(z^L)$.
3. **Backpropagate error**: For $l = L-1, \ldots, 1$: $\delta^l = (W^{l+1})^T \delta^{l+1} \odot f'(z^l)$.
4. **Compute gradients**: $\partial C / \partial W^l = \delta^l (a^{l-1})^T, \quad \partial C / \partial b^l = \delta^l$.
5. **Update parameters**: $W^l \leftarrow W^l - \gamma \partial C / \partial W^l$, etc.

5.1.13 Solution 1.13: Cost Function Choice for Classification

MSE leads to slow learning when using saturating activations (sigmoid), because gradients vanish. *Cross-entropy* loss combined with sigmoid or softmax outputs yields steeper gradients and faster convergence:

$$C = -\big[y \log a + (1-y) \log(1-a)\big].$$

5.1.14 Solution 1.14: Designing Deep Topologies

Each hidden layer composes progressively more abstract features:

- **Layer 1**: edges or simple patterns
- **Layer 2**: motifs or shapes
- **Layer 3+**: high-level concepts

Deeper nets can capture hierarchical structures more efficiently than shallow but wide nets.

5.2 Neural networks: Solution Set 2 (Numeric/Computational)

5.2.1 Solution 2.1: Activation-Function Plots

Solution plot is shown in Fig. 1.

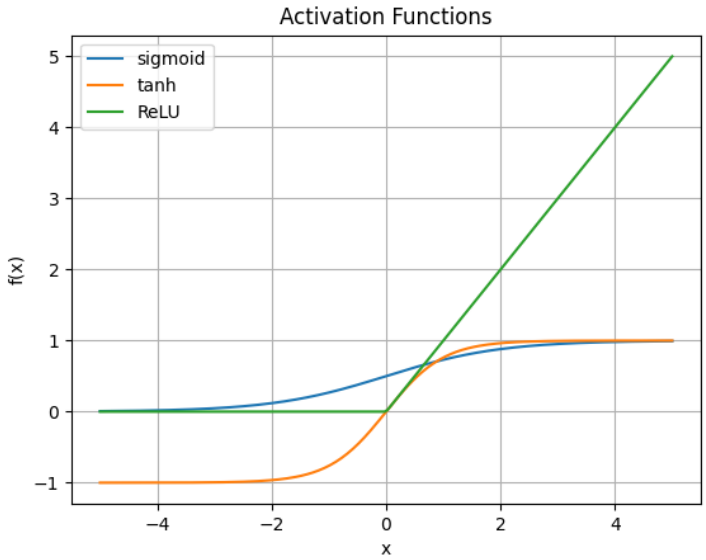

Figure 1: Solution for problem 1. See script `set2_prob1.py` for solution details.

5.2.2 Solution 2.2: Linear Regression by Gradient Descent

Fitted values: $w = 1.841$, $b = -0.034$. Solution plot is shown in Fig. 2.

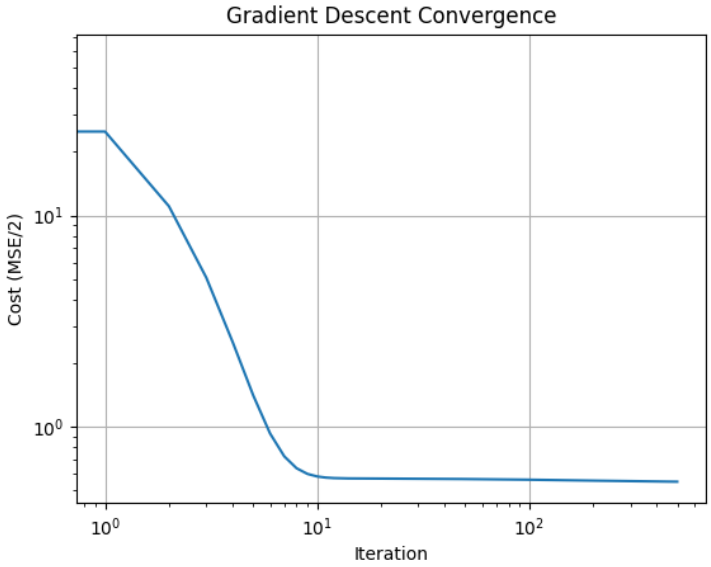

Figure 2: Solution for problem 2. See script `set2_prob2.py` for solution details.

5.2.3 Solution 2.3: Perceptron Learning Algorithm

Final weights: $w = [2.1.]$, $b = -2.0$. Solution plot is shown in Fig. 3.

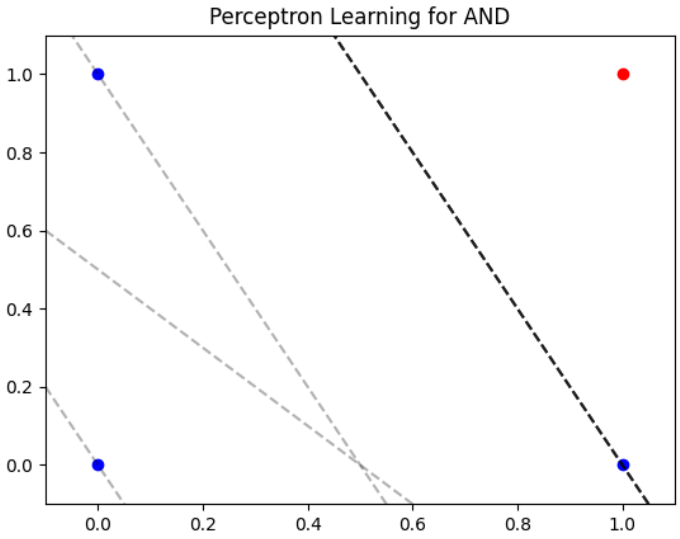

Figure 3: Solution for problem 3. See script `set2_prob3.py` for solution details.

5.2.4 Solution 2.4: Single-Example Backprop

The following values are computed:

$$\frac{\mathrm{d}C}{\mathrm{d}w_1} = 0.0214, \quad \frac{\mathrm{d}C}{\mathrm{d}w_2} = -0.0839$$

5.2.5 Solution 2.5: Cost Surface Contours

Solution plot is shown in Fig. 4.

5.2.6 Solution 2.6: Function Approximation with a 1-Hidden-Layer NN

Solution plot is shown in Fig. 5.

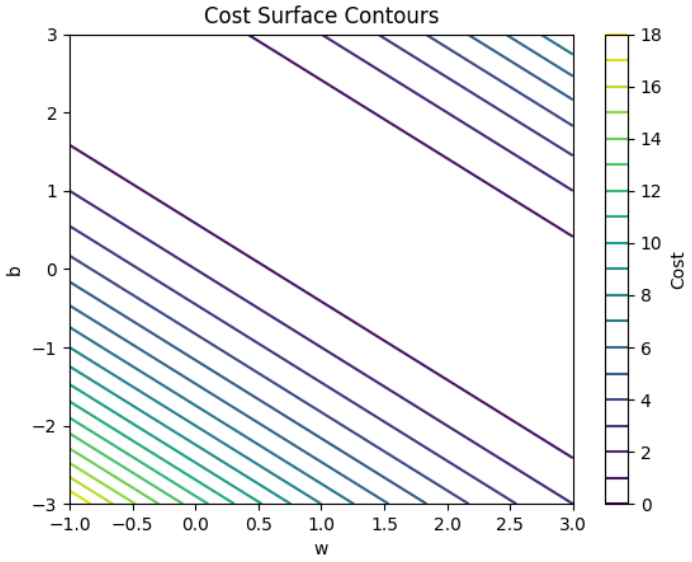

Figure 4: Solution for problem 5. See script `set2_prob5.py` for solution details.

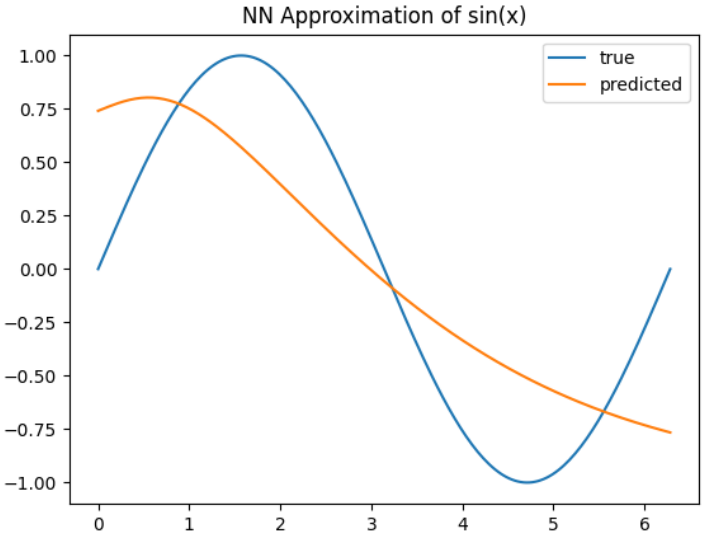

Figure 5: Solution for problem 6. See script `set2_prob6.py` for solution details.

5.3 Neural networks: Solution Set 3 (Analytic/Conceptual)

5.3.1 Solution 3.1: Why Use Neural Networks?

Hard-coded systems require explicit rules for every variation in input. Voice recognition has variability in accent, pitch, and background noise; autonomous vehicles face changing lighting, weather, and traffic. NNs, however, learn hierarchical data representations directly and generalize to unseen scenarios provided the training data is diverse. They adapt without manual reprogramming, making them superior in high-dimensional, variable environments.

5.3.2 Solution 3.2: Digital vs. Analog Accelerators

TinyML on ARM Cortex-M series offers deterministic precision (often 8-bit quantized) and ease of use via frameworks, but energy per MAC is on the order of picojoules. Analog MAC arrays can reach sub-picojoule per MAC, but require careful device calibration to correct variability and suffer lower numeric precision, often limiting them to inference-only tasks with retraining needs.

5.3.3 Solution 3.3: TinyML Constraints

SRAM budgets are typically 16–256 kB, so models must fit within that (often via 8-bit quantization and pruning). Real-time tasks demand inference latencies below ~ 10 ms, requiring efficient layer implementations and minimal dynamic memory usage.

5.3.4 Solution 3.4: Sub-mW Computing Strategies

1. **Duty-cycling**: The MCU sleeps until a sensor interrupt, drastically cutting active time.
2. **Near-threshold operation**: Lower supply voltage reduces energy per operation, at the cost of slower clocks.
3. **Model compression**: Pruning and weight sharing shrink model size and memory-access energy.

5.3.5 Solution 3.5: Feature Extraction Rationale

Raw streams are high-dimensional, leading to large models prone to overfitting. Compact features (RMS, zero-crossings, spectral bins) capture essential signal characteristics, reducing input size, memory footprint, and inference cost while often improving generalization.

5.3.6 Solution 3.6: Event-Driven Spiking Hardware

Spiking hardware processes inputs only when changes or "events" occur, turning off compute elements otherwise. This leads to near-zero dynamic switching for idle inputs, greatly reducing average power compared to continuously clocked digital logic.

5.3.7 Solution 3.7: Deployment Workflow

1. Train and quantize the CNN off-device (e.g., to 8-bit).
2. Convert to a TFLite flatbuffer .tflite.
3. Integrate the model into the MCU project, allocating tensor arena in SRAM.
4. Implement sensor data preprocessing (feature extraction) in C/C++.
5. Compile and flash the firmware; test inference latency and accuracy.

5.3.8 Solution 3.8: Model Compression Effects

Pruning removes redundant weights, reducing memory footprint and the number of MACs. Quantization lowers numeric precision, shrinking storage and enabling faster fixed-point operations. Together they can cut model size by 75% or more and similarly reduce energy per inference.

5.3.9 Solution 3.9: Cross-Validation Techniques

Hold-out splits data into fixed train/validation/test sets—fast but potentially high variance if data is limited. K-fold rotates which fold serves as validation, providing more robust performance estimates at the cost of k$\times$ longer training time. Use k-fold when datasets are small.

5.3.10 Solution 3.10: Confusion Matrix Interpretation

Accuracy can be misleading if one class dominates. Precision (TP/(TP+FP)) measures correctness of positive predictions; recall (TP/(TP+FN)) measures coverage of actual positives; F1 harmonizes both. For safety-critical tasks, high recall may be prioritized; for spam detection, high precision may be more important.

5.4 Neural networks: Solution Set 4 (Numeric/Computational)

5.4.1 Solution 4.1: Energy per MAC Comparison

- **Digital**: $E = CV^2 = 1 \times 10^{-12}\,\text{F} \times (1\,\text{V})^2 = 1 \times 10^{-12}\,\text{J}$.
- **Analog**: $E = 0.1 \times 10^{-12}\,\text{F} \times (1\,\text{V})^2 = 1 \times 10^{-13}\,\text{J}$.

5.4.2 Solution 4.2: RMS Acceleration from Sensor Data

- RMS_x=0.710 g, RMS_y=0.575 g, RMS_z=0.367 g.
- Histogram of a_x shown below in Fig. 6.

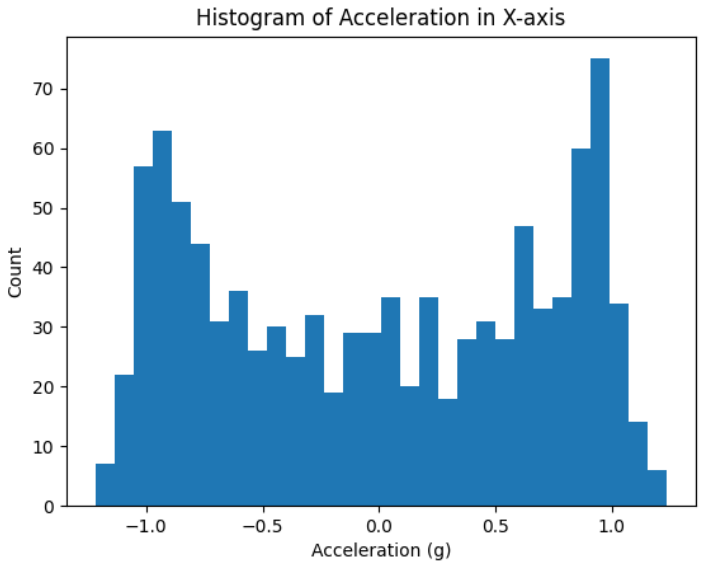

Figure 6: Solution for problem 2. See script `set4_prob2.py` for solution details.

5.4.3 Solution 4.3: Spectral Energy in Frequency Bands

- Energy 0–5 Hz: 242682.39
- Energy 5–10 Hz: 60757.54
- Power spectrum plotted below in Fig. 7.

5.4.4 Solution 4.4: Confusion-Matrix Metrics

- Accuracy = 0.85
- Precision = 0.83
- Recall = 0.91
- F1 = 0.87

5.4.5 Solution 4.5: Training and Validation Curves

- Plot, shown below in Fig. 8, shows training accuracy increasing linearly.
- Validation accuracy lags slightly, indicating mild generalization gap.

5.4.6 Solution 4.6: MAC Throughput & Inference Latency

- 64 MHz/2 cycles = 32 M MAC/s.
- 10 k MACs $\rightarrow$ 0.312 ms latency.

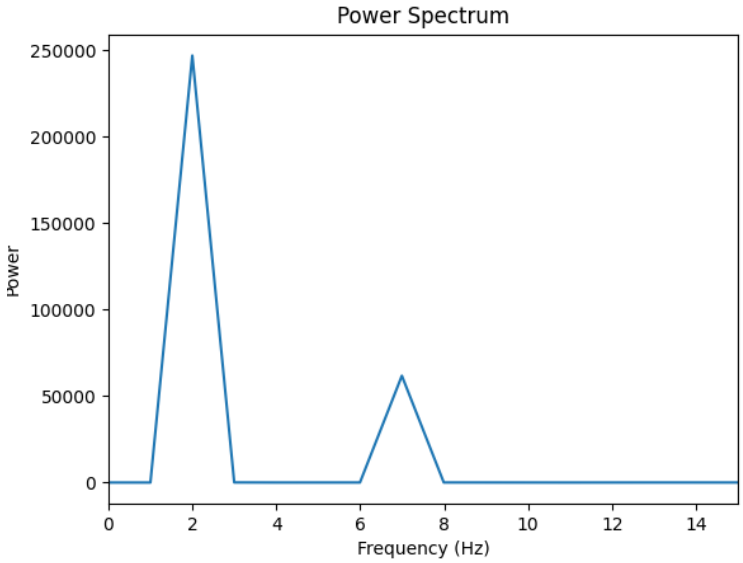

Figure 7: Solution for problem 3. See script `set4_prob3.py` for solution details.

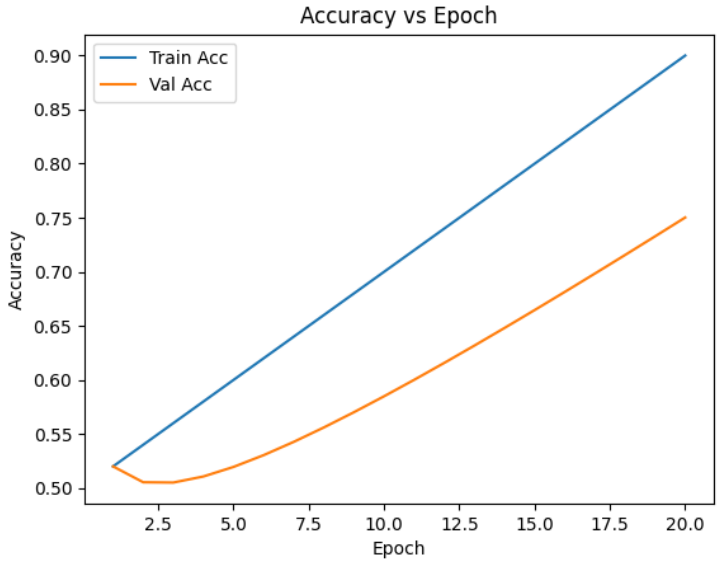

Figure 8: Solution for problem 5. See script `set4_prob5.py` for solution details.

5.4.7 Solution 4.7: Average Power with Duty-Cycling

$$P_{\text{avg}} = \frac{5\text{ms} \times 20\text{mW} + 95\text{ms} \times 0.1\text{mW}}{100\text{ms}} = 1.09\text{mW}.$$

5.4.8 Solution 4.8: 8-bit Quantization of Weights

- **Original**: [0.10, -0.50, 0.75]
- **Quantized**: [13, -64, 95]
- **Dequantized**: [0.102, -0.504, 0.748]

5.4.9 Solution 4.9: Delta MFCC Shape

- MFCC shape (13,100) → delta shape (13,99), representing first-order temporal derivatives.

5.4.10 Solution 4.10: Pruning Savings

- Memory saving = 75.0
- Therefore, estimated energy savings = 75.0%