Quantitative Macroeconomics w/ AI and ML

Lec. 5: Introduction to Machine Learning for Economists

Zhigang Feng

Dec., 2025

Why AI/ML for Economists

Harnessing the Power of Unstructured Data

- Economics has traditionally relied on structured, tabular data. However, a wealth of economic information is locked away in unstructured formats like text, images, and satellite imagery.
- ML algorithms, particularly from the field of deep learning, provide the tools to systematically process and quantify this unstructured data, turning it into valuable economic indicators.
- **Applications for Economists:**
  - Measuring economic sentiment and uncertainty from financial news articles and social media.
  - Analyzing satellite images to track economic activity, such as nighttime light intensity as a proxy for GDP or the number of ships in ports for trade flows.
  - Using image recognition on product pictures to understand non-standardized product features and competition.

Extracting Insights from Textual Data

- A significant portion of unstructured data is in the form of text. Natural Language Processing (NLP), a subfield of ML, offers powerful techniques to analyze and extract meaning from large volumes of text.
- These methods allow economists to move beyond simple word counts and capture nuanced information from documents like central bank statements, corporate filings, and political speeches.
- **Applications for Economists:**
  - Gauging the dovish or hawkish stance of central bankers from their public communications.
  - Identifying key risk factors for businesses from the text of their annual reports.
  - Quantifying the evolution of economic policy uncertainty by analyzing news articles.

Enhancing Causal Inference and Program Evaluation

- A central goal of economics is to identify the causal effects of policies and interventions. ML offers new methods to improve the credibility and precision of causal estimates.
- Techniques like "Double Machine Learning" allow for the flexible inclusion of many control variables (high-dimensional controls) to reduce the risk of omitted variable bias, a common challenge in econometric studies.
- ML is particularly powerful for identifying heterogeneous treatment effects, revealing how the impact of a policy or program varies across different subgroups of the population.
- **Applications for Economists:**
  - More accurately estimating the causal effect of a job training program by controlling for a vast number of individual characteristics.
  - Discovering for which types of students a new educational intervention is most effective.
  - Improving the selection of instrumental variables in complex economic settings.

Improving Economic Forecasting and Nowcasting

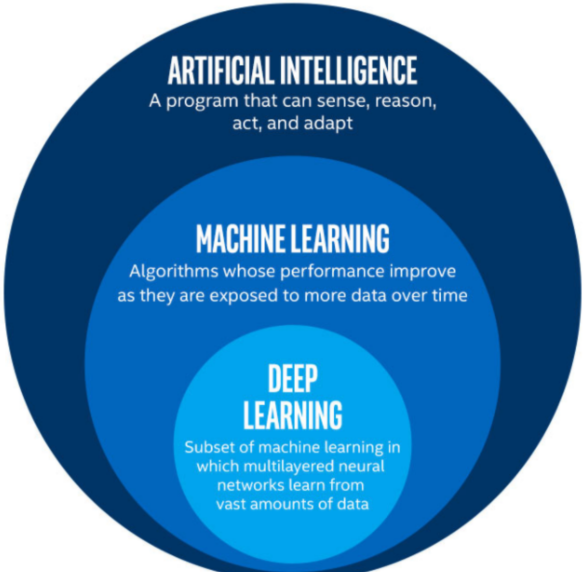
- While economists have long been in the business of forecasting, ML models can often improve predictive accuracy by capturing complex, non-linear relationships in the data that traditional time-series models might miss.
- ML excels at incorporating a massive number of predictors (the "kitchen sink" approach) without suffering from the same degree of overfitting as classical models, thanks to techniques like regularization (e.g., LASSO).
- "Nowcasting" involves predicting the state of the economy in the present or very near future. ML is adept at processing a high-frequency mix of data (e.g., credit card transactions, web searches, satellite imagery) to provide real-time economic indicators.
- **Applications for Economists:**
  - Developing more accurate forecasts of inflation or GDP growth by drawing on hundreds or even thousands of data series.
  - Creating real-time measures of unemployment or consumer spending by analyzing large, alternative datasets.
  - Building early warning systems for financial crises or economic recessions.

Solving Large-Scale Macro and Finance Models

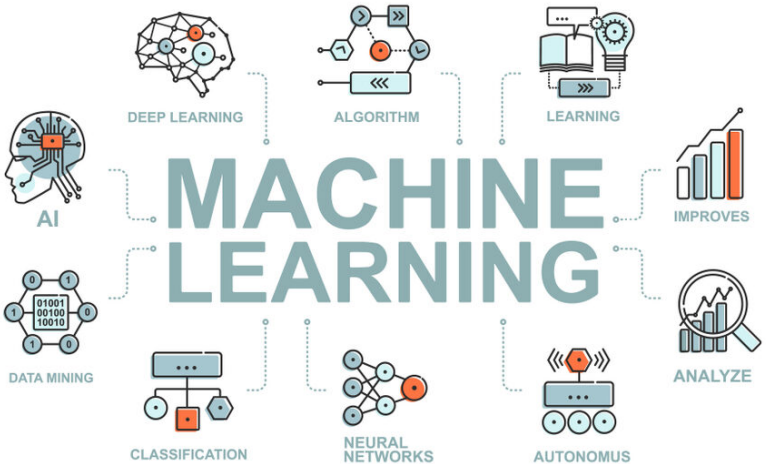
- Many cutting-edge macroeconomic and finance models, especially those featuring heterogeneous agents, are characterized by high dimensionality and complex, non-linear dynamics that make them difficult to solve with traditional methods.
- ML techniques, particularly deep learning and reinforcement learning, provide powerful new numerical methods to approximate the solutions to these complex models.
- **Applications for Economists:**
  - **Optimal Policies in Heterogeneous Agent Models:** Determining the optimal design of fiscal or monetary policy when considering the diverse circumstances of individual households and firms.
  - **Household Portfolio Choice over the Life Cycle:** Solving for the optimal savings and investment decisions of households facing realistic income uncertainty and a wide array of assets.

Machine Learning: Overview

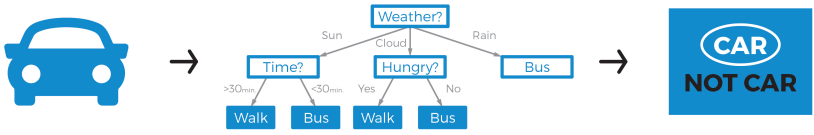
Machine Learning



Machine Learning

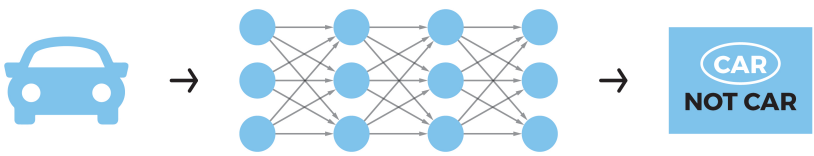


Machine Learning



Input | Decision tree | Output

Deep Learning



Input | Feature extraction + Classification | Output

Machine Learning (ML)

- ML is a subset of artificial intelligence that focuses on teaching computers to learn from data and improve their performance on a specific task without being explicitly programmed.
- It includes various techniques such as linear regression, logistic regression, decision trees, random forests, support vector machines, and more.
- ML algorithms learn from structured data and often require manual feature engineering to extract relevant features from raw data.
- They can handle smaller datasets and require less computational power compared to Deep Learning (DL).
- ML models are generally more interpretable and easier to understand than DL models.

Deep Learning (DL)

- DL is a subset of machine learning that uses artificial neural networks with multiple layers to learn from data.
- It can automatically learn and extract features from unstructured data like images, audio, and text.
- DL models consist of hierarchical layers that learn increasingly abstract representations of the input data.
- They require large amounts of data and significant computational power, often utilizing GPUs for training.
- DL excels in tasks such as image recognition, speech recognition, and natural language processing.
- DL models are often more complex and harder to interpret compared to traditional ML models.

Key differences between ML and DL

- Classical ML methods typically assume a **fixed, structured feature vector**: they can be applied to images or text, but usually *after* manual feature engineering. DL architectures are designed to take **raw, high-dimensional, "unstructured" signals** (pixels, audio waveforms, tokens) and learn useful features automatically.
- DL generally requires larger datasets and more computational power than ML.
- DL models are typically more complex and less interpretable than ML models.
- ML is often suitable when transparency and interpretability are first-order, while DL is better suited for complex tasks like image and speech recognition or working directly with raw economic text and imagery.
- ML and DL are both subfields of AI, with deep learning being a more specialized subset of machine learning.

Machine Learning and Macroeconomics

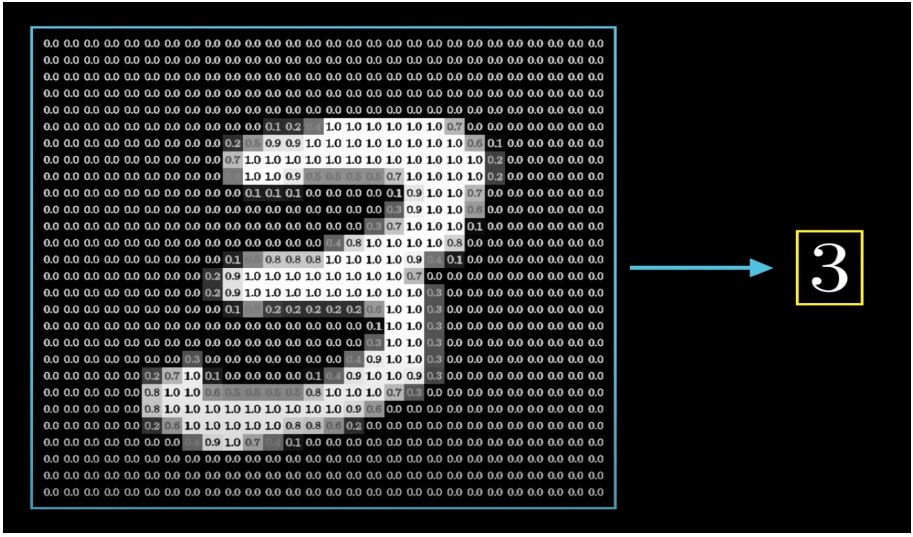
- Curse of Dimensionality (CoD) # 1: dimension of state space
  - $k \to \{k, a, h, ...\}$
  - Γ is an infinite dimensional object
- Curse of Dimensionality (CoD) # 2: unfolding of uncertainty
  - $\mathbb{E}$ over future contingencies.
  - the space for Γ is unknown

- Curse of Dimensionality (CoD) # 1: dimension of state space
  - Traditional methods
    - sparse grids, local approximation, etc.
    - construct grids to generate informations and to solve parameters
  - Machine learning
    - neural network, grid free
    - recover the equilibrium functions via Monte Carlo sampling

- Curse of Dimensionality (CoD) # 2: unfolding of uncertainty
  - Traditional methods
    - discretization and reduction of shock process,
    - impose Markovian
  - Machine learning
    - focus on the ergodic state
    - direct simulation Monte Carlo

How ML Works?

Example: Pattern Recognition



Hand-Written Digit Recognition as a Mathematical Problem

- **Dataset** $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ with images $\mathbf{x}_i \in [0,1]^{784}$ and labels $y_i \in \{0, \dots, 9\}$
- **Signal Extraction**
  - Raw 28×28 grayscale image $\mathbf{x}$
  - Optional preprocessing: centering, scaling, deskewing
- **Input–Output Mapping**

$$f_\theta \colon \mathbf{x} \longmapsto \hat{\mathbf{p}} \in \Delta^9, \qquad \hat{y} = \arg\max_k \hat{p}_k$$

  where Δ^9 is the 10-D probability simplex
- **Learning Objective (Empirical Risk + Reg.)**

$$\min_\theta \ \frac{1}{N} \sum_{i=1}^{N} \underbrace{\mathcal{L}\big(f_\theta(\mathbf{x}_i), y_i\big)}_{\text{cross-entropy}} + \lambda \|\theta\|_p$$

- **Search Strategy**
  - Stochastic gradient descent & variants (Adam, RMSProp)
  - Early stopping, learning-rate schedules, weight decay

Anatomical Constraints in Handwritten Digits

- The anatomy of the human hand and the biomechanics of handwriting influence the way digits are written
- Muscles, tendons, and joints in the hand and arm dictate the possible movements and strokes
- These anatomical constraints lead to certain patterns and variations in the strokes used to create each digit
  - For example, the natural curvature of the fingers and wrist can result in curved strokes, while the limited range of motion can affect the size and orientation of the digits
- Machine learning algorithms can learn these anatomically-driven patterns to better recognize and classify handwritten digits

Geometric Properties of Handwritten Digits

- Each handwritten digit has specific geometric properties that define its appearance
  - These properties include the presence of loops, lines, and curves, which create a set of constraints that limit the possible variations of each digit
  - For instance, the digit '0' is typically characterized by a closed loop, while the digit '1' is often represented by a single vertical stroke
  - The digit '8' combines two loops, one above the other, whereas the digit '7' is composed of a horizontal stroke followed by a diagonal line
- By learning these geometric properties, machine learning algorithms can effectively distinguish between different digits and handle variations within each class

Spatial Relationships in Handwritten Digits

- The spatial relationships between the components of a digit contribute to its recognizable structure
  - These relationships include the relative positions, sizes, and orientations of lines, curves, and loops within a digit
  - For example, in the digit '5', the vertical stroke is typically positioned to the left of the curved portion
  - In the digit '9', the loop on the top is usually smaller than the curved line on the bottom
- Consistency in these spatial relationships helps maintain the identity of the digit, even with individual variations
- Machine learning algorithms can capture these spatial relationships to improve their robustness and accuracy in recognizing handwritten digits

Leveraging Constraints for Improved Pattern Recognition

- Understanding the anatomical constraints, geometric properties, and spatial relationships in handwritten digits is crucial for developing effective machine learning models
  - By incorporating this knowledge into the design and training of the algorithms, we can improve their performance and generalization abilities
  - Techniques such as data augmentation (e.g., applying transformations to the images) can help the models learn these constraints and variations
  - Regularization methods, like dropout or weight decay, can prevent the models from overfitting to noise or irrelevant features
- Exploiting the inherent structure and constraints of the data leads to more efficient and accurate pattern recognition of handwritten digits

Reframing Handwritten Digit Recognition as Input-Output Function

- Input-Output Framework:
  - Handwritten digit recognition as an input-output function in the pixel space.
  - Input: Grayscale image of a handwritten digit, represented as a pixel intensity matrix.
  - Output: Class label (0-9) indicating the recognized digit.
- Dimensionality Reduction:
  - Leveraging anatomical constraints, geometric properties, and spatial relationships.
  - Applying techniques like feature extraction and dimensionality reduction to capture the most relevant information.

From 784 Dimensions to a Low–Dimensional Manifold

- **Manifold Hypothesis** Natural digit images occupy a smooth, d-dimensional manifold $\mathcal{M} \subset \mathbb{R}^{784}$ with $d \ll 784$.
- **Two–Stage Model**

$$\mathbf{x} \xrightarrow{\pi : \mathbb{R}^{784} \to \mathbb{R}^d} \mathbf{z} \xrightarrow{g_\phi : \mathbb{R}^d \to \Delta^9} \hat{\mathbf{p}}, \qquad \hat{y} = \arg\max_k \hat{p}_k$$

  - π: *projection / encoder* that flattens the manifold to a latent code $\mathbf{z} \in \mathbb{R}^d$.
  - g_ϕ: *classifier head* (1–2 fully-connected layers + soft-max) producing class probabilities $\hat{\mathbf{p}}$.

From 784 Dimensions to a Low–Dimensional Manifold (cont.)

- **Typical Choices for π**
  - **Linear:** Principal Component Analysis (PCA), Singular Value Decomposition (SVD), random projections.
  - **Non-linear:** Auto-encoder encoder, t-Distributed Stochastic Nbr. Emb. (t-SNE), Uniform Manifold Approx. & Projection (UMAP), first layers of a CNN.
- **Why Bother?**
  - Decision boundaries become simpler in $\mathbb{R}^d$.
  - Improves generalisation and optimisation speed.
  - Makes the geometry of digit classes (loops, strokes, etc.) explicit.

Defining a Deep Neural Network for Digit Recognition

- Network Architecture:
  - Input Layer: Neurons corresponding to pixel intensities of the digit image.
  - Hidden Layers: Multiple layers applying nonlinear transformations to learn complex patterns.
  - Output Layer: 10 neurons, each representing one digit class (0-9).
- Activation Functions:
  - Nonlinear activation functions (e.g., ReLU, sigmoid) between layers.
  - Customizable number of hidden layers and neurons per layer based on task complexity and computational resources.

CNN in Nutshell

CNNs as Learned Manifold Maps

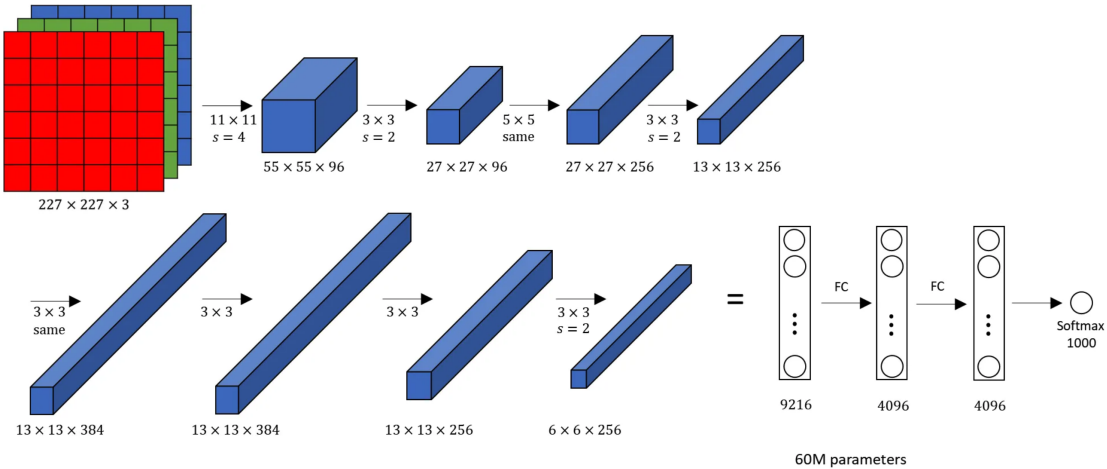
- Recall: we introduced a two–stage view

$$\mathbf{x} \xrightarrow{\pi} \mathbf{z} \xrightarrow{g_\phi} \hat{\mathbf{p}}$$

  where $\pi : \mathbb{R}^{784} \to \mathbb{R}^d$ projects images onto a low–dimensional manifold.

- **In a CNN:**
  - Stacked **convolution + nonlinearity + pooling** layers implement a learned π: they turn raw pixels into feature maps capturing edges, strokes, loops, etc.
  - The final feature tensor (just before fully–connected layers) is a learned low–dimensional representation $\mathbf{z}$ with much fewer degrees of freedom than the original 784 pixels.
- The fully–connected "classifier head" on top of the CNN is precisely the g_ϕ map that turns $\mathbf{z}$ into class probabilities.
- Economically: CNN layers learn a *sufficient statistic* for prediction, just like finding a low–dimensional state in a high–dimensional macro model.

AlexNet


227 × 227 × 3
11 × 11
s = 4
55 × 55 × 96
3 × 3
s = 2
27 × 27 × 96
5 × 5
same
27 × 27 × 256
3 × 3
s = 2
13 × 13 × 256
3 × 3
same
13 × 13 × 384
3 × 3
13 × 13 × 384
3 × 3
13 × 13 × 256
3 × 3
s = 2
6 × 6 × 256
=
9216
FC
4096
FC
4096
Softmax
1000
60M parameters

Step 1: Input $\rightarrow$ 1st Convolution (11$\times$11, stride 4)

- **Raw input tensor**

$$227 \times 227 \times 3$$

  - 227×227: spatial pixels
  - 3: RGB colour channels

- **Conv-1 hyper-parameters**

$$\text{kernel } F = 11, \quad \text{stride } S = 4, \quad \text{padding } P = 0, \quad \text{filters } K = 96$$

- **Output size calculation**

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - F + 2P}{S} \right\rfloor + 1 = \left\lfloor \frac{227 - 11}{4} \right\rfloor + 1 = 55$$

$$\Longrightarrow \quad 55 \times 55 \times 96$$

- **Intuition** Large 11×11 receptive fields sampled every 4 pixels aggressively down-sample the image while each of the 96 filters learns a distinct low-level pattern (edge, colour blob, ...).

Step 2: After Pooling → "Same" Convolution (5×5, stride 1)

- **Input tensor (after 2×2 max-pool)**

$$27 \times 27 \times 96$$

- **Conv-2 hyper-parameters**

$$F = 5, \quad S = 1 \text{ (default)}, \quad \text{padding } P = 2 \text{ (\textit{same})}, \quad K = 256$$

- **Why "same" padding?** Choosing $P = \frac{F-1}{2} = 2$ keeps borders intact:

$$W_{\text{out}} = \left\lfloor \frac{27 - 5 + 2 \times 2}{1} \right\rfloor + 1 = 27,$$

  preserving spatial size while deepening the feature stack.

- **Resulting tensor**

$$27 \times 27 \times 256$$

- **Interpretation** *256* different 5×5 filters scan every pixel ($S = 1$) to capture mid-level motifs—corners, textures, small shapes.

"Same" $\Rightarrow$ output width $\approx$ input width when $S = 1$; formula: $W_{\text{out}} = \lceil W_{\text{in}}/S \rceil$.

Image Classification:Network Architecture

- *Input Layer:*
  - Receives raw image pixels (e.g., 224×224 for AlexNet).
  - Think of each pixel as an "observable data point."
- *Convolutional Layers:*
  - *Strides* (s)*:* Controls how far the filter moves each step.
  - *Padding* (p)*:* Adds zeros around the image to preserve spatial size.
  - Captures local patterns (like local economic multipliers).
- *Max Pooling Layer:*
  - Reduces spatial dimension by taking the maximum over a small region.
  - Analogous to summarizing localized data points into a single "max" indicator.
- *Fully Connected Layers:*
  - Flatten the output of convolutions and poolings into a single vector.
  - Combine learned features for final classification.
- *Output Layer:*
  - Predicts probabilities for each class (e.g., 1000 categories in ImageNet).
  - Each output neuron $\rightarrow$ one class, like categories in an economic dataset.

- **Activation Functions:**
  - *Nonlinear Mapping:*

$$\text{ReLU}(x) = \max(0, x),$$

  - *Economist Perspective:* Think of ReLU as a "shock function" that only activates if $x > 0$. Similar to thresholds or kinked demand curves where effects "turn on" past a boundary.

Image Classification – Feature Extraction

- **Convolution (learned filter):**

$$Y(i,j) = \sum_m \sum_n X(i-m,\ j-n)\, W(m,n)$$

  - *Intuition:* Slide a small "stamp" W over the picture; at every location multiply-and-add to test *how much that pattern is present*.
  - **Stride** S: How many pixels the filter jumps per step (e.g., every 1st, 2nd, 4th pixel).
  - **Padding** P: Add a border of zeros so edge pixels get equal attention.
- **Pooling (spatial summary):**

$$\text{MaxPool}(i,j) = \max_{(m,n)\in R(i,j)} X(m,n)$$

  - *Intuition:* Within each local region R, keep only the strongest signal – similar to reporting "highest value in each neighborhood".
  - Reduces spatial dimensions and provides *translation invariance* (small shifts $\approx$ same maximum response).

Image Classification – Decision Stage

- **Softmax layer:**

$$P(\text{class} = i) = \frac{\exp(z_i)}{\sum_{j=0}^{K-1} \exp(z_j)}$$

  - *Intuition:* Convert raw scores z_i (logits) into probability distribution that sums to 1.
  - Amplifies differences between scores: highest score dominates.

- **Cross-Entropy loss:**

$$\mathcal{L} = -\sum_{k=0}^{9} y_k \log(\hat{y}_k)$$

  - y_k = one-hot ground truth (1 for correct class, 0 otherwise)
  - $\hat{y}_k$ = predicted probability for class k
  - *Intuition:* Measures prediction quality: small when confident and correct, large when confident but wrong.

Training the Neural Network with Backpropagation

- Training Process:
  - Using a labeled dataset of handwritten digits.
  - Repeatedly feed forward batches of data, compute losses, backpropagate errors, and update weights.
  - Adjusting weights and biases to minimize prediction error.
  - Over time, the network "converges" to capture underlying patterns.
- Backpropagation:
  - Calculating gradients of the loss function w.r.t weights and biases via the chain rule (backpropagation).
  - Updating parameters using optimization techniques (e.g., SGD, Adam).
- Learning Rate and Regularization:
  - Adjusting learning rate for speed and stability of convergence.
  - Applying regularization techniques (e.g., L1/L2 regularization, dropout) to prevent overfitting.
- PyTorch's torch.nn Module
  - Provides building blocks for creating neural networks
  - Predefined layers, activation functions, and loss functions

Neural Networks

Neural Networks

- Artificial Neural Networks Basics
  - Inspired by biological neural networks
  - Composed of interconnected neurons organized in layers
  - Learn by adjusting weights based on training data

Neural Networks

- Neuron: The basic unit of a neural network, inspired by neurons in the human brain. It receives inputs, applies a function (the activation function) to them, and produces an output.
- Activation Function: The function applied by a neuron to its inputs to produce an output.
  - Common examples include the sigmoid, tanh, and ReLU functions.
- Loss Function: A measure of how well the model's predictions match the true values. The goal of training a machine learning model is to minimize the loss function.

Deep Neural Networks: single hidden-layer

- Let's represent the input as $x \in \mathbb{R}^M$ and the output as $y \in \mathbb{R}^N$.
- The i-th output of y, can be computed as a function of the input x as follows:

$$h_j^{(1)} = f_j^{(1)}(x) = \sigma\left(b_j^{(1)} + \sum_{m=1}^{M} w_{j,m}^{(1)} x_m\right),\ j = 1, \ldots, U_1. \tag{1}$$

$$y_i = \sum_{j=1}^{U_1} \tilde{w}_{i,j} h_j^{(1)} + \tilde{b}_i,\ i = 1, \ldots, N. \tag{2}$$

  where σ is a nonlinear activation function.

- Common activation functions include: $\sigma(x) = \tanh(x)$, $\sigma(x) = \max(0, x)$, and $\sigma(x) = \frac{1}{1+e^{-x}}$.

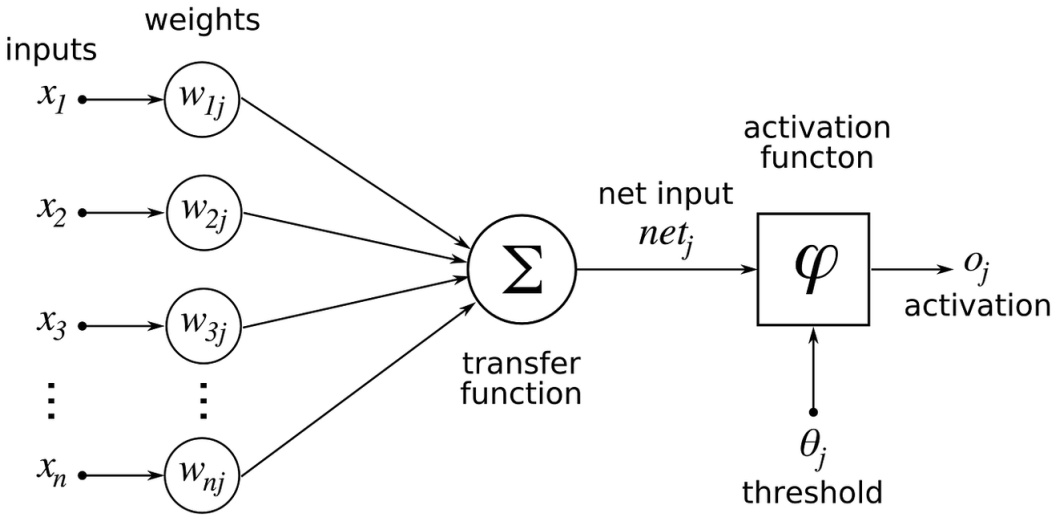
Deep Neural Networks: two hidden-layer

- We take the linear combination of $h^{(1)}$ and pass the result through a nonlinear activation function.

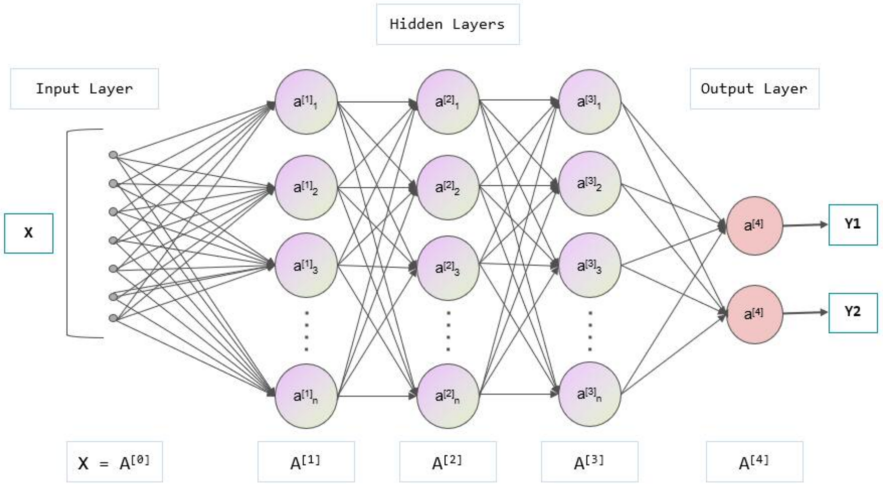
$$h_j^{(2)} = f_j^{(2)}(x) = \sigma\left(b_j^{(2)} + \sum_{k=1}^{U_1} w_{j,k}^{(2)} f_k^{(1)}(x)\right),\ j = 1, \ldots, U_2. \tag{3}$$

$$y_i = \sum_{j=1}^{U_2} \tilde{w}_{i,j} h_j^{(2)} + \tilde{b}_i,\ i = 1, \ldots, N. \tag{4}$$

Deep Neural Networks:



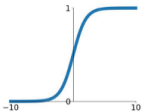
Hidden layers



Activation Functions

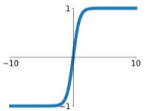
Sigmoid

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



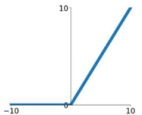
tanh

$\tanh(x)$



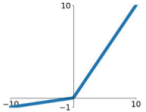
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

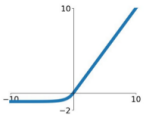


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Function Approximation with ReLU

- **Goal:** Approximate arbitrary nonlinear functions using simple building blocks.
- **Key Idea:** Use a linear combination of shifted ReLU functions to approximate a continuous function.
- **ReLU Function:**

$$\text{ReLU}(x) = \max(0, x)$$

  - Zero for negative inputs.
  - Linear (identity) for positive inputs.
- **Interpretation:**
  - *Shifts:* Define the activation (kink) points capturing changes in marginal slope.
  - *Coefficients:* Scale the contribution of each ReLU to approximate the local slope.

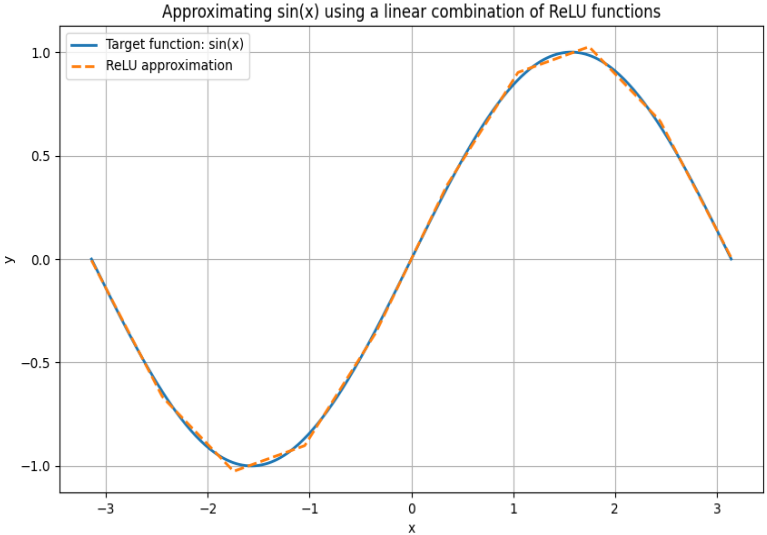
Example: Approximating $\sin(x)$ with ReLU Functions

- **Target Function:** $\sin(x)$ on $[-\pi, \pi]$
- **Design:** Use a linear combination of shifted ReLU functions:

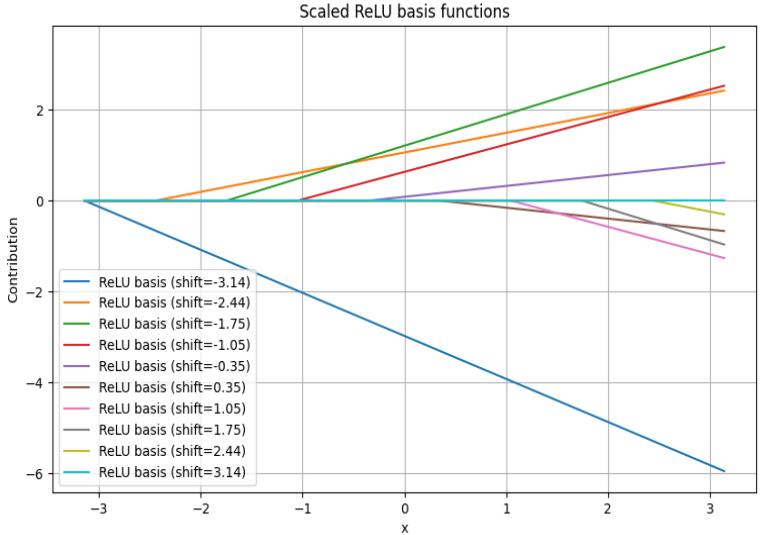
$$\hat{y}(x) = a_0 + \sum_{i=1}^{N} a_i \, \mathsf{ReLU}(x - b_i)$$

- **Implementation:**
  - Generate data points for x and $\sin(x)$.
  - Choose shifts b_i evenly over the domain.
  - Form a design matrix with columns $\mathsf{ReLU}(x - b_i)$ and a bias term.
  - Solve a least-squares problem to obtain the coefficients a_i.
- **Result:** The approximation closely follows $\sin(x)$, demonstrating the power of even simple ReLU functions.

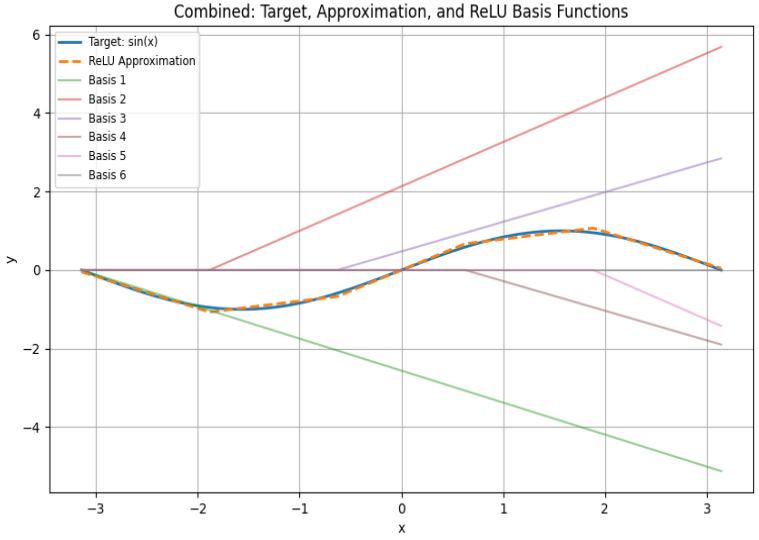
Activation functions



Activation functions



Activation functions



Deep Neural Networks:

- DNN is a combination of linear transformation followed by a non-linear activation function.
- Linear Transformation:
  - each neuron performs a linear transformation on the input data using its weights and biases.
  - akin to projecting the input data onto a new axis (but not necessarily orthogonal as standard approximation);
  - the set of linear transformations (weighted sums) across all neurons in a layer collectively creates a transformed space, or a new representation of the data.
    - Flexibility in Feature Representation;
    - compressing information, distilling the input into a more manageable set of features;
    - capture interactions between inputs in a much more flexible manner.

Deep Neural Networks:

- Non-Linear Activation Function:
  - non-linear activation functions allow neural networks to capture non-linearity;
  - each neuron can now represent more complex patterns than a simple projection;
  - can be thought of as creating a more complex, curved space in which the relationships between data points can be more intricately represented;
  - the non-linearity is essential for the universal approximation property of neural networks.

Some theories

Let us consider the two-layer neural network hypothesis space:

$$\mathcal{H}_m = \left\{ f_m(\mathbf{x}) = \sum_{j=1}^{m} a_j \sigma\left(w_j^T \mathbf{x}\right) \right\}$$

where m is the number of free parameters, σ is activation function. γ will encapsulate all free parameters that characterize Γ_γ.

Theorem (Universal approximation theorem, Cybenko (1989), Theorem 5)
If σ is sigmoidal, in the sense that $\lim_{z\to-\infty}\sigma(z)=0$, $\lim_{z\to+\infty}\sigma(z)=1$, then any function in $C([0,1]^d)$ can be approximated uniformly by two-layer neural network functions.[1]

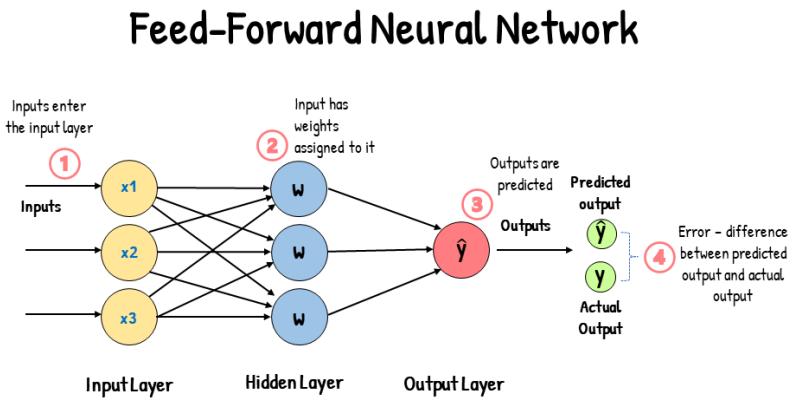
[1] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

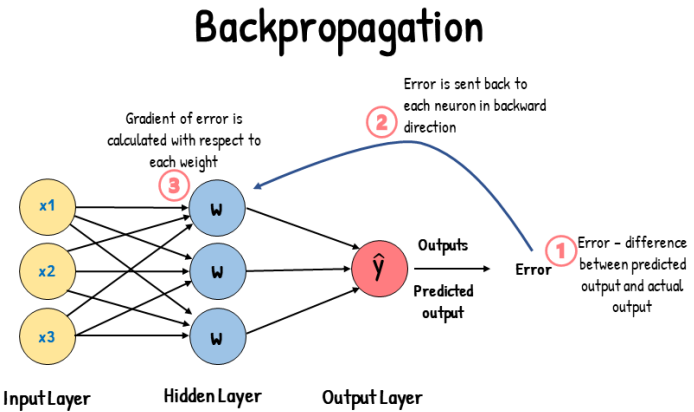
Theorem (Breaking the curse of dimensionality, Barron (1993))
A one-layer neural network achieves integrated square errors of order $O(1/M)$*, where* M *is the number of nodes. In comparison, for series approximations, the integrated square error is of order* $O(1/(M^{2/N}))$ *where* N *is the dimensions of the function to be approximated.*[2]

[2] A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.

Feed-Forward

- Feedforward and Backpropagation Algorithms
  - Feedforward: compute output given input through network layers
  - Backpropagation: compute gradients to update the weights and biases of a neural network by propagating the loss backwards through the network.


Feed-Forward Neural Network
Inputs enter the input layer
1
Inputs
x1
x2
x3
Input has weights assigned to it
2
w
w
w
Outputs are predicted
3
ŷ
Outputs
Predicted output
ŷ
y
Actual Output
4
Error – difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
W
W
W
ŷ
Outputs
Predicted output
Error
1
Error – difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Toy Example: Forward Pass in a Tiny Network

Consider a one-dimensional toy network with:

- Input: $x = 1.0$
- Hidden layer: 1 neuron with sigmoid activation

$$z_1 = w_1 x + b_1, \quad h = \sigma(z_1)$$

- Output layer: 1 neuron with sigmoid activation

$$z_2 = w_2 h + b_2, \quad \hat{y} = \sigma(z_2)$$

- Loss: squared error $L = \frac{1}{2}(\hat{y} - t)^2$ with target $t = 0.0$

Toy Example: Forward Pass in a Tiny Network

Take concrete parameter values:

$$w_1 = 0.5,\ b_1 = 0,\ w_2 = 2.0,\ b_2 = 0$$

$$z_1 = 0.5 \Rightarrow h = \sigma(0.5) \approx 0.62$$

$$z_2 = 2.0 \cdot 0.62 \approx 1.24 \Rightarrow \hat{y} = \sigma(1.24) \approx 0.78$$

$$L = \tfrac{1}{2}(0.78 - 0)^2 \approx 0.30$$

This is a full forward pass: from x to $\hat{y}$ to loss L.

Toy Example: Backpropagation Step (Numbers)

Now propagate the error backwards using the chain rule.

- Output layer:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - t \approx 0.78$$

$$\frac{\partial \hat{y}}{\partial z_2} = \hat{y}(1 - \hat{y}) \approx 0.78 \times 0.22 \approx 0.17$$

$$\Rightarrow \frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \approx 0.78 \times 0.17 \approx 0.13$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial w_2} = 0.13 \cdot h \approx 0.13 \cdot 0.62 \approx 0.08$$

Toy Example: Backpropagation Step (Numbers)

- Hidden layer:

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial h} = 0.13 \cdot w_2 \approx 0.26$$

$$\frac{\partial h}{\partial z_1} = h(1-h) \approx 0.62 \times 0.38 \approx 0.24$$

$$\Rightarrow \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial z_1} \approx 0.26 \cdot 0.24 \approx 0.06$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial w_1} = 0.06 \cdot x \approx 0.06$$

With learning rate $\eta = 0.1$:

$$w_2^{\text{new}} \approx 2.0 - 0.1 \times 0.08 = 1.992, \quad w_1^{\text{new}} \approx 0.5 - 0.1 \times 0.06 = 0.494$$

This is exactly what PyTorch's `.backward()` and the optimizer do for you, just at massive scale.

Gradient descent

- Stochastic batch gradient descent (BGD) method is a popular approach to reduce the cost of computing $\frac{d\mathcal{L}\left(\Gamma_{\gamma^{(j)}}\right)}{d\gamma}$

$$\frac{d\mathcal{L}\left(\Gamma_{\gamma^{(j)}}\right)}{d\gamma} \approx \frac{1}{n}\sum_{i=1}^{n}\frac{d\mathcal{L}\left(\Gamma_{\gamma^{(j)}}; b_i\right)}{d\gamma}$$

  - $n = 1$: stochastic gradient (SGD) method which approximates the expectation function with the value of such a function in one randomly chosen data point;
  - $n = N$: the conventional gradient descent (GD) method in which all data points are used for constructing the gradient.

Gradient descent

- Root Mean Square Propagation, RMSProp: the learning rates of all parameters are adjusted individually by making a step that is inversely proportional to the square root of the exponentially moving average of the previous squared values $g_k \odot g_k$ of the gradient g_k.

$$
\begin{aligned}
g_k &\leftarrow \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L} \left(\Gamma_{\gamma^{(j)}}; b_i \right) \\
r_{k+1} &\leftarrow \rho r_k + (1-\rho) g_k \odot g_k \\
\triangle \theta_{k+1} &= -\frac{\lambda}{\sqrt{\delta + r_{k+1}}} \odot g_k \\
\theta_{k+1} &\leftarrow \theta_k + \triangle \theta_{k+1}
\end{aligned}
$$

Gradient descent

- In a momentum algorithm, there are two additional parameters, a velocity vector v and a hyperparameter $\alpha \in [0, 1)$; the latter determines how quickly the effect of the previous gradients g_k decreases.
- Adaptive Moments (ADAM) calculates an exponential moving average of the gradient and the squared gradient, and use two parameters ρ_1 and ρ_2 to control the decay rates of these moving averages.

Gradient descent: algorithm

- Make an initial guess on the parameters vector
- For $n = 1, 2, ...,$ do the following:
  - draw a random realization for b_n
  - compute a stochastic vector of gradients $g(b_n; \gamma_n)$
  - choose a learning rate $\lambda_n > 0$
  - compute the new parameters vector as $\theta_{n+1} \leftarrow \theta_n - \lambda_n g(b_n; \gamma_n)$
- End iterations when convergence is achieved.

Gradient descent: convergence

- Assumption 1: the objective function F is continuously differentiable; the gradient of F is Lipschitz continuous with a constant $L > 0$.
- Assumption 2: the objective function F and the algorithm satisfies the following three properties:
  - The sequence θ_n is in an open set where the objective function is bounded from below by a scalar F_{inf} for all n;
  - There exists scalars μ_G and μ such that $\mu_G \geq \mu > 0$ and for all n, we have

$$\nabla F(\theta_n)\mathbb{E}_{b_n}\left[g(b_n;\theta_n)\right] \geq \mu||\nabla F(\theta_n)||^2$$
$$||\mathbb{E}_{b_n}\left[g(b_n;\theta_n)\right]|| \leq \mu_G||\nabla F(\theta_n)||$$

  - There exists scalars $H \geq 0$ and $H_V \geq 0$ such that for all n, we have

$$\mathbb{E}_{b_n}\left[||g(b_n;\theta_n)||^2\right] - ||\mathbb{E}_{b_n}\left[g(b_n;\theta_n)\right]||^2 \leq H + H_v||\nabla F(\theta_n)||^2.$$

Gradient descent: convergence

- Theorem (Nonconvex objective and a diminishing learning rate): Suppose Assumptions 1 and 2 hold. Assume that a sequence $\{\lambda_n\}$ satisfies

$$\triangle_n \equiv \sum_{n=1}^{N} \lambda_n = \infty, \sum_{n=1}^{N} \lambda_n^2 < \infty$$

then the expected sum of squares of the gradients of $F(\theta_n)$, weighted by λ_n, satisfies

$$\mathbb{E}\left[\sum_{n=1}^{N} \lambda_n ||F(\theta_n)||^2\right] < \infty$$

and the expected average squared gradients of $F(\theta_n)$, weighted by λ_n, satisfies

$$\mathbb{E}\left[\frac{1}{\triangle_n}\sum_{n=1}^{N} \lambda_n ||F(\theta_n)||^2 \underrightarrow{N \to \infty} 0\right].$$

- Bottou, L., Curtis, F., Nocedal, J., 2018. Optimization methods for large-scale machine learning. Manuscript.

Stopping Criteria for Training

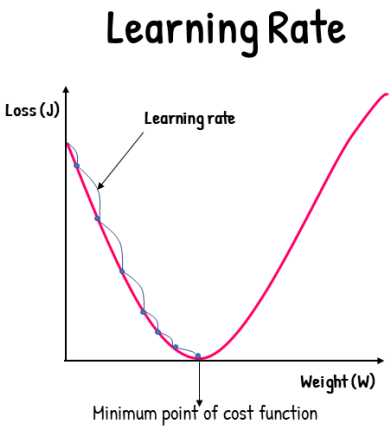
- When to Stop Training:
  - Fixed number of epochs.
  - Monitoring validation loss.
  - Early stopping when validation loss increases, indicating potential overfitting.
- Model Selection:
  - Choosing the model with the best performance on the validation set.

Training of NN

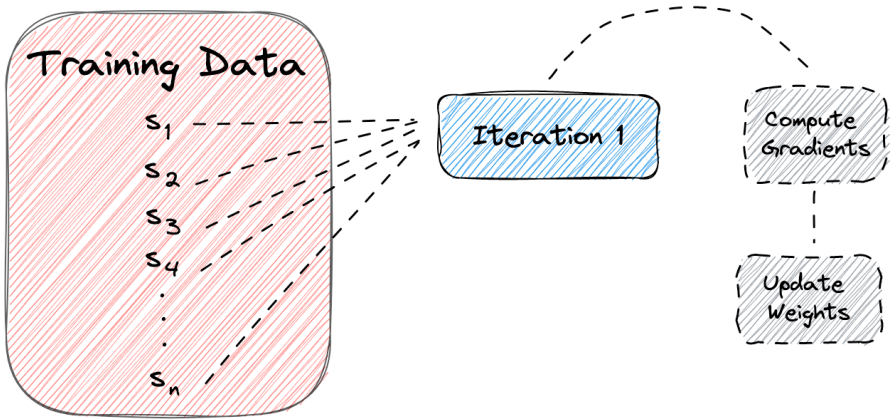
- Automatic Differentiation: A set of techniques for computing derivatives efficiently, which are crucial for back-propagation.
- Mini-Batch: A small subset of the training data. Instead of updating the model based on the entire dataset (batch gradient descent) or a single example (stochastic gradient descent), mini-batch gradient descent updates the model based on a mini-batch of examples.
- Epoch: One complete pass through the entire training dataset during the training of a machine learning model.
- Episodes: This term is often used in the context of reinforcement learning, where an agent interacts with an environment over a sequence of steps. An episode is one sequence of steps from the beginning to the end of the environment.

Local Optima vs. Global Optimum

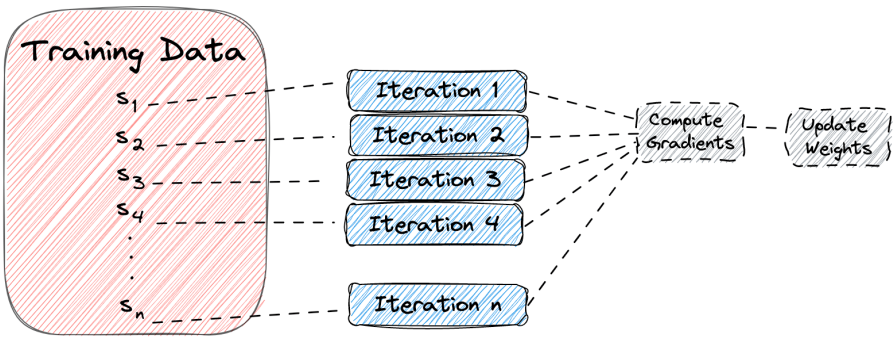
- Non-convex Loss Function:
  - Neural network loss functions are typically non-convex with multiple local optima.
- Local vs. Global Optima:
  - Local Optima: Points where the loss is minimized locally but not globally.
  - Global Optimum: The absolute minimum point of the loss function.
- Mitigating Local Optima:
  - Random initialization of weights.
  - Using momentum or adaptive learning rates.
  - Techniques like simulated annealing or genetic algorithms.
- Practical Considerations:
  - Finding a good local optimum that generalizes well is often sufficient for successful recognition.


Learning Rate
Loss (J)
Learning rate
Weight (W)
Minimum point of cost function

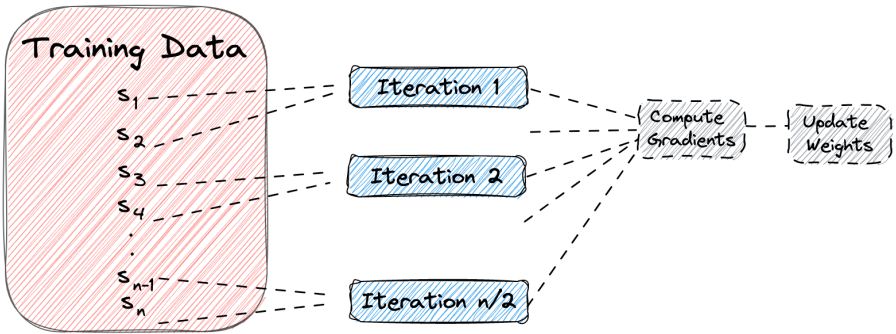
Batch Gradient Descent



Stochastic Gradient Descent

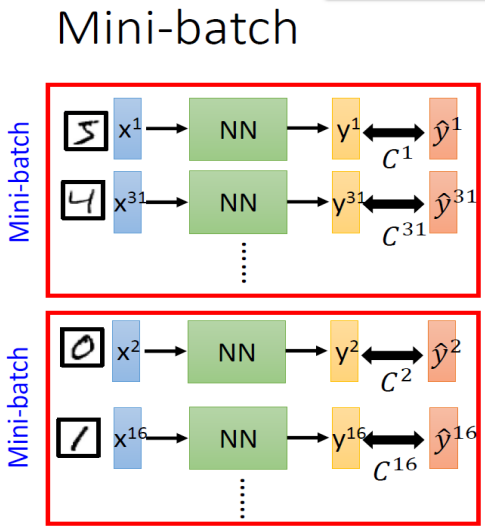


Stochastic Gradient Descent



Mini-batches

We do not really minimize total loss!



- Randomly initialize network parameters
- Pick the 1st batch
  $L' = C^1 + C^{31} + \cdots$
  Update parameters once
- Pick the 2nd batch
  $L'' = C^2 + C^{16} + \cdots$
  Update parameters once
  ⋮
- Until all mini-batches have been picked

one epoch

Repeat the above process

Why mini-batch?

- Computational efficiency: Using the entire dataset to compute the gradient at each iteration can be computationally expensive, especially for large datasets. By using mini-batches, we can update the model parameters more frequently, which can lead to faster convergence. Mini-batches allow for a balance between the speed of SGD and the stability of GD.
- Memory limitations: When dealing with large datasets, it may not be possible to load the entire dataset into memory at once. Mini-batches allow us to process the data in smaller subsets that can fit into memory, making it possible to train models on large datasets that would otherwise be infeasible.

Why mini-batch? (cont.)

- Stochastic approximation: Mini-batches introduce a level of stochasticity in the gradient estimation, which can help the optimization process escape local minima and saddle points. This stochasticity is more pronounced than in GD but less than in SGD, providing a balance between exploration and exploitation in the parameter space.
- Regularization effect: The noise introduced by mini-batches can have a regularizing effect on the model, helping to prevent overfitting. The stochasticity in the gradient estimation can act as a form of implicit regularization, similar to the effect of adding random noise to the input data or model parameters.
- Parallelization: Mini-batch gradient descent is well-suited for parallelization across multiple CPUs or GPUs. Each mini-batch can be processed independently, allowing for efficient utilization of hardware resources and faster training times.

Sample code

```
n1, n2 = 100, 50
n_input, n_output = 1, 1

decision_gk = nn.Sequential(
    nn.Linear(n_input, n1),
    nn.ReLU(),
    nn.Linear(n1, n2),
    nn.ReLU(),
    nn.Linear(n2, n_output),
    nn.Sigmoid()
)
```

Neural Network Model: Class-Based Implementation

- We define a **class-based neural network** using PyTorch's `nn.Module`.
- The model consists of **three fully connected layers** with **ReLU activation** functions.
- Using `nn.Sequential` simplifies the structure and makes the model modular.

```python
import torch.nn as nn

class KinkNet(nn.Module):
    def __init__(self, input_dim, hidden_dim1,
                 hidden_dim2, output_dim=1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim1),
            nn.ReLU(),
            nn.Linear(hidden_dim1, hidden_dim2),
            nn.ReLU(),
            nn.Linear(hidden_dim2, output_dim)
        )

    def forward(self, x):
        return self.net(x)
```

Case Study: Fitting Data

Problem Statement and Motivation

- **Objective:** Approximate a given dataset using multiple methods:
  - Deep Neural Networks (DNNs)
  - Traditional Polynomial Approximation
  - Chebyshev Polynomial Approximation
- **Motivation:** Compare how modern machine learning methods (DNN) perform against classical techniques.
- **Challenges:**
  - Capturing nonlinearities and discontinuities.
  - Avoiding numerical issues such as the Gibbs phenomenon.

Workflow for Function Approximation

1. **Data Generation and Loading:**
   - Create a dataset with a "kink" (non-smooth behavior) to test approximation methods.
   - Load the data using PyTorch's `Dataset` and `DataLoader`.
2. **Method Implementation:**
   - **DNN:** Define a neural network (`KinkNet`) and train it with MSE loss.
   - **Polynomial Approximation:** Use `np.polyfit` to fit a polynomial to the data.
   - **Chebyshev Approximation:** Construct Chebyshev basis functions and solve a least-squares problem.
3. **Training and Evaluation:**
   - Train the DNN model with an AdamW optimizer and monitor convergence with a progress bar.
   - Compare the mean squared error (MSE) of all methods.
4. **Visualization:** Plot the approximated functions versus the true data.

Discussion: Pros, Cons, and Gibbs Phenomenon

- **Polynomial Approximation:**
  - **Pros:**
    - Simple and fast to compute.
    - Closed-form solution via linear algebra.
  - **Cons:**
    - May exhibit **Gibbs phenomenon** near discontinuities or sharp kinks.
    - Higher-degree polynomials can lead to overfitting or oscillations.
- **Chebyshev Approximation:**
  - **Pros:**
    - Better numerical stability compared to ordinary polynomials.
    - Reduced oscillations due to optimal placement of Chebyshev nodes.
  - **Cons:**
    - Still susceptible to Gibbs phenomenon in the presence of discontinuities.
    - Requires transformation/scaling of input data.

Discussion: Pros, Cons, and Gibbs Phenomenon

- **DNNs:**
  - **Pros:**
    - Highly flexible in capturing complex nonlinear relationships.
    - Robust to noise and can generalize well with enough data.
  - **Cons:**
    - Require careful tuning of hyperparameters.
    - Training can be computationally expensive.
- Sample codes: kink_approximate.py; kink_3d_approximate.py

Generate Data

```python
def generate_kinked_data(x_size, alpha, beta, device):
    """
    Returns:
      x: Tensor of shape (x_size, 1)
      y: Kinked function values at each x
    """
    x = torch.linspace(0, 1.0, x_size, device=device).unsqueeze(1)
    y = alpha * beta * x**alpha

    # Apply a different function when x > 0.5
    mask = (x > 0.5)
    y[mask] = 2.0 * alpha * beta * x[mask]**alpha
    return x, y
```

Key Code: Data Loading and DNN

```python
# Data Loading System
class KinkedDataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

def create_dataloader(x, y, batch_size=256, shuffle=True):
    dataset = KinkedDataset(x, y)
    return DataLoader(dataset,
                      batch_size=batch_size,
                      shuffle=shuffle)
```

Key Code: Data Loading and DNN

```python
# Neural Network Definition
class KinkNet(nn.Module):
    def __init__(self, input_dim, hidden_dim1,
                 hidden_dim2, output_dim=1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim1),
            nn.BatchNorm1d(hidden_dim1),
            nn.ReLU(),
            nn.Linear(hidden_dim1, hidden_dim2),
            nn.BatchNorm1d(hidden_dim2),
            nn.ReLU(),
            nn.Linear(hidden_dim2, output_dim)
        )

    def forward(self, x):
        return self.net(x)
```

Key Code: Polynomial Approximation

```python
def polynomial_approx(x, y, degree):
    x_np = x.cpu().numpy().flatten()
    y_np = y.cpu().numpy().flatten()
    coeffs = np.polyfit(x_np, y_np, degree)
    poly_func = np.poly1d(coeffs)
    y_pred = poly_func(x_np)
    return coeffs, y_pred
```

Key Code: Chebyshev Approximation

```python
def chebyshev_basis(x, degree):
    x_scaled = 2 * (x - x.min()) / (x.max() - x.min()) - 1
    basis = torch.zeros((x.shape[0], degree + 1),
                        device=x.device)
    basis[:, 0] = 1.0
    if degree >= 1:
        basis[:, 1] = x_scaled.squeeze()
    for n in range(2, degree + 1):
        basis[:, n] = (2 * x_scaled.squeeze()
                       * basis[:, n - 1]
                       - basis[:, n - 2])
    return basis

def chebyshev_approx(x, y, degree):
    X = chebyshev_basis(x, degree)
    XtX = X.T @ X
    Xty = X.T @ y
    coeffs = torch.linalg.lstsq(XtX, Xty).solution
    y_pred = X @ coeffs
    return coeffs, y_pred
```

Training and Evaluation Workflow

```
def train_model(model, dataloader, config):
    criterion = nn.MSELoss()
    optimizer = optim.AdamW(model.parameters(),
                            lr=config["lr"],
                            weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, 'min', patience=50
    )
    losses = []
    model.train()
```

Training and Evaluation Workflow (cont.)

```
    with tqdm(total=config["epochs"],
              desc="Training") as pbar:
        for epoch in range(config["epochs"]):
            epoch_loss = 0.0
            for batch_x, batch_y in dataloader:
                optimizer.zero_grad()
                outputs = model(batch_x)
                loss = criterion(outputs, batch_y)
                loss.backward()
                optimizer.step()
                epoch_loss += loss.item()
            epoch_loss /= len(dataloader)
            losses.append(epoch_loss)
            scheduler.step(epoch_loss)
            pbar.set_postfix(loss=f"{epoch_loss:.6f}")
            pbar.update(1)
            if epoch_loss < config["tol"]:
                print(f"Early stopping at epoch {epoch}")
                break
    return losses
```

Example: Tiny MLP for GDP Forecasting

Example: Predicting GDP Growth

- **Objective:** Predict next-quarter real GDP growth (g_t) based on the trajectory of the previous year.
- **Econometric Analogy:** A Nonlinear Autoregressive model (AR(4)).
- **Input Vector ($\mathbf{x}_t$):**
$$\mathbf{x}_t = \left(g_{t-1}, g_{t-2}, g_{t-3}, g_{t-4}\right) \in \mathbb{R}^4$$
- **Target Output (y_t):**
$$y_t = g_t \in \mathbb{R}$$
- **Why use a Neural Network?**
  - Standard AR models assume linear persistence.
  - GDP dynamics are often *state-dependent* (e.g., shocks have different persistence in recessions vs. expansions).
  - The MLP allows the coefficients to vary non-linearly with the lag structure.

The Architecture: A "Tiny" MLP

The Network Topology:

$$\text{Input (4)} \xrightarrow{\text{ReLU}} \text{Hidden (16)} \xrightarrow{\text{ReLU}} \text{Hidden (8)} \xrightarrow{\text{Linear}} \text{Output (1)}$$

- **Input Layer:** Dimension 4 (The 4 lags).
- **Hidden Layers:**
  - Layer 1: Expands features to 16 dimensions.
  - Layer 2: Compresses features to 8 dimensions.
  - **Activation:** ReLU ($f(x) = \max(0, x)$) introduces the necessary non-linearity.
- **Output Layer:** Linear transformation to a scalar prediction.
- **Loss Function:** Mean Squared Error (MSE), equivalent to maximizing the likelihood under Gaussian errors.

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{t} (\hat{g}_t - g_t)^2$$

PyTorch Implementation: Defining the Model

```python
import torch
import torch.nn as nn

class GDPNet(nn.Module):
    def __init__(self):
        super().__init__()
        # nn.Sequential stacks layers in order
        self.net = nn.Sequential(
            nn.Linear(4, 16),   # Input: 4 lags -> 16 hidden
            nn.ReLU(),          # Non-linear activation
            nn.Linear(16, 8),   # 16 hidden -> 8 hidden
            nn.ReLU(),
            nn.Linear(8, 1)     # 8 hidden -> 1 scalar output
        )

    def forward(self, x):
        # x shape: [batch_size, 4]
        return self.net(x)      # returns shape: [batch_size, 1]
```

PyTorch Implementation: The Training Loop

The training loop follows the standard optimization "waltz":

```
import torch.optim as optim

model = GDPNet()
criterion = nn.MSELoss()  # Loss function
# Adam optimizer often converges faster than SGD
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Assume X_batch is (N, 4) and y_batch is (N, 1)
for epoch in range(1000):
    optimizer.zero_grad()          # 1. Reset gradients

    y_hat = model(X_batch)         # 2. Forward pass
    loss = criterion(y_hat, y_batch) # 3. Compute loss

    loss.backward()                # 4. Backpropagation
    optimizer.step()               # 5. Update weights
```