MSML610: Advanced Machine Learning

Deep Learning

Instructor: Dr. GP Saggese - gsaggese@umd.edu

References:

- ***Neural networks***
  - Biological inspiration
  - Neural networks
- Advanced Neural Network Architectures

Deep learning

- Deep learning is a family of ML models and techniques with complex expressions and tunable connection strengths

$$\text{Deep Learning} \subseteq \text{Machine Learning}$$

  - "Deep" as circuits are organized in layers with many connection paths between inputs and outputs
  - Represent hypotheses as computation graphs with learnable weights
  - Fit the model to the training data by computing the gradient of the loss function with respect to those weights
- Deep learning is extremely effective for:
  - Machine translation
  - Image recognition/synthesis
  - Speech recognition/synthesis

DL vs ML

- Many ML methods can handle a large number of input variables
  - **But**
    - The path from input to output is very short (e.g., multiply and sum)
    - There are no variable interactions
- E.g., decision trees
  - Allow long computation paths
  - Only a small fraction of variables can interact
- The expressive power of such models is very limited
  - Real-world concepts are far more complex

- Neural networks
    - ***Biological inspiration***
    - Neural networks
- Advanced Neural Network Architectures

Biological Inspiration for Neural Networks

- To perform a function like in a biological system, replicate its structure
- There is a leap of faith: the *structure* matters to achieve a *functionality*
- **Examples**
  - You want to fly, birds fly, birds have wings $\Longrightarrow$ build a contraption with wings
  - You want to learn, the brain learns, the brain has many neurons and synapses $\Longrightarrow$ build a system with many simple units connected together

Neurons in the brain

- The brain is jam-packed with neurons
- Each neuron has inputs (dendrites) and an output (axon)
- Neurons connect their output to inputs of different neurons, sending pulse of electricity
- Senses (e.g., eyes) send pulses to the neurons
- Neurons send pulses to the muscles to make them contract

Neural Network

- Get inspiration from networks of neurons in the brain
  - Human brain consist of layers of interconnected nodes
- Resemblance with neural structures is superficial
  - Architecture is inspired by the brain but does not replicate its complexity
  - Each connection has a weight that adjusts as learning proceeds
  - Neural networks simplify the brain's processes to make them computationally feasible
- Deep learning encompasses a broader range of models and algorithms beyond neural networks

$$\text{Neural Networks} \subseteq \text{Deep Learning}$$

  - Neural networks are building blocks for many deep learning models
  - E.g., a convolutional neural network (CNN) is used for image classification tasks
  - E.g., recurrent neural networks (RNNs) are used for sequence prediction tasks like language modeling

The “one learning algorithm” theory

- The brain can perform various tasks
  - E.g., process vision, sense of touch, do math, play pickleball
  - Doesn’t have thousands of different programs
  - Seems to have a single learning algorithm
- The “one learning algorithm” idea has been experimentally verified
  - Re-route the connection from eyes to the brain’s sound-processing area
  - After training, the brain can “see,” e.g., visual discrimination
- The **AI dream**:
  - If you can implement a (simplified) version of the brain algorithm, you can have a machine that can learn anything

Why resurgence of neural networks?

- AI winter
  - Proposed in the 1950
  - Popular in '80s and '90s
  - Then fell out of fashion due to limitations
- **Key Reasons for Resurgence**
  - **Increased Computational Power**
    - GPUs and TPUs suited for matrix operations in neural nets
  - **Availability of Big Data**
    - Open-source frameworks (e.g., TensorFlow and PyTorch)
    - Massive amounts of labeled data (Internet, IoT, and digital storage)
  - **Algorithmic Improvements**
    - Better activation functions (ReLU)
    - Advanced optimization techniques (Adam, RMSprop)
    - Regularization methods (dropout, batch normalization)
  - **Breakthrough Architectures**
    - CNNs for image tasks
    - RNNs and LSTMs for sequences
    - Transformers for language and vision
  - **Demonstrated Success in Applications**
    - State-of-the-art performance in vision, speech, language, and games
    - Commercial impact in healthcare, finance, and autonomous vehicles

NN vs logistic regression + non-linear transform

- Logistic regression with non-linear transformations might seem sufficient for any problem
  - However, the number of features increases rapidly
  - Neural networks synthesize their own features, offering an advantage
- E.g., in computer vision for 50×50 256-color images
  - There are 7500 bytes available as features
  - Using all cubic terms for a non-linear model requires $\approx 7500^3$ features $\propto (10^4)^3 = 10^{12} = 1$ trillion features
  - Which ones are really needed?
  - The features are predetermined and not learned
- Each neuron in a neural network performs logistic regression
  - Features used are computed by other neurons

- Shallow model has short computation path
- A decision tree has some long paths
- A deep learning network has long paths with many variables interacting

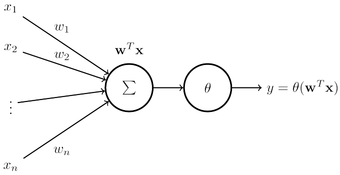
- Neural networks
    - Biological inspiration
    - ***Neural networks***
- Advanced Neural Network Architectures

Neural Network Perceptron

- Aka "artificial neuron", "logistic unit"
- A perceptron has n inputs and 1 output (like a brain neuron)
- The inputs are combined using a non-linear activation function $\theta(s)$ to implement:

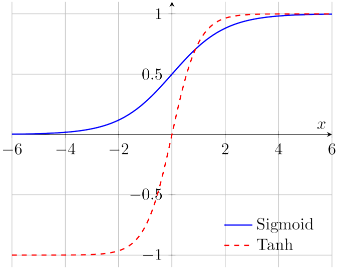
$$y = h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}) = \theta(\underline{\boldsymbol{w}}^T \underline{\boldsymbol{x}})$$

  - Same functional form as logistic regression (θ is logit) or linear classification (θ is sign)
- The parameters $\underline{\boldsymbol{w}}$ are typically called **weights** in neural network literature



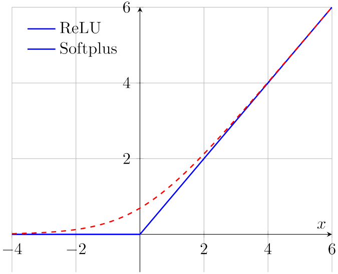
Activation Functions

- **Sigmoid Function**:
  - $\theta(s) = \frac{1}{1+\exp(-s)}$
  - Output range: $[0, 1]$
  - Smooth, differentiable, and used for probabilistic outputs
  - Saturates at extremes; suffers from vanishing gradient
- **Hyperbolic Tangent (tanh)**:
  - $\theta(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$
  - Output range: $[-1, +1]$
  - Zero-centered; improves convergence over sigmoid



Activation Functions

- Map the neuron's input signal s to an output activation value $\theta(s)$
  - Approximately linear for small s
  - Saturate or threshold for large $|s|$
    - Non-linearity enable learning complex functions
  - Many functions cross the origin or have $\theta(0) = 0$
- **ReLU (Rectified Linear Unit)**:
  - $\theta(s) = \max(0, s)$
  - Output range: $[0, \infty)$
  - Non-differentiable at $s = 0$; can cause dead neurons
- **Softplus Function**:
  - $\theta(s) = \log(1 + \exp(s))$
  - Smooth approximation of ReLU
  - Always differentiable; output in $[0, \infty)$



Neural networks to compute boolean functions

- Build AND, OR, NOT functions with linear perceptrons using proper bias and weights
- AND is implemented by a single neuron with 2 inputs x_1 and x_2 and bias
  - Implement in terms of s:

```
x_1   x_2   AND
 -1    -1    -1
 -1    +1    -1
 +1    -1    -1
 +1    +1    +1
```

  - Plot a diagram and find a proper decision boundary:

$$y = \theta(w_0 * 1 + w_1 * x_1 + w_2 * x_2) = \theta(-3 + 2w_1 + 2w_2)$$

Weights are $(w_0, w_1, w_2) = (-3, 2, 2)$

- You cannot classify 4 points in a XOR position in a plane with a *single* perceptron
  - Use two perceptrons to separate two regions of space
  - Then combine the outputs of the 2 perceptrons together

Universal Approximation in Feedforward Networks

- **Power of Composition**
  - Connecting perceptrons enables complex functions
  - A network of sufficient size and depth can approximate any Boolean or continuous function to arbitrary precision
  - Compositional structure: each layer builds on the previous one
- **Role of Nonlinearity**
  - Nonlinear activation functions (e.g., sigmoid, tanh, ReLU) are essential
  - Without nonlinearity, layers reduce to a single linear transformation
  - Nonlinearity allows modeling of complex, non-linear decision boundaries
- **Geometric Intuition**
  - To separate two classes with a circular boundary:
    - Use multiple perceptrons (e.g., 8–16) to approximate the circle with a polygon
    - Combine outputs logically for the final decision
  - This forms a “lookup table” over regions of input space—similar to decision trees

Feedforward vs Recurrent Neural Networks

- **Feedforward Neural Networks**
  - Information flows in one direction from inputs to outputs without cycles in the computational graph
  - Can model static relationships between inputs and outputs
    - E.g., classifying a handwritten digit from an image
  - Limited in handling sequential dependencies (only fixed window of inputs)
- **Recurrent Neural Networks (RNNs)**

$$z_t = f_w(z_{t-1}, x_t)$$

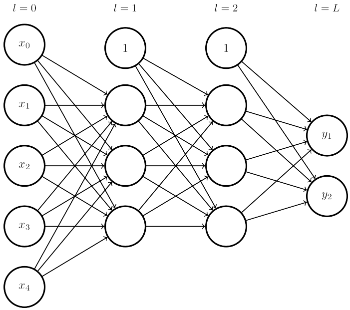
  - Allow cycles in the computational graph with delays
    - Each unit can take inputs from its previous output: adds memory
    - Process sequences: outputs depend on current and previous inputs
  - Suitable for sequential data (e.g., time series, language modeling) and model longer-range dependencies
  - E.g., predicting the next word in a sentence based on previous words

Structure of feedforward neural network

- **Layered Architecture**
  - A feedforward network consists of an ordered set of layers indexed by l
    1. Input layer ($l = 0$)
    2. One or more hidden layers ($0 < l < L$)
    3. Output layer ($l = L$)
  - Each layer l contains $d(l)$ units or neurons, and can vary in size
- **Input Layer ($l = 0$)**
  - Represents the input vector ($x_0 = 1, x_1, x_2, ..., x_d$)
  - $x_0 = 1$ acts as the bias input
  - Booleans are mapped to 0/1 or -1/+1
  - Numeric attributes are left unchanged (scaled to fit a fixed range, logged)
  - Categorical attributes can be one-hot encoded

Structure of feedforward neural network

- **Hidden Layers ($0 < l < L$)**
  - Each neuron computes a weighted sum of inputs from the previous layer
  - Applies a nonlinear activation function θ
  - Includes a bias unit with constant output 1
  - Fully connected: every node in layer $l-1$ connects to every node in layer l
- **Output Layer ($l = L$)**
  - Final layer producing the output vector $\mathbf{y}$
  - Output can be:
    - Linear: for regression tasks
    - Nonlinear (e.g., softmax, sigmoid): for classification tasks
  - Choice of activation depends on the task type and desired output range



Conventions for neurons in a neural network

- Each neuron $x_j^{(l)}$:
  - Belongs to a layer with index l
  - Accepts inputs from the previous layer (scanning index i)
  - Has an index j in the layer for its output
- Weights are identified by 3 indices $w_{ij}^{(l)}$ where:
  - ij are organized as input-output
  - $0 \leq l \leq L$ denotes the layer
  - $0 \leq j \leq d(l)$ denotes the output of the layer (i.e., the neuron in the layer)
  - $0 \leq i \leq d(l-1)$ denotes the inputs: start from 0 to account for the bias
    - Use $l-1$ in $d(l-1)$ since you look at the previous layer

Feedforward propagation algorithm

- The output of the generic l-th layer is $\underline{\boldsymbol{x}}^{(l)}$
- The outputs of the input layer ($l = 0$) are the inputs of the network:

$$\underline{\boldsymbol{x}}^{(0)} = (x_0^{(0)}, x_1^{(0)}, ..., x_{d(0)}^{(0)}) = \underline{\boldsymbol{x}} = (1, x_1, ..., x_d)$$

- The output of the j-th neuron of the l-th layer is $x_j^{(l)}$
  - Has $d(l-1)$ inputs from the previous layer combined with the weights
  - Computes the signal $s_j^{(l)}$
  - The activation function θ is applied:

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta(\sum_{i=0}^{d(l-1)} w_{ij}^{(l)} x_i^{(l-1)})$$

- The output of the network is the output of the neurons in the last layer

$$y = h(\underline{\boldsymbol{x}}) = x_1^{(L)}$$

  - Outputs are $s_1^{(L)}$ for regression or $\theta(s_1^{(L)})$ for classification

Vectorized feedforward propagation algorithm

- Vectorize neuron evaluation:
  - The j-th neuron of the l-th layer uses the $d(l-1)$ outputs of the previous layer to compute its output:

$$x_j^{(l)} = \theta(\sum_i w_{ij}^{(l)} x_i^{(l-1)}) = \theta((\underline{\boldsymbol{w}}_j^{(l)})^T \underline{\boldsymbol{x}}^{(l-1)})$$

- Compute all inputs $\underline{\boldsymbol{s}}^{(l)}$ to the activation function as matrix-vector product:

$$\underline{\boldsymbol{s}}^{(l)} = \underline{\underline{\boldsymbol{W}}}^{(l)} \cdot \underline{\boldsymbol{x}}^{(l-1)}$$

  where $\underline{\underline{\boldsymbol{W}}}^{(l)}$ is a matrix with weight vectors for each neuron in layer l as rows
  - Include the bias by adding a column to weight matrix $\underline{\underline{\boldsymbol{W}}}$ and padding inputs $\underline{\boldsymbol{x}}^{(l)}$ with 1s
- Apply the activation function in vectorized form:

$$\underline{\boldsymbol{x}}^{(l)} = \underline{\boldsymbol{\theta}}(\underline{\boldsymbol{s}}^{(l)}) = \underline{\boldsymbol{\theta}}(\underline{\underline{\boldsymbol{W}}}^{(l)} \cdot \underline{\boldsymbol{x}}^{(l-1)})$$

Cost Function for Single-Class NN Classification

- For binary classification using neural networks, use logistic regression cost function with a regularization term:

$$E_{in}(\underline{\boldsymbol{w}}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log h(\underline{\boldsymbol{x}}_i) + (1-y_i)\log(1-h(\underline{\boldsymbol{x}}_i))) + \frac{\lambda}{N}\sum_{j=1}^{p}\|\underline{\boldsymbol{w}}_j\|^2$$

- By convention, we don't regularize the bias ($j = 1$ and not $j = 0$), as it is constant and does not affect the minimum $\underline{\boldsymbol{w}}$

Multi-output neural networks for multi-class classification

- **In one-vs-all approach:**
  - Train n models, one per class, to recognize each class
  - Pick the model with the highest probability
  - The output is a one-hot encoding of each class
  - E.g.,
    - Use 4 output neurons to discriminate pedestrian, car, motorcycle, truck
    - Encode pedestrian = `(1, 0, 0, 0)`
    - Encode car = `(0, 1, 0, 0)`
    -
- Instead of training n neural networks, train a single neural network with an output layer of n nodes
  - Global optimization vs n local optimizations
  - End-to-end learning

Cost Function for Multi-Class NN Classification

- The loss function for multi-class classification using neural networks
  - encode one-hot
  - the expected outputs $\underline{\boldsymbol{y}}_i$
  - the outputs from the model $\underline{\boldsymbol{h}}(\underline{\boldsymbol{x}}_i)$

$$E_{in}(\underline{\boldsymbol{w}}) = -\frac{1}{N}\sum_i \sum_k \underline{\boldsymbol{y}}_i|_k \log \underline{\boldsymbol{h}}(\underline{\boldsymbol{x}}_i)|_k + (1 - \underline{\boldsymbol{y}}_i|_k) \log(1 - \underline{\boldsymbol{h}}(\underline{\boldsymbol{x}}_i)|_k) +$$

$$\frac{\lambda}{N}\sum_{l=1}^{L}\sum_{j=1}^{d(l)}\sum_{i=1}^{d(l-1)} (w_{ij}^{(l)})^2$$

- Avoid to consider the inputs ($l \neq 0$) and the bias terms ($i \neq 0, j \neq 0$)

Issues with fitting a neural networks

- **Generalization**
  - Match model complexity to data resources
  - High flexibility *to* lots of data is needed
  - Overly complex models can overfit if data is insufficient
  - E.g., a deep neural network with millions of parameters requires a large dataset to train effectively without overfitting
- **Optimization**
  - Several layers of perceptrons with a hard threshold turn the optimization problem into a combinatorial one
  - E.g., challenges in finding the global minimum due to the non-convex nature of the problem

Fitting a neural networks for SGD

- Use Stochastic Gradient Descent (SGD) to determine the weights $\underline{\boldsymbol{w}}$
  - Consider the error on a single example $(\underline{\boldsymbol{x}}, y)$:

$$E_{in}(\underline{\boldsymbol{w}}) = e(h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}), y) = e(\underline{\boldsymbol{w}})$$

  - Same process for both regression and classification:

$$e(h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}), y) = (h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}) - y)^2$$
$$e(h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}), y) = -y \log h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}) - (1-y)\log(1 - h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}))$$

  - Need to compute $\nabla_{\underline{\boldsymbol{w}}} e(\underline{\boldsymbol{w}}_0)$ by computing all the partial derivatives

$$\frac{\partial e(\underline{\boldsymbol{w}})}{\partial w_{ij}^{(l)}} \ \forall i, j, l$$

- The entire formula for the hypothesis $h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}})$ is very convoluted:
  - Non-linearity θ of ...
  - Linear combinations of weights and θ of ...
  - Linear combinations of weights and θ of ...

$$\begin{aligned} h_{\underline{\boldsymbol{w}}}(\underline{\boldsymbol{x}}) &= \theta((\underline{\boldsymbol{w}}^{(L)})^T \cdot \underline{\boldsymbol{x}}^{(L-1)}) \\ &= \theta((\underline{\boldsymbol{w}}^{(L)})^T \cdot \underline{\boldsymbol{\theta}}(\underline{\underline{\boldsymbol{W}}}^{(L-1)} \cdot \underline{\boldsymbol{x}}^{(L-2)}) \\ &= \dots \end{aligned}$$

Computing the gradient

- Compute all partial derivatives to get:

$$\nabla_{\underline{\boldsymbol{w}}} e(\underline{\boldsymbol{w}})$$

- **You can:**
  - Compute the analytic expression of the derivatives by brute force
  - Approximate the derivatives numerically by changing each $w_{ij}^{(l)}$ and computing the variation of e
  - Use a very efficient algorithm (backpropagation) to compute the gradient
- **Backpropagation** (or "backprop")
  - Efficient algorithm for computing gradients of the loss function with respect to all weights in the network
  - Enables training of multi-layer neural networks via gradient descent
  - Based on the chain rule of calculus
  - Updates weights to minimize the overall prediction error
- **Intuition**
  - Forward pass computes predictions
  - Backward pass computes how much each weight contributed to the error
  - Adjust weights to reduce future errors

Backpropagation in Neural Networks

- Initialize weights $\underline{\boldsymbol{w}}$ randomly
  - Avoid $\underline{\mathbf{0}}$ as it is an unstable equilibrium point
- For each iteration:
  - Pick a random input $\underline{\boldsymbol{x}}_n$ (SGD setup)
  - Forward pass: compute outputs of all neurons $x_j^{(l)}$ given $\underline{\boldsymbol{x}}_n$ and current weights $\underline{\boldsymbol{w}}(t)$
  - Backpropagation: compute all $\delta_j^{(l)}$ using backpropagation for current $\underline{\boldsymbol{x}}_n$ and $\underline{\boldsymbol{w}}(t)$
  - Compute derivatives of errors: $\frac{\partial e}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$
  - Update weights using derivatives

$$\underline{\boldsymbol{w}}(t+1) \leftarrow \underline{\boldsymbol{w}}(t) - \eta \nabla_{\underline{\boldsymbol{w}}} e(\underline{\boldsymbol{w}}(t))$$

$$w_{ij}^{(l)}(t+1) \leftarrow w_{ij}^{(l)}(t) - \eta \frac{\partial e}{\partial w_{ij}^{(l)}}$$

- Iterate until termination
- Note that the cost function is not convex
  - No guarantee of finding the global minimum

Backpropagation in Neural Networks

- If we want to use batch gradient descent (instead of SGD)
  - Accumulate the partial derivatives considering all the examples in the training set
  - Update the weights with the accumulated values

Gradient checking

- For some algorithms the analytical expression of the gradient becomes complicated: mistakes are possible
  - E.g., back-propagation in neural networks
- One approach is:
  - Compute the gradient analytically
  - Compute the gradient numerically

$$\frac{\partial E_{in}(\underline{\boldsymbol{w}})}{\partial w_j} \approx \frac{E_{in}(\underline{\boldsymbol{w}} - \hat{w}_j \varepsilon) - E_{in}(\underline{\boldsymbol{w}} + \hat{w}_j \varepsilon)}{2\varepsilon}$$

  - Compare the gradients within numerical approximation
  - Pick ε small (e.g., $\varepsilon = 10^{-4}$) but not so small to cause too many numerical issues
- Automatic differentiation packages solve this issue

Automatic Differentiation

- **Automatic Differentiation**
  - Computes gradients using calculus rules on numerical programs
  - Applies the chain rule efficiently from output to input
  - Avoids manual gradient derivation for new architectures
- **Practical Benefits**
  - Major deep learning frameworks (E.g., tensorFlow, PyTorch) implement automatic differentiation
  - Enables rapid experimentation with network structures, activation functions, and loss functions
  - Frees you from manually re-deriving learning rules
- **Encouragement of End-to-End Learning**
  - Complex tasks (E.g., machine translation) modeled as compositions of trainable subsystems
  - Trained on input-output pairs without explicit internal supervision
  - Requires minimal prior knowledge about internal components or roles

- Neural networks
- ***Advanced Neural Network Architectures***
    - Convolutional Neural Networks
    - Recurrent Neural Networks
    - Deep learning learning algorithms
    - Deep learning for NLP

- Neural networks
- Advanced Neural Network Architectures
    - ***Convolutional Neural Networks***
    - Recurrent Neural Networks
    - Deep learning learning algorithms
    - Deep learning for NLP

Convolutional Neural Networks

- **Motivation**
  - Feedforward networks do not scale with high-dimensional inputs like images
  - Convolutional neural networks (CNNs) are designed to exploit spatial structure in data
- **Key Idea**
  - Local connectivity: adjacent pixels have meaning
  - Weight sharing to detect spatially local patterns
  - Convolutions act as learnable filters applied across input regions
- **Basic Components**
  - Convolutional layer: applies multiple filters across input
  - Activation function: non-linearity (e.g., ReLU) after convolution
  - Pooling layer: reduces spatial dimensions (e.g., max pooling)
  - Fully connected layers: typically at the end for classification

Convolutional Neural Networks

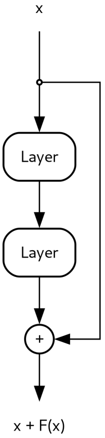
- **Convolutional Layer Mechanics**
  - Filter (or kernel): small matrix of weights (E.g., 3×3)
  - Apply dot product between filter and local patch of input
- **Weight Sharing**
  - Each filter is reused across all spatial locations
  - Spatial invariance: things detectable in an image should be independent of the position in the image
  - Reduces number of parameters and improves generalization
- **Feature map**
  - Output of a convolutional layer after applying filters (kernels) to the input
  - Represents spatial patterns such as edges, textures, or more complex features learned by the network
- **Stacking Convolutions**
  - Multiple layers can detect increasingly abstract features:
    - Early layers detect edges, textures
    - Later layers detect object parts or entire objects
- **Example**
  - An image of size $32 \times 32 \times 3$ with a 5×5 filter creates a 28×28 feature map (ignoring padding)

Convolutional Neural Networks

- **Pros**
  - Parameter efficiency due to local connectivity and weight sharing
  - Invariant to translation and small distortions in input
  - Scalable to large input sizes (E.g., high-resolution images)
- Pooling is a downsampling operation that reduces the spatial dimensions (width and height) of feature maps
  - Reducing computation
  - Controlling overfitting
  - Making features more translation-invariant
  - Common pooling: max pooling, average pooling
- **Common Architectures**
  - LeNet, AlexNet, VGG, ResNet
- **Applications**
  - Vision tasks
    - Image classification
    - Object detection
    - Face recognition
    - Medical imaging
  - NLP
  - Audio processing

Residual Networks (ResNets)

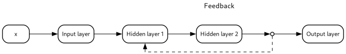
- **Motivation for Deep Networks**
  - Deeper networks perform better on complex tasks
  - Very deep networks suffer from vanishing/exploding gradient problems
- **Key Idea: Residual Learning**
  - Learn a residual function $F(x) = H(x) - x$ instead of a direct mapping $H(x)$
  - Original function becomes $H(x) = F(x) + x$
  - Easier to learn a small change from a function than a new function
- **Residual Block Structure**
  - Each block computes $F(x)$ and outputs $F(x) + x$
  - x is the *shortcut connection* or *skip connection*
- **Pros**
  - Enable training of deeper networks (e.g., hundreds of layers)
  - Help gradients flow during backpropagation
- **Empirical Success**
  - ResNets outperform plain networks of the same depth
  - Achieved state of the art results on ImageNet



- Neural networks
- Advanced Neural Network Architectures
    - Convolutional Neural Networks
    - ***Recurrent Neural Networks***
    - Deep learning learning algorithms
    - Deep learning for NLP

Recurrent Neural Networks (RNNs)

- Maintain a hidden state that evolves over time
  - Process input sequences one element at a time, maintaining a memory of previous inputs
  - Output depends on current input and previous hidden state
  - Useful for tasks where the order of data is crucial
- Trained using backpropagation through time
- Pros
  - Designed for sequential data
    - E.g., time series, text
  - Can, in theory, capture temporal dependencies of arbitrary length
- Cons
  - Struggle with long-term dependencies (vanishing gradients)
  - Training is slow due to sequential processing (no parallelism over time steps)



Vanishing and Exploding Gradient Problem

- Arise due to repeated multiplication of gradients in the chain rule
- **Vanishing Gradient**
  - Gradients become very small in early layers as they are propagated backward
  - Leads to extremely slow learning or no learning
  - Common with activation functions like sigmoid or tanh
- **Exploding Gradient**
  - Gradients grow exponentially during backpropagation
  - Causes unstable updates and potential overflow in weights
  - Often seen when weights are initialized with large values
- **Solutions**
  - Use ReLU or similar activations to mitigate vanishing gradients
  - Apply gradient clipping to handle exploding gradients
  - Normalize inputs
  - Apply batch normalization
- These issues motivated the design of architectures like LSTM and ResNet

Long Short-Term Memory (LSTM) Networks

- Special architecture with memory cells and gates
  - Forget gate: decides what information to discard from the cell state
  - Input gate: controls what new information to store
  - Output gate: determines what to output from the cell
- **Pros**
  - Designed to overcome vanishing gradient problem in standard RNNs
  - Captures long-range dependencies through gated memory cells
  - Flexible architecture for learning when to remember or forget
- **Cons**
  - More complex than RNNs: includes multiple gates and a memory cell
  - Higher computational cost due to increased number of parameters
  - Slower training and inference compared to simpler models (e.g., GRUs)
  - Difficult to parallelize over time steps
- Especially effective in natural language processing and time series tasks
- E.g., used in machine translation, speech recognition, and language modeling

Gated Recurrent Units (GRUs)

- Alternative to LSTMs for sequential modeling
  - Combine **forget** and **input** gates into a single **update gate**
  - Retain essential functionality with fewer components:
    - Update gate controls how much of the past state to keep
    - Reset gate controls how much of the past information to forget
- **Pros**
  - Requires fewer parameters, leading to faster training
  - Comparable or better performance than LSTM on many tasks
  - Easier to tune and implement
  - Suitable for real-time and low-resource applications
- **Cons**
  - May be less expressive than LSTM for very complex patterns
  - Lacks a separate memory cell, which may limit ability to retain long-term information
  - Slightly less studied and standardized than LSTM in some domains
- **Example**
  - GRUs are preferred in real-time speech recognition systems due to their efficiency

- Neural networks
- Advanced Neural Network Architectures
    - Convolutional Neural Networks
    - Recurrent Neural Networks
    - ***Deep learning learning algorithms***
    - Deep learning for NLP

Techniques for Training Deep Neural Networks

- Deep networks are hard to train
  - Due to vanishing/exploding gradients
  - Optimization can be slow and sensitive to hyperparameters
- **Momentum**
  - Modifies gradient descent to accelerate convergence
  - Updates use an exponentially decaying average of past gradients
  - Update rule: $v_t = \beta v_{t-1} + \eta \nabla \theta_t$, $\theta_{t+1} = \theta_t - v_t$
  - Helps overcome local minima and reduces oscillations in ravines
- **Adam (Adaptive Moment Estimation)**
  - Combines momentum with adaptive learning rates
  - Maintains estimates of both first moment (mean) and second moment (uncentered variance) of gradients
  - Update rule uses bias-corrected estimates:
    - $m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla \theta_t$
    - $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla \theta_t)^2$
    - $\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$
  - Works well in practice with little tuning

Techniques for Training Deep Neural Networks

- **Batch Normalization**
  - Normalizes the input of each layer to have zero mean and unit variance
  - Computed over each mini-batch:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

  - Introduces learnable parameters γ and β to scale and shift the normalized values
  - Benefits:
    - Reduces internal covariate shift
    - Enables higher learning rates
    - Acts as a regularizer, reducing need for dropout

Regularization in neural networks

- (Soft) weight elimination: fewer weights $\implies$ smaller VC dimension, so we would like to remove some neurons (i.e., push weights towards 0)
- For any activation function (e.g., tanh()) a small weight means that we work in the linear regime, while a large weight leaves us in the binary regime
- Using a normal regularization

$$\Omega(\underline{\boldsymbol{w}}) = \sum_{i,j,l} (w_{ij}^{(l)})^2$$

  we have the problem that a neuron in binary regime is penalized more than many neurons in linear regime
- So for neural networks we use a regularizer as:

$$\Omega(\underline{\boldsymbol{w}}) = \lambda \sum_{i,j,l} \frac{(w_{ij}^{(l)})^2}{\beta^2 + (w_{ij}^{(l)})^2}$$

  so that the penalization is quadratic for small $\underline{\boldsymbol{w}}$ and then it saturates as function of the weight magnitude

- Neural networks
- Advanced Neural Network Architectures
  - Convolutional Neural Networks
  - Recurrent Neural Networks
  - Deep learning learning algorithms
  - ***Deep learning for NLP***

NLP Tasks

- **Part-of-speech (POS) tagging**
  - Assign a POS (e.g., noun, verb, adjective) to each word in a sentence
  - Data needs: labelled data
- **Co-reference resolution**
  - E.g., *"Mike told that John was sick, so I took him to the hospital"*
    - Who does *"him"* refer to?
    - *"John"* with high probability
  - The output is a distribution over possible antecedents
  - Data needs: labelled data
- **Sentiment analysis**
  - Classify a text as having positive or negative sentiment
    - E.g., "the movie was poorly written and acted" $\rightarrow$ negative
  - Data needs: nothing in theory
- **Machine translation**
  - Translate a sentence from a source language to a target language
    - E.g., from Spanish to English
  - Data needs: large corpus of source/target sentence pairs
- **Information extraction**
  - Automatically extracting structured information from unstructured text
- **Named entity recognition**
  - Identify and classify named entities (e.g., people, organizations, locations) in text

Part-of-speech (POS) tagging

- **Example**
  - Input: "The dog barks loudly"
  - Output: [Determiner (DT), Noun (NN), Verb (VB), Adverb (RB)]
- **Techniques**
  - *Rule-based*: hand-crafted grammar rules
  - *Statistical*: models like Hidden Markov Models (HMMs)
  - *Machine Learning*: classifiers (e.g., SVM)
  - *Neural Networks*: LSTM, BiLSTM, or Transformers
- **Challenges**
  - Ambiguity: "can" can be a verb or a noun
  - Context-dependency: Tags depend on the surrounding words

NLP using Rules-Based Systems

- Systems based on rules for parsing and semantic analysis have shown:
  - Success in many tasks
  - Performance limited by linguistic complexity
  - E.g., “cut” can be a verb or a noun
    - Present, past, infinite verb
    - Transitive or intransitive
    - The meaning depends on the context
- Idea:
  - Vast amount of text in machine-readable form $\rightarrow$ use data-driven ML approach

The bitter lesson

- *"General methods leveraging computation ultimately outperform human-knowledge-based systems"*, Sutton (2019)
- Key Observation
  - Human-designed heuristics and domain knowledge lose
  - General-purpose learning algorithms that scale with computation tend to win
  - E.g.,
    - Chess: hand-crafted rules outperformed by AlphaZero using self-play and deep reinforcement learning
    - Vision: feature-engineered models replaced by end-to-end trained convolutional networks
- The "bitter" part
  - Humans like to inject knowledge into systems
  - But machines learn better when allowed to discover patterns from data themselves

The bitter lesson

- **Implications**
  - Invest in scalable methods over handcrafted knowledge
  - Embrace compute-heavy solutions that learn from data
  - Focus on architectures that can generalize across domains
  - Accept that performance gains come from scaling data and compute
- **Controversy**
  - Seen by some as dismissive of domain expertise and symbolic methods
  - Encourage a shift toward empirical and data-driven AI research

Word Representation

- **Goal**: create representation of words for NLP tasks
- **Requirements**:
  - No need for manual feature engineering
  - Allow for generalization between related words
    - Syntactically: “colorless” and “ideal” are both adjectives
    - Semantically: “cat” and “kitten” are both felines
    - Topically: “sunny” and “rainy” are both weather terms
    - Sentiment: “awesome” and “cringeworthy” are opposite
- **Approaches**:
  1. Encode a word into an input vector (e.g., one-hot vector)
     - Cons: doesn’t capture similarity between words
  2. Represent as a vector of n-gram counts
     - Phrases of *n* words containing each word
     - Very large number of vectors
     - E.g., 100,000-word vocabulary $\rightarrow 100,000^5 = 10^{25}$ vectors
     - Very sparse since most counts are zero
  3. Learn word embeddings
     - Low-dimensional vector representing a word, learned from the data
     - E.g., 100 dimensions

Word Embeddings: Emerging Properties

- Similar words have similar word embeddings
  - Words cluster based on their topics
- Difference between related words seems to have a meaning
  - E.g., `Greece` - `Athens` = country/capital relationship
  - E.g., negative, plural, superlative, past tense
- These properties are:
  - Not really enforced
  - Approximate
  - Emergent

Word Embeddings: Pretrained vs Custom

- Word embedding representations are independent of the task
  - Can pretrain and reuse them
    - Word2vec
    - GloVe (global vectors)
    - FastText
- For some tasks better to learn word embeddings end-to-end with the task
  - E.g., POS tagging: the same word needs to be represented in different ways
    - E.g., "cut" can be verb or noun $\rightarrow$ multiple semantic embeddings

Language Models

- Word embedding are a good representation for words in isolation
- Language consists of sequence of words where context of each word is important
    - E.g., in *"Michael told me that John was sick, so I took him to the hospital"*, "him" can refer to multiple people
- A **language model** is a probability distribution of sequences of words
    - Predict the next word in a text given all the previous words
        - Supervised learning
        - Enormous amount of data
- How to learn a language model?
    1. Feedforward neural network
    2. Recurrent neural network
    3. Transformer architecture

Feedforward network for language models

- **Learning set-up**:
  - Large feedforward network
  - Fixed-size context window of n words
  - Word embeddings
- **Problems:**
  1. Context might need to be very long
     - Not all the words are important
     - E.g., a sentence with 20 words requires a large number of parameters if each word is considered in any context
  2. Problem of asymmetry
     - Weights need to be learned for each word in each position
     - E.g., different weights for the word "cat" when it appears at the beginning versus the end of a sentence
  - Too many parameters to learn
  - Computationally expensive

RNNs for language models

- RNNs process sequence of data, one piece of data at a time
  - Process language, one word at a time
- Architecture
  1. Each input word is encoded as word embedding vector
  2. A hidden layer gets the previous state and the new word
     - Allocate storage space for features of inputs useful for the task
  3. The output is a probability over output words

RNNs for language models: pros and cons

- **Pros**:
  - Context too long:
    - The number of parameters is constant independently of the context (i.e., n-grams)
  - Problem of asymmetry:
    - No problem of asymmetry since the weights are the same for every word position
- **Cons**:
  - Can be difficult to train
    - Solution: Use transformers
  - Solve the context problem, but only in theory
    - In theory, information can be passed along through steps
    - In practice, it is lost or distorted (similar to vanishing gradient problem, but over time)
    - Solution: Use LSTMs

Sequence-to-Sequence Models

- Enable learning of mappings between sequences of differing lengths and structures
  - Used for transforming one sequence into another, e.g.,
    - Machine translation
    - Text summarization
    - Speech recognition
- Composed of two main components:
  - **Encoder**: Compress information from input sequence into a fixed-size context vector
  - **Decoder**: Generates output sequence from the context vector
  - Both encoder and decoder are typically RNNs (e.g., LSTM or GRU)
- Training is supervised with input-output sequence pairs
- Limitation:
  - Fixed-size context vector can become a bottleneck for long sequences
  - Attention mechanisms so decoder can access all encoder states

Sequence-to-Sequence Attention

- **Intuition**
  - Instead of processing input sequentially (like RNNs), attention processes all simultaneously
  - Each token can "look at" others and decide which are most important
    - E.g., in translation, the word "bank" in "river bank" gets higher attention from nearby words like "river"
  - It's like "context-based summarization" of the source sentence into a fixed-dimensional representation
- **Motivation**
  - Enables models to focus on relevant parts of the input sequence
  - Mimics human cognitive attention by weighting important information

Sequence-to-Sequence Attention

- If the RNN model is $\underline{\boldsymbol{h}}_i = RNN(\underline{\boldsymbol{h}}_{i-1}, \underline{\boldsymbol{x}}_i)$
- The sequence-to-sequence attention is the concatenation of $\underline{\boldsymbol{x}}_i$ and the context vector $\underline{\boldsymbol{c}}_i$

$$\underline{\boldsymbol{h}}_i = RNN(\underline{\boldsymbol{h}}_{i-1}, [\underline{\boldsymbol{x}}_i, \underline{\boldsymbol{c}}_i])$$

- The raw attention score r_{ij} is:

$$r_{ij} = \underline{\boldsymbol{h}}_{i-1} \cdot \underline{\boldsymbol{s}}_j$$

  where
  - $\underline{\boldsymbol{h}}_{i-1}$ is the current target state
  - $\underline{\boldsymbol{s}}_j$ is the j-th source word (i.e., the output of the source RNN vector for the word j)

- The attention scores are normalized into probability using a softmax:

$$a_{ij} = \frac{e^{r_{ij}}}{\sum_k e^{r_{ik}}}$$

- Finally the vectors $\underline{\boldsymbol{s}}_j$ are weighted and summed as:

$$\underline{\boldsymbol{c}}_i = \sum a_{ij} \cdot \underline{\boldsymbol{s}}_j$$

Types of Attention

- **Self-Attention**: Each input attends to all inputs, including itself
  - Useful for capturing dependencies within a single sequence
  - E.g., in language models, each word can attend to all other words in a sentence
- **Cross-Attention**: One sequence attends to another
  - Facilitates the interaction between different sequences
  - E.g., in machine translation, the target language sequence attends to the source language sequence
- **Multi-Head Attention**
  - Applies attention multiple times in parallel (with different linear projections)
  - The results are concatenated together to form $\underline{\boldsymbol{c}}_i$
  - Concatenating is better than summing to keep important information
  - Captures different types of relationships at different subspaces
  - E.g., in a Transformer model, different heads might focus on different parts of a sentence, such as subject-verb agreement or noun-adjective relationships

Attention: Vectorized Formula

- Compute a weighted sum of values (V) using weights derived from queries (Q) and keys (K)

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

  where:
  - Q: matrix of query vectors (shape: $n_q \times d_k$)
  - K: matrix of key vectors (shape: $n_k \times d_k$)
  - V: matrix of value vectors (shape: $n_k \times d_v$)
  - d_k: dimensionality of the key vectors
  - QK^T: dot products between each query and all keys (shape: $n_q \times n_k$)
- **Steps**:
  1. **Dot Product**: Compute similarity between each query and all keys
     - QK^T produces attention scores
  2. **Scaling**: Divide scores by $\sqrt{d_k}$
     - To stabilize gradients
  3. **Softmax**: Apply softmax to each row of the scaled scores
     - Converts scores into a probability distribution (attention weights)
  4. **Weighted Sum**: Multiply weights by V to produce context vectors
- **Advantages**
  - Parallelizable computation
  - Better handling of long-range dependencies than RNNs

Sequence-to-Sequence: Decoding

- Training a sequence-to-sequence model requires to:
  - Maximize the probability of each word in the target training sentence
  - Conditioned on the source and previous target words
- At inference time:
  - Given a source sentence
  - Generate the target word one at a time
  - Feed back the target word for the next time step
- Greedy decoding
  - Pick the next word that has the highest probability
  - Pros
    - Fast
  - Cons
    - We need to maximize the probability of the entire target sequence
    - Greedy decoding doesn't have a mechanism to correct a mistake
    - Many times the model needs to see what comes next
- Beam search:
  - Keep the top k hypotheses at each stage
  - Choose the hypothesis with the best score

Transformer Architecture

- Revolutionized sequence modeling through combining several ideas
- **Self-attention**
  - Eliminates recurrence by processing sequences in parallel
  - Model long-distance context without a sequential dependency
  - Multi-head attention to capture different aspects of relationships between tokens
- Transformer has many (6 or more) transformer layers
  - **Transformer layer**
    - Self-attention
    - Residual connection
    - Feedforward layer (with ReLU)
    - Residual connection
- **Positional embedding**: injects sequence order into token embeddings
  - Transformer architecture has no inherent way to capture token order in sequences
  - Add "positional embeddings" to input embeddings to provide information about the position of each token in the sequence
- **Pros**

Pretraining

- For computer vision train using large collections of hand-labeled images (e.g., ImageNet)
- For text use un-labeled data
  - NLP tasks are not easy for labellers (e.g., POS)
  - Internet has large amount of text (100b words added every day)
    - Common crawl
    - Wikipedia
    - FAQs can be used (for question-answering)
    - Websites have multiple version (for translation)
- Pretrain using a large amount of text data
  - Fine tune the model with labeled data
  - It's an example of transfer learning

Masked language models

- Word prediction in language models are made left-to-right
- Sometimes the context comes later in the context
  - E.g., “the river rose five feet”
- Train models using masking a word and predicting it
  - E.g., “the river ________ five feet”
- The sentence provides its own labels