MSML610: Advanced Machine Learning

Deep Learning

Instructor: GP Saggese, PhD - gsaggese@umd.edu

References:

Neural networks

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

DL vs ML

- Many ML methods can:
  - Handle a large number of input variables
  - 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

Biological inspiration

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

Neural network

- Model networks of neurons in the brain with computational circuits
  - Mimic how human brains process information
  - Consist of layers of interconnected nodes or "neurons"
  - Each connection has a weight that adjusts as learning proceeds
- Resemblance with neural cells and structures is superficial
  - Architecture is inspired by the brain but does not replicate its complexity
  - 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., convolutional neural networks)
  - 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

Biological inspiration for neural networks

- To perform a function like in a biological system, just replicate its structure
- There is a leap of faith: the *structure* matters to achieve a *functionality*
- E.g.,
  - Birds fly, birds have wings $\implies$ build a contraption with wings
  - We want to learn, the brain learns. The brain has many neurons and synapses $\implies$ if we get many simple units connected together we can build a model that learns like the brain

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

The “one learning algorithm” theory

- The brain can perform various tasks (process vision, sense of touch, do math, play pickle ball)
    - It doesn’t have thousands of different programs; it 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 we can implement a (simplified) version of the brain algorithm, we can have a ML model that can learn anything

Why resurgence of neural networks?

- Proposed in the 1950, popular in '80s and '90s but then they fell out of fashion
  - Interest declined due to limitations (e.g., XOR problem, lack of data, compute)
- **Key Reasons for Resurgence**
  - **1. Increased Computational Power**
    - GPUs and TPUs enable training of large models
    - Parallel processing suited for matrix operations in neural nets
  - **2. Availability of Big Data**
    - Large datasets crucial for deep learning success
    - Internet, IoT, and digital storage generate massive amounts of labeled data
  - **3. Algorithmic Improvements**
    - Better activation functions (ReLU)
    - Advanced optimization techniques (Adam, RMSprop)
    - Regularization methods (dropout, batch normalization)
  - **4. Breakthrough Architectures**
    - CNNs for image tasks
    - RNNs and LSTMs for sequences
    - Transformers for language and vision
  - **5. Open-Source Ecosystem**
    - Frameworks like TensorFlow and PyTorch simplify experimentation

Neural networks 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 an not learned
- Each neuron in a neural network performs logistic regression
  - Features used are computed by other neurons
  - We are not limited to using inputs $\underline{\boldsymbol{x}}$ or polynomial terms derived from $\underline{\boldsymbol{x}}$; we can infer features automatically

Neural networks

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

Model of a neural network perceptron

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

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

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

Activation Functions

- Map the neuron's input signal s to an output activation value
- Introduce non-linearity to enable learning complex functions
- **General Behavior**:
  - Approximately linear for small s
  - Saturate or threshold for large $|s|$
  - Many functions cross the origin or have $\theta(0) = 0$
- **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
- **ReLU (Rectified Linear Unit)**:

Neural networks to compute boolean functions

- We can build `AND`, `OR`, `NOT` functions with linear perceptrons using proper bias and weights
- Consider a sigmoid activation function $\theta(s)$ with threshold where $y = -1$ for $s < -1$ and $y = 1$ for $s > 1$
- `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:

$$y = \theta(w_0 * 1 + w_1 * x_1 + w_2 * x_2) = 2(-1.5 + w_1 + w_2)$$

    Weights are $(w_0, w_1, w_2) = (-3, 2, 2)$
- For `OR`
  - use weights $(-10, 20, 20)$
  - the only way to get $y = 0$ is for x_1 and x_2 to be 0

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
  - This is due to the compositional structure: each layer builds on the previous one
- **Role of Nonlinearity**
  - Nonlinear activation functions (e.g., sigmoid, tanh, ReLU) are essential
  - Without nonlinearity, stacked layers reduce to a single linear transformation
  - Nonlinearity allows modeling of complex, non-linear decision boundaries
- **Universal Approximation Theorem**
  - A feedforward network with:
    - A single hidden layer of nonlinear units
    - An output layer of linear units
  - Can approximate any continuous function to arbitrary precision
  - Implies shallow networks are theoretically powerful, though they may be impractically large
- **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

Issues with fitting a neural networks

1. Generalization: we need to match the model complexity to the data resources, since the model has so much flexibility we need lots of data
2. Optimization: there are several layers of perceptrons with an hard threshold, which turns the optimization problem into a combinatorial one

Feedforward vs Recurrent Neural Networks

- **Feedforward Neural Networks**
  - Information flows in one direction from input to output 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 temporal or sequential dependencies (can only consider a fixed window of inputs)
- **Recurrent Neural Networks (RNNs)**
  - Allow cycles in the computational graph with delays
    - Each unit can take input from its previous output: adds memory
    - Designed to process sequences: outputs depend on current and previous inputs
  - Update rule: $z_t = f_w(z_{t-1}, x_t)$ defines a time-homogeneous process
  - 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 in terms of layers

- **Layered Architecture**
  - A feedforward network consists of an ordered set of layers indexed by l
  - Typical structure includes:
    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
  - Layers can vary in size: $d(l)$ is layer-specific
- **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
- **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
  - Enables the network to approximate complex, non-linear functions
- **Output Layer ($l = L$)**
  - Final layer producing the output vector $\mathbf{y}$
  - Output can be:

Conventions for neurons and weights 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
  - ij are organized as input-output
- Weights are identified by 3 indices $w_{ij}^{(l)}$ where:
  - $0 \leq l \leq L$ 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: we start from 0 to account for the bias; we use $l-1$ in $d(l-1)$ since we 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 $x_j^{(l)}$:
  - This neuron has $d(l-1)$ inputs from the previous layer combined with the weights to compute the signal $s_j^{(l)}$, and then 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 only neuron in the last layer $y = h(\underline{\boldsymbol{x}}) = x_1^{(L)}$:
  - The last neuron outputs $s_1^{(L)}$ or $\theta(s_1^{(L)})$ depending on regression or classification setup

Vectorized feedforward propagation algorithm

- Neuron evaluation can be vectorized:
  - 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 a matrix-vector product:

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

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

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

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

- Note that by convention, we don't regularize the bias, 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, then 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)`, car = `(0, 1, 0, 0)`, ...
- Instead of training n neural networks, train a single neural network with an output layer of n nodes
  - a global optimization vs n local optimizations

Cost function for multi-class neural network classification

- If we encode one-hot the expected outputs $\underline{\boldsymbol{y}}_i$ and the outputs from the model $\underline{\boldsymbol{h}}(\underline{\boldsymbol{x}}_i)$:

$$E_{in}(\underline{\boldsymbol{w}}) = -\frac{1}{N}\sum_{i}\sum_{k}\underline{\boldsymbol{y}}_i|_k \log \underline{\boldsymbol{h}}(\underline{\boldsymbol{x}}_i)|_k + (1-\underline{\boldsymbol{y}}_i|_k)\log(1-\underline{\boldsymbol{h}}(\underline{\boldsymbol{x}}_i)|_k) + \frac{\lambda}{N}\sum_{l=1}^{L}\sum_{j=1}^{d(l)} \sum_{i}^{d(}$$

- Note that again we avoid to consider the inputs ($l \neq 0$) and the bias terms ($i \neq 0, j \neq 0$)

Fitting a neural networks for SGD

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

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

    - The same reasoning holds for both regression and classification:

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

- 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 ...

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

Computing the gradient

- Need to compute all the partial derivatives to get:

$$\nabla_{\underline{\boldsymbol{w}}} e(\underline{\boldsymbol{w}}_0)$$

- We 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
  - There is a very efficient algorithm (backpropagation or BackProp) that makes computing the gradient efficient
- Backpropagation
  - 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
  - Propagates error backward from the output layer to the input layer
  - Updates weights to minimize the overall prediction error

Backpropagation in Neural Networks

- **Steps of Backpropagation**
  1. **Forward Pass**:
     - Compute outputs of each neuron layer by layer from input to output
     - Store activations and weighted sums (z values) for use in backward pass
  2. **Compute Output Error**:
     - At the output layer, compute the error derivative using:
       $$\delta^L = \nabla_{\mathbf{y}} \mathcal{L} \circ \theta'(z^L)$$
     - Where $\mathcal{L}$ is the loss function and θ' is the derivative of the activation
  3. **Backward Pass**:
     - For each hidden layer $l = L - 1$ down to 1, compute:
       $$\delta^l = (\mathbf{w}^{l+1})^T \delta^{l+1} \circ \theta'(z^l)$$
     - δ^l represents the error signal for layer l
  4. **Gradient Computation**:
     - Gradient of the loss w.r.t. weight w_{ij}^l:
       $$\frac{\partial \mathcal{L}}{\partial w_{ij}^l} = \delta_j^l \cdot a_i^{l-1}$$
     - Where a_i^{l-1} is the activation from the previous layer
  5. **Parameter Update**:
     - Using gradient descent:
       $$w_{ij}^l \leftarrow w_{ij}^l - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}^l}$$
     - η is the learning rate

SGD + backpropagation pseudo-algorithm for 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
- If we want to use batch gradient descent (instead of SGD)

Gradient checking

- For some algorithms (e.g., back-propagation in neural networks), the analytical expression of the gradient becomes complicated, and mistakes are possible
- One approach is to compute the gradient numerically:

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

  and then compare the analytical gradient to the numerical approximation
- One should pick ε small (e.g., $\varepsilon = 10^{-4}$) but not so small to cause numerical issues
- Automatic differentiation packages solve this issue

Automatic Differentiation and End-to-End Learning

- **Automatic Differentiation (AD)**
  - Computes gradients by applying calculus rules to numerical programs
  - Avoids manual gradient derivation for new architectures
- **Backpropagation as Reverse Mode AD**
  - A special case of reverse mode automatic differentiation
  - Applies the chain rule efficiently from output to input
  - Beneficial for models with many inputs but few outputs
- **Practical Benefits**
  - Major deep learning frameworks (e.g., TensorFlow, PyTorch) implement AD
  - Enables rapid experimentation with network structures, activation functions, and loss functions
  - Frees users 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

Advanced Neural Network Architectures

- Neural networks
- **Advanced Neural Network Architectures**
  - Convolutional networks
  - Recurrent Neural Networks (RNNs)
  - Deep learning learning algorithms
  - Deep learning architectures

Convolutional networks

- Neural networks
- Advanced Neural Network Architectures
  - **Convolutional networks**
  - Recurrent Neural Networks (RNNs)
  - Deep learning learning algorithms
  - Deep learning architectures

Introduction to Convolutional Networks

- **Motivation**
  - Standard feedforward networks do not scale well with high-dimensional inputs like images
  - Convolutional neural networks (CNNs) are designed to exploit spatial structure in data
- **Key Idea**
  - Use local connectivity and 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

Convolution Operation and Feature Maps

- **Convolutional Layer Mechanics**
  - Filter (or kernel): small matrix of weights (e.g., 3×3)
  - Applies dot product between filter and local patch of input
  - Produces a **feature map** showing activations across the input
- **Weight Sharing**
  - Each filter is reused across all spatial locations
  - Reduces number of parameters and improves generalization
- **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)

Advantages and Applications of CNNs

- **Advantages**
  - 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)
- **Regularization via Pooling**
  - Pooling layers help summarize features and reduce overfitting
  - Common pooling: max pooling, average pooling
- **Common Architectures**
  - LeNet, AlexNet, VGG, ResNet — widely used in vision tasks
- **Applications**
  - Image classification, object detection, face recognition
  - Also used in NLP, audio processing, and medical imaging

Recurrent Neural Networks (RNNs)

- Neural networks
- Advanced Neural Network Architectures
    - Convolutional networks
    - **Recurrent Neural Networks (RNNs)**
    - Deep learning learning algorithms
    - Deep learning architectures

Recurrent Neural Networks (RNNs)

- Designed for sequential data (e.g., time series, text)
- Maintain a **hidden state** that evolves over time
- Struggle with long-term dependencies (vanishing gradients)
- Suitable for simple sequences
- Output depends on current input and previous hidden state
- Example: predicting next word in a sentence

Long Short-Term Memory (LSTM) Networks

- Solve vanishing gradient problem of RNNs
- Introduce memory cells and gates:
  - Forget, input, and output gates
- Control what to remember and forget
- Able to capture long-term dependencies
- Widely used in NLP and time series forecasting
- Example: LSTM for language modeling tasks

Gated Recurrent Units (GRUs)

- Simplified variant of LSTM networks
- Combine forget and input gates into an **update gate**
- Fewer parameters than LSTM
- Faster training with comparable performance
- Effective for many sequential tasks
- Example: GRU-based network for speech recognition

Transformer Architecture

- Relies on **self-attention** mechanisms
- Processes all input tokens simultaneously (no recurrence)
- Key components:
  - Multi-head attention
  - Positional encoding
- Scales better than RNNs/LSTMs for large datasets
- Backbone of modern NLP models (e.g., BERT, GPT)
- Example: Transformer model for machine translation

Deep learning learning algorithms

- Neural networks
- Advanced Neural Network Architectures
  - Convolutional networks
  - Recurrent Neural Networks (RNNs)
  - **Deep learning learning algorithms**
  - Deep learning architectures

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

Deep learning architectures

- Neural networks
- Advanced Neural Network Architectures
    - Convolutional networks
    - Recurrent Neural Networks (RNNs)
    - Deep learning learning algorithms
    - **Deep learning architectures**

Choosing a NN architecture

- The number of neurons in the input layer is determined by the number of features (fixed in the problem)
- The number of neurons in the output layer is determined by the number of classes (fixed in the problem)
- We need to choose as hyper-parameters
  - The number of hidden layers
  - The number of neurons per hidden layer

Reasonable choices for NN architecture

- A single hidden layer ($l = 1$)
- Multiple hidden layers with the same number of neurons
- Number of neurons in the hidden layers comparable to the number of features (e.g., from 1x to 3x)