Deep Learning: Feedforward Neural Nets and Convolutional Neural Nets

Piyush Rai

Machine Learning (CS771A)

Nov 2, 2016
A Prelude: Linear Models

- Linear models are nice and simple

- Were some of the first models for learning from data (e.g., Perceptron, 1958)
A Prelude: Linear Models

- Linear models are nice and simple
- Were some of the first models for learning from data (e.g., Perceptron, 1958)
- But linear models have limitations: Can’t learn nonlinear functions
A Prelude: Linear Models

- Linear models are nice and simple

- Were some of the first models for learning from data (e.g., Perceptron, 1958)

- But linear models have limitations: Can’t learn nonlinear functions

- Before kernel methods (e.g., SVMs) were invented, people thought about this a lot and tried to come up with ways to address this
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an input layer, one or more hidden layers, and an output layer

Nodes in the hidden layers compute a nonlinear transform of the inputs.

Also called a Feedforward Neural Network.

"Feedforward": no backward connections between layers (no loops).

Note: All nodes between layers are assumed connected with each other.

Universal Function Approximator (Hornik, 1991): A one hidden layer FFNN with sufficiently large number of hidden nodes can approximate any function.

Caveat: This result is only in terms of theoretical feasibility. Learning the model can be very difficult in practice (e.g., due to optimization difficulties).
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an input layer, one or more hidden layers, and an output layer

- Nodes in the hidden layers compute a nonlinear transform of the inputs

- Also called a Feedforward Neural Network

  - "Feedforward": no backward connections between layers (no loops)

  - Note: All nodes between layers are assumed connected with each other

  - Universal Function Approximator (Hornik, 1991): A one hidden layer FFNN with sufficiently large number of hidden nodes can approximate any function

  - Caveat: This result is only in terms of theoretical feasibility. Learning the model can be very difficult in practice (e.g., due to optimization difficulties)
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an input layer, one or more hidden layers, and an output layer

- Nodes in the hidden layers compute a nonlinear transform of the inputs

- Also called a Feedforward Neural Network
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an input layer, one or more hidden layers, and an output layer
- Nodes in the hidden layers compute a nonlinear transform of the inputs
- Also called a Feedforward Neural Network
- “Feedforward”: no backward connections between layers (no loops)
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an **input layer**, one or more **hidden layers**, and an **output layer**
- Nodes in the hidden layers compute a **nonlinear transform** of the inputs
- Also called a **Feedforward Neural Network**
- “Feedforward”: no backward connections between layers (no loops)
- Note: All nodes between layers are assumed connected with each other
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an input layer, one or more hidden layers, and an output layer

- Nodes in the hidden layers compute a nonlinear transform of the inputs

- Also called a Feedforward Neural Network

- “Feedforward”: no backward connections between layers (no loops)

- Note: All nodes between layers are assumed connected with each other

- Universal Function Approximator (Hornik, 1991): A one hidden layer FFNN with sufficiently large number of hidden nodes can approximate any function

Caveat: This result is only in terms of theoretical feasibility. Learning the model can be very difficult in practice (e.g., due to optimization difficulties)
Multi-layer Perceptron

- Composed of several Perceptron-like units arranged in multiple layers

- Consists of an input layer, one or more hidden layers, and an output layer
- Nodes in the hidden layers compute a nonlinear transform of the inputs
- Also called a Feedforward Neural Network
- “Feedforward”: no backward connections between layers (no loops)
- Note: All nodes between layers are assumed connected with each other
- Universal Function Approximator (Hornik, 1991): A one hidden layer FFNN with sufficiently large number of hidden nodes can approximate any function
  - Caveat: This results is only in terms of theoretical feasibility. Learning the model can be very difficult in practice (e.g., due to optimization difficulties)
What do Hidden Layers Learn?

- Hidden layers can automatically extract features from data

Deep neural networks learn hierarchical feature representations

![Diagram of a neural network with input layer, hidden layers 1 to 3, and output layer.]
What do Hidden Layers Learn?

- Hidden layers can automatically extract features from data.

  Deep neural networks learn hierarchical feature representations.

- The bottom-most hidden layer captures very low level features (e.g., edges). Subsequent hidden layers learn progressively more high-level features (e.g., parts of objects) that are composed of previous layer’s features.
A Simple Feedforward Neural Net

- Below: FFNN with 4 inputs, one hidden layer with 3 nodes, and 1 output
A Simple Feedforward Neural Net

- Below: FFNN with 4 inputs, one hidden layer with 3 nodes, and 1 output

- Each hidden node computes a nonlinear transformation of its incoming inputs

![Feedforward Neural Network Diagram](image_url)
A Simple Feedforward Neural Net

- Below: FFNN with 4 inputs, one hidden layer with 3 nodes, and 1 output

- Each hidden node computes a nonlinear transformation of its incoming inputs
  - Weighted linear combination followed by a nonlinear "activation function"
A Simple Feedforward Neural Net

- Below: FFNN with 4 inputs, one hidden layer with 3 nodes, and 1 output

- Each hidden node computes a nonlinear transformation of its incoming inputs
  - Weighted linear combination followed by a nonlinear "activation function"
  - Nonlinearity required. Otherwise, the model would reduce to a linear model
A Simple Feedforward Neural Net

- Below: FFNN with 4 inputs, one hidden layer with 3 nodes, and 1 output

Each hidden node computes a nonlinear transformation of its incoming inputs

- Weighted linear combination followed by a nonlinear “activation function”
- Nonlinearity required. Otherwise, the model would reduce to a linear model

Output $y$ is a weighted comb. of the preceding layer’s hidden nodes (followed by another transform if $y$ isn’t real valued, e.g., binary/multiclass label)
For an FFNN with $D$ inputs $\mathbf{x} = [x_1, \ldots, x_D]$
For an FFNN with \( D \) inputs \( \mathbf{x} = [x_1, \ldots, x_D] \), a single hidden layer with \( K \) hidden nodes \( \mathbf{h} = [h_1, \ldots, h_K] \):

\[
\mathbf{h} = \begin{bmatrix} h_1 \\ \vdots \\ h_K \end{bmatrix} = \begin{bmatrix} f \left( \mathbf{w}_{11}^\top \mathbf{x}_1 \right) \\ \vdots \\ f \left( \mathbf{w}_{K1}^\top \mathbf{x}_D \right) \end{bmatrix}
\]

where \( f \) is the nonlinear activation function.
Feedforward Neural Net

For an FFNN with $D$ inputs $\mathbf{x} = [x_1, \ldots, x_D]$, a single hidden layer with $K$ hidden nodes $\mathbf{h} = [h_1, \ldots, h_K]$, and a scalar-valued output node $y$

$$y = \mathbf{v}^\top \mathbf{h}$$
Feedforward Neural Net

For an FFNN with $D$ inputs $\mathbf{x} = [x_1, \ldots, x_D]$, a single hidden layer with $K$ hidden nodes $\mathbf{h} = [h_1, \ldots, h_K]$, and a scalar-valued output node $y$

$$y = \mathbf{v}^\top \mathbf{h} = \mathbf{v}^\top f(\mathbf{W}^\top \mathbf{x})$$
For an FFNN with $D$ inputs $\mathbf{x} = [x_1, \ldots, x_D]$, a single hidden layer with $K$ hidden nodes $\mathbf{h} = [h_1, \ldots, h_K]$, and a scalar-valued output node $y$

$$y = \mathbf{v}^\top \mathbf{h} = \mathbf{v}^\top f(\mathbf{W}^\top \mathbf{x})$$

where $\mathbf{v} = [v_1 \ v_2 \ldots \ v_K] \in \mathbb{R}^K$
For an FFNN with $D$ inputs $\mathbf{x} = [x_1, \ldots, x_D]$, a single hidden layer with $K$ hidden nodes $\mathbf{h} = [h_1, \ldots, h_K]$, and a scalar-valued output node $y$

$$y = \mathbf{v}^\top \mathbf{h} = \mathbf{v}^\top f(\mathbf{W}^\top \mathbf{x})$$

where $\mathbf{v} = [v_1 \ v_2 \ \ldots \ v_K] \in \mathbb{R}^K$, $\mathbf{W} = [\mathbf{w}_1 \ \mathbf{w}_2 \ \ldots \ \mathbf{w}_K] \in \mathbb{R}^{D \times K}$
Feedforward Neural Net

For an FFNN with $D$ inputs $\mathbf{x} = [x_1, \ldots, x_D]$, a single hidden layer with $K$ hidden nodes $\mathbf{h} = [h_1, \ldots, h_K]$, and a scalar-valued output node $y$

$$y = \mathbf{v}^\top \mathbf{h} = \mathbf{v}^\top f(\mathbf{W}^\top \mathbf{x})$$

where $\mathbf{v} = [v_1 \ v_2 \ \ldots \ v_K] \in \mathbb{R}^K$, $\mathbf{W} = [\mathbf{w}_1 \ \mathbf{w}_2 \ \ldots \ \mathbf{w}_K] \in \mathbb{R}^{D \times K}$, $f$ is the nonlinear activation function
Feedforward Neural Net

For an FFNN with \( D \) inputs \( \mathbf{x} = [x_1, \ldots, x_D] \), a single hidden layer with \( K \) hidden nodes \( \mathbf{h} = [h_1, \ldots, h_K] \), and a scalar-valued output node \( y \)

\[
y = \mathbf{v}^\top \mathbf{h} = \mathbf{v}^\top f(\mathbf{W}^\top \mathbf{x})
\]

where \( \mathbf{v} = [v_1, v_2, \ldots, v_K] \in \mathbb{R}^K \), \( \mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K] \in \mathbb{R}^{D \times K} \), \( f \) is the nonlinear activation function

Each hidden node’s value is computed as: \( h_k = f(\mathbf{w}_k^\top \mathbf{x}) = f(\sum_{d=1}^{D} w_{dk} x_d) \)
(Deeper) Feedforward Neural Net

- Feedforward neural net with \( L \) hidden layers \( h^{(1)}, h^{(2)}, \ldots, h^{(L)} \) where
  \[
  h^{(1)} = f(W^{(1)\top} x) \quad \text{and} \quad h^{(\ell)} = f(W^{(\ell)\top} h^{(\ell-1)}), \quad \ell \geq 2
  \]

  ![Diagram of a feedforward neural network with \( L \) hidden layers.](image)

- Note: The hidden layer \( \ell \) contains \( K_\ell \) hidden nodes, \( W^{(1)} \) is of size \( D \times K_1 \), \( W^{(\ell)} \) for \( \ell \geq 2 \) is of size \( K_\ell \times K_{\ell+1} \), \( v \) is of size \( K_L \times 1 \)
Nonlinear Activation Functions

- Some popular choices for the nonlinear activation function $f$
  - Sigmoid: $f(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$ (range between 0-1)
  - tanh: $f(x) = 2\sigma(2x) - 1$ (range between -1 and +1)
  - Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

![Graphs of Sigmoid, tanh, and ReLU functions]
Nonlinear Activation Functions

- Some popular choices for the nonlinear activation function $f$
  - Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$ (range between 0-1)
  - tanh: $f(x) = 2\sigma(2x) - 1$ (range between -1 and +1)
  - Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

  ![Sigmoid](image1), ![tanh](image2), ![ReLU](image3)

- Sigmoid saturates and can kill gradients. Also not “zero-centered”
Nonlinear Activation Functions

- Some popular choices for the nonlinear activation function $f$
  - Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$ (range between 0-1)
  - tanh: $f(x) = 2\sigma(2x) - 1$ (range between -1 and +1)
  - Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

- Sigmoid saturates and can kill gradients. Also not “zero-centered”
- tanh also saturates but is zero-centered (thus preferred over sigmoid)
Nonlinear Activation Functions

- Some popular choices for the nonlinear activation function $f$
  - Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$ (range between 0-1)
  - tanh: $f(x) = 2\sigma(2x) - 1$ (range between -1 and +1)
  - Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

- Sigmoid saturates and can kill gradients. Also not “zero-centered”
- tanh also saturates but is zero-centered (thus preferred over sigmoid)
- ReLU is currently the most popular (also cheap to compute)
Learning Feedforward Neural Nets

- Want to learn the parameters by minimizing some loss function

- Backpropagation (gradient descent + chain rule for derivatives) is commonly used to do this efficiently
Consider the feedforward neural net with one hidden layer

Recall that \( h = [h_1 \ h_2 \ \ldots \ h_K] = f(W^T x) \)

Assuming a regression problem, the optimization problem would be

\[
\min_{W, v} \sum_{n=1}^{N} (y_n - v^T f(W^T x_n))^2 = \min_{W, v} \sum_{n=1}^{N} (y_n - \sum_{k=1}^{K} v_k f(w^T_k x_n))^2
\]

where \( w_k \) is the \( k \)-th column of the \( D \times K \) matrix \( W \).
Learning Feedforward Neural Nets

- Consider the feedforward neural net with one hidden layer

Recall that

\[ h = [h_1 \ h_2 \ \ldots \ h_K] = f(W^\top x) \]

- Assuming a regression problem, the optimization problem would be

\[
\min_{W, v} \frac{1}{2} \sum_{n=1}^{N} \left( y_n - v^\top f(W^\top x_n) \right)^2 = \min_{W, v} \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right)^2
\]

where \( w_k \) is the \( k \)-th column of the \( D \times K \) matrix \( W \)
Learning Feedforward Neural Nets

- We can learn the parameters by doing gradient descent (or stochastic gradient descent) on the objective function

\[
L = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right)^2 = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - v^\top h_n \right)^2
\]
Learning Feedforward Neural Nets

- We can learn the parameters by doing gradient descent (or stochastic gradient descent) on the objective function

\[ \mathcal{L} = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right)^2 = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - v^\top h_n \right)^2 \]

- Gradient w.r.t. \( \mathbf{v} = [v_1 \ v_2 \ldots \ v_K] \) is straightforward

\[ \frac{\partial \mathcal{L}}{\partial \mathbf{v}} = - \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right) h_n = - \sum_{n=1}^{N} e_n h_n \]
We can learn the parameters by doing gradient descent (or stochastic gradient descent) on the objective function

\[ \mathcal{L} = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right)^2 \]

Gradient w.r.t. \( v = [v_1 \ v_2 \ldots \ v_K] \) is straightforward

\[ \frac{\partial \mathcal{L}}{\partial v} = -\sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right) h_n = -\sum_{n=1}^{N} e_n h_n \]

Gradient w.r.t. the weights \( W = [w_1 \ w_2 \ldots \ w_K] \) is a bit more involved due to the presence of \( f \) but can be computed using chain rule

\[ \frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial f_k} \frac{\partial f_k}{\partial w_k} \]

(note: \( f_k = f(w_k^\top x) \))
Learning Feedforward Neural Nets

- We can learn the parameters by doing gradient descent (or stochastic gradient descent) on the objective function

\[
\mathcal{L} = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right)^2 = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - v^\top h_n \right)^2
\]

- Gradient w.r.t. \( v = [v_1 \ v_2 \ \ldots \ v_K] \) is straightforward

\[
\frac{\partial \mathcal{L}}{\partial v} = -\sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right) h_n = -\sum_{n=1}^{N} e_n h_n
\]

- Gradient w.r.t. the weights \( W = [w_1 \ w_2 \ \ldots \ w_K] \) is a bit more involved due to the presence of \( f \) but can be computed using chain rule

\[
\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial f_k} \frac{\partial f_k}{\partial w_k}
\]

  (note: \( f_k = f(w_k^\top x) \))

- We have:

\[
\frac{\partial \mathcal{L}}{\partial f_k} = -\sum_{n=1}^{N} (y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n)) v_k = -\sum_{n=1}^{N} e_n v_k
\]
Learning Feedforward Neural Nets

- We can learn the parameters by doing gradient descent (or stochastic gradient descent) on the objective function

\[ \mathcal{L} = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right)^2 = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - v^\top h_n \right)^2 \]

- Gradient w.r.t. \( v = [v_1 \ v_2 \ \ldots \ v_K] \) is straightforward

\[ \frac{\partial \mathcal{L}}{\partial v} = - \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n) \right) h_n = - \sum_{n=1}^{N} e_n h_n \]

- Gradient w.r.t. the weights \( \mathbf{W} = [\mathbf{w}_1 \ \mathbf{w}_2 \ \ldots \ \mathbf{w}_K] \) is a bit more involved due to the presence of \( f \) but can be computed using chain rule

\[ \frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial f_k} \frac{\partial f_k}{\partial w_k} \quad \text{(note: } f_k = f(w_k^\top x)) \]

- We have:

\[ \frac{\partial \mathcal{L}}{\partial f_k} = - \sum_{n=1}^{N} (y_n - \sum_{k=1}^{K} v_k f(w_k^\top x_n)) v_k = - \sum_{n=1}^{N} e_n v_k \]

- We have:

\[ \frac{\partial f_k}{\partial w_k} = \sum_{n=1}^{N} f'(w_k^\top x_n)x_n, \text{ where } f'(w_k^\top x_n) \text{ is } f' \text{'s derivative at } w_k^\top x_n \]
Learning Feedforward Neural Nets

- We can learn the parameters by doing gradient descent (or stochastic gradient descent) on the objective function

$$\mathcal{L} = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(\mathbf{w}_k^\top \mathbf{x}_n) \right)^2 = \frac{1}{2} \sum_{n=1}^{N} \left( y_n - \mathbf{v}^\top \mathbf{h}_n \right)^2$$

- Gradient w.r.t. $\mathbf{v} = [v_1 \ v_2 \ \ldots \ v_K]$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}} = -\sum_{n=1}^{N} \left( y_n - \sum_{k=1}^{K} v_k f(\mathbf{w}_k^\top \mathbf{x}_n) \right) \mathbf{h}_n = -\sum_{n=1}^{N} \mathbf{e}_n \mathbf{h}_n$$

- Gradient w.r.t. the weights $\mathbf{W} = [\mathbf{w}_1 \ \mathbf{w}_2 \ \ldots \ \mathbf{w}_K]$ is a bit more involved due to the presence of $f$ but can be computed using chain rule

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_k} = \frac{\partial \mathcal{L}}{\partial f_k} \frac{\partial f_k}{\partial \mathbf{w}_k}, \text{ (note: } f_k = f(\mathbf{w}_k^\top \mathbf{x}))$$

- We have:

$$\frac{\partial \mathcal{L}}{\partial f_k} = -\sum_{n=1}^{N} (y_n - \sum_{k=1}^{K} v_k f(\mathbf{w}_k^\top \mathbf{x}_n)) v_k = -\sum_{n=1}^{N} \mathbf{e}_n v_k$$

- We have:

$$\frac{\partial f_k}{\partial \mathbf{w}_k} = \sum_{n=1}^{N} f'(\mathbf{w}_k^\top \mathbf{x}_n) \mathbf{x}_n, \text{ where } f'(\mathbf{w}_k^\top \mathbf{x}_n) \text{ is } f' \text{'s derivative at } \mathbf{w}_k^\top \mathbf{x}_n$$

- These calculations can be done efficiently using backpropagation
Backpropagation

- Basically consists of a forward pass and a backward pass

- Forward pass computes the errors $e_n$ using the current parameters

- Backward pass computes the gradients and updates the parameters, starting from the parameters at the top layer and then moving backwards
Backpropagation

- Basically consists of a forward pass and a backward pass

- Forward pass computes the errors $e_n$ using the current parameters

- Backward pass computes the gradients and updates the parameters, starting from the parameters at the top layer and then moving backwards

- Also good at reusing previous computations (updates of parameters at any layer depends on parameters at the layer above)
Kernel Methods vs Deep Neural Nets

- Recall the prediction rule for a kernel method (e.g., kernel SVM)

\[ y = \sum_{n=1}^{N} \alpha_n k(x_n, x) \]

This is analogous to a single hidden layer NN with fixed/pre-defined hidden nodes \( \{k(x_n, x)\}_{n=1}^{N} \) and output layer weights \( \{\alpha_n\}_{n=1}^{N} \)

- The prediction rule for a deep neural network

\[ y = \sum_{k=1}^{K} v_k h_k \]

In this case, the \( h_k \)'s are learned from data (possibly after multiple layers of nonlinear transformations)

- Both kernel methods and deep NNs be seen as using nonlinear basis functions for making predictions. Kernel methods use fixed basis functions (defined by the kernel) whereas NN learns the basis functions adaptively from data
Wide vs Deep?

Why might we prefer a deep model over a wide and shallow model?

- Deep "programs" can reuse computational subroutines (and are more compact)
- Learning certain functions may require a huge number of units in a shallow model
Wide vs Deep?

Why might we prefer a deep model over a wide and shallow model?

An informal justification:
- Deep “programs” can reuse computational subroutines (and are more compact)
Wide vs Deep?

Why might we prefer a deep model over a wide and shallow model?

An informal justification:
- Deep “programs” can reuse computational subroutines (and are more compact)

Learning Certain functions may require a huge number of units in a shallow model
Convolutional Neural Network (CNN)

- A feedforward neural network with a special structure
Convolutional Neural Network (CNN)

- A feedforward neural network with a **special structure**
- **Sparse “local” connectivity** between layers (except the last output layer). Reduces the number of parameters to be learned
Convolutional Neural Network (CNN)

- A feedforward neural network with a **special structure**
- **Sparse “local” connectivity** between layers (except the last output layer). Reduces the number of parameters to be learned

  ![Diagram of sparse connectivity between layers]

- **Shared weights** (like a “global” filter). Helps capture the local properties of the signal (useful for data such as images or time-series)

  ![Diagram of shared weights (feature map)]
Convolutional Neural Network (CNN)

- Uses a sequence of 2 operations, convolution and pooling (subsampling), applied repeatedly on the input data.
Convolutional Neural Network (CNN)

- Uses a sequence of 2 operations, convolution and pooling (subsampling), applied repeatedly on the input data

- Convolution: Extract “local” properties of the signal. Uses a set of “filters” that have to be learned (these are the “weighted” $W$ between layers)
Convolutional Neural Network (CNN)

- Uses a sequence of 2 operations, convolution and pooling (subsampling), applied repeatedly on the input data.

Convolution: Extract “local” properties of the signal. Uses a set of “filters” that have to be learned (these are the “weighted” $W$ between layers).

Pooling: Downsamples the outputs to reduce the size of representation.
Convolutional Neural Network (CNN)

- Uses a sequence of 2 operations, convolution and pooling (subsampling), applied repeatedly on the input data

- Convolution: Extract “local” properties of the signal. Uses a set of “filters” that have to be learned (these are the “weighted” $W$ between layers)

- Pooling: Downsamples the outputs to reduce the size of representation

- Note: A nonlinearity is also introduced after the convolution layer
Convolution

- An operation that captures local (e.g., spatial) properties of a signal

Mathematically, the operation is defined as

\[ h_{ij}^k = f((W^k \ast X)_{ij} + b_k) \]

where \( W^k \) is a filter, \( \ast \) is the convolution operator, and \( f \) is a nonlinearity.

- Usually a number of filters \( \{W^k\}_{k=1}^K \) are applied (each will produce a separate “feature map”). These filters have to be learned.

- Size of these filters have to be specified.
Pooling/Subsampling

This operation is used to reduce the size of the representation.
Deep Neural Nets: Some Comments

- Highly effective in learning good feature representations from data in an “end-to-end” manner
Deep Neural Nets: Some Comments

- Highly effective in learning good feature representations from data in an “end-to-end” manner
- The objective functions of these models are highly non-convex
  - But lots of recent work on non-convex optimization, so non-convexity doesn’t scare us (that much) anymore
Deep Neural Nets: Some Comments

- Highly effective in learning good feature representations from data in an “end-to-end” manner

- The objective functions of these models are highly non-convex
  - But lots of recent work on non-convex optimization, so non-convexity doesn’t scare us (that much) anymore

- Training these models is computationally very expensive
  - But GPUs can help to speed up many of the computations

Machine Learning (CS771A)

Deep Learning: Feedforward Neural Nets and CNNs
Deep Neural Nets: Some Comments

- Highly effective in learning good feature representations from data in an “end-to-end” manner
- The objective functions of these models are highly non-convex
  - But lots of recent work on non-convex optimization, so non-convexity doesn’t scare us (that much) anymore
- Training these models is computationally very expensive
  - But GPUs can help to speed up many of the computations
- Training these models can be tricky, especially a proper initialization
  - But now we have several ways to intelligently initialize these models (e.g., unsupervised layer-wise pre-training)
Deep Neural Nets: Some Comments

- Highly effective in learning good feature representations from data in an “end-to-end” manner
- The objective functions of these models are highly non-convex
  - But lots of recent work on non-convex optimization, so non-convexity doesn’t scare us (that much) anymore
- Training these models is computationally very expensive
  - But GPUs can help to speed up many of the computations
- Training these models can be tricky, especially a proper initialization
  - But now we have several ways to intelligently initialize these models (e.g., unsupervised layer-wise pre-training)
- Deep learning models can also be probabilistic and generative, e.g., deep belief networks (we did not consider these here)