Day 7: Convolutional Neural Networks

Summer STEM: Machine Learning

Nikola Janjušević, Akshaj Kumar Veldanda, Jacky Yuan, Tejaishwarya Gagadam

Department of Electrical and Computer Engineering
NYU Tandon School of Engineering
Brooklyn, New York

July 16, 2019

Outline

Extending Logistic Regression

- Motivation: Feature engineering in the model

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$
- Replace $\mathbf{x}$ with $\mathbf{z} = f(W\mathbf{x} + b)$: $\hat{y} = \sigma(Wz + b)$

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$
- Replace $\mathbf{x}$ with $\mathbf{z} = f(W\mathbf{x} + b)$: $\hat{y} = \sigma(Wz + b)$
- So, $\hat{y} = \sigma(W_2 f(W_1\mathbf{x} + b_1) + b_2)$

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$
- Replace $\mathbf{x}$ with $\mathbf{z} = f(W\mathbf{x} + b)$: $\hat{y} = \sigma(Wz + b)$
- So, $\hat{y} = \sigma(W_2 f(W_1\mathbf{x} + b_1) + b_2)$
- **Reminder**: all linear transforms can be represented as matrix multiplication

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$
- Replace $\mathbf{x}$ with $\mathbf{z} = f(W\mathbf{x} + b)$: $\hat{y} = \sigma(Wz + b)$
- So, $\hat{y} = \sigma(W_2 f(W_1\mathbf{x} + b_1) + b_2)$
- **Reminder**: all linear transforms can be represented as matrix multiplication
- We use non-linear function as f to give us a more expressive model

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$
- Replace $\mathbf{x}$ with $\mathbf{z} = f(W\mathbf{x} + b)$: $\hat{y} = \sigma(Wz + b)$
- So, $\hat{y} = \sigma(W_2 f(W_1\mathbf{x} + b_1) + b_2)$
- **Reminder**: all linear transforms can be represented as matrix multiplication
- We use non-linear function as f to give us a more expressive model
  - Recall polynomial transformations and exponential transformations of the data

Extending Logistic Regression

- Motivation: Feature engineering in the model
  - Removes need for domain knowledge
  - Domain knowledge often doesn't exist: ex. object recognition
- Logistic Regression Model: $\hat{y} = \sigma(W\mathbf{x} + b)$
- Replace $\mathbf{x}$ with $\mathbf{z} = f(W\mathbf{x} + b)$: $\hat{y} = \sigma(Wz + b)$
- So, $\hat{y} = \sigma(W_2 f(W_1\mathbf{x} + b_1) + b_2)$
- **Reminder**: all linear transforms can be represented as matrix multiplication
- We use non-linear function as f to give us a more expressive model
  - Recall polynomial transformations and exponential transformations of the data
  - These cannot be expressed as matrix multiplication

Extension to Neural Network

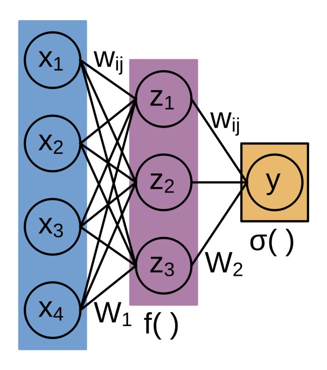
- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values

Extension to Neural Network

- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values
  - Simplest example of a **Neural Network**

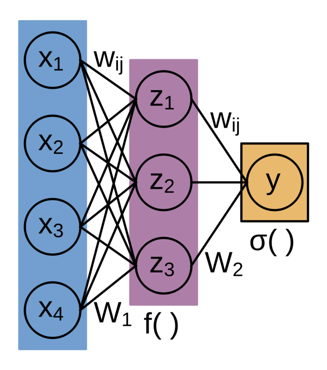
Extension to Neural Network

- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values
  - Simplest example of a **Neural Network**
- $\hat{y} = \sigma(W_2 f_1(W_1\mathbf{x} + \mathbf{b}_1) + b_2)$



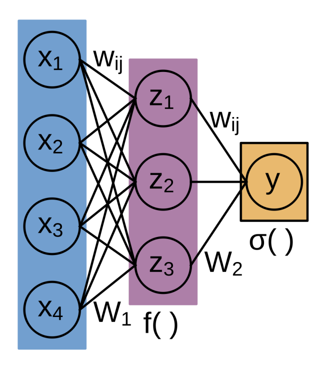
Extension to Neural Network

- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values
  - Simplest example of a **Neural Network**
- $\hat{y} = \sigma(W_2 f_1(W_1\mathbf{x} + \mathbf{b}_1) + b_2)$
- We can optimize for both $W_1, \mathbf{b}_1$ and $W_2, b_2 2$ model-parameters



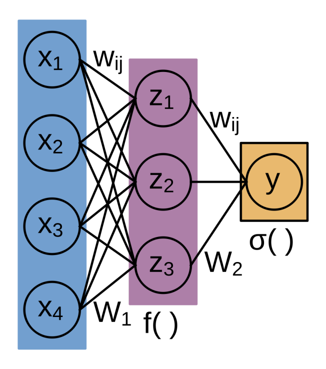
Extension to Neural Network

- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values
  - Simplest example of a **Neural Network**
- $\hat{y} = \sigma(W_2 f_1(W_1\mathbf{x} + \mathbf{b}_1) + b_2)$
- We can optimize for both $W_1, \mathbf{b}_1$ and $W_2, b_2 2$ model-parameters
  - $\nabla J = [\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, ..., \frac{\partial J}{\partial w_P}]^T$



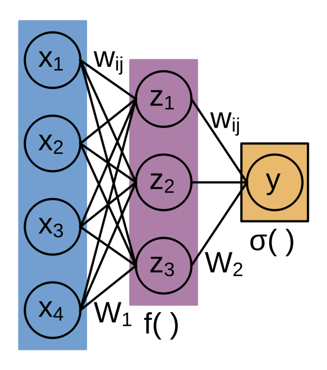
Extension to Neural Network

- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values
  - Simplest example of a **Neural Network**
- $\hat{y} = \sigma(W_2 f_1(W_1\mathbf{x} + \mathbf{b}_1) + b_2)$
- We can optimize for both $W_1, \mathbf{b}_1$ and $W_2, b_2 2$ model-parameters
  - $\nabla J = [\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, ..., \frac{\partial J}{\partial w_P}]^T$
  - Now we're *learning* the feature engineering



Extension to Neural Network

- Restrict $f(\mathbf{x})$ to non-linear function applied to all input values
  - Simplest example of a **Neural Network**
- $\hat{y} = \sigma(W_2 f_1(W_1\mathbf{x} + \mathbf{b}_1) + b_2)$
- We can optimize for both $W_1, \mathbf{b}_1$ and $W_2, b_2 2$ model-parameters
  - $\nabla J = [\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, ..., \frac{\partial J}{\partial w_P}]^T$
  - Now we're *learning* the feature engineering
- But why stop here?...



Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

  - Where, $z_l = f_l(W_l\mathbf{z}_{l-1} + b_l)$ for $1 \leq l \leq L$, $z_0\mathbf{x}$, and L is the number of hidden layers

Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

  - Where, $z_l = f_l(W_l\mathbf{z}_{l-1} + b_l)$ for $1 \leq l \leq L$, $z_0\mathbf{x}$, and L is the number of hidden layers
  - ie. all hidden layers are non-linear activation of linear transform

Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

  - Where, $z_l = f_l(W_l\mathbf{z}_{l-1} + b_l)$ for $1 \leq l \leq L$, $z_0\mathbf{x}$, and L is the number of hidden layers
  - ie. all hidden layers are non-linear activation of linear transform
  - f_{out} depends on type of ML problem: (regression: linear, classification: sigmoid/soft-max)

Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

  - Where, $z_l = f_l(W_l\mathbf{z}_{l-1} + b_l)$ for $1 \leq l \leq L$, $z_0\mathbf{x}$, and L is the number of hidden layers
  - ie. all hidden layers are non-linear activation of linear transform
  - f_{out} depends on type of ML problem: (regression: linear, classification: sigmoid/soft-max)
    - **Regression**: Linear Output

Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

  - Where, $z_l = f_l(W_l\mathbf{z}_{l-1} + b_l)$ for $1 \leq l \leq L$, $z_0\mathbf{x}$, and L is the number of hidden layers
  - ie. all hidden layers are non-linear activation of linear transform
  - f_{out} depends on type of ML problem: (regression: linear, classification: sigmoid/soft-max)
    - **Regression**: Linear Output
    - **Binary Classification**: Sigmoid Output

Mathematical Model: Multi-Layer Perceptron

- Model:

$$\hat{\mathbf{y}} = f_{out}(W_{out}\mathbf{z}_L + b_{out})$$

  - Where, $z_l = f_l(W_l\mathbf{z}_{l-1} + b_l)$ for $1 \leq l \leq L$, $z_0\mathbf{x}$, and L is the number of hidden layers
  - ie. all hidden layers are non-linear activation of linear transform
  - f_{out} depends on type of ML problem: (regression: linear, classification: sigmoid/soft-max)
    - **Regression**: Linear Output
    - **Binary Classification**: Sigmoid Output
    - **Multi-Class Classification**: Soft-max Output

Layers

- **Input**: feature vector, $\mathbf{x}$

Layers

- **Input**: feature vector, $\mathbf{x}$
- **Output**: target vector, $\hat{\mathbf{y}}$

Layers

- **Input**: feature vector, $\mathbf{x}$
- **Output**: target vector, $\hat{\mathbf{y}}$
  - linear/logistic regression

Layers

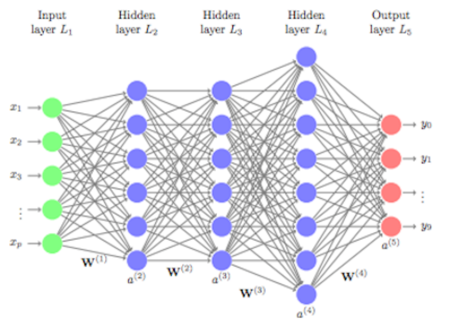
- **Input**: feature vector, $\mathbf{x}$
- **Output**: target vector, $\hat{\mathbf{y}}$
  - linear/logistic regression
- **Hidden**: intermediate vectors, $\mathbf{z}$ or $\mathbf{a}$

Layers

- **Input**: feature vector, $\mathbf{x}$
- **Output**: target vector, $\hat{\mathbf{y}}$
  - linear/logistic regression
- **Hidden**: intermediate vectors, $\mathbf{z}$ or $\mathbf{a}$
  - feature extraction

Layers

- **Input**: feature vector, $\mathbf{x}$
- **Output**: target vector, $\hat{\mathbf{y}}$
  - linear/logistic regression
- **Hidden**: intermediate vectors, $\mathbf{z}$ or $\mathbf{a}$
  - feature extraction



Common Activation Functions

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

Common Activation Functions

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - $\sigma(z) \in (0, 1)$

Common Activation Functions

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - $\sigma(z) \in (0, 1)$
- Tanh (hyperbolic tangent): $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Common Activation Functions

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - $\sigma(z) \in (0, 1)$
- Tanh (hyperbolic tangent): $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - $tanh(z) \in [-1, 1]$

Common Activation Functions

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - $\sigma(z) \in (0, 1)$
- Tanh (hyperbolic tangent): $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - $tanh(z) \in [-1, 1]$
- ReLu (Rectified Linear Unit): $relu(z) = max(0, z)$

Common Activation Functions

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - $\sigma(z) \in (0, 1)$
- Tanh (hyperbolic tangent): $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - $tanh(z) \in [-1, 1]$
- ReLu (Rectified Linear Unit): $relu(z) = max(0, z)$
  - easy to compute, performs well in practice

Guidelines for Designing a NN

- The design space for NN is HUGE

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer
  - bs: batch-size

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer
  - bs: batch-size
  - lr: learning-rate

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer
  - bs: batch-size
  - lr: learning-rate
  - # of epochs

NYU TANDON SCHOOL OF ENGINEERING

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer
  - bs: batch-size
  - lr: learning-rate
  - # of epochs
  - λ: weight-regularization constant

Guidelines for Designing a NN

- The design space for NN is HUGE
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer
  - bs: batch-size
  - lr: learning-rate
  - # of epochs
  - λ: weight-regularization constant
  - J: cost/loss function

Guidelines for Designing a NN

- **The design space for NN is HUGE**
- Hyper-parameters so far:
  - L: # of layers
  - N_L: # hidden units per layer
  - f: activation function for each layer
  - bs: batch-size
  - lr: learning-rate
  - # of epochs
  - λ: weight-regularization constant
  - J: cost/loss function
- This can be overwhelming...

NYU TANDON SCHOOL OF ENGINEERING

Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers

Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - # hidden units ~ 128

Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - # hidden units $\sim$ 128
  - make sure code is working

Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - $\#$ hidden units ~ 128
  - make sure code is working
  - increase size if val good

Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - $\#$ hidden units ~ 128
  - make sure code is working
  - increase size if val good
  - classification acc $\geq$ guessing

Guidelines for Designing a NN

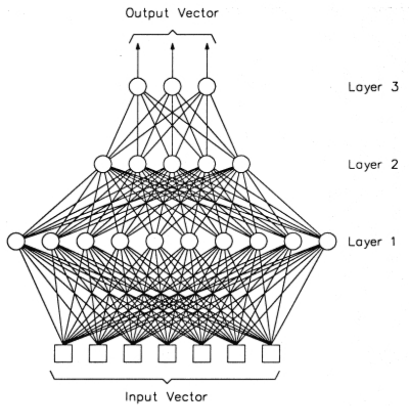
- **Start Small**: 1 or 2 layers
  - # hidden units ~ 128
  - make sure code is working
  - increase size if val good
  - classification acc $\geq$ guessing
- **One activation function**

Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - $\#$ hidden units ~ 128
  - make sure code is working
  - increase size if val good
  - classification acc $\geq$ guessing
- **One activation function**
  - for all hidden layers

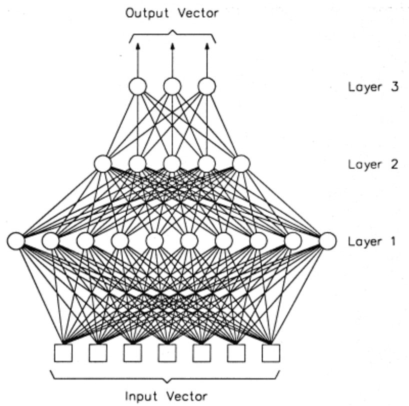
Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - # hidden units ~ 128
  - make sure code is working
  - increase size if val good
  - classification acc $\geq$ guessing
- **One activation function**
  - for all hidden layers
- **Simple MLP Arch**:



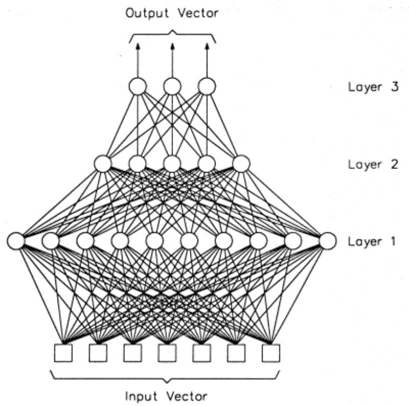
Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - # hidden units $\sim$ 128
  - make sure code is working
  - increase size if val good
  - classification acc $\geq$ guessing
- **One activation function**
  - for all hidden layers
- **Simple MLP Arch**:
  - Pyramid



Guidelines for Designing a NN

- **Start Small**: 1 or 2 layers
  - $\#$ hidden units ~ 128
  - make sure code is working
  - increase size if val good
  - classification acc $\geq$ guessing
- **One activation function**
  - for all hidden layers
- **Simple MLP Arch**:
  - Pyramid
  - Expand, combine & reduce



Outline

Better performance with images

- Encoding locality
- How does an MLP see an image?
- Is this how we see images?

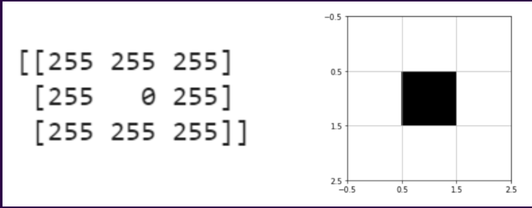
Examples: Lena & Mandrill

Outline

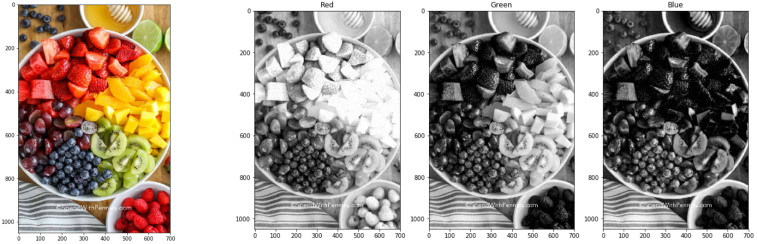
Images in Computer

- Images are stored as arrays of quantized numbers in computers
- Gray scale image: 2D matrices with each entry specifying the intensity (brightness) of a pixel
    - Pixel values range from 0 to 255, 0 being the darkest, 255 being the brightest



Color Images

- Color image: 3D array, 2 dimensions for space, 1 dimension for color
  - Can be thought of as three 2D matrices stacked together into a cube, each 2D matrix specify the amount of each color: Red ,Green ,Blue value at each pixel



- Shape of this image: (1050,700,3)
- There are 1050x700 pixels, 3 channels: R,G,B

Outline

Limitations of Fully Connected Network

- In Fashion MNIST, we used a fully connected network, in which each neuron in the hidden layer is connected to all $28x28 = 784$ pixels

Limitations of Fully Connected Network

- In Fashion MNIST, we used a fully connected network, in which each neuron in the hidden layer is connected to all 28x28 $=$ 784 pixels
- Higher definition images often contain millions of pixels $\rightarrow$ It is not practical to use fully connected network

Limitations of Fully Connected Network

- In Fashion MNIST, we used a fully connected network, in which each neuron in the hidden layer is connected to all 28x28 $=$ 784 pixels
- Higher definition images often contain millions of pixels $\rightarrow$ It is not practical to use fully connected network
- Fully connected network treat each individual pixel as a feature, it does not utilize the positional relationship between pixels

Convolution

- Introducing a new operation: Convolution

Convolution

- Introducing a new operation: Convolution
- An operation on an image(matrix) X with a kernel W

Convolution

- Introducing a new operation: Convolution

- An operation on an image(matrix) X with a kernel W
- $Z = X \circledast W$

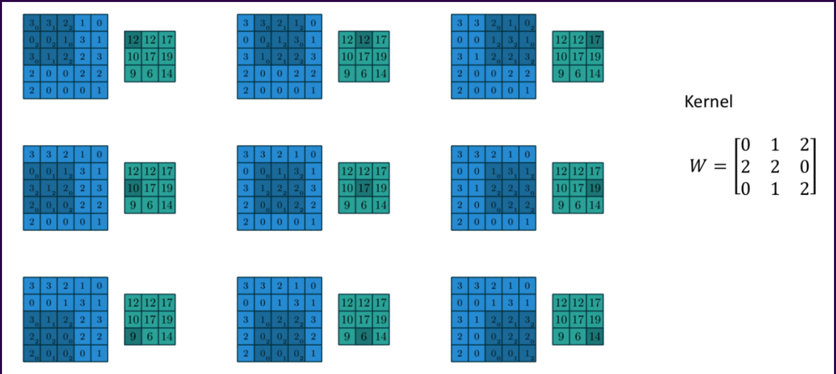
Convolution

- Introducing a new operation: Convolution
- An operation on an image(matrix) X with a kernel W
- $Z = X \circledast W$
- Equation:

At each offset (j_1, j_2) compute:

$$Z[j_1, j_2] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} W[k_1, k_2] X[j_1 + k_1, j_2 + k_2]$$

Example of a Convolultion



Why Convolution?

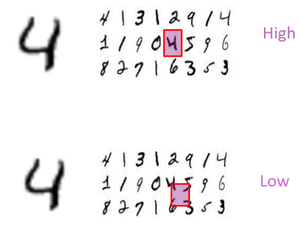
- With convolution, each output pixel depends on only the neighboring pixels in the input

Why Convolution?

- With convolution, each output pixel depends on only the neighboring pixels in the input
- This allows us to learn the positional relationship between pixels

Why Convolution?

- With convolution, each output pixel depends on only the neighboring pixels in the input
- This allows us to learn the positional relationship between pixels
- Use of different kernels allows us to detect features



Convolution for Multiple Channels

- A kernel for each channel. Could be same kernel, or different
- Perform a convolution for each of the channel, with the respective kernel
- Sum the results

Outline

Averaging Kernels

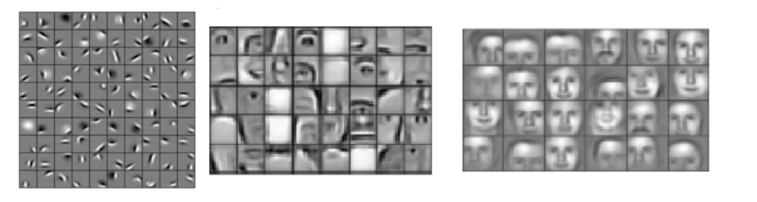
- Uniform Kernel: $\frac{1}{K_x K_y}\begin{bmatrix} 1 & .. & 1 \\ 1 & .. & 1 \\ 1 & .. & 1 \end{bmatrix}$

  $K_x =$ Number of Columns

  $K_y =$ Number of Rows
- Gaussian Kernel is a blurring kernel too.

Edge Detection

- Initial layers in a deep neural networks detect small patterns like lines, curves or edges.
- Subsequent layers combine these local features to create more complex features.



Edge Detection

- Using Sobel filters:
  - Vertical Edge Detection $G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$
  - Horizontal Edge Detection $G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$

Thank You!

- Next Class: Deep Learning and Applications of CNNs