CS231n: Deep Learning for Computer Vision (2024 Spring)

> Created by: *Haoming Cen*
>
> Date: 2024-08-31 (Week 1)

Schedule

(**Week 1**) Lecture 1: Introduction

- [x] Computer vision overview
- [x] Course overview

(**Week 1**) Lecture 2: Image Classification with Linear Classifiers

- [x] The data-driven approach
- [x] K-nearest neighbor
- [x] Linear Classifiers
- [x] Algebraic / Visual / Geometric viewpoints
- [x] SVM and Softmax loss

(**Week 1**) Lecture 3: Regularization and Optimization

- [x] Regularization
- [x] Stochastic Gradient Descent
- [x] Momentum, AdaGrad, Adam
- [x] Learning rate schedules

(**Week 1**) Lecture 4: Neural Networks and Backpropagation

- [x] Multi-layer Perceptron
- [x] Backpropagation

(**Week 1**) Lecture 5: Image Classification with CNNs

- [x] Higher-level representations, image features
- [x] Convolution and pooling

(**Week 1**)

- [x] **Assignment 1**

(Week 2) Lecture 6: CNN Architectures

- [ ] Batch Normalization
- [ ] Transfer learning
- [ ] AlexNet, VGG, GoogLeNet, ResNet

(Week 2) Lecture 7: Recurrent Neural Networks

- [ ] RNN, LSTM, GRU
- [ ] Language modeling
- [ ] Image captioning
- [ ] Sequence-to-sequence

(Week 2) Lecture 8: Attention and Transformers

- ☐ Self-Attention
- ☐ Transformers

(Week 2) Lecture 9: Object Detection and Image Segmentation

- ☐ Single-stage detectors
- ☐ Two-stage detectors
- ☐ Semantic/Instance/Panoptic segmentation

(Week 3) Lecture 10: Video Understanding

- ☐ Video classification
- ☐ 3D CNNs
- ☐ Two-stream networks
- ☐ Multimodal video understanding

(Week 3) Lecture 11: Visualizing and Understanding

- ☐ Feature visualization and inversion
- ☐ Adversarial examples
- ☐ DeepDream and style transfer

(Week 3) Lecture 12: Self-supervised Learning

- ☐ Pretext tasks
- ☐ Contrastive learning
- ☐ Multisensory supervision

(Week 4)

- ☐ **Assignment 2**

(Week 4) Lecture 13: Generative Models

- ☐ Generative Adversarial Network
- ☐ Diffusion models
- ☐ Autoregressive models

(Week 4) Lecture 15: Robot Learning

- ☐ Deep Reinforcement Learning
- ☐ Model Learning
- ☐ Robotic Manipulation

(Week 4) Lecture 18: 3D Vision

- ☐ 3D shape representations
- ☐ Shape reconstruction
- ☐ Neural implicit representations

(Week 5)

- ☐ **Assignment 3**

Lecture 1: Introduction

Definition of Computer Vision: The study of visual data.

Difficult for network to understand

Receptive fields, binocular interaction and functional architecture in the cat's visual cortex by Hubel & Wiesel, 1962: Receptive fields - find edges

Block World by Larry Roberts, 1963: The first Phd. thesis of computer vision - simplifiy the visual world to basic geometric shapes - goal: reconstruct and recognize.

Theory of Edge Detection by David Marr, 1980: Abstract the basic structural elements of a visual scene - Input image $\rightarrow$ edge image (i.e. primal sketch) to recognize edges, virtural lines, boundaries $\rightarrow$ $2\frac{1}{2}$-D sketch to synthesize a coherent structure, combining related features and recognizing the boundaries of different objects $\rightarrow$ 3-D representation (volumetric primitives)

Segmentation: grouping the pixels into meaningful area.

Highlight: *Convolutional Neural Networks*

Lecture 2: Image Classification with Linear Classifiers

Input image $\rightarrow$ label $\in$ predetermined label set

Challenges:

- **Viewpoint variation**. A single instance of an object can be oriented in many ways with respect to the camera.
- **Scale variation.** Visual classes often exhibit variation in their size (size in the real world, not only in terms of their extent in the image).
- **Deformation**. Many objects of interest are not rigid bodies and can be deformed in extreme ways.
- **Occlusion**. The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
- **Illumination conditions**. The effects of illumination are drastic on the pixel level.
- **Background clutter**. The objects of interest may blend into their environment, making them hard to identify.
- **Intra-class variation**. The classes of interest can often be relatively broad, such as chair. There are many different types of these objects, each with their own appearance.

A good image classification model must be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.

Data-driven Approaches: provide the computer with many examples of each class and then develop learning algorithms that look at these examples and learn about the visual appearance of each class (from `predict(){}` to `train(){} predict(){}`).

Nearest neighbors: train $O(1)$, predict $O(n)$

Distance metrics:

- l_1 (Manhattan) distance: $d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$
- l_2 (Euclidean) distance: $d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$

Changing the coordinate frame won't affact the l_2 distance. If the individual entries of the input feature have special meaning, then use l_1.

> Not commonly used in CV

K-nearest neighbors:

- $y_i \in \mathcal{Y} = c_1, \cdots, c_m$
- Training set: $T = (x_1, y_1), \cdots, (x_N, y_N)$
- Nearest K neighbors: $N_k(x)$
- Predict class: $y = \arg\max\limits_{c_j \in \mathcal{Y}} \sum_{x_i \in N_k(x), i \in 1, \cdots, N} \mathbf{1}(y_i = c_j)$

Hyperparameters setting: train, validation, test (underlying statistical assumption: independently and identically distributed)

Linear Classifier: $f(\boldsymbol{x}; \mathbf{W}) = \mathbf{W}\boldsymbol{x} + \boldsymbol{b}$

- Algebraic Viewpoint: matrix operations
- Visual Viewpoint: pattens learned by the weight matrix
- Geometric Viewpoint: regions in representation space seperated by lines

To find a good $\mathbf{W}$:

1. A good loss function that quantifies our unhappiness with the scores across the training data.
2. An efficiently way of finding the parameters that minimize the loss function. (optimization)

SVM loss:

$$L_i = \sum_{j \neq y_i} \max(0, \boldsymbol{s}_j - \boldsymbol{s}_{y_i} + \Delta) = \sum_{j \neq y_i} \max\left(0, (\mathbf{W}\boldsymbol{x}_i)_j - (\mathbf{W}\boldsymbol{x}_i)_{y_i} + \Delta\right)$$

- Q: At initialization \mathbf{W} is small so all $s \approx 0$. What is the loss?: $C + \Delta$
- Q: What if we use $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)^2$?: The squared term has a greater penalty for large deviations

Softmax loss: multinomial logistic regression (interpret raw classifier scores as probabilitie)

$$s = f(x_i; \mathbf{W}), P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} L_i = -\log s = f(x_i; \mathbf{W})$$

- Kullback–Leibler divergence: $D_{KL}(P||Q) = \sum_y P(y) \log \frac{P(y)}{Qy}$
- Cross Entropy: $H(P, Q) = H(p) + D_{KL}(P||Q)$

Softmax classifier provides “probabilities” for each class. Unlike the SVM which computes uncalibrated and not easy to interpret scores for all classes, the Softmax classifier allows us to compute “probabilities” for all labels.

Lecture 3: Regularization and Optimization

Regularization

- strategy #1: random search
- **strategy #2: follow the slope**

Compute gradient:

- #1: Numeric gradient (finite differences approximation): $\frac{\mathrm{d}f(x)}{f(x)} = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
- #2: **Analytic gradient**

Stochastic Gradient Descent (SGD)

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1} NL_i(x_i, y_i, \mathbf{W}) + \lambda R(\mathbf{W})\nabla_{\mathbf{W}} L(\mathbf{W}) = \frac{1}{N}\sum_{i=1} N\nabla_{\mathbf{W}} L_i(x_i, y_i, \mathbf{W}) + \lambda \nabla_{\mathbf{W}} R(\mathbf{W})$$

Update parameters: $\Theta_{t+1} = \Theta_t - \alpha \nabla J(\Theta_t)$

Momentum, AdaGrad, Adam

What if the loss function has a local minima or saddle point? Use momentum

SGD + momentum: $v_{t+1} = \rho v_t + \nabla J(\Theta_t), \Theta_{t+1} = \Theta_t - \alpha v_{t+1}$

How to overcome overshoot?: progress along “steep” directions need to be damped, but progress along “flat” directions need to be accelerated.

RMSProp:

```
# SGD + momentum
v = 0
while True:
    dx = compute_gradient(x)
    v = rho * v + dx
    x = x - lr * v

# RMSProp
v = 0
while True:
    dx = compute_gradient(x)
    # Adds element-wise scaling of the gradient based on the historical sum of squares in
each dimension (with decay)
    v = rho * v + (1 - rho) * dx * dx
    # Per-parameter learning rates
    x = x - lr * dx / (np.sqrt(v) + 1e-7)
```

Adam

RMSProp + momentum

```
moment_1 = 0
moment_2 = 0
beta_1, beta_2 = 0.9, 0.999

for t in range(num_steps):
    dx = compute_gradent(x)
    # momentum
    moment_1 = beta_1 * moment_1 + (1 - beta_1) * dx
    # RMSProp
    moment_2 = beta_2 * moment_2 + (1 - beta_2) * dx * dx
    # RMSProp Bias correction
    beta_1 = moment_1 / (1 - beta_1 ** t)
    beta_2 = moment_2 / (1 - beta_2 ** t)
    # RMSProp
    x = x - learning_rate * moment_1 / (np.sqrt(moment_2) + 1e-7)
```

Learning rate schedules

Learning rate decays (*Step*, *Cosine*, *Linear*, *Inverse sqrt*, *Warmup*, etc.)

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

Mini-batch gradient descent

In large-scale applications, the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over **batches** of the training data.

Lecture 4: Neural Networks and Backpropagation

Neural Networks

Neural Networks are stacks of linear functions and nonlinear activation functions

What if we try to build a neural network without activation function?: We end up with a linear classifier again!

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- tanh: $\tanh(x)$
- ReLU: $\max(0, x)$
- LeakyReLU: $\max(\alpha x, x)$
- Maxout: $\max(w_1^T x + b_1, w_2^T x + b_2)$
- ELU: $ELU(x) = x(x \geq 0), \alpha(e^-1)(x < 0)$

"Fully-connected" layers: hidden layers & output layer

Backpropagation

How to compute gradients?: Computational graphs + Backpropagation

Chain Rule: upstream gradient $\times$ local gradient $=$ downstream gradient

Gates:

- Add: "gradient distributor" $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}$
- Mul: "swap multiplier" $\frac{\partial f}{\partial x} = y\frac{\partial f}{\partial g}, \frac{\partial f}{\partial y} = x\frac{\partial f}{\partial g}$
- Copy: "gradient adder" $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} + \frac{\partial f}{\partial z}$
- Max: "gradient router" $(x > y)$ $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}$

Vector derivatives

- Scalar to Scalar ($x \in \mathbb{R}, y \in \mathbb{R}$): Regular derivative ($\frac{\partial y}{\partial x} \in \mathbb{R}$)
- Vector to Scalar ($x \in \mathbb{R}^N, y \in \mathbb{R}$): Derivative is Gradient ($\frac{\partial y}{\partial x} \in \mathbb{R}^N$, $\left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$)
- Vector to Vector ($x \in \mathbb{R}^N, y \in \mathbb{R}^N$): Derivative is Jacobian ($\frac{\partial y}{\partial x} \in \mathbb{R}^{N\times M}$, $\left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$)

Gradients for vectorized operations Matrix-Matrix multiply gradient is possibly the most tricky operation is the matrix-matrix multiplication operations:

```
# forward pass
W = np.random.randn(5, 10)
X = np.random.randn(10, 3)
D = W.dot(X)

# now suppose we had the gradient on D from above in the circuit
dD = np.random.randn(*D.shape) # same shape as D
dW = dD.dot(X.T) #.T gives the transpose of the matrix
dX = W.T.dot(dD)
```

Tip: use dimension analysis! Note that you do not need to remember the expressions for `dW` and `dX` because they are easy to re-derive based on dimensions. For instance, we know that the gradient on the weights `dW` must be of the same size as `W` after it is computed, and that it must depend on matrix multiplication of `X` and `dD` (as is the case when both `X`, `W` are single numbers and not matrices). There is always exactly one way of achieving this so that the dimensions work out. For example, `X` is of size `[10 x 3]` and `dD` of size `[5 x 3]`, so if we want `dW` and `W` has shape `[5 x 10]`, then the only way of achieving this is with `dD.dot(X.T)`, as shown above.

Recommendation: explicitly write out a minimal vectorized example, derive the gradient on paper and then generalize the pattern to its efficient, vectorized form.

Lecture 5: Image Classification with CNNs

Convolutional Neural Networks

The spatial structure of images could be destroyed by flatten operation

Filters always extend the full depth of the input volume

ConvNet is a sequence of Convolution Layers, **interspersed with activation functions**

Output shape:

$$H_{out} = \frac{H_{in} + 2 \times \text{padding}_H - \text{kernel}_H}{\text{stride}_H} W_{out} = \frac{W_{in} + 2 \times \text{padding}_W - \text{kernel}_W}{\text{stride}_W}$$

Receptive Fields: For convolution with kernel size K, each element in the output depends on a $K \times K$ receptive field in the input

Pooling layer:

- makes the representations smaller and more manageable
- operates over each activation map independently

> 1×1 convolution can change the number of channels of input features, perform an independent linear transformation on each channel, thus changing the number of channels of the input features. Also, it can reduces the number of parameters of the model. In addition, from the perspective of a fully connected layer, the 1×1 convolution can be viewed as a special kind of fully connected layer.