DSC 140B
Representation Learning

Lecture 22 | Part 1

Gradient Descent for NN Training

Empirical Risk Minimization

0. Collect a training set, $\{(\vec{x}^{(i)}, y_i)\}$
1. Pick the form of the prediction function, H.
   - E.g., a neural network, H.
2. Pick a loss function.
3. Minimize the empirical risk w.r.t. that loss.

Minimizing Risk

- To minimize risk, we often use **vector calculus**.
  - Either set $\nabla_{\vec{w}} R(\vec{w}) = 0$ and solve...
  - Or use gradient descent: walk in opposite direction of $\nabla_{\vec{w}} R(\vec{w})$.

- Recall, $\nabla_{\vec{w}} R(\vec{w}) = (\partial R / \partial w_0, \partial R / \partial w_1, \ldots, \partial R / \partial w_d)^T$

In General

- Let ℓ be the loss function, let $H(\vec{x}; \vec{w})$ be the prediction function.

- The empirical risk:

$$R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- Using the chain rule:

$$\nabla_{\vec{w}} R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \ell}{\partial H} \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})$$

Training Neural Networks

- For neural networks with nonlinear activations, the risk $R(\vec{w})$ is typically **complicated.**

- The mininimizer cannot be found directly.

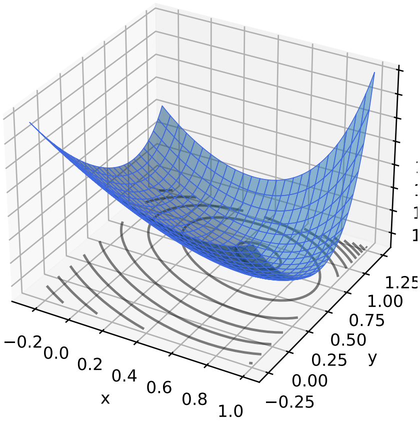
- Instead, we use iterative methods, such as **gradient descent**.

Iterative Optimization

- To minimize a function $f(\vec{x})$, we may try to compute $\vec{\nabla} f(\vec{x})$; set to 0; solve.
- Often, there is **no closed-form solution**.
- How do we minimize f?

Example

- Consider $f(x,y) = e^{x^2+y^2} + (x-2)^2 + (y-3)^2$.



Example

- Try solving $\vec{\nabla} f(x, y) = 0$.

- The gradient is:

$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix}$$

- Can we solve the system?

$$2xe^{x^2+y^2} + 2(x-2) = 0$$
$$2ye^{x^2+y^2} + 2(y-3) = 0$$

Example

- Try solving $\vec{\nabla} f(x, y) = 0$.

- The gradient is:

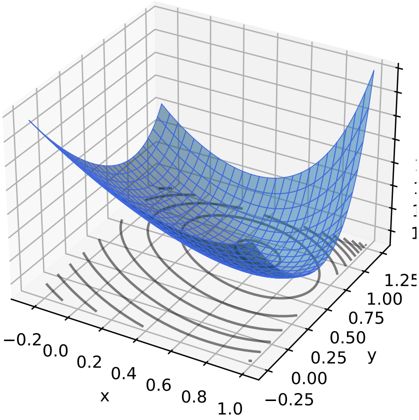
$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix}$$

- Can we solve the system? **Not in closed form.**

$$2xe^{x^2+y^2} + 2(x-2) = 0$$
$$2ye^{x^2+y^2} + 2(y-3) = 0$$

Idea

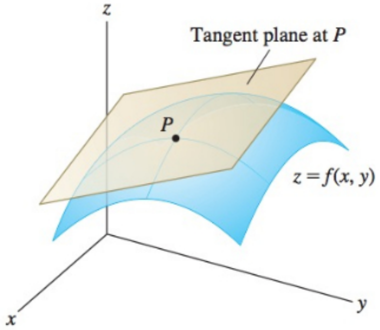
- Apply an iterative approach.
- Start at an arbitrary location.
- “Walk downhill”, towards minimum.



Which way is down?

- Consider a differentiable function $f(x, y)$.
- We are standing at $P = (x_0, y_0)$.
- In a small region around P, f looks like a plane.
- Slope of plane in x, y directions:

$$\frac{\partial f}{\partial x}(x_0, y_0) \quad \frac{\partial f}{\partial y}(x_0, y_0)$$



The Gradient

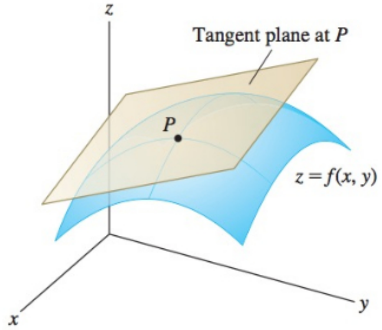
- Let $f : \mathbb{R}^d \to \mathbb{R}$ be differentiable. The **gradient** of f at $\vec{x}$ is defined:

$$\vec{\nabla} f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}(\vec{x}), \frac{\partial f}{\partial x_2}(\vec{x}), \ldots, \frac{\partial f}{\partial x_d}(\vec{x}) \right)^T$$

- **Note:** $\vec{\nabla} f(\vec{x})$ is a **function** mapping $\mathbb{R}^d \to \mathbb{R}^d$.

Which way is down?

- $\vec{\nabla} f(x_0, y_0)$ points in direction of steepest **ascent** at (x_0, y_0).
- $-\vec{\nabla} f(x_0, y_0)$ points in direction of steepest **descent** at (x_0, y_0).



Gradient Properties

- The gradient is used in the linear approximation of f:

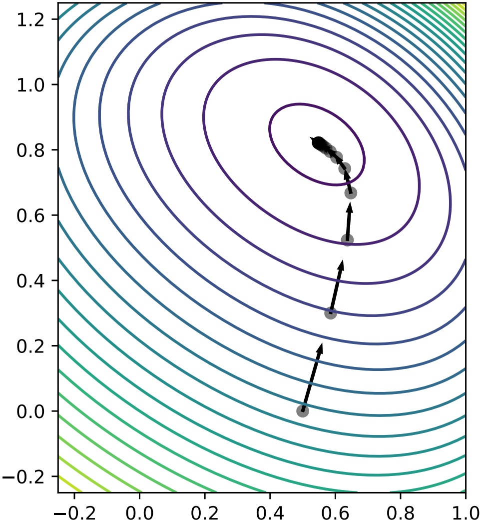
$$f(x_0 + \delta_x, y_0 + \delta_y) \approx f(x_0, y_0) + \vec{\delta} \cdot \vec{\nabla} f(x_0, y_0)$$

- Important properties:
  - $\vec{\nabla} f(\vec{x})$ points in direction of **steepest ascent** at $\vec{x}$.
  - $-\vec{\nabla} f(\vec{x})$ points in direction of **steepest descent** at $\vec{x}$.
  - In directions orthogonal to $\vec{\nabla} f(\vec{x})$, f does not change!
  - $\|\vec{\nabla} f(\vec{x})\|$ measures steepness of ascent

Gradient Descent

- Pick arbitrary starting point $\vec{x}^{(0)}$, **learning rate** parameter $\eta > 0$.
- Until convergence, repeat:
  - Compute gradient of f at $\vec{x}^{(i)}$; that is, compute $\vec{\nabla} f(\vec{x}^{(i)})$.
  - Update $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \eta \vec{\nabla} f(\vec{x}^{(i)})$.
- When do we stop?
  - When difference between $\vec{x}^{(i)}$ and $\vec{x}^{(i+1)}$ is negligible.
  - I.e., when $\|\vec{x}^{(i)} - \vec{x}^{(i+1)}\|$ is small.

```python
def gradient_descent(
        gradient, x, learning_rate=.01,
        threshold=.1e-4
):
    while True:
        x_new = x - learning_rate * gradient(x)
        if np.linalg.norm(x - x_new) < threshold:
            break
        x = x_new
    return x
```



1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

Backprop Revisited

- The weights of a neural network can be trained using **gradient descent**.
- This requires the gradient to be calculated repeatedly; this is where **backprop** enters.
- Sometimes people use “backprop” to mean “backprop + SGD”, but this is not strictly correct.

Backprop Revisited

- Consider training a NN using the square loss:

$$\begin{aligned}\nabla_{\vec{w}} R(\vec{w}) &= \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \ell}{\partial H} \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w}) \\ &= \frac{2}{n}\sum_{i=1}^{n} (H(\vec{x}^{(i)}) - y_i)\, \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})\end{aligned}$$

Backprop Revisited

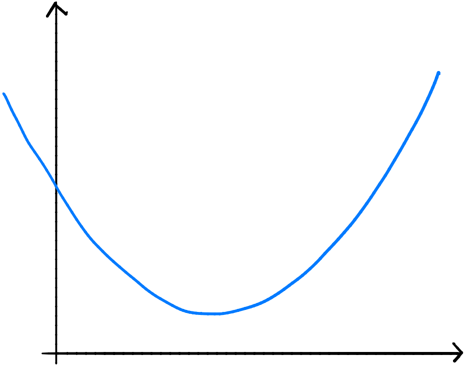
- Interpretation:

$$\nabla_{\vec{w}} R(\vec{w}) = \frac{2}{n} \sum_{i=1}^{n} \underbrace{\left(H(\vec{x}^{(i)}) - y_i\right)}_{\text{Error}} \underbrace{\nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})}_{\text{Blame}}$$

- When used in SGD, backprop “propagates error backward” in order to update weights.

Difficulty of Training NNs

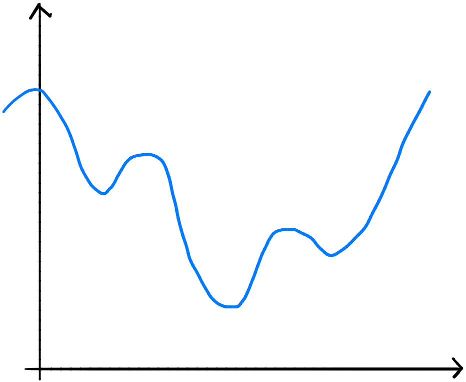
- Gradient descent is guaranteed to find optimum when objective function is **convex**.[1]



[1] Assuming it is properly initialized

Difficulty of Training NNs

- When activations are non-linear, neural network risk is **highly non-convex**:



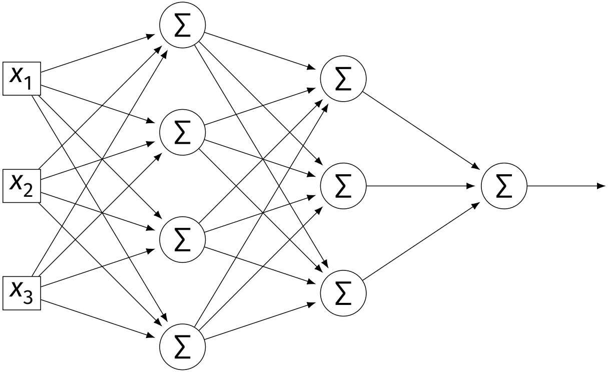
Non-Convexity

- When R is non-convex, GD can get “stuck” in local minima.
  - Solution depends on initialization.

- More sophisticated optimizers, using momentum, adaptation, better initialization, etc.
  - Adagrad, RMSprop, Adam, etc.

Difficulty of Training (Deep) NNs

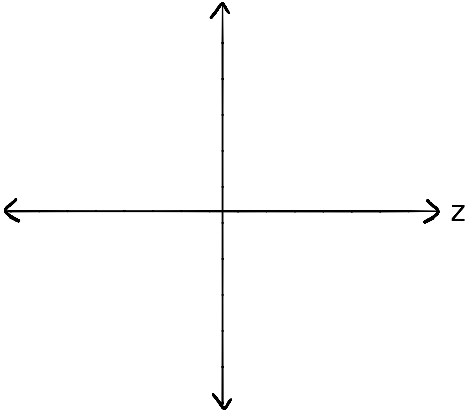
- Deep networks can suffer from the problem of **vanishing gradients**: if w is a weight at the "front" of the network, $\partial H / \partial w$ can be very small



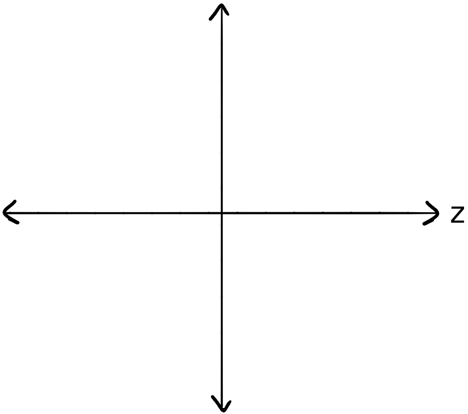
Vanishing Gradients

- If $\partial H/\partial w$ is always close to zero, w is updated **very slowly** by gradient descent.
- In short: early layers are slower to train.
- One mitigation: use ReLU instead of sigmoid.

Vanishing Gradients



Sigmoid



ReLU

DSC 140B

Representation Learning

Lecture 22 | Part 2

Stochastic Gradient Descent

Gradient Descent for Minimizing Risk

- In ML, we often want to minimize a **risk function**:

$$R(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Observation

- The gradient of the risk function is a sum of gradients:

$$\vec{\nabla} R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- One term for each point in training data.

Problem

- In machine learning, the number of training points n can be **very large**.
- Computing the gradient can be **expensive** when n is large.
- Therefore, each step of gradient descent can be **expensive**.

Idea

- The (full) gradient of the risk uses all of the training data:

$$\nabla R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- It is an average of n gradients.
- **Idea:** instead of using all n points, randomly choose $\ll n$.

Stochastic Gradient

- Choose a random subset (**mini-batch**) B of the training data.
- Compute a **stochastic gradient**:

$$\nabla R(\vec{w}) \approx \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Stochastic Gradient

$$\nabla R(\vec{w}) \approx \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- **Good:** if $|B| \ll n$, this is much faster to compute.
- **Bad:** it is a (random) approximation of the full gradient, noisy.

Stochastic Gradient Descent (SGD) for ERM

- Pick arbitrary starting point $\vec{x}^{(0)}$, **learning rate** parameter $\eta > 0$, batch size $m \ll n$.

- Until convergence, repeat:
  - Randomly sample a batch B of m training data points (on each iteration).
  - Compute stochastic gradient of f at $\vec{x}^{(i)}$:

$$\vec{g} = \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

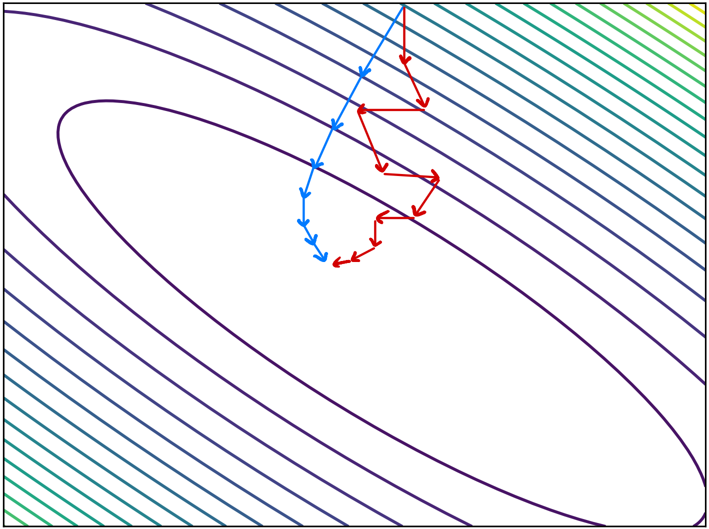
  - Update $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \eta \vec{g}$

Idea

- In practice, a stochastic gradient often works well enough.
- It is better to take many noisy steps quickly than few exact steps slowly.

Batch Size

- Batch size m is a parameter of the algorithm.
- The larger m, the more reliable the stochastic gradient, but the more time it takes to compute.
- Extreme case when $m = 1$ will still work.



Usefulness of SGD

- SGD allows learning on **massive** data sets.
- Useful even when exact solutions available.
  - E.g., least squares regression / classification.

Training NNs in Practice

- There are several Python packages for training NNs:
  - PyTorch
  - Tensorflow / Keras

- This week's discussion was a Tensorflow tutorial.

DSC 140B
Representation Learning

Lecture 22 | Part 3

Output Units

Output Units

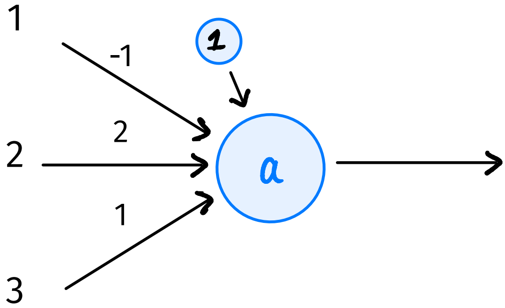
- As with units in hidden layers, can choose different activation functions for the outputs layey.
  - What activation function?
  - How many units?

- Good choice depends on task:
  - Regression, binary classification, multiclass, etc.

- Which loss?

Setting 1: Regression

- Output can be any real number.
- Single output neuron.
- It makes sense to use a **linear activation**.

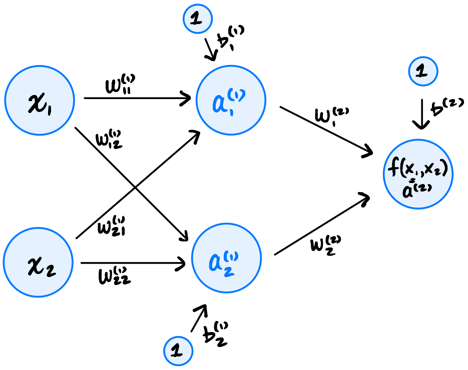


Setting 1: Regression

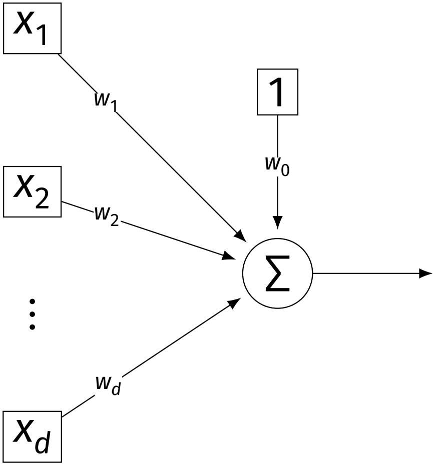
- Prediction should not be too high/low.
- It makes sense to use the **mean squared error**.

Setting 1: Regression

- Suppose we use linear activation for output neuron + mean squared error.
- This is very similar to least squares regression...
- But! Features in earlier layers are **learned**, non-linear.



Special Case: Least Squares



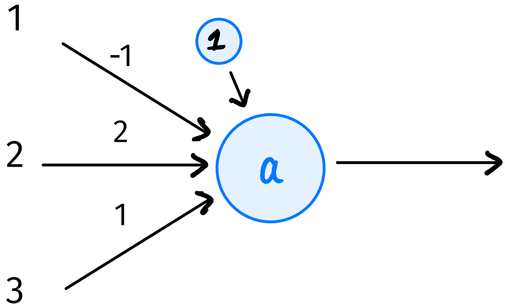
The case of:

- a one layer neural network
- with all linear activations
- trained with square loss

is also called **least squares regression**.

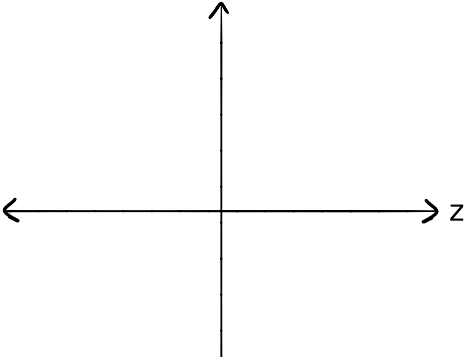
Setting 2: Binary Classification

- Output can be in [0, 1].
- Single output neuron.
- We *could* use a **linear activation**, threshold.
- But there is a better way.



Sigmoids for Classification

- Natural choice for activation in output layer for binary classification: the **sigmoid**.



Binary Classification Loss

- We *could use* square loss for binary classification. There are several reasons not to:
- 1) Square loss penalizes predictions which are “too correct”.
- 2) It doesn’t work well with the sigmoid due to saturation.

The Cross-Entropy

- Instead, we often train deep classifiers using the **cross-entropy** as loss.
- Let $y^{(i)} \in \{0, 1\}$ be true label of ith example.
- The average cross-entropy loss:

$$-\frac{1}{n}\sum_{i=1}^{n}\begin{cases}\log f(\vec{x}^{(i)}), & \text{if } y^{(i)} = 1\\ \log\left[1 - f(\vec{x}^{(i)})\right], & \text{if } y^{(i)} = 0\end{cases}$$

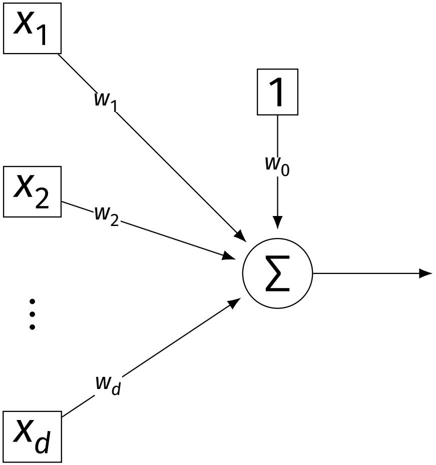
The Cross-Entropy and the Sigmoid

- Cross-entropy “undoes” the exponential in the sigmoid, resulting in less saturation.

Summary: Binary Classification

- Use sigmoidal activation the output layer + cross-entropy loss.
- This will promote a strong gradient.
- Use whatever activation for the hidden layers (e.g., ReLU).

Special Case: Logisitic Regression



The case of:

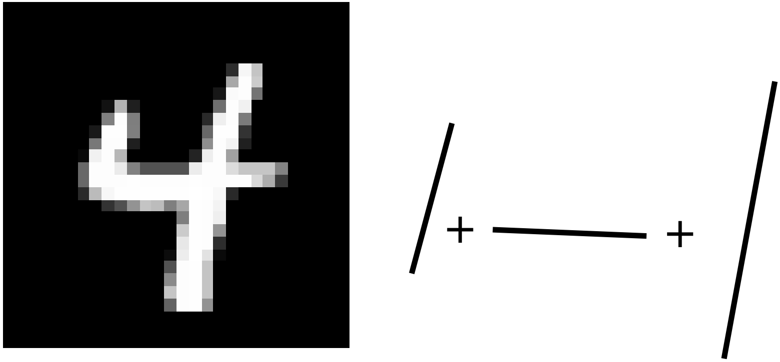
- a one layer neural network
- with sigmoid activation
- trained with cross-entropy loss

is also called **logistic regression**.

DSC 140B
Representation Learning

Lecture 22 | Part 4

Convolutions

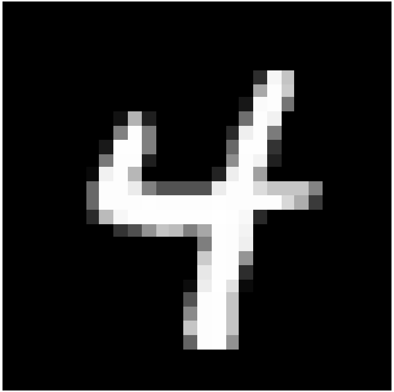


From Simple to Complex

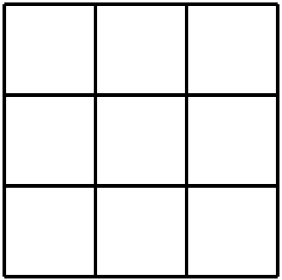
- Complex shapes are made of simple patterns
- The human visual system uses this fact
- Line detector → shape detector → … → face detector
- Can we replicate this with a deep NN?

Edge Detector

- How do we find **vertical edges** in an image?
- One solution: **convolution** with an **edge filter**.



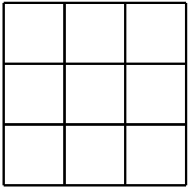
Vertical Edge Filter



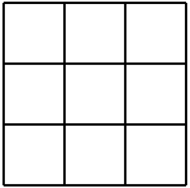
Idea

- Take a patch of the image, same size as filter.
- Perform “dot product” between patch and filter.
- If large, this is a (vertical) edge.

image patch:

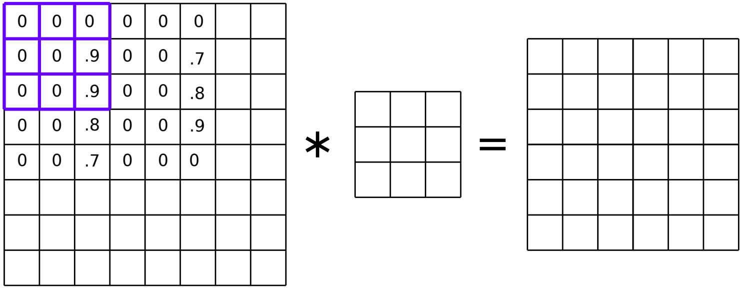


filter:



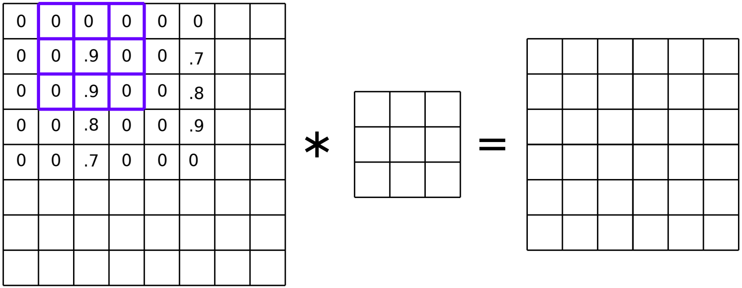
Idea

- Move the filter over the entire image, repeat procedure.



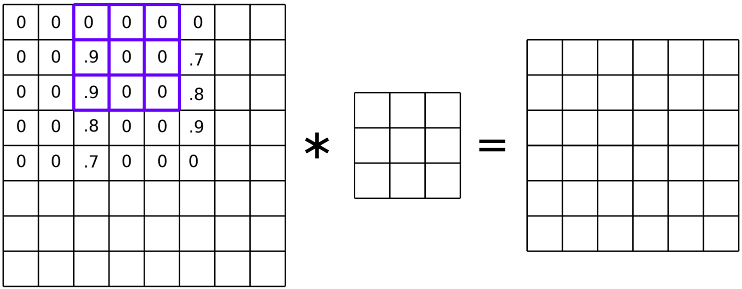
Idea

- Move the filter over the entire image, repeat procedure.



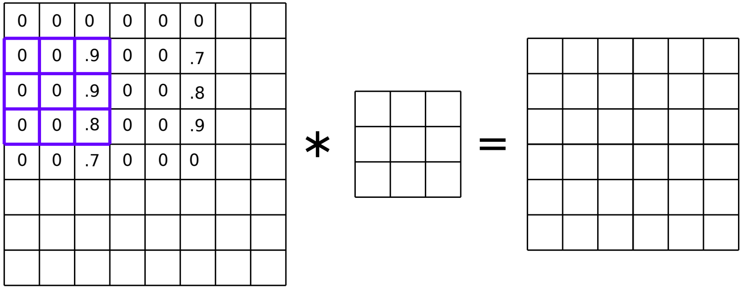
Idea

- Move the filter over the entire image, repeat procedure.



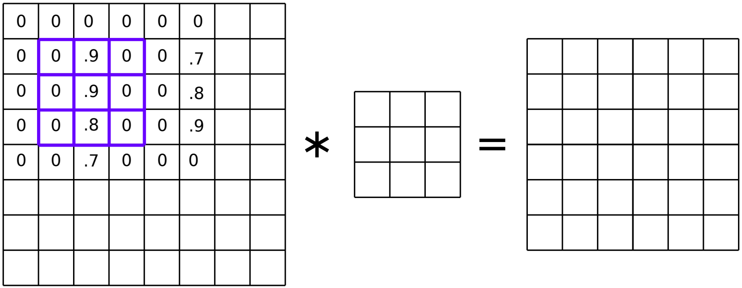
Idea

- Move the filter over the entire image, repeat procedure.



Idea

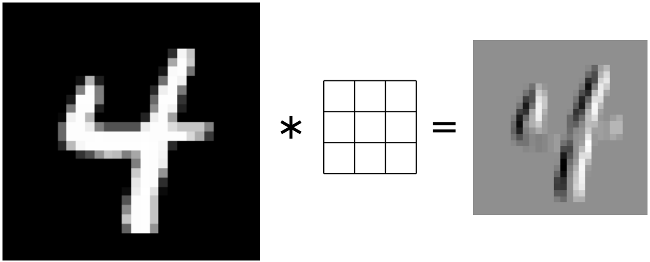
- Move the filter over the entire image, repeat procedure.



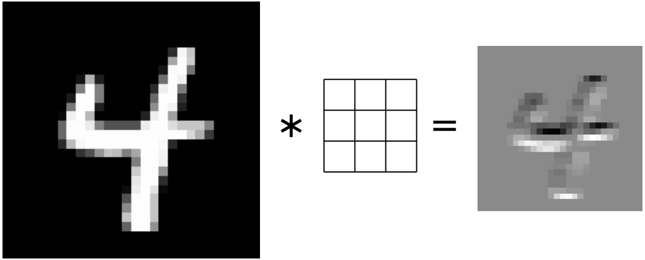
Convolution

- The result is the (2d) **convolution** of the filter with the image.
- Output is also 2-dimensional array.
- Called a **response map**.

Example: Vertical Filter



Example: Horizontal Filter



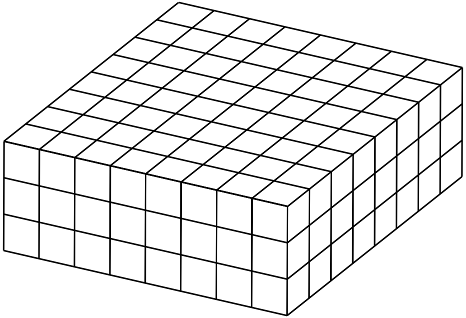
More About Filters

- Typically 3×3 or 5×5.
- Variations: different **stride**, image **padding**.

3-d Filters

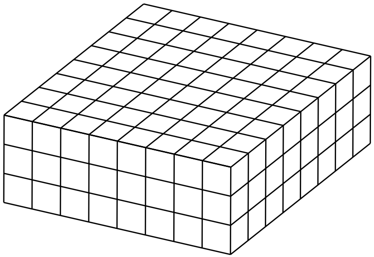
- Black and white images are 2-d arrays.
- But color images are 3-d arrays:
  - a.k.a., **tensors**
  - Three color **channels**: red, green, blue.
  - height × width × 3
- How does convolution work here?

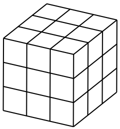
Color Image



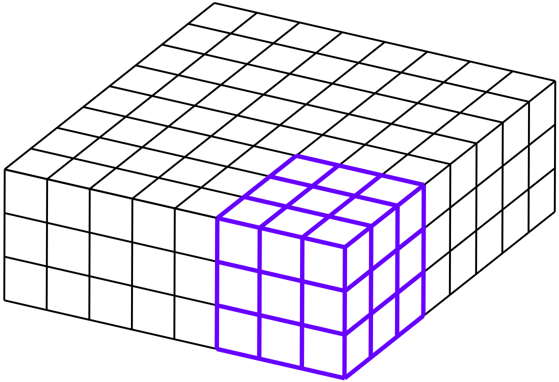
3-d Filter

- The filter must also have three channels:
  - $3 \times 3 \times 3$, $5 \times 5 \times 3$, etc.

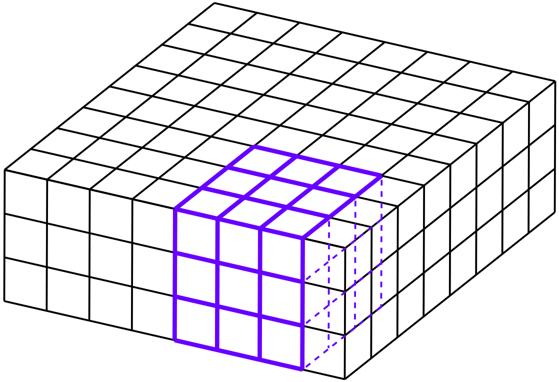




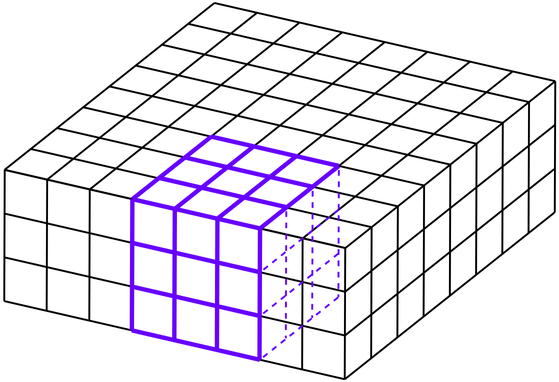
3-d Filter



3-d Filter



3-d Filter



Convolution with 3-d Filter

- Filter must have same number of channels as image.
  - 3 channels if image RGB.
- Result is still a 2-d array.

General Case

- Input "image" has k channels.
- Filter must have k channels as well.
  - e.g., $3 \times 3 \times k$
- Output is still $2-d$