Lecture 20: Convolutional Neural Nets

Mark Hasegawa-Johnson

ECE 417: Multimedia Signal Processing, Fall 2021

Outline

Review: How to train a neural network

1. Find a **training dataset** that contains n examples showing the desired output, $\vec{y}_i$, that the NN should compute in response to input vector $\vec{x}_i$:

$$\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$$

2. Randomly **initialize** the weights and biases, $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
3. Perform **forward propagation**: find out what the neural net computes as $\hat{y}_i$ for each $\vec{x}_i$.
4. Define a **loss function** that measures how badly $\hat{y}$ differs from $\vec{y}$.
5. Perform **back propagation** to improve $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
6. Repeat steps 3-5 until convergence.

Review: Second Layer = Piece-Wise Approximation

The second layer of the network approximates $\hat{y}$ using a bias term $\vec{b}$, plus correction vectors $\vec{w}_j^{(2)}$, each scaled by its activation h_j:

$$\hat{y} = \vec{b}^{(2)} + \sum_j \vec{w}_j^{(2)} h_j$$

- Unit-step and signum nonlinearities, on the hidden layer, cause the neural net to compute a piece-wise constant approximation of the target function. Sigmoid and tanh are differentiable approximations of unit-step and signum, respectively.
- ReLU, Leaky ReLU, and PReLU activation functions cause h_j, and therefore $\hat{y}$, to be a piece-wise-linear function of its inputs.

Review: First Layer = A Series of Decisions

The first layer of the network decides whether or not to “turn on” each of the h_j’s. It does this by comparing $\vec{x}$ to a series of linear threshold vectors:

$$h_k = \sigma\left(\bar{w}_k^{(1)}\vec{x} + b_k\right) \begin{cases} \approx 1 & \bar{w}_k^{(1)}\vec{x} + b_k > 0 \\ \approx 0 & \bar{w}_k^{(1)}\vec{x} + b_k < 0 \end{cases}$$

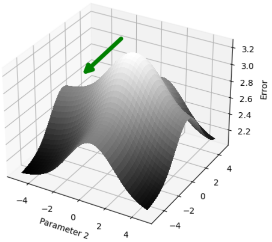
Gradient Descent: How do we improve W and b?

Given some initial neural net parameter, $w_{k,j}^{(\ell)}$, we want to find a better value of the same parameter. We do that using gradient descent:

$$w_{k,j}^{(\ell)} \leftarrow w_{k,j}^{(\ell)} - \eta \frac{d\mathcal{L}}{dw_{k,j}^{(\ell)}},$$

where η is a learning rate (some small constant, e.g., $\eta = 0.02$ or so).

One step of gradient descent on a complicated error surface

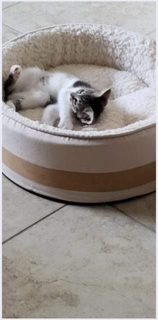


Error Metrics Summarized

- Use MSE to achieve $\hat{y} \rightarrow E\left[\vec{y}|\vec{x}\right]$. That's almost always what you want.
- For a binary classifier with a sigmoid output, BCE loss gives you the MSE result without the vanishing gradient problem.
- For a multi-class classifier with a softmax output, CE loss gives you the MSE result without the vanishing gradient problem.
- After you're done training, you can make your cell phone app more efficient by throwing away the uncertainty:
  - Replace softmax output nodes with max
  - Replace logistic output nodes with unit-step
  - Replace tanh output nodes with signum

Outline

Multimedia Inputs = Too Much Data



Does this image contain a cat?

Fully-connected solution:

$$\hat{y} = \sigma\left(W^{(2)}\vec{h}\right)$$
$$\vec{h} = \text{ReLU}\left(W^{(1)}\vec{x}\right)$$

where $\vec{x}$ contains all the pixels.

- Image size $2000 \times 3000 \times 3 = 18,000,000$ dimensions in $\vec{x}$.
- If $\vec{h}$ has 500 dimensions, then $W^{(1)}$ has $500 \times 18,000,000 = 9,000,000,000$ parameters.
- ...so we should use at least $9,000,000,000$ images to train it.

Shift Invariance

The cat has moved. The fully-connected network has no way to share information between the rows of $W^{(1)}$ that look at the center of the image, and the rows that look at the right-hand side.

How to achieve shift invariance: Convolution

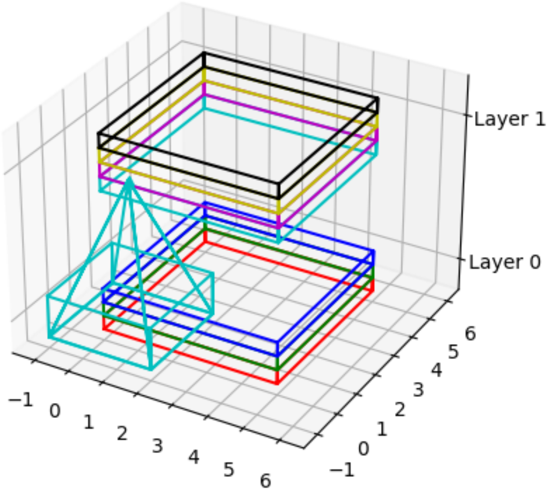
Instead of using vectors as layers, let's use images.

$$\xi^{(l)}[m,n,d] = \sum_c \sum_{m'} \sum_{n'} w^{(l)}[m',n',c,d] h^{(l-1)}[m-m',n-n',c]$$

where

- $\xi^{(l)}[m,n,c]$ and $h^{(l)}[m,n,c]$ are excitation and activation (respectively) of the $(m,n)^{\text{th}}$ pixel, in the c^{th} channel, in the l^{th} layer.
- $w^{(l)}[m,n,c,d]$ are weights connecting c^{th} input channel to d^{th} output channel, with a shift of m rows, n column.

How to achieve shift invariance: Convolution



How to use convolutions in a classifier

- The zero$^{\text{th}}$ layer is the input image, where $c \in \{0, 1, 2\}$ denotes color:

$$h^{(0)}[m, n, c] = x[m, n, c]$$

- Excitation and activation:

$$\xi^{(l)}[m, n, d] = \sum_c \sum_{m'} \sum_{n'} w[m', n', c, d] h^{(l-1)}[m - m', n - n', c]$$

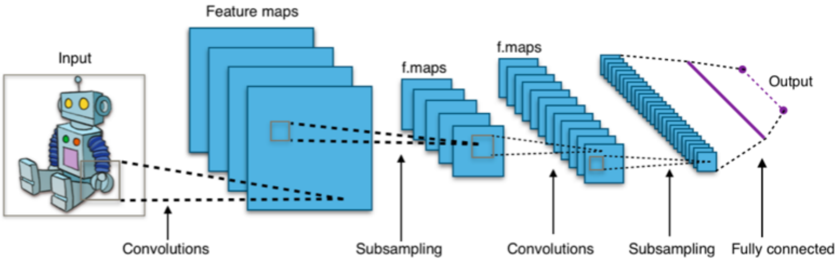
$$h^{(l)}[m, n, d] = \text{ReLU}\left(\xi^{(l)}[m, n, d]\right)$$

- Reshape the last convolutional layer into a vector, to form the first fully-connected layer:

$$h^{(L+1)}_{cN^2+mN+n} = h^{(L)}[m, n, c]$$

where N is the image dimension (both height and width).

How to use convolutions in a classifier



“Typical CNN,” by Aphex34 2015, CC-SA 4.0, `https://commons.wikimedia.org/wiki/File:Typical_cnn.png`

Outline

How to back-prop through a convolutional neural net

You already know how to back-prop through fully-connected layers. Now let's back-prop through convolution:

$$\frac{\partial \mathcal{L}}{\partial h^{(l-1)}[m',n',c]} = \sum_m \sum_n \sum_d \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]} \frac{\partial \xi^{(l)}[m,n,d]}{\partial h^{(l-1)}[m',n',c]}$$

The partial derivative is easy:

$$\xi^{(l)}[m,n,d] = \sum_c \sum_{m'} \sum_{n'} w^{(l)}[m-m',n-n',c,d] h^{(l-1)}[m',n',c]$$

$$\frac{\partial \xi^{(l)}[m,n,d]}{\partial h^{(l-1)}[m',n',c]} = w^{(l)}[m-m',n-n',c,d]$$

Putting all of the above equations together, we get:

$$\frac{\partial \mathcal{L}}{\partial h^{(l-1)}[m',n',c]} = \sum_m \sum_n \sum_d w^{(l)}[m-m',n-n',c,d] \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]}$$

Convolution forward, Correlation backward

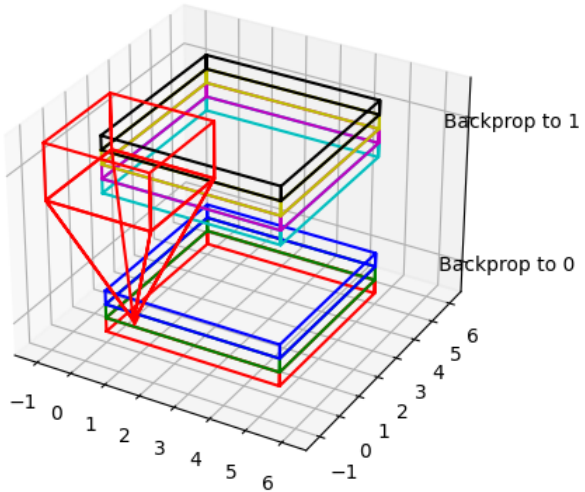
In signal processing, we defined $x[n] * h[n]$ to mean $\sum h[m]x[n-m]$. Let's use the same symbol to refer to this multi-channel 2D convolution:

$$\xi^{(l)}[m,n,d] = \sum_{c}\sum_{m'}\sum_{n'} w^{(l)}[m-m',n-n',c,d]h^{(l-1)}[m',n',c]$$
$$\equiv w^{(l)}[m,n,c,d] * h^{(l-1)}[m,n,c]$$

Back-prop, then, is also a kind of convolution, but with the filter flipped left-to-right and top-to-bottom. Flipped convolution is also known as "correlation."

$$\frac{\partial \mathcal{L}}{\partial h^{(l-1)}[m',n',c]} = \sum_{m}\sum_{n}\sum_{c} w^{(l)}[m-m',n-n',c,d]\frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]}$$
$$= w^{(l)}[-m',-n',c,d] * \frac{d\mathcal{L}}{d\xi^{(l)}[m',n',d]}$$

Back-prop through a convolutional layer



Similarities between convolutional and fully-connected back-prop

- In a fully-connected layer, forward-prop is a matrix multiply. Back-prop is multiplication by the transpose of the same matrix:

$$\vec{\xi}^{(l)} = W^{(l)}\vec{h}^{(l-1)}$$
$$\nabla_{\vec{h}^{(l-1)}}\mathcal{L} = \left(W^{(l)}\right)^T \nabla_{\vec{\xi}^{(l)}}\mathcal{L}$$

- In a convolutional layer, forward-prop is a convolution, Back-prop is a correlation:

$$\xi^{(l)}[m,n,d] = w^{(l)}[m,n,c,d] * h^{(l-1)}[m,n,c]$$
$$\frac{d\mathcal{L}}{dh^{(l-1)}[m,n,c]} = w^{(l)}[-m,-n,c,d] * \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]}$$

Convolutional layers: Weight gradient

Finally, we need to combine back-prop and forward-prop in order to find the weight gradient:

$$\frac{d\mathcal{L}}{dw^{(l)}[m',n',c,d]} = \sum_m \sum_n \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]} \frac{\partial \xi^{(l)}[m,n,d]}{\partial w^{(l)}[m',n',c,d]}$$

Again, the partial derivative is very easy to compute:

$$\xi^{(l)}[m,n,d] = \sum_c \sum_{m'} \sum_{n'} w^{(l)}[m',n',c,d] h^{(l-1)}[m-m',n-n',c]$$

$$\frac{\partial \xi^{(l)}[m,n,d]}{\partial w^{(l)}[m',n',c,d]} = h^{(l-1)}[m-m',n-n',c]$$

Convolutional layers: Weight gradient

$$\frac{\partial \mathcal{L}}{\partial w^{(l)}[m',n',c,d]} = \sum_m \sum_n \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]} \frac{\partial \xi^{(l)}[m,n,d]}{\partial w^{(l)}[m',n',c,d]}$$

$$\frac{\partial \xi^{(l)}[m,n,d]}{\partial w^{(l)}[m',n',c,d]} = h^{(l-1)}[m-m',n-n',c]$$

Putting those together, we discover that the weight gradient is a correlation:

$$\begin{aligned}\frac{d\mathcal{L}}{dw^{(l)}[m',n',c,d]} &= \sum_m \sum_n \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]} h^{(l-1)}[m-m',n-n',c] \\ &= \frac{d\mathcal{L}}{d\xi^{(l)}[m',n',d]} * h^{(l-1)}[-m',-n',c]\end{aligned}$$

Steps in training a CNN

1. Forward-prop:

$$\xi^{(l)}[m,n,d] = w^{(l)}[m,n,c,d] * h^{(l-1)}[m,n,c]$$

2. Back-prop:

$$\frac{\partial \mathcal{L}}{\partial h^{(l-1)}[m,n,c]} = w^{(l)}[-m,-n,c,d] * \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]}$$

3. Weight gradient:

$$\frac{d\mathcal{L}}{dw^{(l)}[m,n,c,d]} = \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]} * h^{(l-1)}[-m,-n,c]$$

Outline

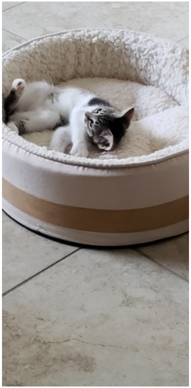
Features and PWL Functions

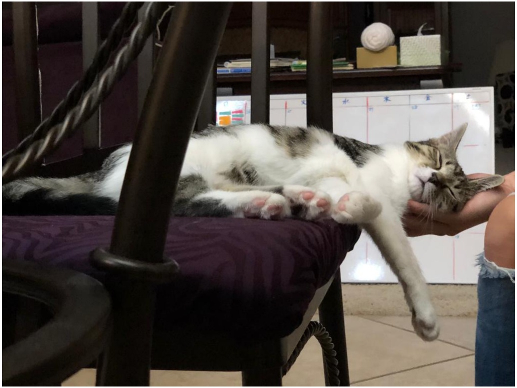
Remember the PWL model of a ReLU neural net:

1. The hidden layer activations are positive if some feature is detected in the input, and zero otherwise.
2. The rows of the output layer are vectors, scaled by the hidden layer activations, in order to approximate some desired piece-wise-linear (PWL) output function.
3. What happens next is different for regression and classification:
    1. Regression: The PWL output function is the desired output.
    2. Classification: The PWL function is squashed down to the $[0, 1]$ range using a sigmoid.

Features and PWL Functions

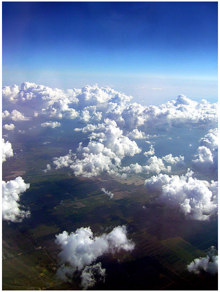
In image processing, often we don't care where in the image the "feature" occurs:





Features and PWL Functions

Sometimes we care **roughly** where the feature occurs, but not exactly. Blue at the bottom is sea, blue at the top is sky:



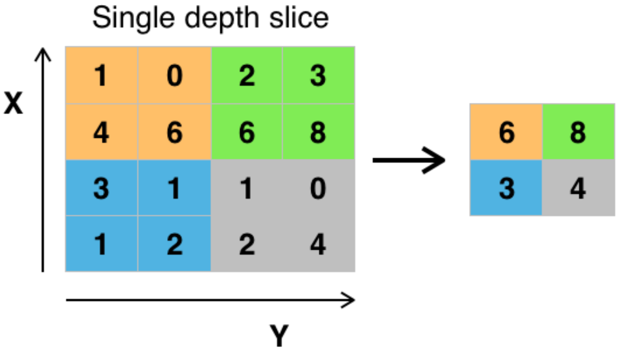
Max Pooling

- Philosophy: the activation $h^{(l)}[m,n,c]$ should be greater than zero if the corresponding feature is detected anywhere within the vicinity of pixel (m,n). In fact, let's look for the *best matching* input pixel.
- Equation:

$$h^{(l)}[m,n,c] = \max_{m'=0}^{M-1} \max_{n'=0}^{M-1} \text{ReLU}\left(\xi^{(l)}[mM+m',nM+n',c]\right)$$

where M is a max-pooling factor (often $M=2$, but not always).

Max Pooling



"max pooling with 2x2 filter and stride = 2," Aphex34, 2015, CC SA 4.0,

https://commons.wikimedia.org/wiki/File:Max_pooling.png

Back-Prop for Max Pooling

The back-prop is pretty easy to understand. The activation gradient, $\frac{d\mathcal{L}}{dh^{(l)}[m,n,c]}$, is back-propagated to just one of the excitation gradients in its pool: the one that had the maximum value.

$$\frac{d\mathcal{L}}{d\xi^{(l)}[mM+m', nM+n', c]} = \begin{cases} \frac{d\mathcal{L}}{dh^{(l)}[m,n,c]} & \begin{array}{l} m'=m^*,\ \ n'=n^*, \\ h^{(l)}[m,n,c]>0, \end{array} \\ 0 & \text{otherwise}, \end{cases}$$

where

$$m^*, n^* = \underset{m',n'}{\operatorname{argmax}}\, \xi^{(l)}[mM+m', nM+n', c],$$

Other types of pooling

- **Average pooling:**

$$h^{(l)}[m,n,c] = \frac{1}{M^2}\sum_{m'=0}^{M-1}\sum_{n'=0}^{M-1}\text{ReLU}\left(\xi^{(l)}[mM+m', nM+n', c]\right)$$

  Philosophy: instead of finding the pixels that best match the feature, find the average degree of match.

- **Decimation pooling:**

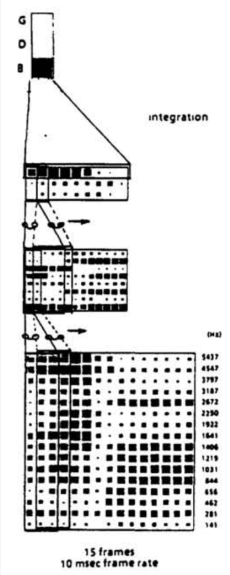
$$h^{(l)}[m,n,c] = \text{ReLU}\left(\xi^{(l)}[mM, nM, c]\right)$$

  Philosophy: the convolution has already done the averaging for you, so it's OK to just throw away the other M^2-1 inputs.

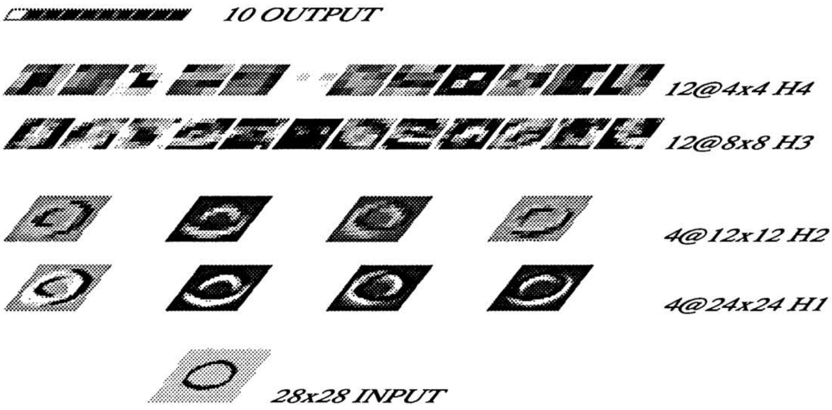
Outline

"Phone Recognition: Neural Networks vs. Hidden Markov Models," Waibel, Hanazawa, Hinton, Shikano and Lang, 1988

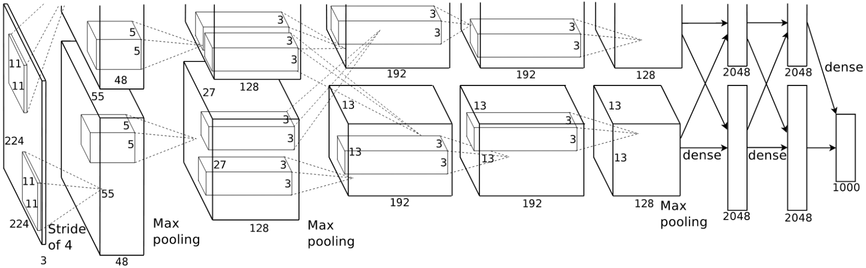
- 1D convolution
- average pooling
- max pooling invented by Yamaguchi et al., 1990, based on this architecture



"Backpropagation Applied to Handwritten Zip Code Recognition," LeCun, Boser, Denker & Henderson, 1989 (2D convolution, decimation pooling)



"Imagenet Classification with Deep Convolutional Neural Networks," Krizhevsky, Sutskever & Hinton, 2012 (GPU training)



Outline

Summary

- Convolutional layers: forward-prop is a convolution, back-prop is a correlation, weight gradient is a correlation.
- Max pooling: back-prop just propagates the derivative to the pixel that was chosen by forward-prop.
- Many-layer CNNs trained on GPUs, with small convolutions in each layer, have won Imagenet every year since 2012, and are now a component in every image, speech, audio, and video processing system.

Outline

Written Example

Suppose our input image is a delta function:

$$x[n] = \delta[n]$$

Suppose we have one convolutional layer, and the weights are initialized to be Gaussian:

$$w[n] = e^{-\frac{n^2}{2}}$$

Suppose that the neural net output is

$$\hat{y} = \sigma\left(\max\left(w[n] * x[n]\right)\right),$$

where $\sigma(\cdot)$ is the logistic sigmoid, and $\max(\cdot)$ is max-pooling over the entire output of the convolution. Suppose that the target output is $y = 1$, and we are using binary cross-entropy loss. What is $d\mathcal{L}/dw[n]$, as a function of n?