07-280 Computer Vision

Nihar B. Shah

An image comprises a grid of pixels. For now, consider grayscale images. Assume each pixel value is normalized to lie in $[0, 1]$.

Suppose you want to perform a classification task. You have access to labeled training data.

- What if you try logistic regression (with pixels as features)?
- What about a fully connected neural network?
  - Does not exploit spatial relations between pixels
  - Not spatially invariant
  - Needs too many parameters; high compute & data

Beneficial to have image-specific models. Two dominant approaches:

- **Convolutional Neural Networks (CNNs):** Less compute, work better under limited data. Will discuss it today.
- **Vision Transformers (ViTs):** Many more parameters, work better with large datasets. Will discuss transformers in the context of language later in this course.

There are also many hybrid approaches that combine ideas from CNNs & ViTs.

Also note that there is another paradigm of diffusion models that is used for image generation and will be covered in 07-380.

CNNs

You are given an image (pixels). You want to blur it. How do you do that?

Do "convolution" with a specific "filter".

Example: A 2×2 "blur" matrix:

$$\begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$

Example image:

$$\begin{bmatrix} 0.1 & 0.2 & 0.6 \\ 0.3 & 0.4 & 0.9 \\ 0.7 & 0.1 & 0 \end{bmatrix} \xrightarrow{\text{convolution with blur filter}} \begin{bmatrix} 0.25 & 0.525 \\ 0.375 & 0.35 \end{bmatrix}$$

For example, from the bottom left of the original image:

$$0.3 \times 0.25 + 0.4 \times 0.25 + 0.7 \times 0.25 + 0.1 \times 0.25 = 0.375$$

Classical Image Processing

- Many such filters, e.g., blur or edge detection etc.
- Designed by hand
- Convolved with image to extract features
- Then used for various downstream tasks

CNN approach: Learn filters using data. E.g., 2×2 filter:

$$\begin{bmatrix} \omega_{00} & \omega_{01} \\ \omega_{10} & \omega_{11} \end{bmatrix}$$

These are parameters which we will learn. Filters are also called "kernels."

Convolution Formula

For a $k_1 \times k_2$ kernel:

$$z[i,j] = \sum_{u=0}^{k_1-1} \sum_{v=0}^{k_2-1} x[i+u,\, j+v] \cdot \omega_{u,v}$$

where z is the output and x is the input.

Multi-Channel Convolution

If the image has C channels (grayscale has $C = 1$):

$$z[i,j] = \sum_{u=0}^{k_1-1} \sum_{v=0}^{k_2-1} \sum_{c=0}^{C-1} x[c,\, i+u,\, j+v] \cdot \omega_{c,u,v}$$

Typical CNN pipeline:

There is not just one but several convolutions done in parallel. Each of these has kernels defined by a different set of parameters. Note that one also adds a bias parameter. These produce multiple **feature maps**.

This multitude of feature maps leads to a problem: too much memory/compute needed. So we downsample them in what is called **pooling**. Two common pooling methods are **average pooling** and **max pooling**. Pooling is applied to each feature map separately.

Conv $\to$ Pool $\to$ Nonlinearity (ReLU) $\to$ Conv $\to$ Pool $\to$ ReLU $\to \cdots \to$ Fully Connected $\to$ Softmax

Properties of Convolutional Layers

- Just like fully connected layers, weights and biases are the parameters, and they apply linearly to the outputs of the previous layer.
- Parameters are shared across sliding window locations.
- Unlike fully connected, nodes in a layer are only connected to a small number of nodes in the previous layer.

- Training is done via backprop (more on training later).
- Each layer takes features from previous layer and combines them.
- Empirically observed to learn interesting representations on its own – see Figure 1.

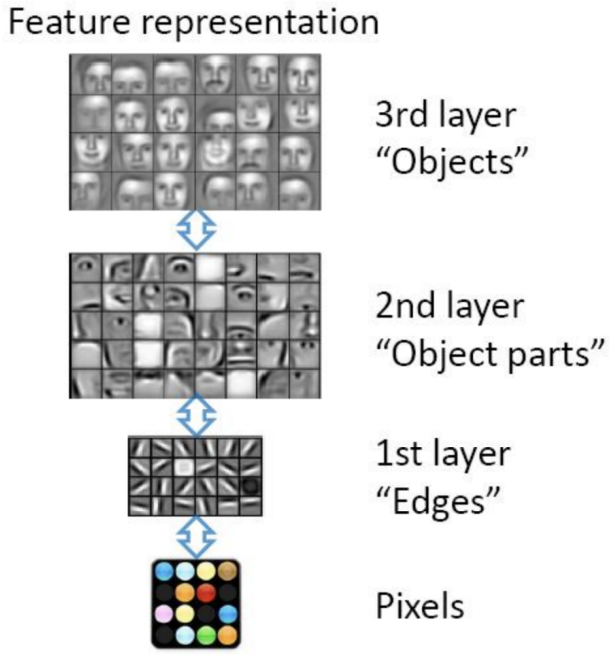


Figure 1: Credit: Honglak Lee `https://deeplearningworkshopnips2010.wordpress.com/wp-content/uploads/2010/09/nips10-workshop-tutorial-final.pdf`

AlexNet

The ImageNet competition was a landmark competition due to its (then) sheer data size and challenge. Until ImageNet, there was far less emphasis on collecting gigantic datasets.

In 2012, AlexNet was proposed which achieved a $\approx 15\%$ error rate. This was way better than the next best $\approx 26\%$ error rate. AlexNet was a pivotal moment in computer vision and deep learning, and this hugely revolutionized computer vision to use deep learning approaches.

Key Contributions

- **ReLU activations:** Previous works used tanh or sigmoid which were more expensive to compute and their gradients weren't well behaved.
- **Training on GPUs:** more on this later.
- **Using deeper architectures.**
- **Data augmentation:** Artificially increase size of their training data. From the original 256×256 images, choose several 224×224 patches. Use this enlarged dataset.

- **Dropout use:** To mitigate overfitting, during each training gradient step, randomly drop a subset of edges.

Graphics Processing Units (GPUs)

A GPU has a large number of simpler cores designed for doing the same simple operation on different data simultaneously. E.g., a matrix multiplication task is split across cores.

Training a neural network often uses multiple GPUs in parallel. E.g., suppose you have two GPUs and 32,768 training datapoints. In each epoch, both GPUs have current model weights. Then:

1. CPU shuffles the datapoints.
2. GPU 0 gets data $[0:32]$, GPU 1 gets data $[32:64]$.
3. Each computes the gradient.
4. Average gradients; both GPUs update weights.
5. GPU 0 gets data $[64:96]$, GPU 1 gets data $[96:128]$.
6. Continue until all data is processed.

ResNets (Residual Networks)

Initially proposed for CNNs but now used much more broadly.

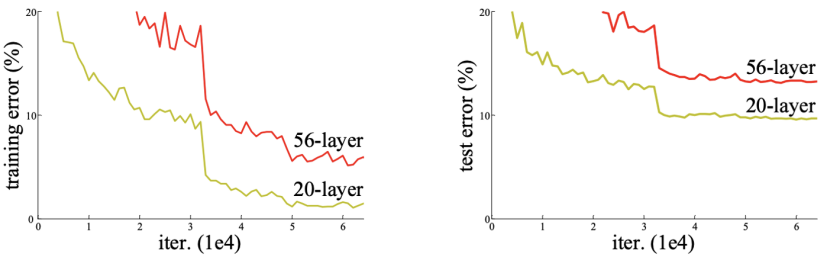
Let us first understand the problem they solve. Consider two neural networks: a 20-layer network and a 56-layer network. The 20-layer network is a "special case" of the 56-layer network, that is: for every value of parameters (ω, b) for the 20-layer network, there is a value of the parameter set (ω', b') of the 56-layer network such that

$$h_{\omega,b}(x) = h'_{\omega',b'}(x) \quad \forall \text{ inputs } x.$$

where $h_{\omega,b}$ denotes the 20-layer NN and $h'_{\omega',b'}$ denotes the 56-layer NN.

You train both networks using the same training data via ERM. Which network (20-layer or 56-layer) would you guess has lower training error?

Since the 20-layer NN is a special case of the 56-layer NN, it is reasonable to expect that the 56-layer NN will fit the training data more and have a lower training error. However:



*Surprisingly, it was observed that the 56-layer network has **higher** training error than the 20-layer network! The test error of the 20-layer NN was also lower than that of the 56-layer NN.*

What is going on?

$$\text{input} \to \boxed{\text{layer } 1} \to \boxed{\text{layer } 2} \to \cdots \to \boxed{\text{layer } L} \to \text{output}$$

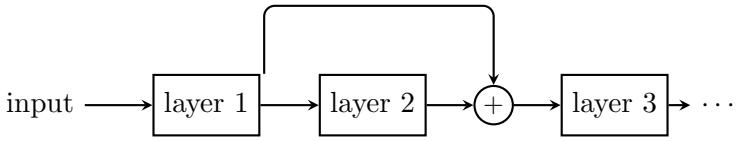
Gradient on the left part of the network is obtained from a product of gradients on the right. If the right gradients are small (or large), this gradient becomes vanishingly small (or explosively large) as the number of layers increases.

Therefore, weights on the left hardly change during training.

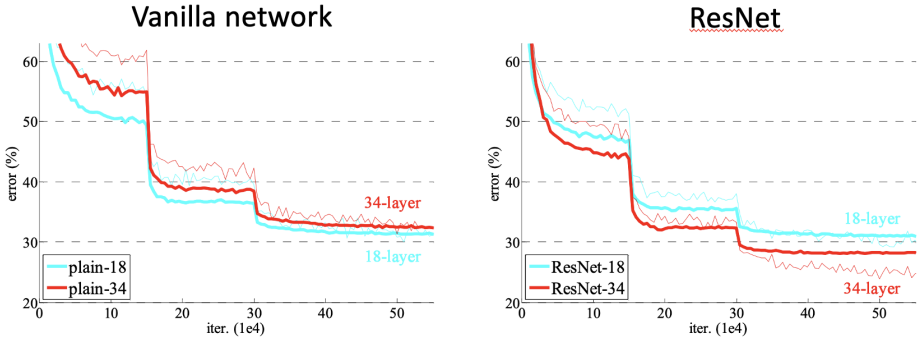
Skip Connections

Observation: Easier to "learn" an all-zero set of weights than an identity mapping.

Idea: Explicitly facilitate identity by adding "skip" connections. The following image shows a simplified diagram of a skip connection where the input to a certain layer is added explicitly to its output.



Training and test error of the deeper network are now lower.



Thin line: Training error
Thick line: Validation error

Batch Normalization

Another idea that helps train deeper networks better.

Imagine you are layer 50. Your job is to learn some transformation of the inputs to you. You think you have this figured out. But then the weights of the previous 49 layers change. Due to this the distribution of inputs to you changes, messing up everything! This is called **internal covariate shift**.

Idea (BatchNorm): Consider a minibatch. After any conv layer, compute the mean & variance of the output across that minibatch. Insert a scale and shift to normalize the mean and variance.

Additional observed benefits:

- Helps smoothen the loss landscape (i.e., gradients become more consistent in direction) allowing larger learning rates without diverging.
- It acts as regularization as the noise due to estimating mean and variance from the minibatch (rather than the full batch) injects slight randomness (like dropout).
- Finally, it reduces sensitivity to how the parameters were initialized.