Lecture 3 U-Net

Shiwei Lan[1]

[1]School of Mathematical and Statistical Sciences
Arizona State University

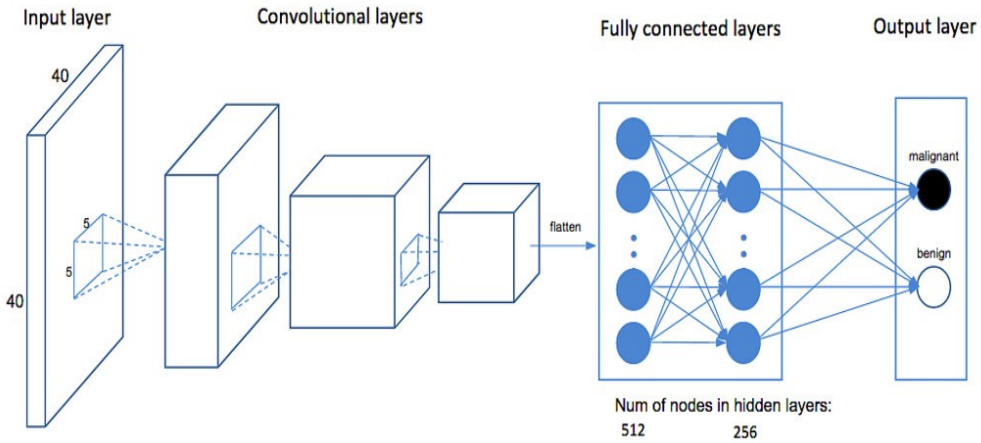
STP598 Advanced Deep Learning Models
Spring 2026

Table of Contents

Convolutional Neural Networks

- Convolutional Neural Networks (CNN) are a specialized kind of neural network for processing data that has a known, grid-like topology, e.g. time-series, images, etc.
- CNNs are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.
- Instead of using all input features to create the linear combination, a "convolutional layer" builds neurons that each takes a subset (a local region) of the input features.
- This is motivated by the fact that biologically, the neurons only take signals from neighboring neurons.

Convolutional Neural Networks



See this hand digit writing recognition example, and this interesting application by Tesla.

Convolution

- Suppose we track the location of a spaceship in real time $x(t)$.
- The sensor is noisy and we need to estimate the position based on noisy measurements.
- More recent measurements are more relevant so we use a weighting function $w(a)$ to emphasize that.
- A smoothed estimate of the position is given by the following *convolution*

$$s(t) = (x * w)(t) := \int x(a)w(t-a)da \tag{1}$$

  where we require $w(a) = 0$ for $a < 0$.
- In the CNN terminology, we have:
  - **input** x
  - **kernel** w
  - **feature map** s

Convolution

- In practice, we observe noisy measure at discrete time points so we have the *discrete convolution*

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2}$$

- In two-dimensional case, e.g. with image I as input, we define the convolution with a 2-d kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3}$$

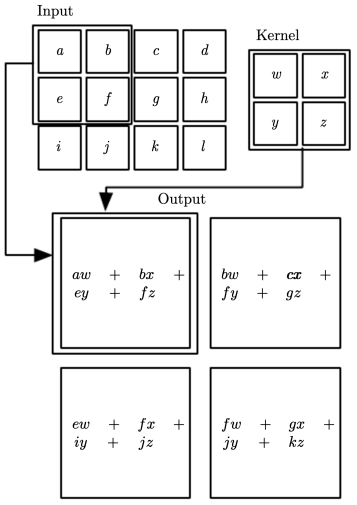
- Or equivalently (convolution is commutative),

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{4}$$

- Many neural network libraries implement *cross-correlation*

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{5}$$

Convolution

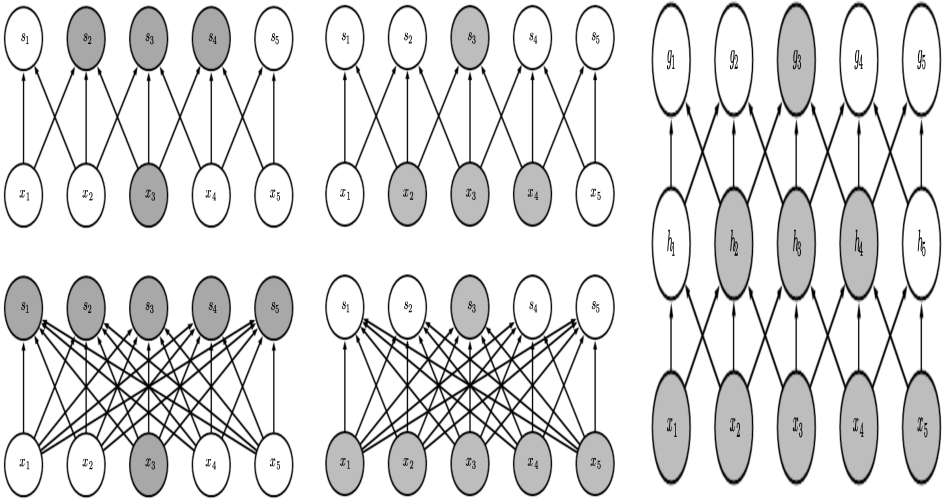


$$T = \begin{bmatrix} w_0 & 0 & 0 & 0 & \cdot & 0 \\ w_1 & w_0 & 0 & 0 & \cdot & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ w_s & w_{s-1} & \cdots & w_0 & 0\cdots & 0 \\ 0 & w_s & \cdots & w_1 & w_0\cdots & 0 \\ \vdots & \ddots & \ddots\ddots & \ddots & \ddots\ddots & \vdots \\ \cdots & \cdots & 0 & w_s & \cdots & w_0 \\ \cdots & \cdots & \cdots & 0 & w_s\cdots & w_1 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots\cdots & \cdots 0 & w_s & w_{s-1} \\ 0 & \cdots & \cdots & \cdots & \cdots 0 & w_s \end{bmatrix}.$$

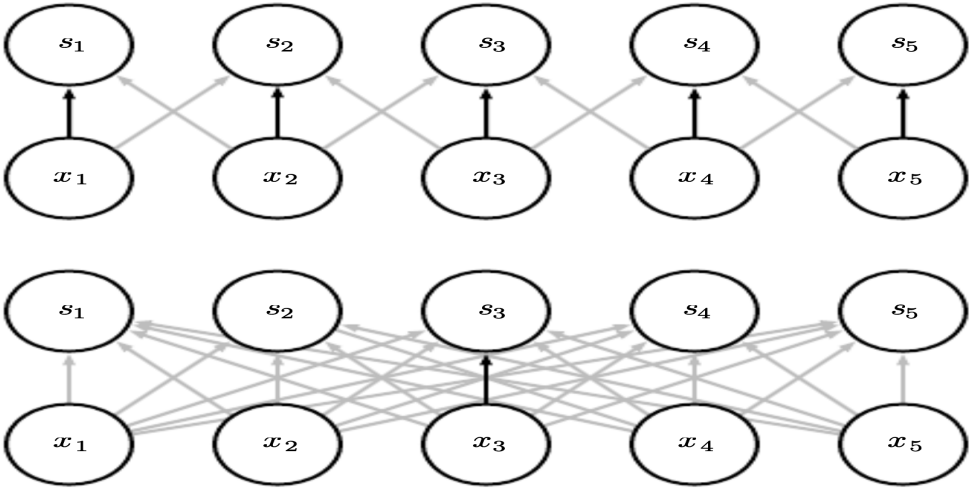
Why convolution?

- Convolution leverages three important ideas that can help improve a machine learning system:
  - **sparse interactions**,
  - **parameter sharing** and
  - **equivariant representations**.
- Sparse connectivity (weights): if there are m inputs and n outputs, then matrix multiplication based algorithms have $\mathcal{O}(mn)$ runtime (per sample). If we limit the number of connections each output may have to k, the runtime can be reduced to $\mathcal{O}(kn)$ with $k \ll m$.
- A function $f(x)$ is *equivariant* to a a function g if $f(g(x)) = g(f(x))$. Convolution with particular parameter sharing makes the layer equivariant to translation.

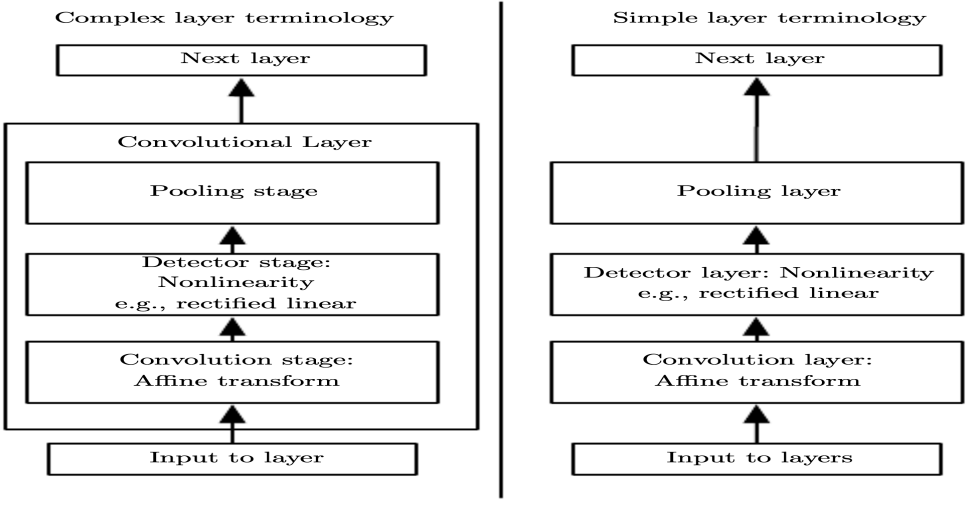
Sparse connectivity



Parameter sharing



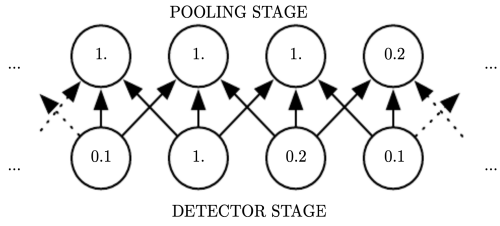
Components of CNN

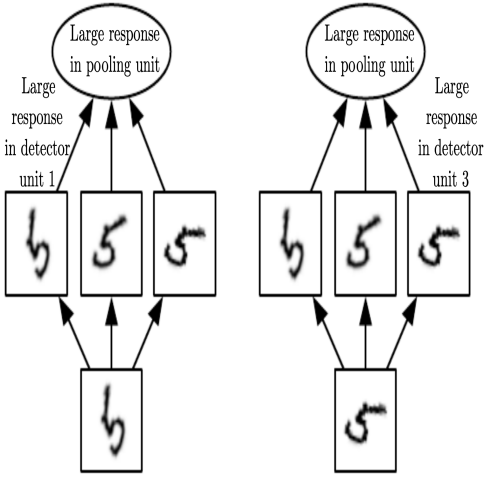


Pooling

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.
- For example, the max pooling (Zhou and Chellappa, 1988) operation reports the maximum output within a rectangular neighborhood. Others include the average, or L^2 norm of a rectangular neighborhood.
- Pooling helps to make the representation become approximately **invariant** to small translations of the input.
- *Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.*

Pooling


POOLING STAGE
1.
1.
1.
0.2
0.1
1.
0.2
0.1
DETECTOR STAGE
POOLING STAGE
0.3
1.
1.
1.
0.3
0.1
1.
0.2
DETECTOR STAGE


Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

Pooling

- The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations.
- Because pooling summarizes the responses over a whole neighborhood, it can be used with *downsampling* by reporting every k pixels apart.
- For many tasks, pooling is essential for handling inputs of varying size. The final pooling layer may be defined to output fixed sets of summary statistics, regardless of the input (image) size.
- Theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau et al., 2010)

Variants of Basic Convolution

- CNN consists of many applications of convolution in parallel.
- The input is a grid of vector-valued observations. For example, a color image has a red, green and blue intensity at each pixel, represented as a 3-D tensor.
- Assume we have:
  - a 4-D kernel tensor $\mathbf{K}$ with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output and the input;
  - input data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k;
  - output data $\mathbf{Z}$ with the same format of $\mathbf{V}$.
- Then we have $\mathbf{Z}$ produced by convolving $\mathbf{K}$ across $\mathbf{V}$:

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n} \tag{6}$$

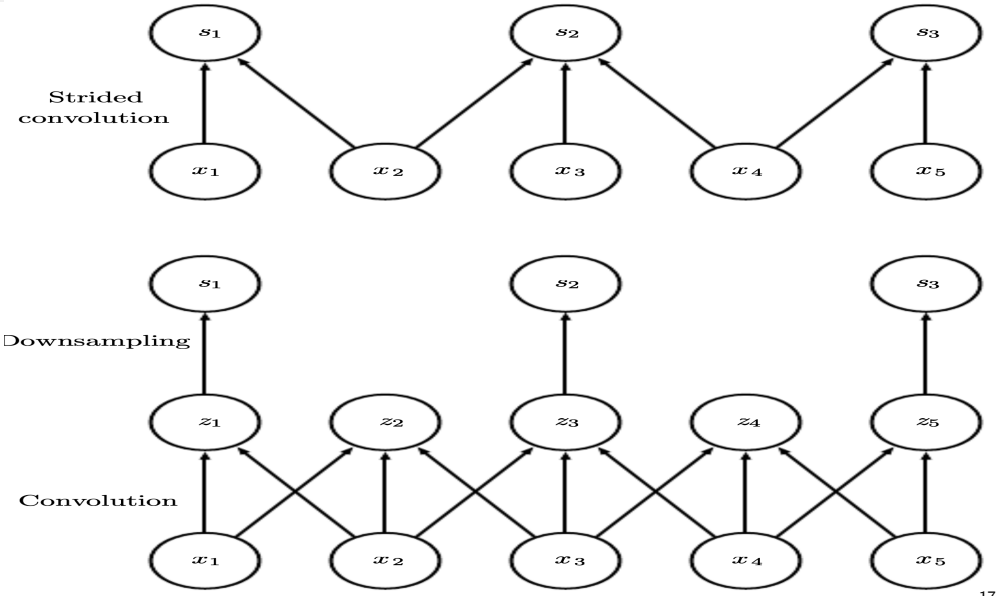
Stride

- We may want to skip over some positions of the kernel in order to reduce the computational cost.
- We can think of this as downsampling the output of the full convolution function.
- If we want to sample only every s pixels in each direction in the output, then we can define a downsampled convolution function c such that

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}] \tag{7}$$

- We refer to s as the **stride** of this downsampled convolution.

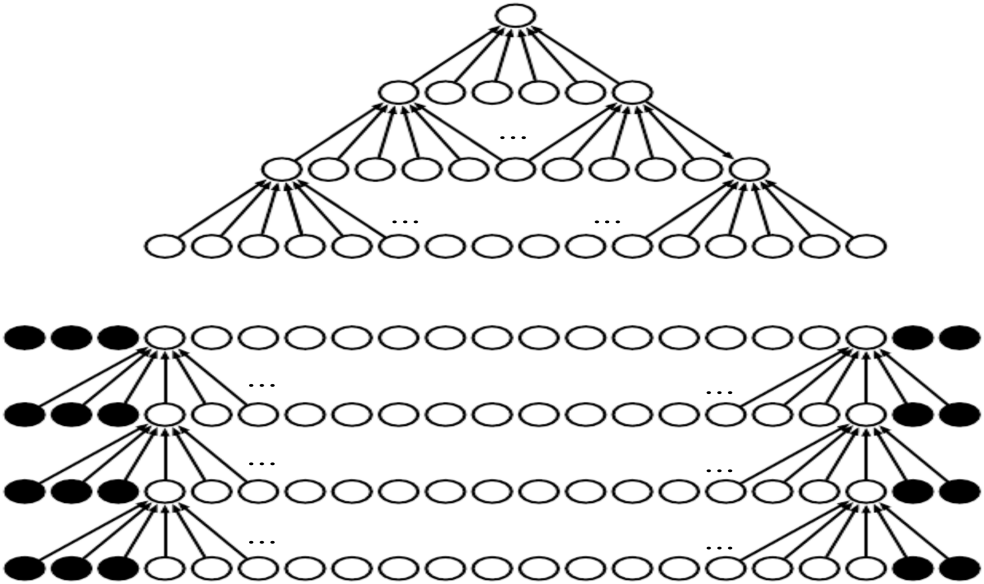
Stride


Strided convolution
s_1
s_2
s_3
x_1
x_2
x_3
x_4
x_5
Downsampling
Convolution
s_1
s_2
s_3
z_1
z_2
z_3
z_4
z_5
x_1
x_2
x_3
x_4
x_5

Zero padding

- One essential feature of CNN implementation is the ability to implicitly zero-pad the input $\mathbf{V}$ in order to make it wider.
- Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer.
- If the input image has width m and the kernel has width k, we have (in MATLAB terminology)
  - **valid** convolution: output will be of width $m - k + 1$
  - **same** convolution: output has the same size of input
  - **full** convolution: enough zeroes are added for every pixel to be visited k times in each direction, resulting in an output image of width $m + k - 1$.
- Usually the optimal amount of zero padding (in terms of test set classification accuracy) lies somewhere between "valid" and "same" convolution.

Zero padding



Quick quizzes

A grayscale image of size $64 \times 64 \times 1$ is processed by the following sequence of layers:

- Convolution layer: number of filters=16, filter size=3×3, stride=1, padding='valid'.
- Max-pooling layer: pool size=2×2, stride=2

Then

- What is the output size of the convolution layer?

Quick quizzes

A grayscale image of size $64 \times 64 \times 1$ is processed by the following sequence of layers:

- Convolution layer: number of filters=16, filter size=3×3, stride=1, padding='valid'.
- Max-pooling layer: pool size=2×2, stride=2

Then

- What is the output size of the convolution layer?
  - $out\_size = \lfloor \frac{in\_size - kernel\_size}{stride} \rfloor + 1$, $(64 - 3)/1 + 1 = 62$.
  - answer: $62 \times 62 \times 16$

Quick quizzes

A grayscale image of size $64 \times 64 \times 1$ is processed by the following sequence of layers:

- Convolution layer: number of filters=16, filter size=3×3, stride=1, padding='valid'.
- Max-pooling layer: pool size=2×2, stride=2

Then

- What is the output size of the convolution layer?
  - $out\_size = \lfloor \frac{in\_size - kernel\_size}{stride} \rfloor + 1$, $(64 - 3)/1 + 1 = 62$.
  - answer: $62 \times 62 \times 16$
- What is the output size of the max-pooling layer?

Quick quizzes

A grayscale image of size $64 \times 64 \times 1$ is processed by the following sequence of layers:

- Convolution layer: number of filters=16, filter size=3×3, stride=1, padding='valid'.
- Max-pooling layer: pool size=2×2, stride=2

Then

- What is the output size of the convolution layer?
  - $out\_size = \lfloor \frac{in\_size - kernel\_size}{stride} \rfloor + 1$, $(64 - 3)/1 + 1 = 62$.
  - answer: $62 \times 62 \times 16$
- What is the output size of the max-pooling layer?
  - $out\_size = \lfloor \frac{in\_size - pool\_size}{stride} \rfloor + 1$, $(62 - 2)/2 + 1 = 31$.
  - answer: $31 \times 31 \times 16$

PyTorch Implementation

- In PyTorch, we build CNN with torch.nn.Conv2d for convolution and torch.nn.MaxPool2d for pooling.
- torch.nn.Conv2d has several arguments:
  - in_channels: number of channels in the input image
  - out_channels: number of channels produced by the convolution
  - kernel_size: size of the convolving kernel
  - stride: stride of the convolution
  - padding: padding added to all four sides of the input
  - padding_mode: 'zeros', 'reflect', 'replicate' or 'circular'
  - bias: whether to add a learnable bias to the output
- torch.nn.MaxPool2d has several arguments:
  - kernel_size: the size of the window to take a max over
  - stride: the stride of the window
  - padding: implicit zero padding to be added on both sides

Table of Contents

AutoEncoder

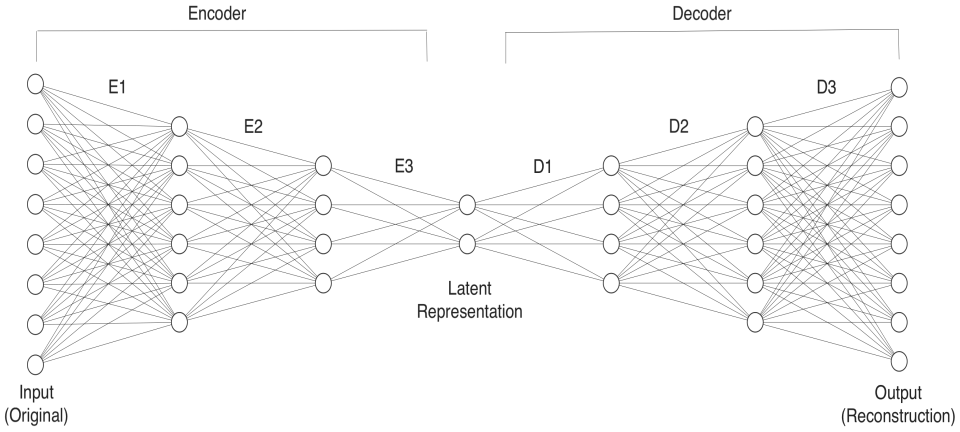


Figure: A typical architecture of autoencoder (AE) neural network.

AutoEncoders

- An autoEncoder (AE) is a neural network that is trained to attempt to copy its input to its output.
- The network consists of two parts:
  1. **encoder**: $f : x \mapsto h$
  2. **decoder**: $g : h \mapsto r$
- The network is trrained to approximately recover (copy) x, i.e. "$r \approx x$".
- The goal of AE is not to perfectly copy, but rather to learn useful (latent) properties of the data!

AutoEncoders

- If the hidden (latent) dimension is smaller than the input dimension, then the AE is called *undercomplete*; otherwise, it is called *overcomplete*.
- The learning process involves minimizing a loss function as follows

$$\min L(x, g(f(x))) \tag{8}$$

  where L is the loss penalizing $g(f(x))$ deviating from x, e.g. mean squared error.
- When the decoder is linear and L is the mean squared error, an undercomplete AE is equivalent to PCA (latent space is spanned by principal directions).
- When f, g are allowed to be nonlinear without constraint, the latent space can be meaningless.

Regularized AutoEncoders

- **Sparse AE** adds sparsity penalty $\Omega(h)$ to the loss: $L(x, g(f(x))) + \Omega(h)$.
- Typical choice of $\Omega(h)$ could be from Laplace prior $p(h; \lambda) = \lambda/2 \exp(-\lambda|h|)$: $\Omega(h) = -\log p(h; \lambda) = \lambda \sum_i h_i$.
- **Denoising AE** minimizes $L(x, g(f(\tilde{x})))$ with $\tilde{x}$ being a copy of x corrupted by some noise and tries to undo such corruption.
- **Contractive AE** regularizes AE with a penalty on the gradients of decoder to learn the distribution of training data:

$$L(x, g(f(x))) + \Omega(h, x), \quad \Omega(h, x) = \lambda \sum_i \|\nabla_x h_i\|^2. \tag{9}$$

Variational AutoEncoder (VAE)

- **Variational AutoEncoder (VAE)** (Kingma and Welling 2014) is probabilistic model for variational Bayesian inference.
- The goal is to approximate posterior distribution $p_\theta(z|x)$ with $q_\Phi(z|x)$ (part of VAE) by minimizing evidence lower bound (ELBO) loss (variation of Kullback–Leibler divergence).
- It reduces to construct a *probabilistic encoder* $q_\Phi(z|x)$ and a *probabilistic decoder* $p_\theta(x|z)$.

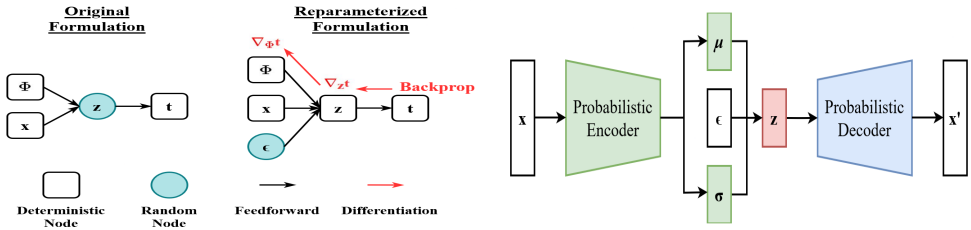


Table of Contents

CNN for Imaging Classification

- Deep CNN (Krizhevsky, A., et al, NIPS 2012) demonstrates strong capability of visual recognition.
- Besides the success of the image classification (single-class label), biomedical images often requires localization (pixel-level classification).
- Ciresan et al. (NIPS 2012) trained a network in a sliding-window setup to predict the class label of each pixel
  - It is quite slow because the network runs separately for each local region (patch).
  - There is a trade-off between localization accuracy and the use of context.
- U-Net (Ronneberger et al 2015) builds up on "fully convolutional network" (Long et al, 2014) to supplement a contracting network by successive layers, and to replace pooling with upsampling.

U-Net

- U-Net is a convolutional neural network (CNN) architecture designed primarily for *image segmentation*, especially in biomedical imaging (e.g., segmenting organs, tumors, cells).
- U-Net takes an image as input and outputs a pixel-wise classification map, meaning every pixel is labeled (e.g., background vs. tumor).
- It's called "U-Net" because its architecture looks like a U-shape:
  - **Contracting path (Encoder)**: Captures context and high-level features
  - **Expanding path (Decoder)**: Combines coarse semantic information with fine details
  - **Skip connections (key idea)**: Preserve fine-grained spatial details (precise localization)
- U-Net won the Grand Challenge for Computer-Automated Detection of Caries in Bitewing Radiography and Cell Tracking Challenge at ISBI 2015.

U-Net Architecture

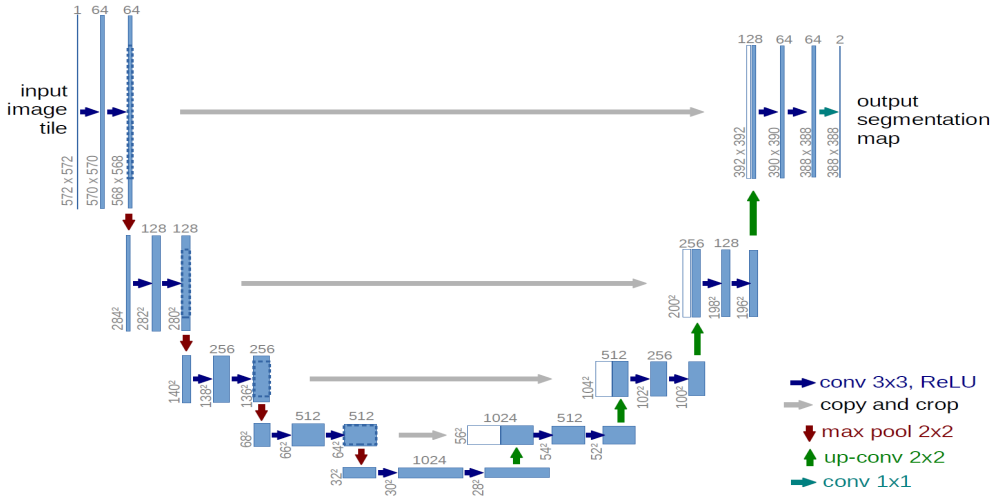


Figure: U-netarchitecture.

Training U-Net

- The energy function is computed by a pixel-wise soft-max over the final feature map combined with the cross entropy loss function.
- The soft-max is $p_k(x) = \exp(a_k(x))/\sum_{k'=1}^{K}\exp(a_{k'}(x))$ where $a_k(x)$ denotes the activation in feature channel k at the pixel position $x \in \Omega$.
- The cross-entropy penalizes at each position the deviation of $p_{\ell(x)}(x)$ from 1

$$E = \sum_{x\in\Omega} w(x)\log(p_{\ell(x)}(x))$$

where $\ell : \Omega \to 1, \cdots, K$ is the true label and $w : \Omega \to \mathbb{R}$ is a weight map.

- The weight map is precomputed for truth segmentation in the training data

$$w(x) = w_c(x) + w_0 \exp\left(-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}\right)$$

where $w_c : \Omega \to \mathbb{R}$ is the weight map to balance the class frequencies, $d_i : \Omega \to \mathbb{R}$ is the distance to the border of the i-th nearest cell.

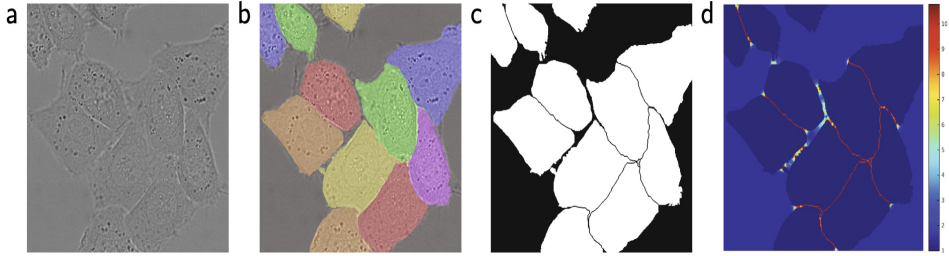


Fig. 3. HeLa cells on glass recorded with DIC (differential interference contrast) microscopy. (**a**) raw image. (**b**) overlay with ground truth segmentation. Different colors indicate different instances of the HeLa cells. (**c**) generated segmentation mask (white: foreground, black: background). (**d**) map with a pixel-wise loss weight to force the network to learn the border pixels.

Numerical Results

Table 1. Ranking on the EM segmentation challenge [14] (march 6th, 2015), sorted by warping error.

Rank	Group name	Warping Error	Rand Error	Pixel Error
	** human values **	0.000005	0.0021	0.0010
1.	u-net	**0.000353**	0.0382	0.0611
2.	DIVE-SCI	0.000355	0.0305	0.0584
3.	IDSIA [2]	0.000420	0.0504	0.0613
4.	DIVE	0.000430	0.0545	**0.0582**
⋮				
10.	IDSIA-SCI	0.000653	**0.0189**	0.1027

U-Net Implementation

- Here is a pre-trained U-Net implemented in PyTorch: https://pytorch.org/hub/mateuszbuda_brain-segmentation-pytorch_unet/
- Check here for another demo of U-Net: https://pyimagesearch.com/2021/11/08/u-net-training-image-segmentation-models-in-pytorch/
- Customized implementation of the U-Net in PyTorch for Kaggle's Carvana Image Masking Challenge from high definition images https://github.com/milesial/Pytorch-UNet
- We will train U-Net on `sol`.