REPETITION LECTURE

INF5860 — Machine Learning for Image Analysis

Ole-Johan Skrede

05.06.2018

University of Oslo

Outline

- Dense neural networks
- CNN architectures
- Object detection and image segmentation
- Unsupervised learning
- Generative adversarial networks

DENSE NEURAL NETWORKS

Supervised learning

- Given a training set with input x and desired output y

$$\Omega_{\text{train}} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$$

- Create a function f that “approximates” this mapping

$$f(x) \approx y, \quad \forall (x, y) \in \Omega_{\text{train}}$$

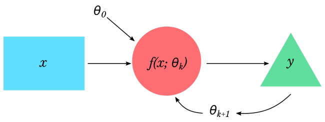
- Hope that this generalises well to unseen examples, such that

$$f(x) = \hat{y} \approx y, \quad \forall (x, y) \in \Omega_{\text{test}}$$

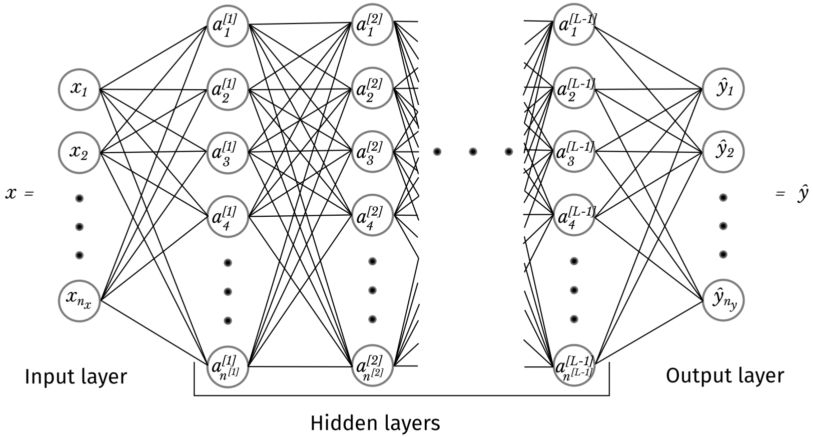
  where Ω_{test} is a set of relevant unseen examples.
- Hope that this is also true for all unseen relevant examples.

Deep learning

1. Build a function f that maps input to output
    - Input: Array of numbers.
    - Output: Probability mass function conditional on observed input.
2. This function will have multiple layers, where each layer is a representation of the previous.
3. Measure how well the output of the function is approximating the true output
4. Use information from the error to update the function
5. Repeat step 3 and 4 with multiple training examples

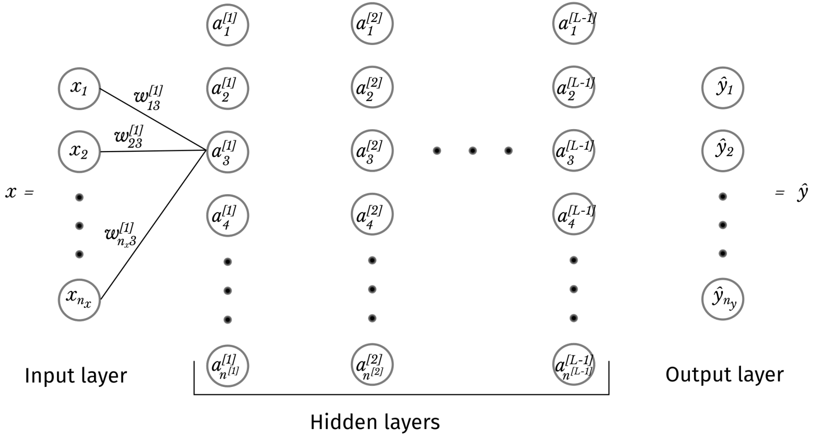


All connections



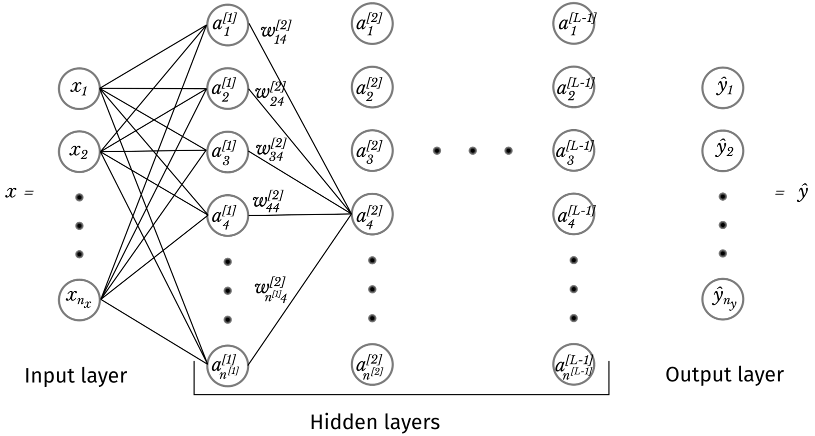
$$a_k^{[l]} = g\left(\sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}\right), \quad k \in \{1, 2, \ldots, n^{[l]}\}, \quad l \in \{1, 2, \ldots, L\}$$

Activation in node 3 of layer 1



$$a_3^{[1]} = g\left(\sum_{j=1}^{n_x} w_{j3}^{[1]} x_j + b_3^{[1]}\right)$$

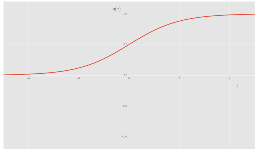
Activation in node 4 of layer 2



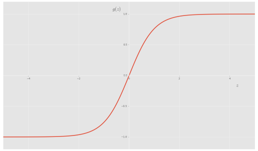
$$a_4^{[2]} = g\left(\sum_{j=1}^{n^{[1]}} w_{j4}^{[2]} a_j^{[1]} + b_4^{[2]}\right)$$

Activation functions

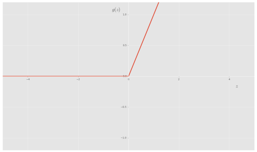
- Functions that introduce non-linearity to our network
- Without it, our network just becomes a linear mapping from input to output
- Enables DNN to become *universal function approximators*
- Can in theory be any function that is
  - Non-linear
  - Differentiable (if you are using a gradient-based optimization)



(a) Sigmoid



(b) Hyperbolic tangent



(c) Rectified linear unit (ReLU)

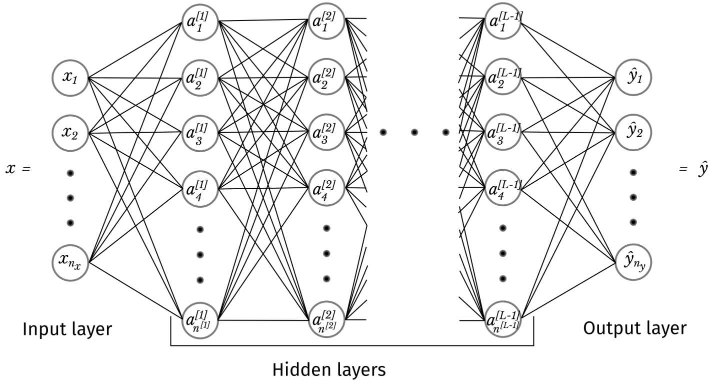
Output layer

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]}$$

$$\begin{aligned} a_k^{[L]} &= s(z_k^{[L]}) \\ &= \hat{y}_k \end{aligned}$$

for

$$\begin{aligned} k &= 1, \ldots, n_y, \\ &= 1, \ldots, n^{[L]}. \end{aligned}$$



Softmax function

$$s(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{n} e^{z_i}}$$

- $\sum_k s(z)_k = 1$, and the softmax can be interpreted as a probability
- Using the softmax as our final activation, we can interpret the output of our network as

$$f(x;\theta)_k = \Pr(Y = k | X = x; \theta) \tag{1}$$

- X is a random vector modeling our input
- Y is a categorical random variable modeling the true output
- θ is the collection of parameters

$$\theta = \{w_{jk}^{[l]}, b_k^{[l]}\}$$

for

$$\begin{cases} j & = 1, \ldots, n^{[l-1]} \\ k & = 1, \ldots, n^{[l]} \\ l & = 1, \ldots, L \end{cases}$$

Choosing network parameters

- The neural network is set up, the next step is to determine the values of the parameters

$$\theta = \{w_{jk}^{[l]}, b_k^{[l]} : j \in \left\{1, \dots, n^{[l-1]}\right\}, k \in \{1, \dots, n^{[l]}\}, l \in \{1, \dots, L\}\right\}$$

- This is done by defining a cost function which is to be minimized by some optimization method.
- In the most common case, we minimize the *cross entropy* cost function using a *stochastic gradient descent* optimizer

Cross entropy cost function

- We can derive the cross entropy cost function from maximum likelihood
- The maximum likelihood estimator (MLE) $\hat{\theta}$ of θ is the parameter

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta; x) \tag{2}$$

- The *likelihood* $\ell(\theta; x)$ is our network $p_Y(y|X = x; \theta)$, when x is a fixed realization of X, and θ is a variable.
- We then showed in the lecture that the maximum likelihood estimator is found by minimizing

$$J(\theta, \Omega_{\text{train}}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}. \tag{3}$$

- $\hat{y}$ is the network output vector, $\tilde{y}$ is the one-hot encoded reference. m is the number of training examples, and n_y is the number of classes.
- In practice, we approximate this by minimizing over a mini-batch $\Omega_{\text{train}}^{mb} \subset \Omega_{\text{train}}$

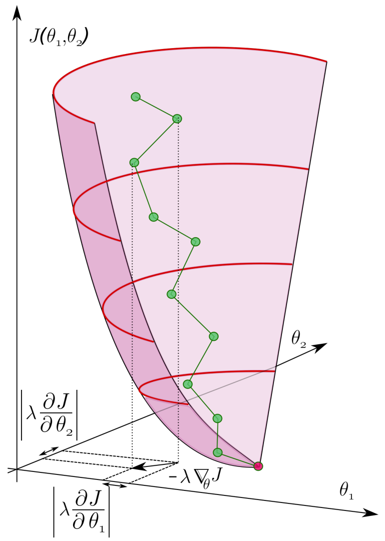
Gradient descent

$$\theta \leftarrow \theta - \lambda \nabla_\theta J(\theta) \tag{4}$$

- The gradient of J w.r.t. a set of variables $\theta = [\theta_1, \ldots, \theta_m]$

$$\nabla_\theta J = \left[\frac{\partial J}{\partial \theta_1}, \ldots, \frac{\partial J}{\partial \theta_m}\right]$$

- $\nabla_\theta J(\theta_k)$ gives the direction of steepest ascent at the point θ_k
- λ determines how long to move in that direction



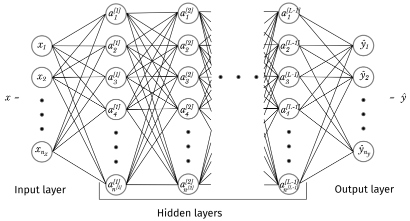
$J(\theta_1,\theta_2)$
θ_2
θ_1
$\left|\lambda \frac{\partial J}{\partial \theta_2}\right|$
$-\lambda \nabla_\theta J$
$\left|\lambda \frac{\partial J}{\partial \theta_1}\right|$

Backpropagation

- We need to compute the values of

$$\frac{\partial J}{\partial w_{jk}^{[l]}} \text{ and } \frac{\partial J}{\partial b_{k}^{[l]}}.$$

  for all nodes and edges in the network.
- This is doen by the so-called *backprop algorithm*
- It works by successive use of the chain rule, from the last layer backward to the first



Backprop summary

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \frac{\partial J}{\partial z_k^{[l]}} a_j^{[l-1]}, \quad l = 1, \dots, L. \tag{5a}$$

$$\frac{\partial J}{\partial b_k^{[l]}} = \frac{\partial J}{\partial z_k^{[l]}}, \quad l = 1, \dots, L. \tag{5b}$$

$$\frac{\partial J}{\partial z_k^{[l]}} = g'(z_k^{[l]}) \sum_{j=1}^{n^{[l+1]}} \frac{\partial J}{\partial z_j^{[l+1]}} w_{kj}^{[l+1]}, \quad l = 1, \dots, L-1 \tag{5c}$$

$$\frac{\partial J}{\partial z_k^{[L]}} = \hat{y}_k - \tilde{y}_k. \tag{5d}$$

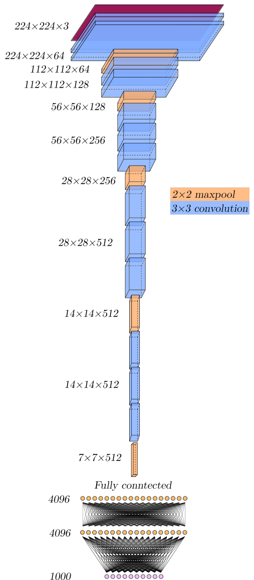
Note that

- Eqs. (5a)— (5c) are generally applicable
- Eq. (5d) assumes that J is the cross-entropy loss, and that $a^{[L]} = s(z^{[L]})$ with s as the softmax function.
- We also have vectorized versions of these equations

CNN ARCHITECTURES

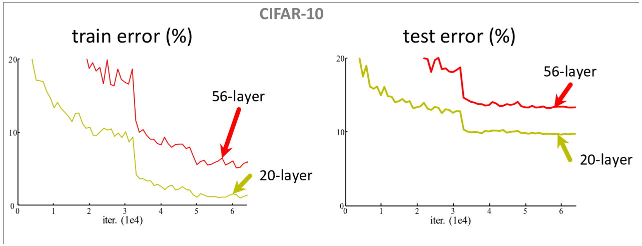
VGG16

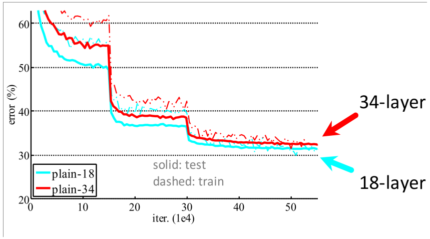
- A well-performing image classification network from 2014
- Simple and elegant design
  - alternating 2×2 maxpool
  - multiple 3×3 convolution layers
- Expencive, both in terms of memory and computation
  - Very many parameters
  - Many layers (at the time)
  - “Unsophisticated” architecture



ResNet's motivational problem

- Deeper models seems to be better
- However, very deep models perform worse
- Not due to overtraining
- Degradation problem



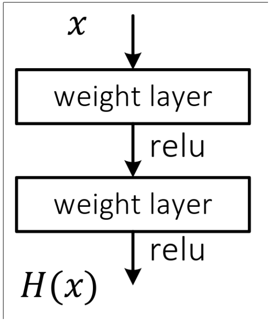


Deep models should be at least as good as shallow

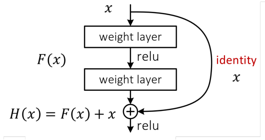
- A deeper model should not have higher training error
- “Proof” by construction
    - Take a shallow model
    - Insert extra layers as identity mappings
    - This deeper mode should have at least as good training error
- How to solve this is the key

Residual learning

- Stack a couple of layers
- Input x
- Let $H(x)$ be the desired mapping to be learned



- Explicitly compose the output as $H(x) = F(x) + x$, by adding the input x
- This means that what has been learnt is the residual $F(x) = H(x) - x$
- This should make identities $H(x) = x$ easier to learn
- Easier to train very deep networks



Residual blocks

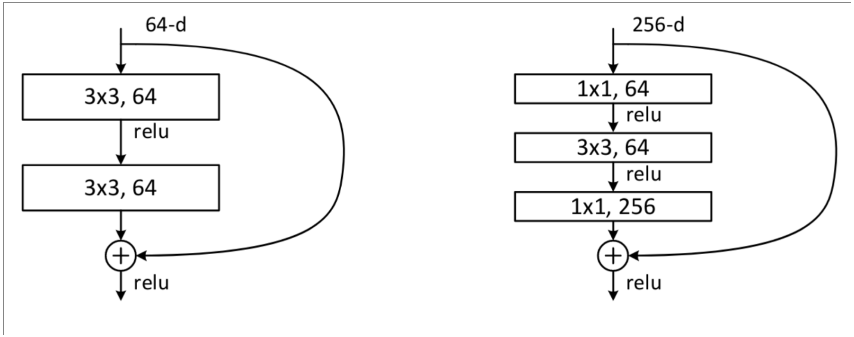
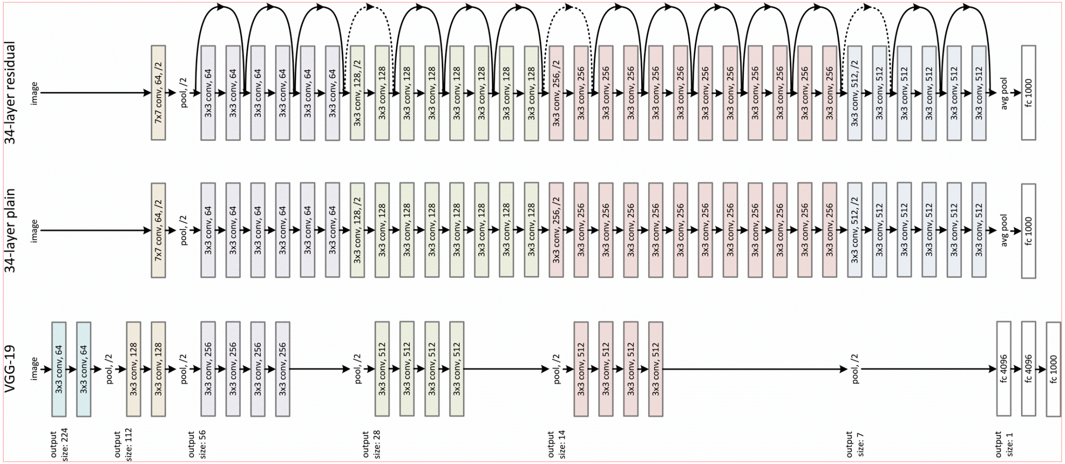


Figure 2: Left: a regular residual block. Right: a "bottleneck" residual block

Base architecture



OBJECT DETECTION AND IMAGE SEGMENTATION

Image classification and object localization

- Classify an image with a single object
- Draw a bounding box around the object

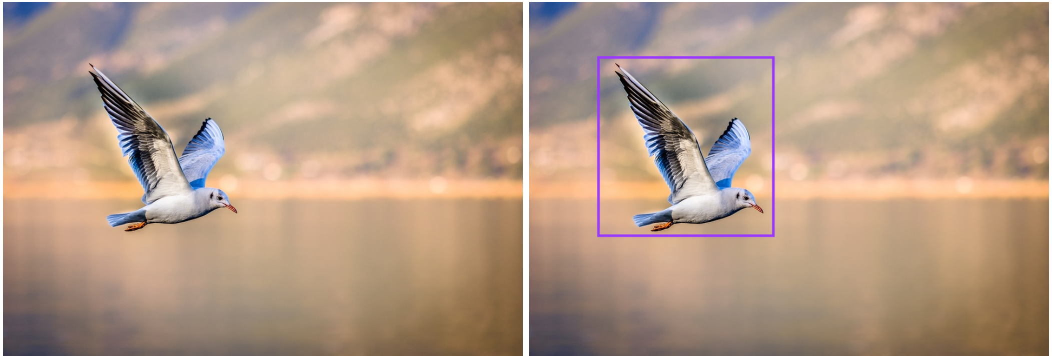


Figure 3: Seagull. Image source: https://www.pixabay.com

Target vector

- Add object/no object indicator c_0
- Interpret c_0 as $c_0 = \Pr(\text{there is an object in this box})$
- c_0 is often referred to as the *objectness*, but can also be thought of as a "catch-all" background class indicator
- Standard category probabilities from classification $(c_1, c_2, \ldots, c_{N_c})$
- Interpret c_i as $c_i = \Pr(\text{class}_i | c_0 = 1)$, $i = 1, \ldots, N_c$
- Add bounding box location specifiers
  - b_r: Center row coordinate
  - b_c: Center column coordinate
  - b_h: Box height
  - b_w: Box width

$$\hat{y} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N_c} \\ b_r \\ b_c \\ b_h \\ b_w \end{bmatrix}$$

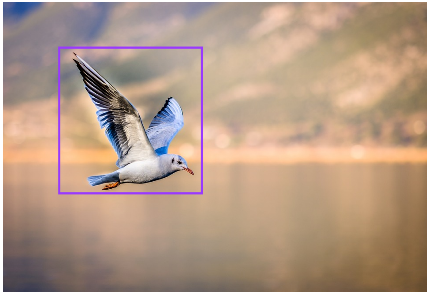


Figure 4: Seagull. Image source: https://www.pixabay.com

Example: Big cats

- c_1: Tiger
- c_2: Leopard
- c_3: Lion

$$\hat{y} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ b_r \\ b_c \\ b_h \\ b_w \end{bmatrix}$$

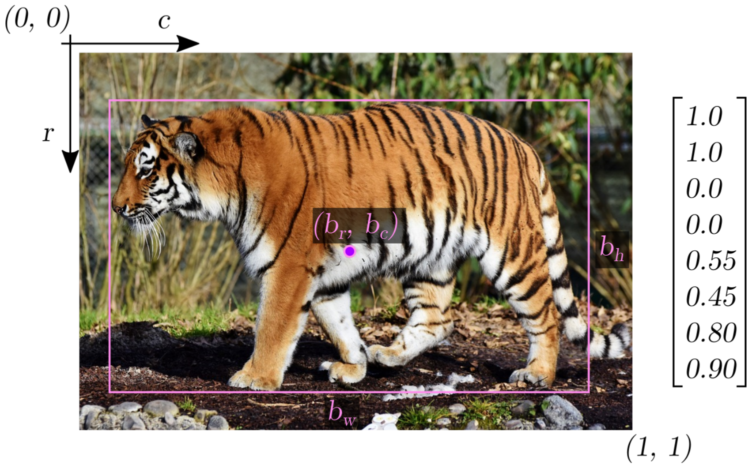


Figure 5: Tiger. Image source: https://www.pixabay.com

Example: Big cats

- c_1: Tiger
- c_2: Leopard
- c_3: Lion

$$\hat{y} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ b_r \\ b_c \\ b_h \\ b_w \end{bmatrix}$$

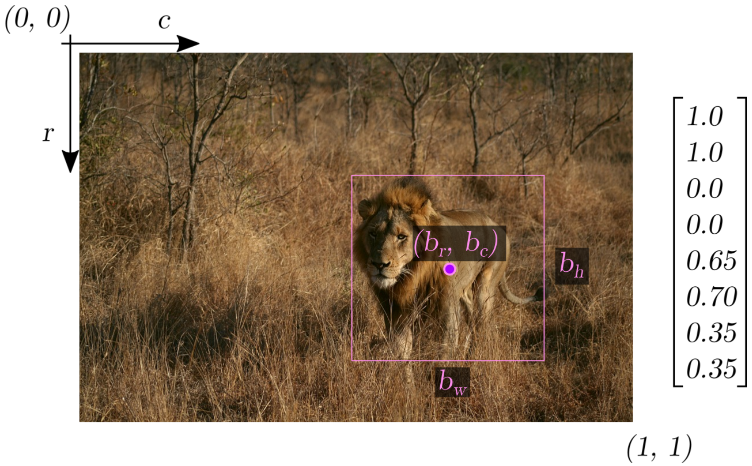


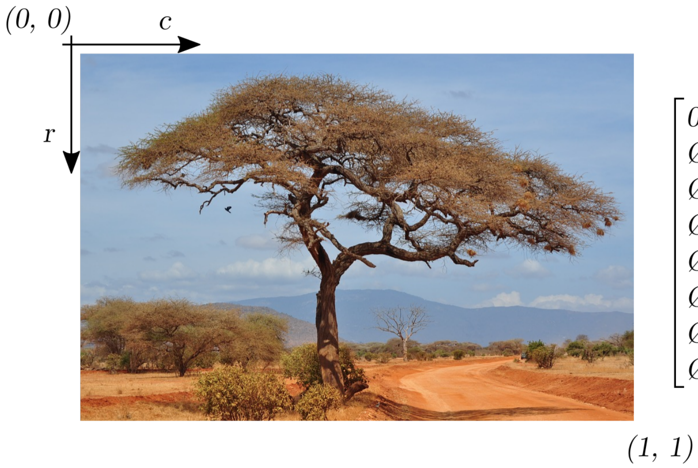
Figure 6: Lion. Image source: https://www.pixabay.com

Example: Big cats

- c_1: Tiger
- c_2: Leopard
- c_3: Lion

$$\hat{y} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ b_r \\ b_c \\ b_h \\ b_w \end{bmatrix}$$

Note that $c_0 = 0$, so we do not care about the rest, symbolized by $\emptyset$.



$$\begin{bmatrix} 0.0 \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Figure 7: Savannah. Image source: https://www.pixabay.com

Multi-task loss function

- Partition y into $y = [c, b]$, with
  - $c = [c_0, c_1, \ldots, c_{N_c}]$
  - $b = [b_r, b_c, b_h, b_w]$
- L_2 loss for object bounding box location b

$$L_b(\hat{b}, b) = \sum_{i \in \{x,y,h,w\}} (\hat{b}_i - b_i)^2$$

- Cross entropy loss for object categories c

$$L_c(\hat{c}, c) = -\sum_{i=1}^{n} c_i \log \hat{c}_i$$

- The total loss can be written as

$$L(\hat{y}, y) = L_c + [c_0 > 0] L_b$$

- Only compare bounding box if there is an object in the image

Sensitivity

- Proportion of positive reference instances labeled as positive by the proposed method

$$tpr = \frac{|B_R \cap B_P|}{|B_R|}$$

- Also known as
  - *true positive rate* (tpr)
  - *recall*
- Example on the right is pixel classification, but it also applies to object instances

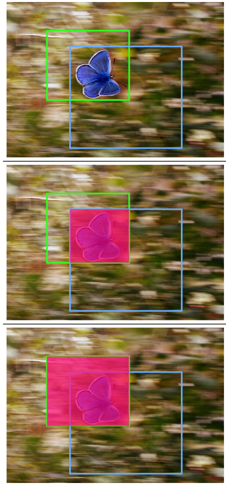


Figure 8: Top: reference (green), proposal (blue). Middle: True positive (red). Bottom: Reference positive (red). Image source: https://www.pixabay.com

Specificity

- Proportion of negative reference instances labeled as negative by the proposed method

$$tnr = \frac{|(B_R \cup B_P)^c|}{|B_R^c|}$$

- Also known as *true negative rate* (tnr)

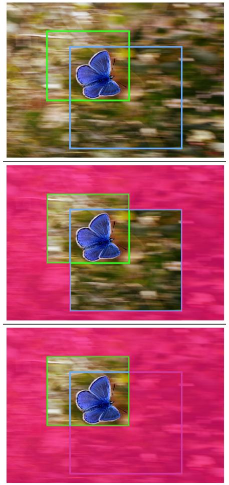


Figure 9: Top: reference (green), proposal (blue). Middle: True negative (red). Bottom: Reference negative (red). Image source: https://www.pixabay.com

Precision

- Proportion of proposed positive instances that are also labeled positive by the reference

$$ppv = \frac{|B_R \cap B_P|}{|B_P|}$$

- Also known as *positive predictive value* (ppv)
- Example on the right is pixel classification, but it also applies to object instances

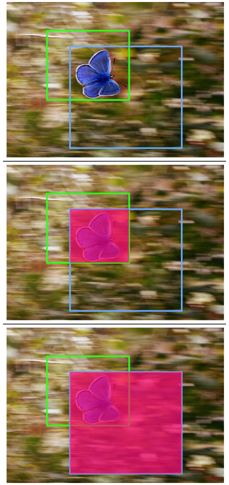


Figure 10: Top: reference (green), proposal (blue). Middle: True positive (red). Bottom: Proposed positive (red). Image source: https://www.pixabay.com

Jaccard index

- The proportion of all instances classified as positive by the reference and/or the proposal method, that are classified as positive by both the reference and the proposal method

$$iou = \frac{|B_R \cap B_P|}{|B_R \cup B_P|}$$

- Also known as
    - Intersection over Union (IoU)
    - Tanimoto index
- Example on the right is pixel classification, but it also applies to object instances

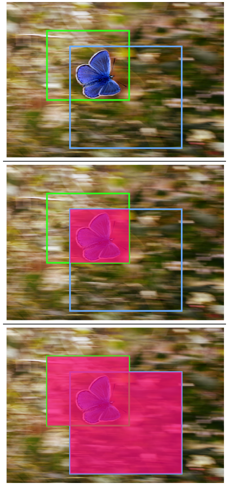


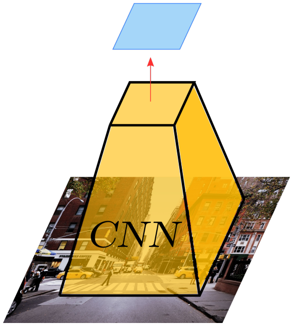
Figure 11: Top: reference (green), proposal (blue). Middle: Intersection (red). Bottom: Union (red). Image source: https://www.pixabay.com

Fast R-CNN

- Get region proposals (as in R-CNN)

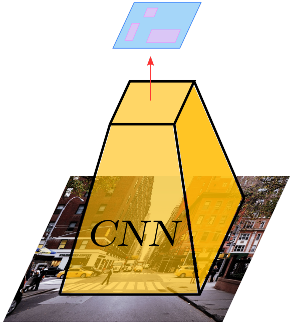
Fast R-CNN

- Get region proposals (as in R-CNN)
- Run a CNN on the entire image



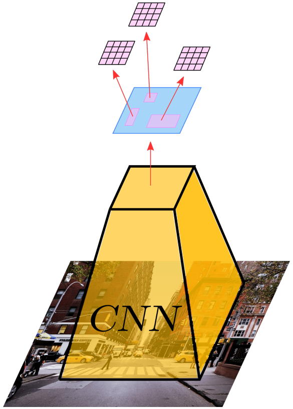
Fast R-CNN

- Get region proposals (as in R-CNN)
- Run a CNN on the entire image
- Project region proposals (ROIs) onto output CNN feature map



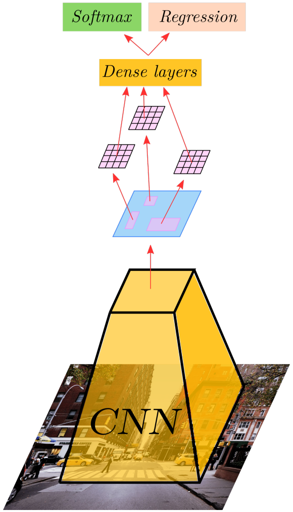
Fast R-CNN

- Get region proposals (as in R-CNN)
- Run a CNN on the entire image
- Project region proposals (ROIs) onto output CNN feature map
- ROI pooling layer



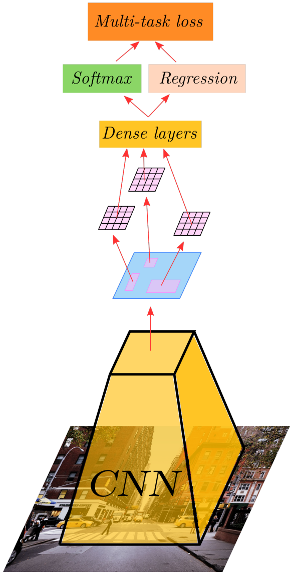
FAST R-CNN

- Get region proposals (as in R-CNN)
- Run a CNN on the entire image
- Project region proposals (ROIs) onto output CNN feature map
- ROI pooling layer
- Feed the fixed-sized pooled region to fully connected layers
- One softmax output for class prediction
- One regression output for the bounding box prediction

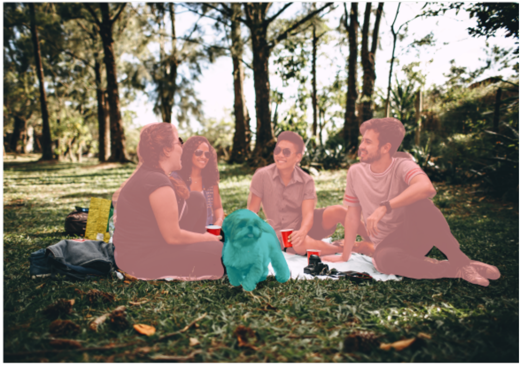


Fast R-CNN

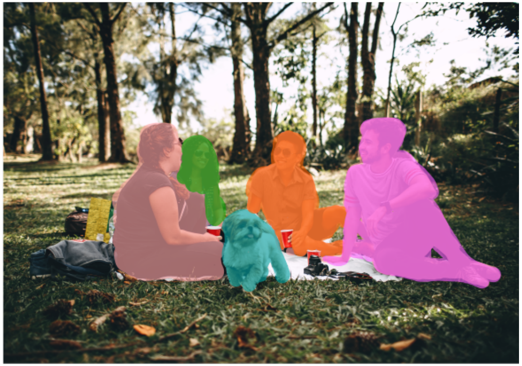
- Get region proposals (as in R-CNN)
- Run a CNN on the entire image
- Project region proposals (ROIs) onto output CNN feature map
- ROI pooling layer
- Feed the fixed-sized pooled region to fully connected layers
- One softmax output for class prediction
- One regression output for the bounding box prediction
- Multi-task loss



Semantic segmentation



Instance segmentation



CNN multiple pixel classification

- Segment image all at once
- Input image shape: $H \times W \times C$
- Output layer shape: $H \times W \times N_c$, where N_c: number of classes
- Pixel-wise cross entropy loss
  - Softmax over channels at a pixel location
  - Repeat, and average over all pixels
- Very expensive on computation and memory

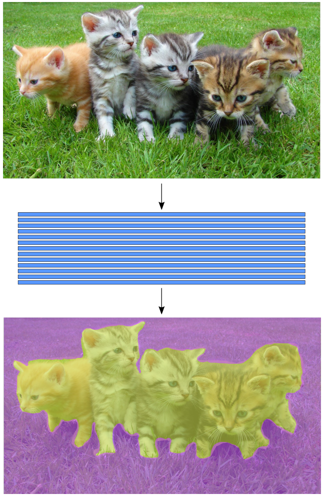


Figure 12: Top: Original. Bottom: Segmented. Image source: `https://www.pexels.com`

Downsampling and upsampling

- Segment image all at once
- Input image shape: $H \times W \times C$
- Output layer shape: $H \times W \times N_c$, where N_c: number of classes
- Spatial downsampling followed by upsampling (encoding, decoding)
- Pixel-wise cross entropy loss
    - Softmax over channels at a pixel location
    - Repeat, and average over all pixels
- Different upsampling techniques

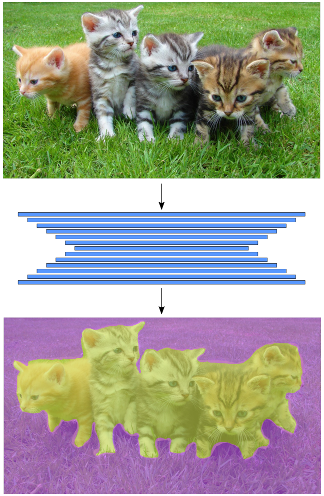
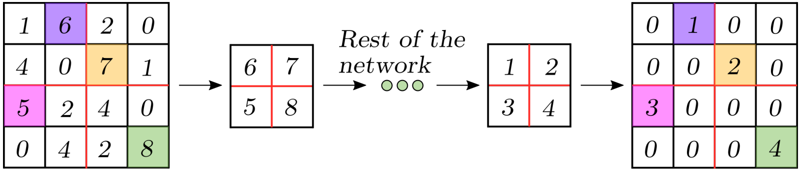
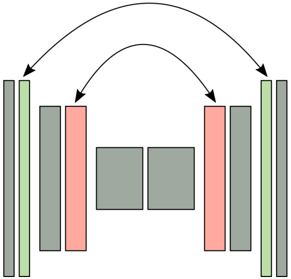


Figure 13: Top: Original. Bottom: Segmented. Image source: https://www.pexels.com

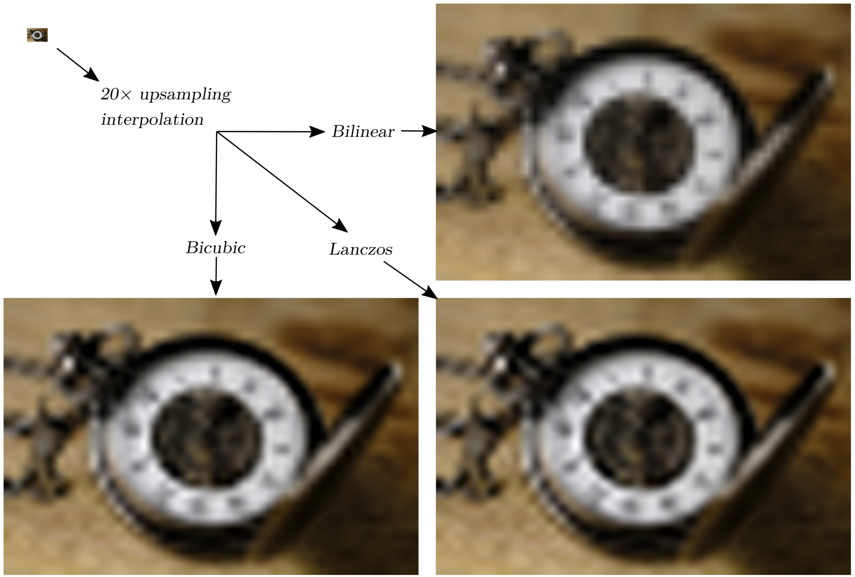
Max unpooling

Remember max locations from max pool downsampling. Reverse this on the “opposite” layer



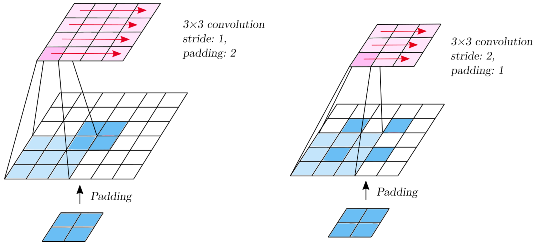


Interpolation upsampling



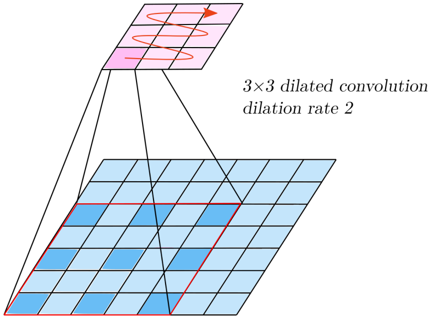
Transposed convolution

- Can view convolution as a matrix-matrix multiplication
- Transposed convolution gets its name by transposing this operation
- Also called
    - fractionally strided convolution
    - econvolution (this is a misnomer)



Dilated convolution

- Insert spacing between convolution kernel cells (dilation rate)
- Also called
    - convolution with holes
    - A-trous convolution (a trous is french for with holes)



UNSUPERVISED LEARNING

t-distributed Stochastic Neighbour Embedding (t-SNE)

- Transforms high-dimensional (hd) data points to low-dimensional (ld) data ponts
- Aims to preserve neighbourhood identity between data points
- Similar (close) hd points should also be similar (close) in the ld representation
- For each point i, we define two distributions:
    - $p_i(x_i, x_j)$: The probability that x_i and x_j are "neighbours",
    - $q_i(y_i, y_j)$: The probability that y_i and y_j are "neighbours",
- p_i are distributions over all hd neighbours x
- q_i are distributions over all ld neighbours y
- For p, we use symmetric gaussian distributions
- For q, we use symmetric student-t distributions
- We make q similar to p by minimizing the KL-divergence between the two
- The KL-divergence is minimized by determining the ld points y with gradient descent

Autoencoders

- An autoencoder f consist of an encoder g and an decoder h
- The encoder maps the input x to some representation z

$$g(x) = z$$

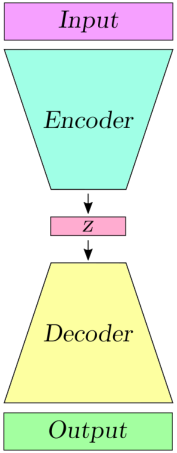
- We often call this representation z for the code or the latent vector
- The decoder maps this representation z to some output $\hat{x}$

$$g(z) = \hat{x}$$

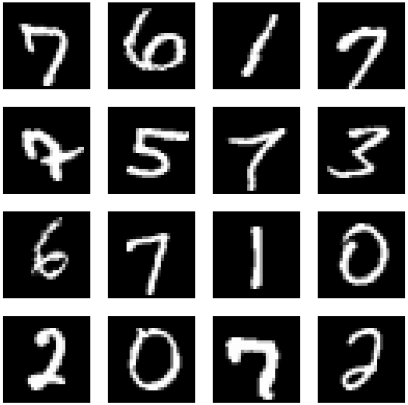
- We want to train the encoder and decoder such that

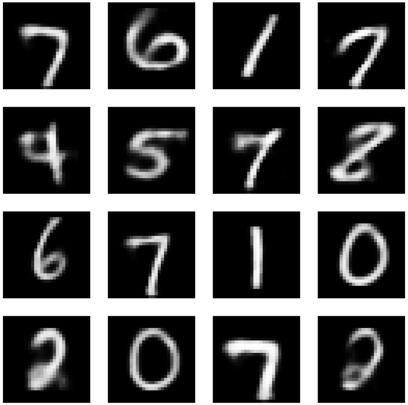
$$f(x) = h(g(x)) = \hat{x} \approx x$$

- Commonly used for compression, feature extraction and de-noising

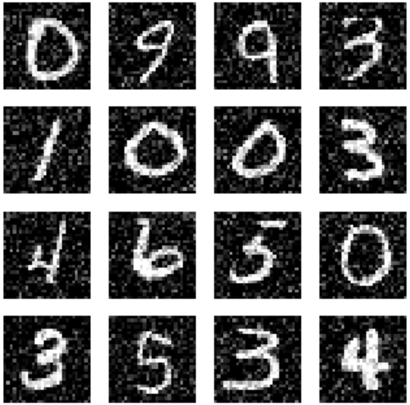


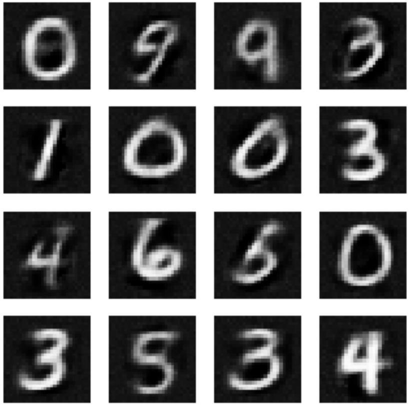
Compression autoencoder — MNIST example


(a) Original


(b) Reconstructed

De-noising autoencoder — MNIST example


(a) Original

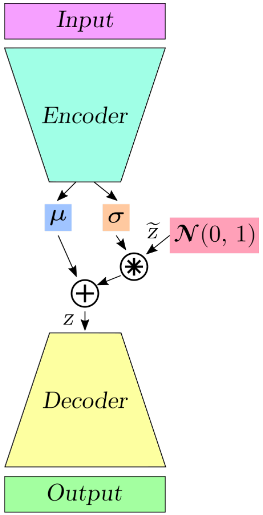

(b) Reconstructed

Variational autoencoders

- A variational autoencoder is designed to have a continuous latent space
- This makes them ideal for random sampling and interpolation
- It achieve this by forcing the encoder g to generate Gaussian representations, $z \sim \mathcal{N}(\mu, \sigma^2)$
- More precisely, for one input, the encoder generates a mean μ and a variance σ^2
- We sample then sample a zero-mean, unit-variance Gaussian $\tilde{z} \sim \mathcal{N}(0, 1)$
- Construct the input z to the decoder from this

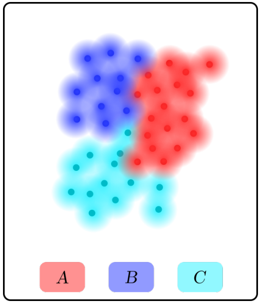
$$z = \mu + \tilde{z}\sigma^2$$

- With this, z is sampled from $q = \mathcal{N}(\mu, \sigma^2)$



Intuition

- This is a stochastic sampling
- That is, we can sample different z from the same set of (μ, σ^2)
- The intuition is that the decoder "learns" that for a given input x:
  - the point z is important for reconstruction
  - but also a neighbourhood of z
- In this way, we have smoothed the latent space, at least locally
- In the previous lecture, we learnt ways to achieve this



Guiding the generative distribution

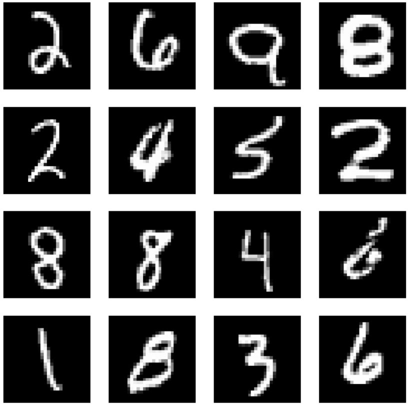
- We can guide the solutions by restricting the generative distribution q
- We do this by making it approximate some distribution p
- In that way, the latent vectors, even for different categories, will be relatively close
- The desired distribution used in variational autoencoders is the standard normal $p = \mathcal{N}(0, 1)$
- We use the familiar KL-divergence between the desired and the generated distribution as a regularizer in the loss function
- With this, the total loss for an example x_i is something like

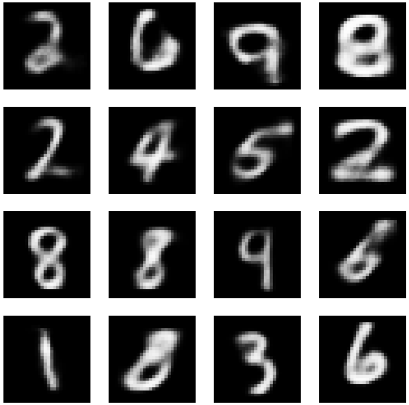
$$J(x_i) = ||x^{(i)} - f(x^{(i)})|| + D_{KL}(p||q_{\mu_i,\sigma_i})$$

- That is, the sum what we call the *reconstruction loss* and the *latent loss*
- The latent loss for a single variable x_i can be shown to be equal to

$$D_{KL}(p||q_{\mu_i,\sigma_i}) = \frac{1}{2}(\mu_i^2 + \sigma_i^2 - \log \sigma_i^2 - 1)$$

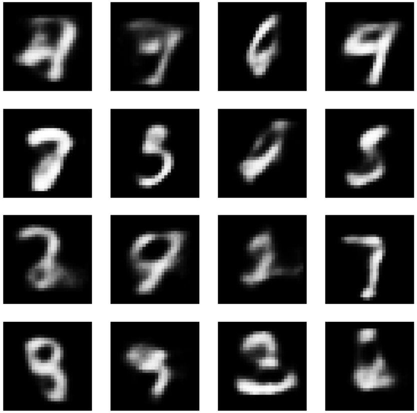
VAE EXAMPLE: RECONSTRUCTION


(a) Original


(b) Reconstructed

VAE example: generation of new signals

- Sample a random latent vector z from $\mathcal{N}(0, 1)$
- Decode z



VAE example: interpolation between samples

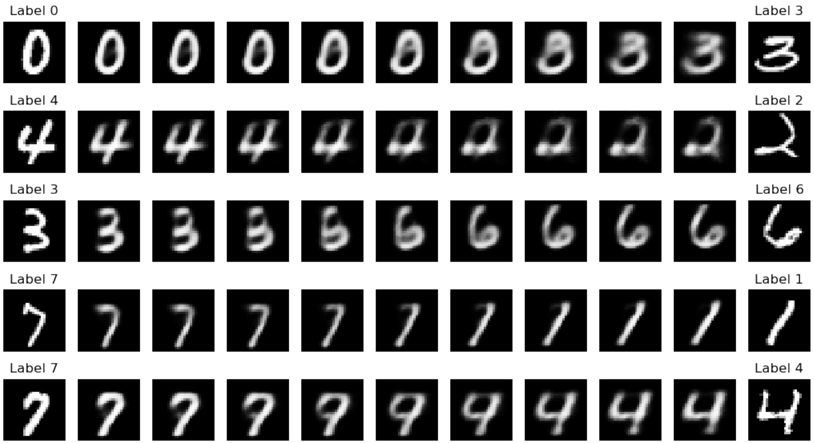
- We generate a signal c that is an interpolation between two signals a and b
- We can do this by a linear interpolation between the means

$$\mu_{c_k} = (1 - w_k)\mu_a + w_k \mu_b$$

where the different interpolation weights can be

$$w_k = \frac{k}{n+1}, \quad k = 1, \ldots, n$$

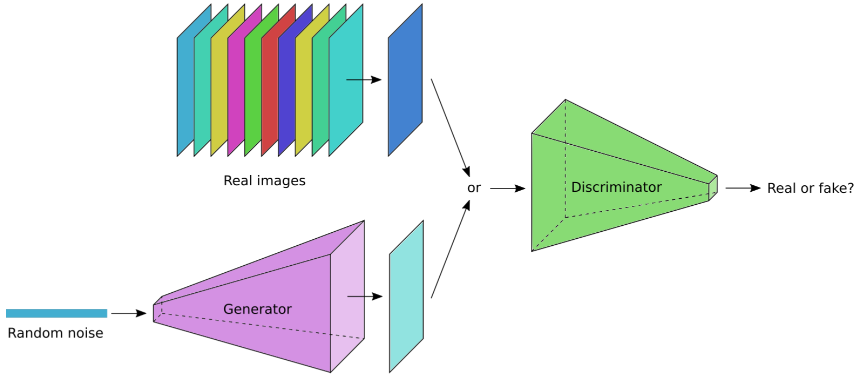
VAE example: interpolation between samples



GENERATIVE ADVERSARIAL NETWORKS

Components

- A *generator* function that tries to create real-looking examples
- A *discriminator* function that tries to distinguish real from fake examples
- Functions are updated in a feedback loop, making each better at its task



Components

- The discriminator is a function

$$D : x \mapsto D(x; \theta_D)$$

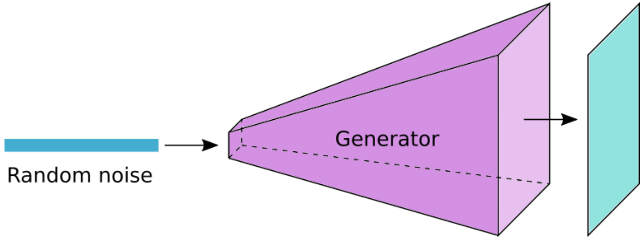
  mapping input x to $D(x; \theta_D)$ with parameters θ_D
- The generator is a function

$$G : z \mapsto G(z; \theta_G)$$

  mapping input z to $G(z; \theta_G)$ with parameters θ_G
- The discriminator has an associated loss $J_D(\theta_D, \theta_G)$, depending on both θ_D and θ_G, but can only control θ_D
- The generator has an associated loss $J_G(\theta_D, \theta_G)$, depending on both θ_D and θ_G, but can only control θ_G
- The optimal solution (θ_D^*, θ_G^*) is a *Nash equilibrium* where
  - θ_D^* is a local minimum of J_D w.r.t. θ_D
  - θ_G^* is a local minimum of J_G w.r.t. θ_G

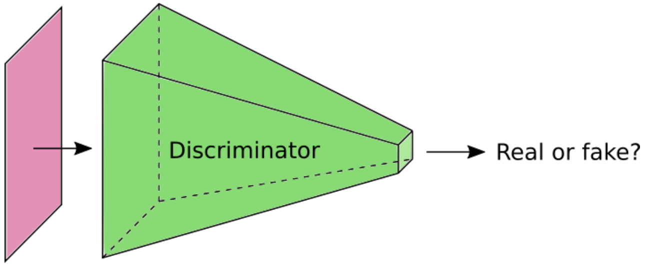
The generator

- The generator is a differentiable function
- The input z is a random vector sampled from some simple prior distribution p_g
- The output $x = G(z)$ is then sampled from p_m
- The most common form of G is some kind of generative neural network
- If we have GAN trained on data from p_d, we can use the generator to sample from p_m
- $p_m \approx p_d$
- With this, samples from the generator will look like the training data



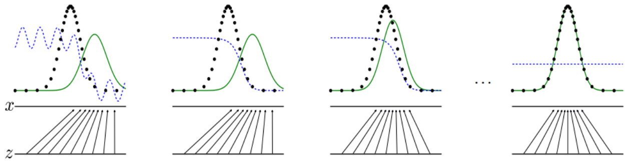
The discriminator

- The discriminator is a standard classification network
- Trained to differentiate between real and fake (generated) images
- Outputs a single number in $[0, 1]$
  - $D(x) = 0 \rightarrow D$ believes x is fake
  - $D(x) = 1 \rightarrow D$ believes x is real



The training process

- At each update step, one mini-batch x of real images, and one mini-batch z of latent vectors are drawn
- z is fed through G, producing $G(z)$
- $D(x)$ is compared with $D(G(z))$
- θ_G is updated using gradients from J_G
- θ_D is updated using gradients from J_D
- The discriminator and generator are updated in tandem using some regular optimization routine (SGD, Adam, etc.)
- Some flexibility with regards to updating one more often than the other



The discriminator cost function

- The generator, G, and discriminator, D, are two distinct networks with distinct cost functions
- The cost functions are optimized separately
- The discriminator cost function is given by

$$\begin{aligned} J_D(\theta_D, \theta_G) &= -E_{x\sim p_d}[\log D(x;\theta_D)] - E_{z\sim p_g}[\log(1 - D(G(z;\theta_G);\theta_D))] \\ &= -E_{x\sim p_d}[\log D(x;\theta_D)] - E_{x\sim p_m}[\log(1 - D(x;\theta_D))] \end{aligned}$$

- With discrete samples, over one mini-batch $\{x_i\}$ and $\{z_i\}$, this becomes

$$J_D(\theta_D, \theta_G) = -\frac{1}{m}\sum_{i=1}^{m}[\log(D(x_i;\theta_D)) + \log(1 - D(G(z_i;\theta_G);\theta_D))]$$

- Binary classification with sigmoid cross entropy where
  - Real images are given label 0
  - Generated (fake) images are given label 1

The generator cost function

- For the generator cost, we propose the following

$$\begin{aligned} J_G(\theta_D, \theta_G) &= -E_{z\sim p_g} \log D(G(z;\theta_G);\theta_D) \\ &= -\frac{1}{m}\sum_{i=1}^{m} \log D(G(z_i;\theta_G);\theta_D) \end{aligned}$$

- With this, the generator maximizes the log-probability of the discriminator being mistaken (assigning label 1 to the generated examples)
- Contrast this with the previous minimax game where we the generator minimizes the log-probability of the discriminator being correct (assigning label 0 to the generated examples)
- Both the generator and the discriminator now have strong gradients when they are “losing the game”

Cost functions overview

- Minimizing the discriminative cost

$$J_D(\theta_D, \theta_G) = -\frac{1}{m}\sum_{i=1}^{m}[\log(D(x_i;\theta_D)) + \log(1 - D(G(z_i;\theta_G);\theta_D))]$$

  "pushes" $D(x)$ to 1 (real class) and $D(G(z))$ to 0 (fake class)

- Minimizing the generative cost

$$J_G(\theta_D, \theta_G) = -\frac{1}{m}\sum_{i=1}^{m}\log(D(G(z_i;\theta_G);\theta_D))$$

  "pushes" $D(G(z))$ to 1 (real class)

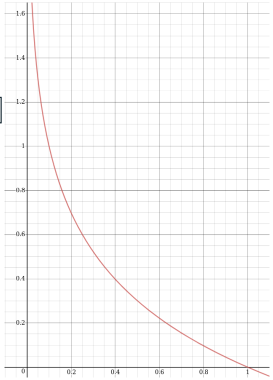


Figure 17: Graph of $f(x) = -\log x$

Questions?