SC 4001 CE/CZ 4042 Neural Network and Deep Learning

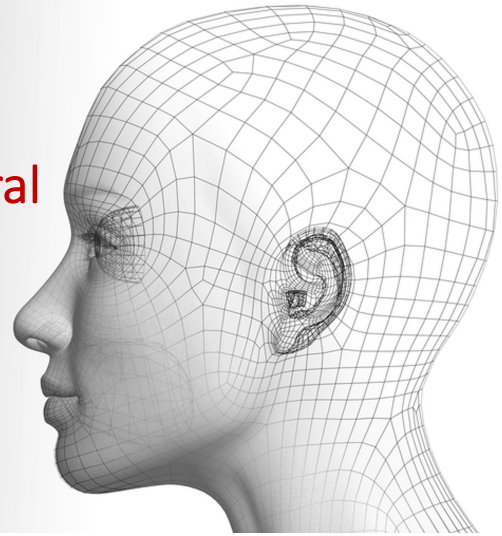
Last update: 25 Feb 2024

Convolutional Neural Networks I

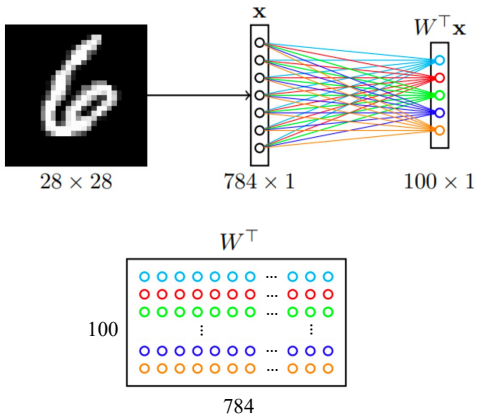
Xingang Pan
潘新钢

https://xingangpan.github.io/

https://twitter.com/XingangP



Motivation



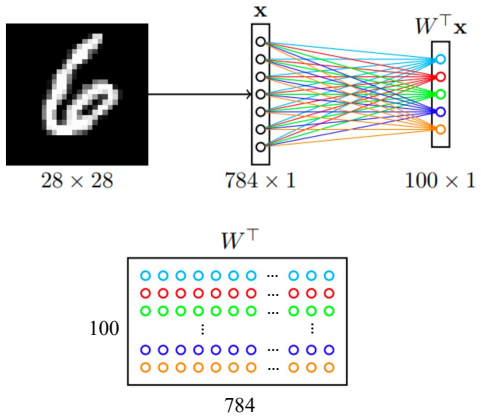
Issues

- Too many parameters: 100 × 784 + 100.
  - What if images are 640 × 480 × 3?
  - What if the first layer counts 1000 units?

Fully connected networks where neurons at one level are connected to the neurons in other layers are not feasible for signals of large resolutions and are computationally expensive in feedforward and backpropagation computations.

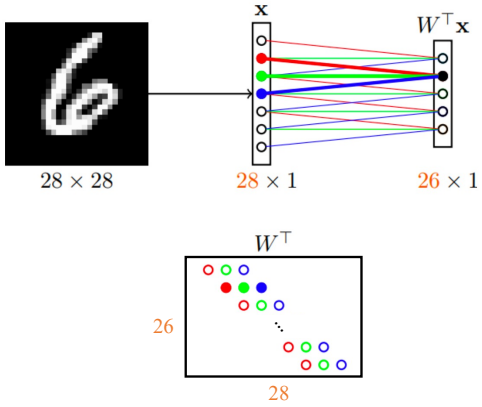
- Spatial organization of the input is destroyed.
  - The network is not invariant/equivariant to transformations (e.g., translation).

Motivation



Let us consider the first layer of a MLP taking images as input. What are the problems with this architecture?

Locally connected networks



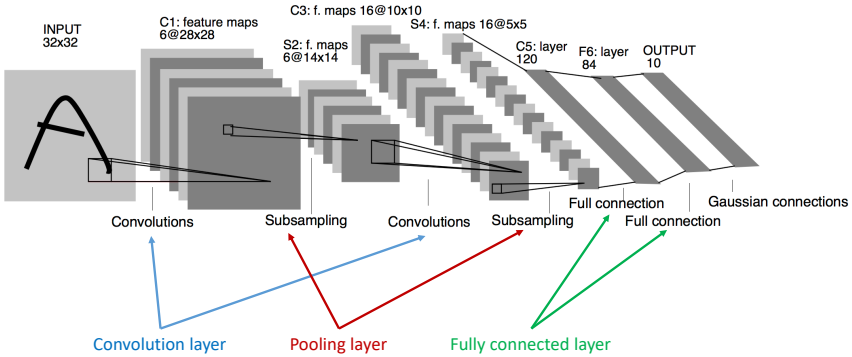
Instead, let us only keep a sparse set of connections, where all weights having the same color are shared.

- The resulting operation can be seen as shifting the same weight triplet (kernel).
- The set of inputs seen by each unit is its receptive field.

This is a 1D convolution, which can be generalized to more dimensions.

Basic Components in CNN

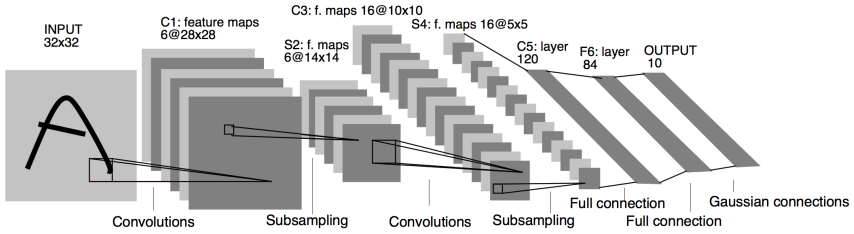
An example of convolutional network: LeNet 5

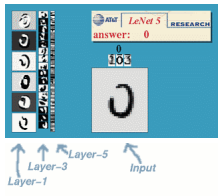


As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

Common layer arrangement: Conv → pool → Conv → pool → fully connected → fully connected → output

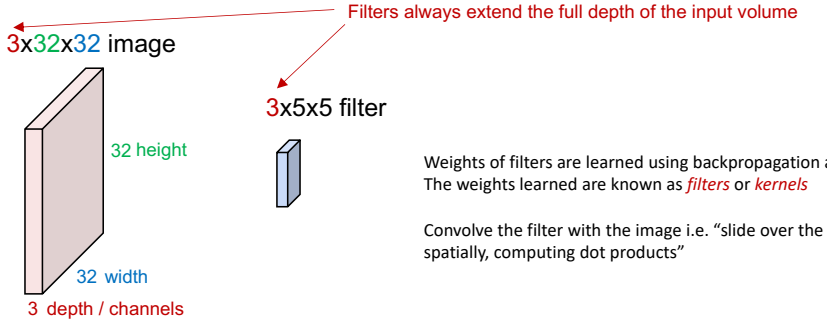
An example of convolutional network: LeNet 5





LeCun et al., 1998
Turing Award (2018)

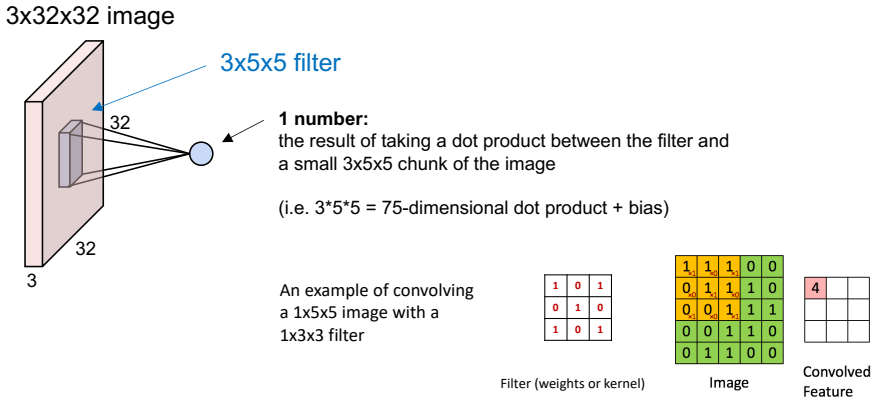
Convolution layer



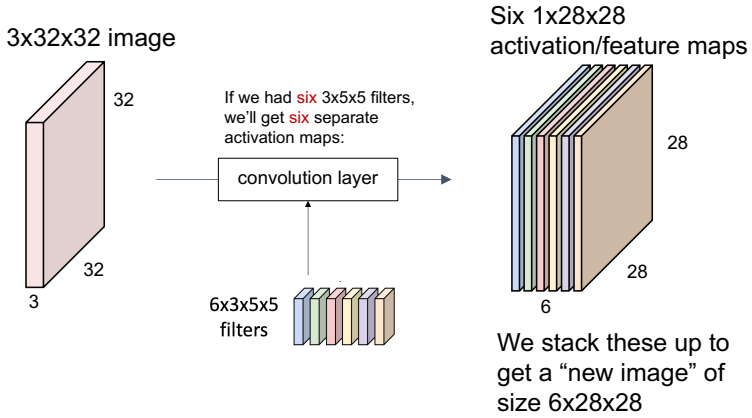
Weights of filters are learned using backpropagation algorithm. The weights learned are known as *filters* or *kernels*

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

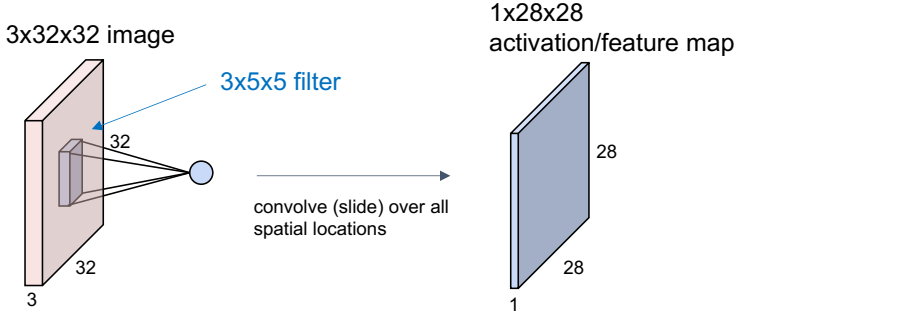
Convolution layer



Convolution layer

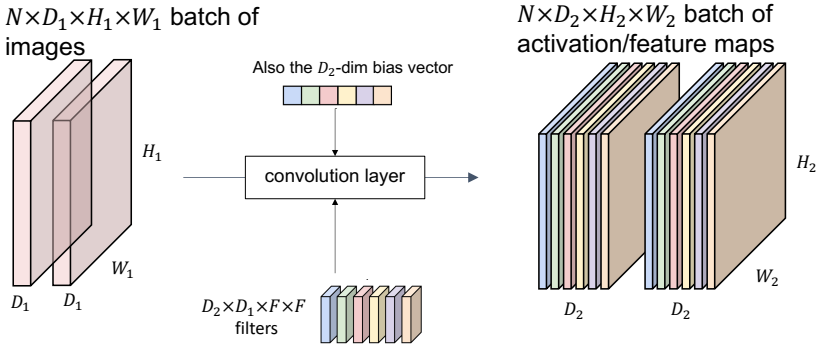


Convolution layer

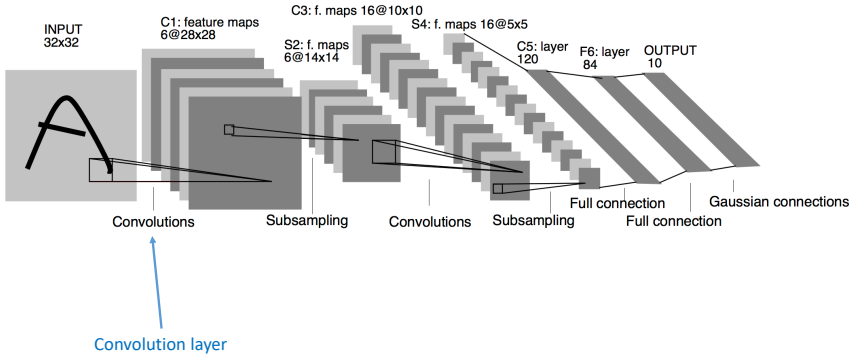


The activations are obtained by convolving the filters (weights) with the input activations. The output activation produced by a particular filter is known as a *activation map / feature map*

Convolution layer



An example of convolutional network: LeNet 5



Back to this example, 6@28x28 means that we have 6 feature maps of size 28*28
You can imagine that the first convolutional layer uses 6 filters and each filter is of size 5 x 5 (how do we know that? We will discuss that later)

Convolution layer

Consider a kernel $\boldsymbol{w} = \{w(l,m)\}$, which has a size of $L \times M, L = 2a + 1, M = 2b + 1$

Synaptic input at location $p = (i,j)$ of the first hidden layer due to a kernel is given by

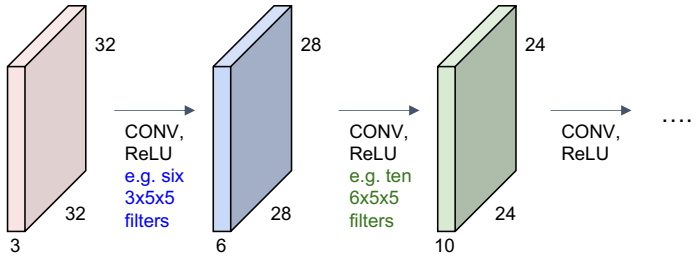
$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m) w(l,m) + bias$$

For instance, given $L = 3, M = 3$

$$u(i,j) = x(i-1,j-1)w(-1,-1) + x(i-1,j)w(-1,0) + \ \ldots$$
$$+x(i,j)w(0,0) + x(i+1,j+1)w(1,1) + bias$$

Convolution layer

CNN is a sequence of convolutional layers, interspersed with activation functions



Convolution layer

The output of the neuron at (i,j) of the convolution layer

$$y(i,j) = f(u(i,j))$$

where f is an activation function. For deep CNN, we typically use ReLU, $f(x) = \max(0,x)$.

Note that one weight tensor $\boldsymbol{w}_k = \{w_k(l,m)\}$ or kernel (filter) creates one feature map:

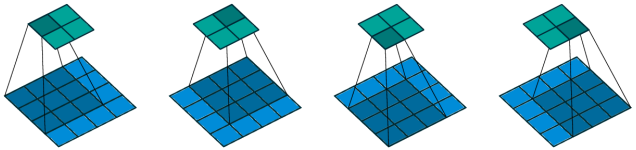
$$\boldsymbol{y}_k = \{y_k(i,j)\}$$

If there are K weight vectors $(\boldsymbol{w}_k)_{k=1}^K$, the convolutional layer is formed by K feature maps

$$\boldsymbol{y} = (\boldsymbol{y}_k)_{k=1}^K$$

Convolution layer

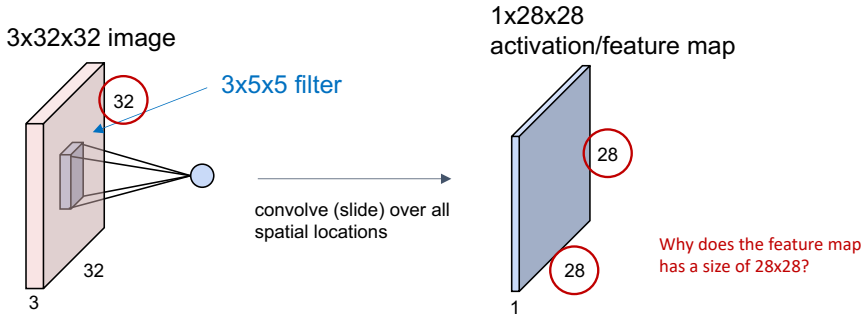
Convolution by doing a sliding window



As a guiding example, let us consider the convolution of single-channel tensors $\mathbf{x} \in \mathbb{R}^{4\times4}$ and $\mathbf{w} \in \mathbb{R}^{3\times3}$:

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

Convolution layer – spatial dimensions



Why does the feature map has a size of 28x28?

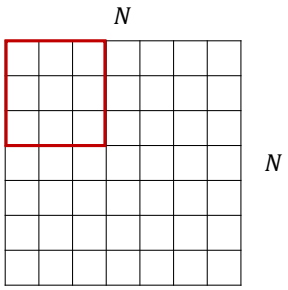
Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m) w(l,m) + bias$$

$$u(1,1) = 4\times1 + 5\times4 + 8\times1 + 1\times1 + 8\times4 + 8\times3 + 3\times3 + 6\times3 + 6\times1 = 122$$
$$u(1,2) = 5\times1 + 8\times4 + 7\times1 + 8\times1 + 8\times4 + 8\times3 + 6\times3 + 6\times3 + 4\times1 = 148$$
$$u(2,1) = 1\times1 + 8\times4 + 8\times1 + 3\times1 + 6\times4 + 6\times3 + 6\times3 + 5\times3 + 7\times1 = 126$$
$$u(2,2) = 8\times1 + 8\times4 + 8\times1 + 6\times1 + 6\times4 + 4\times3 + 5\times3 + 7\times3 + 8\times1 = 134$$

Convolution layer – spatial dimensions



$N\times N$ input (spatially), assume $F\times F$ filter, and S stride

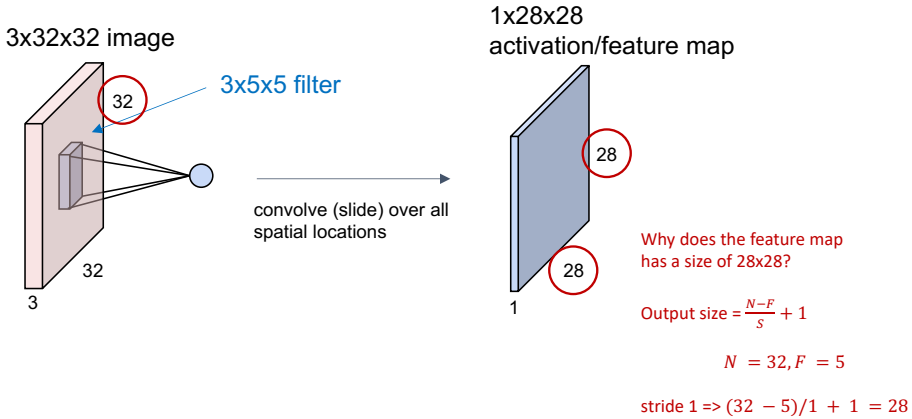
Output size = $\frac{N-F}{S} + 1$

e.g.

$$N = 7, F = 3$$

stride 1 => $(7 - 3)/1 + 1 = 5$
stride 2 => $(7 - 3)/2 + 1 = 3$
stride 3 => $(7 - 3)/3 + 1 = 2.33$:\

Convolution layer – spatial dimensions



Why does the feature map has a size of 28x28?

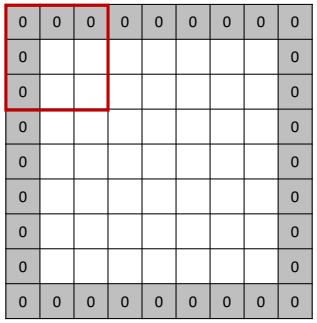
Output size = $\frac{N-F}{S}+1$

$N = 32, F = 5$

stride 1 => $(32 - 5)/1 + 1 = 28$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized, e.g., 'constant', 'reflect', and 'replicate').



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output shape?

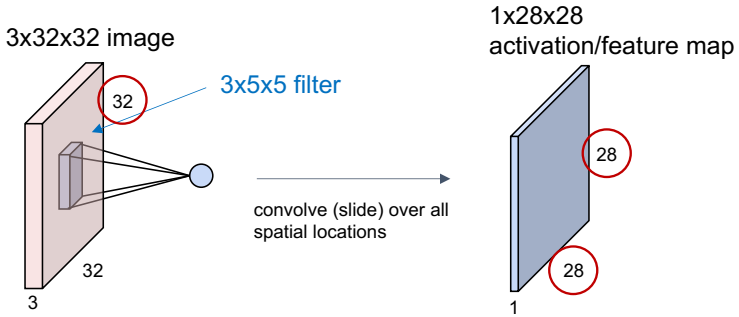
Recall that without padding, output size = $\frac{N-F}{S}+1$

With padding, output size = $\frac{N-F+2P}{S}+1$
e.g.

$N = 7, F = 3$

stride 1 => $(7 - 3 + 2(1))/1 + 1 = 7$

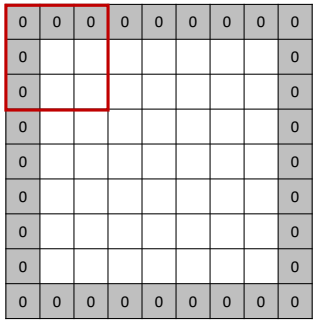
Convolution layer – spatial dimensions



The valid feature map is smaller than the input after convolution.
Without zero-padding, the width of the representation shrinks by the $F - 1$ at each layer
To avoid shrinking the spatial extent of the network rapidly, small filters have to be used

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized).



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output shape?

7×7 output!

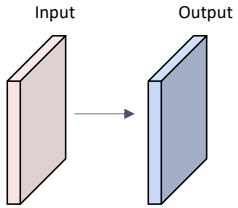
In general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F - 1)/2$. (will preserve size spatially)

e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Example

Input volume: 3×32×32
Ten 3×5×5 filters with stride 1, pad 2

Output volume size: ?



Output size =

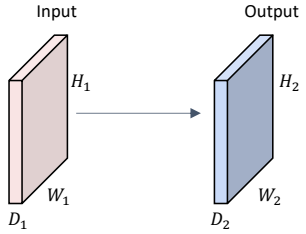
$$\frac{N-F+2P}{S}+1$$

$$\frac{32-5+2(2)}{1}+1 = 32 \text{ spatially}$$

The output volume size is 10×32×32

Convolution layer - summary

A convolution layer

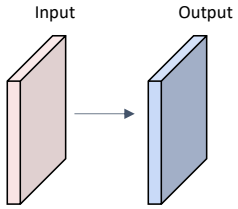


- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
  - Number of filters K
  - Their spatial extent F
  - The stride S
  - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d-th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias

Example

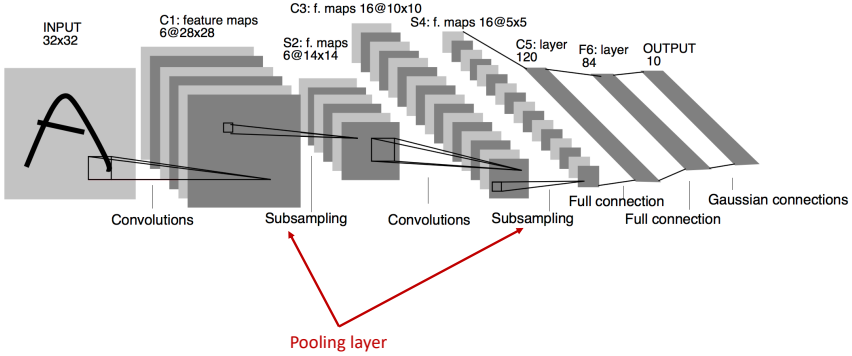
Input volume: 3×32×32
Ten 3×5×5 filters with stride 1, pad 2

Number of parameters in this layer: ?

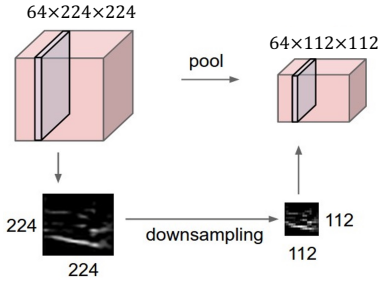


Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)
=> $76 \times 10 = \mathbf{760}$

An example of convolutional network: LeNet 5



Pooling layer



- Operates over each activation map independently
- Either 'max' or 'average' pooling is used at the pooling layer. That is, the convolved features are divided into disjoint regions and pooled by taking either maximum or averaging.

Pooling layer

Consider pooling with non-overlapping windows $\{(l,m)\}_{l,m=-L/2,-M/2}^{L/2,M/2}$, of size $L \times M$

The max pooling output is the maximum of the activation inside the pooling window. Pooling of a feature map $\boldsymbol{y}$ at $p = (i,j)$ produce pooled feature

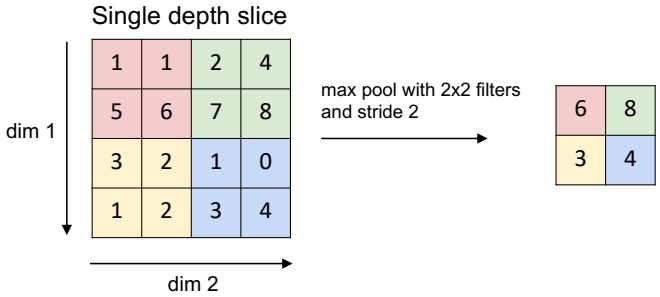
$$z(i,j) = \max_{l,m}\{y(i+l, j+m)\}$$

The mean pooling output is the mean of activations in the pooling window

$$z(i,j) = \frac{1}{L \times M}\sum_{l}\sum_{m} y(i+l, j+m)$$

Pooling layer

MAX Pooling



Pooling is intended to subsample the convolution layer.
The default stride for pooling is equal to the filter width.

Pooling layer

Why pooling?

A function f is invariant to g if $f(g(\boldsymbol{x})) = f(\boldsymbol{x})$.

- Pooling layers can be used for building inner activations that are (slightly) invariant to small translations of the input.
- Invariance to local translation is helpful if we care more about the presence of a pattern rather than its exact position.

Pooling layer - summary

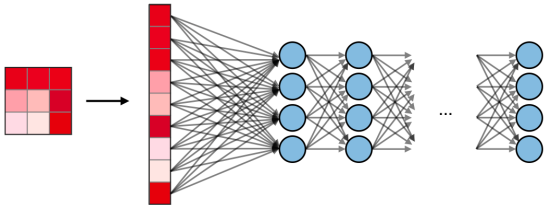
A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
  - Their spatial extent F
  - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

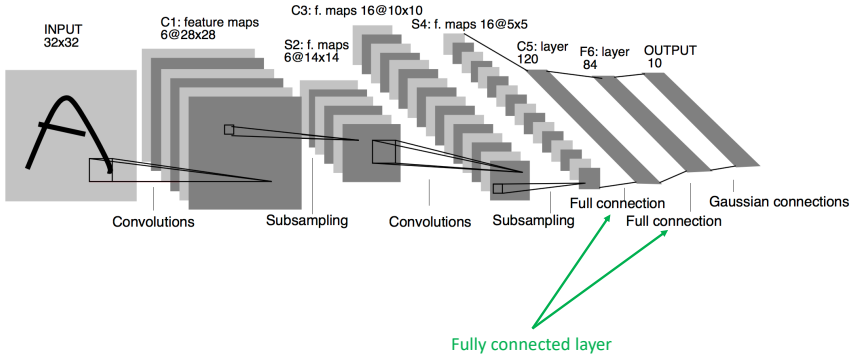
Fully connected layer

The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons.

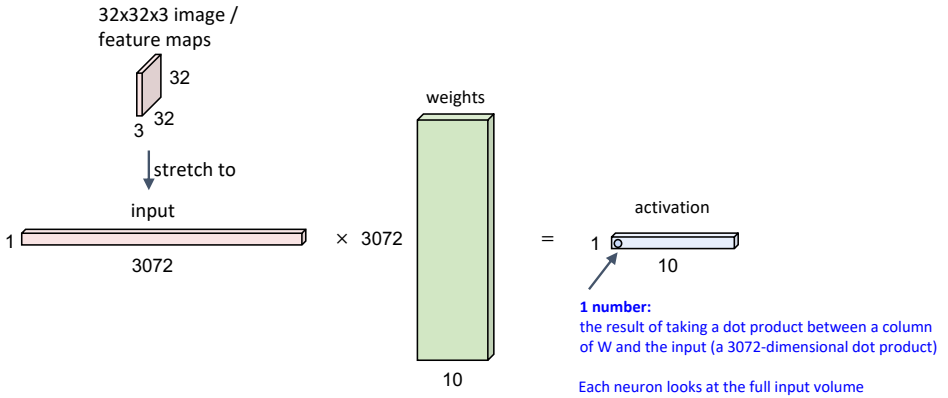
If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



An example of convolutional network: LeNet 5



Fully connected layer

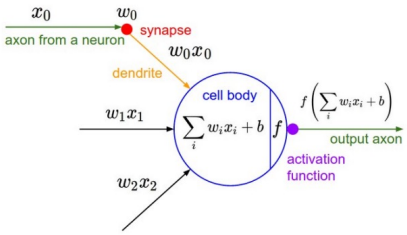


Activation function

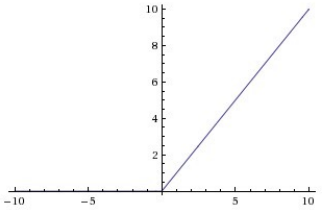
Recall that the output of the neuron at (i, j) of the convolution layer

$$y(i,j) = f(u(i,j))$$

where u is the synaptic input and f is an activation function (It aims at introducing non-linearities to the network.).



Activation function – ReLU

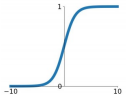


$$f(x) = \max(0, x)$$

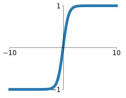
- Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0

Activation function

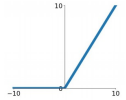
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



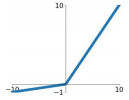
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

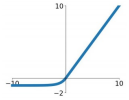


Leaky ReLU
$\max(0.1x, x)$



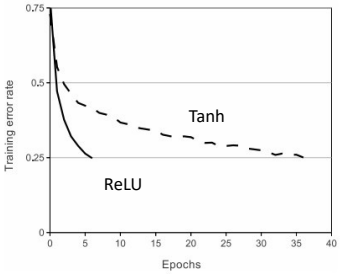
Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



The activation function is usually an abstraction representing the rate of firing in the cell

Activation function – ReLU



- (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.

Example 1

Given an input pattern $\boldsymbol{X}$:

$$\boldsymbol{X} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}$$

The input pattern is received by a convolution layer consisting of one kernel (filter)

$\boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$ and bias = 0.05.

If convolution layer has a sigmoid activation function,

a) Find the outputs of the convolution layer if the padding is VALID at strides = 1
b) Assume the pooling layer uses max pooling, has a pooling window size of 2x2, and strides = 2, find the activations at the pooling layer.
c) Repeat (a) and (b) using convolution layer with SAME padding

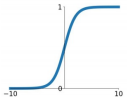
Example 1

Synaptic input to the convolution layer:

$$\boldsymbol{U} = \begin{pmatrix} -0.35 & 1.35 & 0.75 \\ -0.55 & 0.85 & 0.05 \\ 0.75 & 0.05 & 1.15 \end{pmatrix}$$

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



Output of the convolution layer:

$$f(\boldsymbol{U}) = \frac{1}{1+e^{-\boldsymbol{U}}} = \begin{pmatrix} 0.413 & 0.794 & 0.679 \\ 0.366 & 0.701 & 0.512 \\ 0.679 & 0.512 & 0.76 \end{pmatrix}$$

Output of the max pooling layer:

(0.794)

Now find the outputs of the convolution layer if the padding is SAME at strides = 1

See eg6.1.ipynb

Example 1

Find the outputs of the convolution layer if the padding is VALID at strides = 1

$$\boldsymbol{I} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}; \boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Synaptic input to the pooling-layer:

$$u(i,j) = \sum_l \sum_m x(i+l, j+m) w(l,m) + b$$

For VALID padding:

$u(1,1) = 0.5\times0 - 0.1\times1 + 0.2\times1 + 0.8\times1 + 0.1\times0 - 0.5\times1 - 1.0\times1 + 0.2\times1 + 0.0\times0 + 0.05 = -0.35$
$u(1,2) = -0.1\times0 + 0.2\times1 + 0.3\times1 + 0.1\times1 - 0.5\times0 + 0.5\times1 + 0.2\times1 + 0.0\times1 + 0.3\times0 + 0.05 = 1.35$
$u(1,3) = 0.2\times0 + 0.3\times1 + 0.5\times1 - 0.5\times1 + 0.5\times0 + 0.1\times1 + 0.0\times1 + 0.3\times1 - 0.2\times0 + 0.05 = 0.75$
$u(2,1) = 0.8\times0 + 0.1\times1 - 0.5\times1 - 0.1\times1 + 0.2\times0 + 0.0\times1 + 0.7\times1 + 0.1\times1 + 0.2\times0 + 0.05 = -0.55$

⋮

Example 2

Inputs are digit images from MNIST database: http://yann.lecun.com/exdb/mnist/

Input image size = 28x28

First convolution layer consists of three filters:

$$w_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \quad w_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \quad w_3 = \begin{pmatrix} 3 & 4 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 3 \end{pmatrix}$$

Find the feature maps at the the convolution layer and pooling layer. Assume zero bias

For convolution layer, use a stride = 1 (default) and padding = 'VALID'.

For pooling layer, use a window of size 2x2 and a stride if 2.

See eg6.2.ipynb

Example 2

Original Image 28 x 28

Output of convolution layer
3 x 26 x 26

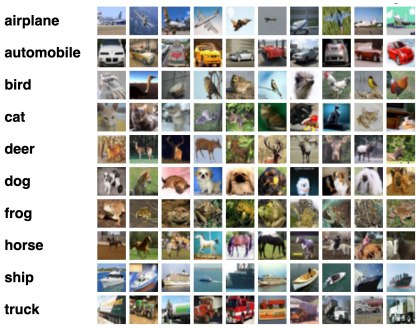
Output of pooling layer
3 x 13 x 13

Training a Classifier

Outline

- Basic components in CNN
- Training a classifier
- Optimizers

Multi-class classification



CIFAR-10

- 10 classes
- 6000 images per class
- 60000 images - 50000 training images and 10000 test images
- Each image has a size of 3x32x32, that is 3-channel color images of 32x32 pixels in size

https://www.cs.toronto.edu/~kriz/cifar.html

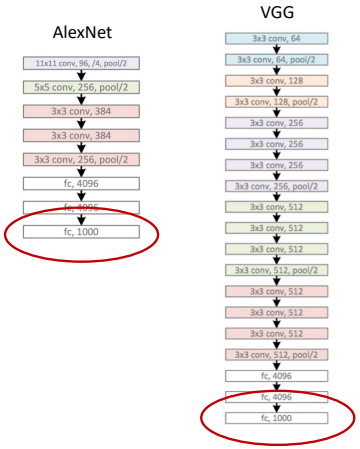
Multi-class classification

MNIST

- Size-normalized and centred 1x28x28 =784 inputs
- Training set = 60,000 images
- Testing set = 10,000 images

http://yann.lecun.com/exdb/mnist/

Softmax function



Where does the Softmax function fit in a CNN architecture?

Softmax's input is the output of the fully connected layer immediately preceding it, and it outputs the final output of the entire neural network. This output is a probability distribution of all the label class candidates.

Softmax function

The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $\boldsymbol{z} \in \mathbb{R}^n$ and outputs a vector of output probability $\boldsymbol{p} \in \mathbb{R}^n$ through a softmax function at the end of the architecture.

exp(2)/(exp(2)+exp(1)+exp(0.1))

$$\boldsymbol{z} \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix} \longrightarrow \boxed{S(z_i) = \frac{e^{z_j}}{\sum_j e^{z_j}}} \longrightarrow \begin{matrix} p = 0.66 \\ p = 0.24 \\ p = 0.10 \end{matrix}$$

Numeric output of the last linear layer of a multi-class classification neural network

Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label

Cross entropy loss

One-hot encoded ground truth

Example:

Class	One-hot vector
Dog	[1 0 0]
Cat	[0 1 0]
Bird	[0 0 1]

Cross entropy loss

$\hat{y}$: 0.66, 0.24, 0.10 — **Cross entropy** $L(\hat{y}, y)$ → y: 1.0, 0.0, 0.0

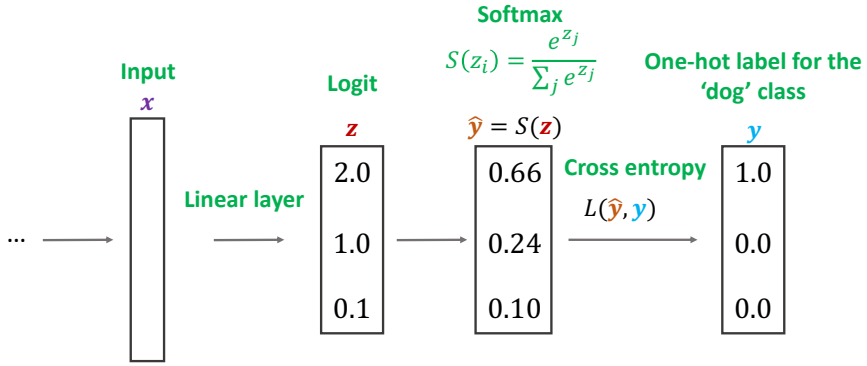
$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.66)) + (0 \times \log_2(0.24)) + (0 \times \log_2(0.10))]$$
$$= 0.6$$

$\hat{y}$: 0.10, 0.66, 0.24 — **Cross entropy** $L(\hat{y}, y)$ → y: 1.0, 0.0, 0.0

$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$
$$= -[(1 \times \log_2(0.10)) + (0 \times \log_2(0.66)) + (0 \times \log_2(0.24))]$$
$$= 3.32$$

What is the min / max possible loss?

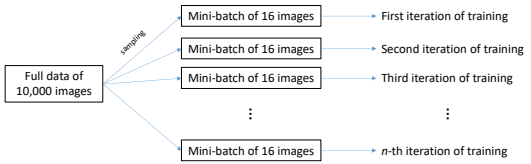
Cross entropy loss



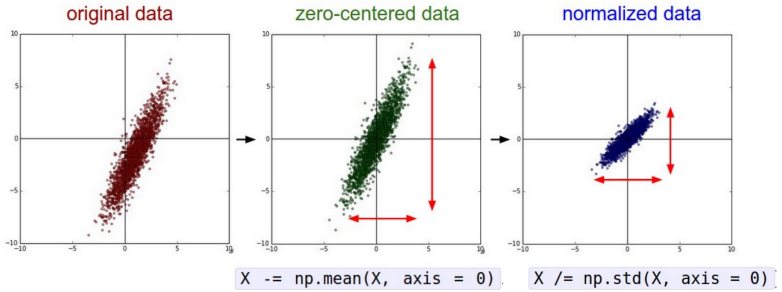
$$L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$

Epoch and mini-batch

- **Epoch** – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.
- **Mini-batch** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on mini batches, where the number of data points in a batch is a hyperparameter that we can tune.



Data pre-processing



(Assume X [NxD] is data matrix,
each example in a row)

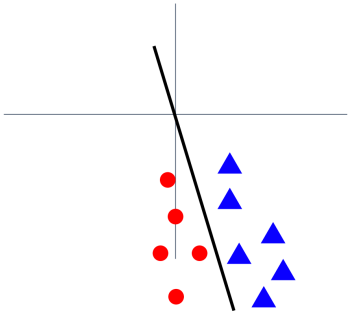
Data pre-processing for images

e.g. consider CIFAR-10 example with [32,32,3] images

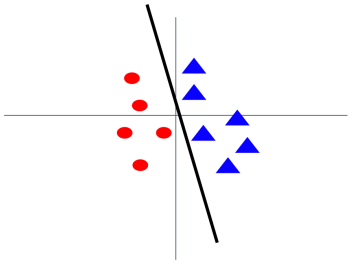
- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Data pre-processing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Outline

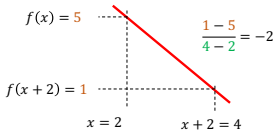
- Basic components in CNN
- Training a classifier
- Optimizers

Optimizers

Optimization

- In 1-dimension, the derivative of a function gives the slope:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension
- The direction of steepest descent is the **negative gradient**

Optimization

- A CNN as composition of functions
  $f_{\boldsymbol{w}}(\boldsymbol{x}) = f_L(\dots\ (f_2(f_1(\boldsymbol{x}; \boldsymbol{w}_1); \boldsymbol{w}_2)\ \dots; \boldsymbol{w}_L)$
- Parameters
  $\boldsymbol{w} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \dots \boldsymbol{w}_L)$
- Empirical loss function
  $L(\boldsymbol{w}) = \frac{1}{n}\sum_i l(y_i, f_{\boldsymbol{w}}(\boldsymbol{x}_i))$

$$\boldsymbol{w}^* = \text{argmin}_{\boldsymbol{w}}\, L(\boldsymbol{w})$$

Random search is a bad idea

Follow the slope

Gradient descent (GD)

- Iteratively step in the direction of the negative gradient
- Gradient descent

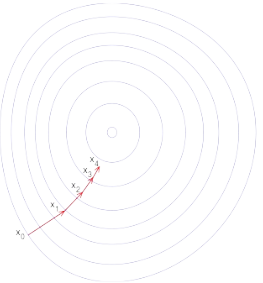
$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t \frac{\partial \boldsymbol{f}(\boldsymbol{w}^t)}{\partial \boldsymbol{w}}$$

New weight

Old weight

Learning rate

Gradient



Gradient descent (GD)

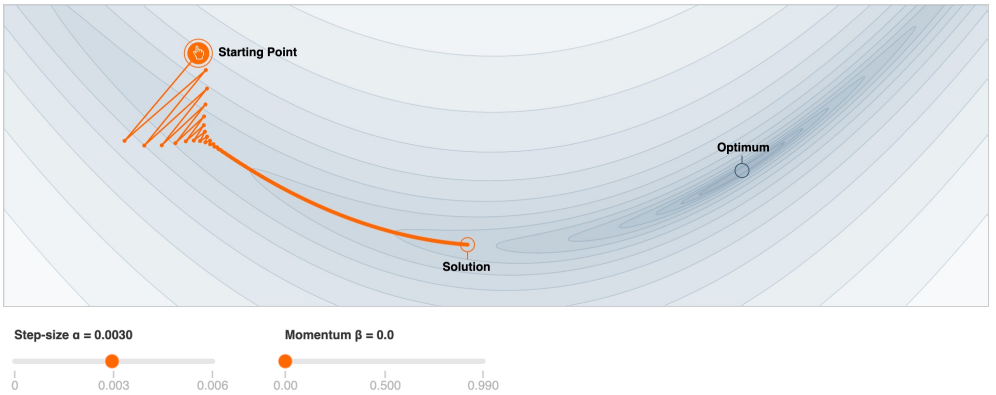
- Batch Gradient Descent
  - Full sum is *expensive* when N is large

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

- Stochastic Gradient Descent (SGD)
  - Approximate sum using a minibatch of examples
  - 32 / 64 / 128 common minibatch size
  - Additional hyperparameter on batch size

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

GD with Momentum



Why Momentum Really Works: https://distill.pub/2017/momentum/

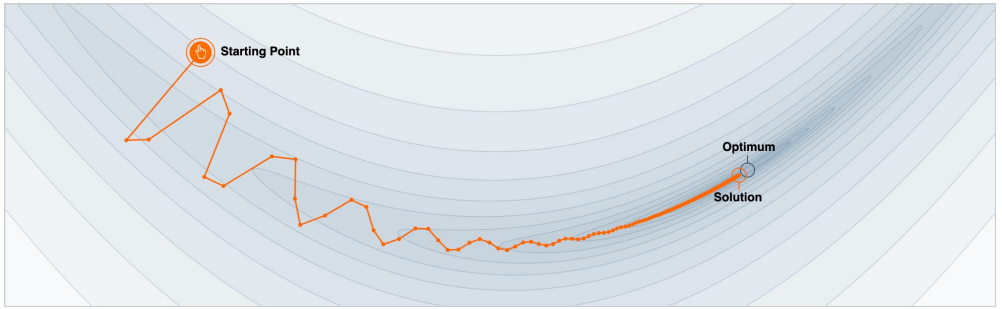
GD with Momentum

Deep neural networks have very complex error profiles. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. This problem is typical in deep learning architecture.

Momentum is one method of speeding the convergence along a narrow ravine.

GD with Momentum



Why Momentum Really Works: https://distill.pub/2017/momentum/

GD with Momentum

Momentum update is given by:

$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

where $\boldsymbol{V}$ is known as the velocity term and has the same dimension as the weight vector $\boldsymbol{W}$.

The momentum parameter $\gamma \in [0,1]$ indicates how many iterations the previous gradients are incorporated into the current update.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

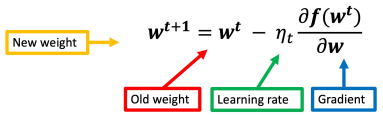
Learning rate

- **Adaptive learning rates**
  - Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution.
  - While Adam optimizer is the most commonly used technique, others can also be useful.
- **Algorithms with adaptive learning rates:**
  - AdaGrad torch.optim.Adagrad()
  - RMSprop torch.optim.RMSprop()
  - Adam torch.optim.Adam()

Adam: A Method for Stochastic Optimization, ICLR 2014

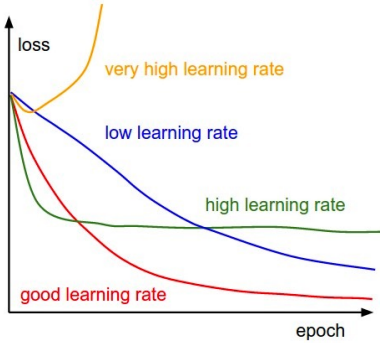
https://pytorch.org/docs/stable/optim.html

Learning rate



Learning rate
The learning rate, often noted α or sometimes η, indicates at which pace the weights get updated. It can be fixed or adaptively changed.

The current most popular method is called Adam, which is a method that adapts the learning rate.



Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to start with a large learning factor and then gradually reducing it.

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and $\alpha(\infty) = 0$.

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.

Learning trajectory of a neural network minimizing non-convex cost function passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. This improves the learning rates, especially in the convex regions of error function.

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + (\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

In other words, learning rate:

$$\tilde{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region.

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\sqrt{\varepsilon + \boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

The decay constant ρ controls the length of the moving average of gradients.

Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer combines RMSprop and momentum methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

Momentum term: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\nabla_{\boldsymbol{W}} J$

Learning rate term: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)(\nabla_{\boldsymbol{W}} J)^2$

$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1}$$
$$\boldsymbol{r} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{s}$$

Note that $\boldsymbol{s}$ adds the momentum and $\boldsymbol{r}$ contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001, \rho_1 = 0.9, \rho_2 = 0.999$, and $\epsilon = 10^{-8}$

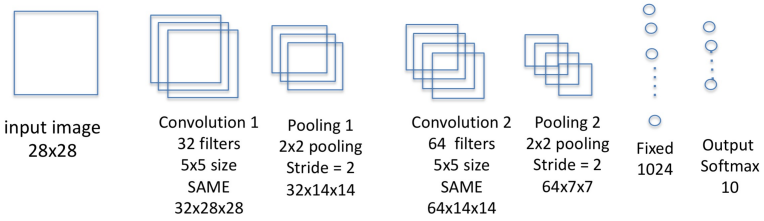
Example 3: MNIST digit recognition

MNIST database: 28x28 =784 inputs

Training set = 12000 images
Testing set = 2000 images

Input pixel values were normalized to [0, 1]

Example 3: Architecture of CNN



ReLU neurons
Gradient descent optimizer with batch-size = 128
Leaning parameter = 10^{-3}

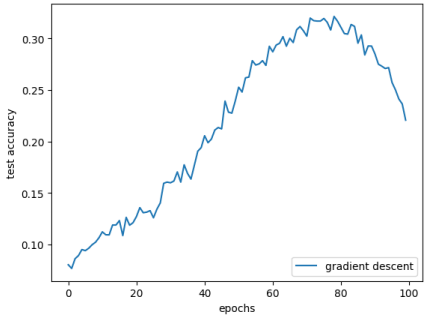
See eg6.3.ipynb

Example 3: Weights learned

Weights learned at convolution layer 1

32x5x5

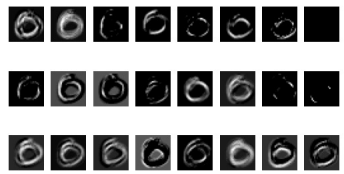
Example 3: Training Curve



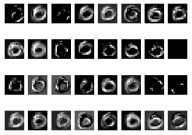
Example 3: Feature maps

Input image
28x28

Feature maps at conv1
32x28x28

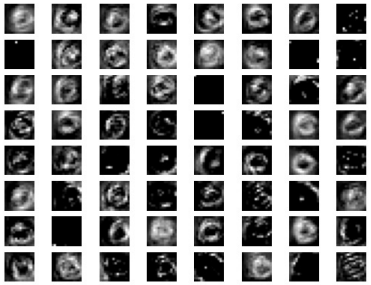


Feature maps at pool1
32x14x14

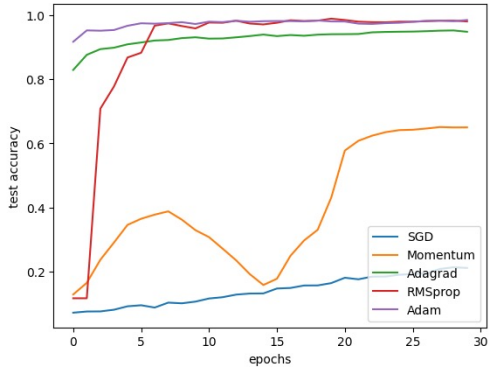


Example 3: Feature maps

Feature maps at conv2
64x14x14



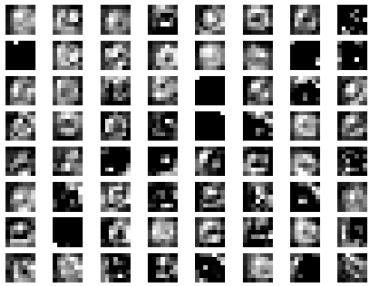
Example 4: MNIST recognition with CNN with different learning algorithms



See eg6.4.ipynb

Example 3: Feature maps

Feature maps at pool2
64x7x7

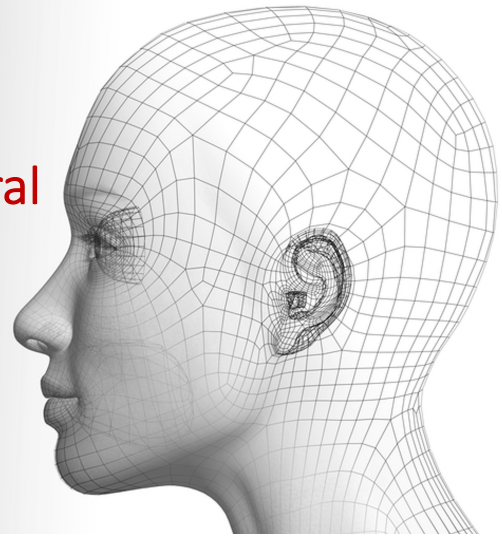


SC 4001 CE/CZ 4042 Neural Network and Deep Learning
Last update: 13 Mar 2024

Convolutional Neural Networks II

Xingang Pan
潘新钢

https://xingangpan.github.io/
https://twitter.com/XingangP



Outline

- CNN Architectures
- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
- Batch normalization
- Prevent overfitting
  - Transfer learning
  - Data augmentation

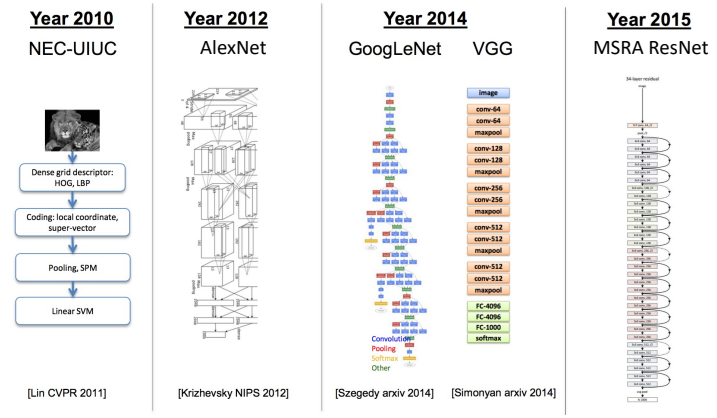
You learn some classic architectures

You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network

You learn an important technique to improve the training of modern neural networks

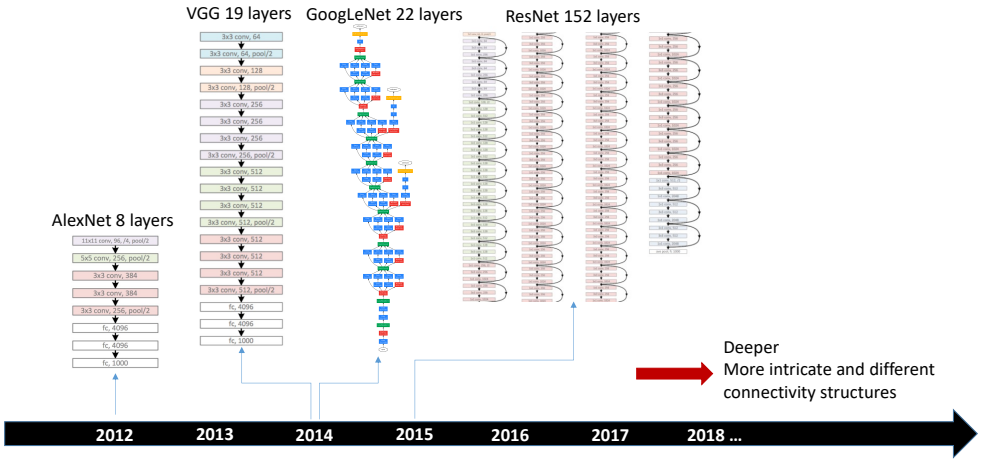
You learn two important techniques to prevent overfitting in neural networks

Deep networks for ImageNet

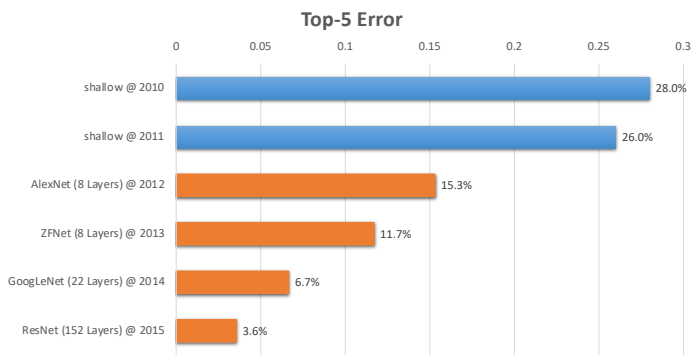


CNN Architectures

Deep architectures

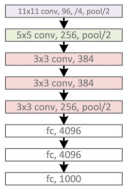


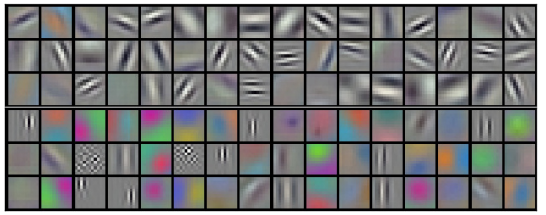
Performance of previous years on ImageNet



Depth is the key to high classification accuracy.

Deep architectures - AlexNet

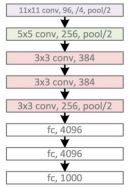


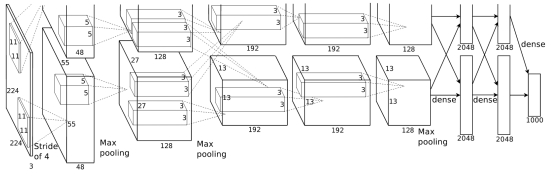


96 kernels learned by first convolution layer; 48 kernels were learned by each GPU

2012 2013 2014 2015 2016 2017 2018 ...

Deep architectures - AlexNet

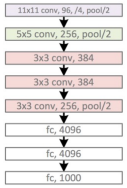


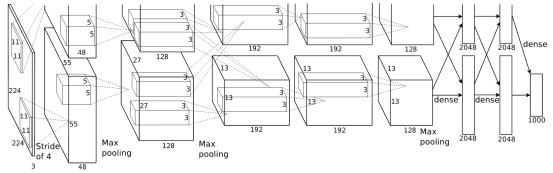


- The split (i.e. two pathways) in the image above are the split between two GPUs.
- Trained for about a week on two NVIDIA GTX 580 3GB GPU
- 60 million parameters
- Input layer: size 227x227x3
- 8 layers deep: 5 convolution and pooling layers and 3 fully connected layers

2012 2013 2014 2015 2016 2017 2018 ...

Deep architectures - AlexNet

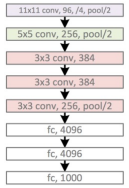




- Escape from a few layers
  - ReLU nonlinearity for solving gradient vanishing
  - Data augmentation
  - Dropout
  - Outperformed all previous models on ILSVRC by 10%

2012 2013 2014 2015 2016 2017 2018 ...

Deep architectures - AlexNet



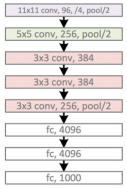
Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels

Number of parameters = (11x11x3 + 1)*96 = 34,944

Note: There are no parameters associated with a pooling layer. The pool size, stride, and padding are hyperparameters.

Deep architectures - AlexNet

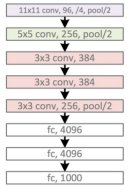


Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256
Size (width) of the output image of the previous Conv Layer = 6

Number of parameters = (6x6x256*4096)+4096 = 37,752,832

Deep architectures - AlexNet

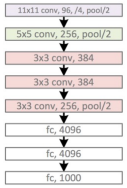


Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

Second convolution layer: 256 kernels of size 5x5x96

Number of parameters = (5x5x96 + 1)*256 = 614,656

Deep architectures - AlexNet



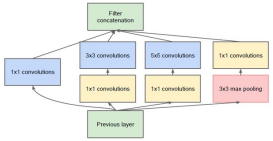
Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	**62,378,344**

The last FC layer:
Number of neurons = 1000
Number of neurons in the previous FC Layer = 4096

Number of parameters = (1000*4096)+1000 = 4,097,000

Deep architectures - GoogLeNet

- An important lesson - go deeper
- Inception structures (v2, v3, v4)
  - Reduce parameters (4M vs 60M in AlexNet)



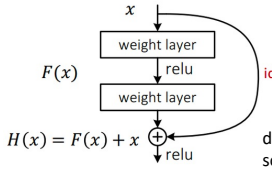
The 1x1 convolutions are performed to reduce the dimensions of input/output

- Batch normalization
  - Normalizes the activation for each training mini-batch
  - Allows us to use much higher learning rates and be less careful about initialization

2012 2013 2014 2015 2016 2017 2018 …

Deep architectures - ResNet

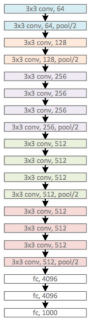
- An important lesson - go deeper
- Escape from 100 layers
  - Residual learning



drives the new layer to learn something different

2012 2013 2014 2015 2016 2017 2018 …

Deep architectures - VGG

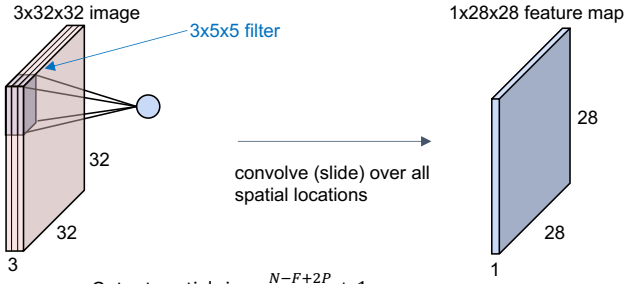


- An important lesson - go deeper
- 140M parameters
- Now commonly used for computing perceptual loss

2012 2013 2014 2015 2016 2017 2018 …

More on Convolution

Recap: How to calculate the spatial size of output?



Output spatial size = $\frac{N-F+2P}{S}+1$

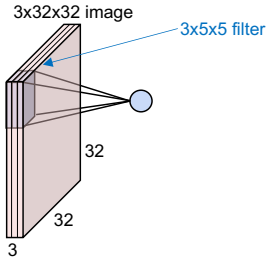
In this example, $N = 32, F = 5, P = 0, S = 1$

Thus, output spatial size = $\frac{32-5+2(0)}{1}+1 = 28$

Recap: How to calculate the number of parameters?

Input volume: 3×32×32
Ten 3×5×5 filters with stride 1, pad 0

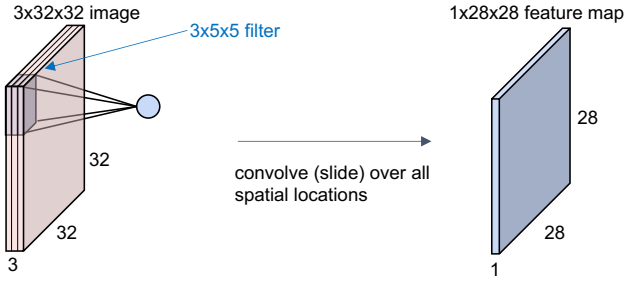
Number of parameters in this layer: ?



Each filter has 5×5×3 + 1 = 76 params (+1 for bias)

Ten filters so 76×10 = **760 parameters**

Recap: How to calculate the number of parameters?



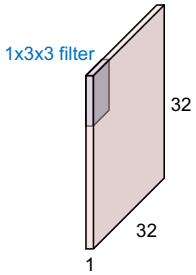
Let say we have ten 3×5×5 filters with stride 1, pad 0

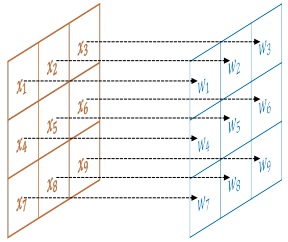
How to calculate the computations involved?

Let's focus on one input channel first

$$u(i,j) = \sum_{l=-a}^{a}\sum_{m=-b}^{b} x(i+l, j+m)w(l,m) + b$$

Element-wise multiplication of input and weights





x_1w_1
x_2w_2
x_3w_3
x_4w_4
x_5w_5
x_6w_6
x_7w_7
x_8w_8
x_9w_9

How many multiplication operations?

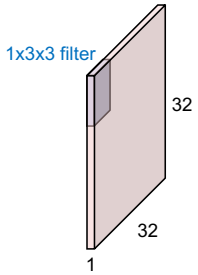
In general, there are F^2 multiplication operations, where F is the filter spatial size

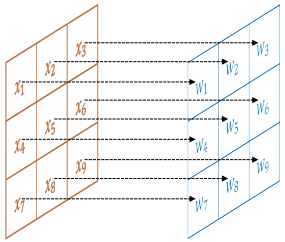
How to calculate the computations involved?

Let's focus on one input channel first

$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m) w(l,m) + b$$

Element-wise multiplication of input and weights





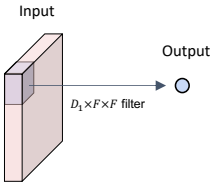
$$x_1w_1 + x_2w_2 + x_3w_3 + x_4w_4 + x_5w_5 + x_6w_6 + x_7w_7 + x_8w_8 + x_9w_9$$

How many adding operations?

Adding the elements after multiplication, we need $n-1$ adding operations for n elements

In general, there are $F^2 - 1$ adding operations, where F is the filter spatial size

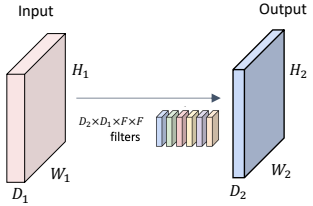
How to calculate the computations involved?



If we have only one filter, and apply it to generate output of one spatial location, the total operations

The total cost is

$$(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$$



If we have D_2 filter, and apply it to generate output of $H_2 \times W_2$ spatial location, the total operations

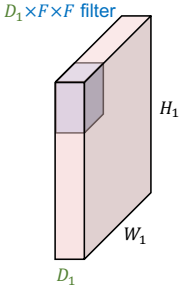
The total cost is

$$[(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$

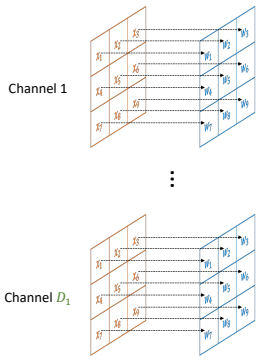
$$= (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

How to calculate the computations involved?

Let's say we have D_1 input channels now



Element-wise multiplication of input and weights, for each channel



F^2 multiplication operations
$F^2 - 1$ adding operations

F^2 multiplication operations
$F^2 - 1$ adding operations

If we have only one filter, and apply it to generate output of one spatial location, the total operations

The total cost is

$$(D_1 \times F^2) + (D_1 \times F^2 - 1)$$

Let's not forget to add the bias

$$(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$$

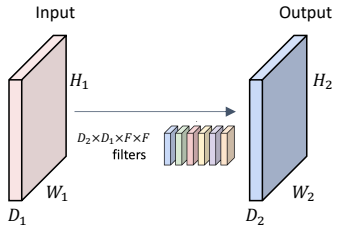
Summary

FLOPs (floating point operations)

Not FLOPS (floating point operations per second)

Assume:
- Filter size F
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Produces a output volume of size $D_2 \times H_2 \times W_2$

The FLOPs of the convolution layer is given by

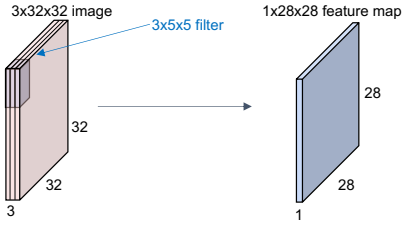


$$\text{FLOPs} = \underbrace{[(D_1 \times F^2)}_{\text{Elementwise multiplication of each filter on a spatial location}} + \underbrace{(D_1 \times F^2 - 1)}_{\text{Adding the elements after multiplication, we need } n-1 \text{ adding operations for } n \text{ elements}} + \underbrace{1}_{\text{Add the bias}}] \times D_2 \times H_2 \times W_2 = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

Try this

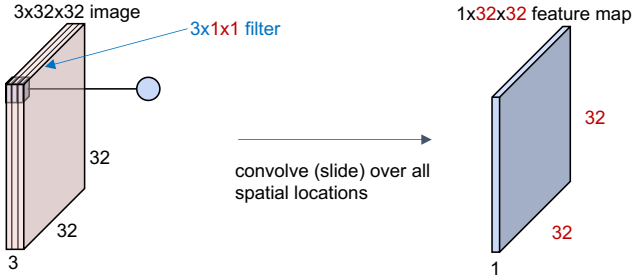
Assume:
- One filter with spatial size of 5×5
- Accepts a volume of size $3\times32\times32$
- Produces a output volume of size $1\times28\times28$

The FLOPs of the convolution layer is given by?



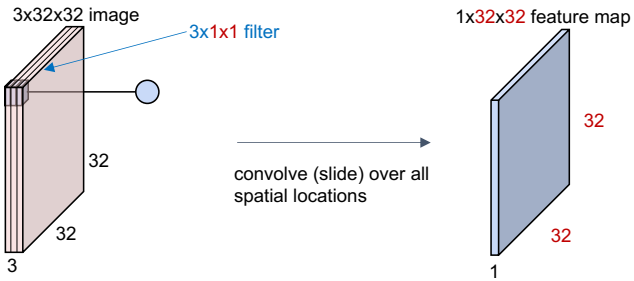
Pointwise convolution

- Why having filter of spatial size 1×1?
  - Change the size of channels
  - "Blend" information among channels by linear combination



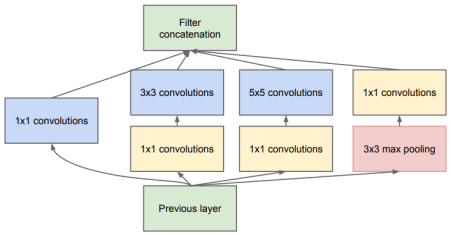
Pointwise convolution

- We have seen convolution with spatial size of 3×3, 5×5, 7×7
  - Can we have other sizes?
  - Can we have filter of spatial size 1×1?



Pointwise convolution

- A real-world example
  - 1×1 convolutions are used for compute reductions before the expensive 3×3 and 5×5 convolutions

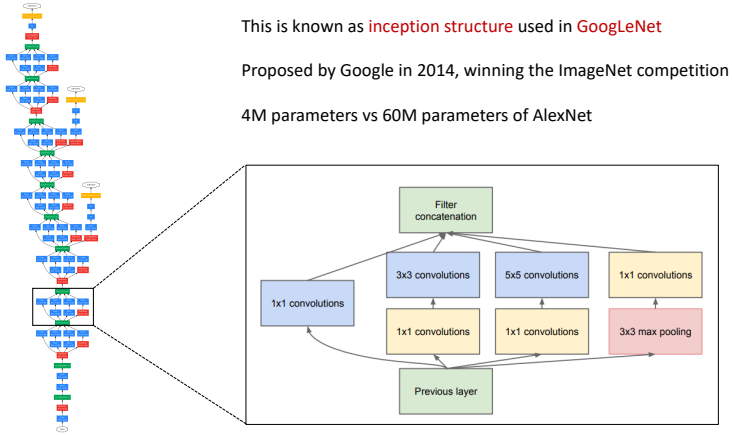


E.g., Given the output from the previous layer, reduce the number of channels of from $256\times32\times32$ to $128\times32\times32$ before the 5×5 convolutions

Referring to the FLOPs equation, reducing input channels help reduce the FLOPs

$$\text{FLOPs} = (2\times D_1\times F^2)\times D_2\times H_2\times W_2$$

Pointwise convolution



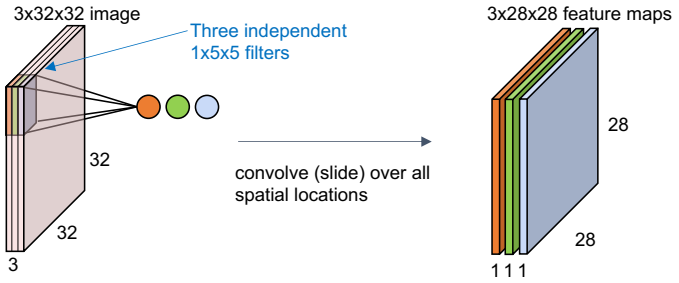
This is known as inception structure used in GoogLeNet

Proposed by Google in 2014, winning the ImageNet competition that year

4M parameters vs 60M parameters of AlexNet

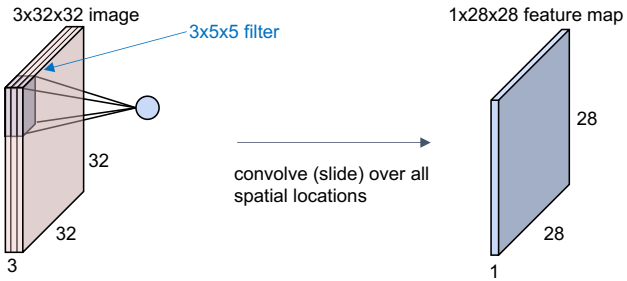
Depthwise convolution

- Depthwise convolution
  - Convolution is performed independently for each of input channels

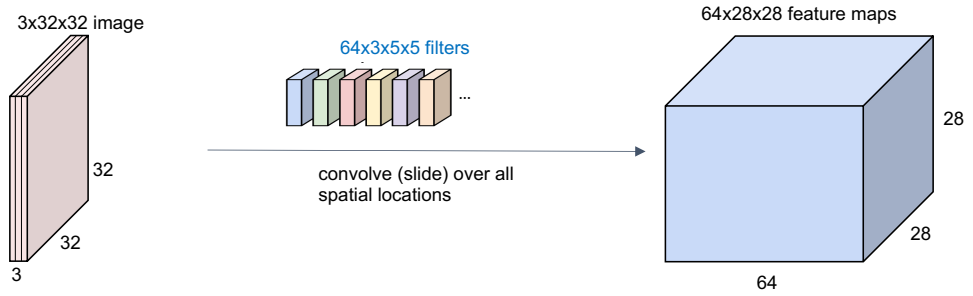


Standard convolution

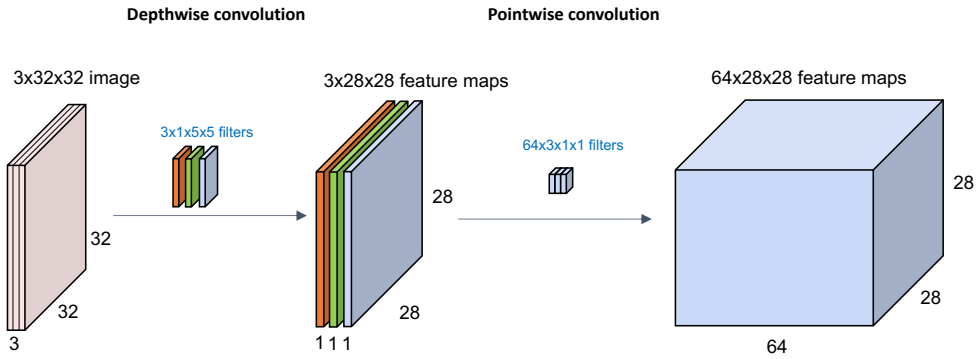
- Standard convolution
  - The input and output are locally connected in spatial domain
  - In channel domain, they are fully connected



Standard convolution



Depthwise convolution + Pointwise convolution



Depthwise convolution + Pointwise convolution

- Why replacing standard convolution with depthwise convolution + pointwise convolution?
  - The computational cost reduction rate is roughly 1/8–1/9 at only a small reduction in accuracy
  - Good if you want to deploy small networks on devices with CPU
  - Used in network such as MobileNet

G. Howard et al., MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, CVPR 2017

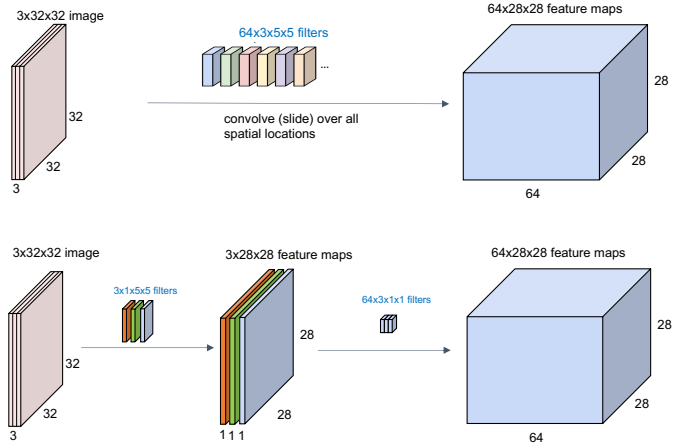
Depthwise convolution + Pointwise convolution

Replace standard convolution

with

Depthwise convolution + pointwise convolution

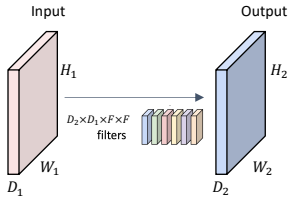
And we still get the same size of output volume!



Depthwise convolution + Pointwise convolution

- Standard convolution cost

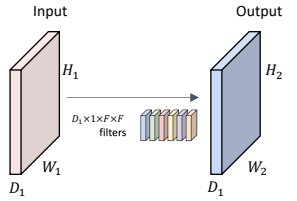
$$\text{FLOPs}_{\text{standard}} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$



- Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = [(1 \times F^2) + (1 \times F^2 - 1) + 1] \times D_1 \times H_2 \times W_2$$

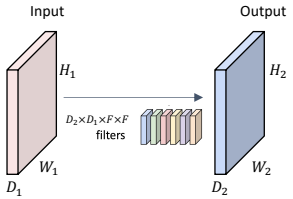
  - Each filter is applied only to one channel
  - The number of output channels equals to the number of input channels



Depthwise convolution + Pointwise convolution

- Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$



- Pointwise convolution cost

$$\text{FLOPs}_{\text{pointwise}} = [(D_1 \times 1) + (D_1 \times 1 - 1) + 1] \times D_2 \times H_2 \times W_2$$

  - The filter spatial size is 1×1

Depthwise convolution + Pointwise convolution

- How much computation do you save by replacing standard convolution with depthwise+pointwise?

$$\text{Ratio} = \frac{\text{Cost of depthwise convolution} + \text{pointwise convolution}}{\text{Cost of standard convolution}}$$

$$\text{Ratio} = \frac{(2 \times F^2) \times D_1 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2} + \frac{(2 \times D_1) \times D_2 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2}$$

$$\text{Ratio} = \frac{1}{D_2} + \frac{1}{F^2}$$

D_2 is usually large. Reduction rate is roughly 1/8–1/9 if 3×3 depthwise separable convolutions are used

Depthwise convolution + Pointwise convolution

- Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

- Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = (2 \times F^2) \times D_1 \times H_2 \times W_2$$

- Pointwise convolution cost

$$\text{FLOPs}_{\text{pointwise}} = (2 \times D_1) \times D_2 \times H_2 \times W_2$$

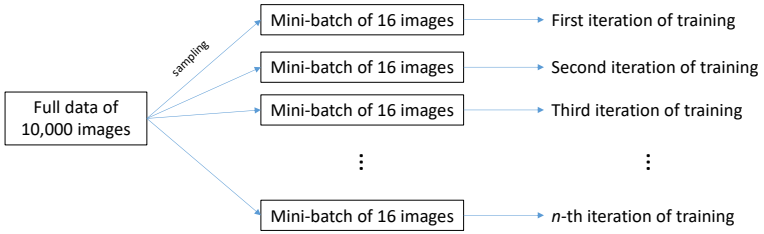
Summary

- More on convolutions
  - You learn how to calculate computation complexity of convolutional layer
  - Pointwise convolution can be used to change the size of channels. This can be used to achieve channel reduction and thus saving computational cost
  - Depthwise convolution + Pointwise convolution yields lower computations than standard convolution

Batch Normalization

Mini-batch

- What is mini-batch?
  - A subset of all data during one iteration to compute the gradient



GoogLeNet

Proposed by Google in 2014, winning the ImageNet competition that year

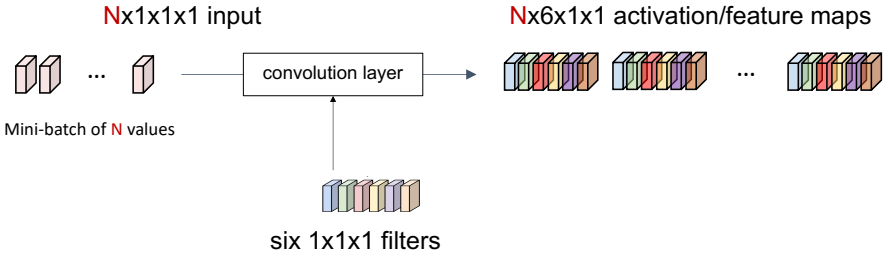
Apart from the inception structure, there is another important technique called **batch normalization**

Problem: deep networks are very hard to train!

Main idea: "Normalize" the outputs of a layer so they have zero mean and unit variance

Effect: allowing higher learning rates and reducing the strong dependence on initialization

Mini-batch



Batch Normalization

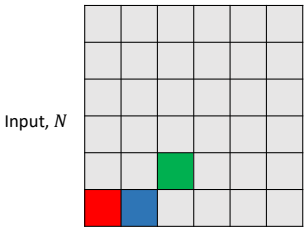
Let's arrange the activations of the mini batch in a matrix of $N \times D$

Nx6x1x1 activation/feature maps

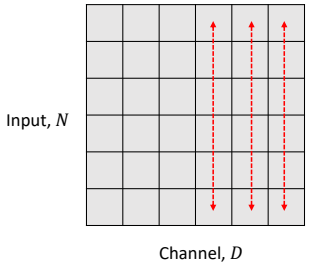
Activation of the first input in the mini-batch

Activation of the second input in the mini-batch

Activation of the N-th input in the mini-batch



Batch Normalization - Training



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

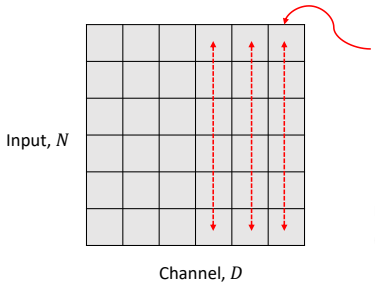
Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the **normalized** values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

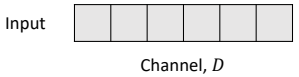
Batch Normalization



The goal of batch normalization is to normalize the values across each column so that the values of the column have zero mean and unit variance

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below

Batch Normalization – Test Time



Problem: Estimates depend on minibatch; can't do this at test-time

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

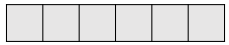
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization – Test Time

Input



Channel, D

Average of values seen during training

Per-channel mean, shape is $1 \times D$

Average of values seen during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization

Batch Normalization for **fully-connected** networks

```
x: N × D
Normalize ↓
μ,σ: 1 × D
γ,β: 1 × D
y = γ(x-μ)/σ+β
```

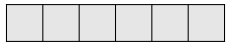
Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

```
x: N×C×H×W
Normalize ↓  ↓  ↓
μ,σ: 1×C×1×1
γ,β: 1×C×1×1
y = γ(x-μ)/σ+β
```

Normalize also on the spatial dimensions

Batch Normalization – Test Time

Input



Channel, D

During testing batchnorm becomes a linear operator! Can be fused with the previous fully-connected or conv layer

Average of values seen during training

Per-channel mean, shape is $1 \times D$

Average of values seen during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

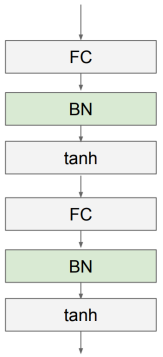
Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

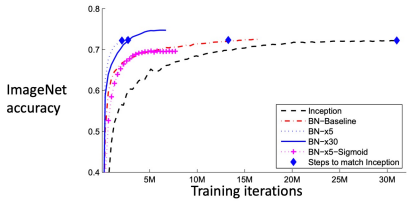
Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

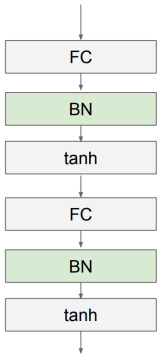
Batch Normalization



- It is usually done after a fully connected/convolutional layer and before a non-linearity layer
- Zero overhead at test-time: can be fused with conv
- Allows higher learning rates, faster convergence
- Networks becomes more robust to initialization



Batch Normalization



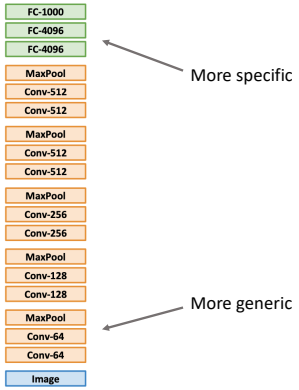
- It is usually done after a fully connected/convolutional layer and before a non-linearity layer
- Zero overhead at test-time: can be fused with conv
- Allows higher learning rates, faster convergence
- Networks becomes more robust to initialization
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a very common source of bugs!

Why overfitting?

- This happens when our model is too complex and too specialized on a small number of training data.
- Increase the size of the data, remove outliers in data, reduce the complexity of the model, reduce the feature dimension

Prevent Overfitting

Transfer learning



In a network with an N-dimensional softmax output layer that has been successfully trained toward a supervised classification objective, each output unit will be specific to a particular class

When trained on images, deep networks tend to learn first-layer features that resemble either Gabor filters or color blobs. These first-layer features are general.

Yosinski et al., How transferable are features in deep neural networks? NIPS'14

Transfer learning

Pre-training + Fine-tuning

Step 1: Pre-training on large-scale dataset like ImageNet

Transfer weights

Step 2: Use pre-trained network as initialization

Remove last layer

Freeze these

Step 3: Fine-tuning

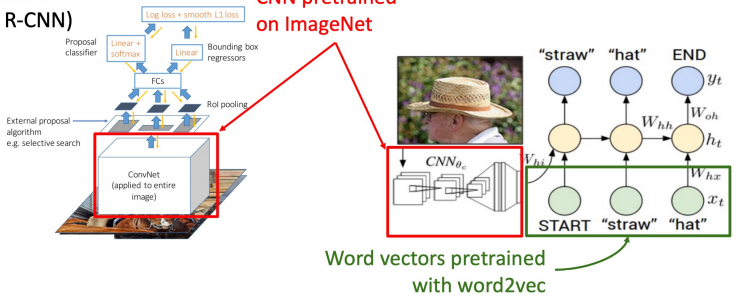
Add new layer correspond to the target class number

Train on new data

Transfer learning

Transfer learning is pervasive

Object Detection (Fast R-CNN)

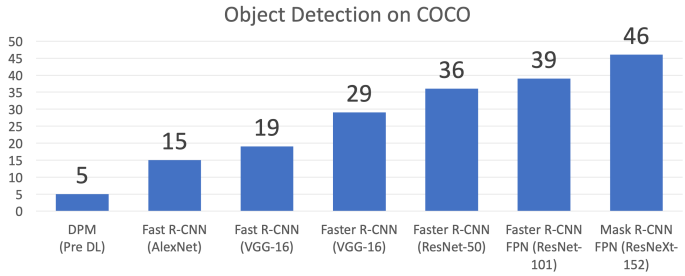


Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

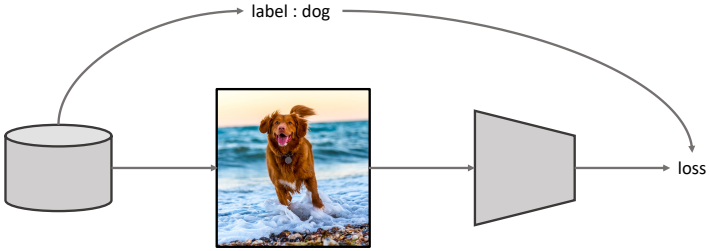
Transfer learning

Architecture matters

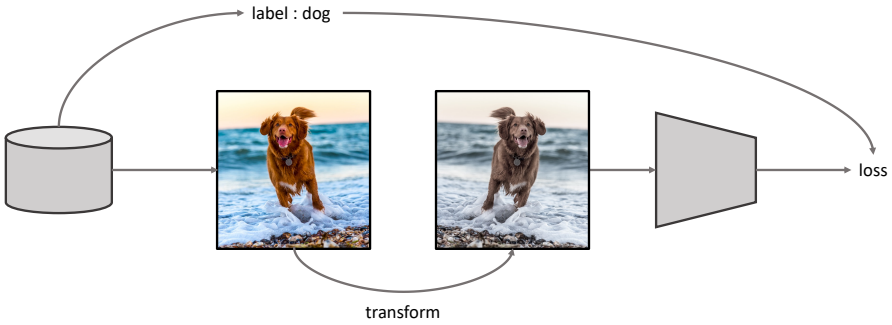


Ross Girshick, "The Generalized R-CNN Framework for Object Detection", ICCV 2017 Tutorial on Instance-Level Visual Recognition

Data augmentation



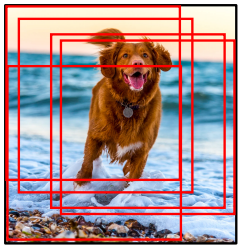
Data augmentation



Data augmentation

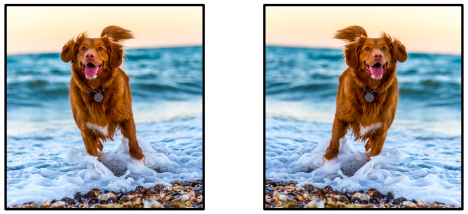
Random crops and scales

Training:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data augmentation

Horizontal flip



Data augmentation

Color jitter

Simple :
1. Randomize contrast and brightness

Complex:
1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(Used in AlexNet, ResNet, etc.)

Data augmentation

There are many more data augmentation schemes

Random mix/combinations of:
translation - rotation - stretching - shearing, - lens distortions ….

Ren Wu et al., Deep Image: Scaling up Image Recognition, https://arxiv.org/pdf/1501.02876v2.pdf?

Last week

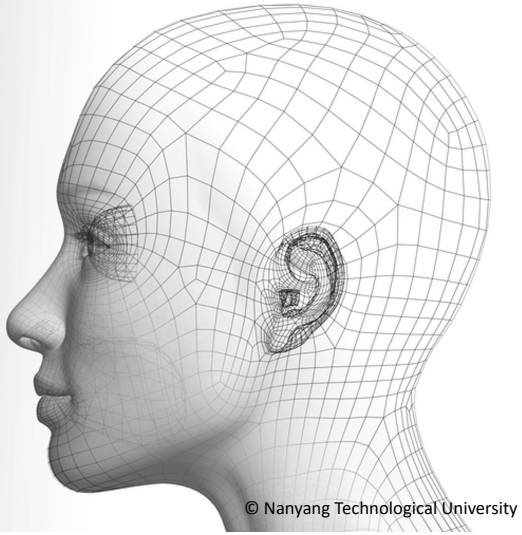
- CNN Architectures
  - You learn some classic architectures
- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution
  - You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network
- Batch normalization
  - You learn an important technique to improve the training of modern neural networks
- Prevent overfitting
  - Transfer learning
  - Data augmentation
  - You learn two important techniques to prevent overfitting in neural networks

SC 4001 CE/CZ 4042 Neural Network and Deep Learning
Last update: 13 Mar 2024

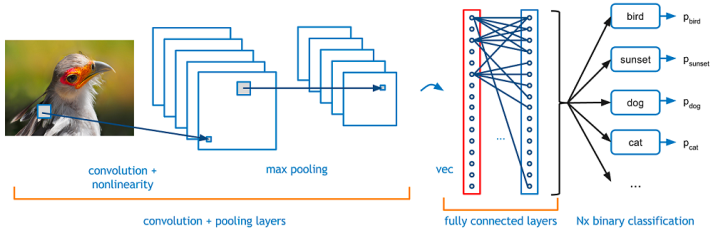
Recurrent Neural Networks (RNN)

Xingang Pan
潘新钢

https://xingangpan.github.io/
https://twitter.com/XingangP



Previous: Convolutional Neural Networks (CNN)



How about sequential information?

- Turn a sequence of sound pressures into a sequence of word identities?
- Speech recognition?
- Video prediction?

Recurrent Neural Network (RNN)

Outline

- Recurrent Neural Network (RNN)
  - Hidden recurrence
  - Top-down recurrence
- Long Short-Term Memory (LSTM)
  - Long-term dependency
  - Structure of LSTM
- Example Applications

Recurrent Neural Networks (RNN)

Recurrent neural networks (RNN) are designed to process sequential information. That is, the data presented in a sequence.

The next data point in the sequence is usually dependent on the current data point.

Examples:

- Natural language processing (spoken words and written sentences). The next word in a sentence depends on the word which comes before it.
- Genomic sequences: a nucleotide in a DNA sequence is dependent on its neighbors.

RNN attempts to capture dependency among the data points in the sequence.

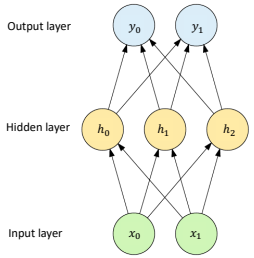
Text Generation with an RNN

QUEENE:
I had thought thou hadst a Roman; for the oracle,
Thus by All bids the man against the word,
Which are so weak of care, by old care done;
Your children were in your holy love,
And the precipitation through the bleeding throne.

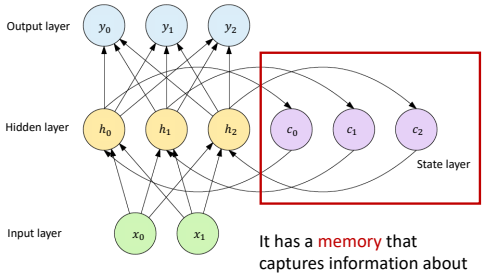
BISHOP OF ELY:
Marry, and will, my lord, to weep in such a one were prettiest;
Yet now I was adopted heir
Of the world's lamentable day,
To watch the next way with his father with his face?

Recurrent Neural Networks (RNN)

Feedforward NN



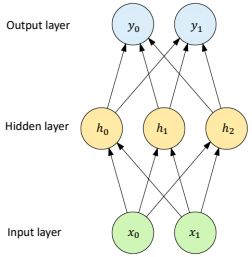
RNN with hidden recurrence



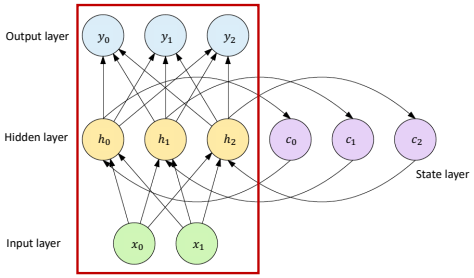
It has a memory that captures information about what has been processed so far.

Recurrent Neural Networks (RNN)

Feedforward NN



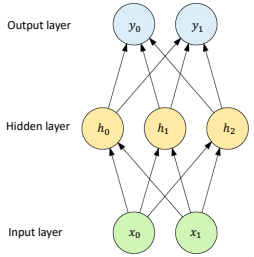
RNN with hidden recurrence



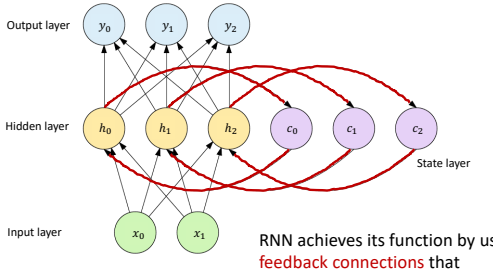
RNNs are called recurrent because they perform the same task for every data element (frame) of a sequence, with the output depending on the previous computations.

Recurrent Neural Networks (RNN)

Feedforward NN



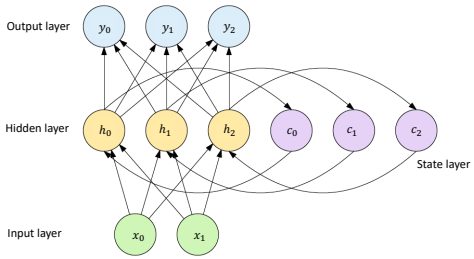
RNN with hidden recurrence



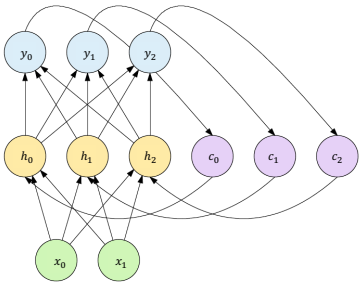
RNN achieves its function by using feedback connections that enables learning of sequential (temporal) information of sequences.

Types of RNN

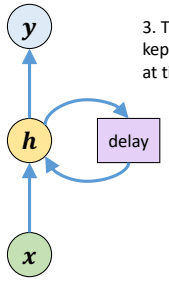
RNN with hidden recurrence (Elman-type)



RNN with top-down recurrence (Jordan-type)



RNN with hidden recurrence (Elman type)



3. The hidden-layer activation at time $t-1$ is kept by the **delay unit** and fed to the hidden layer at time t together with the raw input $\boldsymbol{x}(t)$.

4. The **delay unit** represents that the activation is held for one time unit until the next time instance.

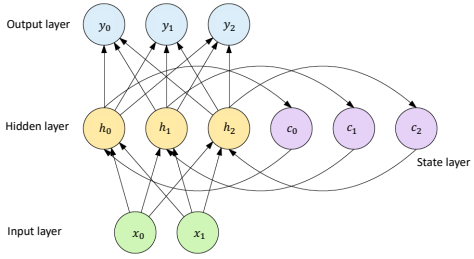
Here, one unit represents the time between two adjacent data points in the sequence .

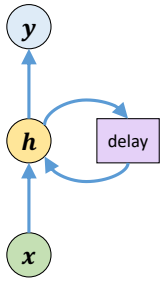
2. Activations are updated at each time instant t.

1. Data is presented as a sequence of instance t (time) that is discretized

RNN with hidden recurrence (Elman type)

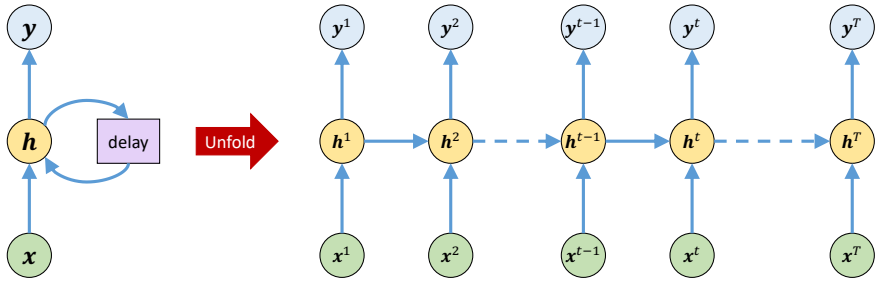
RNN with hidden recurrence (Elman-type)





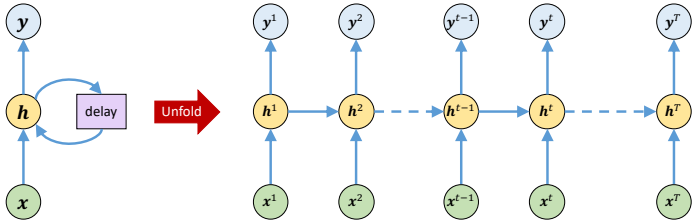
Elman type networks (sometimes called a "Vanilla RNN") that produce an output at each time step and have recurrent connections between hidden units.

RNN with hidden recurrence



The recurrent connections in the hidden-layer can be unfolded to process sequences of arbitrary length.

RNN with hidden recurrence



By considering the unfolded structure, $\boldsymbol{h}(t)$ is dependent on all the inputs at time t and before time t:

$$\boldsymbol{h}(t) = f^t(\boldsymbol{x}(t), \boldsymbol{x}(t-1), \dots, \boldsymbol{x}(2), \boldsymbol{x}(1))$$

The function f^t takes the whole past sequence ,$(\boldsymbol{x}(t), \boldsymbol{x}(t-1), \dots, \boldsymbol{x}(2), \boldsymbol{x}(1))$ as input and produce the hidden layer activation.

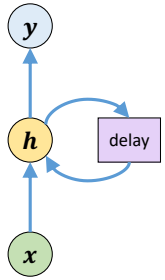
RNN with hidden recurrence

The folded structure introduces two major advantages:

new state $\boldsymbol{h}(t) = f(\boldsymbol{h}(t-1), \boldsymbol{x}(t))$

some function — old state — input vector at some time step

1. Regardless of the sequence length, the learned model always has the same size, rather than specified in terms of a variable-length history of states.
2. It is possible to use same transition function f with the same parameters at every time step.

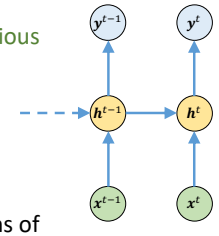


RNN with hidden recurrence

Not an efficient way! The function f^t is dependent to the sequence length.

Let's represent the past inputs by the hidden-layer activations in the previous instant.

$$\boldsymbol{h}(t) = f^t(\boldsymbol{x}(t), \underbrace{\boldsymbol{x}(t-1), \dots, \boldsymbol{x}(2), \boldsymbol{x}(1)}_{\boldsymbol{h}(t-1)})$$



Initial hidden state
Either set it to all zero or learn it

The recurrent structure allows us to factorize f^t into repeated applications of a function f. We can write:

new state $\boldsymbol{h}(t) = f(\boldsymbol{h}(t-1), \boldsymbol{x}(t))$

some function — old state — input vector at some time step

RNN with hidden recurrence

$\boldsymbol{U}$: weight vector that transforms raw inputs to the hidden-layer

$\boldsymbol{W}$: recurrent weight vector connecting previous hidden-layer output to hidden input

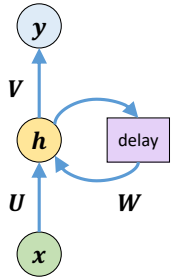
$\boldsymbol{V}$: weight vector of the output layer

$\boldsymbol{b}$: bias connected to hidden layer

$\boldsymbol{c}$: bias connected to the output layer

ϕ: the tanh hidden-layer activation function

σ: the linear/softmax output-layer activation function



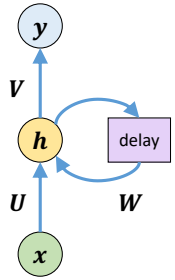
RNN with hidden recurrence

Let $\boldsymbol{x}(t)$, $\boldsymbol{y}(t)$, and $\boldsymbol{h}(t)$ be the input, output, and hidden output of the network at time t.

Activation of the Elman-type RNN with one hidden-layer is given by:

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{h}(t-1) + \boldsymbol{b}\right)$$
$$\boldsymbol{y}(t) = \sigma\left(\boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c}\right)$$

σ is a *softmax* function for classification and a *linear* function for regression.



Example 1

A recurrent neural network with hidden recurrence has two input neurons, three hidden neurons, and two output neurons. The parameters of the network are initialized as $\boldsymbol{U} = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix}$, $\boldsymbol{W} = \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix}$ and $\boldsymbol{V} = \begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}$.

Bias to the hidden layer $\boldsymbol{b} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$ and to the output layer $\boldsymbol{c} = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$.

For a sequence of inputs $\left(\boldsymbol{x}(1), \boldsymbol{x}(2), \boldsymbol{x}(3), \boldsymbol{x}(4)\right)$ where

$$\boldsymbol{x}(1) = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \boldsymbol{x}(2) = \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \boldsymbol{x}(3) = \begin{pmatrix} 0 \\ 3 \end{pmatrix}, \text{ and } \boldsymbol{x}(4) = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

find the output of RNN.

Assume that hidden layer activations are initialized to zero and *tanh* and sigmoid functions for the hidden and output layer activation functions, respectively.

eg8.1.ipynb

RNN with hidden recurrence: batch processing

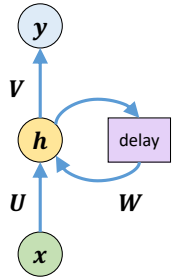
Given P patterns $\{\boldsymbol{x}_p\}_{p=1}^{P}$ where $\boldsymbol{x}_p = \left(\boldsymbol{x}_p(t)\right)_{t=1}^{T}$,

$$\boldsymbol{X}(t) = \begin{pmatrix} \boldsymbol{x}_1(t)^{\mathsf{T}} \\ \boldsymbol{x}_2(t)^{\mathsf{T}} \\ \vdots \\ \boldsymbol{x}_P(t)^{\mathsf{T}} \end{pmatrix}$$

Let $\boldsymbol{X}(t)$, $\boldsymbol{Y}(t)$, and $\boldsymbol{H}(t)$ be batch input, output, and hidden output of the network at time t

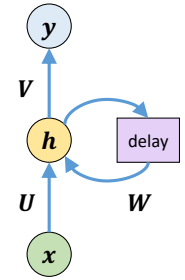
Activation of the three-layer Elman-type RNN is given by:

$$\boldsymbol{H}(t) = \phi(\boldsymbol{X}(t)\boldsymbol{U} + \boldsymbol{H}(t-1)\boldsymbol{W} + \boldsymbol{B})$$
$$\boldsymbol{Y}(t) = \sigma(\boldsymbol{H}(t)\boldsymbol{V} + \boldsymbol{C})$$



Example 1 (con't)

$\boldsymbol{U} = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix}$, $\boldsymbol{W} = \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix}$ and $\boldsymbol{V} = \begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}$



$\boldsymbol{b} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$ and $\boldsymbol{c} = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$

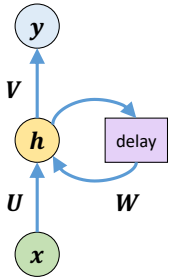
Three hidden neurons and two output neurons

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{U}^{\boldsymbol{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{h}(t-1) + \boldsymbol{b}\right)$$
$$\boldsymbol{y}(t) = \sigma\left(\boldsymbol{V}^{\boldsymbol{T}}\boldsymbol{h}(t) + \boldsymbol{c}\right)$$

$$\phi(u) = \tanh(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}}$$
$$\sigma(u) = \text{sigmoid}(u) = \frac{1}{1 + e^{-u}}.$$

Assume $\boldsymbol{h}(0) = (0 \quad 0 \quad 0)^{T}$.

Example 1 (con't)

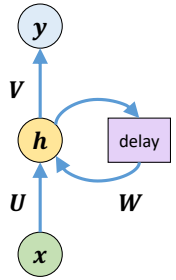


At t=1, $\boldsymbol{x}(1) = \begin{pmatrix}1\\2\end{pmatrix}$:

$$\boldsymbol{h}(1) = \phi\left(\boldsymbol{U}^T\boldsymbol{x}(1) + \boldsymbol{W}^T\boldsymbol{h}(0) + \boldsymbol{b}\right)$$
$$= \tanh\left(\begin{pmatrix}-1.0 & 0.5\\ 0.5 & 0.1\\ 0.2 & -2.0\end{pmatrix}\begin{pmatrix}1\\2\end{pmatrix} + \begin{pmatrix}2.0 & 1.5 & -0.2\\ 1.3 & 0.0 & 1.5\\ -1.0 & -0.5 & 0.4\end{pmatrix}\begin{pmatrix}0\\0\\0\end{pmatrix} + \begin{pmatrix}0.2\\0.2\\0.2\end{pmatrix}\right)$$
$$= \begin{pmatrix}0.2\\0.72\\-1.0\end{pmatrix}$$

$$\boldsymbol{y}(1) = \sigma\left(\boldsymbol{V}^T\boldsymbol{h}(1) + \boldsymbol{c}\right)$$
$$= \text{sigmoid}\left(\begin{pmatrix}2.0 & -1.5 & 0.2\\ -1.0 & 0.5 & 0.8\end{pmatrix}\begin{pmatrix}0.2\\0.72\\-1.0\end{pmatrix} + \begin{pmatrix}0.5\\0.1\end{pmatrix}\right)$$
$$= \begin{pmatrix}0.41\\0.37\end{pmatrix}$$

Example 1 (con't)



Similarly,

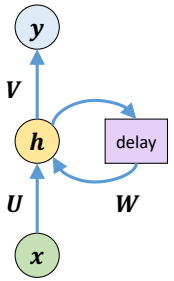
at t=3, $\boldsymbol{x}(3) = \begin{pmatrix}0\\3\end{pmatrix}$; $\boldsymbol{h}(3)\begin{pmatrix}0.99\\0.3\\-1.0\end{pmatrix}$ and $\boldsymbol{y}(3) = \begin{pmatrix}0.86\\0.18\end{pmatrix}$

at t=4, $\boldsymbol{x}(4) = \begin{pmatrix}2\\-1\end{pmatrix}$; $\boldsymbol{h}(4)\begin{pmatrix}0.31\\0.71\\0.79\end{pmatrix}$ and $\boldsymbol{y}(4) = \begin{pmatrix}0.55\\0.68\end{pmatrix}$

The output is $\left(\boldsymbol{y}(1), \boldsymbol{y}(2), \boldsymbol{y}(3), \boldsymbol{y}(4)\right)$ where

$$\boldsymbol{y}(1) = \begin{pmatrix}0.41\\0.37\end{pmatrix}, \boldsymbol{y}(2) = \begin{pmatrix}0.97\\0.11\end{pmatrix}, \boldsymbol{y}(3) = \begin{pmatrix}0.86\\0.18\end{pmatrix} \boldsymbol{y}(4) = \begin{pmatrix}0.55\\0.68\end{pmatrix}$$

Example 1 (con't)



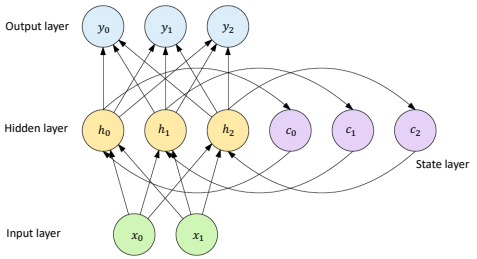
At t=2, $\boldsymbol{x}(2) = \begin{pmatrix}-1\\1\end{pmatrix}$:

$$\boldsymbol{h}(2) = \phi\left(\boldsymbol{U}^T\boldsymbol{x}(2) + \boldsymbol{W}^T\boldsymbol{h}(1) + \boldsymbol{b}\right)$$
$$= \tanh\left(\begin{pmatrix}-1.0 & 0.5\\ 0.5 & 0.1\\ 0.2 & -2.0\end{pmatrix}\begin{pmatrix}-1\\1\end{pmatrix} + \begin{pmatrix}2.0 & 1.5 & -0.2\\ 1.3 & 0.0 & 1.5\\ -1.0 & -0.5 & 0.4\end{pmatrix}\begin{pmatrix}0.2\\0.72\\-1.0\end{pmatrix} + \begin{pmatrix}0.2\\0.2\\0.2\end{pmatrix}\right)$$
$$= \begin{pmatrix}1.0\\-0.89\\-0.99\end{pmatrix}$$

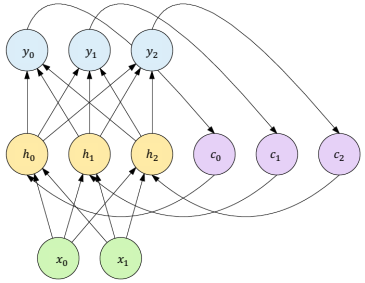
$$\boldsymbol{y}(2) = \sigma\left(\boldsymbol{V}^T\boldsymbol{h}(2) + \boldsymbol{c}\right)$$
$$= \text{sigmoid}\left(\begin{pmatrix}2.0 & -1.5 & 0.2\\ -1.0 & 0.5 & 0.8\end{pmatrix}\begin{pmatrix}1.0\\-0.89\\-0.99\end{pmatrix} + \begin{pmatrix}0.5\\0.1\end{pmatrix}\right) = \begin{pmatrix}0.97\\0.11\end{pmatrix}$$

Types of RNN

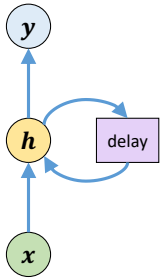
RNN with hidden recurrence (Elman-type)



RNN with top-down recurrence (Jordan-type)

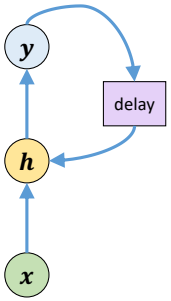


RNN with top-down recurrence (Jordan type)



The **hidden-layer activation** at time $t-1$ is kept by the **delay unit** and fed to the hidden layer at time t together with the raw input $\boldsymbol{x}(t)$.

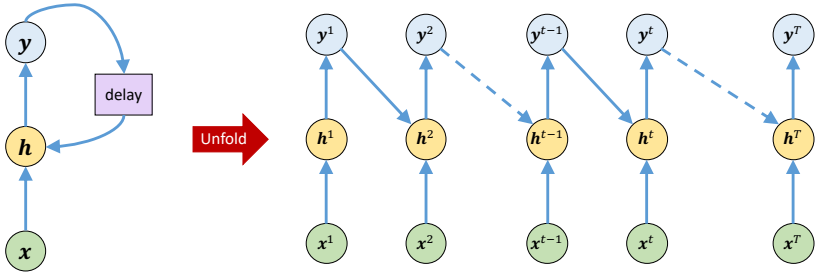
- $\boldsymbol{h}(t) = \phi(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{h}(t-1) + \boldsymbol{b})$
- $\boldsymbol{y}(t) = \sigma(\boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c})$



The **output of the output-layer** at time $t-1$ is kept by the **delay unit** and fed to the hidden layer at time t together with the raw input $x(t)$ for time t

- $\boldsymbol{h}(t) = \phi(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{y}(t-1) + \boldsymbol{b})$
- $\boldsymbol{y}(t) = \sigma(\boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c})$

RNN with top-down recurrence



The recurrent connections in the hidden-layer is unfolded and represented in time for processing a time series $\left(x(t)\right)_{t=1}^{T}$

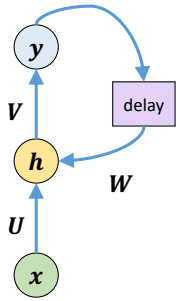
RNN with top-down recurrence

Let $\boldsymbol{x}(t)$, $\boldsymbol{y}(t)$, and $\boldsymbol{h}(t)$ be the input, output, and hidden output of the network at time t.

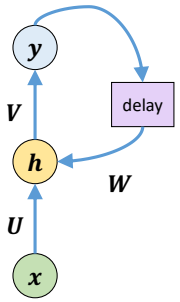
Activation of the Jordan-type RNN with one hidden-layer is given by:

$$\boldsymbol{h}(t) = \phi(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{y}(t-1) + \boldsymbol{b})$$
$$\boldsymbol{y}(t) = \sigma(\boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c})$$

Note that output of the previous time instant is fed back to the hidden layer and $\boldsymbol{W}$ represents the recurrent weight matrix connecting previous output to the current hidden input



RNN with top-down recurrence: batch processing



Given P patterns $\{\boldsymbol{x}_p\}_{p=1}^{P}$ where $\boldsymbol{x}_p = \left(\boldsymbol{x}_p(t)\right)_{t=1}^{T}$,

$$\boldsymbol{X}(t) = \begin{pmatrix} \boldsymbol{x}_1(t)^{\mathsf{T}} \\ \boldsymbol{x}_2(t)^{\mathsf{T}} \\ \vdots \\ \boldsymbol{x}_P(t)^{\mathsf{T}} \end{pmatrix}$$

Let $\boldsymbol{X}(t)$, $\boldsymbol{Y}(t)$, and $\boldsymbol{H}(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Jordan-type RNN is given by:

$$\boldsymbol{H}(t) = \phi(\boldsymbol{X}(t)\boldsymbol{U} + \boldsymbol{Y}(t-1)\boldsymbol{W} + \boldsymbol{B})$$
$$\boldsymbol{Y}(t) = \sigma(\boldsymbol{H}(t)\boldsymbol{V} + \boldsymbol{C})$$

Example 2

A recurrent neural network with top-down recurrence receives 2-dimensional input sequences and produce 1-dimensional output sequences. It has three hidden neurons and following weight matrices and biases:

Weight matrix connecting input the hidden layer $\boldsymbol{U} = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix}$

Recurrence weight matrix connecting previous output to the hidden layer $\boldsymbol{W} = \begin{pmatrix} 2.0 & 1.3 & -1.0 \end{pmatrix}$

Weight matrix connecting hidden layer to the output $\boldsymbol{V} = \begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix}$

Bias to the hidden layer $\boldsymbol{b} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$ and bias to the output layer $\boldsymbol{c} = 0.1$.

Given the following two input sequences:

$$\boldsymbol{x}_1 = \left(\begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 3 \end{pmatrix}\right)$$
$$\boldsymbol{x}_2 = \left(\begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 2 \\ -1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ -1 \end{pmatrix}\right)$$

Using batch processing, find output sequences. Assume outputs are initialized to zero at the beginning. Assume tanh and sigmoid activations for hidden and output layer, respectlvely.

eg8.2.ipynb

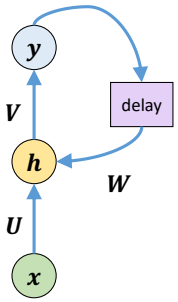
Example 2 (con't)

At $t = 1 : \boldsymbol{X}(1) = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}$

$$\begin{aligned} \boldsymbol{H}(1) &= tanh(\boldsymbol{X}(1)\boldsymbol{U} + \boldsymbol{Y}(0)\boldsymbol{W} + \boldsymbol{B}) \\ &= tanh\left(\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}\begin{pmatrix} 2.0 & 1.3 & -1.0 \end{pmatrix} + \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.2 & 0.72 & -1.0 \\ 0.83 & -0.29 & 0.0 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \boldsymbol{Y}(1) &= sigmoid(\boldsymbol{H}(1)\boldsymbol{V} + \boldsymbol{C}) \\ &= sigmoid\left(\begin{pmatrix} 0.2 & 0.72 & -1.0 \\ 0.83 & -0.29 & 0.0 \end{pmatrix}\begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.31 \\ 0.90 \end{pmatrix} \end{aligned}$$

Example 2 (con't)



$$\boldsymbol{x}_1 = \left(\begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 3 \end{pmatrix}\right)$$
$$\boldsymbol{x}_2 = \left(\begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 2 \\ -1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ -1 \end{pmatrix}\right)$$

Two sequences as batch of sequences:

$$\boldsymbol{X} = \left(\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} \quad \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix} \quad \begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}\right)$$

Three hidden neurons and one output neuron.

$$\boldsymbol{H}(t) = \phi(\boldsymbol{X}(t)\boldsymbol{U} + \boldsymbol{Y}(t-1)\boldsymbol{W} + \boldsymbol{B})$$
$$\boldsymbol{Y}(t) = \sigma(\boldsymbol{H}(t)\boldsymbol{V} + \boldsymbol{C})$$

Initially,

$$\boldsymbol{Y}(0) = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

Example 2 (con't)

At $t = 2 : \boldsymbol{X}(2) = \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}$

$$\begin{aligned} \boldsymbol{H}(2) &= tanh(\boldsymbol{X}(2)\boldsymbol{U} + \boldsymbol{Y}(1)\boldsymbol{W} + \boldsymbol{B}) \\ &= tanh\left(\begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.31 \\ 0.9 \end{pmatrix}\begin{pmatrix} 2.0 & 1.3 & -1.0 \end{pmatrix} + \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \boldsymbol{Y}(2) &= sigmoid(\boldsymbol{H}(2)\boldsymbol{V} + \boldsymbol{C}) \\ &= sigmoid\left(\begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix}\begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix} \end{aligned}$$

Example 2 (con't)

At $t = 3 : \boldsymbol{X}(3) = \begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}$

$$\begin{aligned} \boldsymbol{H}(3) &= tanh(\boldsymbol{X}(3)\boldsymbol{U} + \boldsymbol{Y}(1)\boldsymbol{W} + \boldsymbol{B}) \\ &= tanh\left(\begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix}(2.0 \quad 1.3 \quad -1.0) + \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.00 & 0.94 & 0.99 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \boldsymbol{Y}(3) &= sigmoid(\boldsymbol{H}(3)\boldsymbol{V} + \boldsymbol{C}) \\ &= sigmoid\left(\begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.0 & 0.94 & 0.99 \end{pmatrix}\begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.63 \\ 0.04 \end{pmatrix} \end{aligned}$$

Backpropagation through time (BPTT)

Let's consider vanilla-RNN with hidden recurrence.
Forward propagation equations:

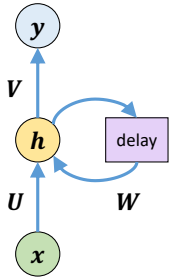
$$\begin{aligned} \boldsymbol{h}(t) &= tanh\big(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{h}(t-1) + \boldsymbol{b}\big) \\ \boldsymbol{u}(t) &= \boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c} \end{aligned}$$

For classification,

$$\boldsymbol{y}(t) = softmax\big(\boldsymbol{u}(t)\big)$$

For regression,

$$\boldsymbol{y}(t) = \boldsymbol{u}(t)$$



Learnable parameters

- $\boldsymbol{U}$: weight vector that transforms raw inputs to the hidden-layer
- $\boldsymbol{W}$: recurrent weight vector connecting previous hidden-layer output to hidden input
- $\boldsymbol{V}$: weight vector of the output layer
- $\boldsymbol{b}$: bias connected to hidden layer
- $\boldsymbol{c}$: bias connected to the output layer

Example 2 (con't)

Output (batch):

$$\begin{aligned} \boldsymbol{Y} &= \big(\boldsymbol{Y}(1) \quad \boldsymbol{Y}(2) \quad \boldsymbol{Y}(3)\big) \\ &= \left(\begin{pmatrix} 0.31 \\ 0.9 \end{pmatrix} \quad \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix} \quad \begin{pmatrix} 0.63 \\ 0.04 \end{pmatrix}\right) \end{aligned}$$

Outputs for inputs

$$x_1 = \left(\begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 3 \end{pmatrix}\right)$$

$$x_2 = \left(\begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 2 \\ -1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ -1 \end{pmatrix}\right)$$

are

$$y_1 = (0.31, 0.83, 0.63)$$

$$y_2 = (0.9, 0.11, 0.04)$$

Backpropagation through time (BPTT)

Let's consider vanilla-RNN with hidden recurrence.
Forward propagation equations:

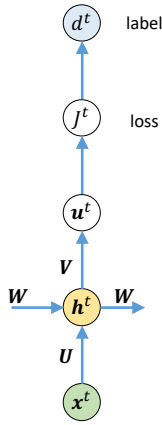
$$\begin{aligned} \boldsymbol{h}(t) &= tanh\big(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{h}(t-1) + \boldsymbol{b}\big) \\ \boldsymbol{u}(t) &= \boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c} \end{aligned}$$

For classification,

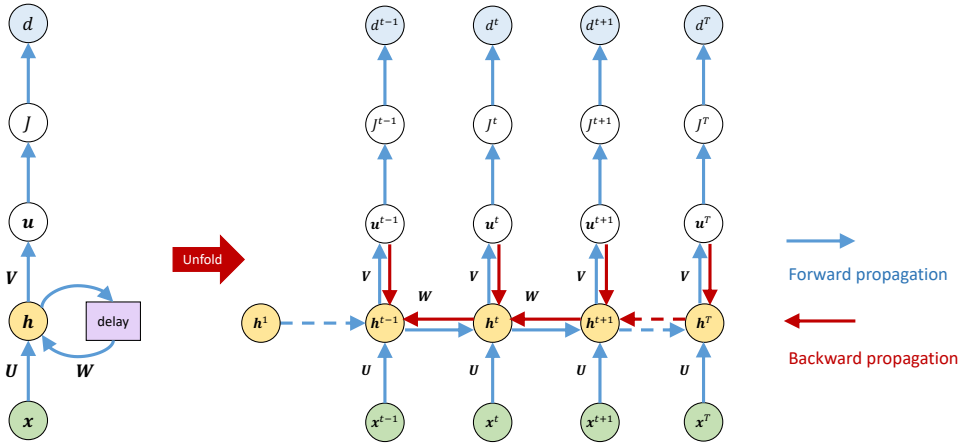
$$\boldsymbol{y}(t) = softmax\big(\boldsymbol{u}(t)\big)$$

For regression,

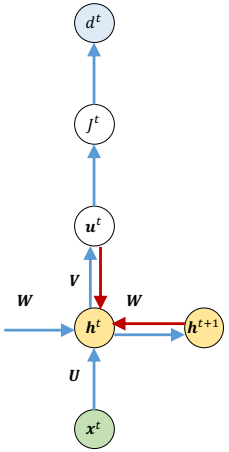
$$\boldsymbol{y}(t) = \boldsymbol{u}(t)$$



Backpropagation through time (BPTT)



Backpropagation through time (BPTT)

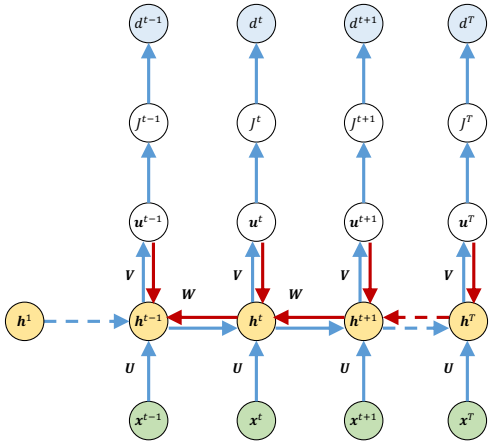


The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph, followed by a backpropagation pass moving right to left through the graph. The gradients are propagated from the final time point to the initial time points .

The runtime is $O(T)$ where T is the length of input sequence and cannot be reduced by parallelization because the forward propagation is inherently sequential. The back-propagation algorithm applied to the unrolled graph with $O(T)$ cost is called *back-propagation through time (BPTT).*

The network with recurrence between hidden units is thus very powerful but also expensive to train.

Backpropagation through time (BPTT)



The gradients propagate backward, starting from the end of the sequence from two directions:

- Top-down direction
- Reverse sequence direction

Example 3

Generate 8-dimensional 16 input sequences of 64 time-steps with each input is a random number between [0.0, 1.0].

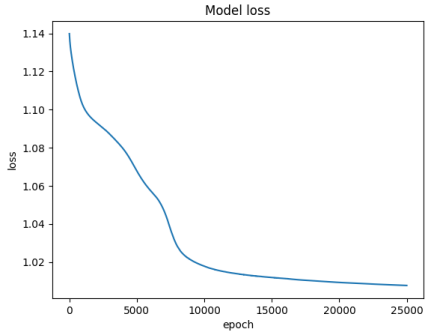
Generate the corresponding 1-dimensional labels by randomly generating a number from [0, 1, 2].

Create an RNN with one hidden layer of 5 neurons.

Plot the learning curves and predicted labels.

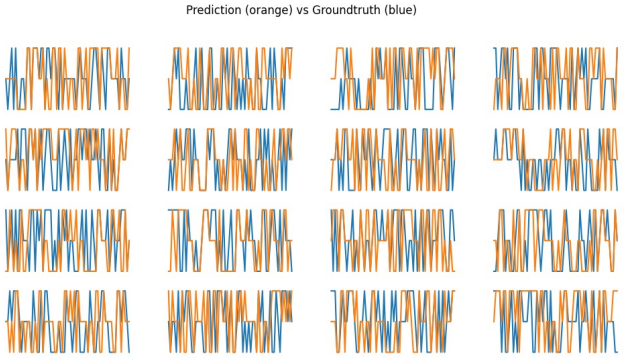
eg8.3.ipynb

Example 3 (con't)



Long Short-Term Memory (LSTM)

Example 3 (con't)



Long-term dependency

"The man who ate my pizza has purple hair"

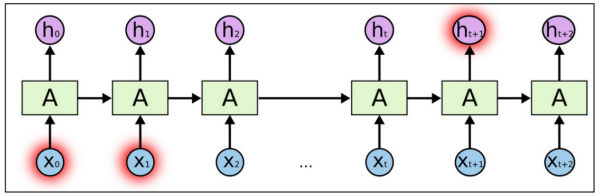
Purple hair is for the man and not the pizza! This is a long term dependency.

There are cases where we need even more context

- To predict last word in "I grew up in France…<long paragraph>…I speak *French*"
- Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is *French*

Long-term dependency

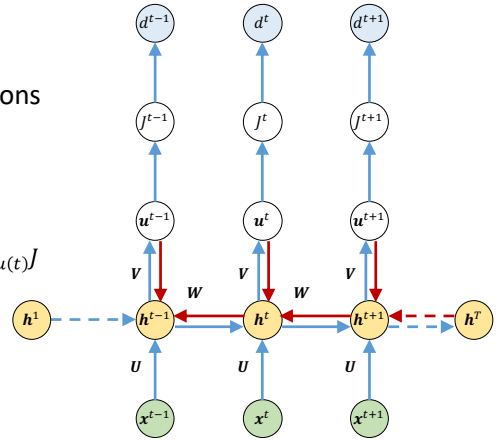
- RNNs work upon the fact that the result of an information is dependent on its previous state or previous n time steps
- RNNs have difficulty in learning long range dependencies
- Gap between relevant information and where it is needed is large



Exploding and vanishing gradients in RNN

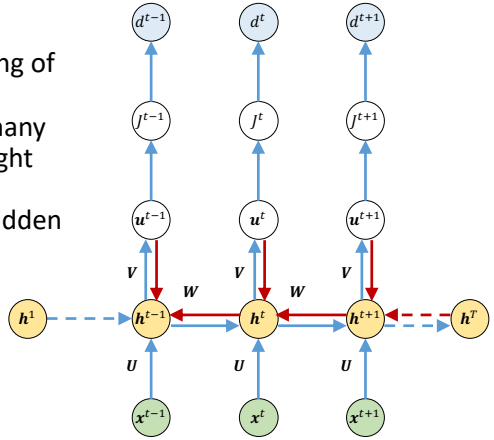
Note that each time the activations are forward propagated in time, the activations are multiplied by $\boldsymbol{W}$ and each time the gradients are back propagated, the gradients are multiplied by $\boldsymbol{W}^{\top}$.

$$\nabla_{\boldsymbol{h}(t)} J = \underbrace{\boldsymbol{W}\,\mathrm{diag}\left(1-\boldsymbol{h}^2(t+1)\right)\nabla_{\boldsymbol{h}(t+1)}J}_{\text{Gradient from reverse direction}} + \boldsymbol{V}\nabla_{u(t)}J$$



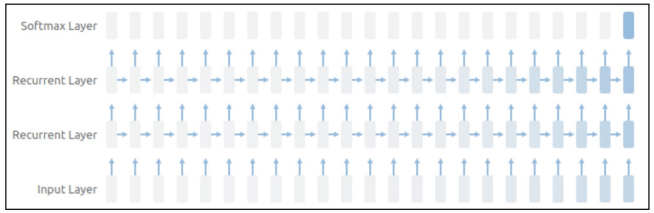
Exploding and vanishing gradients in RNN

During gradient back-propagation learning of RNN, the gradient can end up being multiplied a large number of times (as many as the number of time steps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer.



Exploding and vanishing gradients in RNN

If the weights in this matrix are small, the recursive derivative can lead to a situation called **vanishing gradients** where the gradient signal gets so small that learning either becomes very slow or stops working altogether



Contribution from the earlier steps becomes insignificant in the gradient

Exploding and vanishing gradients in RNN

Conversely, if the weights in this matrix are large, it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as **exploding gradients**.

Exploding gradient easily solved by clipping the gradients at a predefined threshold value.

Vanishing gradient is more concerning

Gradient Clipping

Commonly, two methods are used:

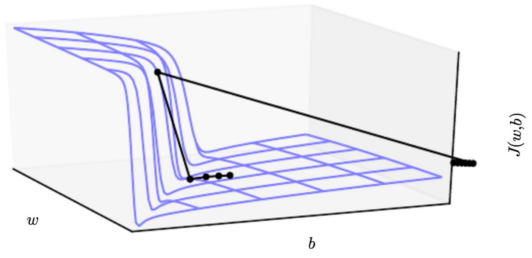
1. Clip the gradient g when it exceeds a threshold:

$$\text{if} \|g\| > v:$$
$$\|g\| \leftarrow v$$

2. Normalize the gradient when it exceeds a threshold:

$$\text{if} \|g\| > v:$$
$$\|g\| \leftarrow \frac{g}{\|g\|} v$$

Gradient Clipping



Due to long term dependencies, RNN tend to have gradients having very large or very small magnitudes. The large gradients resemble cliffs in the error landscape and when the gradient descent encounters the gradient updates can move the parameters away from true minimum.

A gradient clipping is employed to avoid the gradients to become too large.

Exploding and vanishing gradients in RNN

Vanishing and exploding gradients in gradient backpropagation learning makes it difficulty train RNN to learn long-term dependencies.

Solution: Gated RNNs

- Long short-term memory (LSTM), Gated Recurrent Units (GRU)

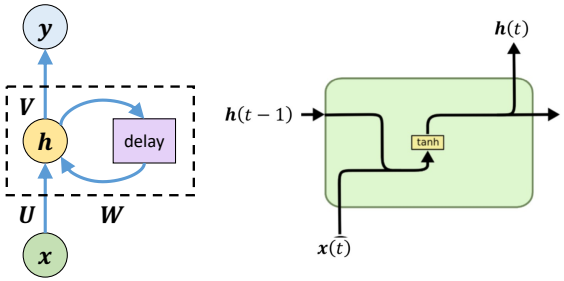
Basic RNN unit

The RNN cell (memory unit) is characterized by

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{h}(t-1) + \boldsymbol{b}\right)$$

where ϕ is the *tanh* activation function and RNN cell is referred to as *tanh* units.

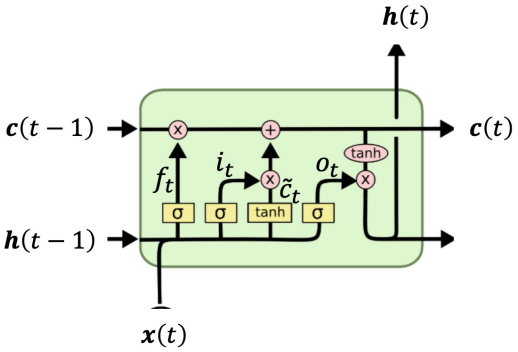
RNN build with simple *tanh* units are also referred to as vanilla RNN.



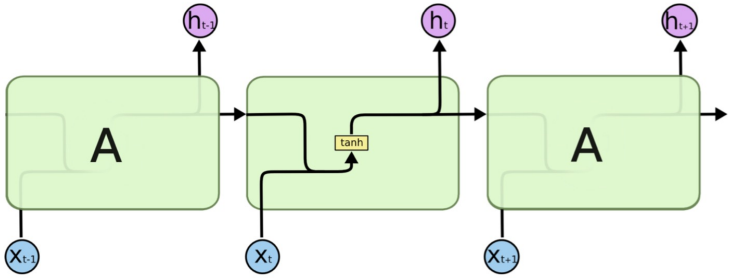
Long short-term memory (LSTM) unit

Key of LSTM is ”state”

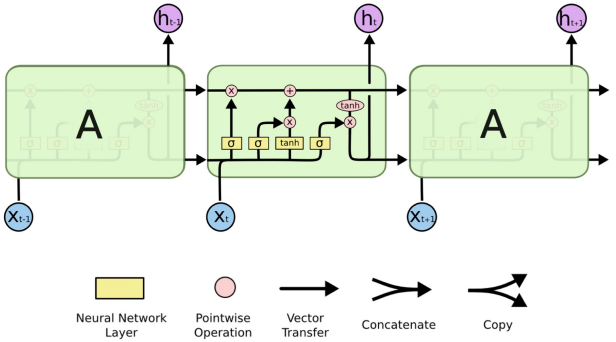
- A persistent module called the cell-state
- “State” is a representation of past history
- It comprises a common thread through time



Basic RNN layer



Long short-term memory (LSTM) unit

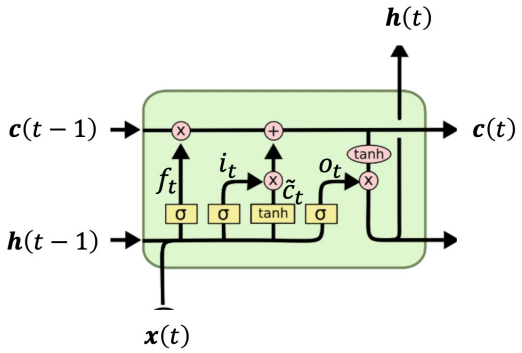


Cells are connected recurrently to each other - Replacing hidden units of ordinary recurrent networks

Long short-term memory (LSTM) unit

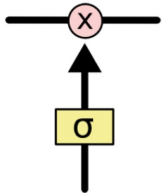
LSTMs provide a solution by incorporating memory units that allow the network to learn when to forget previous hidden states and when to update hidden states given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.



Hochreiter & Schmidhuber (1997)

Gates



The LSTM has the ability to add and remove information to the cell states through gates.

Gates are a way to optionally let information through.

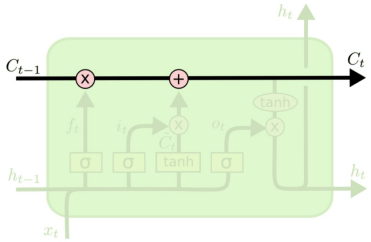
They are composed of sigmoid neural net layer and pointwise multiplication operations.

Long short-term memory (LSTM) unit

The key to LSTM is the cell state $c(t)$ - the horizontal line through the top of the diagram.

The long-term memory.

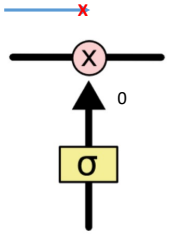
Like a conveyor belt that runs through entire chain with minor interactions

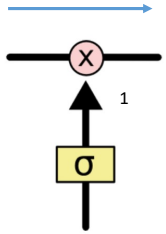


Gates

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.

A value of zero means “let nothing through,” while a value of one means “let everything through”





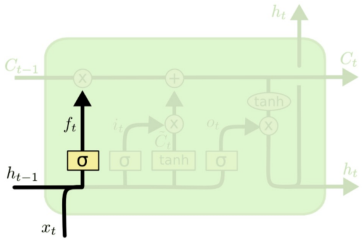
An LSTM has three of these gates to control cell state.

Forget gate

The forget gate can modulate the memory cell's self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.

$$\boldsymbol{f}(t) = \sigma\left(\boldsymbol{U}_f^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_f^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_f\right)$$

Value of $\boldsymbol{f}(t)$ determines if $c(t-1)$ is to be remembered or not.



Input gate

The input gate can allow incoming signal to alter the state of the memory cell or block it. It decides what new information to store in the cell stage.

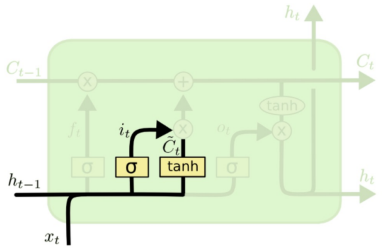
This has two parts:

A sigmoid input gate layer decides which values to update;

A tanh layer creates a vector of new candidate values $\tilde{\boldsymbol{c}}(t)$ that could be added to the state.

$$\boldsymbol{i}(t) = \sigma\left(\boldsymbol{U}_i^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_i^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_i\right)$$

$$\tilde{\boldsymbol{c}}(t) = \phi\left(\boldsymbol{U}_c^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_c^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_c\right)$$



Example: Language modeling
We'd want to add the gender of the new subject to the cell state, to replace the old one we are forgetting

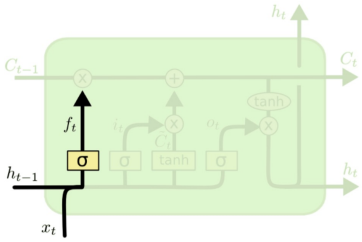
Forget gate

Example: Language modelling

Consider trying to predict the next word based on all previous ones

The cell state may include the gender of the present subject so that the proper pronouns can be used

When we see a new subject we want to forget old subject



Cell state

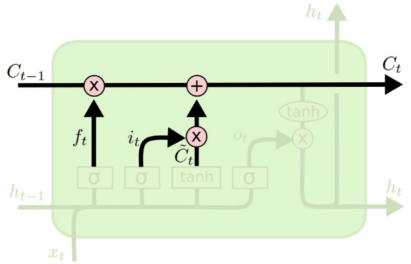
a vector of new candidate values

which values to update

$$\boldsymbol{c}(t) = \tilde{\boldsymbol{c}}(t) \odot \boldsymbol{i}(t) + \boldsymbol{c}(t-1) \odot \boldsymbol{f}(t)$$

input gate

forget gate



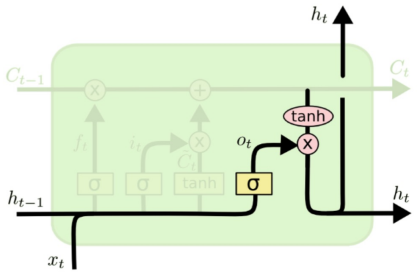
Example: Language modeling
In the Language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in previous steps

Output gate

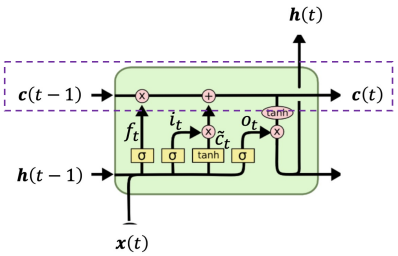
The output gate can allow the state of the memory cell to have an effect on other neurons or prevent it.

$$\boldsymbol{o}(t) = \sigma\left(\boldsymbol{U}_o^\top \boldsymbol{x}(t) + \boldsymbol{W}_o^\top \boldsymbol{h}(t-1) + \boldsymbol{b}_o\right)$$

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{c}(t)\right) \odot \boldsymbol{o}(t)$$



LSTM unit



LSTM introduces a gated cell where the information flow can be controlled.

The most important component is the state unit $c_i(t)$ (for time step t and cell i) which has a **linear self- loop**.

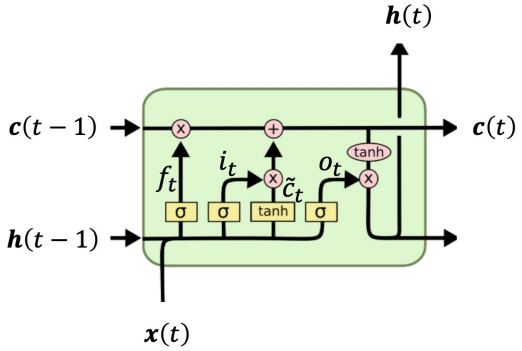
The self-loop weight is controlled by a forget gate unit, which sets this weight to a value between 0 and 1 via a sigmoid unit.

LSTM unit

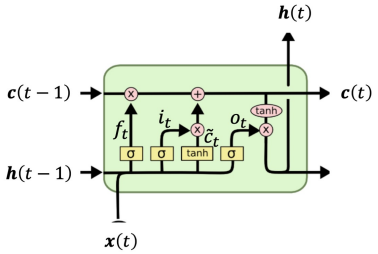
$$\boldsymbol{i}(t) = \sigma\left(\boldsymbol{U}_i^\top \boldsymbol{x}(t) + \boldsymbol{W}_i^\top \boldsymbol{h}(t-1) + \boldsymbol{b}_i\right)$$
$$\boldsymbol{f}(t) = \sigma\left(\boldsymbol{U}_f^\top \boldsymbol{x}(t) + \boldsymbol{W}_f^\top \boldsymbol{h}(t-1) + \boldsymbol{b}_f\right)$$
$$\boldsymbol{o}(t) = \sigma\left(\boldsymbol{U}_o^\top \boldsymbol{x}(t) + \boldsymbol{W}_o^\top \boldsymbol{h}(t-1) + \boldsymbol{b}_o\right)$$

$$\tilde{\boldsymbol{c}}(t) = \phi\left(\boldsymbol{U}_c^\top \boldsymbol{x}(t) + \boldsymbol{W}_c^\top \boldsymbol{h}(t-1) + \boldsymbol{b}_c\right)$$

$$\boldsymbol{c}(t) = \tilde{\boldsymbol{c}}(t) \odot \boldsymbol{i}(t) + \boldsymbol{c}(t-1) \odot \boldsymbol{f}(t)$$
$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{c}(t)\right) \odot \boldsymbol{o}(t)$$



LSTM unit



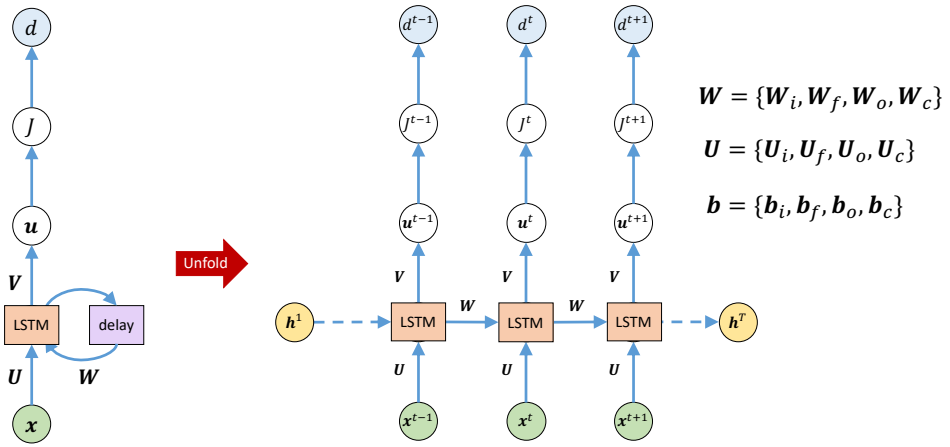
Clever idea!

The self-loop inside the cell is able to produces paths where the gradient can flow for long duration and to make the weight on this self-loop conditioned on the context rather than fixed.

By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically based on the input sequence.

In this way, LSTM help preserve error terms that can be propagated through many layers and time steps.

Training LSTM networks

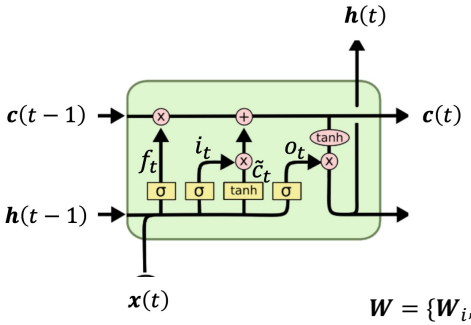


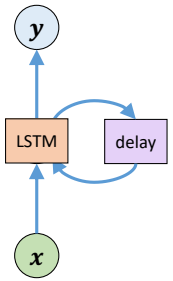
$$\boldsymbol{W} = \{\boldsymbol{W}_i, \boldsymbol{W}_f, \boldsymbol{W}_o, \boldsymbol{W}_c\}$$

$$\boldsymbol{U} = \{\boldsymbol{U}_i, \boldsymbol{U}_f, \boldsymbol{U}_o, \boldsymbol{U}_c\}$$

$$\boldsymbol{b} = \{\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o, \boldsymbol{b}_c\}$$

Example 4





$$\boldsymbol{W} = \{\boldsymbol{W}_i, \boldsymbol{W}_f, \boldsymbol{W}_o, \boldsymbol{W}_c\}$$

$$\boldsymbol{U} = \{\boldsymbol{U}_i, \boldsymbol{U}_f, \boldsymbol{U}_o, \boldsymbol{U}_c\}$$

$$\boldsymbol{b} = \{\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o, \boldsymbol{b}_c\}$$

Example 4

Generate 64 2-dimensional input sequences $\left(\boldsymbol{x}(t)\right)_{t=1}^{16}$ where $\left(x_1(t), x_2(t)\right) \in [0,1]^2$ by randomly generating numbers uniformly.

Generate 1-dimensional output sequences $\left(y(t)\right)_{t=1}^{16}$ where $y(t) \in \boldsymbol{R}$ by following the following recurrent relation:

$$y(t) = 5x_1(t-1)x_2(t-2) - 2x_1(t-7) + 3.5{x_2}^2(t-5) + 0.1\epsilon$$

where $\epsilon \sim N(0,1)$.

Train a LSTM layer to learn the mapping between input and output sequences.

Use a learning factor $\alpha = 0.001$.

Repeat the above using RNN and GRU and compare the performances.

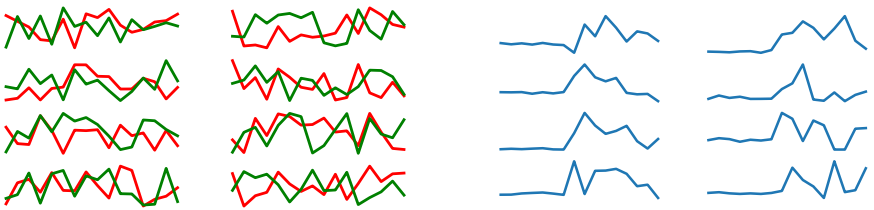
eg8.4a.ipynb

eg8.4b.ipynb

Example 4

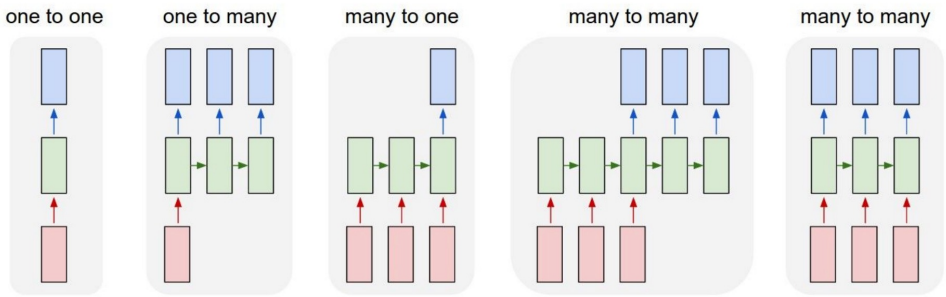
$x_1(t)$ and $x_2(t)$

$$y(t) = 5x_1(t-1)x_2(t-1) - 2x_1(t-7) + 3.5{x_2}^2(t-5)$$



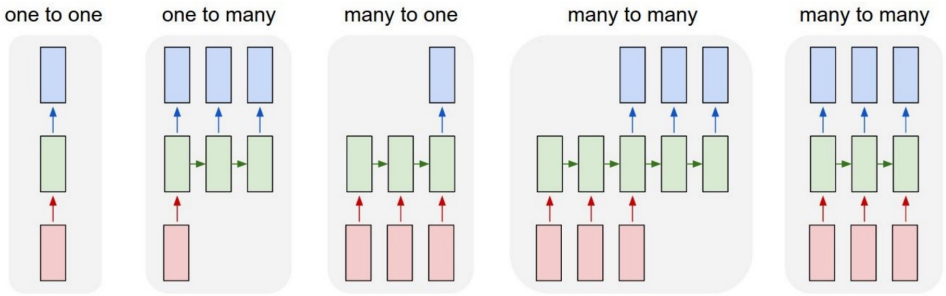
Example Applications

Input-output scenarios



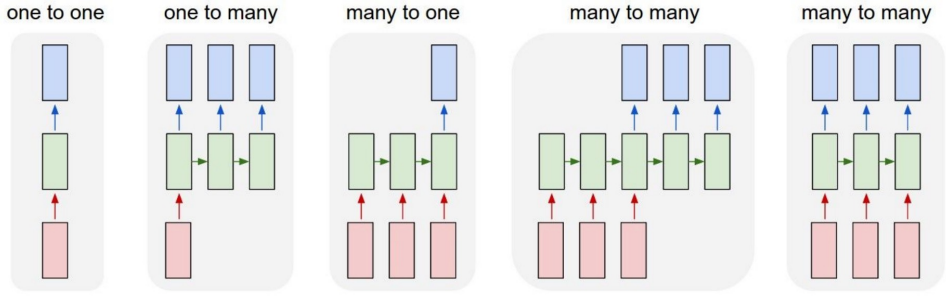
e.g. **Sentiment Classification**
sequence of words -> sentiment

Input-output scenarios



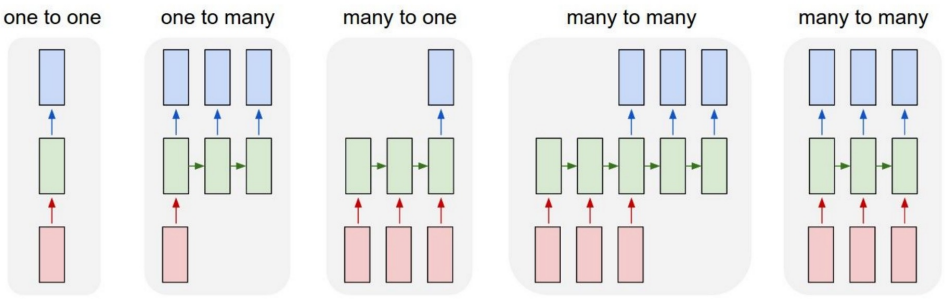
e.g. **Image Captioning**
image -> sequence of words

Input-output scenarios



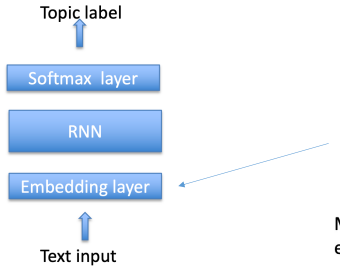
e.g. **Machine Translation**
seq of words -> seq of words

Input-output scenarios



e.g. **Video classification on frame level**

Text classification



$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{14} \\ w_{31} & w_{32} & w_{33} & w_{14} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} w_{11} \\ w_{21} \\ w_{31} \end{bmatrix}$$

Matrix multiply with a one-hot vector just extracts a column from the weight matrix.

The embedding layer converts one-hot vector of word representations (of size of the vocabulary) to a vector of fixed length (embedding_size) vectors.

Embedding layer learns a weight matrix of [embedding size, vocab size]
And then maps word indexes of the sequences into
[batch_size, sequence_length, embedding_size] input to the RNN.

Text classification: Dbpedia dataset

This dataset contains first paragraph of Wikipedia page of 56000 entities and label them as one of 15 categories like people, company, schools, etc.

1	D. C. Thomson & Co.	D. C. Thomson & Company Limited is a Scottish publishing company based in Dundee best known for
1	Pyro Spectaculars	Pyro Spectaculars is headquartered in Rialto California USA and occupies a portion of a former World V
1	Admiral Insurance	Admiral a trading name of EUI Limited is a car insurance specialist which launched in January 1993. Its
1	Pax Softnica	Pax Softnica (パックスソフトニカ) is a Japanese video game developer founded in 1983 under the nan
1	The ACME Laboratories Ltd	The ACME Laboratories Ltd is a major pharmaceutical company based in Bangladesh. It is part of the
2	Dubai Gem Private School & Nursery	Dubai Gem Private School (DGPS) is a British school located in the Oud Metha area of Dubai United Ar
2	Michael Wallace Elementary School	Michael Wallace Elementary School is a Canadian public school in Dartmouth Nova Scotia. It is operate
2	Alma Heights Christian Schools	Alma Heights Christian Schools (AHC) formerly Alma Heights Christian Academy is a private elementar

An input is a text
Requires to find all words in the text and remap them into word IDs, a number per each unit word (word-level inference)

my work is cool! → [23, 500, 5, 1402, 17]

We need to make sure that each sentence is of same length (no_words). The maximum document length is fixed and longer sentences will be truncated and shorter once are padded with zeros.

Sentiment classification

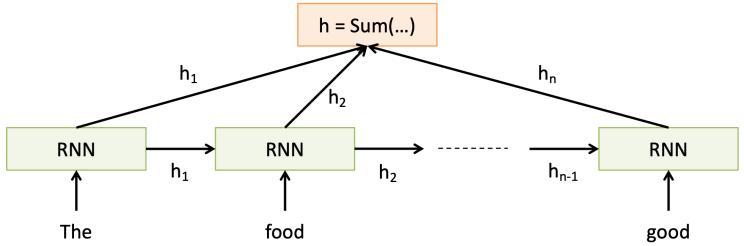
- "The food was really good"

"The chicken crossed the road because it was uncooked"

- Classify a
  restaurant review from Yelp! OR
  movie review from IMDB OR
  …
  as positive or negative

- Inputs: Multiple words, one or more sentences
- Outputs: Positive / Negative classification

Sentiment classification



http://deeplearning.net/tutorial/lstm.html

Image captioning

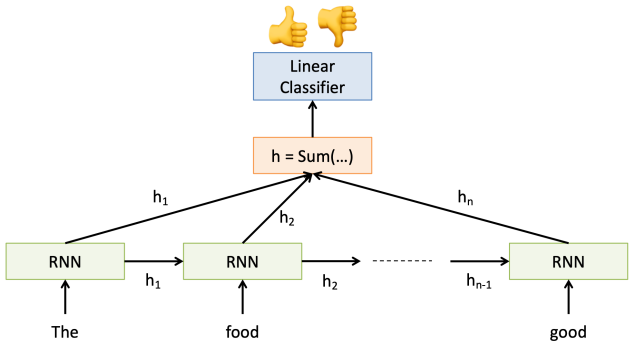
- Given an image, produce a sentence describing its contents
- Inputs: Image feature (from a CNN)
- Outputs: Multiple words (let's consider one sentence)

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

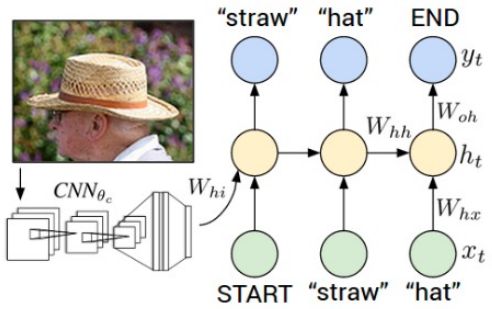
"two young girls are playing with lego toy."

Sentiment classification



http://deeplearning.net/tutorial/lstm.html

Image captioning



https://cs.stanford.edu/people/karpathy/sfmltalk.pdf

Image captioning

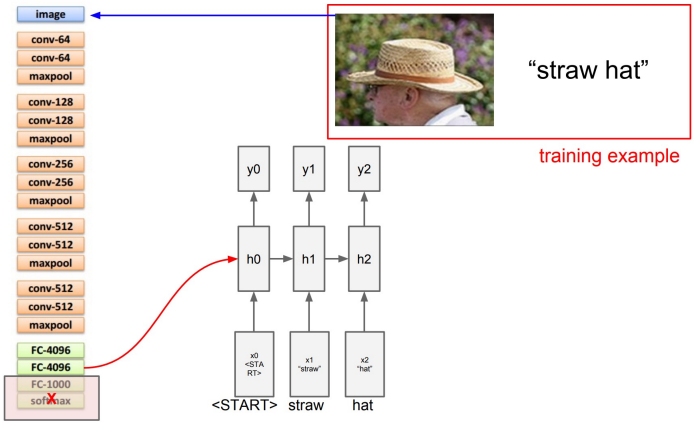


Image captioning

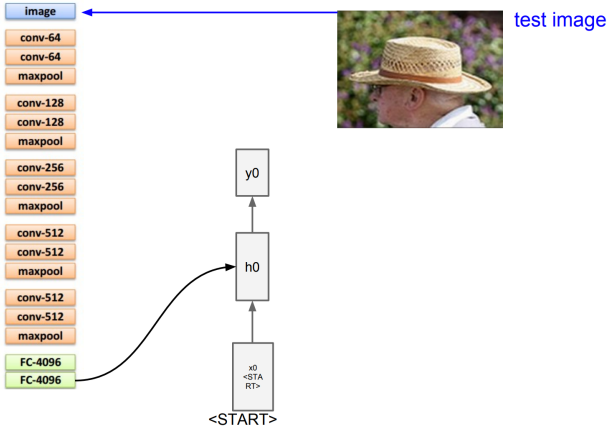


Image captioning

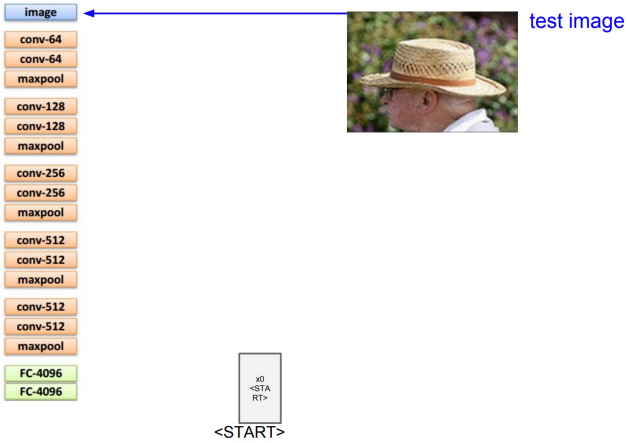


Image captioning

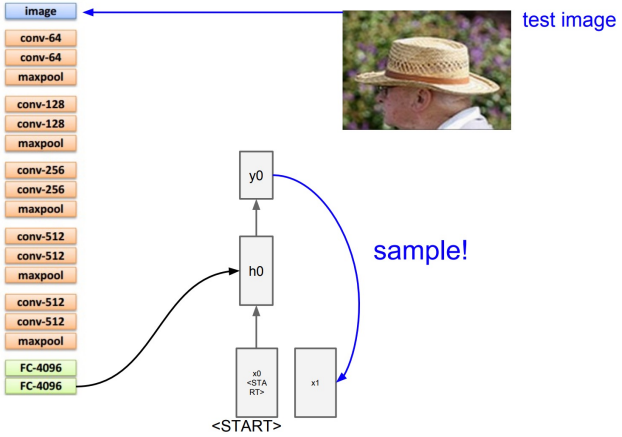


Image captioning

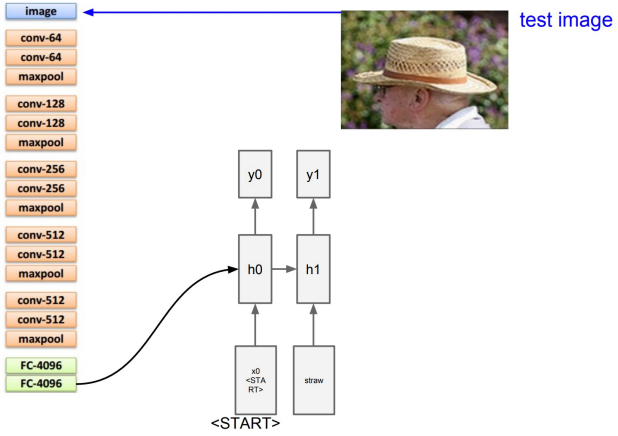


Image captioning

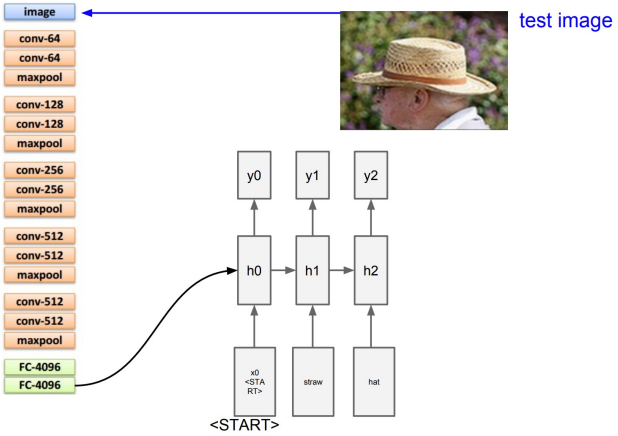


Image captioning

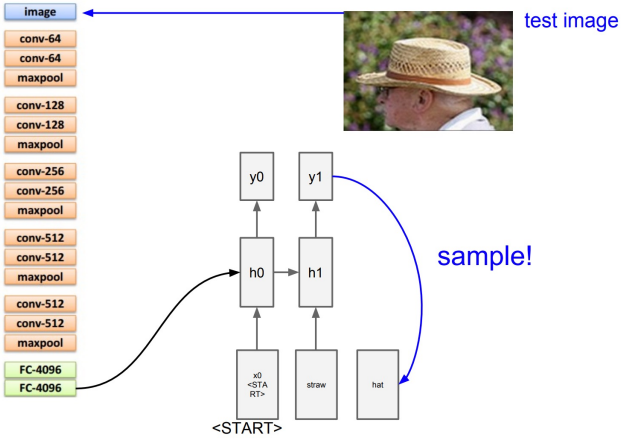


Image captioning

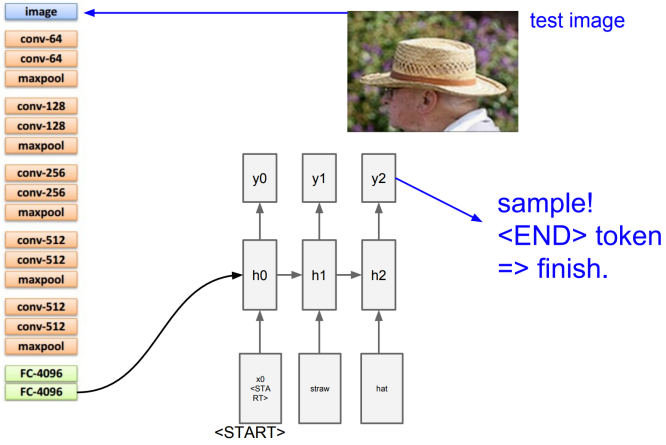
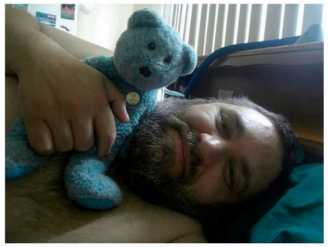


Image captioning

a young boy is holding a baseball bat
logprob: -7.65



a baby laying on a bed with a stuffed bear
logprob: -8.66

Outline

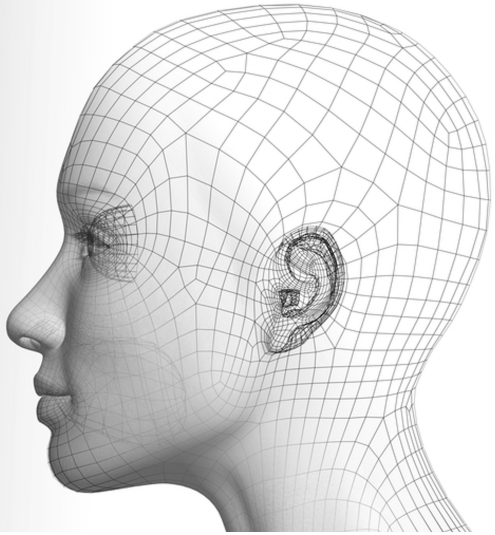
- Attention
- Transformers
- Vision Transformers

SC 4001 CE/CZ 4042 Neural Network and Deep Learning
Last update: 26 Mar 2024

Attention

Xingang Pan
潘新钢

https://xingangpan.github.io/
https://twitter.com/XingangP



Attention

Attention

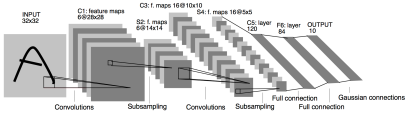
Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is quantified by learned weights and thus the output is usually formed as a weighted average.

Transformers

Background

- Convolutional neural networks (CNN) has been dominating
  - Greater scale
  - More extensive connections
  - More sophisticated forms of convolution



- Transformers
  - Competitive alternative to CNN
  - Generally found to perform best in settings with large amounts of training data
  - Enable multi-modality learning

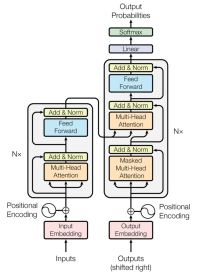
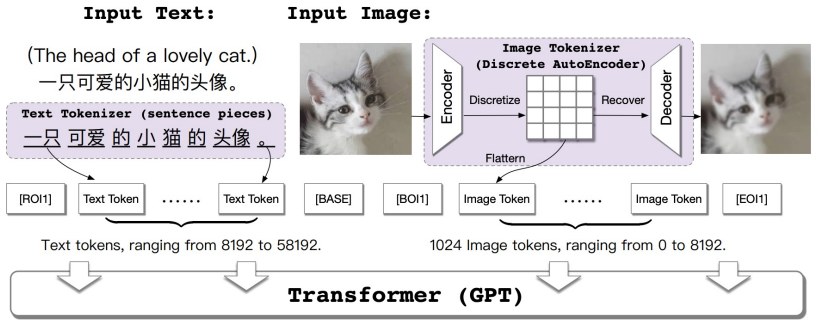


Figure 1: The Transformer - model architecture.

Ashish Vaswani et al., Attention Is All You Need, NIPS 2017 (from Google)

Background



Ding et al., CogView: Mastering Text-to-Image Generation via Transformers, NeurIPS 2021

Transformers

Notable for its use of attention to model long-range dependencies in data

A sequence-to-sequence model

Model of choice in natural language processing (NLP)

Step-by-step guide to self-attention with illustrations and code https://jalammar.github.io/illustrated-transformer/

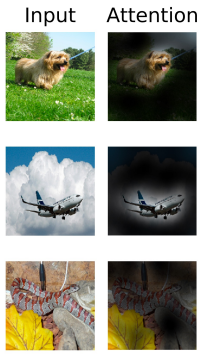
Ashish Vaswani et al., Attention Is All You Need, NIPS 2017 (from Google)

Transformers

Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is quantified by learned weights and thus the output is usually formed as a weighted average.

$$\text{Output} = \mathbf{WV}$$



The model attends to image regions that are semantically relevant for classification

Transformers

Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder)

But it differs from the existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).

- During training, layer outputs can be calculated in parallel, instead of a series like an RNN
- Attention-based models allow modeling of dependencies without regard to their distance in the input or output sequences

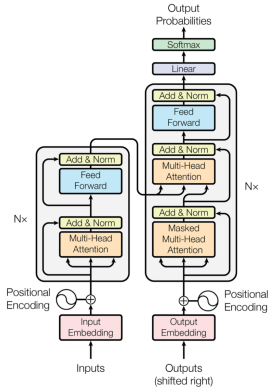
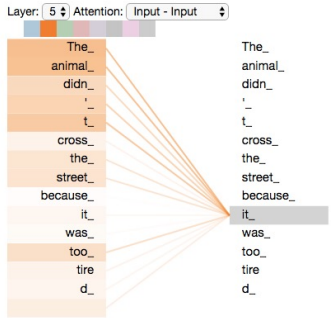


Figure 1: The Transformer - model architecture.

Transformers

It is entirely built on the self-attention mechanisms without using sequence-aligned recurrent architecture

Self-attention is a type of attention mechanism where the model makes prediction for one part of a data sample using other parts of the observation about the same sample.



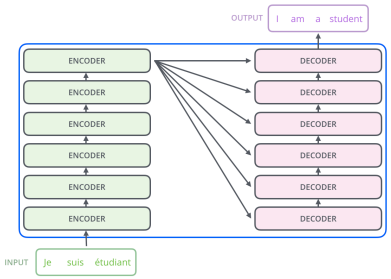
What does “it” in this sentence refer to? Is it referring to the street or to the animal?

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

Transformers

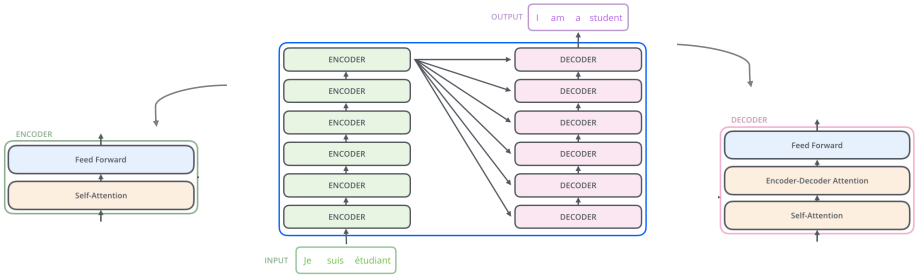
The encoding component is a stack of encoders

The decoding component is a stack of decoders of the same number



Transformers

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence

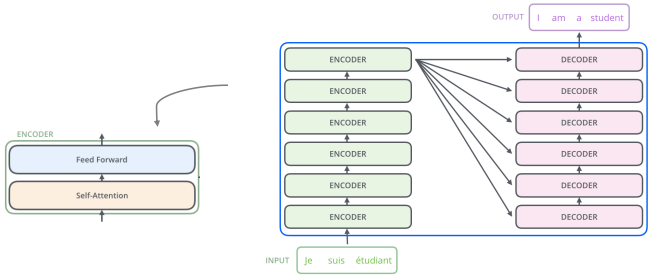


Transformers

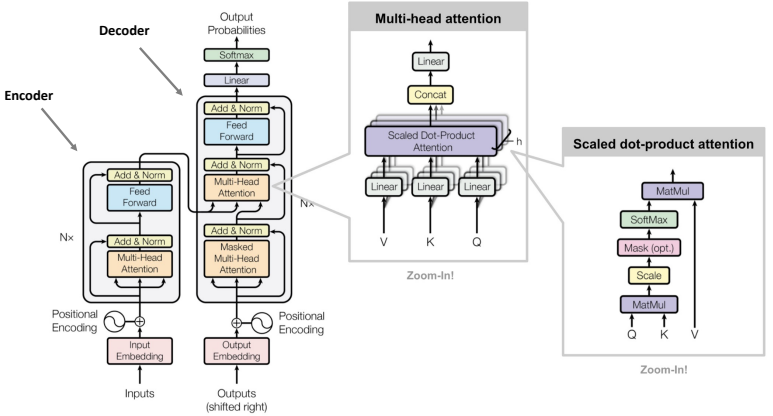
The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word

The outputs of the self-attention layer are fed to a feed-forward neural network.

The exact same feed-forward network is independently applied to each position (each word/token).



Transformers

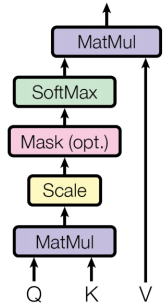


Transformers

Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$



Scaled Dot-Product Attention

Self-Attention in Detail

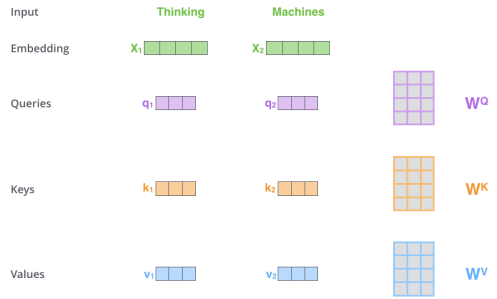
First Step

What are the "query", "key", and "value" vectors?

The names *"query", "key"* are inherited from the field of information retrieval

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}$ depend on the relative similarities between the n-th query and all of the keys

The softmax function means that the key vectors "compete" with one another to contribute to the final result.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

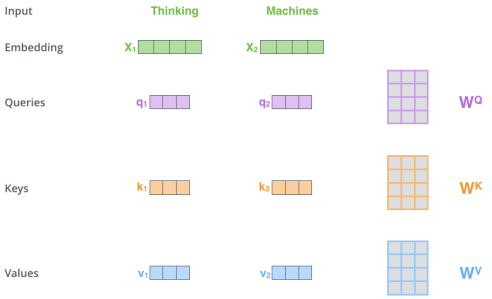
Self-Attention in Detail

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).

So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

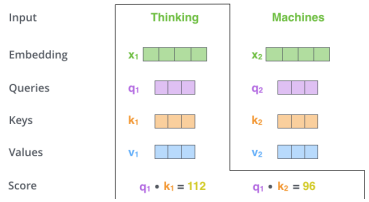
Self-Attention in Detail

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

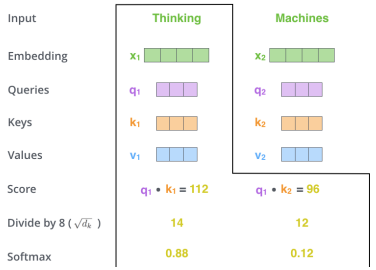
Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)

Fourth Step

Softmax for normalization



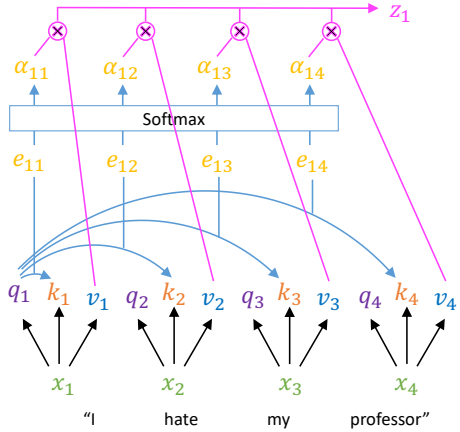
$z_1 = q_1 . k_1 / \sqrt{d_k} = 112/\sqrt{64} = 112/8 = 14, z_2 = q_1 . k_2 / \sqrt{d_k} = 96/\sqrt{64} = 96/8 = 12$

$softmax(z_1) = exp(z_1) / \sum_{i=1}^{2}(exp(z_i)) = 0.88, softmax(z_2) = exp(z_2) / \sum_{i=1}^{2}(exp(z_i)) = 0.12$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

$$e = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}$$



$q = W_q x$

$k = W_k x$

$v = W_v x$

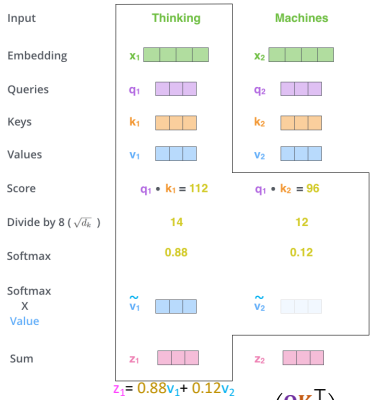
Self-Attention in Detail

Fifth Step

Multiply each value vector by the softmax score

Sixth Step

Sum up the weighted value vectors to get the output of the self-attention layer at this position (for the first word)



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

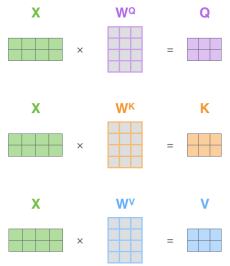
Matrix Calculation of Self-Attention

First Step

Calculate the Query, Key, and Value matrices.

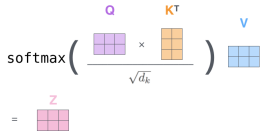
Pack our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V)

Every row in the X matrix corresponds to a word in the input sentence.



Second Step

Calculate the outputs of the self-attention layer.
SoftMax is row-wise



Example

Assuming we have two word embeddings:
$(1, 2, 3)$
$(4, 5, 6)$

And the given $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$ matrices are respectively given as

$$\begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix}, \begin{pmatrix} 0.05 & 0.05 \\ 0.06 & 0.05 \\ 0.07 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix}$$

Compute the output of the scaled-dot product attention, Attention$(\mathbf{Q}, \mathbf{K}, \mathbf{V})$.

eg9.1.ipynb

Example

Step 1: Find $\mathbf{Q}, \mathbf{K}, \mathbf{V}$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^K = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.5 & 0.5 \\ 0.6 & 0.5 \\ 0.7 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.38 & 0.30 \\ 0.92 & 0.75 \end{pmatrix}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^V = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix}$$

eg9.1.ipynb

Example

Step 2: Find Attention$(\mathbf{Q}, \mathbf{K}, \mathbf{V})$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\frac{e^{z_j}}{\sum_j e^{z_j}} = \frac{e^{0.0588}}{e^{0.0588} + e^{0.1441}} = 0.4787$$

Perform row-wise SoftMax

$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) = \text{Softmax}\left(\frac{\begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix}\begin{pmatrix} 0.38 & 0.92 \\ 0.30 & 0.75 \end{pmatrix}}{\sqrt{2}}\right) = \text{Softmax}\begin{pmatrix} 0.0588 & 0.1441 \\ 0.1454 & 0.3566 \end{pmatrix} = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix}$$

$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix}\begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix} = \begin{pmatrix} 0.1326 & 0.1682 \\ 0.1363 & 0.1729 \end{pmatrix}$$

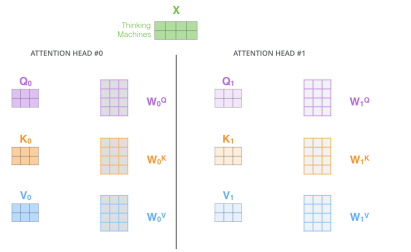
eg9.1.ipynb

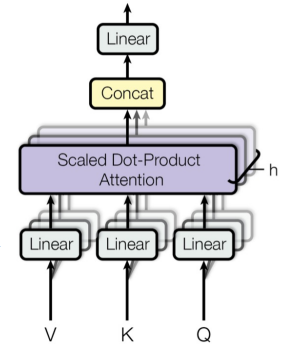
Multi-Head Self-Attention

Multi-Head Self-Attention

Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention multiple times in parallel.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values



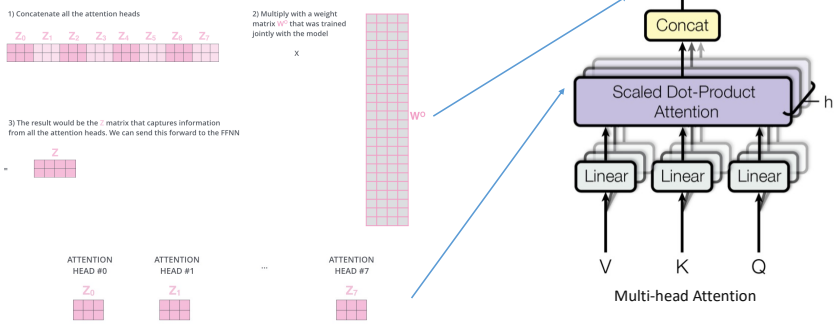


Multi-head Attention

Multi-Head Self-Attention

Multi-Head Self-Attention

The independent attention outputs are simply concatenated and linearly transformed into the expected dimensions.



Multi-head Attention

Multi-Head Self-Attention

Why?

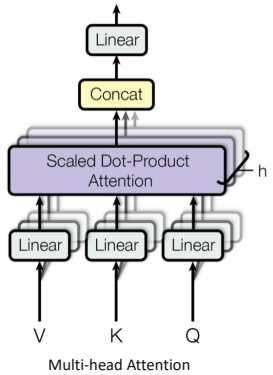
"Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions."

An intuitive example

Given a sentence:

"Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning."

Given "representation learning", the first head attends to "Deep learning" while the second head attends to the more general term "machine learning methods".

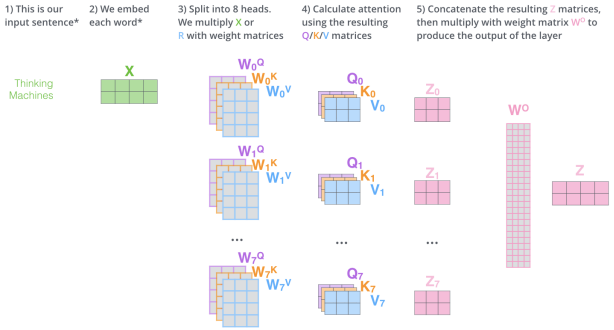


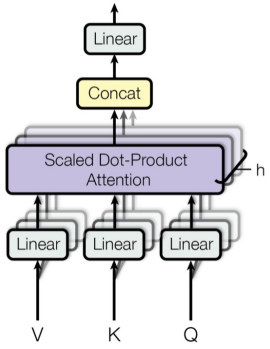
Multi-head Attention

Multi-Head Self-Attention

Multi-Head Self-Attention

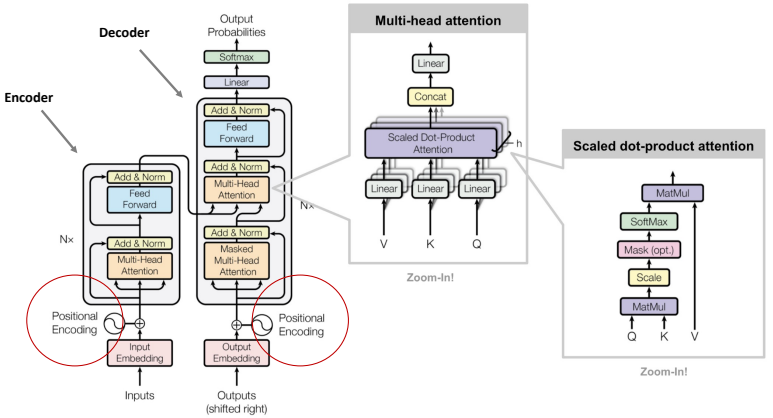
The big picture. Note that after the split each head can have a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.





Multi-head Attention

Positional Encoding



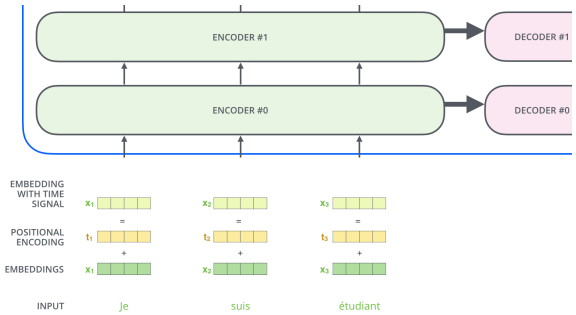
Positional Encoding

Self-attention layer works on sets of vectors and it doesn't know the order of the vectors it is processing

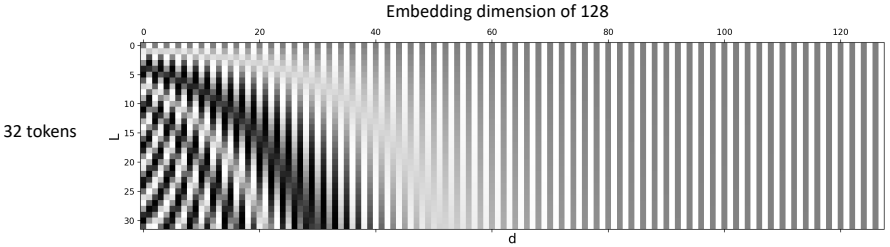
The positional encoding has the same dimension as the input embedding

Adds a vector to each input embedding to give information about the relative or absolute position of the tokens in the sequence

These vectors follow a specific pattern



Positional Encoding



*Sinusoidal positional encoding - interweaves the two signals (*sine for even indices and cosine for odd indice*)*

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension, $[0, \dots, d_{model}/2\ )$

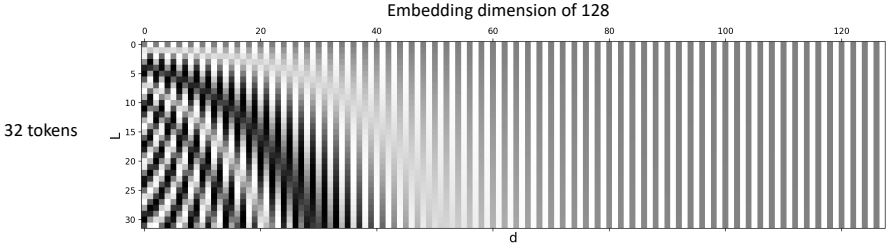
Positional Encoding

What might this pattern look like?

Each row corresponds the a positional encoding of a vector.

The first row would be the vector we'd add to the embedding of the first word in an input sequence.

Each position is uniquely encoded and the encoding can deal with sequences longer than any sequence seen in the training time.



Sinusoidal positional encoding with 32 tokens and embedding dimension of 128. The value is between -1 (black) and 1 (white) and the value 0 is in gray.

Example: Positional Encoding

Example:

Given the following *Sinusoidal positional encoding*, calculate the $PE(pos = 1)$ for the first five dimensions [0, 1, 2, 3, 4]. Assume $d_{model} = 512$

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Solution:

Given $pos = 1$ and $d_{model} = 512$

At dimension 0, $2i = 0$ thus $i = 0$, therefore $PE_{(pos,2i)} = PE_{(1,0)} = \sin(1/10000^{0/512})$

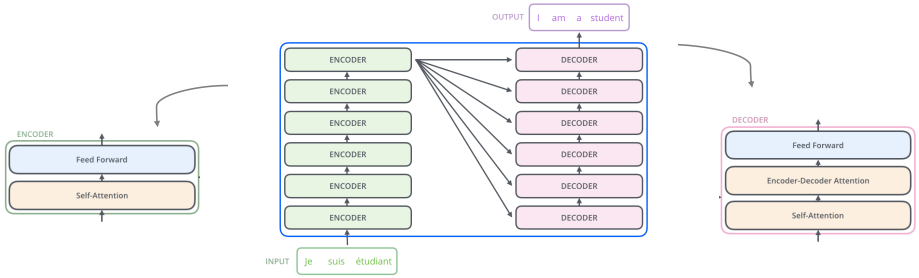
At dimension 1, $2i + 1 = 1$ thus $i = 0$, therefore $PE_{(pos,2i+1)} = PE_{(1,1)} = \cos(1/10000^{0/512})$

At dimension 2, $2i = 2$ thus $i = 1$, therefore $PE_{(pos,2i)} = PE_{(1,2)} = \sin(1/10000^{2/512})$

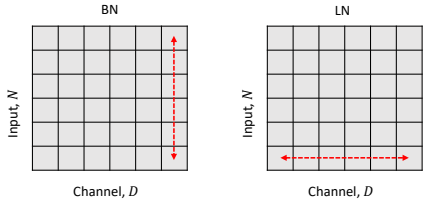
At dimension 3, $2i + 1 = 3$ thus $i = 1$, therefore $PE_{(pos,2i+1)} = PE_{(1,3)} = \cos(1/10000^{2/512})$

At dimension 4, $2i = 4$ thus $i = 2$, therefore $PE_{(pos,2i)} = PE_{(1,4)} = \sin(1/10000^{4/512})$

Transformers

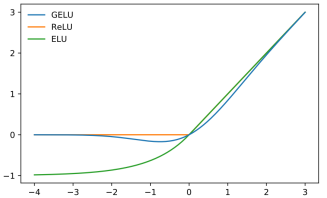


Transformer Encoder



Layer Normalization (LN) [1]

- The pixels along the red arrow are normalized by the same mean and variance, computed by aggregating the values of these pixels.
- BN is found unstable in Transformers [2]
- Works well with RNN and now being used in Transformers



Gaussian Error Linear Units (GELU) [3]

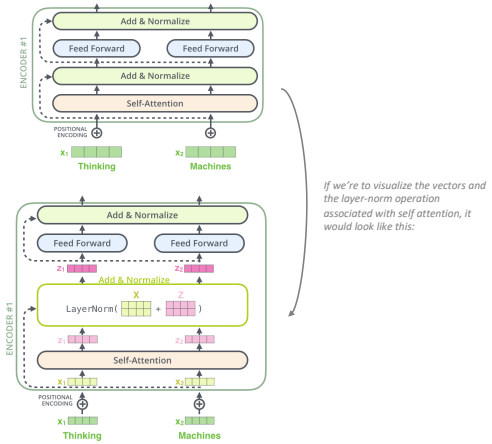
- Can be thought of as a smoother ReLU
- Used in GPT-3, BERT, and most other Transformers

[1] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton, Layer Normalization, arXiv:1607.06450

[2] Sheng Shen, Zhewei Yao, Amir Gholami, Michael Mahoney, Kurt Keutzer, Rethinking Batch Normalization in Transformers, ICML 2020

[3] Dan Hendrycks, Kevin Gimpel, Gaussian Error Linear Units (GELUs), arXiv:1606.08415

Transformer Encoder



If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

Encoder

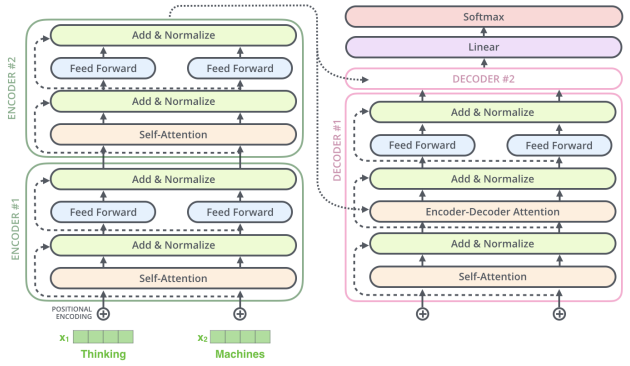
- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

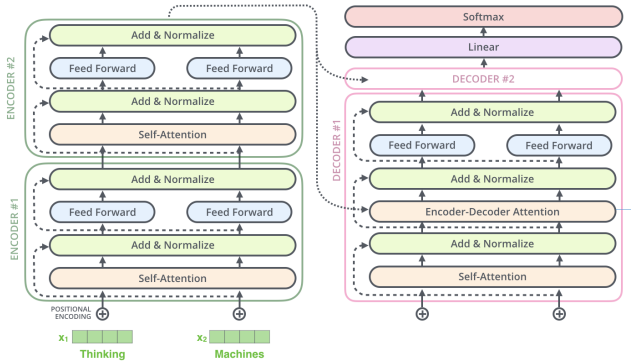
Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



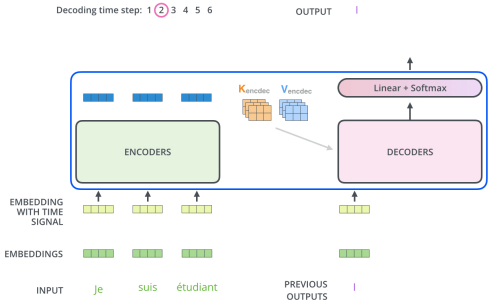
The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

Transformer Decoder

How encoder and decoder work together

- The output of each step is fed to the bottom decoder in the next time step
- Embed and add positional encoding to those decoder inputs. Process the inputs
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.

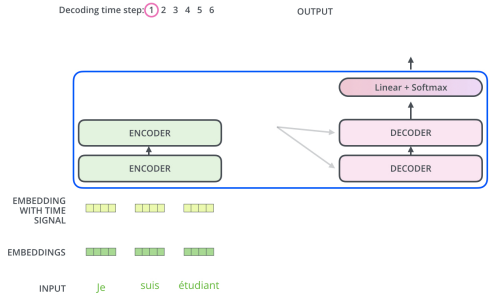
Note:
In the decoder, *the self-attention layer is only allowed to attend to earlier positions in the output sequence*. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.



Transformer Decoder

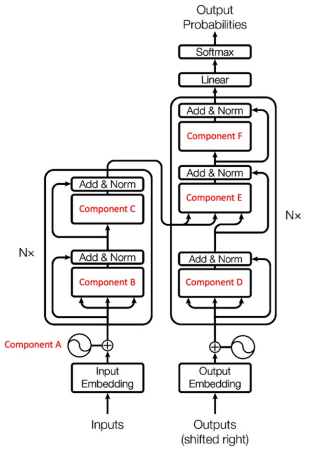
How encoder and decoder work together

- The encoder start by processing the input sequence
- The output of the top encoder is then transformed into a set of attention vectors K and V.
- These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence



After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

Transformer



Vision Transformer

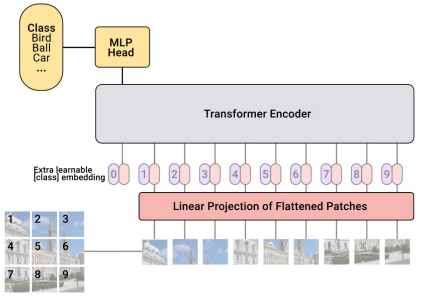
Vision Transformer (ViT)

Prepend a learnable embedding ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}$$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

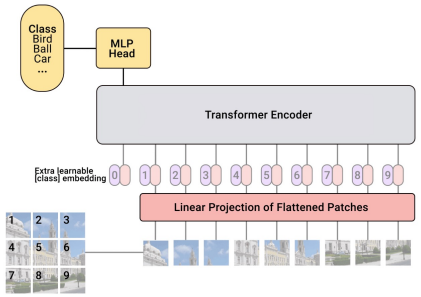
Add learnable position embeddings $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information



Alexey Dosovitskiy et al., An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ICLR 2021

Vision Transformer (ViT)

- Do not have decoder
- Reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
  - (H, W) is the resolution of the original image
  - C is the number of channels
  - (P, P) is the resolution of each image patch
  - $N = HW/P^2$ is the resulting number of patches



Alexey Dosovitskiy et al., An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ICLR 2021

Vision Transformer (ViT)

Feed the resulting sequence of vectors $\mathbf{z}_0$ to a standard Transformer encoder

Transformer Encoder / Classification Head

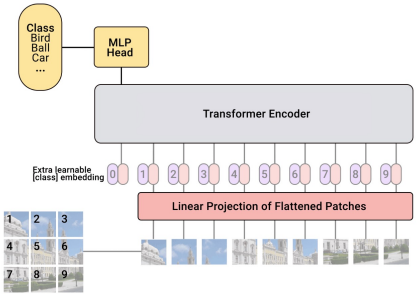
$$\begin{aligned}
\mathbf{z}_0 &= [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \\
\mathbf{z}'_\ell &= \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \\
\mathbf{z}_\ell &= \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \\
\mathbf{y} &= \text{LN}(\mathbf{z}_L^0)
\end{aligned}$$

$\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$, $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$

$\ell = 1 \ldots L$

$\ell = 1 \ldots L$

The output of the additional [class] token is transformed into a class prediction via a small multi-layer perceptron (MLP) with tanh as non-linearity in the single hidden layer.

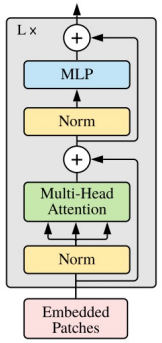


Vision Transformer (ViT)

Transformer Encoder

- Consists of a multi-head self-attention module (MSA), followed by a 2-layer MLP (with GELU)
- LayerNorm (LN) is applied before MSA module and MLP, and a residual connection is applied after each module.

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1},$$
$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell,$$



Transformer Encoder

Vision Transformer (ViT)

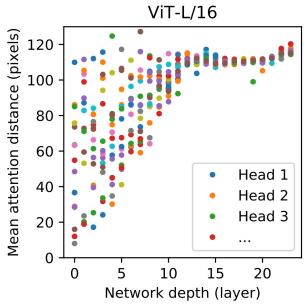
Examine the attention distance, analogous to receptive field in CNN

Compute the average distance in image space across which information is integrated, based on the attention weights.

Attention distance was computed for 128 example images by averaging the distance between the query pixel and all other pixels, weighted by the attention weight.

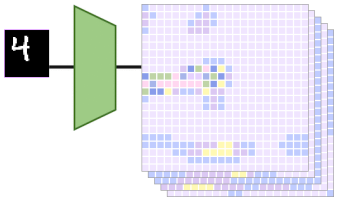
Each dot shows the mean attention distance across images for one of 16 heads at one layer. Image width is 224 pixels.

An example: if a pixel is 20 pixels away and the attention weight is 0.5 the distance is 10.



Inductive Bias

- ViT has much less image-specific inductive bias than CNNs
- Inductive bias in CNN
  - Locality
  - Two-dimensional neighborhood structure
  - Translation equivariance
- ViT
  - Only MLP layers are local and translationally equivariant. Self-attention layer is global
  - Two dimensional neighborhood is used sparingly – i) only at the beginning where we cut image into patches, ii) learnable position embedding (spatial relations have to be learned from scratch)



An equivariant mapping is a mapping which preserves the algebraic structure of a transformation.

A translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping

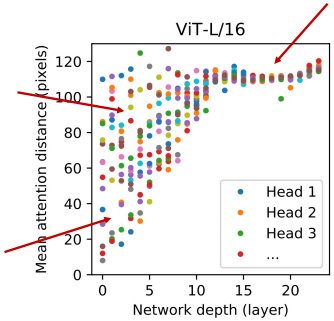
Vision Transformer (ViT)

Examine the attention distance, analogous to receptive field in CNN

The attention distance increases with network depth

Some heads attend to most of the image already in the lowest layers, showing the capability of ViT in integrating information globally

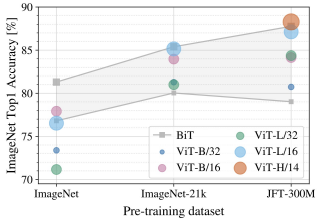
Other attention heads have consistently small attention distances in the low layers



Vision Transformer (ViT)

Performance of ViT

- ViT performs significantly worse than the CNN equivalent (BiT) when trained on ImageNet (1M images).
- However, on ImageNet-21k (14M images) performance is comparable, and on JFT (300M images), ViT outperforms BiT.
- ViT overfits the ImageNet task due to its lack of inbuilt knowledge about images



Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

SC 4001 CE/CZ 4042 Neural Network and Deep Learning

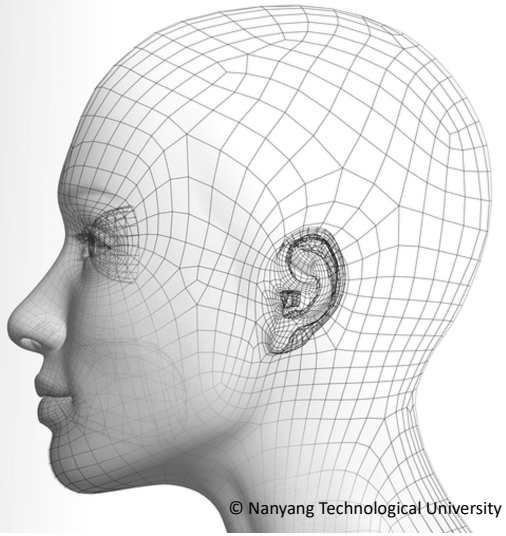
Last update: 17 Mar 2024

Autoencoders

Xingang Pan
潘新钢

https://xingangpan.github.io/

https://twitter.com/XingangP



Vision Transformer (ViT)

ViT conducts global self-attention

- Relationships between a token and all other tokens are computed
- Quadratic complexity with respect to the number of tokens
- Not tractable for dense prediction or to represent a high-resolution image

Outline

- Supervised vs unsupervised learning
- Autoencoders
  - Denoising autoencoders
  - Undercomplete and overcomplete autoencoders
  - Sparse autoencoders
  - Other encoders

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Classification

Cat

Supervised vs Unsupervised Learning

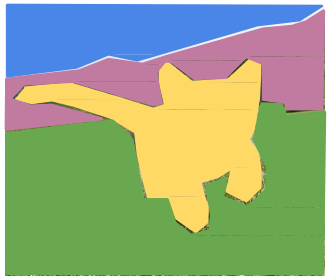
Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Semantic Segmentation



GRASS, CAT, TREE, SKY

Supervised vs Unsupervised Learning

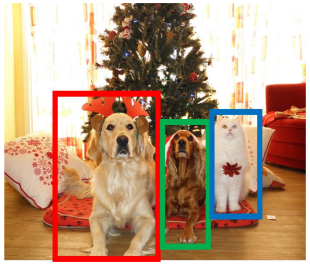
Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Object Detection



DOG, DOG, CAT

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Image captioning

A cat sitting on a suitcase on the floor

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Unsupervised Learning

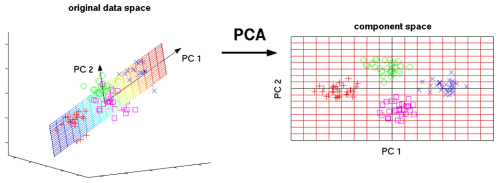
Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Dimensionality Reduction
(e.g. Principal Components Analysis)



3D → 2D

Unsupervised Learning

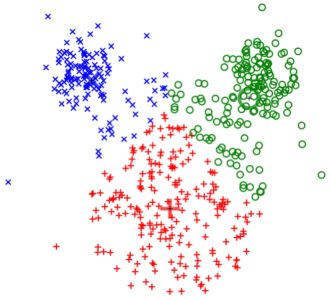
Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Clustering
(e.g. K-Means)



Unsupervised Learning

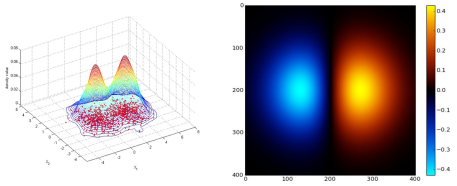
Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Density Estimation



Unsupervised Learning

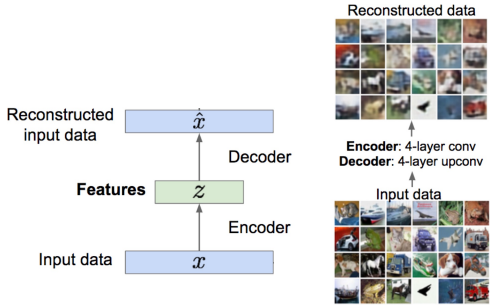
Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Feature Learning
(e.g. autoencoders)



Unsupervised Learning

Data: x

Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

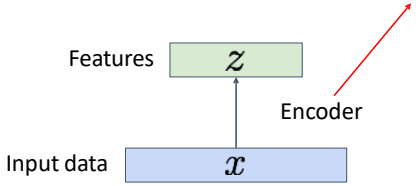
Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

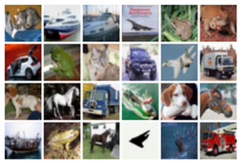
Autoencoders

Problem: How can we learn this feature transform from raw data?

Features should extract useful information (maybe object identities, properties, scene type, etc) that we can use for downstream tasks
But we can't observe features!

Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN



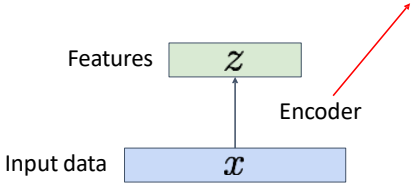

Input Data

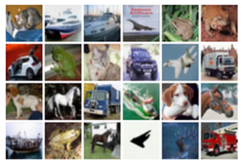
Autoencoders

Unsupervised method for learning feature vectors from raw data x, without any labels

Features should extract useful information (maybe object identities, properties, scene type, etc) that we can use for downstream tasks

Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN



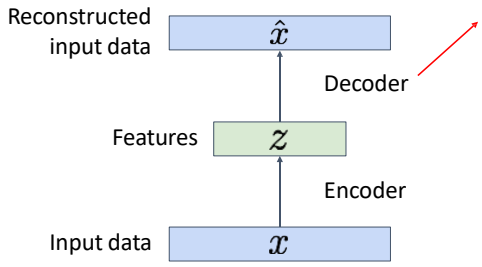

Input Data

Autoencoders

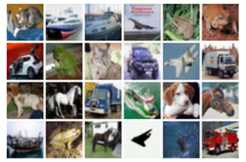
- **Problem**: How can we learn this feature transform from raw data?

Idea: Use the features to <u>reconstruct</u> the input data with a **decoder**

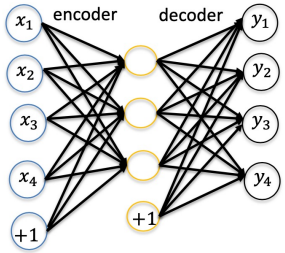
"Autoencoding" = encoding itself



Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN (upconv)


Input Data

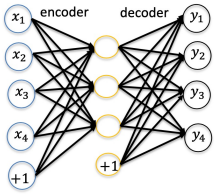
Autoencoders



An autoencoder is a neural network that is trained to attempt to copy its input to its output. It's hidden layer describes a *code* that represents the input.

The network consists of two parts: an encoder and a decoder.

Autoencoders



Given an input $\boldsymbol{x}$, the hidden-layer performs the encoding function $\boldsymbol{h} = \emptyset(\boldsymbol{x})$ and the decoder $\boldsymbol{\psi}$ produces the reconstruction $\mathbf{y} = \boldsymbol{\psi}(\boldsymbol{h})$.

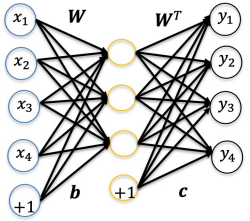
If the autoencoder succeeds:

$$\mathbf{y} = \boldsymbol{\psi}(\boldsymbol{h}) = \boldsymbol{\psi}(\emptyset(\boldsymbol{x})) = \boldsymbol{x}$$

In order to be useful, autoencoders are designed to be unable to copy exactly and enable to copy only inputs that resembles the training data.

Since the model is forced to prioritize which aspects of the input should be copied, the hidden-layer often learns useful properties of the data.

Autoencoders



Reverse mapping from the hidden layer to the output can be optionally constrained to be the same as the input to hidden-layer mapping (tied weights). That is, if encoder weight matrix is $\boldsymbol{W}$, the decoder weigh matrix is $\boldsymbol{W^T}$.

Hidden layer and output layer activation can be then written as

$$\boldsymbol{h} = f(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$$
$$\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{c})$$

$\boldsymbol{f}$ is usually a sigmoid.

Training autoencoders

The cost function of reconstruction can be measured by many ways, depending on the appropriate distributional assumptions on the inputs.

Learning of autoencoders is unsupervised as no specific targets are given.

Given a training set $\{\boldsymbol{x}_p\}_{p=1}^{P}$.

The mean-square-error cost is usually used if the data is assumed to be continuous and Gaussian distributed:

$$J_{mse} = \frac{1}{P}\sum_{p=1}^{P}\|\boldsymbol{y}_p - \boldsymbol{x}_p\|^2$$

where $\boldsymbol{y}_p$ is the output for input $\boldsymbol{x}_p$and $\|\cdot\|$ denotes the magnitude of the vector.

Training autoencoders

If the inputs are interpreted as bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

$$J_{cross-entropy} = -\sum_{p=1}^{P}\left(\boldsymbol{x}_p \log \boldsymbol{y}_p + \left(1-\boldsymbol{x}_p\right)\log\left(1-\boldsymbol{y}_p\right)\right)$$

Learning are done by using gradient descent:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha\nabla_{\boldsymbol{W}}J$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha\nabla_{\boldsymbol{b}}J$$
$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha\nabla_{\boldsymbol{c}}J$$

Example 1

Given input patterns:
(1, 0, 1, 0, 0) , (0, 0, 1, 1, 0) , (1, 1, 0, 1, 1) and (0, 1, 1, 1, 0)

Design an autoencoder with 3 hidden units.
Show the representations of inputs at the hidden layer upon convergence.

$$\boldsymbol{H} = f(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{B})$$
$$\boldsymbol{Y} = f(\boldsymbol{H}\boldsymbol{W}^T + \boldsymbol{C})$$
$$\boldsymbol{O} = 1(\boldsymbol{Y} > 0.5)$$

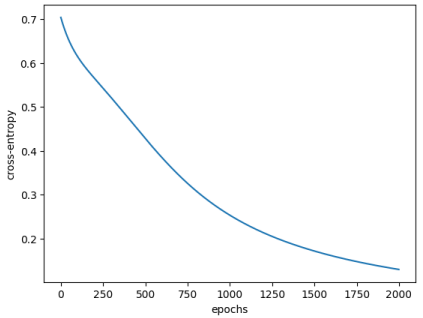
Gradient descent on entropy:

$$J = -\sum_{p=1}^{4}\left(\boldsymbol{x}_p \log \boldsymbol{y}_p + \left(1-\boldsymbol{x}_p\right)\log\left(1-\boldsymbol{y}_p\right)\right)$$

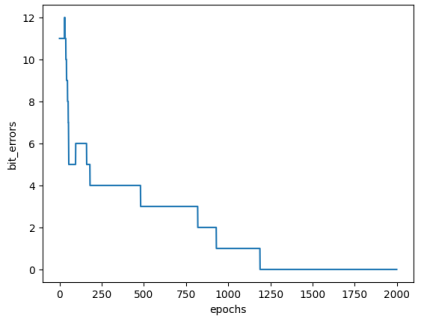
$$bit\ errors = \sum_{p=1}^{4}\sum_{k=1}^{5} 1(x_{pk} \neq o_{pk})$$

eg10.1.ipynb

Example 1



Example 1



Example 1

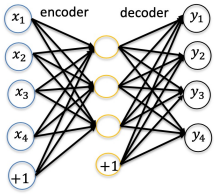
```
# Display weights and biases
print(f'W:\n {autoencoder.W.data}\n')
print(f'b:\n {autoencoder.b.data}\n')
print(f'b_prime:\n {autoencoder.b_prime.data}\n')
```

```
W:
 tensor([[-3.6867, -0.3808,  2.6520],
        [-0.2545, -3.6158, -1.8657],
        [ 1.7148,  3.0138,  0.9254],
        [ 1.9011, -0.9880, -3.5277],
        [-2.4830, -2.5980, -1.1112]])

b:
 tensor([ 0.1496,  0.5881, -0.1604])

b_prime:
 tensor([ 1.2448,  2.3414, -0.7427,  2.1173,  0.9872])
```


Denoising Autoencoders (DAE)



A *denoising autoencoder* (DAE) receives corrupted data points as inputs.

It is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to force the hidden layer to discover more robust features and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to undo the effect of corruption process applied to the input of the autoencoder.

DAE

For training DAE:

- First, input data is corrupted to mimic the noise in the images: $\boldsymbol{x} \rightarrow \tilde{\boldsymbol{x}}$

  $\tilde{\boldsymbol{x}}$ is the corrupted version of data. The corruption process simulates the distribution of data

- The network is trained to produce uncorrupted data: $\boldsymbol{y} \rightarrow \boldsymbol{x}$

DAE: Corrupting inputs

To obtain corrupted version of input data, each input x_i of input data is added with additive or multiplicative noise.

Additive noise:

$$\tilde{x_i} = x_i + \varepsilon$$

where noise ε is Gaussian distributed: $\varepsilon \sim N(0, \sigma^2)$
And σ is the standard deviation that determines the S/N ratio. Usually used for continuous data.

Multiplicative noise:

$$\tilde{x_i} = \varepsilon x_i$$

where noise ε could be Bernoulli distributed: $\varepsilon \sim Bernoulli(p)$
And p is the probability of ones and $1 - p$ is the probability of zeros (noise). Usually, used for binary data.

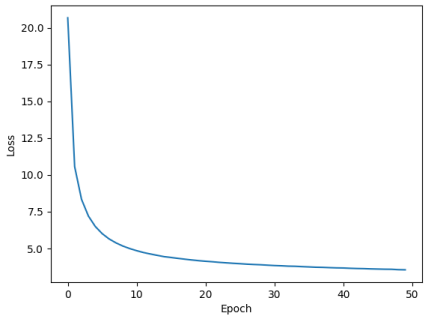
Example 2: Denoising autoencoders

The MNIST database of gray level images of handwritten digits:
http://yann.lecun.com/exdb/mnist/

Each image is 28x28 size.
Intensities are in the range [0, 255] and normalized to [0, 1].
Training set: 60,000, Test set: 10,000

To build a DAE with 500 hidden units.

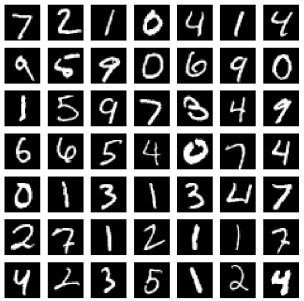
Example 2a: DAE (multiplicative noise)



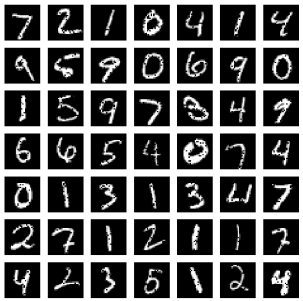
500 hidden units
α = 0.1

Example 2a: DAE (multiplicative noise)

Original data



Corrupted data

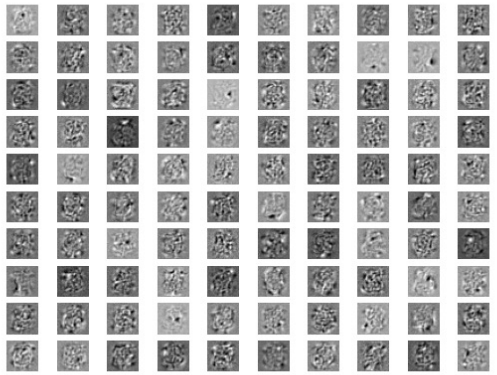


DAE attempts to predict the input data from corrupted input data with multiplicative noise with binomial distribution.
Corruption level 1 – p = 10%

eg10.2a.ipynb

Example 2a: DAE (multiplicative noise)

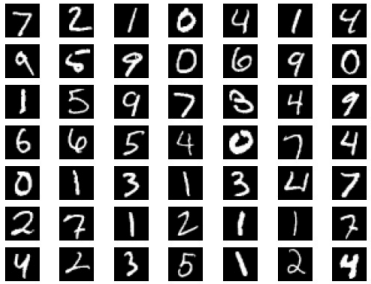
Weights learned by the hidden-layer



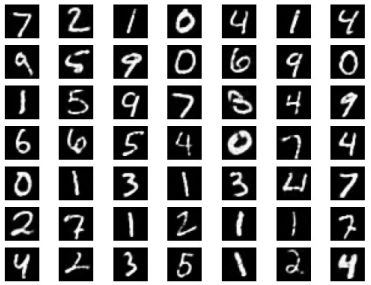
Example 2a: DAE (multiplicative noise)

Sample of reconstructed test images

Input data

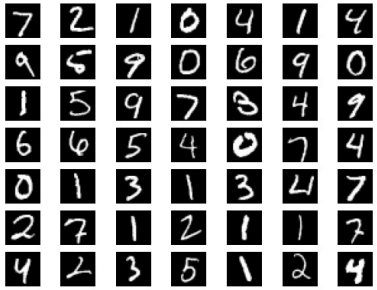


Reconstructed (noise-filtered) data

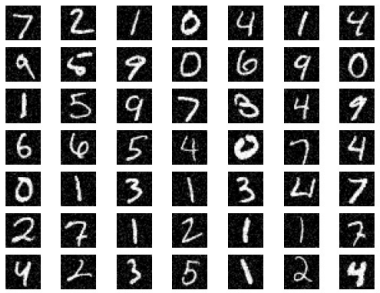


Example 2b: DAE (additive noise)

Original data



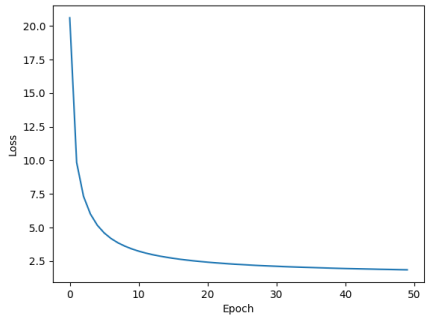
Corrupted data



Additive noise with gaussian distribution.
Data is scaled between[0, 1], noise s.d = 0.1

eg10.2b.ipynb

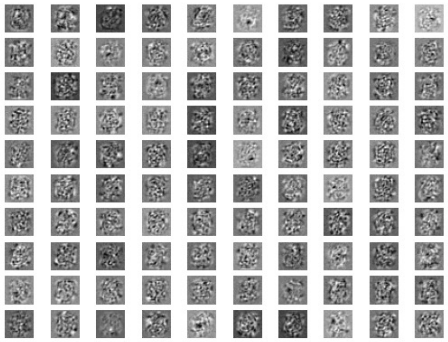
Example 2b: DAE (additive noise)



500 hidden units
$\alpha = 0.1$

Example 2b: DAE (additive noise)

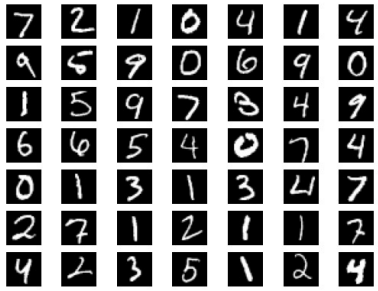
Features (weights) learned by the hidden-layer



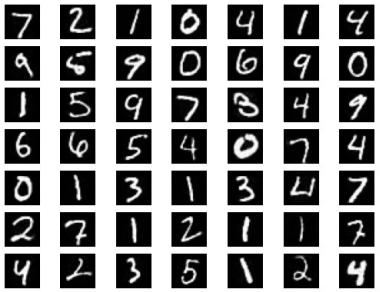
Example 2b: DAE (additive noise)

Sample of reconstructed images

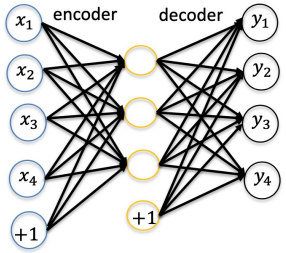
Input data



Reconstructed (noise-filtered) data



Autoencoders

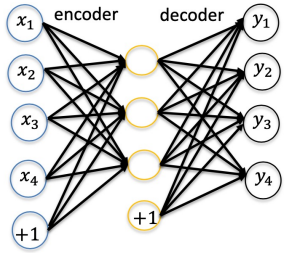


Input dimension n and hidden dimension M:

If $M < n$, *undercomplete* autoencoders

If $M > n$, *overcomplete* autoencoders

Undercomplete autoencoders



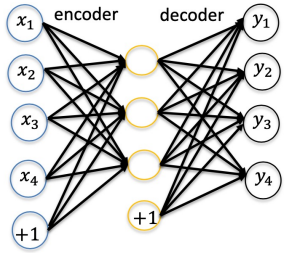
In undercomplete autoencoders, the hidden-layer has a lower dimension than the input layer.

By learning to approximate an *n*-dimensional inputs with *M* (< *n*) number of hidden units, we obtain *a lower dimensional representation* of the input signals. The network reconstructs the input signals from the reduced-dimensional hidden representation.

Learning an undercomplete representation forces the autoencoder to capture the most salient features.

By limiting the number of hidden neurons, hidden structures of input data can be inferred from autoencoders. For example, correlations among input variables, learning principal components of data, etc.

Overcomplete autoencoders

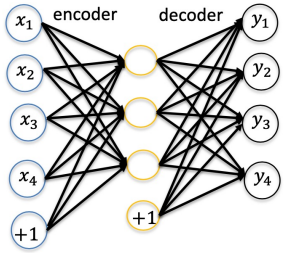


In overcomplete autoencoders, the hidden-layer has a higher dimension than the dimension of the input.

In order to learn useful information from overcomplete autoencoders, it is necessary use some constraints on its characteristics.

Even when the hidden dimensions are large, one can still explore interesting structures of inputs by introducing other constraints such as 'sparsity' of input data.

Regularizing autoencoders



Both undercomplete and overcomplete autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. Some form of constraints are needed in order to make them useful.

Deep autoencoders are only possible when some constraints are imposed on the cost function.

Regularizing autoencoders

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

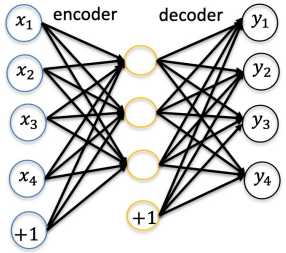
$$J_1 = J + \beta\, \Omega(h)$$

where β is the penalty or regularization parameter. The penalty is usually imposed on the hidden activations.

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn trivial identity function.

The regularized loss function encourages the model to have other properties besides the ability to copy its input to its output.

Regularizing autoencoders



Regularized autoencoders incorporate a penalty to the cost function to learn interesting features from the input.

Regularized autoencoders provide the ability to train any autoencoder architecture successfully by choosing suitable code dimension, the capacity of the encoder and decoder, and the strength of regularization.

With regularized autoencoders, one can use larger model capacity (for example, deeper autoencoders) and large code size.

Sparse Autoencoders (SAE)

A sparse autoencoder (SAE) is simply an autoencoder whose training criterion involves the sparsity penalty Ω_{sparsity} at the hidden layer:

$$J_1 = J + \beta\, \Omega_{sparsity}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be sparse.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be inactive for most of the time.

We say that the neuron is “active” when its output is close to 1 and the neuron is “inactive” when its output is close to 0.

Sparsity constraint

For a set $\{x_p\}_{p=1}^{P}$ of input patterns, the average activation ρ_j of neuron j at the hidden-layer is given by:

$$\rho_j = \frac{1}{P}\sum_{p=1}^{P} h_{pj} = \frac{1}{P}\sum_{p=1}^{P} f(\boldsymbol{x}_p^T \boldsymbol{w}_j + b_j)$$

where h_{pj} is the activation of hidden neuron j for p th pattern, and $\boldsymbol{w}_j$ and b_j are the weights and biases of the hidden neuron j .

We would like to enforce the constraint: $\rho_j = \rho$ such that the *sparsity parameter* ρ is set to a small value close to zero (say 0.05).

That is, most of the time, hidden neuron activations are maintained at ρ on average. By choosing a smaller value for ρ, the neurons are activated selectively to patterns and thereby learn sparse features.

Sparsity constraint

To achieve sparse activations at the hidden-layer, the Kullback-Leibler (KL) divergence is used as the sparsity constraint:

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log\frac{\rho}{\rho_j} + (1-\rho)\log\frac{1-\rho}{1-\rho_j}$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the deviation of the distribution $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ.

The KL divergence is minimum when $\rho_j = \rho$ for all j.
That is, when the average activations are uniform and equal to very low value ρ.

Sparse Autoencoder (SAE)

The cost function for the sparse autoencoder (SAE) is given by

$$J_1 = J + \beta D(\boldsymbol{h})$$

where $D(h)$ is the KL divergence of hidden-layer activations:

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log\frac{\rho}{\rho_j} + (1-\rho)\log\frac{1-\rho}{1-\rho_j}$$

For gradient descent,

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta \nabla_{\boldsymbol{W}} D(\boldsymbol{h})$$

SAE

By chain rule:

$$\frac{\partial D(\boldsymbol{h})}{\partial \boldsymbol{w}_j} = \frac{\partial D(\boldsymbol{h})}{\partial \rho_j}\frac{\partial \rho_j}{\partial \boldsymbol{w}_j} \quad \text{(A)}$$

where

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log\frac{\rho}{\rho_j} + (1-\rho)\log\frac{1-\rho}{1-\rho_j}$$

and

$$\frac{\partial D(\boldsymbol{h})}{\partial \rho_j} = \rho\frac{1}{\frac{\rho}{\rho_j}}\left(-\frac{\rho}{{\rho_j}^2}\right) + (1-\rho)\frac{1}{\left(\frac{1-\rho}{1-\rho_j}\right)}\left(\frac{-(1-\rho)}{\left(\frac{1-\rho}{1-\rho_j}\right)^2}\right)$$

$$= -\frac{\rho}{\rho_j} + \frac{1-\rho}{1-\rho_j} \quad \text{(B)}$$

https://www.mathway.com/popular-problems/Calculus/529317

SAE

Note:

$$\rho_j = \frac{1}{P}\sum_{p=1}^{P} f(u_{pj})$$

where $u_{pj} = x_p^T w_j + b_j$ is the synaptic input of the j th neuron due to p th pattern.

$$\frac{\partial \rho_j}{\partial \boldsymbol{w}_j} = \frac{1}{P}\sum_p f'\left((u_{pj})\right)\frac{\partial (u_{pj})}{\partial \boldsymbol{w}_j} = \frac{1}{P}\sum_p f'\left((u_{pj})\right)\boldsymbol{x}_p \qquad \text{(c)}$$

Substituting (B) and (C) in (A):

$$\frac{\partial D(\boldsymbol{h})}{\partial \boldsymbol{w}_j} = \frac{1}{P}\left(-\frac{\rho}{\rho_j} + \frac{1-\rho}{1-\rho_j}\right)\sum_p f'\left((u_{pj})\right)\boldsymbol{x}_p$$

Example 3

Design the following autoencoder to process MNIST images:
1. Undercomplete autoencoder with 100 hidden units
2. Overcomplete autoencoder with 900 hidden units
3. Sparse autoencoder with 900 hidden units. Use sparsity parameter = 0.5

The MNIST database of gray level images of handwritten digits:
http://yann.lecun.com/exdb/mnist/

Each image is 28x28 size.
Intensities are in the range [0, 255].
Training set: 60,000
Test set: 10,000

SAE

Substituting in (A) , we can find:

$$\nabla_{\boldsymbol{W}} D(\boldsymbol{h}) = (\nabla_{\boldsymbol{w}_1} D(\boldsymbol{h}) \quad \nabla_{\boldsymbol{w}_2} D(\boldsymbol{h}) \quad \cdots \quad \nabla_{\boldsymbol{w}_M} D(\boldsymbol{h}))$$

The gradient of the constrained cost function:

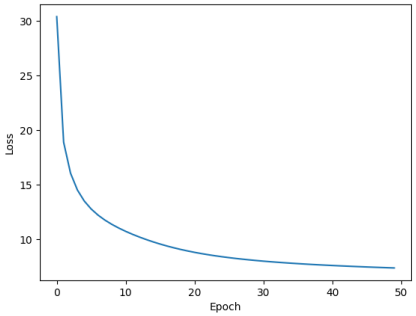
$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta \nabla_{\boldsymbol{W}} D(\boldsymbol{h})$$

Similarly, $\nabla_{\boldsymbol{b}} J_1$ and $\nabla_{\boldsymbol{c}} J_1$ can be derived.

Learning can be done as

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J_1$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J_1$$
$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \nabla_{\boldsymbol{c}} J_1$$

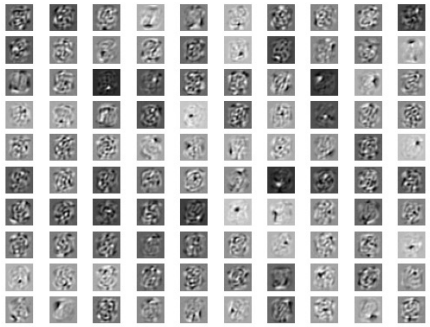
Example 3a: Undercomplete autoencoder



eg10.3a.ipynb

Example 3a: Undercomplete autoencoder

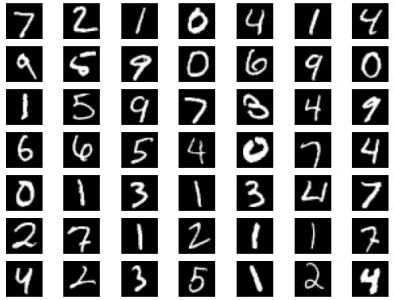
Learned weights



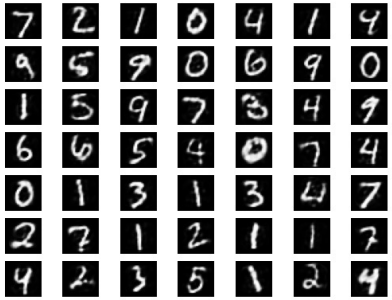
100 hidden units

Example 3a: Undercomplete autoencoder

Test inputs



Reconstructed inputs

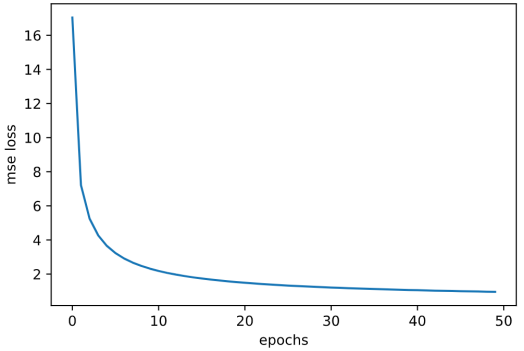


Example 3a: Undercomplete autoencoder

Hidden layer activations of 49 test patterns

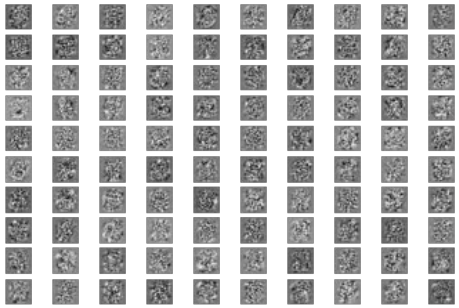
10x10 neurons

Example 3b: Overcomplete autoencoder



Example 3b: Overcomplete autoencoder

Learned weights



900 units

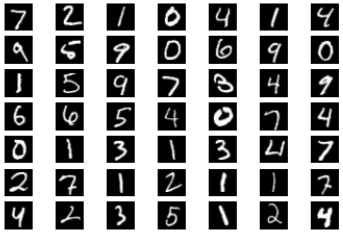
Example 3b: Overcomplete autoencoder

Hidden layer activations

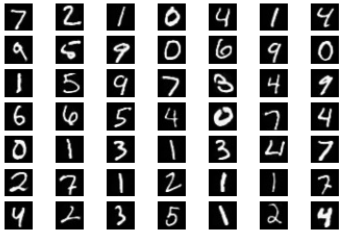
30x30 neurons

Example 3b: Overcomplete autoencoder

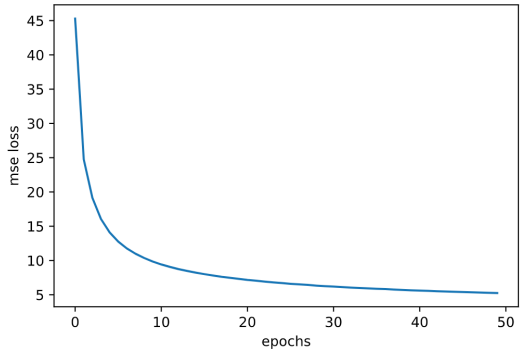
Test inputs



Reconstructed inputs

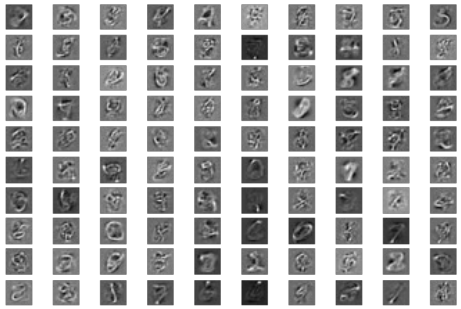


Example 3c: Sparse autoencoder



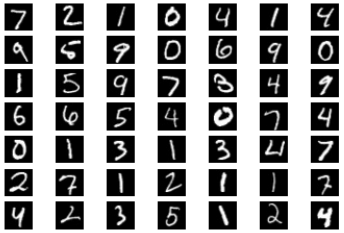
Example 3c: Sparse autoencoder

Learned weights

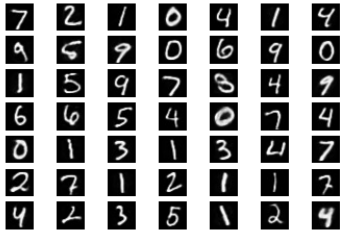


Example 3c: Sparse autoencoder

Test inputs

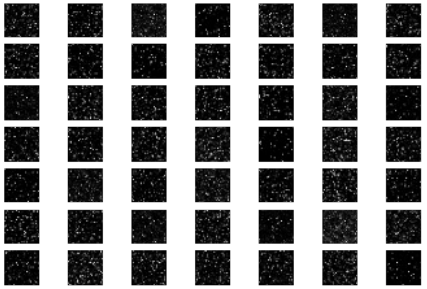


Reconstructed inputs



Example 3c: Sparse autoencoder

Hidden layer activations



30x30 neurons

Deep stacked autoencoders

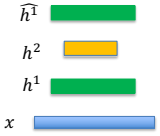
Deep autoencoders can be built by stacking autoencoders one after the other.

Training of deep autoencoders is done in a step-by-step fashion one layer at a time:

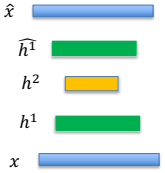
- After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder.
- The second level hidden representation can be used to train the third level of the encoders.
- This process is repeated and deep stacked autoencoder can be realized.

Deep stacked autoencoders

Training first hidden layer



Training second hidden layer. Reconstructs first-hidden layer output

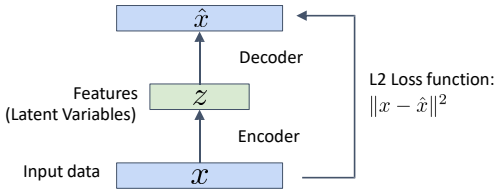


Autoencoder with two hidden layers

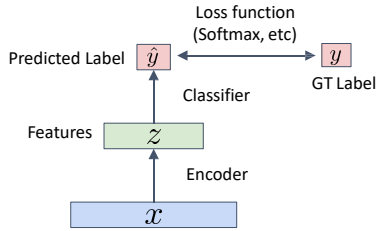
Semi-Supervised Classification

- Many images, but few ground truth labels

start unsupervised
train autoencoder on many images

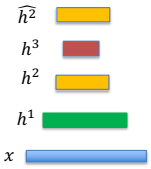


supervised fine-tuning
train classification network on labeled images

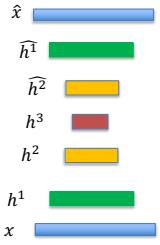


Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Deep stacked autoencoders



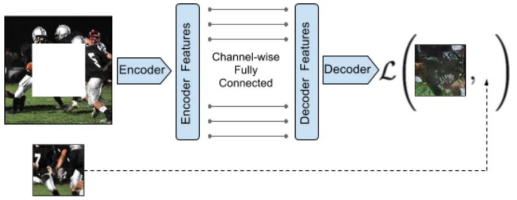
Training the third hidden layer. Reconstructs second hidden layer output



Autoencoder with three hidden layers

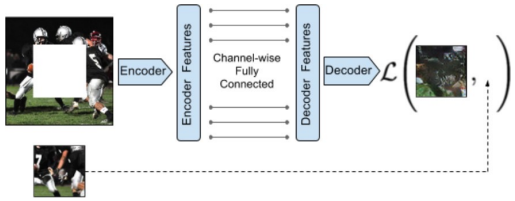
Context Encoders

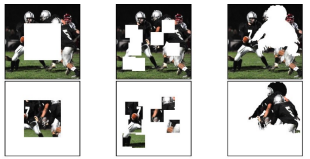
[Pathak et al., 2016]



Context Encoders

[Pathak et al., 2016]

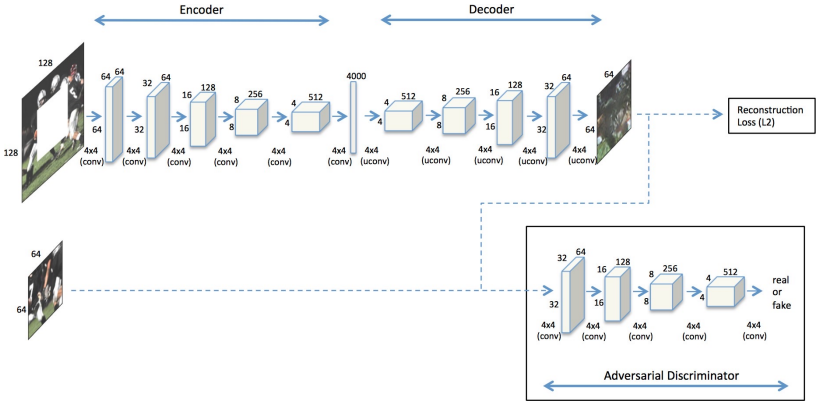




(a) Central region (b) Random block (c) Random region

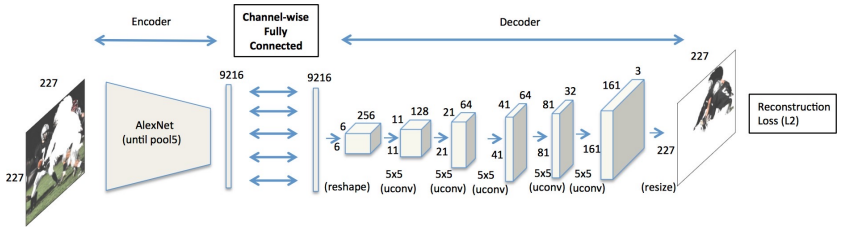
Context Encoders

[Pathak et al., 2016]



Context Encoders

[Pathak et al., 2016]



SC 4001 CE/CZ 4042 Neural Network and Deep Learning

Last update: 17 March 2024

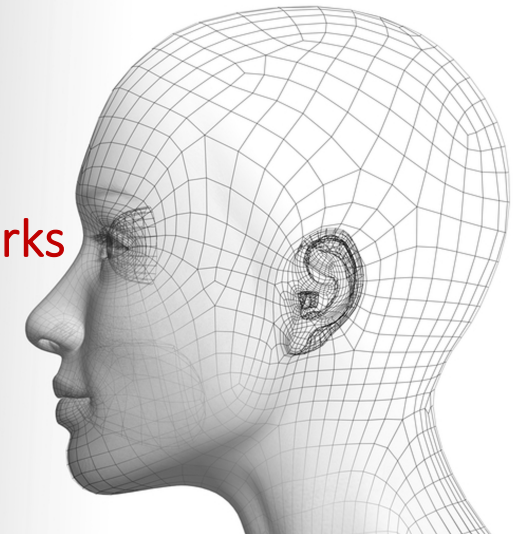
Generative Adversarial Networks

Xingang Pan

潘新钢

https://xingangpan.github.io/

https://twitter.com/XingangP



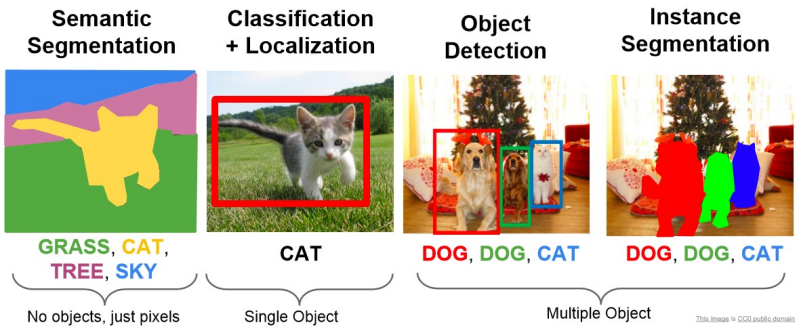
Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Why generative models?

- **We've only seen *discriminative models* so far**
  - Given an data $\boldsymbol{x}$, predict a label $\boldsymbol{y}$
  - Estimates $p(\boldsymbol{y}|\boldsymbol{x})$
- **Discriminative models have several key limitations**
  - Can't model $p(\boldsymbol{x})$, i.e., the probability of seeing a certain data
  - Thus, can't sample from $p(\boldsymbol{x})$, i.e., **can't generate new data**
- ***Generative models* (in general) cope with all of above**
  - Can model $p(\boldsymbol{x})$
  - Can generate new data or images

Applications of supervised learning



Generative modeling

Density estimation – a core problem in unsupervised learning

Sample generation

Training samples – $p_{data}(x)$

Model samples – $p_{model}(x)$
(ideally $p_{model}(x) = p_{data}(x)$)

Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

GAN Basics

Generative Adversarial Networks (GAN)

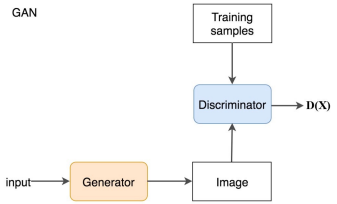
$z \sim \mathcal{N}(0, 1)$
or
$z \sim$ U (-1, 1)

$z = (0.3, 0.2, -0.6, ...)$ $\xrightarrow{g(z)}$ 

$z = (-0.1, 0.1, 0.2, ...)$ $\xrightarrow{g(z)}$ 

GAN samples noise ***z*** using normal or uniform distribution and utilizes a deep network generator ***G*** to create an image ***x (x=G(z))***

Generative Adversarial Networks (GAN)

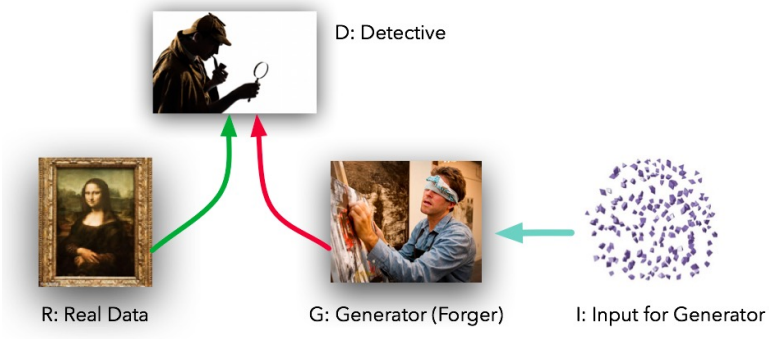


In GAN, we add a discriminator to distinguish whether the discriminator input is real or generated. It outputs a value ***D(x)*** to estimate the chance that the input is real.

D(x) = 1 suggests x is real

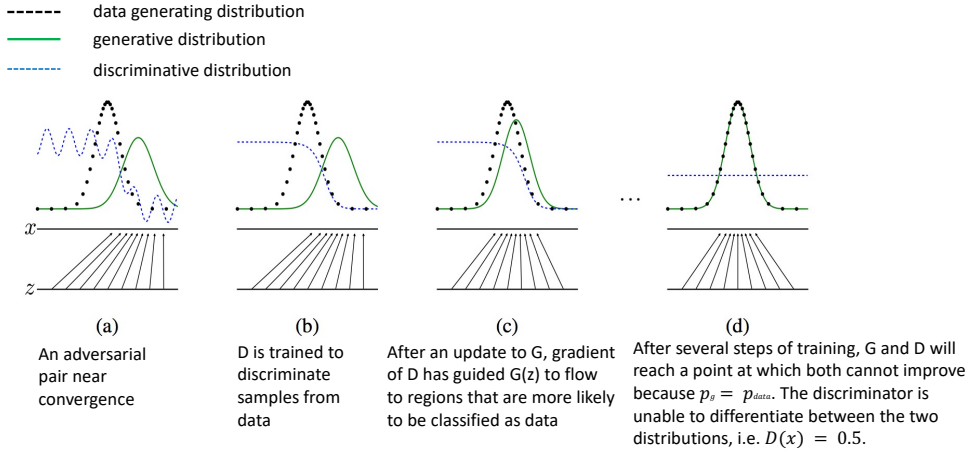
D(x) = 0 suggests x is fake

Generative Adversarial Networks (GAN)



Goodfellow et al., "Generative adversarial networks", NIPS 2014

Generative adversarial networks (GAN)



(a) An adversarial pair near convergence

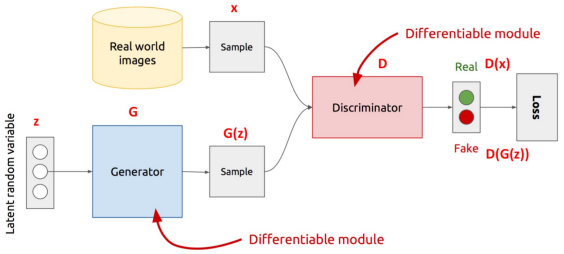
(b) D is trained to discriminate samples from data

(c) After an update to G, gradient of D has guided G(z) to flow to regions that are more likely to be classified as data

(d) After several steps of training, G and D will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(x) = 0.5$.

Generative adversarial networks (GAN)

- Generative model G:
  - Captures data distribution
  - Fool $D(G(z))$
  - Generate an image $G(z)$ such that $D(G(z))$ is wrong (i.e. $D(G(z)) = 1$)
- Discriminative model D:
  - Distinguishes between real and fake samples
  - $D(x) = 1$ when x is a real image, and otherwise



z is some random noise (Gaussian/Uniform).
z can be thought as the latent representation of the data.

Generative adversarial networks (GAN)

- D and G play the following two-player minimax game with value function $V(G, D)$

Generator: input = random vector z
Output: $G(z)$ as synthetic image

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Discriminator: input = generated/real images
Output: $D(x) = 1$ if real image

Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

GAN Training

Training procedure

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

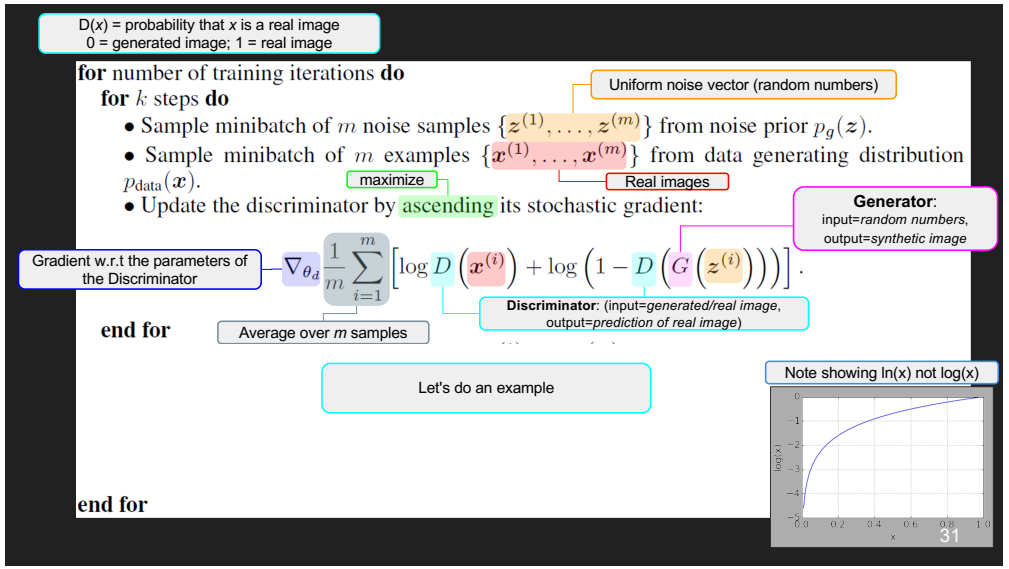
for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient: maximize

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient: minimize

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.



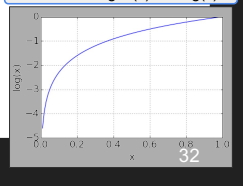
D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
D(x) = 0.8
log(0.8) = -0.2
Imagine for a real image D(x) scores 0.8 that it is a real image (good)
Note showing ln(x) not log(x)



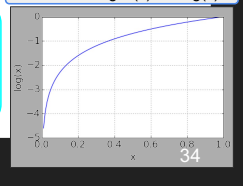
D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
D(x) = 0.8
log(0.8) = -0.2
D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
-0.2 + -0.2 = -0.4
We add them together and this gives us a fairly high (-0.4) loss (note we ascend so we want to maximize this).
Also note that we are adding two negative numbers so 0 is the upper bound
Note showing ln(x) not log(x)



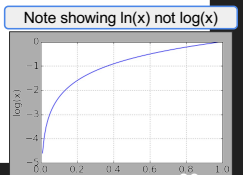
D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
D(x) = 0.8
log(0.8) = -0.2
D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
Then for a **generated** image, D(x) scores 0.2 that it is a real image (good)
Note showing ln(x) not log(x)



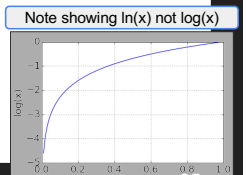
D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
D(x) = 0.8
log(0.8) = -0.2
D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
-0.2 + -0.2 = -0.4
Let's do another example
Note showing ln(x) not log(x)



D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

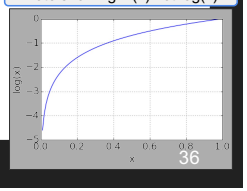
Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
Note showing ln(x) not log(x)

D(x) = 0.8
log(0.8) = -0.2

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
-0.2 + -0.2 = -0.4

D(x) = 0.2
log(0.2) = -1.6

D(x) scores 0.2 that a real image is a real image (bad)



D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
Note showing ln(x) not log(x)

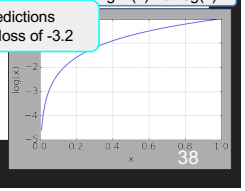
D(x) = 0.8
log(0.8) = -0.2

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
-0.2 + -0.2 = -0.4

D(x) = 0.2
log(0.2) = -1.6

D(G(z)) = 0.8
log(1-0.8) = log(0.2) = -1.6
-1.6 + -1.6 = -3.2

These "bad" predictions combined gives a loss of -3.2



D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
Note showing ln(x) not log(x)

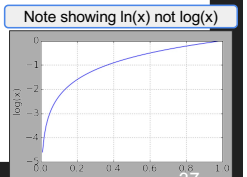
D(x) = 0.8
log(0.8) = -0.2

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
-0.2 + -0.2 = -0.4

D(x) = 0.2
log(0.2) = -1.6

D(G(z)) = 0.8
log(1-0.8) = log(0.2) = -1.6

D(x) scores 0.8 that the generated image is a real image (bad)



D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
end for

Uniform noise vector (random numbers)
Real images
maximize
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
Note showing ln(x) not log(x)

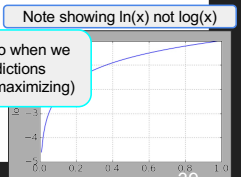
D(x) = 0.8
log(0.8) = -0.2

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
-0.2 + -0.2 = -0.4

D(x) = 0.2
log(0.2) = -1.6

D(G(z)) = 0.8
log(1-0.8) = log(0.2) = -1.6
-1.6 + -1.6 = -3.2

Compare the loss to when we had "good" predictions (remember we are maximizing)



D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

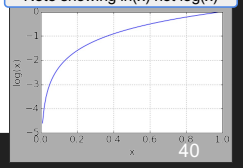
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

Note showing ln(x) not log(x)



end for

D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

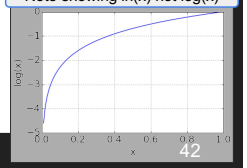
  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

D(G(z)) = 0.2

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

D(G(z)) scores 0.2 that a generated image is a real image = bad :(We didn't fooled D(.)

Note showing ln(x) not log(x)



end for

D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

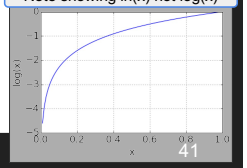
Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

Note showing ln(x) not log(x)



end for

D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

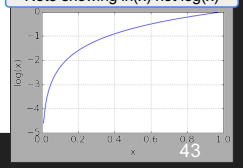
  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

Gets assigned a loss of -0.2

Note showing ln(x) not log(x)



end for

D(x) = probability that x is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

minimize

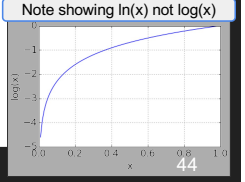
D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

Notice here we want to minimize this loss function (not maximize like before)

  end for

Note showing ln(x) not log(x)



D(x) = probability that x is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

minimize

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2

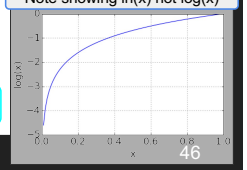
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

D(G(z)) = 0.8

D(G(z)) scores 0.8 that a generated image is a real image = good :) We fooled D(.)!

  end for

Note showing ln(x) not log(x)



D(x) = probability that x is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

minimize

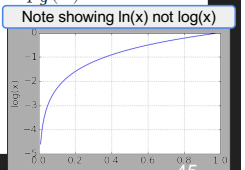
D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

One more example ...

  end for

Note showing ln(x) not log(x)



D(x) = probability that x is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

Uniform noise vector (random numbers)

Real images

maximize

Generator: input=*random numbers*, output=*synthetic image*

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Discriminator: (input=*generated/real image*, output=*prediction of real image*)

Average over *m* samples

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

minimize

D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2

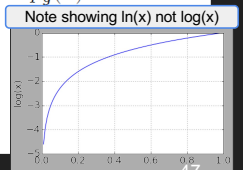
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

D(G(z)) = 0.8
log(1-0.8) = log(0.2) = -1.6

This gives loss of -1.6

  end for

Note showing ln(x) not log(x)



D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

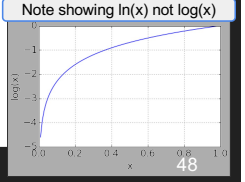
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

Uniform noise vector (random numbers)
maximize
Real images
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
minimize
Note showing ln(x) not log(x)
D(G(z)) = 0.2
log(1-0.2) = log(0.8) = -0.2
D(G(z)) = 0.8
log(1-0.8) = log(0.2) = -1.6
So minimizing this loss means to generate images that fool D(.)



48

Results from GAN

Training a system for rendering arbitrary images (arbitrary within the bounds of the subject matter that we trained the system on)

D(x) = probability that *x* is a real image
0 = generated image; 1 = real image

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

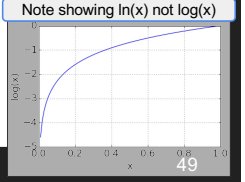
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

Uniform noise vector (random numbers)
maximize
Real images
Generator: input=*random numbers*, output=*synthetic image*
Gradient w.r.t the parameters of the Discriminator
Discriminator: (input=*generated/real image*, output=*prediction of real image*)
Average over *m* samples
minimize
Note showing ln(x) not log(x)
Gradient w.r.t the parameters of the Generator



49

Example 1

Design a GAN to learn Gaussian distributed data with mean = 1.0 and standard deviation = 1.0. The generator receives uniformly distributed 1-dimensional inputs in the range [-1, +1].

Discriminator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 10 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 10 (activation: Tanh)
- Output dimension = 1 (binary classification)

Generator is a 4-layer DNN:

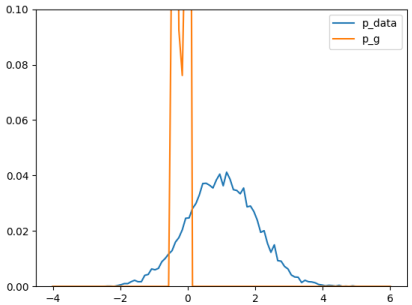
- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 5 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 5 (activation: Tanh)
- Output dimension = 1 (activation: None)

eg11.1.ipynb

Example 1

Histogram of 10000 randomly drawn points from U[-1, 1]

Before training

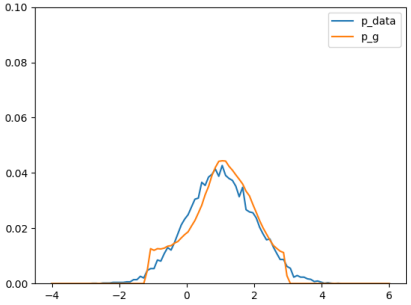


Centered at 0

Example 1

Histogram of 10000 randomly drawn points from U[-1, 1]

70000 iterations

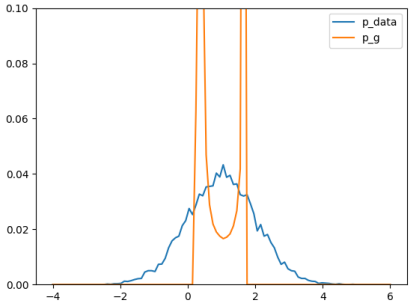


Not perfect, but somehow similar to the data distribution

Example 1

Histogram of 10000 randomly drawn points from U[-1, 1]

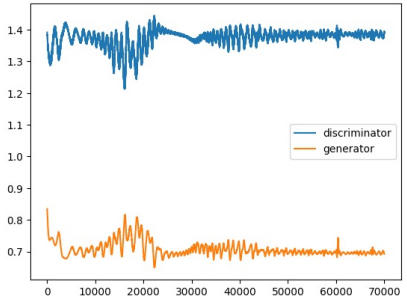
15000 iterations



Symmetric about 1, but not same distribution

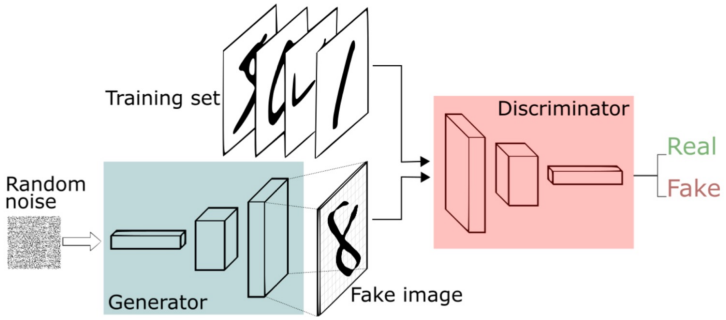
Example 1

Training Curves

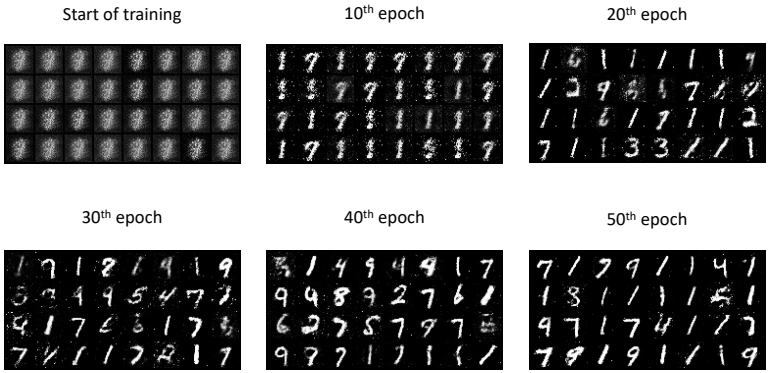


Example 2

Design a GAN to generate MNIST images from a uniformly distributed noise vector of 100 dimensions.



Example 2



Example 2

Generator:

- Input dimension = 100, drawn from a uniform distribution U(-1, 1)
- Hidden layers
  - Number of hidden neurons = 256 (activation: ReLU)
  - Number of hidden neurons = 512 (activation: ReLU)
  - Number of hidden neurons = 1024 (activation: ReLU)
- Output dimension = 784 (activation: Tanh)

Discriminator:

- Input dimension = 784
- Hidden layers
  - Number of hidden neurons = 1024 (activation: ReLU)
  - Number of hidden neurons = 512 (activation: ReLU)
  - Number of hidden neurons = 256 (activation: ReLU)
- Output dimension = 1 (activation: Sigmoid, binary classification)

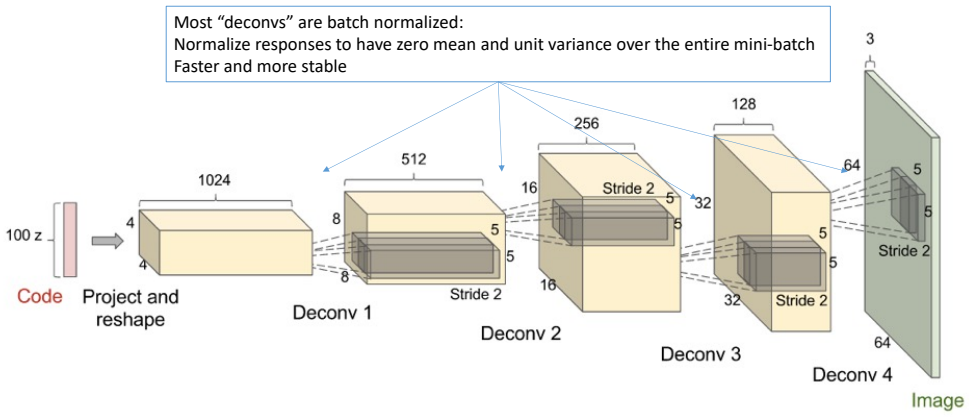
eg11.2.ipynb

Outline

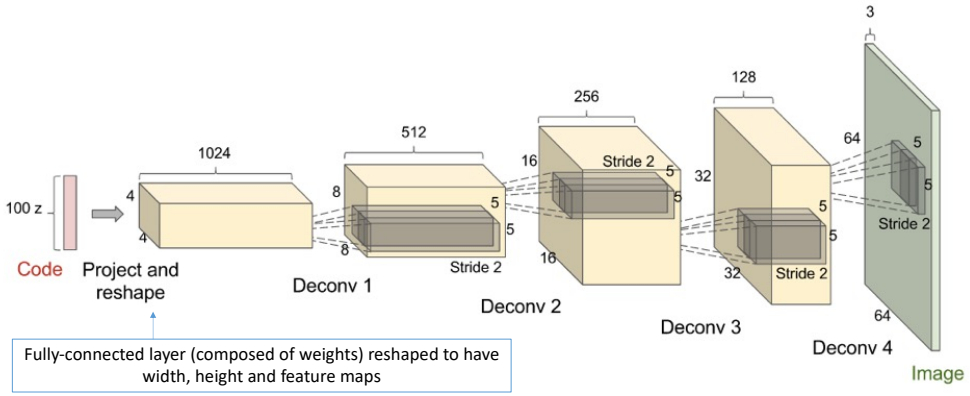
- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

DCGAN

Deep Convolutional GAN (DCGAN)

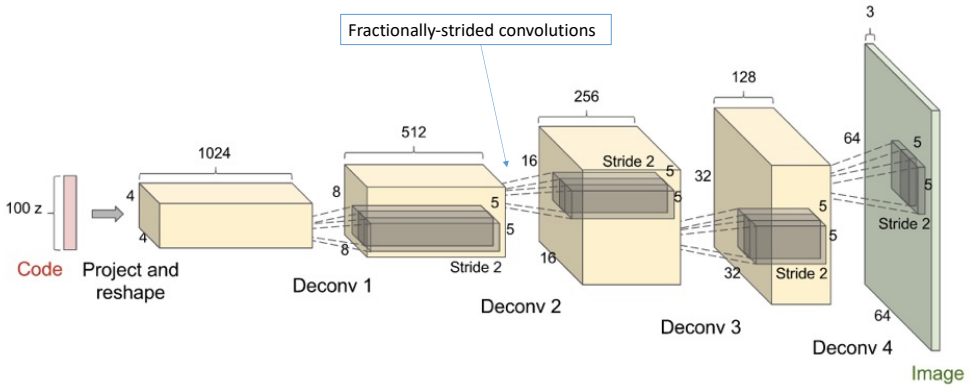


Deep Convolutional GAN (DCGAN)



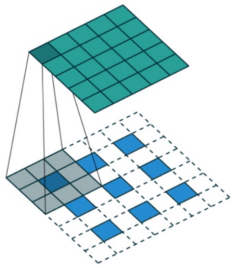
We will go through the code in the tutorial

Deep Convolutional GAN (DCGAN)



Fractionally-strided convolutions

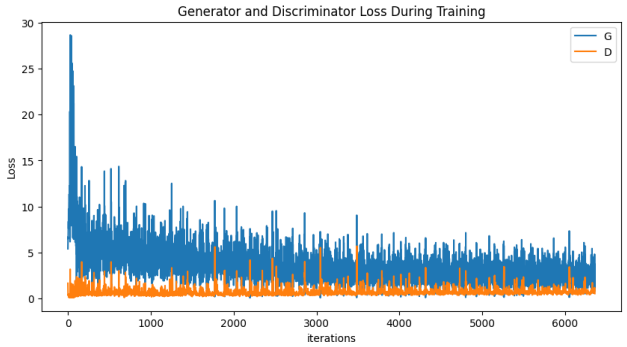
Fractional strides involve: zeros are inserted *between* input units, which makes the kernel move around at a slower pace than with unit strides



https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html

Up-sampling by fractional striding
Input 3x3, output = 5x5
stride =1convolution window = 3 x 3 (with respect to output)

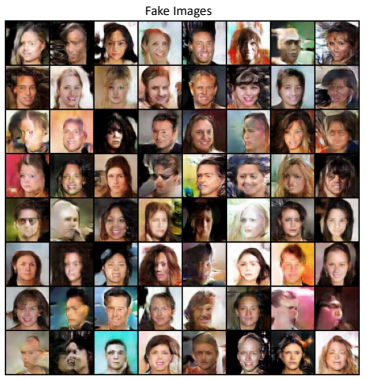
DCGAN



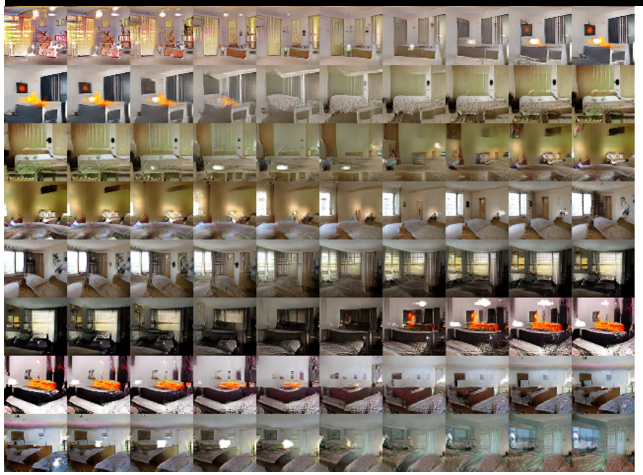
Other tricks proposed by DCGAN

- Adam optimizer (adaptive moment estimation) = similar to SGD but with less parameter tuning
- Momentum = 0.5 (usually is 0.9 but training oscillated and was unstable)
- Low learning rate = 0.0002

DCGAN



Sanity check by walking on the manifold



Interpolating between a series of random points in $\boldsymbol{z}$
($\boldsymbol{z}$ = the 100-d random numbers)
e.g.,

$$\boldsymbol{z}_{new} = m\,\boldsymbol{z}_1 + (1-m)\boldsymbol{z}_2$$

Note the smooth transition between each scene

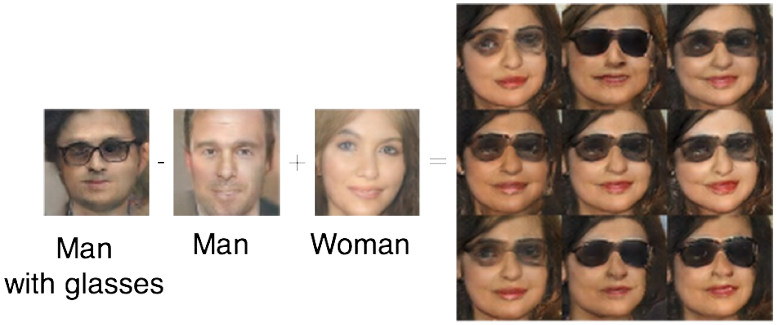
←the window slowly appearing

This indicates that the space learned has smooth transitions!
(and is not simply memorizing)

Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Sanity check by vector space arithmetic



Mode Collapse

Advantages of GANs

- **Plenty of existing work on Deep Generative Models**
  - Boltzmann Machine
  - Deep Belief Nets
  - Variational AutoEncoders (VAE)

- **Why GANs?**
  - Sampling (or generation) is straightforward.
  - Training doesn't involve Maximum Likelihood estimation (i.e. finding the best model parameters that fit the training data the most), believed to cause poorer quality of images (blurry images)
  - Robust to overfitting since the generator never sees the training data.

Non-convergence

- **Deep Learning models (in general) involve a single player**
  - The player tries to maximize its reward (minimize its loss).
  - Use SGD (with Backpropagation) to find the optimal parameters.
  - SGD has convergence guarantees (under certain conditions).
  - **Problem:** With non-convexity, we might converge to local optima.

$$\min_G L(G)$$

- **GANs instead involve two (or more) players**
  - Discriminator is trying to maximize its reward.
  - Generator is trying to minimize Discriminator's reward.

$$\min_G \max_D V(D,G)$$

  - SGD was not designed to find the Nash equilibrium of a game.
  - **Problem:** We might not converge to the Nash equilibrium at all.

Problems with GAN

- **Probability Distribution is Implicit**
  - Not straightforward to compute $p(x)$
  - Thus **Vanilla GANs** are only good for Sampling/Generation.

- **Training is Hard**
  - Non-Convergence
  - Mode-Collapse

Non-convergence example

$$\min_x \max_y V(x,y)$$

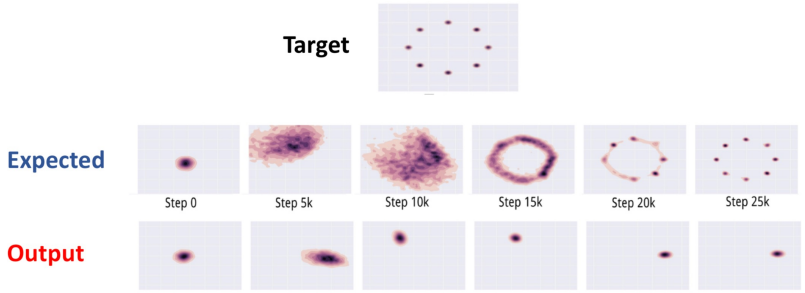
Let $V(x,y) = xy$

State 1:	x > 0	y > 0	V > 0	Increase y	Decrease x
State 2:	x < 0	y > 0	V < 0	Decrease y	Decrease x
State 3:	x < 0	y < 0	V > 0	Decrease y	Increase x
State 4 :	x > 0	y < 0	V < 0	Increase y	Increase x
State 5:	x > 0	y > 0	V > 0	Increase y	Decrease x

== State 1

Mode Collapse

Generator fails to output diverse samples



Metz, Luke, et al. "Unrolled Generative Adversarial Networks." arXiv preprint arXiv:1611.02163 (2016)

Mini-Batch GANs

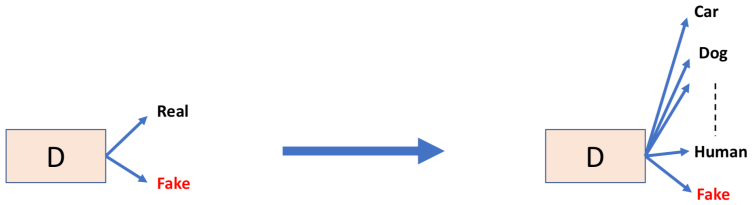
- **Extract features that capture diversity in the mini-batch**
  - For e.g. L2 norm of the difference between all pairs from the batch
  - That is, the errors at an intermediate layer for real and fake samples are minimized.
- **Feed those features to the discriminator along with the image**
- **Feature values will differ b/w diverse and non-diverse batches**
  - Thus, Discriminator will rely on those features for classification
- **This in turn,**
  - Will force the Generator to match those feature values with the real data
  - Will generate diverse batches

How to reward sample diversity

- **At Mode Collapse,**
  - Generator produces good samples, but a very few of them.
  - Thus, Discriminator can't tag them as fake.
- **To address this problem,**
  - Let the Discriminator know about this edge-case.
- **More formally,**
  - Let the Discriminator look at the entire batch instead of single examples
  - If there is lack of diversity, it will mark the examples as fake
- **Thus,**
  - Generator will be forced to produce diverse samples.

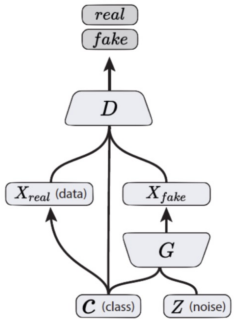
Supervision with Labels

- Label information of the real data might help



- Empirically generates much better samples

Conditional GANs

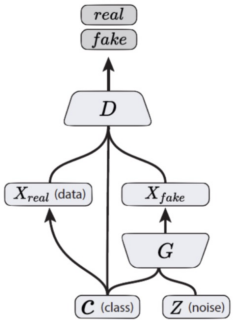


Conditional GAN
(Mirza & Osindero, 2014)

Conditions the inputs to the generator and discriminator with the labels

Conditional GANs

- Simple modification to the original GAN framework that conditions the model on *additional information* for better multi-modal learning.
- Lends to many practical applications of GANs when we have explicit *supervision* available.



Conditional GAN
(Mirza & Osindero, 2014)

Image Credit: Figure 2 in Odena, A., Olah, C. and Shlens, J., 2016. Conditional image synthesis with auxiliary classifier GANs. *arXiv preprint arXiv:1610.09585*.

Conditional GANs

MNIST digits generated conditioned on their class label.

Figure 2 in the original paper.

Mirza, Mehdi, and Simon Osindero. Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784 (2014).