Analysis and Design of Deep Neural Networks

Ahmad Kalhor
Associate Professor
School of Electrical and Computer Engineering
University of Tehran

Spring 2022

Part1
Design of DNNs, by analyzing the well known layers, blocks, modules, and architectures

1. Structure analysis in DNNs

1.1 Layers, Blocks and Modules

- Fully Connected layers and blocks
- Convolution Layers-Blocks-Modules
- Recurrent Layers-Modules
- Attention layers-Modules
- Pooling layers
- Down-sampling/Up-sampling Layers
- Normalization Layers

1.2 Architectures

- General Topologies in DNNs
- CNNs
- GANs and VAEs
- Transformers

1.1 Layers, Blocks and Modules

1.1.1. Fully Connected layers and blocks
1.1.2 Convolution Layers-Blocks-Modules
1.1.3 Recurrent Layers –Modules
1.1.4 Attention layers-Modules
1.1.5 Normalizing/Regularizing Layers
1.1.6 Down-sampling/Up-sampling Layers

1.1.1 Fully Connected layers and their blocks

Ideal to make partitions, maps and encoded/decoded data from a set of distinct inputs.

- One FC layer
- A block of two FC layers
- A block of three FC layers
- A block of more than three FC layers

One FC layer

Definition

- $\boldsymbol{y} = \boldsymbol{f}(\mathrm{W}\boldsymbol{x} + \mathbf{b}),\ \mathrm{W} = [\boldsymbol{w}_{ji}]\ \mathrm{b} = [b_j]$
- $\boldsymbol{x} \epsilon R^n \quad \boldsymbol{y} \epsilon R^m \quad i \in \{1, \cdots, n\}\ j \in \{1, \cdots, n\}$
- Activation function:
- $f \in \{sign, \mathrm{step}, \mathrm{tanh}, sigmoid, Relu, identity \cdots\}$

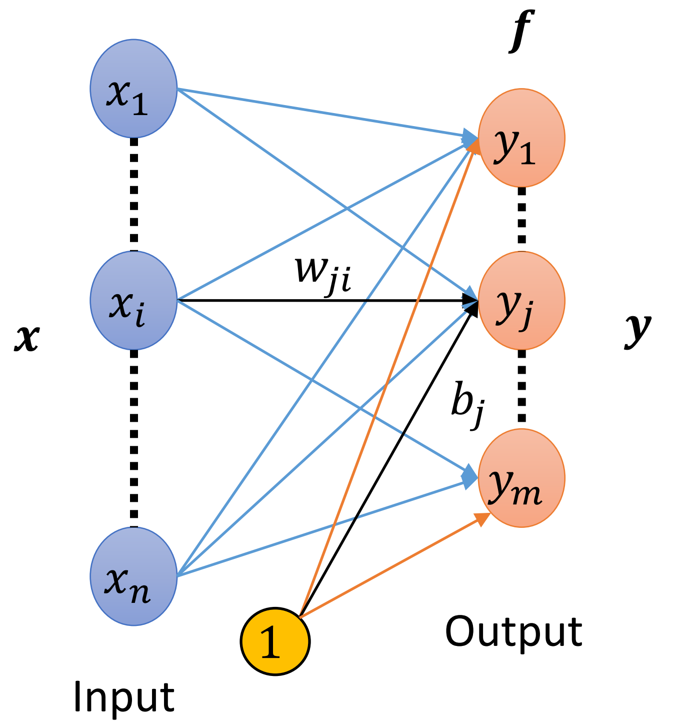
Functionality

(1) Forming an "n"dimensional hyper plane:

$$y_{in_j} = \mathrm{w_{j1}x_1} + \cdots + \mathrm{w_{jn}x_n} + \mathrm{b_j}$$

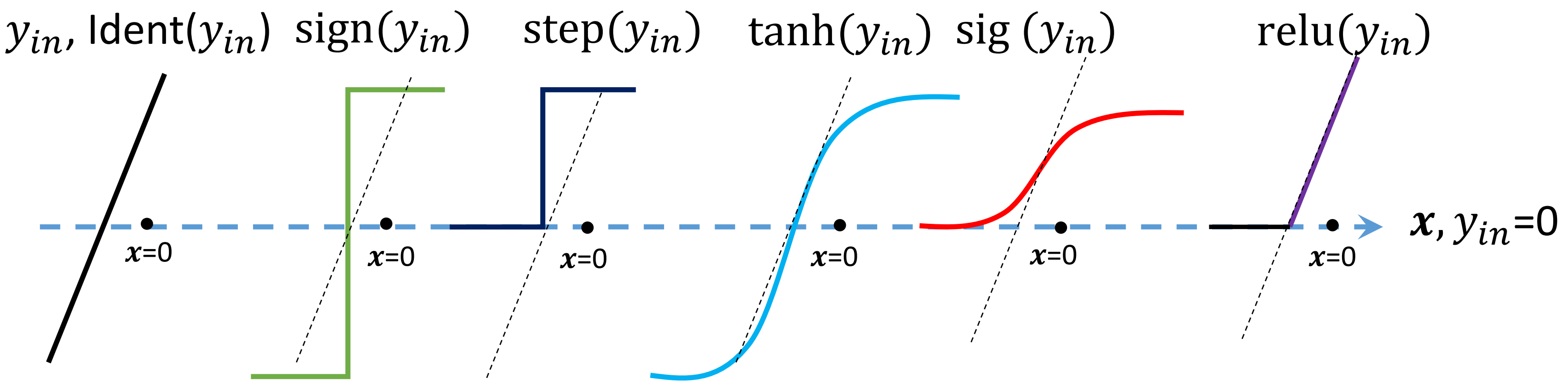
(2) Folding (hard/soft), Rectifying,.. on the hyper plane:

$$y_j = f(y_{in_j})$$

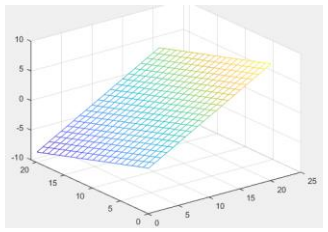


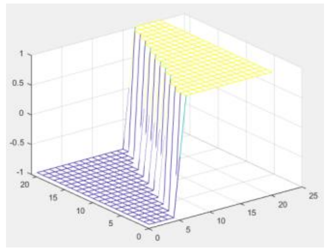
z

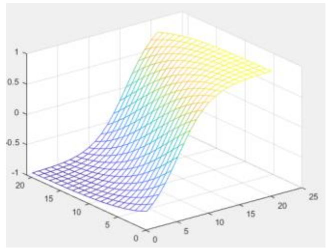
Example(1) for $n = 1$, $f(yin)$, $y_{in} = 2x + 1$ w:slope parameter , b: $(up \uparrow down \downarrow) shift\ parameter$

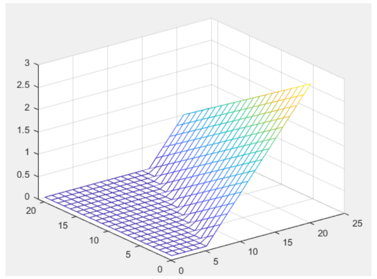


Example(2) for $n = 2$, $f(yin)$, $y_{in} = 2x_2 - x_1$+0


$y_{in}, ident(y_{in})$


sign(10y_{in})


tanh(y_{in})


Relu(y_{in})

❖ A "one FC layer" can transform the input space to a (hard or soft) folded or rectified hyper plane

Some Notes about one FC layer

1. One FC layer (with n inputs and m outputs) is formed from m neurons at output, where each neuron is connected to all n input units (fully connected).
2. Each input send a weighted signal to each neuron.
3. For each neuron, the summation of weighted inputs makes an independent hyper-plane just before activated by it.
4. The parameters of all hyper-planes are opted through a learning process.
5. For each neuron, the corresponding hyper-plane is hardly or softly folded or rectified just after activating by it.
6. Indeed, taking in inputs as a "n“ dimensional vector, the FC layer provides "m" folded, or rectified hyper-planes at the output.
7. Assuming the inputs are binary or bipolar, and the activation functions make binary or bipolar values, a neuron in one FC layer can operate as a simple logic gate like “and”, “or, etc. (M&P neuron)
8. Assuming the activation functions of the neurons are identity, a FC layer operates as a linear transformer, which can approximate a linear regression model in a regression problem.
9. Assuming “r” independent linear or nonlinear correlations among inputs, the order of the formed hyper plan is “n-r”.

A block of two FC layers

Definition

- $\boldsymbol{y} = \boldsymbol{f}^2\left(\mathrm{W}^2\,\underbrace{\boldsymbol{f}^{\mathbf{1}}(\mathrm{W}^1\boldsymbol{x} + \mathbf{b}^{\mathbf{1}})}_{z} + \boldsymbol{b}^2\right)$
- $\boldsymbol{x}\epsilon R^n \boldsymbol{z}\epsilon R^{m_1} \boldsymbol{y}\epsilon R^m \quad i \in \{1, \cdots, n\}\; j \in \{1, \cdots, m_1\}\;\; k \in \{1, \cdots, m\}$
- Activation functions:
- $f^{1,2} \in \{sign, \mathrm{step}, \mathrm{tanh}, sigmoid, Relu, identity \cdots\}$

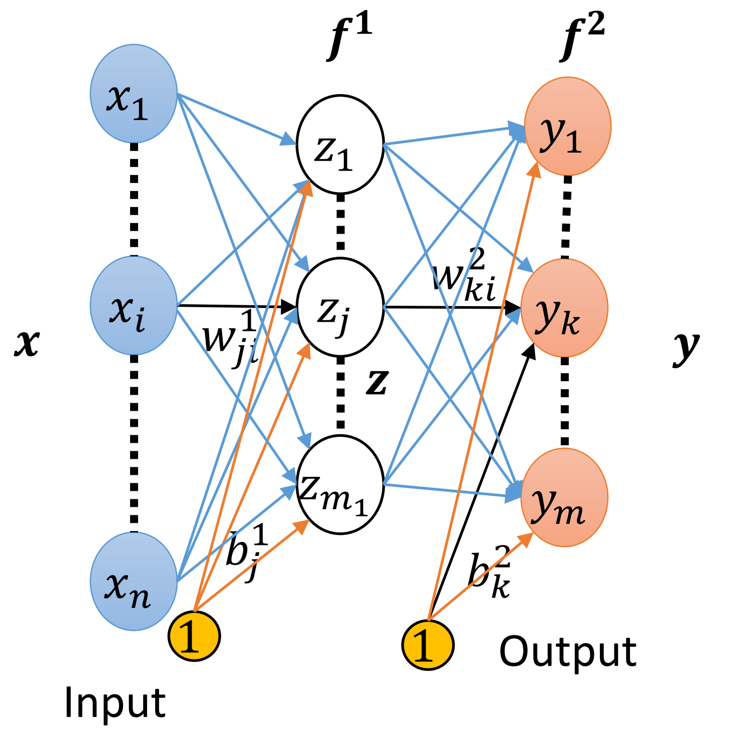
Overall Functionality

(1) A hyper plane: $z_{in_\mathrm{j}} = \mathrm{w}_{\mathrm{j}1}^1\mathrm{x}_1 + \cdots + \mathrm{w}_{\mathrm{jn}}^2\mathrm{x}_\mathrm{n} + \mathrm{b}_\mathrm{j}^1$

(2) Folding, Rectifying,.. on the hyper plane:$z_j = f^1(z_{in_\mathrm{j}})$

(3) A hyper plane: $y_{in_k} = \mathrm{w}_{\mathrm{k}1}^2\mathrm{z}_1 + \cdots + \mathrm{w}_{\mathrm{k}m_1}^2\mathrm{z}_{\mathrm{m}_1} + \mathrm{b}_\mathrm{k}^2$

(4) Folding, Rectifying,.. on the hyper plane:$y_k = f^2(y_{in_k})$



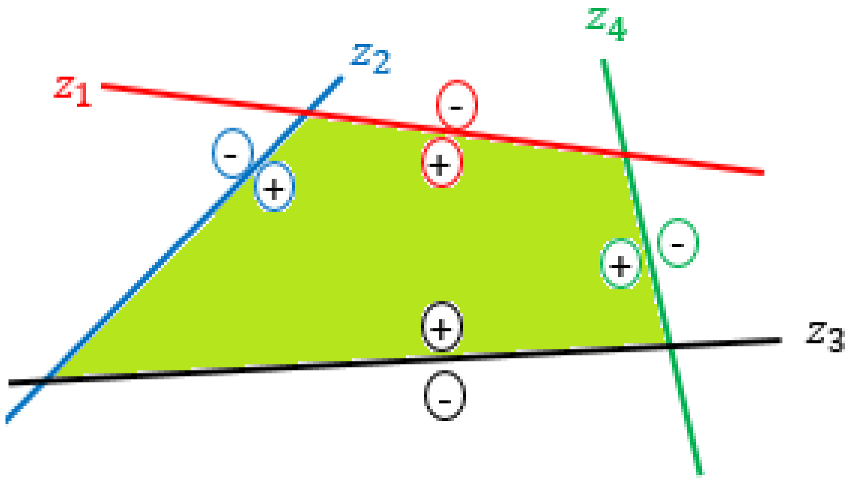
A Desired Functionality in classification /Regression problems

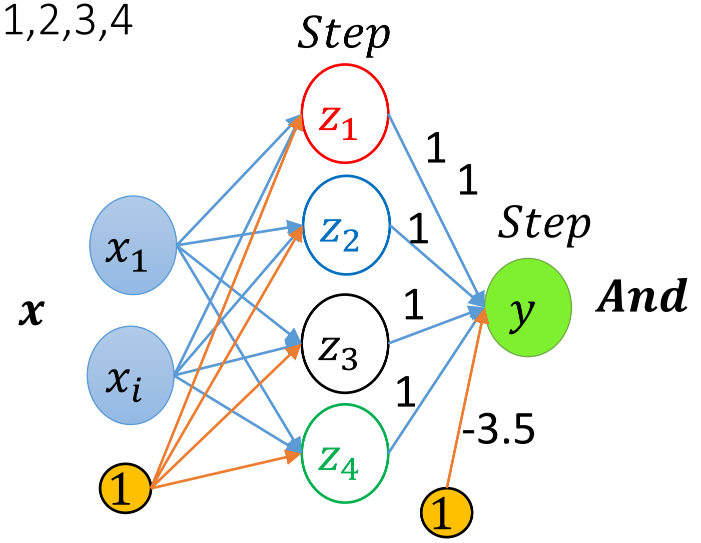
(1) A hyper plane: $z_{in_j} = \mathrm{w}_{j1}^1 \mathrm{x}_1 + \cdots + \mathrm{w}_{jn}^2 \mathrm{x}_n + \mathrm{b}_j^1$

(2) Folding the hyper plane:$z_j = f^1(z_{in_j})$

(3) An "and" or "or logic gate for former folded hyper-planes is defined by "kth" neuron of last layer, by which "kth" convex hyper-polygon will be resulted.

Example for $n = 2, m_1 = 4$, m=1 $z_j = w_{j1}^1 x_1 + w_{j2}^1 x_2 + b_j^1$,   j=1,2,3,4





- ❖ A block of two FC layers can transform the input space to multi convex hyper- polygon partitions
- ❖ Using soft activation functions, a block of two FC layers can transform the input space to multi soft convex hyper- polygon maps

A block of three FC layers

Definition

- $\boldsymbol{y} = f^3\left(\mathrm{W}^3\,\underbrace{\boldsymbol{f}^2\left(\mathrm{W}^2\,\underbrace{\boldsymbol{f}^{\mathbf{1}}(\mathrm{W}^1\boldsymbol{x} + \mathbf{b}^{\mathbf{1}})}_{z} + \boldsymbol{b}^2\right)}_{u} + \boldsymbol{b}^3)\right)$
- '$\boldsymbol{x} \epsilon R^n$ $\quad \boldsymbol{z} \epsilon R^{m_1}$ $\quad \boldsymbol{uz} \epsilon R^{m_2}$ $\quad \boldsymbol{y} \epsilon R^m$
- $i \in \{1, \cdots, n\}\ j \in \{1, \cdots, m_1\}\ k \in \{1, \cdots, m_2\}\ l \in \{1, \cdots, m\}$
- Activation functions:
- $f^{1,2,3} \in \{sign, \text{step}, \text{tanh}, sigmoid, Relu, idenity \cdots\}$

Overall Functionality

(1) A hyper plane: $z_{in_j} = \mathrm{w}_{j1}^1 \mathrm{x}_1 + \cdots + \mathrm{w}_{jn}^1 \mathrm{x}_n + \mathrm{b}_j^1$

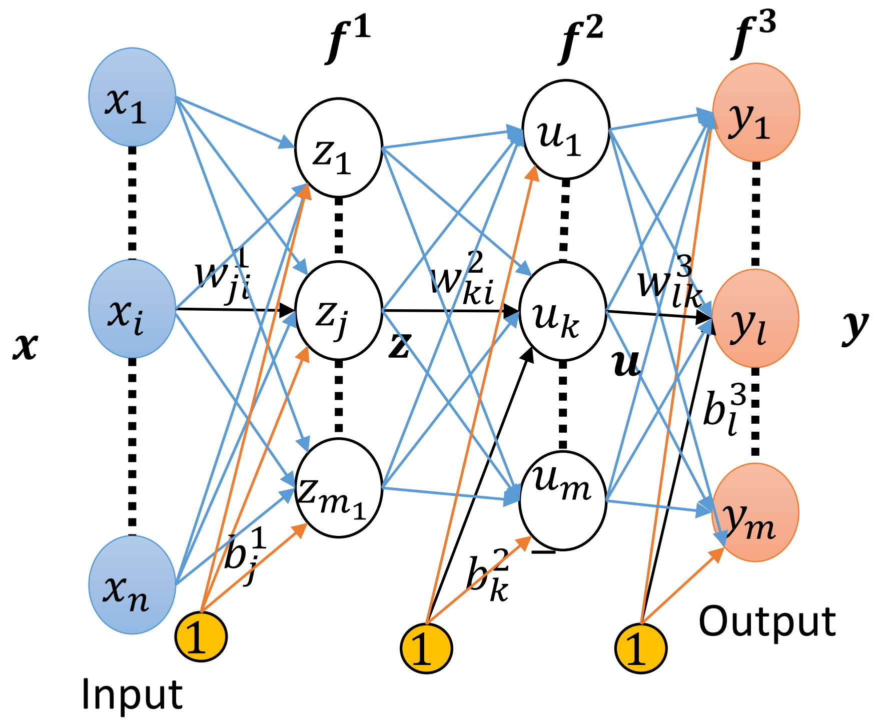
(2) Folding, Rectifying,.. on the hyper plane:$z_j = f^1(z_{in_j})$

(3) A hyper plane: $u_{in_k} = \mathrm{w}_{k1}^2 \mathrm{z}_1 + \cdots + \mathrm{w}_{km_1}^2 \mathrm{z}_{m_1} + \mathrm{b}_k^2$

(4) Folding, Rectifying,.. on the hyper plane:$u_k = f^2(y_{in_k})$

(5) A hyper plane: $y_{in_l} = \mathrm{w}_{l1}^3 \mathrm{u}_1 + \cdots + \mathrm{w}_{lm_2}^3 \mathrm{u}_2 + \mathrm{b}_l^3$

(6) Folding, Rectifying,.. on the hyper plane:$y_l = f^3(y_{in_l})$



A Desired Functionality in classification /Regression problems

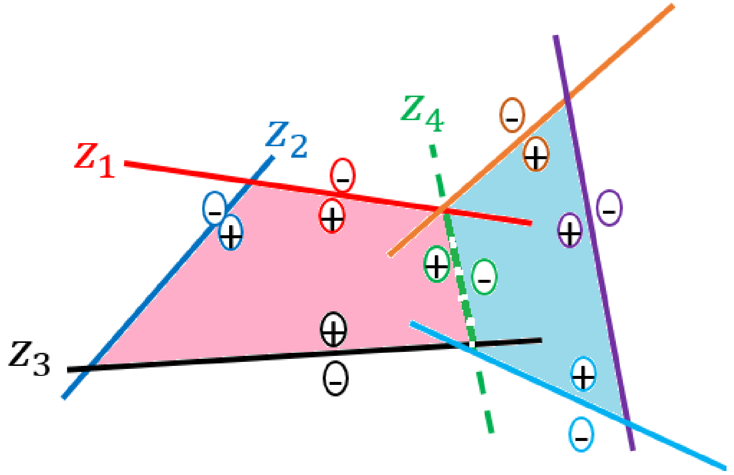
(1) A hyper plane: $z_{in_j} = w^1_{j1}x_1 + \cdots + w^1_{jn}x_n + b^1_j$

(2) Folding the hyper plane: $z_j = f^1(z_{in_j})$

(3) An "and" or "or logic gate for former folded hyper-planes is defined by "kth" neuron of second hidden layer 3, by which "kth" convex hyper-polygon will be resulted.

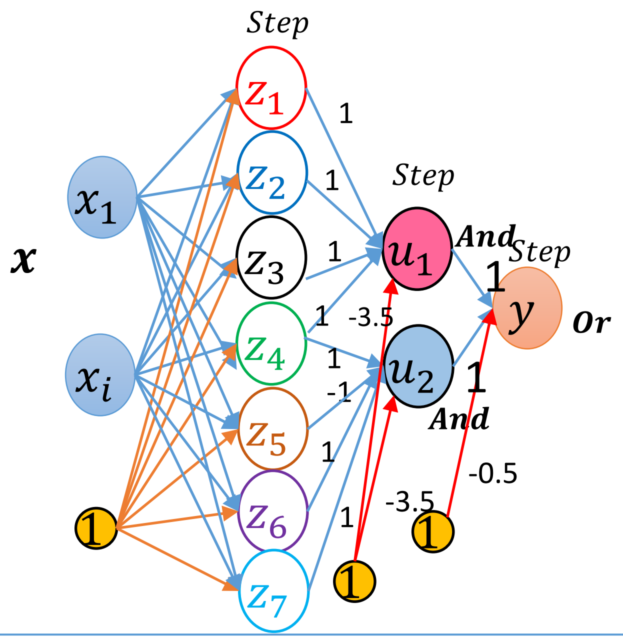
(4) An "and" or "or logic gate for former convex hyper-planes is defined by "lth" neuron of last layer, by which "lth" non-convex hyper-polygon will be resulted.

Example for $n = 2, m_1 = 7, m_2 = 2$ *m=1*



$$z_j = w^1_{j1}x_1 + w^1_{j2}x_2 + b^1_j, \quad \text{j=1,2,3,4,..,7}$$

$$u_k = w^2_{k1}z_1 + w^2_{k2}z_2 + b^2_k, \quad \text{k=1,2}$$

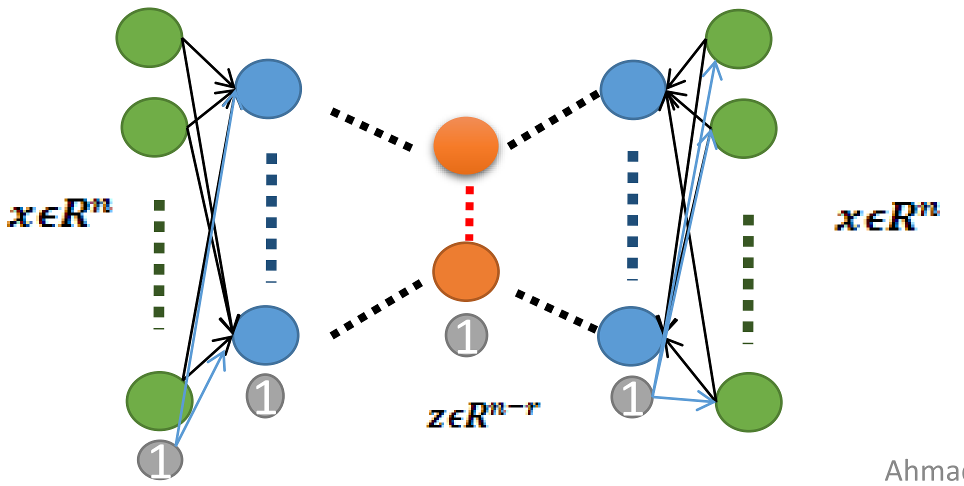


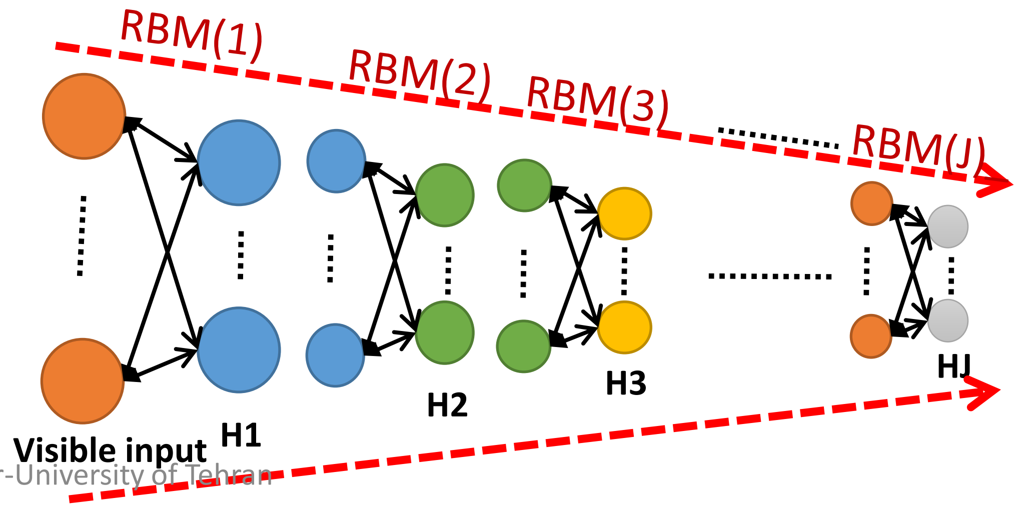
- A block of two FC layers can transform the input space to multi non-convex hyper- polygon partitions
- Using soft activation functions, a block of two FC layers can transform the input space to multi non-convex hyper- polygon volume maps

Some Notes about block of FC layers

1. A block of two or three FC layers are known as universal function approximator, through which any function by any desired accuracy can be approximated.
2. In comparison to a block of two FC layers, a block of three FC layers has better extrapolation (generalization) for unbounded regions and non-convex regions.

A block of FC layers with more than three layers are conventionally used in deep auto-encoders and cascaded restricted Boltzmann machines. In such blocks, each layer learns to encode or decode the taken data with a low change.





Some notes about FC layers

1. Fully connected layers are applied to a set of inputs reshaped as a vector; the spatial or temporal correlations among the inputs are not considered in its operation.
2. For high dimensional data, due to their massive wirings, they suffer from large memories, high computation load, and overfitting.
3. Although, they are appropriate for partitioning and mapping purposes, they are not ideal to remove disturbances, and filter and extract features from temporal, spatial , and multi modal signals.

1.1.2 Convolution Layers-Blocks-Modules*

Ideal to filter and encode/decode various, spatial, temporal and multi modal signals

- Simple Convolution
- Tiled Convolution
- Dilated Convolution
- Deconvolution (Transposed convolution)
- 1x1 Convolutions
- Flattened Convolutions
- Spatial and Cross-Channel convolutions
- Depth-wise Separable Convolutions
- Residual Blocks and types
- Grouped Convolutions
- Shuffled Grouped Convolutions

* Most of exampled are from https://ikhlestov.github.io/pages/machine-learning/convolutions-types/, Illarion Khlestov.

Simple Convolution

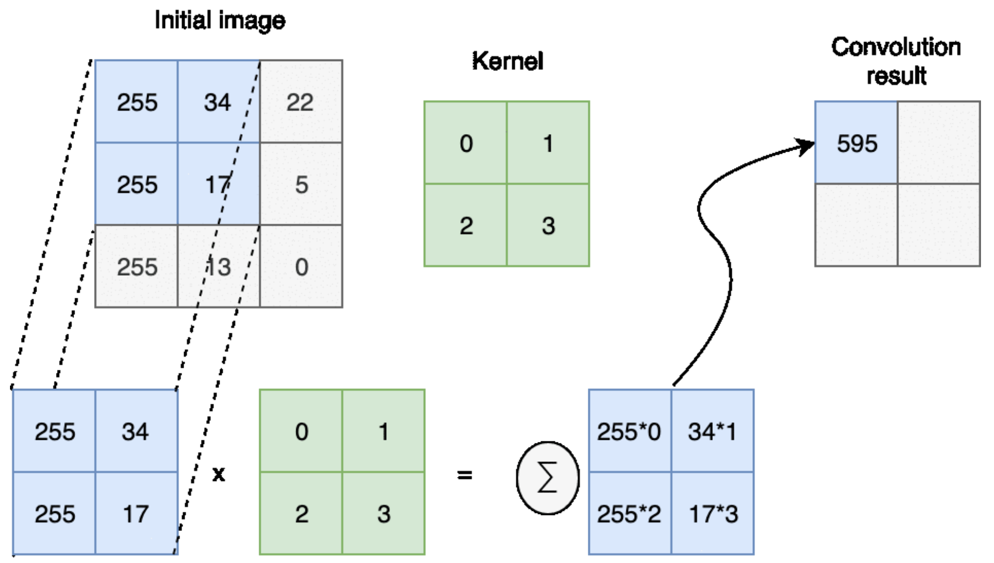
Take multiply dot products with same filter with some width/height shift.

Interesting because:

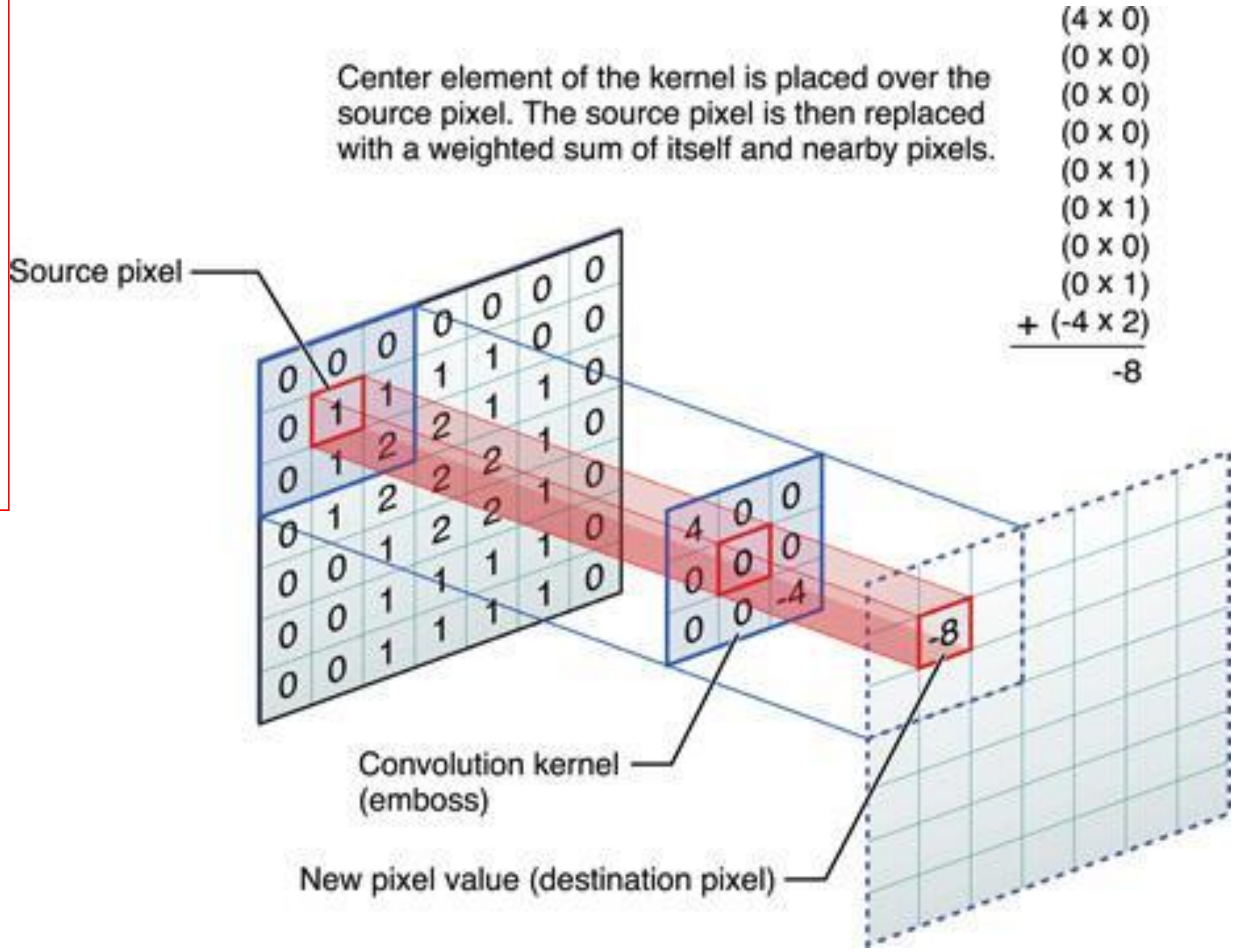
•Weights sharing and local connection

•it can capture and intensify all those patches which are adequately similar to the kernel and remove other ones by “relu”

•It can be interpreted as a filter that remove all patches which have weak linear correlation with the kernel



Example of convolution on two dimensional signal (image)

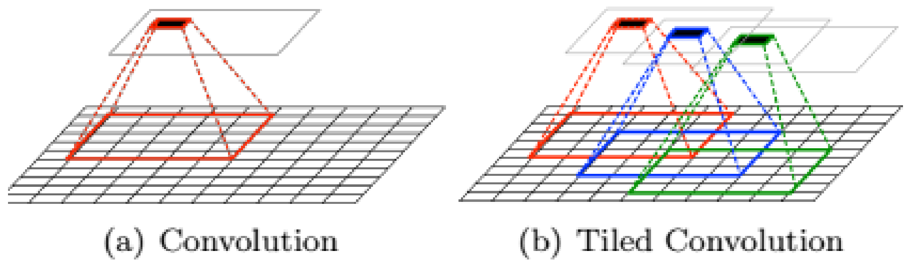

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

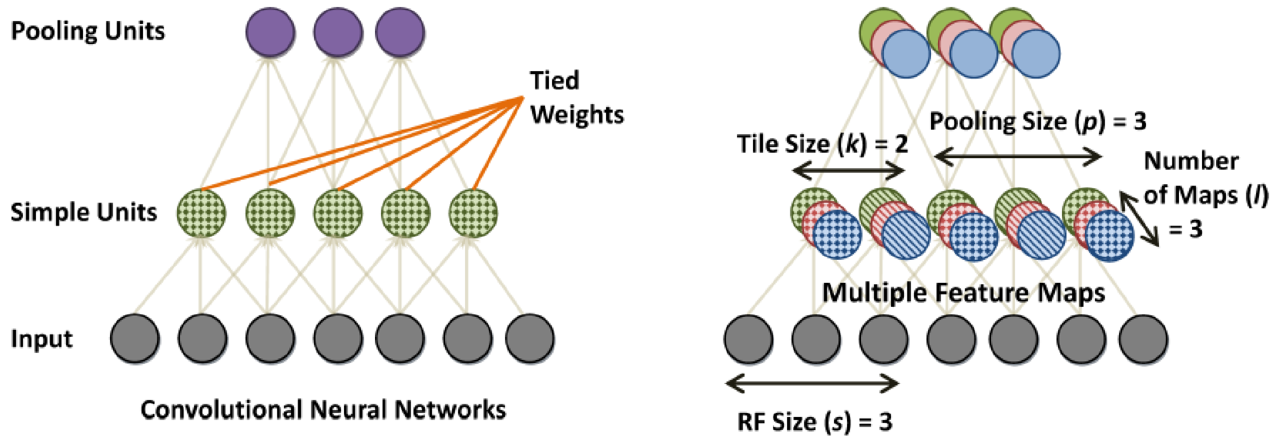
Tiled Convolution

- It means that we can learn multiples feature maps rather than one feature map by considering several different filters
- In tiled convolution we can provide different kinds of invariance.
- Separate kernels are learned within the same layer

Example of Tiled convolution on one dimensional signal(time series)

Example of Tiled convolution on two dimensional signal





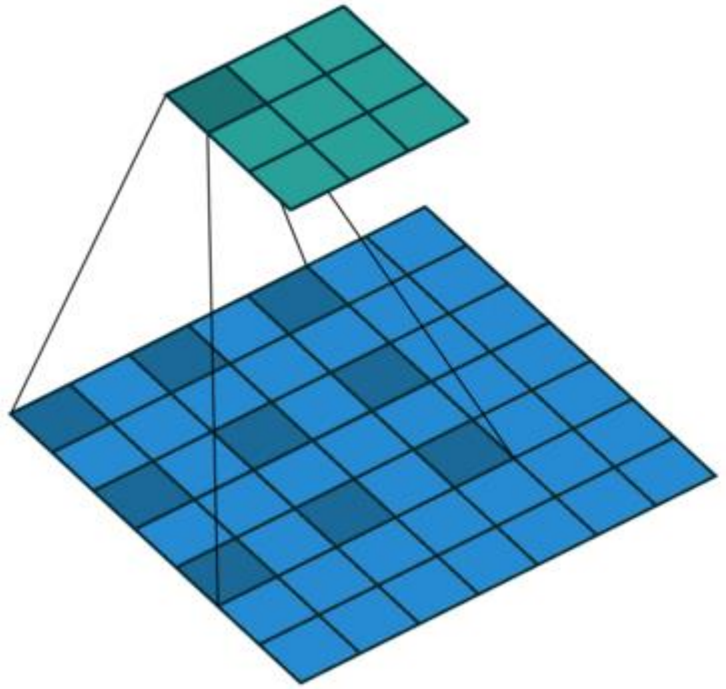
Some notes about the convolution layer

1. The convolution layer is a variant of the convolution operation used in LTI systems
2. A convolution layer actually transform an input (spatial, temporal, or multi modal) signal to several feature maps.
3. All units of feature maps just after convolution may be activated by applying an activation function like “relu”.
4. Using a filter, each feature map captures a certain feature from different local regions of the former signal.
5. Features, which are captured by filters, are actually frequently repeating sub-patterns within a signal.
6. The kernel size and the number of filters depend to the size and the number of the existing independent features, respectively.
7. The resulting feature maps from a convolution layer, can be convolved again through a new convolution layer.
8. Through using a black of sequenced convolution layers (may come with activation functions, pooling and normalizing layers), actually redundancies and disturbances are removed and exclusive features for appropriate classification or regression problem will be appeared.

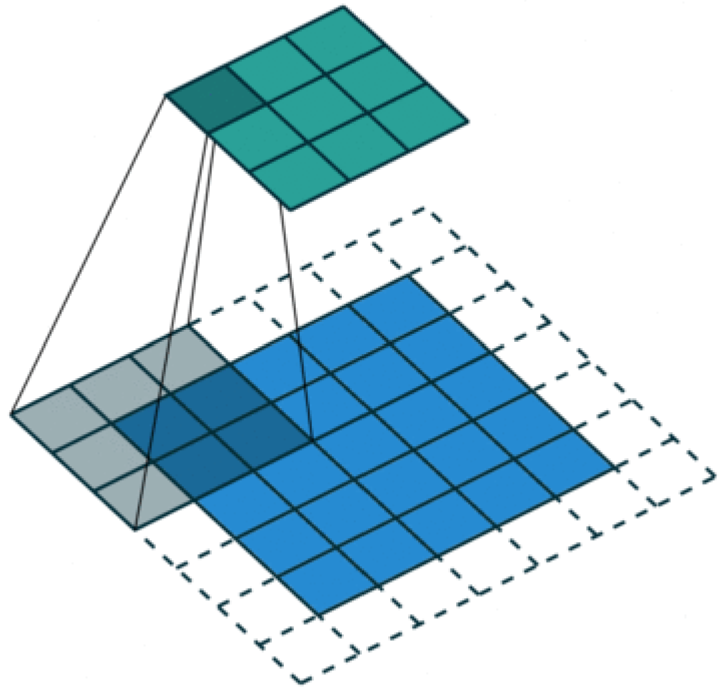
Dilated Convolution

To convolve and down-sample signals with very large spatial/temporal size

It is **a technique that expands the kernel (input) by inserting holes between the its consecutive elements**. In simpler terms, it is same as convolution but it involves pixel skipping, so as to cover a larger area of the input. ... In essence, normal convolution is just 1-dilated convolution



Dilated Convolution (l=2)

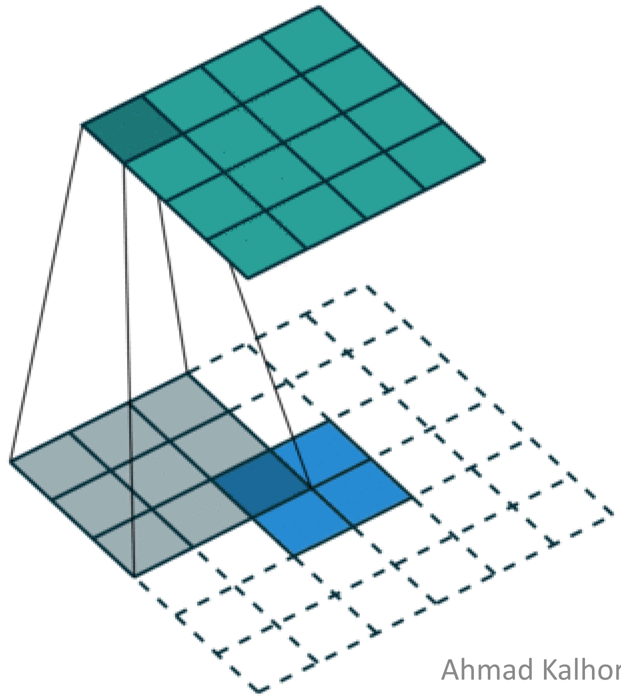


Standard Convolution (l=1)

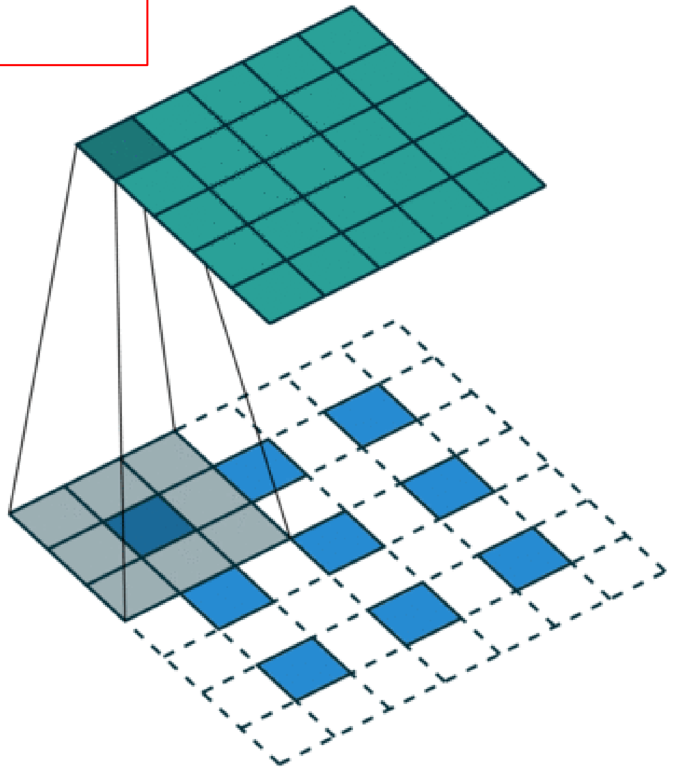
Deconvolution (Transposed Convolution)

Transposed convolution can be seen as the backward pass of a corresponding traditional convolution.
Transpose Convolution is a convolution layer which reverses the operation done by the corresponding convolution.
High pass Filter (corresponding Conv.)→ Low pass Filter (Tran. Conv.)
Smooth Filter (corresponding Conv.) → Difference Filter (Tran. Conv.)

Visually, for a transposed convolution with stride one and no padding, we just pad the original input (blue entries) with zeroes (white entries)).



In case of stride two and padding, the transposed convolution would look like this



1x1 Convolutions

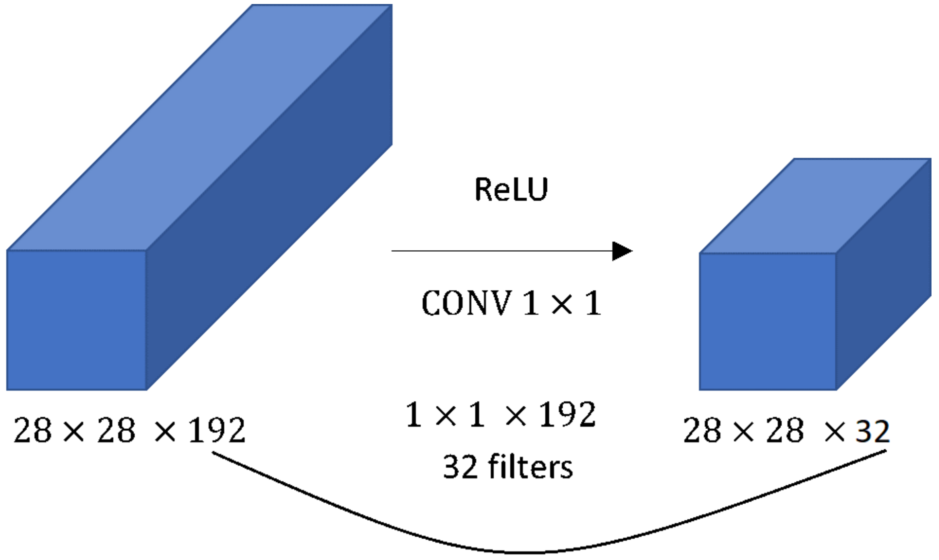
Initially 1x1 convolutions were proposed at Network-in-network(NiN). After they were highly used in GoogleNet architecture. Main features of such layers:

•Reduce or increase dimensionality

•Apply nonlinearity again after convolution

•Can be considered as “feature pooling”

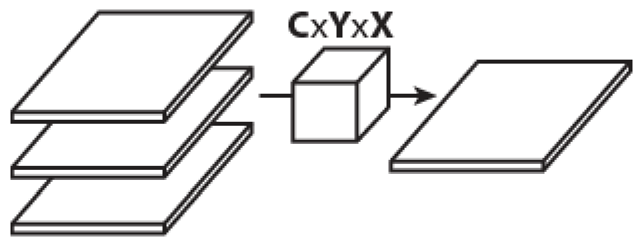
They are used in such way: we have image with size 28x28x192, where 192 means features, and after applying 32 1x1 convolutions filters we will get images with 28x28x32 dimensions.



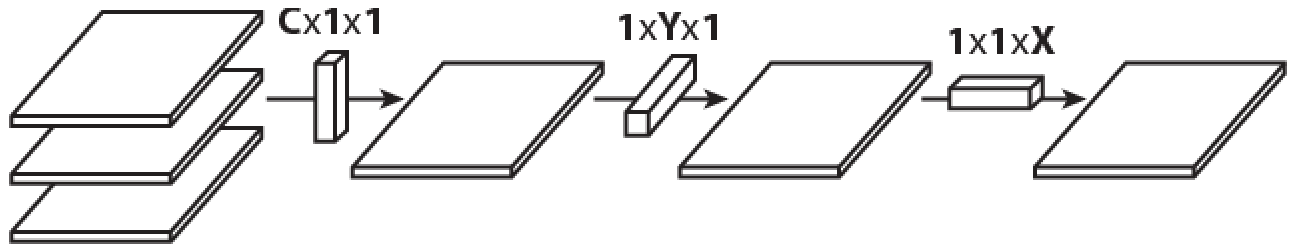
A number of filters goes from 192 to 32.

Flattened Convolutions

Were published in Flattened Convolutional Neural Networks for Feedforward Acceleration.
Reason of usage same as 1x1 convs from NiN networks, but now not only features dimension set to 1, but also one of another dimensions: width or height.



(a) 3D convolution



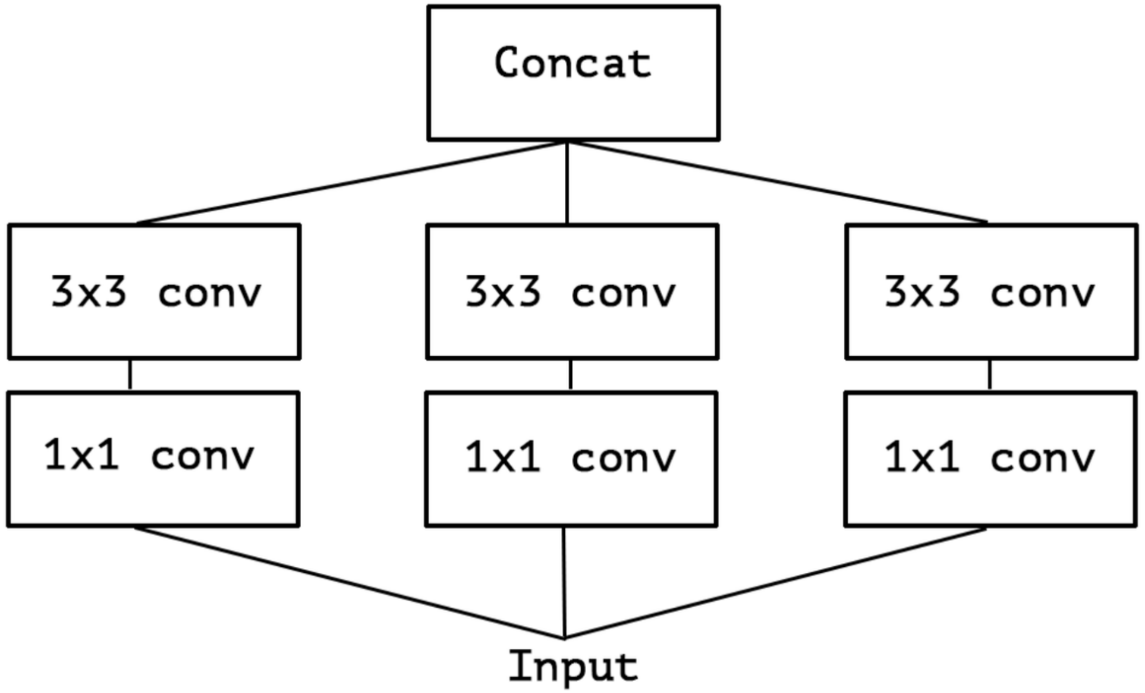
(b) 1D convolutions over different directions

The number of operations at each feature map is= a*a*(C*Y*X)

The number of operations at each feature map is= a*a*(C+Y+X)

Spatial and Cross-Channel convolutions

First this approach was widely used in **Inception network**. Main reason is to split operations for cross-channel correlations and at spatial correlations into a series of independently operations. Spatial convolutions means convolutions performed in **spatial dimensions** - **width** and **height**



Inception Module

Inception module is introduced by Szegedy *et al*. * which can be seen as a logical culmination of NIN. They use variable filter sizes to capture different visual patterns of different sizes, and approximate the optimal sparse structure by the inception modul

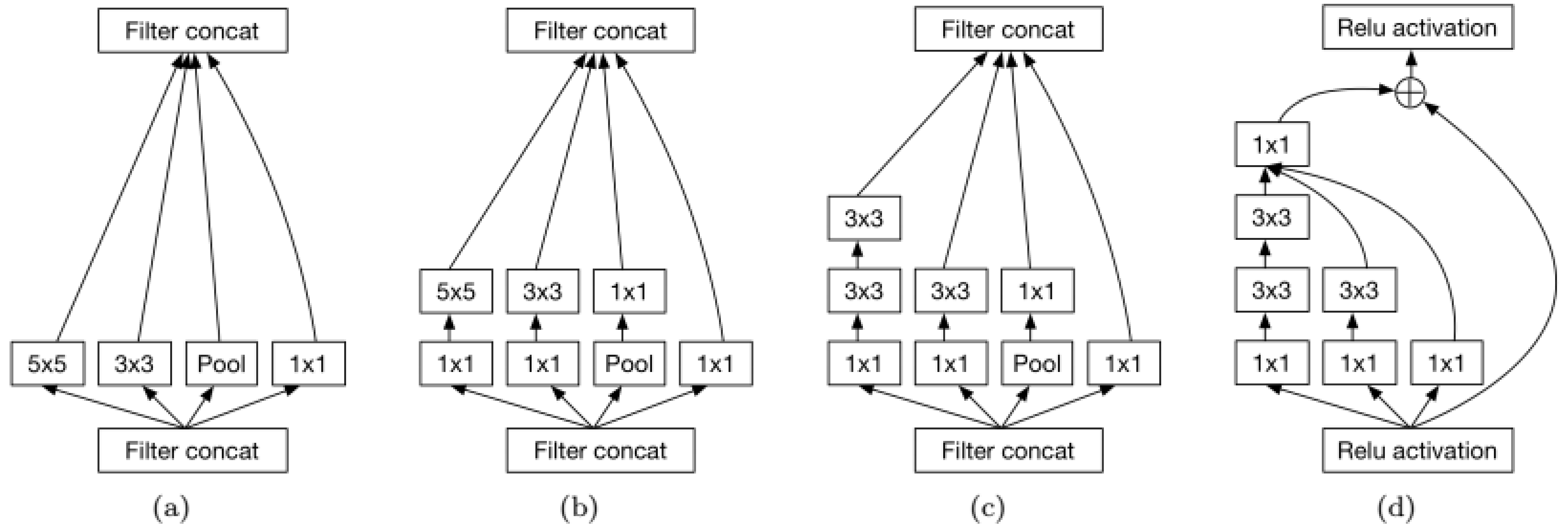
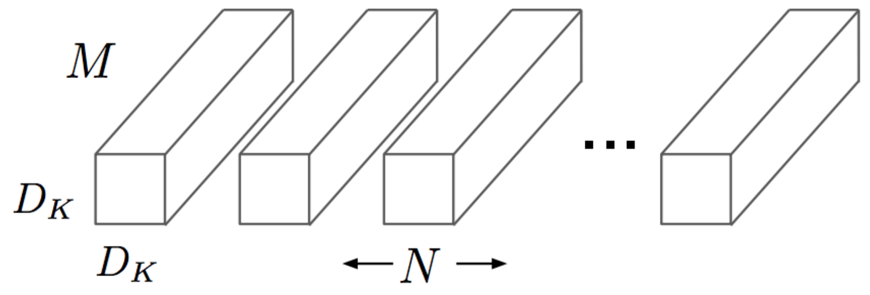


Figure 5: (a) Inception module, naive version. (b) The inception module used in [10]. (c) The improved inception module used in [41] where each 5 × 5 convolution is replaced by two 3 × 3 convolutions. (d) The Inception-ResNet-A module used in [42].

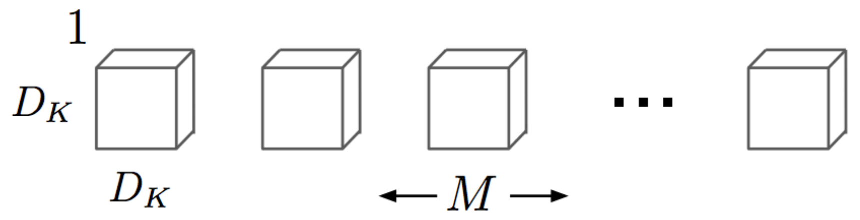
* C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015,
pp. 1–9.

Depthwise Separable Convolutions

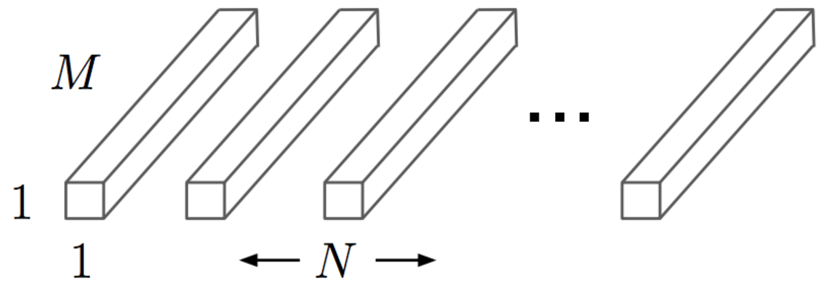
A lot about such convolutions published in the (Xception paper) or (MobileNet paper).
Consist of:
•**Depthwise convolution**, i.e. a spatial convolution performed independently over each channel of an input.
•**Pointwise convolution**, i.e. a 1x1 convolution, projecting the channels output by the depthwise convolution onto a new channel space.
Difference between Inception module and separable convolutions:
•Separable convolutions perform first channel-wise spatial convolution and then perform 1x1 convolution, whereas Inception performs the 1x1 convolution first.
•depthwise separable convolutions are usually implemented without non-linearities



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

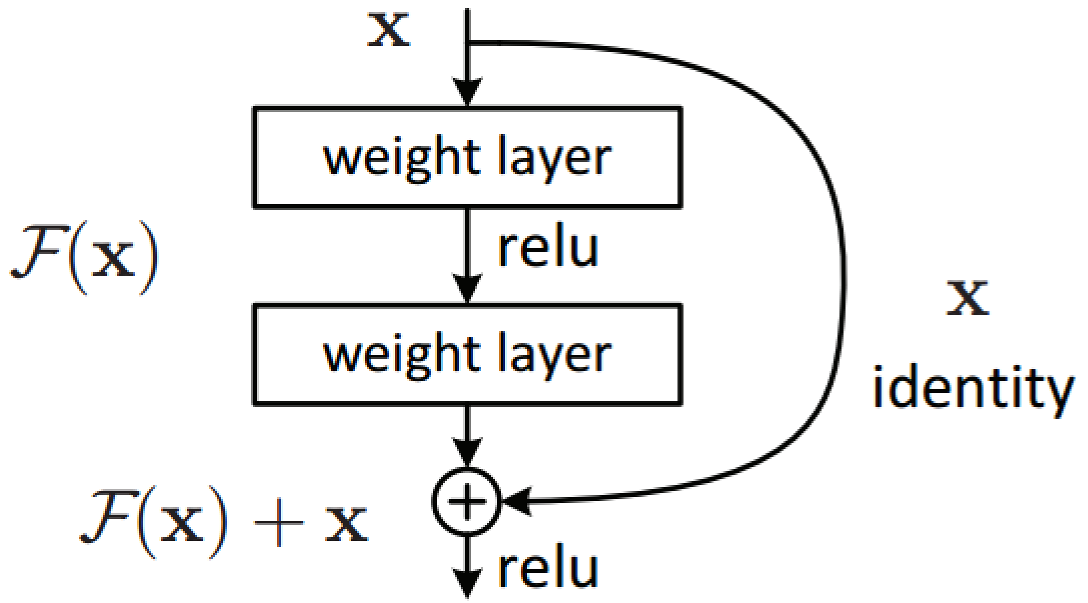


(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Residual Blocks

To avoid missing information and provide a solution for vanishing gradient

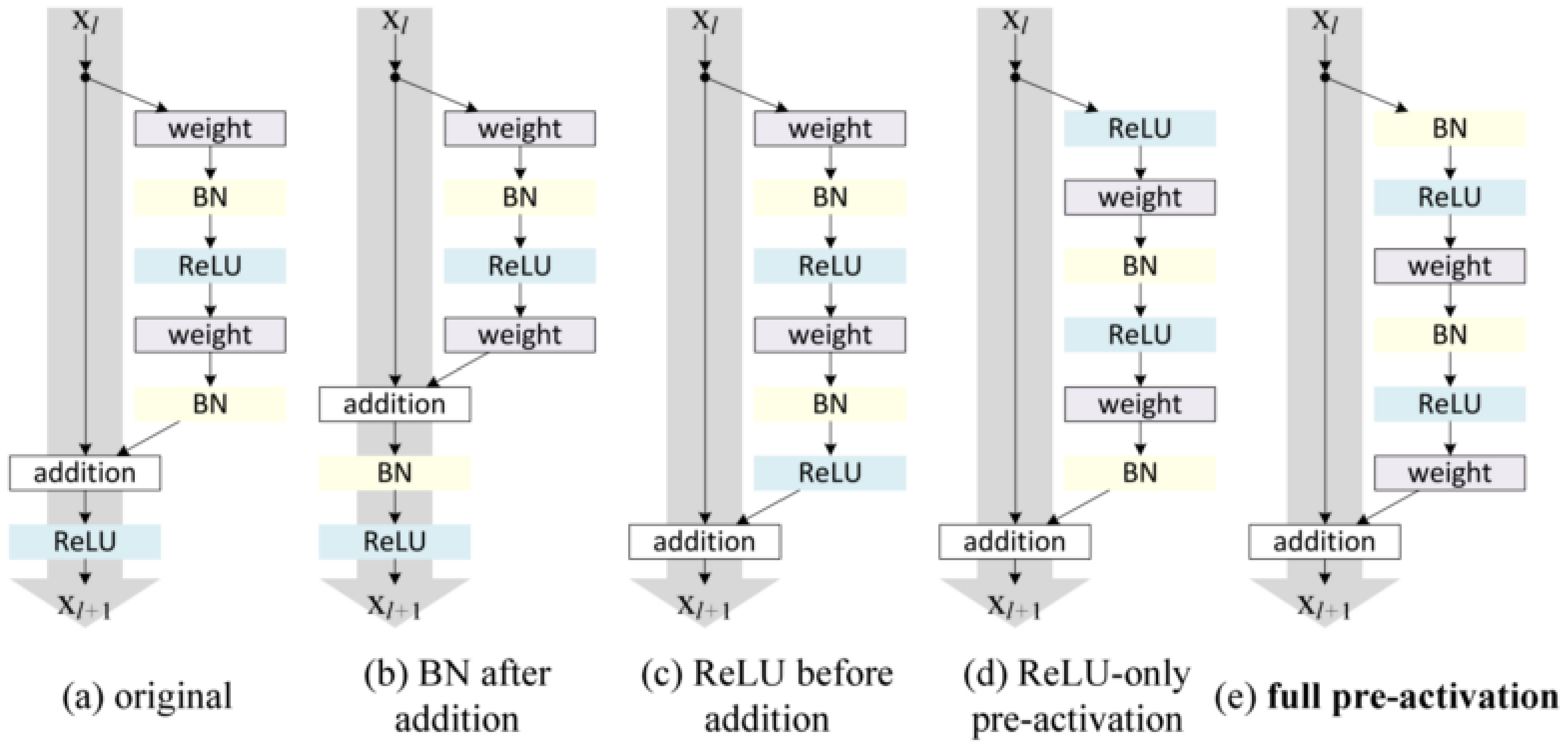
In traditional neural networks, each layer feeds into the next layer. In a network with residual blocks, each layer feeds into the next layer and directly into the layers about 2–3 hops away. That’s it. But understanding the intuition behind why it was required in the first place, why it is so important, and how similar it looks to some other state-of-the-art architectures is where we are going to focus on. There is more than one interpretation of why residual blocks are awesome and how & why they are one of the key ideas that can make a neural network show state-of-the-art performances on a wide range of tasks.



Applying an identity path to convolution, residual block causes that features to be captured in a differential learning manner without missing information.
It allows the CNN to achieve more effective features with using more depth in its architecture.

Different forms of residual blocks

The image below shows how to arrange the residual block and identity connections for the optimal gradient flow. It has been observed that pre-activations with batch normalizations generally give the best results (i.e., the right-most residual block in the image below gives the most promising results).



Grouped Convolutions

Grouped convolutions were initial mentioned in AlexNet, and later reused in ResNeXt. Main motivation of such convolutions is to reduce computational complexity while dividing features on groups.

The image below shows multiple interpretations of a residual block.

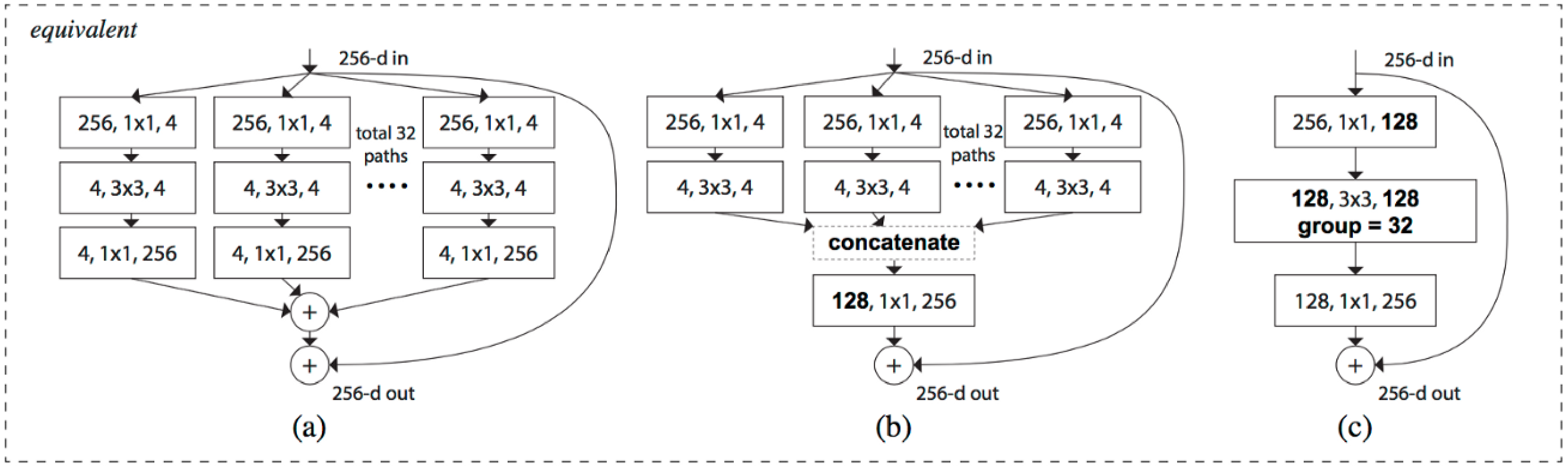


Figure 3. Equivalent building blocks of ResNeXt. **(a)**: Aggregated residual transformations, the same as Fig. 1 right. **(b)**: A block equivalent to (a), implemented as early concatenation. **(c)**: A block equivalent to (a,b), implemented as grouped convolutions [24]. Notations in **bold** text highlight the reformulation changes. A layer is denoted as (# input channels, filter size, # output channels).

Shuffled Grouped Convolutions

Shuffle Net proposed how to eliminate main side effect of the grouped convolutions that “outputs from a certain channel are only derived from a small fraction of input channels”.
They proposed shuffle channels in such way(layer with gg groups whose output has g×ng×n channels):
•reshape the output channel dimension into (g,n)(g,n)
•transpose output
•flatten output bac

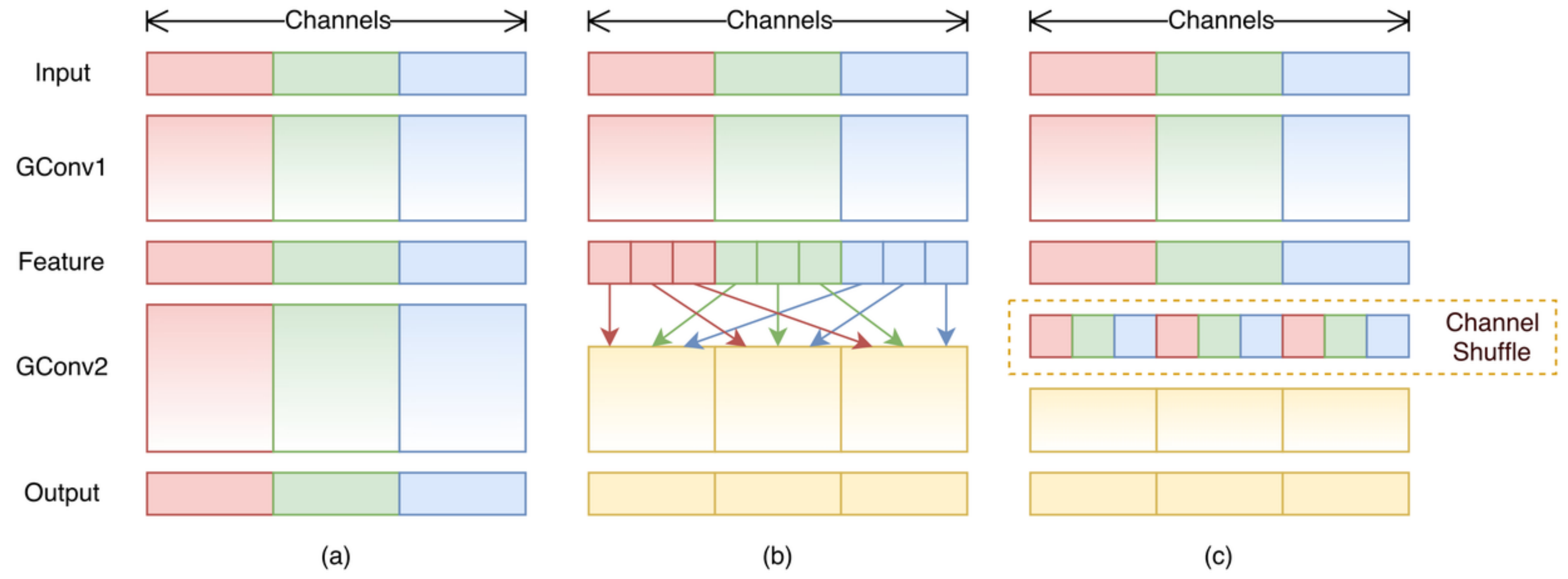


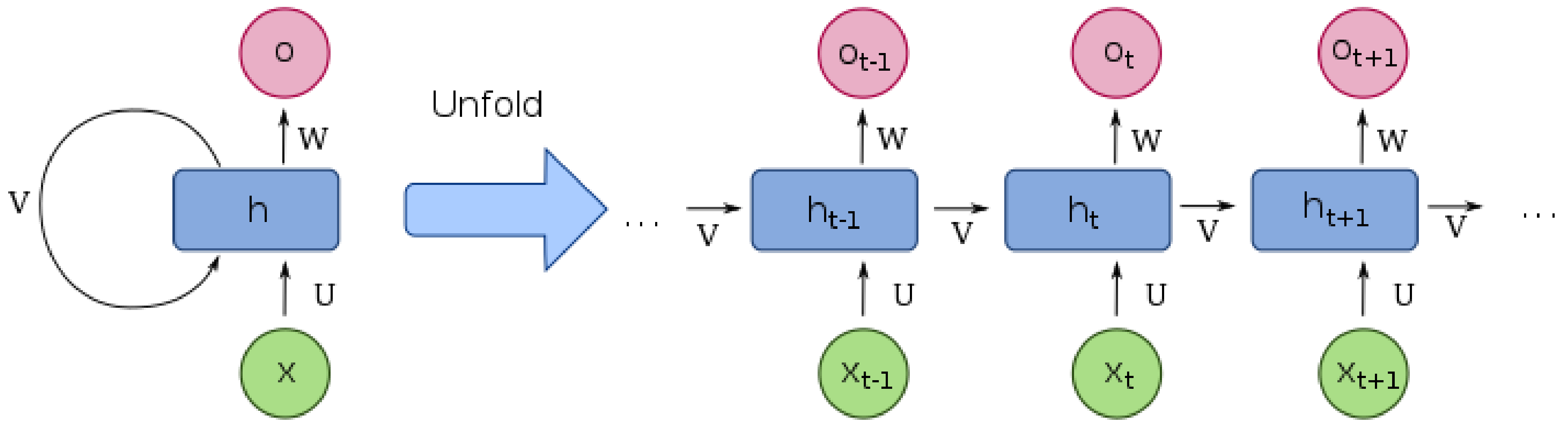
Figure 1: Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

1.1.3 Recurrent Neural Networks

Ideal to retrieve short/long dependencies among the sequenced samples of a signals

- Simple RNN Module
- LSTM Module
- GRU Module
- Bidirectional RNN layer
- Bidirectional Deep RNNs
- ConvLSTM layer
- TimeDistributed layer

Simple RNN Module



$$\boldsymbol{h}_t = \mathbf{f}(\mathrm{V}\boldsymbol{h}_{t-1} + \mathrm{U}\boldsymbol{x}_t)$$

$$\boldsymbol{o}_t = \mathbf{g}(\mathrm{W}\boldsymbol{h}_t)$$

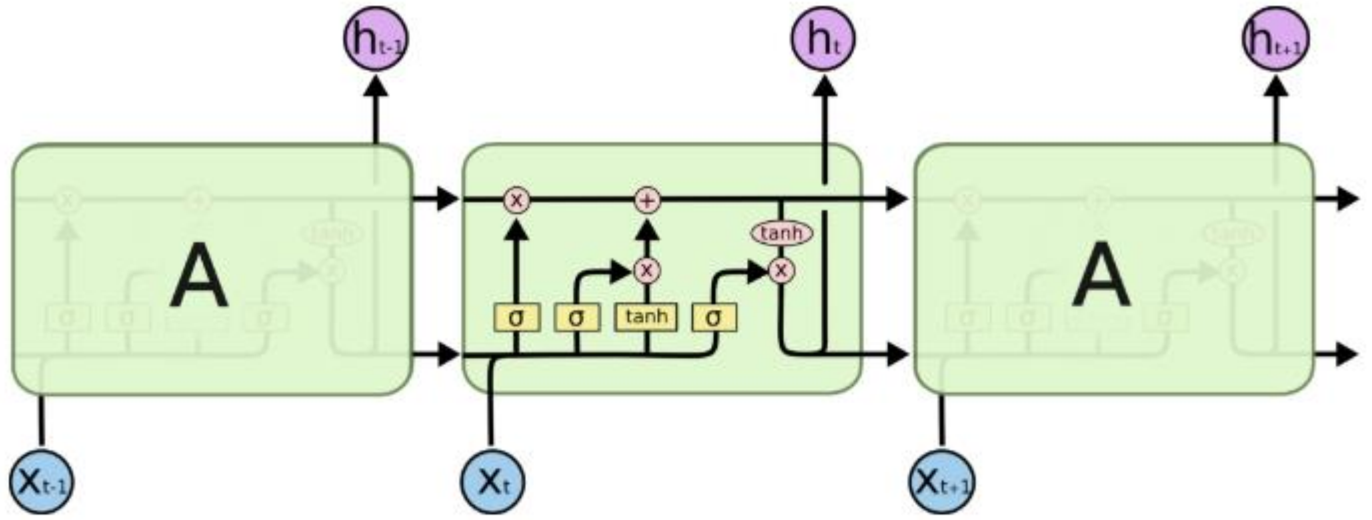
$$\boldsymbol{o}_t = \mathbf{g}\Big(\mathrm{W}\mathbf{f}(\mathrm{V}\mathbf{f}(\mathrm{V} \dots (\mathrm{V}\mathbf{f}(\mathrm{V}\boldsymbol{h}_0 + \mathrm{U}\boldsymbol{x}_1) \dots + \mathrm{U}\boldsymbol{x}_{t-1})) + \mathrm{U}\boldsymbol{x}_t)\Big), t = 1,2,\cdots$$

Some notes

1. RNN is equal to a set of nonlinear first order difference equations.
2. The current state $\boldsymbol{h}_t$ is affected by both exogenous input $\boldsymbol{x}_t$ and the former state $\boldsymbol{h}_{t-1}$.
3. The output $\boldsymbol{o}_t$ is a map of hidden state $\boldsymbol{h}_t$ at each time t .
4. Such equations can provide a memory from the short dependencies in a sequenced patterns
5. $Activation\ functions: f, g \epsilon \{Relu, tanh, sigm, sign, identity \dots.\}$

LSTM (long short term memory)

Hochreitor & Shmidhuber 1997



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \;+\; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \;+\; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \;+\; b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \;+\; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

$$\hat{y}_t = f(V.h_t + b_v)$$

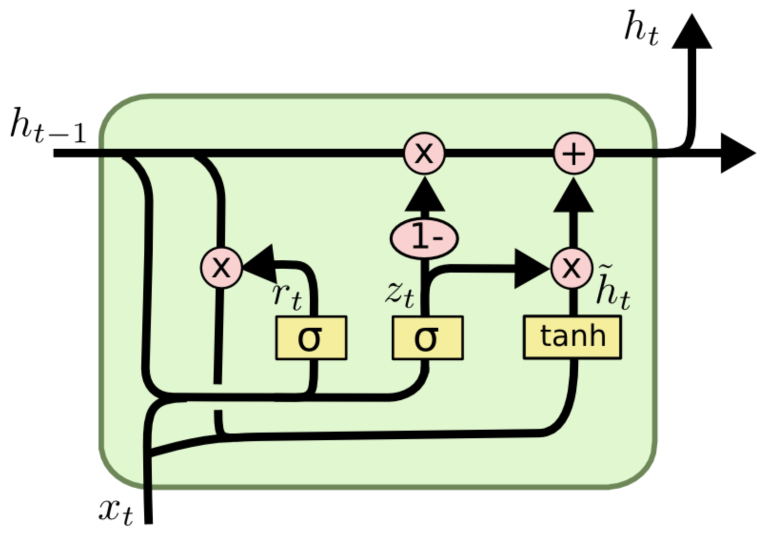
Some notes:

1. LSTM is a module of $four\ interacted\ layers$.
2. A cell state is the main concept in the LSTM providing all informative data required to save long/short dependencies among the sequenced patterns
3. Using exogenous input and the hidden state, cell state interacts by three independent gates.
4. Some information on the cell state is forgotten by the forget gate.
5. Using exogenous input and the hidden state, a package of informative data is added to the cell state by write gate.
6. The hidden state is provided from the cell state by read gate.
7. The output is a function of the hidden state.

GRU

Cho, et al. (2014)

A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU. It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



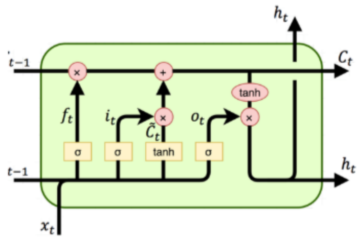
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

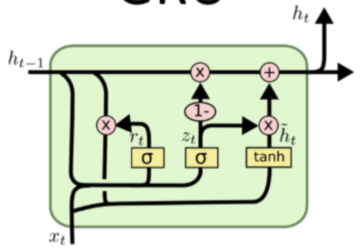
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTM

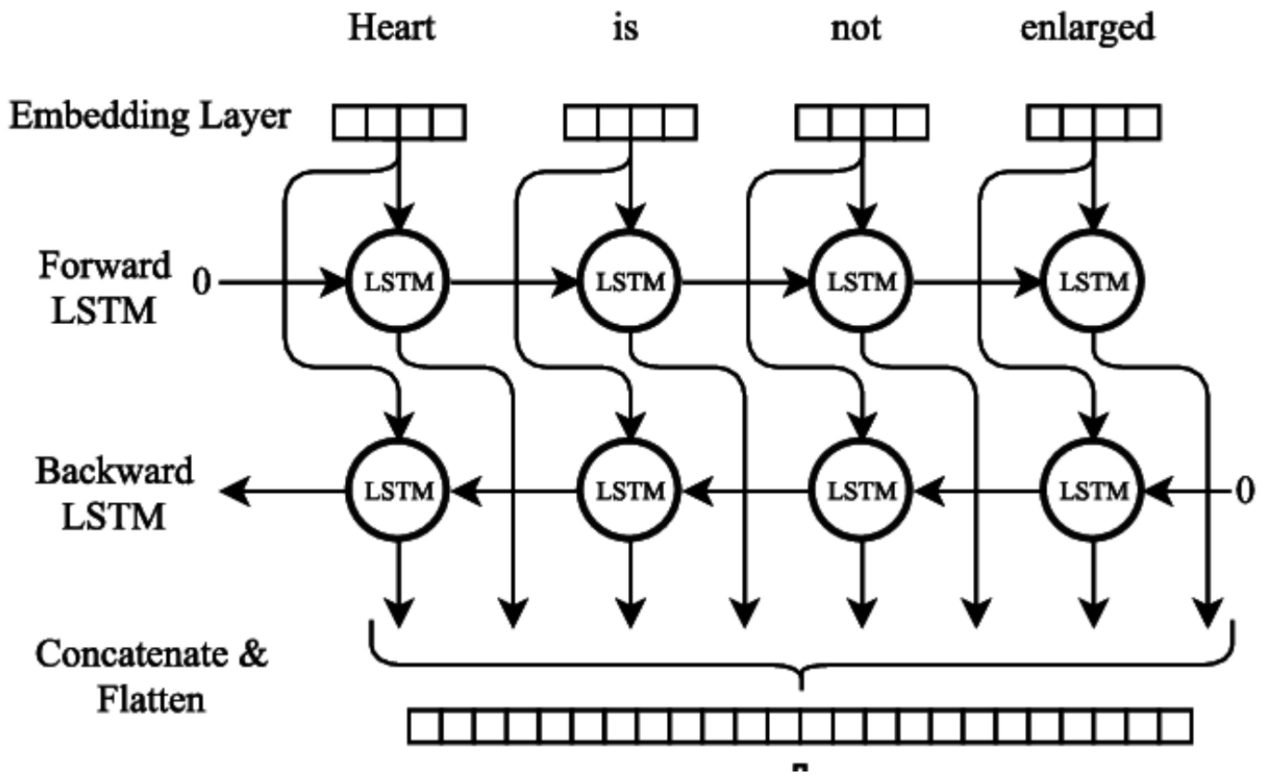


GRU



Bidirectional RNNs

A **Bidirectional LSTM**, or **biLSTM**, is a sequence processing model that consists of two LSTMs: one taking the input in a forward direction, and the other in a backwards direction. BiLSTMs effectively increase the amount of information available to the network, improving the context available to the algorithm (e.g. knowing what words immediately follow *and* precede a word in a sentence).
Image Source: Modelling Radiological Language with Bidirectional Long Short-Term Memory Networks, Cornegruta et al

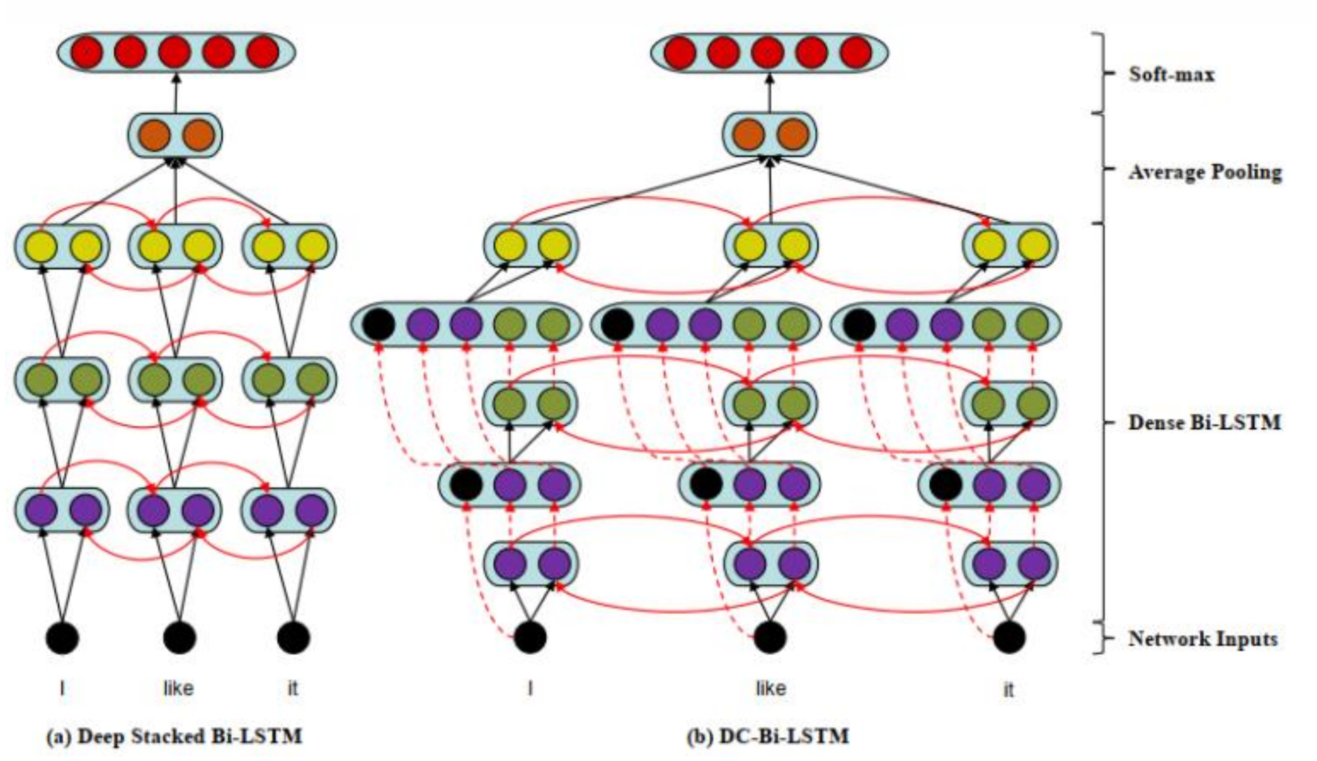


It can be used in categorization
Missed words prediction and so on.

Deep (Bidirectional) RNNs

A block of one or more than one LSTMs or Bi-LSTMs

- **Deep (Bidirectional) RNNs** are similar to Bidirectional RNNs, only that we now have multiple layers per time step. In practice this gives us a higher learning capacity (but we also need a lot of training data)



Deep Stacked Bi-LSTM

Densely Connected Bidirectional

ConvLSTM

Xingjian Shi Zhourong, et al (2015)

- In our case, sequencial images, one approach is using **ConvLSTM** layers. It is a Recurrent layer, just like the LSTM, but internal matrix multiplications are exchanged with convolution operations. As a result, the data that flows through the ConvLSTM cells keeps the input dimension (3D in our case) instead of being just a 1D vector with features.

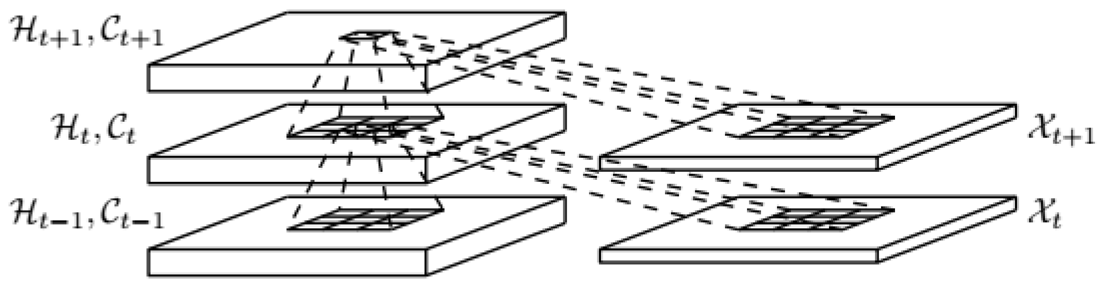
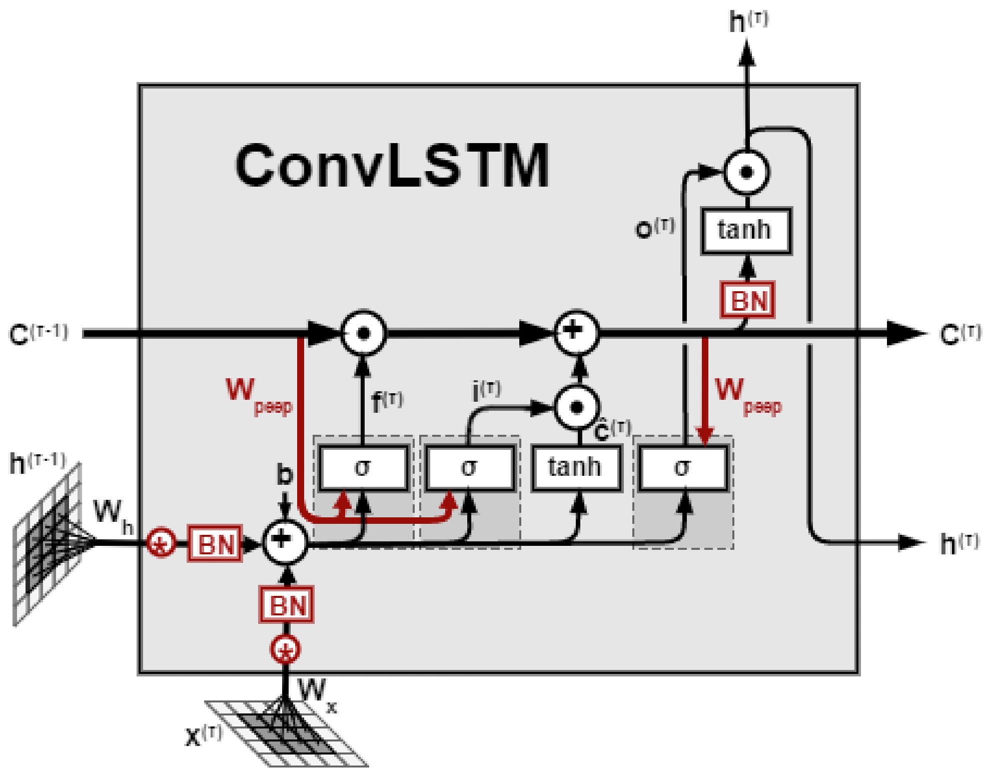

$\mathcal{H}_{t+1}, \mathcal{C}_{t+1}$
$\mathcal{H}_t, \mathcal{C}_t$
$\mathcal{H}_{t-1}, \mathcal{C}_{t-1}$
$\mathcal{X}_{t+1}$
$\mathcal{X}_t$

Figure 2: Inner structure of ConvLSTM

$$
\begin{aligned}
i_t &= \sigma(W_{xi} * \mathcal{X}_t + W_{hi} * \mathcal{H}_{t-1} + W_{ci} \circ \mathcal{C}_{t-1} + b_i) \\
f_t &= \sigma(W_{xf} * \mathcal{X}_t + W_{hf} * \mathcal{H}_{t-1} + W_{cf} \circ \mathcal{C}_{t-1} + b_f) \\
\mathcal{C}_t &= f_t \circ \mathcal{C}_{t-1} + i_t \circ \tanh(W_{xc} * \mathcal{X}_t + W_{hc} * \mathcal{H}_{t-1} + b_c) \\
o_t &= \sigma(W_{xo} * \mathcal{X}_t + W_{ho} * \mathcal{H}_{t-1} + W_{co} \circ \mathcal{C}_t + b_o) \\
\mathcal{H}_t &= o_t \circ \tanh(\mathcal{C}_t)
\end{aligned}
$$

where '$*$' denotes the convolution operator and '$\circ$', as before, denotes the Hadamard product

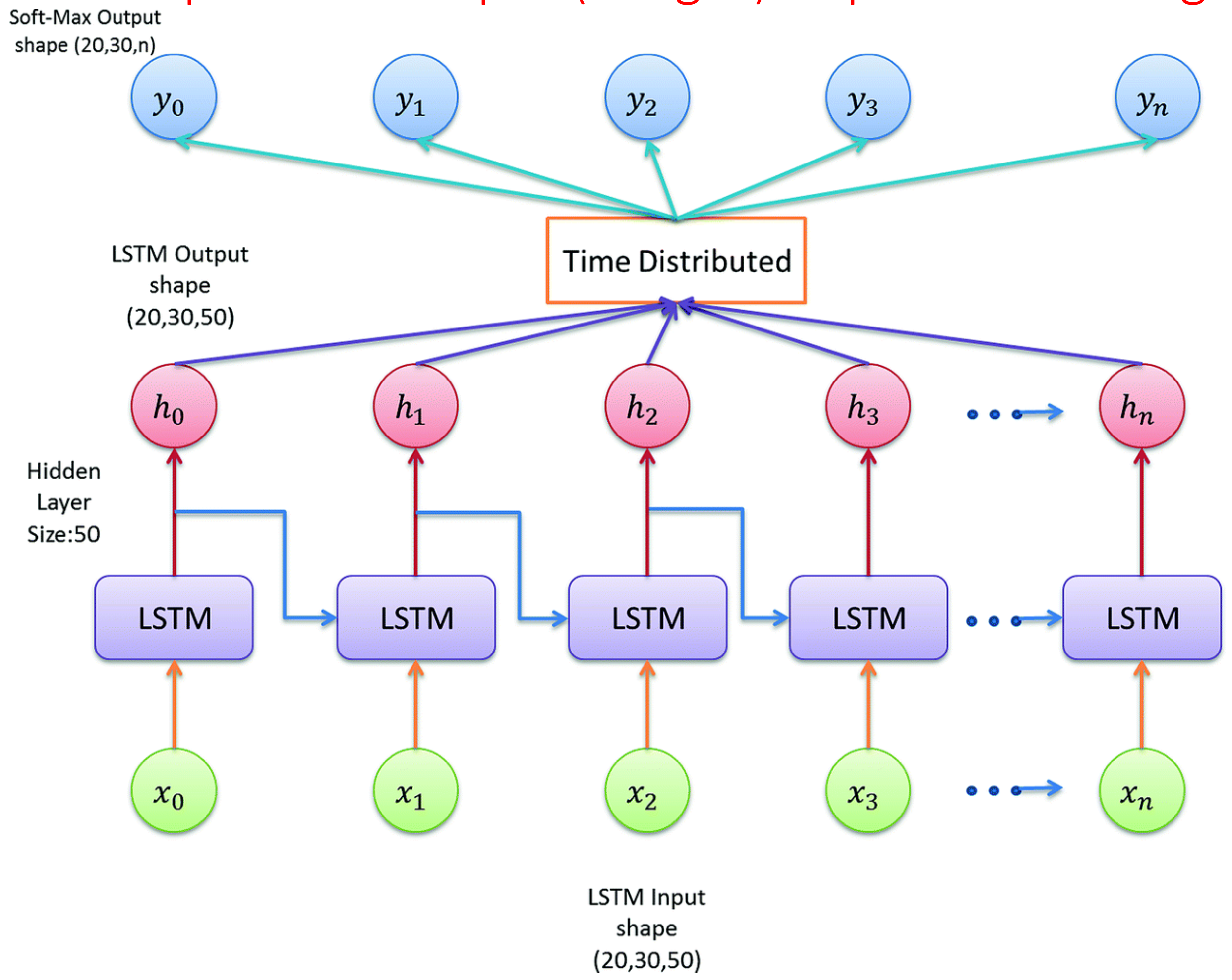


ConvLSTM 2D,3D

Time distributed Layer

to reduce the required memory in processing multi sequenced samples(images) to predict the target

Time Distributed layer is very useful to work with time series data or video frames. It allows **to use a layer for each input**. That means that instead of having several input "models", we can use "one model" applied to each input. Then GRU or LSTM can help to manage the data in "time".
This function adds an independent layer for each time step in the recurrent model. So, for instance, if we have 10 time steps in a model, a Time Distributed layer operating on a Dense layer would produce 10 independent Dense layers, one for each time step. The activation for these dense layers is set to be softmax in the final layer of our Keras LSTM model.



Time step: **the number of past samples of a sequence which are used in a LSTM to predict the targets.**

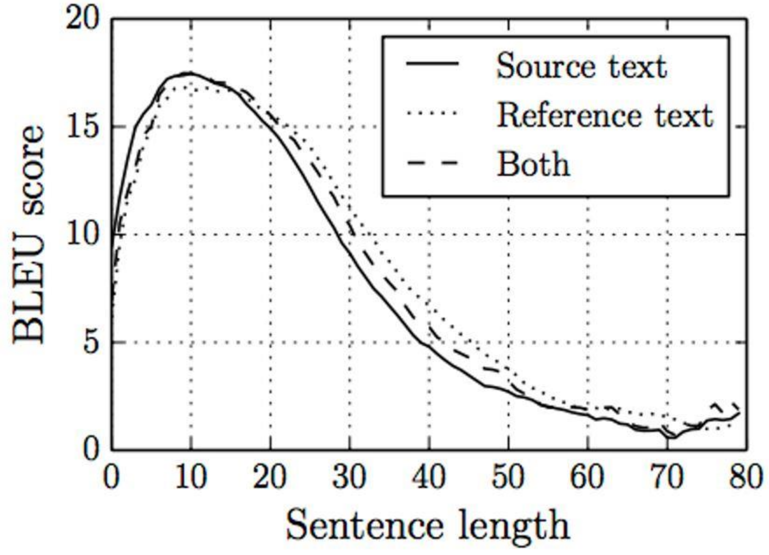
Attention Layers

- In translation machines, LSTM/GRU as a seq2seq model can not predict the translation of a sentences in sense of BLEU (Bilingual Evaluation Understudy) , while the length of the sentence increases more than 20 words.

What is attention layer in Deep Learning?

- Every RNN/LSTM can be interpreted as an Encoder and Decoder.
  - Encoder encodes the input samples of a chronological signal to hidden state.
  - Decoder decodes the hidden state to the output.

Attention is an interface connecting the encoder and decoder that provides the decoder with information from every encoder hidden state. With this framework, the model is able to selectively focus on valuable parts of the input sequence and hence, learn the association between them.

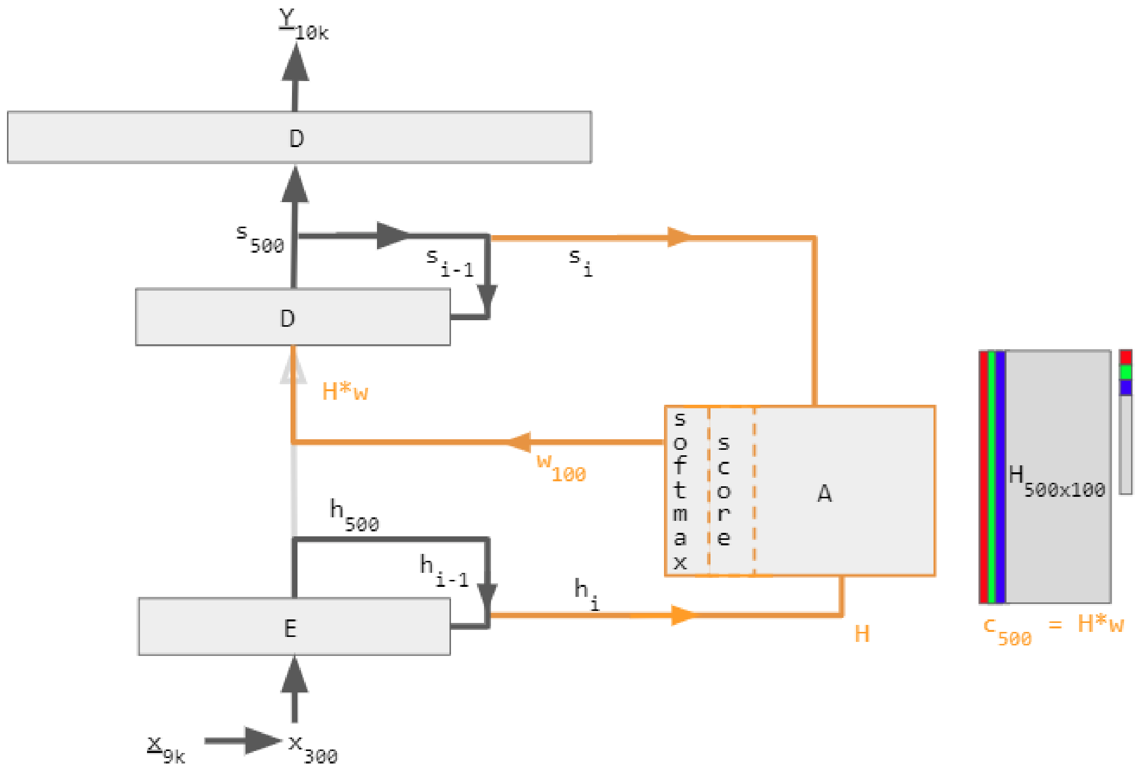


*Neural Machine Translation by Jointly Learning to Align and Translate

Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio

A language translation example

To build a machine that translates English-to-French (see the shown diagram), one starts with an Encoder-Decoder and grafts an attention unit to it. In the simplest case such as the example below, the attention unit is just lots of dot products of recurrent layer states and does not need training. In practice, the attention unit consists of 3 fully connected neural network layers that needs to be trained. The 3 layers are called Query, Key, and Value.
Encoder-Decoder with attention. This diagram uses specific values to relieve an already cluttered notation alphabet soup. The left part (in black) is the Encoder-Decoder, the middle part (in orange) is the attention unit, and the right part (in grey & colors) is the computed data. Grey regions in H matrix and w vector are zero values.
Numerical subscripts are examples of vector sizes. Lettered subscripts i and i-1 indicate time step.

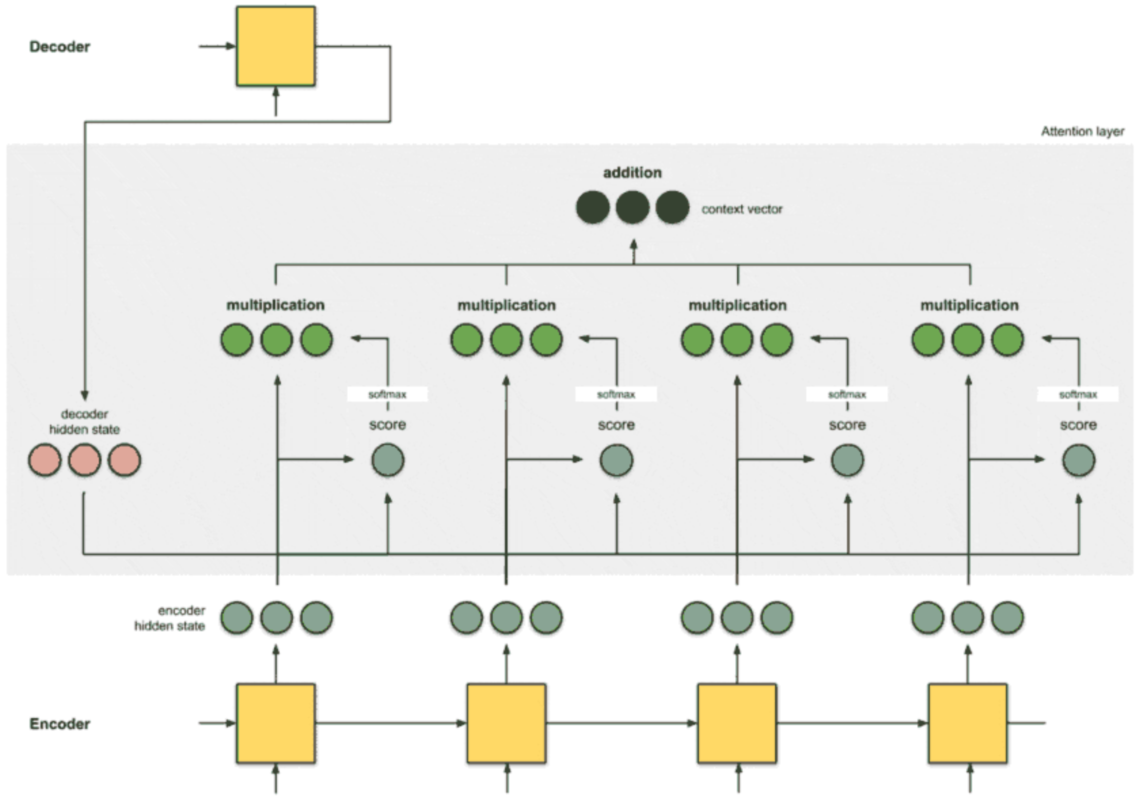


label	description
100	max sentence length
300	embedding size (word dimension)
500	length of hidden vector
9k, 10k	dictionary size of input & output languages respectively.
x, Y	9k and 10k 1-hot dictionary vectors. x → x implemented as a lookup table rather than vector multiplication. Y is the 1-hot maximizer of the linear Decoder layer D; that is, it takes the argmax of D's linear layer output.
x	300-long word embedding vector. The vectors are usually pre-calculated from other projects such as GloVe or Word2Vec.
h	500-long encoder hidden vector. At each point in time, this vector summarizes all the preceding words before it. The final h can be viewed as a "sentence" vector, or a thought vector as Hinton calls it.
s	500-long decoder hidden state vector.
E	500 neuron RNN encoder. 500 outputs. Input count is 800–300 from source embedding + 500 from recurrent connections. The encoder feeds directly into the decoder only to initialize it, but not thereafter; hence, that direct connection is shown very faintly.
D	2-layer decoder. The recurrent layer has 500 neurons and the fully connected linear layer has 10k neurons (the size of the target vocabulary).[7] The linear layer alone has 5 million (500 * 10k) weights -- ~10 times more weights than the recurrent layer.
score	100-long alignment score
w	100-long vector attention weight. These are "soft" weights which changes during the forward pass, in contrast to "hard" neuronal weights that change during the learning phase.
A	Attention module — this can be a dot product of recurrent states, or the Query-Key-Value fully connected layers. The output is a 100-long vector w.
H	500x100. 100 hidden vectors h concatenated into a matrix
c	500-long context vector = H * w. c is a linear combination of h vectors weighted by w.

Another Intuition

The below figure demonstrates an Encoder-Decoder architecture with an attention layer.

The idea is to keep the decoder as it is, and we just replace sequential RNN/LSTM with bidirectional RNN/LSTM in the encoder.
Here, we give attention to some words by considering a window size *Tx* (say four words *x1*, *x2*, *x3*, and *x4*). Using these four words, we'll create a context vector *c1*, which is given as input to the decoder. Similarly, we'll create a context vector *c2* using these four words. Also, we have α1, α2, and α3 as weights, and the sum of all weights within one window is equal to 1. Similarly, we create context vectors from different sets of words with different α values. The attention model computes a set of attention weights denoted by α(t,1), α(t,2),..,α(t,t) because not all the inputs would be used in generating the corresponding output. The context vector *ci* for the output word *yi* is generated using the weighted sum of the annotations. The attention weights are calculated by normalizing the output score of a feed-forward neural network described by the function that captures the alignment between input at *j* and output at *i*.



$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

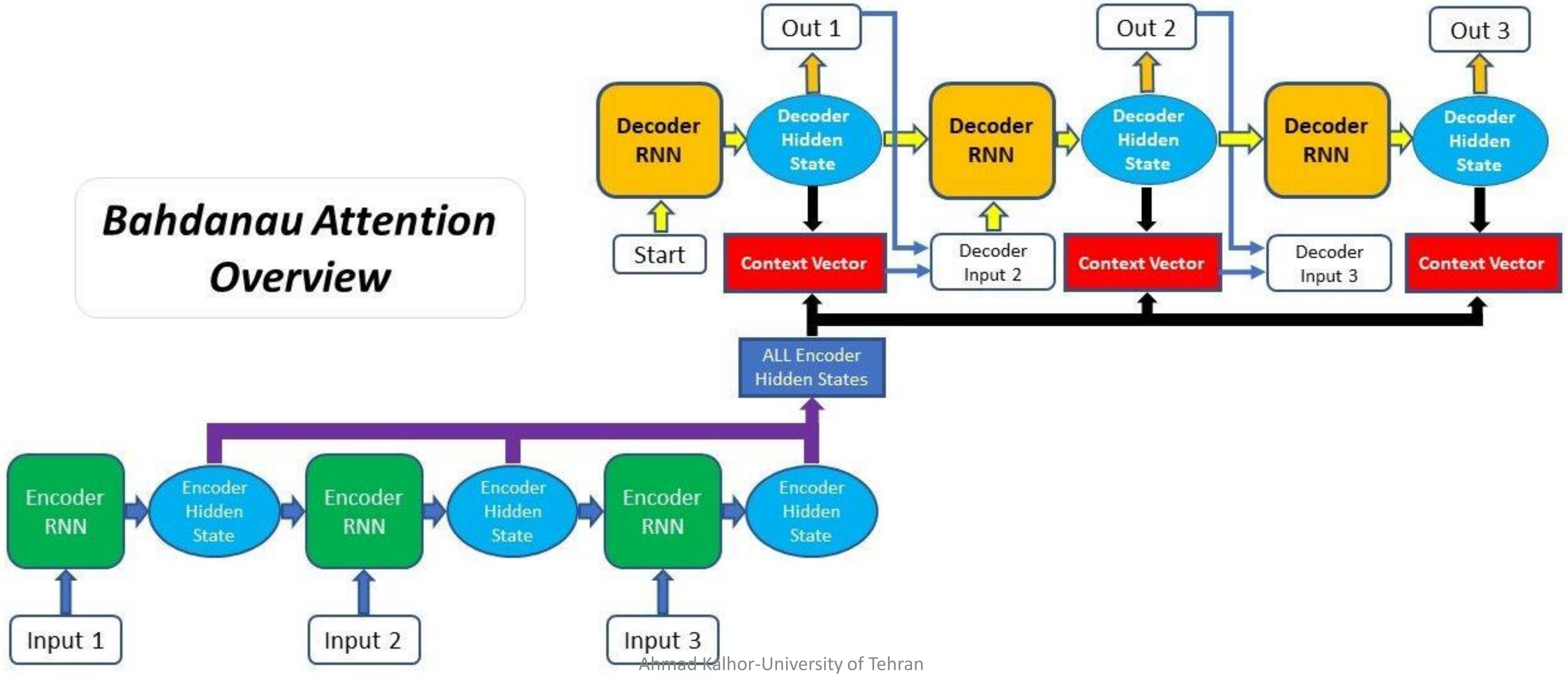
$$e_{ij} = a(s_{i-1}, h_j)$$

***Encoder-Decoder Recurrent Neural Network Models for Neural Machine Translation**
by **Jason Brownlee** on January 1, 2018 in **Deep Learning for Natural Language Processing**

The implementations of an attention layer can be broken down into 4 steps.

- **Step 0: Prepare hidden states.**
- First, prepare all the available encoder hidden states (green) and the first decoder hidden state (red). In our example, we have 4 encoder hidden states and the current decoder hidden state. (Note: the last consolidated encoder hidden state is fed as input to the first time step of the decoder. The output of this first time step of the decoder is called the first decoder hidden state.)
- **Step 1: Obtain a score for every encoder hidden state.**
- A score (scalar) is obtained by a score function (also known as alignment score function or alignment model). In this example, the score function is a dot product between the decoder and encoder hidden states.
- **Step 2: Run all the scores through a softmax layer.**
- We put the scores to a softmax layer so that the softmax scores (scalar) add up to 1. These softmax scores represent the attention distribution.
- **Step 3: Multiply each encoder hidden state by its softmax score.**
- By multiplying each encoder hidden state with its softmax score (scalar), we obtain the alignment vector or the annotation vector. This is exactly the mechanism where alignment takes place.
- **Step 4: Sum the alignment vectors.**
- The alignment vectors are summed up to produce the context vector. A context vector is an aggregated information of the alignment vectors from the previous step.
- **Step 5: Feed the context vector into the decoder.**

Bahdanau Attention Overview



Attention variants

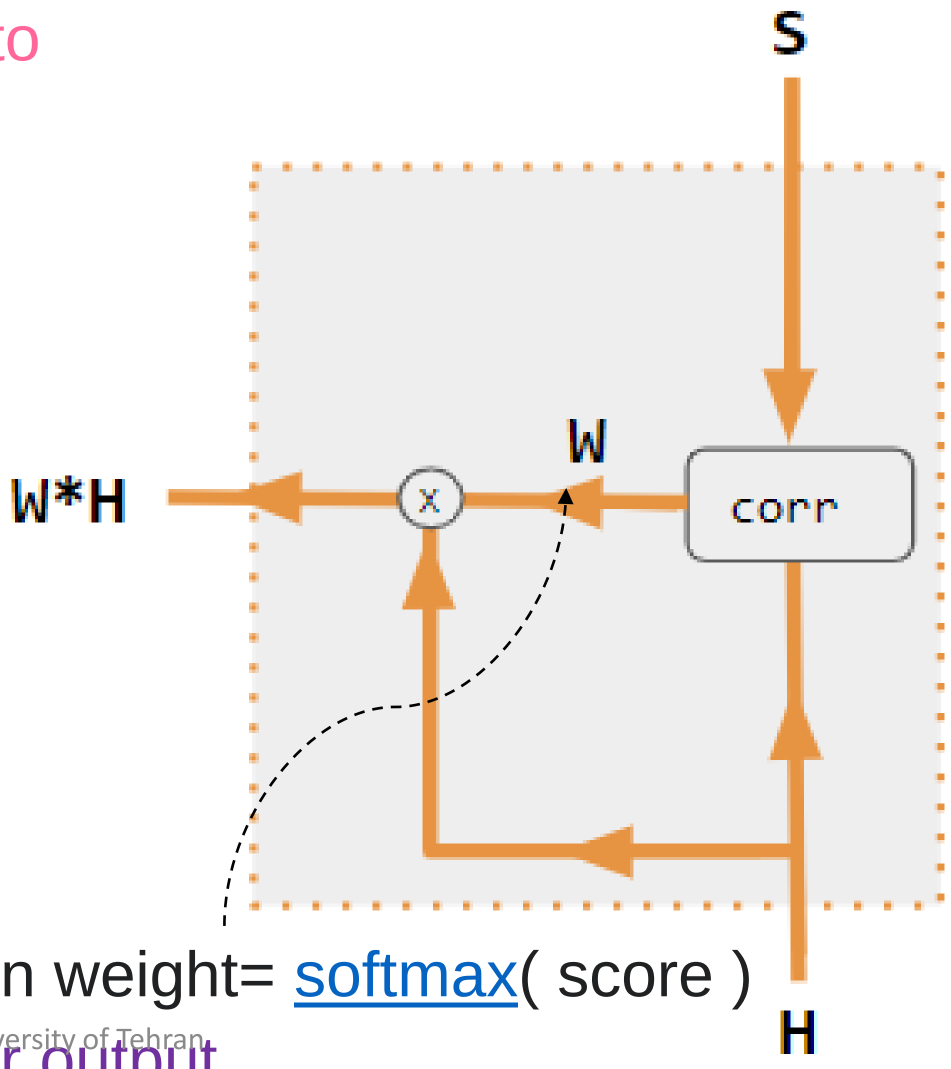
1. encoder-decoder dot product
2. encoder-decoder QKV
3. encoder-only dot product
4. encoder-only QKV
5. Pytorch tutorial

1. encoder-decoder dot product

s = decoder input to Attention
=Decoder hidden state

Both Encoder & Decoder are needed to calculate Attention.

c = context vector = H*w



1.*Luong, Minh-Thang (2015-09-20). "Effective Approaches to Attention-based Neural Machine Translation". arXiv:1508.04025v5 [cs.CL].*

w = attention weight= softmax(score)

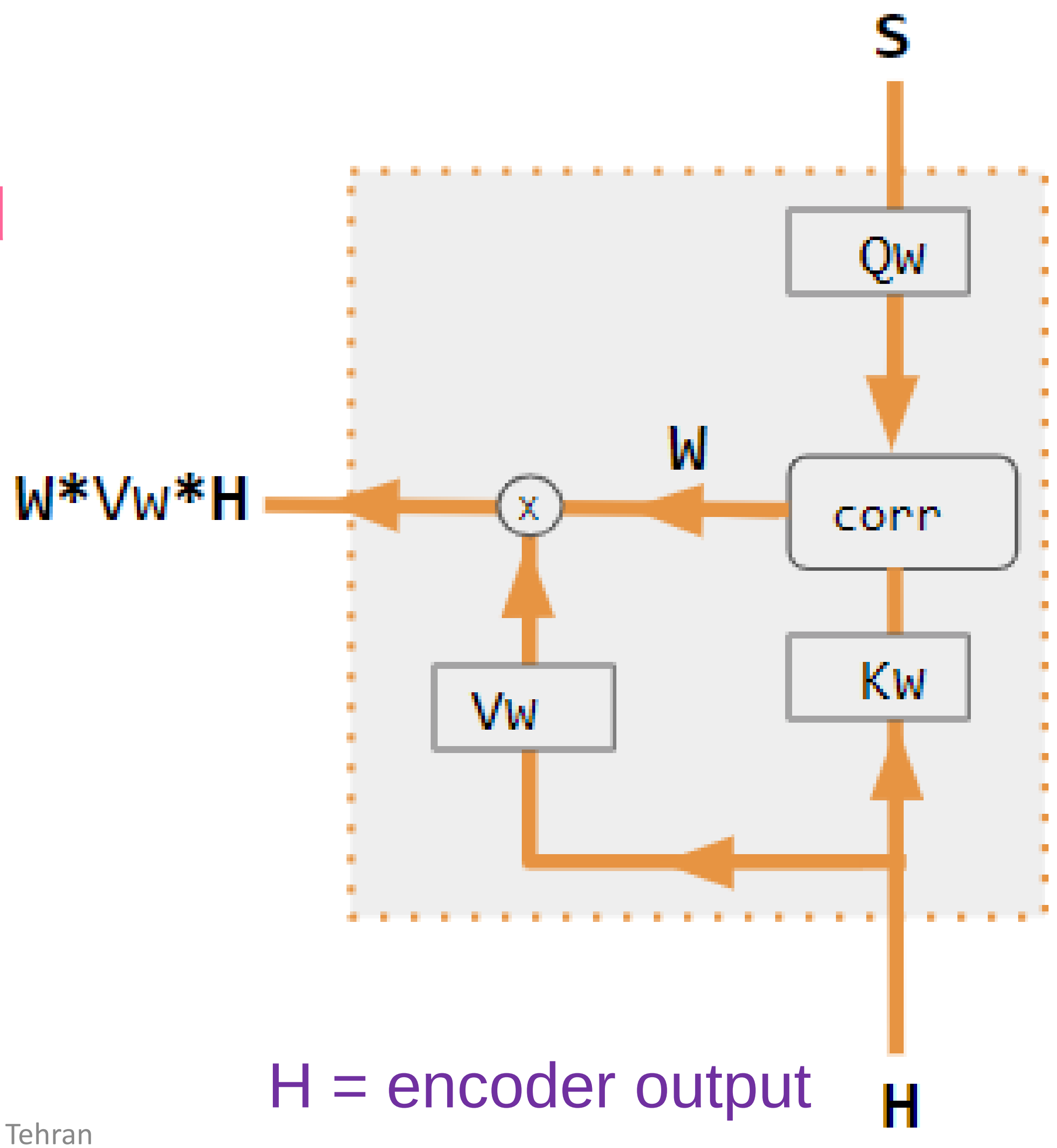
H = encoder output

2. encoder-decoder QKV

s = decoder input to Attention

Both Encoder & Decoder are needed to calculate Attention.

c = context vector

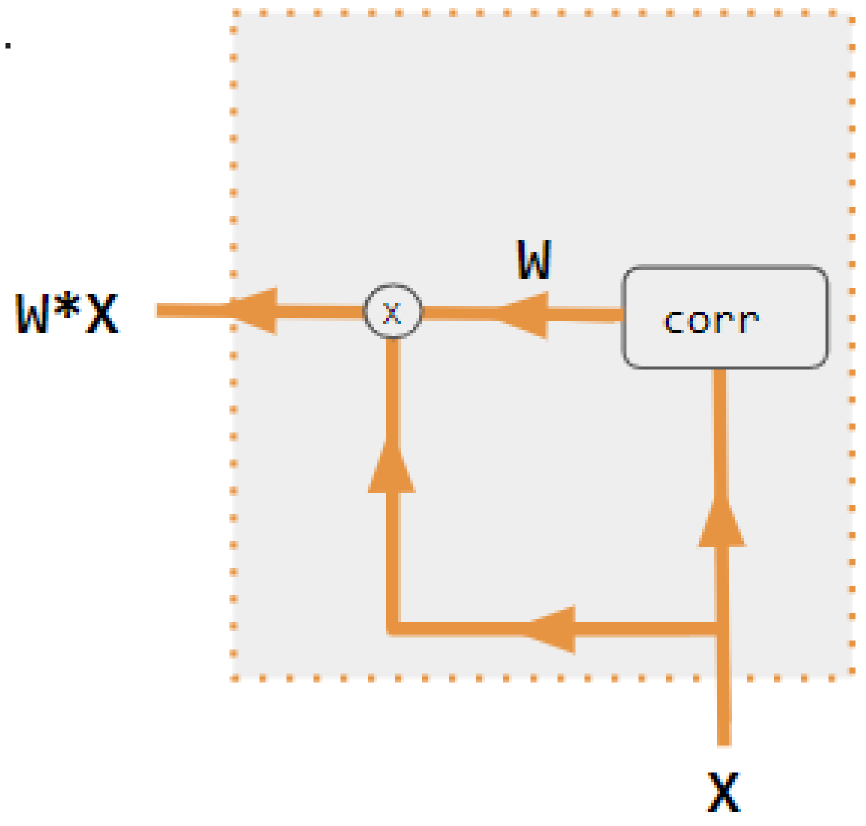


H = encoder output

1. *Neil Rhodes (2021).* *[CS 152 NN—27: Attention: Keys, Queries, & Values](). Event occurs at 06:30. Retrieved 2021-12-22.*

3. encoder-only dot product

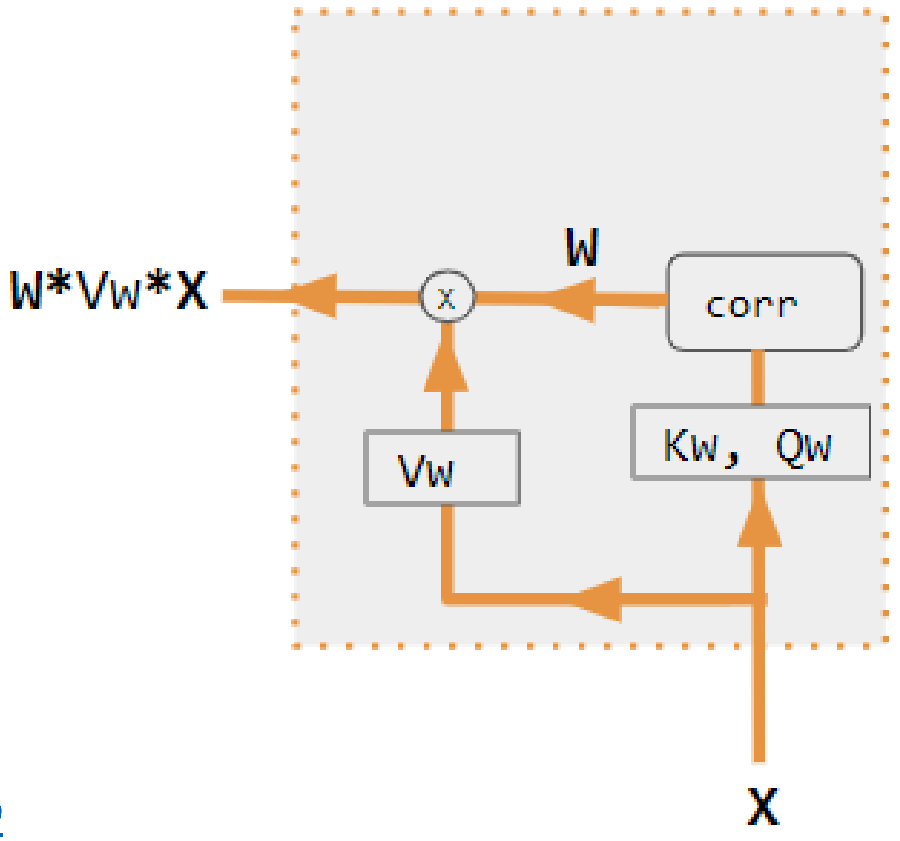
Decoder is NOT used to calculate Attention. With only 1 input into corr, W is an auto-correlation of dot products.
$w_{ij} = x_i * x_j$ [10]



1. *Alfredo Canziani & Yann Lecun (2021). NYU Deep Learning course, Spring 2020. Event occurs at 05:30. Retrieved 2021-12-22.*

4. encoder-only QKV

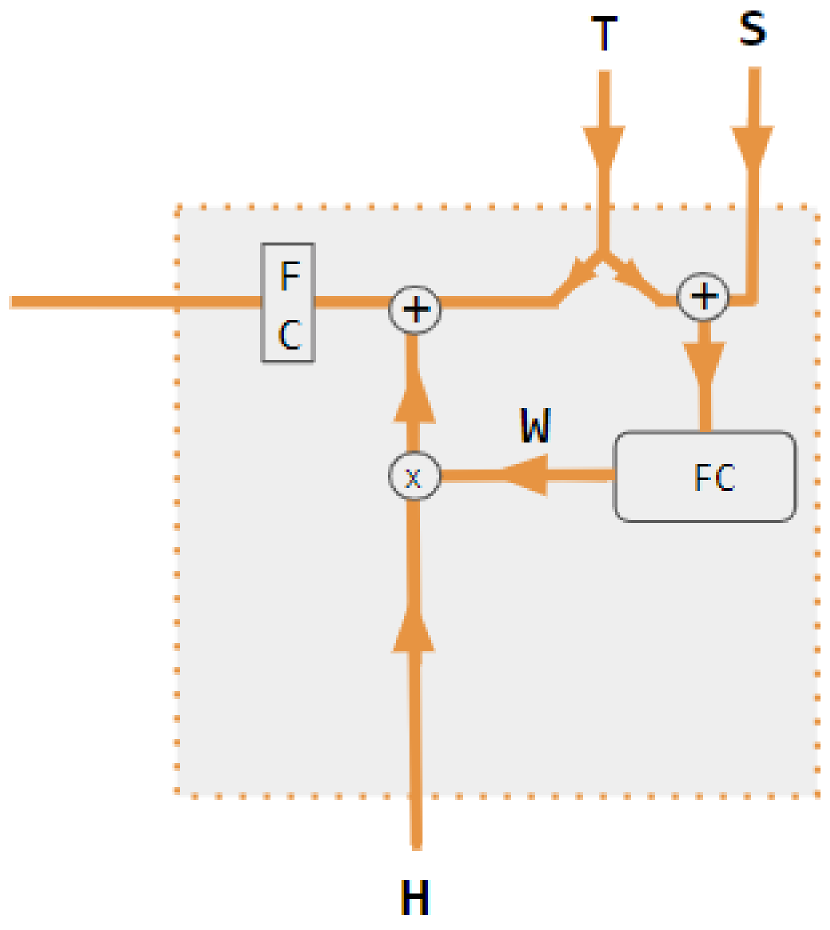
Decoder is NOT used to calculate Attention.[11]



1. *Alfredo Canziani & Yann Lecun (2021).* *NYU Deep Learning course, Spring 2020. Event occurs at 20:15. Retrieved 2021-12-22.*

5. Pytorch tutorial

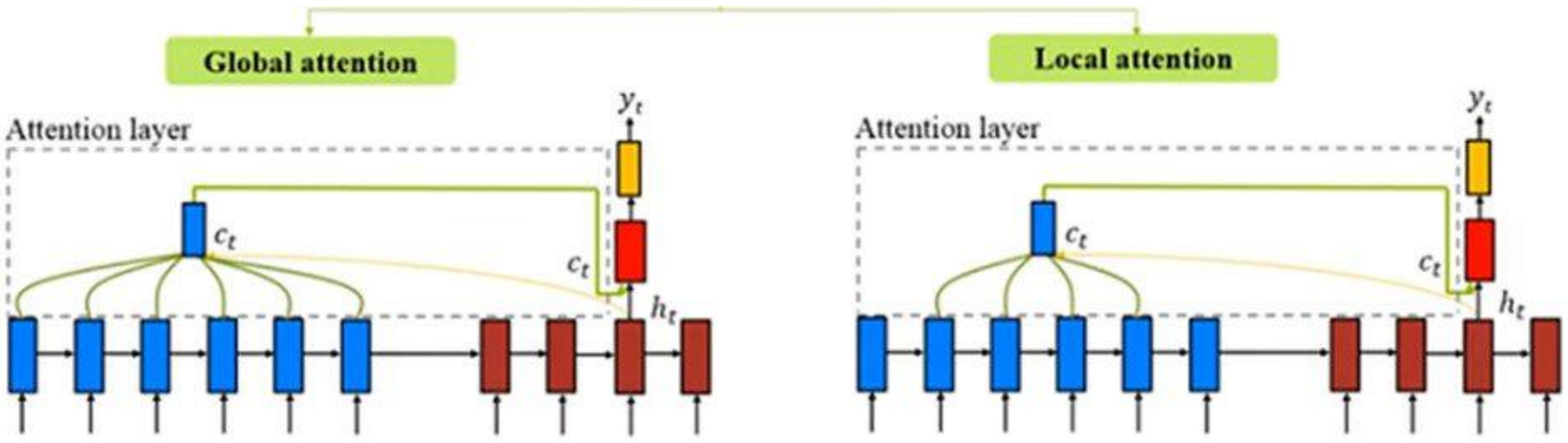
A FC layer is used to calculate Attention instead of dot product correlation.



1. *Robertson, Sean. ["NLP From Scratch: Translation With a Sequence To Sequence Network and Attention"]. pytorch.org. Retrieved 2021-12-22.*

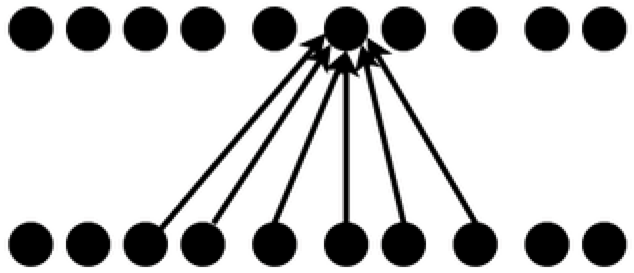
Types of attention

Depending on how many source states contribute while deriving the attention vector (α), there can be three types of attention mechanisms:

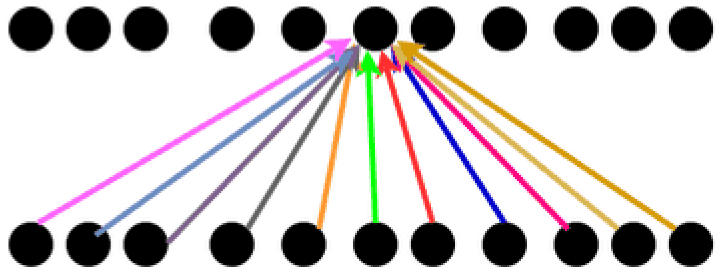


•**Global Attention**: When attention is placed on all source states. In global attention, we require as many weights as the source sentence length.
•**Local Attention**: When attention is placed on a few source states.
•**Hard Attention**: When attention is placed on only one source state.
You can check out my Kaggle notebook or GitHub repo to implement NMT with the attention mechanism using TensorFlow.

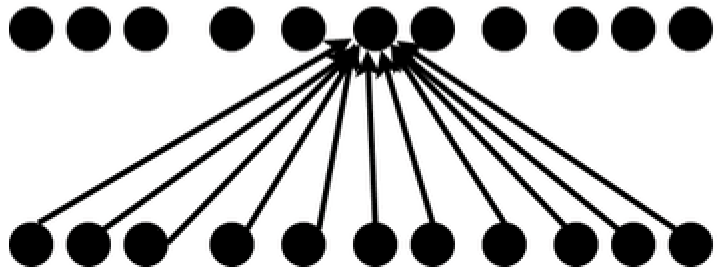
Convolution



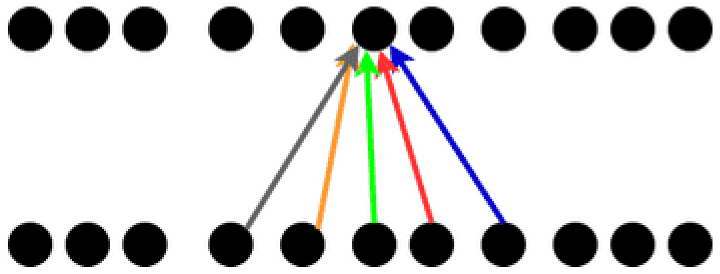
Global attention



Fully Connected layer



Local attention



Normalizing Layers

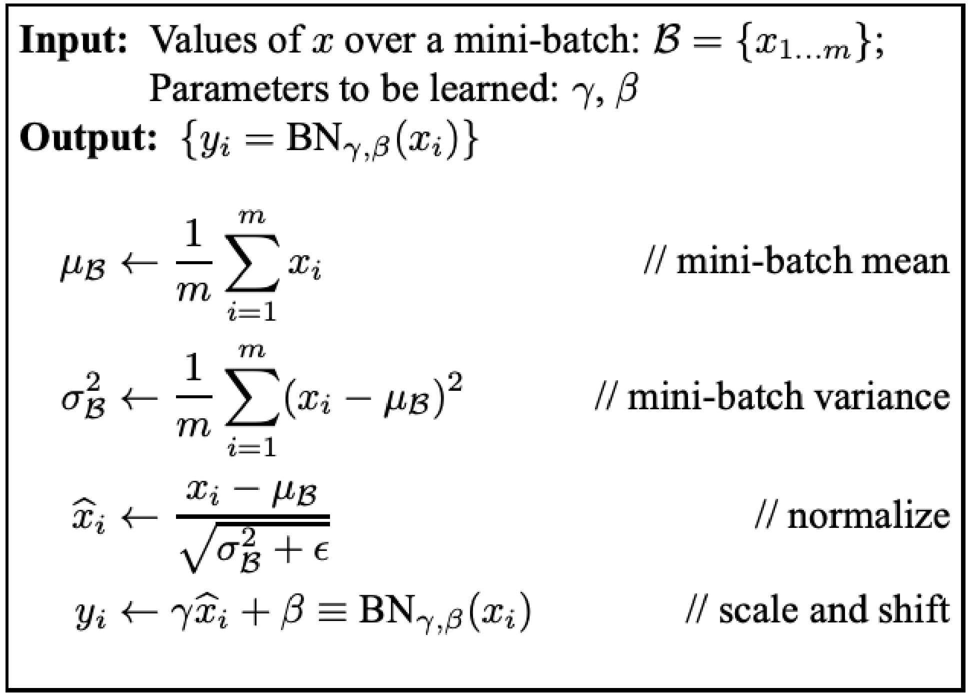
- Batch Normalization
- Weight Normalization
- Layer Normalization
- Group Normalization
- Weight Standarization

*Nilesh Vijayrania
Intrigued about Deep learning and all things ML.
Dec 10, 2020

Batch Normalization(BN)

Batch Normalization focuses on standardizing the inputs to any particular layer(i.e. activations from previous layers). Standardizing the inputs mean that inputs to any layer in the network should have approximately zero mean and unit variance. Mathematically, BN layer transforms each input in the current mini-batch by subtracting the input mean in the current mini-batch and dividing it by the standard deviation.
But each layer doesn’t need to expect inputs with zero mean and unit variance, but instead, probably the model might perform better with some other mean and variance. Hence the BN layer also introduces two learnable parameters γ and β.
The whole layer operation is as follows. It takes an input x_i and transforms it into y_i as described in the below table

The question is how BN helps NN training? Intuitively, In gradient descent, the network calculates the gradient based on the current inputs to any layer and reduce the weights in the direction indicated by the gradient. But since the layers are stacked one after the other, the data distribution of input to any particular layer changes too much due to slight update in weights of earlier layer, and hence the current gradient might produce suboptimal signals for the network. But BN restricts the distribution of the input data to any particular layer(i.e. the activations from the previous layer) in the network, which helps the network to produce better gradients for weights update. Hence BN often provides a much stable and accelerated training regime.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

However below are the few cons of Batch Normalization.
•BN calculates the batch statistics(Mini-batch mean and variance) in every training iteration, therefore it requires larger batch sizes while training so that it can effectively approximate the population mean and variance from the mini-batch. This makes BN harder to train networks for application such as object detection, semantic segmentation, etc because they generally work with high input resolution(often as big as 1024x 2048) and training with larger batch sizes is not computationally feasible.
•BN does not work well with RNNs. The problem is RNNs have a recurrent connection to previous timestamps and would require a separate β and γ for each timestep in the BN layer which instead adds additional complexity and makes it harder to use BN with RNNs.
•Different training and test calculation: During test(or inference) time, the BN layer doesn't calculate the mean and variance from the test data mini-batch(steps 1 and 2 from the algorithm table above) but uses the fixed mean and variance calculated from the training data. This requires cautious while using BN and introduces additional complexity. In pytorch `model.eval()` makes sure to set the model in evaluation model and hence the BN layer leverages this to use fixed mean and variance from pre-calculated from training data.

Weight Normalization

- Due to the disadvantages of Batch Normalization, T. Saliman and P. Kingma proposed Weight Normalization. Their idea is to decouple the length from the direction of the weight vector and hence reparameterize the network to speed up the training.
- What does reparameterization mean for Weight Normalization?
- The authors of the Weight Normalization paper suggested using two parameters g(for length of the weight vector) and v(the direction of the weight vector) instead of w for gradient descent in the following manner.

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

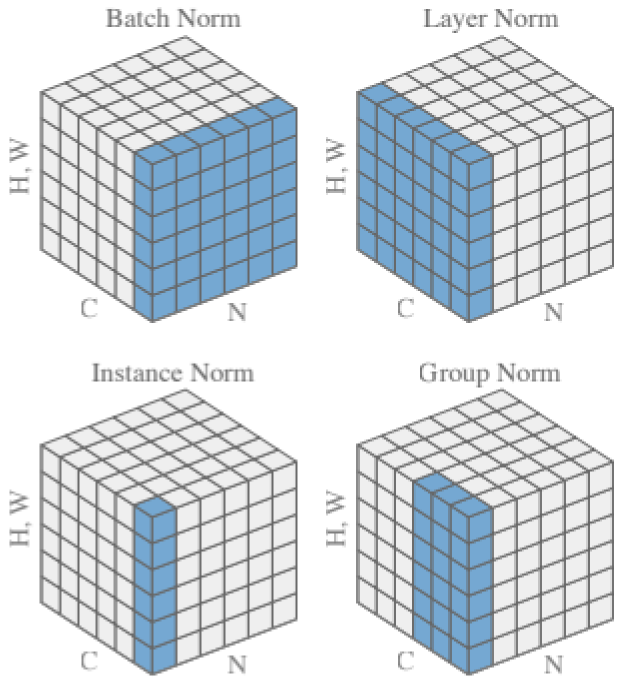
- Weight Normalization speeds up the training similar to batch normalization and unlike BN, it is applicable to RNNs as well. But the training of deep networks with Weight Normalization is significantly less stable compared to Batch Normalization and hence it is not widely used in practice.

Layer Normalization(LN)

- Inspired by the results of Batch Normalization, Geoffrey Hinton et al. proposed Layer Normalization which normalizes the activations along the feature direction instead of mini-batch direction. This overcomes the cons of BN by removing the dependency on batches and makes it easier to apply for RNNs as well.
- In essence, Layer Normalization normalizes each feature of the activations to zero mean and unit variance.

Group Normalization(GN)

- Similar to layer Normalization, Group Normalization is also applied along the feature direction but unlike LN, it divides the features into certain groups and normalizes each group separately. In practice, Group normalization performs better than layer normalization, and its parameter *num_groups* is tuned as a hyperparameter.
- If you find BN, LN, GN confusing, the below image summarizes them very precisely. Given the activation of shape (N, C, H, W), BN normalizes the N direction, LN and GN normalize the C direction but GN additionally divides the C channels into groups and normalizes the groups individually.

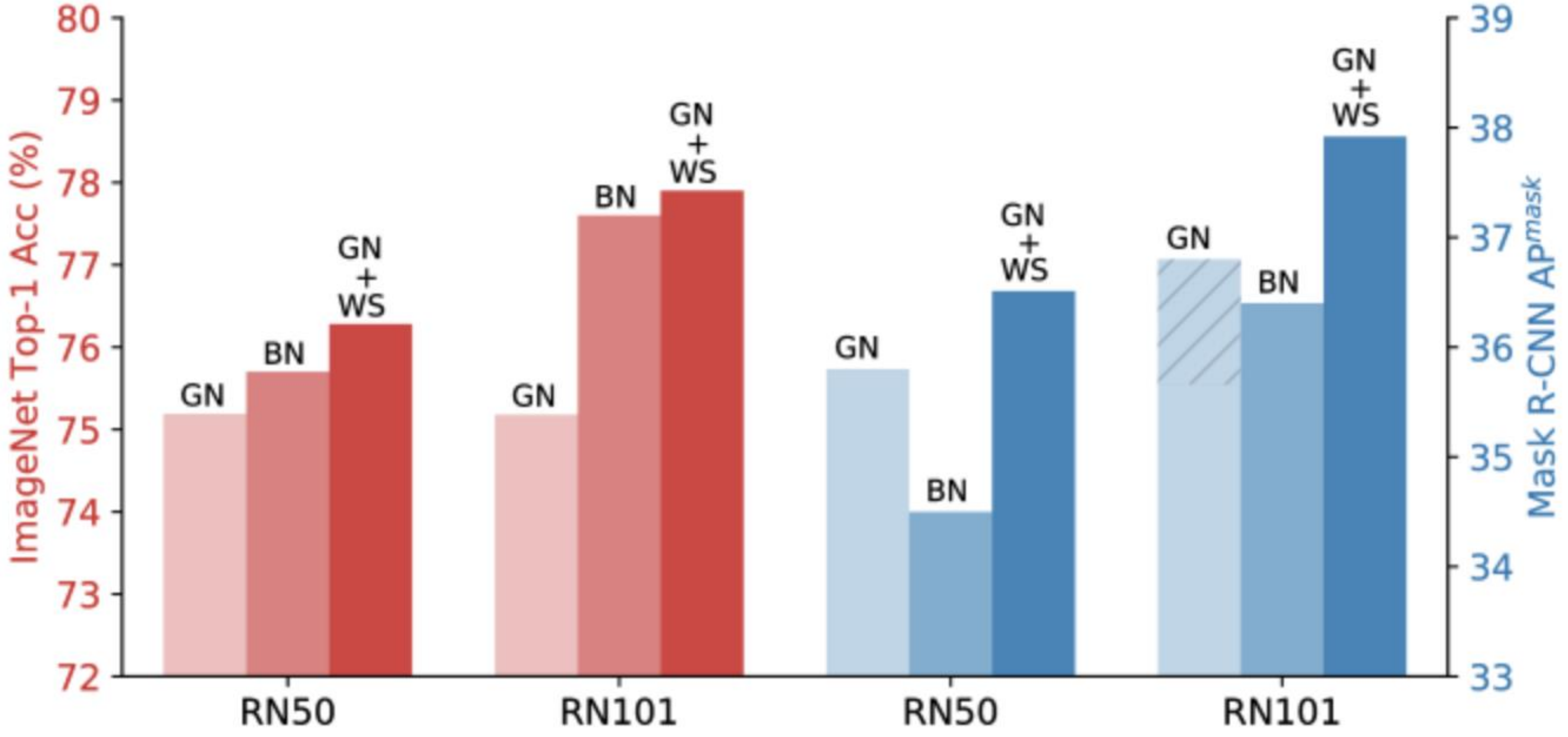

Batch Norm
Layer Norm
Instance Norm
Group Norm
H, W
C
N

Weight Standardization(WS)

Weight Standardization is transforming the weights of any layer to have zero mean and unit variance. This layer could be a convolution layer, RNN layer or linear layer, etc. For any given layer with shape(N, *) where * represents 1 or more dimensions, weight standardization, transforms the weights along the * dimension(s).
Below is the sample code for implementing weight standardization for the 2D conv layer in pytorch.

The basic idea is to only transform the weights during the forward pass and calculate activations accordingly. Pytorch will handle the backward pass out of the box. Similarly, it could be implemented for the linear layer as well. Recently, Siyun Qiao et al. introduced Weight Standardization in their paper “Micro-Batch Training with Batch-Channel Normalization and Weight Standardization” and found that group normalization when mixed with weight standardization, could outperform or perform equally well as BN even with batch size as small as 1. Shown below in the graph, the authors trained GN, BN, combination of GN+WS with Resnet50 and Resnet101 on Imagenet classification and MS COCO object detection task and found that GN+WS consistently outperforms the BN version even with much smaller batches than BN uses. This has attracted attention in dense prediction tasks such as semantic segmentation, instance segmentation which are usually not trainable with larger batch sizes due to memory constraints.

Image Credits: Siyuan Qiao et al. Weight Standardization. GN+WS effect on classification and object detection task[4]



- In conclusion, Normalization layers in the model often helps to speed up and stabilize the learning process. If training with large batches isn’t an issue and if the network doesn’t have any recurrent connections, Batch Normalization could be used. For training with smaller batches or complex layer such as LSTM, GRU, Group Normalization with Weight Standardization could be tried instead of Batch Normalization.
- One important thing to note is, in practice the normalization layers are used in between the Linear/Conv/RNN layer and the ReLU non-linearity(or hyperbolic tangent etc) so that when the activations reach the Non-linear activation function, the activations are equally centered around zero. This would potentially avoid the dead neurons which never get activated due to wrong random initialization and hence can improve training.
- Below is the list of references used for this post and should be considered for further experiment details.

- Ioffe, Sergey, and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” *arXiv preprint arXiv:1502.03167* (2015).
- Salimans, Tim, and Durk P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." *Advances in neural information processing systems* 29 (2016): 901-909.
- Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. “Layer normalization.” *arXiv preprint arXiv:1607.06450* (2016).
- Qiao, Siyuan, et al. “Weight standardization.” *arXiv preprint arXiv:1903.10520* (2019)

Down-sampling/Up-sampling Layers

- Nowadays, Deep Neural Networks are among the main tools used in various sciences. Convolutional Neural Network is a special type of DNN consisting of several convolution layers, each followed by an activation function and a pooling layer. The pooling layer is an important layer that executes the down-sampling on the feature maps coming from the previous layer and produces new feature maps with a condensed resolution. This layer drastically reduces the spatial dimension of input. It serves two main purposes. The first is to reduce the number of parameters or weights, thus lessening the computational cost. The second is to control the overfitting of the network. An ideal pooling method is expected to extract only useful information and discard irrelevant details. There are a lot of methods for the implementation of pooling operation in Deep Neural Networks. In this paper, we reviewed some of the famous and useful pooling methods

Pooling Methods in Deep Neural Networks, a Review Hossein Gholamalinezhad1 , Hossein Khosravi*,2021

The three types of pooling operations are:

- Max pooling: The maximum pixel value of the batch is selected.
- Min pooling: The minimum pixel value of the batch is selected.
- Average pooling: The average value of all the pixels in the batch is selected.
- Mixed Pooling
- Lp ppoling
- Stochastic ppoling
- Spatial Pyramid Pooling
- Region of Interest Pooling

Other pooling layers

- 3. Novel Pooling Methods
- 3.1. Multi-scale order-less pooling (MOP)
- 3.2. Super-pixel Pooling
- 3.3. PCA Networks
- 3.4. Compact Bilinear Pooling
- 3.5. Lead Asymmetric Pooling (LAP)
- 3.6. Edge-aware Pyramid Pooling
- 3.7. Spectral Pooling
- 3.8. Row-Wise Max-Pooling
- 3.9. Intermap Pooling
- 3.10. Per-pixel Pyramid Pooling
- 3.11. Rank-based Average Pooling
- 3.12. Weighted Pooling
- 3.13. Genetic-Based Pooling

Down-sampling/Up-sampling layers

1.2 Architectures

- General Topologies in DNNs
- CNNs
- GANs and VAEs
- RNNs and Transformers
- Other architectures