ECE 1513: Introduction to Machine Learning

Lecture 10: Convolutional Neural Networks and Sequence Data

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2025

Recap: *Building and Training NN*

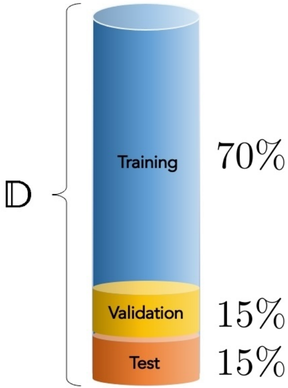
```
TrainingLoop():
```

Build NN $\boldsymbol{y} = f_{\mathbf{w}}(\boldsymbol{x})$ with some hyperparameters and *initial weights*
Split training set to mini-batches
Specify the loss function $\mathcal{L}$
for *epochs* = $1, \ldots, E$ **do**
    Keep applying mini-batch SGD
end for
Return final weights $\mathbf{w}^*$, average training loss, and accuracy on training set

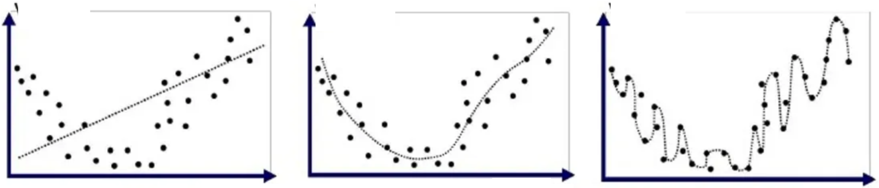
Recap: *Generalization*

? *How can we measure generalization?*

! *We try samples that we did not use for training*



Recap: *Underfitting and Overfitting*



- *Left* $\rightsquigarrow$ *Underfitting*
- *Middle* $\rightsquigarrow$ *Learning fairly*
- *Right* $\rightsquigarrow$ *Overfitting*

Recap: *Bias and Variance*

Generalization Error Components

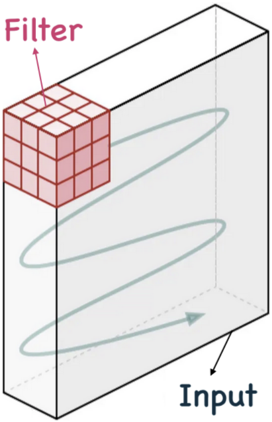
Generalization error is proportional to the model bias and variance

We can only minimize the bias and variance of our model output

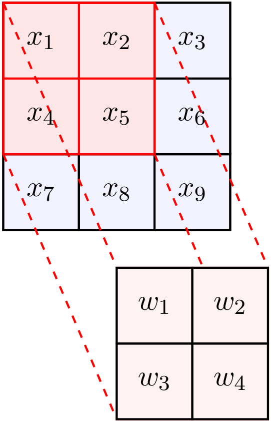
- *If we set the model to give us zero bias* $\rightsquigarrow$ *unbiased estimator*
  - ↳ *But this may lead to higher variance*
- *There is always a minimum error that we cannot beat* $\rightsquigarrow$ *Bayes error*

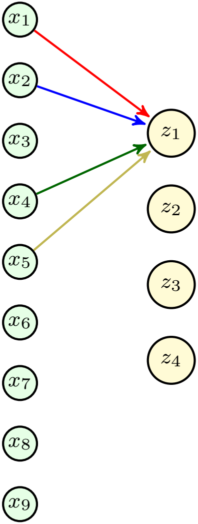
Recap: *Convolution*

In convolution, we slide a filter over the input sample

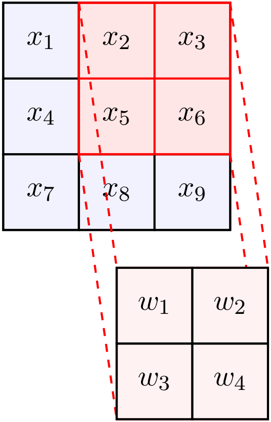


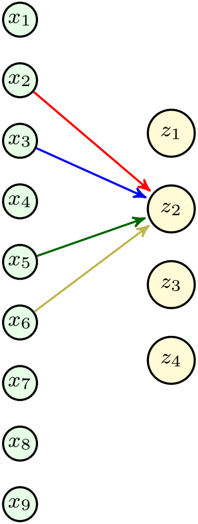
Recap: *Convolution as Sparse Linear Transform*



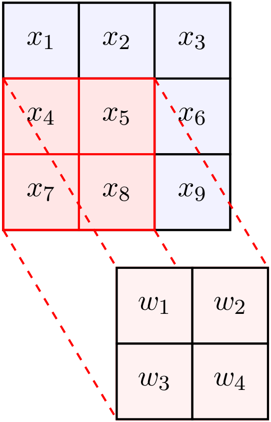


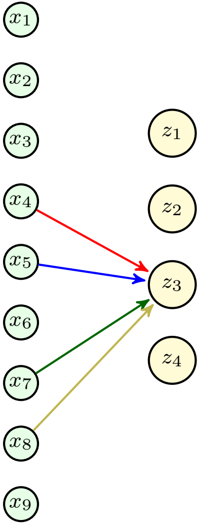
Recap: *Convolution as Sparse Linear Transform*



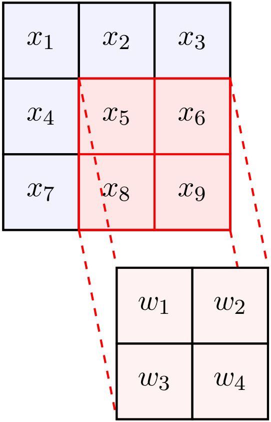


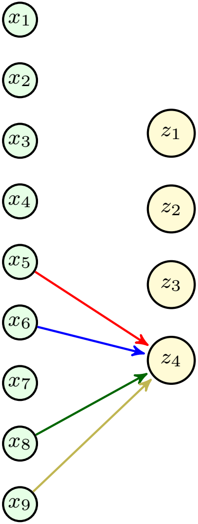
Recap: *Convolution as Sparse Linear Transform*





Recap: *Convolution as Sparse Linear Transform*

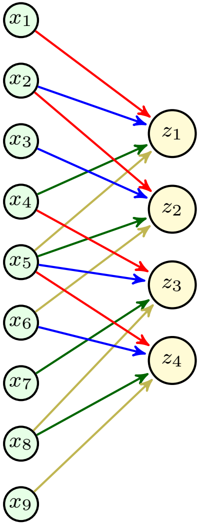




Recap: *Convolution as Sparse Linear Transform*

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

w_1	w_2
w_3	w_4



Today's Agenda: *CNNs and Sequence Data*

Today, we use the convolution to build more efficient FNNs called

Convolutional Neural Networks

In this way, we learn

- *Convolutional layers*
- *Pooling layers*
- *Architecture of deep CNNs*

We then start our final journey which explores briefly

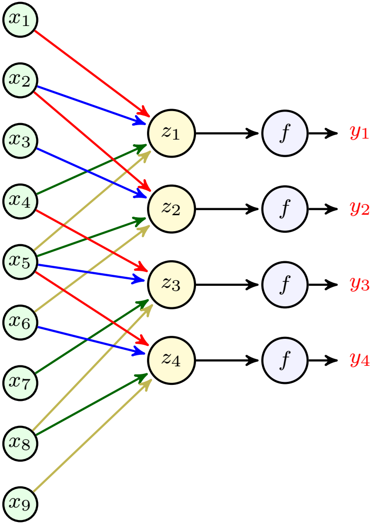
Advanced Topics in Machine Learning

In today's episode, we talk about

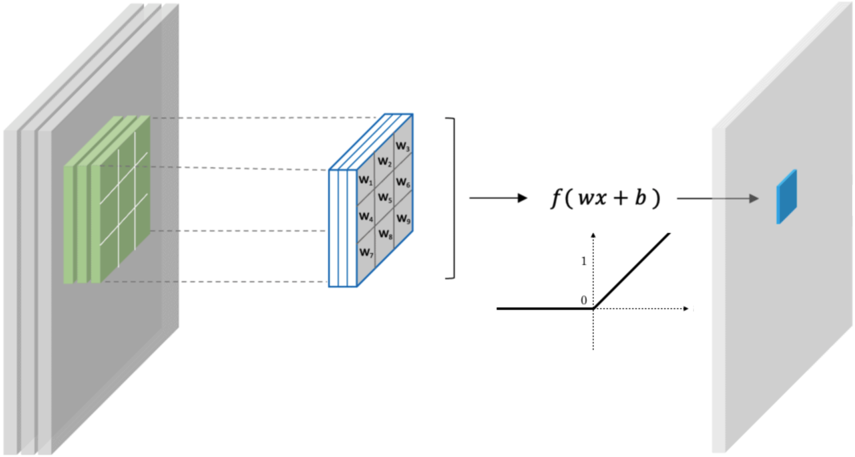
- *Sequence Data and Seq2Seq Models*

Building Neural Layers with Convolution

We can activate the convolution output to make a partially-connected layer

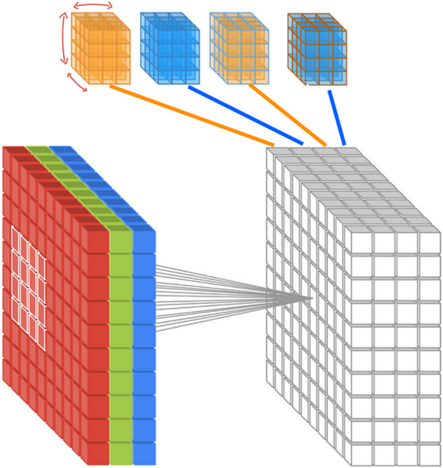


Convolutional Layer: *Single-Channel Output*



Convolutional Layer: *Multi-Channel Output*

To have multiple output channels, we can use multiple filters



Pooling: *Max-Pooling*

Pooling

Pooling is a convolution-like operation that computes a fix function in each slide

Example: *Max-Pooling*

$$\mathbf{X} = \begin{bmatrix} \boxed{\begin{matrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{matrix}} & \cdots & \cdots & X_{1,M} \\ & \cdots & \cdots & X_{2,M} \\ \vdots \quad \vdots & \vdots & \vdots & \vdots \\ X_{N,1} \quad X_{N,2} & \cdots & \cdots & X_{N,M} \end{bmatrix} \dashrightarrow Z = \max \{\text{window}\}$$

we pool the maximum

$$\mathbf{Z} = \begin{bmatrix} \quad \end{bmatrix}$$

Pooling: *Max-Pooling*

Pooling

Pooling is a convolution-like operation that computes a fix function in each slide

Example: *Max-Pooling*

$$\mathbf{X} = \begin{bmatrix} \boxed{\begin{matrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{matrix}} & \cdots & \cdots & X_{1,M} \\ & \cdots & \cdots & X_{2,M} \\ \vdots \quad\quad \vdots & \vdots & \vdots & \vdots \\ X_{N,1} \quad X_{N,2} & \cdots & \cdots & X_{N,M} \end{bmatrix} \dashrightarrow Z = \max\{\text{window}\}$$

we pool the maximum

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & & \\ & & \\ & & \end{bmatrix}$$

Pooling: *Max-Pooling*

Pooling

Pooling is a convolution-like operation that computes a fix function in each slide

Example: *Max-Pooling*

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & \boxed{X_{1,2} \quad \cdots} & \cdots & X_{1,M} \\ X_{2,1} & \boxed{X_{2,2} \quad \cdots} & \cdots & X_{2,M} \\ \vdots & \vdots \quad \vdots & \vdots & \vdots \\ X_{N,1} & X_{N,2} \quad \cdots & \cdots & X_{N,M} \end{bmatrix} \dashrightarrow Z = \max \{\text{window}\}$$

we pool the maximum

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & & \\ & & & \\ & & & \end{bmatrix}$$

Pooling: *Max-Pooling*

Pooling

Pooling is a convolution-like operation that computes a fix function in each slide

Example: *Max-Pooling*

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & \dots & \dots & X_{1,M} \\ X_{2,1} & X_{2,2} & \dots & \dots & X_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ X_{N,1} & X_{N,2} & \dots & \dots & X_{N,M} \end{bmatrix} \qquad Z = \max \{\text{window}\}$$

we pool the maximum

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2			

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2		

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2			

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8		

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8	3.8	

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3			

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3	3.8		

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3	3.8	3.8	

Max Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

3.2	3.2	2.1	1.5
3.2	3.8	3.8	4.3
2.3	3.8	3.8	4.3

Mean-Pooling

Mean-pooling is another approach in which we compute the average

$$\mathbf{X} = \begin{bmatrix} \boxed{\begin{matrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{matrix}} & \ldots & \ldots & X_{1,M} \\ & \ldots & \ldots & X_{2,M} \\ \vdots \quad \vdots & \vdots & \vdots & \vdots \\ X_{N,1} \quad X_{N,2} & \ldots & \ldots & X_{N,M} \end{bmatrix} \dashrightarrow \quad \texttt{mean}\{ \qquad \}$$

In each window, we pool the average

$$\mathbf{Z} = \begin{bmatrix} \quad \end{bmatrix}$$

Mean-Pooling

Mean-pooling is another approach in which we compute the average

$$\mathbf{X} = \begin{bmatrix} \boxed{\begin{matrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{matrix}} & \cdots & \cdots & X_{1,M} \\ & \cdots & \cdots & X_{2,M} \\ \vdots \quad \vdots & \vdots & \vdots & \vdots \\ X_{N,1} \quad X_{N,2} & \cdots & \cdots & X_{N,M} \end{bmatrix} \qquad \texttt{mean}\left\{ \qquad \right\}$$

In each window, we pool the average

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & & \\ & & \\ & & \end{bmatrix}$$

Mean-Pooling

Mean-pooling is another approach in which we compute the average

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & \boxed{X_{1,2} \quad \cdots} & \cdots & X_{1,M} \\ X_{2,1} & \boxed{X_{2,2} \quad \cdots} & \cdots & X_{2,M} \\ \vdots & \vdots \quad \vdots & \vdots & \vdots \\ X_{N,1} & X_{N,2} \quad \cdots & \cdots & X_{N,M} \end{bmatrix} \qquad \texttt{mean}\left\{ \qquad \right\}$$

In each window, we pool the average

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & & \\ & & & \\ & & & \end{bmatrix}$$

Mean-Pooling

Mean-pooling is another approach in which we compute the average

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & \dots & \dots & X_{1,M} \\ X_{2,1} & X_{2,2} & \dots & \dots & X_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ X_{N,1} & X_{N,2} & \dots & \dots & X_{N,M} \end{bmatrix} \qquad \mathtt{mean}\left\{ \qquad \right\}$$

In each window, we pool the average

$$\mathbf{Z} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05			

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9		

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85			

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275		

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	

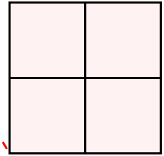
Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1



2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525			

Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9		

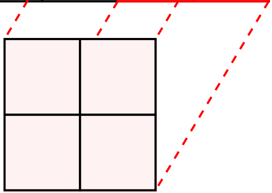
Mean-Pooling: *Numerical Example*

0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1

2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	

Mean-Pooling: *Numerical Example*

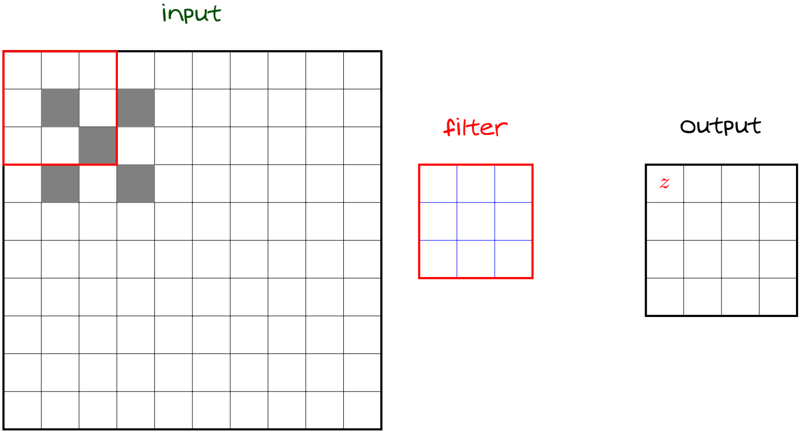
0.4	2.3	0	1.3	0
1.9	3.2	2.1	1.5	0
2.3	0	3.8	0	4.3
1.5	2.3	1.5	0.7	2.1



2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	1.775

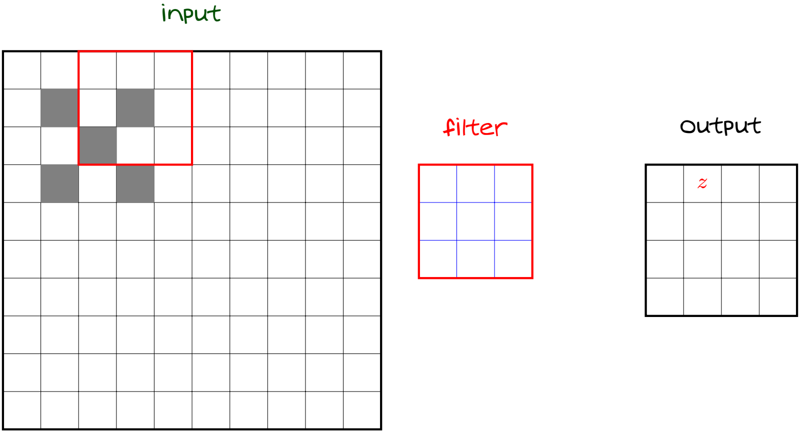
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



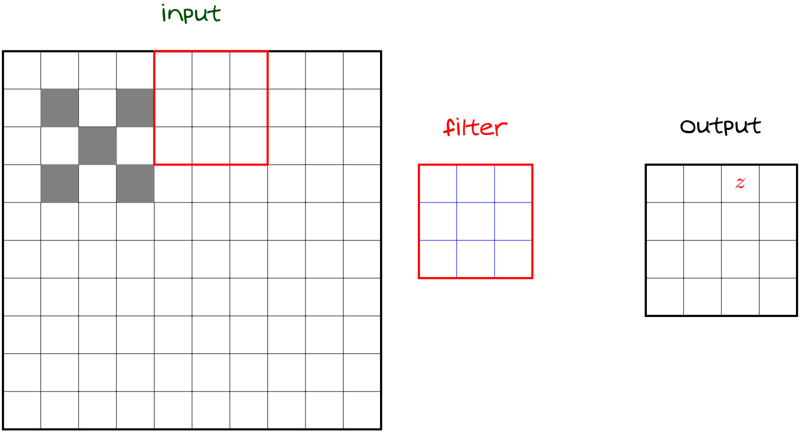
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



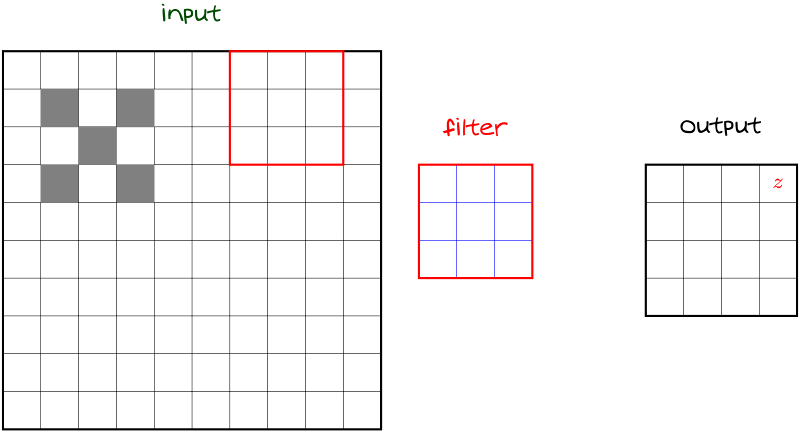
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



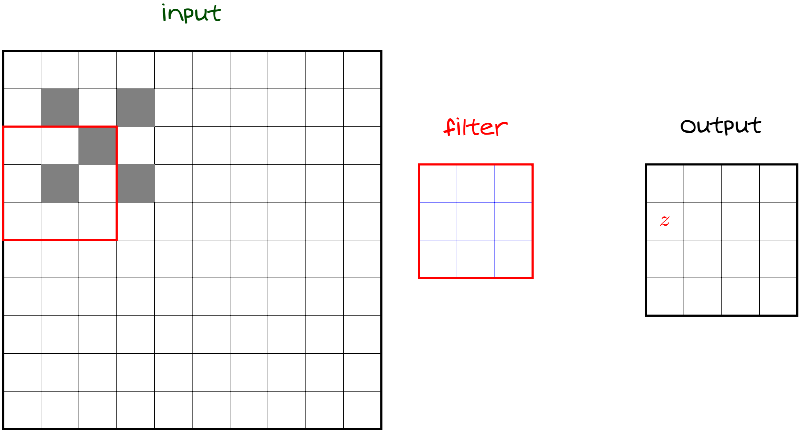
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



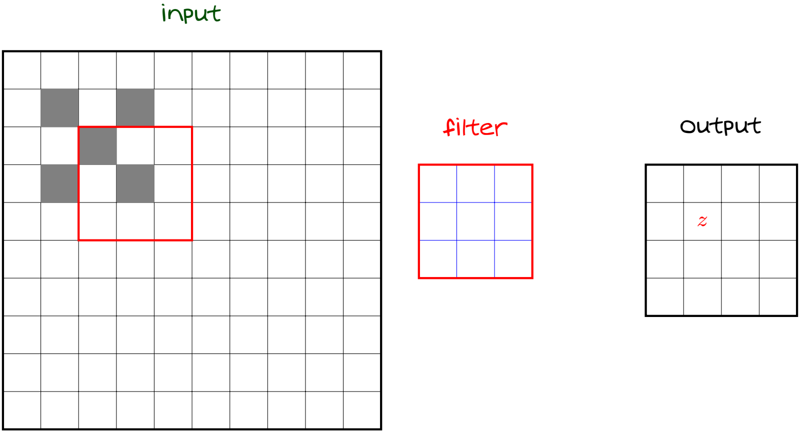
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



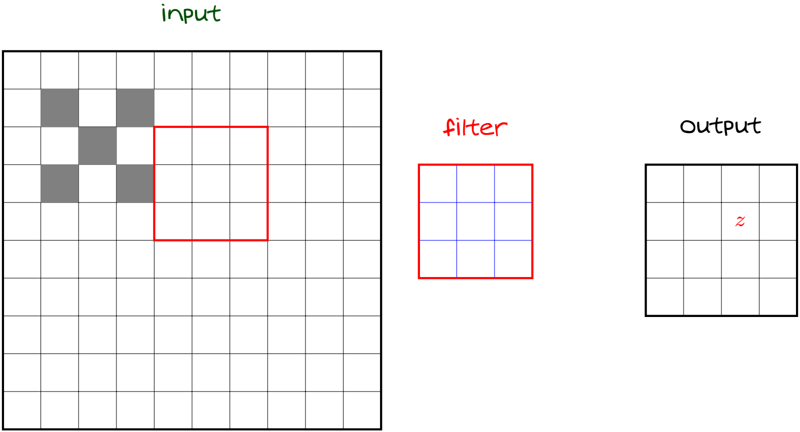
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



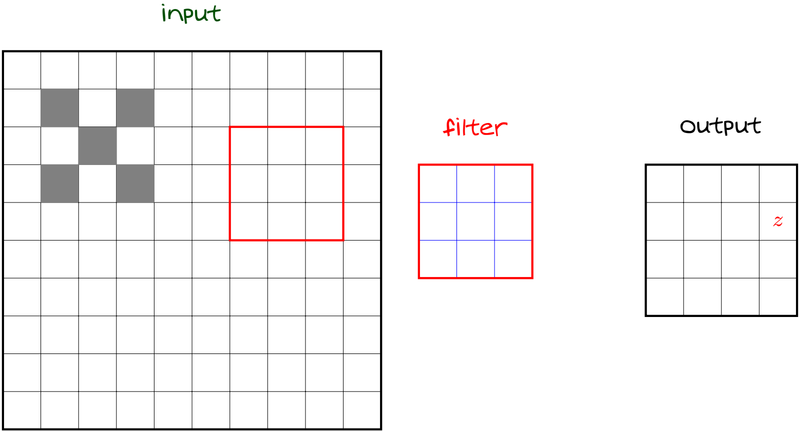
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



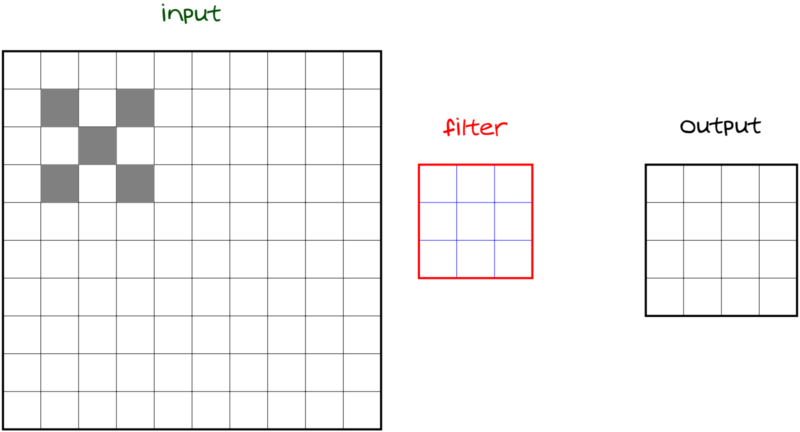
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



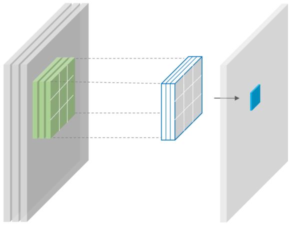
Convolution *with Stride*

We can perform all operations with *stride*, *e.g.*, *stride 2*



Convolutional Unit: *Complex Convolutional Layer*

A convolutional unit is made as

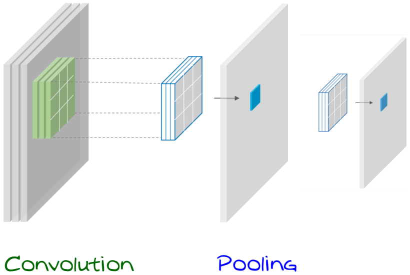


Convolution

- *The convolution performs less complex linear operation*

Convolutional Unit: *Complex Convolutional Layer*

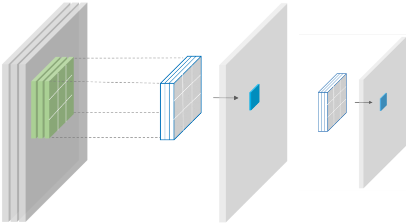
A convolutional unit is made as



- *The convolution performs less complex linear operation*
- *Pooling make the output smooth, i.e., less varying*

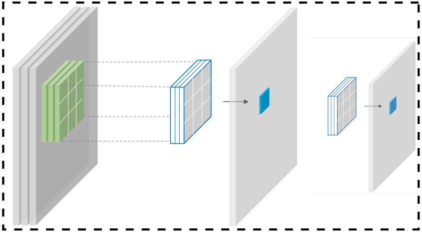
CNN: General Architecture

A CNN consists of multiple convolutional units and a fully-connected NN



CNN: General Architecture

A CNN consists of multiple convolutional units and a fully-connected NN

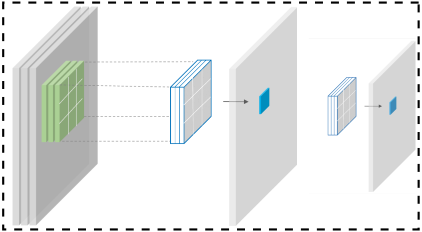


$\times L$

We can repeat the convolutional unit multiple times

CNN: General Architecture

A CNN consists of multiple convolutional units and a fully-connected NN



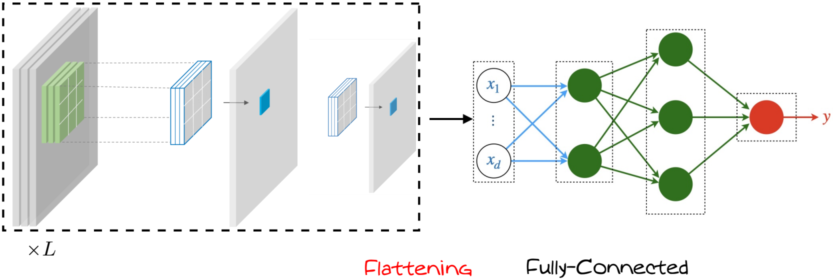
$\times L$

Flattening

We can repeat the convolutional unit multiple times

CNN: General Architecture

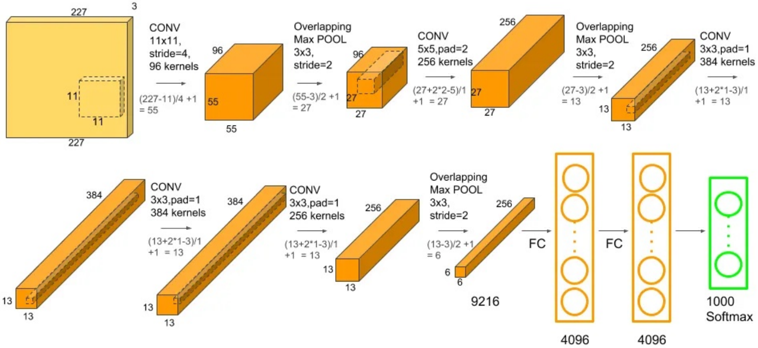
A CNN consists of multiple convolutional units and a fully-connected NN



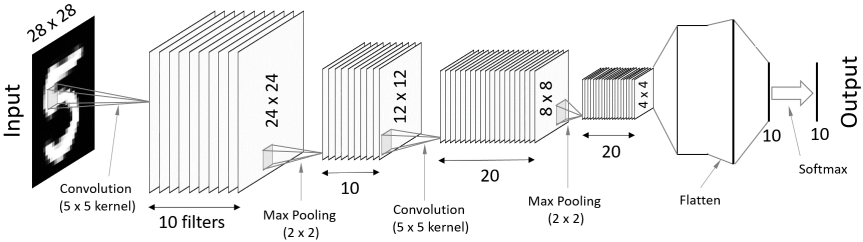
We can repeat the convolutional unit multiple times

Example: *AlexNet*

Let's look at the winner of the ImagNet challenge in 2012



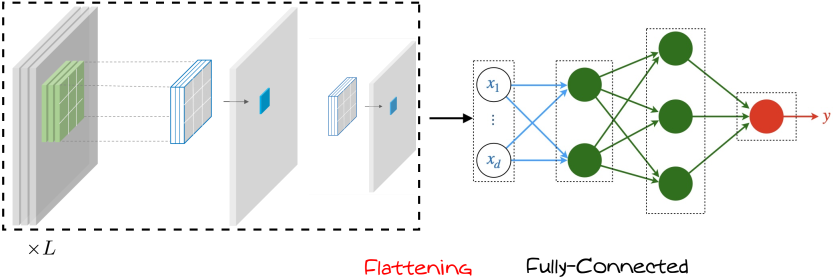
Example: *Custom CNN for MNIST Classification*



In this network, we do the following

- *We apply convolution with 10 filters*
  - ↳ *We apply pooling with stride 2*
- *We apply convolution with 20 filters*
  - ↳ *We apply pooling with stride 2*
- *We flatten and pass through one fully-connected layer*
  - ↳ *We compute Softmax for classification*

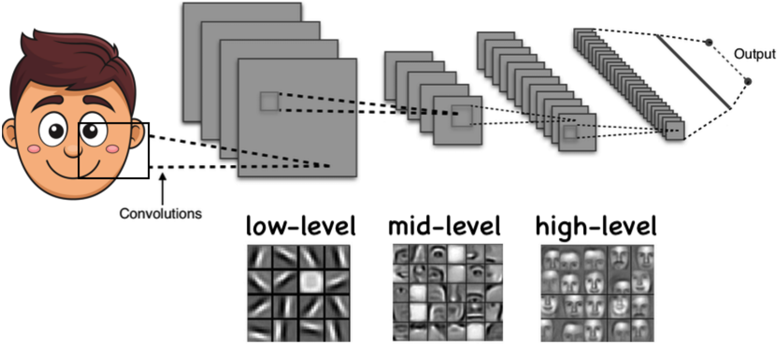
Backpropagation



It is easy to see that backward pass is dual to the forward pass

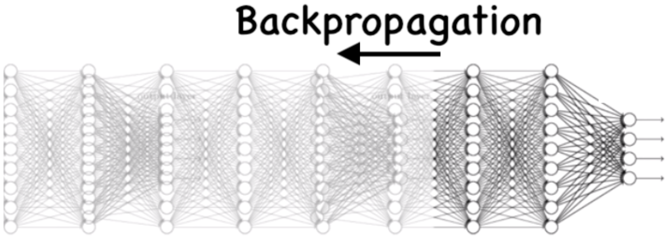
- *We can compute sample gradients with backpropagation*
- *Using mini-batch SGD, we can efficiently train CNNs*

Gradual Feature Extraction



As we go deeper, CNN extracts higher levels of features

Vanishing or Exploding Gradient

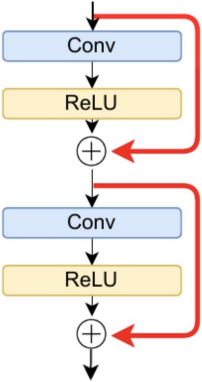


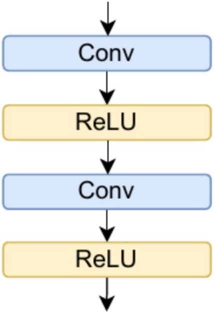
As we go very deep in FNNs, we could experience

- *Decrease in gradient values through depth*
  - ↳ *This results in vanishing gradient*
- *Increase in gradient values through depth*
  - ↳ *This results in exploding gradient*

ResNet: *Skip Connection*

ResNet uses skip connections to overcome this issue





Further Read

- Goodfellow
  - ↳ Chapter 9 **_CNNs_**

CNNs are further discussed in details in

- *ECE1508: Applied Deep Learning*
  - ↳ *Given in both Fall and Winter Semesters*

Sequence Data: *Many Applications*

"This is the first lecture on ..."

Classic/Jazz/Pop/Hip-hop

"This product is useless!"

A cute puppy in snow

Sport/Drama/Documentary

Sequence Learning Problem

Basic FNNs cannot be used in practice: *we need a huge input and/or output*

Sequence Learning Problem

Basic FNNs cannot be used in practice: *we need a huge input and/or output*

Sequence Learning Model

Models that get sequence inputs and return sequence outputs

Sequence Learning Problem

Basic FNNs cannot be used in practice: *we need a huge input and/or output*

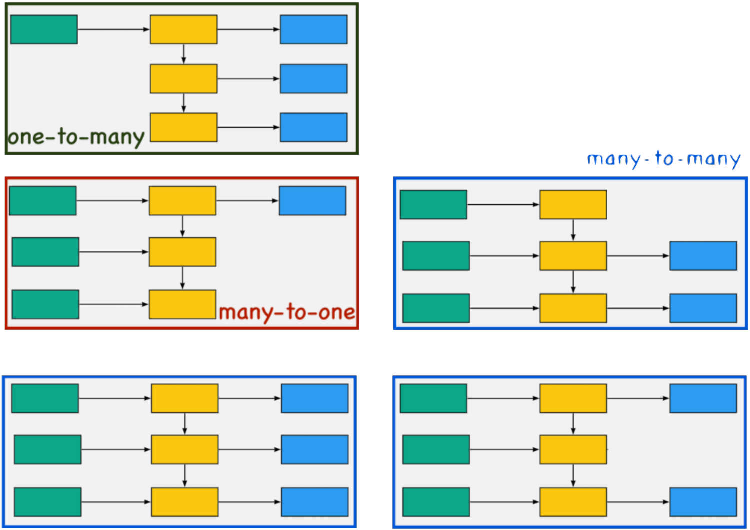
Sequence Learning Model

Models that get sequence inputs and return sequence outputs

There are various approaches in the literature

- *Recurrent Neural Networks (RNNs)*
- *Encoder-Decoder Architecture (Seq2Seq Models)*
- *Transformers*

Types of Sequence Problems

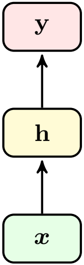


Recurrent Neural Networks

We can feed a sequence to a neural network, if we include recurrence

Recurrence

NN takes a state as input and returns the next state as output

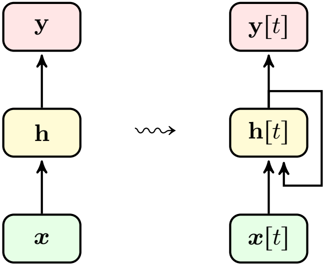


Recurrent Neural Networks

We can feed a sequence to a neural network, if we include recurrence

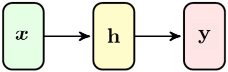
Recurrence

NN takes a state as input and returns the next state as output



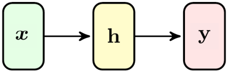
Example: *Turning Shallow FNN to Basic RNN*

Say we have a shallow fully-connected FNN



Example: *Turning Shallow FNN to Basic RNN*

Say we have a shallow fully-connected FNN

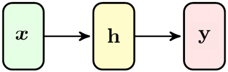


- *It computes hidden feature from the input* $\boldsymbol{x}$

$$\mathbf{h} = f\left(\mathbf{W}_1 \boldsymbol{x}\right)$$

Example: *Turning Shallow FNN to Basic RNN*

Say we have a shallow fully-connected FNN



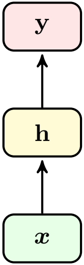
- *It computes hidden feature from the input* $\boldsymbol{x}$

$$\mathbf{h} = f\left(\mathbf{W}_1 \boldsymbol{x}\right)$$

- *It computes the output from the hidden features*

$$\mathbf{y} = f\left(\mathbf{W}_2 \mathbf{h}\right)$$

Example: *Turning Shallow FNN to Basic RNN*



The model in this case gives us

$$\mathbf{y} \propto P(v|\boldsymbol{x})$$

and when we train it, we maximize its likelihood

Example: *Turning Shallow FNN to Basic RNN*

We can make an RNN by using the previous feature in each time: at time t

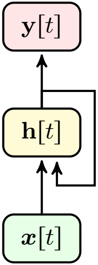
- *compute new features from the input* $\boldsymbol{x}[t]$ *and previous features* $\mathbf{h}[t-1]$

$$\mathbf{h}[t] = f\left(\mathbf{W}_1 \boldsymbol{x}[t] + \mathbf{W}_0 \mathbf{h}[t-1]\right)$$

- *computes the output from the new features*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$$

Example: *Turning Shallow FNN to Basic RNN*

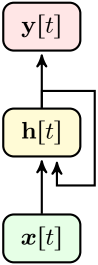


This recursive model determines for us

$$\mathbf{y}[t] \propto P\left(v[t] | \mathbf{h}[t-1], \boldsymbol{x}[t]\right)$$

and we should maximize the likelihood over time

Example: *Turning Shallow FNN to Basic RNN*



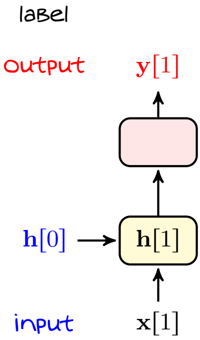
This recursive model determines for us

$$\mathbf{y}[t] \propto P\left(v[t] | \mathbf{h}[t-1], \boldsymbol{x}[t]\right) \equiv P\left(v[t] | \boldsymbol{x}[1], \ldots, \boldsymbol{x}[t]\right)$$

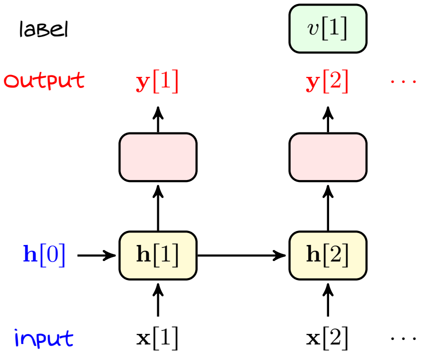
and we should maximize the likelihood over time

↳ *information about* $\boldsymbol{x}[1], \ldots, \boldsymbol{x}[t-1]$ *is somehow encoded in* $\mathbf{h}[t-1]$

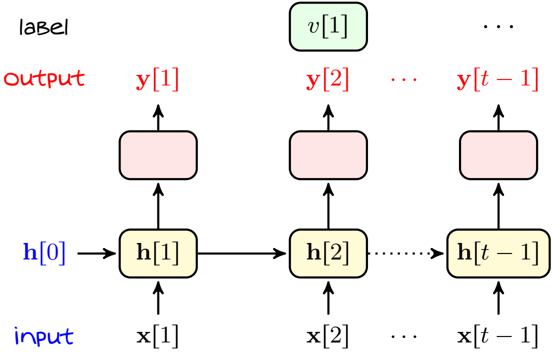
Example: *Turning Shallow FNN to Basic RNN*



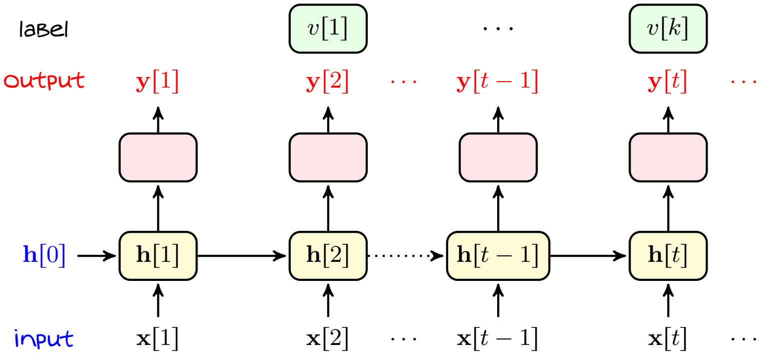
Example: *Turning Shallow FNN to Basic RNN*



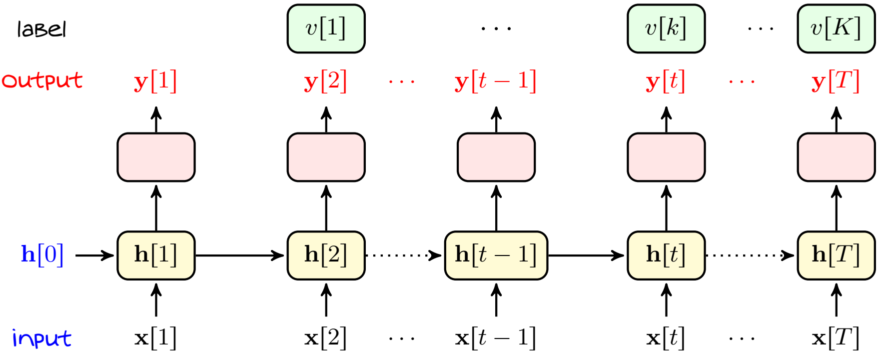
Example: *Turning Shallow FNN to Basic RNN*



Example: *Turning Shallow FNN to Basic RNN*

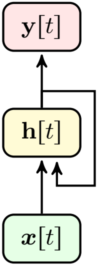


Example: *Turning Shallow FNN to Basic RNN*



Generic RNN

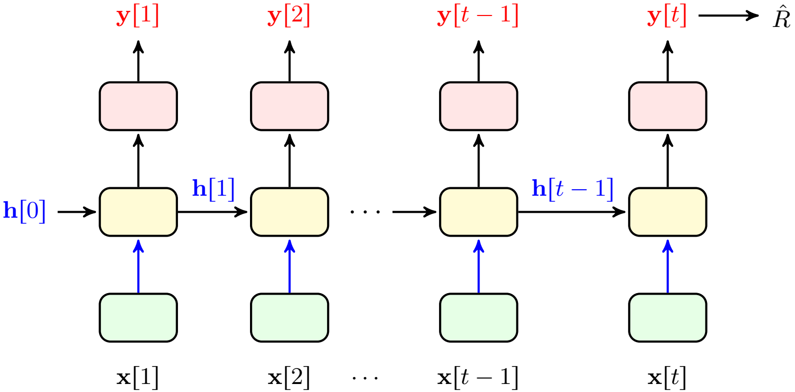
We can do the same thing with any FNN, including CNNs



$\mathbf{h}[t]$ *is the output of a ny number of hidden layers*

Training RNNs

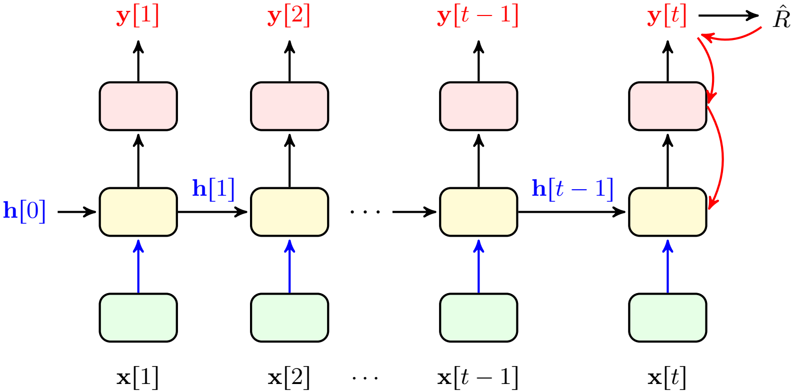
We can apply the same risk minimization; this time over time



To train RNNs, we need to backpropagate through time

Training RNNs

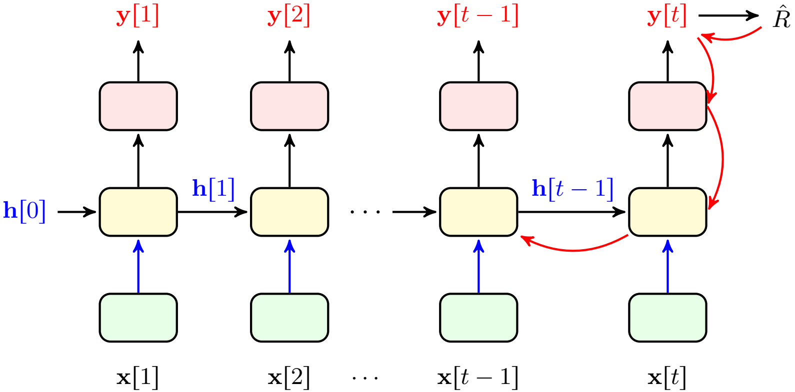
We can apply the same risk minimization; this time over time



To train RNNs, we need to backpropagate through time

Training RNNs

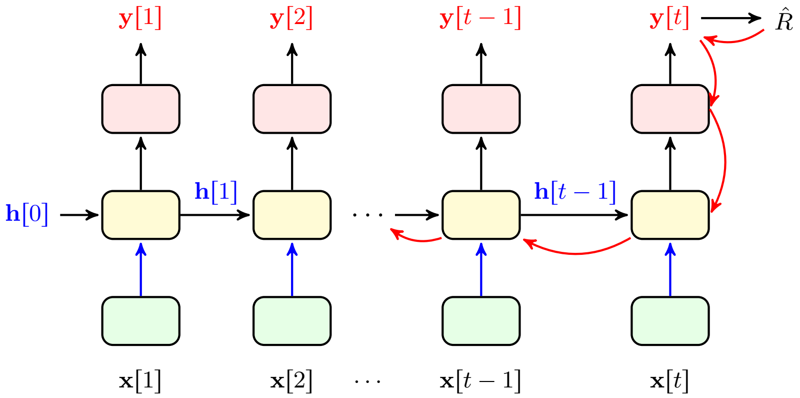
We can apply the same risk minimization; this time over time



To train RNNs, we need to backpropagate through time

Training RNNs

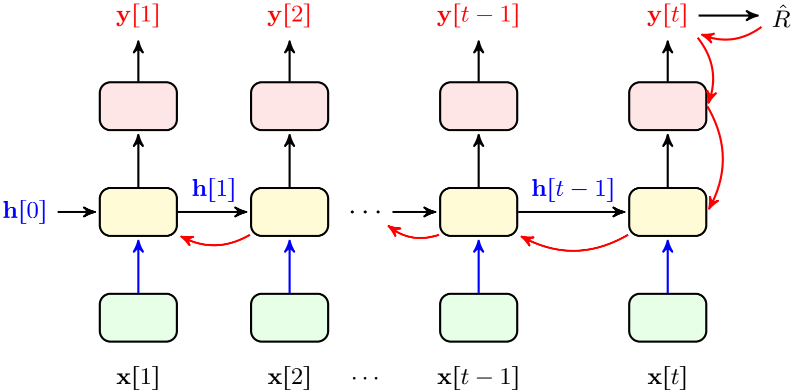
We can apply the same risk minimization; this time over time



To train RNNs, we need to backpropagate through time

Training RNNs

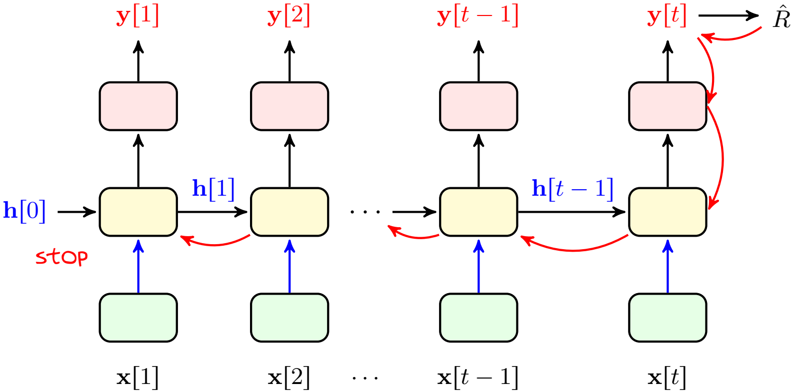
We can apply the same risk minimization; this time over time



To train RNNs, we need to backpropagate through time

Training RNNs

We can apply the same risk minimization; this time over time



To train RNNs, we need to backpropagate through time

Main Challenge: *Vanishing Gradient Through Time*

With long sequences

backpropagation through time $\approx$ *backpropagation through deep* NNs

- *Exploding Gradients*
- *Vanishing Gradients*

Main Challenge: *Vanishing Gradient Through Time*

With long sequences

backpropagation through time $\approx$ *backpropagation through deep* NNs

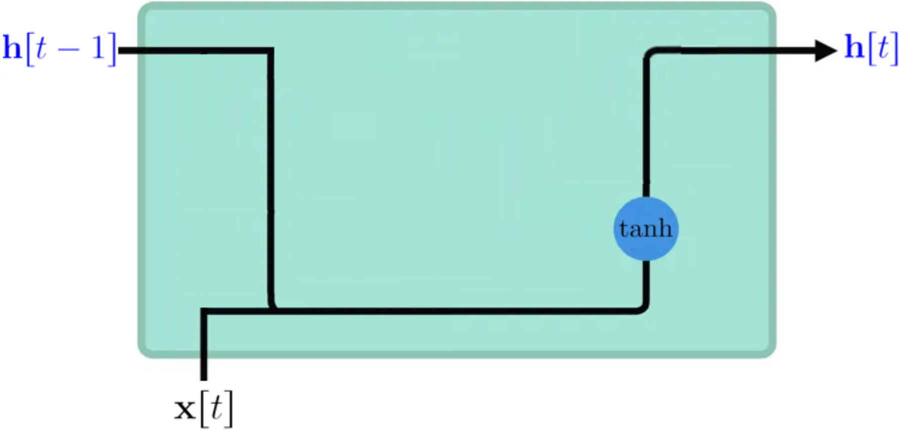
- *Exploding Gradients*
- *Vanishing Gradients*
  - ↳ *Gradients become small over time*
  - ↳ *Weights of RNN are updated only with most recent time instances*
  - ↳ *RNN has a limited memory!*

Classical Remedies

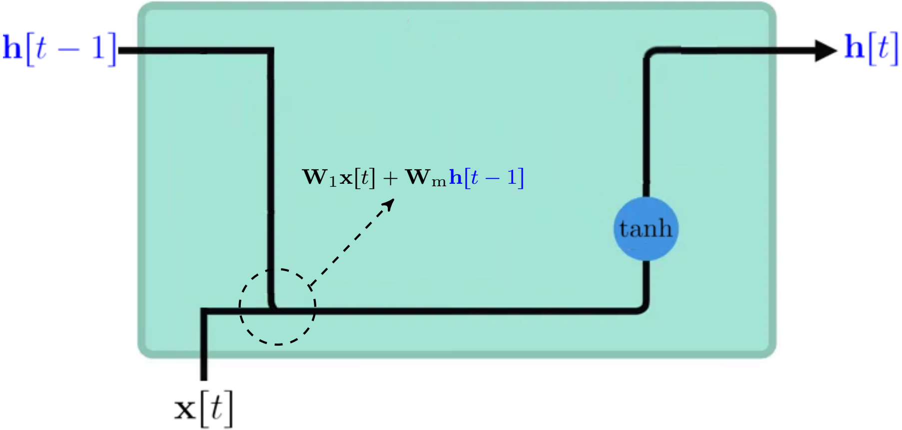
There are various approaches to handle these issues

- *Clipping gradients when exploding*
  - ↳ *Usually used to deal with exploding gradient*
- *Truncated backpropagation through time*
  - ↳ *Updated multiple times in the middle to carry memory forward*
- *Using gated units*
  - ↳ *Use the so-called gate to control better the memory*

Basic RNN as a Unit

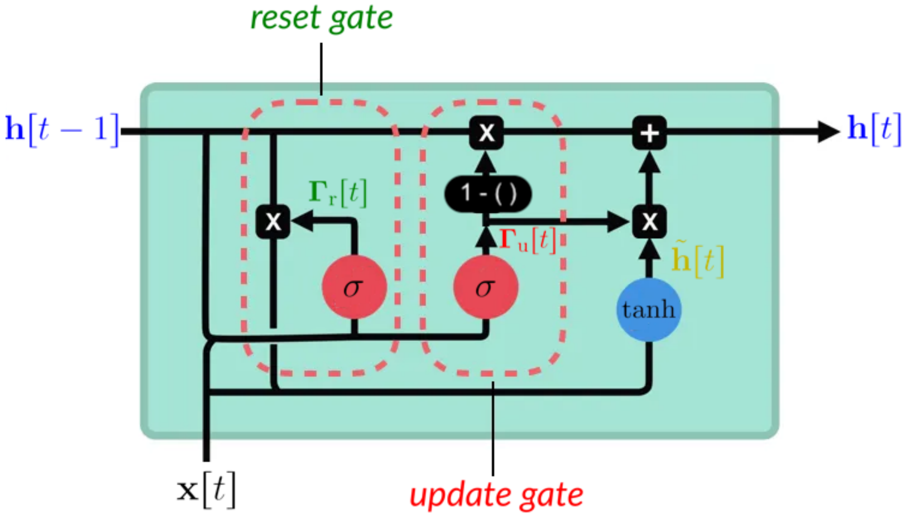


Basic RNN as a Unit



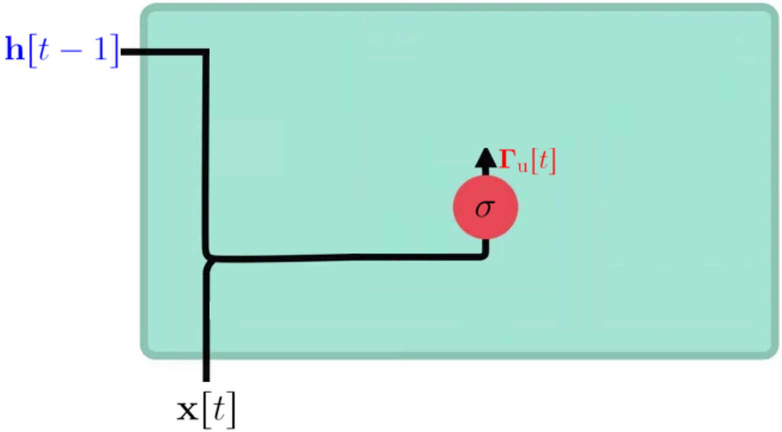
Gated Recurrent Unit: *GRU*

This is what's going on in a GRU cell



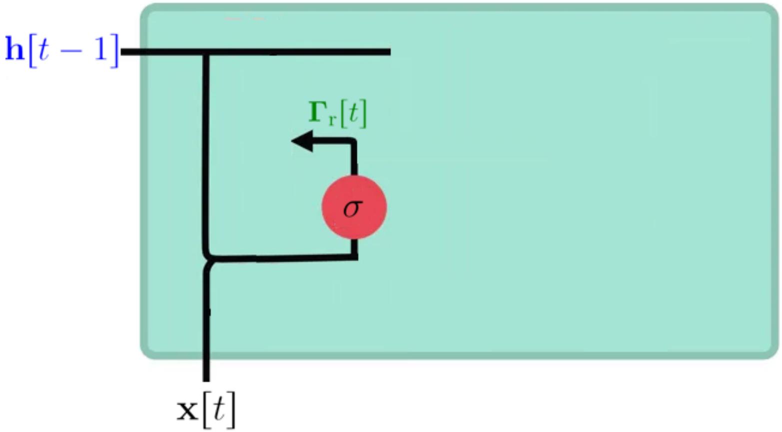
Gated Recurrent Unit: *GRU*

Compute update gate $\boldsymbol{\Gamma}_{\mathrm{u}}[t] = \sigma\left(\mathbf{W}_{\mathrm{u,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{u,m}}\mathbf{h}[t-1]\right)$



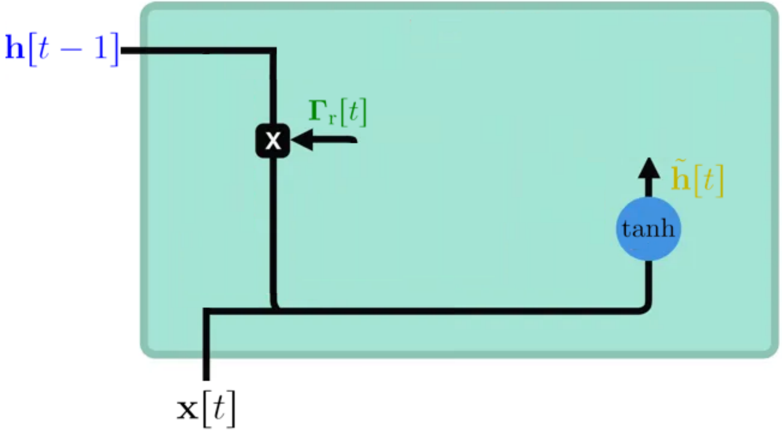
Gated Recurrent Unit: *GRU*

Compute reset gate $\mathbf{\Gamma}_{\rm r}[t] = \sigma\left(\mathbf{W}_{\rm r,in}\mathbf{x}[t] + \mathbf{W}_{\rm r,m}\mathbf{h}[t-1]\right)$



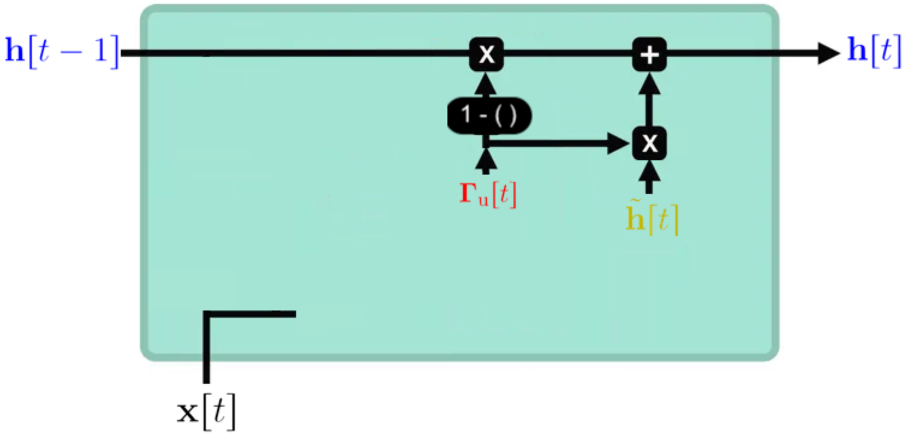
Gated Recurrent Unit: *GRU*

Compute actual memory $\tilde{\mathbf{h}}[t] = f(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_\mathrm{m}\boldsymbol{\Gamma}_\mathrm{r}[t] \odot \mathbf{h}[t-1])$



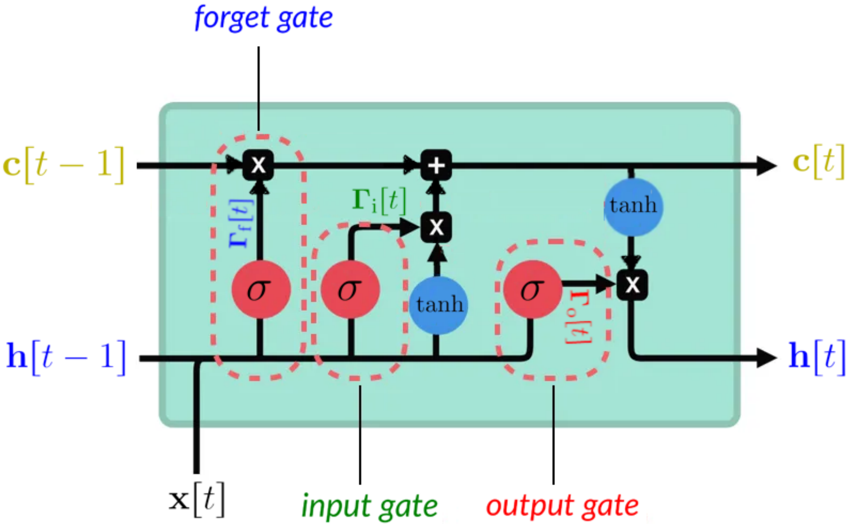
Gated Recurrent Unit: *GRU*

Update hidden state as $\mathbf{h}[t] = (1 - \mathbf{\Gamma}_{\mathrm{u}}[t]) \odot \mathbf{h}[t-1] + \mathbf{\Gamma}_{\mathrm{u}}[t] \odot \tilde{\mathbf{h}}[t]$



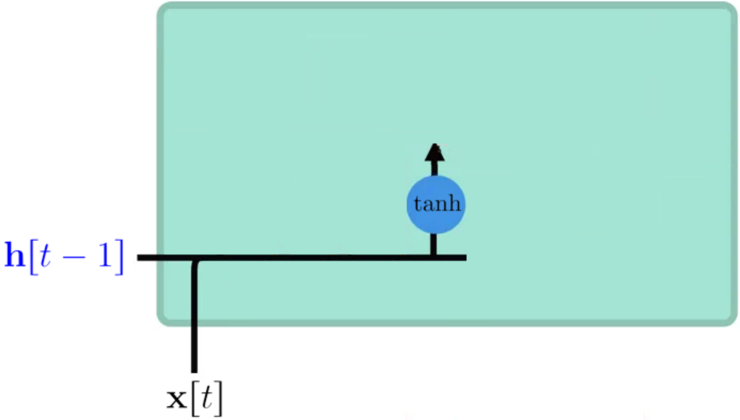
Long Short-Term Memory: *LSTM*

This is how inside an LSTM unit looks like



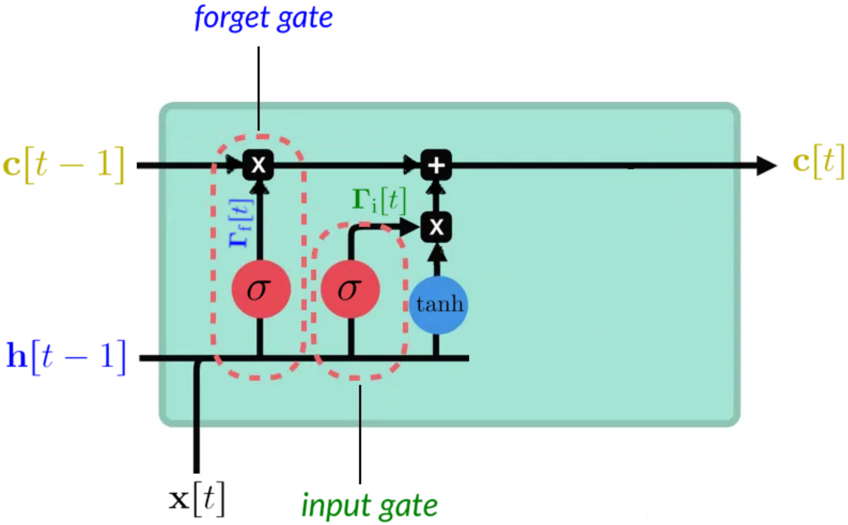
Long Short-Term Memory: *LSTM*

Actual cell state $\tilde{c}[t] = f(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_{\mathrm{m}} \mathbf{h}[t-1])$



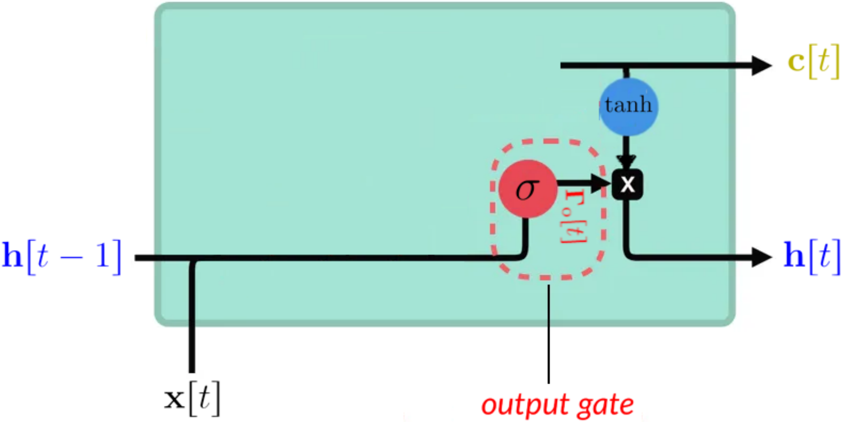
Long Short-Term Memory: *LSTM*

We use forget gate and update gate to update cell state



Long Short-Term Memory: *LSTM*

We use output gate to control fellow of memory to the hidden state



Gating is Helpful

- *Using gating we can control vanishing and exploding gradient*

Gating is Helpful

- *Using gating we can control vanishing and exploding gradient*
- *LSTM used to be a robust sequence-based model*
  - ↳ *We can still use it for simple tasks like price prediction*
  - ↳ *It was one of the first models used for text generation*

Gating is Helpful

- *Using gating we can control vanishing and exploding gradient*
- *LSTM used to be a robust sequence-based model*
  - ↳ *We can still use it for simple tasks like price prediction*
  - ↳ *It was one of the first models used for text generation*
- *At the end of the day, RNNs will still carry limited memory*
  - ↳ *This is why Attention Mechanism is developed*
  - ↳ *We can do whole sequence processing based on Attention*
  - ↳ *This is the idea of Transformers*

Further Read

- Goodfellow
  - ↳ Chapter 10 ***RNNs***

RNNs and Transformers are discussed in

- *ECE1508: Applied Deep Learning*
  - ↳ *Given in both Fall and Winter Semesters*
- *ECE1786: Creative Applications of NLP*
  - ↳ *Given in Fall Semesters*