

Introduction to Machine Learning

Lecture 17: Regularization and Convolution

Ali Bereyhi

`ali.bereyhi@utoronto.ca`

Department of Electrical and Computer Engineering
University of Toronto

Winter 2026

Today's Agenda: *Generalization and Convolution*

Today, we talk about

Generalization

In this way, we discuss the following topics

- *Generalization concept and bias-variance trade-off*
- *Memory-capacity trade-off and overfitting*
- *Techniques to reduce overfitting*

Reducing Overfitting

Most basic approach is to tune hyperparameters

Hyperparameter

Any parameter we set to realize our model, e.g., depth, width, etc

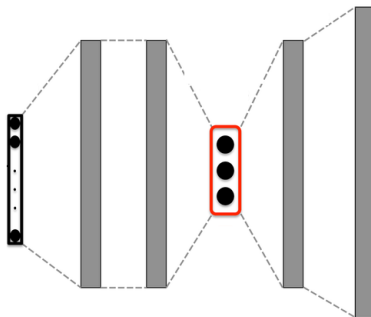
Start with initial setting

- *Train the model and save the training loss*
- *Evaluate it on the validation dataset*
 - ↳ *If the loss is too different from training loss*
 - ↳ *change hyperparameters and re-train*
 - ↳ *Else*
 - ↳ *training is over*

? *How can we search?*

! *Depending on the complexity: random or grid search*

Bottleneck



Bottleneck can drop the model capacity

- Say we add a 10-neuron layer between two 200-neuron layers
 - ↳ Without bottleneck we have 40,000 weights
 - ↳ With bottleneck we have 4,000 weights!

Regularization: *General Form*

In regularization, we add a penalty to empirical risk: *training minimizes*

$$\tilde{R}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \mathcal{L}(y_n, v_n) + \lambda \Pi(\mathbf{w})$$

Penalty is proportional to the property of weights when model overfits

Example

If we know that weights get large when model overfits we could set

$$\Pi(\mathbf{w}) = \|\mathbf{w}\|^2 = \sum_i w_i^2$$

- We typically pre-set the regularization function, i.e., penalty
- λ is a hyperparameter that we call *regularizer* and tune by validation

Regularization: ℓ_p -Norm

Famous choice of penalty is ℓ_p -norm: we set

$$\Pi(\mathbf{w}) = \|\mathbf{w}\|_p^p = \sum_i |w_i|^p$$

- ℓ_2 -norm is the classic choice $\rightsquigarrow$ Tikhonov
 - ↳ It avoids weights to get very large
 - ↳ You did it in Assignment 2 for regression $\rightsquigarrow$ Ridge regression
- ℓ_1 -norm is another classic choice $\rightsquigarrow$ Lasso
 - ↳ It enforces the weights to be sparse $\rightsquigarrow$ contain too many zeros
 - ↳ Great piece of work by Robert Tibshirani 😊

Regularization and Weight Decay

Let's set $\Pi(\mathbf{w}) = \|\mathbf{w}\|^2/2$ and write GD

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \eta \nabla \tilde{R}(\mathbf{w}) \\ &\leftarrow \mathbf{w} - \eta \nabla \left(\frac{1}{N} \sum_{n=1}^N \mathcal{L}(y_n, v_n) + \frac{\lambda}{2} \|\mathbf{w}\|^2 \right)\end{aligned}$$

Now focus on a single w_i : it would be like a **weight decay**

$$\begin{aligned}w_i &\leftarrow w_i - \eta \left(\frac{\partial}{\partial w_i} \hat{R}(\mathbf{w}) + \frac{\lambda}{2} \frac{\partial}{\partial w_i} \|\mathbf{w}\|^2 \right) \\ &\leftarrow w_i - \eta \lambda w_i - \eta \frac{\partial}{\partial w_i} \hat{R}(\mathbf{w}) = w_i (1 - \eta \lambda) - \eta \frac{\partial}{\partial w_i} \hat{R}(\mathbf{w})\end{aligned}$$

Some people call ℓ_2 -norm regularization weight decay

Stochastic Regularization

We can also apply stochastic regularization: *stating training with SGD*

- *In each gradient computation*
 - ↳ *Apply a stochastic operation on neuron outputs*
- *Average all sample gradients over mini-batch*
- *Apply an iteration of GD*

The most famous approach is *dropout*

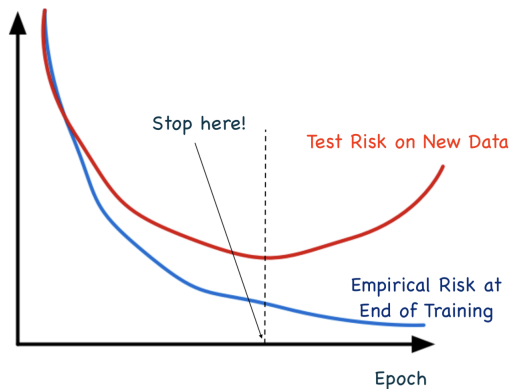
- *In each gradient computation*
 - ↳ *we randomly drop some neurons*

$$y_i \leftarrow m_i y_i$$

where $m_i \in \{0, 1\}$ is a random mask

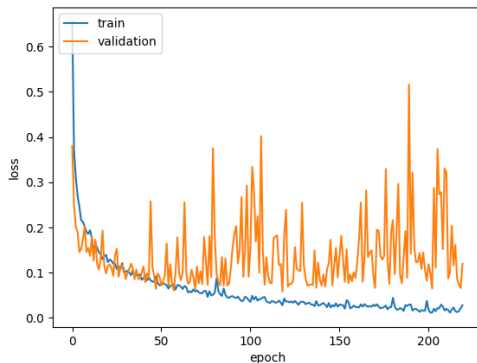
Early Stopping

We may stop when we the model is well-trained: *validate after each epoch*



Early Stopping

In practice though it's hard: *we usually see*



We may use moving average; however, it's not always effective

Ensemble

We could use multiple model: *give the task to m model*

- *Each model predicts a label y_i*
- *We set the average as the final prediction*

$$y = \frac{1}{m} \sum_{i=1}^m y_i$$

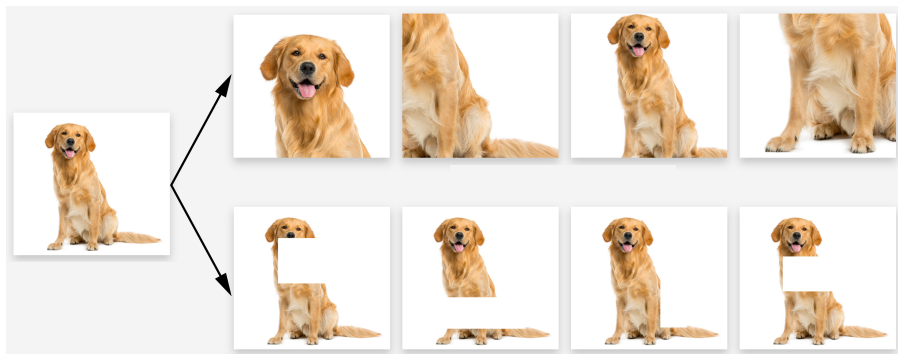
? *How to realize multiple models?*

We can take different approaches

- *Bagging $\rightsquigarrow$ use the same model on different data*
- *Stacking $\rightsquigarrow$ use different models on the same data*
- *Boosting $\rightsquigarrow$ modify the data based on previous predictions and re-train*

Data Augmentation

We can augment to increase samples



Further Read

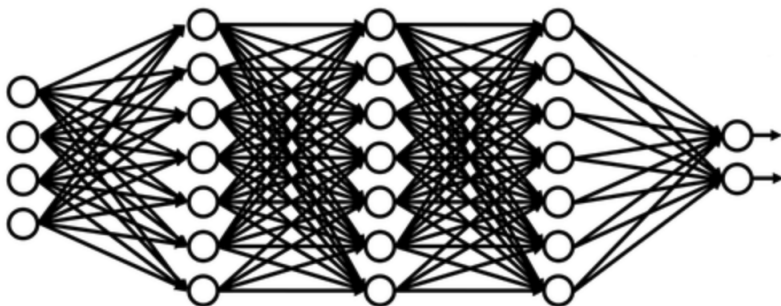
- Goodfellow
 - ↳ Chapter 6

Regularization

Challenges of Fully-Connected Layers

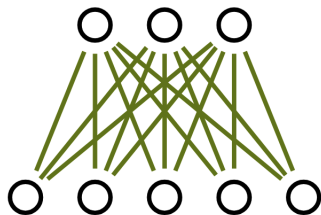
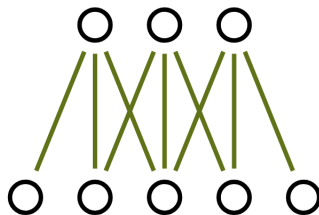
In fully-connected layers we have too much weights

- *Even for a simple task, we end up with high dimensional w*
- *Many of these w do nothing!*



Need for Sparse Linear Operation

We would benefit if we *drop some weights*!



We need a *sparse*-weighted linear operation

Mathematical Convolution: 1D Signals

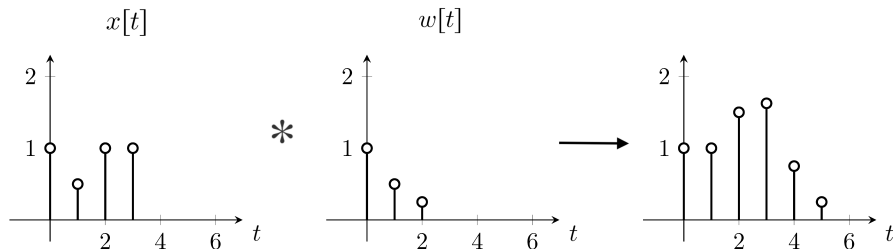
Convolution

For two 1D signals $x[t]$ and $w[t]$, the convolution is defined as

$$(x * w)[t] = \sum_{i=-\infty}^{\infty} x[i]w[t-i]$$

We typically deal with finite signals: *we assume signal is zero anywhere else*

Convolution: *Example*



$$\dots \quad (x * w)[0] = \sum_{i=-\infty}^{\infty} x[i]w[-i]$$

$$(x * w)[1] = \sum_{i=-\infty}^{\infty} x[i]w[1-i] \quad \dots$$

Convolution: *Sliding Flipped Filter*

We can look at convolution as *sliding a flipped filter over input*

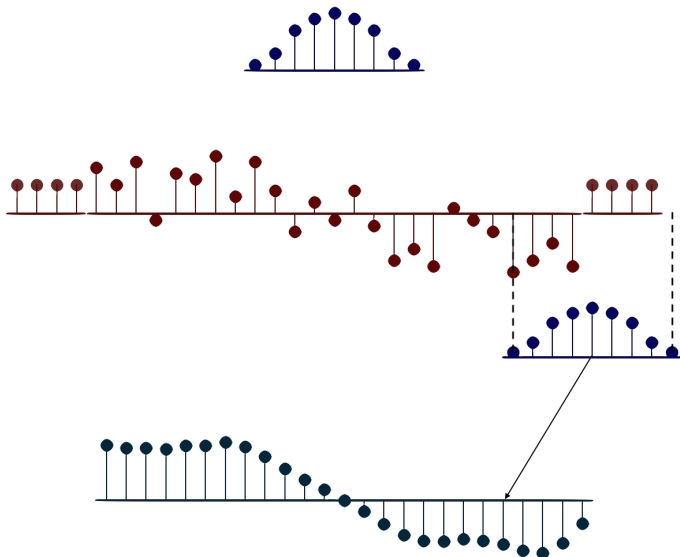
$$(x * w)[t] = \sum_{i=-\infty}^{\infty} x[i]w[t-i]$$

Say $\hat{w}[-t] = w[t]$; then, we have

$$(x * w)[t] = \sum_{i=-\infty}^{\infty} x[i]\hat{w}[i-t]$$

This is like sliding $\hat{w}[t]$ over $x[t]$ and averaging

Convolution: *Sliding Flipped Filter*



Mathematical Convolution: 2D Signals

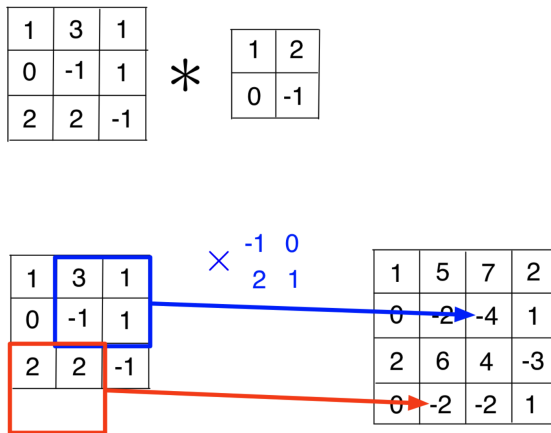
Convolution

For two 2D signals $\mathbf{X}[t]$ and $\mathbf{W}[t]$, the convolution is defined as

$$(\mathbf{X} * \mathbf{W})[t, s] = \sum_{i, j=-\infty}^{\infty} \mathbf{X}[i, j] \mathbf{W}[t - i, s - j]$$

Convolution: *Sliding Flipped Filter*

We can again show that we are *sliding the 2D filter after flipping it*

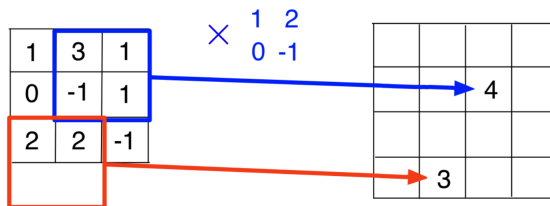


Cross-Correlation: *Convolution in Practice*

Cross-correlation $\equiv$ Convolution without Flipping

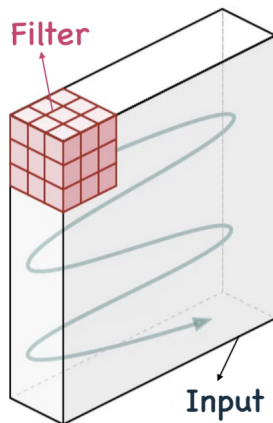
For two 2D signals $\mathbf{X}[t]$ and $\mathbf{W}[t]$, the cross-correlation is

$$(\mathbf{X} * \mathbf{W})[t, s] = \sum_{i, j=-\infty}^{\infty} \mathbf{X}[i, j] \mathbf{W}[i + t, j + s]$$

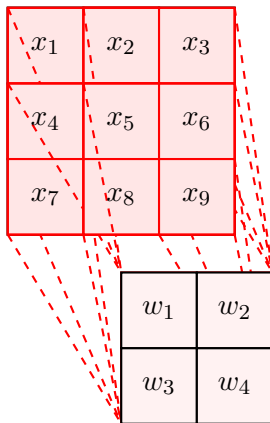
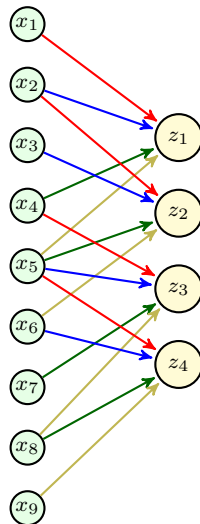


Convolution with 3D Inputs

Convolution can be also done on 3D objects: *we slide 3D filters*

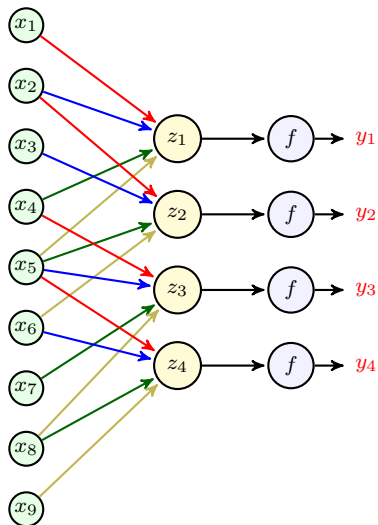


Convolution $\equiv$ *Sparse Linear Operation with Shared Weights*

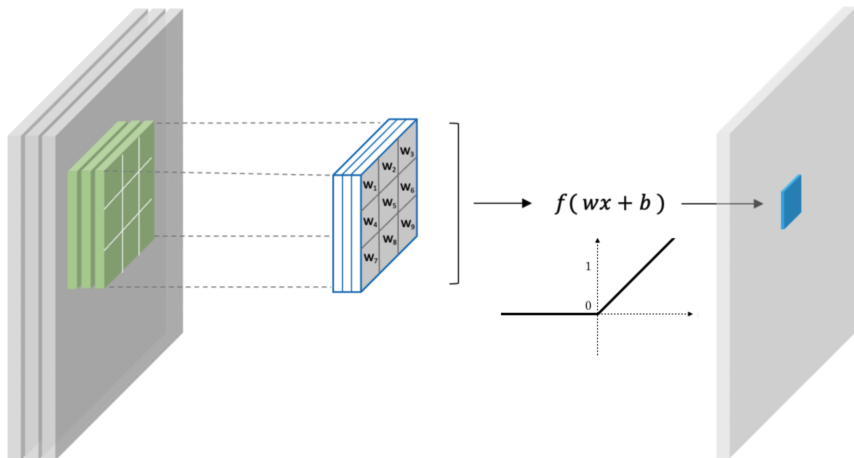


Building Neural Layers with Convolution

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

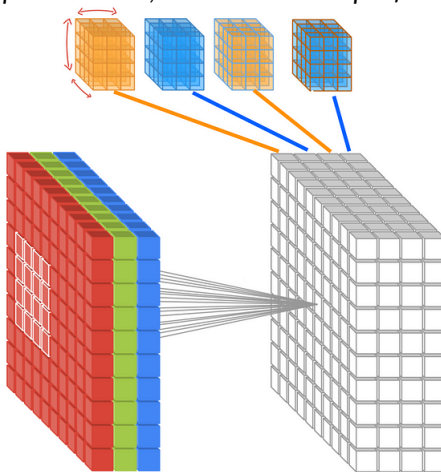


Convolutional Layer: *Single-Channel Output*



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

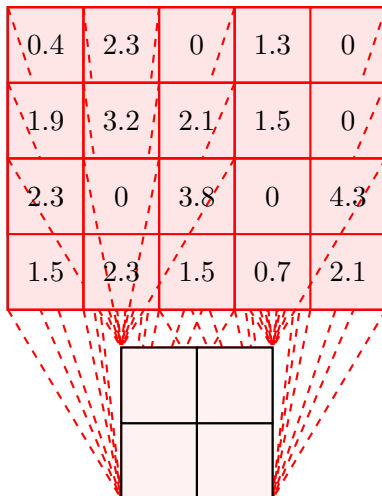
Example: Max-Pooling

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

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



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

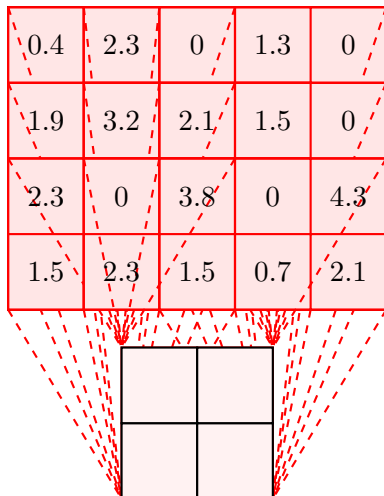
$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & \dots & \dots & X_{1,M} \\ X_{2,1} & X_{2,2} & \dots & \dots & X_{2,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ X_{N,1} & X_{N,2} & \dots & \dots & X_{N,M} \end{bmatrix}$$

mean { }

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

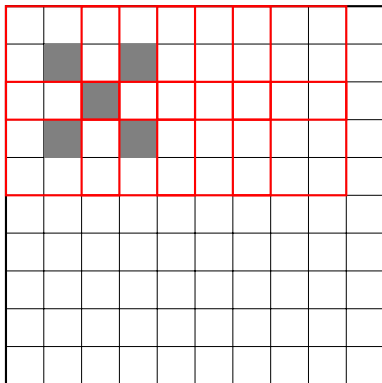


2.05	1.9	1.225	0.7
1.85	2.275	1.85	1.45
1.525	1.9	1.5	1.775

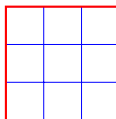
Convolution with Stride

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

input



filter

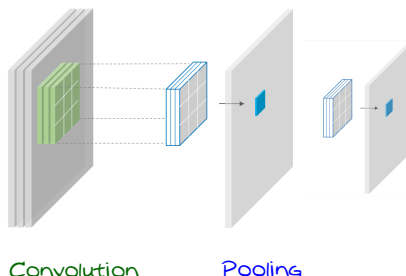


output



Convolutional Unit: Complex Convolutional Layer

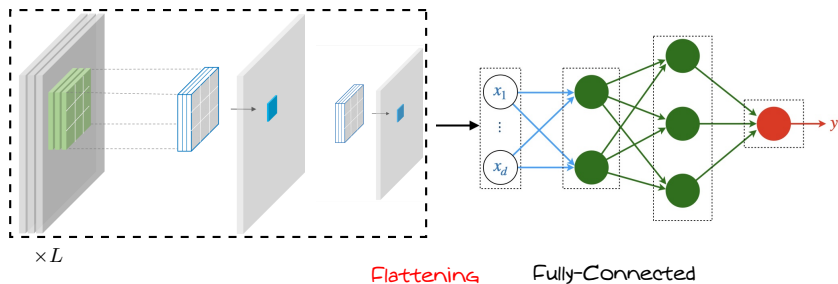
A convolutional unit is made as



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

CNN: General Architecture

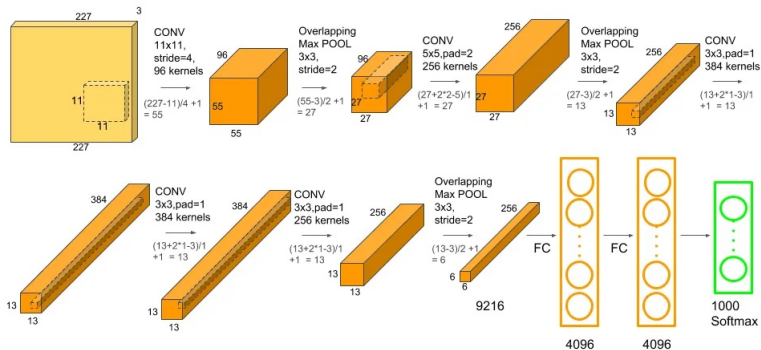
A CNN consists of multiple convolutional units and a fully-connected NN



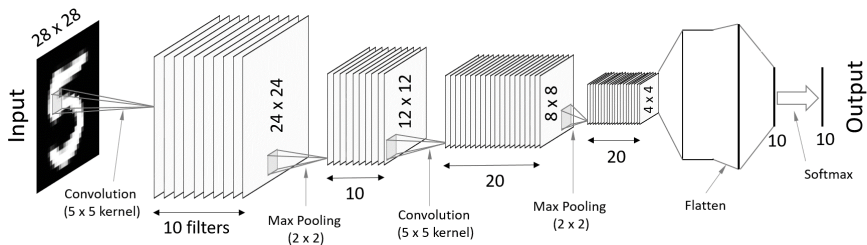
We can repeat the convolutional unit multiple times

Example: AlexNet

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



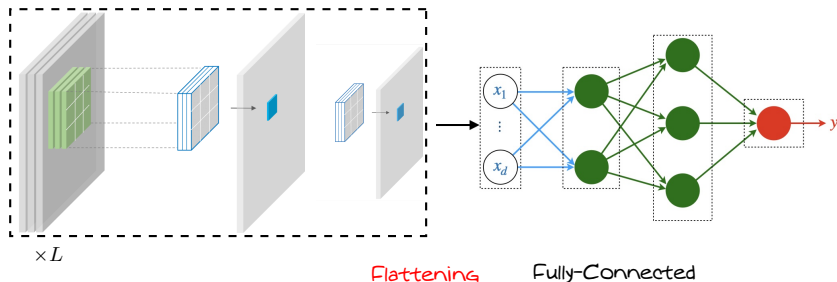
Example: Custom CNN for MNIST Classification



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

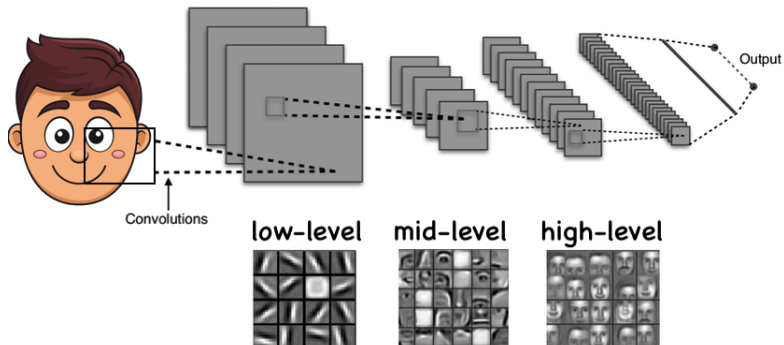
Backpropagation



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

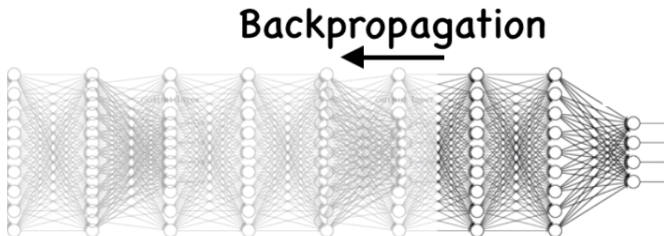
- We can compute sample gradients with backpropagation
- Using mini-batch SGD, we can efficiently train CNNs

Gradual Feature Extraction



As we go deeper, CNN extracts higher levels of features

Vanishing or Exploding Gradient

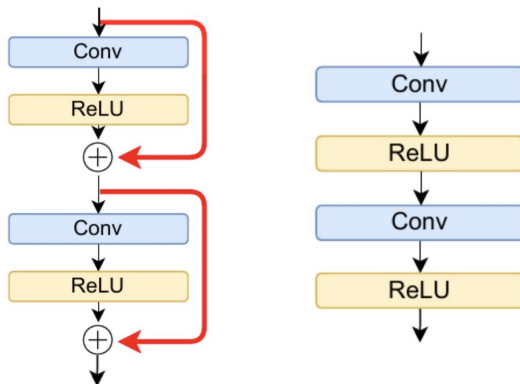


As we go very deep in FNNs, we could experience

- 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*