DEEP LEARNING FOR AI

Object recognition

IDEA: use a sample (the chair) as TEMPLATE, and proceed by a point to point correlation between the template and the image ==> Highest correlation = MATCHING!

Challenges

- Variable viewpoint
- Variable illumination
- Scale
- Deformation
- Occlusion
- Background clutter
- Intra-class variability

Why not using SIFT (local descriptor of key-points) matching for everything?

- Work well for object instances
- Difficulties when the problem get harder

→ need of "some" artificial intelligence!

Deep learning:

• has revolutionized many areas of machine intelligence, with particular impact **on image understanding tasks**
• particularly effective...
– for unstructured data
– to learn good representations
– to learn good "models

❖ Image classification
❖ Detect and segment objects
❖ Style transfer
❖ Synthesize faces
❖ Image captioning

What is intelligence?

Big open question, but for our purposes ... the ability to process information, to inform future decisions

Artificial intelligence : any techniques that enables computers to mimic human behaviour

- **Machine learning**: ability to learn without explicitly being programmed
  - **Deep learning** : extract patterns from data using neural networks

Why Deep Learning?

Traditional ML:

• Hand-engineered features
• LIMITS and PROBLEMS:

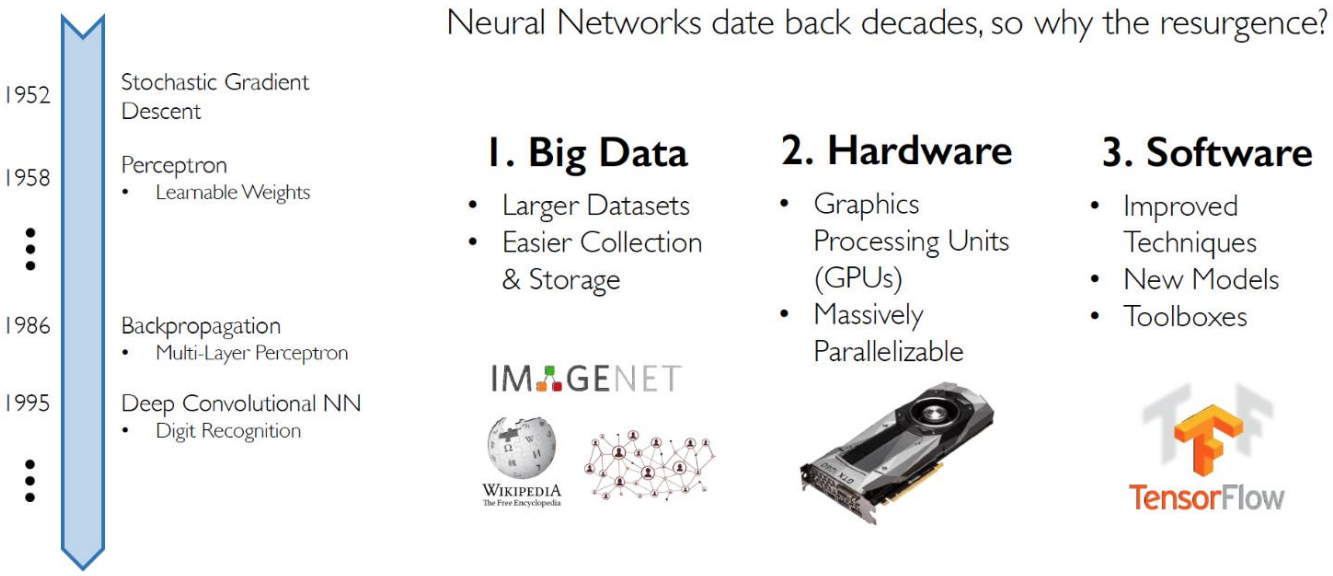
- Time consuming
- Not robust
- Not scalable

Challenge:

Can we learn the underlying features directly from data? Deep Learning :

- learns features directly from data
- learns features in a hierarchical manner

HOW CAN WE GO FROM RAW DATA (e.g. Pixels) TO A MORE AND MORE COMPLEX REPRESENTATION AS THE DATA FLOWS THROUGH THE MODEL?



→ **Preliminary operations on Colab**
→ **Python review and extras**
→ **PyTorch introduction**

AUTOMATIC DIFFERENTIATION

Derivatives for vectors

With slight abuse of notation by ∂:

		Scalar	Vector
		x	$\mathbf{x}$
Scalar	y	$\frac{\partial y}{\partial x}$	$\frac{\partial y}{\partial \mathbf{x}}$
Vector	$\mathbf{y}$	$\frac{\partial \mathbf{y}}{\partial x}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$

$\left[\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right]_{ij} = \frac{\partial y_i}{\partial x_j}$ "numerator layout" notation: # of rows = # of entries in y

# of columns = # entries in x

$\rightarrow \frac{\partial y}{\partial \mathbf{x}} = \nabla y(\mathbf{x})$ is a row vector

y	a	au	$sum(\mathbf{x})$	$\| \mathbf{x} \|^2$	$u+v$	uv	$\langle \mathbf{u}, \mathbf{v} \rangle$
$\frac{\partial y}{\partial \mathbf{x}}$	$\mathbf{0}^T$	$a\frac{\partial u}{\partial \mathbf{x}}$	$\mathbf{1}^T$	$2\mathbf{x}^T$	$\frac{\partial u}{\partial \mathbf{x}} + \frac{\partial v}{\partial \mathbf{x}}$	$\frac{\partial u}{\partial \mathbf{x}} v + \frac{\partial v}{\partial \mathbf{x}} u$	$\mathbf{u}^T \frac{\partial \mathbf{v}}{\partial \mathbf{x}} + \mathbf{v}^T \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$

($\mathbf{0}$ and $\mathbf{1}$ are vectors)

Chain rule

Scalars : $y = f(u), u = g(x) \rightarrow \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$

Vectors : $\underset{(1,n)}{\frac{\partial y}{\partial \mathbf{x}}} = \underset{(1)}{\frac{\partial y}{\partial u}}\underset{(1,n)}{\frac{\partial u}{\partial \mathbf{x}}}$ $\underset{(1,n)}{\frac{\partial y}{\partial \mathbf{x}}} = \underset{(1,k)}{\frac{\partial y}{\partial \mathbf{u}}}\underset{(k,n)}{\frac{\partial \mathbf{u}}{\partial \mathbf{x}}}$ $\underset{(m,n)}{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}} = \underset{(m,k)}{\frac{\partial \mathbf{y}}{\partial \mathbf{u}}}\underset{(k,n)}{\frac{\partial \mathbf{u}}{\partial \mathbf{x}}}$

Example 1

Assume $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$, $y \in \mathbb{R}$ and $z = (\langle \mathbf{x}, \mathbf{w} \rangle - y)^2$

Compute $\frac{\partial z}{\partial \mathbf{w}} \rightarrow \frac{\partial z}{\partial \mathbf{x}} = \frac{\partial z}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial \mathbf{w}} = \frac{\partial b^2}{\partial b}\frac{\partial a - y}{\partial a}\frac{\partial \langle \mathbf{x}, \mathbf{w} \rangle}{\partial \mathbf{w}} = 2b \cdot 1 \cdot \mathbf{x}^T = 2(\langle \mathbf{x}, \mathbf{w} \rangle - y)\mathbf{x}^T$

Decompose $a = \langle \mathbf{x}, \mathbf{w} \rangle$, $b = a - y$, $z = b^2$

Example 2

Assume $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{w} \in \mathbb{R}^n$, $y \in \mathbb{R}$ and $z = \|\mathbf{X}\mathbf{w} - y\|^2$

Compute $\frac{\partial z}{\partial \mathbf{w}} \rightarrow \frac{\partial z}{\partial \mathbf{x}} = \frac{\partial z}{\partial \mathbf{b}}\frac{\partial \mathbf{b}}{\partial \mathbf{a}}\frac{\partial \mathbf{a}}{\partial \mathbf{w}} = \frac{\partial \|\mathbf{b}\|^2}{\partial \mathbf{b}}\frac{\partial \mathbf{a} - \mathbf{y}}{\partial \mathbf{a}}\frac{\partial \mathbf{X}\mathbf{w}}{\partial \mathbf{w}} = 2\mathbf{b}^T \times \mathbf{I} \times \mathbf{X} = 2(\mathbf{X}\mathbf{w} - y)^T\mathbf{X}$

Decompose $\mathbf{a} = \mathbf{X}\mathbf{w}$, $\mathbf{b} = \mathbf{a} - \mathbf{y}$, $z = \|\mathbf{b}\|^2$

Automatic differentiation evaluates gradients of a function specified by a (computer) program at given values

It differs from :

– Symbolic differentiation with some Computer Algebra System (CAS) software, eg. Wolfram Mathematica:

```
In[1] := D[4x^3+x^2+3, x]
Out[1] = 2 x + 12 x^2
```

– Numerical differentiation: $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x-h)}{2h} \simeq \frac{f(x+h)-f(x-h)}{2h}$

It is conceptually based on the simple structure of **computation graph** and the **Chain Rule** in differentiation

- First: Decompose the calculation code into primitive operations
- Then: Build a directed acyclic graph (DAG) to represent the computations
- How it is processed in PyTorch:
  - keeps a record of **tensors & all executed operations** (along with the resulting new tensors) in a DAG: **leaves** are the **input** tensors, **roots** are the **output** tensors
  - by tracing (visiting) this graph from roots to leaves, you can automatically compute the gradients **using the chain rule**

Two modes, using the chain rule:
suppose y (output) depends on x (input) through composition of functions $u_n, u_{n-1}, \ldots, u_1$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u_n}\frac{\partial u_n}{\partial u_{n-1}} \cdots \frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial x}$$

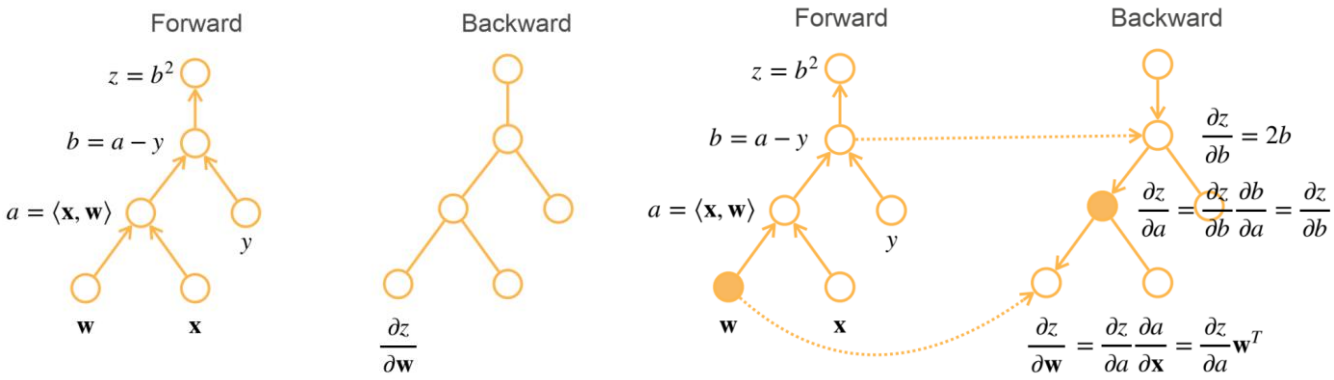
- forward computation (notice backets forcing the order): $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u_n}\left(\frac{\partial u_n}{\partial u_{n-1}}\left(\ldots\left(\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial x}\right)\right)\right)$
- backward computation, also known as **backpropagation**: $\frac{\partial y}{\partial x} = \left(\left(\left(\frac{\partial y}{\partial u_n}\frac{\partial u_n}{\partial u_{n-1}}\right)\ldots\right)\frac{\partial u_2}{\partial u_1}\right)\frac{\partial u_1}{\partial x}$

Using chain rule in forward direction for computing derivatives/gradients is straight-foward, hence not an issue.

Backpropagation

For example: let $z = (\langle \mathbf{x}, \mathbf{w}\rangle - y)^2$. We want : $\frac{\partial z}{\partial \mathbf{w}}$

First, **construct** the DAG decomposing the computations
Second, **evaluate** the DAG forward, storing intermediate results
Then backward computation of the gradient...



Notice: while computing backward we can discard unneeded paths

Automatic differentiation engines in deep learning frameworks are really necessary for making the model training phase effective

Computational complexity: n = #operations

- **Forward** computation of **derivatives**:
  - $O(n)$ time complexity for scalar derivative; $O(nk)$ time complexity for gradient with k variables
  - $O(1)$ space complexity
- **Forward evaluation**: $O(n)$ space complexity, needed to record all intermediate results in the forward visiting
- **Backpropagation**: no additional significant complexity than $O(n)$ of forward computation

autograd

- Pytorch already includes a module named **autograd** (torch.autograd) which is used for automatic differentiation
- **Mechanism**: when willing to compute a certain derivative starting from some data (variable), we need to
  1. declare to Pytorch which floating-point tensors/variables we want to calculate the gradient **with respect to** (by requires_grad=True argument).
     It is said that we attach the gradient data to some tensor/variable, or that the tensor requires a gradient
  2. **compute** the intermediate and final **results**
  3. then **apply the backward** computation of derivatives starting from the final result (the backward() method)
  4. **access** the gradients w.r.t. the declared variables

Eg.: suppose we want to write the computations like

```
a,b,x = 1,2,3
y = a*x
z = y**2 + b
```

which corresponds to $z = z(x) = (ax)^2 + b$, and compute the derivative $\frac{dz}{dx}(x)$ at the preassigned value $x = 3$.

```
a = torch.tensor(1)
b = torch.tensor(2)
x = torch.tensor(3.0, requires_grad=True)
```

Alternative to last statement: initialize tensor `x`, then call `x.requires_grad_(True)`
Requirement: `x` needs to be a floating-point type tensor

```
1. a = torch.tensor(1)
   b = torch.tensor(2)
   x = torch.tensor(3.0, requires_grad=True)
2. y = a*x
   z = y**2 + b
3. z.backward()    # call backward computation
4. x.grad # access the gradient w.r.t. x
```

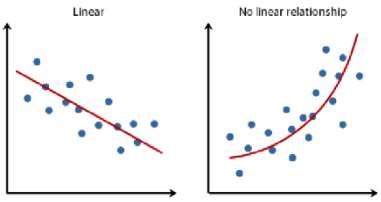
```
[3]  a = torch.tensor(1)
     b = torch.tensor(2)
     x = torch.tensor(3., requires_grad=True)
     y = a*x
     z = y**2 + b
     z.backward()
```

```
[4]  x.grad

     tensor(6.)
```

→ **autograd Notebook**

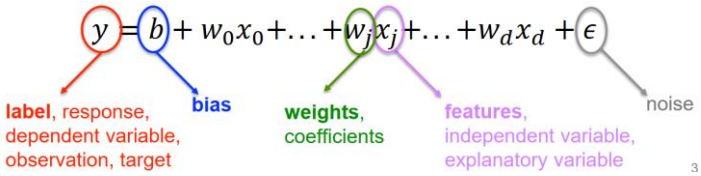
LINEAR NEURAL NETWORKS

Regression methods for modeling the relationship between one or more independent variables and a dependent variable. In machine learning regression methods allow for **prediction**



LINEAR REGRESSION

Assumptions:
– **linear relationship** between independent variables $\mathbf{x}$ (input) and dependent variable y (output):
– **additive noise**: $\sim N(0, \sigma^2)$ distribution



Example

Estimate the prices (dollars) of houses based on their area (feet) and ages (years) applying a linear model

$$price = w_{area} \cdot area + w_{age} \cdot age + b$$

We need a **Training set**: observations of the **sample** characterized by `<area, age; price>`, where area and age are features and price are the labels.

Choose the best weights $\mathbf{w}$ and the bias b s.t., on average, the prediction y made by our model on $\mathbf{x}$ best fit the true prices y observed in the training set $\hat{y} = \mathbf{w}T\mathbf{x}+b$

Given multiple examples $X \in \mathbb{R}^{n\times d}$,we get multiple predictions $\mathbf{y} \in \mathbb{R}^n$: $\hat{\mathbf{y}} = X\mathbf{w}+b$

LOSS FUNCTION

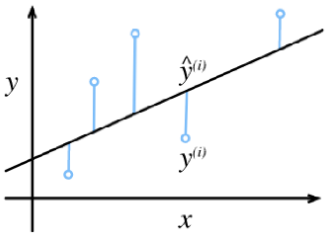
Quantifies the distance between the real $y^{(i)}$ and the predicted $\hat{y}^{(i)}$ value of the target

Squared/quadratic loss

On a single example i: $\ell^{(i)}(\mathbf{w}, b) = \frac{1}{2}(\hat{y}_i - y_i)^2$

On the whole dataset: "Mean Squared Error"

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \ell^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^T x^{(i)} - y^{(i)}\right)^2$$



FIND THE BEST MODEL BY TRAINING

Goal: find the parameter(s) $(\mathbf{w}^*, b^*)$that minimize the total Loss: $(\mathbf{w}^*, b^*) \in argmin_{\mathbf{w},b} L(\mathbf{w}, b)$

Technically sound discussion on optimization:
- existence
- uniqueness

Solution (closed-form solution for linear model)

Augment weights with bias by: $\mathbf{X} \leftarrow [\mathbf{X}, \mathbf{1}]$, $\mathbf{w} \leftarrow \begin{bmatrix}\mathbf{w}\\ b\end{bmatrix}$

The **Loss** is convex quadratic polynomial function: $L(\mathbf{X},\mathbf{y},\mathbf{w}) = \frac{1}{n} \| \mathbf{y} - \mathbf{X}\mathbf{w} \|^2$

Global minimizers via **derivatives** (w.r.t. $\mathbf{w}$) equated to 0: $0 = \frac{\partial}{\partial \mathbf{w}} L(\mathbf{X},\mathbf{y},\mathbf{w}) = \frac{2}{n}(\mathbf{y} - \mathbf{X}\mathbf{w})^T\mathbf{X} \Leftrightarrow (\mathbf{y} - \mathbf{X}\mathbf{w})^T\mathbf{X} = 0$

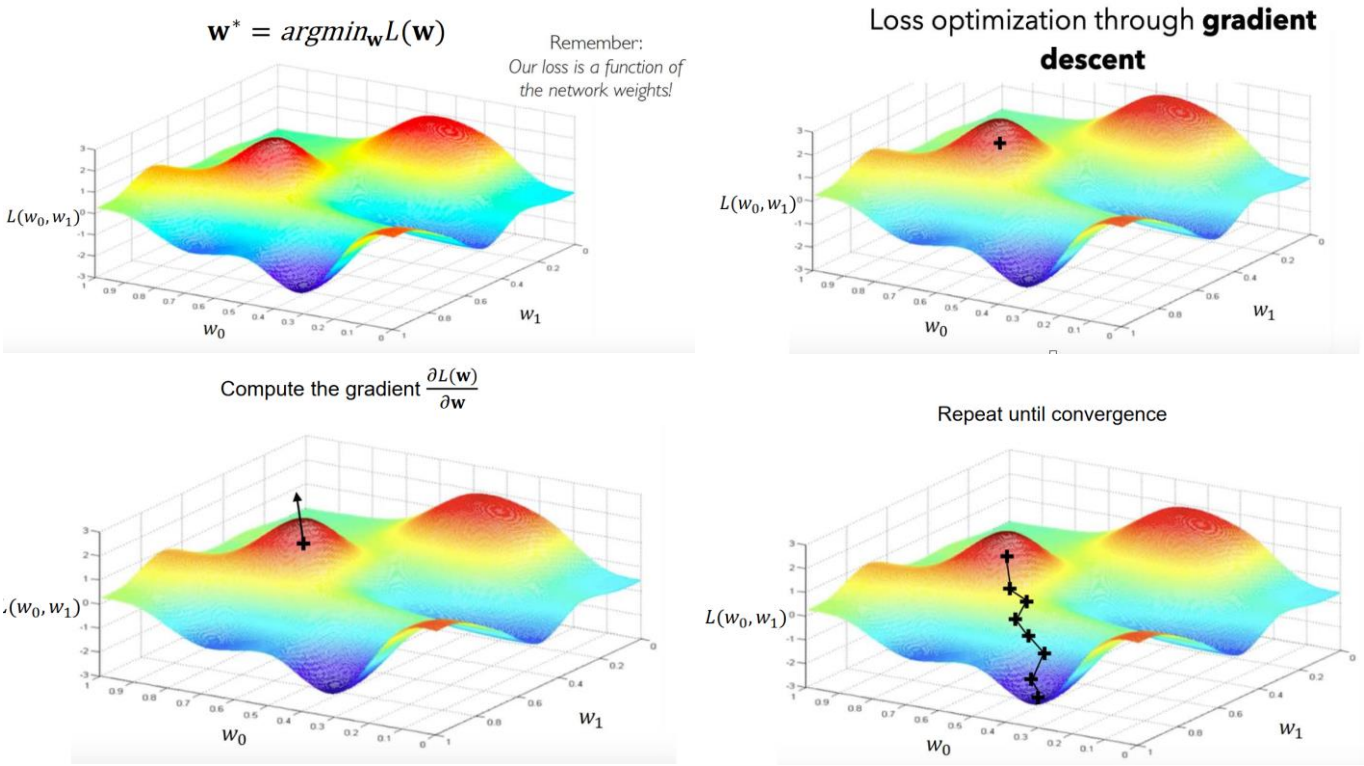
When $\mathbf{X}$ has full column-rank , Normal Equations give: $\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

Alternative solution:

iterative method yielding sequence $\{(w_i^*, b_i^*)\}_{i=0}^{\infty}$ converging to a solution

Machine learning community says "**by training**":
iterative optimization algorithm for updating the parameters toward some direction that incrementally lowers L

- eg.: **Gradient Descent** method, if applicable
- drawback: computing gradient of L over the whole training set $(\mathbf{X},\mathbf{y})$to produce a single update is too expensive



→ Take small step in opposite direction of gradient and repeat until convergence

Gradient descent

Algorithm

1. Initialize the weights randomly, e.g. $\sim N(0, \sigma^2)$
2. Loop until "convergence" a. Compute gradient $\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$
   b. Update weights $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$
3. Return weights

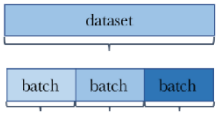
Stochastic Gradient Descent (SGD) algorithm:

The general **iteration** consists in:
1) randomly select a **mini-batch** $\beta \subset \{1, \ldots, n\}$(a chosen subset of training examples)
2) compute the gradient of Lrestricted on β w.r.t. the model parameters, $\partial_{(\mathbf{w},b)} \ell^{(i)}(\mathbf{w}, b)$
3) update the parameters:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\beta|} \sum_{i \in \beta} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b)$$

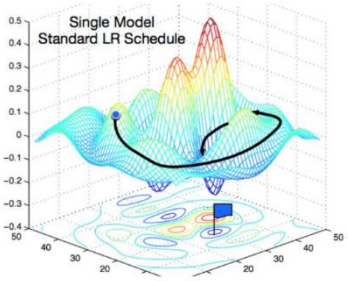
learning rate, step size
batch size

Learning rate and batch size: **hyperparameters** ; if β >1: "Minibatch SGD"
Manually **pre-specified** and eventually subjected to **tuning** (for instance in a validation procedure) → learning rate: typically scheduled by specific optimization algorithms

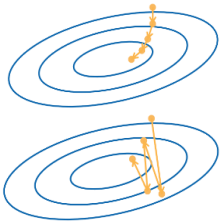


Why "stochastic"? due to random choice of β
• simple approach: subsequently pick all (mini)batches in random order
• a full pass iterating through the whole dataset: **epoch**

We would like iterations to converge to a minimizer point
How to choose η and $|\beta|$?
– Analytically grounded methods: many hypotheses
– Trial and error: most practitioners



Choose a Learning Rate η



- Not too small : too many iterations or epochs
- Not too big : iterates are too wandering

Choose a Batch Size $|\beta|$

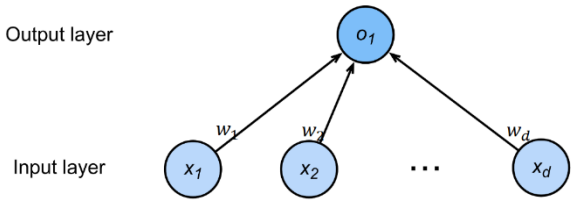
- Not too small : Workload is too small, hard to fully utilize computation resources
- Not too big : Memory occupancy, Waste computation, e.g. when many observations x_i are collinear or so

Terminology summary

- **batch size**: number of training examples in a mini-batch
- **iteration**: one time of update for the current solution ($\mathbf{w}$,b), based on a mini-batch in SGD case
- **epoch** in SGD: a full pass through entire dataset, by as many iterations as # of mini-batches
- Minibatch SGD: SGD where each iteration considers neither a single example nor the full dataset, i.e. a proper "minibatch" version
- Eg. For a dataset of 10000 sample with mini-batch size 1000, 10 iterations will complete 1 epoch

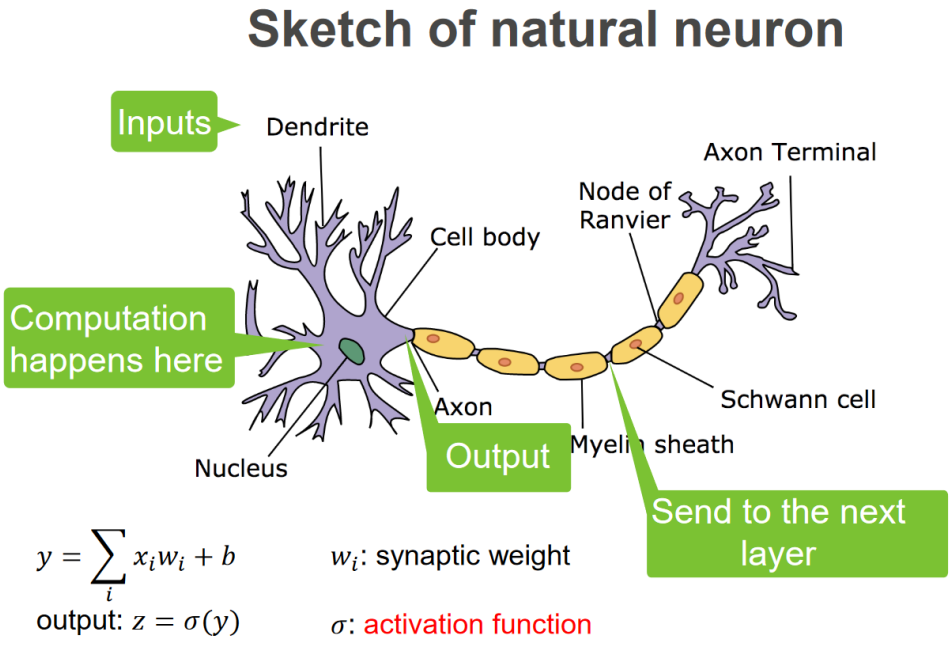
Linear Model viewed as single-layer NN

Linear regression as a NN with one layer (input is not counted).
Input are d-dimensional features (feature vectors)
Output is 1-dimensional



By stacking multiple layers and wiring 1 layer's outputs to subsequent layer's inputs, we get deep neural networks

NNs arise from Neuroscience discoveries



$y = \sum_i x_i w_i + b$ w_i: synaptic weight

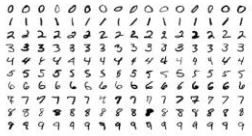
output: $z = \sigma(y)$ σ: activation function

→ **Concise Implementation of Linear Regression Notebook**

Regression vs **Classification**

explains a continuous value — assigns to a discrete category/classes

MNIST:
classify hand-written digits
(10 classes)



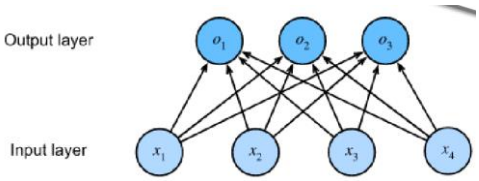
ImageNet:
classify nature objects
(1000 classes)

SOFTMAX REGRESSION

To solve **classification** problems (eg: classify an image according to its content)

Example

- 4 pixels as input and 3 categories as output (cat, dog, pig)
- class label coded with the one-hot encoding: $y \in \{(1,0,0),(0,1,0),(0,0,1$
- estimate $\mathbf{o}$ applying: $\mathbf{o} = \mathbf{W}\mathbf{x}+\mathbf{b}$, where $\mathbf{W}$ is 3×4, $\mathbf{b}$ is 3-dimensional
  called fully-connected layer



Softmax operation. We want:

- output $\hat{y}_j$ be interpreted as "the probability the input belongs to class j" ; e.g.: $\hat{y}_1 = 0.1,\ \hat{y}_2 = 0.8,\ \hat{y}_3 = 0.1$
- easy classification rule: $argmax_j\{\hat{y}_j\}$; if not unique: break the tie
- Unfortunately: $\hat{y}_j \neq \mathbf{o}_j$, since
  - $\mathbf{o}_j$ can be negative, unbounded
  - $\sum_j \mathbf{o}_j \neq 1$
  - $\mathbf{o}_j$ cannot be thus interpreted as a probability

Softmax function

Idea: activation function for transforming **logits** into probabilities (using some differentiable function):

$\hat{\boldsymbol{y}} = softmax(\boldsymbol{o})$, where $\hat{y}_j = \frac{exp(o_j)}{\sum_k exp(o_k)}$

It's also used in multinomial logistic regression. Parameters?

Properties: • $\sum_j \hat{y}_j = 1$ • $0 \leq \hat{y}_j \leq 1$ • $argmax_j\{\hat{y}_j\} = argmax_j\{\hat{o}_j\}$

Loss function

It quantifies how "poor" the prediction is

Common choices:

- For **regression** problems often **SQUARED LOSS**: $L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- For **multiclass classification** problems: **CROSS ENTROPY LOSS**: $L = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot log(\hat{y}_i)$
- **binary classification** case: **BINARY CROSS ENTROPY LOSS** $L = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)$

Linear Model vs Softmax Regression

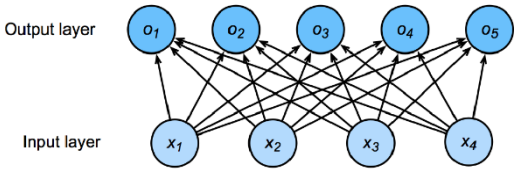
Problem	**Single value regression**	**k-class classification**
Model	$\langle \boldsymbol{w}, \boldsymbol{x} \rangle + \boldsymbol{b}$ $\boldsymbol{w} \in \mathbb{R}^{\boldsymbol{n}}, \boldsymbol{b} \in \mathbb{R}$	$softmax(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ $\boldsymbol{W} \in \mathbb{R}^{\boldsymbol{k}\times\boldsymbol{n}}, \boldsymbol{b} \in \mathbb{R}^{\boldsymbol{k}}$
Loss	Squared loss	Cross-entropy loss

→ **Image classification dataset Notebook**
→ **Softmax Regression concise Notebook**

MULTI-LAYER PERCPTRONS

Limits of one layer NN

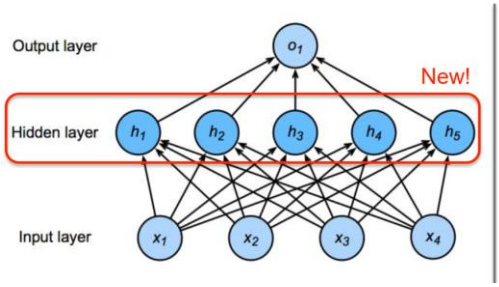
Softmax regression: map input directly to the output via a single affine transformation + softmax operation.



Limit: assumption of linearity in affine transformation
- rather restrictive: can work well for **linearly separable** classes
- need for **more expressive** models

SINGLE HIDDEN LAYER

Does introducing hidden layer(s) help?
No. Combining several linear layers we still get a linear model with some unobserved variables



- Input $\mathbf{x} \in \mathbb{R}^n$
- Hidden $\mathbf{W}_1 \in \mathbb{R}^{m \times n}, \mathbf{b}_1 \in \mathbb{R}^m$
- Output $\mathbf{w}_2 \in \mathbb{R}^m, b_2 \in \mathbb{R}$

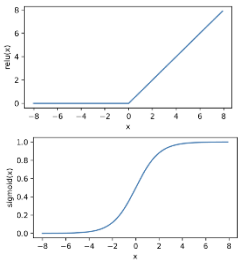
$$\mathbf{h} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{o} = \mathbf{w}_2^T\mathbf{h} + b_2$$
$$\text{hence } o = \mathbf{w}_2^T\mathbf{W}_1\mathbf{x} + b'$$

We need introducing a nonlinear activation function!

$\mathbf{h} = \boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ where σ is an element-wise activation function
$\mathbf{o} = \mathbf{w}_2^T\mathbf{h} + b_2$

Multilayer Perceptron: loosely refers to NN with fully connected layers and 1 or more hidden layers, especially with nonlinear activation function

Common activation functions



- **ReLU**: rectified linear unit : $\text{ReLU}(x) = max(x, 0)$

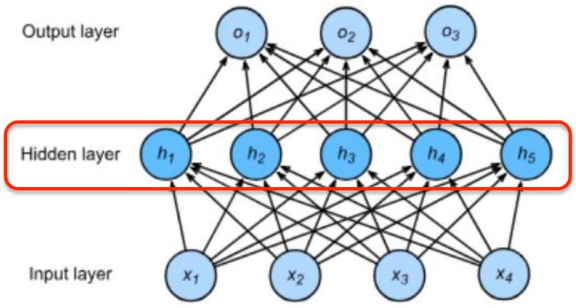
- **Sigmoid**: map $\mathbb{R}$ input into (0, 1), a soft version of $\sigma(x) = \begin{cases} 1, & if x > 0 \\ 0, & otherwise \end{cases}$

$$sigmoid(x) = \frac{1}{1+exp(-x)}$$

Multiclass Classification, with single hidden layer

For multiclass classification we can introduce a hidden layer and an activation function:

$y_1, y_2, \ldots, y_k = softmax(o_1, o_2, \ldots, o_k)$



- Input
  $\mathbf{x} \in \mathbb{R}^n$
- Hidden
  $\mathbf{W}_1 \in \mathbb{R}^{m \times n}$ and $\mathbf{b}_1 \in \mathbb{R}^m$
  $\mathbf{W}_2 \in \mathbb{R}^{m \times d}$ and $\mathbf{b}_2 \in \mathbb{R}^d$
- Output
  $\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
  $\mathbf{o} = \mathbf{w}_2^T\mathbf{h} + \mathbf{b}_2$
  $\mathbf{y} = \text{softmax}(o)$

Single hidden layer vs Deep networks

Single hidden layer networks (with sufficient nodes) with non linear activation functions can model any well-behaved function: we have the informally stated

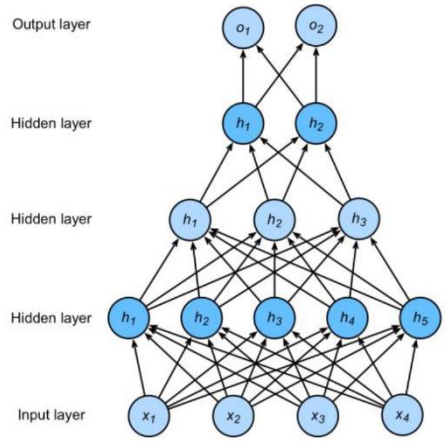
– **Universal Approximation Theorem**: for any continuous function f, closed & bounded set K and $\varepsilon > 0$, there ∃ a 1-hidden-layer NN with continuous activation function σ (on the hidden layer) and remaining linear layers such that the function $f_{\varepsilon,K}$ computed by the NN is ε distant from f, i.e.:

$$\|f_{\varepsilon,K} - f\|_{\infty} = \sup_{x \in K} \|f_{\varepsilon,K}(x) - f(x)\| < \varepsilon$$

- Take home: given a continuous function we can build a suitable NN approximating it at arbitrary given precision ε on a given closed & bounded set K

Why deep networks? they can model the function **more compactly**

MULTIPLE HIDDEN LAYERS



Hyper-parameters:
• # of hidden layers
• Size of each hidden layer

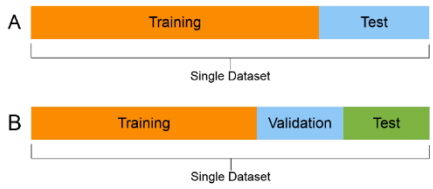
$$\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$
$$\mathbf{o} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4$$

Training error: model error on the training data

Generalization error: model error on new data ; *over/underfitting symptom*: bad generalization capability

Example: practice a future exam with past exams.
Doing well on past exams (training error) doesn't guarantee a good score on the future exam (generalization error)

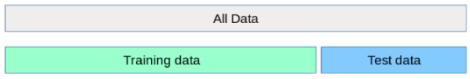
Validation dataset: a dataset used to evaluate model's generalization ability:
• for hyperparameters selection: how many hidden layers? which activation function? ...
• evaluate generalization error
• not to be mixed with the training data



Test dataset: for final evaluation of the model's performance
• not used during model conception and training
• separate from training and validation set

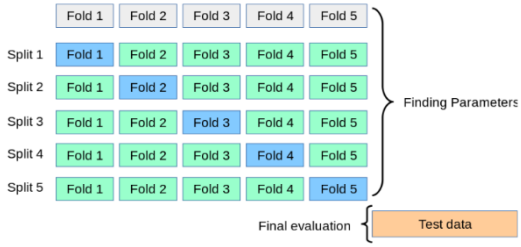
K-fold Cross Validation

It's the more methodical way and it's useful when not sufficient data



Method:
• Partition the training data into K parts
• For $i = 1, \ldots, K$,
use the i-th part as the validation set, the rest for training
• Report the averaged the K validation errors

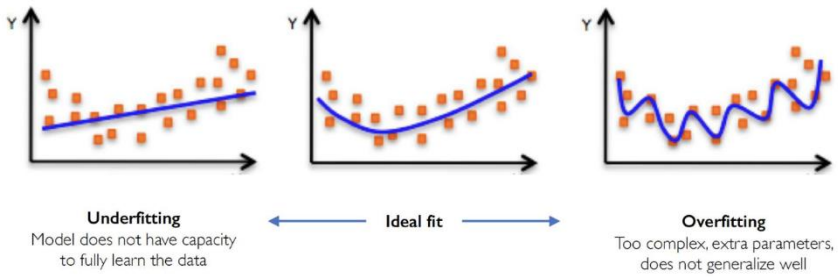
Popular choices: $K = 5$ or 10



MODEL CAPACITY

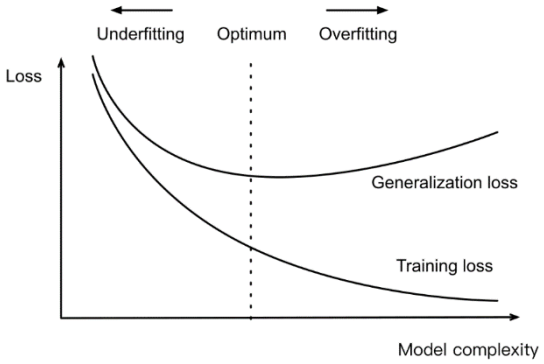
The ability to fit variety of functions

- **Underfitting** : Low capacity models struggles to fit training set (too simple model)
- **Overfitting** : High capacity models can memorize the training set (training error significantly lower than validation error)



Model capacity \ Data complexity	Simple	Complex
Low	Normal	Underfitting
High	Overfitting	Normal

Influence



→ **Concise Implementation Multilayer Perceptrons Notebook**

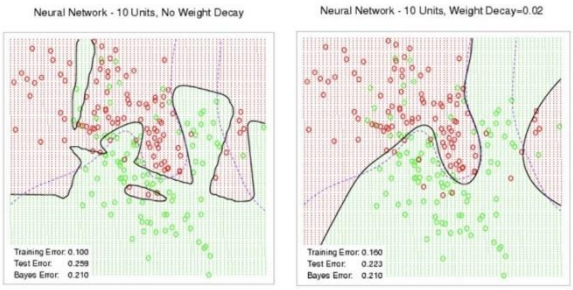
REGULARIZATION

It's a technique that constrains our optimization problem to discourage complex models

Why do we need it?

- avoid overfitting
- Improve generalization of our model on unseen data

Weight decay

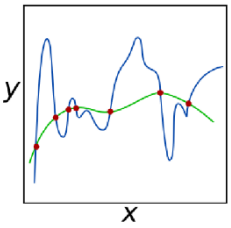


Squared Norm Regularization as Hard Constraint

Same technique, different names over different scientific fields: **Tikhonov regularization, weight decay, $L2$ regularization, ridge regression, ...** • a technique for regularizing parametric ML models

Reduce **model complexity** by **limiting value range :** $\min \ell(\mathbf{w}, b)$ subject to $\|\mathbf{w}\|^2 \leq \theta$

- Often do not regularize bias b (would produce little difference in practice)
- A small θ means stronger regularization



Squared Norm Regularization as Soft Constraint

For each θ, we can find λ to rewrite the hard constraint version into a Lagrangian formulation:

$\min \ell(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$

Hyper-parameter λ controls regularization importance

$\lambda = 0$: no regularization

$\lambda \to \infty \Rightarrow \mathbf{w}^* \to \mathbf{0}$

DROPOUT

A good model should be robust under modest changes in the input
- **Training with input noise** equivalent to regularization
- **Dropout**: inject **noises** into **internal layers**

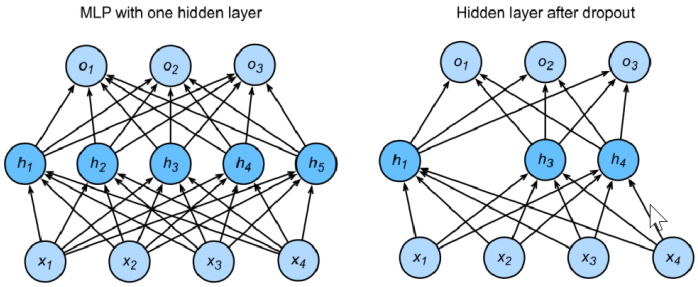
Add Noise without Bias
- Adding noise into $\boldsymbol{x}$ to get $\boldsymbol{x}'$, we hope $\mathbb{E}[\boldsymbol{x}'] = \boldsymbol{x}$
- Dropout perturbs each i-th element by independently tossing a Bernoulli(p) coin, then

$$x_i' = \begin{cases} 0, & with\ probability\ p \\ \frac{x_i}{1-p}, & otherwise \end{cases}$$

- Easily guaranteed: $\mathbb{E}[\boldsymbol{x}'] = \boldsymbol{x}$

Often apply dropout on the output of hidden fully-connected layers

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}' = \text{dropout}(\mathbf{h})$$
$$\mathbf{o} = \mathbf{W}_2\mathbf{h}' + \mathbf{b}_2$$
$$\mathbf{y} = \text{softmax}(o)$$



→ **Dropout Notebook**

EARLY STOPPING - regularization

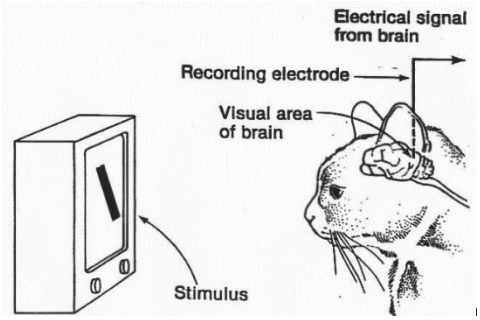
Stop training before we have a chance to overfit.
Overfitting : when a model starts to perform worse on the test (validation) set than on the training set.

INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS

Vision

Main sense in humans → Origin: 540 mln years ago → Our vision system is pre-trained on images seen in 540 mln years! Re-trained in our life

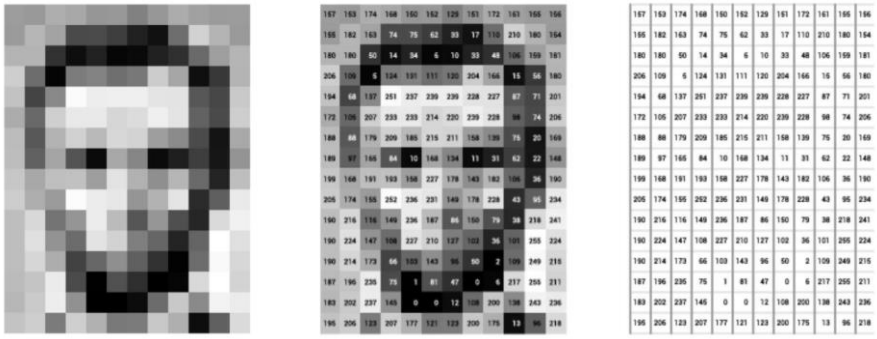
1950s: David Hubel and Torsten Wiesel experiments on the visual cortex of a cat



Findings:
- Certain neurons fire only to very specific patterns and orientation
- Neural mechanisms are spatio-invariant
- Neural layers are organized hierarchically

What computers "see"

Images are Numbers

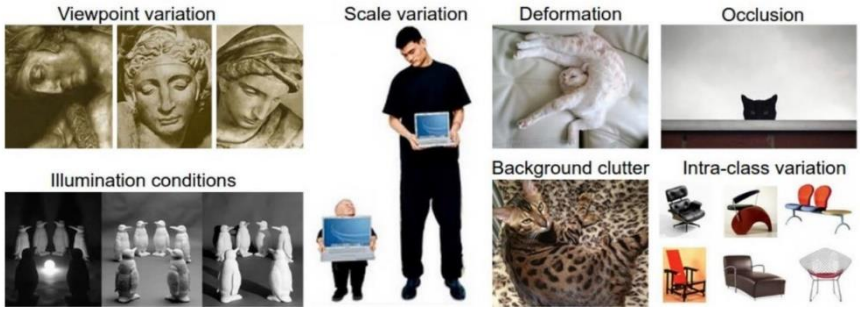


Greyscale image: a matrix of integers in [0, 255] or values in [0,1] to represent grey's intensity
Colour image: one intensity value for each (pixel and) channel; additive colour synthesis
RGB format has 3 channels: Red, Green, Blue. Eg.: 3 x 1080 x 1080 tensor for an RGB image

Manual Feature Extraction

Define features → Domain knowledge → Detect features to classify

Main problem : which features?



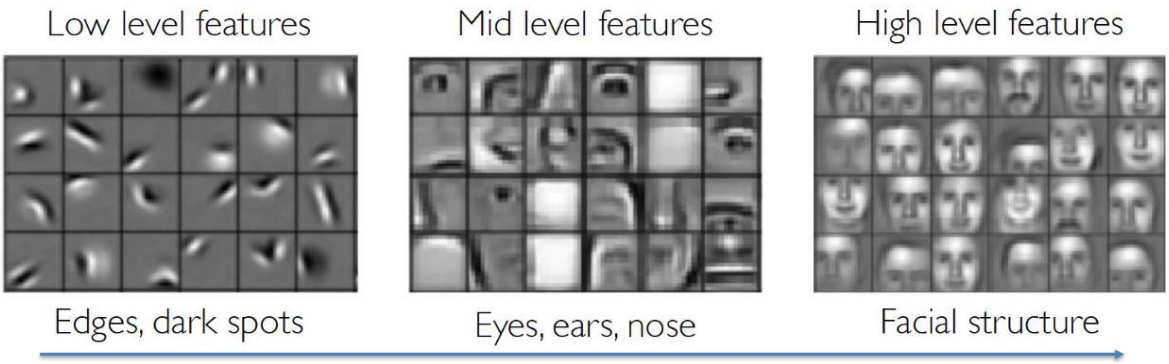
The desired property is **robustness**
- View point variation
- Illumination conditions
- Scale variation
- Deformation
- Occlusion
- Background clutter
- Intra-class variation

Identify **key features** for each category

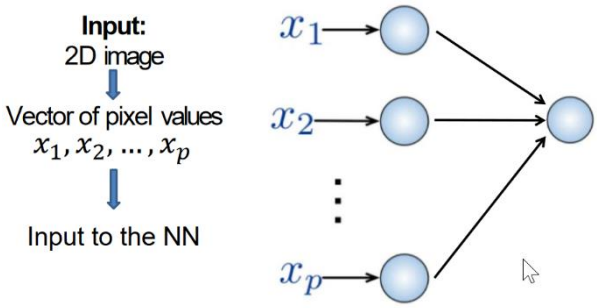
How can computers do that?
It's difficult that it classify a person for the nose, eyes and mouth….

LEARNING VISUAL FEATURES

Can we **learn** a **hierarchy of features** directly from the data instead of hand engineering?



Fully Connected Neural Network?



- Connect neuron in hidden layer to all neurons in input layer
- No spatial information!
- Way too many parameters!

How can we use **spatial structure** in the input to inform the architecture of the network?

Example : searching Waldo

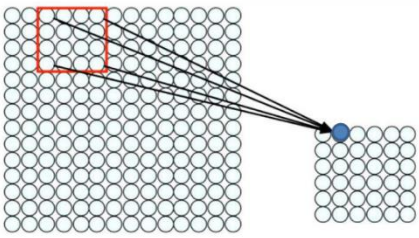
Patch: small region of the image. Localizing waldo on an image is on respect on two principles:

- Translation invariance
- Locality

CONVOLUTIONAL LAYER

Spatial Convolution

- **Filter** or **kernel** of size $(n \times n)$: n^2 different weights
- Apply this same filter to each and all $(n \times n)$ patches of input image: locality
- Not necessary all locations: shift by $s$pixels for next patch (**stride**)

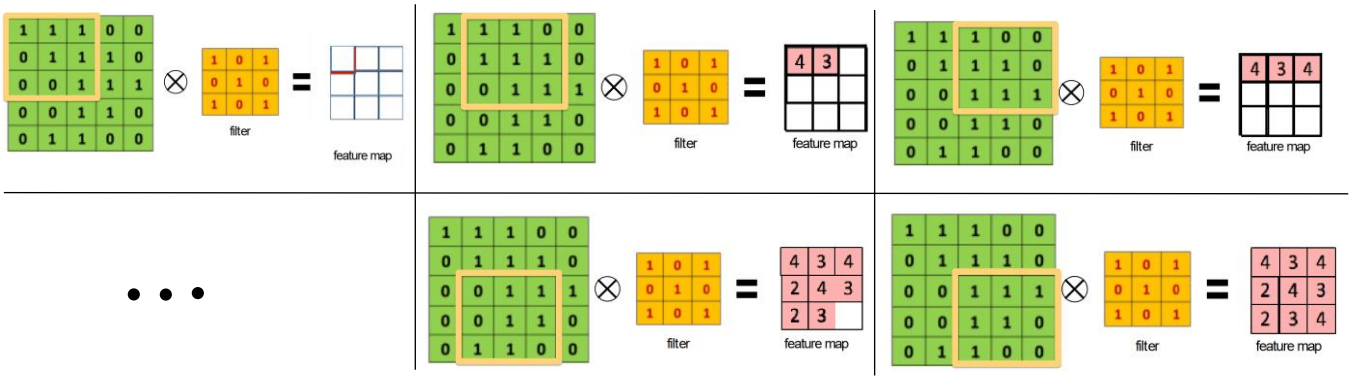


The Convolution Operation

Shall be properly called: **correlation**

Suppose we want to compute the **convolution** of a 5x5 image and a 3x3 filter:

We slide the 3x3 filter over the input image, element-wise multiply, and sum up the outputs...



Producing Feature Maps

Different weights (filters) extract different features!
Which weights?

Convolutional layer

1) connect patches of input to neurons in hidden layer.
2) Slide the patch window across the image.

Different **weights (filters):** detect different features
Sliding across whole image: translation invariance

2D convolution con be easily reduced to 1D or extended to 3D or higher

- **1D** suitable for data: Text , Voice , Time series
- **3D** suitable for data: Video , Medical images

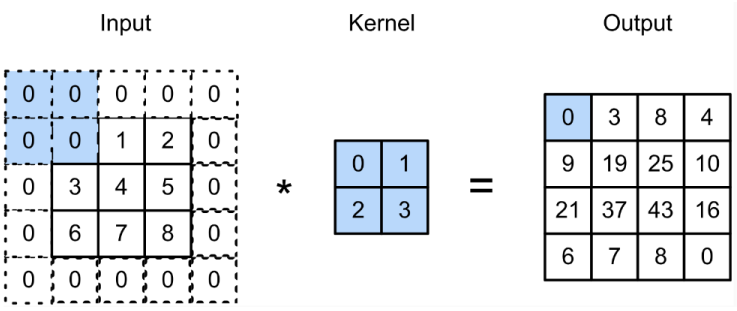
INGREDIENTS

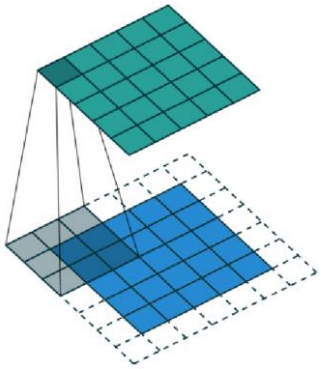
Padding

Example: given a $n_h \times n_w = 32 \times 32$ image

- apply conv. layer with $k_h \times k_w = 5 \times 5$ kernel
  - 28 x 28 output with 1 layer
  - 4 x 4 output with 7 layers chained
- Shape decreases faster with larger kernels
  Shape reduces from $n_h \times n_w$ to $(n_h - k_h + 1) \times (n_w - k_w + 1)$

Padding adds zero-value rows/columns around input.





$$0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$$

Padding p_h rows and p_w columns, output shape will be

Common choice: $p_h = k_h - 1$and $p_w = k_w - 1$

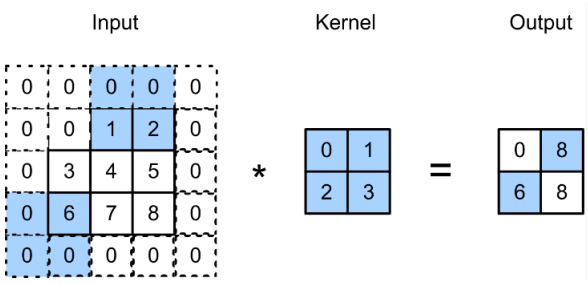
- Odd k_h: pad p_h 2on both sides
- Even k_h: pad $\lceil p_h/2 \rceil$on top, $\lfloor p_h/2 \rfloor$on bottom

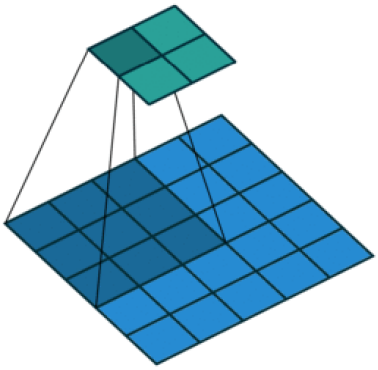
Padding reduces shape linearly with # of layers

- given 224 x 224 input with 5 x 5 kernel: needs 44 layers to reduce the shape to 4 x 4
- Large amount of computation if taking all locations! It can be mitigated by striding

Stride

Stride is the # of rows/columns per slide step ***Example*** : Strides of (height,width) = (3,2)





$$0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$$
$$0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$$

Given stride of shape (height,width) = $s_h \times s_w$ the output shape is

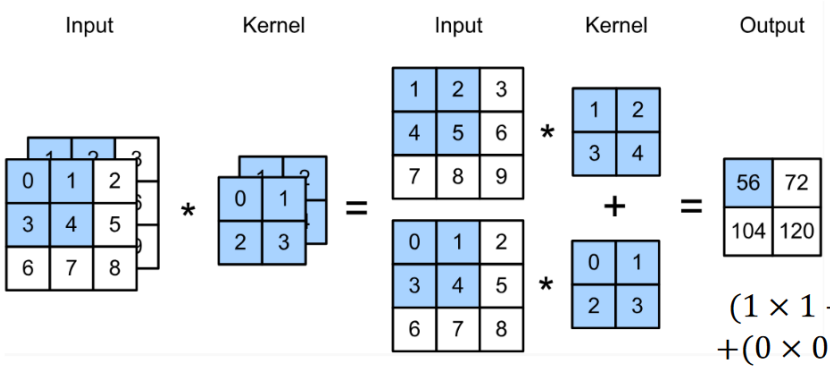
$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

With $p_h = k_h - 1$ and $p_w = k_w - 1$: $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$
If input height/width are divisible by strides $(n_h/s_h) \times (n_w/s_w)$

Multiple input channels

Example : Color image may have three RGB channels. Converting to grayscale loses information of course

Have a kernel for each channel, and then sum results over channels



$$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$$

$\mathbf{Y} = \sum_{i=0}^{c_i} \boldsymbol{X}_{i,:,:} \star \boldsymbol{W}_{i,:,:}$

$\mathbf{X}$: $c_i \times n_h \times n_w$ input
$\mathbf{W}$: $c_i \times k_h \times k_w$ kernel
$\mathbf{Y}$: $m_h \times m_w$ output
$\star$: convultion

No matter how many inputs channels, so far we always got a single output channel
We can have **multiple** 2D kernels, **each** one generates an **output channel**

$\boldsymbol{Y}_{i,:,:} = \boldsymbol{X} \star \boldsymbol{W}_{i,:,:,:}$
"for $i = 1, \ldots, c_o$"

$\mathbf{X}$: $c_i \times n_h \times n_w$ input
$\mathbf{W}$: $c_0 \times c_i \times k_h \times k_w$ kernel
$\mathbf{Y}$: $c_0 \times m_h \times m_w$ output

Each output channel may recognize a particular pattern
Input channels kernels recognize and combines patterns in inputs

Pooling

A single convolution is sensitive to position (Eg: Detect vertical edges)

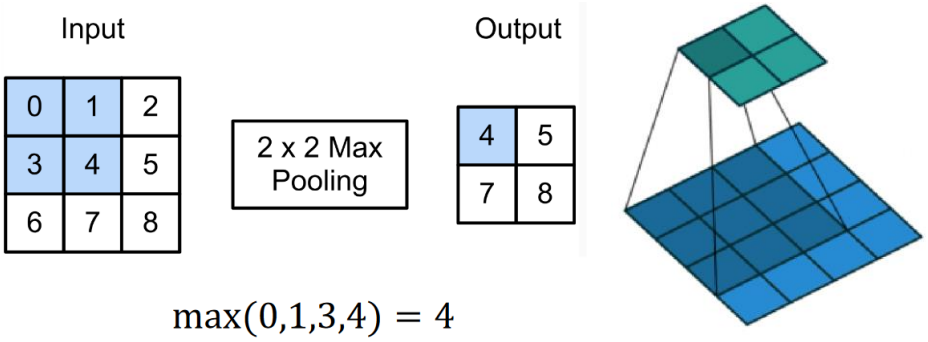
```
X= [[1. 1. 0. 0. 0.
    [1. 1. 0. 0. 0.
    [1. 1. 0. 0. 0.
    [1. 1. 0. 0. 0.

Y= [[ 0.  1.  0.  0.
    [ 0.  1.  0.  0.
    [ 0.  1.  0.  0.
    [ 0.  1.  0.  0.
```

0 output with 1 pixel shift

We need some degree of **invariance** to translation : lighting, object positions, scales, appearance vary among images

2-D Max Pooling

Returns the maximal value in a sliding window



Detect vertical edge on

```
[[1. 1. 0. 0. 0.
 [1. 1. 0. 0. 0.
 [1. 1. 0. 0. 0.
 [1. 1. 0. 0. 0.
```

Conv. output

```
[[ 0.  1.  0.  0.
 [ 0.  1.  0.  0.
 [ 0.  1.  0.  0.
 [ 0.  1.  0.  0.
```

2 x 2 max pooling

```
[[ 1.  1.  1.  0.
 [ 1.  1.  1.  0.
 [ 1.  1.  1.  0.
 [ 1.  1.  1.  0.
```

Tolerant to 1 pixel shift

Padding, Stride, and Multiple Channels

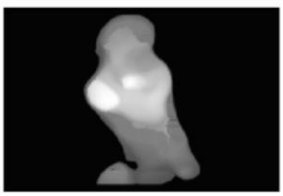
- Pooling layers have similar padding and stride as convolutional layers
- No learnable parameters
- Apply pooling for each input channel to obtain the corresponding output channel

#output channels = #input channels

Average Pooling

- Max pooling: the "strongest pattern signal" in a window
- **Average pooling**: replace max with mean in max pooling
- The average signal strength in a window

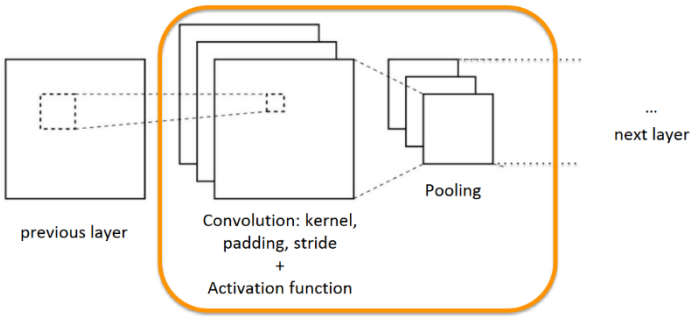
Max pooling



Average pooling

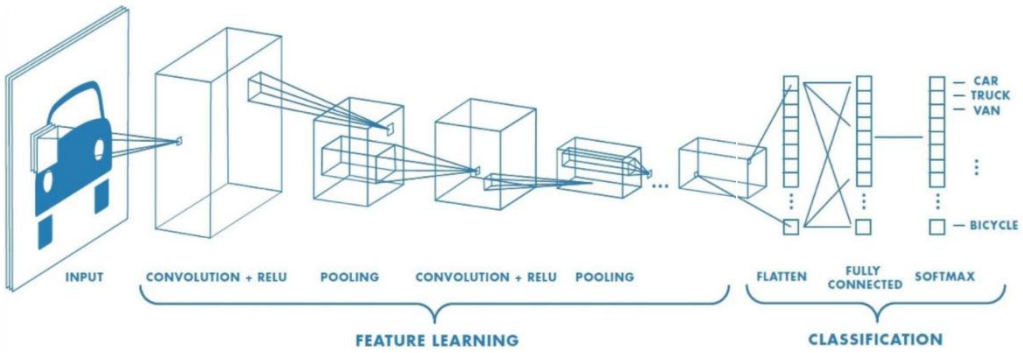
CONVOLUTIONAL NEURAL NETWORKS (CNNs)

CNN: chaining convolutional layers



CNNs for Classification:

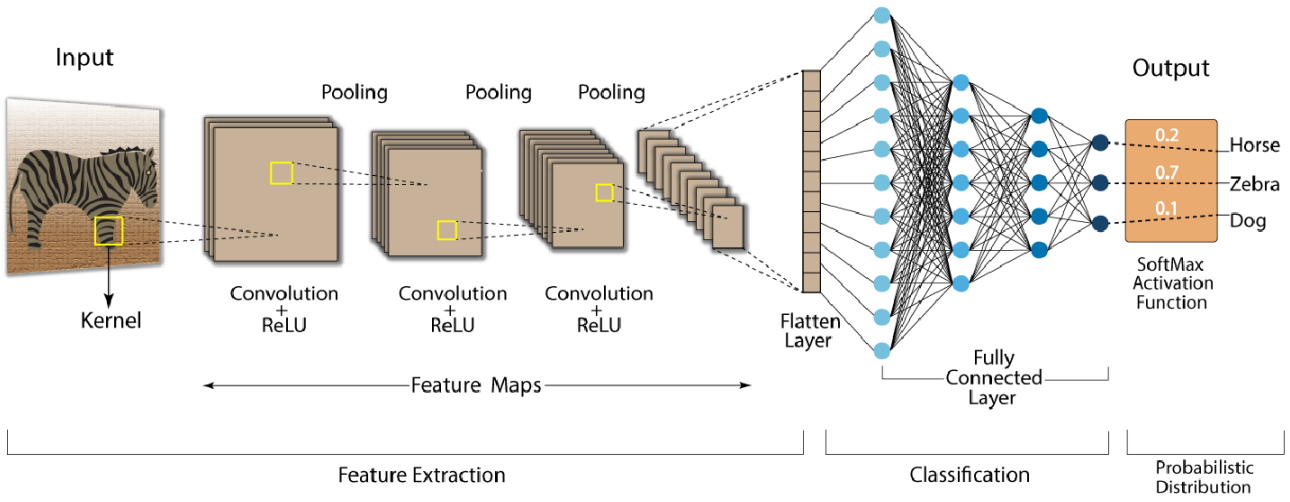
- Feature Learning
- Class probabilities



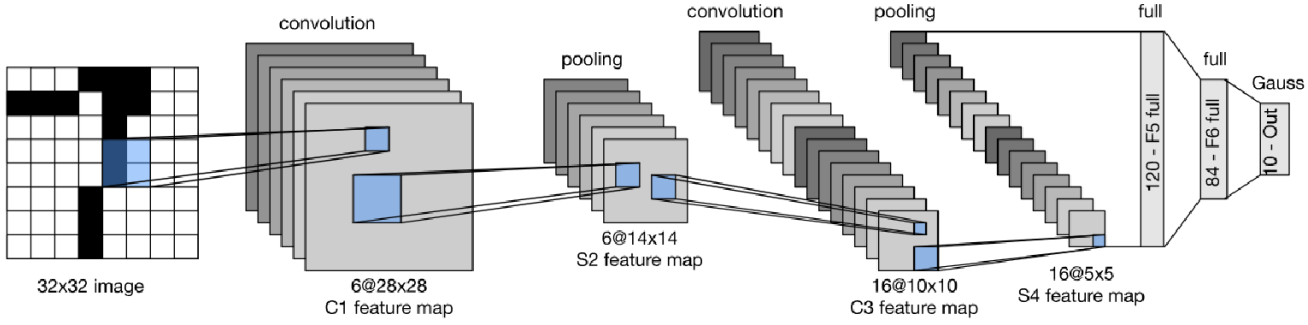
- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as probability of image belonging to a particular class (using the softmax activation function)

$$softmax(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Example architecture

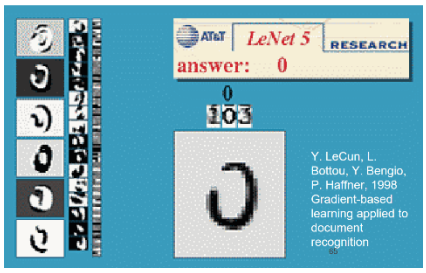


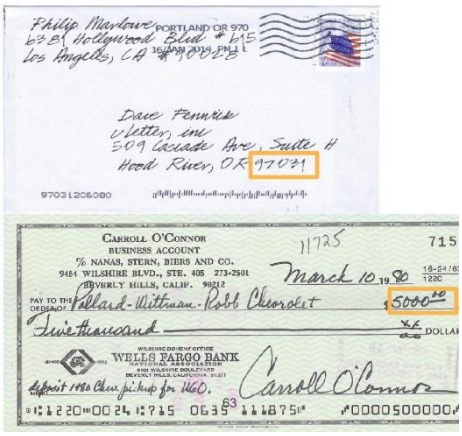
LeNet architecture (Y. LeCun 1998)



Example : Handwritten Digit Recognition (MNIST)

- *Centered and scaled*
- *50,000 training data*
- *10,000 test data*
- *28 x 28 images*
- *10 classes*





LeNet in pyTorch

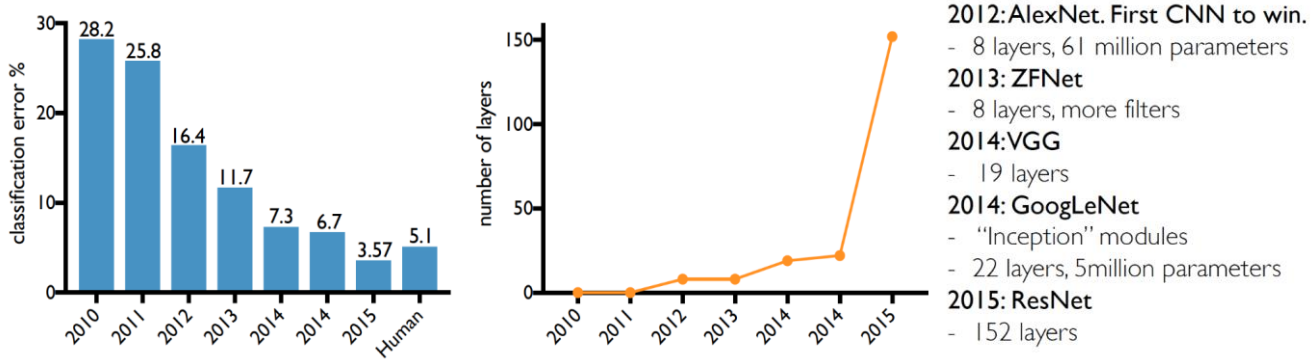
```
net = torch.nn.Sequential(
      Reshape(),
      nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
      nn.AvgPool2d(kernel_size=2, stride=2),
      nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
      nn.AvgPool2d(kernel_size=2, stride=2),
      nn.Flatten(),
      nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
      nn.Linear(120, 84), nn.Sigmoid(),
      nn.Linear(84, 10))
```

→ **Convolutional layer Notebook**

MODERN CNN

CNNs for Classification: ImageNet

Classification task: produce a list of object categories present in image. 1000 categories.



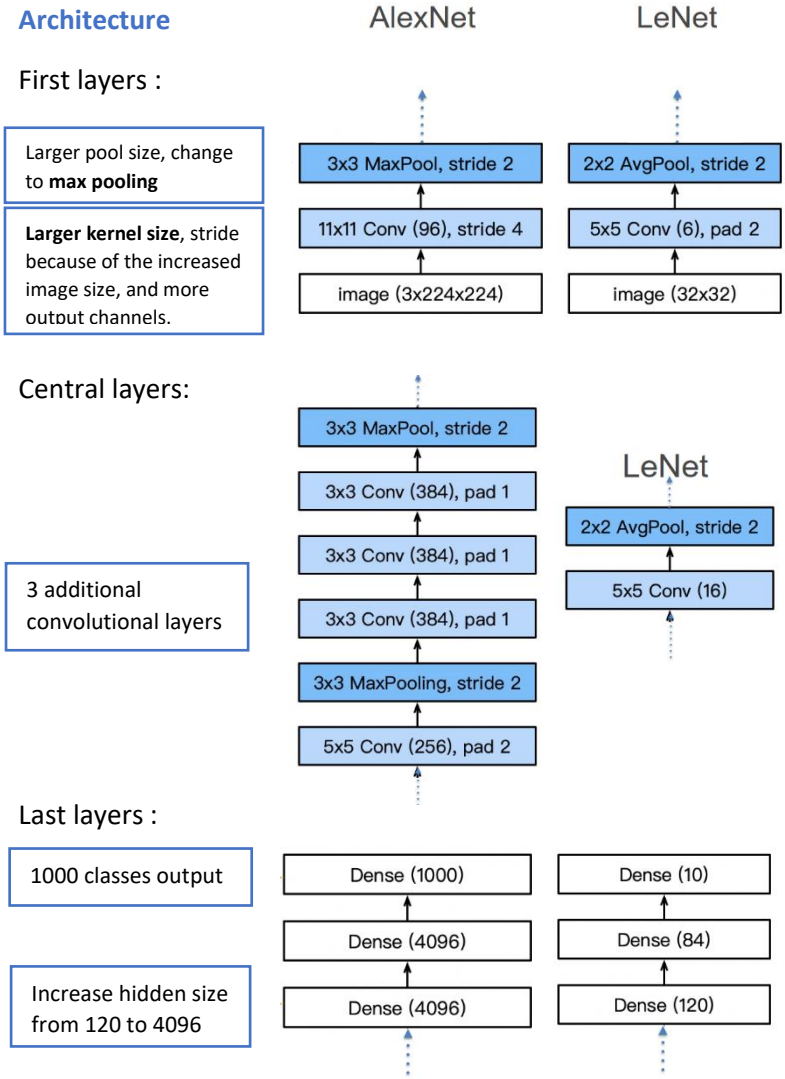
"Top5 error": rate at which the model does not output correct label in top 5 predictions

Other tasks include: single-object localization, object detection from video/image, scene classification, scene parsing

ALEXNET

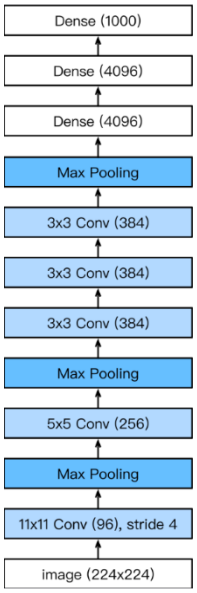
AlexNet won ImageNet competition in 2012. Deeper and bigger LeNet. Paradigm shift for computer vision.
Key modifications : - Dropout (regularization) - ReLu (training) - MaxPooling

Architecture



More Tricks

- Change activation function from Sigmoid to **ReLu** (no more vanishing gradient)
- Add a **dropout** layer after two hidden dense layers (better robustness / regularization)
- **Data augmentation**



Complexity

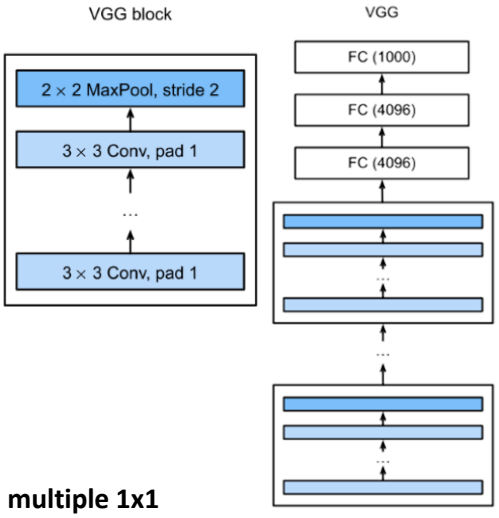
	#parameters		FLOP	
	AlexNet	LeNet	AlexNet	LeNet
Conv1	35K	150	101M	1.2M
Conv2	614K	2.4K	415M	2.4M
Conv3-5	3M		445M	
Dense1	26M	0.48M	26M	0.48M
Dense2	16M	0.1M	16M	0.1M
Total	46M	0.6M	1G	4M
Increase	11x	1x	250x	1x

→ **AlexNet Notebook**

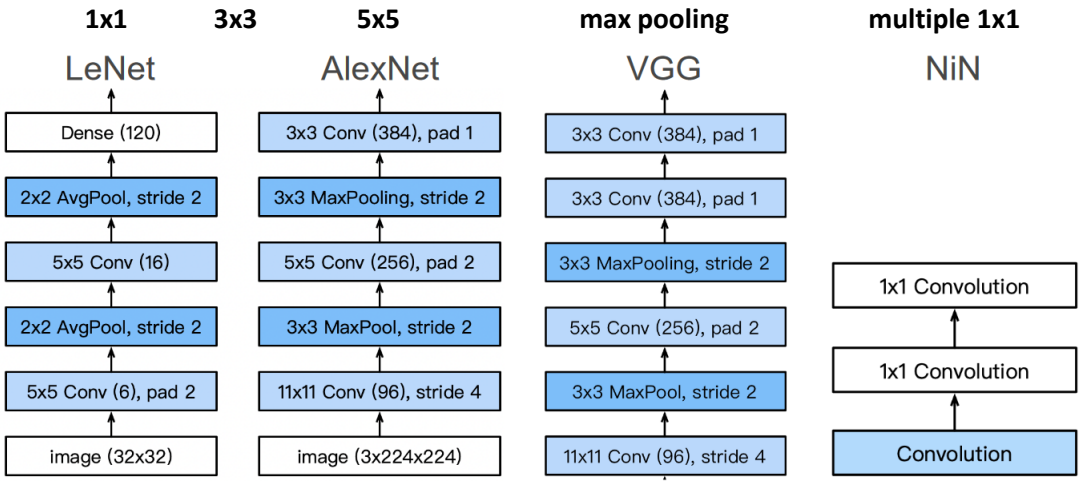
VGG

Main idea : abstract from neurons and layers to Blocks

VGG: concatenation of blocks



Picking the best convolution



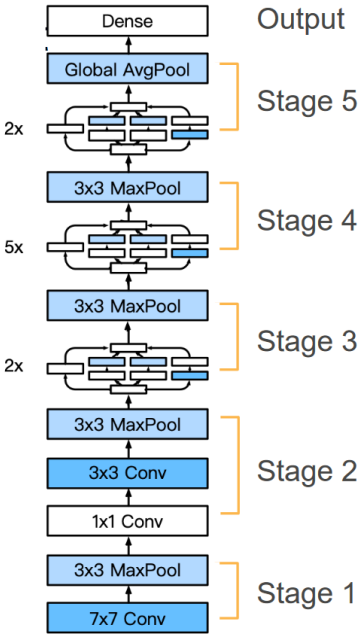
Do we really need choosing? Just pick them all

GoogLeNet

5 stages with 9 "inceptions" blocks

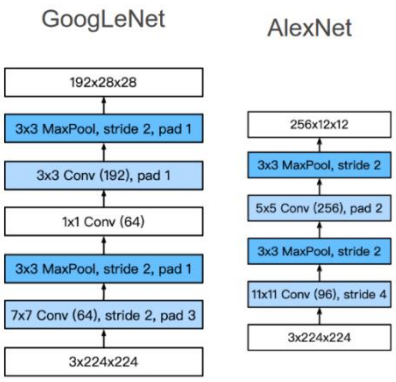
Inception block:

- "We need to go deeper", from sci-fi film "Inception"
- goal: deeper but computationally tractable
- idea: stack more layers, but insert several different layers working in "parallel" then concatenated



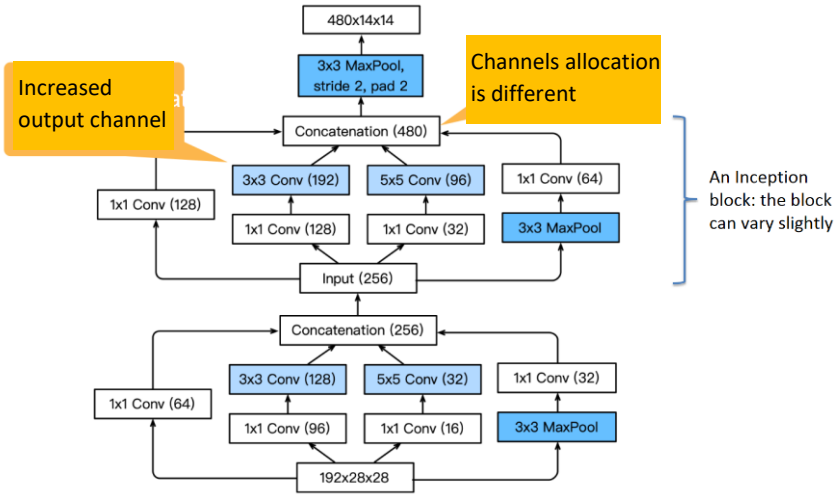
→ **GoogLeNet Noebook**

Stage 1 & 2

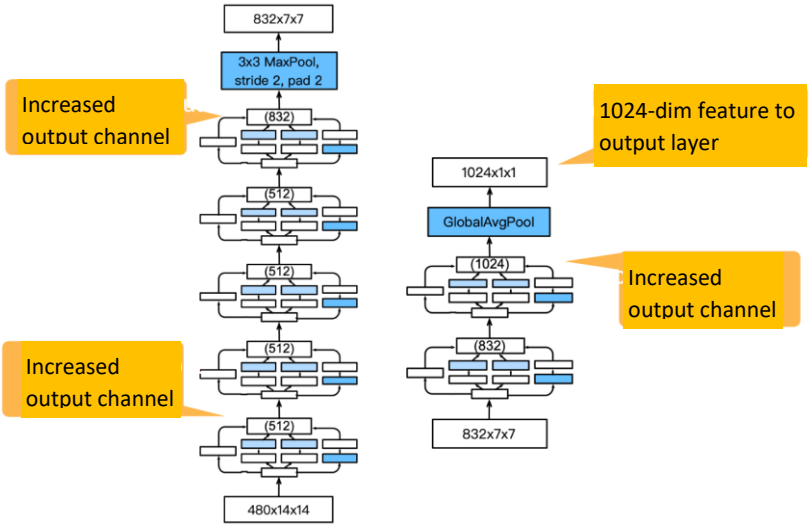


Smaller kernel size and output channels due to more layers

Stage 3



Stage 4 & 5



Batch Normalization

Loss occurs at last layer : last layers learn quickly
Data is inserted at bottom layer

- Bottom layers change - everything changes
- Last layers need to relearn many times: values can change dramatically
  ->long time before convergence
- This is like covariate shift (still debated). Can we avoid changing last layers while learning first layers?

The variables in intermediate layers may take values with widely varying magnitudes.
Add a normalization:

- Estimate the mean and std of the current mini-batch: $\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x},$

$$\hat{\boldsymbol{\sigma}}^2_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon$$

- Normalize input data of each layer accordingly $\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}$

Compute new mean and variance for every minibatch

- Effectively acts as regularization
- Common minibatch size is ~128

Batch Normalization: frequently used in ResNet, belonging to an architecture type called residual networks
Can be inserted into other architectures

RECURRENT NEURAL NETWORKS

Given an image of a ball, can you predict where it will go next?

Sequences in the wild : riascolto ??+ slides

Many others contexts:
- analysis of medical signals like ECGs
- predicting stock trends
- processing genomic data
- in Computer Vision: given the behavior of an object for a certain period, predict its behavior in next frames

A Sequence Modeling Problem: Predict the Next Word

"This morning I took my cat for a walk" given the green words we want to predict what's next.

Solution using DNN? *Problem*: Input with fixed size, while sentences are **variable**

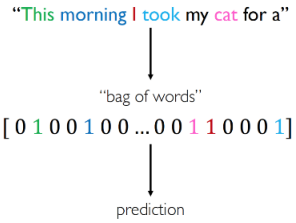
Other ideas :

1) Use a fixed window

One hot feature encoding : tells us what each word is

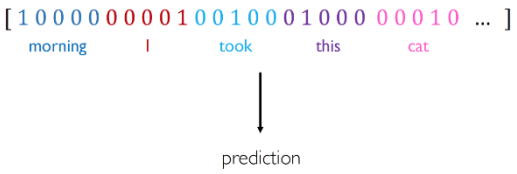
Problem : can't model long-term dependencies
Example :
"France is where I grew up, but now I live in Boston. I speak a fluent ____"
We need information from the distant past to accurately predict the current word

2) Use entire sequence as set of counts

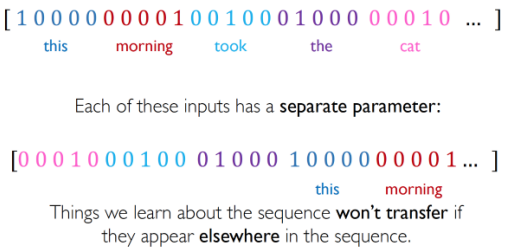


Problem: counts don't preserve order
Example :
The food was good, not bad at all
vs The food was bad, not good at all.

3) Use a really big fixed window



Problem : no parameter sharing



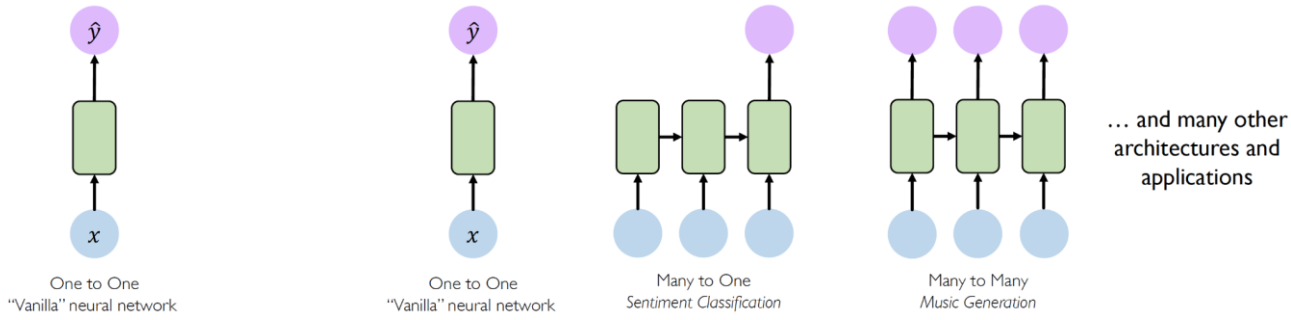
Sequence modelling: design criteria
To model sequences, we need to:
1) Handle **variable-length** sequences
2) Track **long-term** dependencies
3) Maintain information about **order**
4) **Share parameters** across the sequence

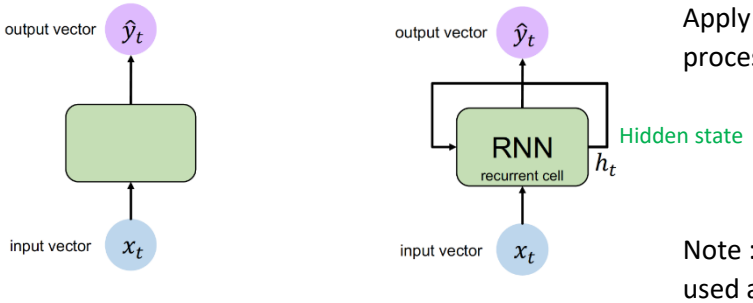
Recurrent Neural Networks (RNNs)

Recurrent: information is passed internally from one time step to the next

Standard feed-forward "vanilla" neural network

RNN : sequence modelling





Apply a recurrence relation at every time step to process a sequence :

$$h_t = f_W(h_{t-1}, x_t)$$

new state — function parameterized by W — old state — input vector at time step t

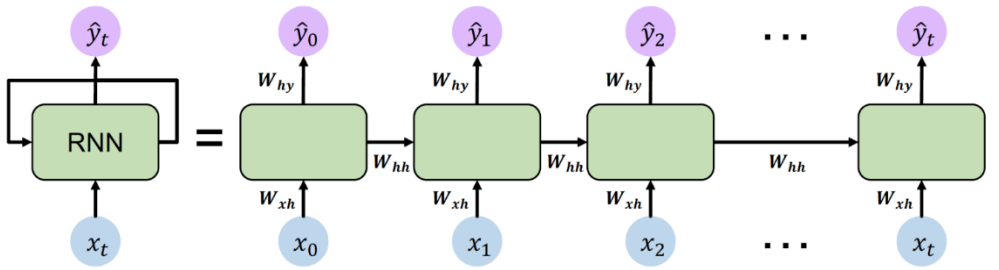
Note : the same function and set of parameters are used at every time step

RNN intuition

```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]
sentence = ["I", "love", "recurrent", "neural"]

for word in sentence :
      prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"
```

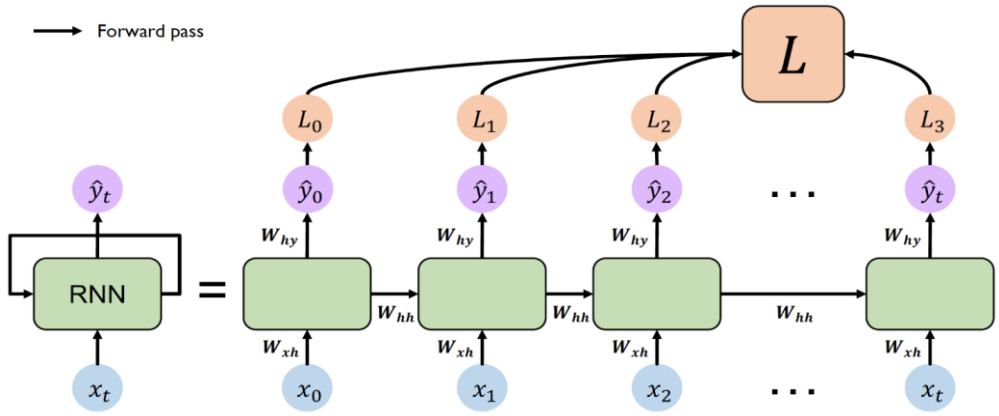
An **RNN** is represent as **computational graph** unrolled **across time**

$W_{xh}: x \to h$
$W_{hh}: h \to h$
$W_{hy}: h \to y$



Re use the same weight matrices at every time step

Then …



Preliminary operations on Colab

- Colab timeouts:
  - notebook's runtime **session** limit: 12 hrs
    But in the session, from Colab's FAQ: "Colaboratory is intended for interactive use. Long-running background computations, particularly on GPUs, may be stopped"
  - notebook **idle** (no user interaction) timeout: 30 min
- We will need PyTorch software and D2L textbook's companion Python module for working with the PyTorch-based exercises of the textbook: before importing these modules in Python we need to install them in Colab
- The **import**s have effect in the single notebook only
- Colab currently runs on top of Linux operating system: we can issue a Linux command in a cell by prefixing it with `!`
  - Linux commands may have arguments, called options, specified by prefixing with `-` or `--`, e.g. `--help`

```
[1]  !uname --help
     # uname is a Linux command that shows OS infos
```

```
Usage: uname [OPTION]...
Print certain system information.  With no OPTION, same as -s.

  -a, --all                print all information, in the following order,
                             except omit -p and -i if unknown:
  -s, --kernel-name        print the kernel name
  -n, --nodename           print the network node hostname
  -r, --kernel-release     print the kernel release
  -v, --kernel-version     print the kernel version
  -m, --machine            print the machine hardware name
  -p, --processor          print the processor type (non-portable)
  -i, --hardware-platform  print the hardware platform (non-portable)
  -o, --operating-system   print the operating system
      --help     display this help and exit
      --version  output version information and exit

GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
Full documentation at: <http://www.gnu.org/software/coreutils/uname>
or available locally via: info '(coreutils) uname invocation'
```

```
[2]  !uname -a
     # The option a tells it to show "all" basic infos
```

```
Linux f023cf48a682 5.4.104+ #1 SMP Sat Jun 5 09:50:34 PDT 2021 x86_64 x86_64 x86_64 GNU/Linux
```

- ... few OS commands are accepted with no ! prefixing if it is on single line: eg. the following `cat` (concatenation) command takes a file path (argument) and prints the file content as output

```
[3]  cat /root/.profile
```

```
# ~/.profile: executed by Bourne-compatible login shells.

if [ "$BASH" ]; then
  if [ -f ~/.bashrc ]; then
    . ~/.bashrc
  fi
fi

mesg n || true
```

- Usual IPython notebook "magic" commands are accepted as well: %pwd, %ls, %cd, ...

```
[4]  %ls -al /
     # Options a and l stand for: "all" the files including "hidden" ones (those named with . prefix
     # in Linux) and "long" infos about each file/dir
```

```
total 108
drwxr-xr-x   1 root root 4096 Oct 27 12:13 ./
drwxr-xr-x   1 root root 4096 Oct 27 12:13 ../
drwxr-xr-x   1 root root 4096 Oct  8 13:36 bin/
drwxr-xr-x   2 root root 4096 Apr 24  2018 boot/
drwxr-xr-x   1 root root 4096 Oct  8 13:45 content/
drwxr-xr-x   1 root root 4096 Oct 14 13:12 datalab/
drwxr-xr-x   5 root root  360 Oct 27 12:13 dev/
-rwxr-xr-x   1 root root    0 Oct 27 12:13 .dockerenv*
drwxr-xr-x   1 root root 4096 Oct 27 12:13 etc/
drwxr-xr-x   2 root root 4096 Apr 24  2018 home/
drwxr-xr-x   1 root root 4096 Oct  8 13:37 lib/
drwxr-xr-x   2 root root 4096 Oct  8 13:24 lib32/
drwxr-xr-x   1 root root 4096 Oct  8 13:24 lib64/
drwxr-xr-x   2 root root 4096 Nov 19  2020 media/
drwxr-xr-x   2 root root 4096 Nov 19  2020 mnt/
drwxr-xr-x   1 root root 4096 Oct  8 13:39 opt/
dr-xr-xr-x 176 root root    0 Oct 27 12:13 proc/
drwxr-xr-x  14 root root 4096 Oct  8 13:38 python-apt/
drwx------   1 root root 4096 Oct 27 12:13 root/
drwxr-xr-x   1 root root 4096 Oct  8 13:28 run/
drwxr-xr-x   1 root root 4096 Oct 27 12:13 sbin/
drwxr-xr-x   2 root root 4096 Nov 19  2020 srv/
dr-xr-xr-x  12 root root    0 Oct 27 12:13 sys/
drwxr-xr-x   4 root root 4096 Oct  8 14:10 tensorflow-1.15.2/
drwxrwxrwt   1 root root 4096 Oct 27 12:13 tmp/
drwxr-xr-x   1 root root 4096 Oct 14 13:12 tools/
drwxr-xr-x   1 root root 4096 Oct  8 13:39 usr/
drwxr-xr-x   1 root root 4096 Oct 27 12:13 var/
```

Let's install PyTorch main module (torch) and the submodule for some Computer Vision functionalities (torchvision), as well as the D2L textbook's companion package (d2l). PIP is accepted as single line system command.

```
[5]  pip install torch torchvision
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.9.0+cu111)
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packages (0.10.0+cu11
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: pillow>=5.3.0 in /usr/local/lib/python3.7/dist-packages (from torc
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from torchvision)
```

```
[6]  pip install d2l
```

```
Collecting d2l
  Downloading d2l-0.17.0-py3-none-any.whl (83 kB)
     |████████████████████████████████| 83 kB 1.4 MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from d2l) (1.1
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from d2l) (
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from d2l)
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-packages (from d2l) (1
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from d2l) (1.
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-packages (from jupyte
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-packages (from jupy
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-packages (from jupyt
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-packages (from jupyt
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-packages (from jupyt
Requirement already satisfied: tornado>=4.0 in /usr/local/lib/python3.7/dist-packages (from ip
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: traitlets>=4.1.0 in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: ipython>=4.0.0 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7/dist-packages (fro
```

These following groups of **import**s gather all the modules required by textbook codes, so it is safe to import them all.

```
[7]  import numpy as np
     import torch
     import torchvision
     from PIL import Image
     from torch import nn
     from torch.nn import functional as F
     from torch.utils import data
     from torchvision import transforms
```

```
[8]  import collections
     import hashlib
     import math
     import os
     import random
     import re
     import shutil
     import sys
     import tarfile
     import time
     import zipfile
     from collections import defaultdict
     import pandas as pd
     import requests
     from IPython import display
     from matplotlib import pyplot as plt
     d2l = sys.modules[__name__]
```

We can check whether D2L's software is properly installed by importing torch submodule from d2l module and then showing info on d2l variable:

```
[9]  from d2l import torch as d2l
```

```
[10] d2l
```

```
<module 'd2l.torch' from '/usr/local/lib/python3.7/dist-packages/d2l/torch.py'>
```

Python Review and Extras

This notebook contains

- some review sections you can skip, in black colour
- some topics in red coloured sections we will use throughout the course

Topics

- **Basic Python**: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- **Numpy**: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- **Matplotlib**: Plotting, Subplots, Images

```
[1]  !python --version
```

```
Python 3.7.12
```

A Brief Note on Python Versions

As of Janurary 1, 2020, Python has officially dropped support for `python2`. We'll be using Python 3.7 for this iteration of the course. You can check your Python version at the command line by running `python --version`. In Colab, we can enforce the Python version by clicking `Runtime -> Change Runtime Type` and selecting `python3`. Note that as of April 2020, Colab uses Python 3.6.9 which should run everything without any errors.

Basics of Python

- Python is a high-level, dynamically typed multiparadigm programming language
- Python code is often said to be almost like pseudocode
- As an example, here is an implementation of the classic quicksort algorithm in Python:

```
[2]  # Quicksort

     def quicksort(arr):
         if len(arr) <= 1:
             return arr
         pivot = arr[len(arr) // 2]
         left = [x for x in arr if x < pivot]
         middle = [x for x in arr if x == pivot]
         right = [x for x in arr if x > pivot]
         return quicksort(left) + middle + quicksort(right)

     # apply
     print(quicksort([3,6,8,100,1,2,3,5,6,2,1,6,3,21,9,6,4,2,7,89,9,1]))
```

```
[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 5, 6, 6, 6, 6, 7, 8, 9, 9, 21, 89, 100]
```

Basic data types

Numbers

- Integers and floats work as you would expect from other languages:

```
[3]  x = 3
     print(x, type(x))
```

```
3 <class 'int'>
```

```
[4] print(x + 1)   # Addition
    print(x - 1)   # Subtraction
    print(x * 2)   # Multiplication
    print(x ** 2)  # Exponentiation
```

```
4
2
6
9
```

```
[5] x += 1
    print(x)
    x *= 2
    print(x)
```

```
4
8
```

```
[6] y = 2.5
    print(type(y))
    print(y, y + 1, y * 2, y ** 2)
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation.

Booleans

- Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

```
[7] t, f = True, False
    print(type(t))
```

```
<class 'bool'>
```

```
[8] print(t and f) # Logical AND;
    print(t or f)  # Logical OR;
    print(not t)   # Logical NOT;
    print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings

```
[9] hello = 'hello'   # String literals can use single quotes
    world = "world"   # or double quotes; it does not matter
    print(hello, len(hello))
```

```
hello 5
```

```
[10] hw = hello + ' ' + world  # String concatenation
     print(hw)
```

```
hello world
```

```
[11] hw12 = '{} {} {}'.format(hello, world, 12)  # string formatting
     print(hw12)
```

```
hello world 12
```

String methods

- String objects have a bunch of useful methods

```
[12] s = "hello"
     print(s.capitalize())  # Capitalize a string
     print(s.upper())       # Convert a string to uppercase; prints "HELLO"
     print(s.rjust(7))      # Right-justify a string, padding with spaces
     print(s.center(7))     # Center a string, padding with spaces
     print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another
     print('  world '.strip())  # Strip leading and trailing whitespace
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

Exercises

1. Write a program that tells if a given string S1 is sub-string of a another string S.
2. Two words are anagrams if you can rearrange the letters from one to spell the other. Write a program that takes two strings and returns `True` if they are anagrams

You can find a list of all string methods in the documentation.

Containers

- Python includes several built-in container types:
    - **lists**
    - **dictionaries**
    - **sets**
    - **tuples**

Lists

A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

```
[13] xs = [3, 1, 2]   # Create a list
     print(xs, xs[2])
     print(xs[-1])     # Negative indices count from the end of the list; prints "2"
```

```
[3, 1, 2] 2
2
```

```
[14] xs[2] = 'foo'    # Lists can contain elements of different types
     print(xs)
```

```
[3, 1, 'foo']
```

```
[15] xs.append('bar') # Add a new element to the end of the list
     print(xs)
```

```
[3, 1, 'foo', 'bar']
```

```
[16] x = xs.pop()     # Remove and return the last element of the list
     print(x, xs)
```

```
bar [3, 1, 'foo']
```

As usual, you can find all the gory details about lists in the documentation.

Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

```
[17] nums = list(range(5))    # range is a built-in function that creates a list of integers
     print(nums)          # Prints "[0, 1, 2, 3, 4]"
     print(nums[2:4])     # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
     print(nums[2:])      # Get a slice from index 2 to the end; prints "[2, 3, 4]"
     print(nums[:2])      # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
     print(nums[:])       # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
     print(nums[:-1])     # Slice indices can be negative; prints ["0, 1, 2, 3]"
     nums[2:4] = [8, 9] # Assign a new sublist to a slice
     print(nums)          # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

Loops

You can loop over the elements of a list like this:

```
[18] animals = ['cat', 'dog', 'monkey']
     for animal in animals:
         print(animal)
```

```
cat
dog
monkey
```

Enumerate

If you want access to the **index** of each **element** within the body of a loop, use the built-in `enumerate` function:

```
[19] animals = ['cat', 'dog', 'monkey']
     for idx, animal in enumerate(animals):
         print(f"#{idx+1}: {animal}") # we used an f-string here
```

```
#1: cat
#2: dog
#3: monkey
```

Zip

Given two sequences $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, we may need to iterate over the pairs (a_i, b_i) with $a_i \in A, b_i \in B$

- we can "zip" the two "tapes" A and B to produce the interleaved sequence $((a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n))$: `zip()` returns an iterable object

```
[20] tuple(zip(range(3), ['fee', 'fi', 'fo', 'fum'])) # it stops zipping by exhausting the shortest
```

```
((0, 'fee'), (1, 'fi'), (2, 'fo'))
```

```
[ ] for a,b in [[1,'one'], [2, 'two'], [3, 'three']]: # iterating over pairs
        print(a,b)
    for num, flavour in zip( [1, 2, 3], ['sugar', 'spice', 'everything nice'] ):
        print(num, flavour)
```

```
1 one
2 two
3 three
1 sugar
2 spice
3 everything nice
```

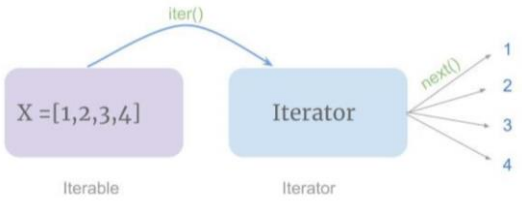
Iterables and iterators protocols

An **iterable** is an object, whose elements we can iterate over, eg. a list, a dictionary, ...

- it must implement the `__iter__()` method that returns an **iterator**
    - method invokable with `iter()` built-in function

An **iterator** is an iterable object having a method for "stepping forward" through the elements:

- the `__next__()` method, invokable with `next()` built-in function




```
[22] x = iter([1, 2, 3])
     print(x)
     next(x)

     <list_iterator object at 0x7f95eae96990>
     1
```

```
[23] next(x), next(x)

     (2, 3)
```

```
[31] next(x) # If iterator was already exhausted: next() raises the StopIteration exception!

     ---------------------------------------------------------------------------
     StopIteration                             Traceback (most recent call last)
     <ipython-input-31-0c05a801fdc8> in <module>()
     ----> 1 next(x) # If iterator was already exhausted: next() raises the StopIteration exception!

     StopIteration:

      SEARCH STACK OVERFLOW
```

Iterator object can be iterated over once only

```
[26] xnew = iter(x)
     list(xnew)

     []
```

Generators

Python functions may produce **many subsequent results**, instead of returning the data all at once, alternating between function's body execution and result production:

- using `yield` statements in place of `return` in the function body
- the function will actually return a **generator** object: a particular *iterator* object used to iterate through the many subsequent results
    - ⇒ we can use `next()` on the generator object
    - such iterator is "lazy": it yields one subsequent result upon calling `next()`, then it *stops*; and it *resumes* the function execution (till reaching the next `yield` statement) only when `next()` is **called again**, and so on.

```
[27] def foo():
         print("begin")
         for i in range(3):
             print("before yield", i)
             yield i
             print("after yield", i)
         print("end")
```

```
gen = foo()
next(gen)
```

```
begin
before yield 0
0
```

```
[28] next(gen)
```

```
after yield 0
before yield 1
1
```

```
[29] next(gen)
```

```
after yield 1
before yield 2
2
```

```
[32] next(gen) # the generator object was already exhausted
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-32-6491bbafdf39> in <module>()
----> 1 next(gen) # the generator object was already exhausted

StopIteration:
```

SEARCH STACK OVERFLOW

- the function containing `yield`, thus returning a generator object, is called a *generator function*. Eg. above `foo()` is a generator function

List comprehensions

- If we want to construct a sequence with a pattern, usually we need writing a loop
- For example: consider the following code that computes square numbers:

```
[33] nums = [0, 1, 2, 3, 4]
     squares = []
     for x in nums:
         squares.append(x ** 2)
     print(squares)
```

```
[0, 1, 4, 9, 16]
```

- You can make this code simpler using a **list comprehension** with "for" clause:

```
[34] nums = [0, 1, 2, 3, 4]
     squares = [x ** 2 for x in nums]
     print(squares)
```

```
[0, 1, 4, 9, 16]
```

- List comprehensions can also contain **conditions** with "if" clause:

```
[35] nums = [0, 1, 2, 3, 4]
     even_squares = [x ** 2 for x in nums if x % 2 == 0]
     print(even_squares)
```

```
[0, 4, 16]
```

Dictionaries

- A dictionary stores (key, value) pairs, similar to a Map in Java or an object in Javascript

```
[36] d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
     print(d['cat'])       # Get an entry from a dictionary; prints "cute"
     print('cat' in d)     # Check if a dictionary has a given key; prints "True"
```

```
cute
True
```

```
[37] d['fish'] = 'wet'    # Set an entry in a dictionary
     print(d['fish'])      # Prints "wet"
```

```
wet
```

```
[38] print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-38-78fc9745d9cf> in <module>()
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

SEARCH STACK OVERFLOW

```
[39] print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
     print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
```

```
N/A
wet
```

```
[40] del d['fish']        # Remove an element from a dictionary
     print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

You can find all you need to know about dictionaries in the documentation.

Iterate over dict

- It is easy to iterate over the keys in a dictionary:

```
[41] d = {'person': 2, 'cat': 4, 'spider': 8}
     for animal, legs in d.items():
         print('A {} has {} legs'.format(animal, legs))
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Dictionary comprehensions:

- These are similar to list comprehensions, but allow you to easily construct dictionaries

```
[42] nums = [0, 1, 2, 3, 4]
     even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
     print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

Sets

- A set is an unordered collection of distinct elements

```
[43] animals = {'cat', 'dog'}
     print('cat' in animals)   # Check if an element is in a set; prints "True"
     print('fish' in animals)  # prints "False"
```

```
True
False
```

```
[44] animals.add('fish')      # Add an element to a set
     print('fish' in animals)
     print(len(animals))       # Number of elements in a set;
```

```
True
3
```

```
[45] animals.add('cat')       # Adding an element that is already in the set does nothing
     print(len(animals))
     animals.remove('cat')    # Remove an element from a set
     print(len(animals))
```

```
3
2
```

Loops

- **Iterating** over a **set** has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set

```
[46] animals = {'cat', 'dog', 'fish'}
     for idx, animal in enumerate(animals):
         print('#{}: {}'.format(idx + 1, animal))
```

```
#1: dog
#2: fish
#3: cat
```

Set comprehensions

- Like lists and dictionaries, we can easily construct sets using set comprehensions

```
[47] from math import sqrt
     print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

Tuples

- A tuple is an (immutable) ordered list of values
- A tuple is in many ways similar to a list
- One of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot

```
[48] d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
     t = (5, 6)       # Create a tuple
     print(type(t))
     print(d[t])
     print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

Functions

- A function is a named sequence of statements that performs a computation
- When you define a function, you specify the name and the sequence of statements
- Later, you can “call” the function by name
- Python functions are defined using the `def` keyword

```
[51] def sign(x):
         if x > 0:
             return 'positive'
         elif x < 0:
             return 'negative'
         else:
             return 'zero'
```

Function calls

- Examples of a function call: specify the name, the arguments of the function (expression in parentheses)

```
[52] for x in [-1, 0, 1]:
         print(sign(x))
```

```
negative
zero
positive
```

```
[53] type(10)
```

```
int
```

Math functions

- Python has a math module that provides most of the familiar mathematical functions. A module is a file that contains a collection of related functions.
- Before we can use the functions in a module, we have to import it with an `import` statement

```
[54] import math  # this statement creates a module object named math

     ratio = 15.2
     decibels = 10 * math.log10(ratio)
     print('decibels: ', decibels)

     radians = 0.7
     height = math.sin(radians)
     print('height: ', height)
```

```
decibels:  11.818435879447726
height:  0.644217687237691
```

- We will often define functions to take optional keyword arguments

```
[55] def hello(name, loud=False):
         if loud:
             print('HELLO, {}'.format(name.upper()))
         else:
             print('Hello, {}!'.format(name))

     hello('Bob')
     hello('Fred', loud=True)
```

```
Hello, Bob!
HELLO, FRED
```

Exercises

1. Write a function that draws a grid like the following:

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

   Hint: to print more than one value on a line, you can print a comma-separated sequence of values: `print('+', '-')`. By default, print advances to the next line, but you can override that behavior and put a space at the end, like this:

```
print('+', end=' ')
print('-')
```

   The output of these statements is '+ -' on the same line. The output from the next print statement would begin on the next line.

2. Write a function that draws a similar grid with four rows and four columns.

Classes

- Python has built-in types but in many cases it is necessary to define new types
- As an example, we will create a type called Point that represents a `point` in two-dimensional space
- There are several ways we might represent points in Python:
    - We could store the coordinates separately in two variables, `x` and `y`
    - We could store the coordinates as elements in a list or tuple
    - We could create a new type to represent points as objects

```
[56] class Point:
         """Represents a point in 2-D space."""
```

- The class object is like a factory for creating objects
- To create a Point, you call Point as if it were a function

```
[57] # The return value is a reference to a Point object, which we assign to blank.
     blank = Point()
     blank
```

```
     <__main__.Point at 0x7f95eadcc790>
```

Attributes

- You can assign values to an instance using dot notation
- This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or string.whitespace
- we are assigning values to named elements of an object
- These elements are called **attributes**

```
[58] blank.x = 3.0
     blank.y = 4.0

     print(blank.x)
     print(blank.y)
```

```
     3.0
     4.0
```

Methods

- A method is a function that is associated with a particular class
- There are methods for strings, lists, dictionaries and tuples
- It is also possible to define methods for programmer-defined types
- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit
- The syntax for invoking a method is different from the syntax for calling a function

```
[59] class Point:
         """Represents a point in 2-D space

             attributes: x, y
         """
         def print_point(p):
             print('(%d, %d)' % (p.x, p.y))
```

- To represent a point, instantiate a Point object and assign values to the attrinutes

```
[60] p = Point()
     p.x = 3
     p.y = 5
     Point.print_point(p)
```

```
     (3, 5)
```

```
[61] # By convention, the first parameter of a method is called self,
     # so it would be more common to write print_time like this:

     class Point:
         """Represents a point in 2-D space

             attributes: x, y
         """
         def print_point(self):
             print('(%d, %d)' % (self.x, self.y))
```

The init method

- The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated
- Its full name is `__init__` (two underscore characters, followed by init, and then two more underscores).

```
[62] # stores the value of the parameter x, y as an attribute of self

     class Point:
         """Represents a point in 2-D space
            attributes: x, y
         """
         def __init__(self, x=0, y=0):
             self.x = x
             self.y = y
```

```
         def print_point(self):
             print('(%d, %d)' % (self.x, self.y))
```

```
[63] p = Point(x=3, y=5)
     p.print_point()
```

```
     (3, 5)
```

Instances as return values

- Functions can return instances
- For example, translate takes a `Point` as an argument and returns a new `Point` that contains the new coordinates

```
[64] def translate(p, x0, y0):
         x1 = p.x + x0
         y1 = p.y + y0
         return Point(x1, y1)

     # Here is an example that passes a Point as an argument and assigns the resulting Point to cente
     new_point = translate(p, 10, 20)
     new_point.print_point()
```

```
     (13, 25)
```

Exercices

- Write a definition for a class named 'Circle' with attributes center and radius, where center is a Point object and radius is a number
- Instantiate a Circle object that represents a circle with its center at (150, 100) and radius 75
- Write a function named point_in_circle that takes a Circle and a Point and returns True if the Point lies in or on the boundary of the circle.

Numpy

- Numpy is the core library for scientific computing in Python
- It provides a high-performance multidimensional array object, and tools for working with these arrays

```
[65] # To use Numpy, we first need to import the numpy package

     import numpy as np
```

Arrays

- A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers
- The number of dimensions is the rank of the array
- The shape of an array is a tuple of integers giving the size of the array along each dimension

- We can initialize numpy arrays from nested Python lists, and access elements using square brackets

```
[66] a = np.array([1, 2, 3])  # Create a rank 1 array
     print(type(a), a.shape, a[0], a[1], a[2])
     a[0] = 5                 # Change an element of the array
     print(a)

     <class 'numpy.ndarray'> (3,) 1 2 3
     [5 2 3]
```

```
[67] b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
     print(b)

     [[1 2 3]
      [4 5 6]]
```

```
[68] print(b.shape)
     print(b[0, 0], b[0, 1], b[1, 0])

     (2, 3)
     1 2 4
```

- Numpy also provides many functions to create arrays

```
[69] a = np.zeros((2,2))  # Create an array of all zeros
     print(a)

     [[0. 0.]
      [0. 0.]]
```

```
[70] b = np.ones((1,2))   # Create an array of all ones
     print(b)

     [[1. 1.]]
```

```
[71] c = np.full((2,2), 7) # Create a constant array
     print(c)

     [[7 7]
      [7 7]]
```

```
[72] d = np.eye(2)        # Create a 2x2 identity matrix
     print(d)

     [[1. 0.]
      [0. 1.]]
```

```
[73] e = np.random.random((2,2)) # Create an array filled with random values
     print(e)

     [[0.15203753 0.40237367]
      [0.04187215 0.28410145]]
```

Array indexing

- Slicing: Similar to Python lists, numpy arrays can be sliced.
- Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
[74] import numpy as np

     # Create the following rank 2 array with shape (3, 4)
     # [[ 1  2  3  4]
     #  [ 5  6  7  8]
     #  [ 9 10 11 12]]
     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

     # Use slicing to pull out the subarray consisting of the first 2 rows
     # and columns 1 and 2; b is the following array of shape (2, 2):
     # [[2 3]
     #  [6 7]]
     b = a[:2, 1:3]
     print(a)
     print(b)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[2 3]
 [6 7]]
```

- A slice of an array is a view into the same data, so modifying it will modify the original array.

```
[75] print(a[0, 1])
     b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
     print(a[0, 1])
```

```
2
77
```

- You can also mix integer indexing with slice indexing
- However, doing so will yield an array of lower rank than the original array

```
[76] # Create the following rank 2 array with shape (3, 4)
     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
     print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Accessing the data in the middle row

- Mixing integer indexing with slices yields an array of lower rank
- Using only slices yields an array of the same rank as the original array

```
[77] row_r1 = a[1, :]    # Rank 1 view of the second row of a
     row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
     row_r3 = a[[1], :]  # Rank 2 view of the second row of a
     print(row_r1, row_r1.shape)
     print(row_r2, row_r2.shape)
     print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

```
[78] # We can make the same distinction when accessing columns of an array:
     col_r1 = a[:, 1]
     col_r2 = a[:, 1:2]
     print(col_r1, col_r1.shape)
     print()
     print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing

- When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array
- In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array

```
[79] a = np.array([[1,2], [3, 4], [5, 6]])

     # An example of integer array indexing.
     # The returned array will have shape (3,) and
     print(a[[0, 1, 2], [0, 1, 0]])

     # The above example of integer array indexing is equivalent to this:
     print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
[1 4 5]
```

```
[80] # When using integer array indexing, you can reuse the same
     # element from the source array:
     print(a[[0, 0], [1, 1]])

     # Equivalent to the previous integer array indexing example
     print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

- One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix

```
[81] # Create a new array from which we will select elements
     a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
     print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
[82] # Create an array of indices
     b = np.array([0, 2, 0, 1])

     # Select one element from each row of a using the indices in b
     print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[ 1  6  7 11]
```

```
[83] # Mutate one element from each row of a using the indices in b
     a[np.arange(4), b] += 10
     print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing

- Boolean array indexing lets you pick out arbitrary elements of an array
- Frequently this type of indexing is used to select the elements of an array that satisfy some condition

```
[84] import numpy as np

     a = np.array([[1,2], [3, 4], [5, 6]])

     bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                         # this returns a numpy array of Booleans of the same
                         # shape as a, where each slot of bool_idx tells
                         # whether that element of a is > 2.

     print(bool_idx)
```

```
     [[False False]
      [ True  True]
      [ True  True]]
```

```
[85] # We use boolean array indexing to construct a rank 1 array
     # consisting of the elements of a corresponding to the True values
     # of bool_idx
     print(a[bool_idx])

     # We can do all of the above in a single concise statement:
     print(a[a > 2])
```

```
     [3 4 5 6]
     [3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

Datatypes

- Every numpy array is a grid of elements of the same type
- Numpy provides a large set of numeric datatypes that you can use to construct arrays
- Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype

```
[86] x = np.array([1, 2])  # Let numpy choose the datatype
     y = np.array([1.0, 2.0])  # Let numpy choose the datatype
     z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

     print(x.dtype, y.dtype, z.dtype)
```

```
     int64 float64 int64
```

You can read all about numpy datatypes in the documentation.

Array math

- Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module

```
[87] x = np.array([[1,2],[3,4]], dtype=np.float64)
     y = np.array([[5,6],[7,8]], dtype=np.float64)

     # Elementwise sum; both produce the array
     print(x + y)
     print(np.add(x, y))
```

```
     [[ 6.  8.]
      [10. 12.]]
     [[ 6.  8.]
      [10. 12.]]
```

```
[88] # Elementwise difference; both produce the array
     print(x - y)
     print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
[89] # Elementwise product; both produce the array
     print(x * y)
     print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
[90] # Elementwise division; both produce the array
     # [[ 0.2         0.33333333]
     #  [ 0.42857143  0.5       ]]
     print(x / y)
     print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
[91] # Elementwise square root; produces the array
     # [[ 1.          1.41421356]
     #  [ 1.73205081  2.        ]]
     print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Dot operation

- The `dot` function compute the inner products of vectors

```
[92] x = np.array([[1,2],[3,4]])
     y = np.array([[5,6],[7,8]])

     v = np.array([9,10])
     w = np.array([11, 12])

     # Inner product of vectors; both produce 219
     print(v.dot(w))
     print(np.dot(v, w))
```

```
219
219
```

Note

Unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects

- You can also use the `@` operator which is equivalent to numpy's `dot` operator.

```
[93] print(v @ w)
```

```
219
```

```
[94] # Matrix / vector product; both produce the rank 1 array [29 67]
     print(x.dot(v))
     print(np.dot(x, v))
     print(x @ v)
```

```
[29 67]
[29 67]
[29 67]
```

```
[95] # Matrix / matrix product; both produce the rank 2 array
     # [[19 22]
     #  [43 50]]
     print(x.dot(y))
     print(np.dot(x, y))
     print(x @ y)
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

- Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`

```
[96] x = np.array([[1,2],[3,4]])

     print(np.sum(x))  # Compute sum of all elements; prints "10"
     print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
     print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Reshape

- Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays
- The simplest example of this type of operation is transposing a matrix
    - to transpose a matrix, simply use the T attribute of an array object:

```
[97] print(x)
     print("transpose\n", x.T)
```

```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

```
[98] v = np.array([[1,2,3]])
     print(v )
     print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations
- Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.
- For example, suppose that we want to add a constant vector to each row of a matrix

```
[99] # We will add the vector v to each row of the matrix x,
     # storing the result in the matrix y
     x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
     v = np.array([1, 0, 1])
     y = np.empty_like(x)   # Create an empty matrix with the same shape as x

     # Add the vector v to each row of the matrix x with an explicit loop
     for i in range(4):
         y[i, :] = x[i, :] + v

     print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

NOTE

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`

- Adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv

```
[100] vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
      print(vv)                # Prints "[[1 0 1]
                               #          [1 0 1]
                               #          [1 0 1]
                               #          [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

```
[101] y = x + vv  # Add x and vv elementwise
      print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

- Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v

```
[102] import numpy as np

      # We will add the vector v to each row of the matrix x,
      # storing the result in the matrix y
      x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
      v = np.array([1, 0, 1])
      y = x + v  # Add v to each row of x using broadcasting
      print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

NOTE

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

- Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible

[103]
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

[104]
```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

[105]
```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

[106]
```
# Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

[107]
```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

Exercices

- Consider a random array with shape (100,2) representing coordinates, find point by point distances
- Considering a 2-dimensional array (matrix), how to get sum over the columns at once?
- Consider an arbitrary random array: compute its mean, median, and the standard deviation
- Compute a matrix rank (hint: `U, S, V = np.linalg.svd(Mat) # Singular Value Decomposition`)

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

Matplotlib

- Matplotlib is a plotting library
- In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB

```
[108] import matplotlib.pyplot as plt
```

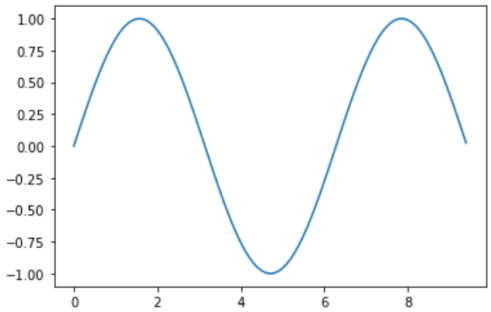
- By running this special iPython command, we will be displaying plots inline:

```
[109] %matplotlib inline
```

Plotting

- The most important function in `matplotlib` is plot, which allows you to plot 2D data

```
[110] # Compute the x and y coordinates for points on a sine
      x = np.arange(0, 3 * np.pi, 0.1)
      y = np.sin(x)

      # Plot the points using matplotlib
      plt.plot(x, y)
      plt.show()
```



- With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

```
[111] y_sin = np.sin(x)
      y_cos = np.cos(x)

      # Plot the points using matplotlib
      plt.plot(x, y_sin)
      plt.plot(x, y_cos)
      plt.xlabel('x axis label')
      plt.ylabel('y axis label')
      plt.title('Sine and Cosine')
      plt.legend(['Sine', 'Cosine'])
      plt.show()
```