Part 1. AI/ML foundation

1.1: AI/ML overview

Insop, 7/23/2024, Ver 0.4

AI learning modules

Part 1 AI/ML foundation
1.1 AI/ML overview

We're here

Part 2 Generative AI
2.1: Generative AI/LLM overview
2.2: LLM applications

Part 3 AI/ML systems
3.1: AI/ML systems overview
3.2: AI/ML systems optimization

1.1: AI/ML overview

Why do we need to know?

- AI/ML are used in everywhere…
- Core ideas are simple and knowing these will help us to use AI tools
- There are many AI algorithms and knowing what they are helpful us to know when to use
- Many of ideas can be used as lego blocks, knowing core blocks will help us to build larger systems
- Despite that there are well established frameworks (hw/sw), it is still very flexible fields and moving fast

Plan for today

Motivational example

Three key ingredients in ML

- Hypothesis
- Loss
- Optimization

Neural network

CNN

Word embedding

RNN

Terminology

AI (artificial intelligence): A set techniques used to try to imitate human intelligence. Field of study to improve automated performance on tasks that are required cognition to solve.

- What is this written character?
- How do I navigate to the office by noon?

ML (machine learning): Using large amounts of data, machines learn without being explicitly programmed

- ML relies on *statistics* that models which properties of the inputs are associated with the expected outputs
- Supervised learning: train model using inputs with true labels, such as classification
- Unsupervised learning: learn the characteristics of input data without explicit labels, such as clustering

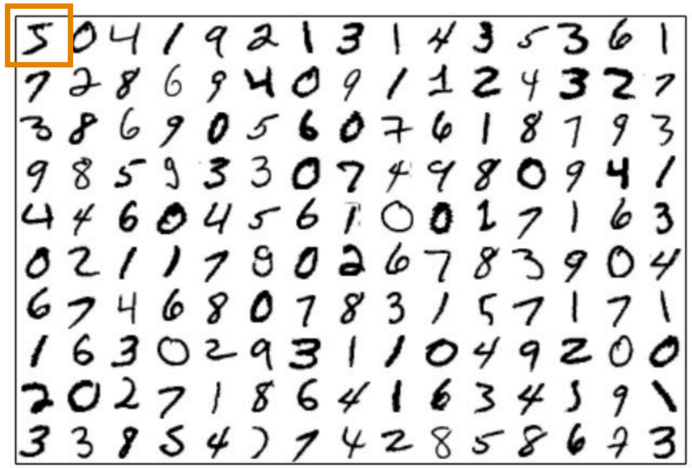
Generative AI: Extending machine learning to create text, images, video, audio and other content.

- ChatGPT
- StableDiffusion model generate images based on text

Ref: [4]

ML learns from training dataset

Example: write a program that recognizes handwritten digits in 0 through 9 *categories*



MNIST Dataset

Dataset consist of digit images and true digit labels

Train a model with example data so that after the training the trained model can predicts the output given *unseen* inputs

The *supervised* ML prepare a *training dataset* with true *labels* and feed these into ML algorithm. After the training process, the ML model can automatically solve the task

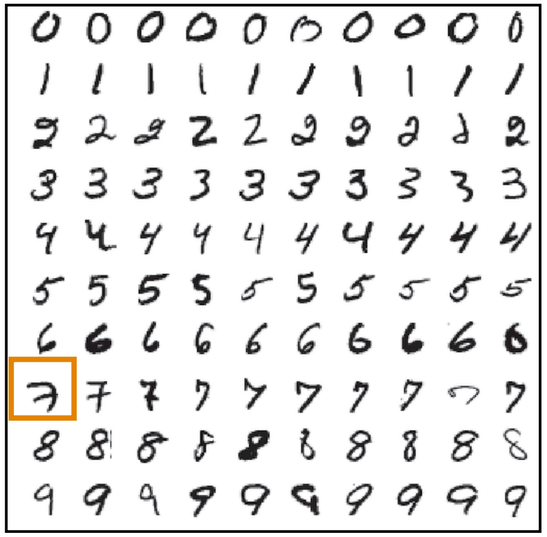
Image source: [2]

Example digit recognition

An example input image size of 28x28 with grey (8bit) scale and flatten to a single array (vector),

$$28 \cdot 28 = 784$$

Inputs to ML training is a form of vector or matrix, as in here 784x1 vector.

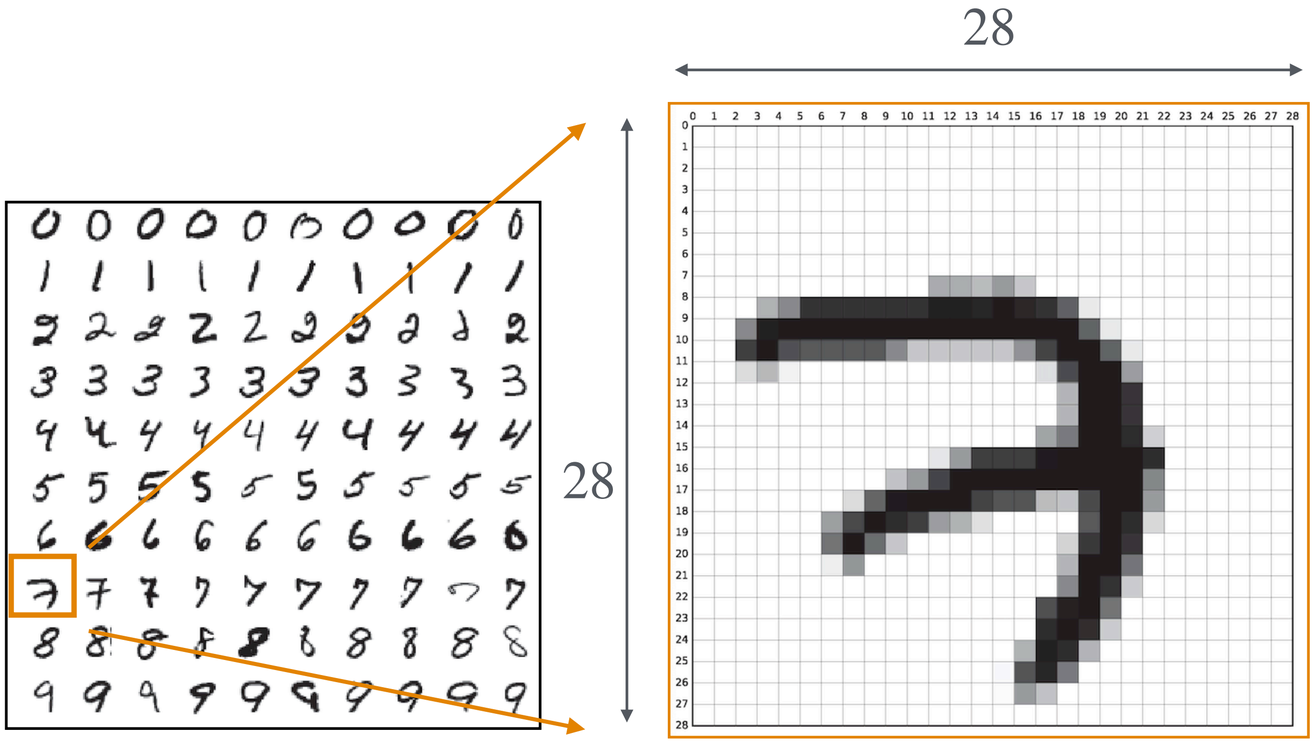


Example digit recognition

An example input image size of 28x28 with grey (8bit) scale and flatten to a single array (vector),

$$28 \cdot 28 = 784$$

Inputs to ML training is a form of vector or matrix, as in here 784x1 vector.

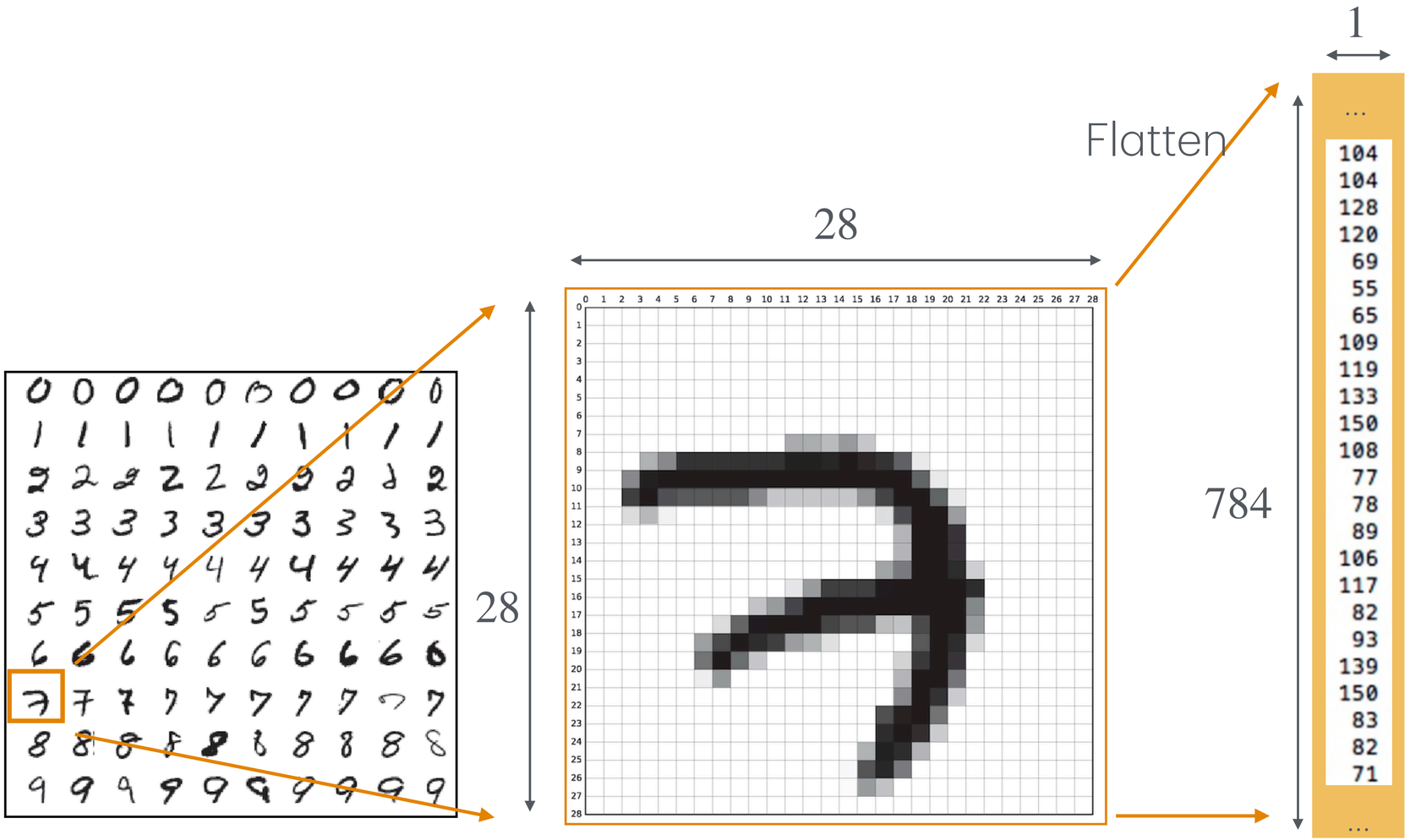


Example digit recognition

An example input image size of 28x28 with grey (8bit) scale and flatten to a single array (vector),

$$28 \cdot 28 = 784$$

Inputs to ML training is a form of vector or matrix, as in here 784x1 vector.



ML learns from training dataset

Training time: feed dataset and learn

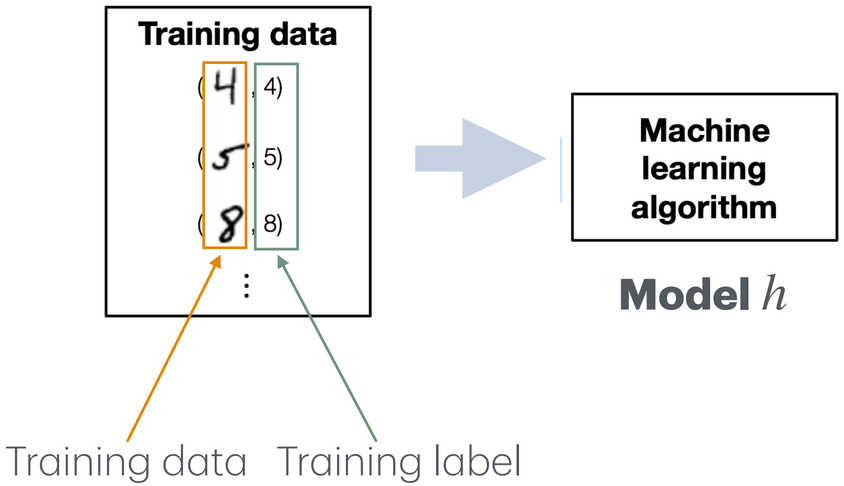
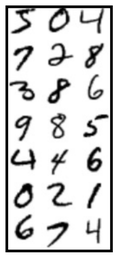


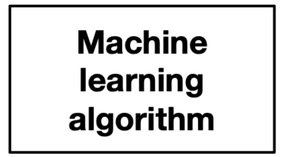
Image source: [2]

ML learns from training dataset

Inference time: automatically predicts



Unseen digits



Model h

Image source: [2]

ML learns from training dataset

Inference time: automatically predicts

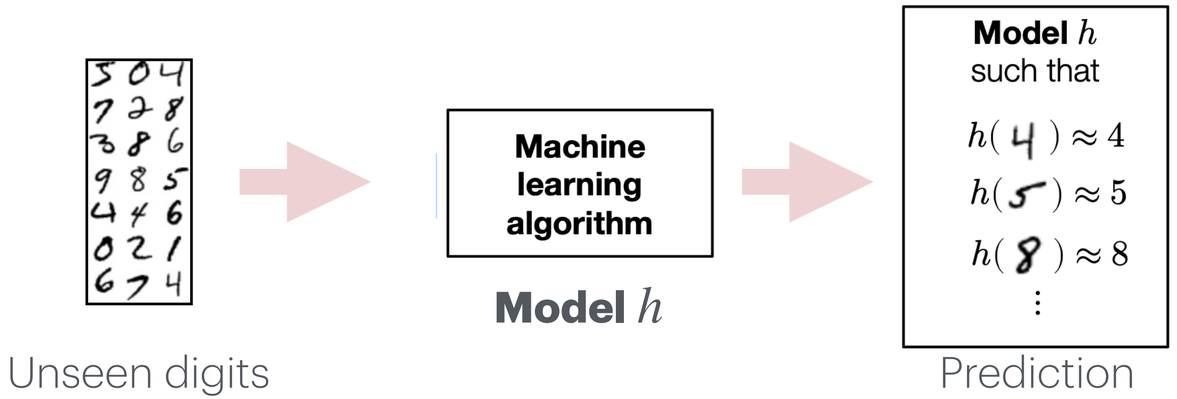


Image source: [2]

Three key ingredients in ML

Three key ingredients in ML

Every ML algorithm consists of the following three key ingredients

1. hypothesis: ML algorithm (model), how to prepare *parameters* that can map inputs (digit images) to outputs (class labels)

2. Loss: a function that measures how *well* a given hypothesis (i.e., a choice of parameters) performs on the task of interest

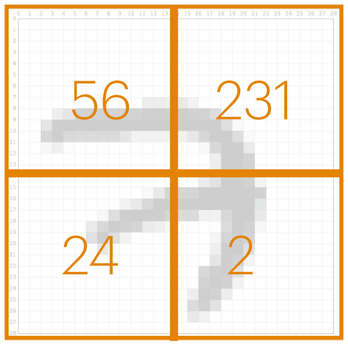
3. Optimization: a procedure for determining a set of parameters that (approximately) *minimizes* the sum of *losses* over the training dataset

Ref: [2]

Linear hypothesis function

Linear classifier

A *motivational* example with an image with 4 pixels, and 3 digit classes (7,8,9)



Input image

Machine learning algorithm

Model h

"7" score

"8" score

"9" score

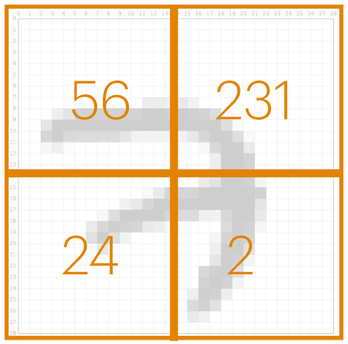
Output scores

Adapted from: [3]

Linear hypothesis function

Linear classifier

A *motivational* example with an image with 4 pixels, and 3 digit classes (7,8,9)



Input image

$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 20 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0.0 & 0.25 & 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} + \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix} =$$

W, weights

Flattened input

b, bias

"7" score

"8" score

"9" score

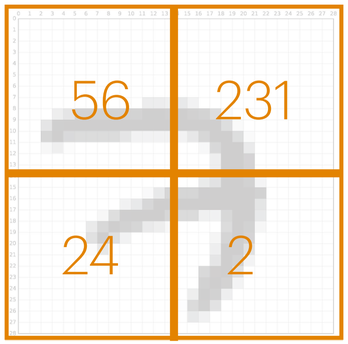
Flatten images to input vector

Adapted from: [3]

Linear hypothesis function

Linear classifier

A *motivational* example with an image with 4 pixels, and 3 digit classes (7,8,9)



Input image

$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 20 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0.0 & 0.25 & 0.2 & -0.3 \end{bmatrix}$$

W, weights

$$\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}$$

Flattened input

$$+ \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}$$

b, bias

$$= \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}$$

Output scores

"7" score

"8" score

"9" score

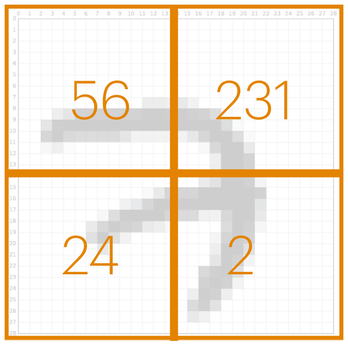
Flatten images to input vector

Adapted from: [3]

Linear hypothesis function

Linear classifier

A *motivational* example with an image with 4 pixels, and 3 digit classes (7,8,9)



Input image

$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 20 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0.0 & 0.25 & 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} + \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix} = \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}$$

"7" score
"8" score
"9" score

W, weights

Flattened input

b, bias

Output scores

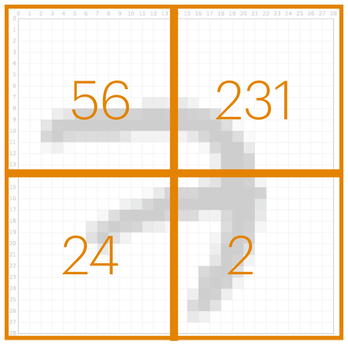
Flatten images to input vector

Adapted from: [3]

Linear hypothesis function

Linear classifier

A *motivational* example with an image with 4 pixels, and 3 digit classes (7,8,9)



Input image

$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 20 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0.0 & 0.25 & 0.2 & -0.3 \end{bmatrix}$$

W, weights

$$\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}$$

Flattened input

$$+ \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix} = \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}$$

b, bias

Output scores

"7" score

"8" score

"9" score

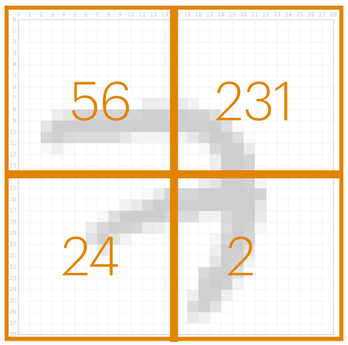
Flatten images to input vector

Adapted from: [3]

Linear hypothesis function

Linear classifier

A *motivational* example with an image with 4 pixels, and 3 digit classes (7,8,9)



Input image

$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 20 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0.0 & 0.25 & 0.2 & -0.3 \end{bmatrix}$$

W, weights

$$\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}$$

Flattened input

$+$

$$\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}$$

b, bias

$=$

$$\begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}$$

Output scores

"7" score

"8" score

"9" score

Flatten images to input vector

Adapted from: [3]

Linear hypothesis function

Linear classifier

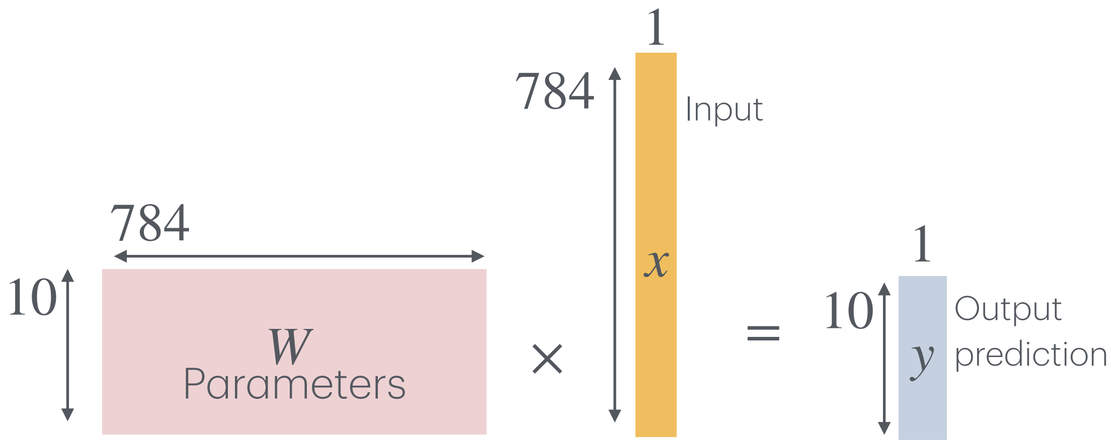
Linear hypothesis function uses a linear equation (matrix multiplication) to predict class scores

Example of hand-written digit classification:

- Training data: 28x28 pixel MNIST digits, $x^{(i)} \in \mathbb{R}^n, n = 28 \cdot 28 = 784$
- 10 digit classification, $y^{(i)} \in \{1,..k\}, k = 10$
- Number of data in dataset, $i = 1,...,m$

A linear hypothesis function maps inputs to outputs with this transformation

- $h(x) = Wx$, for parameters $W \in \mathbb{R}^{k \times n}$
- Wx : matrix multiplication of W and x shown below



Ref: [2]

Loss

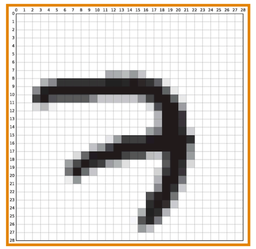
A *motivational* example with an image with 4 pixels, and *3 digit classes* (7,8,9).

We want to find a better representation of goodness from raw score.

$$\mathbb{P}(\text{label} = y) = \frac{\exp(h_y(x))}{\sum_j \exp(h_j(x))}$$

Softmax

Input



“7” score	3.2
“8” score	5.1
“9” score	-1.7

Raw score

Adapted from: [3]

Loss

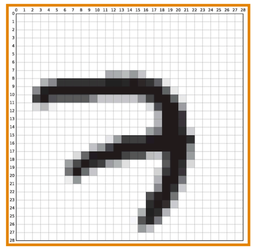
A *motivational* example with an image with 4 pixels, and *3 digit classes* (7,8,9).

We want to find a better representation of goodness from raw score.

$$\mathbb{P}(\text{label} = y) = \frac{\exp(h_y(x))}{\sum_j \exp(h_j(x))}$$

Softmax

Input



"7" score	3.2
"8" score	5.1
"9" score	-1.7

Raw score

Apply exponential →

Probability must be >=0

	24.5
	164.0
	0.8

Unnormalized probabilities

Adapted from: [3]

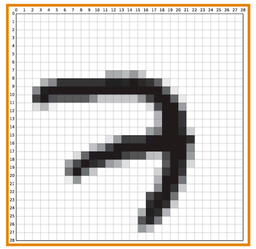
Loss

A *motivational* example with an image with 4 pixels, and *3 digit classes* (7,8,9).

We want to find a better representation of goodness from raw score.

$$\mathbb{P}(\text{label} = y) = \frac{\exp(h_y(x))}{\sum_j \exp(h_j(x))}$$

Softmax



Input

	Raw score
"7" score	3.2
"8" score	5.1
"9" score	-1.7

Apply exponential

Probability must be >=0

	Unnormalized probabilities
	24.5
	164.0
	0.8

Normalize

Probabilities must sum to 1

	probabilities
	0.13
	0.87
	0.00

Adapted from: [3]

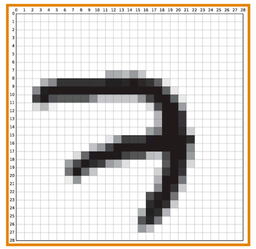
Loss

A *motivational* example with an image with 4 pixels, and *3 digit classes* (7,8,9).

We want to find a better representation of goodness from raw score.

$$\mathbb{P}(\text{label} = y) = \frac{\exp(h_y(x))}{\sum_j \exp(h_j(x))}$$

Softmax



Input

	Raw score
"7" score	3.2
"8" score	5.1
"9" score	-1.7

Apply exponential →

Probability must be >=0

	Unnormalized probabilities
	24.5
	164.0
	0.8

Normalize →

Probabilities must sum to 1

	probabilities
	0.13
	0.87
	0.00

Compare

True labels

	Correct probabilities
	1.00
	0.00
	0.00

Adapted from: [3]

Loss

$\ell(h(x), y)$: A loss function defines how good our hypothesis (model) classifier is

Softmax: converts the hypothesis classifier output to "probability". We can do this by exponentiating and normalizing its entries (making them all positive and sum to one). Note, exp(. .) is exponential function, $e^{(..)}$.

$$\text{softmax}(h(x)) = \mathbb{P}(\text{label} = y) = \frac{\exp(h_y(x))}{\sum_j \exp(h_j(x))}$$

Exponentiation makes anything positive

Normalize over entire h to give probability distribution

Cross-entropy loss: negative log probability (softmax) of the true label (class)

$$\ell_{ce}(h(x), y) = -\log \mathbb{P}(\text{label} = y) = -\log\left(\frac{\exp(h_y(x))}{\sum_j \exp(h_j(x))}\right) = -h_y(x) + \log \sum_j \exp(h_j(x))$$

Ref: [2]

Optimization

Motivational idea, optimization finds a way to achieve the minimum loss

Image source: link

Optimization

Motivational idea, optimization finds a way to achieve the minimum loss.

Follow the descending slop...

Image source: link

Optimization

Motivational idea, optimization finds a way to achieve the minimum loss.

Follow the descending slop...

Image source: link

Optimization

Third ingredient of ML is *optimization* that finds parameters that *minimize* the *loss*. The optimization is a method that minimizes the average loss on the training dataset.

$\min_W (\dots)$ finds W that minimizes $(\dots)$. m is number of training dataset.

$$\min_W \frac{1}{m} \sum_{i=1}^{m} \ell_{ce}(h(x^{(i)}), y^{(i)})$$

Ref: [2]

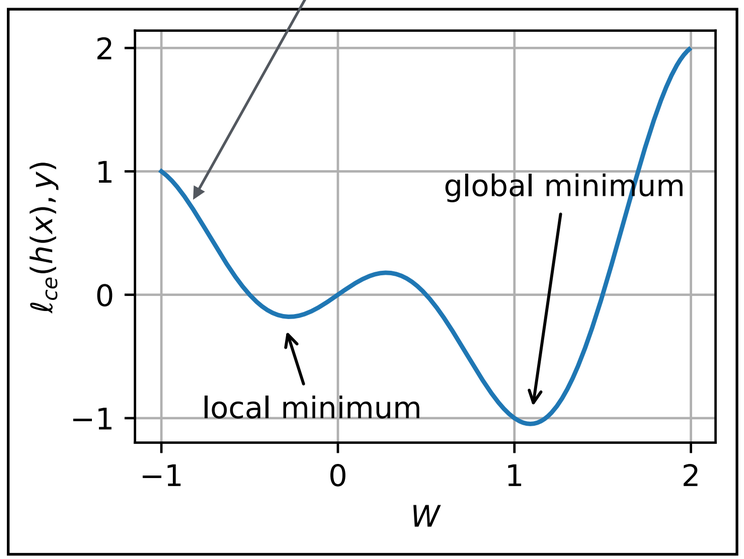
Optimization

Third ingredient of ML is *optimization* that finds parameters that *minimize* the *loss*. The optimization is a method that minimizes the average loss on the training dataset.

$\min_W (\dots)$ finds W that minimizes $(\dots)$. m is number of training dataset.

$$\min_W \frac{1}{m} \sum_{i=1}^{m} \ell_{ce}(h(x^{(i)}), y^{(i)})$$

A simplified loss curve in 1 dimensional space, practical loss curves are in multidimensional space



W is vector, but shown in as 1 dim for simplicity

Ref: [2]

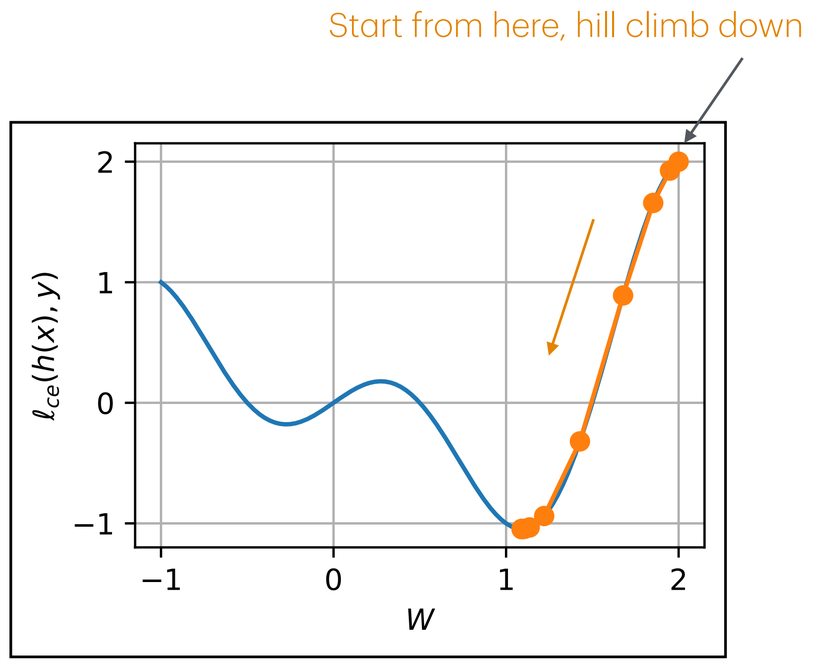
Gradient descent

Let's use a simple view of how to find minimum value using the following. The figure shows loss value on the y axis and shows W on the x axis. W is high dimensional vector but only shows as 1 dimensional number for simplicity.

- To find minimum loss, we need to follow the slop of the loss curve
- Derivatives of function defines the direction that most increases the function. So we use the negative of the derivative of the function, then we will find the direction that most decreases the function (loss function).

Gradient descent algorithm: iteratively taking steps in the direction of the negative gradient. α is *learning rate* (step) adjust the changes, ∇_W is gradient, which defines matrix of partial derivatives

$$W \leftarrow W - \alpha \nabla_W(\ell_{ce}(h(x), y))$$



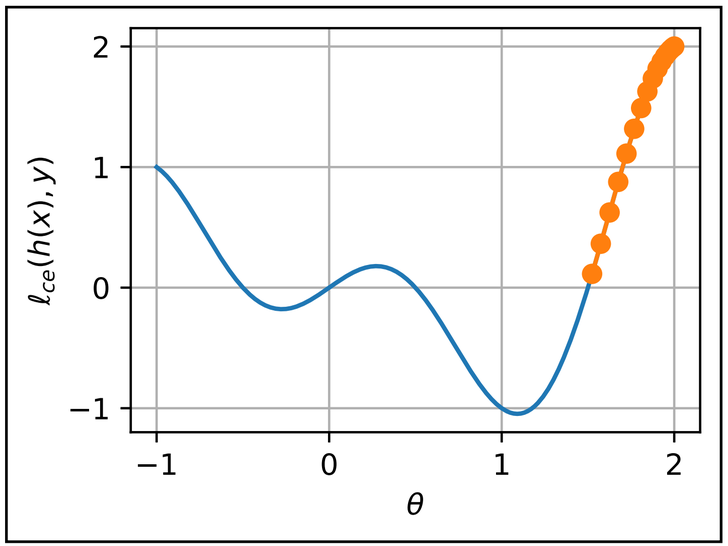
Gradient descent

Learning rate

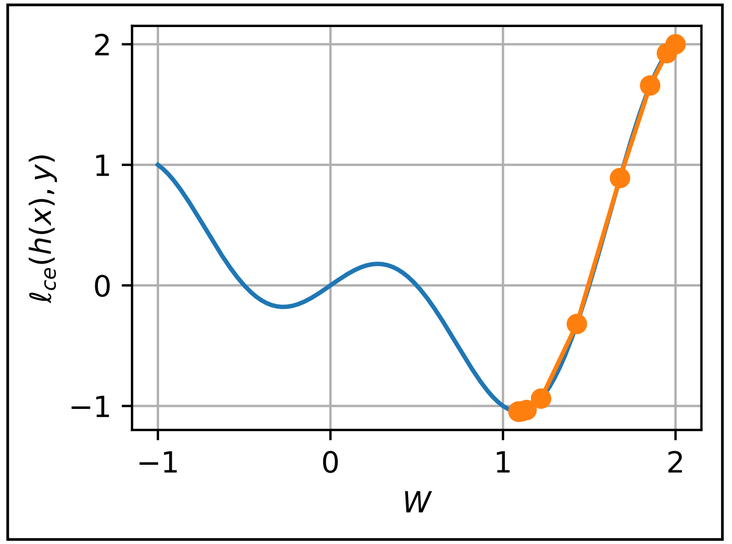
Learning rate, α, determine how we update the weights.

It matters in gradient descent, if it too small, convergence is too slow, and if it too large, then it may not converge.

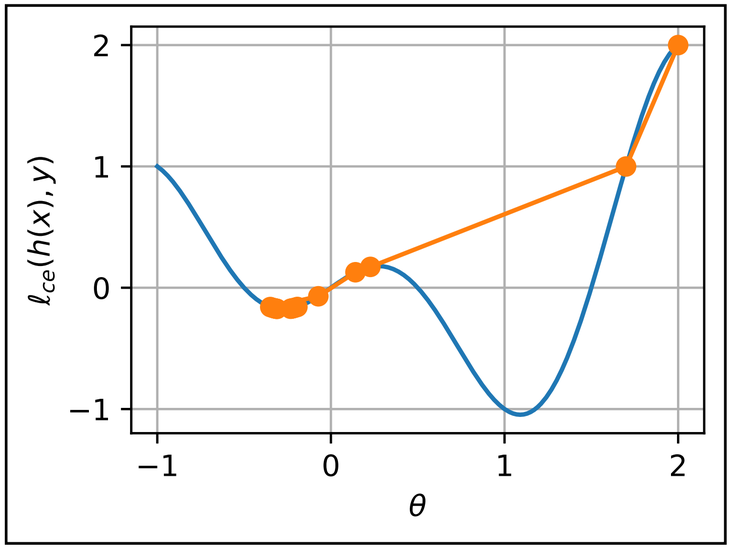
$$W \leftarrow W - \alpha \nabla_W(\ell_{ce}(h(x), y))$$



Learning rate=0.01, slow convergence



Learning rate=0.05, converging

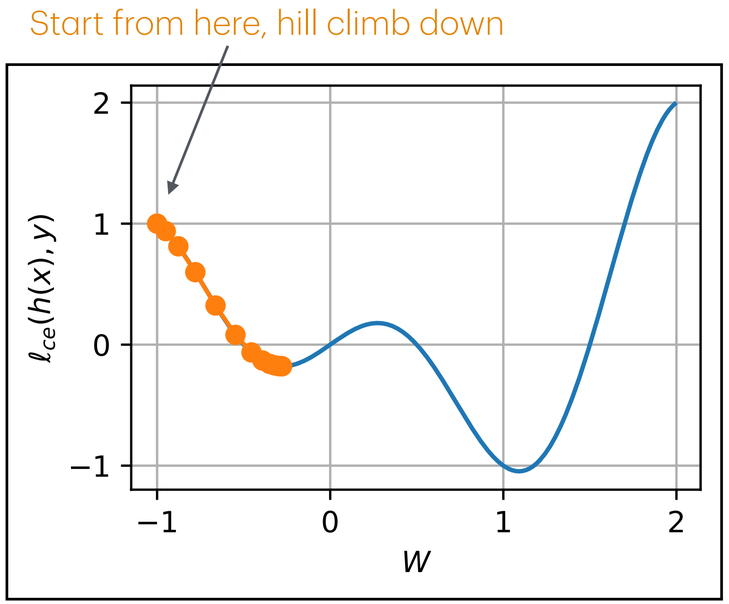


Learning rate=0.3, failed to converge

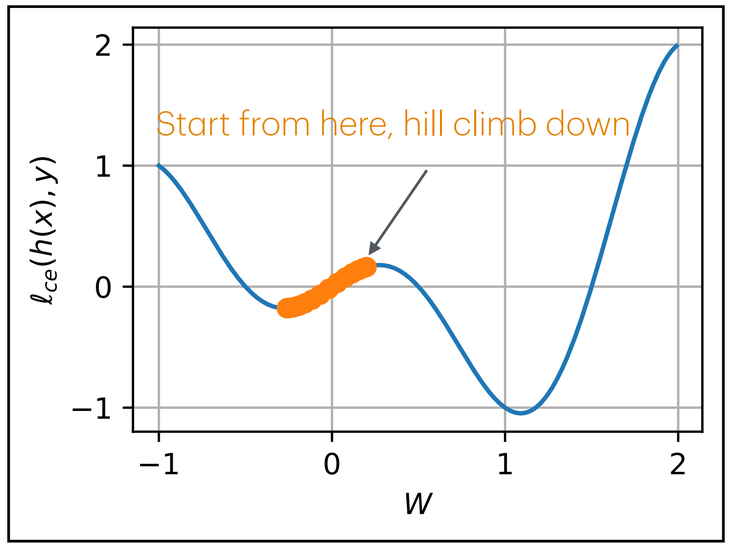
Gradient descent

Weight initialization

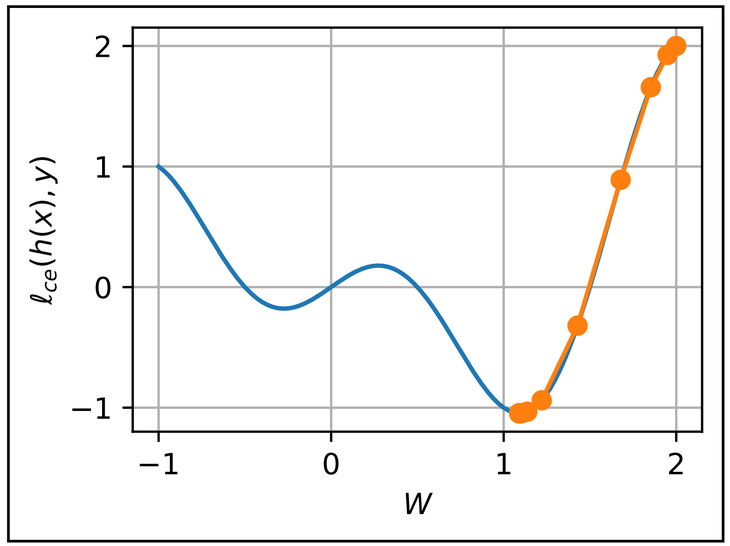
What values to initialize weight also matters. The following figures show with different initial values for W with the learning rate 0.05.



Initial value = -1, stuck in local minimum



Initial value = 0.05, stuck in local minimum



Initial value = 2, found global minimum

Stochastic gradient descent

Gradient descent uses all training examples to compute the gradient and update the weight. This can be too expensive to compute and slow to converge.

```
# Gradient descent
# init weights
while True:
    # evaluate gradient for all data points
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    # update weights,
    weights += - learning_rate * weights_grad
```

Stochastic gradient descent

Gradient descent uses all training examples to compute the gradient and update the weight. This can be too expensive to compute and slow to converge.

Stochastic gradient descent takes many gradient steps for mini-batch (small partition) of training datasets

```
# Gradient descent
# init weights
while True:
    # evaluate gradient for all data points
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    # update weights,
    weights += - learning_rate * weights_grad
```

```
# Stochastic gradient descent
# init weights
while True:
    # sample mini-batch of data
    data_batch = sample_training_data(data, 256)
    # evaluate gradient for mini-batch
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    # update weights
    weights += - learning_rate * weights_grad
```

Neural network

Linear hypothesis (classifier)

Linear hypothesis maps inputs (x) in $\mathbb{R}^n$ to outputs in $\mathbb{R}^k$ linearly

$$h(x) = Wx, \quad W \in \mathbb{R}^{k \times n}$$

This model uses k linear classifier that predicts the class with the largest values. Essentially partitioning the input into k linear reasons with class.

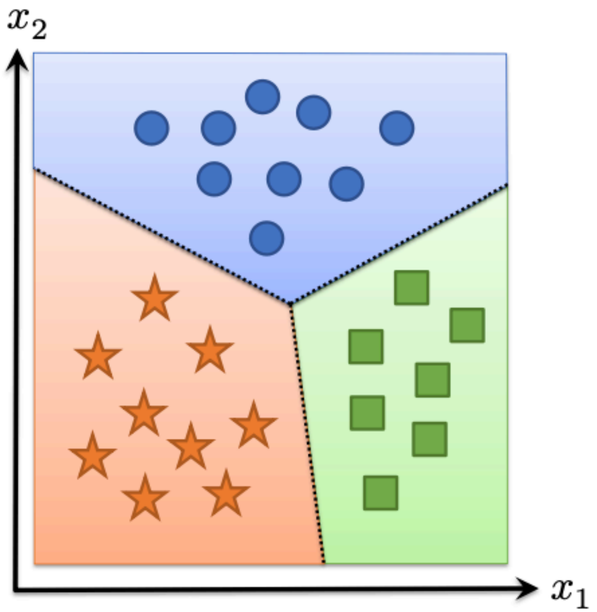


Image source: [2]

Nonlinear classification boundary

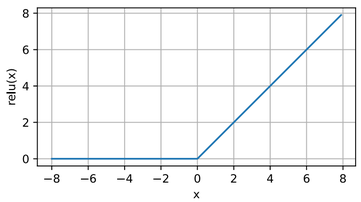
When we have data that cannot be separated by a set of linear classifier functions?

We need a way to separate these data using a nonlinear set of classification boundaries.

Apply a non-linear function, $\phi(\,.\,)$, to element-wise to linear hypothesis (features)

$$\phi(x) = \sigma(Wx), \;\; W \in \mathbb{R}^{k \times n}$$

Example: $\phi(x) = \max(Wx, 0)$



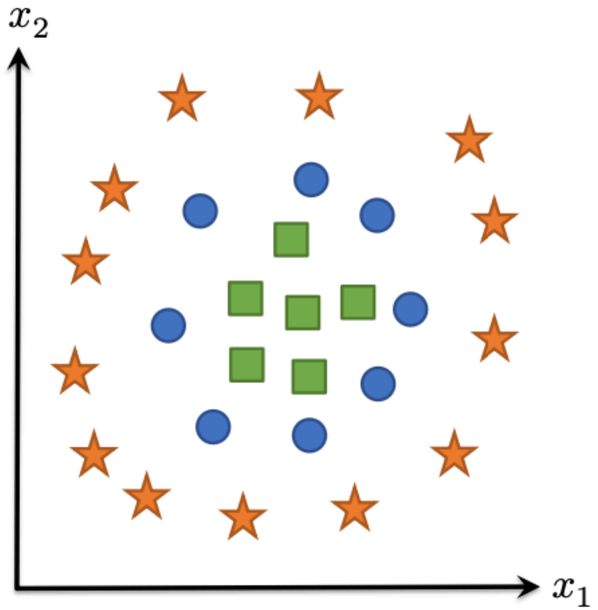
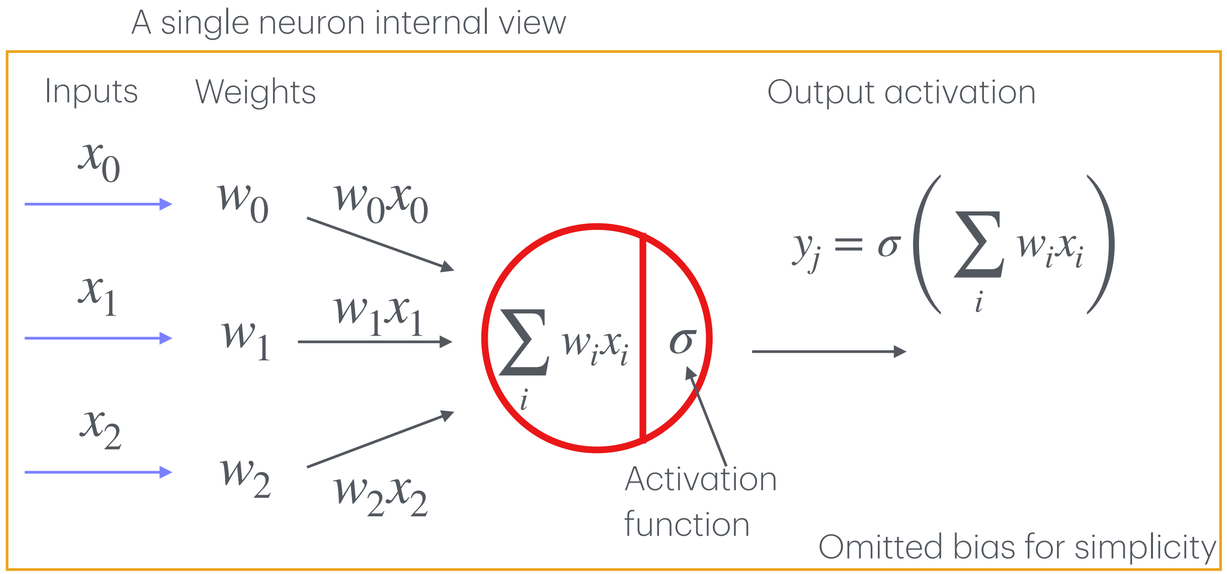


Image source: [2]

Neural network

Neural network is a model consist of *"fully" connected* layers with non-linear function called activation function.

- A neuron combines all inputs with weights and apply activation function
- Also called as deep learning when there is large number layers are used.
- Most widely used ML model type, i.e. a useful lego block

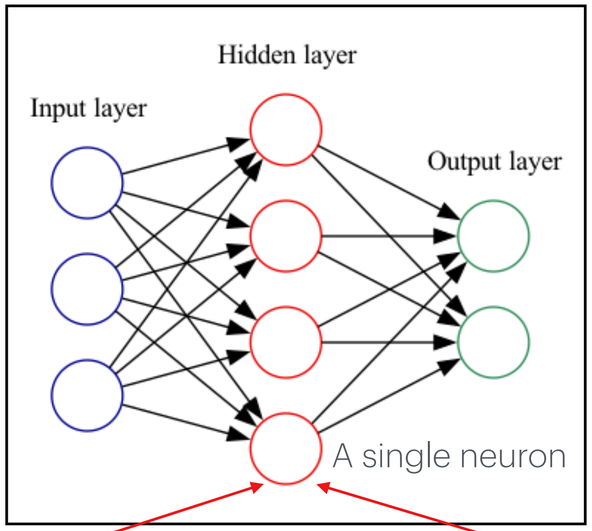


Neural network

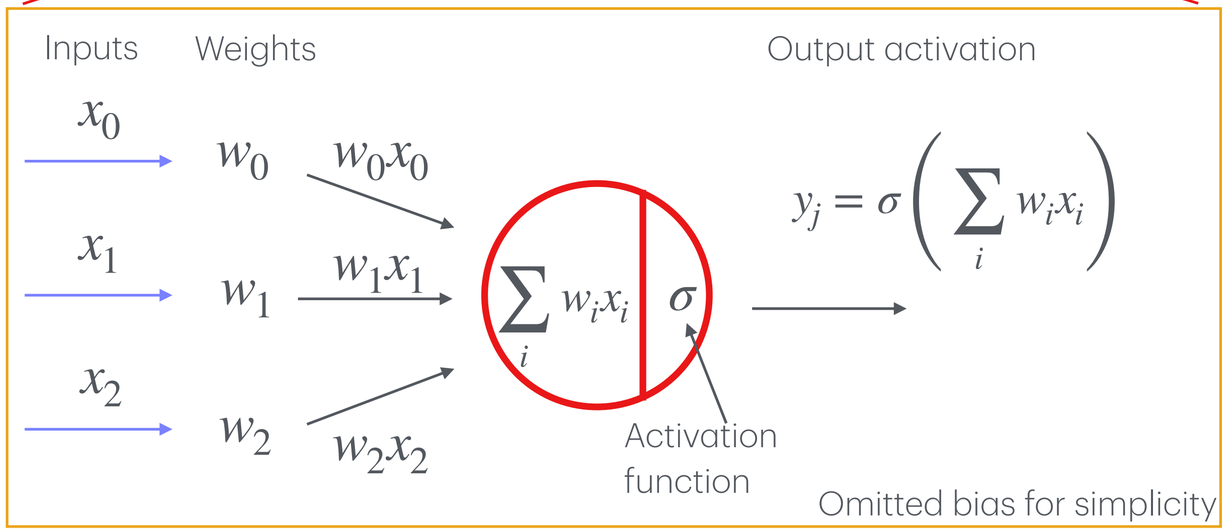
Neural network is a model consist of *"fully" connected* layers with non-linear function called activation function.

- A neuron combines all inputs with weights and apply activation function
- Also called as deep learning when there is large number layers are used.
- Most widely used ML model type, i.e. a useful lego block

2 layers neural network with 1 hidden layer

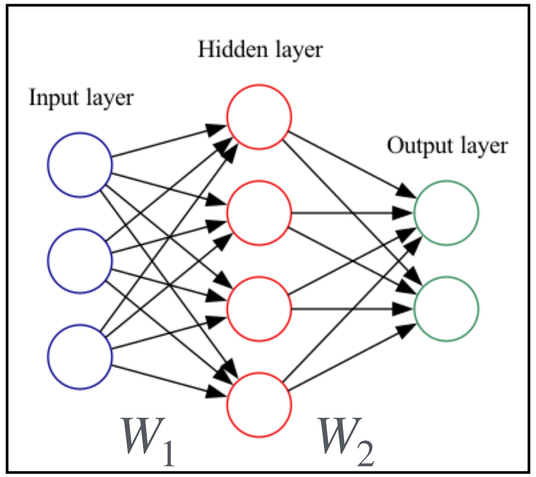


A single neuron internal view



Parameters and computation of FC

FC (fully connected) neural network



2 layer neural network

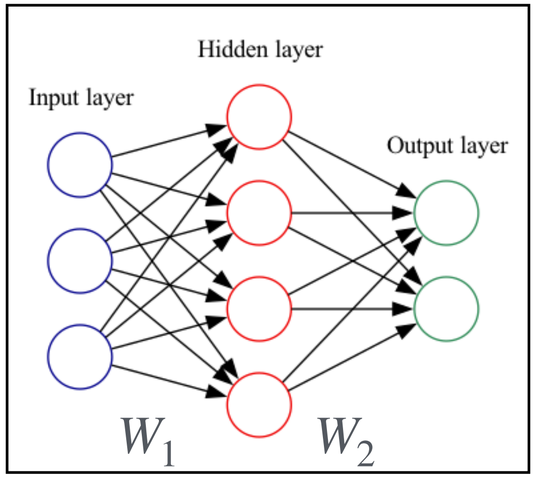
$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{4\times 3}, W_2 \in \mathbb{R}^{2\times 4}$$

$$x \in \mathbb{R}^3, y \in \mathbb{R}^2$$

Parameters and computation of FC

FC (fully connected) neural network



2 layer neural network

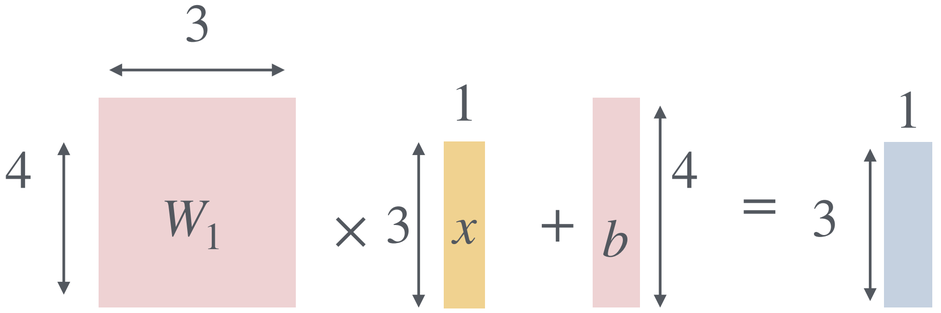
$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{4\times 3}, W_2 \in \mathbb{R}^{2\times 4}$$

$$x \in \mathbb{R}^3, y \in \mathbb{R}^2$$

Matrix multiplication: $W_1 x$

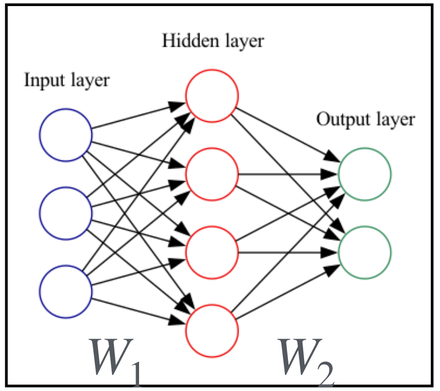
$$\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \\ w_{30} & w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$



Simple representation of matmul with showing dimensions

Parameters and computation of FC

FC (fully connected) neural network

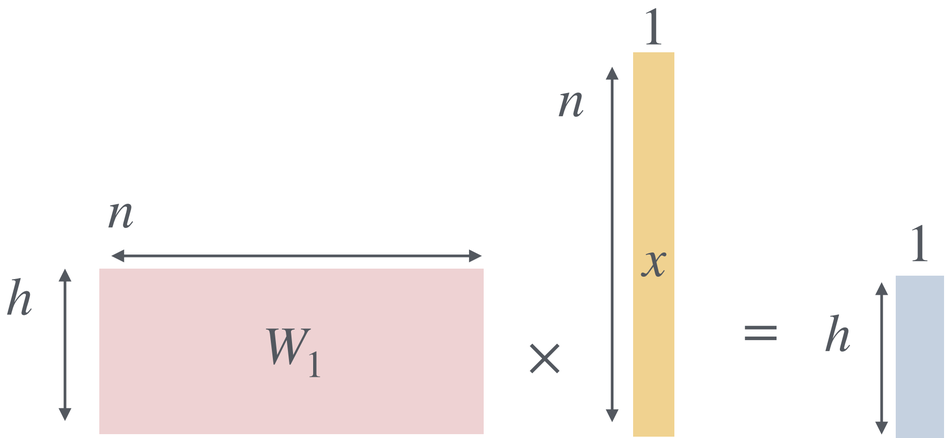


2 layer neural network for single input

$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{h\times n}, W_2 \in \mathbb{R}^{k\times h}$$

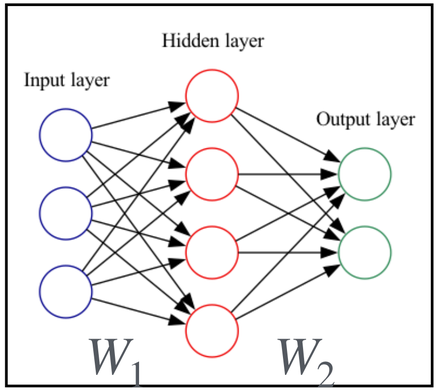
$$x \in \mathbb{R}^{n}, y \in \mathbb{R}^{k}$$



Matrix multiplication

Parameters and computation of FC

FC (fully connected) neural network

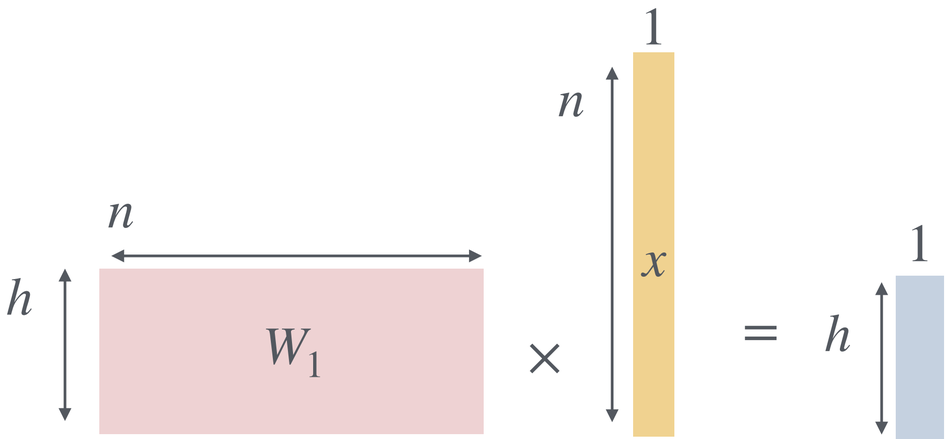


2 layer neural network for single input

$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{h\times n}, W_2 \in \mathbb{R}^{k\times h}$$

$$x \in \mathbb{R}^{n}, y \in \mathbb{R}^{k}$$



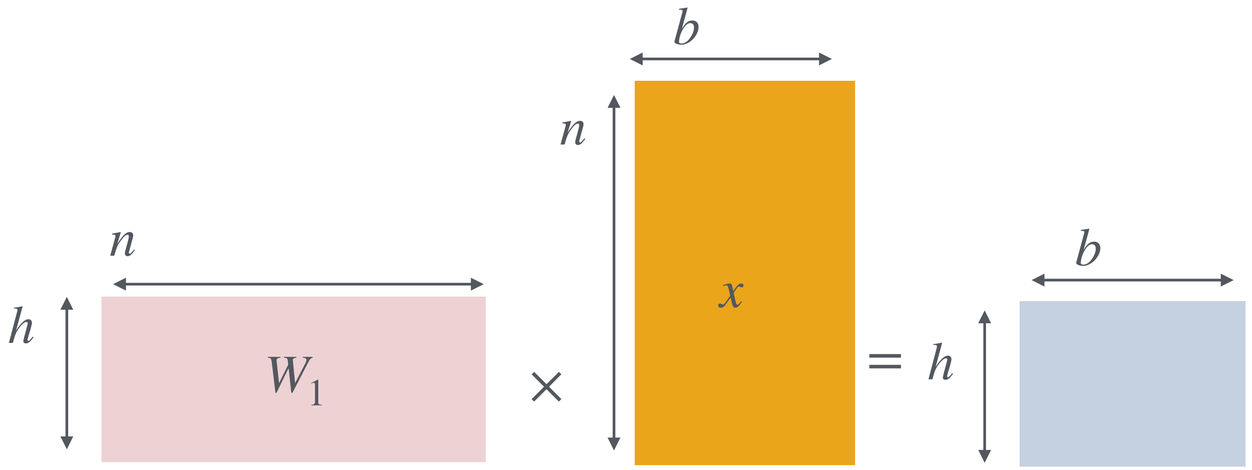
Matrix multiplication

2 layer neural network for mini-*batch* *input (size* b*)*

$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{h\times n}, W_2 \in \mathbb{R}^{k\times h}$$

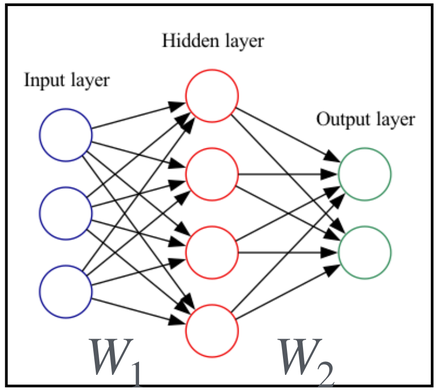
$$x \in \mathbb{R}^{n\times b}, y \in \mathbb{R}^{k\times b}$$



Matrix multiplication

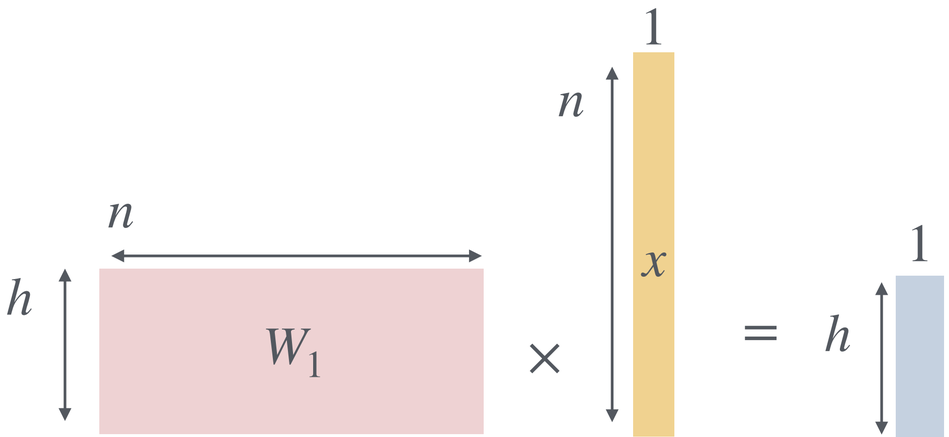
Parameters and computation of FC

FC (fully connected) neural network



2 layer neural network for single input

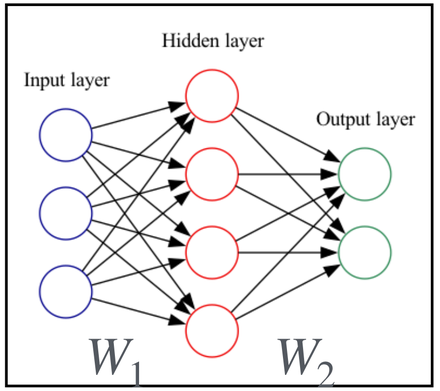
- # parameters: $h \times n$
- Memory: $2(h \times n + n + h)$, 2 for FP16, hn: weight, n: input activation, h: output activation
- Compute (FLOPs): $2(h \times n)$, 2 for mul + add



Matrix multiplication

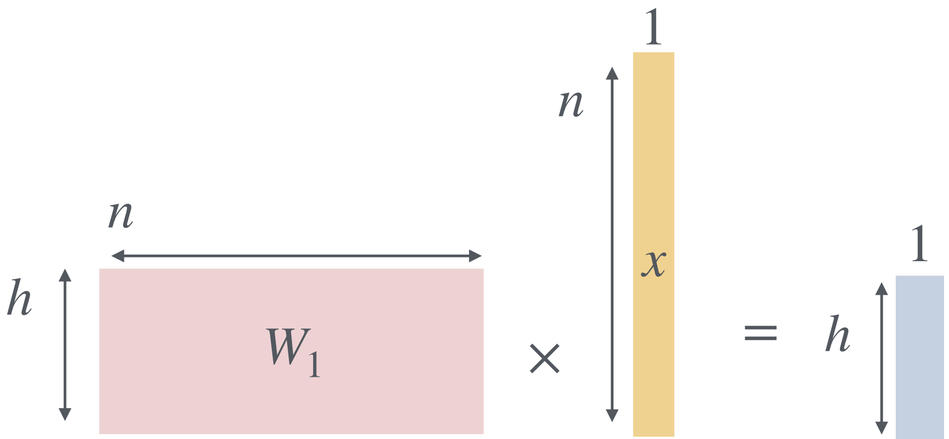
Parameters and computation of FC

FC (fully connected) neural network



2 layer neural network for single input

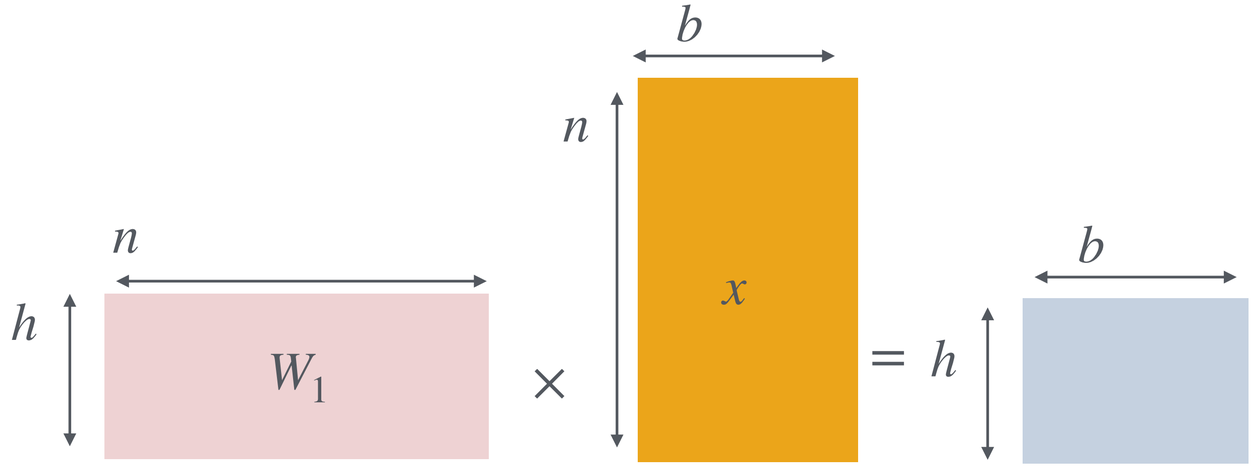
- # parameters: $h \times n$
- Memory: $2(h \times n + n + h)$, 2 for FP16, hn: weight, n: input activation, h: output activation
- Compute (FLOPs): $2(h \times n)$, 2 for mul + add



Matrix multiplication

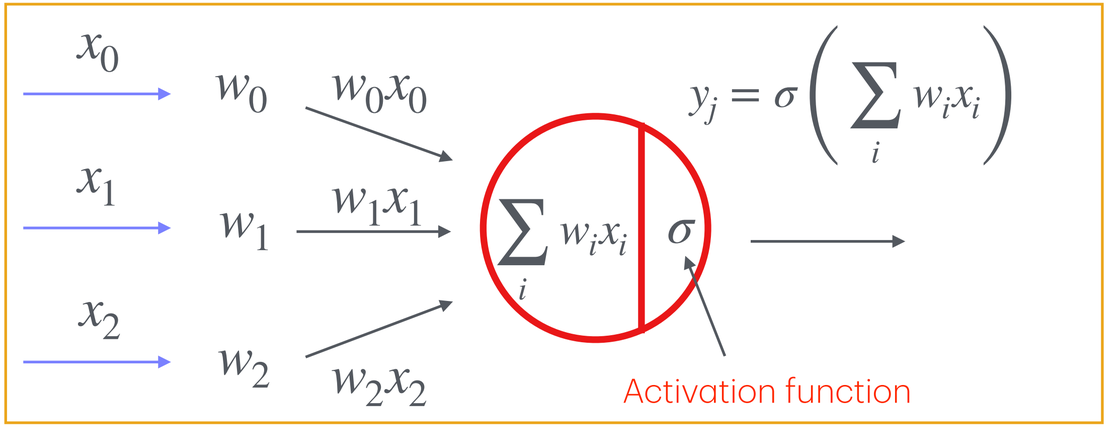
2 layer neural network for mini-*batch* *input (size* b*)*

- # parameters: $h \times n$
- Memory: $2(h \times n + b \times n + b \times h)$, 2 for FP16, hn: weight, bn: input activation, bh: output activation
- Compute (FLOPs): $2(h \times n \times b)$, 2 for mul + add



Matrix multiplication

Activation functions

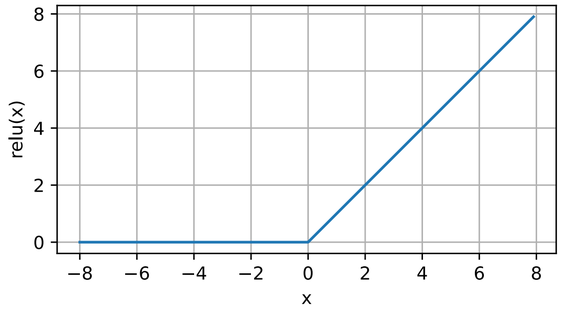


Activation function decides if a neuron should activate the calculated sum.

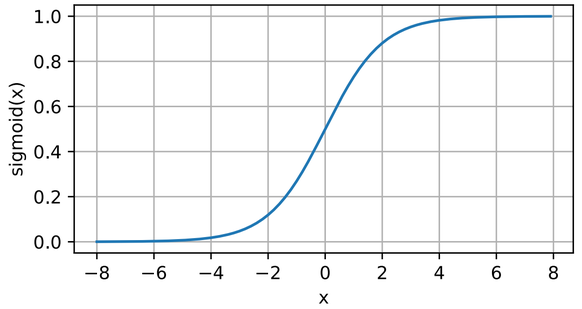
They are differentiable and applied to input element-wise non-linear function

- ReLU (Rectified Linear Unit) is a good default for most problems
- Sigmoid squashes all inputs to interval between (0, 1)
- Tanh (hyperbolical tangent) squashes all inputs to interval between (-1, 1)

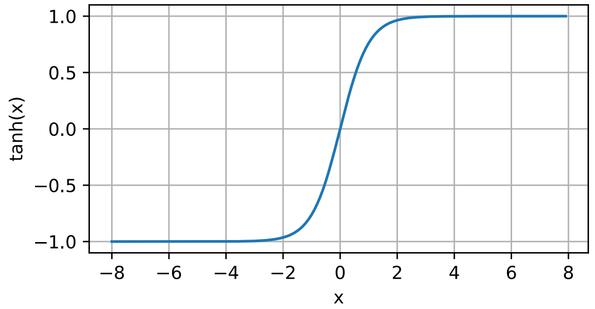
$$\mathrm{ReLU}(x) = \max(x, 0)$$



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



$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$



Neural network

Forward propagation: compute the input through the network using weights/ activations to get the output

Backward propagation: computing gradients using chain rule from calculus.

- Software framework automatically calculates gradients for this propagation (automatic differentiation)

```
Neural network training steps:
Loop through dataset:
  # compute prediction and loss
  Forward_prop(dataset)
  # compute gradient
  Backward_prop()
  # gradient descent
  Update_weight()
```

Building ML model using blocks

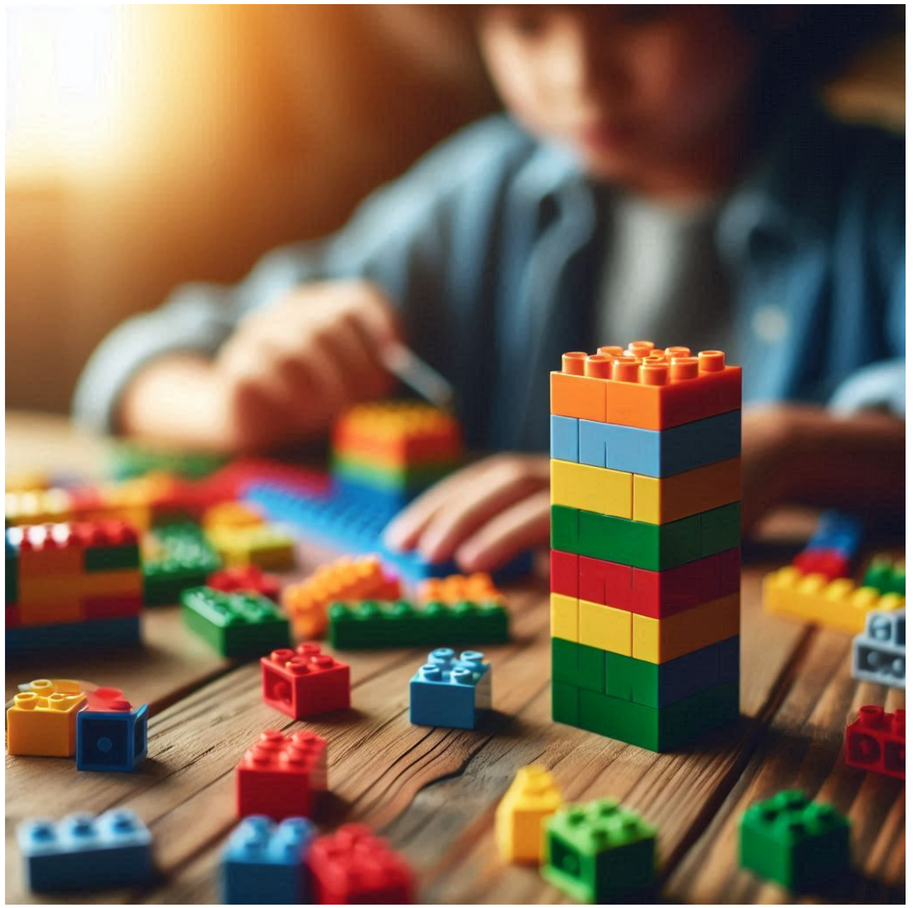
We have learned three key ingredients of ML model training. We learned two models: linear classifier and neural network.

ML model using building blocks:

- We can create complex ML models by *composing* layers of neural network. Then we train the model using the loss designed and gradient descent optimization method

- We will learn different types of model lego blocks in the following slides

Since AlexNet debuted in the ImageNet competition, the size of models has increased with the help of many new ideas such as the residual method, which helps train much deeper networks. We can use these ideas with model blocks in many applications.



AI generated image

Computer vision

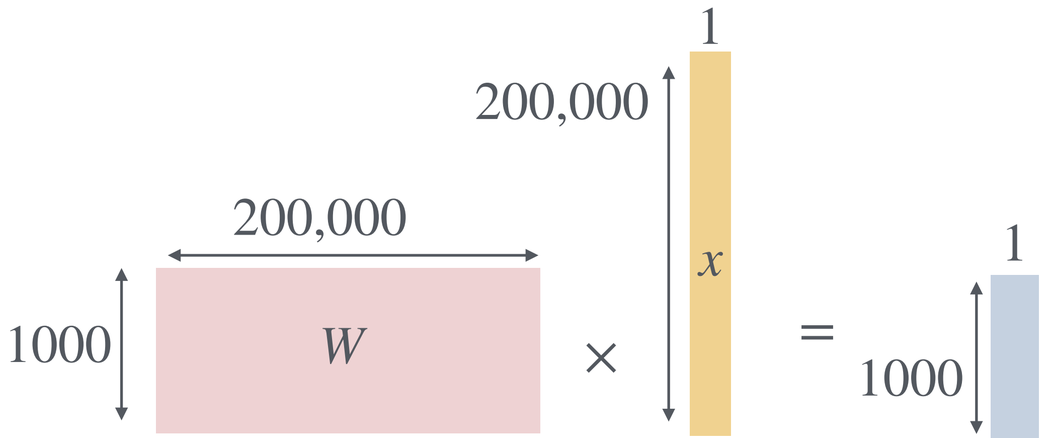
Issues in handling image with fully connected neural network

Image data is flattened to a vector and then feed to neural network. To handle large size image, then we need large number of parameters to handle even in a single hidden layer.

- For example, an input image (256x256 RGB) is ~200K dimension vector, $x \in \mathbb{R}^n, n = 200K$. To map this hidden layer with 1000 hidden neurons, we need ~ 200 million parameters.

$$z = Wx,\ W \in \mathbb{R}^{h \times n},\ h = 1000$$

Spacial information is not captured, such as a slight shift on image will generate different activation output to the next layer.



Matrix multiplication

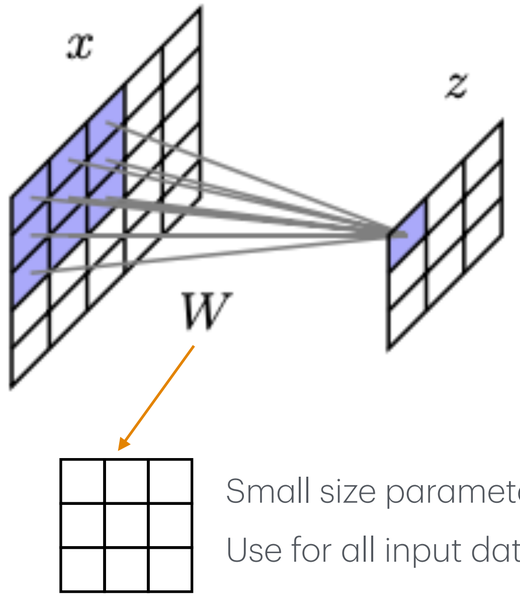
Convolutional Neural Network (CNN)

Use small size parameters to capture local information only to capture the structure of the image. Keep handling output activations with spacial structure saved with hidden layers.

Share small number parameters for all data points in each layer.

Advantages of CNN

- Drastically reduce the number of parameters
- Captures spacial information of the image



Small size parameters.

Use for all input data by sliding through input

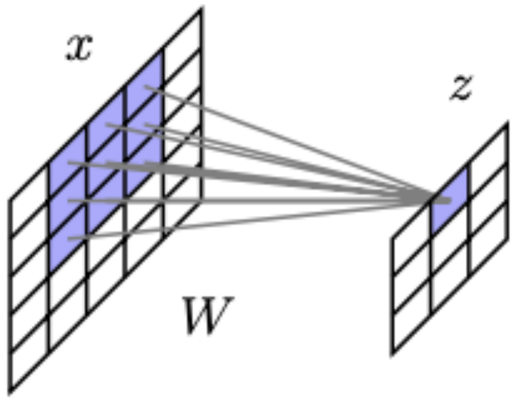


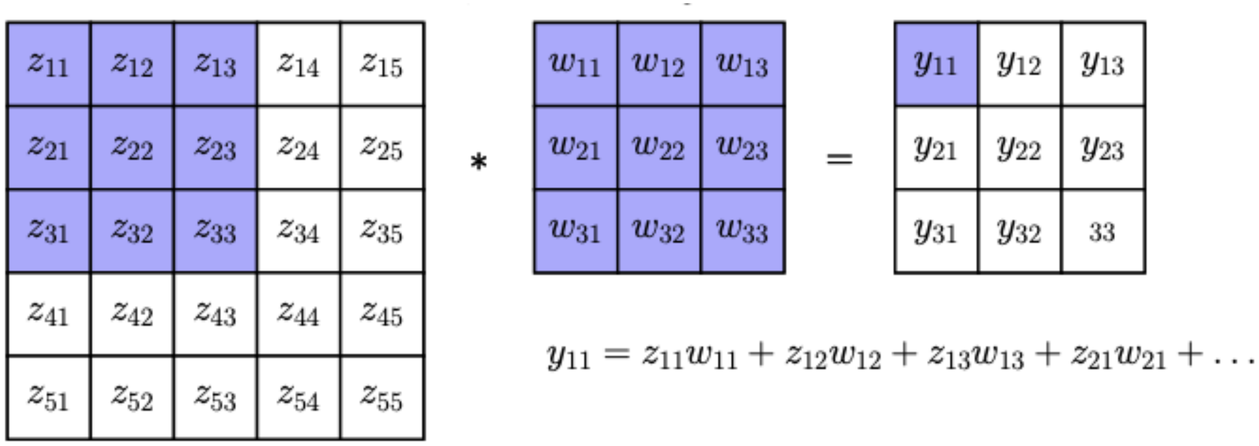
Image source: [2]

How convolution works

Slide the parameters with $k \times k$ dimension, also called kernel/filter size of k over the input, and produce an output activation. We write as $y = z * w$.

Convolution kernel does a dot product (multiple and add) of the range of the input that kernel covers with the kernel. If the input image is multiple channels (such as RGB), then the kernel will have the same number of channels and the dot product is extended to channel dimensions.

We use a single channel input and size of 3 kernel show below



$$y_{11} = z_{11}w_{11} + z_{12}w_{12} + z_{13}w_{13} + z_{21}w_{21} + \dots$$

Single channel input 3x3 kernel Output

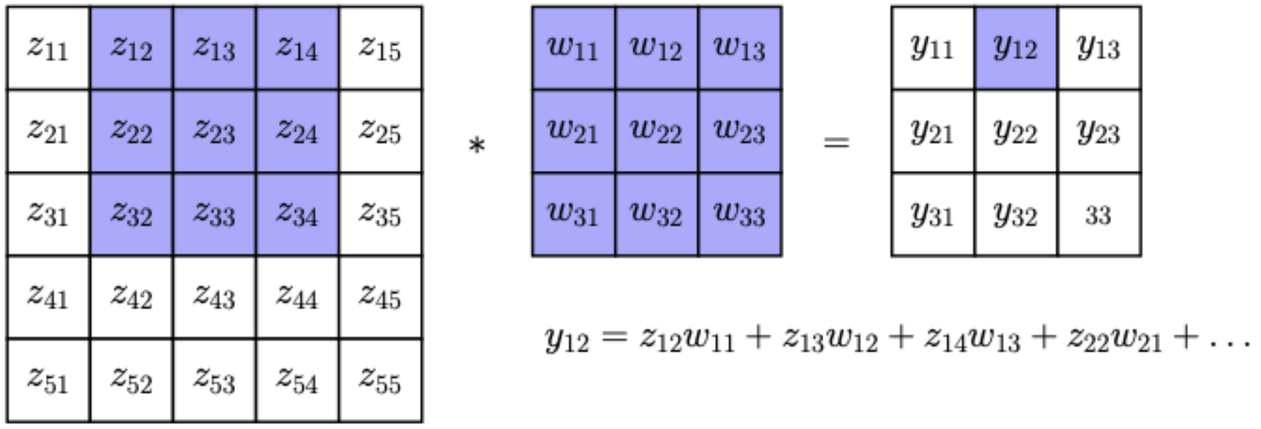
Ref. and Image source: [2]

How convolution works

Slide the parameters with $k \times k$ dimension, also called kernel/filter size of k over the input, and produce an output activation. We write as $y = z * w$.

Convolution kernel does a dot product (multiple and add) of the range of the input that kernel covers with the kernel. If the input image is multiple channels (such as RGB), then the kernel will have the same number of channels and the dot product is extended to channel dimensions.

We use a single channel input and size of 3 kernel show below



$$y_{12} = z_{12}w_{11} + z_{13}w_{12} + z_{14}w_{13} + z_{22}w_{21} + \dots$$

Single channel input 3x3 kernel Output

Ref. and Image source: [2]

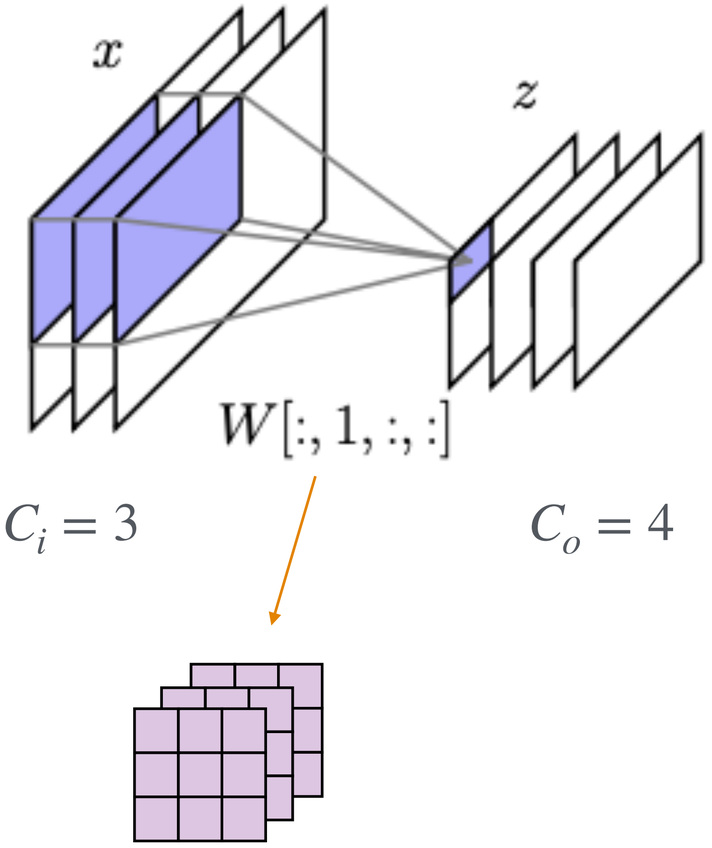
Convolutional neural network

CNN processes multi-channel inputs (such as RGB) and generates multi-channel activations. A single kernel should have the same channel size as input channel.

- $x \in \mathbb{R}^{h_i \times w_i \times c_i}$, c_i input channel, $h_i \times w_i$ input size
- $z \in \mathbb{R}^{h_o \times w_o \times c_o}$, c_o output channel, $h_o \times w_o$ output size
- $W \in \mathbb{R}^{c_i \times c_o \times k_h \times k_w}$, 4 dimensional vector, $k_h \times k_w$ kernel size

Single channel output computation is sum of convolution over all input channels. Below equation shows the computation for output channel s.

$$z[:, :, s] = \sum_{r=1}^{c_i} x[:, :, r] * W[r, s, :, :]$$



$W \in \mathbb{R}^{3 \times 1 \times 3 \times 3}$, kernel for 1 output channel

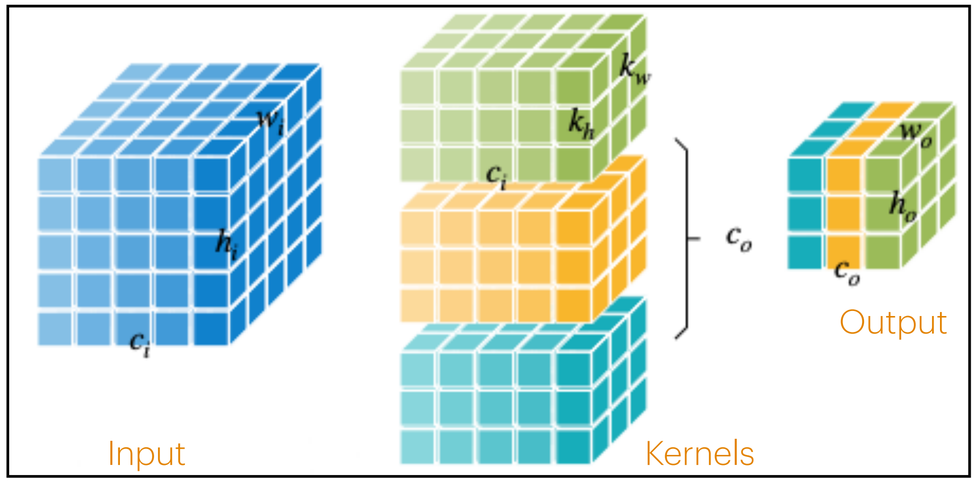
We have total 4 (c_o) kernels

Ref. and Image source: [2]

Parameters and computation in convolution

parameters: $c_o \cdot c_i \cdot k_h \cdot k_w$

Compute (FLOPs): $2 \cdot c_i \cdot k_h \cdot k_w \cdot h_o \cdot w_o \cdot c_o$, 2 for mul+add



- $x \in \mathbb{R}^{h_i \times w_i \times c_i}$, c_i input channel, $h_i \times w_i$ input size
- $z \in \mathbb{R}^{h_o \times w_o \times c_o}$, c_o output channel, $h_o \times w_o$ output size
- $W \in \mathbb{R}^{c_i \times c_o \times k_h \times k_w}$, 4 dimensional vector, $k_h \times k_w$ kernel size

Image source: [1]

Naive implementation

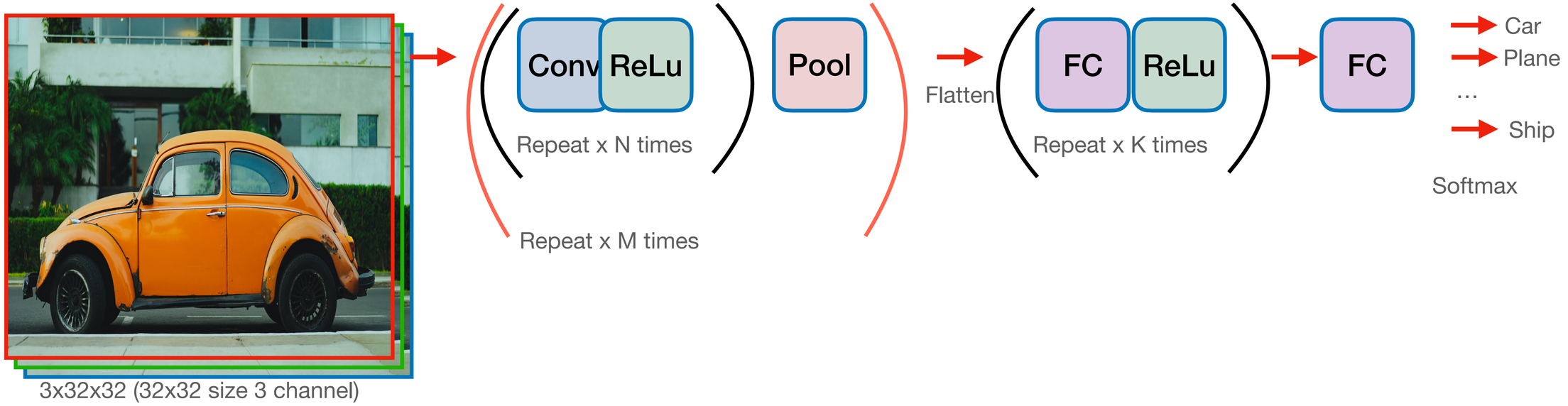
```
def conv_naive(Z, weight):
    # N: batch size
    # C_in: input channel
    # H x W: input dimension
    N,H,W,C_in = Z.shape
    # K: kernel size
    # C_out: output channel
    K,_,_,C_out = weight.shape

    out = np.zeros((N,H-K+1,W-K+1,C_out));
    # loop over batch size
    for n in range(N):
        # loop over input channel
        for c_in in range(C_in):
            # loop over output channel
            for c_out in range(C_out):
                # main convolution loop
                for y in range(H-K+1):
                    for x in range(W-K+1):
                        for i in range(K):
                            for j in range(K):
                                out[n,y,x,c_out] += Z[n,y+i,x+j,c_in] *
                                weight[i,j,c_in,c_out]
    return out
```

CNN model

CNN model are consist of repeated conv layers followed by activation (such as ReLU) and pooling layer (down sample). At the end, FC (fully connected layers) are added for the classification task.

Note: more information on padding, pooling and example CNN models will be at additional slide section.



Natural language processing (NLP)

Representation of words

Words can be represented as discrete symbols, such as one-hot vectors:

- Vector dimension = number of words in the vocabulary (~ 500,000+)

Problem: how to find the similarity of words?

Solution: use a model to encode similarity in the vector themselves. So that vectors of similar meaning will be close to each other.

motel = [0 0 0 0 0 0 0 1 0 0 0]

hotel = [0 0 0 1 0 0 0 0 0 0 0]

Ref. : [5]

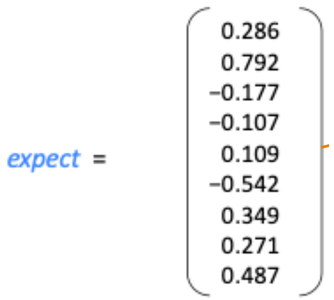
Word embedding

Word2vec: a model converts words to word embeddings (vector)

It learns the word embedding by training the model to predict probability of given word and its contexts.

Similar words are clustered each other. In the example high dimensional vectors are projected to 2 dim for visualization.

Relationship of words also captured by word embedding. In the example, an answer for "man:woman=king:?" can be with cosine distance in the embedding space.



Word embedding

Image source: [5]

Word embedding

Word2vec: a model converts words to word embeddings (vector)

It learns the word embedding by training the model to predict probability of given word and its contexts.

Similar words are clustered each other. In the example high dimensional vectors are projected to 2 dim for visualization.

Relationship of words also captured by word embedding. In the example, an answer for "man:woman=king:?" can be with cosine distance in the embedding space.

expect =

0.286
0.792
−0.177
−0.107
0.109
−0.542
0.349
0.271
0.487

Word embedding

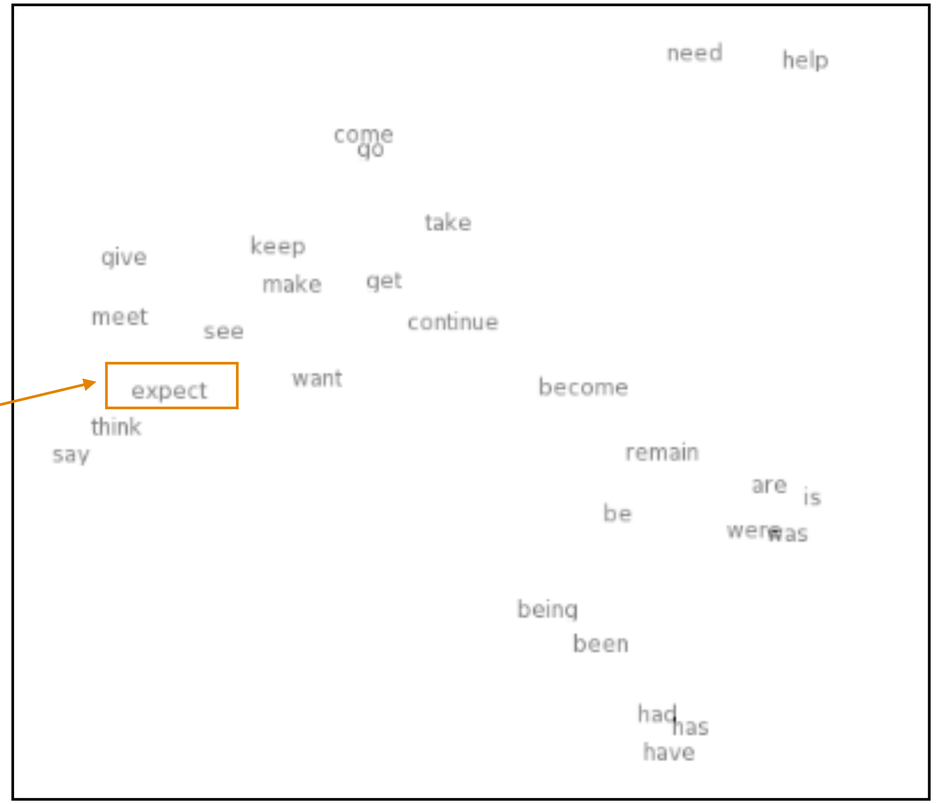


Image source: [5]

Word embedding

Word2vec: a model converts words to word embeddings (vector)

It learns the word embedding by training the model to predict probability of given word and its contexts.

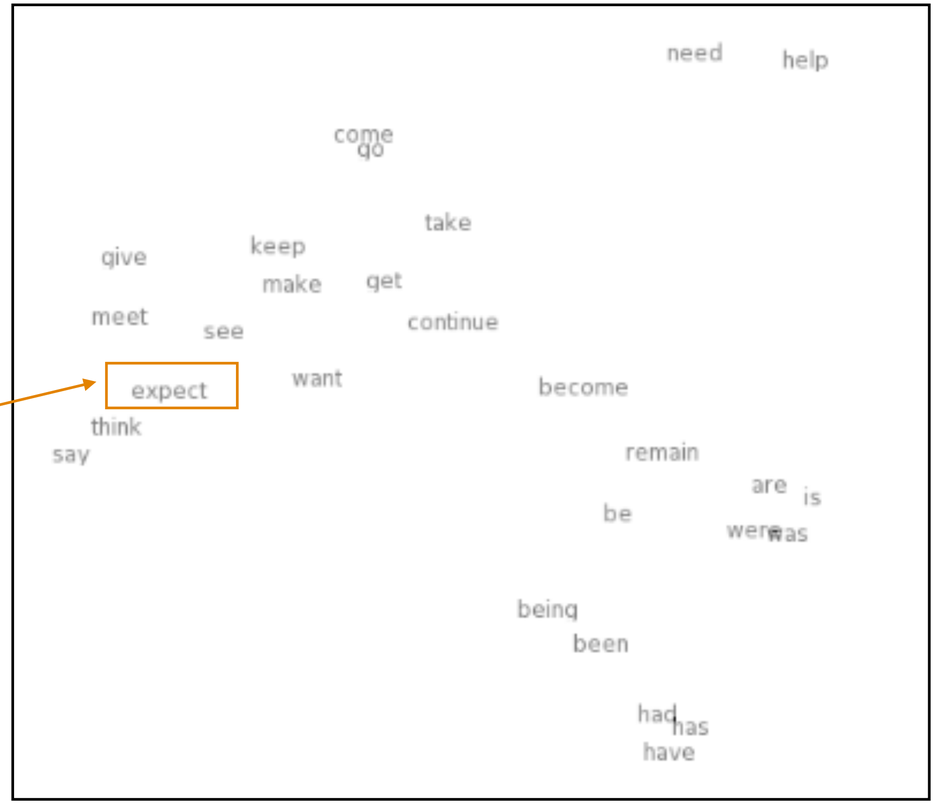
Similar words are clustered each other. In the example high dimensional vectors are projected to 2 dim for visualization.

Relationship of words also captured by word embedding. In the example, an answer for "man:woman=king:?" can be with cosine distance in the embedding space.

expect =

0.286
0.792
−0.177
−0.107
0.109
−0.542
0.349
0.271
0.487

Word embedding



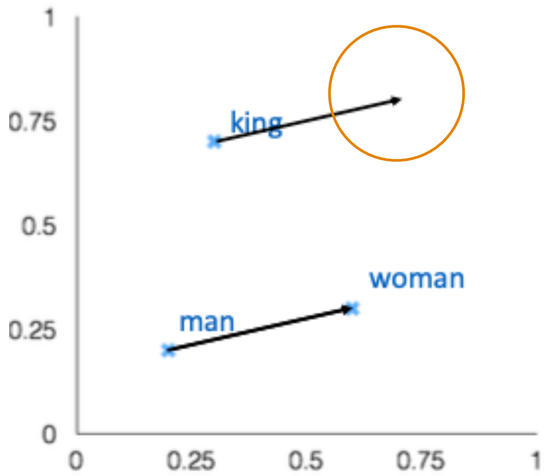


Image source: [5]

Language model

LM (language model): predicts next word given input

- Input: text
- Output: predict the next word by sampling

Ref. : [5]

Language model

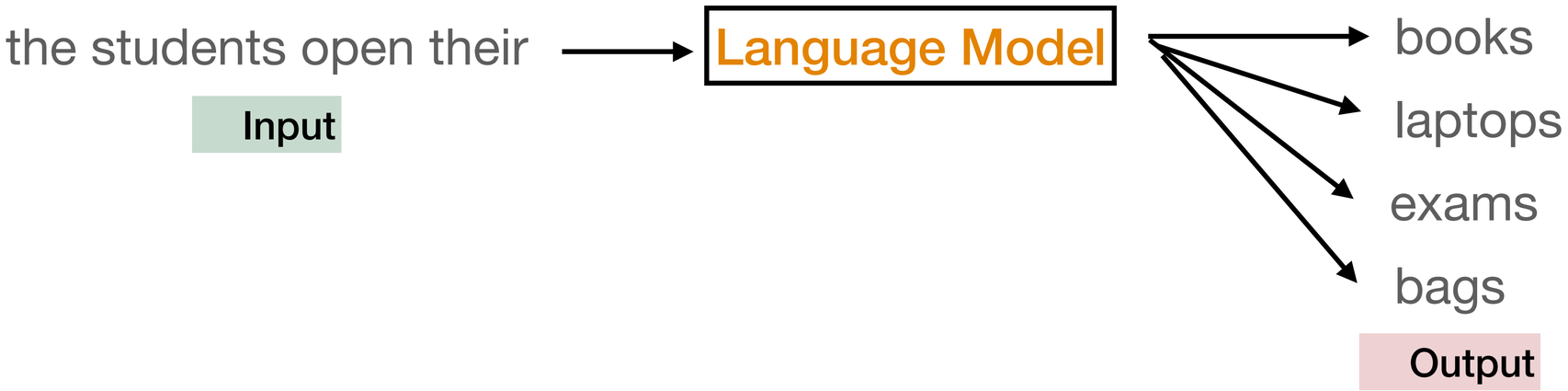
LM (language model): predicts next word given input

- Input: text
- Output: predict the next word by sampling

Formal definition: given a sequence of words, $x^1, x^2, \ldots, x^t$, compute the probability distribution of next word,

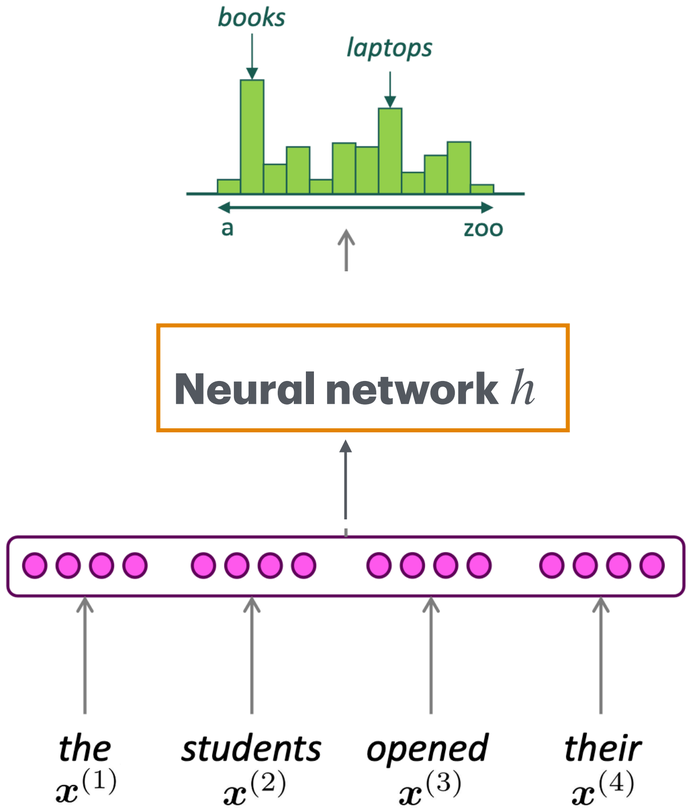
$$P(x^{(t+1)} \mid x^1, \ldots, x^t)$$

where $x^{(t+1)}$ can be a word in the dictionary $V = \{w_1, \ldots w_{|V|}\}$



Ref. : [5]

Neural language model w/ fixed input window



Fixed window sized input, such as 4 words

Use neural network to predict next word given input words.

Fixed window neural network based LM

- Keep input within fixed window, and feed to a neural network.
- Neural network based language model predicts the probability of the next word.

Problems with this approach?

- Small fixed window
- Larger window means larger model

We need a model that can process any length of input

Ref. and Image source: [5]

Neural language model w/ fixed input window

output distribution

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

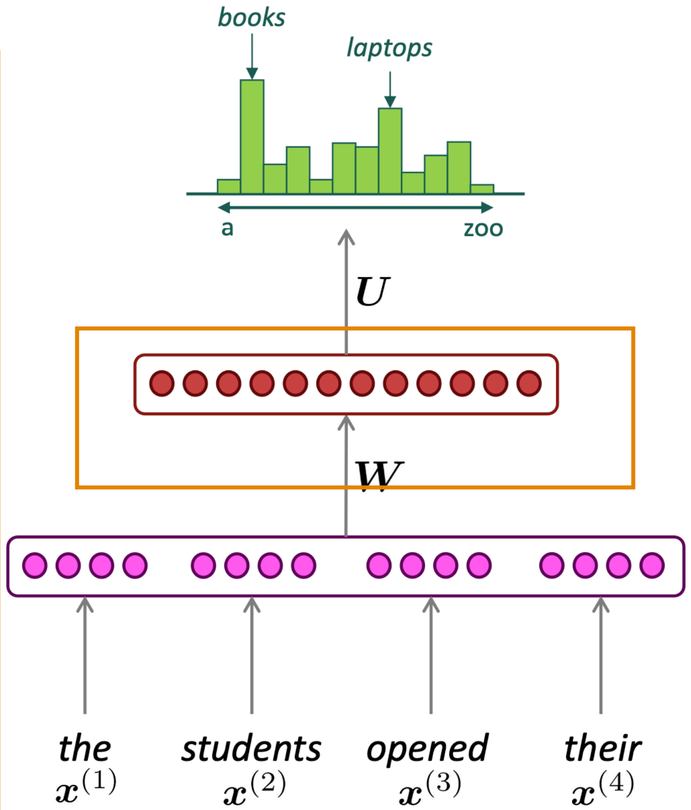
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



Fixed window sized input, such as 4 words

Use neural network to predict next word given input words.

Fixed window neural network based LM

- Keep input within fixed window, and feed to a neural network.
- Neural network based language model predicts the probability of the next word.

Problems with this approach?

- Small fixed window
- Larger window means larger model

We need a model that can process any length of input

Ref. and Image source: [5]

RNN (Recurrent neural network)

Use the same weight $\boldsymbol{W}$ for processing each input.

Allows stateful computation since it uses hidden state from previous time $(t-1)$

Any input sequence lengths can be processed

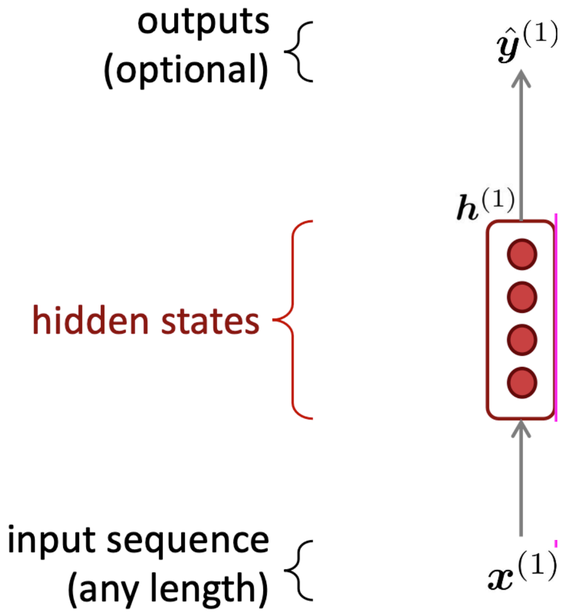


Image source: [5]

RNN (Recurrent neural network)

Use the same weight W for processing each input.

Allows stateful computation since it uses hidden state from previous time $(t-1)$

Any input sequence lengths can be processed

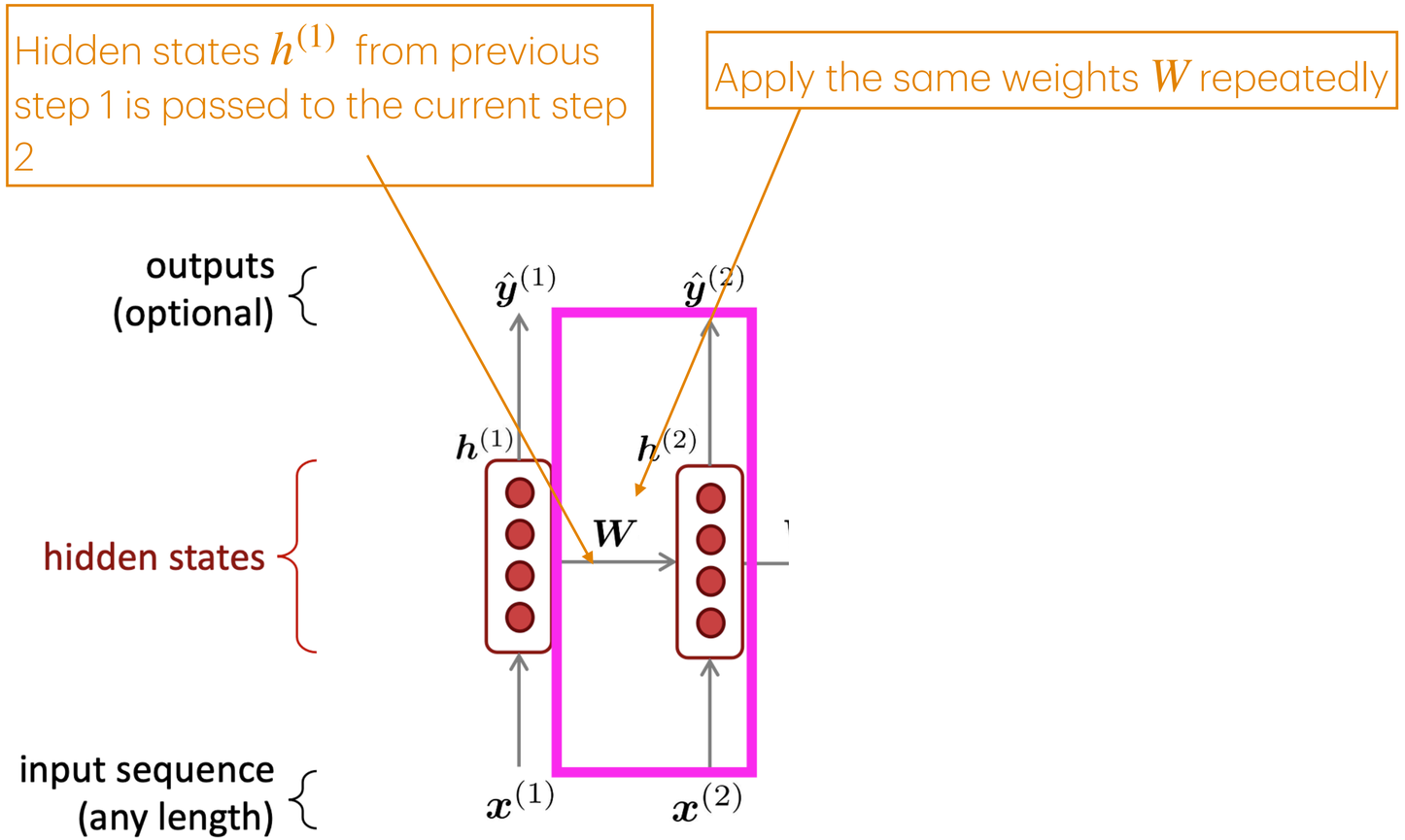


Image source: [5]

RNN (Recurrent neural network)

Use the same weight W for processing each input.

Allows stateful computation since it uses hidden state from previous time $(t-1)$

Any input sequence lengths can be processed

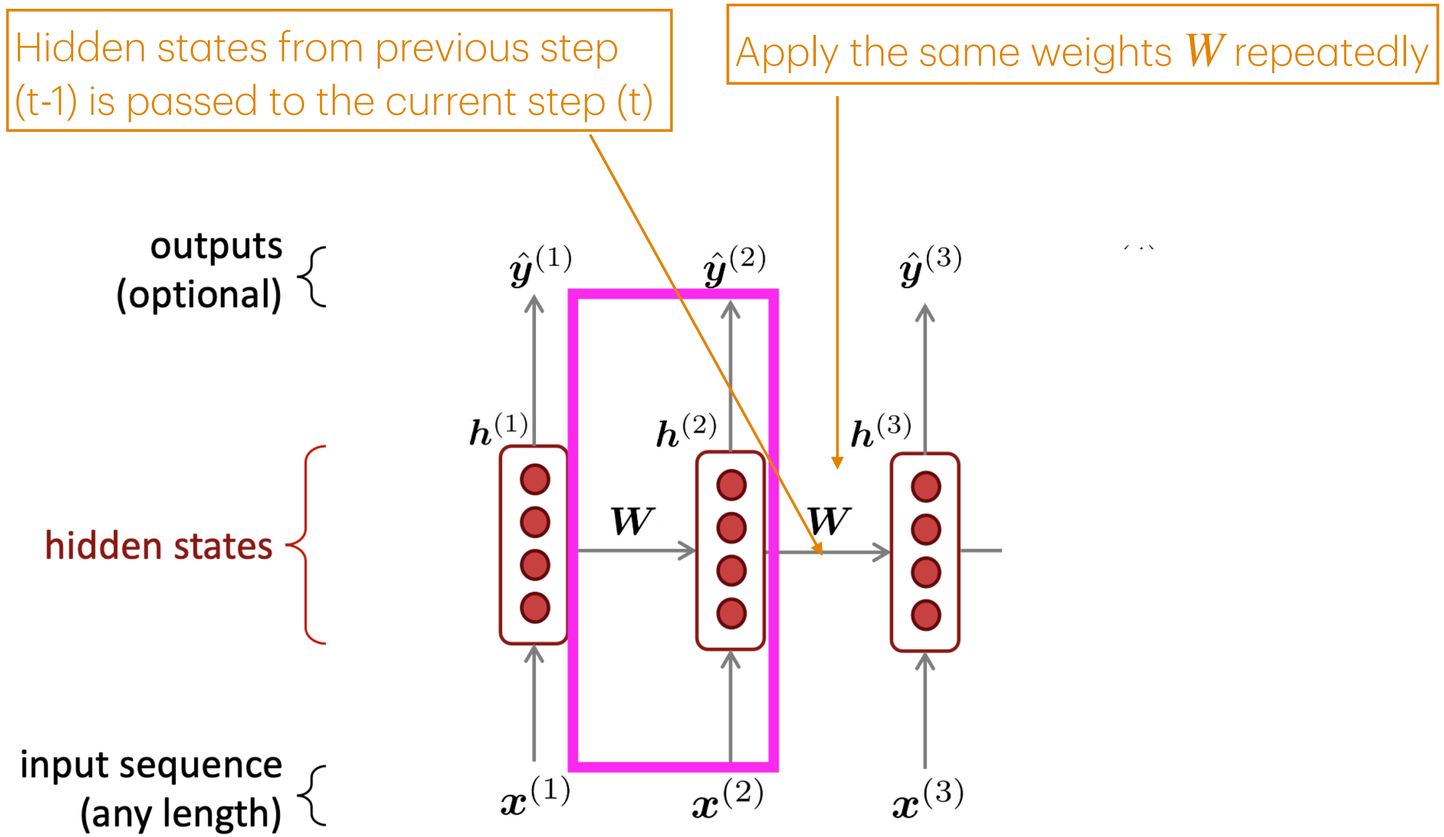


Image source: [5]

RNN (Recurrent neural network)

Hidden states from previous step (t-1) is passed to the current step (t)

Apply the same weights W repeatedly

Use the same weight W for processing each input.

Allows stateful computation since it uses hidden state from previous time $(t-1)$

Any input sequence lengths can be processed

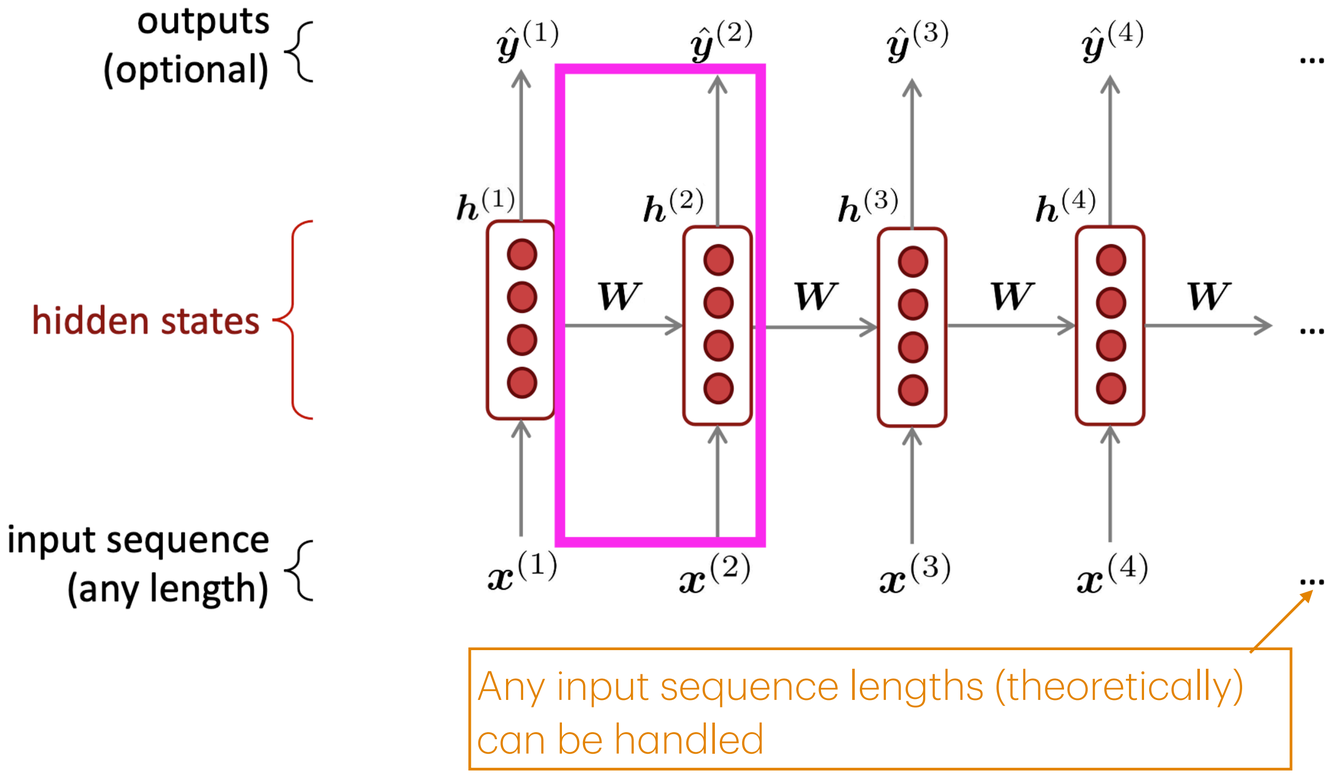


Image source: [5]

RNN based Language model

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

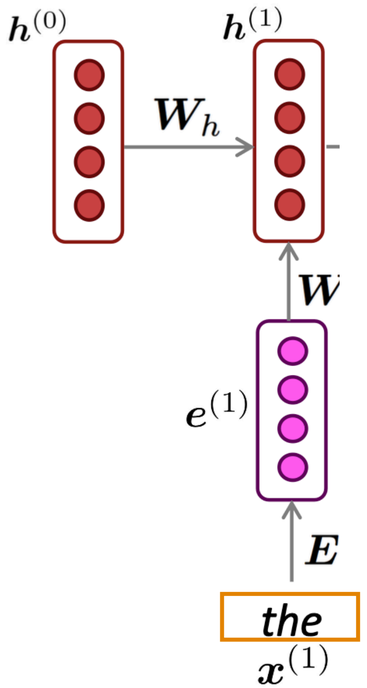
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



Compared to previous neural network based LM, RNN based LM can handle much longer input sequence now.

Sequential states are processed using hidden state from the previous step.

Image source: [5]

RNN based Language model

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

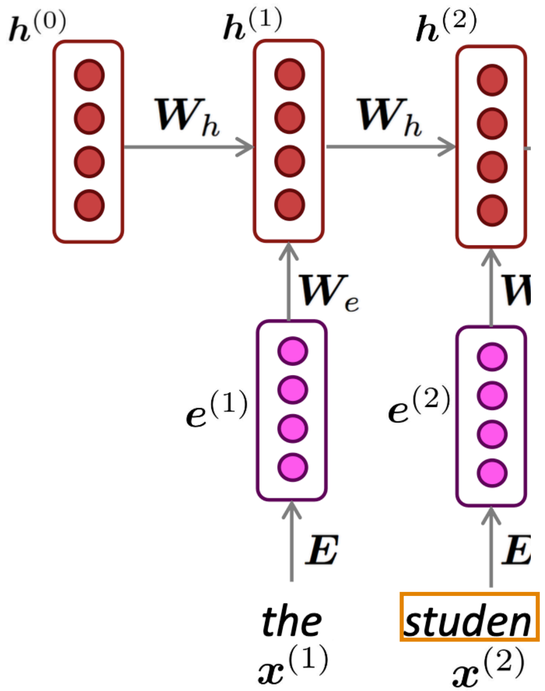
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



Compared to previous neural network based LM, RNN based LM can handle much longer input sequence now.

Sequential states are processed using hidden state from the previous step.

Image source: [5]

RNN based Language model

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

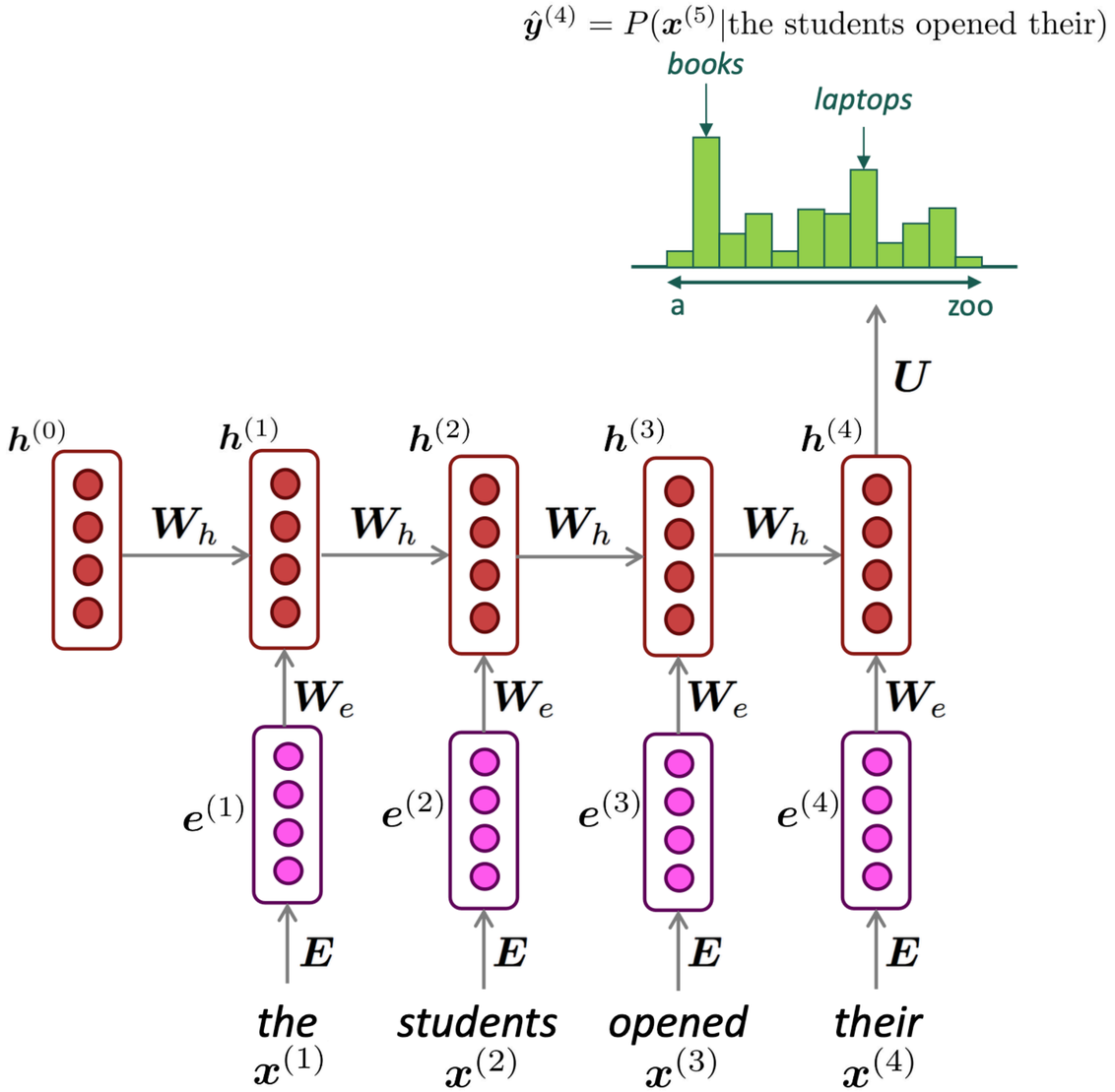
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



Compared to previous neural network based LM, RNN based LM can handle much longer input sequence now.

Sequential states are processed using hidden state from the previous step.

Image source: [5]

Fancy RNNs

Vanilla RNN has difficulty of capturing long term relations due to vanishing gradient

- Fancy RNN that allows to capture past state better, LSTM (long short term memory), GRU (Gated recurrent unit)

Stacked RNN: stacking RNNs to capture more complex information

Bi-directional RNN: to capture in both forward and backward directional relations

Does RNN model solve NLP problems?

Slow training due to the sequential nature of hidden state propagation

Hard to parallelize training:

- Forward and backward propagation takes O(seq len) and not parallelizable while GPUs can perform many parallel tasks at once

Hard to learn long-distance dependencies

- It takes O(seq len) steps to model the interaction between two words

"Attention is all you need" proposed a solution to the above problems:

- Transformer model: a parallelizable model that can effectively attend to all sequence

What is a Large Language Model (LLM)?

- An LLM is a model takes text input and generates a high probability continuation of that text output
- The probabilities are trained form massive quantities of text data
- The probabilities are learned from highly abstract matrix operations:
  - No symbolic computer code
  - No interpretable database storage
  - No analytical control over behaviors
- It is a prediction mechanism for next text.

Ref: [4]

Summary

- Machine Learning (ML) learns how to solve problems through data and statistical methods.
- There are three key ingredients in ML: *hypothesis* (model), *loss* (defines the goodness of the model), and *optimization* (how to find the parameters that minimize the loss).
- A neural network is a basic building block of deep learning and consists of fully connected layers with non-linear activation functions that can learn non-linear boundaries.
- Convolutional Neural Networks (CNNs) use convolutional kernels to learn representations of images.
- Word embedding is a learned representational vector of words, such as Word2vec.
- Recurrent Neural Networks (RNNs) are a type of neural network that can be used for sequential input of any size, such as natural language.

References

1. TinyML and Efficient Deep Learning Computing
2. Deep Learning Systems, Algorithms and Implementation
3. Deep Learning for Computer Vision
4. Generative AI: Technology, Business, and Society
5. Natural Language Processing with Deep Learning

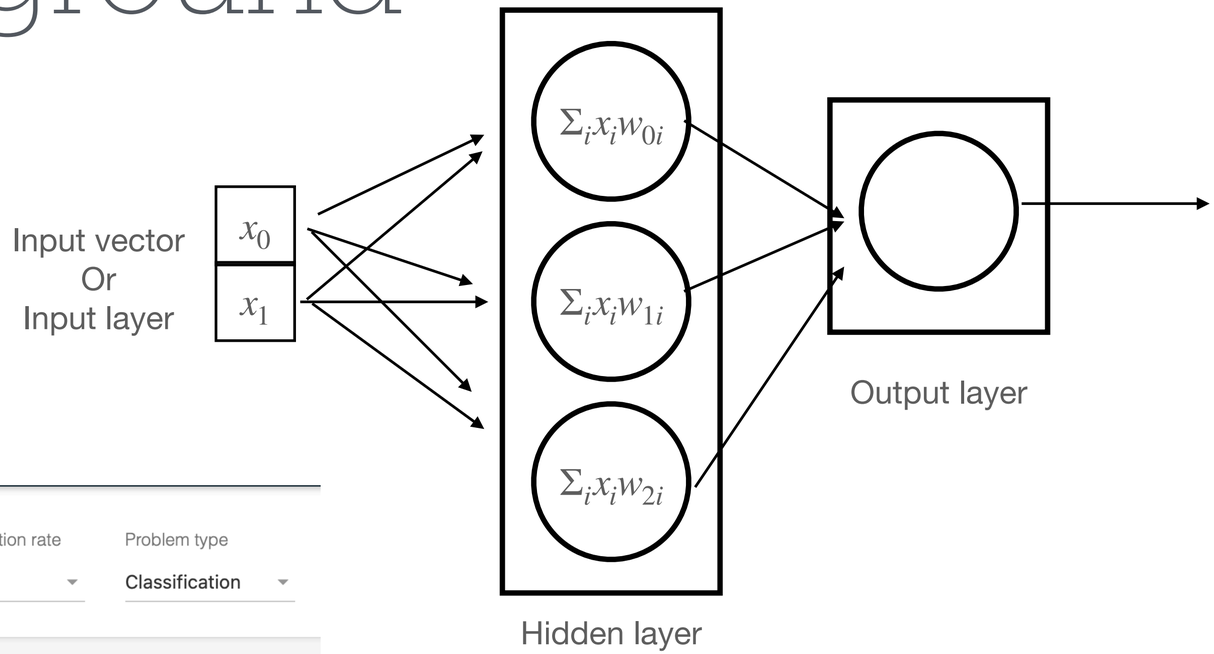
Free Books and additional material

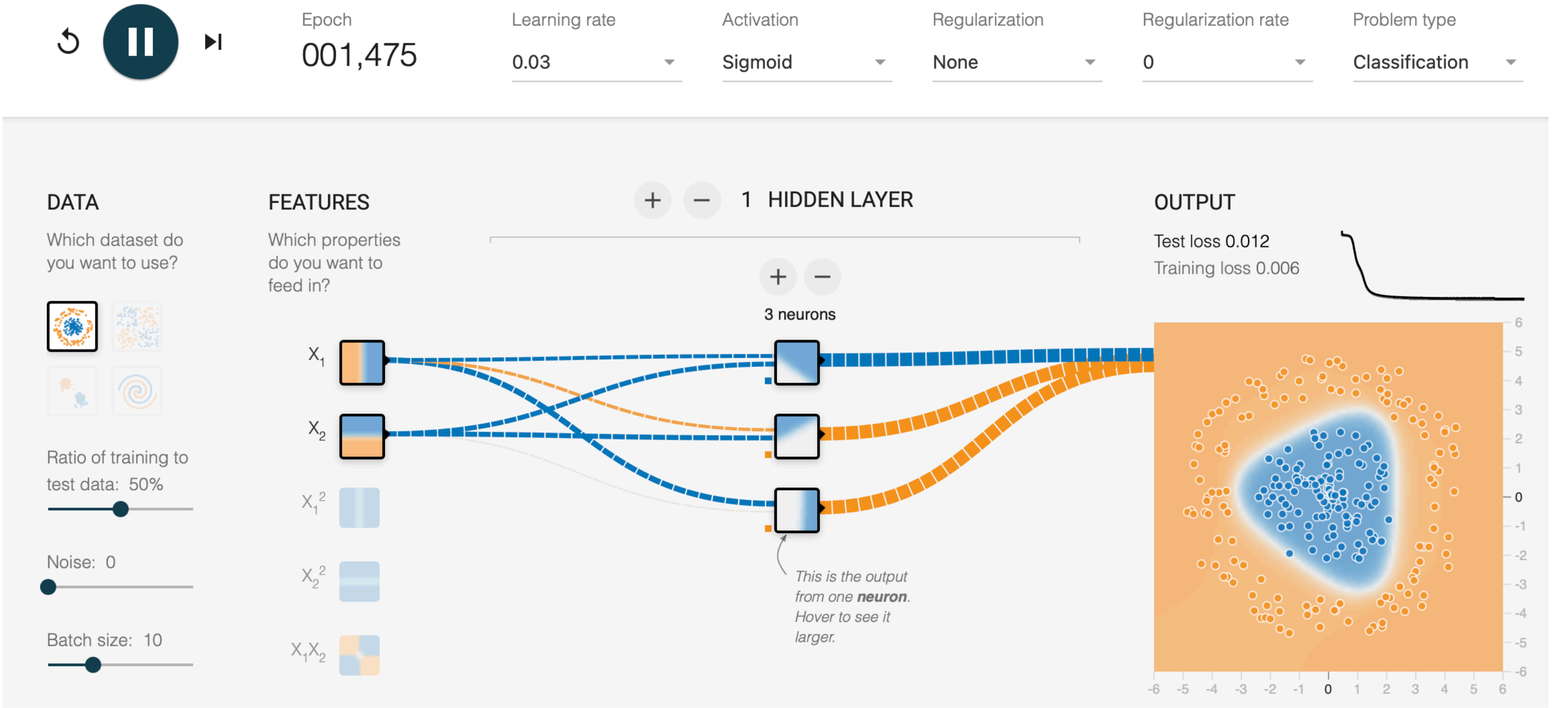
1. Dive into Deep Learning, Interactive deep learning book with code, math, and discussions
2. Understanding Deep Learning, book with Jupyter notebooks
3. Deep Learning, in-depth details theory and math
4. Neural network playground, http://playground.tensorflow.org
5. Word2vec visualization: https://projector.tensorflow.org/
6. MLU-EXPLAIN, Visual explanations of core machine learning concepts
7. Excellent Neural network introduction book: Neural Networks and Deep Learning
8. Excellent neural network tutorial

Additional slides

Neural network playground

- Great way to gain intuition
- http://playground.tensorflow.org





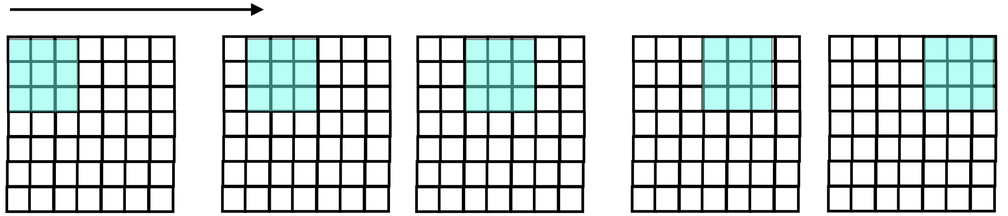
Simple PyTorch code for neural network

- Open in Colab

Stride in CNN

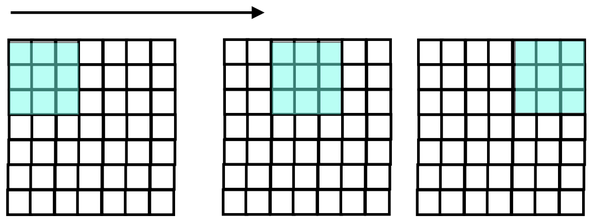
- Define how many steps to move
- Stride 3? Won't fit
- Output = $\frac{(N-F)}{S}+1$
- N: input size (assume same height&width)
- F: kernel size
- S: stride

Stride 1
3x3 kernel moves 1 step, output dim is 5x5



7x7 input

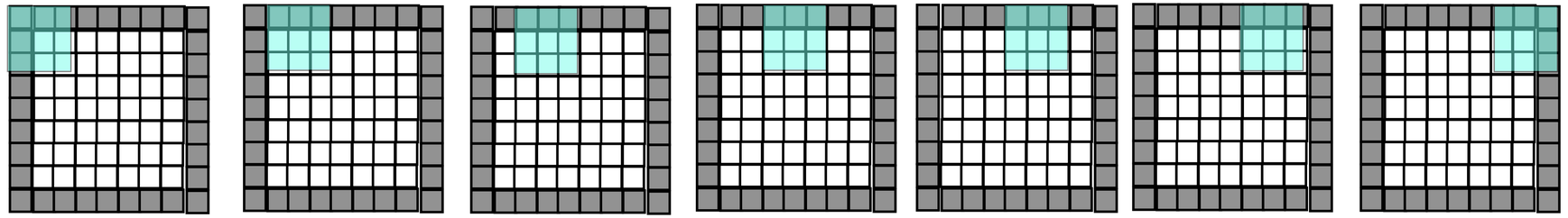
Stride 2
3x3 kernel moves 2 step2



Padding in CNN

- To preserve the same dimension for output
- Add paddings (0) to the input
- Output :
  - $\frac{(N+2P-F)}{S}+1$
- N: input size (assume same height&width)
- F: kernel size
- S: stride
- P: padding
- To preserve the size, add padding size
  $P=(F-1)/2$
  - When F=3 -> P=1, F=5 -> P=2

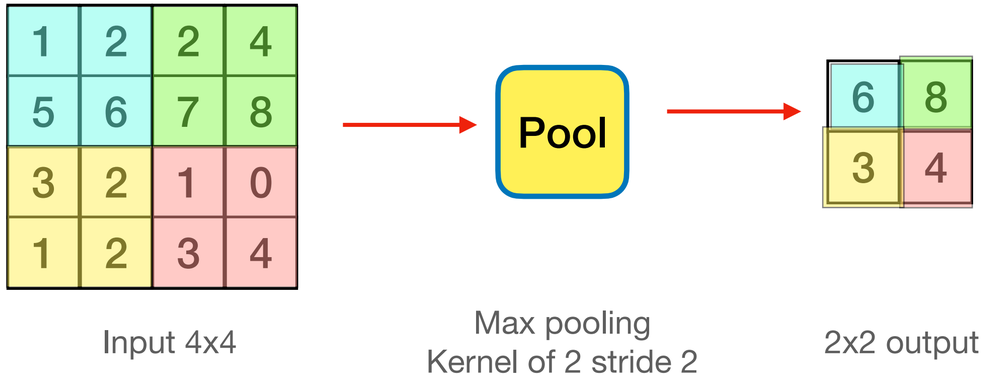
Slide 1, padding 1
3x3 kernel moves 1 step, output dim is 7x7

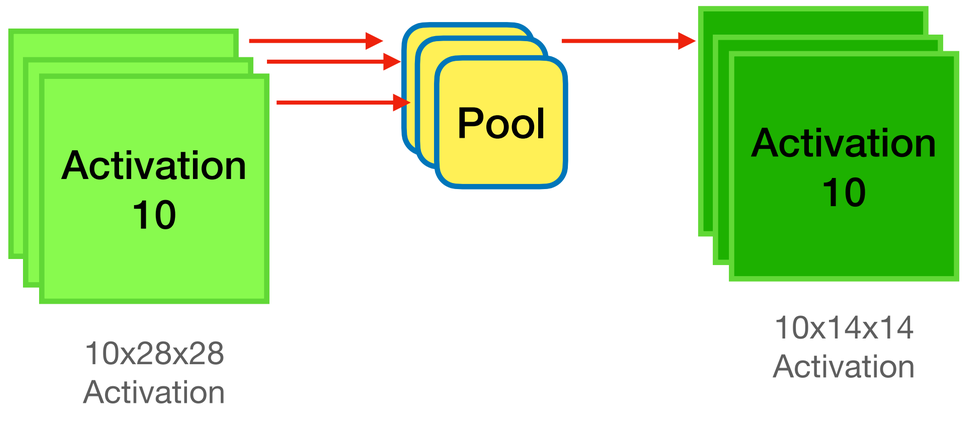


7x7 input
Gray area is zero padding

Pooling in CNN

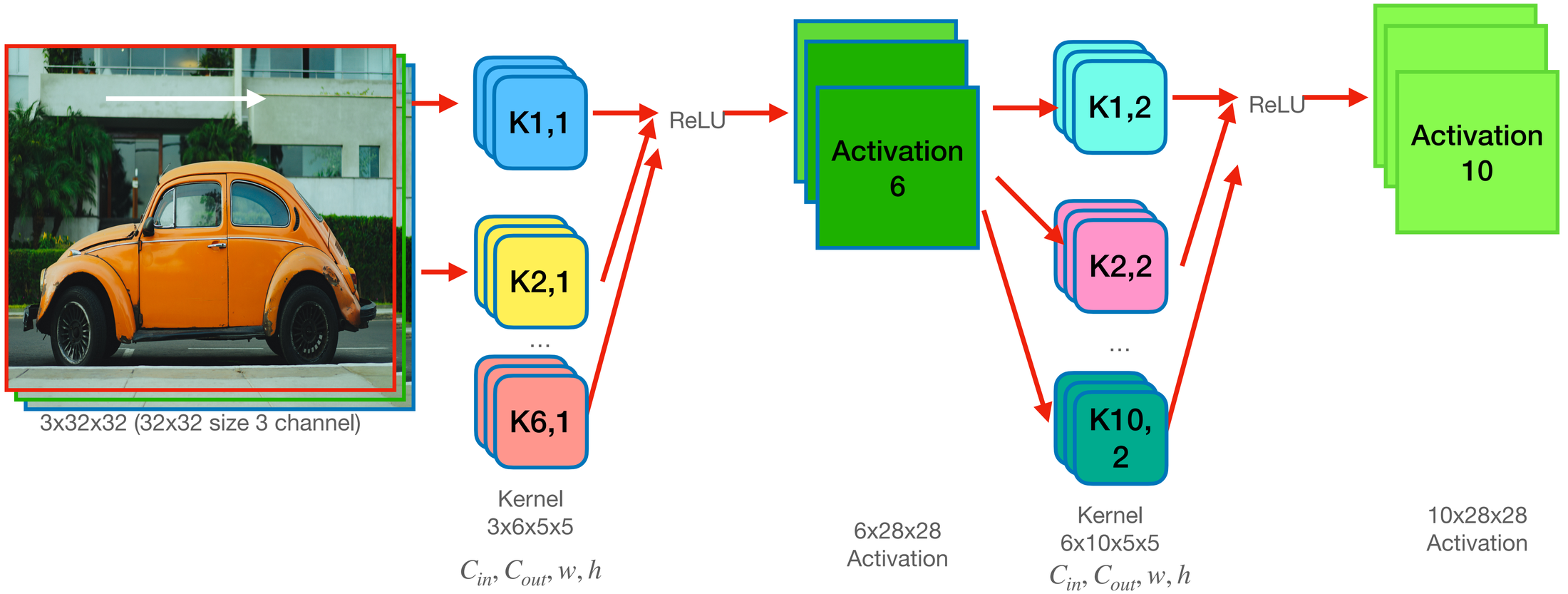
- Reduce the activation size
- Applied to each activation layers independently
- Downsample the input
- Max, average pool are used
- No learnable parameters





CNN structure example

- Multiple Conv kernel layer with non-linear layer (ReLU)



CNN architecture

Alexnet

- CONV1
- MAX POOL1
- NORM1
- CONV1
- MAX POOL2
- NORM2
- CONV3
- CONV4
- CONV5
- MAX POOL3
- FC6
- FC7
- FC8

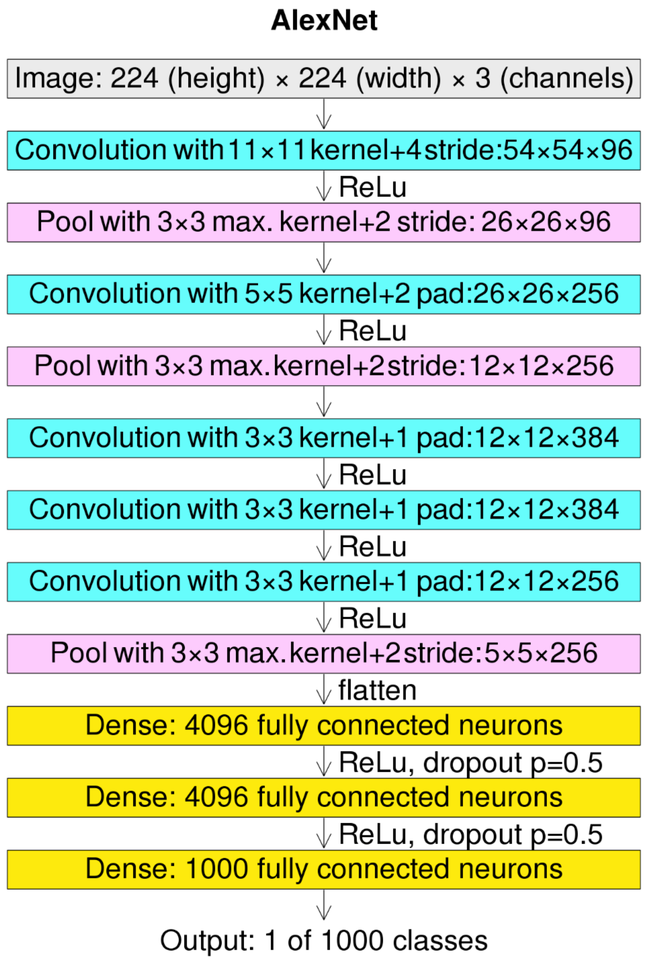


Image source: https://en.wikipedia.org/wiki/AlexNet

CNN architecture

ResNet

- Accomplished very deep network using skip connection
- Original Resnet is 152 layers
- Showing a smaller example of Resnet-18

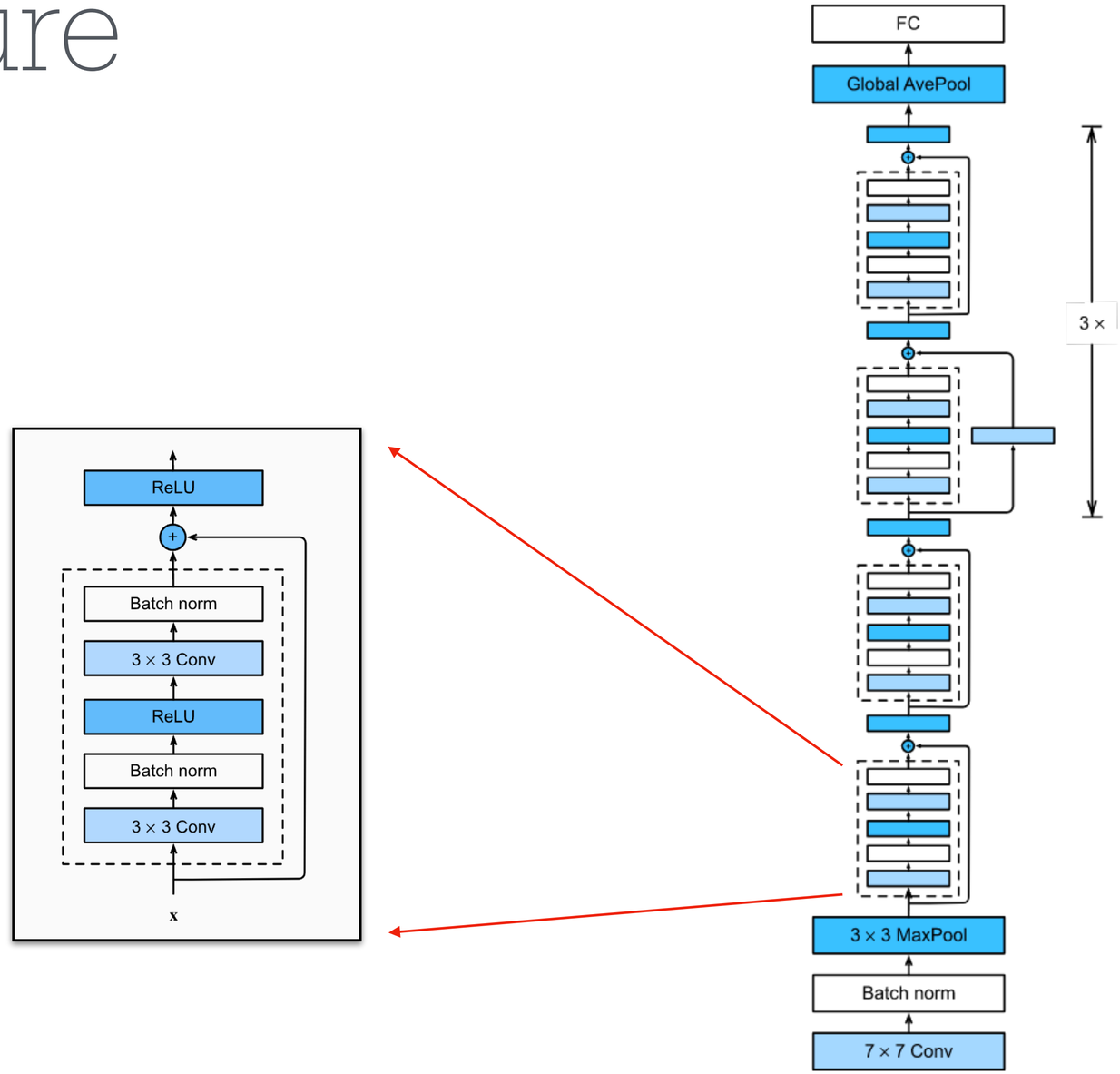
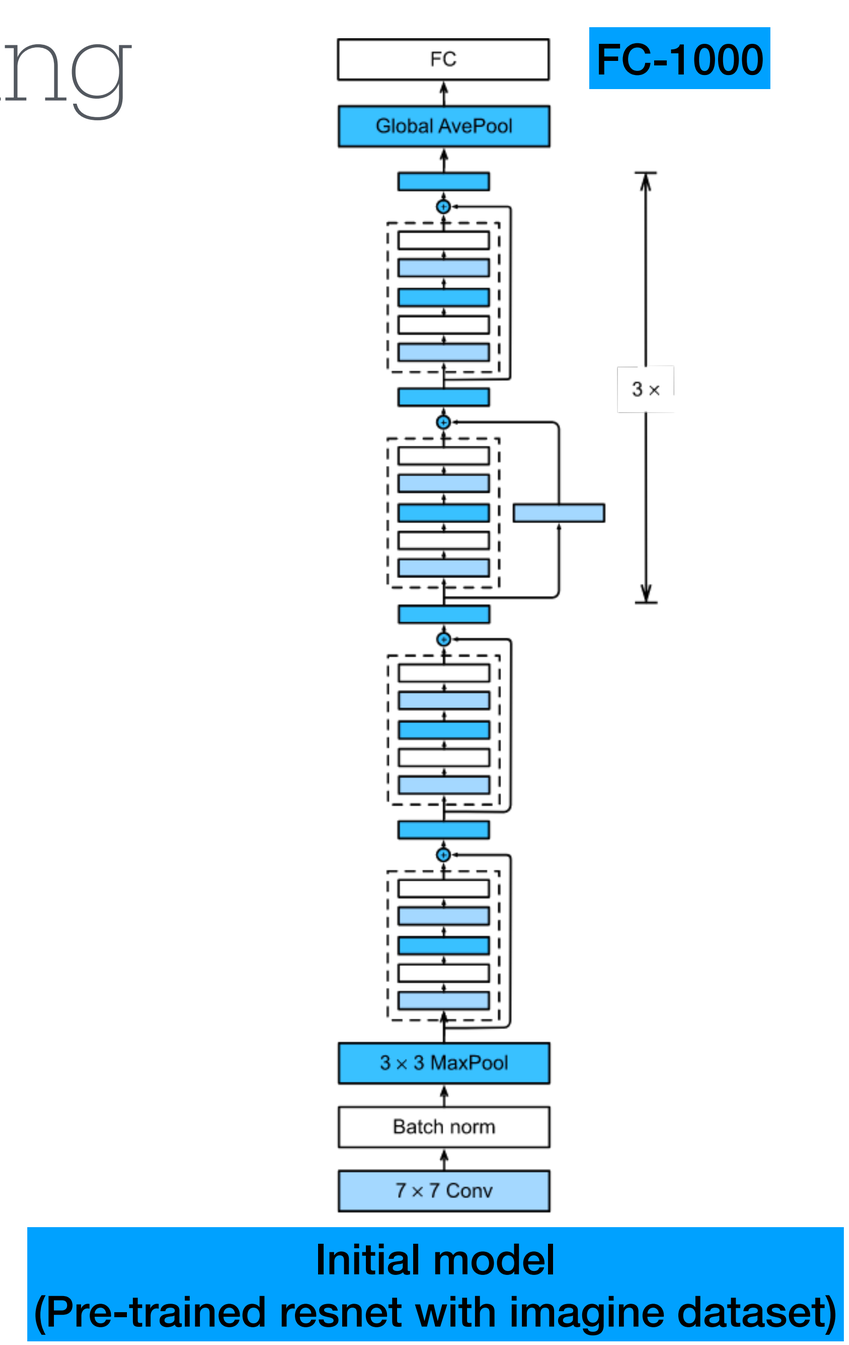
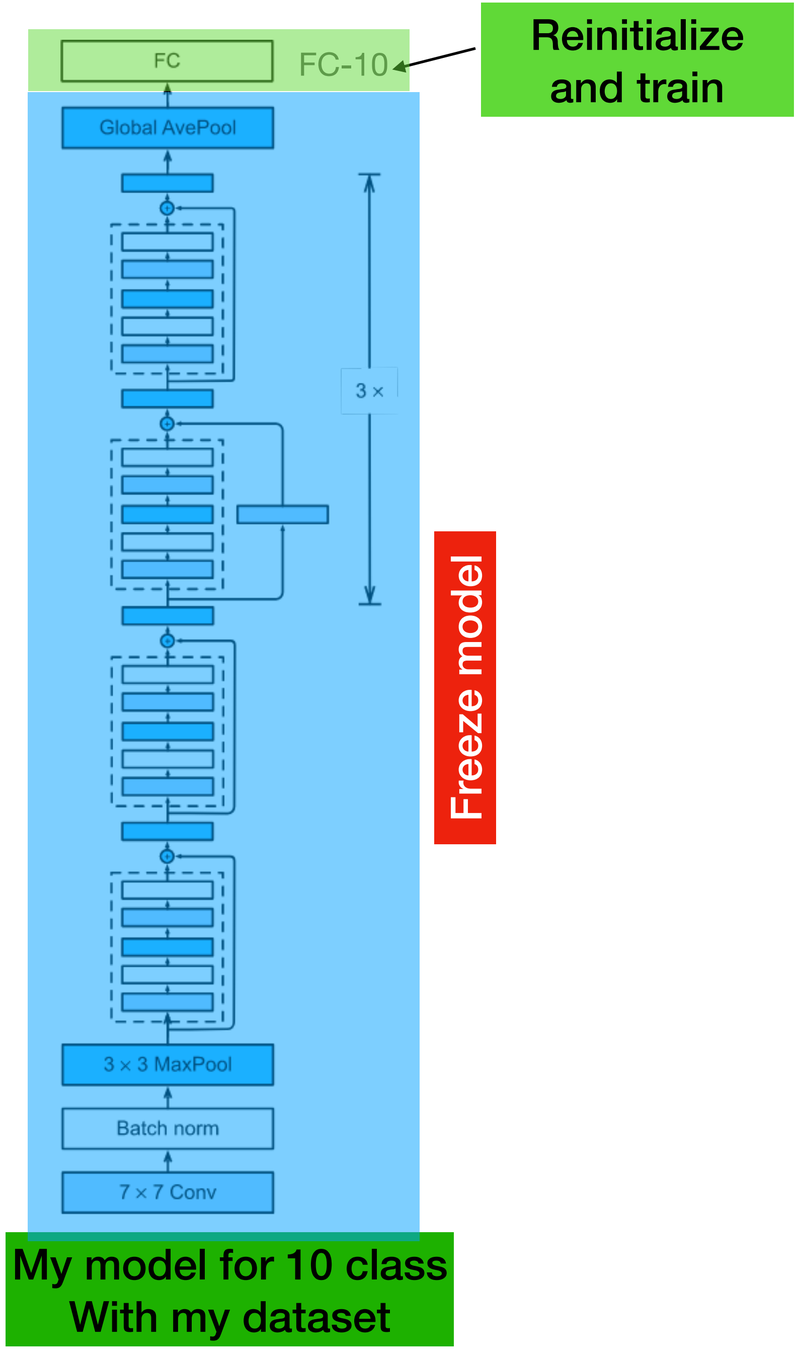


Image source: https://d2l.ai/chapter_convolutional-modern/resnet.html

Transfer learning

- Pre-trained model:a model with large dataset (Imagenet)
- Keep the Conv layers as a backbone network
- Replace FC(s) to number classes for the model
- Pre-trained backbone network is used everywhere





Sequential problems

There are many applications requires sequential data processing

Sequential data: data at time t has relationship with previous time steps

Various types of input and outputs used in sequential applications

A regular neural network might not use the sequential nature of the input data

A neural network might not generate sequential outputs

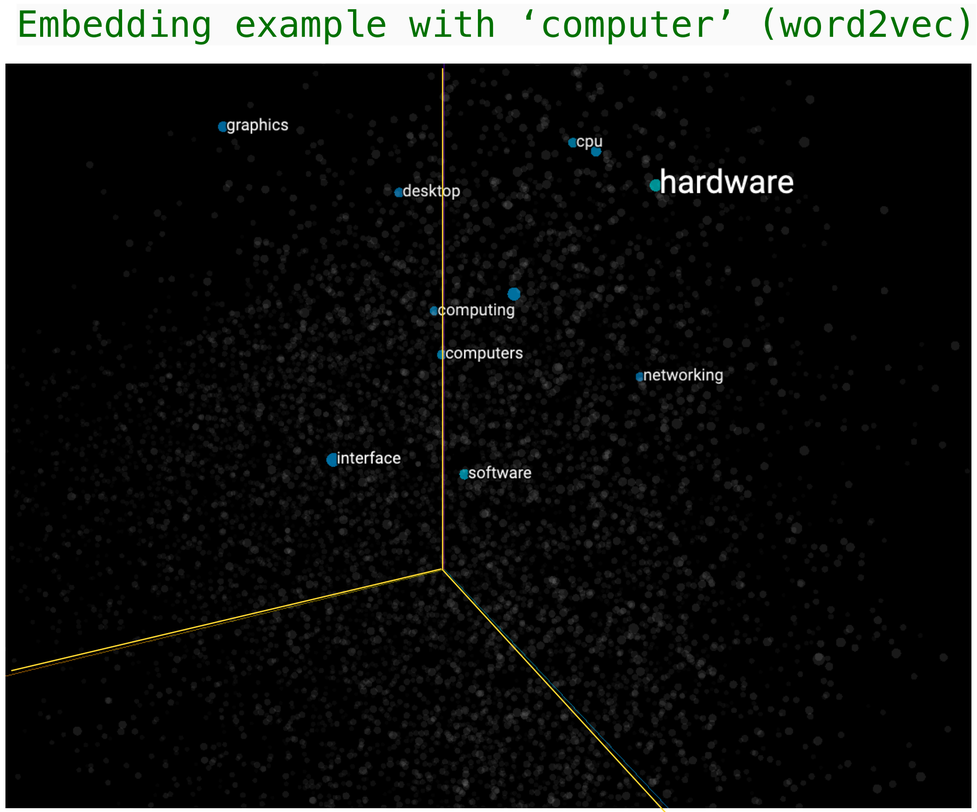
Types	Input	Output
Time series forecasting	Time series 	Next time step prediction
Sentiment analysis	movie review Rotten Tomatoes	Predicted sentiment 
Language translation	English text	日本語 普通话 한국어
Speech recognition		Transcribed text
Document summarization	Document text	Summarized text
Image captioning		Describing scene, planes flying
Question answering	what is the tallest building in the world?	Shanghai Tower

Photo by Anton on Unsplash Photo by Andrey Kremkov on Unsplash

Adapted from : "RNNs" section of Full Stack Deep Learning

Word2vec visualization

- Visualization: https://projector.tensorflow.org



Sequential problems

Problem setup of sequential problems

One to many:

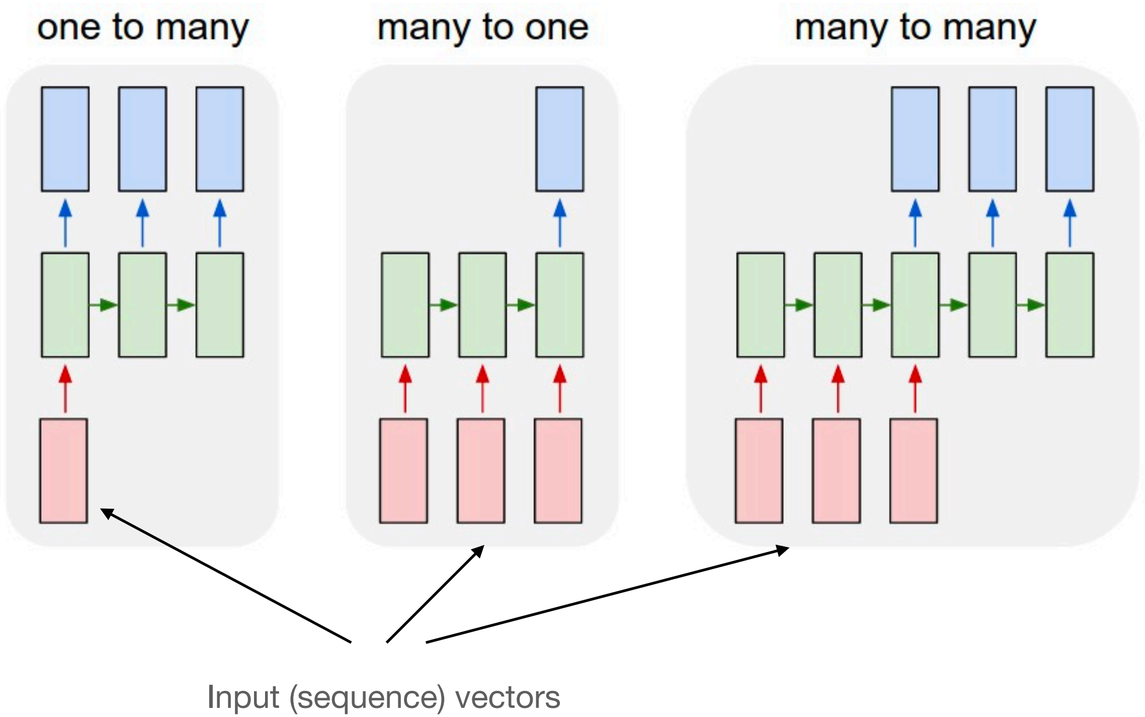
- Image captioning

Many to one:

- Sentiment analysis, time series forecast

Many to many:

- Machine translation



Adapted from : http://karpathy.github.io/2015/05/21/rnn-effectiveness/

RNN in code

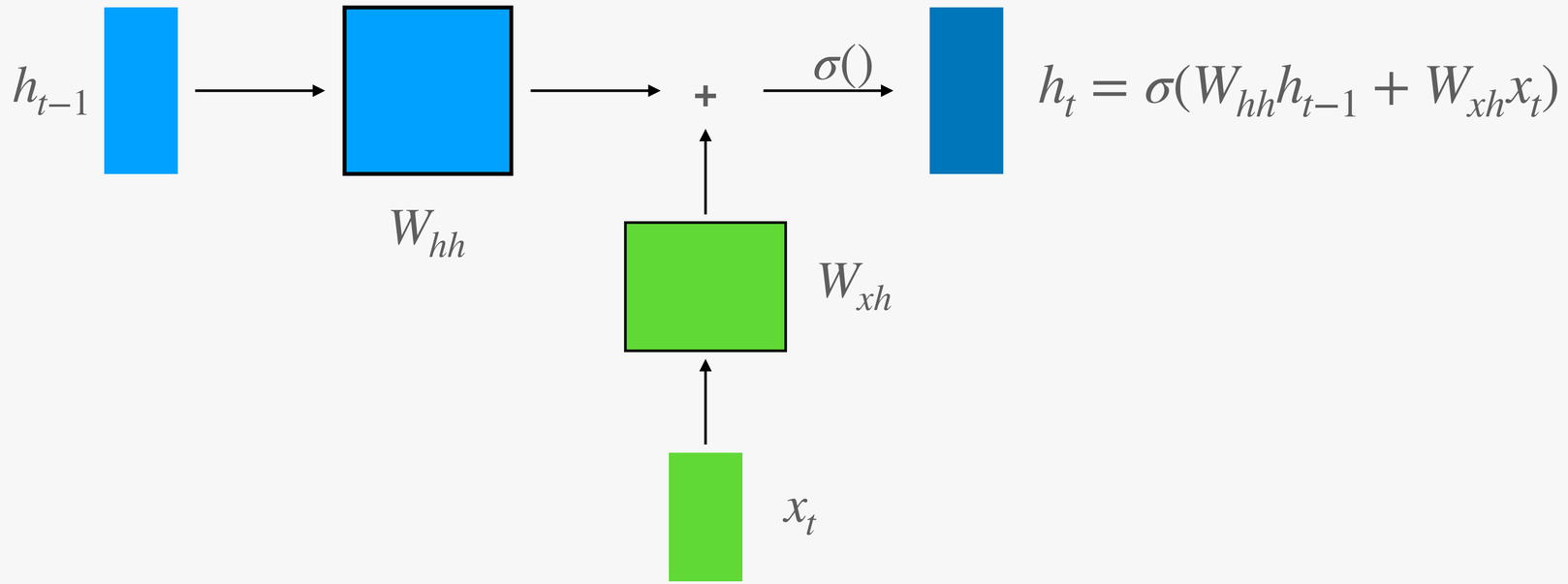
```
class RNN:
  # ...
  def compute_next_h(self, x):
    # Vanilar RNN hidden computation
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    return h
  # ...
  def step(self, x):
    # update the hidden state
    self.h = self.compute_next_h(x)
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t)$$

Adapted from : http://karpathy.github.io/2015/05/21/rnn-effectiveness/

RNN in code 2

```
class RNN:
```

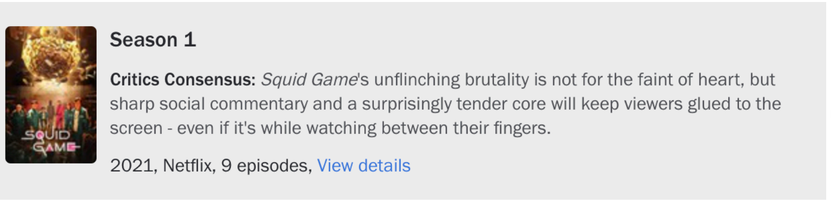


Adapted from : https://drive.google.com/file/d/1bn801i0Brs2FypxDyjBIzJyrNYQof80J/view

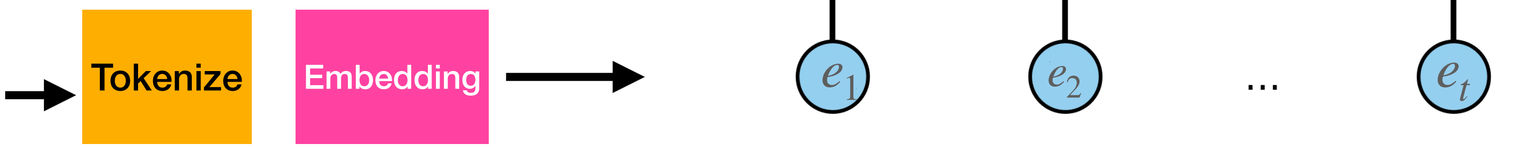
RNN applications

Many to one setup

- Add the fully connected layer at the last hidden state
- Use it as a classification
- RNN is working as encoder, encodes input text
- FC is working as a decoder that use the encoded information from RNN to use it for classification



Input text



Adapted from : https://drive.google.com/file/d/1bn801i0Brs2FypxDyjBIzJyrNYQof80J/view

RNN applications

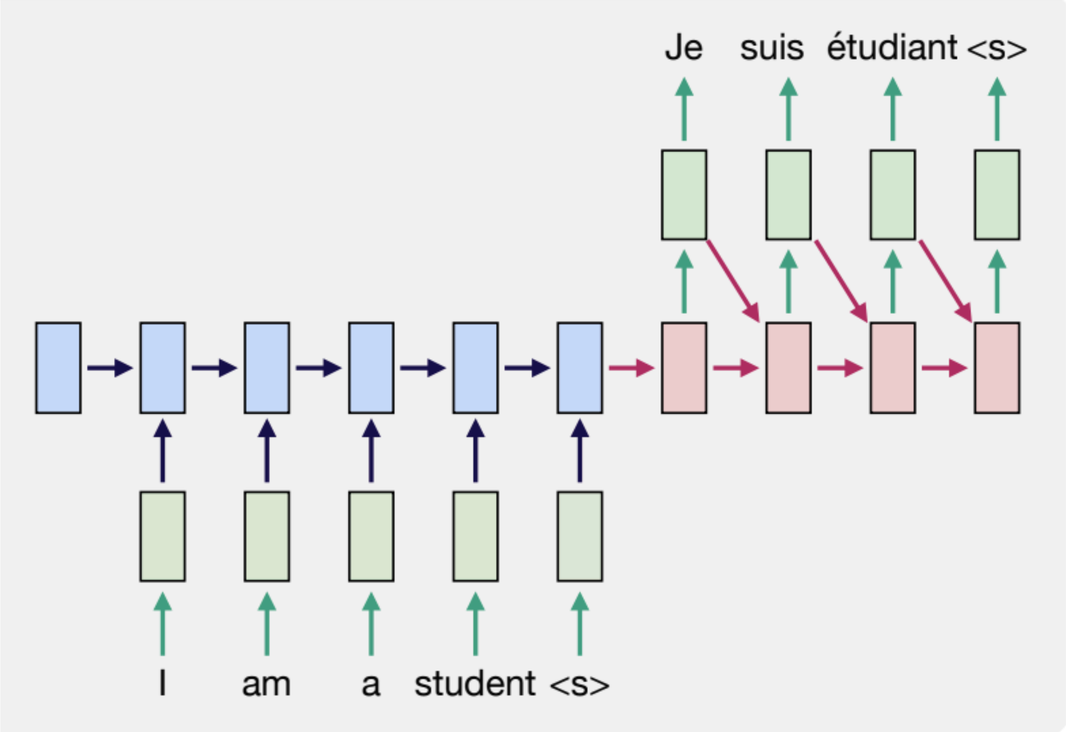
Language translation

Many to many setup consists of encoder RNN and decoder RNN

Encoder decoder architecture

- Also called sequence to sequence (seq2seq)

Dramatically replace all statistical machine translation (SMT) method to RNN based neural machine translation (NMT) in 2014



Note: Another changes in NLP: attention based improvement and transformer type model have been dominating in NLP since 2019

Source: https://drive.google.com/file/d/1bn801i0Brs2FypxDyjBIzJyrNYQof80J/view