UCLA Samueli Computer Science

CS M146 Discussion: Week 6

Neural Networks, Learning Theory, Kernels, PyTorch

Junheng Hao

Friday, 02/12/2021

Roadmap

- Announcement
- Neural Nets: Back Propagation
- Learning Theory
- Programming Guide: PyTorch

Announcements

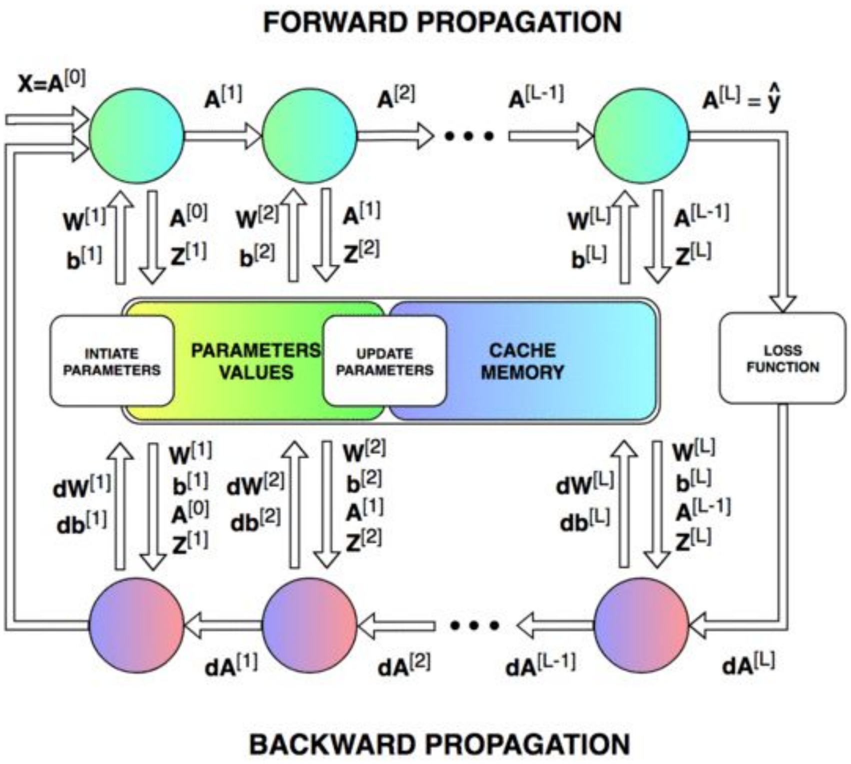
- **5:00 pm PST, Feb 12 (Friday):** Weekly Quiz 6 released on Gradescope.
- **11:59 pm PST, Feb 14 (Sunday):** Weekly quiz 6 closed on Gradescope!
  - Start the quiz before **11:00 pm Feb 14, Feb 14** to have the full 60-minute time
- **Problem set 1: Regrade request due today**
- **Problem set 3: Problem set 1:** Will be released later today, due **Feb 26 11:59PM PST**
- **Problem set 2** submission on Gradescope.
  - Please assign pages of your submission with corresponding problem set outline items on GradeScope.
  - Due on **TODAY 11:59pm PST, Feb 12 (Friday)**

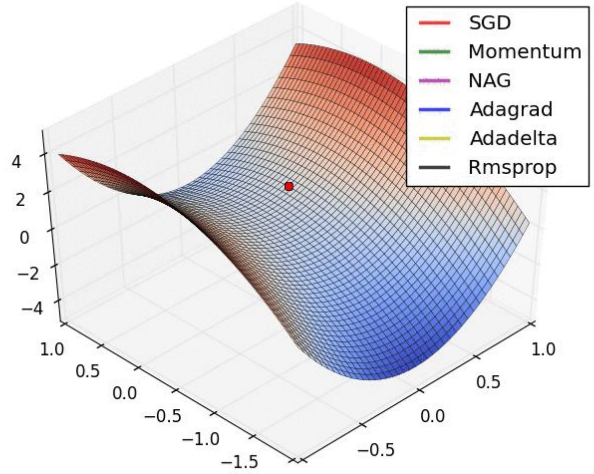
Late Submission of PS will NOT be accepted!

About Quiz 6

- Quiz release date and time: **Feb 12, 2021 (Friday) 05:00 PM PST**
- Quiz due/close date and time: **Feb 14, 2021 (Sunday) 11:59 PM PST**
- You will have up to **60 minutes** to take this exam. → Start before **11:00 PM** Sunday
- You can find the exam entry named "Week 4 Quiz" on GradeScope.
- Topics: **Neural Nets, Learning Theory**
- Question Types
  - True/false, multiple choices
  - Some questions may include several subquestions.
- Some light calculations are expected. Some scratch paper and one scientific calculator (physical or online) are recommended for preparation.

Neural Networks: Backpropagation





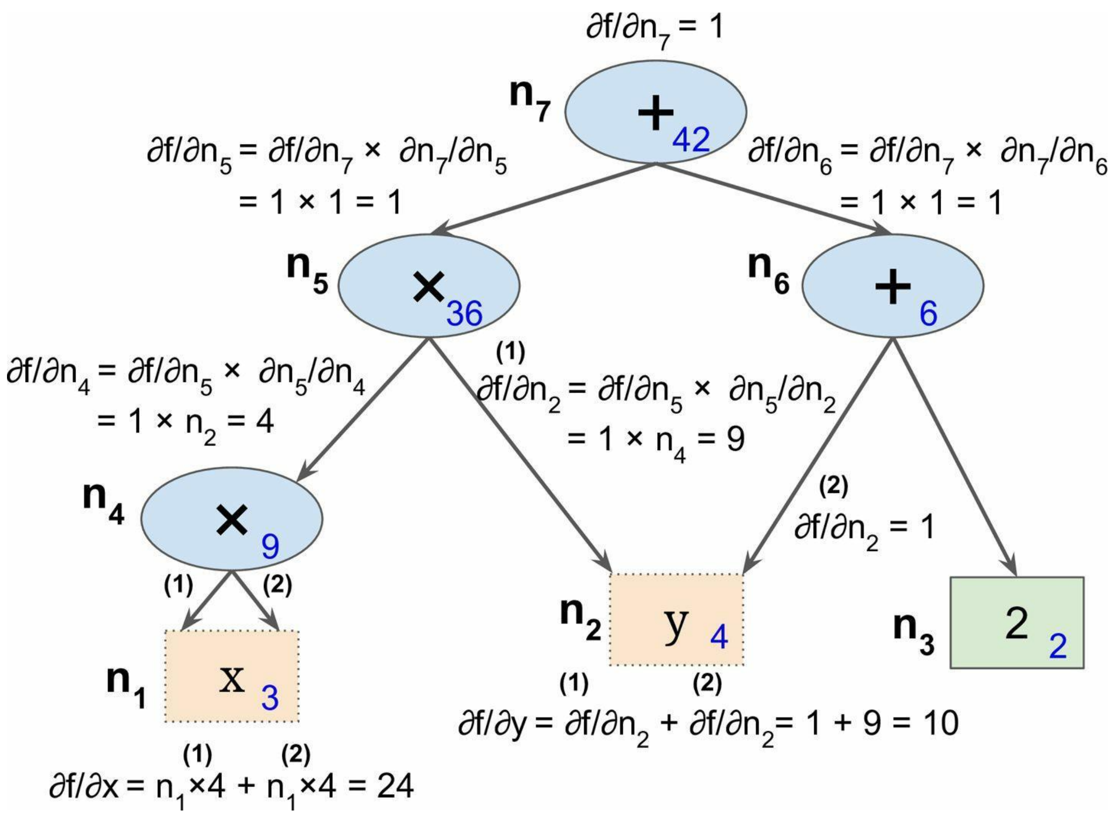
https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e

Neural Networks: Backpropagation

- A simple example to understand the intuition
- **f(x,y) = x²y + y + 2**
- Forward pass:
  - x=3, y=4 → f(3,4)=42
- Backward pass:
  - Chain rule:

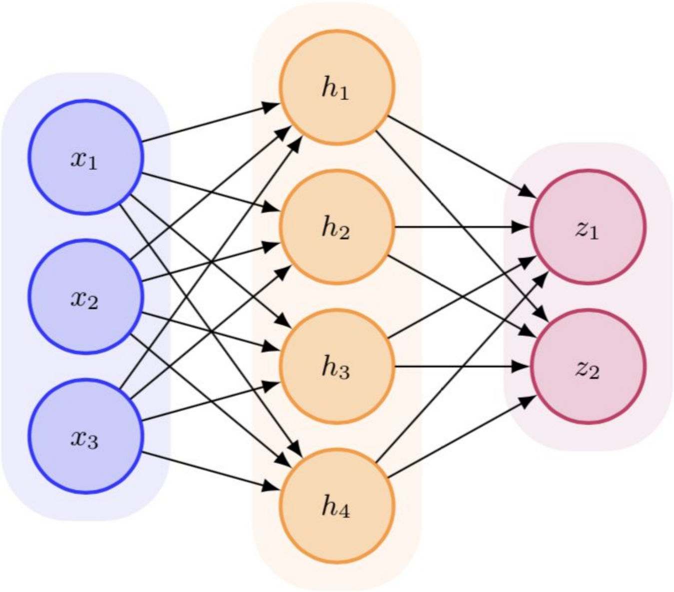
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Another better demo: http://colah.github.io/posts/2015-08-Backprop/

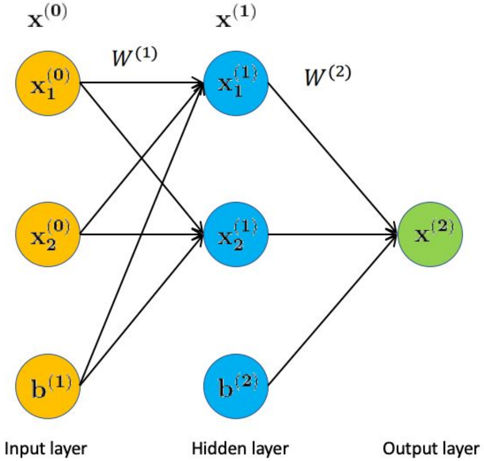


2-Layer NN Example

Demo in class : Back propagation for a 2-layer network



Backprop: Exercise

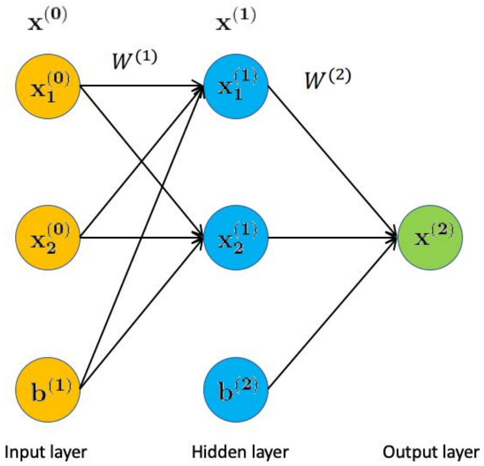


In this question, let's consider a simple two-layer neural network and manually do the forward and backward pass. For simplicity, we assume our input data is two dimension. Then the model architecture looks like the following. Notice that in the example we saw in class, the bias term `b` was not explicit listed in the architecture diagram. Here we include the term `b` explicitly for each layer in the diagram. Recall the formula for computing $\mathbf{x}^{(l)}$ in the l-th layer from $\mathbf{x}^{(l-1)}$ in the $(l-1)$-th layer is $\mathbf{x}^{(l)} = \mathbf{f}^{(l)}(\mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})$. The activation function $\mathbf{f}^{(l)}$ we choose is the `sigmoid` function for all layers, i.e. $\mathbf{f}^{(l)}(z) = \frac{1}{1+\exp(-z)}$. The final loss function is $\frac{1}{2}$ of the `mean squared error` loss, i.e. $l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||^2$.

We initialize our weights as

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix}, \quad \mathbf{W}^{(2)} = [0.4, 0.45], \quad \mathbf{b}^{(1)} = [0.35, 0.35], \quad \mathbf{b}^{(2)} = 0.6$$

Backprop: Exercise



$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix}, \quad \mathbf{W}^{(2)} = [0.4, 0.45],$$

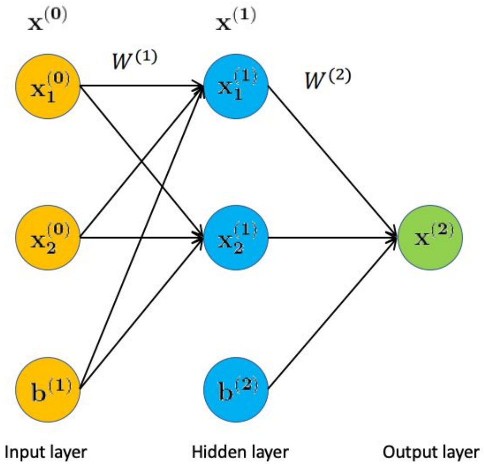
$$\mathbf{b}^{(1)} = [0.35, 0.35], \quad \mathbf{b}^{(2)} = 0.6$$

$$\text{input } \mathbf{x}^{(0)} = [0.05, 0.1]$$

Forward Pass

1. When the input $\mathbf{x}^{(0)} = [0.05, 0.1]$, what will be the value of $\mathbf{x}^{(1)}$ in the hidden layer? (Show your work).
2. Based on the value $\mathbf{x}^{(1)}$ you computed, what will be the value of $\mathbf{x}^{(2)}$ in the output layer? (Show your work).
3. When the target value of this input is $y = 0.01$, based on the value $\mathbf{x}^{(2)}$ you computed, what will be the loss? (Show your work).

Backprop: Exercise



$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix}, \quad \mathbf{W}^{(2)} = [0.4, 0.45],$$

$$\mathbf{b}^{(1)} = [0.35, 0.35], \quad \mathbf{b}^{(2)} = 0.6$$

$$\text{input } \mathbf{x}^{(0)} = [0.05, 0.1]$$

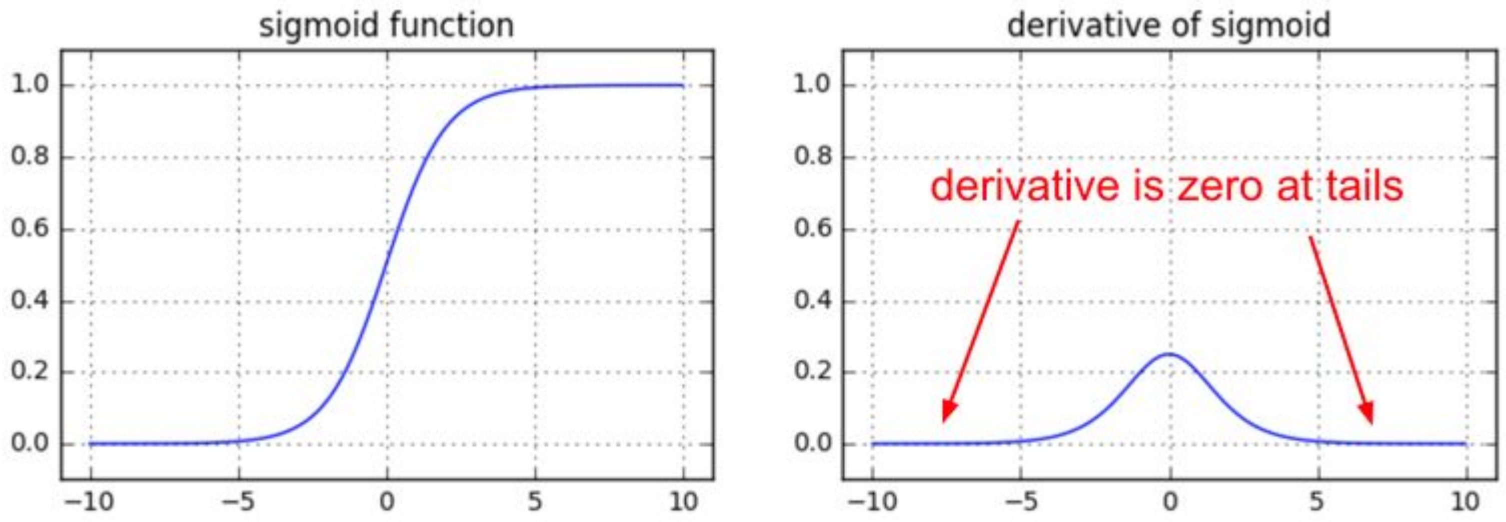
$\boxed{\text{target value of this input is } y = 0.01}$

Back Propagation

1. Consider the loss l of the same input $\mathbf{x}^{(0)} = [0.05, 0.1]$, what will be the update of $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ when we backprop, i.e. $\frac{\partial l}{\partial \mathbf{W}^{(2)}}$, $\frac{\partial l}{\partial \mathbf{b}^{(2)}}$
2. Based on the result you computed in part 1, when we keep backproping, what will be the update of $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$, i.e. $\frac{\partial l}{\partial \mathbf{W}^{(1)}}$, $\frac{\partial l}{\partial \mathbf{b}^{(1)}}$

Why understanding activation function?

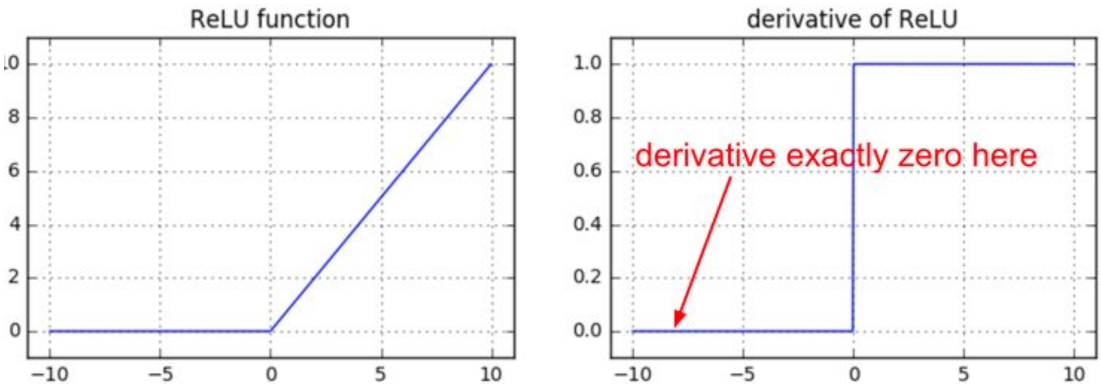
- “Why do we have to write the backward pass when frameworks in the real world, such as TensorFlow/PyTorch, compute them for you automatically?”
- Vanishing gradients on Sigmoids



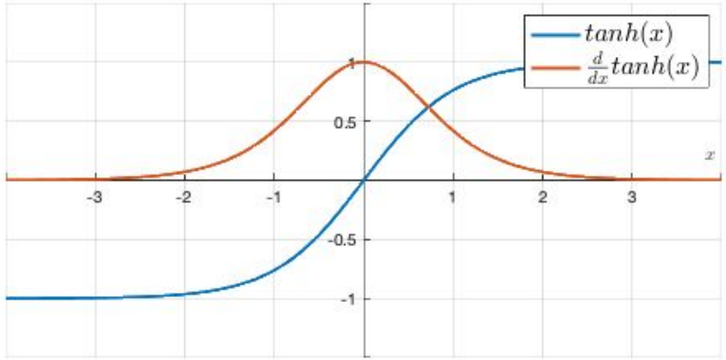
https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

Why understanding backpropagation?

- ReLUs



- tanh

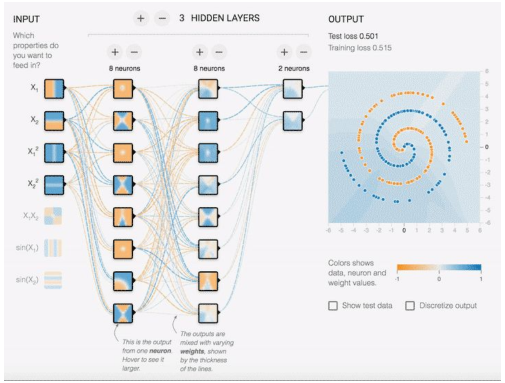


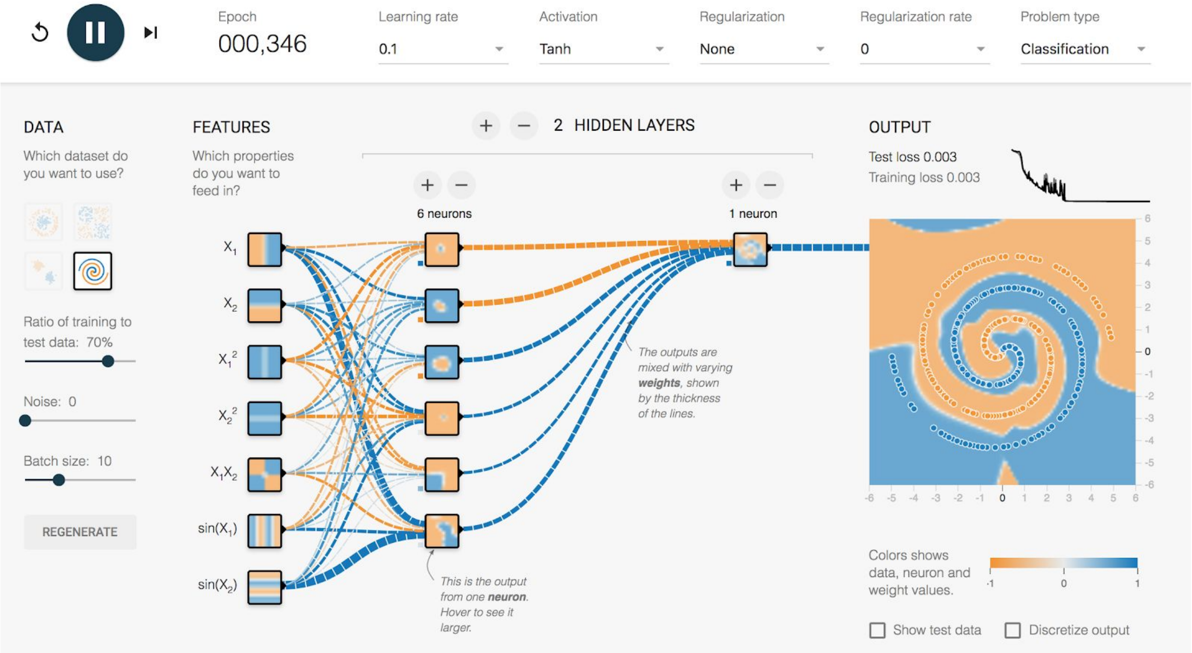
Why understanding backpropagation?

- Examples of non-linear activation functions: `Sigmoid, ReLU, leaky ReLU,` **`tanh,`** `etc`
- Properties we focus on:
  - Differentiable
  - Range: Whether saturated or not? (
  - Whether zero-centered or not?
- Activation function family
  - Wiki: https://en.wikipedia.org/wiki/Activation_function

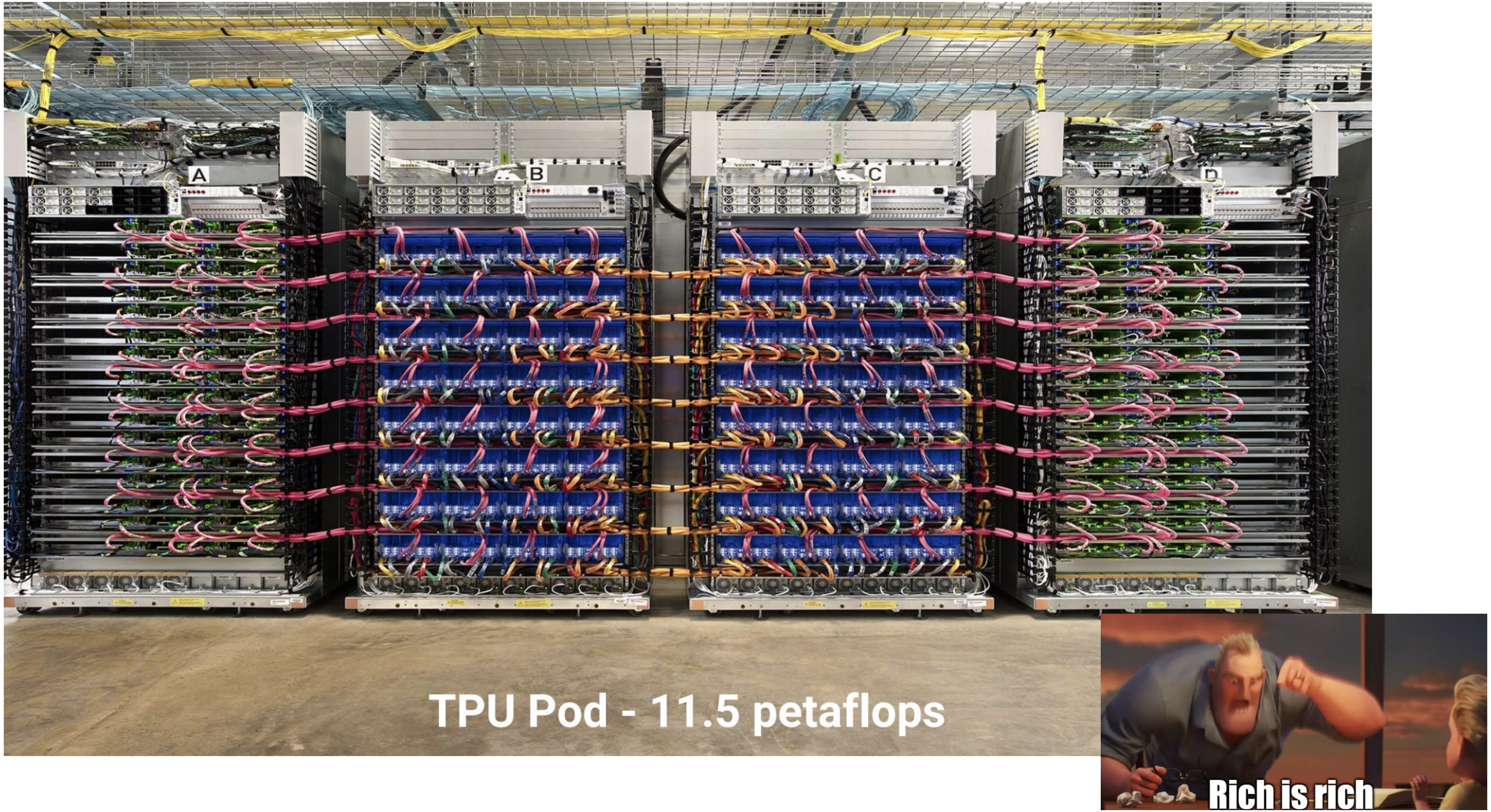
Neural Networks: Online Demo

- Let’s play with it: https://playground.tensorflow.org/



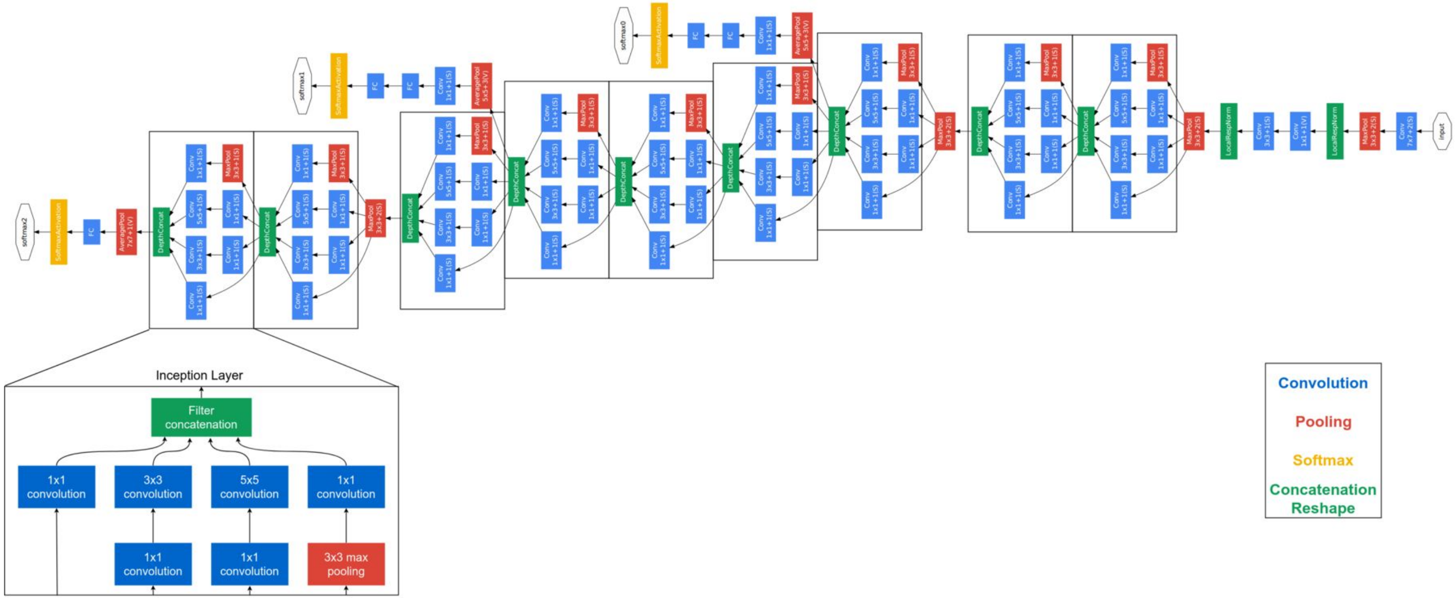


Story of Computing



UCLA
Engineer Change.

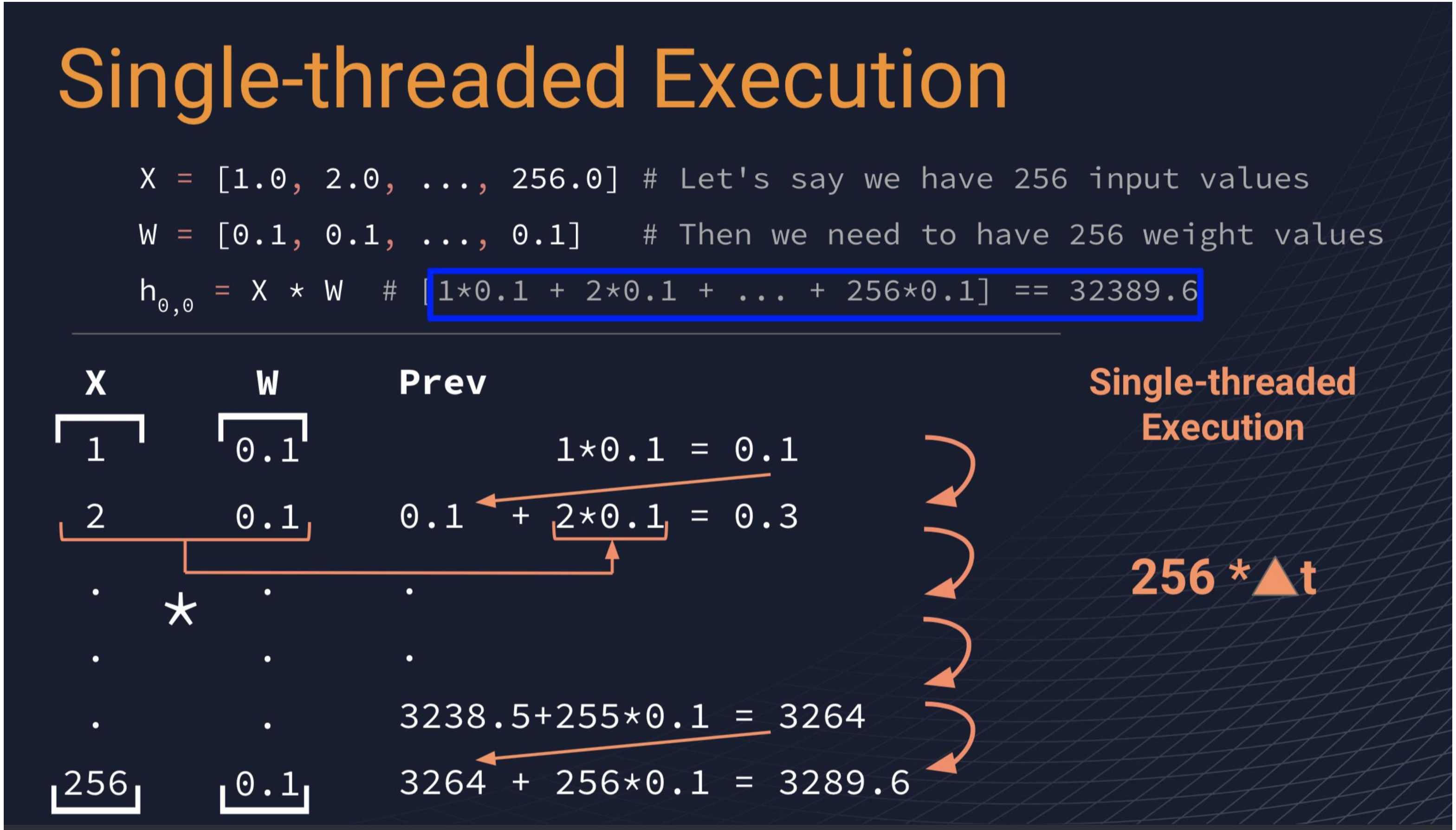
Story of Computing



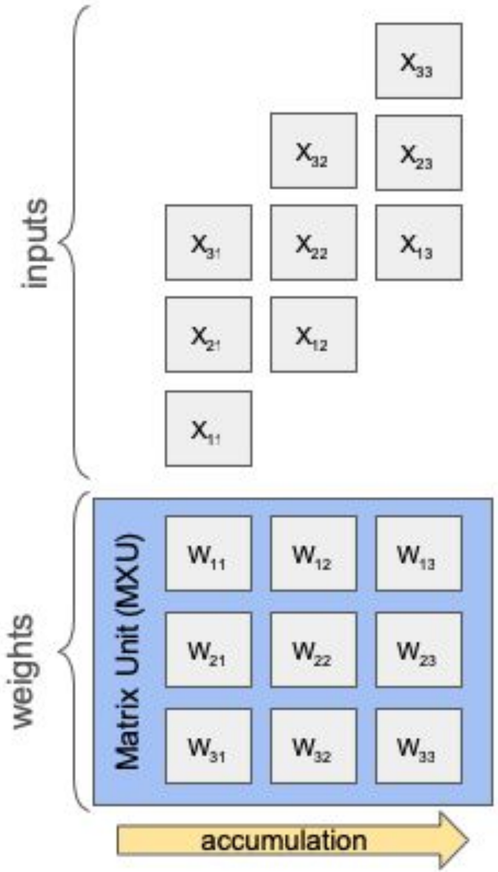
Matrix Multiplication is Eating (the computing resource of) the World!

UCLA Engineer Change.

Single-thread Computing of *X*W*



Neural Networks: Computation Example



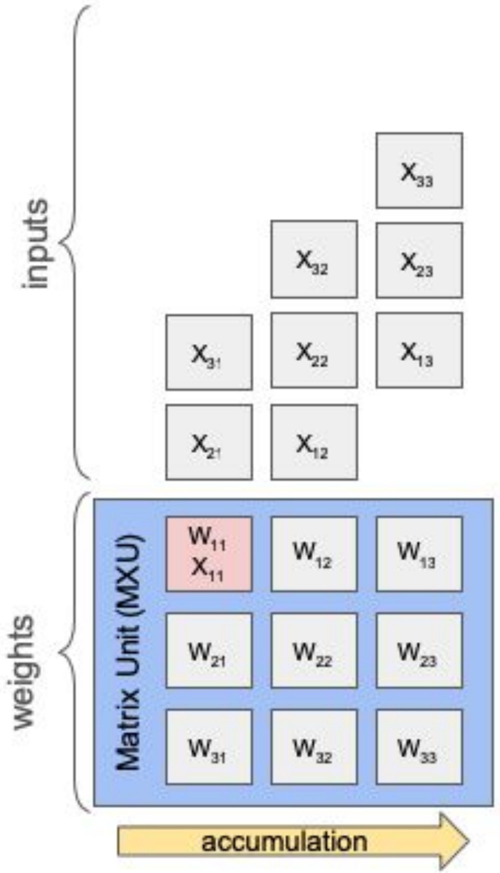
Matrix Unit Systolic Array

Computing y = Wx

3x3 systolic array
W = 3x3 matrix
Batch-size(x) = 3

UCLA Engineer Change.

Neural Networks: Computation Example

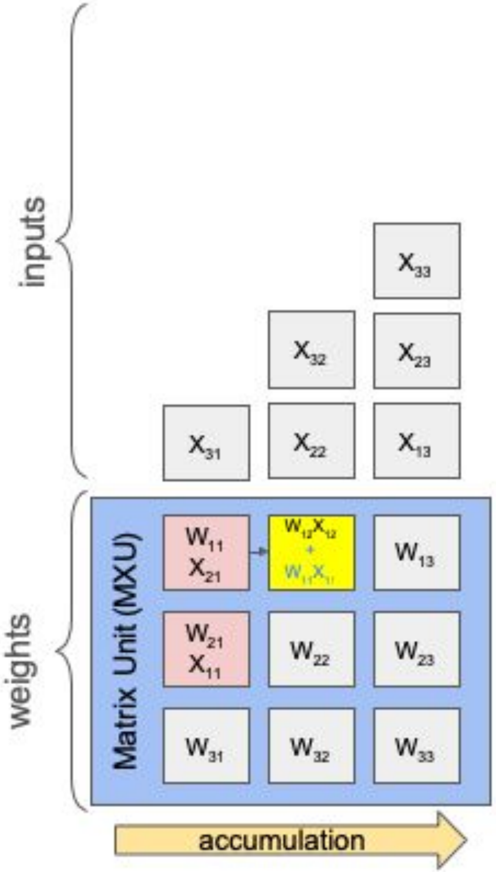


Matrix Unit Systolic Array

Computing y = Wx

with W = 3x3, batch-size(x) = 3

Neural Networks: Computation Example

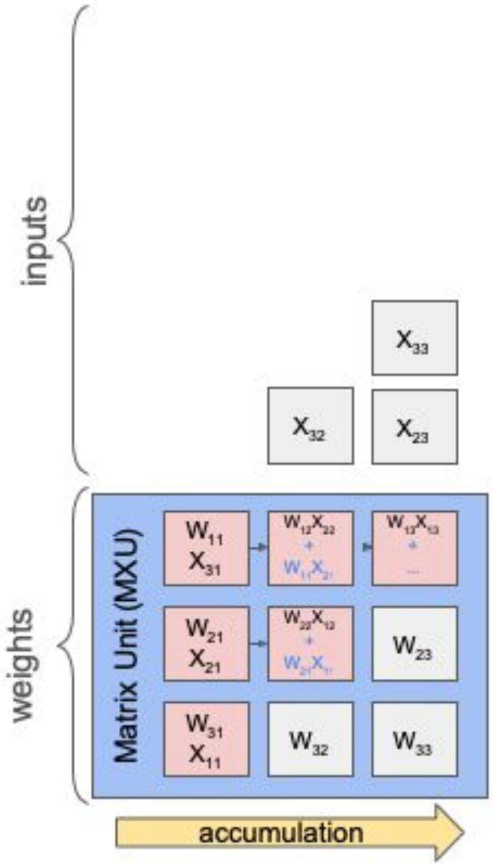


Matrix Unit Systolic Array

Computing $y = Wx$

with $W = 3x3$, batch-size$(x) = 3$

Neural Networks: Computation Example

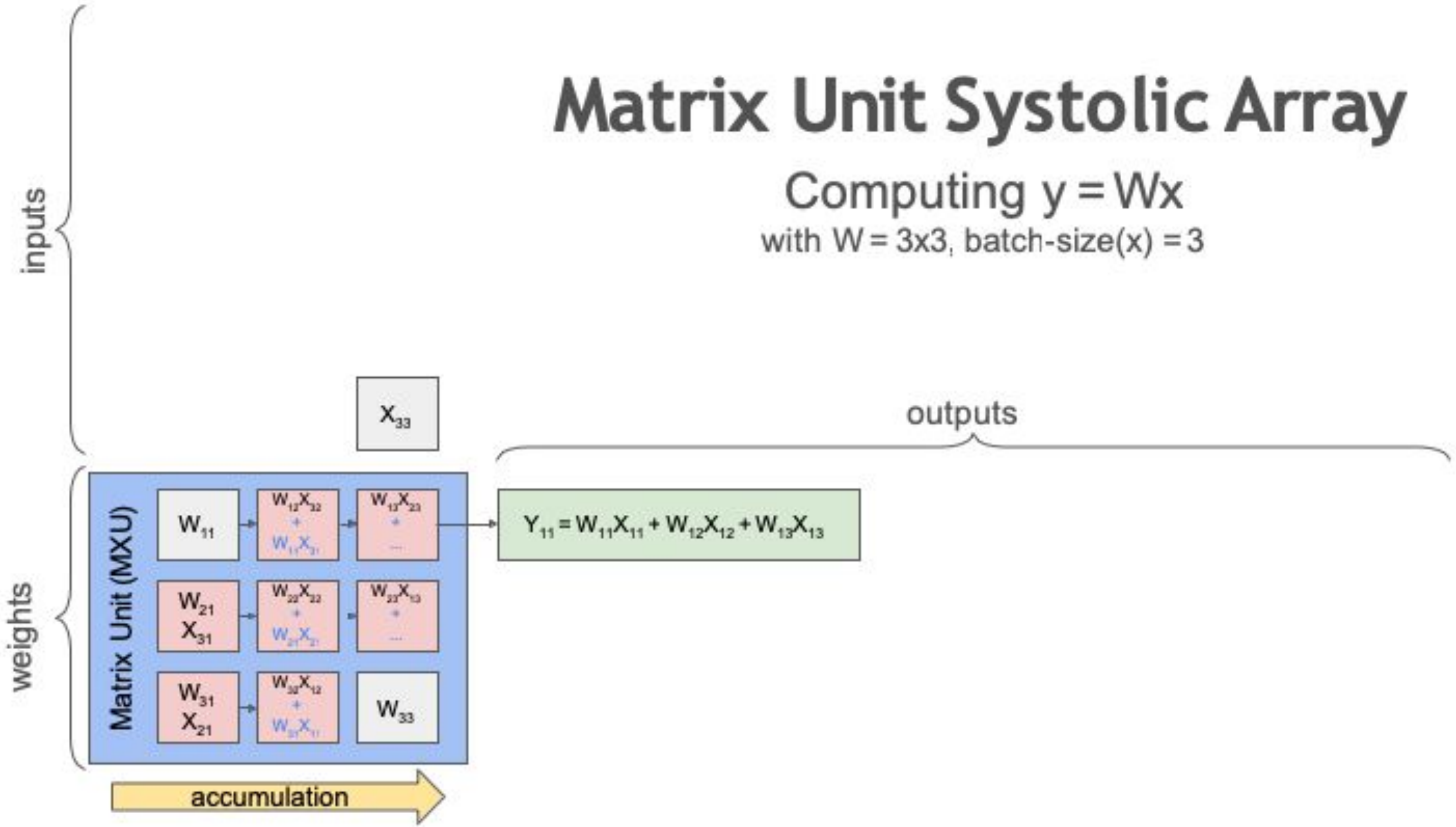


Matrix Unit Systolic Array

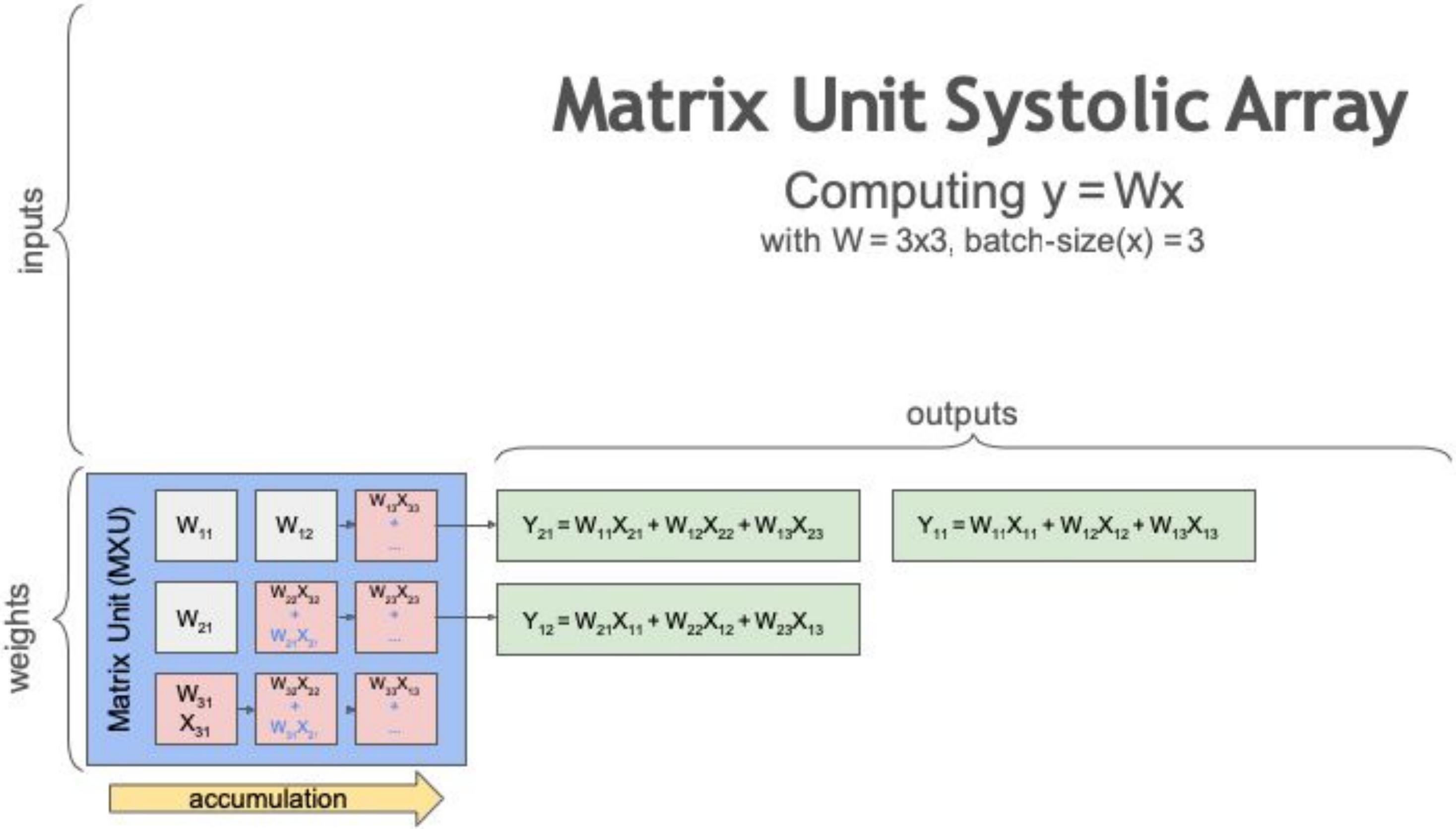
Computing $y = Wx$

with $W = 3x3$, batch-size(x) = 3

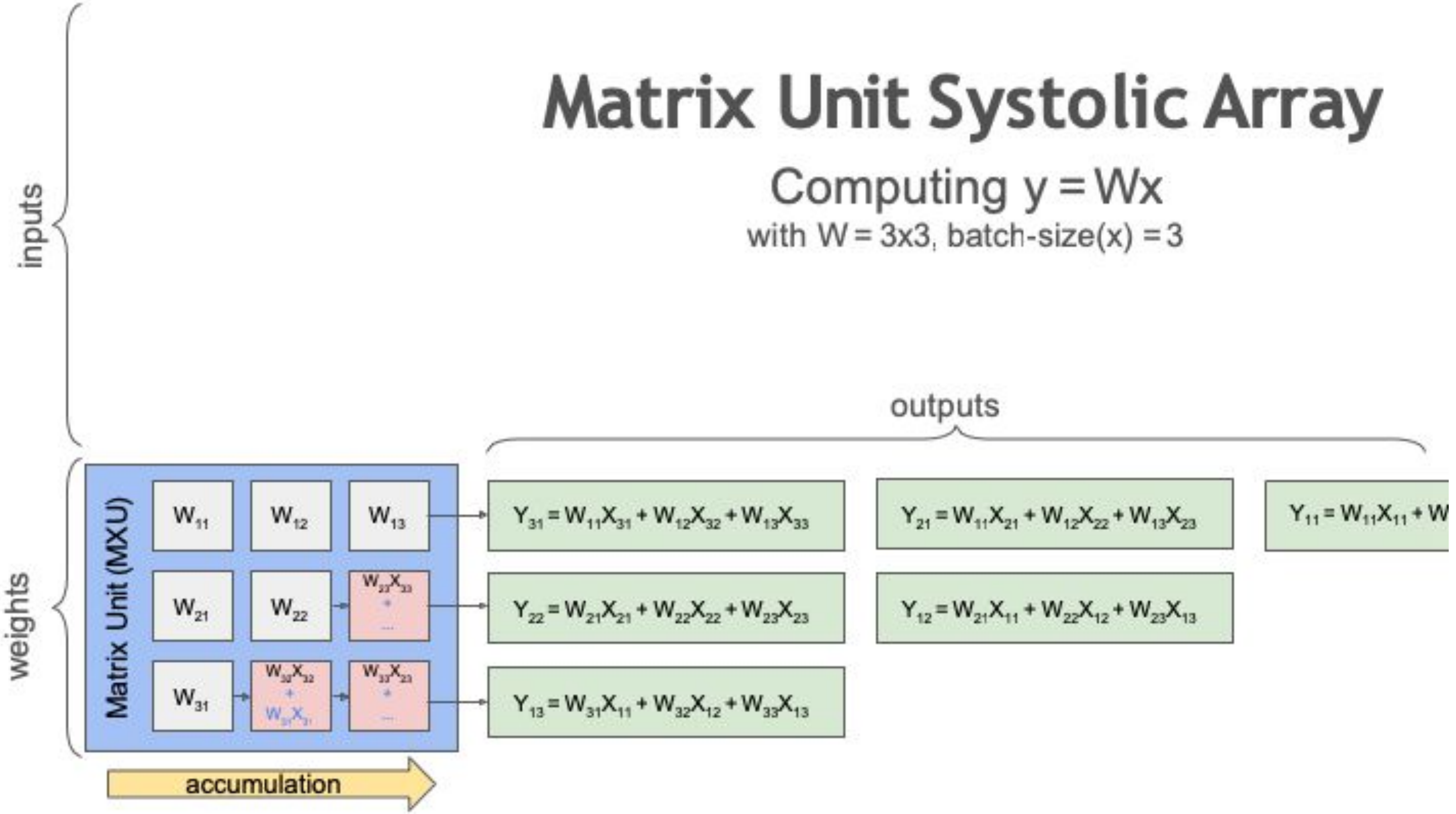
Neural Networks: Computation Example



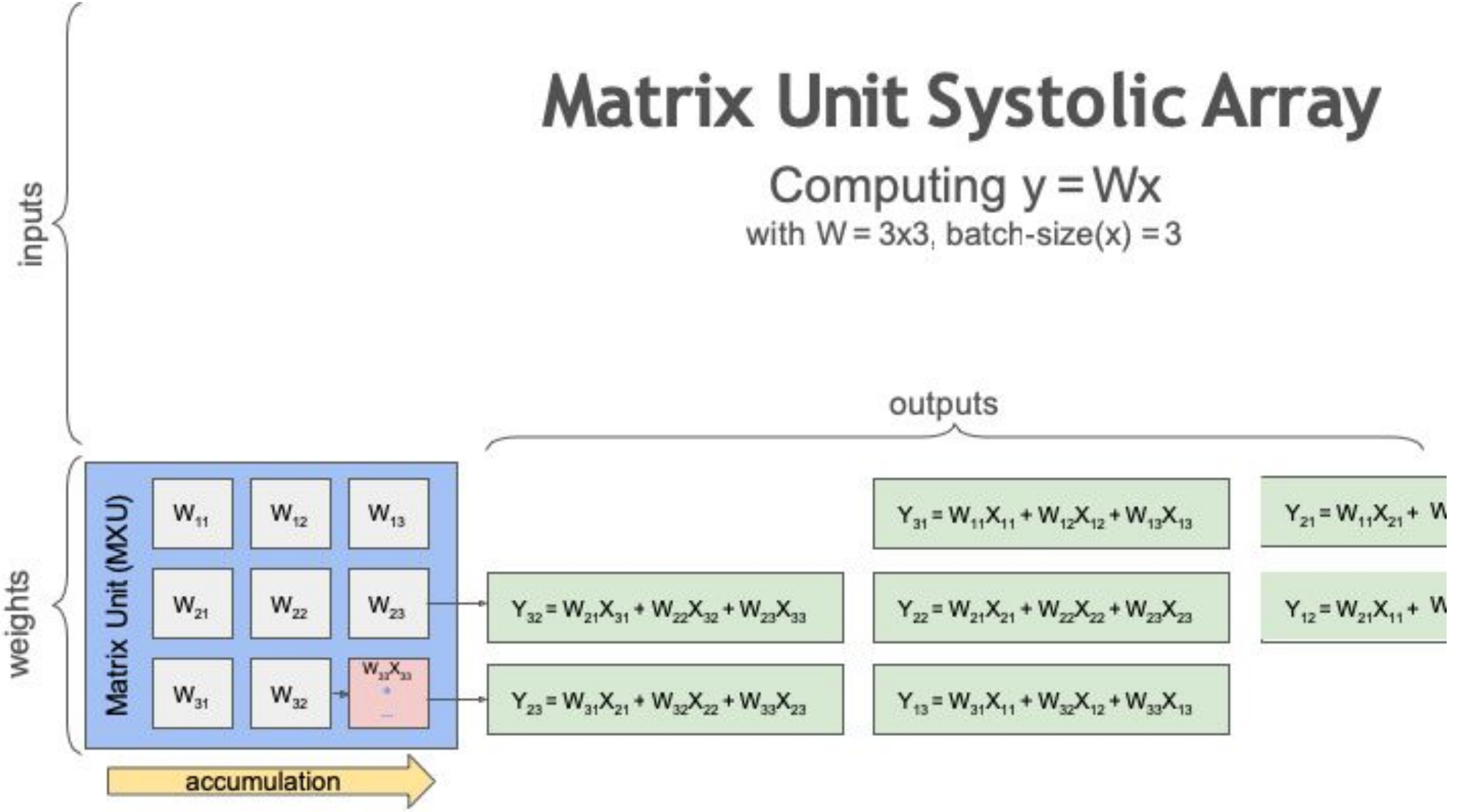
Neural Networks: Computation Example



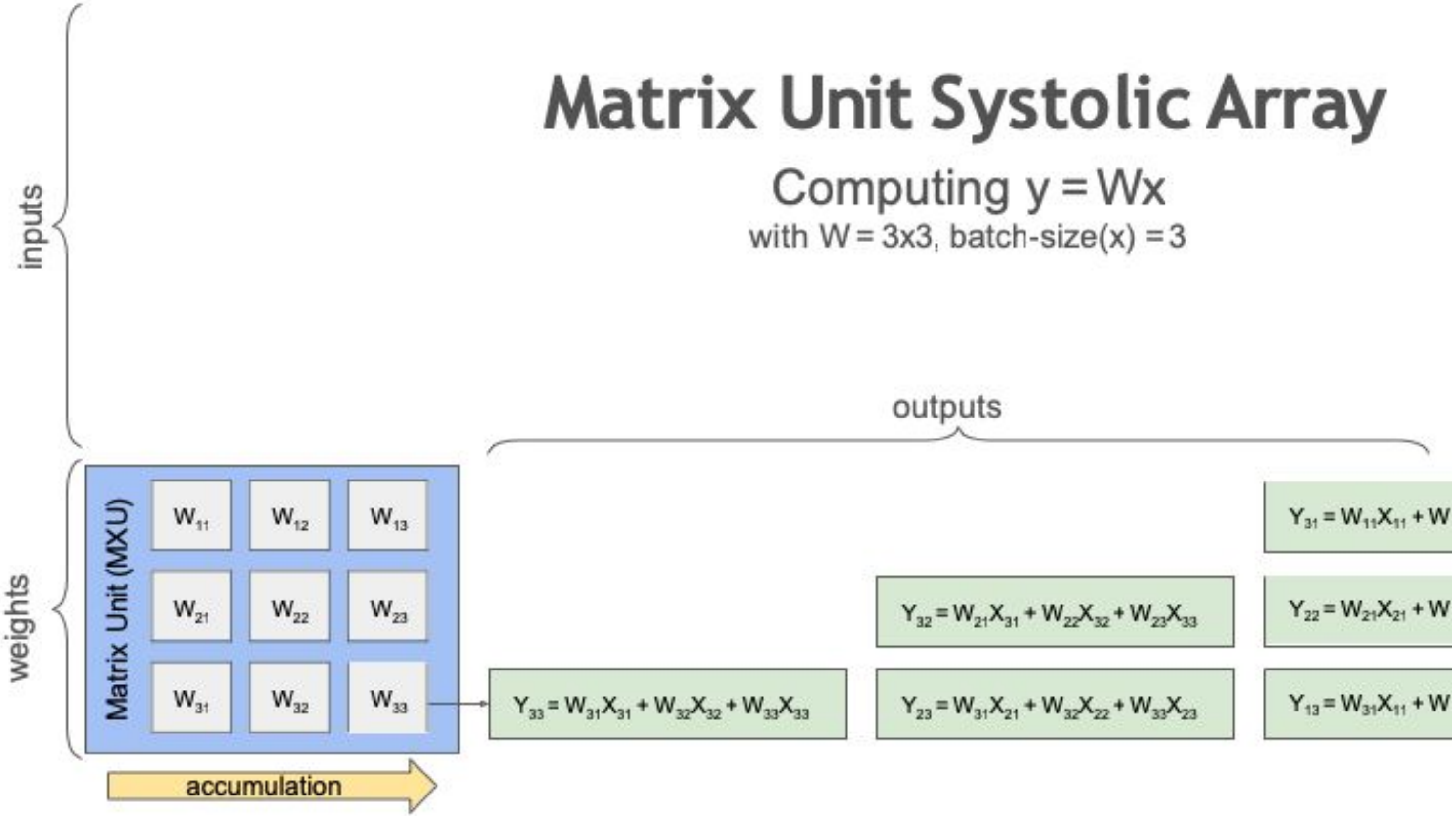
Neural Networks: Computation Example



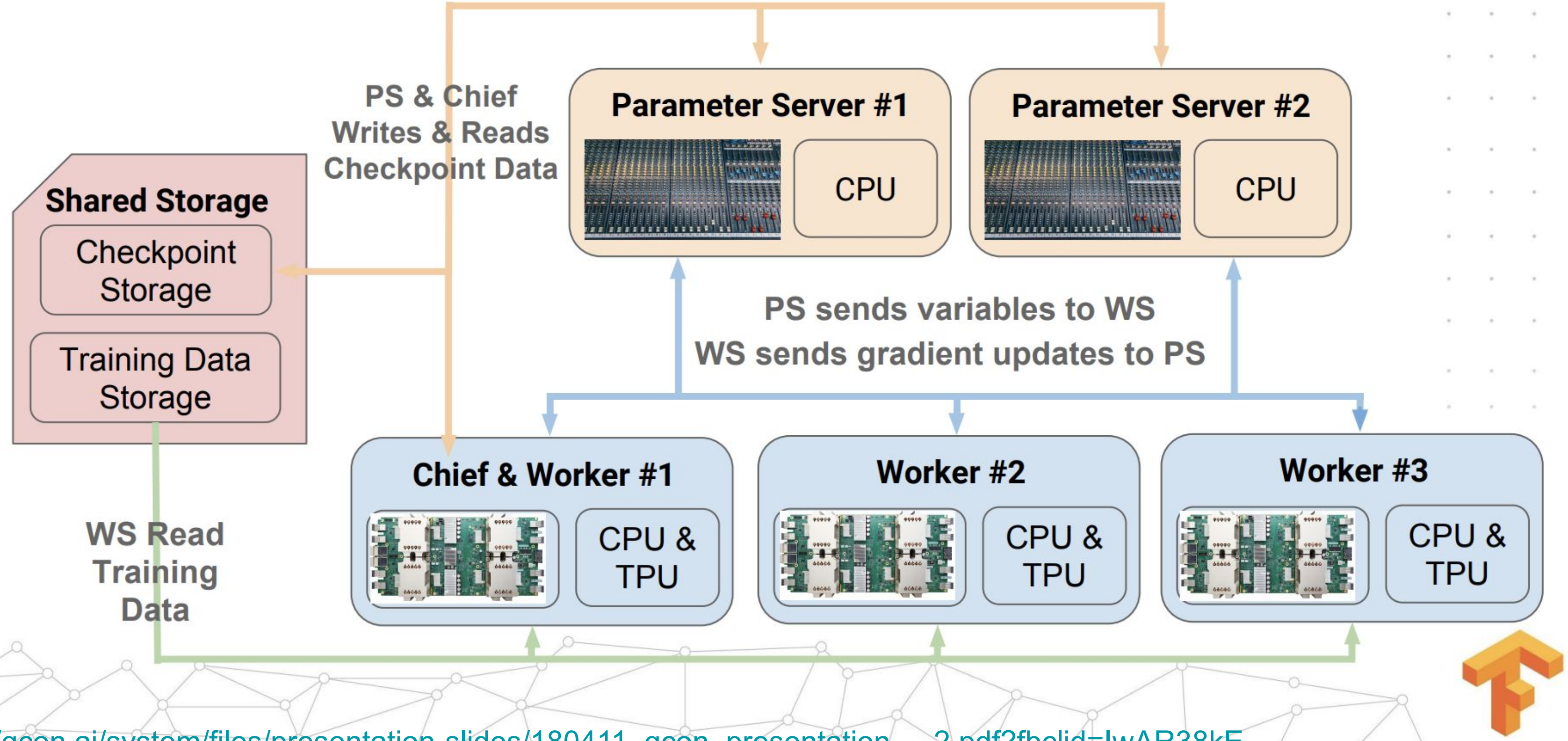
Neural Networks: Computation Example



Neural Networks: Computation Example



Neural Networks: Computation Example



https://qcon.ai/system/files/presentation-slides/180411_qcon_presentation_-_2.pdf?fbclid=IwAR38kESwm8e2NAhkj5JFqgz0F0VtnCpFyBp1HH5itsoSQllYvkyYEwsc9uY

UCLA
Engineer Change.

Learning Theory

- Let H be any finite hypothesis space. With probability 1 -δ a hypothesis $h \rightarrow H$ that is consistent with a training set of size m will have an error $< \epsilon$ on future examples if

$$m > \boxed{\frac{1}{\epsilon}}\left(\boxed{\ln(|H|)} + \boxed{\ln\frac{1}{\delta}}\right)$$

1. Expecting lower error increases sample complexity (i.e more examples needed for the guarantee)

3. If we want a higher confidence in the classifier we will produce, sample complexity will be higher.

2. If we have a larger hypothesis space, then we will make learning harder (i.e higher sample complexity)

Credit: http://cs229.stanford.edu/summer2020/cs229-notes4.pdf

VC Dimension

- Given a **hypothesis class** $\boldsymbol{H}$ over instance space X, we then define its Vapnik Chervonenkis dimension, written as $VC(H)$, to be the size of the largest finite subset of X that is shattered by H.
- In general, the VC dimension of an $\boldsymbol{n}$**-dimensional** linear function is $\boldsymbol{n+1}$

This term may decrease

This term will decrease

$$err_D(h) \leq err_S(h) + \sqrt{\frac{VC(H)\left(\ln \frac{2m}{VC(H)} + 1\right) + \ln \frac{4}{\delta}}{m}}$$

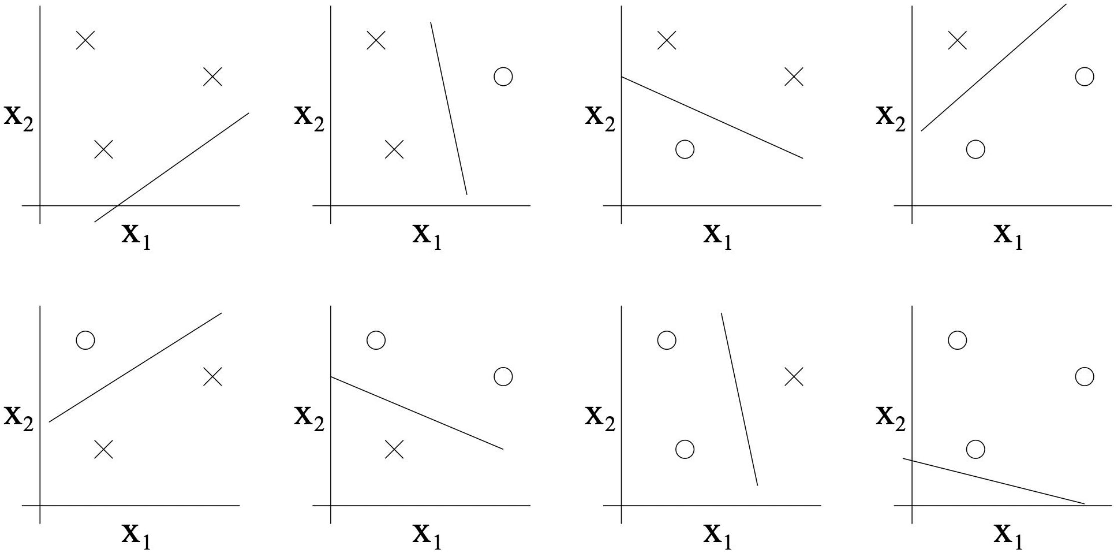
- Sample size for infinite H

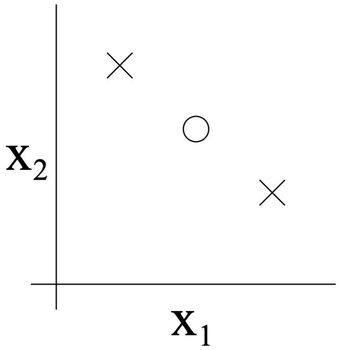
$$m \geq \frac{1}{\varepsilon}\left(4\log_2\left(\frac{2}{\delta}\right) + 8 \cdot VC(H)\log_2\left(\frac{13}{\varepsilon}\right)\right)$$

UCLA
Engineer Change.

VC Dimension of Half Space

- How to determine the set H of linear classifiers in two dimension has a VC(H)=3?

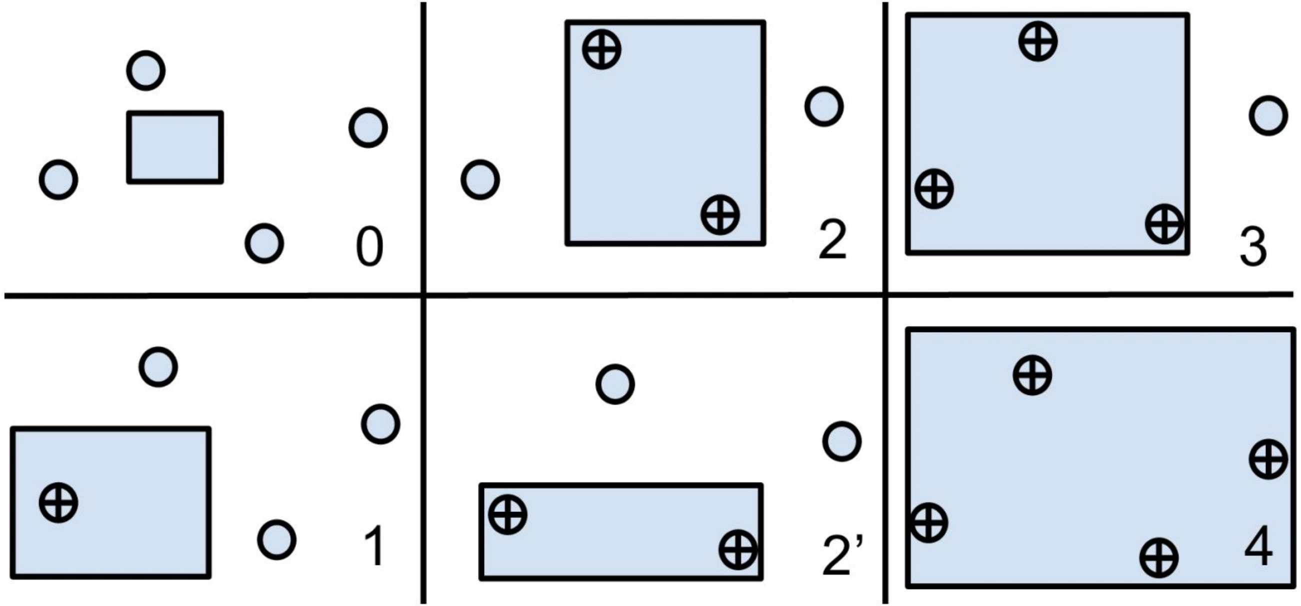




VC dimension of H here is 3 even though there may be sets of size 3 that it cannot shatter.

VC Dimension of Rectangles

- What is the VC Dimension of Axis-aligned rectangles?



Credit: https://www.cs.princeton.edu/courses/archive/spring14/cos511/scribe_notes/0220.pdf

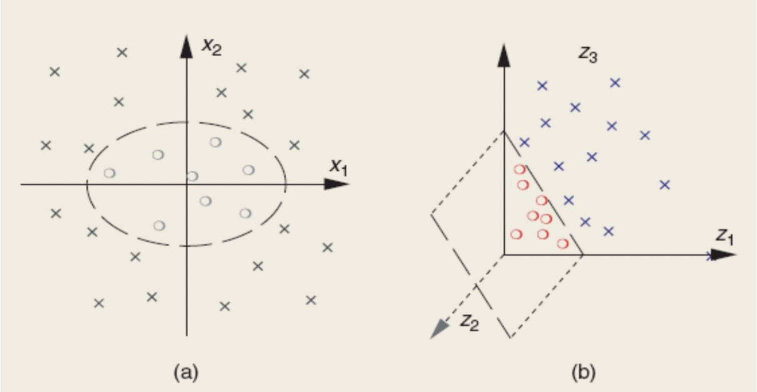
Kernels

- Motivation: Transformed feature space

- Basic idea: Define K, called kernel, such that:

$$K : X \times X \rightarrow \mathbb{R} \qquad \Phi(x) \cdot \Phi(y) = K(x, y)$$

which is often as a similarity measure.



- Benefit:
  - Efficiency: is often more efficient to compute than and the dot product.
  - Flexibility: can be chosen arbitrarily so long as the existence of is guaranteed (Mercer's condition).

Credit: https://cs.nyu.edu/~mohri/icml2011-tutorial/tutorial-icml2011-1.pdf

Polynomial Kernels

- **Definition:**

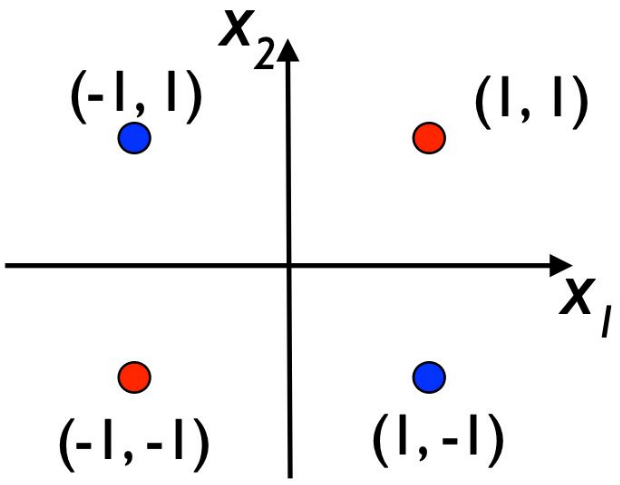
$$\forall x, y \in \mathbb{R}^N, \;\; K(x,y) = (x \cdot y + c)^d, \quad c > 0.$$

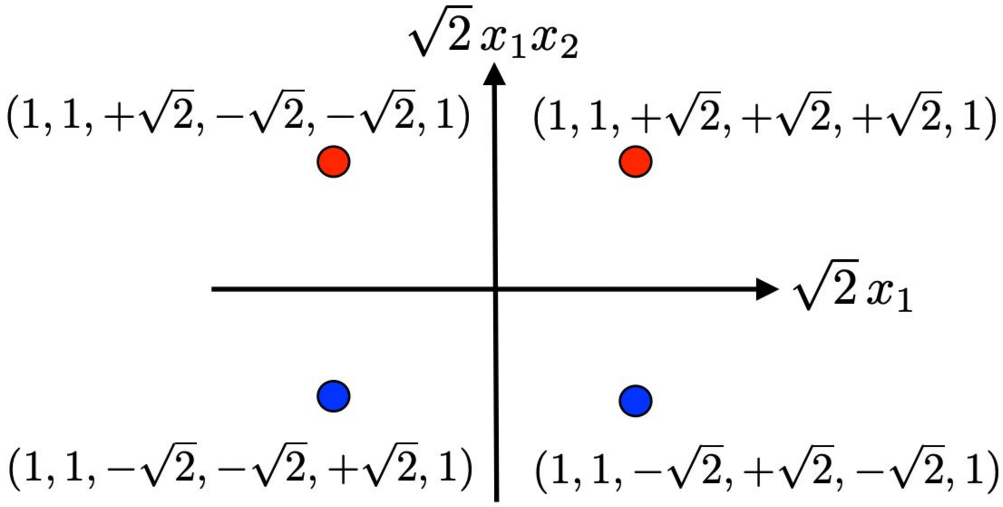
- **Example:** for $N=2$ and $d=2$,

$$K(x,y) = (x_1 y_1 + x_2 y_2 + c)^2$$

$$= \begin{bmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}\, x_1 x_2 \\ \sqrt{2c}\, x_1 \\ \sqrt{2c}\, x_2 \\ c \end{bmatrix} \cdot \begin{bmatrix} y_1^2 \\ y_2^2 \\ \sqrt{2}\, y_1 y_2 \\ \sqrt{2c}\, y_1 \\ \sqrt{2c}\, y_2 \\ c \end{bmatrix}.$$

Kernels: XOR Example





Linearly non-separable

Linearly separable by $x_1x_2 = 0.$

Other Kernel Options

Gaussian kernels:

$$K(x, y) = \exp\left(-\frac{||x - y||^2}{2\sigma^2}\right), \quad \sigma \neq 0.$$

Also known as "Radial Basis Function Kernel"

Sigmoid Kernels:

$$K(x, y) = \tanh(a(x \cdot y) + b), \; a, b \geq 0.$$

Note: The RBF/Gaussian kernel as a projection into infinite dimensions, commonly used in kernel SVM.

$$K(x, x') = \exp\left(-(x - x')^2\right)$$

$$= \exp\left(-x^2\right) \exp\left(-x'^2\right) \underbrace{\sum_{k=0}^{\infty} \frac{2^k (x)^k (x')^k}{k!}}_{\exp(2xx')}$$

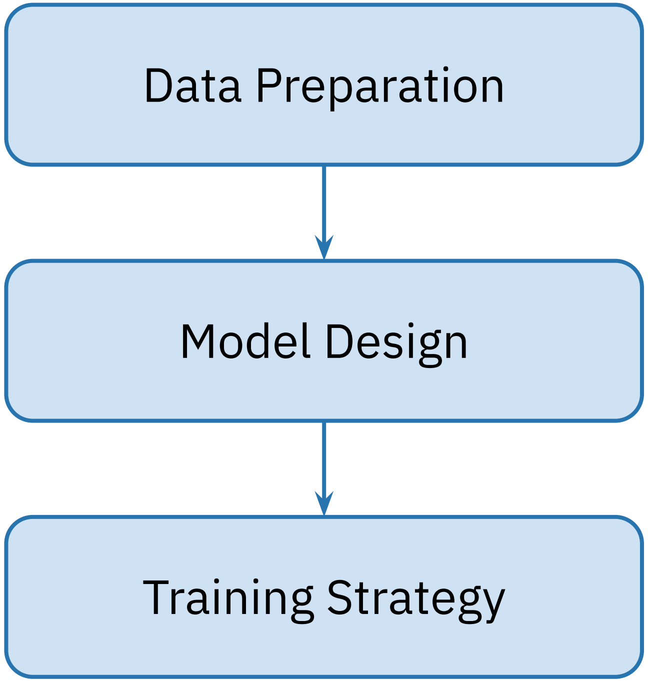
Taylor Expansion

Credit: http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf

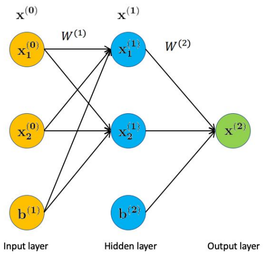
Programming Guide for PS3: PyTorch

- Important Concept Checklist
  - Tensors, Variable, Module
  - Autograd
  - Creating neural nets with provided modules: `torch.nn`
  - Training pipeline (loss, optimizer, etc): `torch.optim`
  - Util tools: Dataset
  - *(most important)* Search on official document or google
- A Not-so-short Tutorial: https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-winter2018/Pytorch_Tutorial.pdf → Details and demo code in another slides
- Youtube: https://www.youtube.com/playlist?list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m

PyTorch Project Pipeline



Use PyTorch to check your gradient calculation




```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(2, 2, bias=True)
        self.l2 = nn.Linear(2, 1, bias=True)

        self.l1.weight.data = torch.Tensor([[0.15, 0.2], [0.25, 0.3]])
        self.l2.weight.data = torch.Tensor([[0.4, 0.45]])
        self.l1.bias.data = torch.Tensor([0.35, 0.35])
        self.l2.bias.data = torch.Tensor([0.6])

    def forward(self, x0):
        z1 = self.l1(x0)
        x1 = torch.sigmoid(z1)
        z2 = self.l2(x1)
        x2 = torch.sigmoid(z2)
        print("z1:", z1)
        print("x1:", x1)
        print("z2:", z2)
        print("x2:", x2)
        return x2

    def loss(self, x2, y):
        l = nn.MSELoss()
        return 0.5 * l(x2, y)
```

```python
x = torch.Tensor([0.05, 0.1])
y = torch.Tensor([0.01])
net = Net()
y_hat = net(x)
loss = net.loss(y_hat, y)
print(loss)
loss.backward()
```

```
z1: tensor([0.3775, 0.3925], grad_fn=<AddBackward0>)
x1: tensor([0.5933, 0.5969], grad_fn=<SigmoidBackward>)
z2: tensor([1.1059], grad_fn=<AddBackward0>)
x2: tensor([0.7514], grad_fn=<SigmoidBackward>)
tensor(0.2748, grad_fn=<MulBackward0>)
```

```python
print("d[W1]", list(net.l1.parameters())[0].grad)
print("d[b1]", list(net.l1.parameters())[1].grad)
print("d[W2]", list(net.l2.parameters())[0].grad)
print("d[b2]", list(net.l2.parameters())[1].grad)
```

```
d[W1] tensor([[0.0007, 0.0013],
        [0.0007, 0.0015]])
d[b1] tensor([0.0134, 0.0150])
d[W2] tensor([[0.0822, 0.0827]])
d[b2] tensor([0.1385])
```

Colab Link: https://colab.research.google.com/drive/1FHo_mkFaTatKgpBw5VRRVUSBJzoMt_U8?usp=sharing

UCLA Samueli Computer Science

Thank you!

Q & A

Whiteboard

- Content