南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

STA-5007: Advanced Natural Language Processing

# Lecture 2: Basics of Deep Learning

陈冠华 CHEN Guanhua

Department of Statistics and Data Science

# Content

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

- Introduction
- History
- Model
- Optimization
- Training
- Coding

# Data Analyses

- To create a function to map an input X into an output Y, $Y = f(X)$
- Examples:

| Input $X$ | Output $Y$ | Task |
|---|---|---|
| Text | Text in Other Language | Translation |
| Text | Response | Dialog |
| Text | Label | Text Classification |
| Text | Linguistic Structure | Language Analysis |

- To create such a system, we can use
  - Manual creation of rules
  - Machine learning from paired data <X, Y>

# Machine Learning

- Statistical approach
  - Generalized linear model, linear regression, logistic regression
  - Gaussian mixture models
  - Support vector machine (SVM)
  - Decision trees, random forests
- Deep learning approach
  - Modeling with different deep neural networks

# Why didn’t They Work Before?

- Datasets too small
  - For machine translation, not really better until you have 1M+ parallel sentences (and really need a lot more)
- Optimization not well understood
  - Good initialization
  - Momentum (Adagrad/Adam) work best out-of-the-box
- Other innovations
  - Word embedding
  - Dropout, layer normalization, residual connection
  - Large-scale computing system

# Deep Learning



- Modeling with deep neural networks
- Optimized on big data
- Research Contributions

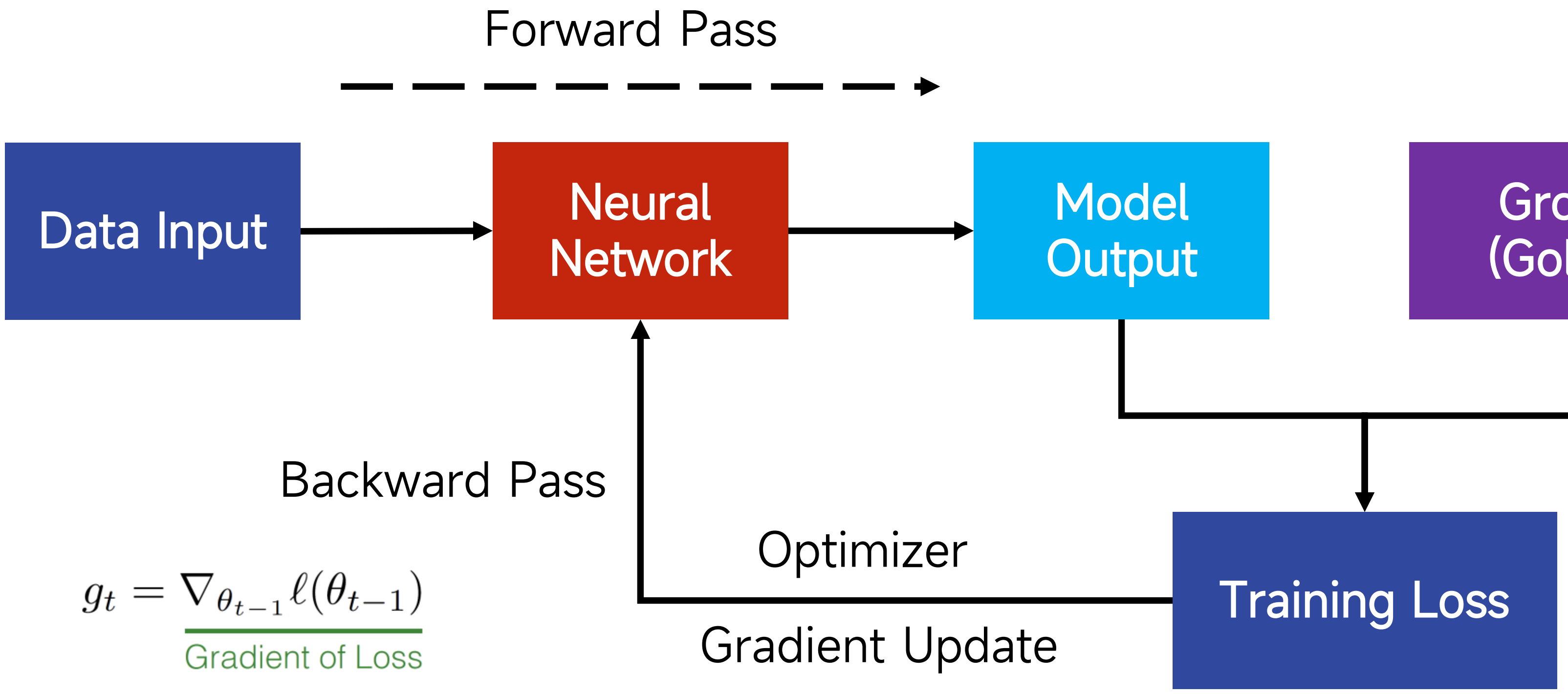
| | |
|---|---|
| Collect/create the dataset | Design a model |
| Design a task | Optimize a model |
| Design an evaluation metric | Analyze and understand the model and results |

# Deep Learning Algorithm Sketch

- Create a model and define a loss
- For each example
  - Forward process: calculate the result (prediction & loss) of that example
  - if training
    - Perform back propagation
    - Update parameters

# Deep Learning Algorithm Sketch






Python files

- Model.py
- Loss.py
- Data.py
- Trainer.py
- Main.py

$$g_t = \underbrace{\nabla_{\theta_{t-1}} \ell(\theta_{t-1})}_{\text{Gradient of Loss}}$$

$$\theta_t = \theta_{t-1} - \underbrace{\eta}_{\text{Learning Rate}} g_t$$

$$\underbrace{v_t}_{\text{Momentum}} = \underbrace{\gamma}_{\text{Momentum Conservation Parameter}} \underbrace{v_{t-1}}_{\text{Previous Momentum}} + \eta g_t$$

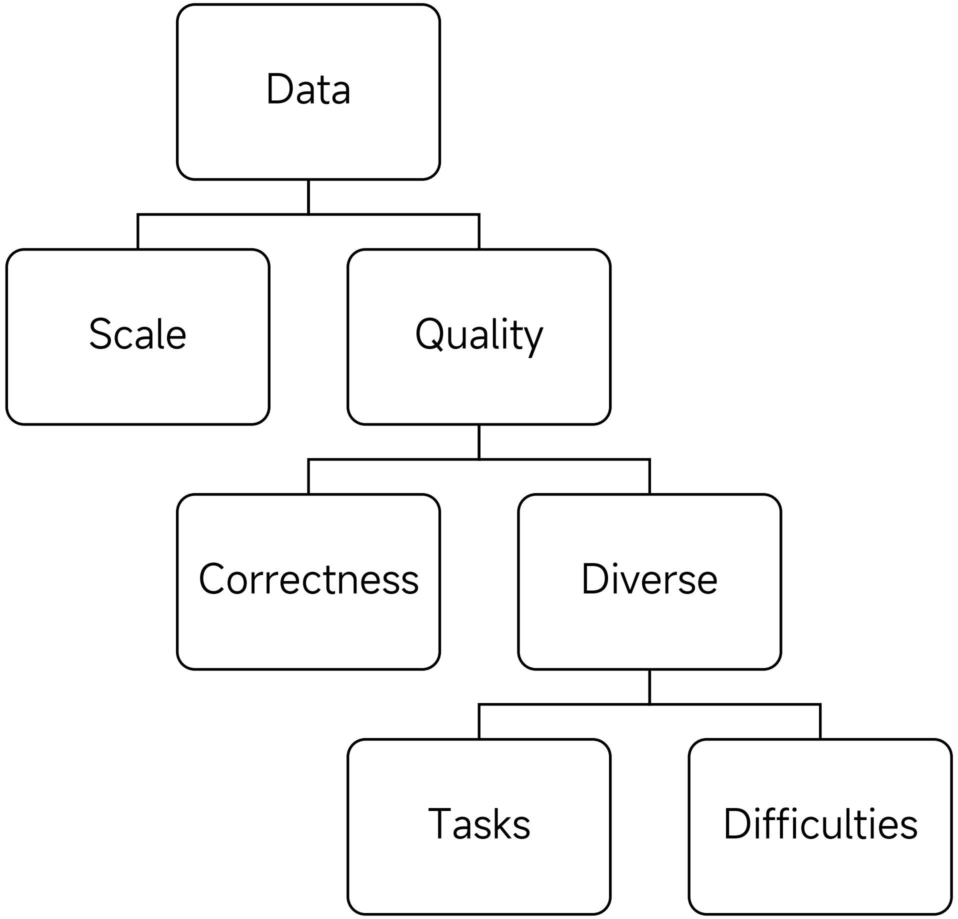
$$\theta_t = \theta_{t-1} - v_t$$

# Dataset Split

When creating a system, use three sets of data

- Training Set
    - Generally larger dataset,  used during system design, creation, and learning of parameters.
- Development/validation Set
    - Smaller dataset for testing different design decisions ("hyper-parameters").
- Test Set
    - Dataset reflecting the final test scenario, do not use for making design decisions.

# Data is Very Important

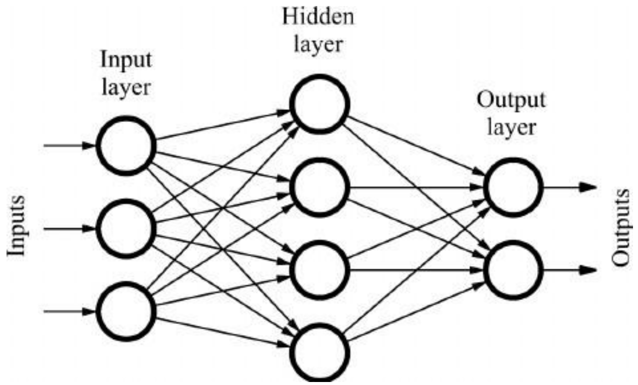


# Deep Learning Algorithm Sketch

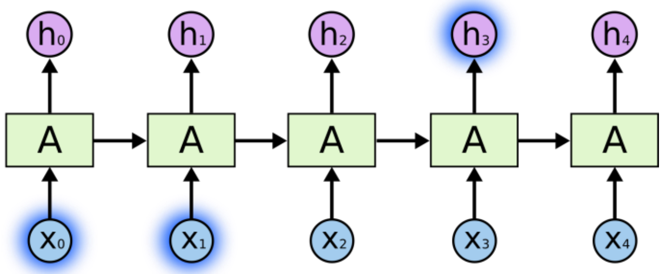
- Create a model and define a loss
- For each example
  - Forward process: calculate the result (prediction & loss) of that example
  - if training
    - Perform back propagation
    - Update parameters

# Different Model Structures

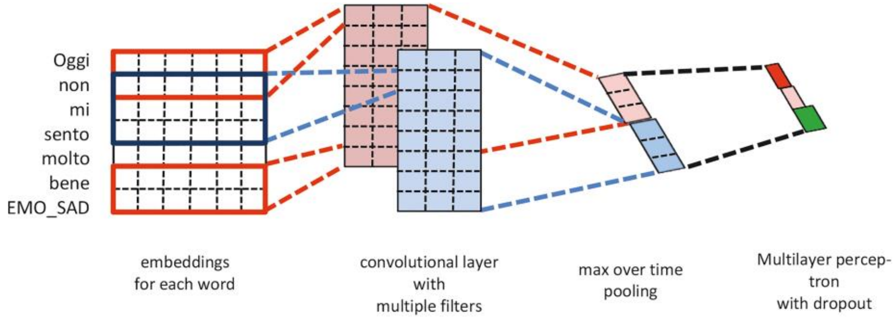
Feed-forward NNs



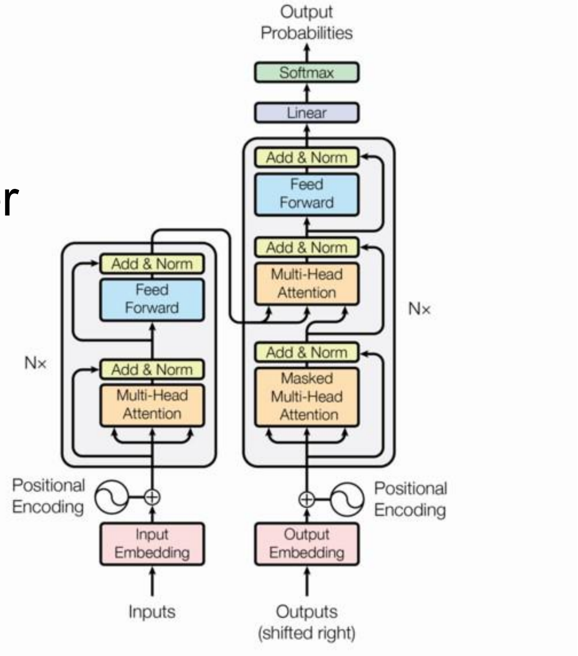

Recurrent NNs




Convolutional NNs

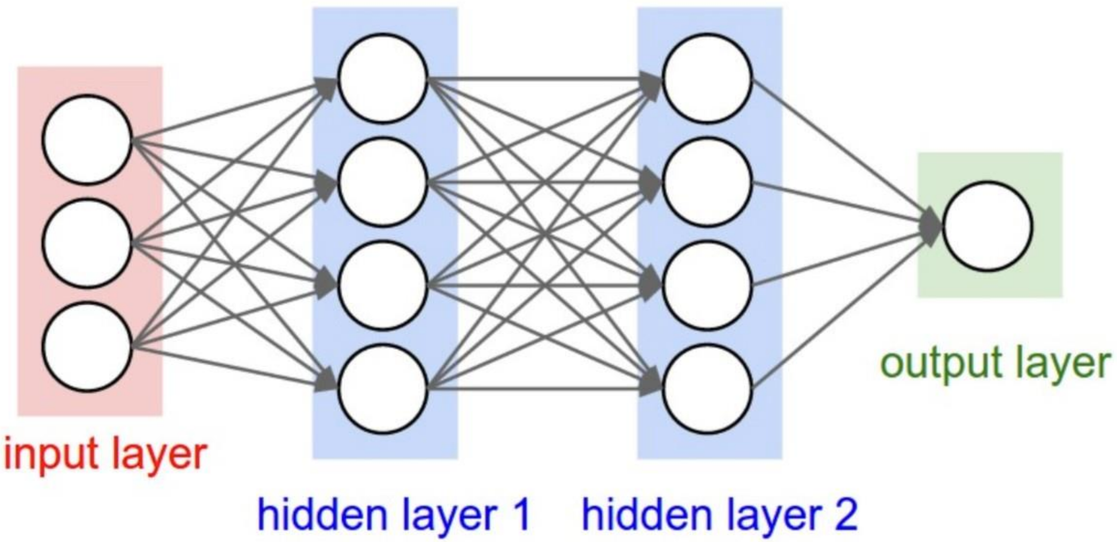



Transformer



# Feed-forward Neural Network

- The units are connected with no cycles
- The outputs from units in each layer are passed to units in the next higher layer
- No outputs are passed back to lower layers
- Fully-connected (FC) layers

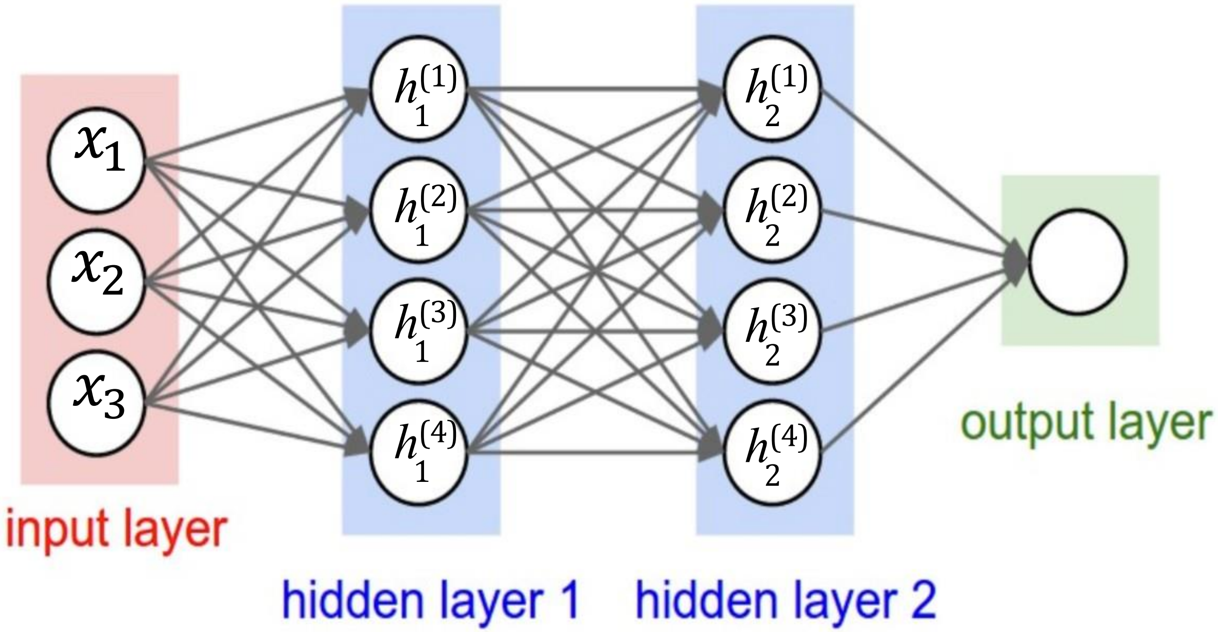



```python
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

# Feed-forward Neural Network




- Input layer: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer 1:

$$\mathbf{h}_1 = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \in \mathbb{R}^{d_1}$$

$$\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}, \mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$$

- Hidden layer 2:

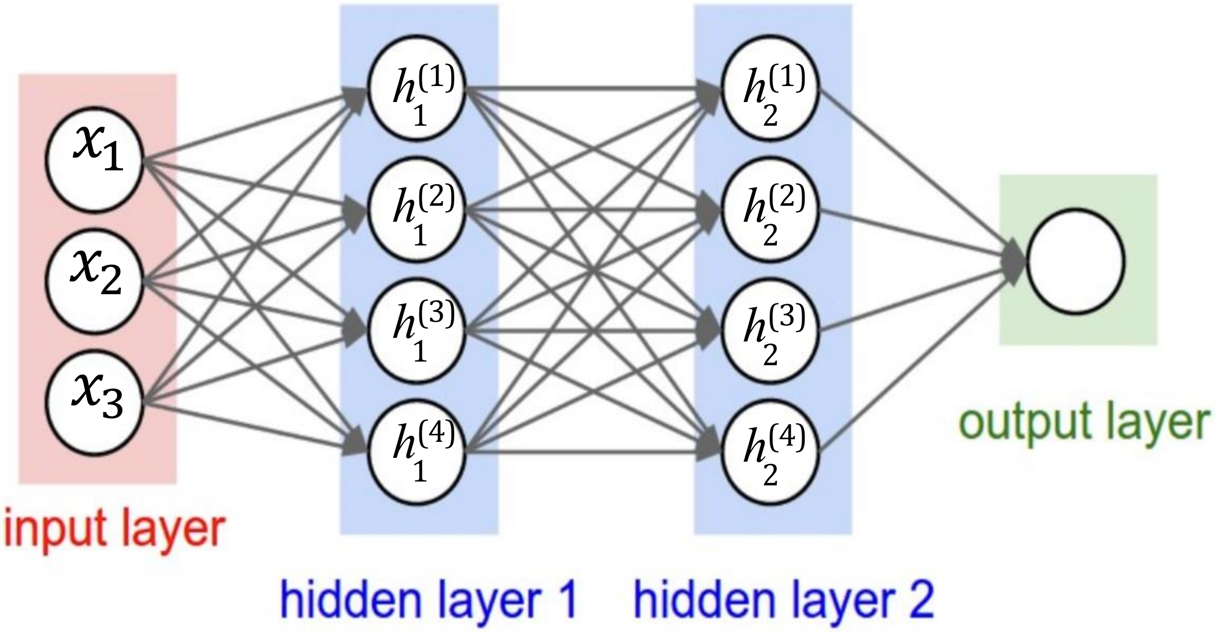
$$\mathbf{h}_2 = f(\mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)}) \in \mathbb{R}^{d_2}$$

$$\mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1}, \mathbf{b}^{(2)} \in \mathbb{R}^{d_2}$$

- Output layer:

$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}$$

# Feed-forward Neural Network




$$h_1^{(1)} = f(w_{1,1}^{(1)}x_1 + w_{1,2}^{(1)}x_2 + w_{1,3}^{(1)}x_3)$$

$$h_2^{(3)} = f(w_{3,1}^{(2)}h_1^{(1)} + w_{3,2}^{(2)}h_1^{(2)} + w_{3,3}^{(2)}h_1^{(3)} + w_{3,4}^{(2)}h_1^{(4)})$$

Non-linearity (activation function) $f$: $\tanh$ or $ReLU$

# Feedforward Neural Network for Classification

- Use `softmax` to get the probability distribution

$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{y}) \qquad \text{softmax}(\mathbf{y})_k = \frac{\exp(y_k)}{\sum_{j=1}^{C}\exp(y_j)} \qquad \mathbf{y} = [y_1, y_2, \ldots, y_C]$$

Training loss:

$$\min_{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(o)}} - \sum_{(\mathbf{x},y)\in D} \log \hat{\mathbf{y}}_y$$

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{W}^{(1)}\mathbf{x})$$

$$\mathbf{h}^{(2)} = \text{ReLU}(\mathbf{W}^{(2)}\mathbf{h}^{(1)})$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}^{(o)}\mathbf{h}^{(2)})$$

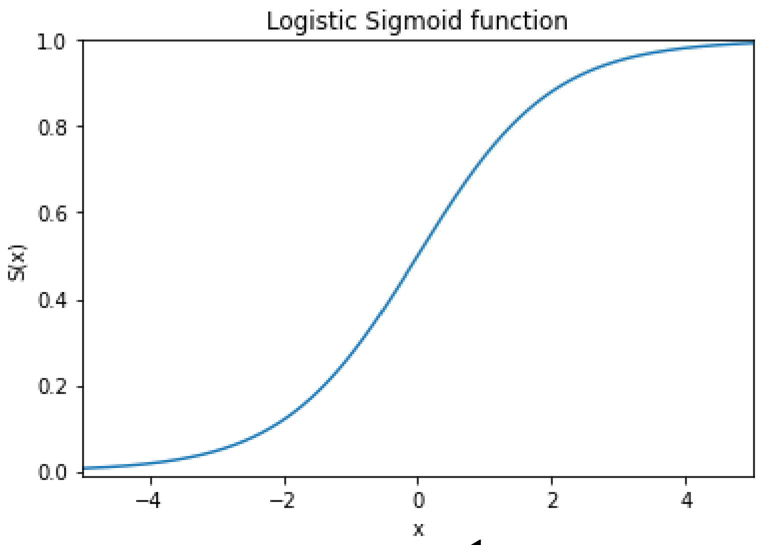
Neural networks are difficult to optimize.

SGD can only converge to local minimum.

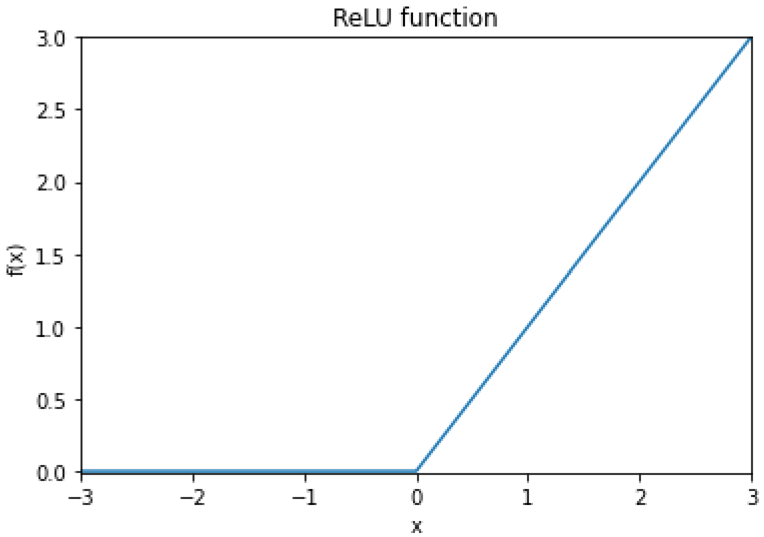
Initializations and optimizers matter a lot!

# Activation Functions

- Add non-linearities into neural networks
- Allowing the neural networks to learn powerful operations




$$f(x) = \frac{1}{1 + e^{-x}}$$




$$f(x) = \max(x, 0)$$

# Activation Functions



- GeLU (Gaussian Error Linear Unit)
  - Used in GPT-3, BERT, and many other models

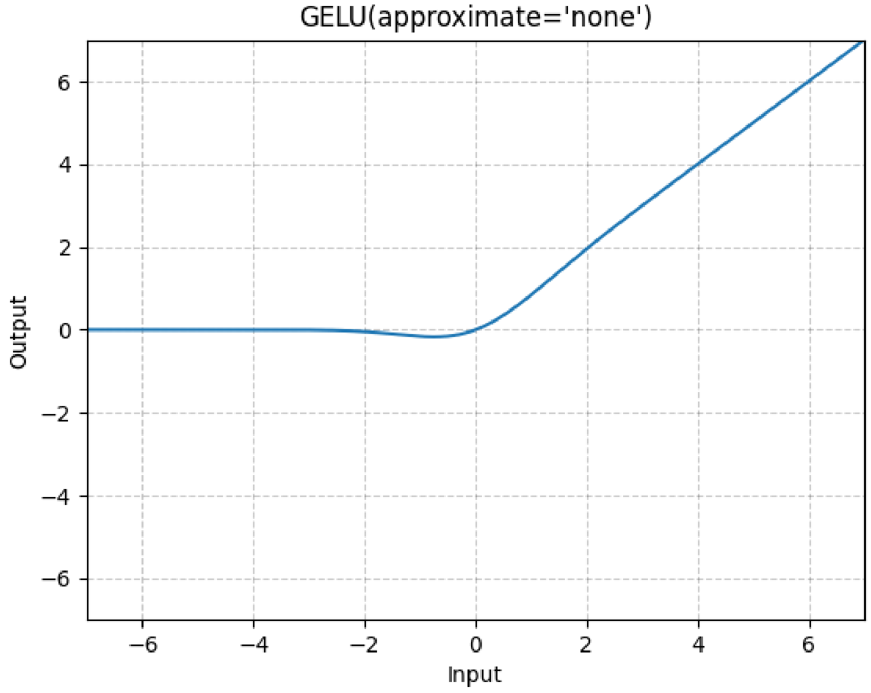
Applies the Gaussian Error Linear Units function:

$$\text{GELU}(x) = x * \Phi(x)$$

where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

When the approximate argument is 'tanh', Gelu is estimated with:

$$\text{GELU}(x) = 0.5 * x * (1 + \text{Tanh}(\sqrt{(2/\pi)} * (x + 0.044715 * x^3)))$$



# Loss Functions



- Given a labeled example $(x, \hat{y})$, we use a neural network to estimate the conditional probability and predict the label as

$$\hat{y} = \text{argmax}\, P_\theta(y|x)$$

- We compute how close our prediction w.r.t. the true label by a loss function
- Classification: Cross-Entropy

$$\mathcal{L}(x, y^*) = -\log P_\theta(y = y^*|x)$$

- Regression: L1 loss, L2 loss (a.k.a. Mean Square Error)

$$\mathcal{L}(x, y^*) = \|f(x) - y^*\|_1$$
$$\mathcal{L}(x, y^*) = \|f(x) - y^*\|_2$$

# Deep Learning Algorithm Sketch

南方科技大學
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

- Create a model and define a loss (i.e., construct a computation graph)
- For each example
  - Forward process: calculate the result (prediction & loss) of that example
  - if training
    - Perform back propagation
    - Update parameters

# Backpropagation

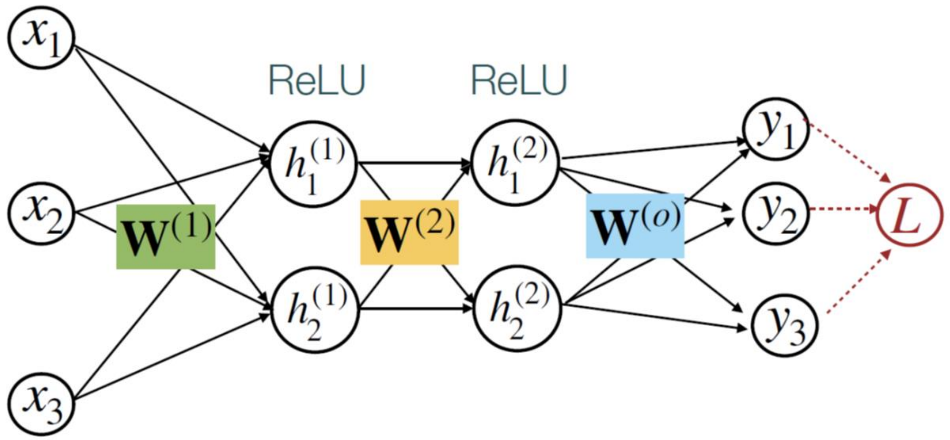
**Forward propagation:** from input to output layer

**Forward step 1:** Compute $h_1^{(1)}, h_2^{(1)}$

**Forward step 2:** Compute $h_1^{(2)}, h_2^{(2)}$

**Forward step 3:** Compute $y_1, y_2, y_3$ and $[\hat{y}_1, \hat{y}_2, \hat{y}_3] = \text{softmax}[y_1, y_2, y_3]$

**Given:** $x_1, x_2, x_3$ and the class label $y$ (a single training example)




**Forward step 4:** Compute loss $L = -\log \hat{\mathbf{y}}_y$

**Back propagation:** from output to input layer

**Goal:** $\dfrac{\partial L}{\partial W^{(1)}}, \dfrac{\partial L}{\partial W^{(2)}}, \dfrac{\partial L}{\partial W^{(o)}}$

**Back step 4:** Compute $\dfrac{\partial L}{\partial W^{(1)}}$

**Back step 3:** Compute $\dfrac{\partial L}{\partial h_1^{(1)}}, \dfrac{\partial L}{\partial h_2^{(1)}}, \dfrac{\partial L}{\partial W^{(2)}}$

**Back step 2:** Compute $\dfrac{\partial L}{\partial h_1^{(2)}}, \dfrac{\partial L}{\partial h_2^{(2)}}, \dfrac{\partial L}{\partial W^{(o)}}$

**Back step 1:** Compute $\dfrac{\partial L}{\partial y_1}, \dfrac{\partial L}{\partial y_2}, \dfrac{\partial L}{\partial y_3}$

# Back-propagation in PyTorch



```python
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```python
import torch.optim as optim

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```python
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

PyTorch did back-propagation for you in this one line of code!

[A toy pytorch example to train an NN model]

# Deep Learning Algorithm Sketch

- Create a model and define a loss (i.e., construct a computation graph)
- For each example
  - Forward process: calculate the result (prediction & loss) of that example
  - if training
    - Perform back propagation
    - Update parameters

# Optimizer Update

- Most deep learning toolkits implement the parameter updates by calling `optimizer.step()` function

```
[59] # Before gradient update
     for name, param in model.named_parameters():
         print(name, param)

     A Parameter containing:
     tensor([[-0.2267,  0.6521, -0.8193,  0.7723, -0.6456],
             [ 1.2410, -2.4380, -0.5612, -0.1144, -0.2687],
             [ 0.4792,  0.4543, -1.3530,  1.2934, -0.9943],
             [ 0.7565,  0.9449,  0.2796,  0.4703,  0.2926],
             [ 0.4143,  0.5891,  0.4370,  0.6060,  0.0161]]
     b Parameter containing:
     tensor([ 0.3592,  0.3455, -0.2517, -0.5678, -0.6016],
     c Parameter containing:
     tensor([-1.5490], requires_grad=True)
```

Before optimizer update

```
[61] # Gradient update:
     optimizer.step()
     # After gradient update
     for name, param in model.named_parameters():
         print(name, param)

     A Parameter containing:
     tensor([[-0.2167,  0.6621, -0.8093,  0.7623, -0.6556],
             [ 1.2510, -2.4280, -0.5512, -0.1244, -0.2787],
             [ 0.4892,  0.4643, -1.3430,  1.2834, -1.0043],
             [ 0.7465,  0.9349,  0.2696,  0.4803,  0.3026],
             [ 0.4043,  0.5791,  0.4270,  0.6160,  0.0261]]
     b Parameter containing:
     tensor([ 0.3492,  0.3355, -0.2617, -0.5578, -0.5916],
     c Parameter containing:
     tensor([-1.5390], requires_grad=True)
```

After optimizer update

Can be updated with standard SGD or Adam optimizer

# Standard SGD

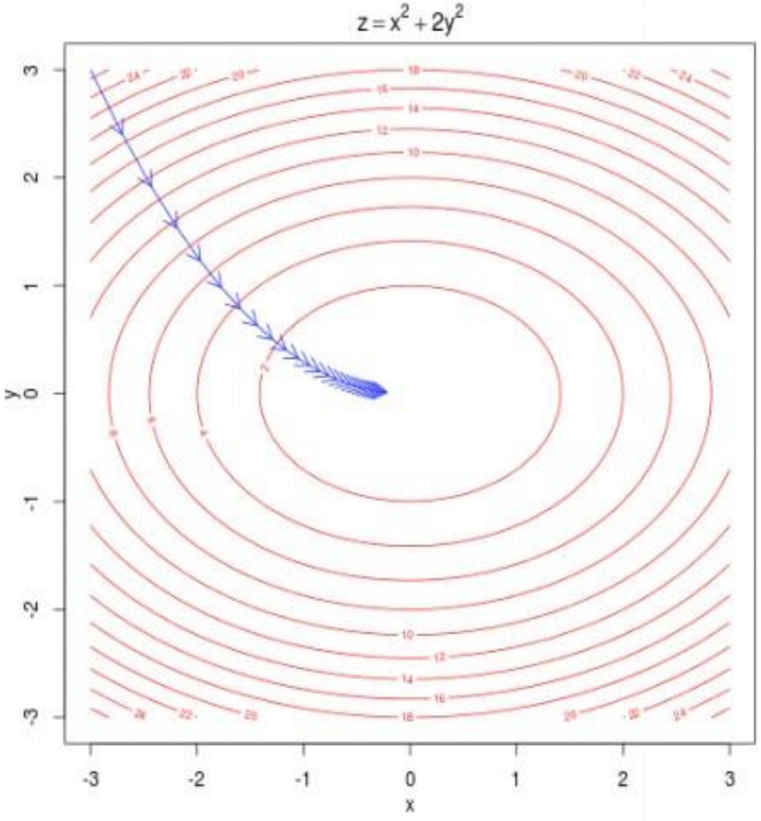


- Standard stochastic gradient descent

$$g_t = \underbrace{\nabla_{\theta_{t-1}} \ell(\theta_{t-1})}_{\text{Gradient of Loss}}$$

$$\theta_t = \theta_{t-1} - \underbrace{\eta}_{\text{Learning Rate}} g_t$$

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```



# Adam Optimizer



- Most standard optimization option in NLP and beyond
- Considers rolling average of gradient $g_t$, and momentum $m_t$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \longrightarrow \text{Momentum}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t \odot g_t \longrightarrow \text{Rolling Average of Gradient}$$

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)^t} \quad \hat{v}_t = \frac{v_t}{1-(\beta_2)^t} \longrightarrow \text{Correction of bias}$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \longrightarrow \text{Final update the parameter}$$

[Further reading: how to use the optimizer in Pytorch]

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005, betas=(0.99, 0.999))
```

# Adam Optimizer



- Gradient descent | Khan Academy
- Intuition of Adam Optimizer
- Blog: An updated overview of recent gradient descent algorithms
- (paper) Convex Optimization: Algorithms and Complexity
- Course: Optimization for Machine Learning

# Learning Rate

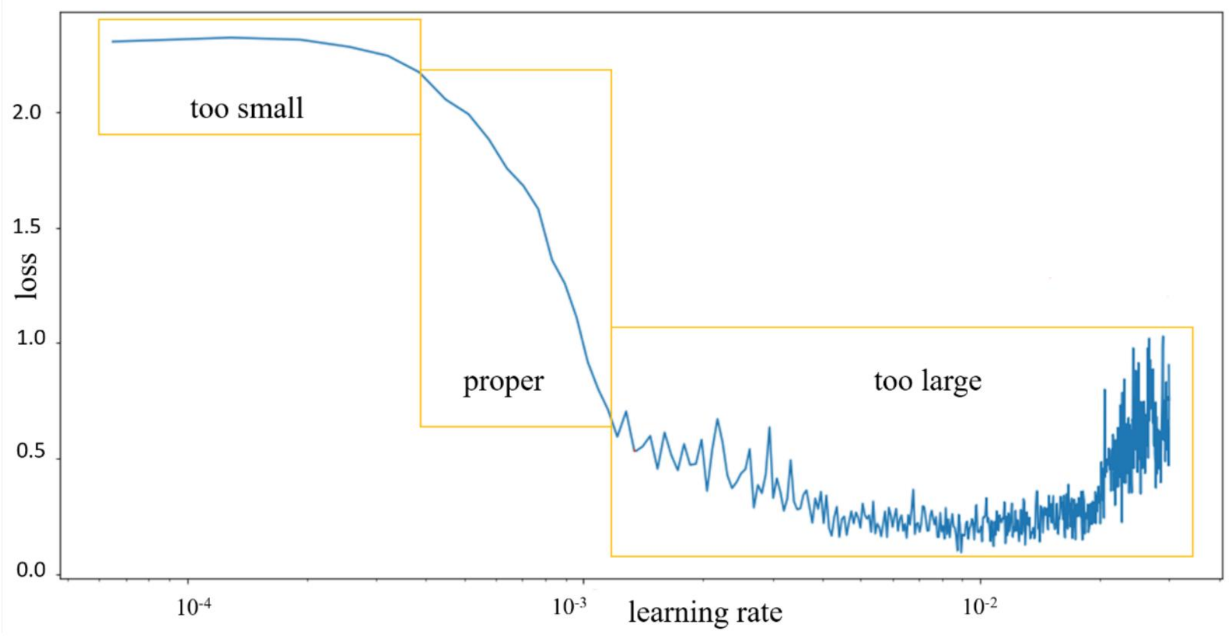
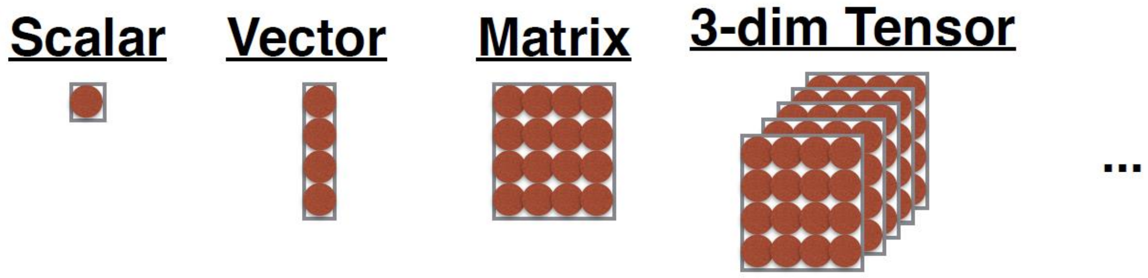



Figure 4: Effect of learning rate (adapted figure from: https://www.jeremyjordan.me/nn-learning-rate/)

learning rate schedule [another link], warmup

# Tensors

- An n-dimensional array




$x \in \mathbb{R}$ $\quad \mathbf{x} \in \mathbb{R}^4$ $\quad \mathbf{X} \in \mathbb{R}^{4\times4}$ $\quad \mathbf{X} \in \mathbb{R}^{4\times4\times5}$

- Widely used in neural networks
- Parameters in NNs consist of different shape of tensors, which store both their values and gradients (e.g., `x, x.grad`)

# Tensor Operations

```
import numpy as np
x = torch.Tensor([[2, 3], [1, 2]])
x = torch.Tensor(np.array([[-1, 1], [2, 4]]))
x = torch.zeros([2, 3], dtype=torch.int32)
```

create tensors from list, numpy.array

```
import torch
import torch.nn as nn

x = torch.randn((4, 2))
W = torch.randn((3, 4))
print(x)
print(W)
```

```
tensor([[-1.5372,  0.0845],
        [ 0.5752,  0.7634],
        [ 0.4265, -0.1287],
        [-1.8629, -0.8520]])
tensor([[-1.1272, -1.1810,  0.0867,  0.0676],
        [-0.1070,  3.2586,  0.7446, -1.2094],
        [-1.7670,  0.2900, -1.0881, -1.5555]])
```

```
[10] torch.matmul(W, x)  # results in a [3,2] matrix
```

```
tensor([[ 0.9646, -1.0656],
        [ 4.6092,  3.4132],
        [ 5.3167,  1.5373]])
```

matrix multiply

```
x = torch.randn((4,2))
z = torch.randn((4,2))
print(x)
print(z)
```

```
tensor([[ 0.0762,  1.5145],
        [-0.4747, -0.9141],
        [ 0.7106,  0.4888],
        [ 0.6959, -0.5305]])
tensor([[-0.1766,  0.6187],
        [ 0.9254, -0.5931],
        [-0.9162,  0.3209],
        [ 0.0216, -0.7116]])
```

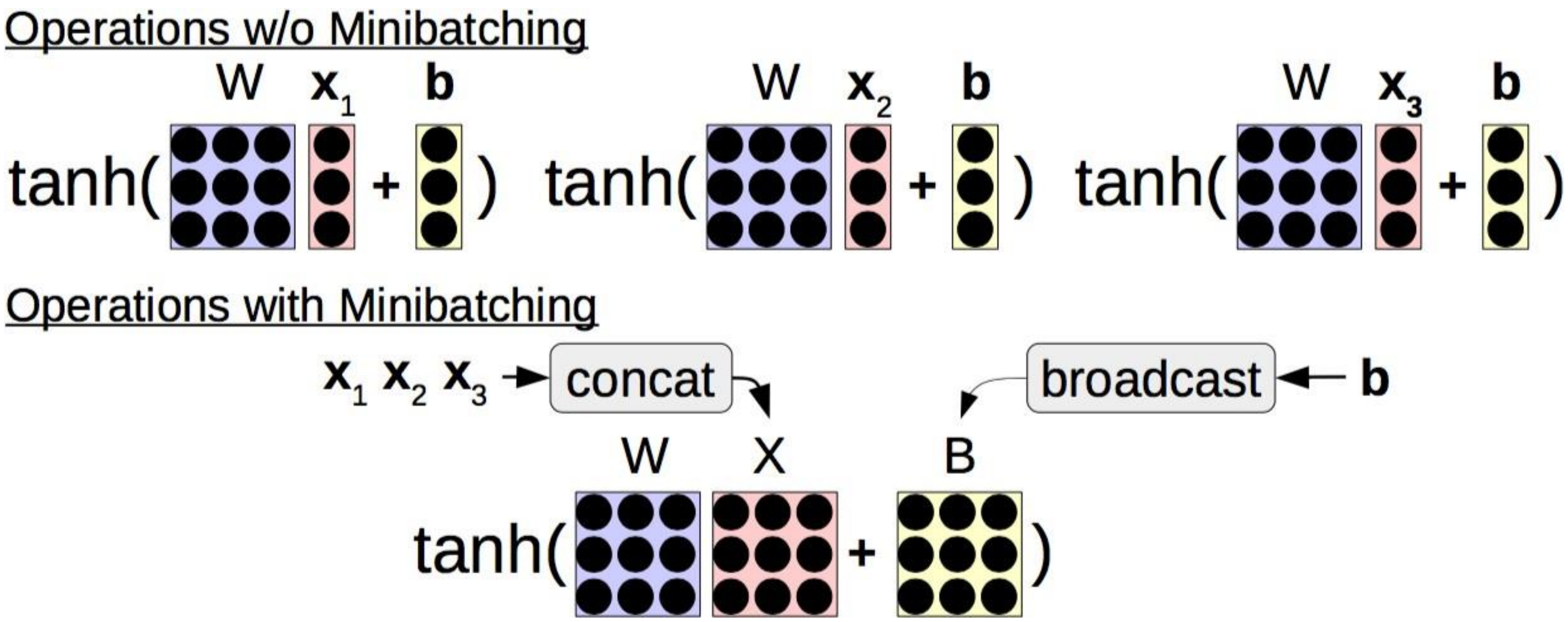
```
[13] x * z    # results in a [4,2] matrix
```

```
tensor([[-0.0135,  0.9371],
        [-0.4392,  0.5421],
        [-0.6510,  0.1569],
        [ 0.0151,  0.3775]])
```

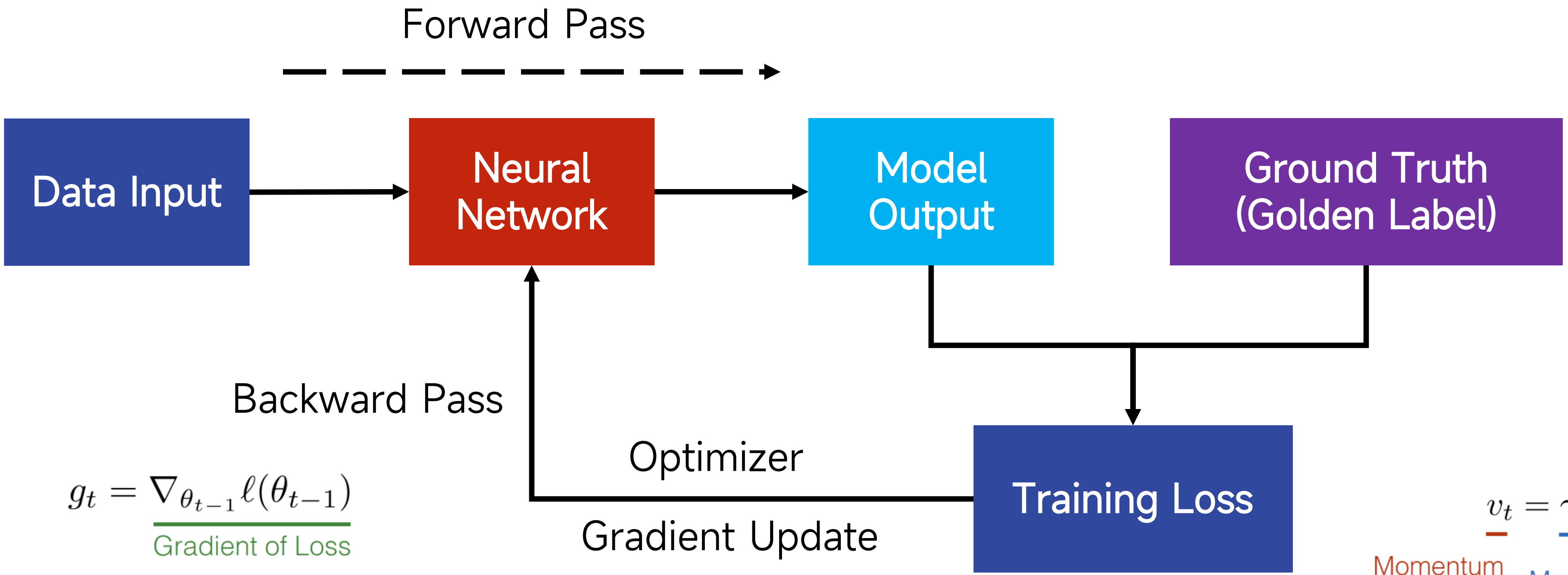
Element-wise matrix multiply

# Efficiency Tricks: Mini-batching

- On modern hardware 10 operations of size 1 is much slower than 1 operation of size 10
- Mini-batching combines together smaller operations into one big one
- About padding



# Deep Learning Algorithm Sketch




$$\underbrace{g_t = \nabla_{\theta_{t-1}} \ell(\theta_{t-1})}_{\text{Gradient of Loss}}$$

$$\theta_t = \theta_{t-1} - \underbrace{\eta}_{\text{Learning Rate}} g_t$$

Python files

- Model.py
- Loss.py
- Data.py
- Trainer.py
- Main.py

$$\underbrace{v_t}_{\text{Momentum}} = \underbrace{\gamma}_{\text{Momentum Conservation Parameter}} \underbrace{v_{t-1}}_{\text{Previous Momentum}} + \eta g_t$$

$$\theta_t = \theta_{t-1} - v_t$$

# Different Learnings

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

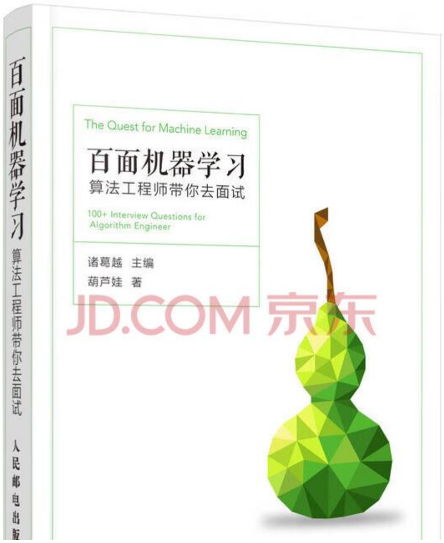
- Supervised/unsupervised learning
- Self-supervised learning
- Transfer learning
- Few-shot/zero-shot learning

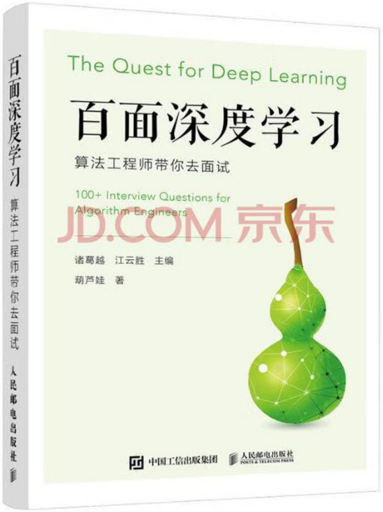
Stanford STATS214 / CS229M: Machine Learning Theory

# Further Reading



- (book) Information Theory From Coding to Learning
- Neural Networks: Zero to Hero (Andrej Karpathy)
- Course: Introduction to Deep Learning
- Course: MIT Introduction to Deep Learning
- Github free resources for students



Thank you