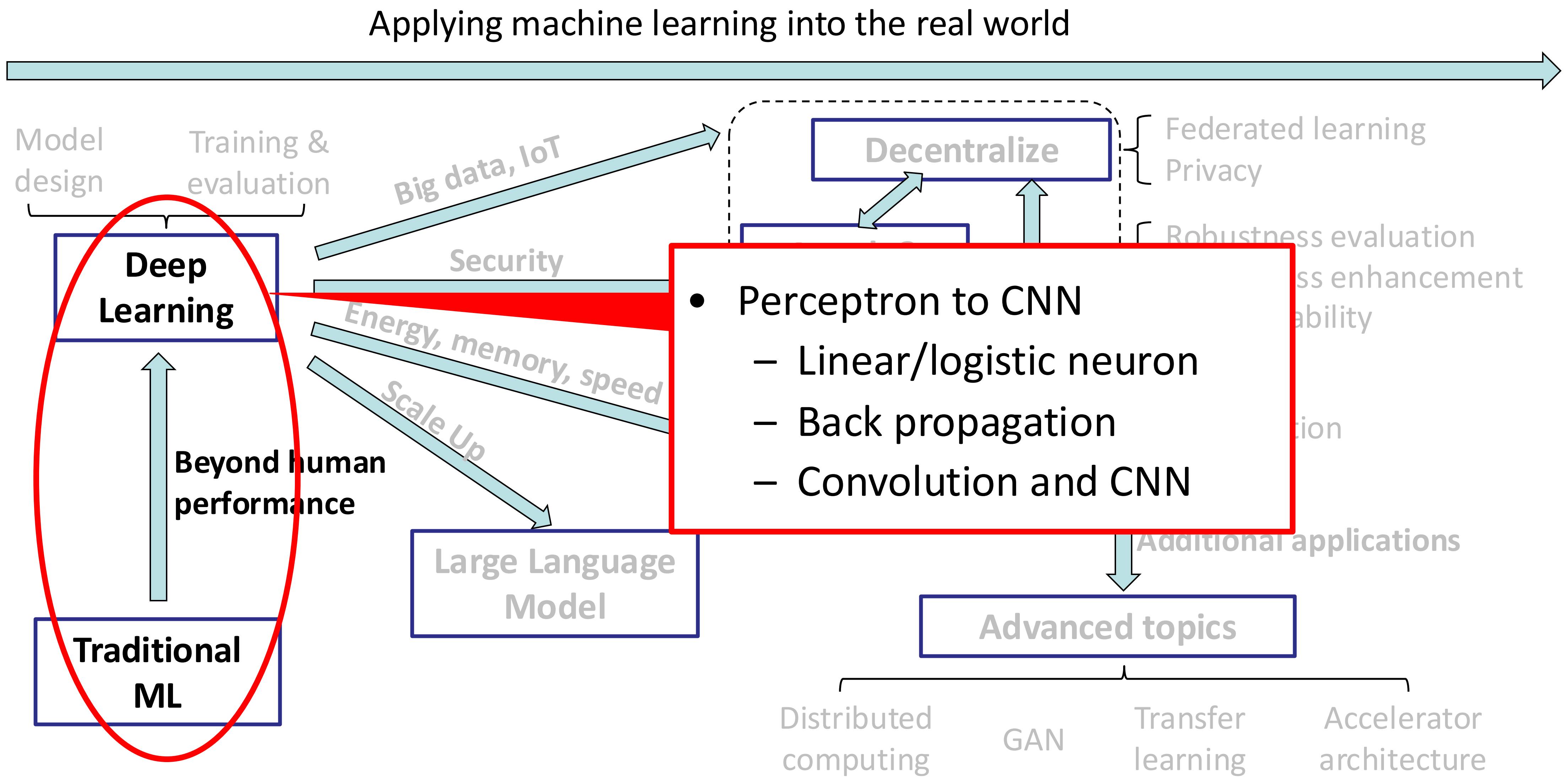
ECE 661 COMP ENG ML & DEEP NEURAL NETS

5. CNN TRAINING – BASIC

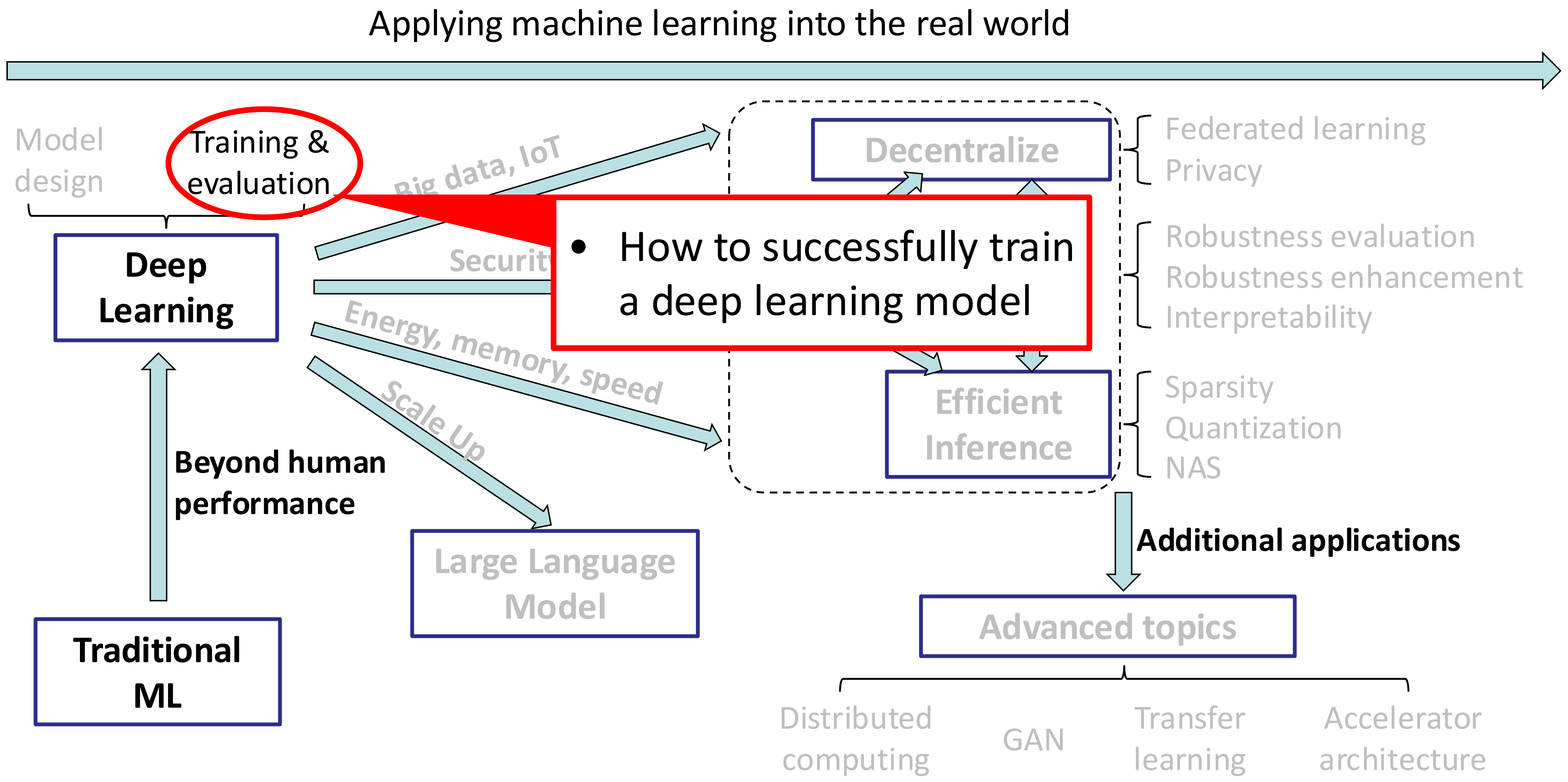
HAI "HELEN" LI, SPRING 2025

Previously



- Perceptron to CNN
  - Linear/logistic neuron
  - Back propagation
  - Convolution and CNN

Next two lectures



- How to successfully train a deep learning model

Babysitting the CNN training

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

  - Regularization
  - Data preprocessing
  - Hyperparameter tuning

Babysitting the CNN training

Lecture 5: Basic techniques

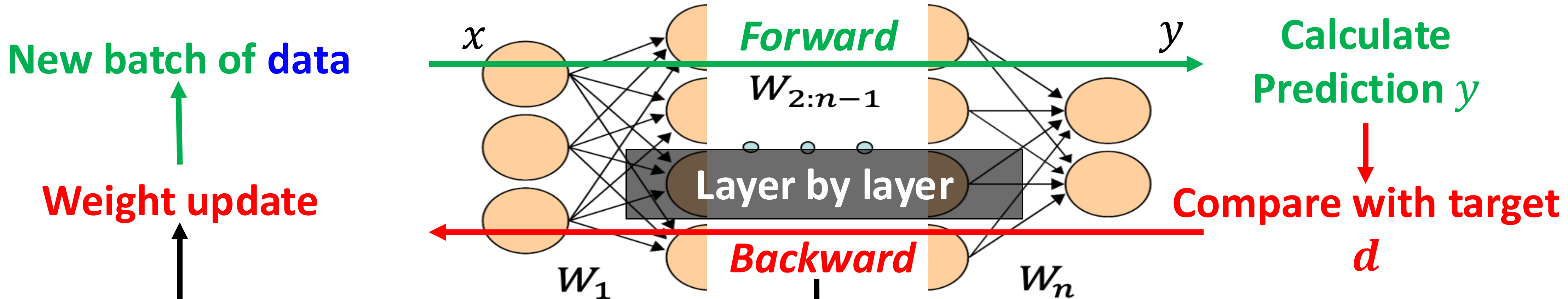
- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

Recall: Training overview

$f_i(W_i,\cdot)$: i_{th} layer (linear/conv) with **weight** W_i followed by **Activation function**

$$\boxed{y = F(x) := f_n(W_n, f_{n-1}(W_{n-1}, f_{...} f_1(W_1, x)))}$$



$$\boxed{Loss = L(y, d)}$$

Loss Function

Optimizer

Gradient of i_{th} layer's weights

$$\boxed{\frac{\partial Loss}{\partial W_i} = \frac{\partial L}{\partial f_n} \cdot \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdots \frac{\partial f_i}{\partial W_i}}$$

Babysitting the CNN training

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

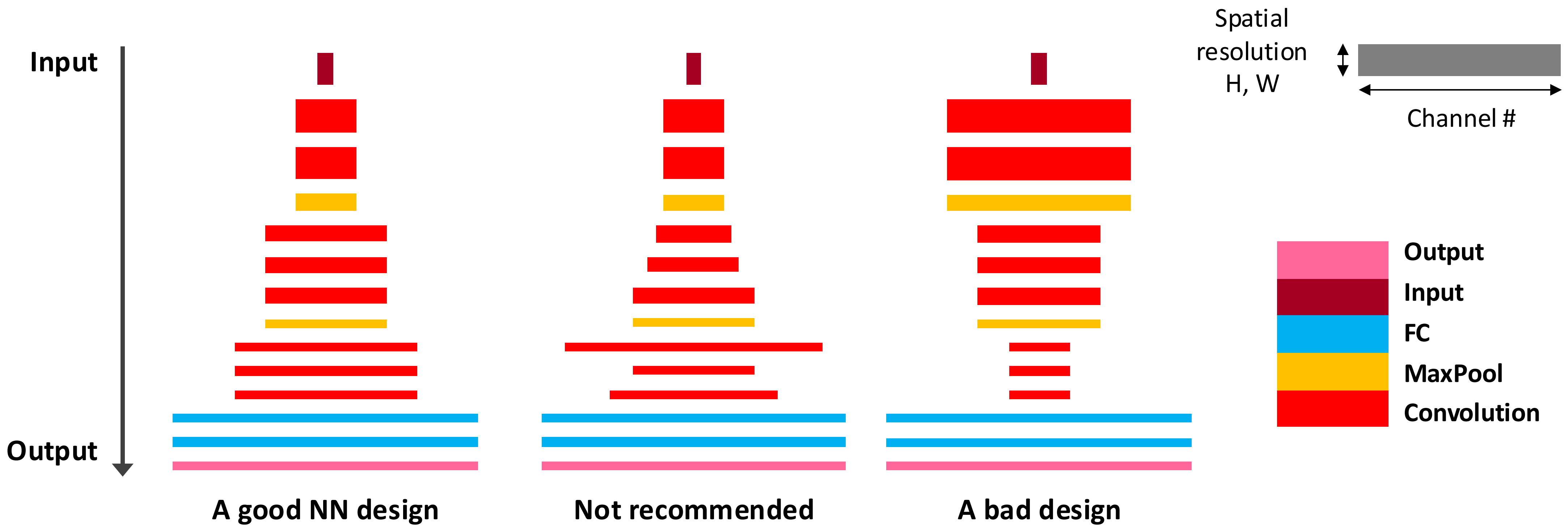
Neural network design: A bird's-eye view

What does a good practice of CNN design look like?

✓ "Thinner at the top, wider at the bottom"

✓ Deeper feature maps should have a lower spatial resolution, but more channels.

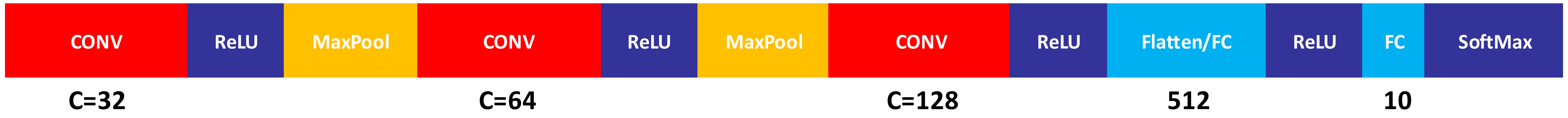
Two considerations: 1) information preservation 2) feasible computation



Neural network design: # filters

How to choose the number of filters in a CNN?

- By convention, the filter numbers C is in the form of 2^n.
- Number of filters should increase (most likely by a factor of **2**) after each down-sampling module (e.g., MaxPool, strided convolution).
- Here is an example configuration which is good enough for training on MNIST digit classification.

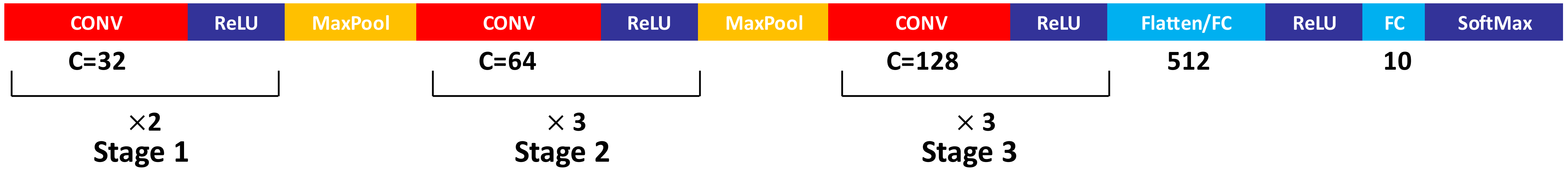


- The channel number increases with 32 → 64 → 128 while spatial feature maps get smaller.

Neural network design: # layers

How to decide the number of convolutional layers?

- Usually stack **2-4 convolutional layers** before applying a down-sampling module.
  - Do not stack thousands of convolution layers, as very deep convolutional neural networks are extremely difficult to train.
- A combination of N convolutional layers and a pooling layer (CONV*N-POOL) is called as a **stage**.
- Typically, a CNN model for MNIST/CIFAR-10 has **3** stages.



- The channel number increases with 32 → 64 → 128 while spatial feature maps get smaller.
- Each stage stacks multiple convolution layers.
- Typically 1~3 FC layers are enough.

Babysitting the CNN training

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

Activation functions

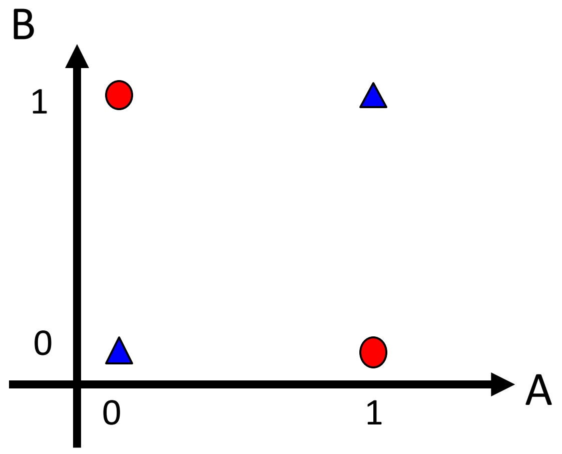
Why is activation function necessary?

- Activation functions enable NNs to learn complicated representations.
- Without any nonlinear activation functions, a multi-layer NN degrades to a single-layer NN models.
- A single-layer NN model cannot fit linearly non-separable cases.

Truth Table of XOR Gate

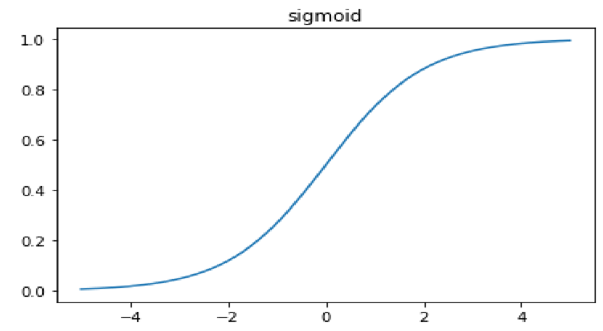
Input		Output
A	**B**	**Y**
0	0	0
0	1	1
1	0	1
1	1	0

Recall: Can we use a **linear** neural network to model the XOR Gate with binary inputs A and B?



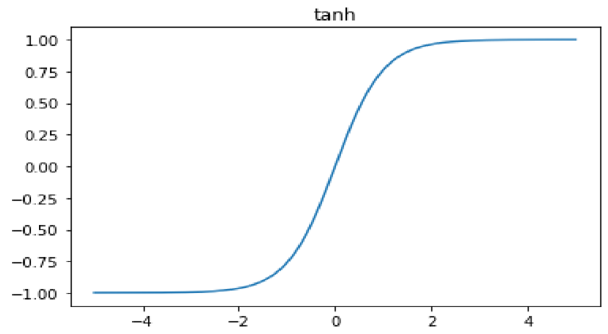
An overview of activation functions

There are many activation functions!



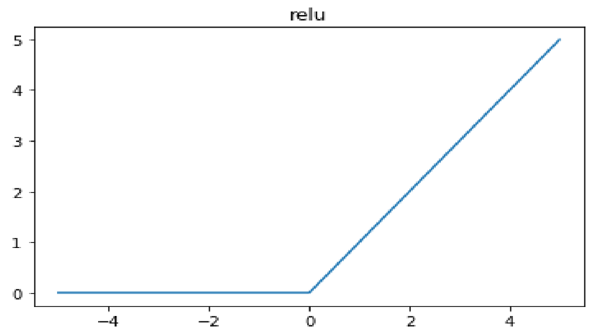
Sigmoid

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



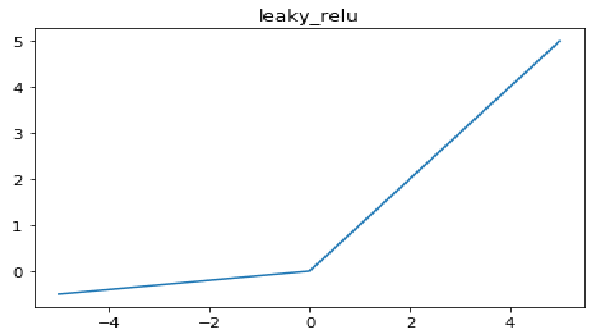
Tanh

$a(x) = \tanh(x)$



ReLU

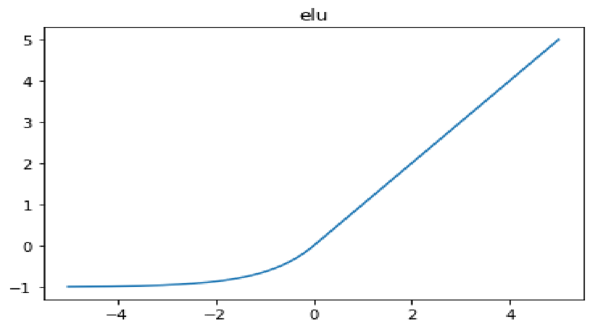
$a(x) = \max(0, x)$



Leaky ReLU

$a(x) = \max(\alpha x, x)$

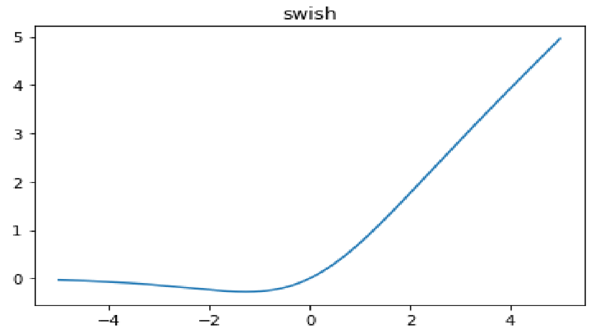
$\alpha \in (0,1)$



ELU

$$a(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

$\alpha \in (0,1)$



Swish

$a(x) = x\sigma(x)$

Sigmoid

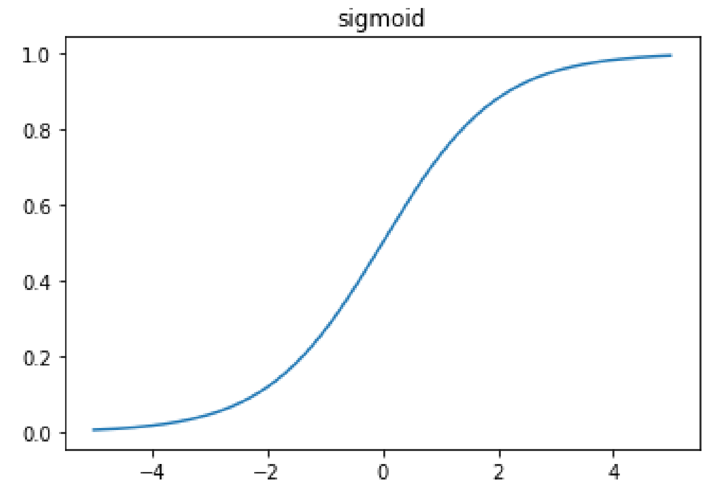
Sigmoid function normalizes inputs to $[\mathbf{0}, +\mathbf{1}]$.

Advantage:

- **Bounded output** makes training the model more stable and efficient.

Problems:

- S-shape curve suffers from gradient vanishing. This makes the neural networks harder to learn in some layers.
  - Gradients are close to zero when the activation value gets saturated
- Non-zero-centered output.



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

$$\frac{dy}{dx} = y(1 - y)$$

- Typically, sigmoid activation is only used after the last layer to generate a valid probability distribution for binary classification problems.

Hyperbolic tangent (tanh)

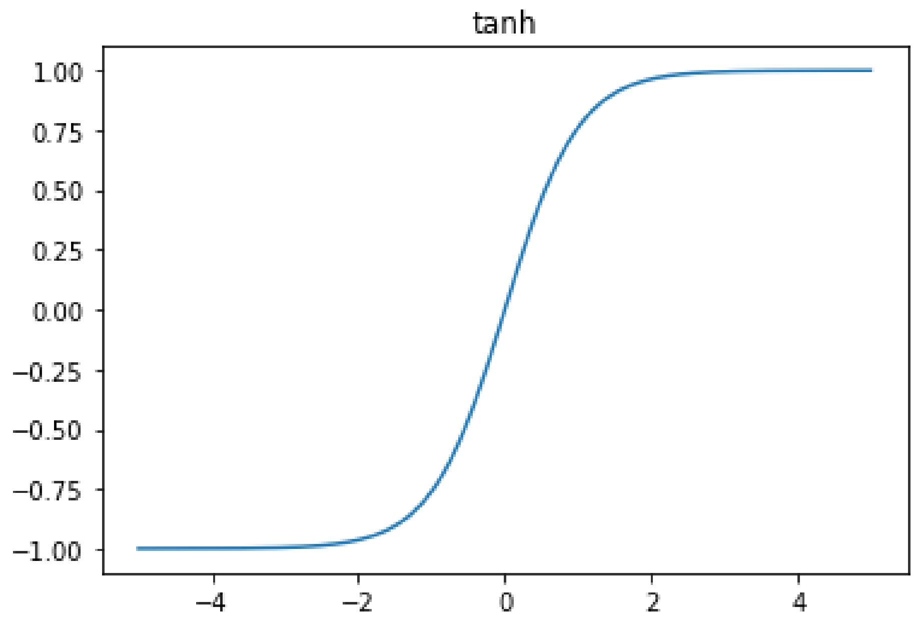
Hyperbolic tangent (tanh) normalizes inputs to $[\mathbf{-1}, \mathbf{+1}]$.

Advantage:

- **Bounded** & **zero-centered output.** This makes training the model more stable and efficient.

Problem:

- Gradient vanishing in saturation region. This makes DNNs with many layers harder to train.
  - Gradients are close to zero when the neuron's input gets saturated.



$$y = tanh(x)$$

$$\frac{dy}{dx} = 1 - \tanh^2(x)$$

REctified Linear Unit (ReLU)

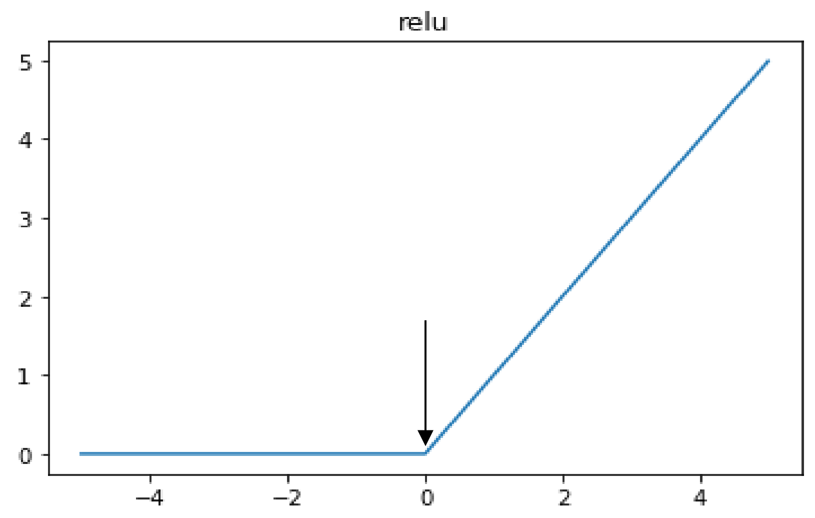
REctified Linear Unit (ReLU) transforms negative input neurons to 0 and keeps positive inputs.

Advantages:

- Cheap to compute
- Converge faster

Problems:

- Dead neurons in negative region. Weights related to inactive neurons will never be updated.
  - The gradient is non-zero only if the input neuron is positive (i.e., activated).
- Non-zero-centered output. This may trigger problem in some early neural network designs.



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

$$\frac{dy}{dx} = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

- **ReLU is the most popular choice.**
- ReLU activation is not commonly used for the output, which usually should not be bounded by (0, ∞)

Leaky ReLU

Leaky ReLU uses a slope for negative inputs. Thus, negative neurons can update related weights. This solves the ‘dead neuron’ problem.

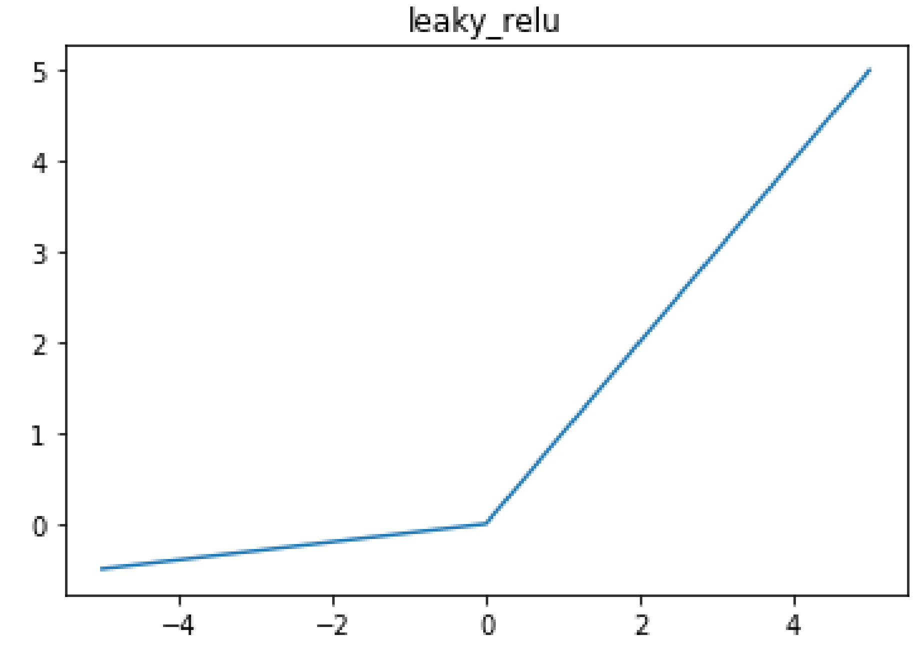
- Stable gradients are provided when the input neuron is negative.

Advantages:

- Computationally efficient
- No dead neurons

Problems:

- Non-zero-centered output.
- Inconsistent slope makes the training process of some neural network architectures unstable. Under most cases, using leaky ReLU may take more time to reach convergence.



$$y = \max(\alpha x, x)\ ,\ \alpha \in (0,1)$$

$$\frac{dy}{dx} = \begin{cases} \alpha & x \le 0 \\ 1 & x > 0 \end{cases}$$

Exponential linear unit (ELU)

Exponential learning unit (ELU) bounds negative inputs with exponential function in the negative region.

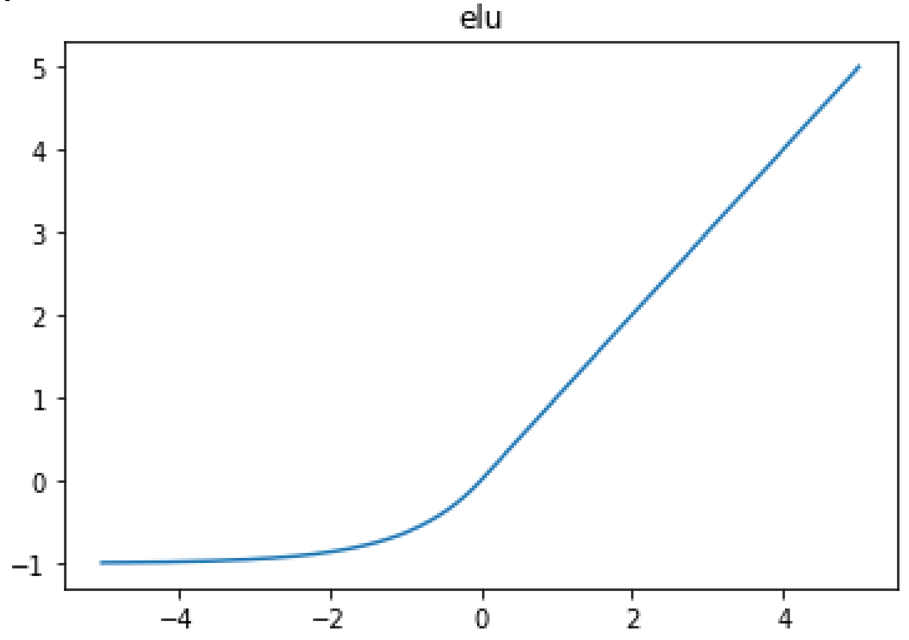
- Exponential function adds to the performance of DNNs.
- ELU is more robust to noise in the negative saturation part.

Advantages:

- Closer to zero-mean outputs. Thus, ELU has similar functionality as **Batch Normalization** (We will cover it later).
- Less dependent of weight initialization
- Better performance

Problem:

- Computationally inefficient. **ELU** induces additional cost to the model.



$$\alpha(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

$\alpha \in (0,1)$, usually we use α=1

Swish

Swish is an activation function discovered by AutoML algorithms.

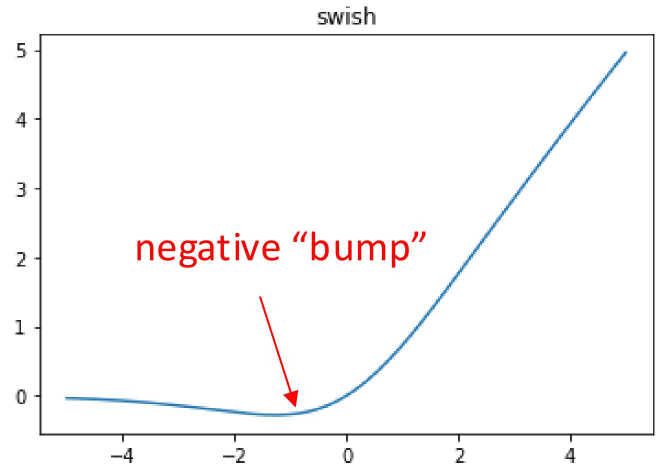
- The non-monotonic "bump" of Swish increases the complexity of neural networks while processing negative inputs.

Advantages:

- Increase representation power for DNNs, especially small models.
- Consistent performance gain compared to other activation functions over different models.

Problem:

- Computationally inefficient. **Swish** induces additional cost to the model.



$$y = x\sigma(x)$$

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

Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. "Searching for activation functions." (2017)

Summary of activations

	Performance on CNN?	Efficiency?	Zero-mean?
Sigmoid	Poor	Good	No
Tanh	Poor	Poor	Yes
ReLU	Good	Good	No
Leaky ReLU	Good	Good	No
ELU	Good	Poor	Yes
Swish	Good	Poor	Yes

Performance: Swish gives the best performance on CNN.
Efficiency: **ReLU** is a good choice for efficient CNNs.
Zero-mean: Zero-mean activations such as **ELU** and **Swish** make CNN training easier.

Neural network design: activation function

Which activation function should we use?

- We recommend starting with ReLU activation function.
- Using activation functions other than ReLU makes it harder for debugging.
  - For example, even if a neural network is not correctly configured, ELU/Swish may still make the loss decreasing during the optimization process, which will lead to sub-optimal performance.
- If your DNN design works fine with ReLU, you may switch to other kinds of activations, which may bring a higher performance gain.
- ELU and Swish can improve performance, with additional cost.
- Sigmoid activation is commonly used at the output in **binary classification problems**.

Babysitting the CNN training

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

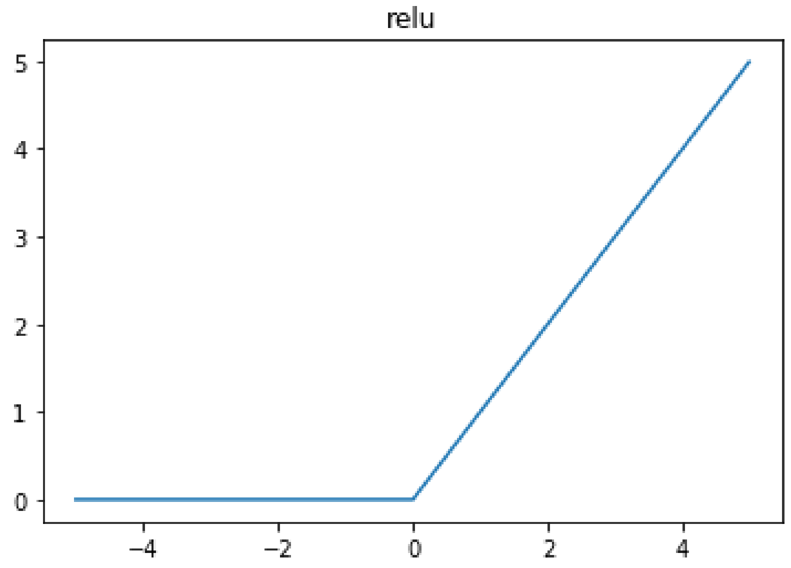
Lecture 6: Advanced techniques

Weight initialization

Why is weight initialization necessary?

Without weight initializations, DNN can be tricky to train.

Question: Suppose all the layers in a DNN use ReLU as the activation function. What will happen if all weight and bias parameters are set to 0 at the beginning of training?



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

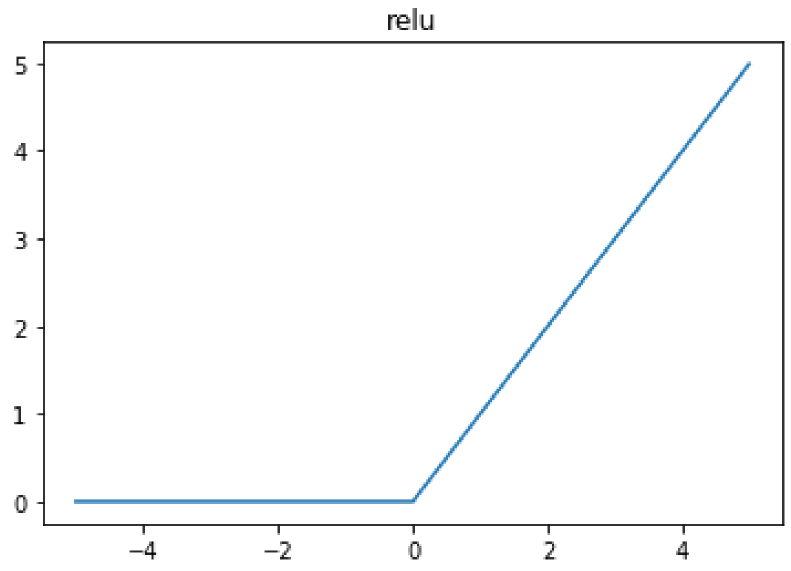
$$\frac{dy}{dx} = \begin{cases} 0 & x \le 0 \\ 1 & x > 0 \end{cases}$$

Weight initialization

Why is weight initialization necessary?

Without weight initializations, DNN can be tricky to train.

Question: Suppose all the layers in a DNN use ReLU as the activation function. What will happen if all weight and bias parameters are set to 0 at the beginning of training?



$$y = \max(0, x)$$
$$\frac{dy}{dx} = \begin{cases} 0 & x \le 0 \\ 1 & x > 0 \end{cases}$$

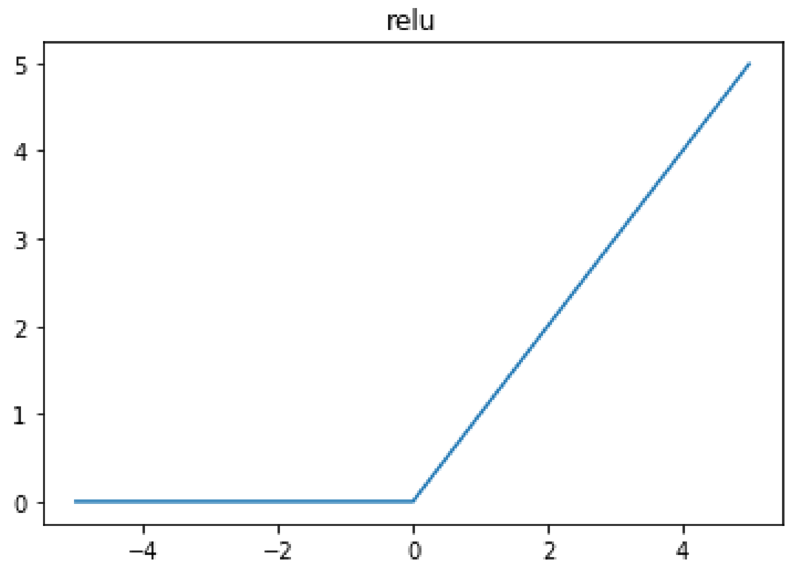
Answer: All output neurons are stuck at 0 and all neurons are 'dead' forever. The weights will never be updated despite of the incoming training data. As a result, the training is trivial.

Weight initialization

Why is weight initialization necessary?

Without weight initializations, DNN can be tricky to train.

Question: Suppose all the layers in a DNN use ReLU as the activation function. What will happen if all weight and bias parameters are set to 0 at the beginning of training?



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

$$\frac{dy}{dx} = \begin{cases} 0 & x \le 0 \\ 1 & x > 0 \end{cases}$$

Answer: All output neurons are stuck at 0 and all neurons are ‘dead’ forever. The weights will never be updated despite of the incoming training data. As a result, the training is trivial.

How to address this? Weight initialization activates some of the neurons to generate some non-zero outputs and kick off the weight update process.

Random initialization

- Random initialization is performed to start the learning process.
- Usually, we randomly initialize the weight parameters to have zero mean and small standard deviation (e.g., 0.03).

- There are two commonly used random initializations:
  - Random uniform initializations

    ```
    torch.nn.init.uniform_
    ```
  - Random normal initializations

    ```
    torch.nn.init.normal_
    ```

- However, random initialization is too arbitrary. It does not utilize any information from network structure, thus can be sub-optimal under many cases. Unless carefully tuned, random initialization does not significantly improve the performance.

Xavier initialization

- To utilize the information of network structures, Xavier initialization initializes weights according to different input units and output units.
- There are two kinds of Xavier initializations.
  - Xavier (Glorot) normal initializations

$$W \sim N\left(0, \sqrt{\frac{2}{fan\_in + fan\_out}}\right)$$

`torch.nn.init.xavier_normal_`

  - Xavier (Glorot) uniform initializations

$$W \sim U\left[-\sqrt{\frac{6}{fan\_in + fan\_out}}, \sqrt{\frac{6}{fan\_in + fan\_out}}\right]$$

`torch.nn.init.xavier_uniform_`

- Assumption for Xavier initialization mostly works for linear or quasi-linear activations.
- It does not consider the usage of non-linear rectifiers, thus may be sub-optimal in some complex CNN architectures.

Refer (https://pytorch.org/docs/stable/_modules/torch/nn/init.html#xavier_uniform_) for fan_in and fan_out calculation

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." (2010)

Xavier initialization: uniform vs. normal

- In most cases, DNN training ends up with a weight distribution that is the same as the weight distribution at initialization. Therefore, it is important to decide using uniform weight initialization or normal weight initialization.

- We recommend choosing **normal initialization** over uniform initialization. Normal initialization is more natural and stable. In addition, normal initialization brings some concrete performance gain compared to uniform initialization.

- In PyTorch, the default weight initialization method is `kaiming_uniform`. Most neural networks obtain performance gain if changing the initialization method to `xavier_normal`. You may also try `kaiming_normal`, which is a small adaptation to the `xavier_normal` method.

MSRA

- **MSRA** extends the intuition from Xavier and considers the impact of non-linear rectifiers (e.g., ReLU). As a result, **MSRA** scales the variance of current weights by only the number of input units.

- For fully-connected layers with *fan_in* input units, **MSRA** initializes the weights by

$$W \sim N\left(0, \sqrt{\frac{2}{fan\_in}}\right)$$

- For convolutional layers with $K \times K$ kernel and C input channels, MSRA initializes the weights by

$$W \sim N\left(0, \sqrt{\frac{2}{K \times K \times C}}\right)$$

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." (2015)

Neural network design: Weight initialization

How to choose weight initialization for your CNN?

- Xavier normal initialization can be taken as a good start.
  - Try other initialization options only if Xavier normal works well for your design.

- Unless instructed by the original paper, DO NOT use any form of uniform initialization as it will lead to inferior and/or unexpected performance.

- With the introduction of batch normalization, neural networks are less sensitive to initialization.

https://pytorch.org/docs/stable/nn.init.html#nn-init-doc

Babysitting the CNN training

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

The output of CNN classifiers

- The output of the last FC layer is the logits/activations of each class.
- We can further transform them into probabilities:
  - Binary classification: sigmoid $\sigma(z) = \frac{e^z}{1+e^z}$
  - Multi-class classification: softmax $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$

$$\begin{tikzcd} \arrow[r] & \boxed{\text{Model}} \arrow[r] & 2.0 \arrow[r, "\sigma"] & 0.88 \end{tikzcd}$$

88% confident

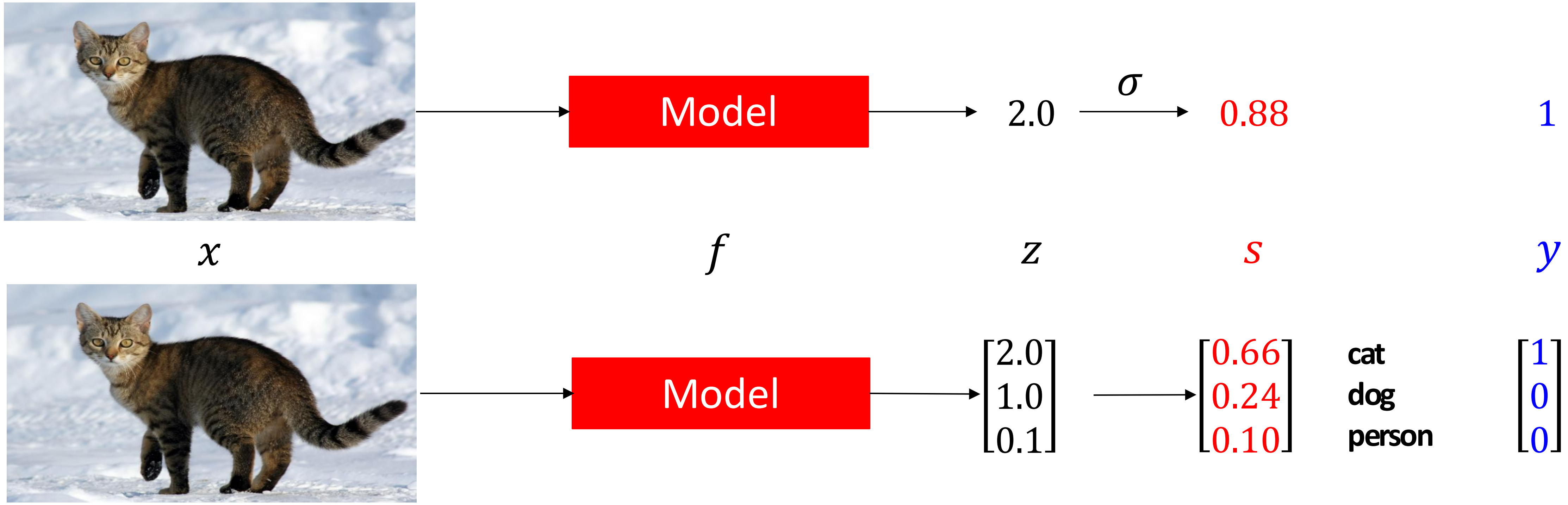
x f z s

$$\begin{tikzcd} \arrow[r] & \boxed{\text{Model}} \arrow[r] & \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix} \arrow[r] & \begin{bmatrix} 0.66 \\ 0.24 \\ 0.10 \end{bmatrix} \end{tikzcd} \quad \begin{array}{l} \text{cat} \\ \text{dog} \\ \text{person} \end{array}$$

66% confident

Cross-entropy Loss – high level

- CNN classifiers are generally trained by minimizing the cross-entropy loss between predicted distribution and target (ground-truth) distribution



- By minimizing cross-entropy, we are making two distributions as close as possible
  - Minimizing cross-entropy for classifier is equivalent to Maximum Likelihood Estimation
  - Useful posts/discussions [1] [2] [3]
- There are other loss functions (e.g. MSE), but CE loss generally work better

Cross-entropy loss – details

Notation: y_j is the probability of the jth class in the target distribution. s_j is the probability of the jth class in the predicted distribution.

- **Cross-entropy loss**

$$l_{\text{CE}} = -\sum_j y_j \log(s_j)$$

Note that s_i must be a valid probability distribution. Otherwise, the cross-entropy loss will explode.

$$\begin{tikzcd} x \arrow[r] & \boxed{\text{Model}}\ f \arrow[r] & z = \begin{bmatrix}2.0\\1.0\\0.1\end{bmatrix} \arrow[r] & s = \begin{bmatrix}0.66\\0.24\\0.10\end{bmatrix} & \begin{array}{l}\textbf{cat}\\\textbf{dog}\\\textbf{person}\end{array} & y = \begin{bmatrix}1\\0\\0\end{bmatrix}\end{tikzcd}$$

$$l_{\text{CE}} = -(1 \times \log 0.66 + 0 \times \log 0.24 + 0 \times \log 0.10) = 0.415$$

Cross-entropy loss

Notation: y_j is the probability of the jth class in the target distribution. s_j is the probability of the jth class in the predicted distribution.

- **Cross-entropy loss**

$$l_{\text{CE}} = -\sum_j y_j \log(s_j)$$

Question: Suppose there are *N* classes. What is the value of the loss function if the weights ***W*** are initialized with evenly random numbers?

Cross-entropy loss

Notation: y_j is the probability of the jth class in the target distribution. s_j is the probability of the jth class in the predicted distribution.

- **Cross-entropy loss**

$$l_{\mathrm{CE}} = -\sum_j y_j \log(s_j)$$

Question: Suppose there are *N* classes. What is the value of the loss function if the weights ***W*** are initialized with evenly random numbers?

Answer: $\log N$. Because the random guess is performed when DNN has learned nothing. All ***s*** will be approximately equal (i.e., $\frac{1}{N}$).

You may do a sanity check on the initial loss value before starting the training. For example, for CIFAR-10 dataset with N=10 classes, the initial loss shall be around 2.31.

Babysitting the CNN training

Lecture 5: Basic techniques

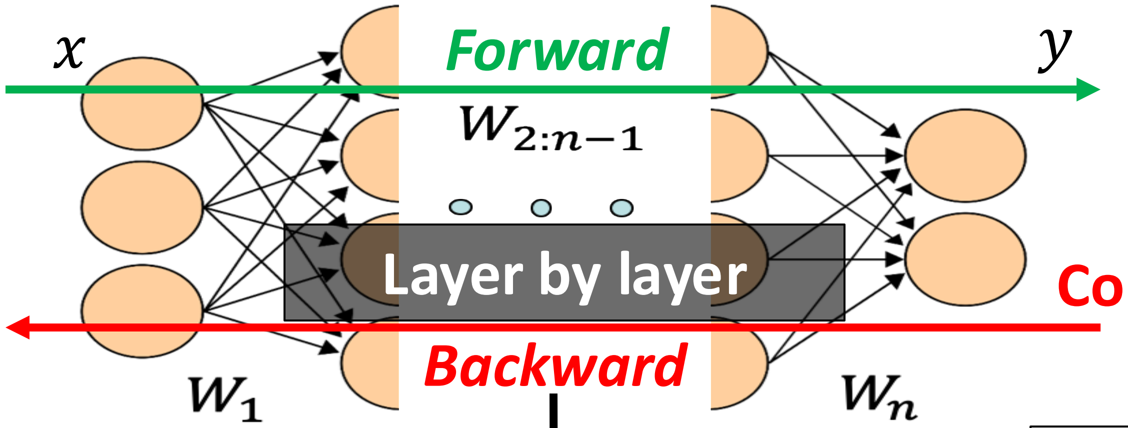
- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

Recall: Training overview

$f_i(W_i,\cdot)$: i_{th} layer (linear/conv) with **weight** W_i followed by **Activation function**

$$\boxed{y = F(x) \coloneqq f_n(W_n, f_{n-1}(W_{n-1}, f_{\dots} f_1(W_1, x)))}$$



New batch of data

Weight update

Optimizer

Calculate Prediction y

Compare with target d

$$\boxed{Loss = L(y, d)}$$

Loss Function

Gradient of i_{th} layer's weights

$$\boxed{\frac{\partial Loss}{\partial W_i} = \frac{\partial L}{\partial f_n} \cdot \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdots \frac{\partial f_i}{\partial W_i}}$$

Recall: Optimizing the loss function

- The loss function is complex and there's no closed-form solution to the gradient formula.
- We don't have all the data for loss function computation.

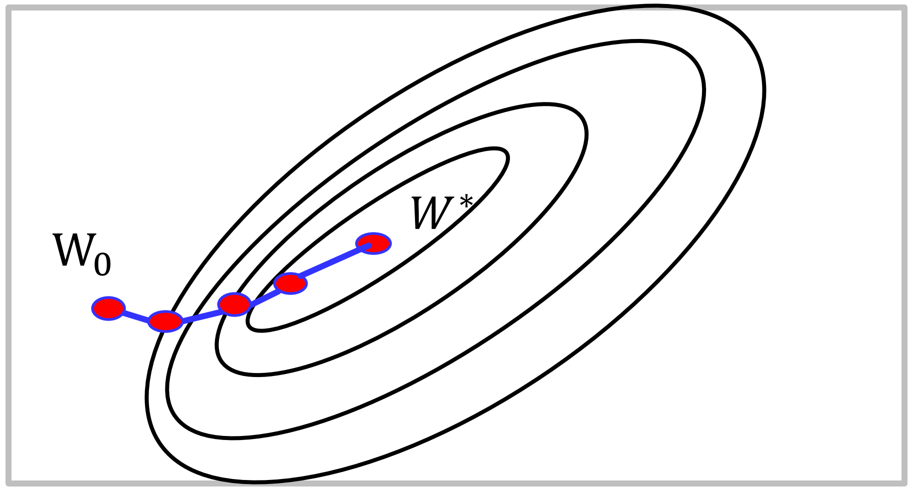
Solution:

- We can find the W^* in an iterative way

$$W^{t+1} = W^t - \alpha \frac{\partial l}{\partial W}(W^t)$$

α: learning rate, defining the step size

- This process is called **Gradient Descent!**
- Three variants
  - **Batch Gradient Descent**: Take **all** samples from the whole dataset at each step
    - Better convergence to the minima; heavy computation (e.g., millions of samples)
  - **Stochastic Gradient Descent**: Randomly take **one** sample at each step
    - Fast computation; will oscillate severely around the minima
  - **Mini-batch Gradient Descent**: Randomly sample a **mini-batch** of examples at each step (SGD is a special case where batch size is 1)
- Mini-batch Gradient Descent is often referred to as SGD (also in our later slides)

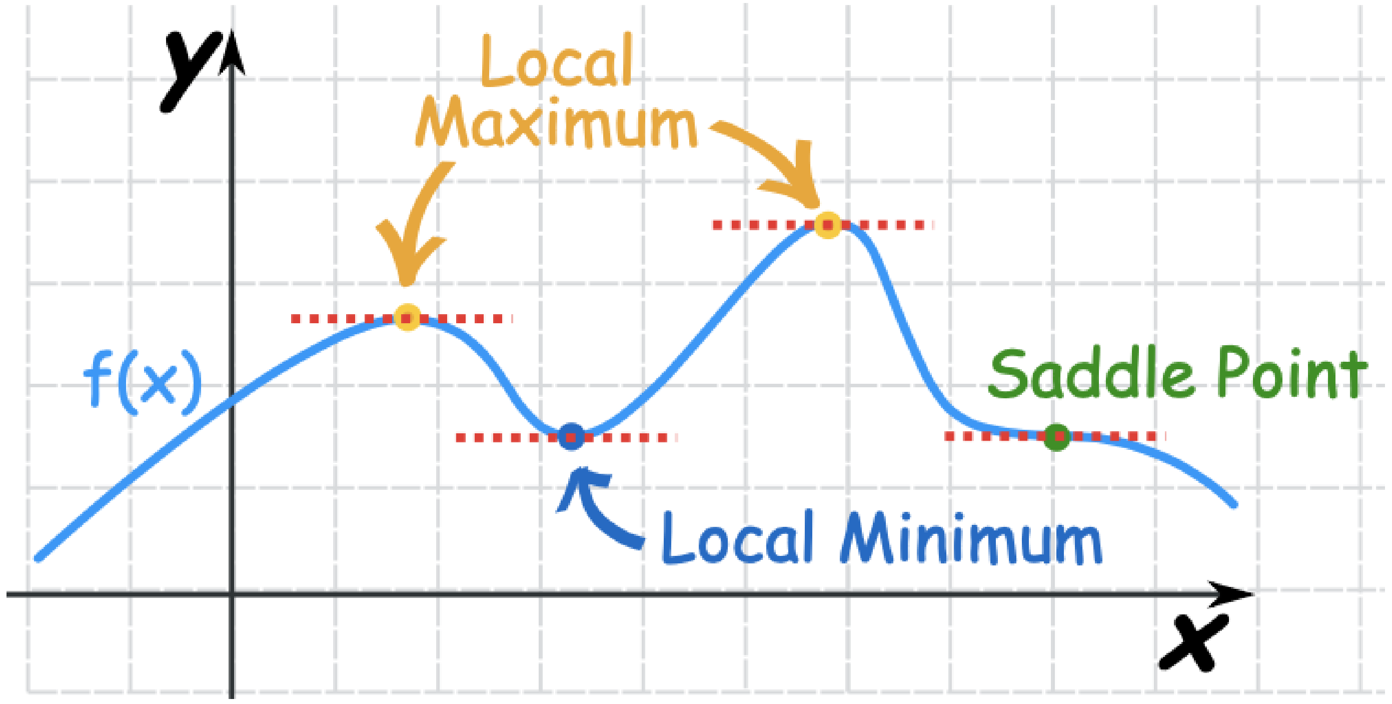


Vanilla SGD

Vanilla SGD updates the weights using the computed **vanilla gradient**:

$$W^{t+1} = W^{t} - \alpha \nabla_{W^{t}} l(W^{t}) \quad \alpha\text{: Learning rate}$$

SGD is likely to get stuck into local minimum and saddle point. Once SGD gets into a local minimum, it can never get out.



SGD with momentum

SGD with momentum avoids the saddle point by “overshooting” local optimal values with accumulated momentum.

- Gradient with **momentum** is computed and used to speed up the converging process. The momentum is accumulated in each optimization step.

$$v^{t+1} = \mu v_t - \alpha \nabla_{W^t} l(W^t)$$
$$W^{t+1} = W^t + v^{t+1}$$

α: Learning rate

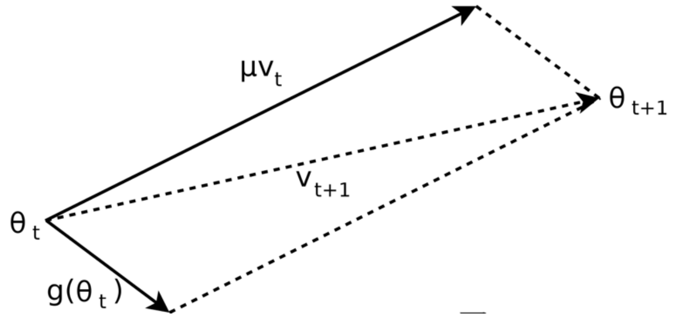
μ: Momentum (factor)

Advantages:

- Little computational cost during weight updates.
- Can reach the state-of-the-art performance

Problems:

- Requires more tuning time
- May overshoot the global minimum.



SGD with Nesterov momentum

Nesterov momentum accumulates the momentum by "looking ahead."

- More specifically, it mixes the current weight parameter with velocity to compute the gradient.

$$v^{t+1} = \mu v_t - \alpha \nabla_W l(W^t + \mu v_t)$$
$$W^{t+1} = W^t + v^{t+1}$$

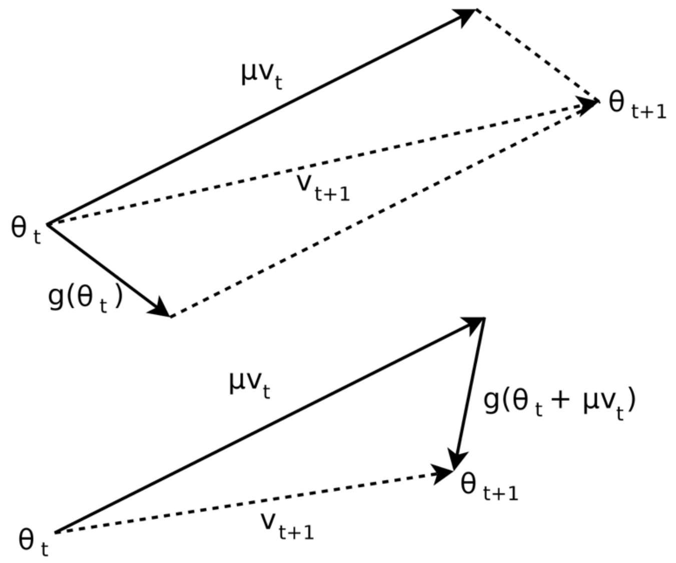
α: Learning rate
μ: Momentum (factor)

Advantages:

- Avoid oscillations during optimization
- Can reach state-of-the-art performance

Problems:

- Convergence still take a long time



SGD + Momentum

SGD + Nesterov momentum

Source: Sutskever et al., (2013), On the importance of initialization and momentum in deep learning

Adaptive gradient (Adagrad)

Adagrad speeds up the training process by using adaptive learning rate for each weight parameter.

$$W^{t+1} = W^t - \frac{\alpha \nabla_W l(W^t)}{\sqrt{G+\epsilon}}$$

$$G = \sum_{i=0}^{t} \left(\nabla_W l(W^t)\right)^2$$

α: Learning rate
ϵ : A small number to avoid zero division

Advantage:

- Increase the robustness/smoothness of SGD update
- Leads to faster convergence speed

Problem:

- Accumulation of gradients in the denominator grows very fast. It may shrink learning rate too quickly
- May yield sub-optimal weights

Root mean square propagation (RMSprop)

To address the problem in Adagrad, RMSprop uses a moving average of incoming gradients as the denominator.

$$G^{t+1} = \mu G^t + (1-\mu)\left(\nabla_W l(W^t)\right)^2$$

$$W^{t+1} = W^t - \frac{\alpha \nabla_W l(W^t)}{\sqrt{G^{t+1}+\epsilon}}$$

α: Learning rate
μ: Momentum (factor)
G^t: gradient norm at time step t.
ϵ : Small number to avoid zero division

Advantages:

- Prevent learning rate shrinkage
- Faster convergence speed

Problem:

- Oscillation at the end of convergence

Adam

Adam combines the advantages of **SGD with momentum** and **RMSprop Optimizer**.

- $g_t = \nabla_W l(W^t)$
- $m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t$ first momentum estimate
- $v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2$ second momentum estimate
- $\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}$ bias-correlated first-order momentum estimate
- $\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$ bias-correlated second-order momentum estimate
- $W^{t+1} = W^t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ update, ϵ is a small number to avoid zero division

The use of the first order and second order statistics gives Adam the fastest convergence speed. However, Adam usually oscillates loss values at the end of convergence. As a result, Adam is not stable and may yield sub-optimal performance compared to SGD optimizers.

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." (2014)

Neural network design: Optimizer

Which optimizer should we use for training?

- **Adam** is a good default choice in many cases because it has the fastest convergence speed.
- **SGD with momentum/Nesterov momentum** can outperform **Adam**, but it takes a longer time to tune the weight parameters and reach convergence (e.g., 300~400 epochs for a CNN training).

Pytorch Optimizer	Description
`torch.optim.Adagrad`	Implement Adagrad algorithm
`torch.optim.Adam`	Implement Adam algorithm
`torch.optim.ASGD`	Implement averaged SGD
`torch.optim.RMSprop`	Implement RMSprop algorithm
`torch.optim.SGD`	Implement SGD (optionally with momentum)

Gradient clipping

- There are cases where some batch of data may provide large gradients. It may harm the training process and mess up with the weights.
- **Gradient clipping** was introduced to solve the problem of exploding gradients and stabilize the training process.
  - Gradient clipping removes the outliers in the gradient and caps them within a given range.
  - It is a common practice to clip the gradient values to (-5, 5) to stabilize training for CNNs.
  - You can also try to clip the norm of the gradient.

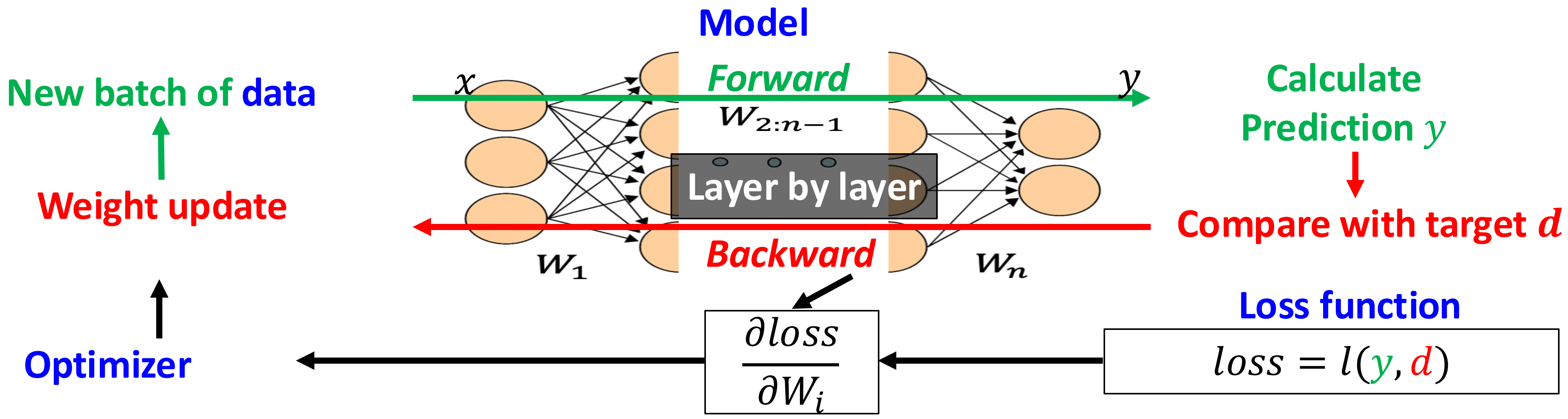
PyTorch API:

- `torch.nn.utils.clip_grad_norm_(parameters, max_norm, norm_type=2.0, error_if_nonfinite=False, foreach=None)`
- `torch.nn.utils.clip_grad_value_(parameters, clip_value, foreach=None)`

In this lecture, we learned:

The basic techniques to design and train a CNN model by answering the following questions:

- What does a good practice of **CNN design** look like?
- How to choose **the number of filters** in a CNN?
- How to decide **the number of convolutional layers**?
- Which **activation function** should we use?
- How to choose **weight initialization** for your CNN?
- Which **loss function** and **optimizer** should we use for training?


Model
New batch of data
Weight update
Optimizer
x
Forward
$W_{2:n-1}$
y
Layer by layer
Backward
W_1
W_n
$\frac{\partial loss}{\partial W_i}$
Calculate Prediction y
Compare with target d
Loss function
$loss = l(y, d)$