Lecture 18: Physical Layer Implementations and Troubleshooting

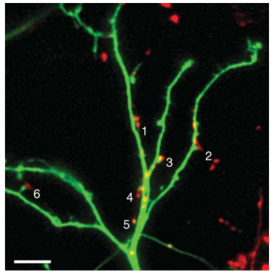
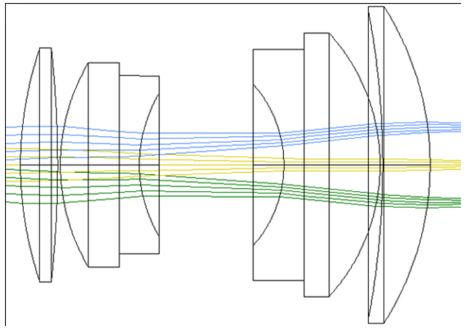
Machine Learning and Imaging

BME 590L

Roarke Horstmeyer

Summary of two models for image formation

- **Interpretation #1: Radiation (*Incoherent*)**
- Model: Rays

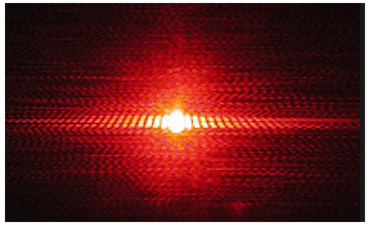




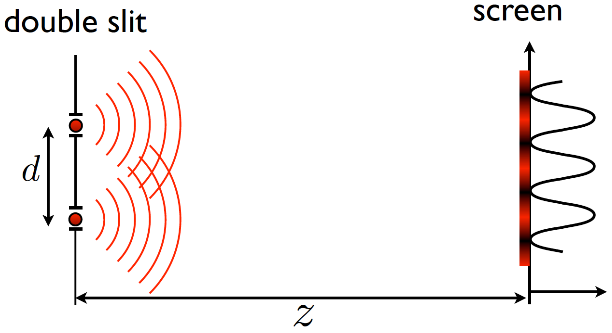
- Real, non-negative

$$\boxed{I_s = H\,B\,S_0}$$

- Sample absorption **S**
- Illumination brightness **B**
- Blur in **H**

- **Interpretation #2: Electromagnetic wave (*Coherent*)**
- Model: Waves





- Complex field

$$\boxed{I_C = |\,H\,C\,S_C|^2}$$

- Sample abs./phase **S**
- Illumination wave **B**
- Blur in **H**

deep imaging

Questions to address in this lecture

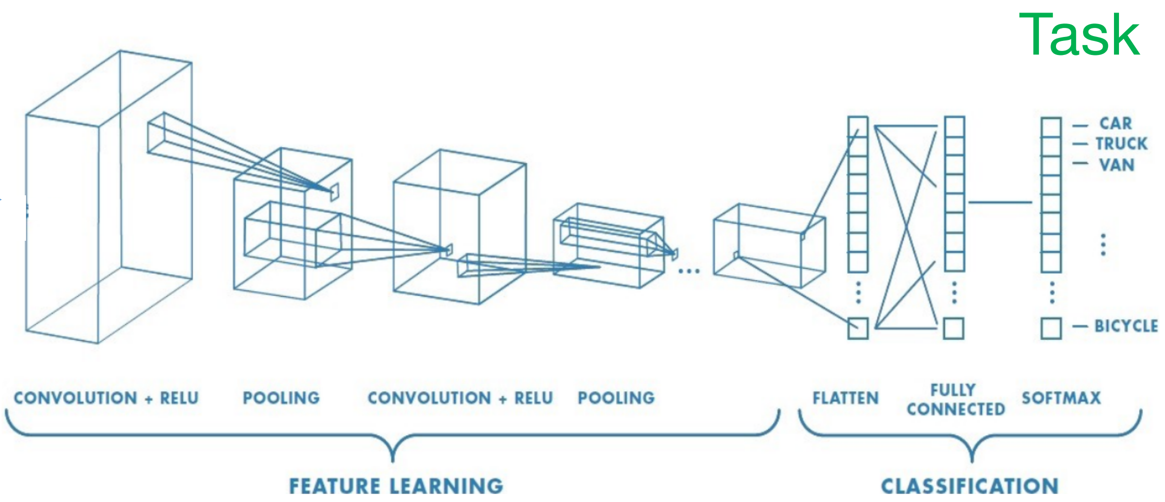
- Where and how should I implement my physical layer?
  - Simulation data
  - Experimental data
- How can I add some constraints to the physical weights that I'm optimizing?
- What are some common issues and pitfalls?

Physical Layers

Digital Layers

Input image data $\mathbf{I_0}$ → $f[\;]$ → Digitized $\mathbf{I_s}$ →

$$\mathbf{I_s} = f[\mathbf{I_0}]$$



Some Examples:

- Optimized illumination
- Optimized sensor specifications
- Number of measurements and locations
- Radiation dosage, biomarkers

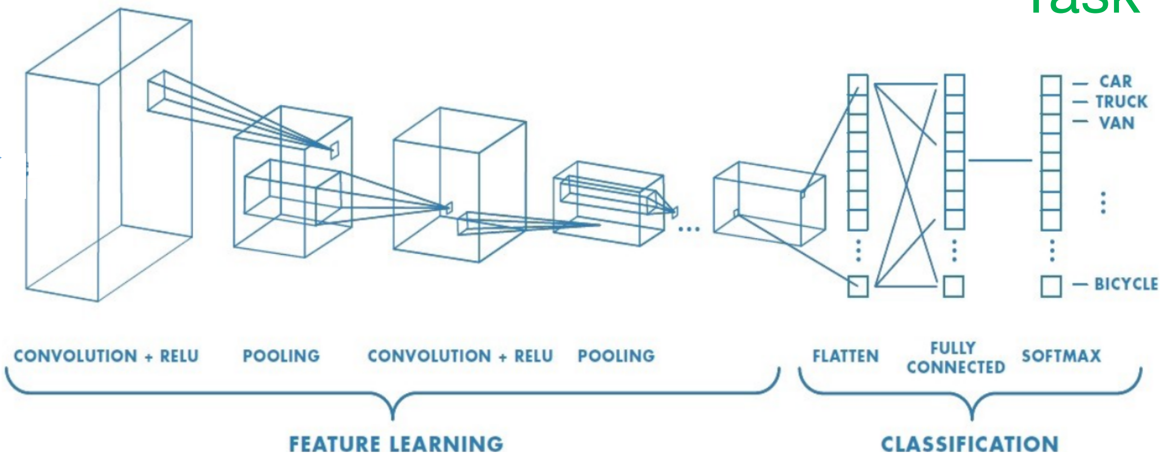
Physical Layers

Digital Layers

Input image data $\mathbf{I}_0$ → f[ ] → Digitized $\mathbf{I}_s$ →

$$\mathbf{I}_s = f[\mathbf{I}_0]$$

Task



Q: Where and how should I implement my physical layer?

deep imaging

Physical Layers

Digital Layers

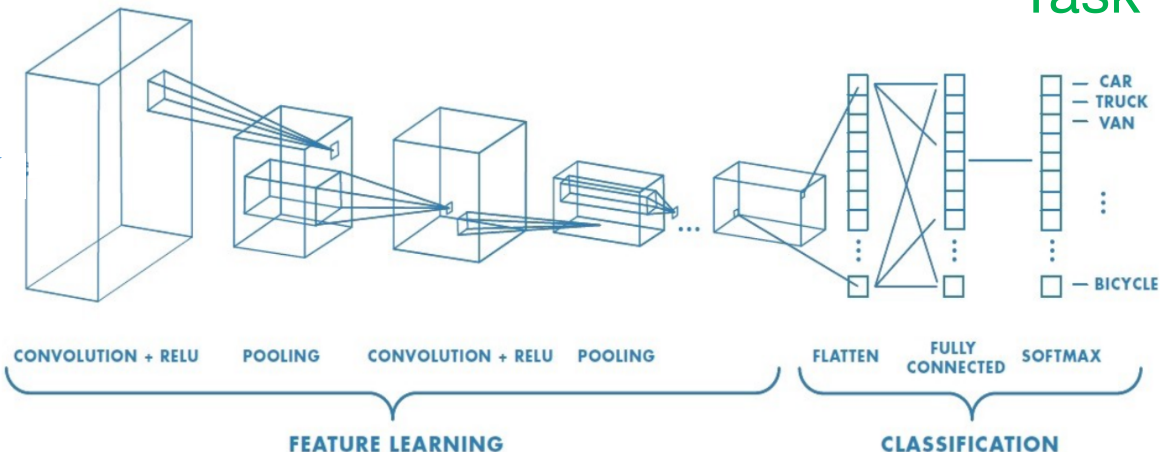
Input image data $\mathbf{I}_0$

f[ ]

$\mathbf{I}_s = f[\mathbf{I}_0]$

Digitized $\mathbf{I}_s$

Task



Q: Where and how should I implement my physical layer?

A: It depends on your data and implementation

- Situation #1: Fully simulated physical layers
- Situation #2: Experimentally-driven physical layers

Situation #1: Fully simulated physical layers

Option (a): Simulate the input images and the labels from scratch

Full simulation of different samples and their labels

Physical Layers

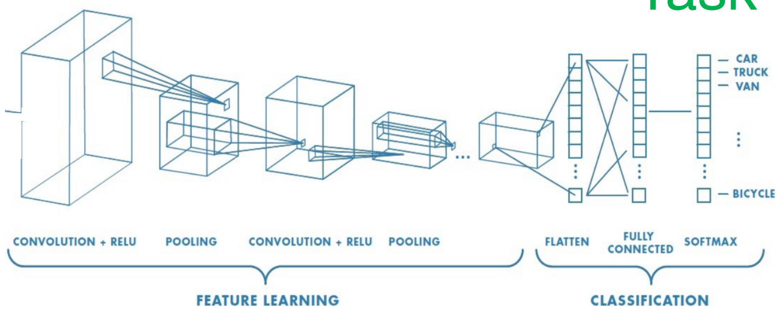
$f[\ ]$

$$\mathbf{I_s} = f[\mathbf{I_0}]$$

Digitized $\mathbf{I_s}$

Digital Layers

Task



Simulated labels

Situation #1: Fully simulated physical layers

Option (a): Simulate the input images and the labels from scratch

Physical Layers

Digital Layers

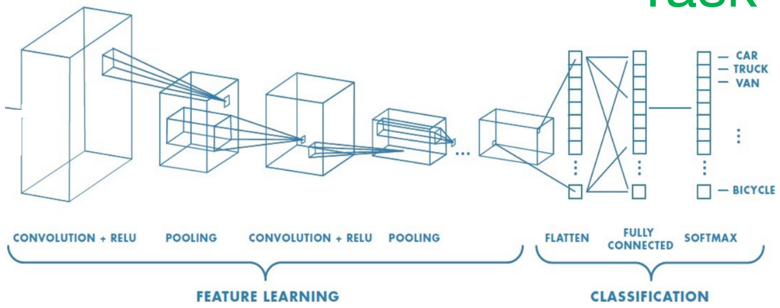
Full simulation of different samples and their labels

$f[\ ]$

$$\mathbf{I_s} = f[\mathbf{I_0}]$$

Digitized $\mathbf{I_s}$

Task



Simulated labels

Examples:

- [Ultrasound scatterers, segmentation boundary]
- [Simulated cell body types, location]
- [CT phantom, 3D mesh surfaces]

Situation #1: Fully simulated physical layers

Option (b): Augment an existing dataset that you download

Existing labels

Physical Layers

Digital Layers

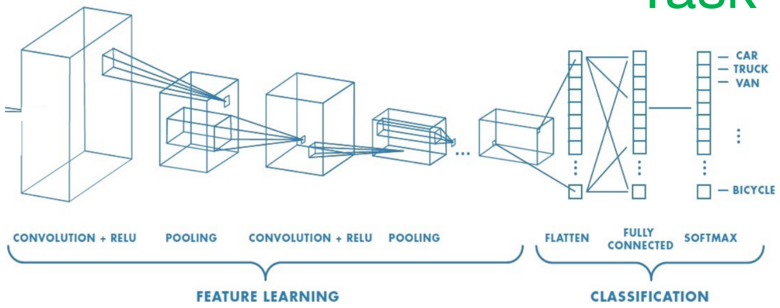
Existing annotated dataset $\mathbf{I}_0$

f[ ]

$$\mathbf{I}_s = f[\mathbf{I}_0]$$

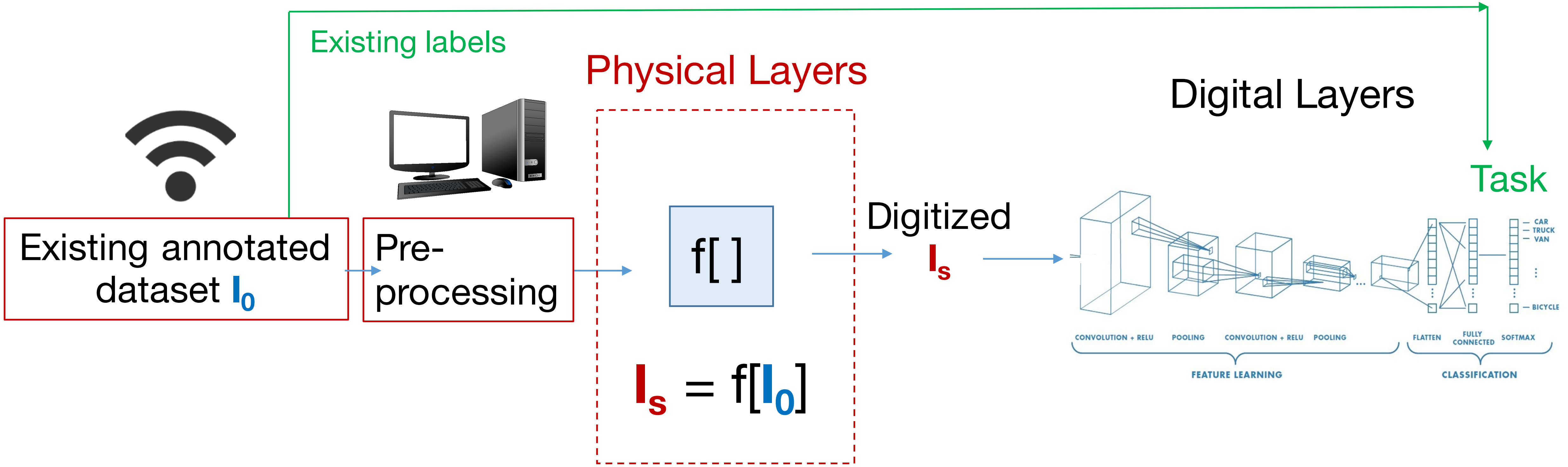
Digitized $\mathbf{I}_s$

Task



Situation #1: Fully simulated physical layers

Option (b): Augment an existing dataset that you download



Situation #1: Fully simulated physical layers

Option (b): Augment an existing dataset that you download

Existing labels

Physical Layers

Digital Layers

Existing annotated dataset $\mathbf{I}_0$

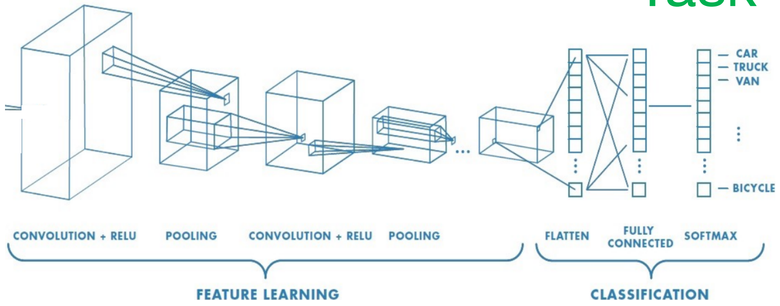
Pre-processing

f[ ]

$$\mathbf{I}_s = f[\mathbf{I}_0]$$

Digitized $\mathbf{I}_s$

Task



Examples:

- MNIST Image set
- Segmented cells from Celltracker
- Segmented CT dataset from lab
- Thickness map
- Multispectral image stack
- Stitch together in a 3D composite

Situation #1: Fully simulated physical layers

Option (a) or Option (b): Choice on where and how to simulate/pre-process

Simulation and/or pre-processing

ML Optimization

Python/Matlab/other

Big dataset

TensorFlow

TensorFlow

TensorFlow

deep imaging

Situation #1: Fully simulated physical layers

Option (a) or Option (b): Choice on where and how to simulate/pre-process

Simulation and/or pre-processing

ML Optimization

Python/Matlab/other → Big dataset → TensorFlow

Pros: Utilize old code, easier to archive, troubleshoot

Cons: Large datasets are slow to load, hard to fit in GPU memory, code in 2 places

TensorFlow → TensorFlow

Pros: batch processing, all in one place, easily incorporate additional physical layers

Cons: Harder to bug-check /compare to prior work if closely integrated

Situation #2: Experimentally-driven physical layers

deep imaging

Generated Labels

Experimental measurements → "Expert" annotation

Physical Layers

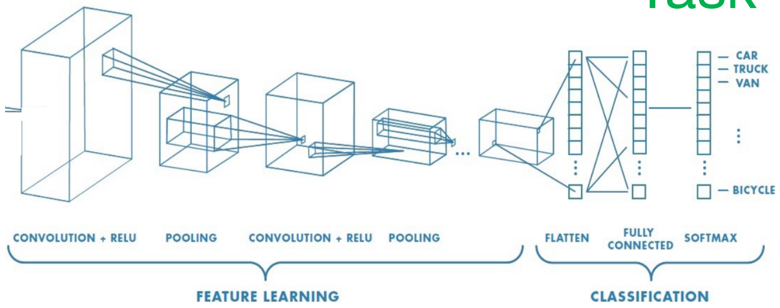
f[ ]

$\mathbf{I_s} = f[\mathbf{I_0}]$

Digitized $\mathbf{I_s}$

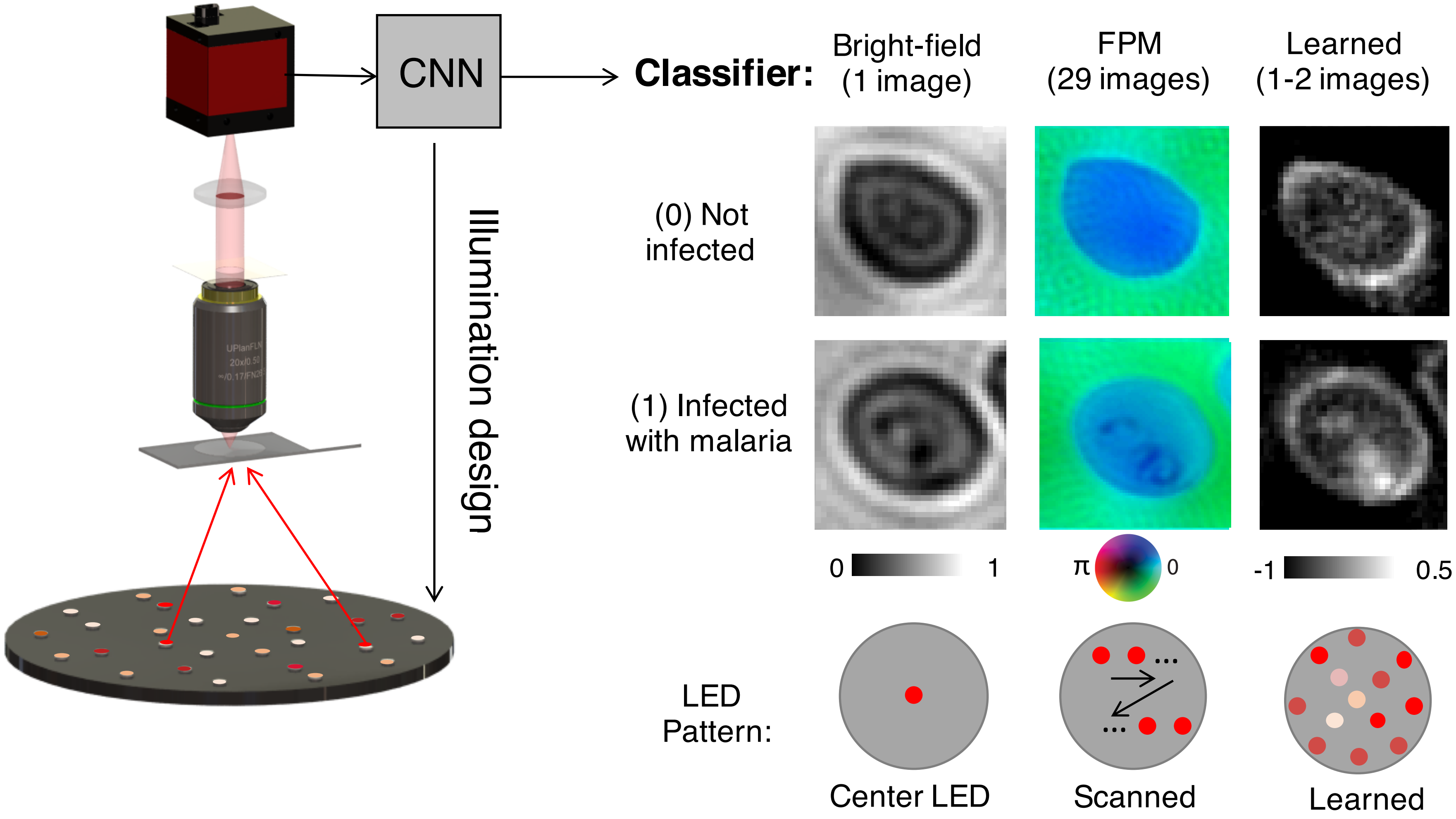
Digital Layers

Task



Situation #2: Experimentally-driven physical layers

Example: CNN-Optimized illumination for classification of malaria:

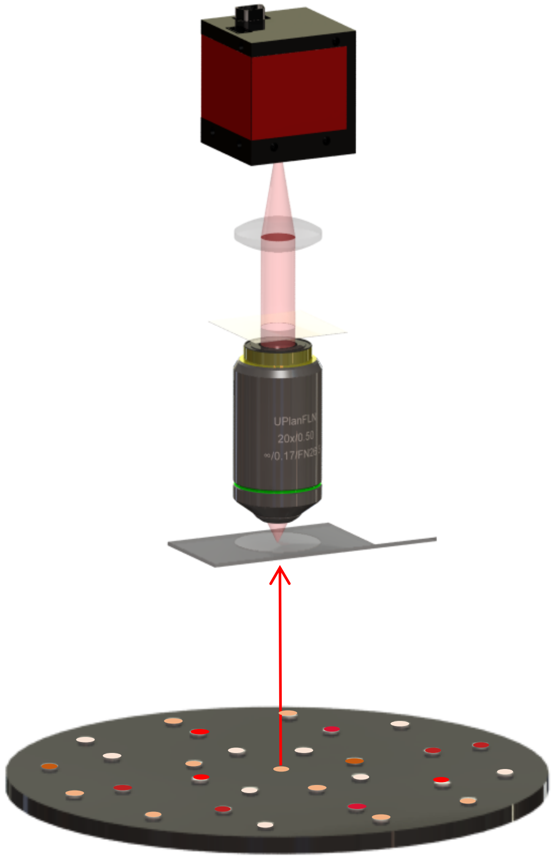


Classification Accuracy: **75%** **98.5%** **97%**

R. Horstmeyer et al., "Convolutional neural networks that teach microscopes how to image," arXiv (2017)

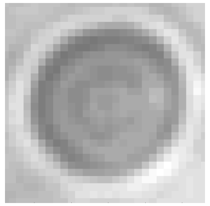
Situation #2: Experimentally-driven physical layers

Example: CNN-Optimized illumination for classification of malaria:



Data set for physical layer optimization:

Turn on LED 1, capture image 1:



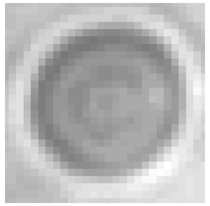
R. Horstmeyer et al., "Convolutional neural networks that teach microscopes how to image," arXiv (2017)

Situation #2: Experimentally-driven physical layers

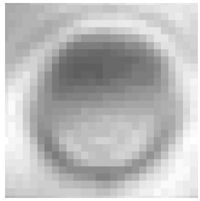
Example: CNN-Optimized illumination for classification of malaria:

Data set for physical layer optimization:

Turn on LED 1, capture image 1:



Turn on LED 1, capture image 2:



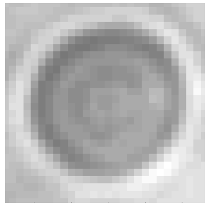
R. Horstmeyer et al., "Convolutional neural networks that teach microscopes how to image," arXiv (2017)

Situation #2: Experimentally-driven physical layers

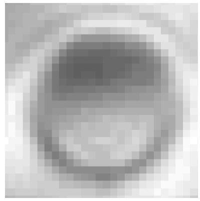
Example: CNN-Optimized illumination for classification of malaria:

Data set for physical layer optimization:

Turn on LED 1, capture image 1:

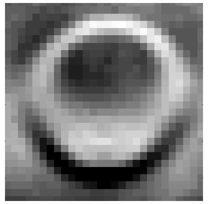


Turn on LED 1, capture image 2:



⋮

Turn on LED 32, capture image 32:

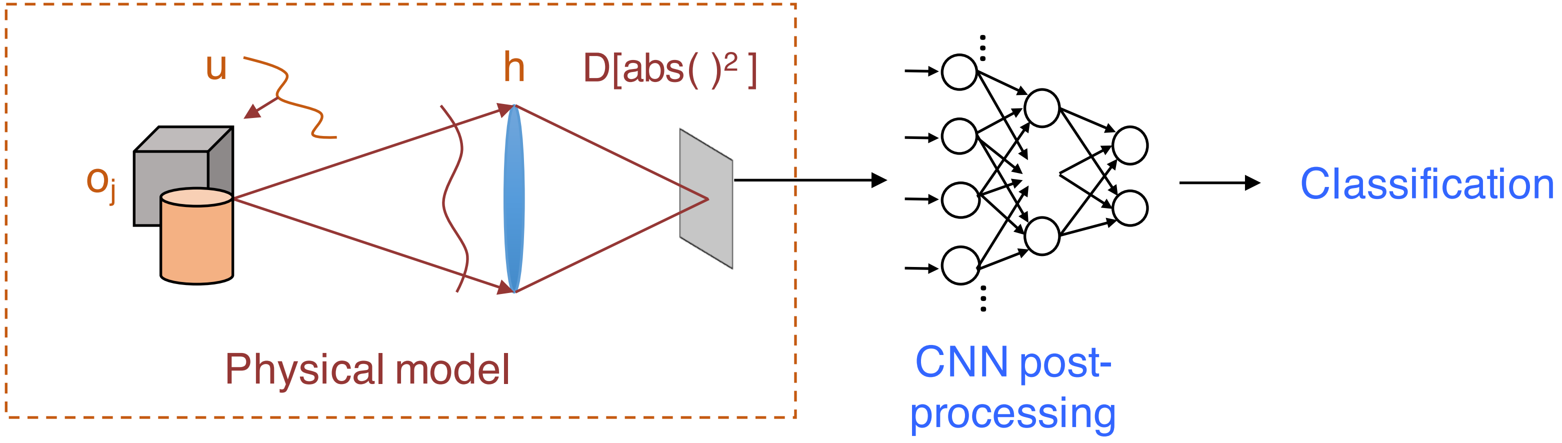


Save all 32 images (96 with 3 colors)

R. Horstmeyer et al., "Convolutional neural networks that teach microscopes how to image," arXiv (2017)

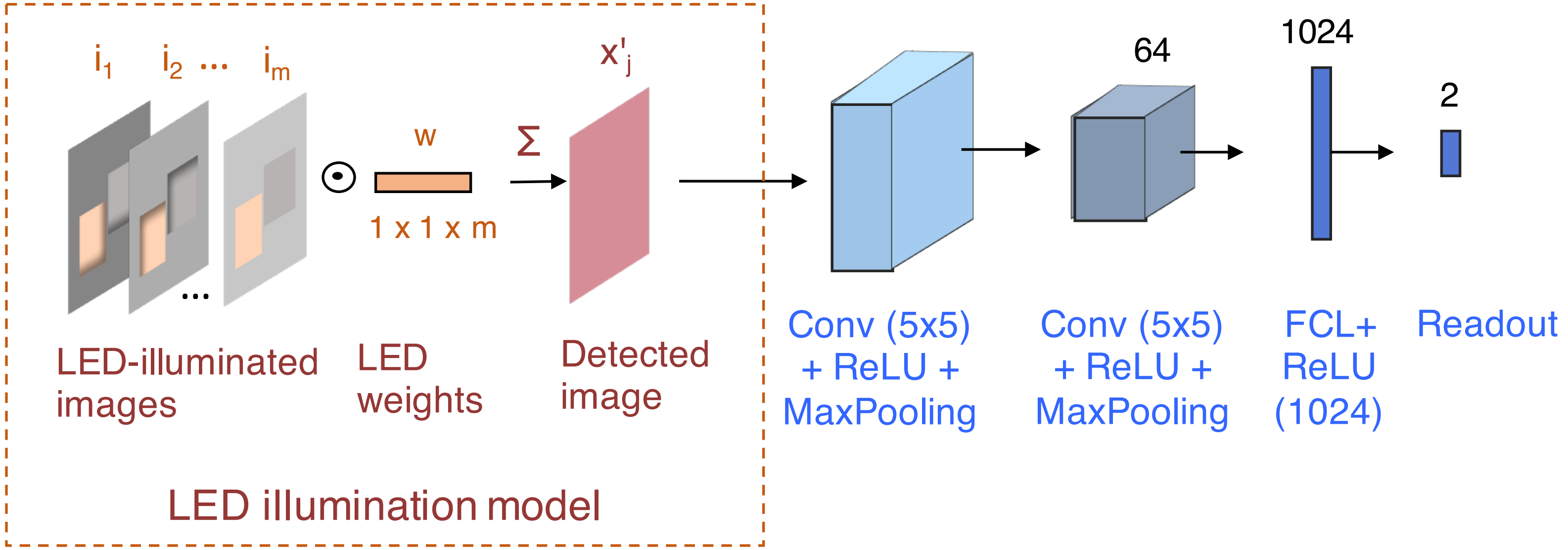
Situation #2: Experimentally-driven physical layers

Example: CNN-Optimized illumination for classification of malaria:



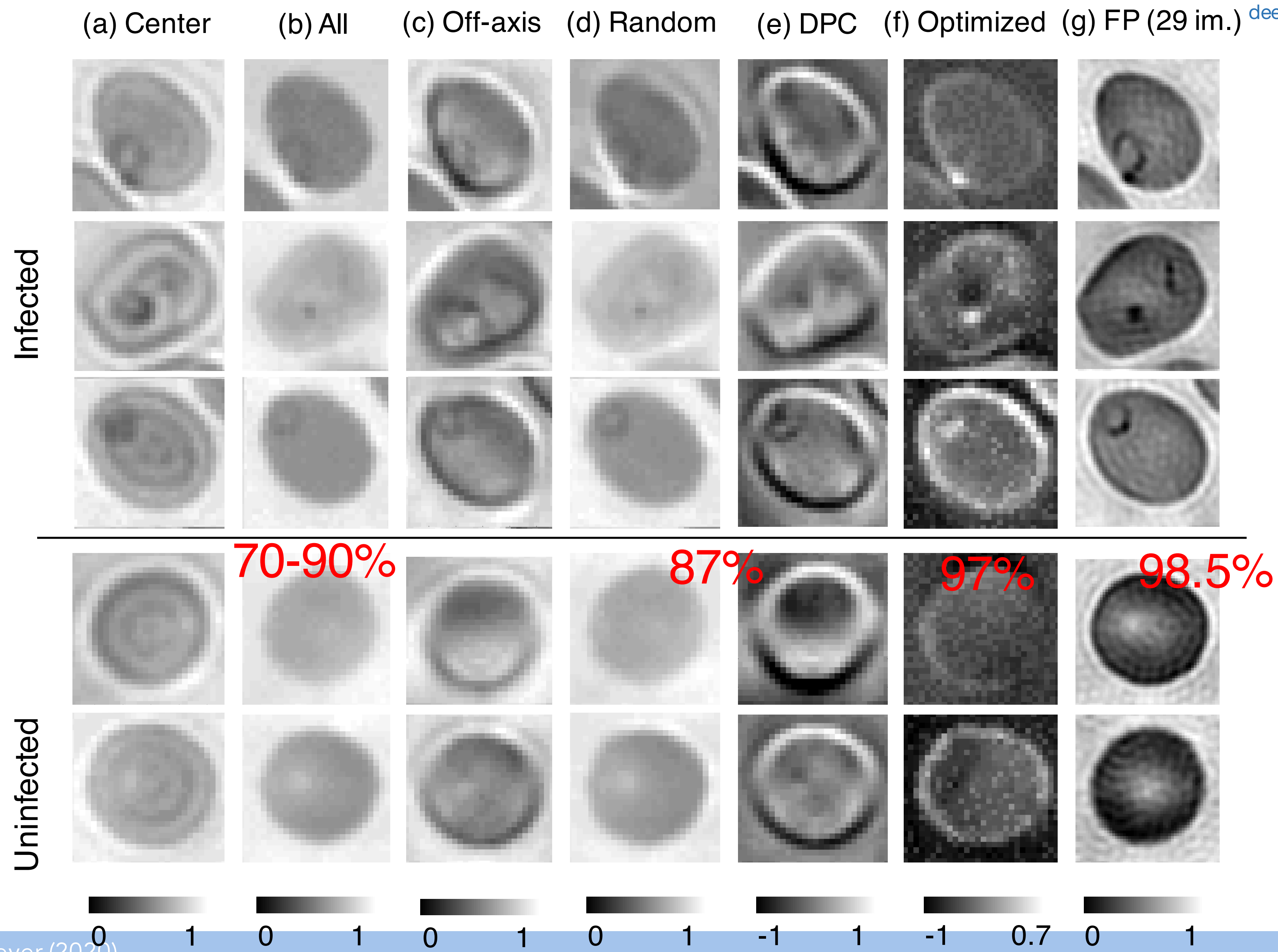
Physical layer:

$$I_s = \sum w_j I_j$$



Situation #2: Experimentally-driven physical layers

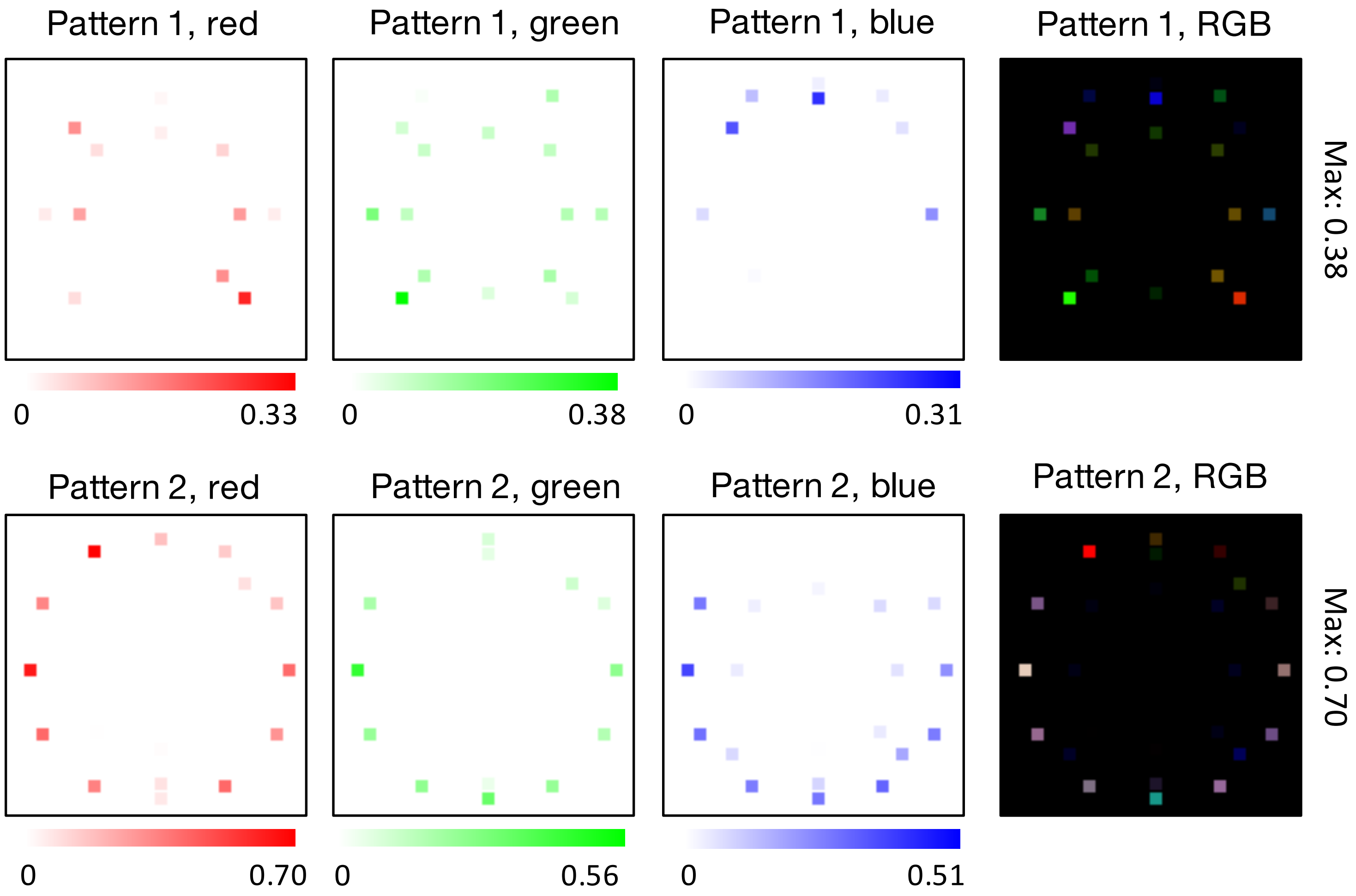
Example: CNN-Optimized illumination for classification of malaria:



deep imaging

Situation #2: Experimentally-driven physical layers

Optimized color LED patterns to classify malaria



Situation #2: Experimentally-driven physical layers

Generated Labels

Experimental measurements → "Expert" annotation

Physical Layers

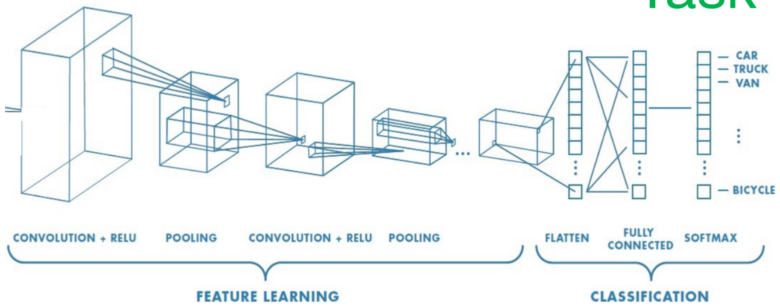
f[ ]

$$\mathbf{I_s} = f[\mathbf{I_0}]$$

Digitized $\mathbf{I_s}$

Digital Layers

Task



<u>Pro's of experimental measurements</u>: Don't need to worry about making your simulations match the setup! (HUGE WIN)

<u>Con's of experimental measurements:</u> You'll need to label them, limited access to desired sample information, often need to exploit some fundamental physical property

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can’t realize any real or physical number?

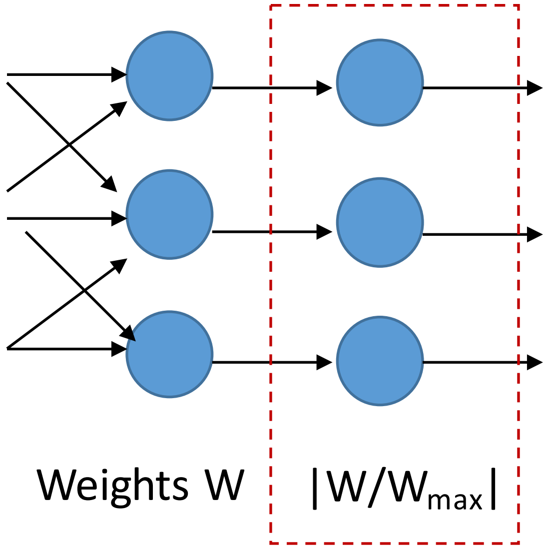
Example: Constrain weights to be non-negative values less than one

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can't realize any real or physical number?

Example: Constrain weights to be non-negative values less than one

Solution: add constraint as an extra "differentiable" layer (operation)



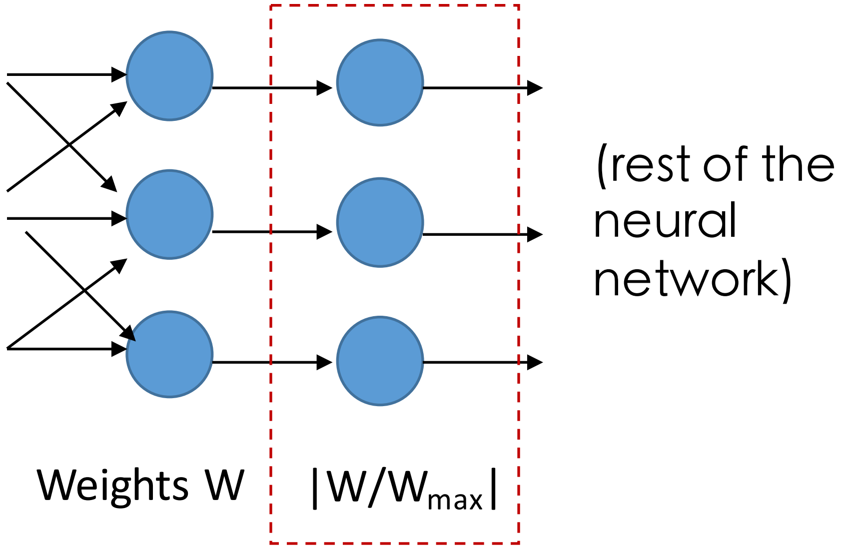
(rest of the neural network)

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can't realize any real or physical number?

Example: Constrain weights to be non-negative values less than one

Solution: add constraint as an extra "differentiable" layer (operation)



Pros:
- Easy to implement
- Constraints are obvious

Cons:
- Not always a well-behaved derivative

deep imaging

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can't realize any real or physical number?

Example: Constrain weights to be either 0 or 1

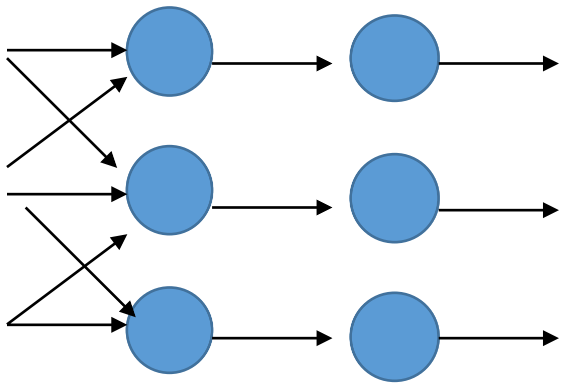
Solution: Perform annealing with a temperature parameter

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can't realize any real or physical number?

Example: Constrain weights to be either 0 or 1

Solution: Perform annealing with a temperature parameter



(rest of the neural network)

Weights W $I(n) = \textbf{Soft-max}\left[\alpha_t w(n)\right]$

Increase α with iteration number

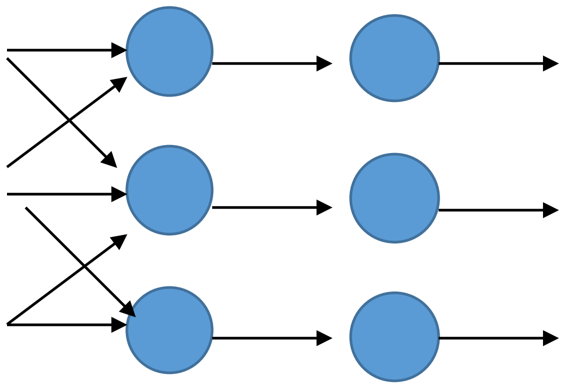
Soft-max(x) = exp(-x)/ Σ exp(-x)

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can't realize any real or physical number?

Example: Constrain weights to be either 0 or 1

Solution: Perform annealing with a temperature parameter

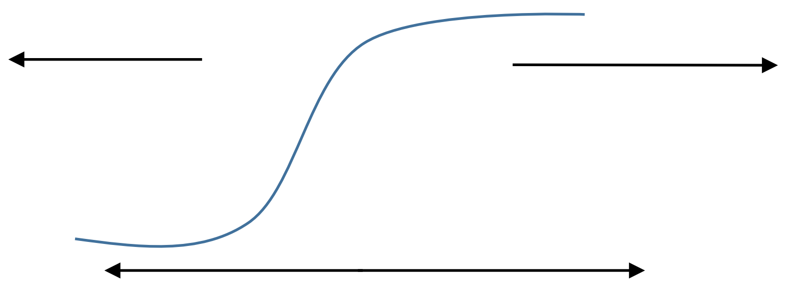


(rest of the neural network)

Weights W $I(n) = \textbf{Soft-max}\left[\alpha_t w(n)\right]$

Increase α with iteration number

Drive w to be large, so softmax(w) -> 0 or 1



Weight w

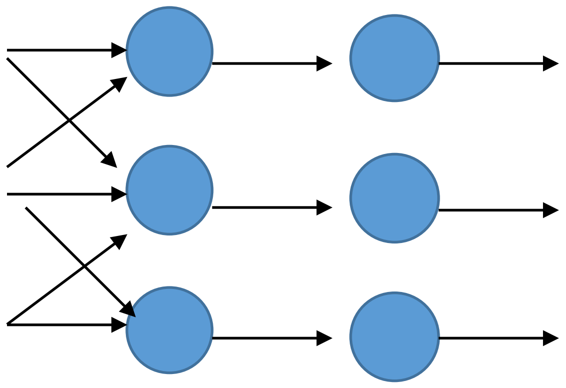
Soft-max(x) = exp(-x)/ Σ exp(-x)

How can I add some constraints to my physical weights?

Without any constraints, weights can be any real (or complex) number What if you physically can't realize any real or physical number?

Example: Constrain weights to be either 0 or 1

Solution: Perform annealing with a temperature parameter



(rest of the neural network)

Pros:

- Works pretty well
- Flexibly address convergence issues

Cons:

- A bit sensitive

Weights W $$I(n) = \textbf{Soft-max}\left[\alpha_t w(n)\right]$$

Increase α with iteration number

Soft-max(x) = exp(-x)/ Σ exp(-x)

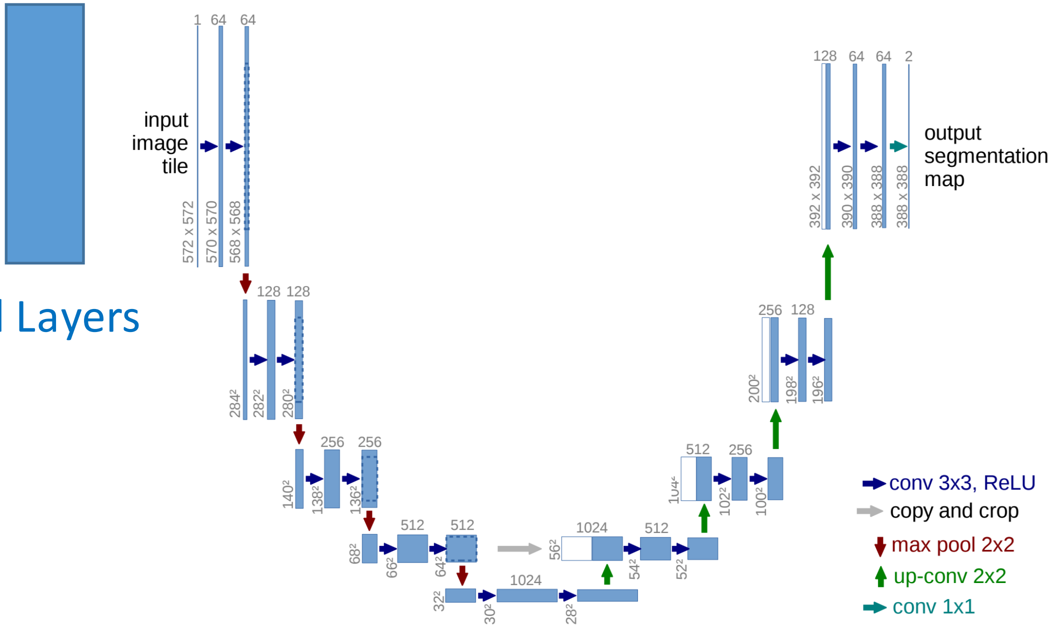
What are some common issues and pitfalls with physical layers?

- Most common issue – you have a bug in your CNN!
  - Solution: "Disable " physical layer (set to constant), and get network to work!
  - Good practice: always compare performance with and without physical layer
- Another common challenge - vanishing gradients

deep imaging

What are some common issues and pitfalls with physical layers?

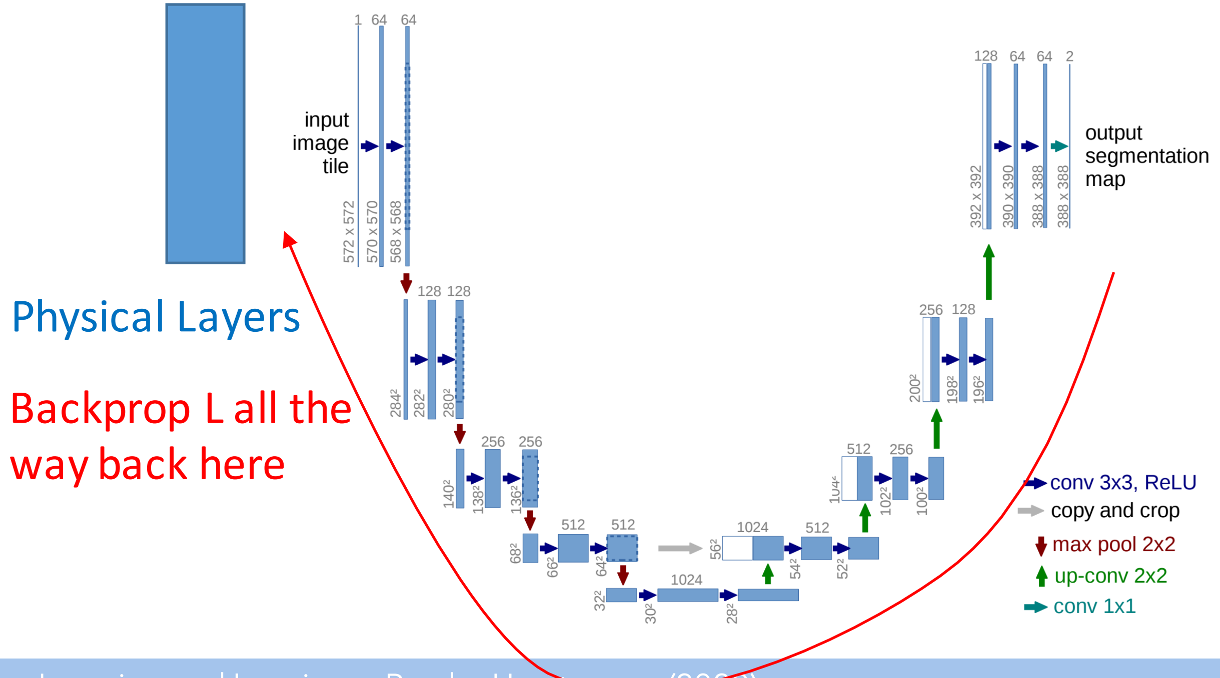
- Most common issue – you have a bug in your CNN!
    - Solution: “Disable “ physical layer (set to constant), and get network to work!
- Another common challenge - vanishing gradients



Physical Layers

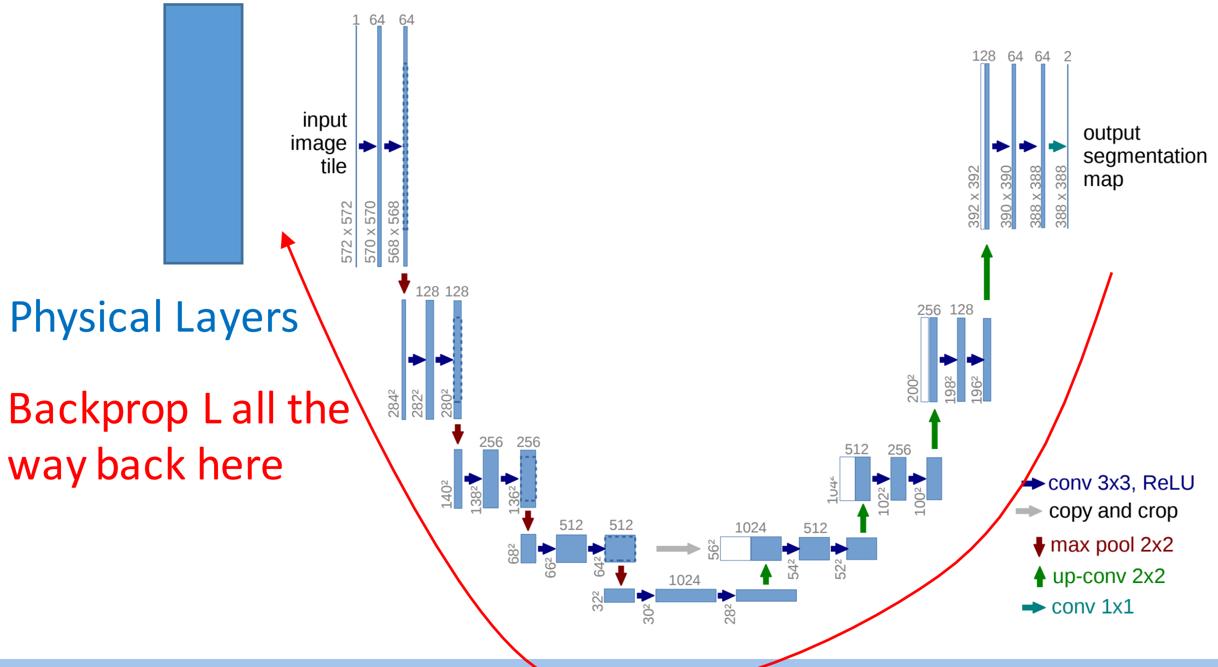
What are some common issues and pitfalls with physical layers?

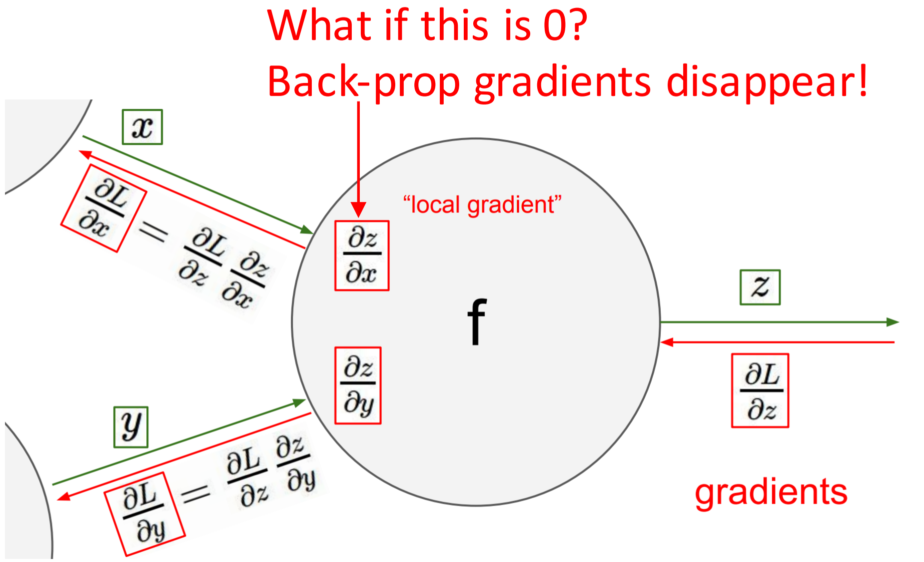
- Most common issue – you have a bug in your CNN!
  - Solution: “Disable “ physical layer (set to constant), and get network to work!
- Another common challenge - vanishing gradients



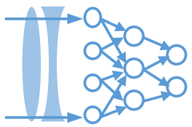
What are some common issues and pitfalls with physical layers?

- Most common issue – you have a bug in your CNN!
    - Solution: “Disable “ physical layer (set to constant), and get network to work!
- Another common challenge - vanishing gradients



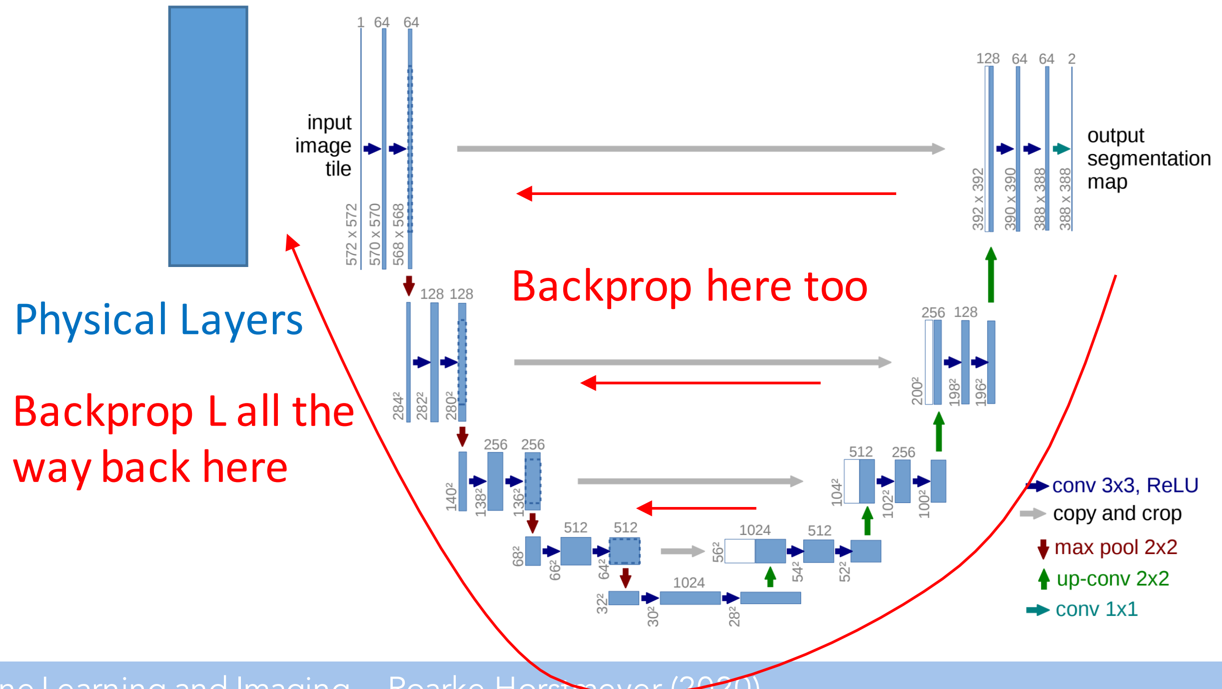


From Stanford CS231n



What are some common issues and pitfalls with physical layers?

- Most common issue – you have a bug in your CNN!
    - Solution: “Disable “ physical layer (set to constant), and get network to work!
- Another common challenge - vanishing gradients

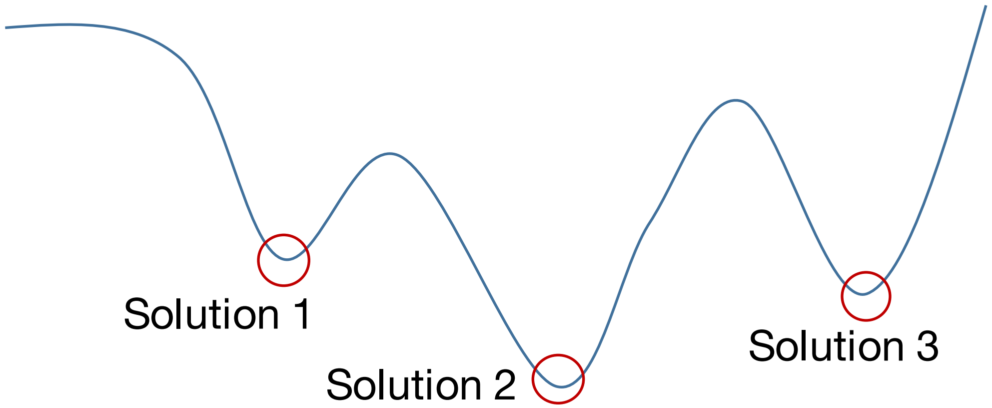


Solution: Introduce skipped connections

deep imaging

What are some common issues and pitfalls with physical layers?

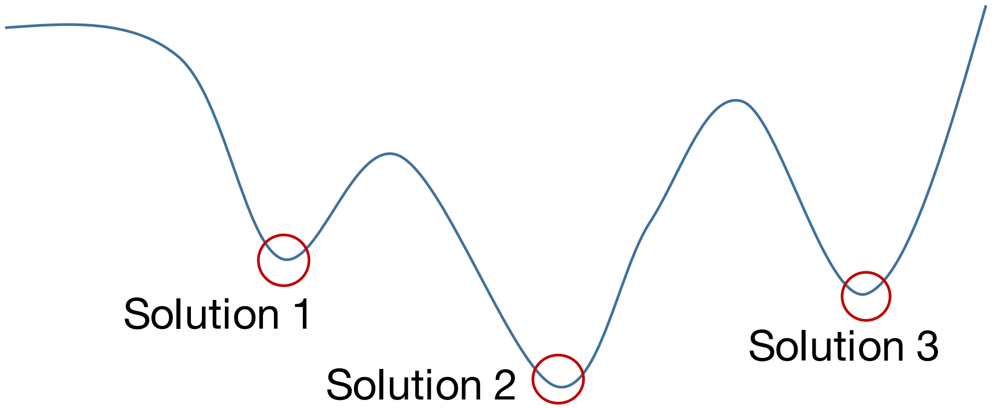
- Most common issue – you have a bug in your CNN!
  - Solution: “Disable “ physical layer (set to constant), and get network to work!
- Another common challenge - vanishing gradients
- Third issue - physical layer results are not very repeatable...



deep imaging

What are some common issues and pitfalls with physical layers?

- Most common issue – you have a bug in your CNN!
  - Solution: “Disable “ physical layer (set to constant), and get network to work!
- Another common challenge - vanishing gradients
- Third issue - physical layer results are not very repeatable…



Effective Solution: Add a small amount of noise to the physical layer output:

$$I_s = \Sigma\, w_j I_j + n$$

(tf.keras.layers.GaussianNoise)

deep imaging

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

(Typically easy)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

(Typically hard)

deep imaging

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

(Typically easy)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

(Typically hard)

What I did in grad school to get ready for an experiment:

Simulated "Real-world" objects → Forward model → data → Inverse problem solver → Simulated "Real-world" objects

Noise, perturbations, etc. → Forward model

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

(Typically easy)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

(Typically hard)

What I did in grad school to get ready for an experiment:

Simulated "Real-world" objects → Forward model → data → Inverse problem solver → Simulated "Real-world" objects

Known, easy (Forward model)

Noise, perturbations, etc. → Forward model

Hard to get working… (Inverse problem solver)

But you have insights and guarantees!

deep imaging

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

(Typically easy)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

(Typically hard)

What I did in grad school to get ready for an experiment:

Simulated "Real-world" objects → Forward model (Known, easy) → data → Inverse problem solver (Hard to get working…) → Simulated "Real-world" objects

Classic examples: Inverse Radon Transform, US image reconstruction, image deblurring/denoising, diffraction tomography, phase retrieval, super-resolution (structured illumination, STORM/PALM), etc.

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

(Typically easy)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

(Typically hard)

What you can do now with CNN's:

Simulated "Real-world" objects → Forward model → data → CNN as an inverse model → Simulated "Real-world" objects

Noise, perturbations, etc. → Forward model

deep imaging

Aside on simulated data: Combining forward and inverse solvers

Forward problem: Start with the causes (objects in the real world) and compute the results (captured data)

(Typically easy)

Inverse problem: Start with the results (captured data) and infer about the causes (objects in the real world)

(Typically hard)

What you can do now with CNN's:

Simulated "Real-world" objects → Forward model → data → CNN as an inverse model → Simulated "Real-world" objects

Known, easy

Noise, perturbations, etc.

Also easy….

But it's a black box!