UVA DATA SCIENCE

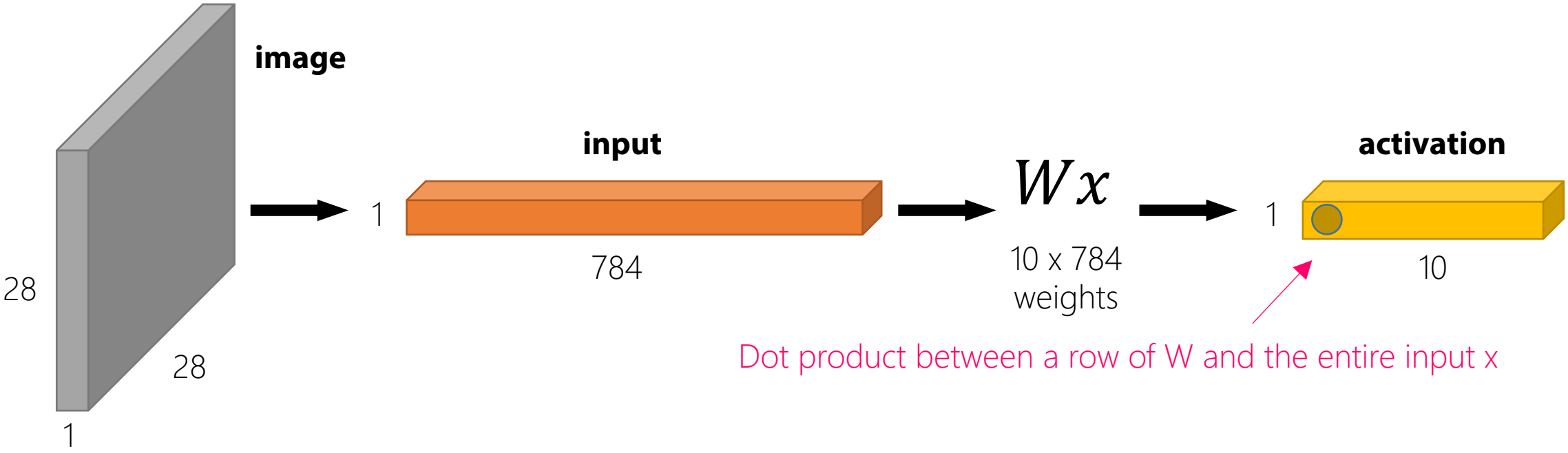
DS 6050 Deep Learning

4. How to Train Convolutional Neural Networks Part I

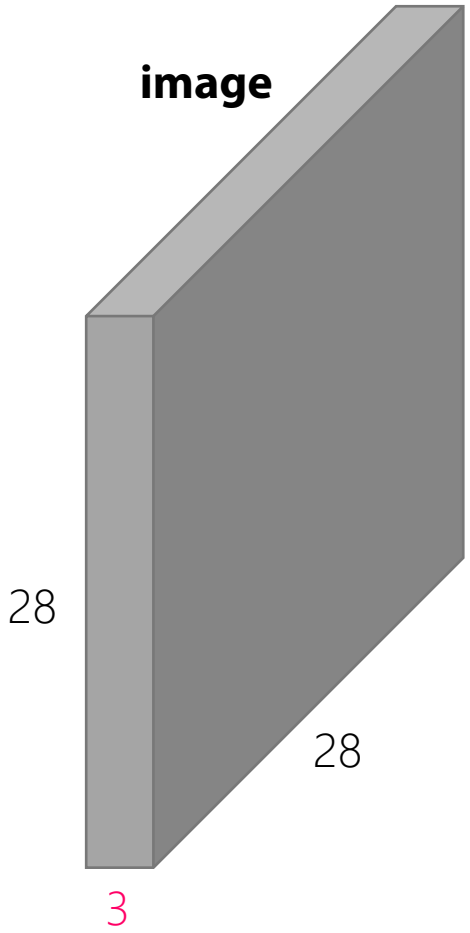
Sheng Li
Associate Professor
School of Data Science
University of Virginia

Recap: Fully Connected Layer

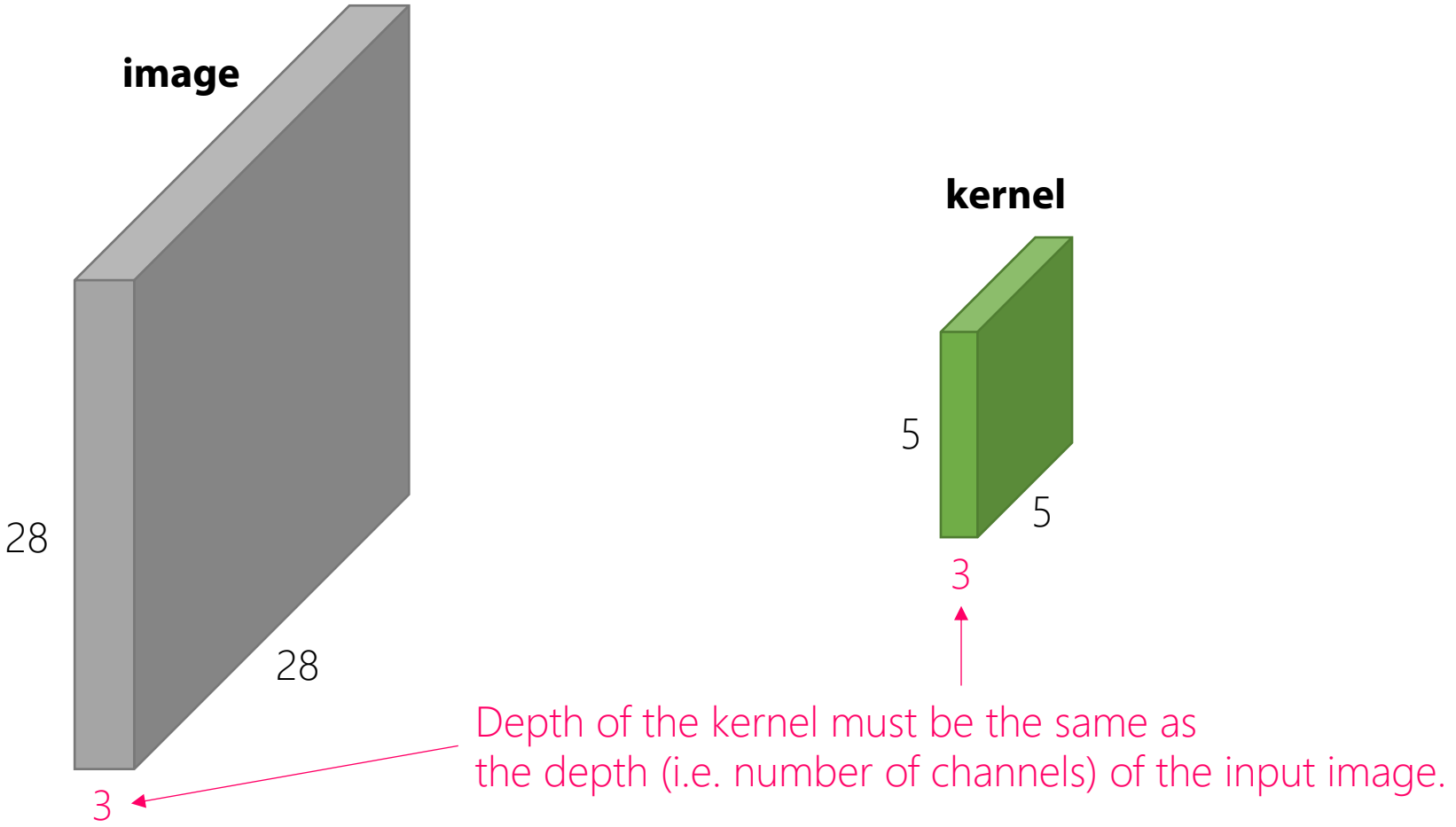
- 28 x 28 image → stretch to 784 x 1
- 64 x 64 x 3 image → stretch to 12288 x 1
- …



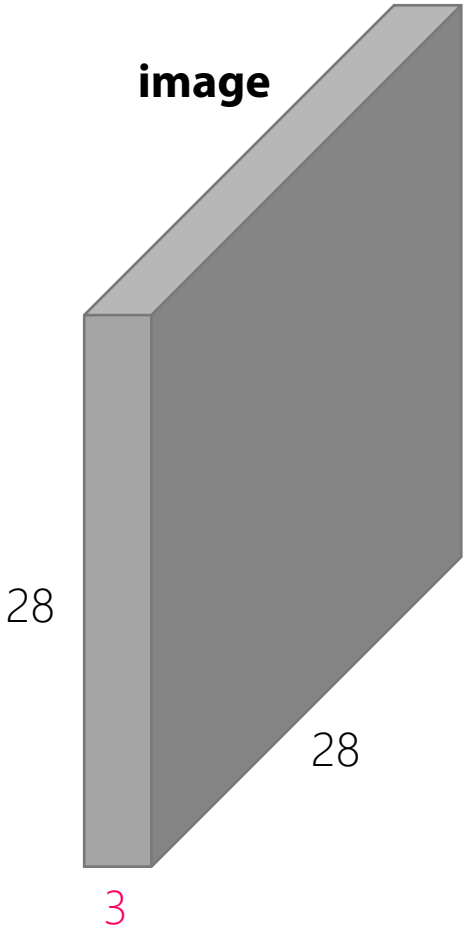
Recap: Convolution Layer

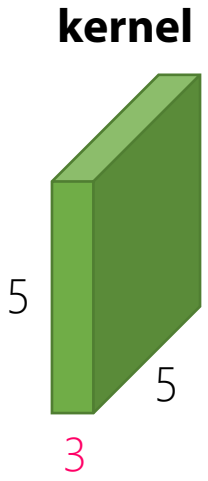


Recap: Convolution Layer



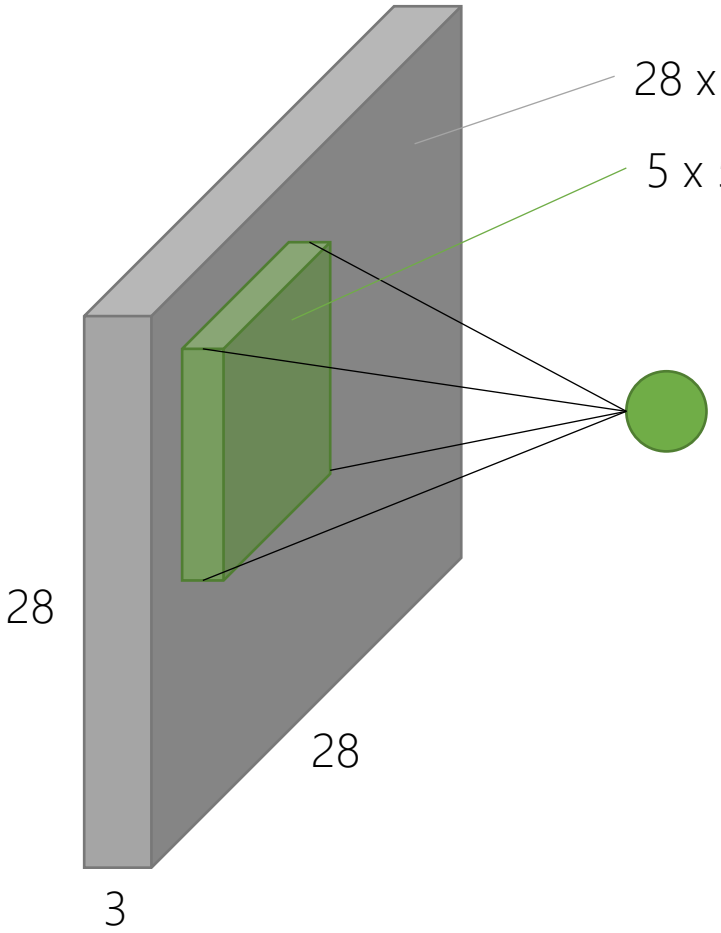
Recap: Convolution Layer





Convolve the kernel with the image!
In other words…
"slide the kernel over the image, compute dot products each time."

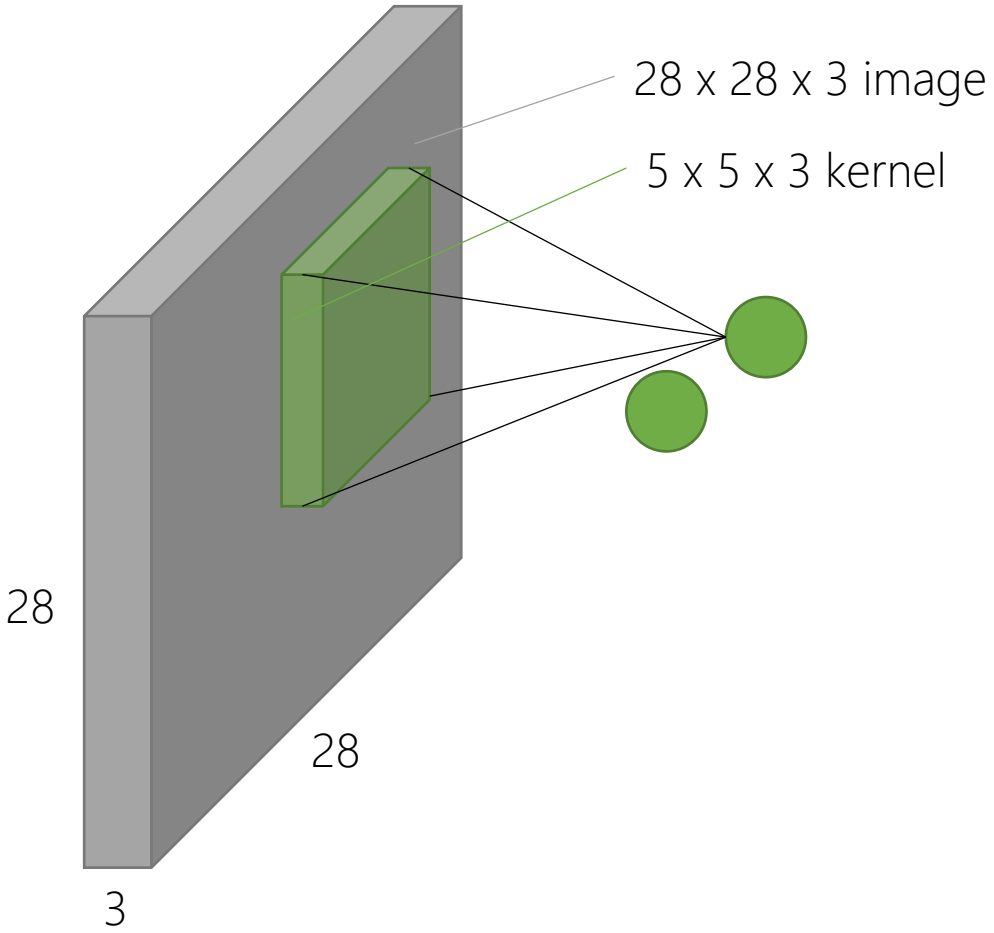
Recap: Convolution Layer



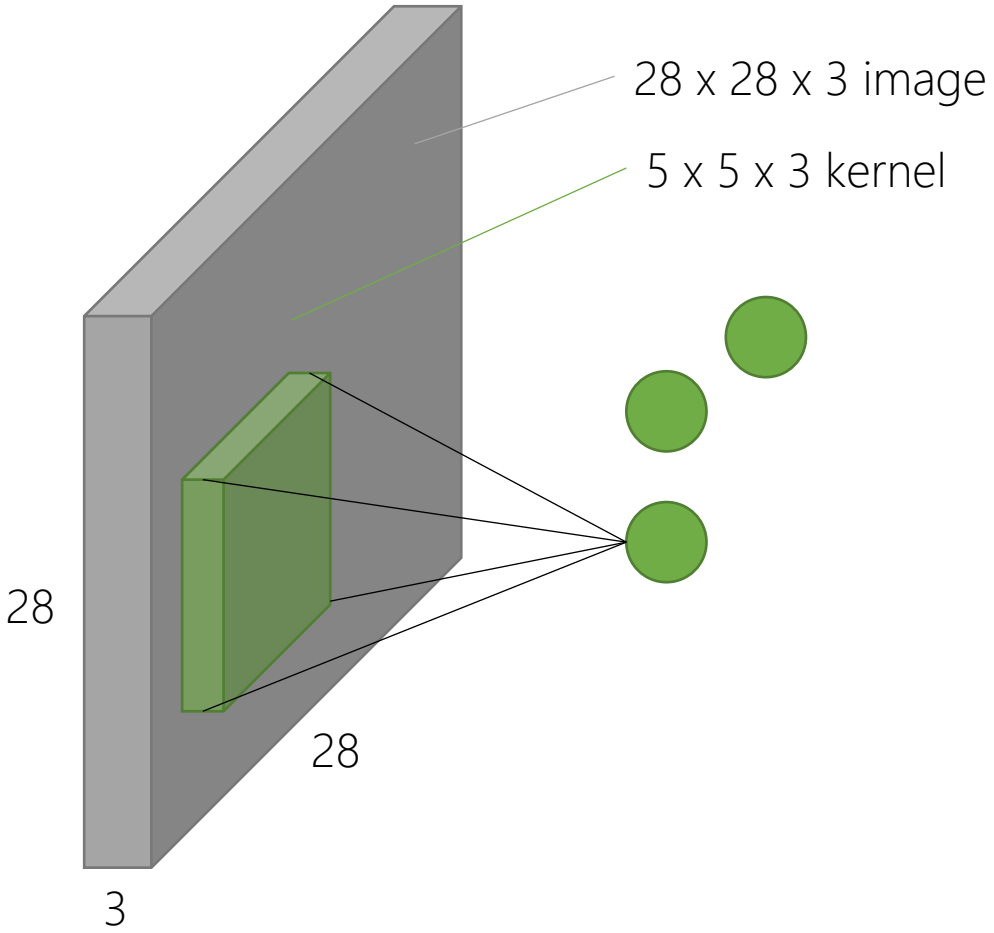
1 number as a result of taking dot product between the kernel and a small 5 x 5 x 3 chunk of the image, plus bias:

$$w^T x + b$$

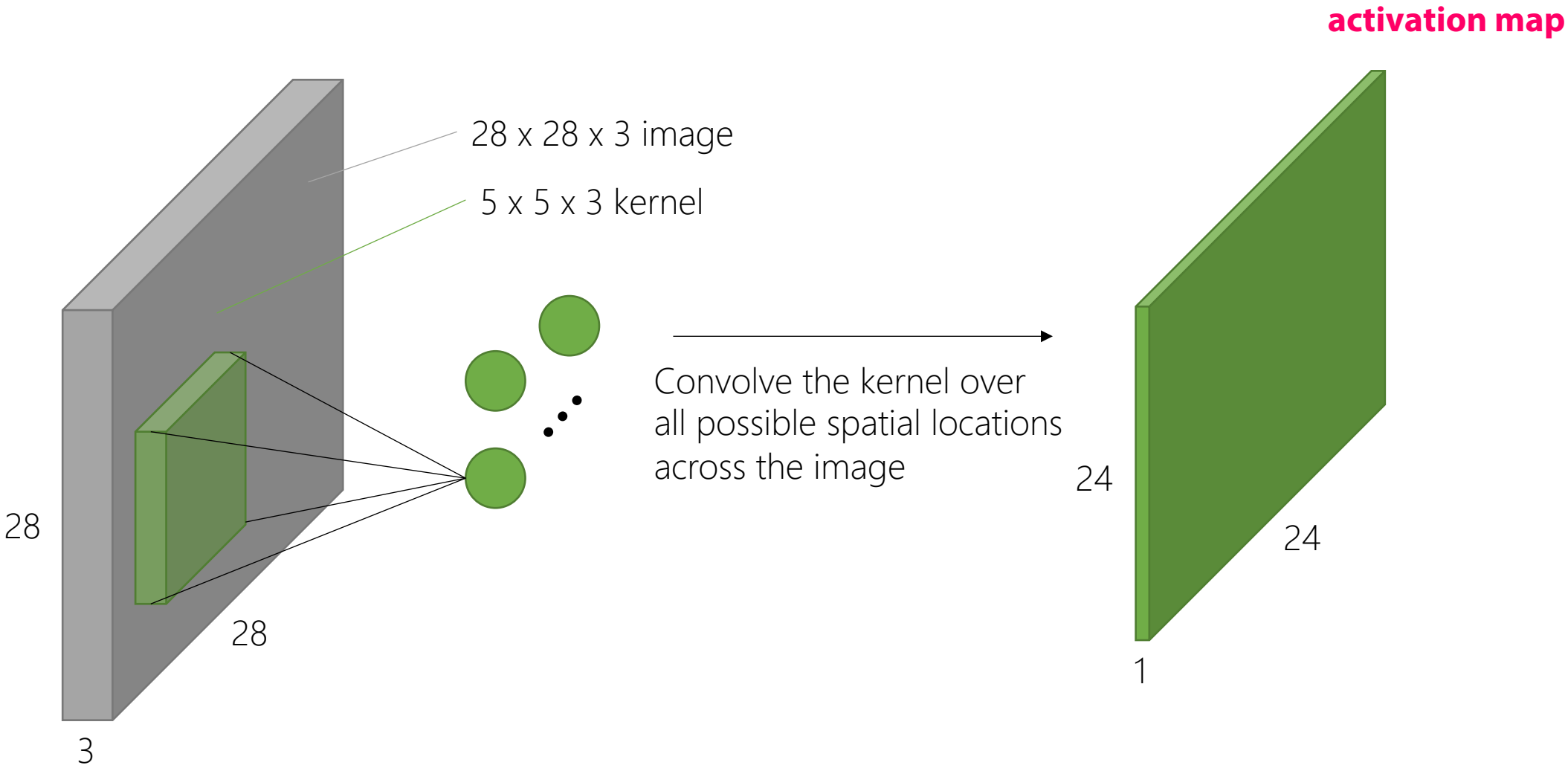
Recap: Convolution Layer



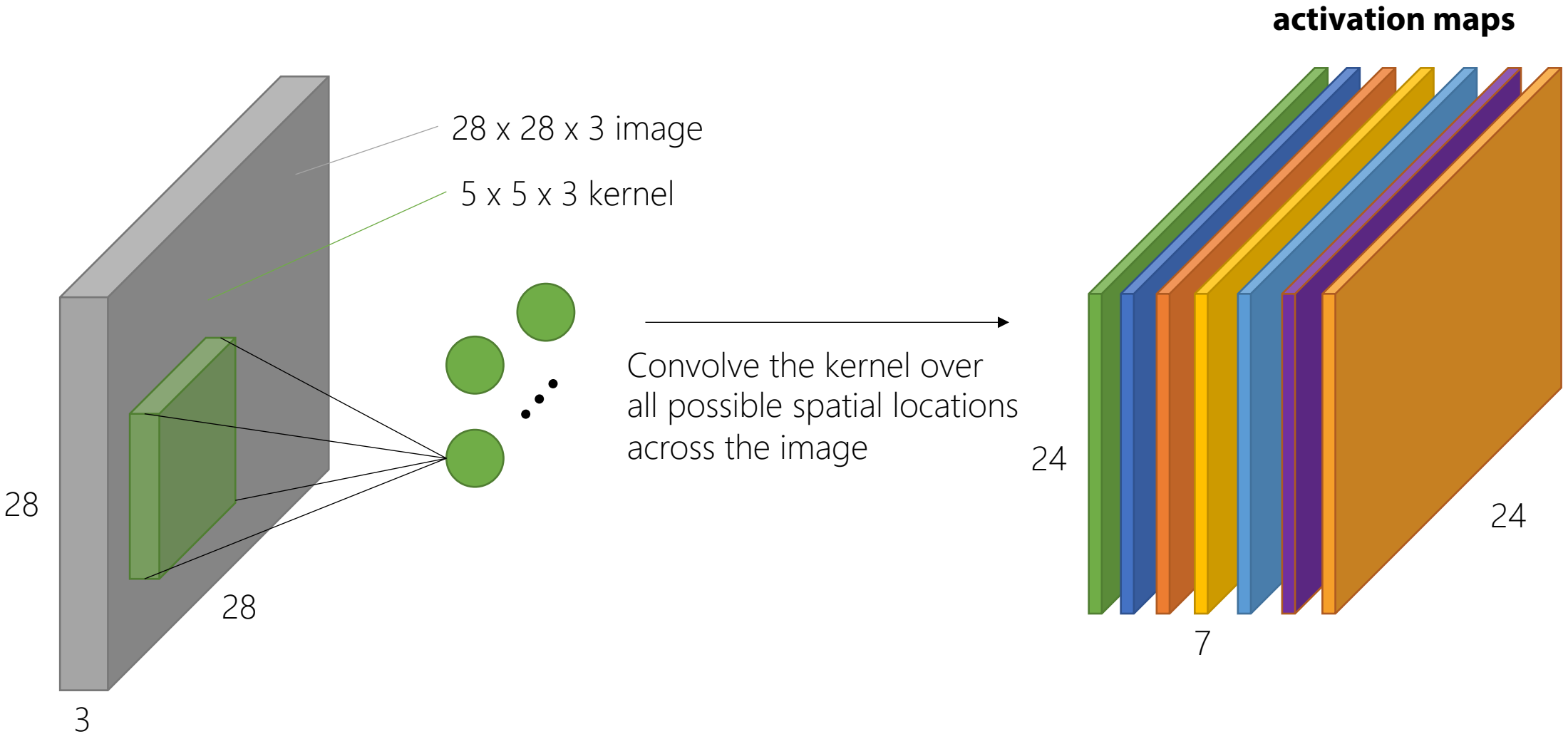
Recap: Convolution Layer



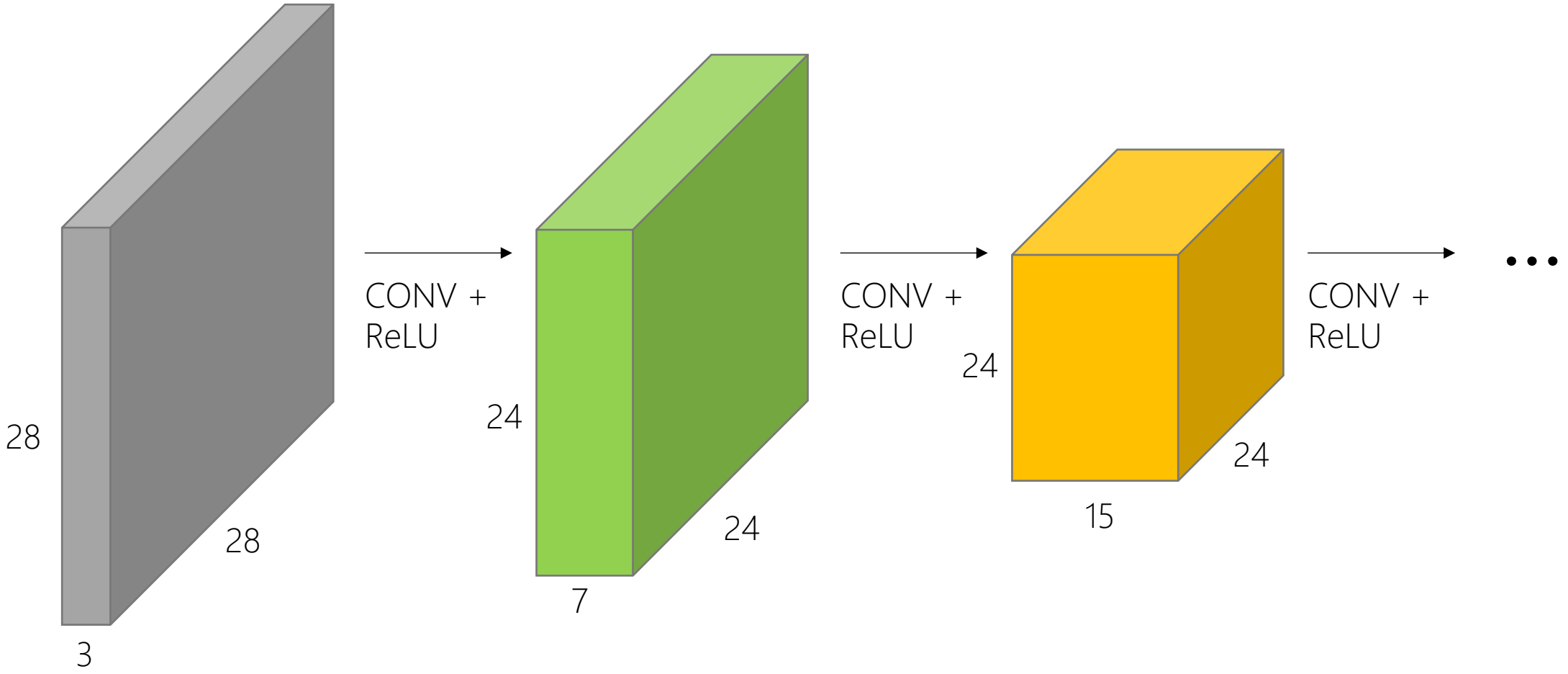
Recap: Convolution Layer



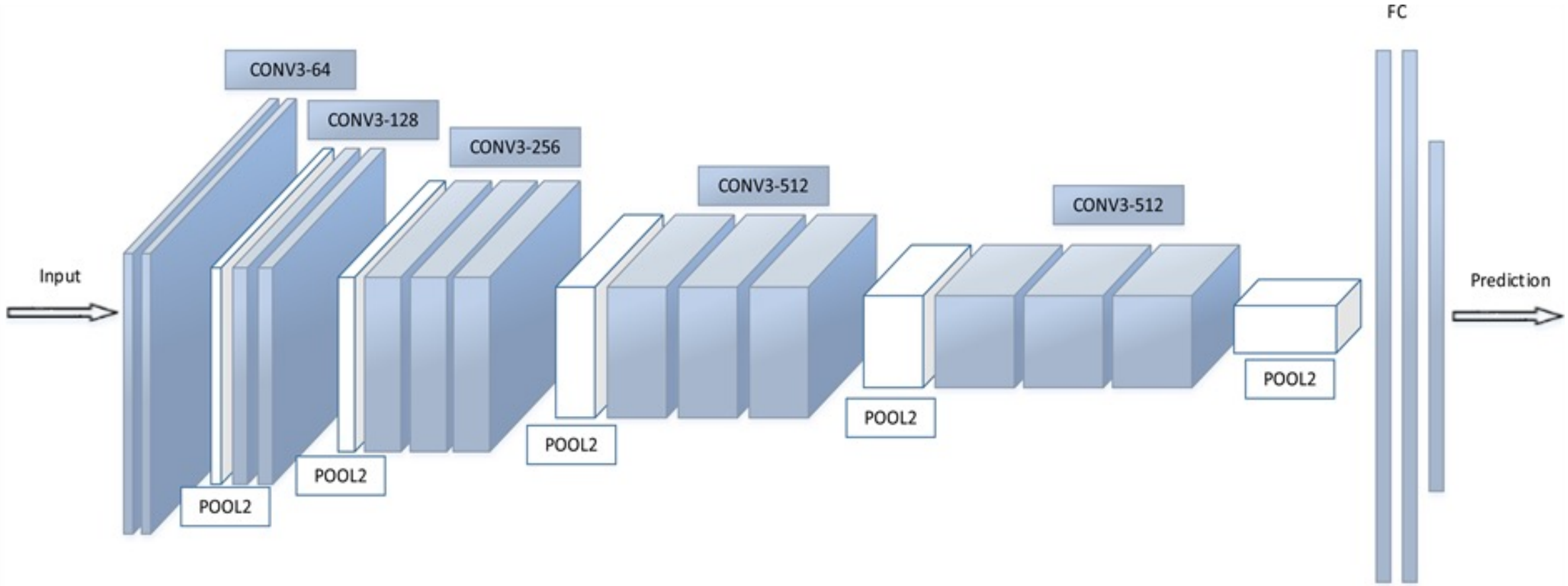
Recap: Convolution Layer



Recap: ConvNet is a sequence of convolution layers



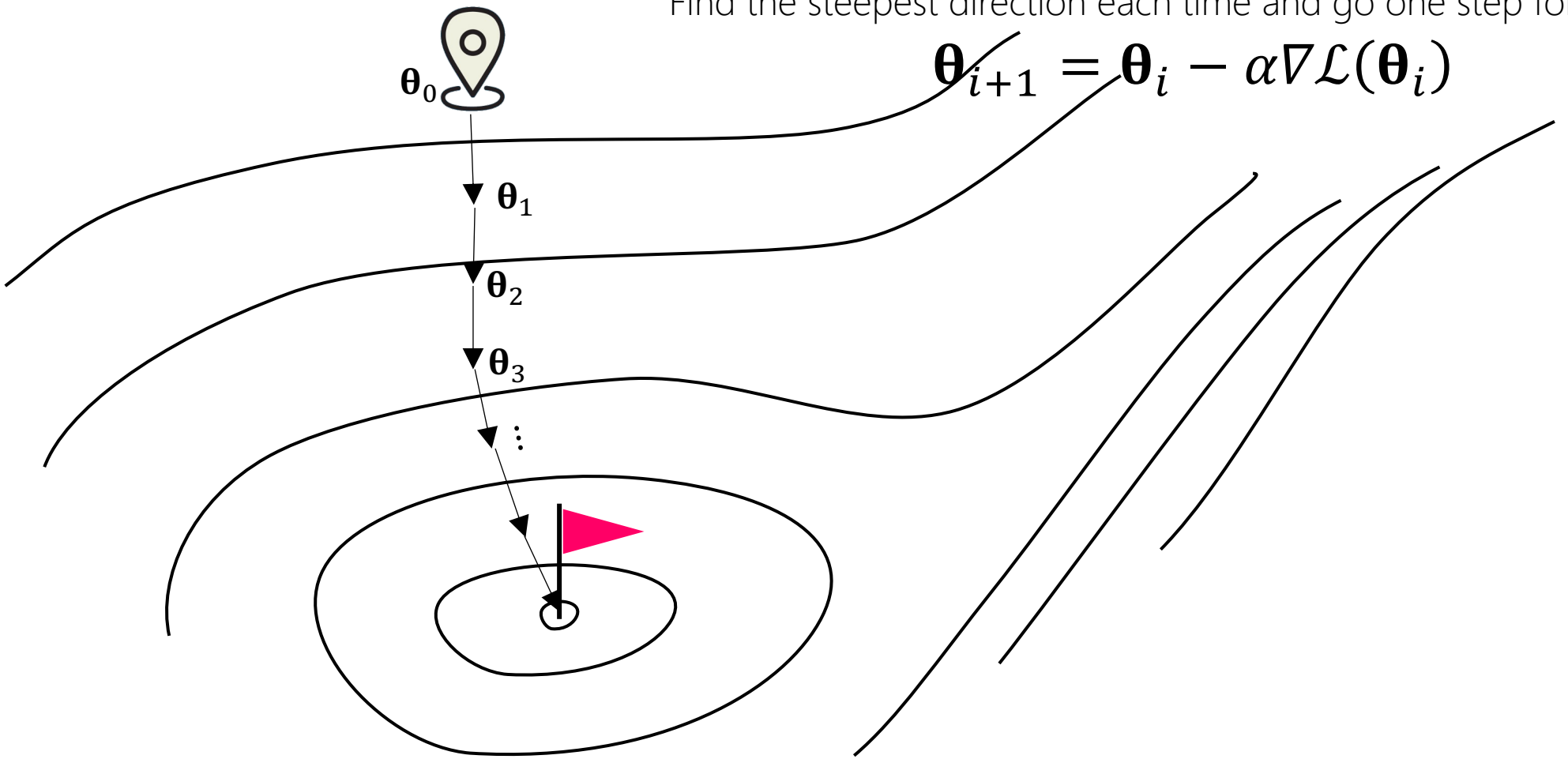
Recap: VGG Networks



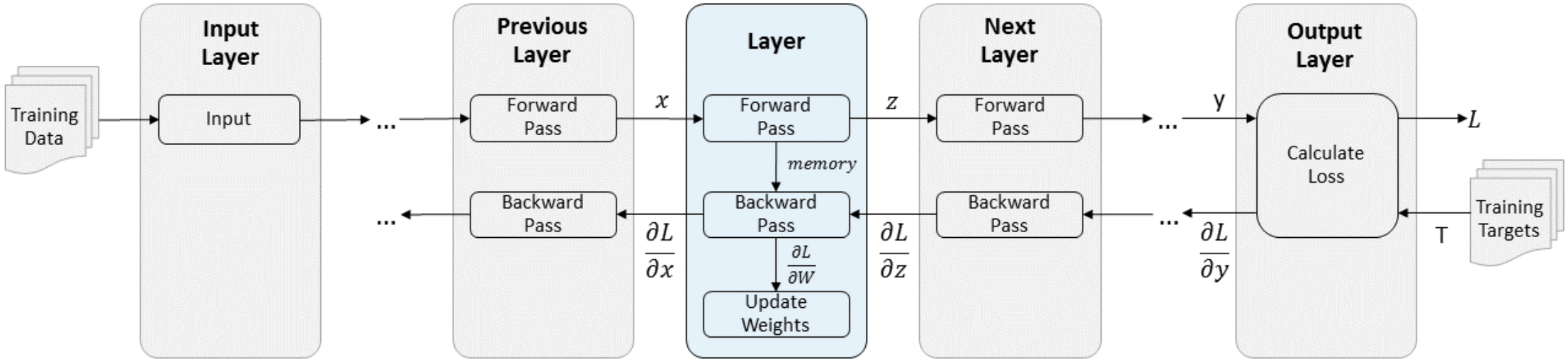
Recap: Steepest (Gradient) Descent

"Find the steepest direction each time and go one step forward."

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \alpha \nabla \mathcal{L}(\boldsymbol{\theta}_i)$$



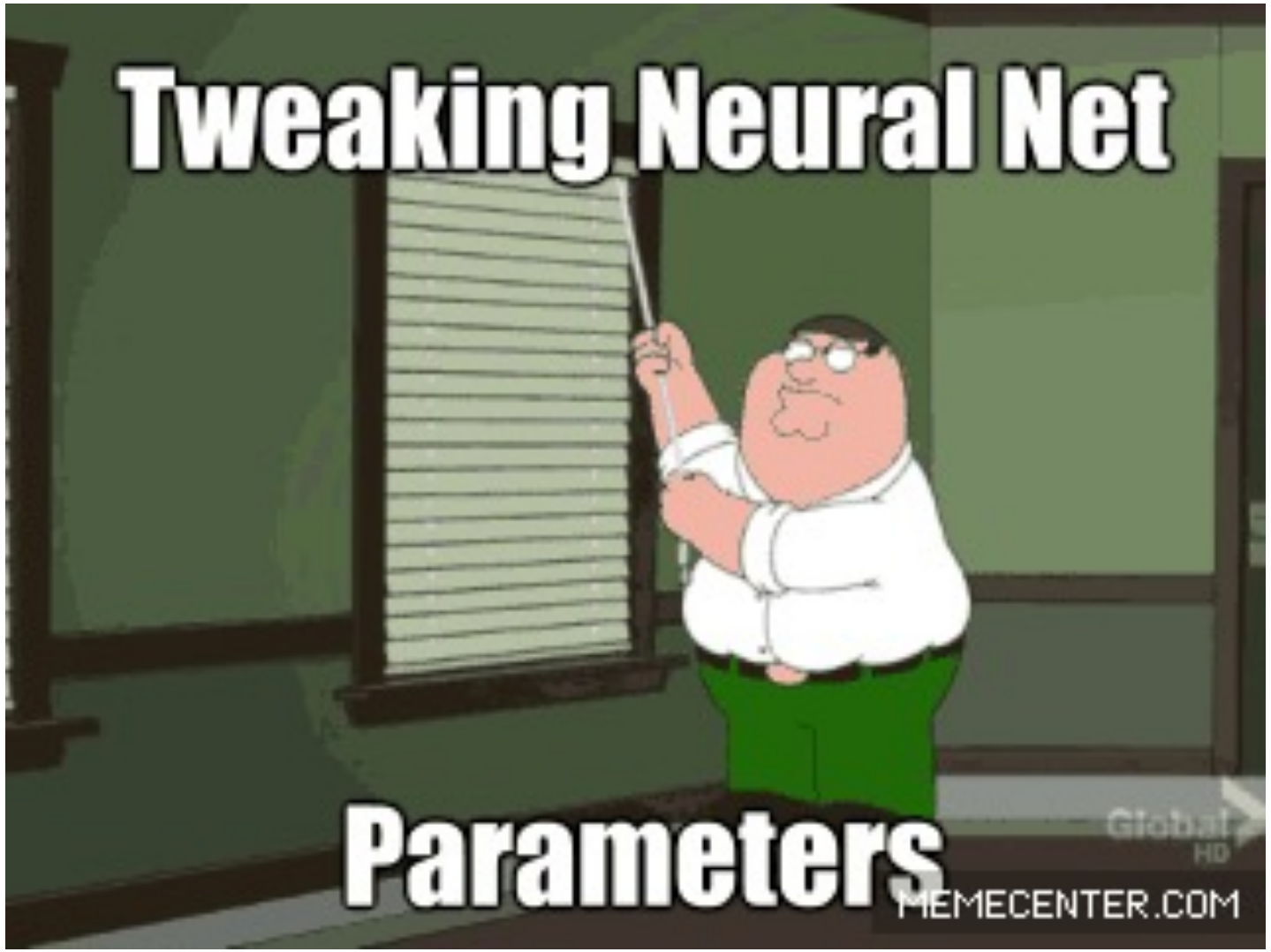
Recap: Backpropagation



https://www.mathworks.com/help/nnet/ug/define-custom-deep-learning-layers.html

Factors Affecting Your Training Result:

- Network architecture
  - How many layers? How many neurons? Kernel size?
  - Skip connection? Inception (network-in-network)?
- Choice of activation functions
  - Sigmoid? Tanh? ReLU? LeakyReLU? PReLU? Others?
- Data preprocessing and sampling (Data distribution)
  - Data mean and standard deviation, minimum, maximum, noise
  - Train-val-test split, batching
  - Data augmentation, synthetic data, etc.
- ConvNet initialization
  - Initial solution for gradient descent (initial neuron weights and biases)
  - Transfer learning
- Network regularization
  - Dropouts, L1/L2 regularizers, early stopping, multi-task learning, ensemble, etc.
- Loss functions
  - MAE? MSE? LogCosh? Huber? Cross-entropy? Kullback-Leibler? Others?
- Optimization methods
  - Gradient descent? Momentum? Nesterov correction? Adaptive gradient? Adam? Nadam?
- So many other parameters to consider


Tweaking Neural Net
Parameters
MEMECENTER.COM

Factors Affecting Your Training Result:

- Network architecture
  - How many layers? How many neurons? Kernel size?
  - Skip connection? Inception (network-in-network)?
- Choice of activation functions
  - Sigmoid? Tanh? ReLU? LeakyReLU? PReLU? Others?
- Data preprocessing and sampling (Data distribution)
  - Data mean and standard deviation, minimum, maximum, noise
  - Train-val-test split, batching
  - Data augmentation, synthetic data, etc.
- ConvNet initialization
  - Initial solution for gradient descent (initial neuron weights and biases)
  - Transfer learning
- Network regularization
  - Dropouts, L1/L2 regularizers, ensemble, etc.
- Loss functions
  - MAE? MSE? LogCosh? Huber? Cross-entropy? Kullback-Leibler? Others?
- Optimization methods
  - Gradient descent? Momentum? Nesterov correction? Adaptive gradient? Adam? Nadam?
- So many other parameters to consider

Module 6 (in two weeks… stay tuned!)

This module!
(theme: "gradients & distributions")

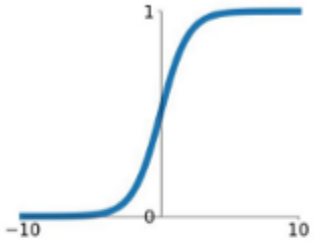
Part II (next week)
(theme: "optimization")

Activation Functions

Recap: Activation Functions

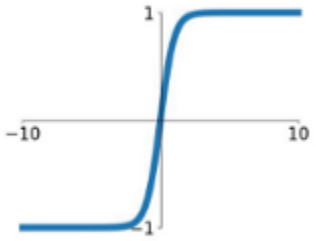
Sigmoid

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



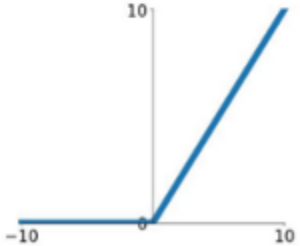
tanh

$$\tanh(x)$$



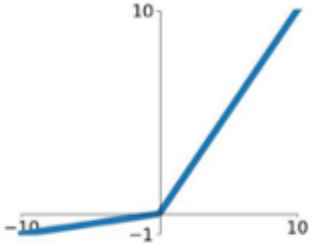
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

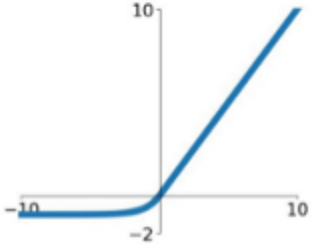


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



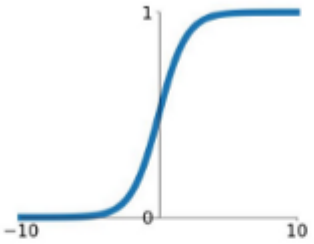
https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

Historically the most popular choice: Sigmoid

- Brain analogy: saturating “firing rate” of a neuron
- Crushes input values to [0, 1].
  - Problems:
    1. Kills off gradients when saturated.
    2. Outputs are always non-negative (not zero-centered).
    3. Exponential! (computationally expensive)

Sigmoid

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

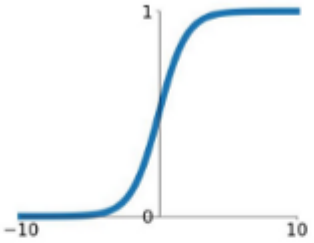


Historically the most popular choice: Sigmoid

- Brain analogy: saturating "firing rate" of a neuron
- Crushes input values to [0, 1].
  - Problems:
    1. Kills off gradients when saturated.
    2. Outputs are always non-negative (not zero-centered).
    3. Exponential! (computationally expensive)

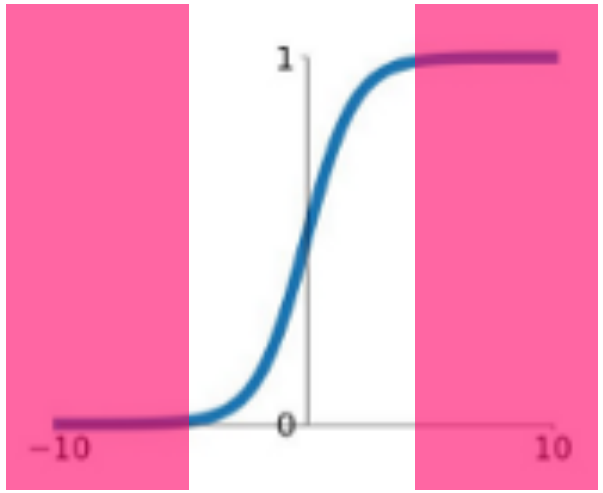
Sigmoid

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

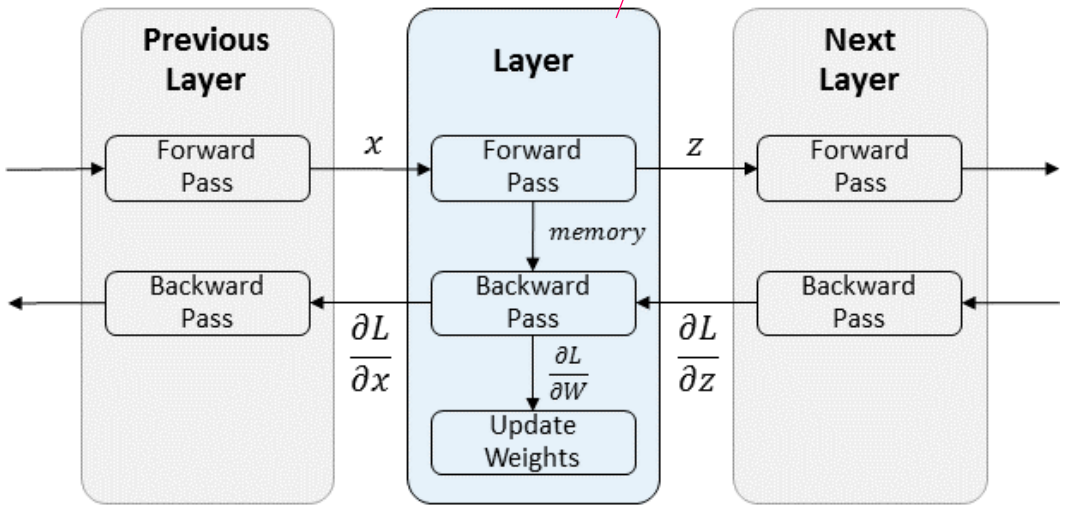


$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial \sigma}\frac{\partial \sigma}{\partial x}$$

$\frac{\partial \sigma}{\partial x}$ → 0



$$\frac{\partial \sigma}{\partial x} = 0$$



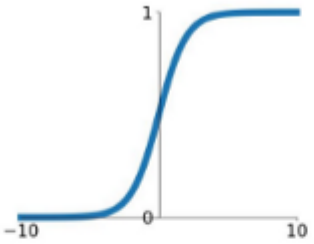
https://www.mathworks.com/help/nnet/ug/define-custom-deep-learning-layers.html

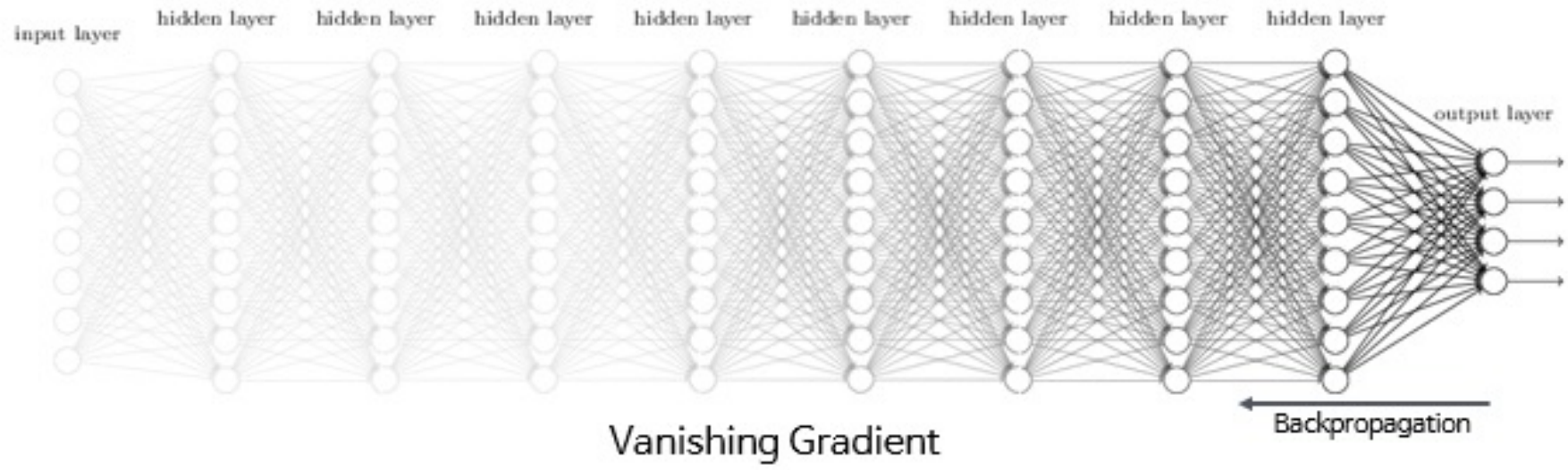
Historically the most popular choice: Sigmoid

- Brain analogy: saturating "firing rate" of a neuron
- Crushes input values to [0, 1].
  - Problems:
    1. Kills off gradients when saturated.
    2. Outputs are always non-negative (not zero-centered).
    3. Exponential! (computationally expensive)

Sigmoid

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



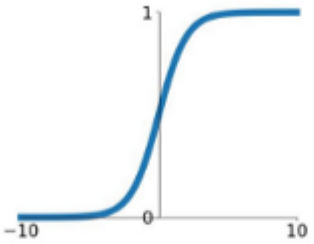


Historically the most popular choice: Sigmoid

- Brain analogy: saturating "firing rate" of a neuron
- Crushes input values to [0, 1].
  - Problems:
    1. Kills off gradients when saturated.
    2. Outputs are always non-negative (not zero-centered).
    3. Exponential! (computationally expensive)

Sigmoid

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

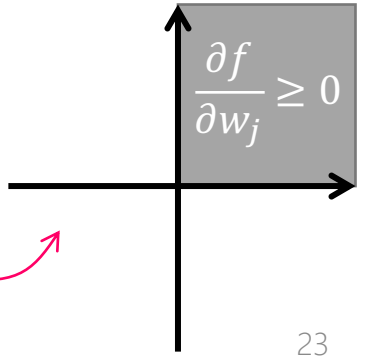


Q. What happens to the gradient if all the input to a neuron is always non-negative? i.e., $x_j \geq 0, \ \forall x_j$

$$f(x \mid \mathbf{w}, b) = \sum_j w_j x_j + b$$

$$\frac{\partial f}{\partial w_j} = x_j \geq 0 \ \ \forall j \quad \Rightarrow \quad \mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{L}}{\partial f} \frac{\partial f}{\partial \mathbf{w}}$$

scalar vector (gradient)

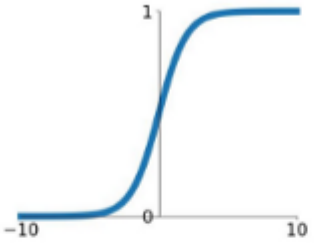


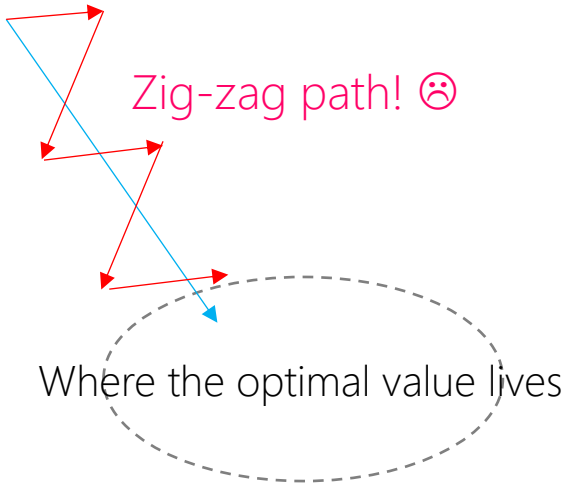
Historically the most popular choice: Sigmoid

- Brain analogy: saturating "firing rate" of a neuron
- Crushes input values to [0, 1].
  - Problems:
    1. Kills off gradients when saturated.
    2. Outputs are always non-negative (not zero-centered).
    3. Exponential! (computationally expensive)

Sigmoid

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



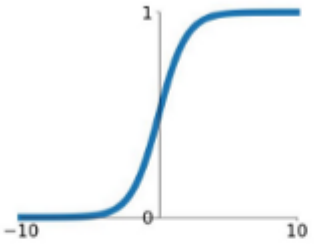


Historically the most popular choice: Sigmoid

- Brain analogy: saturating "firing rate" of a neuron
- Crushes input values to [0, 1].
  - Problems:
    1. Kills off gradients when saturated.
    2. Outputs are always non-negative (not zero-centered).
    3. Exponential! (computationally expensive)

Sigmoid

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



Operation	Input	Output	Algorithm	Complexity
Addition	Two *n*-digit numbers *N*, *N*	One *n*+1-digit number	Schoolbook addition with carry	$\Theta(n)$, $\Theta(\log(N))$
Subtraction	Two *n*-digit numbers *N*, *N*	One *n*+1-digit number	Schoolbook subtraction with borrow	$\Theta(n)$, $\Theta(\log(N))$
Multiplication	Two *n*-digit numbers	One 2*n*-digit number	Schoolbook long multiplication	$O(n^2)$
			Karatsuba algorithm	$O(n^{1.585})$
			3-way Toom–Cook multiplication	$O(n^{1.465})$
			k-way Toom–Cook multiplication	$O(n^{\log(2k-1)/\log k})$
			Mixed-level Toom–Cook (Knuth 4.3.3-T)[2]	$O(n\, 2^{\sqrt{2\log n}} \log n)$
			Schönhage–Strassen algorithm	$O(n \log n \log\log n)$
			Fürer's algorithm[3]	$O(n \log n\, 2^{O(\log^* n)})$
Division	Two *n*-digit numbers	One *n*-digit number	Schoolbook long division	$O(n^2)$
			Newton–Raphson division	$O(M(n))$
Square root	One *n*-digit number	One *n*-digit number	Newton's method	$O(M(n))$
Modular exponentiation	Two *n*-digit numbers and a *k*-bit exponent	One *n*-digit number	Repeated multiplication and reduction	$O(M(n)\, 2^k)$
			Exponentiation by squaring	$O(M(n)\, k)$
			Exponentiation with Montgomery reduction	$O(M(n)\, k)$

https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

Hyperbolic Tangent (LeCun et al. 1991)

- "Rescaled sigmoid"

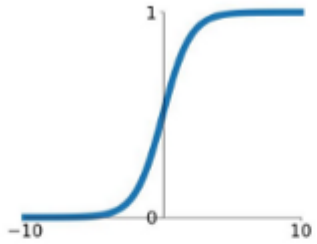
$$2\frac{1}{1+e^{-x}} - \frac{(1+e^{-x})}{1+e^{-x}} = \frac{1-e^{-x}}{1+e^{-x}}$$

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

- Now it's zero-centered ☺
  - but still…
    - Kills off gradients when saturated.
    - Exponential! (computationally expensive)

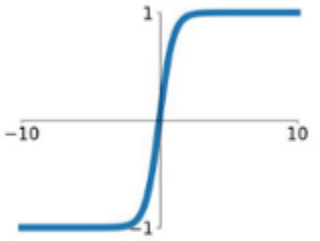
Sigmoid

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



tanh

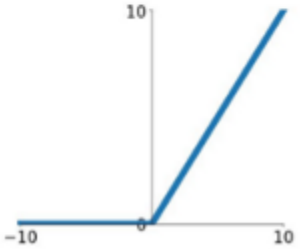
$$\tanh(x)$$



Rectified Linear Unit (Krizhevsky et al., 2012)

ReLU

$$\max(0, x)$$



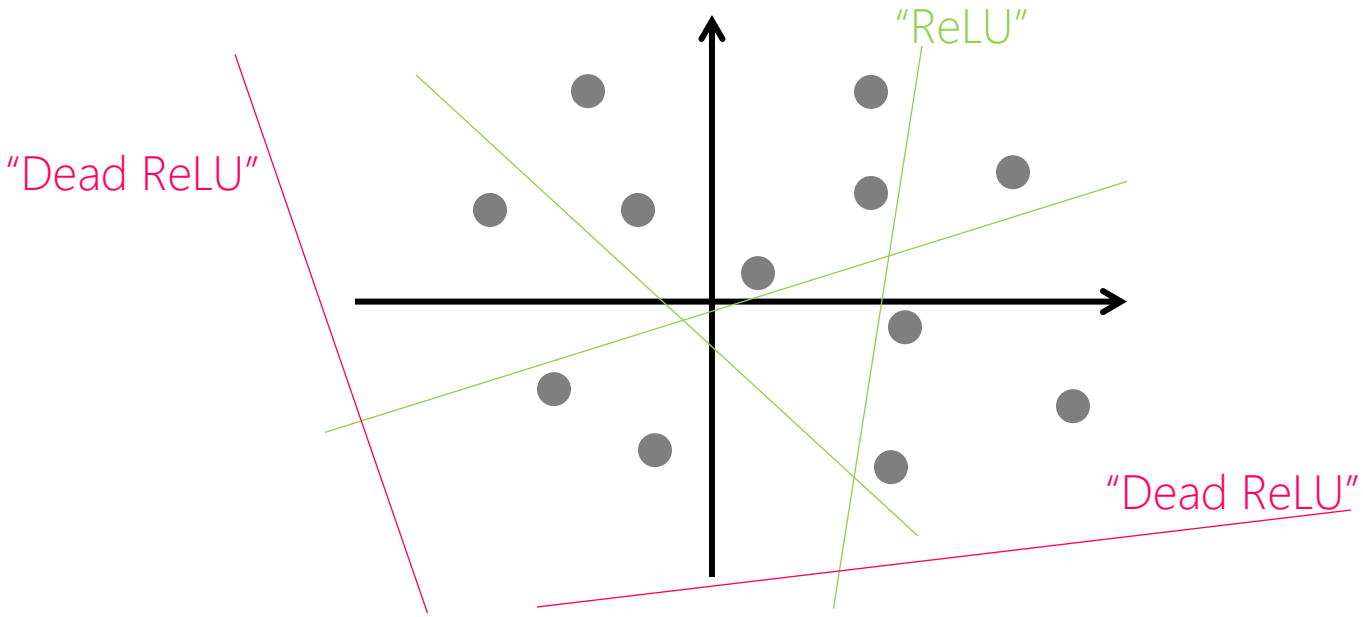
- No saturation when $x > 0$ ☺
- Computationally super efficient. ☺
- Actually, biologically more plausible (modern neuroscience) ☺

- In practice, converges much faster than sigmoid or tanh. ☺

- Not zero-centered though… ☹
- Zero gradient when $x < 0$ → "Dead ReLU" neurons ☹

Rectified Linear Unit (Krizhevsky et al., 2012)

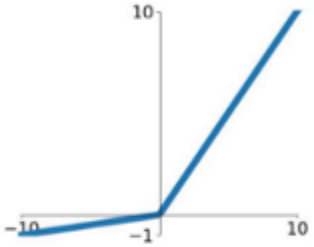
- Dead ReLU
  - A "dead" ReLU always outputs zero for any input.
  - This happens when a large negative bias term is developed.
  - In turn, the dead ReLU neuron will not take any role in the network.
  - “Decision plane” outside the data space.



“Leaky” ReLU (Mass et al., 2013)

- No saturation
- No dead ReLU situation

Leaky ReLU
$\max(0.1x, x)$



- Leaky ReLU

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

  - α is set by the user

- Parametric Leaky ReLU (PReLU) (He et al., 2015)

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

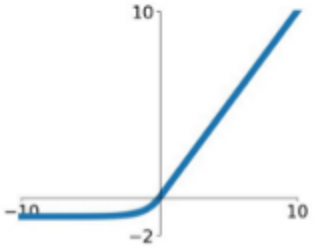
  - α is learned from data (backprop)

Exponential Linear Units (Clevert et al., 2015)

- All the benefits of ReLU, plus…
  - Close to zero-mean output
  - Negative saturation → robustness to noise than Leaky ReLUs

ELU

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



- … but at expenses of computing exponential… ☹

Maxout Neurons (Goodfellow et al. 2013)

- Prepare two different sets of weights.
- Output the max of these two.
- Generalizes ReLU and Leaky ReLU.
- Linear, non-saturate, no dead neurons.

- … however, doubles the number of parameters… ☹

Maxout

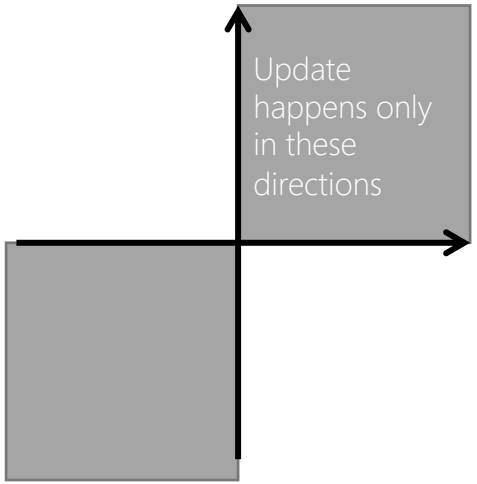
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

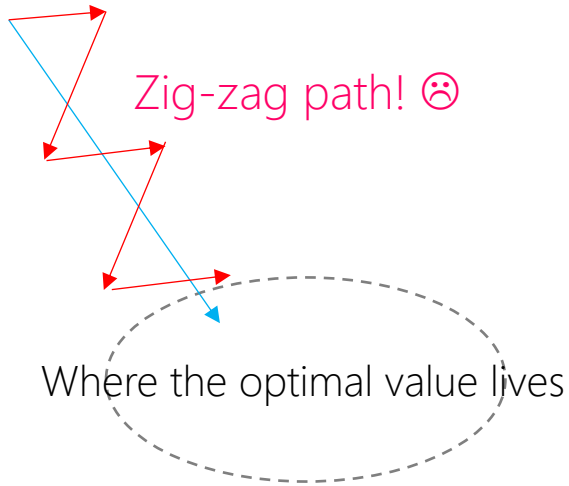
TL;DR

- “Never” use sigmoid
- tanh is good for some rare cases.
- ReLU is always the good starting point.
- Try out Leaky ReLU, PReLU, ELU, and maxout. See if they give any better result.

Data Normalization & Whitening

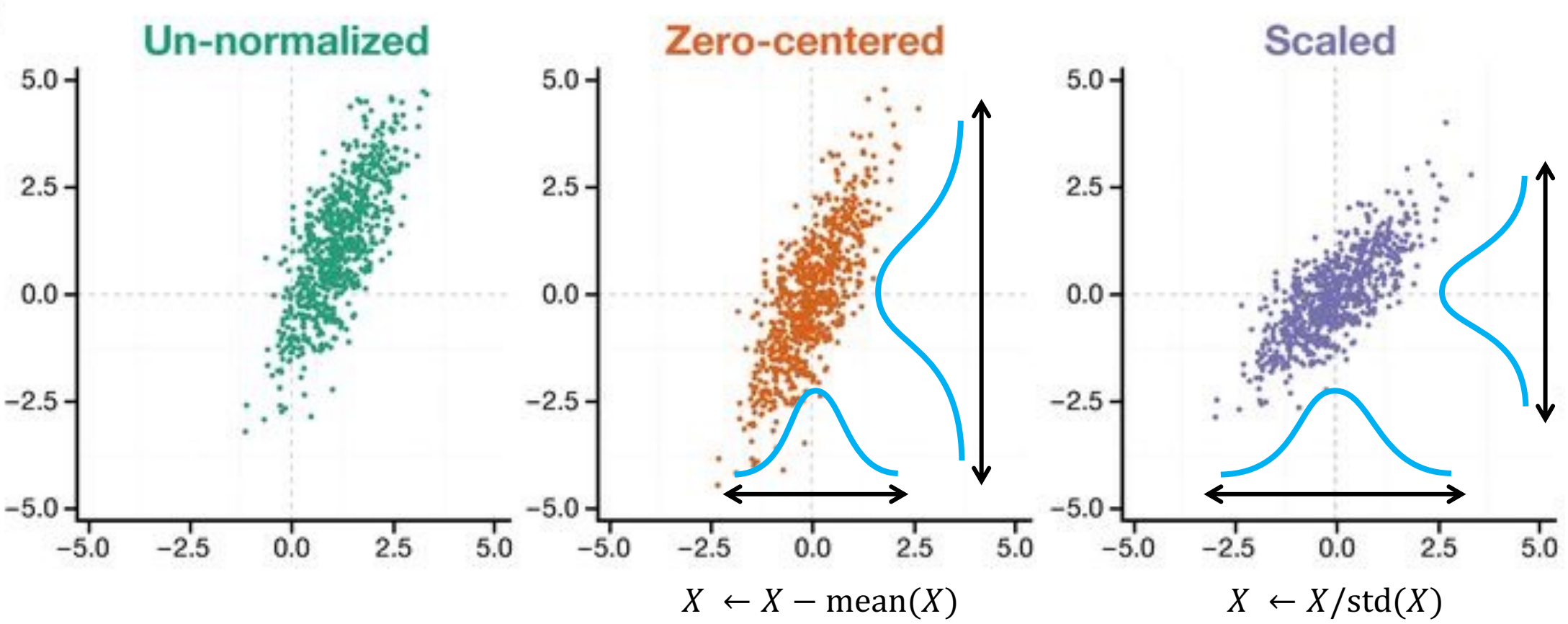
Recall: when inputs to a neuron ≥ 0





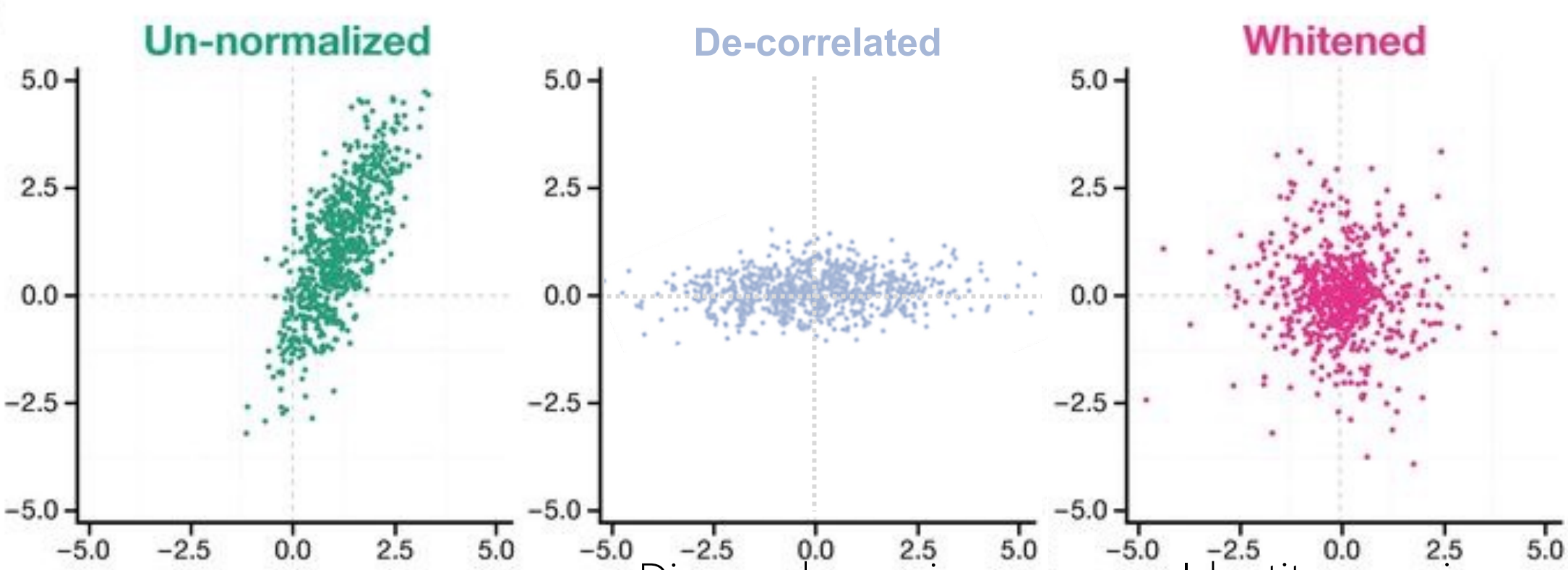
- This also means you need zero-centered data!

Standard Normalization



Angermueller et al. (2016). Deep learning for computational biology. Molecular Systems Biology 12:878

PCA Whitening



Angermueller et al. (2016). Deep learning for computational biology. Molecular Systems Biology 12:878

In practice

- Images:
  - Not so common to do whitening (but some people do).
  - Subtract the mean image (mean across the entire dataset).
  - Subtract the mean color (mean across an image, per channel).

- Medical Images:
  - Occasionally, each image is whitened per patient.
  - Entire Hounsfield unit (HU) is rescaled/readjusted to the range (-1, 1).

- There's no golden rule or anything. But data normalization is nearly a MUST. Zero-centering can already help a lot.

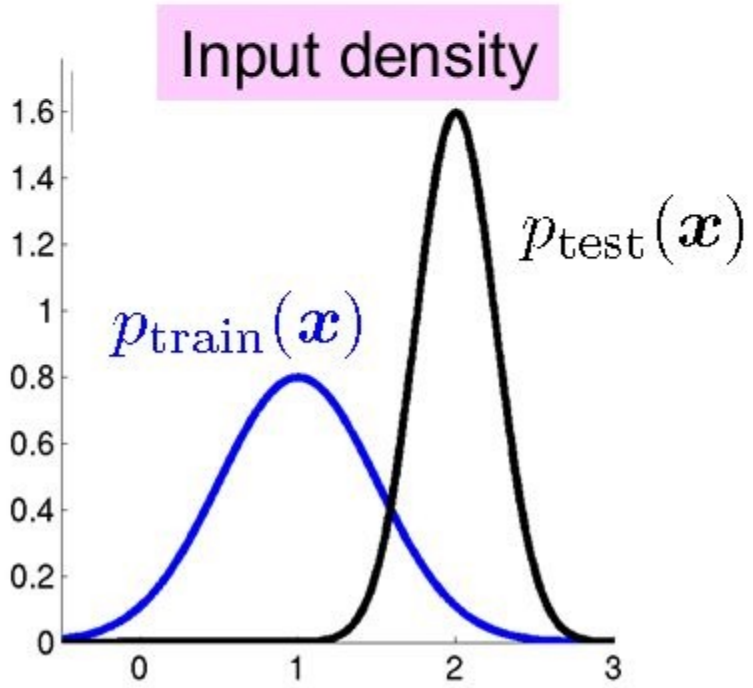
Batch Normalization

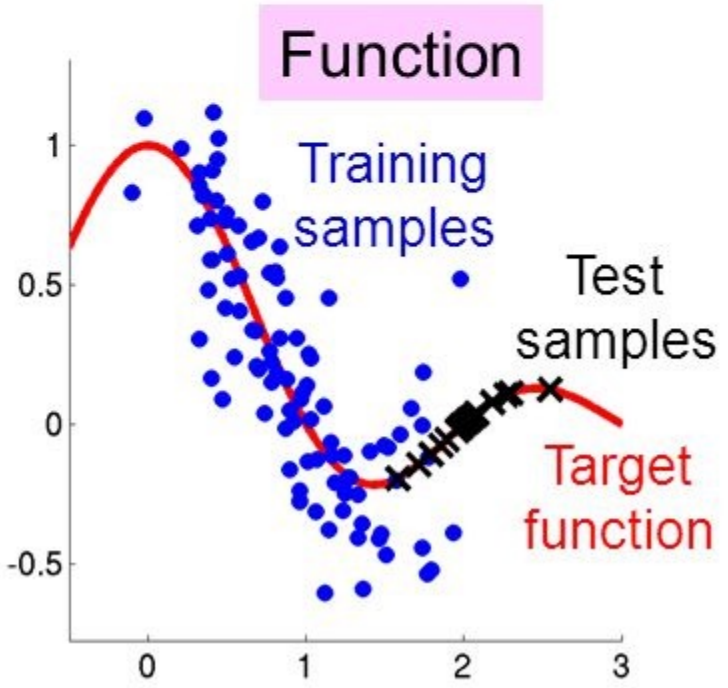
- So, we just learned that the input layer can benefit from normalization.
- Why not do the same thing also for the hidden layers?

- Ioffe and Szegedy. (2015) https://arxiv.org/pdf/1502.03167.pdf

Covariate Shift

- Test samples have different distribution than training.
- Bit of “extrapolation” is required.





Batch Normalization

- Idea: Improve training by mitigating internal covariate shift.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

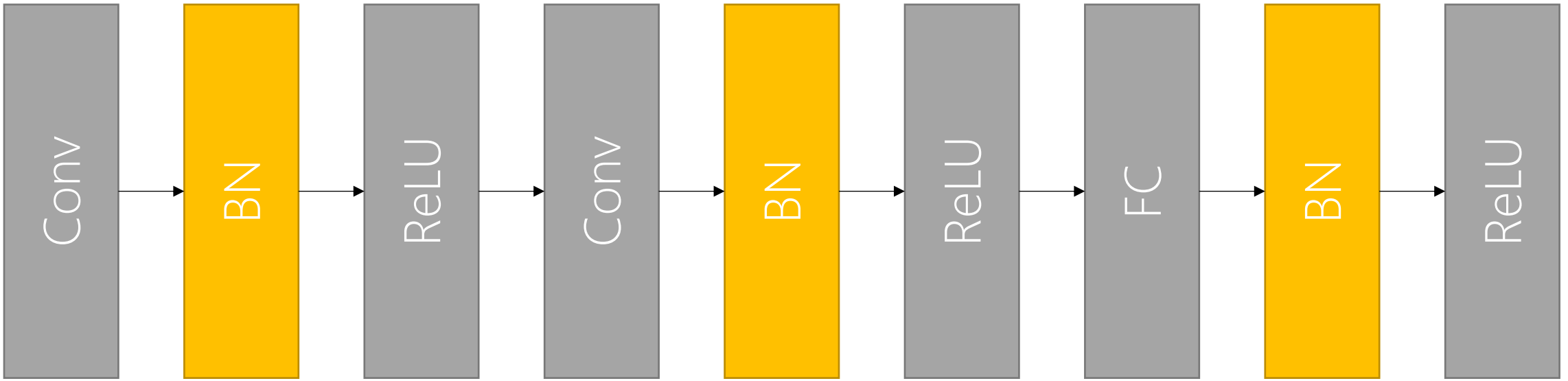
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Ioffe and Szegedy (2015).

Batch Normalization

- Batch normalization layers are added typically after FC/Conv and before nonlinearity (activation).



Batch Normalization (Advantages)

- Allows each layer of a network to learn by itself a little bit more independently of other layers.
- Allows higher learning rates because BN makes sure that there's no neuron that goes extremely high or extremely low (i.e. no gradient explosion).
- Enables stable training at larger batch sizes (Bjorck et al., 2018; De & Smith, 2020)
- Noise statistics among batch → Slight regularization effect (Hoffer et al., 2017; Luo et al., 2019).
- Smoothens the loss landscape (Santurkar et al., 2018).
- Becomes less dependent to initialization

Reading (Optional):
- Bjorck et al. 2018: https://arxiv.org/pdf/1806.02375.pdf
- De & Smith 2020: https://arxiv.org/pdf/2002.10444.pdf
- Hoffer et al. 2017: https://arxiv.org/pdf/1705.08741.pdf
- Luo et al. 2019: https://arxiv.org/pdf/1809.00846.pdf
- Santurkar et al. 2018: https://proceedings.neurips.cc/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf

Batch Normalization (Disadvantages)

- Surprisingly expensive computationally. Incurs memory overhead (Rota Bulo et al., 2018)
- Significantly increases the time required to evaluate the gradient in some networks (Gitman & Ginsburg, 2017)
- Introduces a discrepancy between the behavior of the model during training and at inference time (Summers & Dinneen, 2019; Singh & Shrivastava, 2019)

Reading (Optional):

- Rota Bulo et al. 2018: https://openaccess.thecvf.com/content_cvpr_2018/papers/Bulo_In-Place_Activated_BatchNorm_CVPR_2018_paper.pdf
- Gitman & Ginsburg 2017: https://arxiv.org/pdf/1709.08145.pdf
- Summers & Dinneen 2019: https://arxiv.org/pdf/1906.03548.pdf
- Sing & Shrivastava 2019: https://arxiv.org/pdf/1904.06031.pdf

Batch Normalization (Disadvantages)

- Breaks the independence among training examples in the minibatch (Brock et al. 2021)
  - Batch normalized networks are often difficult to replicate precisely on different hardware
  - For some loss functions, interaction between training examples enables the network to 'cheat' (Chen et al., 2020; He et al., 2020)
  - Performance of BN is sensitive to the batch size. Networks can perform poorly when the batch size is too small (Hoffer et al., 2017; Ioffe, 2017; Wu & He, 2018)

Reading (Optional):

- Brock et al. 2021: https://arxiv.org/pdf/2102.06171
- Chen et al. 2020: https://arxiv.org/pdf/2002.05709.pdf
- He et al. 2020: https://openaccess.thecvf.com/content_CVPR_2020/papers/He_Momentum_Contrast_for_Unsupervised_Visual_Representation_Learning_CVPR_2020_paper.pdf
- Hoffer et al. 2017: https://arxiv.org/pdf/1705.08741.pdf
- Ioffe 2017: https://arxiv.org/pdf/1702.03275.pdf
- Wu & He 2018: https://openaccess.thecvf.com/content_ECCV_2018/papers/Yuxin_Wu_Group_Normalization_ECCV_2018_paper.pdf

Alternative Strategies

- Use better regularization (Zhang et al. (2019a) and De & Smith (2020))
- Weight Standardization (Huang et al., 2017; Qiao et al., 2019)

$$\hat{W}_{ij} = \frac{W_{ij} - \mu_i}{\sqrt{N}\sigma_i}$$

- Gradient Clipping (Pascanu et al., 2013)

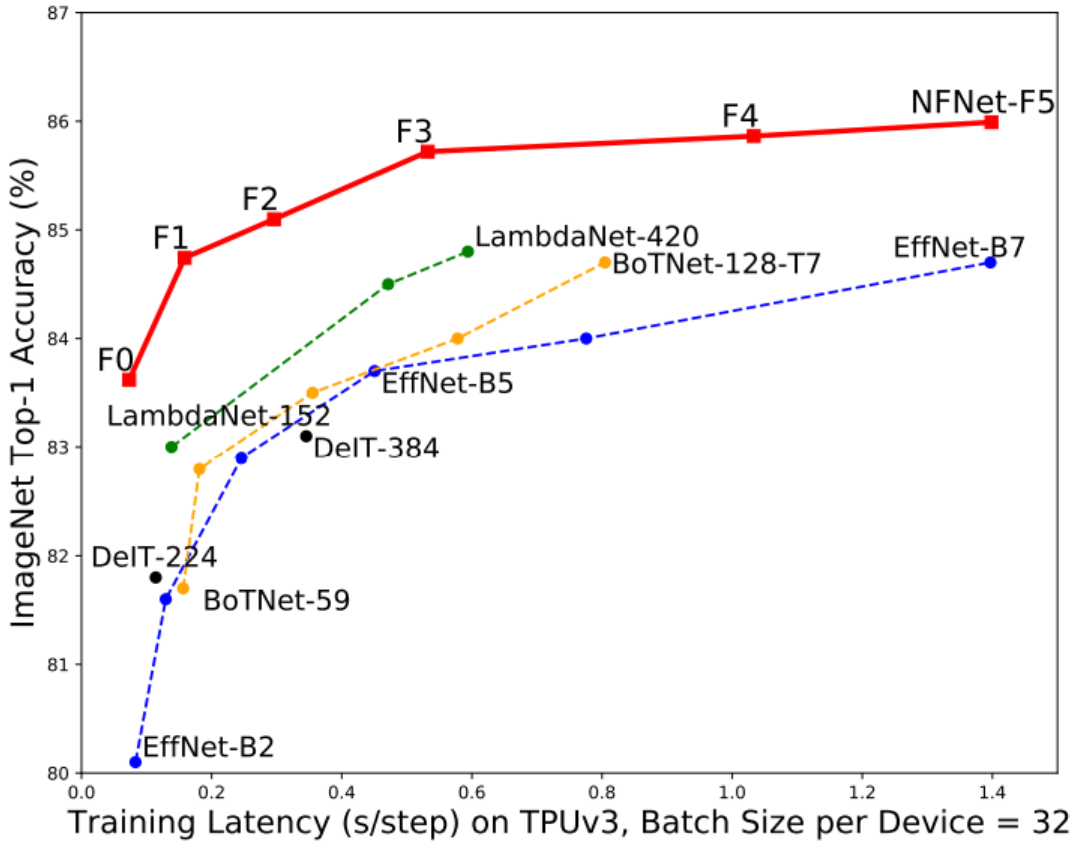
$$G \to \begin{cases} \lambda \frac{G}{\|G\|} & \text{if } \|G\| > \lambda \\ G & \text{otherwise.} \end{cases}$$

Zhang, H., Dauphin, Y. N., and Ma, T. Fixup initialization: Residual learning without normalization. arXiv preprint arXiv:1901.09321, 2019a
De, S. and Smith, S. Batch normalization biases residual blocks towards the identity function in deep networks. Advances in Neural Information Processing Systems, 33, 2020..

Normalization Free Networks (NFNet, 2021)

- Recent DeepMind paper (Brock et al. 2021)
  - Adaptive Gradient Clipping

$$G_i^\ell \rightarrow \begin{cases} \lambda \frac{\|W_i^\ell\|_F^\star}{\|G_i^\ell\|_F} G_i^\ell & \text{if } \frac{\|G_i^\ell\|_F}{\|W_i^\ell\|_F^\star} > \lambda, \\ G_i^\ell & \text{otherwise.} \end{cases}$$



Initializing Weights

Weight Initialization

- Simplest way: W = 0 (or some constant)
  - What happens in this case?

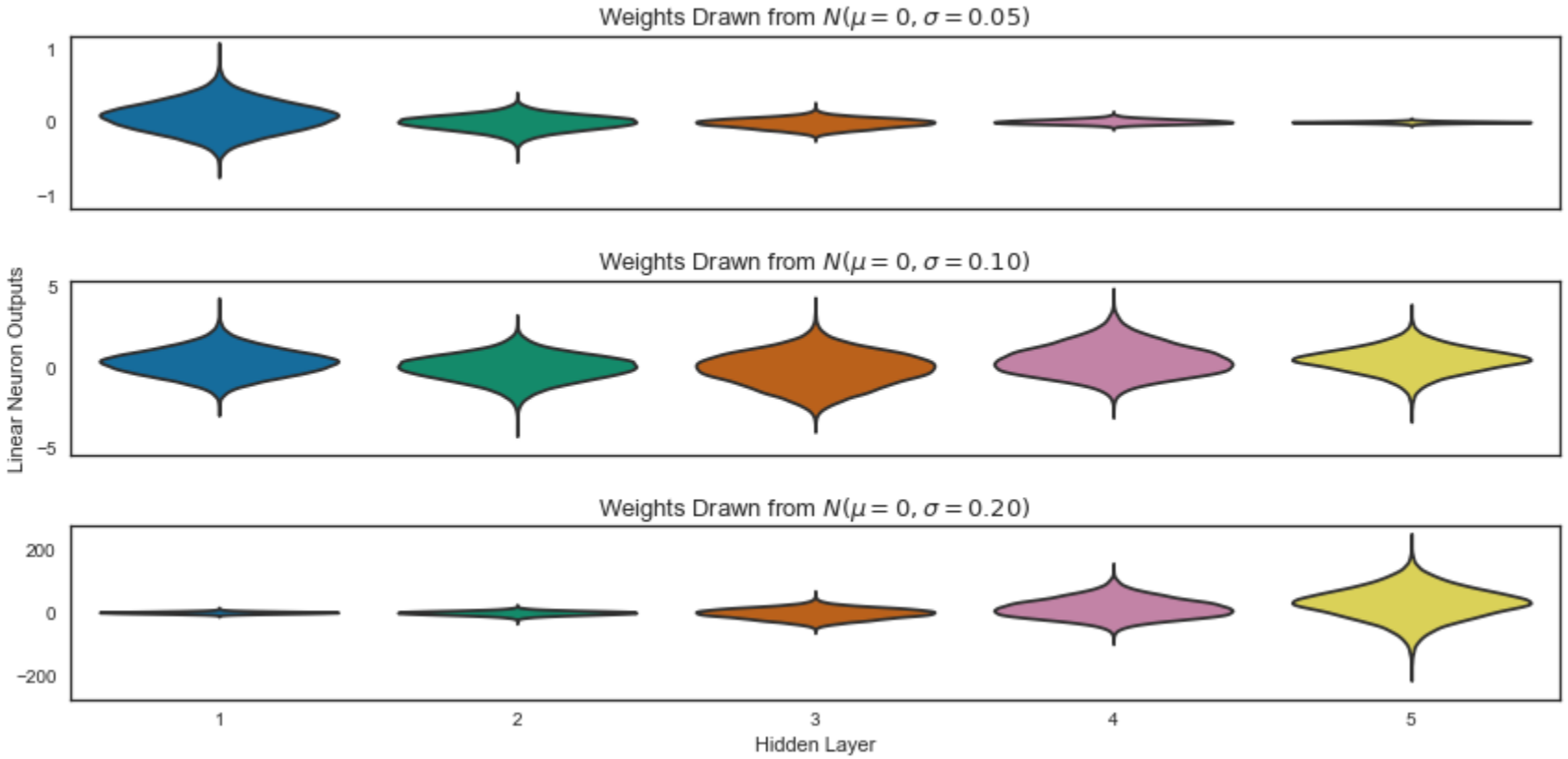
- Weight values move in the same direction!
- Never do this!

Random Initialization of Weights

- In practice, neural weights are initialized with some random number.
- Magnitude of the random number matters.
  - Too small → weights dwindle to zero
  - Too big → weights explode
- You need to find an “appropriate” sized random numbers.

Random Initialization of Weights

- Activations of each hidden layer after one forward pass through the network.



https://intoli.com/blog/neural-network-initialization/

How to find the appropriate range?

- Xavier Initialization (Glorot and Bengio 2010): $\mathrm{W_i} \sim \mathcal{N}(0, \frac{1}{n})$
  - Suppose input X with n components and a linear neuron with random weights W:
$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$
  - Then, assuming W_iX_i are uncorrelated (Bienayme formula),
$$\mathrm{Var}(Y) = \mathrm{Var}\left(\sum_i W_iX_i\right) = \sum_i \mathrm{Var}(W_iX_i) = n\mathrm{Var}(W_iX_i)$$
  - Meanwhile,
$$\mathrm{Var}(W_iX_i) = E(X_i)^2\mathrm{Var}(W_i) + E(W_i)^2\mathrm{Var}(X_i) + \mathrm{Var}(W_i)\mathrm{Var}(X_i) = \mathrm{Var}(W_i)\mathrm{Var}(X_i)$$
  - which gives…
$$\mathrm{Var}(Y) = \mathrm{nVar}(W_i)\mathrm{Var}(X_i)$$
  - Therefore, to constrain $\mathrm{Var}(Y) = \mathrm{Var}(X_i)$, we need $\mathrm{nVar}(W_i) = 1$

$$\begin{aligned}
Var(XY) &= E[(XY)^2] - \{E[XY]\}^2 && (\because Var(A) = E[A^2] - \{E[A]\}^2)\\
&= E[X^2Y^2] - \{E[X]E[Y]\}^2 && (\because \text{X and Y are independent})\\
&= E[X^2]E[Y^2] - E[X]^2E[Y]^2\\
&= (\sigma_x^2 + \mu_x^2)(\sigma_y^2 + \mu_y^2) - \mu_x^2\mu_y^2 && (\because E[A^2] = Var(A) + \{E[A]\}^2)\\
&= \sigma_x^2\sigma_y^2 + \sigma_x^2\mu_y^2 + \sigma_y^2\mu_x^2 + \mu_x^2\mu_y^2 - \mu_x^2\mu_y^2\\
&= \sigma_x^2\sigma_y^2 + \sigma_x^2\mu_y^2 + \sigma_y^2\mu_x^2
\end{aligned}$$

How to find the appropriate range?

- In case of ReLU, Xavier initialization doesn't hold.
  - Derivation of Xavier initialization: assumes a linear neuron.
  - Works alright for Sigmoid and tanh, but not for "rectifiers"

- To over-simplify, half of the outputs (the negative part) dies out under ReLU

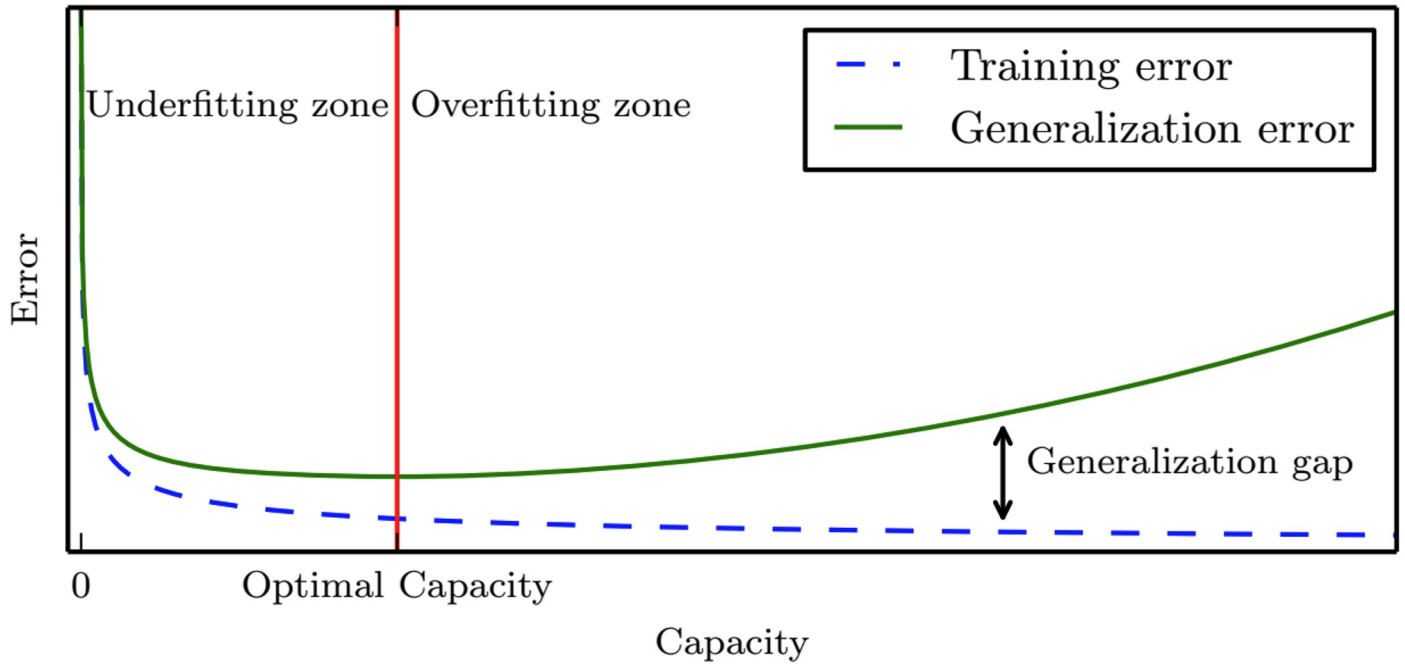
- He et al. (2015): $\mathbf{std}(W_i) = 2/\sqrt{n}$

Dropout


THE BEST WAY TO
EXPLAIN OVERFITTING

Basic Concepts

- The ability to perform well on *unobserved inputs* is called generalization.
- Underfitting: when the model is not able to obtain sufficient training error.
- Overfitting: when the gap between training and generalization error too large.



Dropout

- More abstract-level view of overfitting

What network has seen from the training set

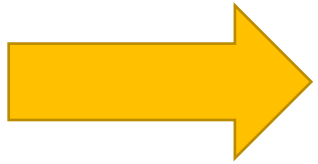
"Too fluffy, it's not a cat"

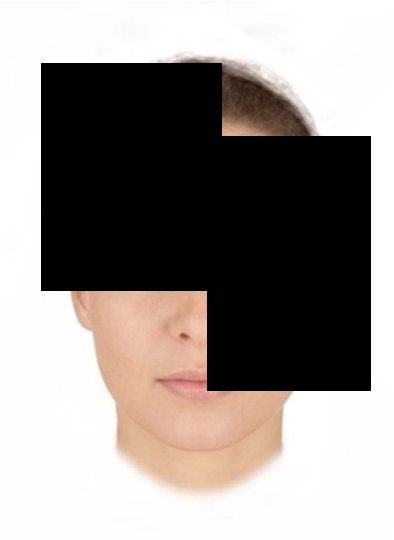
"Too fat. Not a cat"

...

Dropout

- An idea: “let’s hide some information while training”
  - Missing information → incentivize generalization of observation





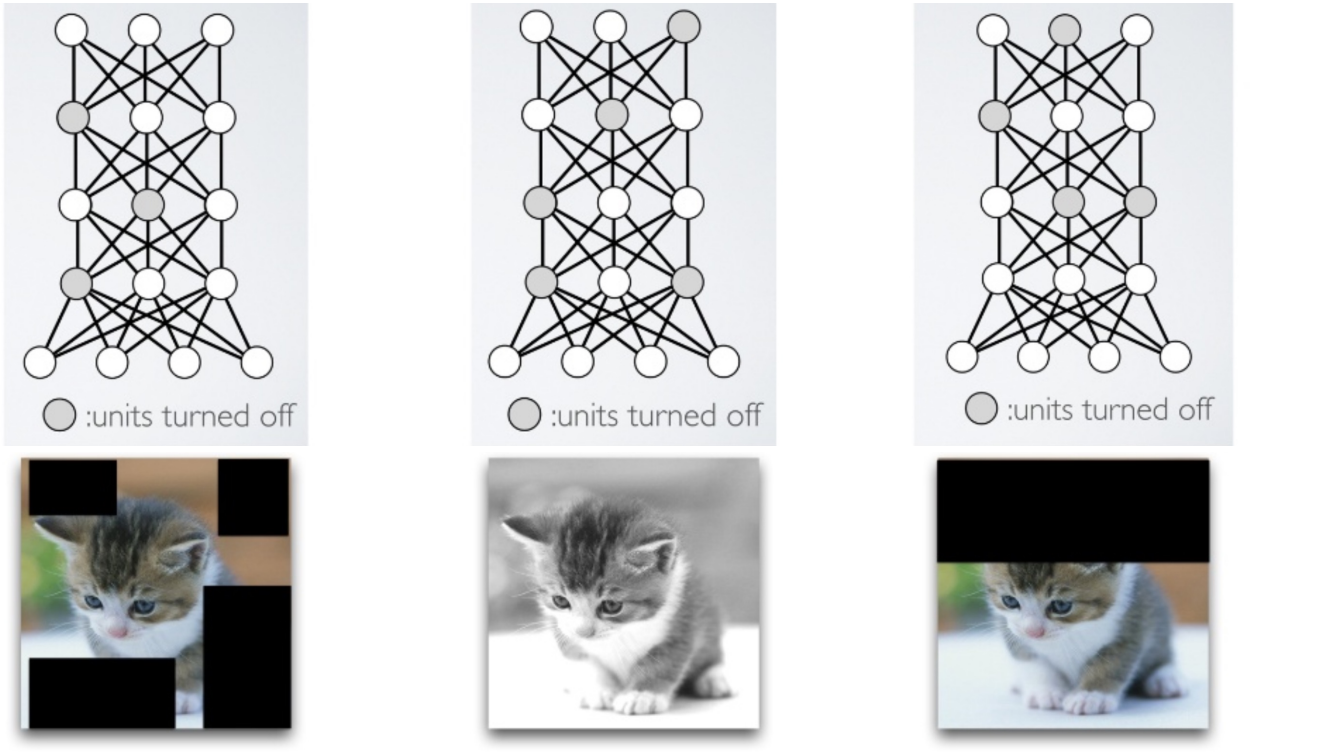
Human face:
“has to have two eyes, two eyebrows,
one nose with two nostrils, upper and lower lips,
two ears, …”

“an eye and a nose
→ face”

“chin, lips, and nose
→ face”

Dropout

- But how?
  - Randomly turn off neurons (set inputs to zero) while training



Dropout

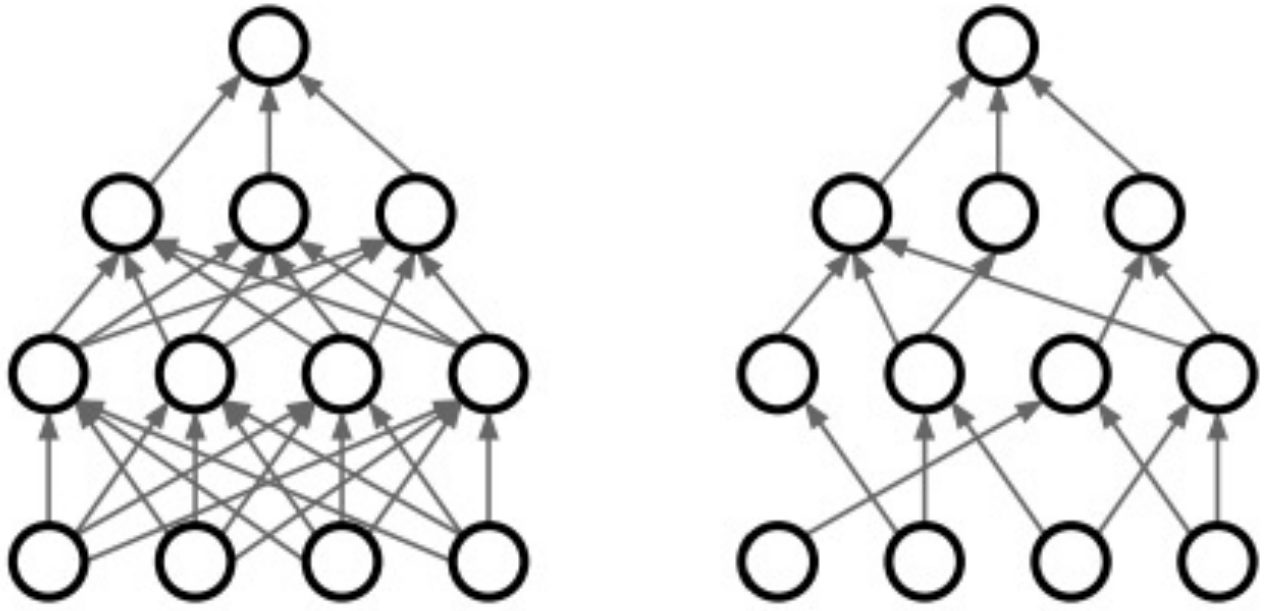
- A bonus: increase of training data in a funny way!
  - How many?
    - Example: a FC layer with 4096 units
      - Possible combinations of turning on/off: $\mathbf{2^{4096} \approx 10^{1233}}$ cases
      - cf) the total number of atoms in the universe: $\mathbf{10^{82}}$ approx.

Dropout

- Dropout happens only at training time. No dropout during test time.
- Dropout layers can be added to any location in the computational graph.
  - Typically before Conv or FC layers.
- Dropout rate can be controlled independently per each dropout layer.

A Variation of Dropout: DropConnect

- Same flavor, but instead of zeroing out neurons, cut the connections by turning off some values in the weight matrix.



Wan et al. (2013)

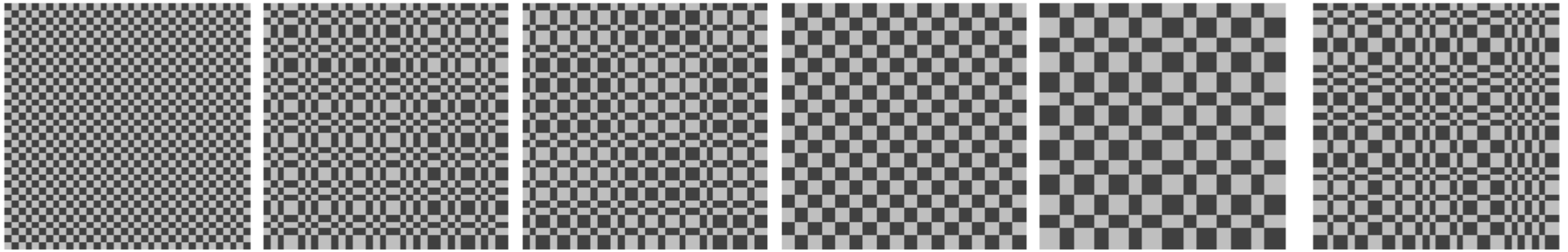
Dropout & DropConnect: What are we trying to achieve?

- Both Dropout and DropConnect introduce some stochasticity (or noise) to the learning process.
- The stochasticity is incentivizing the neural network to better generalize.

- Similar to people!
  - Person with broader experience, tend to think more broadly and see things from multiple different perspectives.
  - Person with narrow experience, tend to believe what they already know is absolute truth.

Fractional Max-Pooling (Graham, 2014)

- So, if we were able to introduce stochasticity to conv & fc layers, why not then to pooling layers?



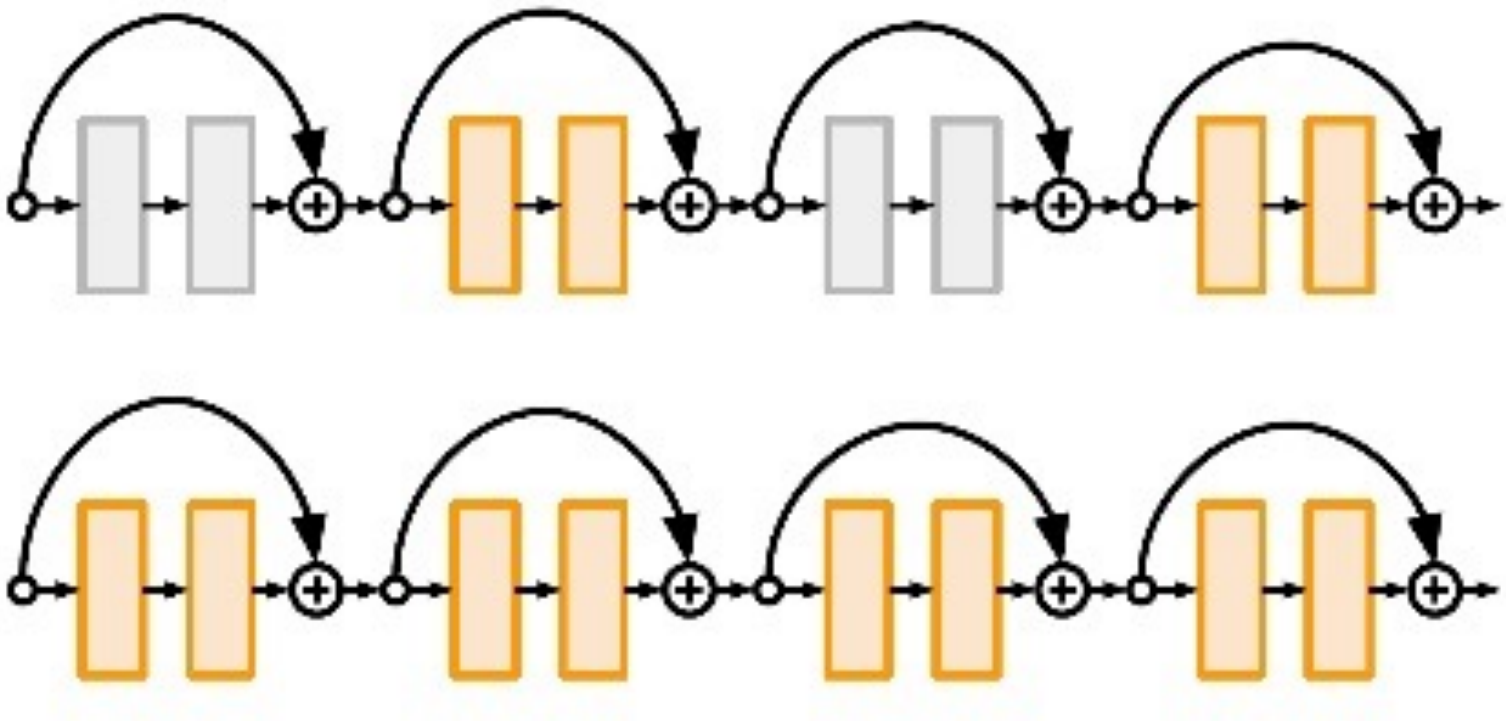
"random pooling grids"

Fractional Pooling (Graham, 2014)

- Sanity check: What are they?

Stochastic Depth (Huang et al. 2016)

- Randomly skip some of the layers during training time…



Data Augmentation

Is this a cat?

How about this?

How about now?

How about now?

How about now?

How about now?

How about now?

How about now?

How about now?

Data Augmentation

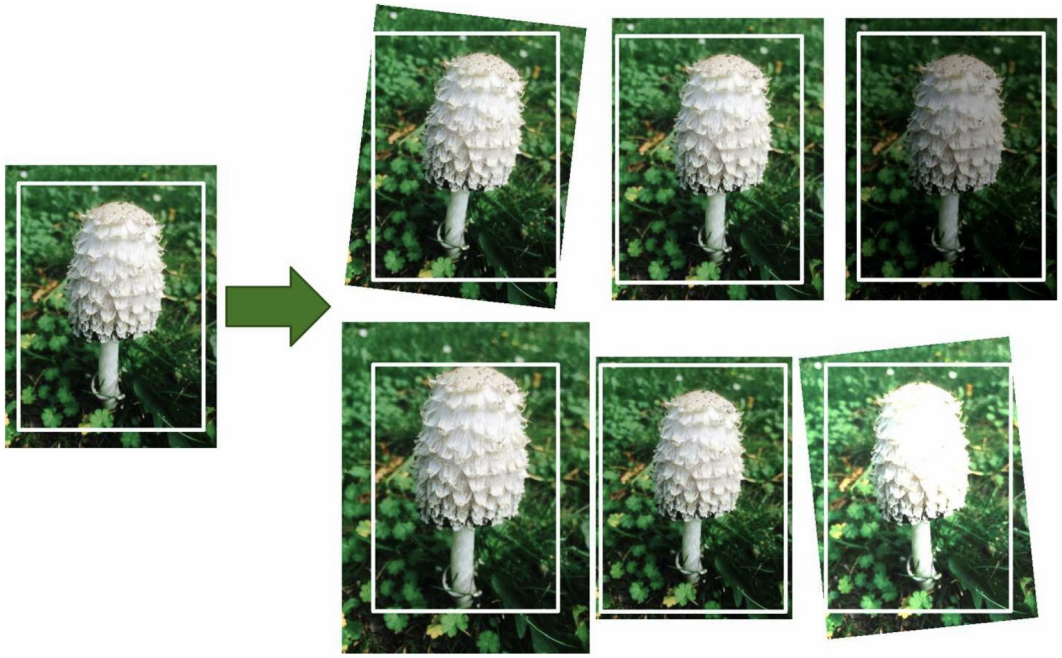
- Random rotation, scale, crop, color change in training set

- Even some crazy ideas… because why not?:
  - Stretch
  - Shear
  - Lens distortion

- Point: Stochasticity!

Data Augmentation

- Make the model generalize better by training it on more data
- If the amount of data available is limited, get around by creating “fake” data and augment them into the training set. How?
- Augmentation has been particularly effective for object recognition. Image transformations (ie. translating, rotating, cropping, brightness correcting, ect) often greatly improve generalization.



Injecting noise and label smoothing

- Injecting (random) noise in the inputs can be a form of data augmentation.
- Injecting noise to output labels? It can be harmful to maximize log p(y|x) when y is a mistake.
- Prevent this by explicitly model the noise on the label: y is correct with probability 1 - $\boldsymbol{\varepsilon}$ (epsilon) with small constant $\boldsymbol{\varepsilon}$
- Label smoothing regularizes a model on a softmax with k output values by replacing hard label 0 and 1 with

$$\frac{\epsilon}{k-1} \text{ and } 1-\epsilon$$

Was it a lot to digest?

- Network architecture
  - How many layers? How many neurons? Kernel size?
  - Skip connection? Inception (network-in-network)?

Module 6 (in two weeks… stay tuned!)

- Choice of activation functions
  - Sigmoid? Tanh? ReLU? LeakyReLU? PReLU? Others?
- Data preprocessing and sampling (Data distribution)
  - Data mean and standard deviation, minimum, maximum, noise
  - Train-val-test split, batching
  - Data augmentation, synthetic data, etc.
- ConvNet initialization
  - Initial solution for gradient descent (initial neuron weights and biases)
  - Transfer learning

This module!
(theme: "gradients & distributions")

- Network regularization
  - Dropouts, L1/L2 regularizers, ensemble, etc.
- Loss functions
  - MAE? MSE? LogCosh? Huber? Cross-entropy? Kullback-Leibler? Others?
- Optimization methods
  - Gradient descent? Momentum? Nesterov correction? Adaptive gradient? Adam? Nadam?
- So many other parameters to consider

Part II (next week)
(theme: "optimization")

Takeaways

- Think differential: gradients, gradients gradients.
  - Will the gradient vanish/explode?
  - Can the gradient point everywhere? Or is it going to be stuck at some specific directions?

- Data and weight distribution is way more important than you think!
  - Again, gradients: can cause vanishing/exploding gradients.
  - Never get lazy on checking the distribution of your data.

- Some stochasticity in data is good!
  - Some noises and stochasticity in your data is not a bad thing, especially with deep nets.
  - Bayesian?