Lecture 06: Neural network building blocks

Data C182 (Fall 2024). Week 04. Tuesday Sept 17th, 2024

Speaker: Eric Kim

Announcements

- Add/drop deadline: **Wednesday September 18th**
- "Course recommended prerequisites": [link]

HW01: Google Colab!

- Important update: you can now do HW01 on Google Colab! [link]
- If you're having trouble with setting up your machine (eg installing Docker), then I recommend going with Colab
- Reasons to go with Colab
  - Easier setup
  - Possibly better compute available (depends on how powerful your laptop is)
  - Access to GPUs (possibility: education GPU credits!)
- Note: if local setup works fine for you (eg Docker), then feel free to continue using that if you prefer.
- Fun fact: this lecture covers much of HW01 (FullyConnected, BatchNorm, DropOut)
  - Thursdays lecture will cover the remainder of HW01 ("ConvNets")

Overview

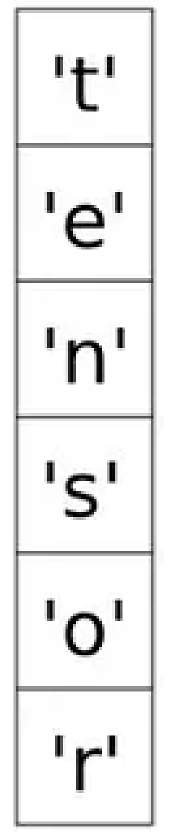
- Pytorch quick intro
  - torch.Tensor, torch.nn.Module
  - torch.nn.Linear
  - Two-layer NN in pytorch
- Standardization/Normalization
  - Batch Norm, Layer Norm
- Activation Functions, Skip Connections
- Weight Initialization, Dropout, Data Augmentation, Model Ensembling

pytorch

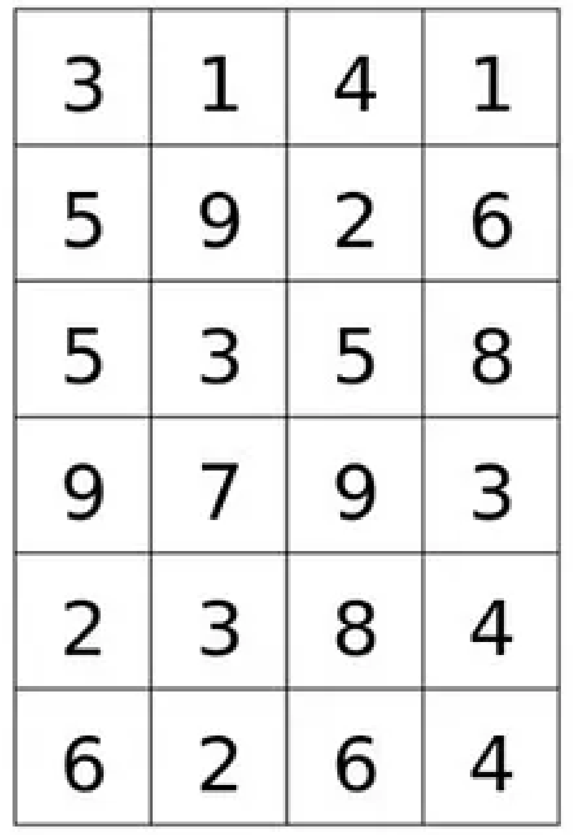
- Deep learning library (Python)
  - Initial release: 2016
  - Pytorch 2.0: 2023
  - Developed by Meta
  - libtorch: a C++ runtime env useful for serving torchscript models in production
- Main idea: make ML experimentation as **easy and flexible** as possible ("dynamic/eager" execution, vs "static" computation approach taken by Tensorflow 1.0)
- Widely used in both industry and academia for both training and serving DNN models.

torch.Tensor

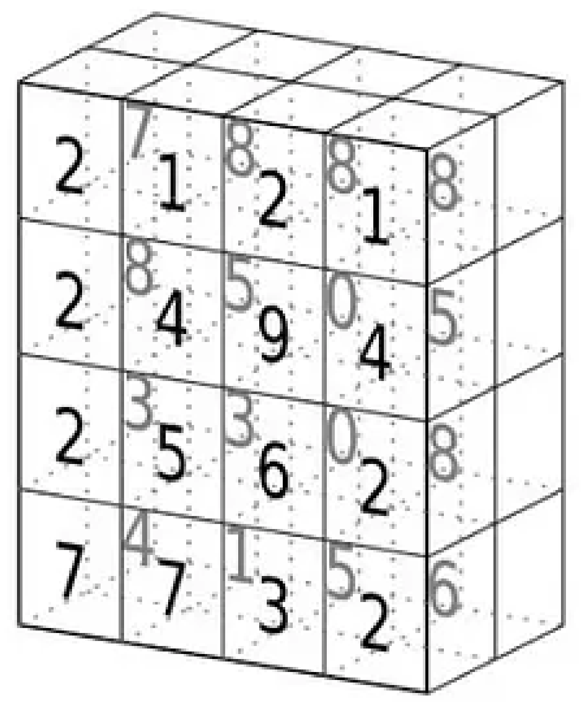
- "Fundamental datatype": torch.Tensor
- "Multidimensional array"
  - Vector: shape=[N]
  - Matrix: shape=[N, M]
  - N-dim array: shape=[d_0, d_1, ..., d_N]
    - Aka "Tensor" (but not in the physics-y mathematically sense)
- Similar in spirit to: numpy.ndarray, pandas.DataFrame
- Docs: https://pytorch.org/docs/stable/tensors.html



tensor of dimensions [6]
(vector of dimension 6)



tensor of dimensions [6,4]
(matrix 6 by 4)



tensor of dimensions [4,4,2]

https://aadhil-imam.medium.com/introduction-pytorch-tensors-tutorial-01-9dca5b4a1590

torch.Tensor properties

- Shape
  - Ex: a matrix with 2 rows and 3 columns has shape [2, 3]
- Data type
  - Ex: torch.float32, torch.int32, torch.float16
  - Tip: it's common to train/serve in reduced precision for improved performance (eg torch.float16, bfloat16)
- Device (aka "CPU" vs "GPU")

```
# a matrix with shape=[2, 3] (2 rows, 3 columns)
>>> my_matrix = torch.tensor(
    [
        [1, 2, 3],
        [4, 5, 6],
    ],
    dtype=torch.float32
)
>>> print("my_matrix: ", my_matrix)
my_matrix:  tensor([[1., 2., 3.],
        [4., 5., 6.]])
>>> print("my_matrix.shape: ", my_matrix.shape)
my_matrix.shape:  torch.Size([2, 3])
```

Operations with torch.Tensor

- Arithmetic operations with Tensors (+, -, *, /)
- Assignment
- Indexing/slicing
- Dot products, matrix multiply
- [DEMO: tensor_demo.py, two_layer_linear_nn_demo.py]

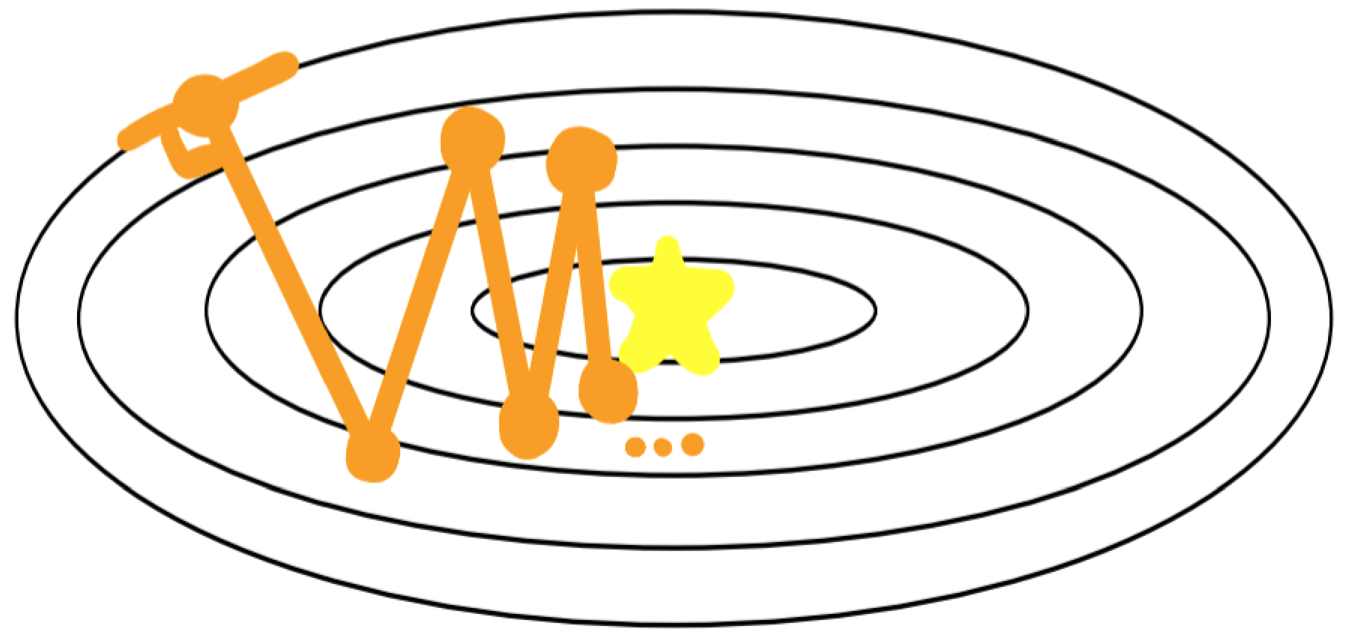
Today’s lecture

- Today’s lecture is the “collected wisdom” of techniques, tips, and tricks for how to build and train the best neural networks
- We focus on techniques that have “stood the test of time”
  - Normalization, activations, weight initialization, hyperparameter optimization, …
- Nevertheless, new and better techniques are introduced all the time
  - The best deep learning practitioners and researchers typically are also the best at keeping up with the latest trends

Standardization and normalization

Some motivation for input standardization

- Suppose the input $\mathbf{x}$ is 2D and x_1 is usually much larger than x_2 — what could go wrong?
  - Adjusting the part of θ corresponding to x_2 may have a bigger effect on the loss
- We saw that momentum and Adam can suffer less from issues like oscillation
  - Compared to vanilla gradient based optimization
- Nevertheless, **standardization** of the input dimensions is typically an important *preprocessing* step and never hurts performance
  - Think of it like helping to "circularize" the loss landscape



Input standardization

- Input standardization is carried out for each dimension of the input separately
- For each training input, for each dimension d, we subtract the mean $\mu_d = \frac{1}{N}\sum_{i=1}^{N} x_d$ and divide by $\sigma_d = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_d - \mu_d)^2}$
- Ex: if an input feature is "User's Age", then we'd preprocess the "age" column via: new_age = (age - age_mean) / (age_std).
    - Network now reasons in terms of "how many std-devs is the input age from the mean?" instead of "raw age units"

Motivation for standardizing features

- Question: why might we want our input features to be standardized?
- **Hint**: consider if input feature x_0 is REALLY large in magnitude relative to input feature x_1. Consider a Linear layer with weight A (no bias for now):

Linear forward pass:

$$\begin{bmatrix} \underset{\text{BIG}}{x_0} & \underset{\text{small}}{x_1} \end{bmatrix} \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

$$= [\underbrace{A_{00} * x_0}_{\text{BIG}} + \underbrace{A_{10} * x_1}_{\text{small}} \quad A_{01} * x_0 + A_{11} * x_1 \quad A_{02} * x_0 + A_{12} * x_1]$$

Issue 1: layer outputs are dominated by x_0. If x_1 were important to the final model outputs (eg classifications), then it's "washed away".

Motivation for standardizing features

- Question: in theory, can the NN learn to rescale x_0, x_1 so that they're at a comparable magnitude? Suppose x_0 is typically 1000x larger in magnitude than x_1. What would the resulting weight matrix A look like?

Linear forward pass:

$$\begin{bmatrix} \underset{\text{BIG}}{x_0} & \underset{\text{small}}{x_1} \end{bmatrix} \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

$$= [\underbrace{A_{00} * x_0}_{\text{BIG}} + \underbrace{A_{10} * x_1}_{\text{small}} \quad A_{01} * x_0 + A_{11} * x_1 \quad A_{02} * x_0 + A_{12} * x_1]$$

Answer: A_00, A_01, A_02 (first row of A) should be roughly 1000x smaller than A_10, A_11, A_12. [DEMO normalization_motivation.py linear_demo()]

$$\begin{bmatrix} 0.001 & 0.001 & 0.001 \\ 1 & 1 & 1 \end{bmatrix}$$

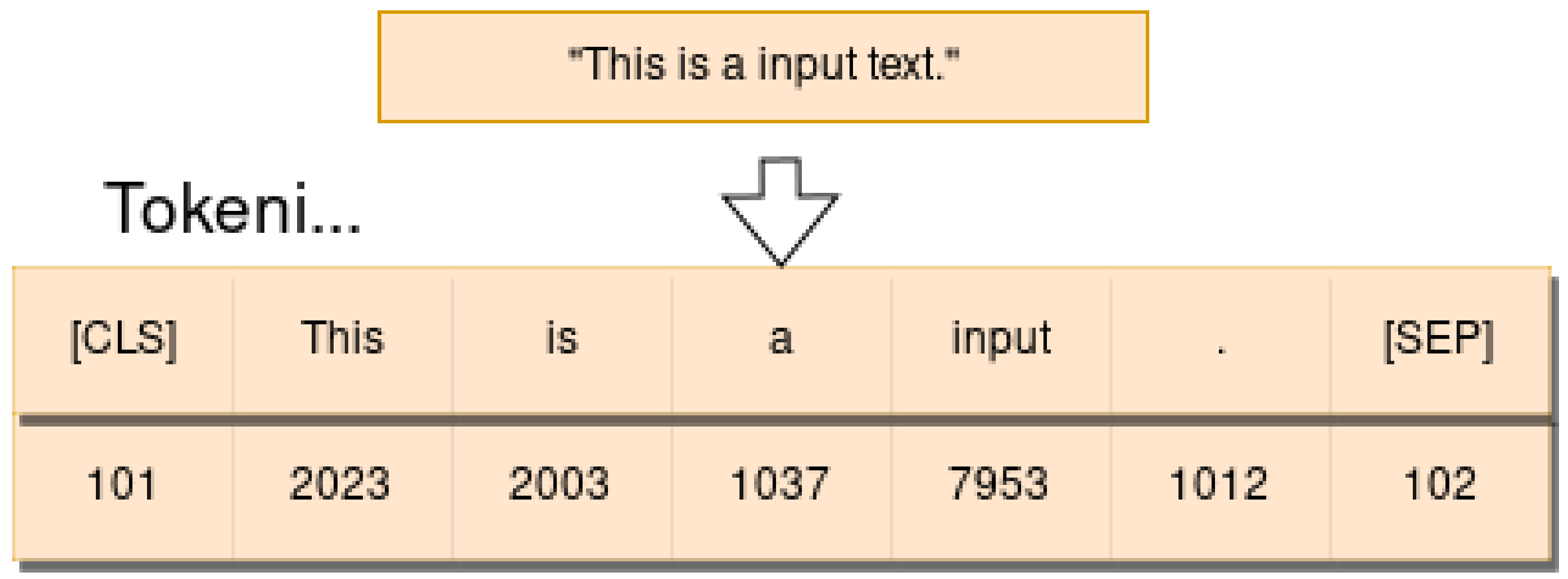
Motivation for standardizing features

- Although the NN can in theory learn to figure out scale difference, a common NN technique is to make the learning problem as easy as possible.
  - In this case: standardizing input features so that they're in the same "general range" so that the NN doesn't have to learn scale differences.

$$\underset{\text{Input features}}{\begin{bmatrix} 1e9 & 0.01 \\ 1.2e9 & 0.03 \\ 1.1e9 & 0.01 \\ \ldots & \ldots \end{bmatrix}} \xrightarrow[\text{Standardize}]{} \underset{\substack{\text{Input features} \\ \text{("standardized")}}}{\begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.5 \\ 0.2 & 0.2 \\ \ldots & \ldots \end{bmatrix}}$$

Input standardization

- There are some variations on this, e.g., this is usually done *per channel* for image inputs rather than per dimension
- And for discrete inputs, such as in language (eg token IDs), this is typically not done at all

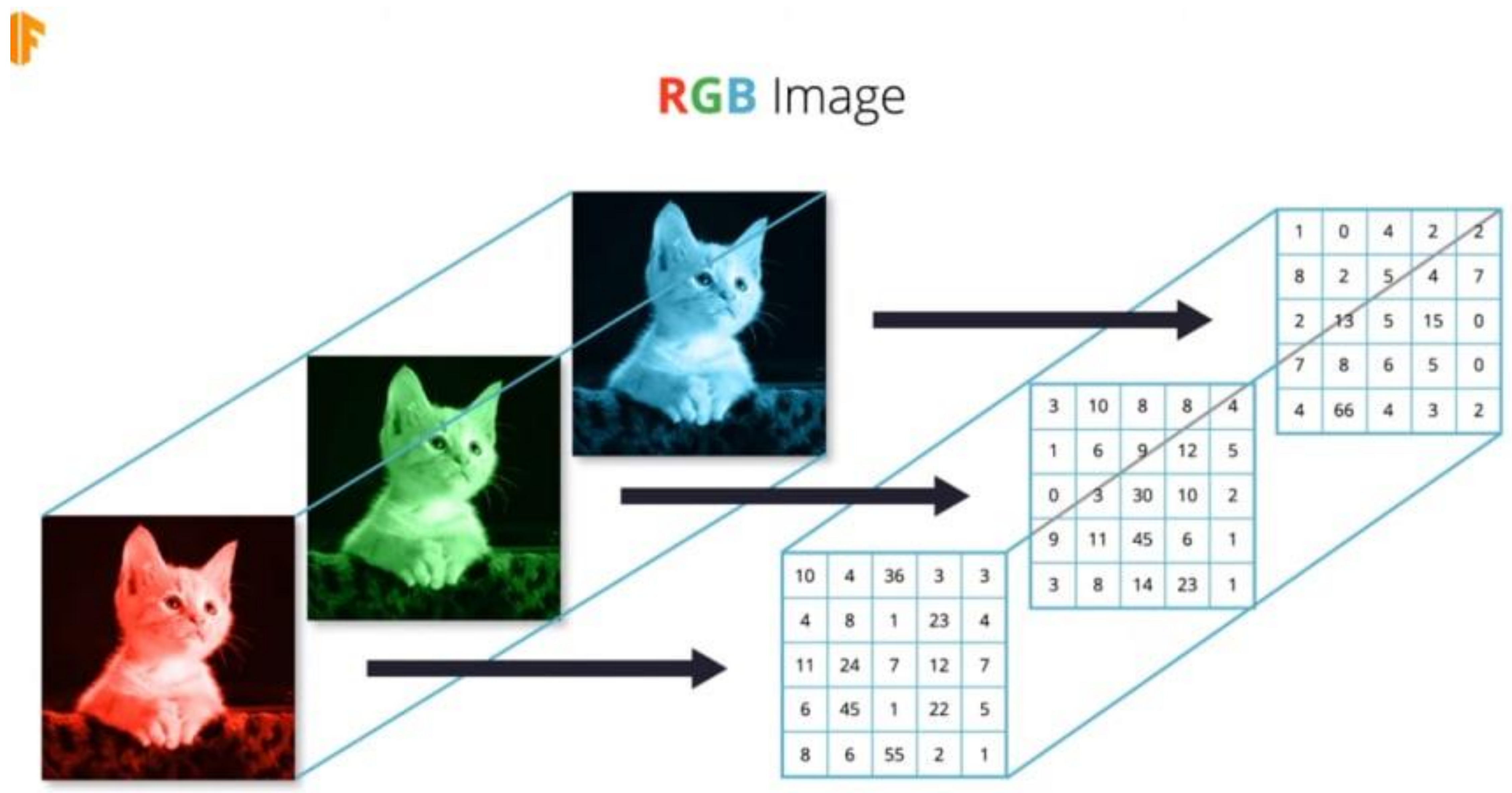


"This is a input text."

Tokeni...

[CLS]	This	is	a	input	.	[SEP]
101	2023	2003	1037	7953	1012	102

NLP: tokenizer outputs are typically int IDs, eg ID=2023 corresponds to "This", ID=2003 corresponds to "is". Doesn't make sense to standardize these.



https://dev.to/sandeepbalachandran/machine-learning-going-furthur-with-cnn-part-2-41km
https://vaclavkosar.com/ml/Tokenization-in-Machine-Learning-Explained

A few more comments on standardization

- The far more common (but incorrect) term for standardization is **normalization**
  - For the rest of this lecture and beyond, we will use this term instead
- Beyond normalizing inputs, outputs are often also normalized if they are continuous values (but not if they are discrete values such as labels)
  - Just like normalizing inputs, think of it like “circularizing” the loss landscape
- Maybe we can also consider… normalizing intermediate activations $\mathbf{z}^{(l)}$ or $\mathbf{a}^{(l)}$?
  - What might be trickier about this?

Normalizing intermediate activations

- Activations change throughout the course of training!
- This means that we have to recompute these *normalization statistics* (μ_d and σ_d) every time we update our neural network parameters
  - And it would be prohibitively expensive to recompute using all the training data
- Let’s discuss the two most commonly used methods for normalizing activations that get around this issue by using only *mini batches* or *single data points*
  - These are **batch normalization (BN)** and **layer normalization (LN)**, respectively

Batch normalization (BN)

- Consider normalizing the intermediate activation $\mathbf{z}^{(l)}$ (same story for $\mathbf{a}^{(l)}$)
- Recall that, during training, we use mini batches of B data points for each update
- We can compute the per dimension mean and standard deviation of $\mathbf{z}^{(l)}$ using just this mini batch, rather than the entire training set
  - This should be a good approximation for large enough B and if the points in the mini batch are sampled i.i.d. (they're not, but close enough)
- BN refers to normalizing $\mathbf{z}^{(l)}$ using these *mini batch statistics*

The BN “layer”

- Typically, we normalize either the $\mathbf{z}^{(l)}$ or the $\mathbf{a}^{(l)}$, but not both
- We can think of this as putting a BN “layer” either before or after the nonlinearity
  - Both choices usually work, it is usually easy enough to try both
- The BN layer also includes one more thing: learnable *scale* and *shift* parameters
  - That is, after normalization, we multiply each dimension by γ_d and add β_d
  - This is done so that the neural network doesn’t lose expressivity — if needed, it could even learn to undo the normalization!

BN: training vs. testing

- Models with BN layers operate in two different modes: “train” vs. “test” or “eval”
  - These are used during training and testing time, as the names suggest
- Train mode is what has been described — compute statistics using the mini batch
- Eval mode instead uses the average statistics computed during train time
  - That is, we additionally maintain an *exponential running average* of the normalization statistics during model training, for use at test time
  - This is important if, e.g., we only are able to see one test point at a time
- Otherwise, the normalization, scaling, and shifting work identically in both modes

The pros and cons of BN

- BN enables higher learning rates and therefore faster training
- BN fixes many of the training stability issues that people used to worry about
  - Before BN, this course would have talked a lot more about these issues
- But BN also requires a large enough B for a good estimate of the statistics
- It’s also kind of weird that the model works differently for training vs. testing…
- It’s also kind of weird, at training time, for the model’s predictions on a data point to depend on the other points in the mini batch…
- [DEMO batch_norm_demo.py]

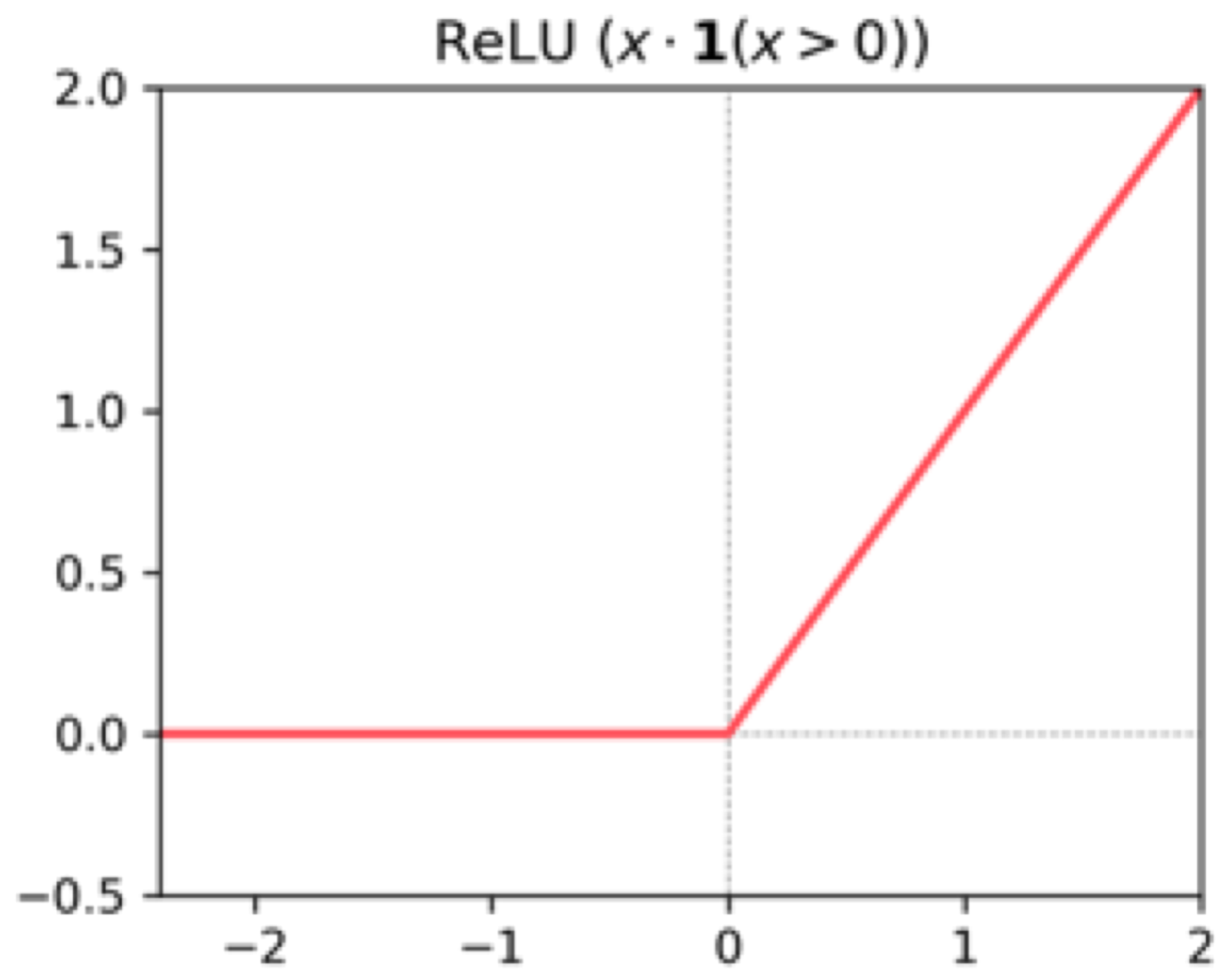
Layer normalization (LN)

- LN is a different normalization approach that does not use mini batch information
  - So it operates on single data points, and it is identical at training vs. test time
- LN is basically the "transpose" of BN: compute the mean and standard deviation of $\mathbf{z}^{(l)}$ *across the feature dimensions*, rather than per dimension
  - Now, each data point will have different normalization statistics, but these statistics are shared across dimensions
  - We still have learnable *scale* and *shift* parameters that are applied after the normalization step, to produce the final output of the LN layer
  - [DEMO layer_norm_demo.py]

Network architecture choices

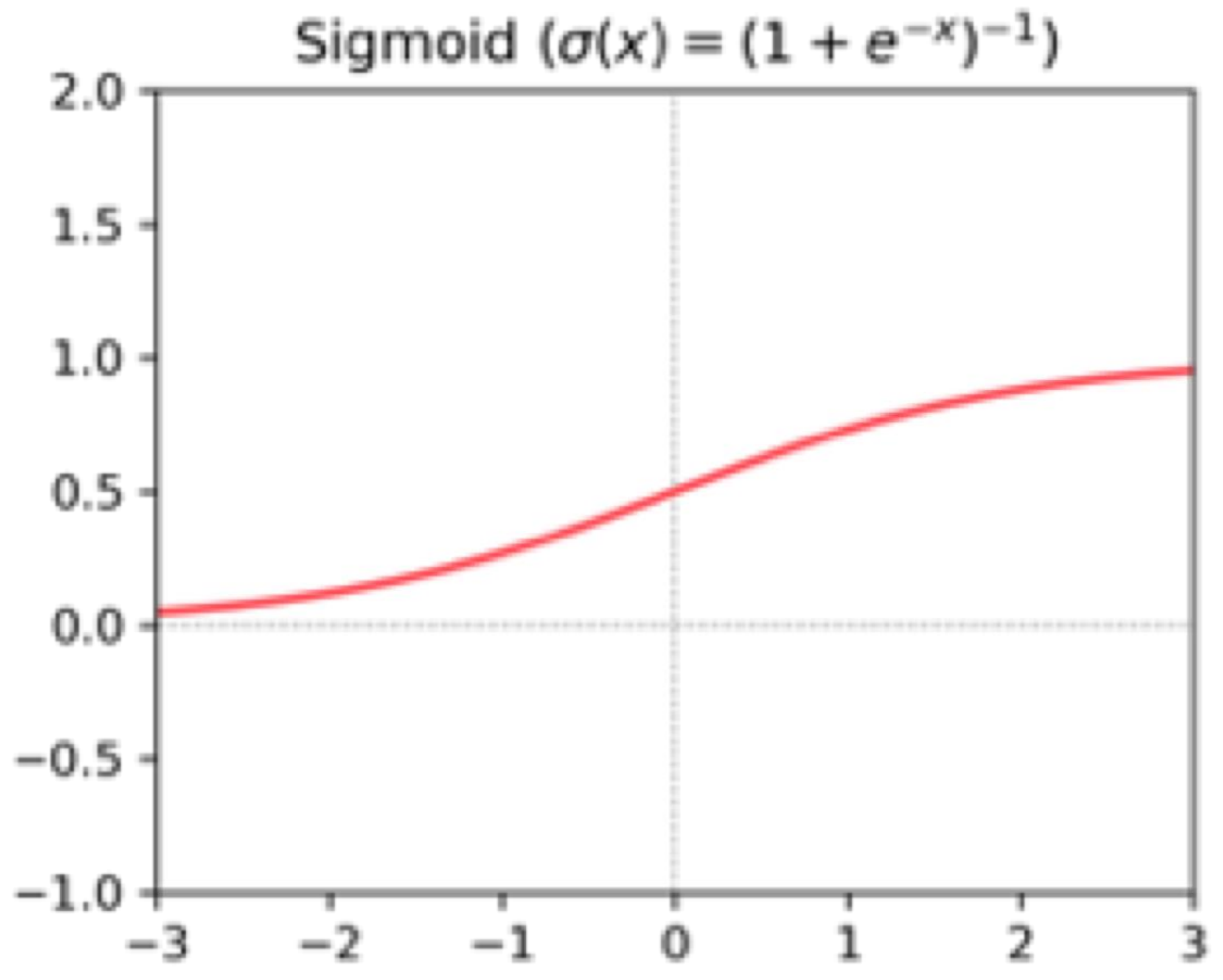
Nonlinearities — rectified linear units (ReLUs)

- $ReLU(\mathbf{v}) = max\{0, \mathbf{v}\} = \mathbf{v} \odot \mathbf{1}[\mathbf{v} > 0]$
    - Therefore, $\nabla_{\mathbf{v}} ReLU(\mathbf{v}) = diag(\mathbf{1}[\mathbf{v} > 0])$
- A very common choice for hidden layer activations
- “Gates” inputs based on their sign
- May be suboptimal because, for negative values, the gradient provides no update direction



Nonlinearities — sigmoid

- $sigmoid(\mathbf{v}) = \frac{1}{1+\exp\{-\mathbf{v}\}} = \frac{\exp\{\mathbf{v}\}}{\exp\{\mathbf{v}\}+1}$
- Along with $tanh$, has really fallen out of favor as a hidden layer activation
- Why? Very small gradient values for large inputs
  - $\nabla_{\mathbf{v}} sigmoid(\mathbf{v}) = diag(sigmoid(\mathbf{v}) \odot (1 - sigmoid(\mathbf{v})))$
- Used as the output "activation" for binary classification

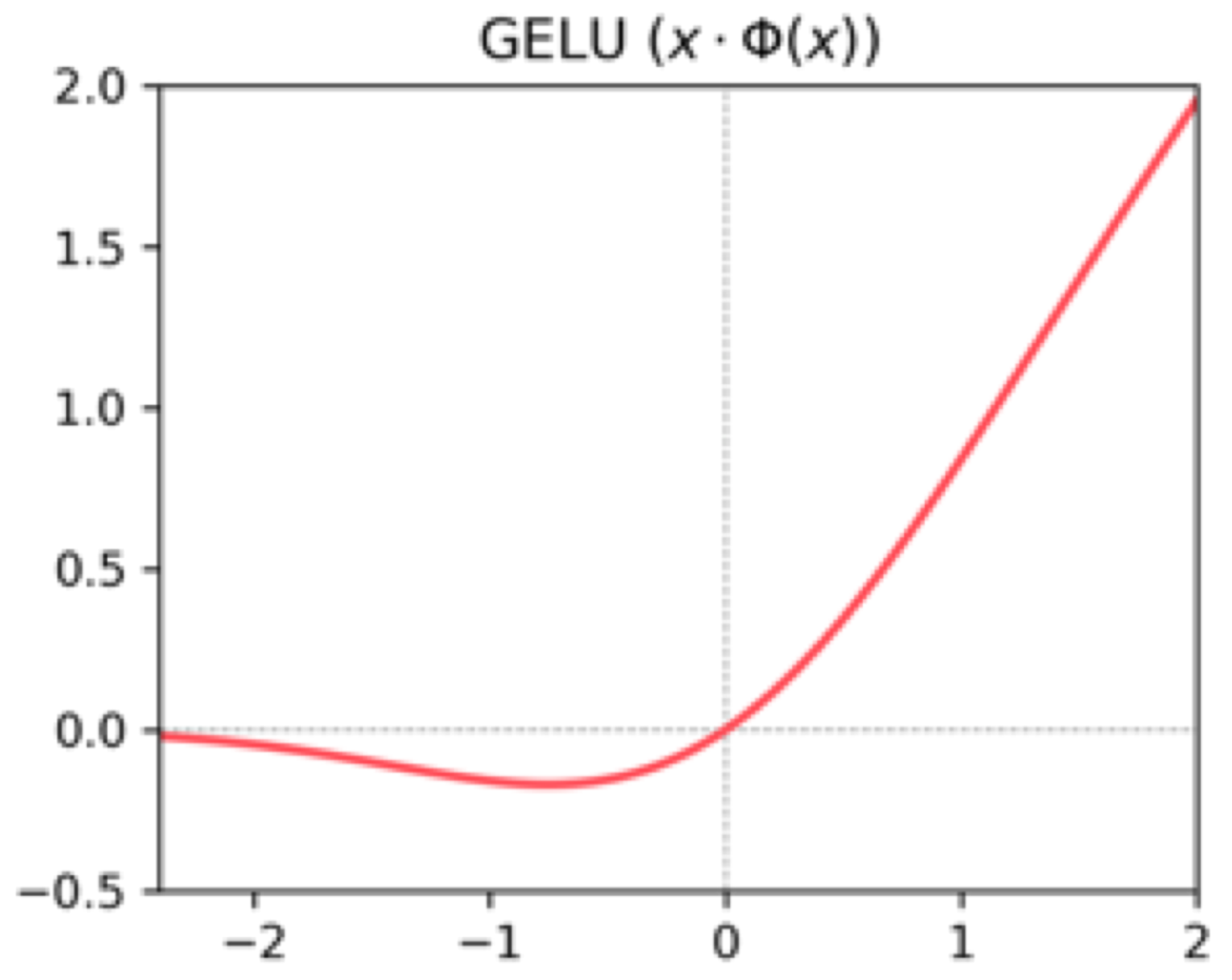


Nonlinearities — Gaussian error linear units

- GELUs (and friends)
- Both $ReLU$s and $sigmoid$s have gradient issues
- Another function that sidesteps some of these issues is the Gaussian error linear unit (GELU)

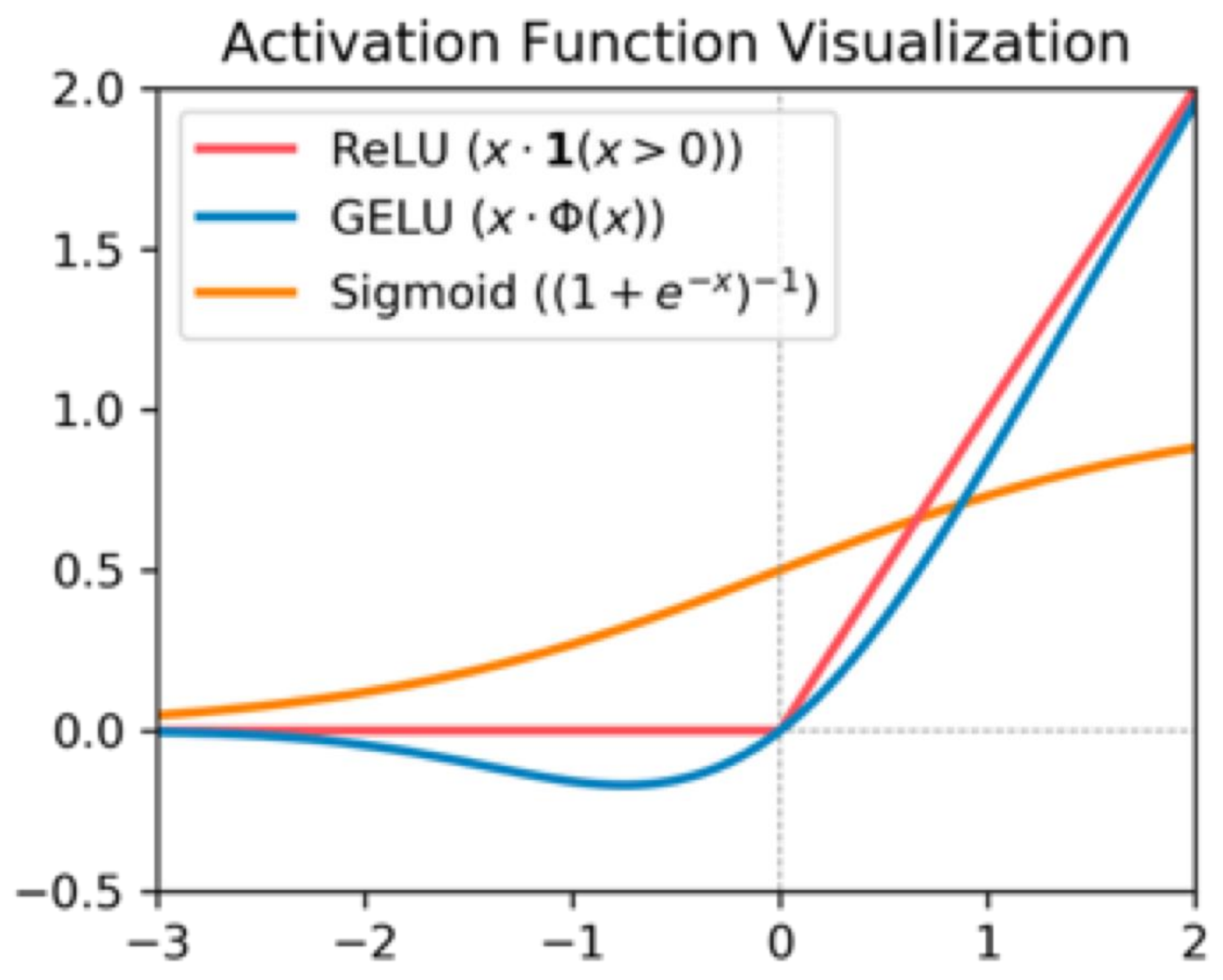
$$GELU(\mathbf{v}) = \mathbf{v} \odot \Phi(\mathbf{v})$$

  - Φ evaluates the CDF of $\mathcal{N}(0,1)$ element wise
- Closely related to other functions that pass the input through a "soft gate" — e.g., $\mathbf{v} \odot sigmoid(\mathbf{v})$ is quite similar (sometimes called $SiLU$ or $swish$)



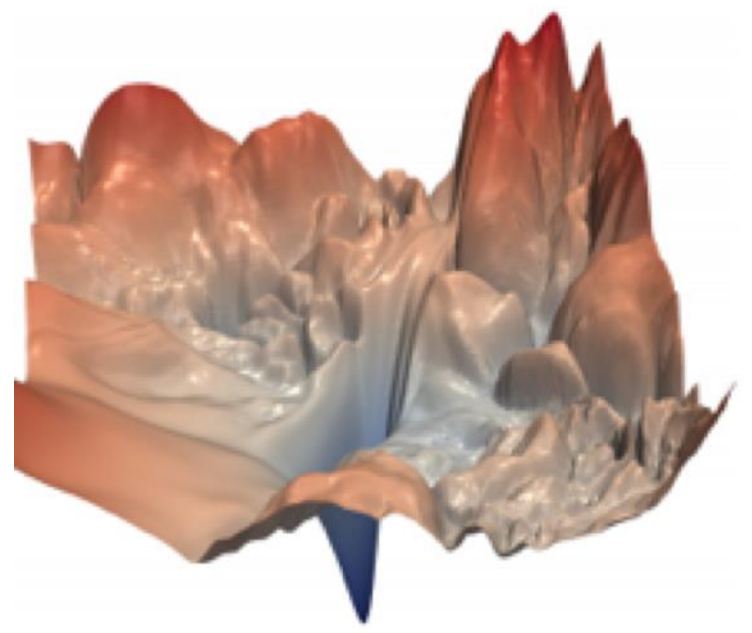
Comparing these nonlinearities

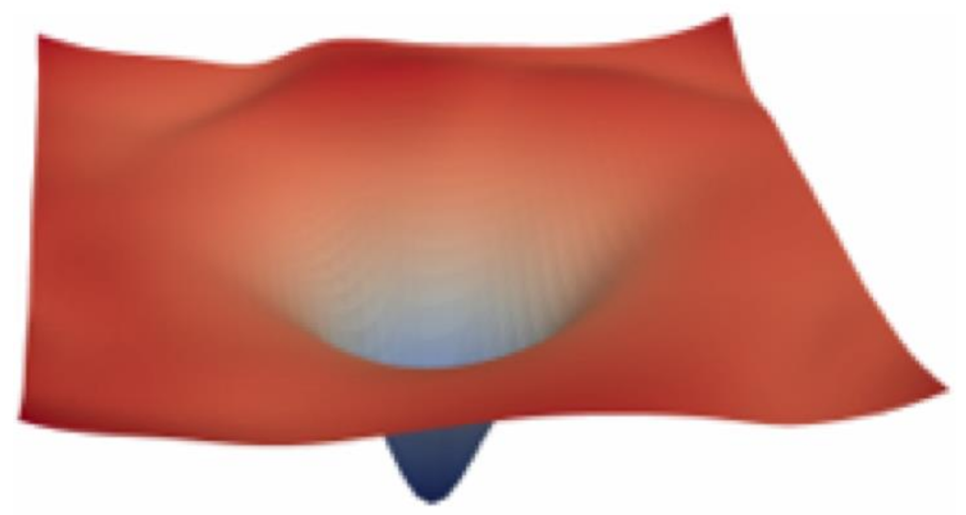
- Both $sigmoid$ and $ReLU$ are non negative and monotonically non decreasing
- $sigmoid$ and $GELU$ are smooth, which is sometimes important from an optimization perspective
- $sigmoid$ is historically an important activation but is rarely the only nonlinearity used in today's neural networks



Skip connections

- Basically every state-of-the-art neural network uses **skip connections**
- Very simple high level idea: $\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) + \mathbf{a}^{(l-1)}$, rather than just $\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)})$
- This idea was popularized by *residual convolutional networks (ResNets)*
  - Allowed for training much deeper, more performant models
- The loss "landscape" of neural networks with residual connections looks much nicer


Li et al, NIPS '18


Li et al, NIPS '18

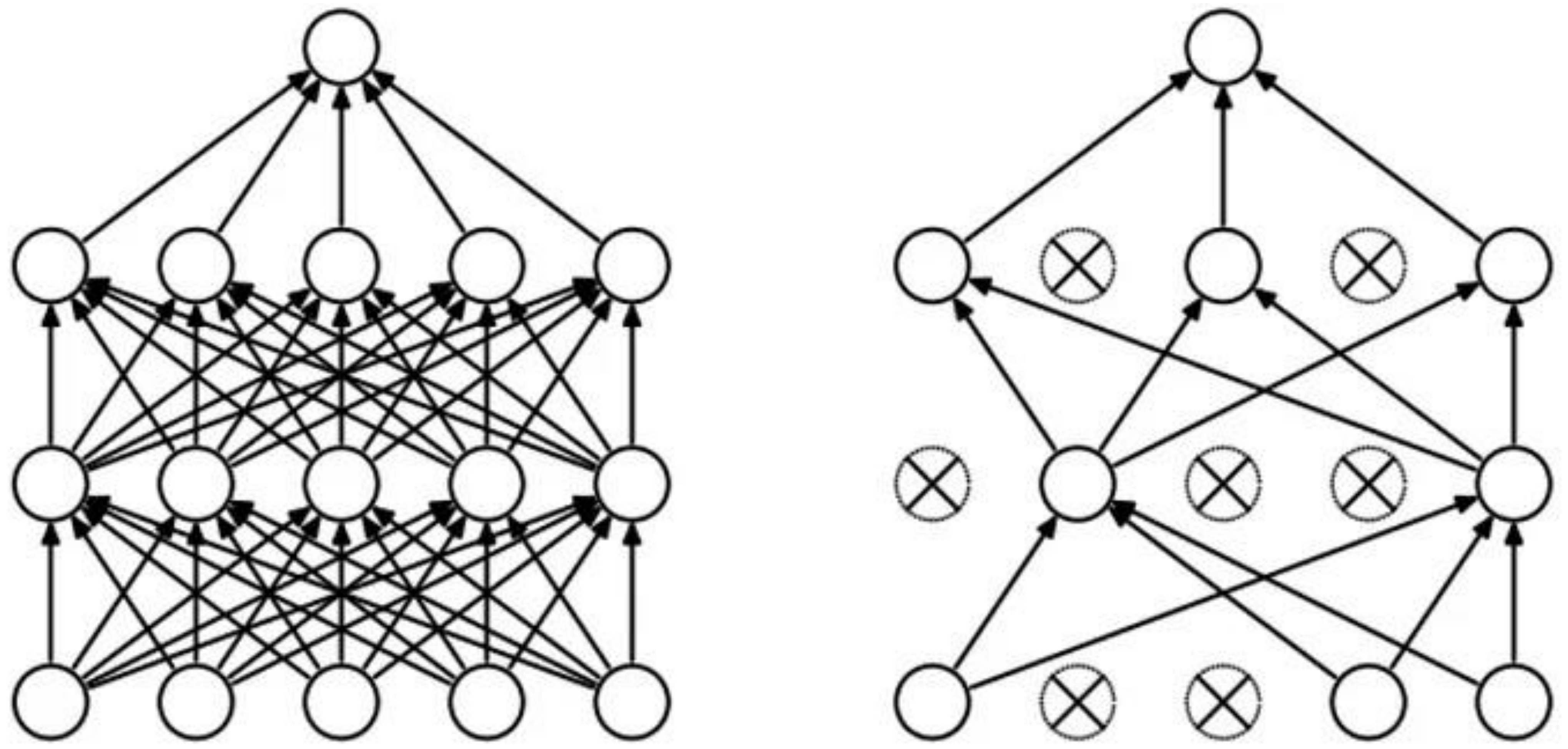
Training considerations

Weight initialization

- A thought exercise
  - What should we initialize our neural network parameters (weights) to? This question is less important with the advent of BN and LN, but it is still interesting to think about
  - If $x_j \sim \mathcal{N}(0,1)$ in each dimension j, and we initialize each $\mathbf{W}_{ij}^{(1)} \sim \mathcal{N}(0, \sigma_W^2)$…
    - …then we get $\mathbb{E}[z_i^2] = \sum_j \mathbb{E}[(\mathbf{W}_{ij}^{(1)})^2]\mathbb{E}[x_j^2] = d\sigma_W^2$
  - Therefore, picking $\sigma_W^2 = \frac{1}{d}$ gives us outputs similar in magnitude to the inputs
    - We can do this at every linear layer, i.e., initialize each $\mathbf{W}^{(l)}$ with variance inversely proportional to the *input dimensionality* to that layer
  - In practice: it's slightly more complicated, but it's done for you by deep learning libraries
    - Ex: see pytorch's weight initialization strategies: [link]

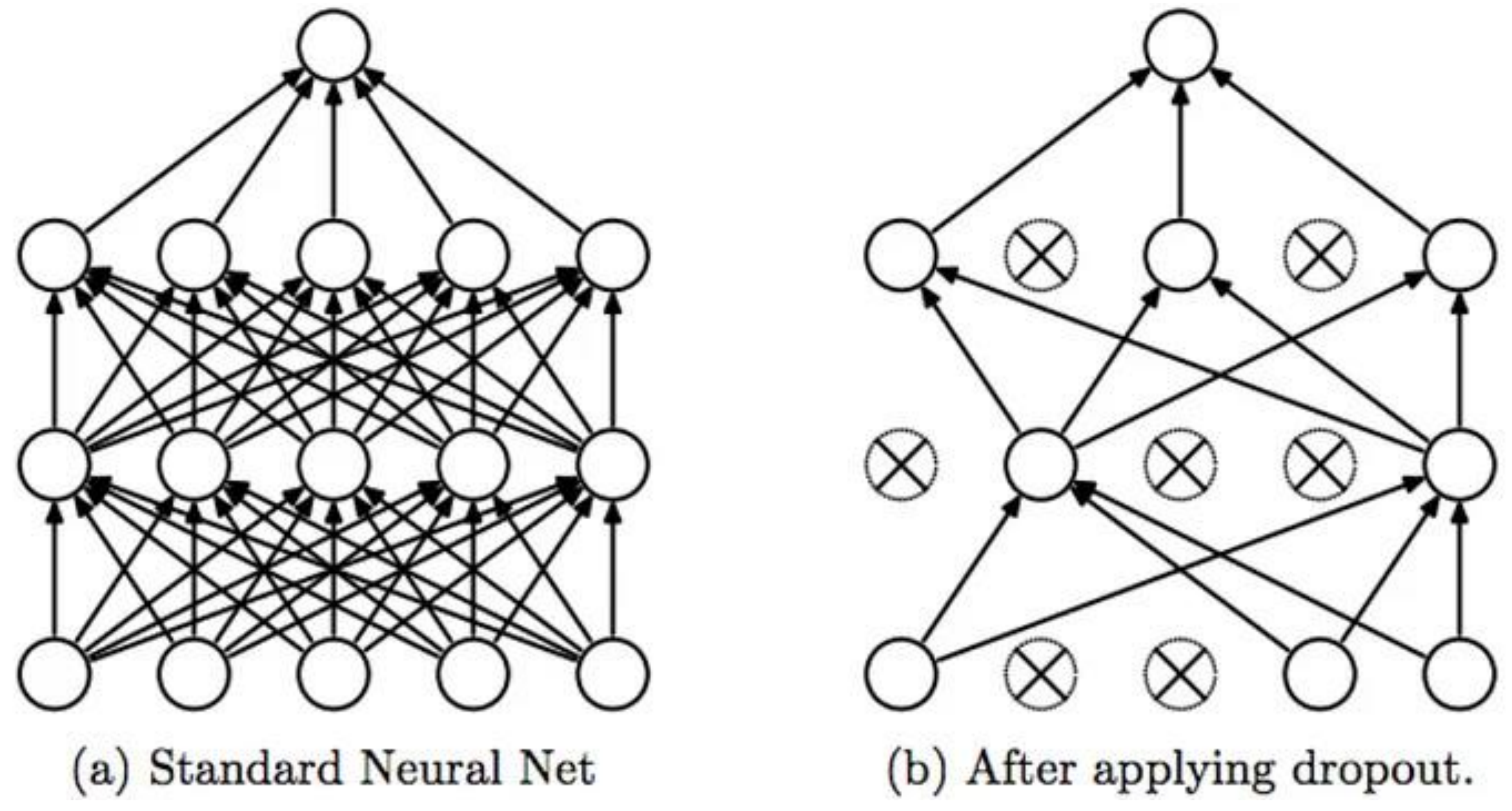
Dropout

- Often, **dropout** is applied to our model during training
- The basic idea is: randomly disable ("drop out") neurons in the network.
- Implementation: randomly zero out some fraction p of the $\mathbf{W}_{ij}$
    - Can implement as element wise multiplication of each $\mathbf{W}^{(l)}$ with a *boolean mask*


(a) Standard Neural Net (b) After applying dropout.

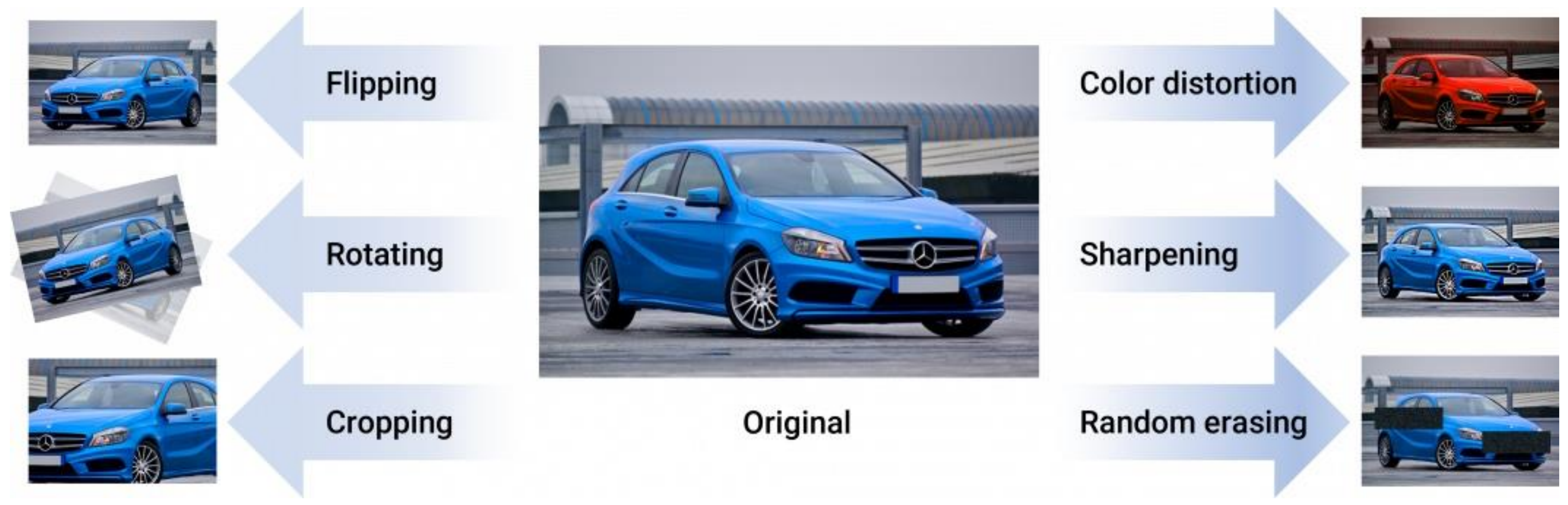
Dropout

- Dropout builds *redundancies* into the model, such that it doesn't rely too much on any particular "pathways" through the network
  - Yet another example of inductive biases at work!
- Can be thought of as a form of regularization specific to NN's.
- Some care should be taken to make training vs. test output magnitudes consistent
  - Ex: if drop probability is 0.8, then at test time, multiply activations by (1 / (1 - 0.8)) = 5



Data augmentations, briefly

- For some problems, **data augmentations** are an indispensable part of training
  - E.g., for image classification: we apply random flips and crops to the images
- This is useful for encoding **invariances**, e.g., flipping and cropping do not change the image class
  - Another inductive bias!
- For some domains, such as natural language, it is harder to come up with good data augmentation schemes



https://viso.ai/computer-vision/image-data-augmentation-for-computer-vision/

Neural network ensembles

- If you have enough compute, training multiple neural networks is often useful
- Same concept as ***bagging*** for other machine learning models — an **ensemble** of models reduces variance and combats overfitting
  - Turns out, also very good at ***uncertainty quantification***
- In theory: create different ***bootstrap samples*** of the dataset to train the models
  - In practice for neural networks: just train them all on all of the data
- In theory: when predicting, average all of their output probabilities together
  - In practice: just take a majority vote

Hyperparameter optimization

- We briefly talked last lecture about tuning hyperparameters such as learning rate, momentum, regularization strength, etc.
  - Training loss helps diagnose underfitting, validation loss for overfitting
- We are adding in even more hyperparameters to tune with this lecture!
  - Normalization, architecture choices (nonlinearities, skip connections), dropout, …
- It is definitely daunting to try and tune all of these — here are some tips

Hyperparameter optimization

- Typically, tuning hyperparameters goes from “coarse to fine”
    - E.g., first find the right order of magnitude for the learning rate, then zero in
- Hyperparameter ***search*** can be done with randomly sampled values or in a grid
- When ***grid searching***, it is standard to space values evenly in log space
- For example, to cover [0.001, 0.01] approximately evenly, use:
    - [0.001, 0.003, 0.01] if grid searching with three values
    - [0.001, 0.002, 0.005, 0.01] if grid searching with four values
- Putting it all together: [DEMO: two_layer_nn_bells_and_whistles.py]