Lecture 7:
Neural network building blocks

CS 182/282A (“Deep Learning”)

2022/02/09

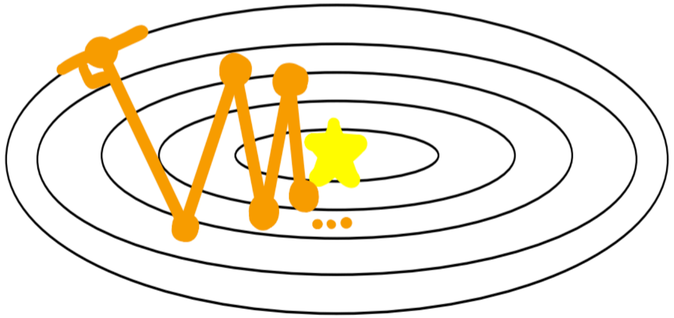
Today's lecture

- 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 $\sigma_d = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_d - \mu_d)^2}$
- 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, this is typically not done at all

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…

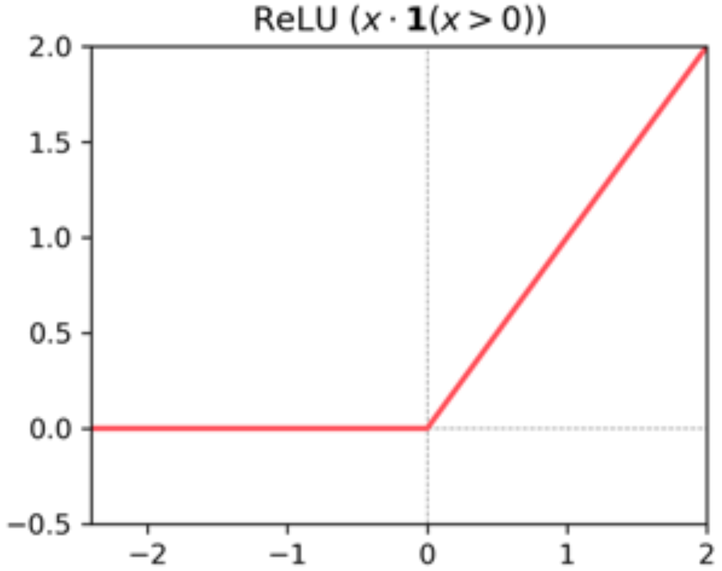
Layer normalization (LN)

- 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

Network architecture choices

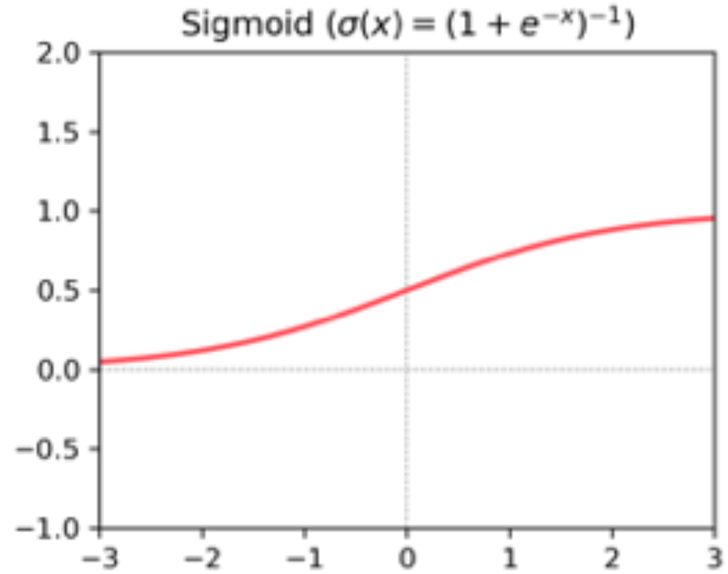
Nonlinearities — rectified linear units (ReLUs)

- $\text{ReLU}(\mathbf{v}) = \max\{0, \mathbf{v}\} = \mathbf{v} \odot \mathbf{1}[\mathbf{v} > 0]$
  - Therefore, $\nabla_{\mathbf{v}} \text{ReLU}(\mathbf{v}) = \text{diag}(\mathbf{1}[\mathbf{v} > 0])$
- 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

- $\text{sigmoid}(\mathbf{v}) = \dfrac{1}{1+\exp\{-\mathbf{v}\}} = \dfrac{\exp\{\mathbf{v}\}}{\exp\{\mathbf{v}\}+1}$

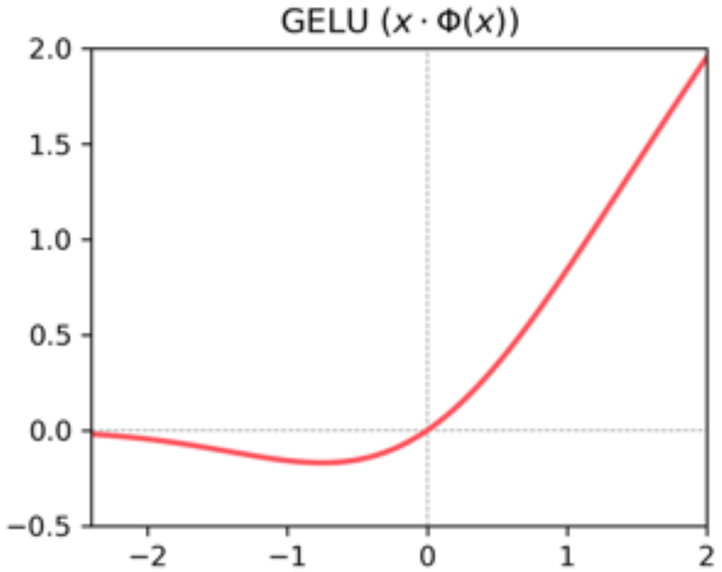


- Along with tanh, has really fallen out of favor as a hidden layer activation
- Why? Very small gradient values for large inputs
  - $\nabla_{\mathbf{v}}\text{sigmoid}(\mathbf{v}) = \text{diag}(\text{sigmoid}(\mathbf{v}) \odot (1 - \text{sigmoid}(\mathbf{v})))$
- Used as the output “activation” for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

Nonlinearities — Gaussian error linear units

GELUs (and friends)

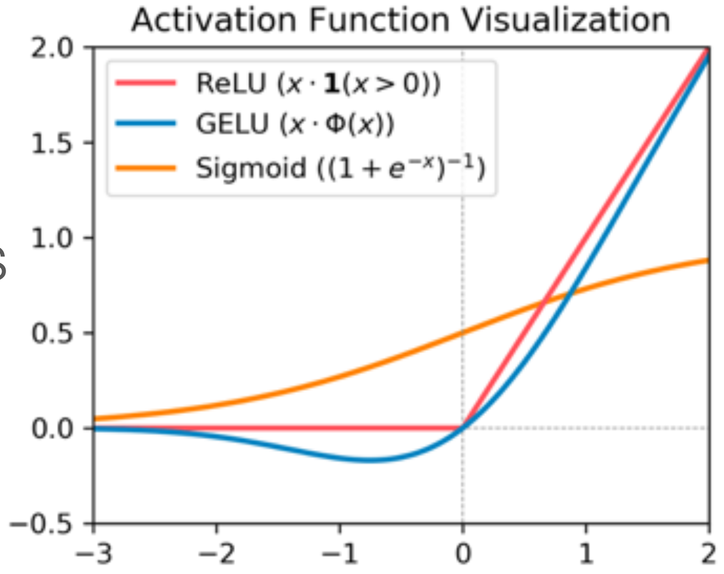
- Both ReLUs and sigmoids have gradient issues
- Another function that sidesteps some of these issues is the Gaussian error linear unit (GELU)
  $\text{GELU}(\mathbf{v}) = \mathbf{v} \odot \Phi(\mathbf{v})$
  - Φ evaluates the CDF of $\mathcal{N}(0, 1)$ element wise



- Closely related to other functions that pass the input through a “soft gate” — e.g., $\mathbf{v} \odot \text{sigmoid}(\mathbf{v})$ is quite similar (sometimes called SiLU or swish)

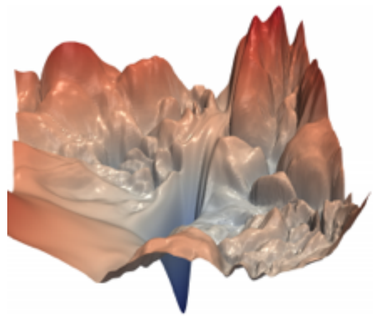
Comparing these nonlinearities

- 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

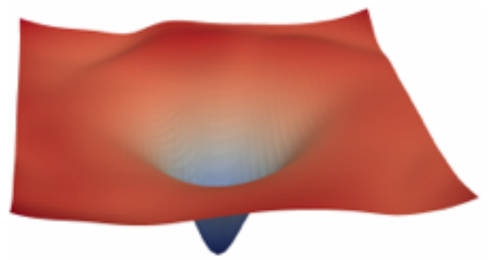


Skip connections

- 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

Dropout

- Often, **dropout** is applied to our model during training
- The basic idea is very simple: 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*
- 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!
- Some care should be taken to make training vs. test output magnitudes consistent

Data augmentations, briefly

We'll talk more about this topic later in the course

- 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://neptune.ai/blog/data-augmentation-in-python

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