EECS 182 Midterm Review

Stephen Krider, Joyee Chen

Saturday, April 8 for the "lecture" of March 20

This is designed as a midterm review sheet. Where possible, at least in the first few weeks, the providing of essential equations is shared with a strong focus on conceptually describing the key topics and ideas – because after all, your midterm cheat sheets will already have a natural tendency to cram themselves with equations under pressure. These reviews, on the other hand, focus as much on the "what" as on the "why".

Contents

Weeks 1-2: the basics of machine learning, optimization, and regularization

A great amount of credit is due to the various Spring 2023 scribes for 182 who I distilled here.

Machine learning: the basics

Week 1 Monday (1/23)
This lecture is, in general, about ML review.

The simplest application of ML is in supervised learning, which is defined roughly as the learning of patterns among datasets where each piece of data already has a label or target.

One paradigm of supervised learning is the optimization-based paradigm, where the basic structures – the basic types of objects – are "training data" and "model". What's the relationship between the two? Training data acts on the model. What are the specifics within the two? Training data must have "labels" attached to each datapoint, and models must be tunable or changeable by selecting certain values for variables called "parameters".

Training via Empirical Risk Minimization: Choose the ideal parameters, such that the average loss is minimized.

However, the word "average loss" is implicitly with respect to an argument called "the data", which may be misaligned with what we really want – minimizing error upon the real world.

This is where the idea of a **mathematical proxy** comes in: an assumption about the underlying data's probability distribution, P(X,Y). The obvious complication: what happens when we aren't just given a P(X, Y), or even have any access to it? Solution: partition or slice all the data into a training-set and a test-set. Then train the model – i.e. fit all the parameters using the training-set only. Finally, calculate test error (basically just average-error) on the test error. The independence generated by not mixing the two sets in either phase means test error should be a good representation of real-world performance.

Loss functions are basically functions mapping vectors, or matrices, or collections of data into a real number. That real number should represent the amount of error sustained by the model with respect to some ground data.

But what if our optimizer's loss function (at least naively) fits the loss function that we want, L_{true}, like a square peg into a round hole? The solution seems to be to define an "adaptor loss" function, or a *surrogate loss*: a function that has all the abstract properties L_{true} needs, but also meets the concrete requirements needed to be pluggable into the optimizer framework.

Overfitting: the problem where your trained model's results (weights) adheres too precisely to a particular dataset. (Like preparing too specifically for the SAT math section instead of building up general math skills, so you get confused whenever any open ended problems come up in college.) *Its symptoms*: crazily precise/crazily large values of weights, often combined with bad performance on a test set. *Its cures*: Adding a regularizer function during training, or choosing to make extra complexity "not exist in the first place" by rewriting your model to have fewer parameters.

Optimizers and Gradient Descent and Neural Networks

Week 1 Friday (1/27)
This lecture is about optimizers and neural networks.

The primal question we'll start today off with is, "How do we tune a model's parameters in the first place?" We'll have to use an optimizer, or metaphorically a "tuning-machine", and the most basic one is gradient descent.

In a nutshell: gradient descent takes in a loss function and a vector of parameters, and then behaves as if the loss function was a slope, and the parameter location was a ball rolling down the slope. More precisely, at each timestep from t=0 (until a specified "close-enough" convergence goal is met), gradient descent will first update the position of (or value of) the parameter vector itself based on one timestep of motion down where the steepest downslope is. Then it will find the training loss for that new position, efficiently based on the aforementioned slope/gradient value.

The devil's in the details, though. One problem with that is in the loss function itself – in order to get a loss value, you have to loop through every single point in the dataset, a real hassle. Literally O(dataset size) time at best.

So what do we do?

Letting n be the dataset size, we can actually calculate the training loss using just a random subset of points $n_{batch} \ll n$, and use that sample to represent the dataset in its entirety.

This seems to have solved so many of our issues. Now let's turn to seeing how the "tuning machine" itself operates.

What does the step size do? It generally determines how fast the gradient descent algo converges, though it's not as simple as "increasing step size = zoom zoom". A great result is that in order to converge *at all*, the learning rate/step size itself has to decay.

On neural networks:

For the thorny question of what a "neural network" actually is, I defer to Mathias Weiden in his scribe notes for Sp2023 Lecture 3: "A neural net is a computation graph/analog circuit that plays nice with automatic differentiation or "back propagation."

So what? Can we clear up the magician's fog here? What use do these so-called "circuits" have?

We want these circuits to do the heavy lifting of prediction or classification for us, of course. And we want to bake into them two qualities:

1. Expressibility, which is being able to "represent a wide variety of functions".
2. Learnability, which is equivalent to speed: "the ability to learn functions without too much trouble."

So how do we approximate such a "wide variety" of functions?

Answer: "linearize them", approximating them by a series of short straight lines!

From this ideal answer comes the real function of ReLUs, which are basically elbows connecting two lines at different angles from each other...connect enough ReLUs together and you can approximate almost any function.

Various views of OLS

Week 2 Monday (1/30)
In this lecture we discuss gradient descent, OLS, and ridge regression.

Reminder: a single fully-connected-layer ReLU network can be represented by $f(x) = W_2 ReLU(W_1 x + b_1) + b_2$.

Let's start with the basics: what's the purpose of regularization in the first place?

It's helpful when you want to bias the optimization to learn a certain property *along with* the patterns that we like, or when we want the optimizaiton to learn a certain property of the *representations* of the patterns.

A favorite example of this is sparsity – favoring more zeros in the vector representation of the learned patterns, i.e. "simpler" models of fewer parameters.

Let's concretize this idea of regularization by considering a standard equation and use case.

The standard equation where regularization is used:

$L = (1/n)\Sigma l_{train}(y_i, f(x_i)) + R$
where R is the regularization term.

The standard use case is that of least squares with ridge regression: adjust w so as to get y closest to Xw, or minimize $(||y - Xw||_2)^2 + \lambda(||w||_2)^2$.

This simple use case can be interpreted in five different perspectives:

1. Perspective 1: Gradient descent:
   $w_{t+1} = (1 - 2\eta\lambda) \cdot w_t + 2\eta X^{\mathrm{T}}(y - Xw_t).$

2. Perspective 2: Leveraging SVD for better understanding via OLS:
   $Xw \approx y$ can be rewritten as $\Sigma\tilde{w} \approx \tilde{y}$, which can be expanded to

   $$\tilde{w}_i = \begin{cases} \frac{1}{\sigma_i}\tilde{y}_i & : i \leq \min(n, d) \\ 0 & : \text{o.w.} \end{cases}.$$

   with two cases for Σ being tall or being wide.

3. Perspective 3: Leveraging SVD for better understanding via Ridge Regression

4. Perspective 4: Gradient Descent

Now out of intellectual curiosity, how can we implicitly use the idea of ridge regularization, without explicitly adding an additive term "$+R$"?

Answer: Implicit regularization!

1. "Adding extra data"

2. "Adding extra features"

Regularization

Week 2 Friday (2/3)
This lecture is on a potpourri of topics surrounding regularization.

We start off where we ended last lecture, on different interpretations of regularization in a least squares use case.

One that didn't come to mind: an argument from probability!

Assume, as a leap of faith, that the we actually know a multivariate normal distribution generates the predictor variables.

Now let's translate this into the language of EECS 126 (upper div probability): what is the MLE of w, the maximum likelihood estimate? (More precisely: given w, what's the maximum likelihood of the data?) Some 126 wizardry shows it's exactly the same as the plain OLS solution.

The ridge regression solution (adding a λI term to the $X^T X$ term) likewise has a probabilistic interpretation as the MAP estimate of w, or the maximum a posteriori estimate. (More precisely: given the data, what's the most probable value of w?)

Another interpretation (that might appeal to some CS 170 takers): kernel ridge interpretation, prominent for its XX^T term in place of the usual.

$w_{ridge} = X^T(XX^T + \lambda I_n)^{-1}y$

You can easily do algebra to show kernel-ridge gives the same equation as regular OLS. But the key motivation for kernel-ridge is its ability to resolve the parameters (our model can learn) as clearly direct linear combinations of data points (which we have).

A corollary: the actual meaning of XX^T itself is a matrix containing just inner products between certain data points, i.e. a similarity metric.

What is inductive bias, in a nutshell? It's the tendency for gradient descent to go in one direction over another. A SVD expansion of gradient descent equation will show that, roughly, gradient descent will "move" the parameter-vector in a decreasing priority of subspaces, where "priority" is given by their singular values.

Week 3: Initialization and advanced optimization

A great amount of credit is due to the various Spring 2023 scribes for 182 who I distilled here.

Initialization and Optimizers

Week 3 Monday (Feb 6)
This lecture is about initialization and optimizers, leading up to the concept of momentum.

In particular, we start off with the little-considered question of how to initialize ReLU weights w and b. (Reminder: elbow location at $e = -b/w$.)

But what are our **goals** for an ideal ReLU initialization again? We want the input to each ReLU layer to be centered near 0, to effectively discriminate between points...so a next preliminary goal might be to initialize ReLU weight w to N(0, 1).

(*Important: the next two paragraphs are just about initializing w, not the bias b.*)

First attempt: Xavier initialization, where we try to twiddle weights so as to standardize variances across each layer...more specifically, choosing them i.i.d. from N(0, 1/d). Here, we face the significant problem of actually misjudging the variance-sum by a factor of two because on average, half of the outputs are 0 so the real number of inputs that actually behave like **random** variables is d/2.

Second attempt: He initialization (Kaiming initialization). Basically the same as Xavier initialization...but drawn from N(0, 2/d) solving the problem!

Optimization w.r.t. learning rates and singular values

Week 3 Friday (Feb 10)
This lecture focuses on optimization in a deep learning context, as opposed to a traditional context.

What's the big difference, you might say?

In traditional optimization, according to the Lecture 7 scribe notes, "the objective function is exactly what we would want to optimize over." But in deep learning, we're often forced into introducing a surrogate loss, a middleman between the raw training data and the clean loss metric; that surrogate loss renders the idea of "optimizing" more vague.

Also, convergence speeds are more severe of a matter in neural networks.

Consider learning rates for example, on the simple OLS gradient descent. We know that the convergence rate for the ith dimension is $|1 - 2\eta\sigma_i^2|$, and in general, smaller values of that absolute value imply faster convergence (error decaying quicker).

Amidst such seeming certainty, let's shift our focus to underdetermined cases (read: non-square data matrices). In the case of "billion of parameters, just a few points", EECS 127 (upper div optimization) showed us that blindly applying OLS formula resulted in min-norm solution value of w.

As it turns out, gradient descent (with parameters initially at 0) also converges to a min-norm solution (an argument you can make by seeing everything "lives" in the plane of $col(X^T)$).

Can you see how a similar argument applies to SGD?

But we aren't out of the woods yet when it comes to gradient descent. Remember the $|1 - 2\eta\sigma_i^2|$ equation for convergence rate? If we make the learning rate small, convergence will be slow, but if we make the learning rate too large, then combining that with large singular values can create oscillation at high frequency among successive w. This problem leads us to the abstract solution of a low pass filter. This solution has often been concretely implemented as an exponential moving average of the input.

Such an exponential moving average can be encapsulated in the idea of **momentum**: a weighting system that adjustably trades off current gradient against previous gradients.

$$\vec{a}_{t+1} = (1-\beta)\vec{a}_t + \beta\vec{u}(t)$$
$$\vec{w}_{t+1} = \vec{w}_t - \eta\vec{a}_{t+1}$$

(Source: Spring 2023 CS182 Lecture 7 scribe notes)
Here the a terms represent a summary statistic or weighted average of all the gradients so far, and β represents the amount of importance you think the weighted average should give to the current gradient.

But a subtle problem still arises: even though momentum ensures representation across all different gradients, it doesn't have good representation across all the different singular values, with only σ_{max} and σ_{min} represented. This justifies the idea of **adaptive momentum**, or **adam**, method:

$$\vec{a}_{t+1} = (1-\beta)\vec{a}_t + \beta\nabla l(\vec{w}_t)$$
$$\vec{v}_{t+1} = (1-\beta')\vec{v}_t - \beta' \begin{bmatrix} (\frac{\partial}{\partial w[1]} l(\vec{w}_t))^2 \\ \vdots \\ (\frac{\partial}{\partial w[n]} l(\vec{w}_t))^2 \end{bmatrix}$$
$$\vec{w}_{t+1}[i] = \vec{w}_t[i] - \eta \frac{\vec{a}_{t+1}[i]}{\sqrt{\vec{v}_{t+1}[i]} + \varepsilon}$$

(Source: Spring 2023 CS182 Lecture 7 scribe notes)

Week 4: Conv nets, batch norms, and dropout, oh my!

A great amount of credit is due to the various Spring 2023 scribes for 182 who I distilled here.

Conv Net Basics

Week 4 Monday: Feb 13
In this lecture we'll move from the microscale to the macroscale: from particularly simplified and abstractified models of how an idealized weight variable gets changed, to more real-world models of where learning can be used with less reference to particular algorithms in their entirety.

Consider the case of convolutional neural networks (convnets, or CNNs). You might associate them with the image classifiers used to locate the faces in self driving cars or cute kitty photos.

They have three great properties that will motivate anything designed on them:

1. 1) Respect locality
2. 2) Respect invariances/equivariances within dataset
3. 3) Support a hierarchical structure and multiple resolutions

But those ideals do not help us actually visualize a convnet. What does: convnets are based around the notion of a filter, an array or matrix (often called h) which is dragged along all parts of a signal/image. At each stop along the dragging, the filter gets convolved with the local signal/pixel values (a form of dot product) and the resulting product becomes the output of a convnet. Or at least one layer of it.

The nifty equation:

$$o(t) = (u * h)[t] = \sum_{\tau=-\infty}^{\infty} u[\tau]h[t-\tau]$$

where u is the input signal, h is the filter (also called a "kernel"), and o is the output.

The real trick, that turns this filter-and-dragger notion from a skeleton into an evolving organism, is that *the filter values are learnable.*

Let's try to visualize a more concrete notion of a ConvNet.

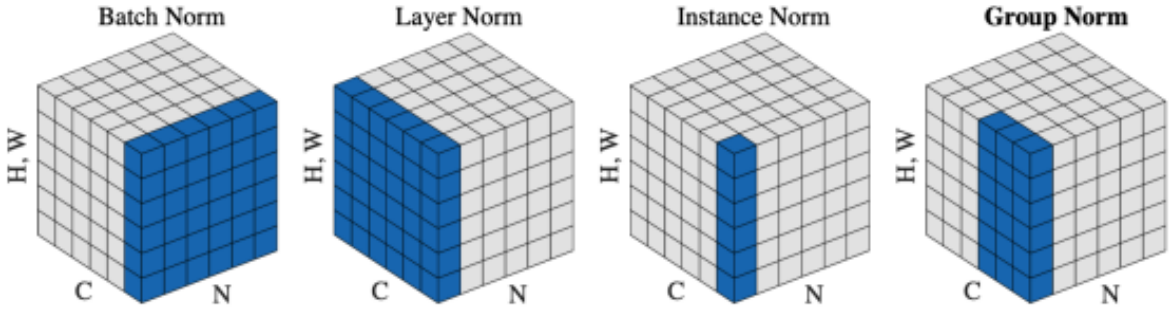
Suppose we have a 2D picture. Those are 2 dimensions, but when working with color images, we'll add a third dimension called depth (with "channels" red, green, and blue). But a great workaround is that one can collapse height and width into a single dimension, by "lining up" all the pixels in a certain order: left to right first row, then left to right second row, ...

Now consider a 2D video of many frames of pictures. Here the notion of "batch" manifests itself as a collection of objects of type "entire picture", rather than "pixel": we can choose all the frames from, say, the 0-second mark to the 10-second mark as one batch, from the 10 to 20 second mark as batch 2, ...

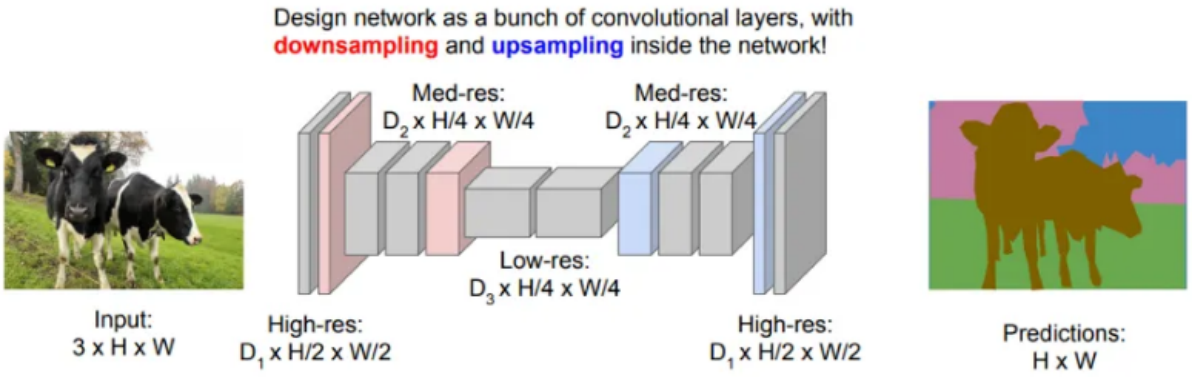
Now, for this 2D video case, we can represent an entire dataset – an entire video – as a 3 dimensional rectangular block: pixel position (height and width combined) as dimension 1, channel as dimension 2, and batch as dimension 3.

Now that we understand how inputs are represented, lwe can see how they are normed, or grouped together in certain ways depending on certain processing needs:

Figure 1: Different types of norms (Source: CS189 Fa22 Lecture 9 Scribe Notes)



Convolution Output Size Considerations
But still, we haven't yet touched the matter of a convolution's output size. Think first of an idealized filter, a one-dimensional strip.
That convolution will likely have characteristics like **stride**, the number of spaces the filter skips over from one convolving to another, and **padding**, the number of spaces on the edges (both edges!) of the input signal that ought to be replaced with a standard number (usually zero). Then, by a simple counting argument, the **dimension of the output** can be shown to equal
$\frac{W-K+2P}{S}+1$
where W is the input dimension.
Downsampling and Upsampling (source: "Transposed Convolution Demystified", towardsdatascience.com)



In general, downsampling is the extraction of information, densifying the amount of information from a picture to a smaller one; upsampling can be said to be the sparsifying of information, copying it and distributing it over a wider area.

Batch Norm and Dropout

Week 4 Friday: Feb 17
In this section we focus on the concepts of batch normalization as a sort of "classical" improvement to convnets, but then shift our attention to the much more "modern" improvement generalizable across all neural networks: dropout.

Standard normalization, upon a dataset D, has the goal of making the elements of D have mean equal to zero and variance equal to 1 (with many statistical advantages and efficiencies resulting from it). This is classically done by calculating the mean and standard deviation of the entire D, then subtracting the mean from all D's points, and dividing the result by the st dev.

Batch normalization is a way to make standard normalization yet more efficient in the computation stage: because calculating means and variances are expensive, we can just calculate them both from a randomly chosen batch of datapoints B from D, instead of D itself. Then do normalization *using those mean/sd values as though they represented all D.*

Let's flesh this out with different colors, courtesy of Geleta et al. in their scribe notes for Lecture 9:

The essence of batchnorm (BN) lies in computing $\hat{x}$ from x in the test set, both represented by blue, but normalizing using means and standard deviations in the *training set*, both in *red*.

$$\tilde{\boldsymbol{x}} = \frac{\boldsymbol{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

A motivating problem: a single batch might not represent the entire dataset very well (think of picking 30 people at random from the American population!).
A partial solution: pick multiple different batches B_k from D, and try to include them all in normalization. But how do we "include them all"? That's the motivating question for our next BN refinement: compare x to the average of the batch-means and batch-SDs!

$$\tilde{\boldsymbol{x}} = \frac{\boldsymbol{x} - \frac{1}{K}\sum_{k=1}^{K} \boldsymbol{\mu}_k}{\frac{1}{K}\sum_{k=1}^{K} \boldsymbol{\sigma}_k}$$

A further refinement to BN can come from the concept of normalizing x by the mean and sd of the last *epoch* of training:

$$\mathrm{BN}(\boldsymbol{x}) = \frac{\boldsymbol{x} - \boldsymbol{\mu}^{(t)}}{\boldsymbol{\sigma}^{(t)}}$$

And for the greatest flexibility, in those rare cases you want to stretch your data to a known SD that isn't 1, or a known mean that isn't zero, you can add *trainable* parameters γ and β like so:

$$\mathrm{BN}_{\gamma,\beta}(\boldsymbol{x}) = \gamma\frac{\boldsymbol{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} + \beta$$

Dropout addresses a different problem. It is all well and good to start training on fully connected layers, but that feels somewhat like saying "everything in life is causally connected, in some way or another"...it does not bode well for good interpretability of the model, when a sparser model with fewer causal links will capture all that is needed.

The gist of dropout is to go over all the output of neurons, and randomly set some of those outputs to zero. The probability of zero-setting, p, is a hyperparameter you can control or optimize.

On a deeper level, dropout can be motivated in terms of the resilience it brings to neural network training as a whole: instead of the brittleness of one model when it comes to the bias variance tradeoff, running dropout multiple times can lead to creating a horde of different models on one dataset, each slightly jittered – and the advantages of CS189 style ensemble learning ensue.

Week 5: ResNets and ConvNeXT

A great amount of credit is due to the various Spring 2023 scribes for 182 who I distilled here.

ResNets and ConvNeXT

Week 5 Monday: Feb 20
In this section I discuss ResNets in theory and in practice.

The essence of ResNets is the residual connection, a sort of express train amongst the local trains that is the general layer-by-layer connection: residual connections skip the process of having info passing and likely modified through different layers, and instead have the info travel directly from the input to the output.

Now we ask, what do we put in the F block? The two choices discussed in lecture, at least:

Thankfully, the choice of the best (for trying to model a function itself) is rather more down to a technicality: the left block directly outputs from the ReLU's teat, so can only emulate functions with nonnegative outputs! But the right block outputs from a weighted output of convolutions, so can model any real-output function.

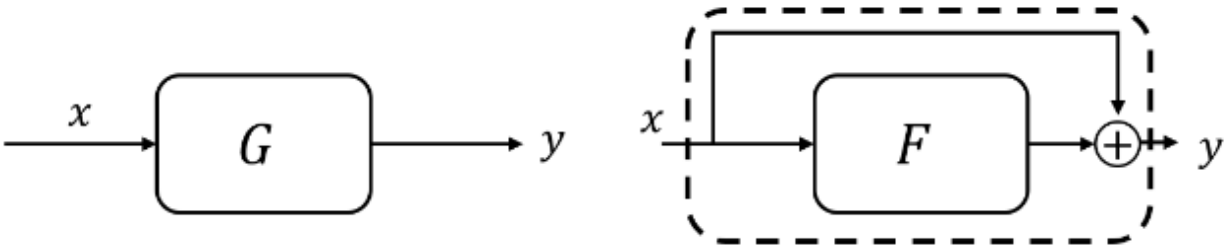


Figure 2: The essence of resnet: a skip-connection (Sp23 Lecture 10 scribe notes)

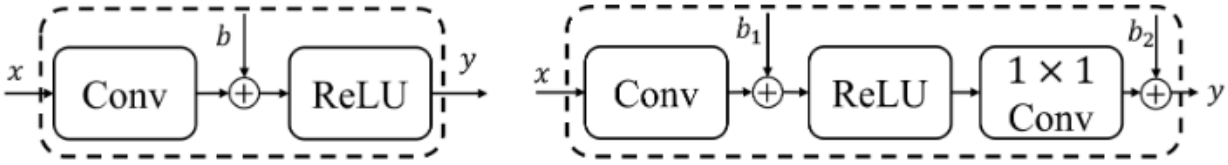
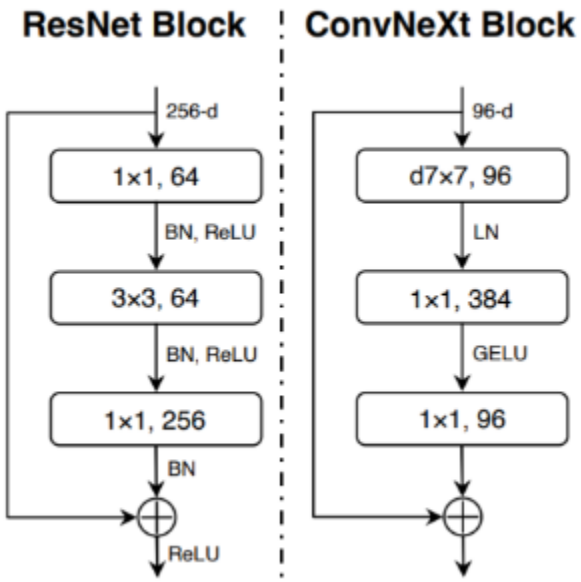


Figure 3: Two proposals for architectures (Sp23 Lecture 10 scribe notes)

ConvNeXT is considered a radical improvement on ResNet, with only surface level architectures the same. ResNet vs ConvNeXt architecture (Sp23 Lecture 10 scribe notes):



In particular, a new activation function called the GeLU (Gaussian ReLU) has been introduced: $GeLU(x) = x\Phi(x)$ for $\Phi(x)$ being the Gaussian cdf of x.

Week 6-8: Graph Neural Networks, RNNs, and attention

Graph Neural Networks

Graph neural networks are a type of neural network designed to work with graph-structured data.

Definition. A graph $G = (V, E)$ consists of a set of **vertices** V and **edges** $E = \{(u, v) : u, v \in V\}$. We often define **labels** on E and/or V: $w_E(e) : E \mapsto \mathbb{R}^d$, $w(v) : V \mapsto \mathbb{R}^d$.

The purpose of a GNN is to infer some value from the graph's labels. In practice, this task manifests in two forms:

- **Graph-Level Tasks**. Inferring a value from the entire graph.
- **Vertex/Edge-Level Tasks**. Inferring a value for each edge/vertex.

Where our task is to learn θ given training data.

> Note that sometimes we have only one single graph to work with - for instance, a social network. In this case, we split the graph into $(V_{\text{test}}, E_{\text{test}})$ and $(V_{\text{train}}, E_{\text{train}})$.

Aggregation Functions

GNN layers have the inductive bias of spatial locality, that all relevant information about a node is contained in its local neighborhood. Formally, we wish to learn an **aggregation function** f_θ:

$$f_\theta(v, N) : v \in V, N = \{u : (v, u) \in E\} \tag{1}$$

Where N is the **neighborhood** of v. Importantly, f_θ must be **permutation invariant** in N, as we have the inductive bias that E, V are unordered.

Possible aggregation functions:

1. Sum aggregation:

$$f(v, N) = w_0^T v + \sum_{u \in N} w^T u \tag{2}$$

2. Mean aggregation:

$$f(v, N) = w_0^T v + \frac{1}{|N|} \sum_{u \in N} w^T u \tag{3}$$

3. Max aggregation:

$$f(v, N) = w_0^T v + \max_{u \in N} w^T u \tag{4}$$

Note the similarity to convolutional neural networks in that we learn a single function / filter / kernel that is then applied to every position.

GNN Layers

A single GNN layer consists of multiple learned aggregation functions, mapping multiple input channels to multiple output channels in much the same way was CNNs. Additionally, we incorperate **nonlinearities** to increase the expressivity of our network.

Depth in a GNN has an important implication. As a GNN grows deeper, the **receptive field** of the network increases. For example, a 2-layer GNN can take into account information from a node's "two-hop" neighbors.

We also incorporate **residual connections**. Formally:

$$x^{(\ell)} = f^{(\ell)}(x^{(\ell-1)}, N) + x^{(\ell-1)} : x \in V \tag{5}$$

Residual connections assist in allowing the gradient to flow to earlier layers in the network without "dying" or "blowing up".

Graph Representation

Defintion. The adjacency matrix of a graph is defined as

$$[A]_{ij} = \begin{cases} 1 & \text{if there is an edge from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

A need not be symmetric, such as when G is directed

Definition. The degree matrix of a graph is defined as

$$[D]_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

We use the degree matrix to define $A^{\text{Normalized}} = AD^{-1}$ and \$Aˆ SymNorm = D{-1/2}AD{-1/2} **Definition.** The Laplacian matrix of G is $L \doteq D - A$. Note that L_i sums to zero. Additionally, we define $L^{\text{SymNorm}} = I - A^{\text{SymNorm}}$.

The Laplacian provides a convenient way of representing the graph as a matrix. Importantly, A^k has $[A^k]_{ij} = n$ where n is the number of ways j can be reached with exactly k hops from i.

> With our definition of $A \in \mathbb{R}^{N \times N}$ and vertex labels $X \in \mathbb{R}^{N \times C}$, applying the sum aggregation function to G is similar to a convolution of dimension $1 \times N$ on AX. This establishes the parallel between GNNs and CNNs.

Normalization

Layer norms in GNNs are similar to layer norms in CNNs. In GNNs, we normalize across batch entries and nodes, keeping channels separate.

Loss in RNNs

One important design choice in RNNs involves deciding whether to consider only the final hidden state or the entire sequence. In the former case, we use a loss function that is only dependent on the final hidden state. In the latter case, we use a loss function that is dependent on the entire sequence.

In the former, we may often run into issues where earlier inputs have less of an impact on model output.

Pooling

In GNNs, pooling is a technique applied to downsample the graph structure and reduce the computational requirements of the network. Usually, it involves merging a node into its neighbors. However, it is often domain-specific, so for our purposes we assume the dimensionality of the graph stays constant.

RNNs

RNNs are generalizations of a learned Kalman filter. For more information about Kalman filters, see https://people.eecs.berkeley.edu/~pabbeel/cs287-fa15/slides/lecture14-KalmanFiltering.pdf

$$\hat{h}_t = W_{\hat{h},\hat{h}}\hat{h}_{t-1} + W_{\hat{h},x}\vec{x}_t + \vec{b}$$

Where we learn $W, \vec{b}$ from data, modeled as one of the following:

1. An $\left\{(\vec{h}_t, \vec{x}_t)\right\}$ sequence.
2. An x_t sequence, where x_{t+1} is predicted conditioned on \$h_t−3. $\{x_t, z_t\}$ where z_t is a linear function of \$h_t To generalize to RNNs, we add a nonlinearity - for example, rather than h_t as an affine function in x_t we use a MLP.

To increase expressivity, we have a few options:

- Increase the dimensionality of the hidden state
- Compose RNN filters (standard approach)

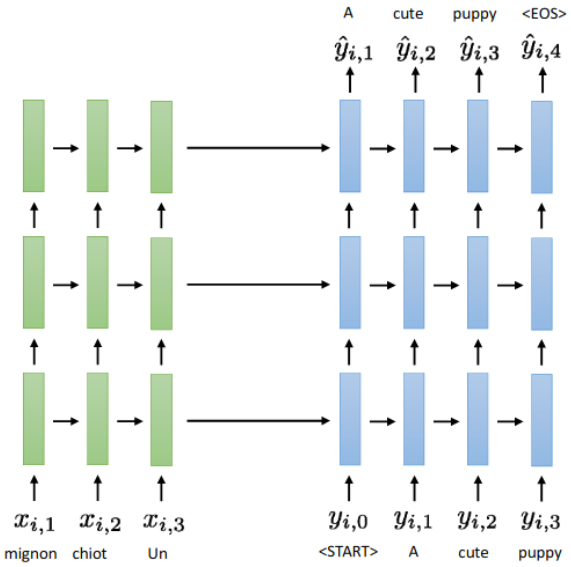


Figure 4: A composed RNN filter. Source: EECS 189 Fall 22

Nonlinearities

Main challenge in RNNs: very long chains of input/output to backprop through, leading to dying or exploding gradients. Exploding gradients prevent the network from learning anything, while dying gradients prevent the network from handling long-range dependencies.

To handle this, we employ **saturating nonlinearities** (figure 2). Rather than using a nonlinearity such as ReLU with no upper bound, we use a nonlinearity like sigmoid or tanh.

Because saturating nonlinearities are upper-bounded, gradients are less susceptible to explosion.

Normalization in RNNs

Why not just normalize the whole output of an RNN layer? A couple subtle reasons why not:

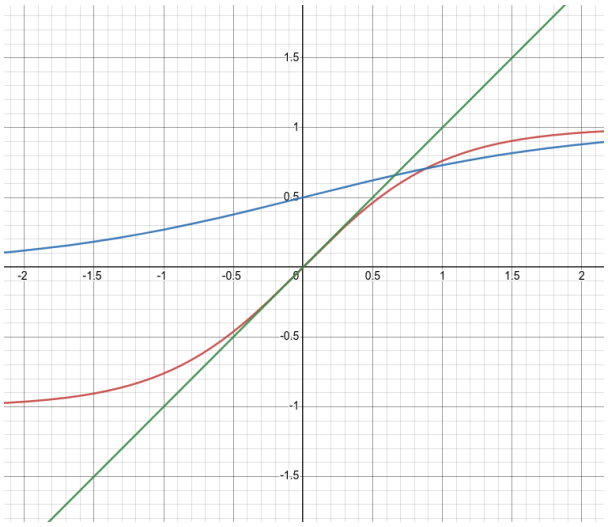


Figure 5: Sigmoid (blue), tanh (red), "ReLU" (green)

- Outputs at a timestep depends on future inputs - could be an issue at inference time
- Could we layer norm the entire hidden state at each timestep? Or perhaps a batch norm?
- Batch norm is possible and done occasionally, but is hard to reason about as sequences are variable length

  Intuition: The layer-norm of single hidden state is like forcing all $\vec{h}_i$ to lie on a sphere (or with learned μ, σ, an ellipsoid)

Beam Search

In top-k beam search, at each time step t, the algorithm maintains a set of k partial sequences (called beams or hypotheses) with the highest probabilities. At each time step, all possible extensions of the k "beams" are calculated, and only the top k are kept.

The top-k beam search algorithm can be summarized as follows:

1. Initialize the set of beams with an empty sequence, i.e., $B_0 = \{()\}$.
2. For each time step $t = 1, 2, \ldots, T$:
   a. For each beam $b \in B_{t-1}$:
   b. Compute the conditional probabilities $P(y_t|b, x)$ for all possible tokens y_t.
   ii. Extend the beam b by appending each token y_t and calculate the probability of the extended sequence using the conditional probabilities.
   b. Select the top k extended sequences with the highest probabilities and set B_t to these sequences.
3. Return the sequence in B_T with the highest probability.

Top-k beam search strikes a balance between exploration and exploitation. However, top-k beam search is not guarantueed to find the optimal solution.

Attention

We learn keys, queries, values, and output sum of values weighted by the softmax of the inner products of keys and queries

$$
\begin{aligned}
x_i &\to k_i, v_i \\
y_i &\to q_i \\
e_j &= \frac{\langle k_j, q_j \rangle}{\sqrt{d}} \\
\alpha_i &= \frac{\exp e_i}{\sum_j \exp e_j} \\
a_i &= \sum_j \alpha_j v_j
\end{aligned}
$$

The $\sqrt{d}$ is added to normalize the e_j to variance 1, assuming random key, query vectors.

We have a traditional recurrent neural network-based encoder-decoder setup. How do we add attention?

We take the hidden state output for each decoder cell, learn a query from it, and then learn a key/value pair for each encoder cell at each layer. Then concatenate the output from the attention with the output from the hidden state, and input to the next layer.

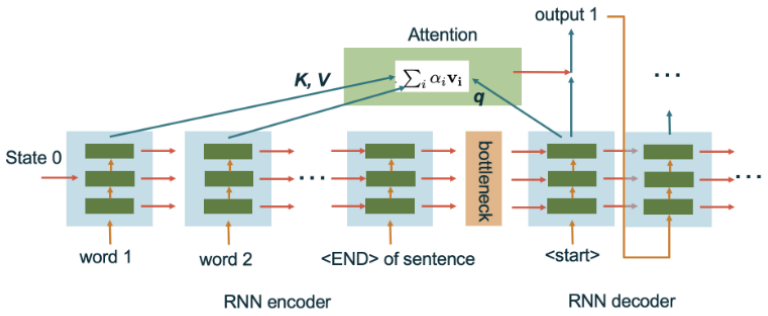


Figure 6: Diagram of attention. Source: Fa22 Scribe, Lecture 14

Definition. Self-attention is where the decoder queries itself.

Definition. Cross-attention is where the decoder queries the encoder.

Positional Encoding

The inputs to the attention are an unordered set. How can we add positional information? We use "circular time" and express time as a sum of sinusoids. This is known as **rotary encoding**.

$$
\begin{bmatrix}
e^{i\omega_1 t} \\
e^{i\omega_2 t} \\
\vdots \\
e^{i\omega_N t}
\end{bmatrix}
$$

Why can't we use different ω, perhaps depending on the length of the input sequence? This would introduce issues with different sequences having different positional embedding schemes.

Self-Supervision / Autoencoders

> "If you have a strong inductive bias, then you can get away with less data - if your inductive bias is weak, then you need more data"

How can I train a network with unlabeled data?

> "Self-supervised learning is unsupervised learning"

"Data Mining" is the beginning of this - we have data, but don't know any patterns in it - how do we find them?

Two classical approaches to unsupervised learning:

1. PCA, t-SNE - Dimensionality Reduction
2. k-Means - Clustering

Consider PCA. Justified by the Eckert-Young-Mirsky theorem:

Theorem. The best rank-k approximation to a matrix X is $U\Sigma_k V^T$. Formally,

$$U\Sigma_k V^T = \arg\min_{\hat{X}} ||X - \hat{X}||_F$$

Note that our objective function here is a kind of loss - is there therefore a way that we can formulate this into a problem solveable by automatic differentiation and gradient descent?

However, the constraint is rather difficult to relax, meaning we can't use pure gradient descent to solve the problem.

Idea: compress x into a latent space, and then expand it back out to the original space again, taking the squared reconstruction error as a loss.