CS 182

Preston Fu

Fall 2023

These are course notes for the Fall 2023 rendition of CS 182, Deep Neural Networks, by Prof. Anant Sahai, i.e. a summary of the lecture videos. They are a strict subset, covering maybe half the material (that's pretty generous).

Contents

2 Machine Learning Review (08/28)

The corresponding video is here.

What is Machine Learning?

Concept 2.1. The ML setup is as follows: we are given some training data $\mathcal{D} = \{(x_i, y_i)\}$, a model $f_\theta(\cdot)$, and loss $\ell(\hat{y}, y)$. Our goal is to achieve good performance in the real world.

For example, one possible way of trying this is **empirical risk minimization**,

$$\hat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^n \ell_{\text{train}}(y_i, f_\theta(x_i)).$$

Unfortunately, this suffers because this bases our performance on our data rather than the true objective. Instead, we can frame our optimization problem as follows: make an assumption about the underlying distribution of the data, and call it P. Then our goal is to compute $\arg\min_\theta \mathbb{E}_{X,Y\sim P}[\ell(Y, f_\theta(X))]$.

We (briefly) discuss several complications of this setup and the typical solution:

1. We have no access to P. So we partition $\mathcal{D} = \mathcal{D}_{\text{train}} \sqcup \mathcal{D}_{\text{test}}$ and use the test error as a faithful representation of the real world.
2. Our loss function ℓ_{true} (e.g. Hamming loss for binary classification) might be incompatible with our optimizer. So we use a **surrogate loss** ℓ_{train} that satisfies the conditions of the optimizer and evaluate the true loss with ℓ_{true}.
3. We might get crazy values for $\hat{\theta}$ or bad overfitting. Adding an explicit **regularizer** $R(\theta)$ (e.g. $\lambda \|\theta\|^2$) or changing the model handles this.
4. A **hyperparameter** can be considered something that, if handled directly by the optimizer, will go crazy. For example, using λ as a parameter in ridge regularization will probably set it to $-\infty$. Often there are dozens of hyperparameters, so we blindly look at others' because it is intractable to grid search over more than 4 or 5 of them.

Gradient Descent

Recall the Taylor expansion $L_{\text{train}}(\theta_t + \Delta\theta) = L_{\text{train}}(\theta_t) + \frac{\partial}{\partial\theta} L_{\text{train}}|_{\theta_t} \Delta\theta + o(\|\Delta\theta\|^2)$. We want to move in such a way that maximizes the change in the loss, i.e. move in the negative direction of the gradient. So we update

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L_{\text{train}}(\theta).$$

An example of $L_{\text{train}}(\theta)$ might be $\frac{1}{n}\sum_{i=1}^n \ell_{\text{train}}(y_i, f_\theta(x_i)) + R(\theta)$. But usually n is quite large, and computing this summation is time-consuming. So we instead use stochastic gradient descent, using a sample of size $n_{\text{batch}} \ll n$.

Neural Networks

We provide a bare-bones example of a "universal function approximator." For our baseline, observe that you can approximate any function linearly, i.e. you can express it as the sum of some $\max(0, wx + b)$ for some $w, b \in \mathbb{R}$. So in general, you can optimize W and $\mathbf{b}$ so that $\text{ReLU}(W\mathbf{x} + \mathbf{b}) \approx \mathbf{y}$.

3 Gradient Descent, Regularization, SGD (08/30)

The corresponding video is here.

Least Squares

Recall that the problem is computing $\arg\min_{\mathbf{w}} \|X\mathbf{w} - \mathbf{y}\|^2$. The OLS solution is $(X^\top X)^{-1}X^\top \mathbf{y}$, and the ridge regression solution is $(X^\top X + \lambda I)^{-1}X^\top \mathbf{y}$. The idea behind ridge is that the inverted term is positive definite, and thus more numerically stable.

Suppose that for some reason we want to run gradient descent. Then the update step for OLS is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta 2X^\top(y - X\mathbf{w}_t),$$

and that for ridge regression is

$$\mathbf{w}_{t+1} = (1 - 2\eta\lambda)\mathbf{w}_t + \eta 2X^\top(\mathbf{y} - X\mathbf{w}_t).$$

The latter is "weight decay," which draws $\mathbf{w}$ toward zero.

Warning 3.1. It is a common mistake to apply both an explicit regularization and enabling weight decay. They are redundant and will result in more of a weight decay effect than you might've intended.

SVD Perspective

Let $X = U\Sigma V^\top$, where $U \in n \times n$, $\Sigma \in n \times d$, and $V \in d \times d$. Changing coordinates, let $\tilde{\mathbf{w}} = V^\top \mathbf{w}$ and $\tilde{\mathbf{y}} = U^\top \mathbf{y}$, so that $\Sigma\tilde{\mathbf{w}} \approx \tilde{\mathbf{y}}$. In either the wide or tall case, one can do some math to compute

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

In the ridge regression case, we find

$$\tilde{w}_i = \begin{cases} \frac{\sigma_i}{\sigma_i^2 + \lambda}\tilde{y}_i & i \le \min(n, d) \\ 0 & \text{else.} \end{cases}$$

Thus, $\lambda \ll \sigma_i$ makes $\tilde{w}_i \approx \frac{1}{\sigma_i} y_i$, i.e. the usual solution from OLS. If instead $\lambda \gg \sigma_i$, then $\tilde{w}_i \approx \frac{\sigma_i}{\lambda} y_i$, preventing the case of blowup for small singular values.

Gradient Descent

We have

$$\mathbf{w}_t = \mathbf{w}_{t-1} + 2\eta\Sigma^\top(\tilde{\mathbf{y}} - \Sigma\tilde{\mathbf{w}}_{t-1}).$$

Coordinate-wise, the update is

$$\mathbf{w}_t[i] = \mathbf{w}_{t-1}[i] + 2\eta\sigma_i(\tilde{\mathbf{y}}[i] - \sigma_i\tilde{\mathbf{w}}_{t-1}[i]).$$

The interpretation is that gradient descent fits first in the largest σ_i direction. For small σ_i (that give large solutions), we will converge to something very large, but very slowly.

4 Gradient Descent, Regularization, SGD (09/06)

The corresponding video is here.

Initialization

Question 4.1. Suppose you have a two-layer MLP with ReLU activations. What happens when you initialize all weights to zero?

Answer. All the gradients are zero. This only happens when you have a neural net with depth > 1.

As a result, we want a principled way of initializing them. (Treating them as hyperparameter is too expensive for deep learning.) For some tasks, someone else might've already found decent weights for a similar problem, and you can use them as a starting point. Otherwise, suppose they are distributed as iid Gaussians. We still have to choose the corresponding mean and variance. How can we do so?

Recall that in standard machine learning, we usually normalize our inputs to mean 0, variance 1, so that large singular values are representative of important features, rather than a misnomer of the feature simply containing large values.

In a similar fashion, we may want to initialize weights such that $\mathcal{N}(0,1)$ inputs result in $\mathcal{N}(0,1)$ outputs. **Xavier initialization** does this, distributing weights according to $\mathcal{N}(0, \text{indegree}^{-1})$.

Remark. Xavier initialization was originally justified with a similar Bayesian justification as sigmoid. However, sigmoid, arctan, etc. suffer from saturation, i.e. your derivatives go to zero. Generally, people like to use ReLUs for this reason.

However, this also doesn't work well sometimes; recall that the elbow is at $-b/w$. But since w has a high probability density at 0, the tails are heavy. If the elbow is too high, it's possible for the gradients to be stuck at zero. (We call this a "dead ReLU.") In practice, as you train more, you'll end up with more dead ReLUs.

Note that in the real world, half your (normalized) data will be negative, and thus the corresponding ReLUs will die instantly. We fix this with **He initialization**, i.e. taking $\sigma^2 = 2\text{indegree}^{-1}$.

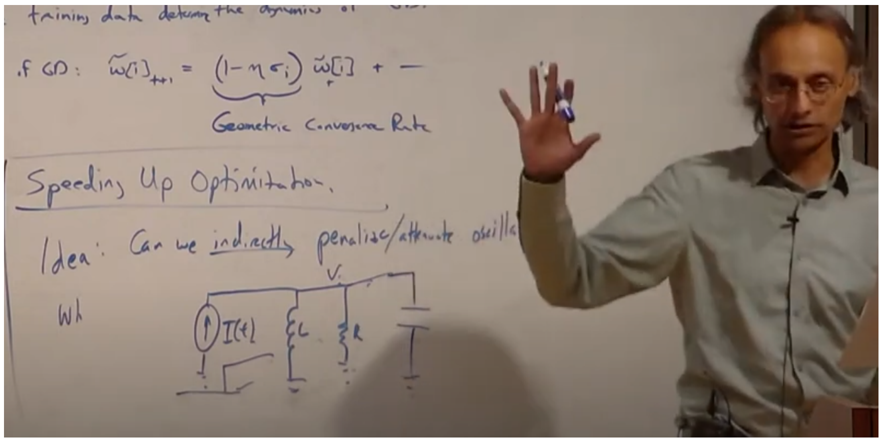
training data determine the dynamics of
of GD: $\tilde{\omega}[i]_{t+1} = \underbrace{(1-\eta\sigma_i)}_{\text{Geometric Convergence Rate}} \tilde{\omega}[i] + -$
Speeding Up Optimization.
Idea: Can we indirectly penalize/attenuate oscilla
Wh
V
I(t)
L
R

Figure 1: What the fuck is this man doing?

Optimization

Last time, we considered gradient descent from an SVD perspective. The update rule looks like

$$\tilde{\mathbf{w}}_{t+1}[i] = (1 - \eta\sigma_i)\tilde{\mathbf{w}}_t[i] + [\text{something}].$$

In particular, we want convergence to occur at a consistent rate, so we want to choose η such that $|1 - \eta\sigma_{\max}| = |1 - \eta\sigma_{\min}|$. We'll see this again on the homework.

Suppose we want to speed up optimization. The idea is to reduce oscillation by applying a low-pass filter. Sahai represents this in a scuffed analogy in Figure 1. The answer is **momentum**.

The idea, then, is to add a capacitor, or add "memory/state" to learning dynamics. Our update looks like

$$\begin{aligned} \mathbf{a}_{t+1} &= (1-\beta)\mathbf{a}_t + \beta\nabla\ell(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \eta\mathbf{a}_{t+1} \end{aligned}$$

We get more details when we convert to SVD coordinates in homework.

5 Adam, SGD (09/11)

Recall that GD with momentum is given as follows:

$$\begin{aligned} \mathbf{a}_{t+1} &= (1-\beta)\mathbf{a}_t + \beta\nabla L(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w} - \eta\mathbf{a}_{t+1}. \end{aligned}$$

Alternatively, you can use "Nesterov" momentum, with update $\mathbf{a}_{t+1} = (1-\beta)\mathbf{a}_t + \beta\nabla L(\mathbf{w}_t - \eta(1-\beta)\mathbf{a}_t$. (Take the gradient where you know you'll end up.) In practice, this doesn't make too much a difference in deep learning.

Adam

Observe that GD with momentum doesn't change behavior if we rotate coordinates or translate; we are still limited in the directions of small gradients.

The idea is to pretend that the coordinates are independent, and as such you can use an adaptive learning rate for each coordinate. Conceptually, we want small gradients to have larger learning rates.

$$\mathbf{v}_{k+1} = (1-\beta')\mathbf{v}_k + \beta' \nabla_{\mathbf{w}} \begin{bmatrix} \vdots \\ \left(\frac{\partial L}{\partial w_i}\right)^2 \\ \vdots \end{bmatrix}$$

$$\mathbf{a}_{k+1} = (1-\beta)\mathbf{a}_k + \beta \nabla L(\mathbf{w}_k)$$

$$\mathbf{w}_{k+1}[i] = \mathbf{w}_k[i] - \eta \frac{\mathbf{a}_{k+1}[i]}{\sqrt{\mathbf{v}_{k+1}[i]} + \varepsilon}$$

SGD

Instead of $\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell_i(\mathbf{w})$, we use a minibatch of size k where we randomly sample from all n. The idea is that the loss is the same in expectation, and likewise for its gradient. In practice, you just randomly shuffle the data and take batches in a loop.

Example 5.1. Suppose we have an invertible X and want to solve for $\mathbf{w}$ with $X\mathbf{w} = \mathbf{0}$. Let the rows of X be $\mathbf{x}_i^\top$, and let $\lambda_{\max} \geq \lambda_{\min} \geq 0$ be the eigenvalues of $X^T X$. Suppose you have batch size 1, so our update looks like

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \cdot 2(\mathbf{x}_i \mathbf{x}_i^\top)\mathbf{w}_t$$

where $i \sim U\{1, \ldots, n\}$.

6 ConvNets, Normalization (09/13)

Suppose you are given an image and want to perform some classification task (is it a cat, dog, airplane, frog)? If you have $100 \times 100 \times 3$ images, then we want to pick an architecture we think will be good at expressing specific patterns; building an MLP with layer size 30000 is massive and will learn lots of noise. The key ideas in expressivity are as follows:

- Respect locality: many interesting patterns we want to learn are local, so we should use a convolutional structure with filters.
- Respects symmetries: can use weight sharing and data augmentation.
- Support hierarchical structure: understanding depth, multi-resolution perspective.

And the key ideas in stabilization in implementation:

- Normalizations
- Residual/skip connections
- Dropout

Next, we motivate several important aspects of ConvNets:

- **Weight sharing**: we apply the same filter at different patches in the image. We think of this as weight sharing instead of just one filter because of the following framework: you have a filter that you slide around; different patches contribute to the learned filter additively.

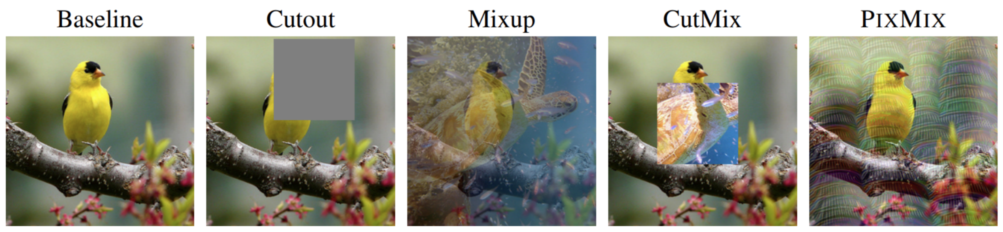

Figure 2: An overview of different augmentation approaches [2]. Cutout emphasizes using all of the image as clues, not just the few that you need. Mixup provides an image of 90% bird and 10% turtle and expects the same proportions as its output (don't throw away unlikely outcomes). CutMix is a combination. PixMix applies some terrible filters, reminiscent of psychedelics.

- **Channels**: if you have one filter, your output will have depth 1. Maybe you want to learn multiple filters, in which case your depth will just be that number. Note that the intermediate layers aren't images, but they have the same topology as images.

  Question. At a given layer, how many filters do we want to use?
- **Nonlinearities**: For a long time, it was common to apply a nonlinearity after every convolution. Now, it isn't so common to do so anymore.
- **Padding**: You might want the image to stay the same size (same padding). There are a couple ways of doing this (padding with zero, mirror, etc).
- **Pooling**: Suppose you have some $100 \times 100 \times 3$ image, and you want to convert it to $1 \times 1 \times 4$ scores for cat, dog, airplane, frog. It is natural to want to downsample it over several layers. **Stride** is one way, throwing away some of your convolutions by not computing them at all. More generally, we can also do things like average and max pooling.
  - With average pooling, everything has equal contribution to the gradient. With max pooling, only one pixel contributing, i.e. the gradient is "routed" to the max.
  - Each pixel has a "receptive field," i.e. the pixels that contributed to it in the previous layer. (For example, the receptive field of any output pixel in a fully-connected layer is the entire input.) As you go back through the layers, the receptive field expands. With pooling, this expansion happens more quickly.
- Eventually, you want to get to 1×1. So towards the end, people add some fully-connected layer(s).
- This works fine for translation invariance. But it doesn't work for small rotations. In practice, you apply data augmentations (contrast, posterize, shear, rotate, translate), so that the network never sees the same image twice.

  Even better, you can give robustness with augmentation. See Figure 2. The key takeaway is that a lot of creativity and domain-specific knowledge is required to regularize effectively.

7 Normalization Layers, ResNets (09/18)

Normalization

Recall that we like to normalize each feature in our training data because neural nets like to move in the direction of big things (specifically, large singular values). So we want to make our data zero mean and variance one so that we can avoid "accidental bigness."

Furthermore, we like to initialize weights with Xavier/He so that we can avoid the same problem.

However, these don't provide guarantees during training; left to their own devices, neural nets often produce exploding or vanishing gradients. The most common intervention is normalization layers, which normalize the outputs of each layer.

This raises two question:

1. What should we normalize to?

   Answer 1: Use $\mu = 0, \sigma = 1$.
   Answer 2: Make μ and σ learnable.

2. What do you average together? (In the case of ConvNets, you can visualize the output as $HW \times C \times N$, where (H, W) are the image size, C is the number of channels, and N is the number of mini-batches, and we are interested in determining which blocks should participate in normalization.)

   Answer 1: Average over all batches for a particular pixel and channel.

   Answer 2: Answer 1 doesn't work well because your batch estimate might be bad (e.g. if your batch size is 16, there might be too much variance). However, we still want to average over things we think might be similar.

   So we introduce **batch normalization**: we average over pixels and batches for each channel. Observe that this is SGD-friendly and backprop-friendly, but backprop is nontrivial because now the gradients depend on the rest of the pixels. (We'll see this in homework, probably.) It's a bit tricky because you have a different mean and variance for each batch, so in implementation you might remember the normalization constants for each batch and take the average at the end. As a result, passing in the same input at train and test time will result in different losses, but we hope that they aren't too far off.

3. *Answer 3:* This last part is a bit troublesome. Moreover, different channels might represent different components of the same "signal" (i.e. gradient direction), and normalizing channels separately might distort those signals.

   Layer normalization averages over pixels and channels for each batch.

ResNets

In practice, however, initialization, normalization, and ConvNet structures were insufficient in training very deep neural nets. These work well at preventing gradients from becoming too big, but aren't great at preventing them from getting too small. One interpretation to neural nets getting stuck is that reaching a local minimum often makes the last layer unstable, and that uncertainty gets propagated back into the first layer as garbage. So we want a way of allowing upstream gradients to be propagated directly downstream, such that they don't have to be passed through

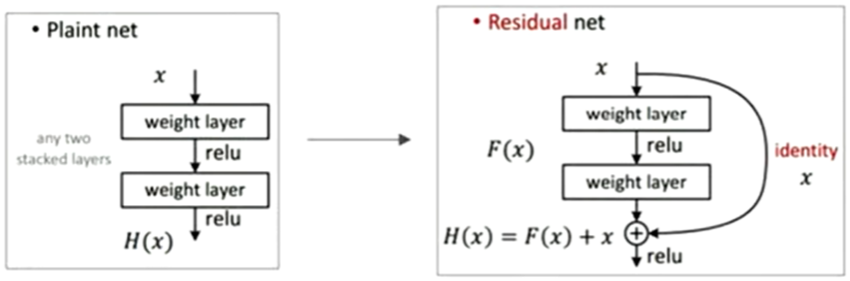

Figure 3: ResNet architecture

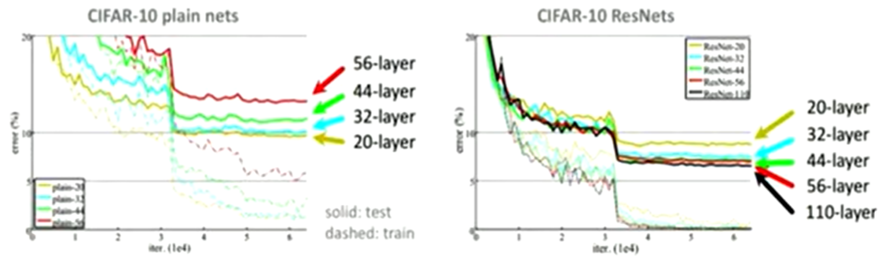

Figure 4: ResNets achieve better performance for deeper neural nets.

so many activations.

Question 7.1. How do we decide which **skip (aka residual) connections** we want? For instance, adding $\binom{n}{2}$ connections will make gradients add exponentially, and they'll explode.

Answer. Add a connection between the input to each layer and the input to the next one. This way, it's possible to for the gradient to pass from the end to the start without passing through all the intermediate layers. See Figure 3.

Functionally, the optimal performance of a ResNet isn't any better than that of your old neural net: adding the input can be achieved by carefully setting weights. It succeeds, however, because setting those weights takes a lot of time and tuning, and it can be much better to use this architecture.

Question 7.2. How do you downsample?

Answer. Apply a projection.

8 ResNets, Fully Convolutional Nets (09/20)

This lecture contained a bunch of separate ideas, so the notes are also jumbled.

Remark. Supposedly, there is a way to consider a ResNet as an ODE. It seems weird to add the input of one layer to the output of the next because they don't seem "compatible" at first glance. The idea is that when you run gradients through, the units "make sense" because of the update $\mathbf{x}(t+\Delta) = \mathbf{x}(t) + \Delta f(\mathbf{x}(t), t)$. If you write it as an ODE and it has some form, there may be some way to implement backprop without needing to store gradients of all intermediate activations.

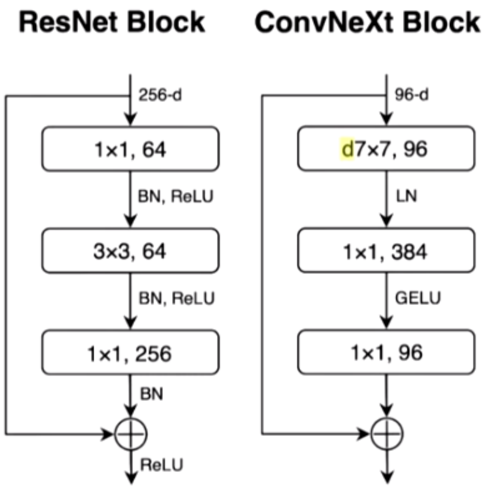

Figure 5: ConvNeXt aggregates some methods from the previous several years: residual connections, cosine learning rate schedule, AdamW, and aggressive data augmentation

Note. GeLU is given by $x\Phi(x)$, where Φ denotes the Gaussian cdf. It is inspired by a theoretical variant where we randomly draw from a Gaussian and only pass x through if it is nonnegative. Its gradient is non-convex and non-monotonic.

Classification In the pre-ResNet days, people downsampled images until they hit some size, then flattened them, then ran them through a fully-connected layer and finally a linear layer. Later, people introduced global average pooling, which reduces everything to a 1×1.

Segmentation Suppose you want to associate each pixel in an image with a class. One idea is to use a convnet where each layer has the same size; this has the issue that some classes exist at different scales. One fix is to apply convolutional layers until you hit low-res (never 1×1, but pretty small) and high channels, then up-sample at the end.

Question 8.1. How do you upsample?

Answer. In the past, we used stride as a means to downsample. We can use the inverse process (we can regard it as fractional stride) as a means of upsampling.

Max un-pooling requires that we remember which elements were the max and use those same positions.

In particular, we can pair downsampling and upsampling layers, where we concatenate activations from conv layers to their corresponding upsampling layers. This is called a **U-net**; see Figure 6. Generally, they are good for anything producing images, and as such are good for segmentation and generation in particular.

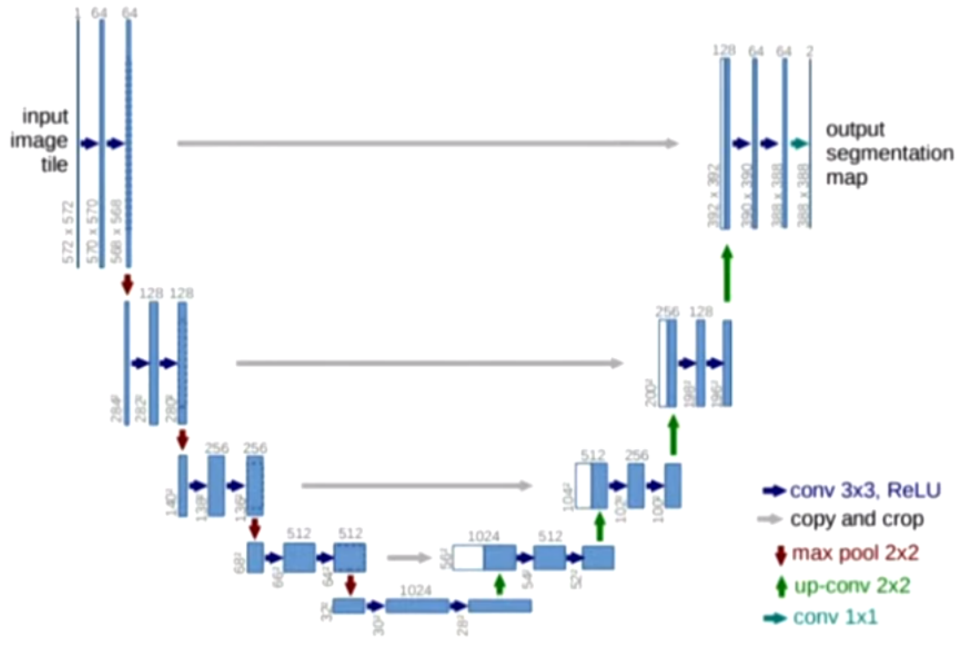

Figure 6: U-net

9 Dropout, SGD (09/25)

The corresponding video is here.

Dropout

Previously, we saw the analogy

input : deeper layers :: input standardization : batch and layer normalization.

In this analogy, what does data augmentation correspond to? Its purpose (e.g. with rotations, reflections) is that the output will not become overly dependent on any particular activation. **Dropout**, i.e. randomly zeroing some activations, has this effect: we care about the image as a whole, so getting rid of some things shouldn't cause any problems.

Suppose that every activation has iid probability p of being zeroed. In expectation, its value is multiplied by $1 - p$. So we account for this at test time: multiply its output by $1 - p$.

SGD

Suppose you want to solve solve $W\mathbf{x} = \mathbf{0}$ (W is the data with rows $\mathbf{w}_i^\top$, and $\mathbf{x}$ are the parameters). The SGD update is $\mathbf{x}_{t+1} = \mathbf{x}_t - 2\eta \mathbf{w}_I(\mathbf{w}_I^\top \mathbf{x}_t)$, where I is sampled uniformly at random from $[n]$.
Consider $\mathcal{L}(\mathbf{x}) = \frac{1}{n}\mathbf{x}^\top W^\top W\mathbf{x} = \mathbb{E}_I[\mathcal{L}_I(\mathbf{x})]$, where $\mathcal{L}_I(\mathbf{x}) = (\mathbf{w}_I^\top \mathbf{x})^2 = \mathbf{x}^\top(\mathbf{w}_I \mathbf{w}_I^\top)\mathbf{x}$. Then:

Claim 9.1. $\mathbb{E}[\mathcal{L}(\mathbf{x}_{t+1}) \mid \mathbf{x}_t] \leq (1-\rho)\mathcal{L}(\mathbf{x}_t)$ for some constant $\rho > 0$.

Proof.

$$\begin{aligned}\mathcal{L}(\mathbf{x}_{t+1}) &= \mathcal{L}(\mathbf{x}_t + (\mathbf{x}_{t+1} - \mathbf{x}_t)) \\ &= \mathcal{L}(x_t) + \frac{2}{n}\mathbf{x}_t^\top W^\top W(\mathbf{x}_{t+1} - \mathbf{x}_t) + \frac{1}{n}(\mathbf{x}_{t+1} - \mathbf{x}_t)^\top W^\top W(\mathbf{x}_{t+1} - \mathbf{x}_t).\end{aligned}$$

Let's bound each term separately:

$$\begin{aligned}\mathbb{E}\left[\frac{2}{n}\mathbf{x}_t^\top W^\top W(\mathbf{x}_{t+1} - \mathbf{x}_t) \,\middle|\, \mathbf{x}_t\right] &= -\frac{2}{n}\mathbf{x}_t^\top W^\top W 2\eta \mathbb{E}_I[\mathbf{w}_I \mathbf{w}_I^\top]\mathbf{x}_t \\ &= -\frac{4}{n^2}\eta \mathbf{x}_t^\top W^\top W W^\top W \mathbf{x}_t \\ &\leq -\frac{4}{n^2}\eta \mathbf{x}_t^\top W^\top \sigma_{\min}(WW^\top) W \mathbf{x}_t \\ &\leq -\frac{4}{n}\eta \lambda_{\min}(WW^\top)\mathcal{L}(\mathbf{x}_t), \qquad (\clubsuit)\end{aligned}$$

and

$$\begin{aligned}\mathbb{E}\left[\frac{1}{n}(\mathbf{x}_{t+1} - \mathbf{x}_t)^\top W^\top W(\mathbf{x}_{t+1} - \mathbf{x}_t) \,\middle|\, \mathbf{x}_t\right] &= \mathbb{E}_I\left[\frac{1}{n}4\eta^2 \mathbf{x}_t^\top \mathbf{w}_I \mathbf{w}_I^\top W^\top W \mathbf{w}_I \mathbf{w}_I^\top \mathbf{x}_t \,\middle|\, \mathbf{x}_t\right] \\ &\leq \frac{1}{n}4\eta^2 \lambda_{\max}(W^\top W)\mathbb{E}_I\left[\mathbf{x}_t^\top \mathbf{w}_I \mathbf{w}_I^\top \mathbf{w}_I \mathbf{w}_I^\top \mathbf{x}_t \,\middle|\, \mathbf{x}_t\right].\end{aligned}$$

Letting $\beta^2 = \max_i \|\mathbf{w}_i\|^2$, we have

$$\begin{aligned}\cdots &\leq \frac{1}{n}4\eta^2\beta^2\lambda_{\max}(W^\top W)\mathbb{E}_I\left[\mathbf{x}_t^\top \mathbf{w}_I \mathbf{w}_I^\top \mathbf{x}_t \,\middle|\, \mathbf{x}_t\right] \\ &= 4\eta^2\beta^2\lambda_{\max}(W^\top W)\mathcal{L}(\mathbf{x}_t) \qquad (\spadesuit)\end{aligned}$$

Since β, $\lambda_{\min}$, and $\lambda_{\max}$ are constants we can compute ahead of time, choosing sufficiently small η will make $|\clubsuit| > |\spadesuit|$, which suffices. ■

10 Graph Neural Networks (09/27)

The corresponding video is here.

Question 10.1. Can we generalize principles from CNNs for images to general graphs?

The idea behind CNNs is that they exploit local structures. We want to do the same thing using undirected graphs with no edge labels.

In images, we had image-level tasks (e.g. classification) and pixel-level tasks (e.g. segmentation). Likewise, we'll have graph-level and node-level tasks.

First, let's give an overview of the big CNN ideas:

- Convolutions (local processing):

  - Weight-sharing (translation invariance) and filter-banks.
  - Depth of network as "seeing" the entire image: growth of the receptive field.
- Residual connections
- Normalization layers
- Downsampling
- Data augmentation (problem-specific)

Next, let's see their analogs:

- **Convolutions** work in images because there is an ordering: each entry in the filter is paired with a specific pixel as we move the filter across the image. This is not the case for graphs: each vertex is only associated with an (unordered) set of vertices, which can have different size for different vertices. As a result, most of the time, we'll learn three functions:
  - f, which accepts a vertex and an aggregate of its neighbors as input
  - s, representing a similarity score between a vertex and a particular neighbor (a scalar)
  - g, a function of a particular neighbor (possibly a vector).

  Overall, it'll look something like:

$$f_{W_1}\left(v, \sum_{u \text{ neighboring } v} s_{W_2}(v,u) g_{W_3}(u)\right),$$

  where $\sum$ is a placeholder aggregator (might be max, average for example).
- **Residual connections** are built in because the graph is fixed.
- **Normalization:** Recall our 3D box visualization. Batch norm considered a fixed channel and normalized over pixels and images. Layer norm considered a fixed image and normalized over pixels and channels. The naive approach is just to replace "pixels" with "nodes." This works in the case of small graphs. Otherwise, you don't have enough memory. In the case of large graphs, you can deal with it in a problem-specific way.
- **Pooling:** One approach is to compute a clustering of the graph and downsampling to clusters. Another is to do this in a content-specific way, for example learning a similarity measure to simulate clustering.
- **Data augmentation** is problem-dependent (built in).

Remark. We only look at local structures at any point, so the graph structure need not remain fixed.

11 GNN (10/02)

The corresponding video is here.

He started off with a review of last lecture, and I didn't really care to follow.

Question 11.1. Suppose we want to run some node-level task. What does it mean to have a train/val/test split?

Answer. Simply deleting the held-out nodes and all incident edges doesn't work. It is easy to implement, but we can't see the "not-missing" parts of the nodes to learn patterns. It can also disconnect the graph.

In practice, we want to maintain the graph structure. So we'll mask a subset of the nodes during training. This approach works especially well when the number of labeled nodes is small relative to the number of unlabeled nodes (ignore noise).

Question 11.2. How do you speed up the spread of globally-relevant information?

Answer. Augment the graph with "global" nodes, which is connected to everything.

Question 11.3. How do you train when the graph is massive? How do you sample batches?

Answer. Some ideas are sampling nodes and taking all of its neighbors, or taking random walks in a graph. In the former case, the receptive field scales exponentially, and in both cases, the central node contributes much more to the loss than its surroundings. This can work well, but as a result, we should choose our loss function carefully.

Question 11.4. How do you deal with padding?

Answer. You can get away with doing nothing, or try to establish boundary nodes.

12 GNN (10/04)

The corresponding video is here.

Again he did a lot of talking and not a lot of writing, so not much of it got copied here. For a proper transcription see s23 notes.

Question 12.1. How do you train on huge graphs?

Answer. It's hard to sample a minibatch. If you just select some connected subset of nodes and the corresponding boundary, you run into several problems:

- No guarantee on the relative size of the selected graph and the boundary (whereas for images the boundary is small).
- Nodes near the boundary of a minibatch get less information, because they are within distance of only a few other nodes. Some techniques to mitigate this are adaptive sampling, importance sampling, and graph coarsening.

He then did a general review, which I didn't bother watching.

14 RNN, Self-supervision, Autoencoders (10/18)

The corresponding video is here.

RNNs

Recall the Kalman filter setup: nature has some latent transition variable $x_t = Ax_{t-1} + Bu_{t-1} + w_t$, and we observe $y_t = Cx_t + v_t$, where u_t are known control inputs and w_t and v_t are zero-mean, uncorrelated noise. Our goal is to use past controls $\{u_0, \ldots, u_{t-1}\}$ and past and current observations $\{y_0, \dots, y_t\}$ to estimate x_t. KF gives us some recursive form

$$\hat{x}_t = \tilde{A}\hat{x}_{t-1} + \tilde{B}_u u_{t-1} + \tilde{B}_y y_t$$

for some deterministic $\tilde{A}$, $\tilde{B}_u$, $\tilde{B}_y$ (dependent on $A, B, C, \Sigma(w), \Sigma(v)$) true in expectation. [There is some assumption on Gaussians, but I forget exactly and this isn't too important.]

Suppose that we want to approach this problem with learning, where we don't know any of the required values for KF (namely $A, B, C, \Sigma(w), \Sigma(v)$). Now, let us consider the two possibilities:

- In the supervised case, suppose we have traces $\{(x_t, y_t, u_t)\}_{t=0}^n$. We can make $\tilde{A}$, $\tilde{B}_u$, $\tilde{B}_y$ learnable parameters with loss $\ell(x_t, \hat{x}_t) = \|x_t - \hat{x}_t\|^2$, and proceed as usual.
- In the unsupervised case, suppose we are solving the original problem, i.e. we are given $\{(y_t, u_t)\}_{t=0}^n$. We have to be careful to not pass in these values as-is, since y will be passed in as an input but is also used as a label (we don't just want to copy it over). The solution is to set $\hat{y}_{t+1} = C(A\hat{x}_t + Bu_t) =: \tilde{C}_x\hat{x}_t + \hat{C}_u u_t$ (since we don't know A, B, C), and make $\tilde{C}_x$, $\tilde{C}_u$ learnable. Our loss will be $\ell(y_{t+1}, \hat{y}_{t+1}) = \|y_{t+1} - \hat{y}_{t+1}\|^2$.

He went on to say something about introducing a hidden state that can be obtained from an invertible transformation $\tilde{T}$ on $\hat{x}_t$, and how making another head with a separate loss $\ell(x_t, \hat{x}_t)$ might result in some conflicts because the losses pull in different directions. It didn't seem super important to I skipped most of it, but it's probably a segway into the following thing.

In practice, there are some complications with RNNs:

- Where do you introduce nonlinearities in RNNs?

  The whole purpose in an RNN is that applying $\tilde{A}$ many times will make eigenvalues explode or vanish. We apply $h_t = \sigma(Ah_t + \cdots)$ to combat this, i.e. the purpose of a nonlinearity is to force eigenvalues to 1.

  In RNNs, we often use saturating nonlinearities, e.g. tanh and sigmoid. This keeps values reasonable but kills gradients.
- How do you use residual connections?

  Vertical skips (i.e. between different RNN layers) work well, and is basically the same that is used in a ConvNet. Horizontal ones (i.e. temporal) don't, and aren't really used in practice.
- Normalization?

  You can add normalization blocks horizontally.

Self-supervision

Above, we used "unsupervised" (the old term) to mean the absence of an x. Really, this was self-supervision, since we just used the data to supervise itself.

In CS 189, we saw PCA and k-means, which felt very different in nature than everything else we've been doing. In fact, it is possible to understand these methods as though they are a self-supervised regime, as we will see in homework.

15 Attention and Transformers (10/25)

The corresponding video is here.

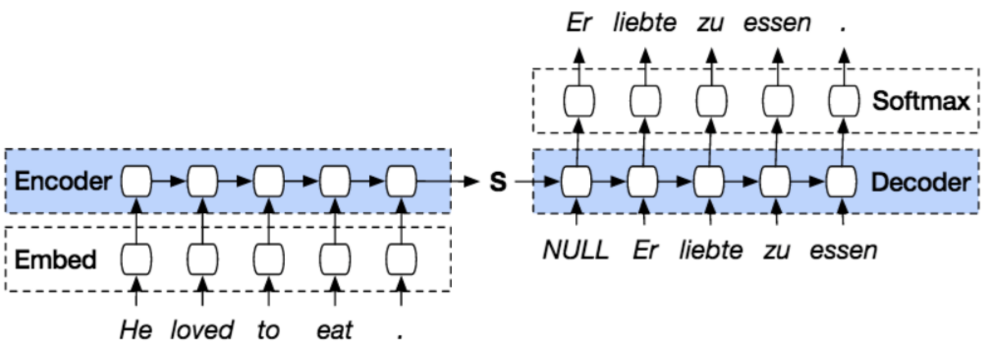

Figure 7: Sequence-to-sequence with RNN

One can interpret a transformer as a graph neural net with a fully connected graph. This is non-trivial because we retain the idea permutation invariance. At a high level, we will use a similarity-based aggregation of messages $\sum_v \mathrm{sim}(u, v) f(v)$. So we have an MLP with an additional constraint, that one might not have thought of without a GNN.

The more historically accurate explanation for the transformer is the following: "How do you produce an analog for U-nets in sequence-to-sequence models?"

Encoder-Decoder Approach

Suppose we want to translation from English to German. With an RNN, our model will look something like Figure 7. The idea is that there will be an encoder RNN, which produces a final hidden state after the last token. Then you will have a decoder RNN, which runs the translation. There are many reasonable ways to do translation, so our output will be randomized with probabilities given by a softmax. As a result of this randomization, we need to pass each output token as an input to the next hidden state.

This approach doesn't work great. Sure, the order of words matters. But there is more structure to a sentence than its order — there is some hierarchical structure (e.g. an adjective modifies a noun, a noun is performing the action indicated by a verb), similarly to images, that isn't captured by an RNN.

Our goal, then, is to have the decoder be able to look up relevant local context for each corresponding hidden state in the encoder. (This is exactly like a U-net!) We want this memory to be semantically meaningful — we don't care to be able to index into the third element in our memory array, but rather want to be able to query it by content.

Attention

We want a differentiable, softened, approximate hash table. Suppose our memory consists of some key-value pairs, and we want to run some queries. (The keys come from the encoder, and the queries come from the decoder.) We'll run some ideas and see why they do or don't work:

- Take the k closest to q, and return the corresponding v.
  - This produces no useful gradients on k or q, since changing either of them by a small amount will not affect the answer. It only produces useful gradients on the winning v,

and doesn't do anything to the rest of them.

- Use the GNN approach, $\sum_i \text{sim}(k_i, q)v_i$.
  - Now we have good gradients on everything.
  - We must decide what to use for our similarity. We can take $e_i = k_i^\top q$ and $\alpha_i := \text{sim}(k_i, q) = \frac{\exp e_i}{\sum_j \exp e_j}$.
    * Consider the e_i's. Each k_i is initialized to some random direction, and each q is too. So their inner product is the sum of several iid things, and by CLT scales like a Gaussian with variance $O(d)$, where $k_i, q \in \mathbb{R}^d$.
    * The e_i's will be scaled on the order of $O(\sqrt{d})$, and through the softmax, the problem will worsen considerably (it's basically a hard max). Even though initialization is just noise, most gradients will die immediately.
    * Instead, take $e_i = k_i^\top q / \sqrt{d}$.

This latter approach is known as **cross-attention**, because our keys and queries come from different neural nets. It may also be the case that we want to query from the original neural net, called **self-attention**: maybe it is insufficient to keep passing the outputs through the decoder RNN, and it'll be better to query for them too. This is useful if your output is "causal."

In practice, it is also good to look up ℓ things from ℓ tables, called **multi-headed attention**. Intuitively, they function similarly to channels in CNNs.

The insight in [3] is that you can run self-attention and cross-attention everywhere, and you no longer need the RNN.

16 Transformers (10/30)

The corresponding video is here.

Last lecture, we saw a graph neural net interpretation of a transformer. A key idea is that the softmax is symmetric; we can't distinguish the neighbors. However, this doesn't make sense in natural language: although we maintained before that literally indexing sequences doesn't work well, we have to have some notion of how we'll order things (else "dog bites man" and "man bites dog" are equivalent).

Each decoder block will consist of three parts:

- An embedder, which accepts (discrete) tokens and outputs a (continuous) vector. A simple model for this is a lookup table; they are learnable parameters.

  Notes:
  - We can't take gradients over discrete things. Output tokens from each decoder block are inputs to the following decoder block; gradients don't flow across them.
  - Your training data might not include some words, so your embeddings won't support them (random initialization!). So in practice, you'll use a coarser embedding that allows extra words not in your dictionary.
- Decoder blocks, which consist of cross-attention, self-attention, MLP, and normalization and are connected via residual connections. There are lots of possible ways to arrange them. How do we decide what works best?

We choose the one that performs better in compute time: it turns out that it works best to put all of these blocks in parallel. In practice, we put our normalization before the other three. The idea is that we want to make our vectors roughly the same size before they're passed through a softmax.

In each attention block, we have some keys, values, and queries, which are constructed with matrices W^k, W^v, and W^q.

At training time, we apply **causal masking**: we don't want the future to impact our current predictions, so equivalently we can set scores in the softmax to $-\infty$.

Positional encodings are done via **RoPE**. Intuitively, the idea is that you can encode time/-position with something cyclic; the trick is to think in $\mathbb{C}^{d/2}$ instead of $\mathbb{R}^d$. RoPE says that of $q = W^q x$, we use $q = \Theta_m W^q x$, where $\Theta_m = \text{diag}\left\{e^{im\theta_k}\right\}_k$, where $\theta_k = 10000^{-2(k-1)/d}$. For example, $\theta_1 = 1$, while $\theta_{d/2+1} = 10000^{-1}$.

- An "inverse-embedder," which accepts a vector and outputs probabilities for each token. This uses weight sharing with the embedding.

  We have to be careful with scaling (aka temperature); recall that we had to do $d^{-1/2}$ scaling in the past, and similarly should do this before passing dot products through a softmax.

17 Transformers (again) (11/01)

The corresponding video is here.

Transformers

Last time, we used RoPE as a modification to attention as positional encodings. Note that there are no additional learnable parameters.

Another method is **relative position embedding**, where you replace $\langle q_j, k_i \rangle$ with $\langle q_j, k_i \rangle + b_{j-i}$ for some learnable biases b per attention head.

Fine-tuning

Start with a trained network. Suppose we now have some data for a new task, and we want to adjust our network to do well. Here are some approaches:

- **Full fine-tuning:** Treat the trained network as an initialization; just train on your new data.
- **Linear probing:** Typically, your network will have a linear head that produces scores for the classes you care about. So one idea is to:
  - "Decapitate" the old network (i.e. remove its head), which you expect to have learned good features for your task
  - Add your head with random initialization
  - Freeze the old network, and train just your linear model
- We can also do a combination of these two ideas, e.g. if you have the resources to train the full network, you can run linear probing then full fine-tuning.

Question 17.1. Our task might not have much data, but you can count on a pretrained model. What things are ok to forget for our task, and what things aren't?

- There is a large continuum of ways to freeze the network. For instance, you might want to freeze the middle layers but make the top and bottom learnable.
- Don't decapitate, and combine the old and new training data. Instead, just add a new linear head, and learn two tasks.
- Any other regularizing effects: adjusting learning rates, explicit regularizers, early stopping, etc.

Self-supervision and pre-training

Suppose we have a large unlabeled dataset. We want to be able to use it but don't have any specific task in mind. So we can instead make a "universal" surrogate task for pre-training. Here are some common tasks:

- Autoencoders: given X, reconstruct X.
- Denoising autoencoder: Given $X + N$, reconstruct X.
- Masked autoencoder: Given $N \odot \{\text{binary noise}\}$, reconstruct X.
- Next-token prediction: Given $X_1, \ldots, X_T$, predict X_{T+1}.

18 Self-supervised training, Prompting, ViT (11/06)

The corresponding video is here.

Self-supervised training

To learn a pattern effectively with a large model, either you must have a strong inductive bias and hope that it works well, or have a lot of data.

Example 18.1. **GPT** is a decoder-only transformer, i.e. we just use causal self-attention. We will pass in tokens in order, beginning with [START]. Then, we will have some loss on each output token, maybe cross-entropy or the like, and run backprop, where each head attends to all preceding tokens. Some implementation details:

- We use AdamW as an optimizer, as we want gradients to remain stable in such a large model.
- Generally, it is better to use longer context lengths (whatever will fit in memory; rather than several short ones) due to the overall purpose of a transformer model.
- One epoch training: data is only seen once. (In homework, we'll see empirically that this tends to work best for a fixed compute budget.)

Example 18.2. **BERT** is an encoder-only transformer, i.e. just bidirectional self-attention, designed solely for the pretraining-finetuning paradigm.

Due to the encoder structure, it is insufficient to accept input [START], $T_1, \ldots, T_N$ and produce outputs $T_1, \ldots, T_N$ (the model will just copy over those tokens). We resolve this by running a combination of masked and noisy auto-encoding (that is, we replace some inputs with [MASK] and some with other words).

BERT produces activations at each self-attention layer, so for pretraining-finetuning purposes you have lots of choices for which one(s) to use.

Example 18.3. Word2Vec was an unsupervised model some time ago that mapped similar words to similar embeddings. An interesting phenomenon was that the resulting embeddings E displayed properties from the data that it wasn't explicitly designed to do, e.g. $E(\text{king}) - E(\text{man}) + E(\text{woman}) \approx E(\text{queen})$.

Likewise, the model picked up on stereotypes present in the data. This led to alignment research, which aims to suppress artifacts present in the data that we don't want to learn.

Prompting

- GPT-1 was designed as a pre-training + fine-tuning regime, and could be used for specific tasks.
- GPT-2 had the realization that prompting for next-word prediction works, e.g. "the capital of France is ____."
- GPT-3 introduced in-context learning. Prompts looked like this: "let's play the capital game! The capital of ____ is ____. [Some more examples.] The capital of France is ____."

ViT

Vision transformers work the same way as in language. We break images up into small patches and run attention. Similar to our above discussion on text, ViT does not have any inductive biases, but through pretraining on a large dataset alone learned the importance of locality in images.

Remark. In language, each token is quite valuable. In vision, patches are redundant, and we can get away with dropping [MASK] tokens entirely to save compute.

19 Language models (11/08)

The corresponding video is here.

BERT

A slight continuation of last lecture.

BERT has an input sequence that is randomly masked and noised. It has two objectives:

- Reconstruct the input.
- Augment it with another input sequence, separated by a learnable `<SEP>` token, that appears in the same long passage. The order of the input sequences is randomized. Then we have a binary classification problem where we predict which sequence came first.

Typically, we want embeddings to be smaller than the original thing; for example, we looked at autoencoders and PCA in the past. In a transformer network, however, it is sometimes good to consider an expanded set of features (in classical ML, for instance, we might've looked into polynomials applied to our features). The size of each layer is exactly the same (recall that we have residual connections), so looking at the activations from these layers gives us an expanded "set of features." Some research has shown that averaging the last four layers does pretty well.

Tokenization

A tokenizer maps variable-length strings into fixed-length vectors. The solution is called **byte-pair encoding** (and is still roughly used today in LLMs), and is an algorithmic problem. It works similarly to Huffman codes, where we map the most common substrings in a provided corpus to some tokens. There is no ML here!

Beam search

In our usual setup, the encoder consists of several blocks with a token as input and a token as output. Those output tokens are fed in as inputs to the following block. The output tokens are generated by sampling from our distribution from the softmax outputs.

Suppose our model is fairly small, and the outputs aren't great. Also suppose that we have some fixed compute budget. One theoretical idea from tree traversal is to maintain some fixed number of several paths, each generated by sampling. For each path, we will maintain its log probability by summing the log prob on each token, and sometimes we will throw paths out or expand our tree by branching.

This approach tends to work well with smaller models; for LLMs it is usually better to just produce one sample (acting greedily).

Parameter-efficient finetuning

Previously, we discussed two methods of finetuning: linear probing and full finetuning. There is a very big gap between these two approaches, and we want to bridge that gap. After all, we might not have enough memory to even store the large model (all the activations and the optimizer state would triple your required number of parameters), and training it can be quite hard.

20 Parameter-efficient finetuning and Meta-learning (11/13)

The corresponding video is here.

Parameter-efficient finetuning

To adjust our finetuning, we can either change the initial condition or the dynamics.

Soft-prompting

At a high level, the idea is that a language model with prompting solves a differential equation with a very strong initial condition, so engineering the prompt can lead to a variety of results.

Recall that your prompt is treated as a collection of vectors that are outputs of an embedder; as a result, it is possible to optimize the prompt. (Just run gradient descent, and your vectors will change.)

The idea of soft-prompting is that the prompt felt by the model is the vectors and not the text.

A key discovery in LLMs is that the entire model might have lots of parameters, say in the billions. You might have a prompt with length in the thousands or tens of thousands. Even so, optimizing on the vectors from the prompt embedding provides a very substantial improvement over hand-designed prompts. In fact, this usually performs as well or better than finetuning.

Suppose we have a larger budget for parameters. At each location in your model, a layer sees the prompt through key-value pairs corresponding to preceding layers and tokens. It does not experience the effect of MLPs or queries directly, but rather indirectly through those activations. In the same way, as we broke the link between the vectors and the prompt with soft-prompting, you can break the consistency relationship between the layers: perturb the keys and values, and stop the gradients there. That is, your gradients flow left but not down. Then you have a range of soft-prompting strategies based on how many key, value layers you tweak. (In general, tweaking just the final layer is usually quite good.)

Low-rank adaptation

A transformer model has some W_k, W_v, W_q, and W_{MLP} at each layer. Typically, the number of weights in W_{MLP} is the largest, but the rest can be quite substantial as well.

The idea is to replace W_{MLP} with two components, namely the sum of W_{freeze} and ΔW (which allows gradient flow), with $\Delta W = BA$ where we can fix the number of parameters. For example, if $W_{\text{MLP}} \in \mathbb{R}^{500\times 1000}$ (requiring 500,000 parameters), then taking $B \in \mathbb{R}^{500\times k}$ and $A \in \mathbb{R}^{k\times 1000}$ requires $1500k$ parameters. Taking $k = 20$, for example, only needs 30,000 parametersa but does pretty well.

We also need to initialize B and A. We want ΔW to start at around 0, but setting $B = A = 0$ means that the input to B is 0, and all gradients on B are 0. This isn't a problem as long as exactly one of A and B is nonzero, and we can set the other one randomly (it turns out that this doesn't work if you have both A and B nonzero for some reason).

One could alternatively have set ΔW with a random sparsity pattern, i.e. it has the same dimensions but only 30000 nonzero entries. It turns out that this doesn't work as well. Why?

Suppose we take $B = 0$ and A random. We are effectively taking a 1000-dimensional space and projecting onto k random directions. By running gradient descent, we adjust our basis to better fit the data. In the sparse case, on the other hand, gradient descent can't help you change the sparsity pattern.

At a high level, the idea behind LoRA is similar to that of PCA: we want the variations in the directions that matter be significant, and we don't really care about everything else. So we don't lose very much through a low-rank approximation.

Question 20.1. We know that large models do better. Yet in practice, reducing their size to speed up training still does very well. Contradiction?

Answer. This is an open question. It may be possible that having a large model is helpful to reaching a good initial condition, at which point you don't need too many parameters to finetune.

Meta-learning

Question 20.2. Can I learn a model that is good for finetuning?

Answer.

- So far, we have used the following approach: pick a self-supervised task and collect all the data you can.
- Use the first approach, then finetune on a good task.

To do better, we must assume that we have a family of tasks with labeled training data that should do well on unseen new tasks.

This is given by **Model-Agnostic Meta-Learning (MAML)** [1].

21 Meta-learning, Generative Models (11/15)

The corresponding video is here.

Meta-learning

Assume you have some base model parameterized by θ_0, and you have a meta training set of tasks with labeled training data.

The idea of MAML is that you will have several tasks. For each task, you'll have some loss, which you optimize using your training set to obtain a new set of parameters $\theta_{\text{finetuned}}$. Then you will evaluate $\theta_{\text{finetuned}}$ on a holdout set. Then the objective is to find θ_0, i.e. find a model such that the entire thing can be easily fine-tuned.

Algorithm 1 MAML

```
1: while not converged do
2:     pick a training task and its corresponding data
3:     $\theta \leftarrow \theta_0$
4:     for $k$ steps do
5:         $\theta \leftarrow \theta - \eta_1 \nabla_\theta \ell(y, f_\theta(x))$
6:     end for
7:     sample $n$ holdout points $(x_i, y_i)$ uniformly at random
8:     $g \leftarrow \frac{1}{n} \sum \ell(y_i, f_\theta(x_i))$
9:     $\theta_0 \leftarrow \theta_0 - \eta_2 \nabla_{\theta_0} g$
10: end while
```

In previous lectures, we saw an approach to the same problem where we have a shared network and several task-specific heads, and train the whole network. Then we attach a new head and finetune just the head. Clearly this is much cheaper than MAML (and in some cases performs quite well). One trick to speed it up even further is to set head weight matrices according to ridge regression rather than running SGD on so many separate points.

Generative models

The approach we saw to language problems is called auto-regressive generation, i.e. GPT. Effectively, we produce output tokens $x_0, x_1, \ldots$, where we learn $p_\theta(X_t = x_t \mid X_{t-1} = x_{t-1}, \ldots, X_0 = x_0)$.

22 VAE (11/20)

The corresponding video is here.

Suppose we have a learned system that can generate images of cats. When we call the model, we want it to be able to generate a new image of a cat. How do we do this?

Typically, we will feed some random vector into the learned system and output an image of a cat. This raises some questions: (i) what sorts of systems would facilitate this process, and (ii) if we have data, how do we train the system?

Idea 0: Use a classifier. Suppose you have a classifier that accepts images and outputs scores for each category. SO you can initialize $\mathbf{x}_0$ randomly (e.g. Gaussian noise), which will produce some dog classification score. Then you can run gradient ascent on the score.

In reality, the dog score increases, but it will look like static. Likewise, if you start with an image of a sheep, you'll end up with a sheep, but your classifier will be convinced that it is a dog. This is known as the **adversarial example phenomenon**.

We will investigate this phenomenon with the following toy model: you have some $\mathbf{x} \in \mathbb{R}^{d+1}$, with $x[1] \sim \mathcal{N}(0, \sigma^2)$ and $\sigma^2 \gg 1$ and $x[i] \sim \mathcal{N}(0,1)$ for $i \in \{2, \ldots, d+1\}$, with the entries of $\mathbf{x}$ jointly independent. The ground truth label is $\operatorname{sgn}(x[1])$, and the model learns a classifier $\operatorname{sgn}(\alpha^\top \mathbf{x})$.

Now, suppose $\alpha[1] = \beta$ and $\alpha[i] = 1$. Then $\alpha^\top x = \beta x[1] + \sum_{i=2}^{d+1} x[i] = u + v$, where $u = \beta x[i] \sim \mathcal{N}(0, \beta^2\sigma^2)$ and $v = \sum_{i=2}^{d+1} x[i] \sim \mathcal{N}(0, d)$. For α to be a good classifier, we require $\beta\sigma \gg \sqrt{d}$.

The gradient update is $x' = x + \eta\alpha$, or equivalently $\alpha^\top x' = u + v + \eta(\beta^2 + d)$. Suppose we want to find an adversarial example, Then:

- For our gradient steps to be useless, we want the Δx to be small enough that it is negligible. Since $u \in O(\beta\sigma)$ and $\Delta x \in O(\beta^2)$, we can enforce this by taking $\sigma \gg \beta$.
- For our classifier to be bad (the influence of $x[2 : d+1]$ is greater than that of $x[1]$, we want $d \gg \beta\sigma$.

If we can satisfy all three of these conditions, we have a confused classifier. For example, taking $\sigma = d^{1/2}$, $\beta = d^{1/4}$, and $\eta = d^{1/8}$ suffices.

Idea 1: Use an autoencoder. Run the decoder on random noise.

What do we mean by random? What are we sampling from?

In fact, we find that this process alone is insufficient, even when x is Gaussian and $\mathcal{E}, \mathcal{D}$ are linear. Let $x = U\Sigma V^\top \in \mathbb{R}^{d \times n}$. We showed in homework that the autoencoder will learn

$$\mathcal{E} = \begin{bmatrix} u_1^\top \\ \vdots \\ u_k^\top \end{bmatrix} \implies \mathcal{E}(x) \sim \mathcal{N}(0, \operatorname{diag}(\sigma_1^2, \ldots, \sigma_k^2)).$$

So really we want to learn

$$\mathcal{E} = \begin{bmatrix} u_1^\top/\sigma_1 \\ \vdots \\ u_k^\top/\sigma_k \end{bmatrix} \implies \mathcal{E}(x) \sim \mathcal{N}(0, I),$$

but we can't do this a priori because we know nothing about the distribution of x. Moreover, once our encoder and decoder become nonlinear, we cannot simply fit a Gaussian to x.

Idea 2: VAE We want it to satisfy the following conditions:

(i) We want the encoder to be robust to noise. So it will accept an input x and samples outputs from a distribution. (Learn a mean network and a covariance network.)

(ii) We will add a regularizer during training so that the latent distribution is close to what we want to sample from. (Impose a KL regularizer.)

(iii) We want to be able to run SGD despite sampling. (Reparameterization trick.)

22 VAE (11/22)

The corresponding video is here.

Last time, we suggested using an autoencoder, where we feed noise in through the decoder.

- Q: What distribution of latents?
  A: $\mathcal{N}(0, I)$.
- Q: What loss function?
  A: $D_{\mathrm{KL}}(Q\|P) = \mathbb{E}_{z\sim Q}\left[\log \frac{Q(z)}{P(z)}\right]$.
  Remark. Some intuition for KL-divergence: Take n iid samples from P. Then

  $$\mathbb{P}(\text{“samples look like } Q\text{”}) \approx \exp(-nD_{\mathrm{KL}}(Q\|P)).$$

  Example 22.1. If $P = \mathcal{N}(0, I)$ and $Q = \mathcal{N}(\mu, \Sigma)$, then

  $$D_{\mathrm{KL}}(Q\|P) = \frac{1}{2}(\operatorname{tr}\Sigma + \mu^\top\mu - k - \log\det\Sigma).$$

One idea is to have $\mathcal{E}(x)$ produce outputs $\mu(x)$ and $\Sigma(x)$, and inputs to the decoder will be sampled from $\mathcal{N}(\mu(x), \Sigma(x))$. We will have two losses: a sample loss given by $D_{\mathrm{KL}}(\mathcal{N}(\mu(x) \,\|\, \Sigma(x)), \mathcal{N}(0, I))$ (in the case that we want our samples to look like $\mathcal{N}(0, I)$), as well as a reconstruction loss. If we too heavily weight KL, then our reconstructions will be poor. On the other hand, if we too heavily weight reconstructions, then our reconstructions will look fine but at test time sampling from $\mathcal{N}(0, I)$ won't work.

Question 22.2. How do you run backprop through samples?

Answer. Reparameterization trick: instead learn $\Sigma^{1/2}$. Let $\varepsilon \sim \mathcal{N}(0, I)$, and let $z = \mu + \Sigma^{\frac{1}{2}}\varepsilon$. Then you can treat ε as an input (no need to push gradients on it), and you can easily get gradients for μ and $\Sigma^{1/2}$.

Next, we will discuss VQ-VAE. Suppose you have some real number $X = \sum_{i=-\infty}^{\infty} X_i 2^i$ for digits d_i. Now, take $X + U$, where $U \sim U(-\frac{1}{2}, \frac{1}{2})$. Intuitively, this randomly flips the bits in the digits in X after the decimal point. However, some digits survive, and in essence the amount of information that remains is the number of digits preceding the decimal.

This motivates the use of quantization as a noisy process. In a VQ-VAE, latents are quantized via k-means. With intuition following the noisy process, simply pass gradients through.

The benefit of quantizzation is that discrete tokens are handled well by methods in natural language.

24 GAN & Diffusion Models (11/27)

The corresponding video is here.

GAN

A generator accepts randomness and produces an image. To help, we make a discriminator, which is a classifier that accepts a candidate image and outputs a binary "real or fake." The idea is to go back and forth between training the generator and discriminator, since they are mutually helpful.

In practice, GANs are very tricky to train due to "mode collapse": the generator and discriminator have opposing objectives even though we want them to work together. Thus, it is easy for the generator to produce bad samples, i.e. the discriminator only thinks one kind of images is correct. In this case, gradients through the discriminator are small, and neither thing moves. There is a lot of research trying to get GANs to work well.

The benefit of GANs is that they are very fast. They can use a relatively lightweight model,

Diffusion Models

The *manifold hyothesis* says that the set of satisfactory things is quite small, probably within some small low-dimensional space rather than scattered throughout. By chance, you'll never land on this set. The idea of diffusion is that starting with some random noise, we can find a path to end up there.

Diffusion gradually adds noise to a ground truth image X_0 in a process to $X_1, \ldots, X_T$. T should be quite large, so that X_T is roughly pure noise. We will train a neural net to learn the reverse process, $\tilde{X}_T \to \tilde{X}_{T-1} \to \cdots \to \tilde{X}_0$.

The typical approach is $N_i \overset{\text{iid}}{\sim} \mathcal{N}(0,1)$, and take $X_{i+1} = \sqrt{1-\beta} X_i + \sqrt{\beta} N_i$. The idea is that X_i always has mean 0 and variance 1, and X_i is fully attenated and ultimately replaced by Gaussians. In the limit, it is $\mathcal{N}(0,1)$.

References

[1] Chelsea Finn, Pieter Abbeel, and Sergey Levine. *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*. 2017. arXiv: `1703.03400 [cs.LG]`.

[2] Dan Hendrycks et al. *PixMix: Dreamlike Pictures Comprehensively Improve Safety Measures*. 2022. arXiv: `2112.05135 [cs.LG]`.

[3] Ashish Vaswani et al. “Attention Is All You Need”. In: arXiv:1706.03762 (Aug. 2023). arXiv:1706.03762 [cs]. DOI: `10.48550/arXiv.1706.03762`. URL: `http://arxiv.org/abs/1706.03762`.