CS 182/282A: Designing/Visualizing and Understanding Deep Neural Networks

Fall 2022

Lecture 2: August 30 (Tuesday)

Lecturer: Prof. Anant Sahai

Scribes: Connie Huang, Jaewoong Lee

Today

1. Recap. Basic Standard ML Doctrine
2. Empirical Risk Minimization (ERM)-Optimization Perspective (e.g. Generalization)
3. Hyperparameters vs. Parameters
4. Gradient Descent & SGD
5. Intro. to Neural Nets via ReLU Nets

1 Recap. Basic Standard ML Doctrine

1.1 Typical Supervised ML Setup

- Training Data: x_i, y_i, where x_i is input (or covariants), y_i is label, and i = 1, 2, ..., n
- Model: $f_\theta(-)$
- Loss Function: $l(y, \hat{y})$
- Optimizer

Our goal of a supervised ML setup is to make an inference $\hat{y}$ on new data X as follows: $\hat{y} = f_{\hat{\theta}}(X)$, where $\hat{\theta}$ are the learned parameters.

2 Empirical Risk Minimization (ERM)-Optimization Perspective

How do we learn parameters θ? The basic approach is to find the optimal θ for our optimization problem,

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l(y_i, f_\theta(x_i)) \tag{2.1}$$

We can then extend this to the probabilistic interpretation, maximum likelihood (ML) estimation, where our loss function $l(y, \hat{y})$ is interpreted as the negative log-likelihood function.

The big picture goal for supervised machine learning is to achieve good performance in the real world when the model is deployed. In practice, this is difficult to achieve because in the real world, there are unexpected circumstances that we do not have data for and therefore cannot actually represent in our model. As a result, we must use a mathematical proxy so that our model has a

low generalization error. We can model the real world using a probability distribution $P(X, y)$ and aim to minimize the expectation of our loss function with respect to this probability function:

$$E_{X,y}[l(y, f_{\hat{\theta}}(X))]$$

However, our mathematical proxy introduces a few complications.

Complication 1: We do not have access to $P(X, y)$.

Solution: collect a test set $(x_{i,test}, y_{i,test})_{i=1}^{n_{test}}$ to be used once to evaluate our learned model by getting test error.

$$\frac{1}{n_{test}} \sum_{i=1}^{n_{test}} l(y_{i,test}, f_{\hat{\theta}}(x_{i,test}))$$

The model is desired to be tested once because it is not only hard to collect test data but also there is a risk of data incest of test data while designing the model. Test data are not supposed to affect the model.

Complication 2: The loss we care about may be incompatible with our optimizer. For example, our optimizer will use derivatives to find optimal parameters, but our loss function may not have nice derivatives everywhere.

Solution: Use a surrogate loss function $l_{train}(.,.)$ that does have nice derivatives, computes fast, and works with the optimizer. We use this surrogate loss function to *guide* learning of the model. The real loss function is used to evaluate the model. Some standard loss functions include squared error (regression); logistic, hinge, and exponential loss (binary classification); and cross-entropy loss (multiclass classification). You may want to choose a loss function based on the application settings of the problem and model.

This surrogate loss function is different from the evaluation loss function from *complication 1.* The surrogate loss function is used for training the model, and the evaluation loss function is to see how well your model works with new test data. A few things to remember for choosing an appropriate surrogate loss function are *it should be compatible with the optimizer, guide the model to the correct solutions, and run fast enough (e.g. easy to take derivatives).* The squared loss ($l_{train} = (y_i - \hat{y}_i)^2$ or in the vector form, $l_{train} = ||y - \hat{y}||^2$) is a good example of running fast enough.

3 Hyper-parameters & Parameters

Complication 3: We might get *crazy* values for $\hat{\theta}$ (e.g. *over-fitting*). How do we solve this problem?

Solution A: Add a *regularizer* during training.

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\frac{1}{n} \sum_{i=1}^{n} l_{train}(y_i, f_\theta(x_i)) + R(\theta) \right] \tag{3.1}$$

In the above equation, $R(\theta)$ is the regularizer that can be chosen based on what *loss function* is used. For example, if squared loss is used as the loss function, then the *ridge regularization* ($R(\theta) = \lambda||\theta||^2$) might be used as the corresponding *regularizer*. The *ridge regularization* prevents the $\hat{\theta}$ values from becoming too big. The probability interpretation of *regularization* is **Maximum**

A Posteriori (MAP) estimation where $R(\theta)$ corresponds to a prior, which means we want to achieve optimal thetas, given $R(\theta)$, that find a good balance between the unpenalized loss function and $R(\theta)$. Now, realize that we added a new parameter λ to the regularizer. How do we handle λ?

Solution B: Split parameters into two groups: The normal parameters (θ) and hyperparameters (θ_H).

A hyperparameter is a parameter that cannot be trained or the optimizer cannot deal with. For example, if λ was considered as a normal parameter in the above *ridge regularization* example, then λ would end up with an absurd number being assigned (e.g. *0* and *-inf*). Another example of a hyperparameter is the model order (degree) of a polynomial function $f_\theta(x_i)$.

How does the optimizer work with hyperparameters?

Figure 3.1 shows the classical division of data into three categories for hyperparameter fitting. The process of parameters and hyperparameters fitting can be represented as a nested optimization problem with the equations below. Notice that equation (3.3) is equal to equation (3.1) except $R_{\theta_H}(\theta)$.

$$\hat{\theta}_H = \underset{\theta_H}{\operatorname{argmin}} \frac{1}{n_{val}} \sum_{i=1}^{n_{val}} l_{val}(y_{i,val}, f_{\tilde{\theta},\theta_H}(x_{i,val})) \tag{3.2}$$

$$\tilde{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\frac{1}{n} \sum_{i=1}^{n} l_{train}(y_i, f_\theta(x_i)) + R_{\theta_H}(\theta) \right] \tag{3.3}$$

Process

1. Initiate the values of hyperparameters (θ_H)
2. Based on the values of hyperparameters (θ_H), compute the regularized loss with $R_{\theta_H}(\theta)$ on the training data set to get $\tilde{\theta}$ in equation (3.3)
3. Based on the values of normal parameters ($\tilde{\theta}$) and hyperparameters (θ_H), find the best $\hat{\theta}_H$ on the validation data set using equation (3.2)

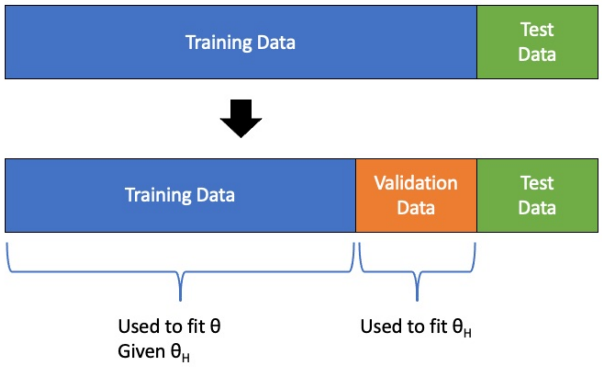


Figure 3.1: Partitioning data for hyperparameter Tuning

You may split the original training data set into the new training and validation data set. However, be careful about data contamination. (e.g. Duplicated data points in each data set. The training and validation data set should be distinct)

Complication 4: The optimizer might have "knobs" (other parameters) associated with it. This might include, for example, the learning rate (or step size) η in gradient descent.

Solution: Include these in θ_H or ignore this problem (i.e. pick a value that has worked in the past. This is a reasonable approach in the light of the limit of the experimentation budget).

4 Gradient Descent and SGD

Gradient Descent (GD) is an iterative approach to optimization (with the spirit of Newton's method) that seeks the local optima taking repeated steps in the opposite direction of the gradient around the current point. Also, Gradient Descent operates under the assumption that it's a linear system. What does the linear assumption mean? The basic idea is to look at the loss function $L_\theta = \frac{1}{n}\sum_{i=1}^{n} l_{train}(y_i, f_\theta(X_i)) + R(\theta, \theta_H)$ in the neighborhood of θ_0 in any place using Taylor Expansion.

$$L_\theta(\theta_0 + \Delta\theta) \approx L_\theta(\theta_0) + (\boldsymbol{\nabla}_\theta L_\theta(\theta_0))^T \Delta\theta$$

Here, θ_0, $\Delta\theta$, and $(\boldsymbol{\nabla}_\theta L_\theta(\theta_0))$ are vectors, and $L_\theta(\theta_0)$ is a scalar. From the equation above, $(\boldsymbol{\nabla}_\theta L_\theta(\theta_0))$ is the gradient around θ_0.

Using this approximation, we can iterate to find our optimal θ.

$$\hat{\theta}_{t+1} = \theta_t + \eta(-\boldsymbol{\nabla}_\theta L_\theta(\theta_t))$$

Notice that the gradient $((\boldsymbol{\nabla}_\theta L_\theta(\theta_t))$, multiplied by a scalar factor (η), at the current time step t is subtracted (taking a negative step) from θ_t. η is the learning rate, which we set to be small enough so that the system converges and big enough so that optimization is not too slow. One problem we introduce with this method is that computing gradients for extremely large datasets can be very computationally intensive. As a result, we introduce **Stochastic Gradient Descent (SGD)**, where instead of using the entire dataset of size n, we randomly choose a representative subset of size n_{batch} from n every iteration to reduce the computation of gradients to only this batch. Because we randomly choose a subset of size n_{batch} every iteration, the overall result over time is a good estimate and trustful.

5 Intro. to Neural Nets via ReLU (Rectified Linear Unit) Nets

5.1 What is a Neural Net (Differentiable Programming)?

A neural net is an object that is easy to take derivatives (e.g. Analog circuits realized as computation graphs with (mostly) differentiable operations compatible with nice vectorization). Moreover, differentiable operations allow nonlinearities.

5.2 Two goals of the analog circuits

1. **Expressivity**: Use the circuit to express the patterns that we want to learn. In other words, the circuit is realization of the function $f_\theta(-)$, and the θs are tunable resistors in the circuit.

2. **Reliably Learnable**: Think of your machine learning system as a microscope where you look at data and the right patterns come into focus.

5.3 Example of Neural Networks

Figure 5.1 shows a 1-D nonlinear (Blue) function, piecewise linear (Red) functions, and data points (Black). As shown in the figure, the piecewise functions describe the nonlinear function pretty well. Our goal is to find a set of piecewise linear functions (Red) that best match the nonlinear function (Blue) based on the data points using Neural Nets. Then, how do we create the piecewise linear functions? A linear combination of elbows (Rectified Linear Units) in **Figure 5.2**! The rectifier circuit in **Figure 5.2** is composed of a diode and a resistor. The diode prevents the current from flowing in the opposite (or negative) direction. Setting the positive direction of the current to be from left to right, it means that the current can never flow in the negative direction (from right to left). All negative currents will be set to zero resulting in V_{out} readings being zero. On the other hand, positive currents (from left to right) will go through the diode resulting in V_{out} readings on the other side of the diode. The standard ReLU function is shown below.

$$f(x) = max(0, x) = \begin{cases} 0 & \text{if x} \leq 0 \\ x & \text{if x} > 0 \end{cases}$$

In this example, V_{in} is x and V_{out} is $f(x)$. Also, the standard ReLU function can be modified if needed. For example, x can be replaced with $wx + b$, so that the modified ReLU function becomes $f(x) = max(0, wx + b)$. Here, w and b are the parameters(θ) we want to minimize using a loss function as mentioned in the previous sections. More details and visualization will be covered in the discussion session and next lecture.

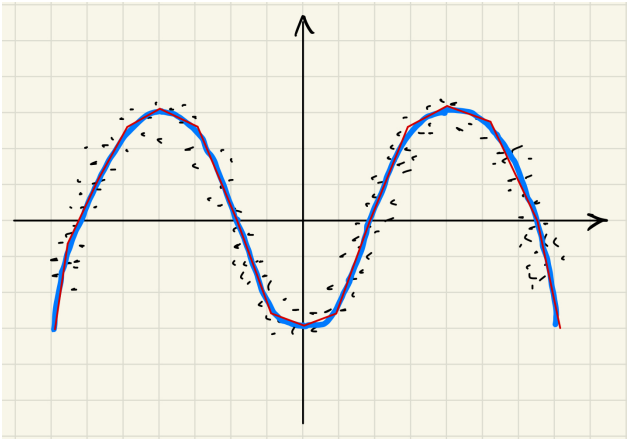


Figure 5.1: 1-D nonlinear and Piecewise linear functions

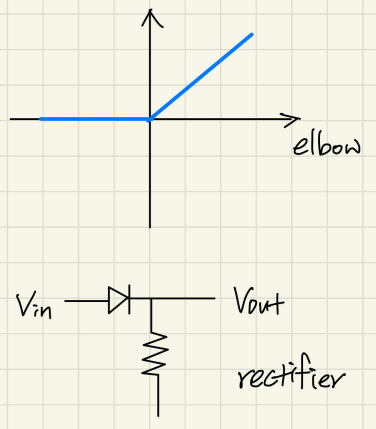


Figure 5.2: Elbow and Rectifier

6 What we wish this lecture also had to make things clearer?

- It would be helpful if more Empirical Risk Minimization (ERM) is covered in this lecture more directly and potentially with diagrams.

CS 182 Lecture 3: Initialization and Regularization

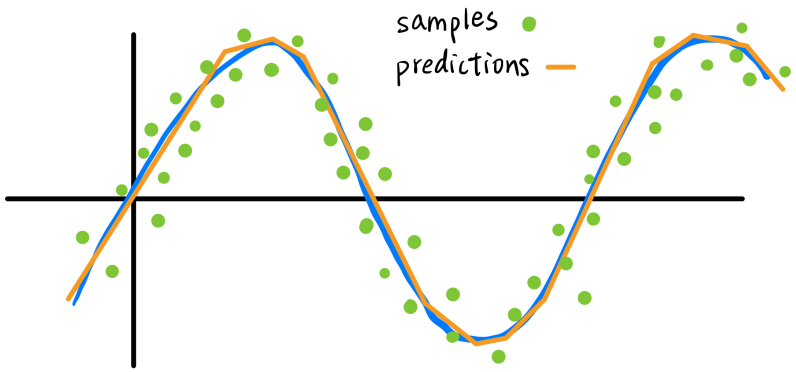
September 1, 2022

Lecturer: Anant Sahai — Scribes: Shreyas Krishnaswamy

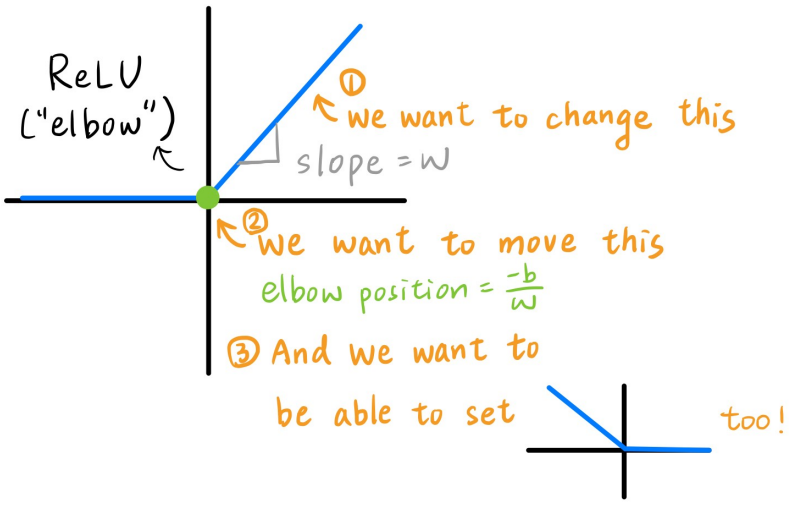
1 Finish up Basic ReLU Net

Neural Nets are easily (for a computer) differentiable function approximators.

1.1 Approximation (e.g., piecewise linear)

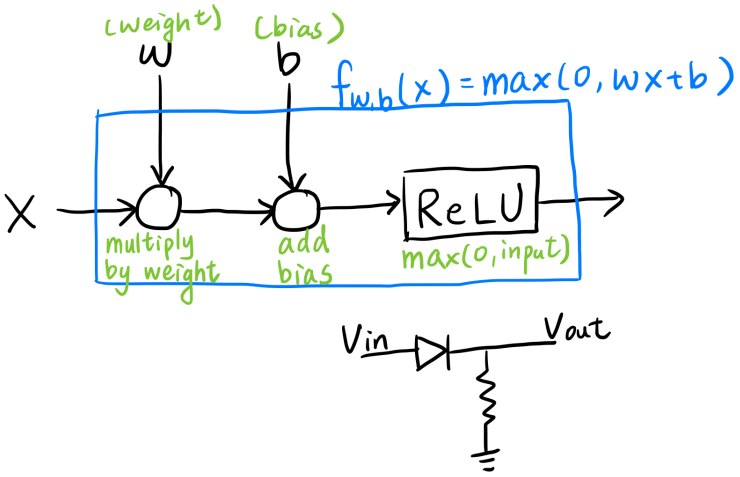


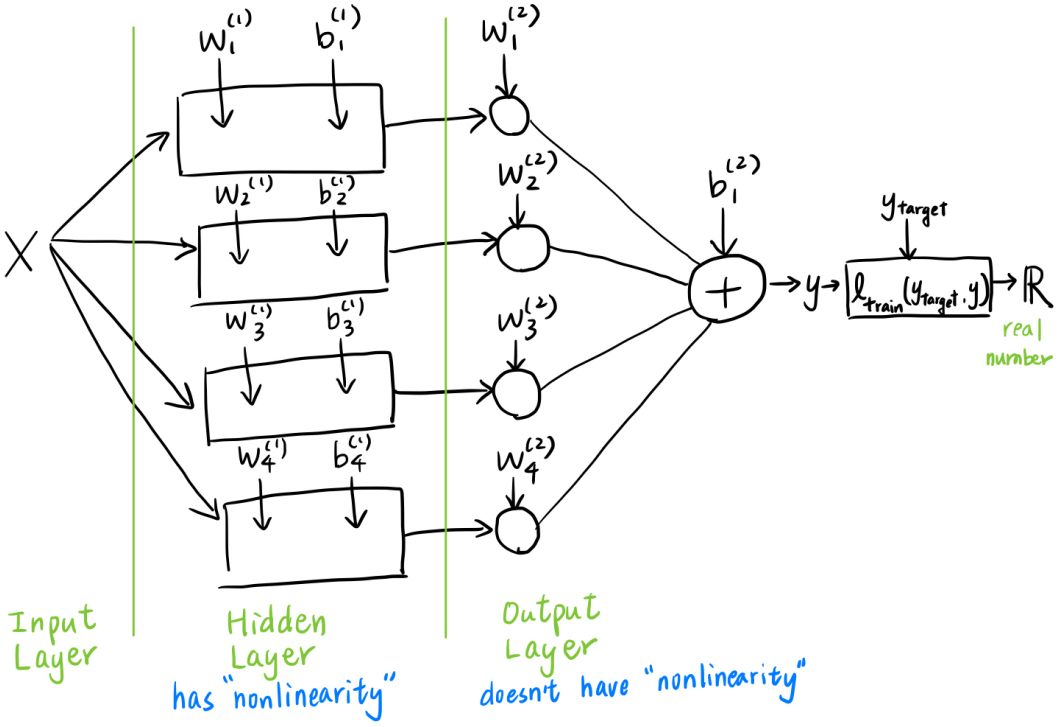
- How do you represent piecewise linear functions in a way that is differentiable for a computer?
  → Build out of "elbows"!



- How does the "elbow" depend on w (weight) and b (bias)?
  → Elbow located at $-\frac{b}{w}$ and slope $= w$

- How do you move the "elbow"?
  → Change w and/or b.





- The optimizer is trying to make the final real number (loss) as small as possible across all the training points. To push the loss down, how much do I have to move y, b's, and w's? This process goes backward and the elbow changes.

1.2 Initialization

Current Basic Folk Wisdom:

- Use whatever worked on a related problem. (e.g., pre-trained network, literature review, etc)
- Random initialization using Gaussian $N(0, \sigma^2)$.
    * Xavier Initialization: $\sigma^2 = \frac{1}{d}$, where d is the fan-in of this unit.
    * $Var(\sum_{i=1}^{d} X_i) = dk$ (X_i are independent, has $mean = 0$ and $var = k$). Note that all variances add up to 1.

Xavier initialization is appropriate when the nonlinearity is not a ReLU (e.g. hyperbolic tangents, sigmoids, etc.). For ReLUs, **He initialization** (a.k.a. Kaiming initialization) is appropriate.

He Initialization

$$\text{Gaussian } N\left(0, \frac{2}{d}\right)$$

Why $\frac{2}{d}$ instead of $\frac{1}{d}$? When initializing ReLU weights, we want our ReLU elbows to be close to where the action is. In other words, we want the ReLUs to actually produce non-linearities, so our model can emulate nonlinear phenomena.

If this is the case, about half of our ReLUs should be in the off-state, so the actual fan-in for ReLUs is halved, which is why there's a two in the numerator of $\frac{2}{d}$.

There are many possibilities for initializing the biases. Some include:

- Xavier initialization on the biases
    * Motivation: the bias can be considered another weight. Instead of using d, use $d + 1$ (i.e. add the bias's fan-in) when initializing the corresponding unit.
- Make them all 0
- Make them all small random numbers
- Make them all the same small number (e.g. 0.01)

For different situations, some of these work better than others.

1.3 Aside: Dead ReLUs

If both weights and biases are distributed with normal distributions, the ratio $\frac{b}{w}$ will be a Cauchy distribution. This causes some ReLUs to be located far apart from the others. These ReLUs are considered "dead" since they output 0, and changing the weights or biases slightly doesn't affect their output.

2 Revisiting Regularization

2.1 Regularization in the Loss Function

This section explores **regularization**. Let's begin by looking at general loss functions with and without regularization.

Loss function without regularization

$$L_\theta = \frac{1}{n}\sum_{i=1}^{n} \ell_{train}\left(y_i, f_\theta\left(x_i\right)\right)$$

$x_i = i$th data point (model input)
$y_i =$ Ground truth (i.e. expected) output for data point x_i
$f_\theta =$ Model with parameters θ
$n =$ Number of training data points
$\ell_{train} =$ Loss on training data point i
$L_\theta =$ Total loss

This function tracks the loss between our parameterized model's outputs and the expected outputs. We can add a regularization function to it to create a loss function with regularization.

Loss function with regularization

$$L_\theta = \frac{1}{n}\sum_{i=1}^{n} \ell_{train}\left(y_i, f_\theta\left(x_i\right)\right) + R(\theta)$$

$R(\theta) =$ Regularization term on model parameters θ

2.2 Regularization in Least Squares

For a more intuitive understanding of regularization, we can look at least squares. Let's start by reviewing the problem: we have a matrix X and vector $\vec{y}$, and we want to calculate the weights $\vec{w}$ that best approximates $X\vec{w} = \vec{y}$.

Ordinary Least Squares (OLS)

Problem: $\text{argmin}_w \; ||\vec{y} - X\vec{w}||^2$

Solution: $\hat{\vec{w}} = \left(X^T X\right)^{-1} X^T \vec{y}$

We can add regularization to OLS. One version of regularized OLS is called ridge regression:

Ridge Regression

$$\text{Problem: } \text{argmin}_w \ ||\vec{y} - X\vec{w}||^2 + \lambda||\vec{w}||^2$$

$$\text{Solution: } \hat{\vec{w}} = \left(X^T X + \lambda I\right)^{-1} X^T \vec{y}$$

λ is the **regularization parameter**. It penalizes high-magnitude weight vectors (can you see why?). We can select different λ values to control the behavior of ridge regression. When λ is higher, the weight magnitude penalty is more severe; when it's lower, the penalty is lighter, which allows ridge regression to settle on weight vectors with higher magnitudes.

Note that when $\lambda = 0$, ridge regression becomes unregularized. This unregularized formulation simplifies to the OLS problem and solution. Intuitively, setting $\lambda = 0$ tells ridge regression to disregard the weight vector's magnitude altogether, allowing it to solve the problem as though it were simply OLS.

2.2.1 Gradient Descent

We can also solve least squares with **gradient descent**. Gradient descent is a process where we aim to improve our model by repeatedly moving its parameters in the direction that minimizes the cost function. For least squares, the parameters are stored in $\hat{\vec{w}}$.

The gradient for least squares without regularization is

OLS Gradient Descent

$$\text{Gradient (with step size): } \eta 2X^T \left(\vec{y} - X\hat{\vec{w}}\right)$$

$$\text{Update step: } \hat{\vec{w}}_{(t+1)} = \hat{\vec{w}}_{(t)} + \eta 2X^T \left(\vec{y} - X\hat{\vec{w}}_{(t)}\right)$$

$$\begin{aligned}
\hat{\vec{w}}_{(t+1)} &= \text{Weights } \hat{\vec{w}} \text{ at timestep } t+1 \\
\hat{\vec{w}}_{(t)} &= \text{Weights } \hat{\vec{w}} \text{ at timestep } t \\
\eta &= \text{Step size} \\
\vec{y} - X\hat{\vec{w}}_{(t)} &= \text{Residual at timestep } t
\end{aligned}$$

We can create a similar update step for ridge regression:

Ridge Regression Gradient Descent

$$\text{Gradient (with step size): } = \eta \left(2X^T \left(\vec{y} - X\hat{\vec{w}}\right) - 2\lambda\hat{\vec{w}}\right)$$

$$\text{Update step: } \hat{\vec{w}}_{(t+1)} = (1 - 2\eta\lambda)\, \hat{\vec{w}}_{(t)} + \eta 2X^T \left(\vec{y} - X\hat{\vec{w}}_{(t)}\right)$$

Note that the ridge regression update step can be derived by rearranging the gradient and adding it to the weights at time t.

The coefficient $(1 - 2\eta\lambda)$ causes an effect called **weight decay**. On top of adding an update, the regularized update also decays the weights, making the system of gradient descent more stable.

2.3 Data Augmentation View

We can also look at regularization through the **data augmentation** lens. Suppose we want to solve an OLS problem where we have appended fake data points to our X matrix. For example:

Data-Augmented System

$$\begin{bmatrix} X \\ \sqrt{\lambda}I \end{bmatrix} \vec{w} \approx \begin{bmatrix} \vec{y} \\ \vec{0} \end{bmatrix}$$

Dimensions

$$\begin{aligned} X &: n \times d \\ \sqrt{\lambda} &: \text{constant} \\ I &: d \times d \ \text{(identity)} \\ \vec{w} &: d \times 1 \\ \vec{y} &: n \times 1 \\ \vec{0} &: d \times 1 \end{aligned}$$

$\sqrt{\lambda}I$ represents some fake data. We append zeros $\left(\vec{0}\right)$ to $\vec{y}$ in order to match our output's dimensions to the augmented X matrix. Once again, λ is a regularization parameter.

We can apply the OLS solution to this system:

$$\left(\begin{bmatrix} X \\ \sqrt{\lambda}I \end{bmatrix}^T \begin{bmatrix} X \\ \sqrt{\lambda}I \end{bmatrix} \right)^{-1} \begin{bmatrix} X \\ \sqrt{\lambda}I \end{bmatrix}^T \begin{bmatrix} \vec{y} \\ \vec{0} \end{bmatrix} = \left(X^T X + \lambda I\right)^{-1} X^T \vec{y}$$

The output matches the ridge regression solution!

What's happening here? Intuitively, each of the data points added to the X matrix is trying to make the corresponding feature go to 0 in order to fit to the zeros we appended to $\vec{y}$ – just like a regularizer! Recall that ridge regression similarly penalized high-magnitude weight vectors, which also added pressure to reduce the weight vector's magnitude.

The data augmentation view is useful because it's intuitive to generalize. We can add modified or fake data like we did in this example to other machine learning problems or architectures in order to regularize them.

2.4 Feature Augmentation

Another approach to regularization is **feature augmentation**, where we add irrelevant features to our system.

Feature-Augmented System

$$\begin{bmatrix} X & \sqrt{\lambda}I \end{bmatrix} \begin{bmatrix} \vec{w} \\ \vec{f} \end{bmatrix} = \vec{y}$$

Dimensions

$$\begin{aligned} X &: n \times d \\ \sqrt{\lambda} &: \text{constant} \\ I &: n \times n \ \text{(identity)} \\ \vec{w} &: d \times 1 \\ \vec{f} &: n \times 1 \\ \vec{y} &: n \times 1 \end{aligned}$$

$\sqrt{\lambda}I$ represents some fake features that we have added to the X matrix. To make the dimensions match, we have appended fake weights $\vec{f}$ (that we don't care about) to $\vec{w}$ (which we do care about).

Note that since the augmented X matrix is wider than it is tall, the problem uses strict equality. This also means the system has infinitely many solutions, and we need to pick one of them.

For now, let's use the Moore-Penrose Pseudoinverse to find a minimum-norm solution to this problem.

$$\begin{aligned} \begin{bmatrix} \hat{\vec{w}} \\ \hat{\vec{f}} \end{bmatrix} &= \begin{bmatrix} X^T \\ \sqrt{\lambda}I \end{bmatrix} \left(\begin{bmatrix} X & \sqrt{\lambda}I \end{bmatrix} \begin{bmatrix} X^T \\ \sqrt{\lambda}I \end{bmatrix} \right)^{-1} \vec{y} \\ \hat{\vec{w}} &= X^T \left(XX^T + \lambda I \right)^{-1} \vec{y} \end{aligned}$$

We arrive at the second expression above by discarding the fake weights $\hat{\vec{f}}$ since we're only looking for the weights stored in $\hat{\vec{w}}$.

Note that the final expression for the $\hat{\vec{w}}$ vector looks a bit different than the OLS solutions from the other setups. This expression is actually the dual perspective of ridge regression, called **kernel ridge regression**.

This means that augmenting with features actually has a regularizing effect. This phenomenon is one of the reasons people say deep neural networks are effective. Not only are they more expressive, but by adding many extra features, they may also be more effective regularizers.

EECS 182/282A Deep Neural Networks Lecture 4 - 09/06/2022

Lecture 4: Basic Principles Part II

Instructor: Anant Sahai *Scribe: Austin Zane*

1 Regularization

1.1 Explicit Regularization

This is a recap of the previous lecture. For the ordinary least squares problem $\boldsymbol{X}\boldsymbol{w} \approx \boldsymbol{y}$, the solution is given by

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y}.$$

However, this can give us very large and not very useful values for the parameter vector. In such cases, it is often beneficial to "regularize" the parameters when training so that they stay reasonable. A common choice is ridge regularization, which uses a modified version of the least-squares loss function:

$$\arg\min_{\boldsymbol{w}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2.$$

A bit of vector calculus shows that the new solution for $\hat{\boldsymbol{w}}$ is given by

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y} = \boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y},$$

where the first formula is the classic ridge form and the second is kernel ridge form. The equivalence can be seen as follows,

$$\begin{aligned}
\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y} &= \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y} \\
&= \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{X}^\mathsf{T}\right)\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y} \\
&= \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)\boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y} \\
&= \boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y}.
\end{aligned}$$

There is a way of thinking of the first formulation as solving the primal and the second as solving the dual.

1.2 Data and Feature Augmentation

Instead of explicitly changing the loss function, we can add d "fake" data points to our data matrix to achieve the same regularizing effect (as proven in previous lecture):

$$\begin{bmatrix}\boldsymbol{X} \\ \sqrt{\lambda}\boldsymbol{I}_d\end{bmatrix}\boldsymbol{w} \approx \begin{bmatrix}\boldsymbol{y} \\ \boldsymbol{0}_d\end{bmatrix},$$

where the new data matrix is in $\mathbb{R}^{(n+d)\times d}$ and the new response is a vector in $\mathbb{R}^{n+d}$. This is an important concept and can be thought of as improving the conditioning number of the data matrix. Plugging the new data matrix and response vector into the classic OLS solution immediately gives us the solution for the ridge regularized problem.

Another equivalent option is adding n fake features:

$$\begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\begin{bmatrix}\boldsymbol{w}\\ \boldsymbol{f}\end{bmatrix} = \boldsymbol{y}.$$

We use the Moore-Penrose pseudoinverse to solve for the new weight vector,

$$\begin{bmatrix}\boldsymbol{w}\\ \boldsymbol{f}\end{bmatrix} = \begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}^{\dagger}\boldsymbol{y} = \begin{bmatrix}\boldsymbol{X}^{\mathsf{T}}\\ \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\left(\begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\begin{bmatrix}\boldsymbol{X}^{\mathsf{T}}\\ \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\right)^{-1}\boldsymbol{y}.$$

We are only interested in solving for $\boldsymbol{w}$, so we disregard the n rows corresponding to the false parameter $\boldsymbol{f}$. In the end, we are left with the minimum norm solution to the under-determined system specified above,

$$\boldsymbol{w} = \boldsymbol{X}^{\mathsf{T}}\left(\begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\begin{bmatrix}\boldsymbol{X}^{\mathsf{T}}\\ \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\right)^{-1}\boldsymbol{y} = \boldsymbol{X}^{\mathsf{T}}\left(\boldsymbol{X}\boldsymbol{X}^{\mathsf{T}} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y}.$$

1.3 Using Singular Value Decomposition to Simplify Regularization

Using the singular value decomposition (SVD) in place of $\boldsymbol{X}$ in the above equations allows us to simplify things and make the algorithms far easier to handle. Namely, we are able to update each weight individually instead of solving systems of equations. This can be seen by taking the SVD of $\boldsymbol{X}$ and solving the unregularized problem

$$\begin{aligned}\boldsymbol{X}\boldsymbol{w} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathsf{T}}\boldsymbol{w} &\approx \boldsymbol{y}\\ \Rightarrow\ \boldsymbol{\Sigma}\widetilde{\boldsymbol{w}} &\approx \widetilde{\boldsymbol{y}},\end{aligned}$$

where $\widetilde{\boldsymbol{w}} := \boldsymbol{V}^{\mathsf{T}}\boldsymbol{w}$, $\widetilde{\boldsymbol{y}} := \boldsymbol{U}^{\mathsf{T}}\boldsymbol{y}$, and $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times d}$. Note that $\boldsymbol{U}$, $\boldsymbol{V}$ are orthogonal matrices so they merely rotate our vectors while preserving their norms. Let σ_i denote the i^{th} singular value. Because $\boldsymbol{\Sigma}$ is diagonal and we are assuming that $n > d$, only the first d equations will be meaningful here:

$$\begin{aligned}\text{for } d \text{ equations: } \sigma_i\widetilde{\boldsymbol{w}}[i] \approx \widetilde{\boldsymbol{y}}[i] \ \Rightarrow\ \widetilde{\boldsymbol{w}}[i] &\approx \frac{1}{\sigma_i}\widetilde{\boldsymbol{y}}[i],\\ \text{for } n-d \text{ equations: } 0 &\approx \widetilde{\boldsymbol{y}}[i].\end{aligned}$$

As previously stated, using these coordinates removes the system of equations and allows us to simply solve for the individual weight components. Observe that we are dividing by the singular values so problems may arise if they become too small (i.e. if the matrix is ill-conditioned). This is the underlying cause of OLS sometimes giving us wild values.

Next, we consider using ridge regression. In this setting, the solution is obtained by plugging the SVD of $\boldsymbol{X}$

into the previously mentioned classic solution for ridge regression.

$$\begin{aligned}\hat{\boldsymbol{w}} &= \left(\boldsymbol{V}\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\boldsymbol{V}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{V}\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{U}^\mathsf{T}\boldsymbol{y} \\ &= \boldsymbol{V}\left(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{V}^\mathsf{T}\boldsymbol{V}\boldsymbol{\Sigma}^\mathsf{T}\widetilde{\boldsymbol{y}} \\ &= \boldsymbol{V}\left(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Sigma}^\mathsf{T}\widetilde{\boldsymbol{y}} \\ \Rightarrow \widehat{\widetilde{\boldsymbol{w}}} &= \left(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Sigma}^\mathsf{T}\widetilde{\boldsymbol{y}}.\end{aligned}$$

In the end, we are left with solutions of the form

$$\widetilde{\boldsymbol{w}}[i] = \frac{\sigma_i}{\sigma_i^2 + \lambda}\widetilde{\boldsymbol{y}}[i].$$

If $\lambda << \sigma_i^2$, then we are in the same situation as the unregularized case. If $\lambda >> \sigma_i^2$, then the weights are forced to stay small instead of behaving wildly.

1.4 Implicit Regularization

Implicit regularization is the regularization that occurs when we aren't consciously doing any regularization. When performing explicit regularization as above, we must specify a specific regularization hyperparameter and modify the training data, model architecture, or loss function. In contrast, implicit regularization is an unexpected benefit stemming from our choice of optimizer. We will see that choosing gradient descent as our optimization algorithm, combined with the large size of DNNs, provides enough regularization for DNNs to generalize well without us intentionally restricting the parameters.

To gain intuition, let's look at gradient descent (GD) updates for OLS in SVD coordinates:

$$\widetilde{\boldsymbol{w}}_{t+1} = \widetilde{\boldsymbol{w}}_t + 2\eta\boldsymbol{\Sigma}^\mathsf{T}\left(\widetilde{\boldsymbol{y}} - \boldsymbol{\Sigma}\widetilde{\boldsymbol{w}}_t\right),$$

where $\widetilde{\boldsymbol{w}}_t$ represents the parameter vector at the current step, $\widetilde{\boldsymbol{w}}_{t+1}$ represents the updated parameter vector, $\boldsymbol{\Sigma}$ represents the diagonal matrix of singular values from the above SVD, $\widetilde{\boldsymbol{y}} := \boldsymbol{U}^\mathsf{T}\boldsymbol{y}$ as defined above, and η is our learning rate hyperparameter.

Note that because we are dealing with a diagonal matrix, this works out to updating each component of the weight vector individually. They don't interact with each other at all during GD, so each is being modified as follows:

$$\widetilde{\boldsymbol{w}}_{t+1}[i] = \widetilde{\boldsymbol{w}}_t[i] + 2\eta\sigma_i\left(\widetilde{\boldsymbol{y}}[i] - \sigma_i\widetilde{\boldsymbol{w}}_t[i]\right).$$

This is potentially unstable because we are not reducing $\widetilde{\boldsymbol{w}}_t[i]$ at each step as we did in the last lecture. This means that, subject to a bounded input, we might get an unbounded output if we allow the algorithm to run forever. Observe that the stationary point is the solution we discussed earlier, $\widetilde{\boldsymbol{w}}[i] = \frac{1}{\sigma_i}\widetilde{\boldsymbol{y}}[i]$. If σ_i is tiny, then we find ourselves in a bad situation.

Let's carefully calculate the first few steps of GD to see what's going on:

$$\begin{aligned}&\widetilde{\boldsymbol{w}}_0[i] = 0 \\ &\widetilde{\boldsymbol{w}}_1[i] = 2\eta\sigma_i\widetilde{\boldsymbol{y}}[i] \\ &\widetilde{\boldsymbol{w}}_2[i] = 2\eta\sigma_i\widetilde{\boldsymbol{y}}[i] + 2\eta\sigma_i\left(\widetilde{\boldsymbol{y}}[i] - \sigma_i 2\eta\sigma_i\widetilde{\boldsymbol{y}}[i]\right) \approx 4\eta\sigma_i\widetilde{\boldsymbol{y}}[i].\end{aligned}$$

Observe that this is roughly a linear function with an extremely small slope if σ_i is tiny. In this situation, GD barely moves in the early stages even though it will eventually converge to a very large value, as previously discussed. Together with early stopping, this means that GD is trying to do something like ridge regularization for us because it will resist enlarging the directions corresponding to small singular values. Early stopping is when we stop the training process because validation performance has gotten worse or has not improved for a long time. It is important to note that GD, when initialized at zero, will converge to the minimum-norm solution. This is a good exercise for the reader to verify.

To summarize, there are three kinds of regularization: explicit regularization, data augmentation (adding fake observations or features), and implicit regularization (optimizer has implicit regularizing effect). Regarding DNNs, the combination of the min-norm seeking behavior of gradient descent and the feature augmentation that is implicit when using large networks gives a lot of regularization even if we weren't thinking about it.

2 Trade-offs Between Qualitatively Different Sources of Error

Suppose we have learned a model $\hat{\theta} \to f_{\hat{\theta}}$. At "test time", we look at the error, $\left(Y_{observed} - f_{\hat{\theta}}(x)\right)$. There are three main sources of error:

1. *Irreducible error*: This is due to noise or randomness from $Y|X$ itself. In short, there is some level of noise that is impossible for our model to account for. One possible situation is that the underlying response is a deterministic function of X (i.e. not random), but there is randomness in the measurement. It is possible that our model perfectly predicts the underlying signal, but the model will still disagree with $Y_{observed}$ and contribute to the error. In the classic setup of $y = f_{true}(x) + \boldsymbol{\varepsilon}_{noise}$, the $\boldsymbol{\varepsilon}_{noise}$ term contributes to irreducible error.
2. *Approximation error*: This error comes from limited expressive power of f_θ as a finitely-parameterized model. In other words, our model isn't "flexible" enough to capture the true signal. For example, trying to fit the function $y = \cos(x)$ using a single 6th degree polynomial model, $f_\theta(x) = \sum_{i=0}^{6} a_i x^i$.
3. *Estimation error*: There are two components to this type of error, both of which are well-covered in prerequisite machine learning courses:
   (a) *Bias*: Bias captures the systematic error of our learning algorithm and training data in terms of making predictions. We write this mathematically by $\mathbb{E}_{\boldsymbol{\varepsilon},\mathcal{D}}[f_{\hat{\theta}(\mathcal{D})}(X) - Y \ |X]$. Note that $\boldsymbol{\varepsilon}$ respresents the randomness in $Y|X$ and $\mathcal{D}$ represents the randomness in the training process used to get $\hat{\theta}$. For example, this could include the training data set we used or the splits made in random forest trees. Traditionally, X is separate from $\mathcal{D}$ and is not random. It is common to look at the squared bias to prevent positive and negative biases for different observations from canceling.
   (b) *Variance*: This is the variable part of error. We write it as $\mathbb{E}_{\mathcal{D}}[(f_{\hat{\theta}(\mathcal{D})}(X) - \mathbb{E}_{\mathcal{D}}[f_{\hat{\theta}(\mathcal{D})}(X)])^2|X]$, where $\mathcal{D}$ again represents the randomness of the training process. It describes how much our prediction "shakes" as a function of the randomness in the training.

This perspective can be useful and many papers utilize the bias-variance decomposition in their derivations. However, it doesn't always match what our intuition might be, especially in deep learning settings. The following figure is intended to help us understand this point.

This drawing is for high-level intuition, so we musn't allow ourselves to become confused over the exact dimensions and projections. We first turn our attention to the figure on the left. The line passing through

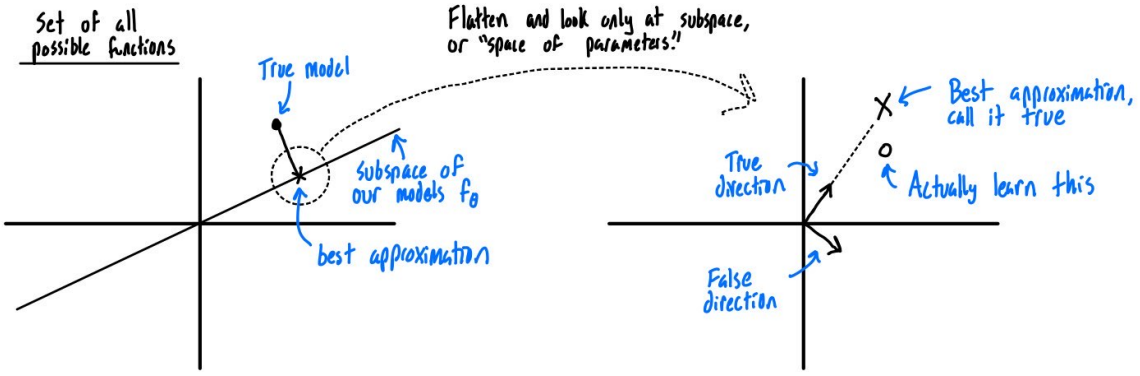


Figure 1: Approximation and Estimation Error

the origin represents the subspace models that we are considering. Note that we are only capable of providing estimates that fall along this line. As we can see, the true model is not in this subspace so we will suffer a certain amount of approximation error.

Turning our attention to the figure on the right, we project the true model onto the subspace of models that we are considering. The result is the best possible approximation and is the goal of our learning algorithm. We see in the figure that our predicted model is not quite equal to the best approximation. In this perspective, instead of thinking of things in terms of bias and variance, we consider how much of the "true" and "false" directions we are incorporating into our prediction.

We make this a bit more formal by discussing *survival* and *contamination*. Survival reflects, in expectation, how much of the true pattern survives the estimation/learning process. Contamination is how much useless information get "learned", e.g. spurious features that our model is capable of picking up but don't help with prediction. Survival and contamination can be thought of as the intuition behind bias and variance, respectively.

3 What are "Features"?

The following is a simplified sketch of a neural network with ℓ hidden layers.

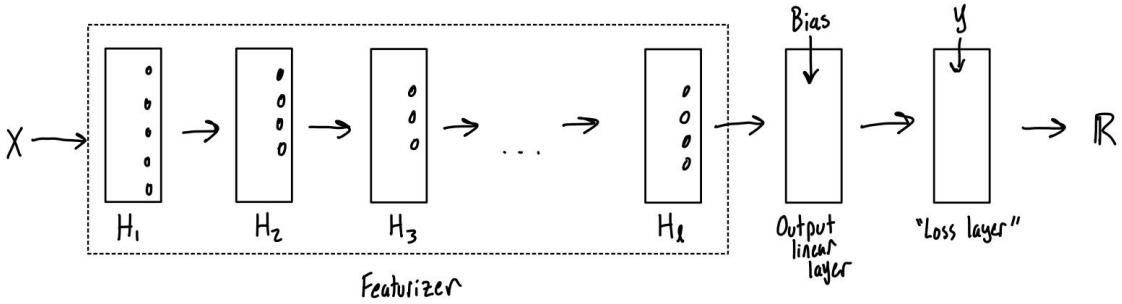


Figure 2: Simplified DNN

If we treat everything before the output of H_ℓ as a black box, we can think of things from the perspective of a generalized linear model. We have a featurizer, some linear function of those new features, and a loss that we are optimizing. The featurizer lifts or distills the input X into a nicer feature space. In this perspective, the "learned" features are the outputs of the penultimate layer in the featurizer, H_ℓ. We want the featurization to be data-driven instead of hand-picked, so the layers essentially find a representation of the data that allows the generalized linear model (GLM) to work well.

However, there is another important point of view. Suppose a generalized linear model is given by $\hat{y} = \sum_{i=1}^{\lambda} w_i \phi_i(\boldsymbol{x})$, where $\boldsymbol{\phi(x)}$ is the output of the "featurizer" in the figure above. When we are actually using the model on new data, these are simply the features. However, from the training perspective, they also determine the gradient. The derivative of the linear model with respect to the i^{th} parameter is $\frac{\partial \hat{y}}{\partial w_i} = \phi_i(\boldsymbol{x})$.

In short, we don't understand nonlinear systems well, so our standard approach to understanding a nonlinear system is local linearization. We zoom in until things are roughly linear around a certain point. In terms of DNNs, we say that the deep network is Taylor expanded around the features in such a way that it is some constant term plus a local GLM in which small increments of the features change the predictions in a small way. This will be covered in greater detail during the next lecture.

CS282 Lecture 5: Survey of Architectures and Problems

Lecturer: Anant Sahai, Scribes: Gabrielle Hoyer, Kevin Tsai

September 2022

1. Last Lecture Recap

In Lecture 4, we talk about using regularization to prevent our dependence on direction in the SVD space (direction with small singular values) that we don't trust. Various methods of regularization are mentioned and discussed, in particular we analyze:

1.1 Explicitly Adding Terms to the Cost Function

This corresponds to transforming the optimization problem to:

$$\min_{\boldsymbol{\theta}} \quad (l_{\text{train}}(\boldsymbol{\theta}) + R(\boldsymbol{\theta}))$$

where $R(\boldsymbol{\theta})$ is a regularization term, e.g., for l2 regularization this could be:

$$R(\boldsymbol{\theta}) = \lambda ||\boldsymbol{\theta}||_2^2$$

Under l2 regularization and least squares loss function, the transformed problem is the classical ridge regression with solution $\boldsymbol{\theta}^*$ where X is the data matrix and $\mathbf{y}$ is the target vector.

$$\boldsymbol{\theta}^* = (X^T X + \lambda I)^{-1} X^T \mathbf{y}$$

1.2. Weight Decay (Explicit in the Algorithm)

With ridge regression, the penalty term $\lambda ||\boldsymbol{\theta}||_2^2$ inside the cost function has a gradient that is proportional to the weight ($\boldsymbol{\theta}$) itself. When performing gradient descent, this amounts to scaling down the norm of the weight (provided that the learning rate is sufficiently small), i.e., $\boldsymbol{\theta}_{t+1} = (1 - 2\lambda\zeta)\boldsymbol{\theta}_t +$ *usual terms for gradient descent*, where ζ is the learning rate (this is like pulling the weights down). This form of regularization is listed separately from the first one even though it might appear to be equivalent; this is because for applications other than ridge regression (with gradient descent), sometimes a general optimizer (or algorithm) might not work well with the l2-penalized cost function, in which case, we can still perform the weight decay regardless.

1.3. Data Augmentation

It is also possible to mimic regularization by inserting artificial observations into the data matrix, i.e.

$$\begin{bmatrix} X \\ \sqrt{\lambda} I_d \end{bmatrix} \boldsymbol{\theta} \approx \begin{bmatrix} \mathbf{y} \\ \mathbf{0}_d \end{bmatrix}$$

This is important for neural networks, e.g., adding flipped/rotated images into the training data for a convolutional neural network.

1.4. Feature Augmentation

Instead of extending the data matrix row-wise, we can also extend it column-wise, e.g., by appending artificial features to each existing observation.

$$\begin{bmatrix} X & \sqrt{\lambda} I_d \end{bmatrix} \begin{bmatrix} \boldsymbol{\theta} \\ \boldsymbol{\theta}' \end{bmatrix} \approx \begin{bmatrix} \mathbf{y} \end{bmatrix}$$

This is also important for neural networks; we will see in section 2 that we have just as many weights (parameters) as we have features in a neural net, and these many features can have the desired regularization effect.

1.5. Implicit Regularization during Optimization

Optimization algorithms such as gradient descent implicitly implement regularization, even when the objective function does not include the regularization penalty. Features (in the SVD space) are more favored if they correspond with large singular values (i.e., in each iteration, they take bigger steps), whereas those with small singular values move hardly at all. For those "good" features with large singular values, we hope that they are likely to correspond to true patterns. On the contrary, those with small singular values, we hope they are spurious signals.

In addition to the regularization effect introduced by how the optimization algorithm responds to different singular values, early stopping is another implicit mechanism we have for most iterative algorithms; this effectively prevents the weights from becoming arbitrarily large.

1.6. Bias-variance Trade-off

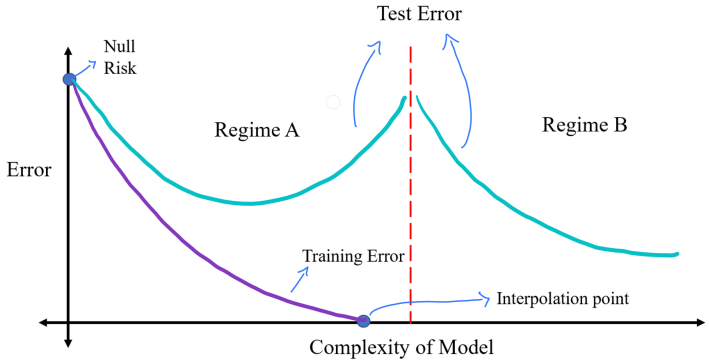


Figure 1: Training Error and Test Error vs Model Complexity

A common plot in traditional machine learning is the evolution of training error and test error as the complexity of a model increases. The complexity of a model can be the number of parameters estimated, the degrees of polynomials used, or the number of hidden layers/units in a neural network, etc. Figure 1 shows a simple demonstration of such a plot: When the model has zero complexity (e.g., an intercept model) the training error is equal to the test error and their specific values are called the null risk (note that they need not be identical). As the model becomes more complicated, the training error usually decreases faster than the test error, and at some point, the test error ceases to decrease and begins to increase (what people usually call overfitting) whereas the training error approaches zero

eventually (which could potentially be concerning if there is non-reducible error in the data generating process and a zero error implies the model is fitting the noise). The reason why the test error begins to increase can be attributed to the increase in estimation error with the increased complexity and the decrease in approximation error (from the more complex model) can not compensate for that.

It is possible, however, that the test error in the overfit regime can again start to decrease when the model becomes even more complicated. This is apparently what practitioners of deep neural networks observe and most deep neural networks that work reasonably well are thought to be in this regime (Regime B in Figure 1). One would be concerned if a traditional machine learning model ends up with zero training error (whatever it means in the context) as it indicates overfitting, but when it comes to deep neural networks, people would actually only start to consider the model if it can attain zero training error because they believe a well-designed deep neural net should be operating in Regime B. The 4th (feature augmentation) and 5th (implicit regularization) regularization methods mentioned above may be at work to contribute to the reduction in test error in this regime (a potential decrease in approximation error is also a possible cause of reduction). It is likely that these two regularization methods help control the estimation error when deep neural networks perform well.

Although we know the 4th and 5th regularization methods are important for neural networks, for practical reasons, it is often the case that practitioners would try all 5 methods of regularization mentioned above. One of the reasons why someone might want to do that is because these methods do not work independently from each other; e.g., if we want to take advantage of implicit regularization and we want the feature direction with large singular values to match what we believe to be true, we probably need other regularization techniques to encourage the behavior.

Last, despite the fact that our discussion centers around training a neural network, the behavior in Regime B (small test error with a really complex model) can also be observed in other traditional machine learning models, e.g., tree-based models and kernel methods. One of the first realizations of such behavior is said to have been discovered with boosted trees.

2. Features

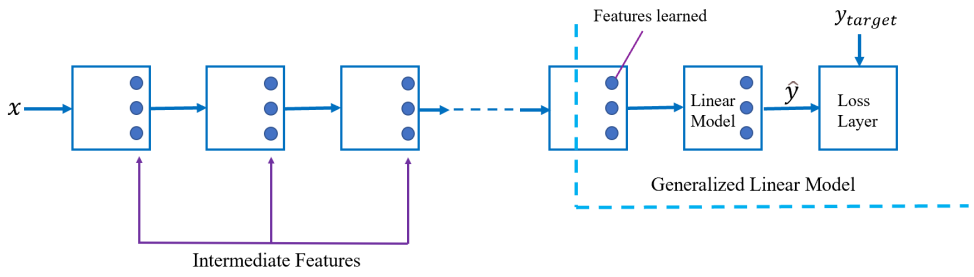


Figure 2: Generalized Linear Model with intermediate features

2.1 Neural Net Represented as a Generalized Linear Model

When thinking about deep neural networks, it is useful to consider the optimization algorithm from a local linear perspective. The below equation states an interesting relationship between $\hat{y}$, our model, x, the training data, and θ, the learned weights. Specifically, the equation describes that $\hat{y}(x)$ for given current parameters relative to the parameters to change, is equal to $\hat{y}$ for x relative to current parameters and outcomes, plus the partial derivative of $\hat{y}$ relative to all parameters, evaluated at θ_0, times the desired

hypothetical parameter change.

$$\hat{y}(x) = \hat{y}(x, \theta_0) + \left.\frac{d\hat{y}}{d\theta}\right|_{\theta_0} \cdot \Delta\vec{\theta}$$

From this perspective, one can see that despite the many possible layers within a neural net, the algorithm is working on a generalized linear model (though differently centered) with a feature corresponding to every parameter. The $\Delta\vec{\theta}$ vector is the size of the number of parameters that can be learned; therefore, $\frac{d\hat{y}}{d\theta}$ is this same size. In this way, the generalized linear model can be described by the following equation in which our model, $\vec{\theta}$ is acting on some featurization vector, $\vec{\phi}(x)$.

$$y = \vec{\theta}^T \vec{\phi}(x)$$

Specifically, this construes the lifting of x to a set of features, which are functions of x, to return scalars. This is turn is multiplied by the weights, and the output is this linear combination. While we do not fully understand the nuances of deep neural networks, we understand linear models very well, thus the simplification of our optimization problem by our algorithm in a local perspective is quite useful.

When thinking about our deep neural network, it is important to understand that the Gradient Descent direction is not determined from the features of the penultimate layer,"features learned" in Figure 2, but rather the the derivative feature matrix $\frac{d\hat{y}}{d\theta}$ described in our equation. Indeed, it is this feature matrix that helps determine the singular value, big or tiny σ_i, in Gradient Descent, thereby determining our next trusted direction to move.

2.2 Discussion Example

Below is an example from our latest discussion. Figure 3 displays a neural network with affine layers, ReLU non-linearity, and a fully-connected layer. This neural network has a single hidden layer, a scalar input and scalar output. The neural network has a width of k. The parameters of this network can be viewed in two different ways:

1) $3k + 1$ total parameters, stemming from kW_1, kb_1, kW_2, kb_2

2) $k + 1$ parameters from the generalized linear model view.

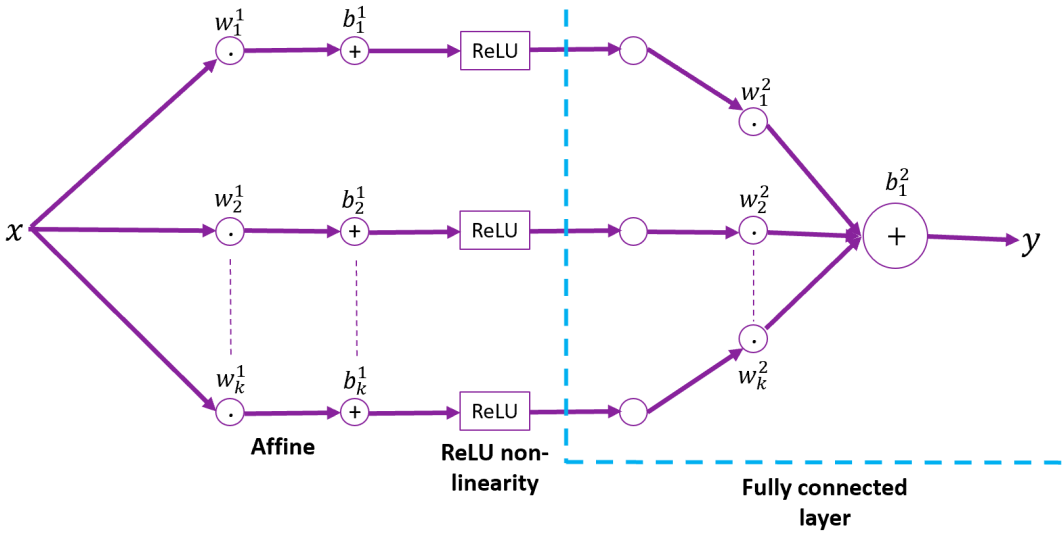


Figure 3: ReLU Neural Network

Consider the following equations. The partial derivative of our network output y in respect to our weights, W, and biases, b, elucidate the dependence of earlier layer features on the features of the later layers. This is interesting, but also introduces the necessity of decoupling these features, a topic of normalization which will be discussed at a later time.

$$\frac{dy}{db_j^2}(x) = 1$$

$$\frac{dy}{dW_j^2}(x) = max(0, W_j^1 x + b_j^1)$$

$$\frac{dy}{db_j^1}(x) = \begin{cases} 0 & ReLU - off \\ W_j^2 & ReLU - on \end{cases}$$

$$\frac{dy}{dW_j^1}(x) = \begin{cases} 0 & ReLU - off \\ W_j^2 x & ReLU - on \end{cases}$$

The above example corresponds to a neural network with a single output. In the case of a network with multiple outputs, there will be features which correspond to each output. Furthermore, in the case of the general linear model each output of the penultimate layer is assigned its own weight. However, in the case of a deep neural network such as for multi-classification, there will be features corresponding to each output with weight sharing.

In the case of a very large linear model with many features, each feature performs a small part of the work of fitting the residual; the amount each corresponding weight moves is quite small. Interestingly, there is a hypothesis in the field to consider which states that for a very large model, a sufficient distance from initialization is never reached to change the features (or derivatives of the features) in a notable way, nor is it truly necessary. This is a perspective that deep neural nets do indeed behave quite like a general linear model.

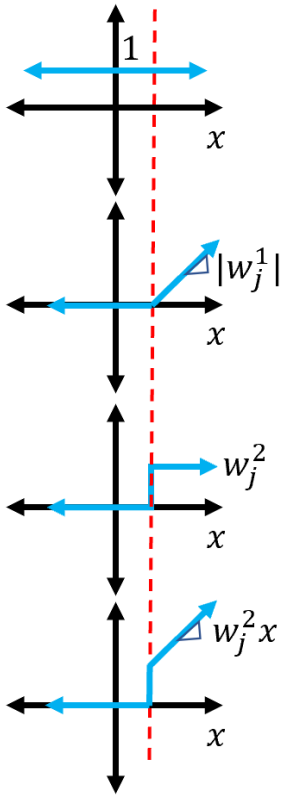


Figure 4: Gradient plots, First: $\frac{dy}{db_j^2}$ Second: $\frac{dy}{dW_j^2}$ Third: $\frac{dy}{db_j^1}$ Fourth: $\frac{dy}{dW_j^1}$

3. Survey

There are a variety of Neural Network architectures and specific applications in which they can be used. Technical application domains and the approaches in which you engage with them will be further explored in detail at a future time. Our multi-dimensional architecture-application grid can be seen below:

Neural Net Architectures (families)					
Area of Use	Multilayer Perceptron	Convolutional NN	Recurrent NN	Graph NN	Transformers
Vision Natural Lang. Processing Time Series Recommendation Engines Scientific Control					

CS 182/282A Lecture 6:

Michelle Tong, Nikhil Potu Surya Prakash

UC Berkeley - Fall 2022

Lecture Topics

1. Deep Learning Survey
2. Deep Learning Problems
3. Optimization Overview

1 Deep Learning Survey

Conceptually these 4 sections: network architecture, problem domains, problem types, and engineering concerns. These 4 topics can be thought of as the dimensions of a 4D grid describing the field of DL.

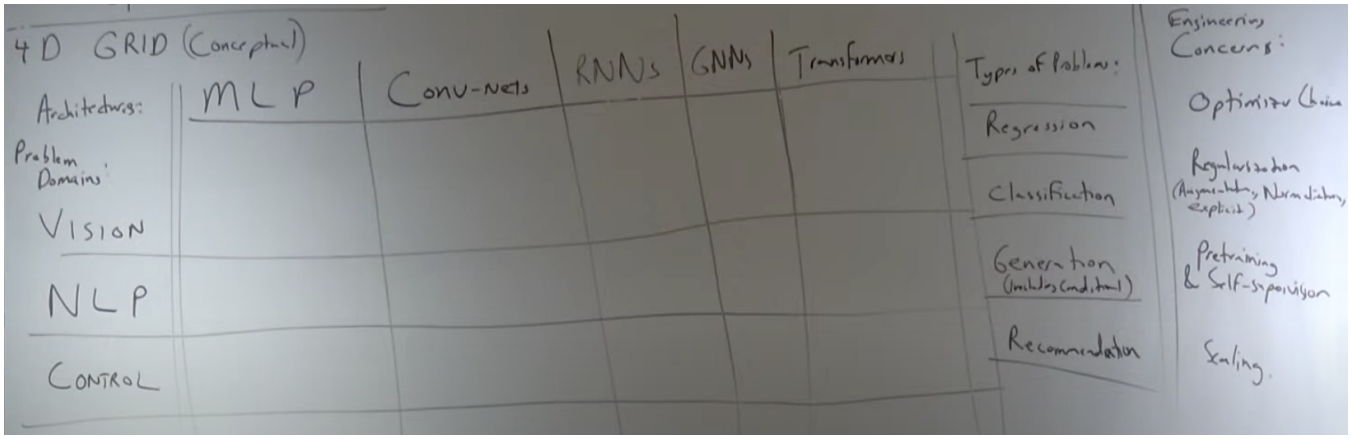


Figure 1:

1.1 Network Architectures

- Multi-Layer Perceptron (MLP)
  - network has fully connected layers
- Convolutional Neural Nets (CNNs)
  - useful for images
  - spatial regularity is embedded in the network architecture
- Recurrent Neural Nets (RNN)
  - the model architecture is different than CNNs but both architectures have a sense of internal state over time from array and weight sharing are over time
- Graph Neural Nets (GNN)

  - nearby items are more related

- Transformers
  - can access input data elsewhere and weight share
- We can tune these networks with various levels of specificity but for the scope of this class we will focus on common underlying problems that may occur.

1.2 Problem Domains

- Vision
- Natural Language Processing (NLP)
- Control
  For the scope of this class, we will explore certain domains to build intuition and experience designing networks and understand trade offs. Additionally research in this field is commonly in one of these domains so literacy is a plus.

1.3 Types of Problems

- Regression - to predict reals numbers
- Classification - to categorize
- Generation - to make/synthesize - generation as opposed to recommendation focuses on new outputs, one such example is the generation of new images of the same scene but in a different style (photo into a painting)
- Recommendation (including conditional generation) - often to commercialize and make money
  Deep learning aims to identify underlying regularities for these problems.

1.4 Engineering concerns

- Optimizer choice
- Regularization (augmentation, normalization, explicit, weight-sharing)
- Pre-training and self-supervision
  - learning models need large amounts of data to be trained well
  - Can we use external data to enhance the ML model? Yes. In theory, ML networks are able to learn regularities that are present elsewhere in large datasets. When new data is presented to the network, the model is able to focus on optimizing the nuances in the external data.
- Scaling
  - larger models (more layers, units, data) tend to work better but it needs to be trained first
  - the network is tweaked to work and also scaled to run on various components and parallel clusters
  - the network is also often scaled down to run on devices for deployment
- Experimentation
  - there are various ways to design experiments such as varying the input data or model architecture (we will cover this more in depth later in class)
- Debugging
  - there are various ways to troubleshoot models (we will cover this more in depth later in class)

2 Deep Learning Problems

2.1 Standard Computer Vision problems

1. Object classification - What is the object? Is the image a cat or dog? This method assumes there is only one object in the image.
2. Object location - Where is the object? Where is the cat? One challenge with localization is determining how many of the object is in the image.

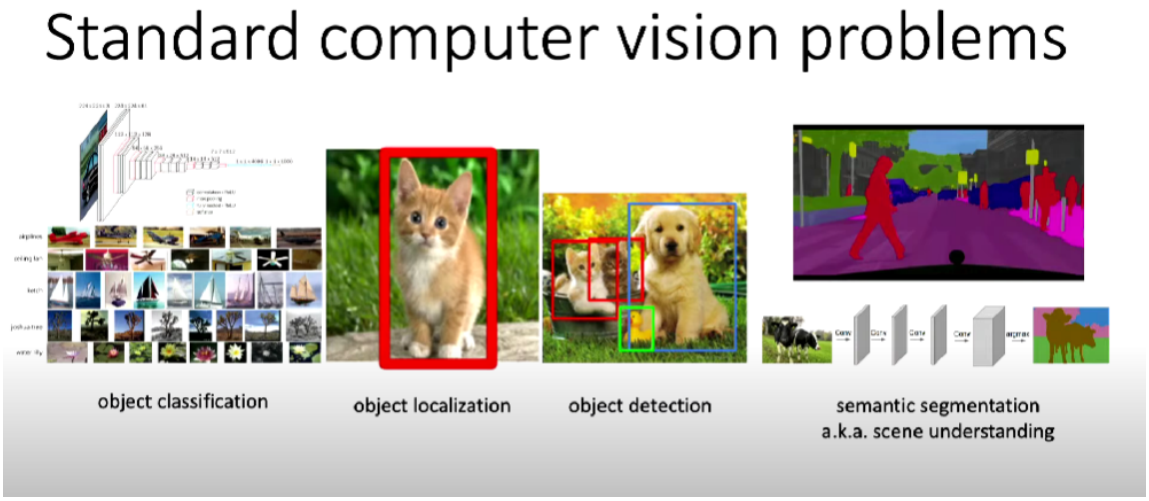


Figure 2:

3. Object detection - What and where are the objects? Where is the dog, cat, and duck in the image?
4. Semantic segmentation - scene understanding (Can we modify architecture to be better than building a classified for each pixel?)
5. Style transfer - ex. change a picture to an impressionist painting
6. GANs (Fig. 3) - making fake realistic images (Can you generate images from a class?)

Figure 3:

7. Unpaired data testing - How does network perform with data that is not paired? Appropriately paired data generally works well.
    - Example: draw the outline of bread and tell the network to make a cat

2.2 Natural Language Processing (NLP) Problems

- OpenAI GPT-2 (Fig. 4)
    - NLP was previously rule based, but now networks can learn patterns of language
    - Example: the network learns words, grammatical types, sentence structure, flow, and pragmatics but does not necessarily learn how to reflect reality

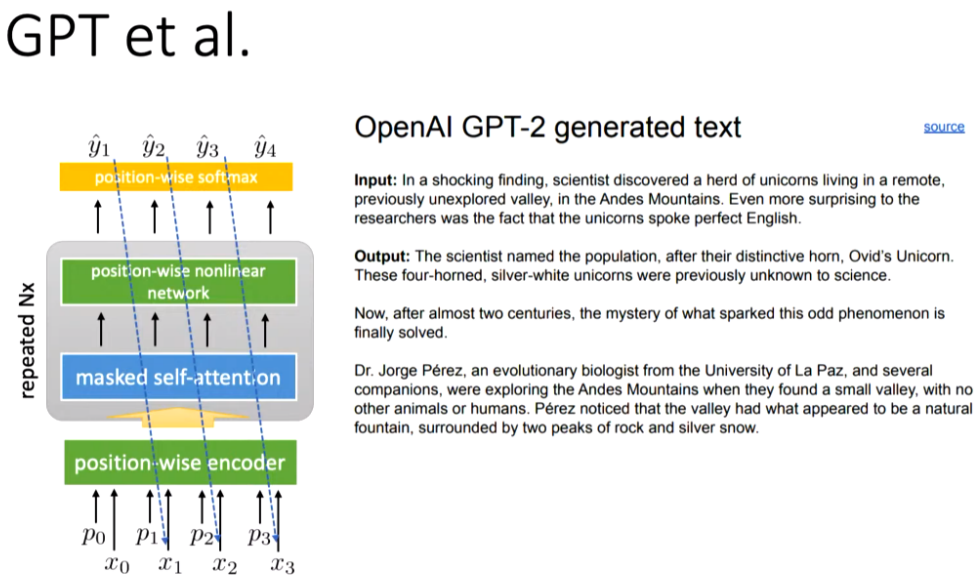


Figure 4:

2.3 Datasets

- CIFAR-10 and CIFAR-100 - dataset of images with 10 or 100 classes, 50,000 training images and 10,000 test images, labels were assigned by humans, image dimensions are 32x32x3
- imageNet - images with 1,000 classes,1.2 million training images, 50,000 evaluation images, labels were assigned by humans

2.4 Networks

- AlexNet (Fig. 5)
    - classic medium depth network
    - widely known to be the first NN to attain state of the art results on ImageNet challenge
- ResNet (Fig. 6)
    - very deep, trainable network
    - does not include a large FC layer at the end, instead just average pools over all positions and has 1 linear layer
    - network development was driven by trying to improve the optimizer

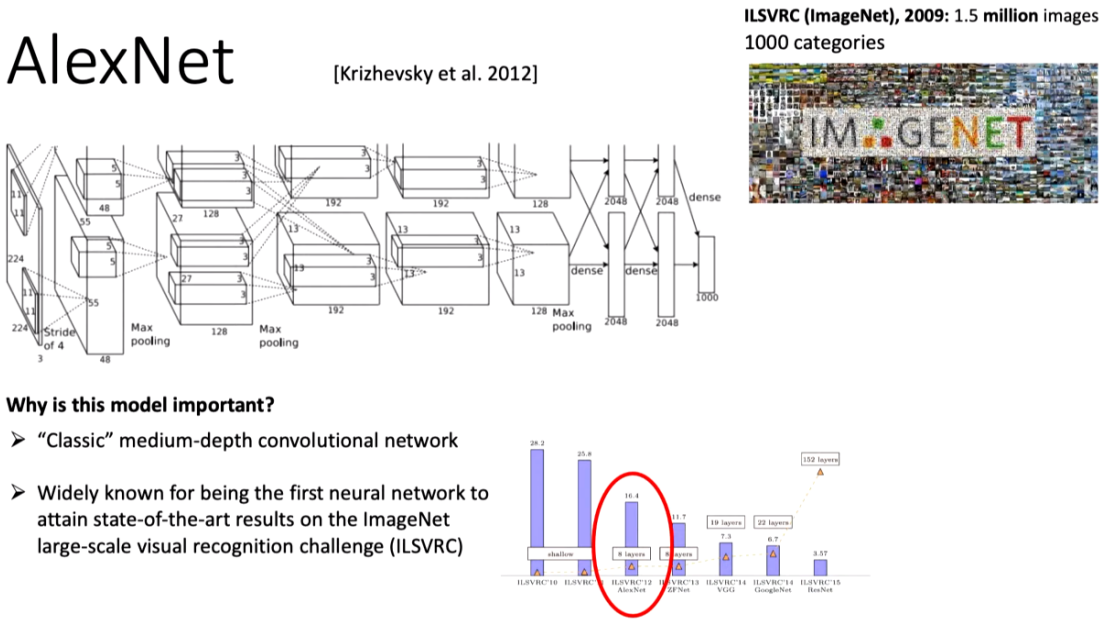


Figure 5:

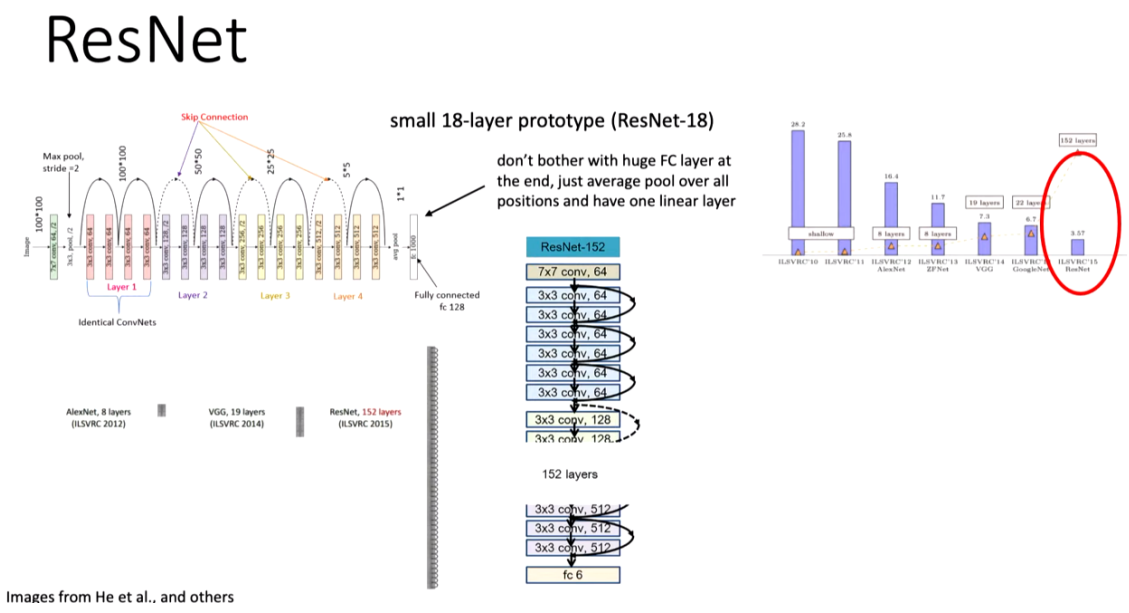


Figure 6:

  - network is leading to super-human performance which means the performance is better than that of humans doing the task, it actually achieved superhuman accuracy on some tasks

- fully convolutional networks
  - low-res (but high-depth) processing in the middle integrates context from the entire image
  - up-sampling at the end turns these low-res feature vectors into high-res per pixel predictions
- U-Net architecture
  - concatenate activations from conv layers to upsampling layers
- RNNs (Fig. 7)

  - the network addresses the question, how can time oriented data, such as time series or sequential data, be digested and used for different problems

- Transformers (Fig. 8)
  - aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease

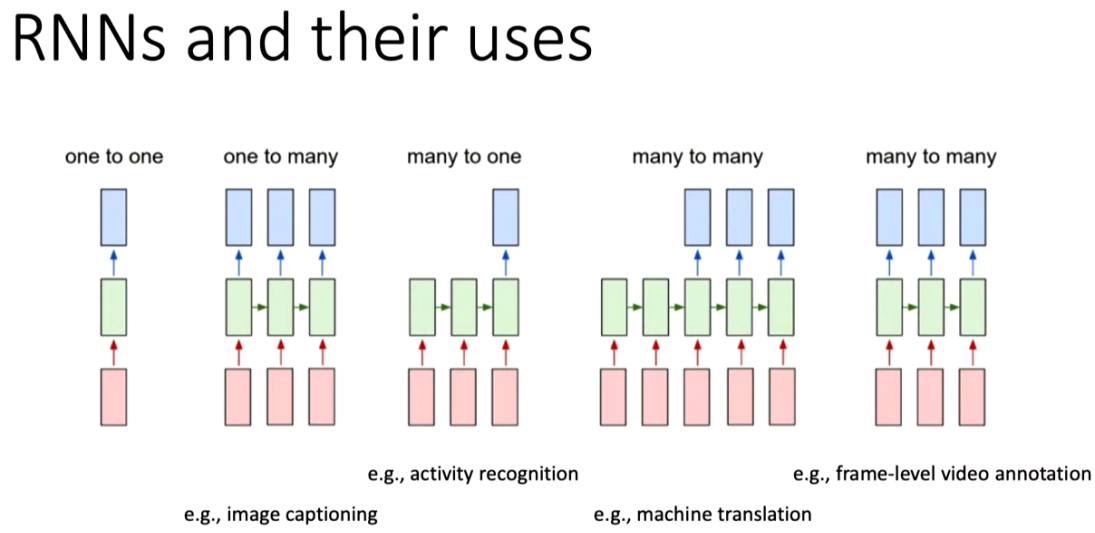


Figure 7:

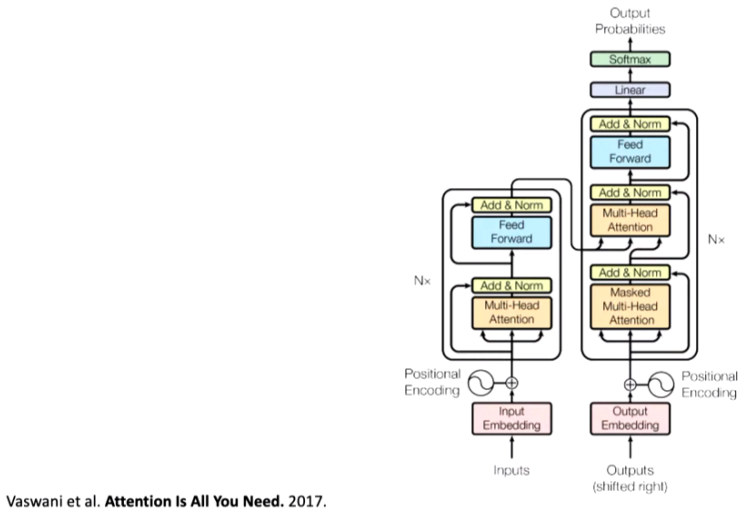


Figure 8:

3 Optimization Overview

In this section, a brief overview of important ideas in numerical optimization algorithms are presented. For a detailed understanding of what each of the methods does, refer to the course material of EE 227C. For ML networks, we need to solve the algorithm using an optimizer. In practice, people use optimizers that were used for similar problems before as a starting point and hyper-parameter tuning.

- Important optimization considerations
  - Learning rate - What rate are we moving down the gradient? How large is our step size?
  - Momentum based methods
  - Adaptive approaches
  - two common optimizer choices - Stochastic Gradient Descent which is mostly influenced by the learning rate hyper-parameter or ADAM optimizer which is an adaptive approach

First we want to understand optimization in terms of GD then we will understand optimization in terms of SGD. Let's begin by breaking down the learning rate in terms of a least-squares perspective.

3.1 Singular value Decomposition (SVD)

Let us recall the singular value decomposition of a matrix X. For real matrices X, its singular value decomposition can be written as

$$X = U\Sigma V^T \text{with } X \in \mathscr{R}^{m\times n}, U \in \mathscr{R}^{m\times m}, V \in \mathscr{R}^{n\times n} \text{ and } \Sigma \in \mathscr{R}^{m\times n}. \tag{1}$$

The matrices U and V are orthonormal matrices and satisfy the properties $U^TU = UU^T = I_{m\times m}$, $V^TV = VV^T = I_{n\times n}$. The matrix Σ is a collection of singular values of X along its diagonal. The singular values of X are the positive square roots of non-zero eigenvalues of XX^T or X^TX. If X has rank 'r' then there would be r singular values of X. Let the singular values of X be denoted by σ_i for $i \in \{1, \ldots, r\}$. The matrix Σ can be written as

$$\Sigma = \left[\begin{array}{c|c} diag(\sigma_1 \ldots \sigma_r) & 0_{r\times(n-r)} \\ \hline 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{array}\right]$$

We can also see that columns of U are the extended eigenvectors of XX^T and similarly the columns of V are the extended eigenvectors of X^TX.

3.2 Least Squares

Optimization algorithms can be understood and analysed easily using simple optimization objective to which closed form minimizers exist. Least Squares problem is one such elegant optimization problem.

The least squares approximate solution of the equation $Xw = y$ can be found using the following optimization problem.

$$w^* = \underset{w}{\text{argmin}} \quad ||Xw - y||_2^2 \tag{2}$$

Utilizing the SVD of X The same optimization problem can be formulated using change of coordinates as follows

$$\begin{aligned} &\min_w \quad ||Xw - y||_2^2 \\ =&\min_w \quad ||U\Sigma V^Tw - y||_2^2 \\ =&\min_w \quad ||U(\Sigma V^Tw - U^Ty)||_2^2 \end{aligned} \tag{3}$$

Notice that the norm of a vector doesn't change when it is just rotated without stretching. The orthonormal matrices U and V have orthonormal columns and hence just rotate the vectors without stretching them. Therefore,

$$\begin{aligned} &\min_w \quad ||U(\Sigma V^Tw - U^Ty)||_2^2 \\ =&\min_w \quad ||\Sigma V^Tw - U^Ty||_2^2 \\ =&\min_{\tilde{w}} \quad ||\Sigma\tilde{w} - \tilde{y}||_2^2 \end{aligned} \tag{4}$$

where $V^T w = \tilde{w}$ and $U^T y = \tilde{y}$ In eq.(4), since Σ has singular values only along its diagonal, the objective can be decoupled into sums of squares of multiple scalar differences as follows

$$\min_{\tilde{w}[1],\tilde{w}[2],\ldots,\tilde{w}[r]} \quad \sum_{k=1}^{r} (\sigma_k \tilde{w}[k] - \tilde{y}[k])^2 \tag{5}$$

3.3 Gradient Descent

The gradient descent update equation for the optimization problem in (2) with a learning rate η can be written as

$$w_{t+1} = w_t - 2\eta X^T (y - X w_t) \tag{6}$$

Similarly, the gradient descent update equation for the equivalent optimization problem in (4) can be written as

$$\begin{aligned} \tilde{w}_{t+1}[i] &= \tilde{w}_t[i] - 2\eta \Sigma^T (\Sigma \tilde{w}_t[i] - \tilde{y}) \\ &= (1 - 2\eta\sigma_i^2)\tilde{w}_t[i] + 2\eta \Sigma^T \tilde{y} \end{aligned} \tag{7}$$

Notice that the update rule is just written for the i^{th} element of $\tilde{w}$. For the stability of the difference equation in (7), we need

$$\begin{aligned} & 1 - 2\eta\sigma_i^2 > -1 \; \forall \; i \\ \implies & \eta < \frac{1}{\sigma_i^2} \; \forall \; i \\ \implies & \eta < \frac{1}{\sigma_{max}^2} \end{aligned} \tag{8}$$

Here σ_{max} and σ_{min} are the largest and smallest singular values of X respectively.

It can be seen from the above choice of η, $1 - 2\eta\sigma_{min}^2$ can be close to 1 and the convergence might take an extremely long time to converge along certain directions with small corresponding singular values. One of the ideas to improve the speed of convergence is to use the concept of 'momentum' inspired from the 'Proportional + Integral (PI) action controller' which is described in the next section.

3.4 Momentum based methods

Idea: Find a way to make the learning rate bigger without causing trouble for the large singular values.
Observation: The weights associated with the large singular values oscillate at high frequency as the learning rate is increased. So, to dampen the oscillations out, a low pass filter can be added. It is known from circuit analysis that a low pass filter outputs an exponential average of the input. This averaging can help us use larger learning rates compared to gradient descent as averaging dampens out the oscillations due to large singular values.
Implementation:

$$\begin{aligned} \tilde{w}_{t+1}[i] &= \tilde{w}_t[i] - \eta a_{t+1}[i] \\ a_{t+1}[i] &= (1 - \beta) a_t[i] + \beta(\text{ "current gradient"}) \end{aligned} \tag{9}$$

For more intuition about The term "current gradient", think about how the gradient was obtained in the previous section on least squares.

Here a_t is the internal state which dictates the averaging behavior (exponential average) and β controls how fast we average i.e., controls the weight given to the past events in the exponential average. This is very much similar to the behavior of an RLC circuit. For momentum based methods, both the weight and internal state average gradient are evolving. Note, there are several ways to mathematically implement this circuit.

3.5 Adaptive approaches:

There's a limit to how much the learning rate can be increased even by the momentum based methods. Momentum still respects SVD and the movement along the directions that are small is still small. The idea in adaptive approaches is to change the learning rates for different singular values (different directions) - More about this in the next lecture.

3.6 Citation

- Sergey Levine's slides (Fig 2-8), https://static.us.edusercontent.com/files/azVHcuABtM3V2ja2DrXVVo91

EECS 182/282 - Designing, Visualizing, and Understanding Deep Neural Networks

Lecturer: Anant Sahai

September 15, 2022

Scribe: Hasitha Sithadara Wijesuriya

1 Momentum and Adaptive Gradient Descent Methods

Vanilla gradient descent has many benefits, but speed is not one of them. The reason behind that is the constraint on the learning rate. This effect can be illustrated by solving the simple scalar equation given in equation 1 which considers the squared loss as the loss function ($L(w)$). We want to minimize $L(w)$ over w until we get as close as possible to the ground truth (y).

$$\begin{aligned} &\sigma w = \hat{y} \\ &\min_{w} L(w) = (y - \sigma w)^2 \end{aligned} \tag{1}$$

Gradient descent step at k^{th} step is given in 2 with a learning rate (η).

$$\begin{aligned} w_{k+1} &= w_k - \eta \nabla_w L(w^k) \\ w_{k+1} &= w_k + 2\eta\sigma(y - \sigma w_k) \\ w_{k+1} &= (1 - 2\eta\sigma^2)w_k + 2\eta\sigma y \\ \left(w_{k+1} - \frac{y}{\sigma}\right) &= (1 - 2\eta\sigma^2)\left(w_k - \frac{y}{\sigma}\right) \end{aligned} \tag{2}$$

Then the w^{k+1} can be written in terms of the initial guess of w_0 by considering the pattern in equation 2, as below (3)

$$w_{k+1} = (1 - 2\eta\sigma^2)^{k+1}\left(w_0 - \frac{y}{\sigma}\right) + \frac{y}{\sigma} \tag{3}$$

In order to make sure the (3) is recurrence stable, $-1 < (1 - 2\eta\sigma^2) < 1$ this condition should be satisfied. From that, we get a constraint $\eta < \frac{1}{\sigma^2}$. Figure 1 shows the behavior of gradient descent update with the choice of η. As the η is small enough, the solution converges to the optimal solution but takes a lot of iterations. But when η passes some value, it shows an oscillatory behavior, and after increasing it further, it starts to diverge from the solution.

A slight modification called momentum for the gradient descent is applied to solve these problems.

1.1 Momentum

The idea of momentum is finding a safe way to make the η bigger. We know that the dimensions with larger singular values start to oscillate quickly. So the thought is to somehow low pass filter (LPF) those directions that would otherwise oscillate if we take smaller steps. By doing that, instead of moving in the direction of the gradient, the update is moved towards the direction of the average gradient. The functioning of the simplest LPF is where this idea of averaging originally came from, Figure 2.

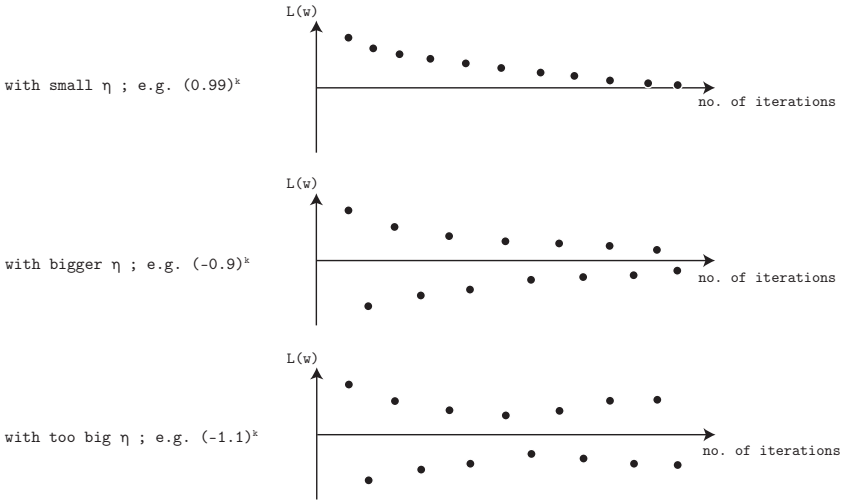


Figure 1: Gradient descent update with the choice of η

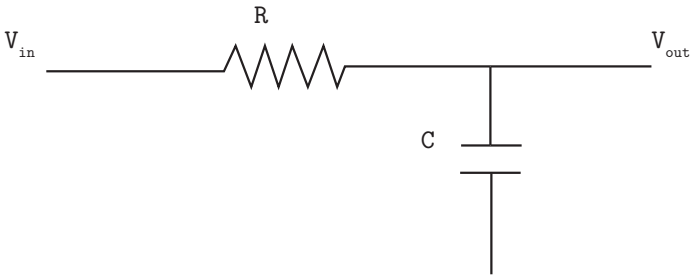


Figure 2: Simplest Low Pass Filter (LPF)

We can solve the first order differential equation that governs the LPF on 2 by equation 4 with a dummy variable τ. The exponential weighted average part is taken as $h(t-\tau)$, which is the impulse response that defines the filter. Since $h(t-\tau)$ is an exponentially weighted average of the V_{in}, the dying exponential has lesser weight as the τ gets further into the past.

$$\begin{aligned} V_{out}(t) &= \int_{-\infty}^{t} V_{in}(\tau)\frac{e^{-\frac{1}{RC}(t-\tau)}}{RC}d\tau \\ V_{out}(t) &= \int_{-\infty}^{t} V_{in}(\tau)h(t-\tau)d\tau \end{aligned} \tag{4}$$

This is also known as convolution integral as what it does is sliding along the input and taking averages by re-centering the input to t. And it integrates to 1 as it expresses a normalized average. In the discrete-time point of view, we can write 4 as a sum, equation 5. The geometrically dying exponential ($h(t-\tau)$ can be written with normalization parameter β as shown in 5. $\beta(1-\beta)^{t-\tau}$ makes sure that sums to 1.

$$V_{out}(t) = \sum_{-\infty}^{t} V_{in}(\tau)\beta(1-\beta)^{t-\tau} \tag{5}$$

This discrete time solution is also the solution to a first-order difference equation. Instead of input V_{in} we can plug in the gradients of $L(w)$ at current step, equation 6. β controls the averaging, as closer it gets to zero, the more averaging we get through the recurrence relationship of past gradients given by a_k.

$$a_{k+1} = (1-\beta)a_k + \beta\nabla_w L(w^k) \tag{6}$$

This momentum term a_{k+1} is used to update the $(k+1)^{st}$ gradient update rather than directly using the gradient as in vanilla gradient descent, equation 7.

$$w_{k+1} = w_k - \eta a_{k+1} \tag{7}$$

In this case, we have two hyper-parameters that can be adjusted, β & η. With the appropriate choice of β, we can increase the η than the limit imposed in vanilla gradient descent. The slower directions converge faster by allowing that increment, increasing the overall speedup.

The current gradient used in 6 can have two interpretations depending on which variant we want to use. The two variants are "Vanilla momentum" and "Nesterov Momentum", equation 8.

$$\begin{aligned} &\text{"Vanilla" Momentum } \nabla L(w_t) \\ &\text{"Nesterov" Momentum } \nabla L(w_t - \eta(1-\beta)a_t) \end{aligned} \tag{8}$$

The vanilla momentum uses the gradient where the current weights are. In the Nesterov momentum, we take advantage of the fact where we are going by peeking into the future, as given from the first part of the momentum term (6). This is because we already know where we are going to end up, and by taking that new information to the calculation of the gradient at this step, we can take a bit of an advantage on learning.

1.2 Adaptive (e.g. Adam) Methods

The simple perspective of the origin of the adaptive methods comes from the fact that even with momentum, the largest singular value and the smallest singular value are the ones that govern the process. Values in between are not relevant for setting up the parameters. So the simple idea is that instead of having a single step size, use different step sizes (η_i) along different dimensions of the parameter vector. In vanilla gradient descent, we go down in the steepest gradient direction. But in the adaptive methods, we no longer go in the steepest direction but still in a downward direction.

The formulation of Adam's method where different step sizes in a different direction are shown in equation 9. In the large gradient directions, it takes smaller steps and vice versa. That way, it takes evenly sized steps along different directions. To track the size of the gradient, a vector $\vec{V}$ is introduced, which depends on the element-wise product of the gradient vector. Then the gradient update for the i^{th} parameter is computed with the square root of elements of $\vec{V}$ to make the units right. A constant (ϵ) is added to the denominator to make sure it is stable.

$$\begin{aligned} \vec{a}_{k+1} &= (1-\beta)\vec{a}_k + \beta \nabla_w L(w^k) \\ \vec{V}_{k+1} &= (1-\beta')\vec{V}_k + \beta' \nabla_w \begin{pmatrix} \cdots \\ \left(\frac{\partial L(w)}{\partial w_i}\right)^2 \\ \cdots \end{pmatrix} \\ w_{k+1}[i] &= w_k[i] - \eta \frac{a_{k+1}[i]}{\sqrt{V_{k+i}[i]} + \epsilon} \end{aligned} \tag{9}$$

Typically $\beta' << \beta$, to make sure the average size of the gradient is over a longer period than the average gradient. Otherwise, it would create oscillatory movement when it is closer to the convergence. This can be observed by the numerator and the denominator in the weight update step. Also, this creates another challenge when the β' is very small (which is typically the case). In the start, this creates problems as the $\vec{V}$ is going to be small for a while until it builds up. This is partially protected by the (ϵ). The solution for this problem is normalizing, as shown in equation 10. This normalization dies away as the iterations keep going but solve the problem at the start.

$$\hat{\vec{a}}_{k+1} = \frac{\vec{a}_{k+1}}{1-\beta^k}$$
$$\hat{\vec{V}}_{k+1} = \frac{\vec{V}_{k+1}}{1-(\beta')^k} \tag{10}$$
$$w_{k+1}[i] = w_k[i] - \eta \frac{\hat{\vec{a}}_{k+1}[i]}{\sqrt{\hat{\vec{V}}_{k+1}[i]} + \epsilon}$$

2 Convolution neural networks

Convolutional Neural Networks (CNN) is a class of neural networks that is most commonly used to analyze images. Why do we need CNN's in the first place? Figure 3 shows the naive approach to analyzing images. As the input is high dimensional and it itself is very big. For an image with 128×128 resolution with three color channels, it has around 50,000 pixels. If we approach this problem with a fully connected network, we get roughly around 3 million weights for the first layer. And it is very large for just one layer. That is why people adopt the idea of convolution for analyzing images with fewer weights.

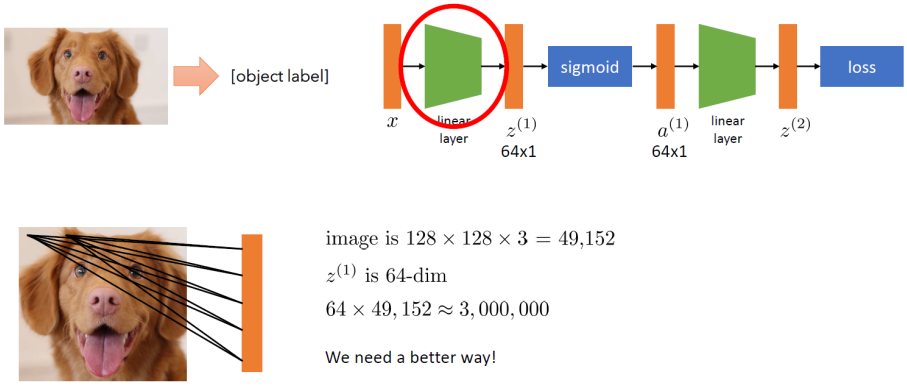


Figure 3: Naive approach with images [Lev21]

As the goal of any neural network, the objective is to learn the pattern in our data. And our objective is to build an architecture toward the kind of patterns we expect to see. There are several ways that CNN's achieves this objective, as shown below.

- Respecting "locality"

In the learned functions, the pixels that are near each other are important to figure out the relationship of the learned function. This idea is manifest as the convolution structure itself. Figure 4 shows the local features in small neighborhoods inside of an image. If we learn a very narrow field of view of an image, we sometimes see edges as the edges are the smallest local signature that we can observe. when we make the network deeper, it sees the larger and larger parts of the image, and we can maybe start observing parts of the objects.

- Respecting "invariances"

This idea can be simply explained as the learned function should not be affected by the translation of objects within the dataset. For example, in the case of a classifier, it should give the same prediction even if the object has moved within the input image. This idea is manifested by weight sharing and data augmentation during the training process. Figure 5 shows the idea of weight sharing. If we have different blocks with different weights for every patch of the image, that doesn't capture the idea of invariance. If we use the same patch to cover the whole image by shifting the patch by a pixel at a time, it is much more tractable to lift it to the 64 channels. And it is a much more reasonable number of parameters.

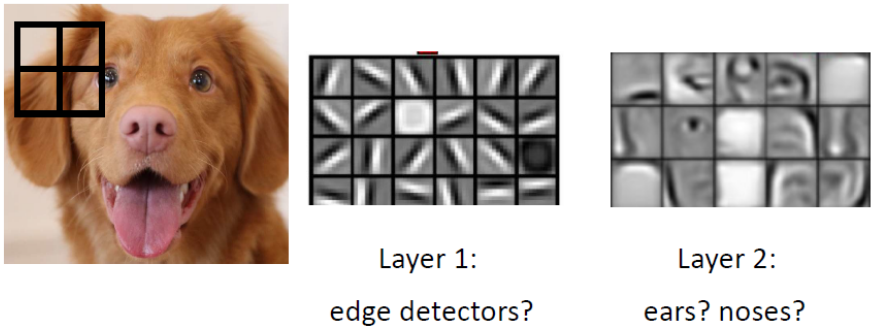


Figure 4: Locality in CNN's[Lev21]

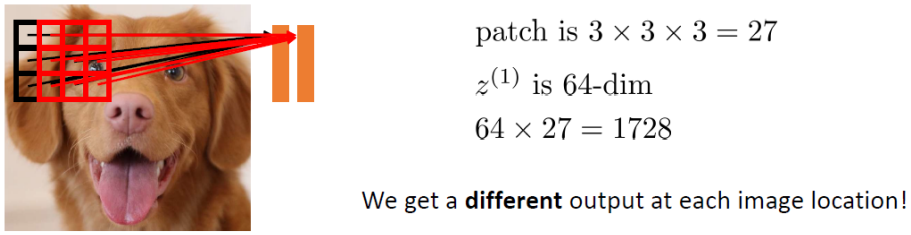


Figure 5: Idea of weight sharing[Lev21]

- Support hierarchical structure and multi-resolution understanding

This idea is explained as the patterns that we are trying to learn are visible not at the level of local level but only when we see the whole image. For example, the edges can be identified at the local level, but the parts of the objects we are considering are only visible at another level, and the object themselves is only visible at the global level. This manifests as the combination of depth of the network, downsampling with depth (stride and pooling), and lifting from pixel space to abstract level by increasing the number of channels with depth.

- "Room to play" & "redundancy"

This idea is during the learning process, if things are too tightly constrained, we are usually stuck in local minima, and it is hard to move on from there. As the learning process is going, we should be able to have room to build new features to get it solved. In other words, we should have redundancy to work on different features as we progress with our learning. This manifests as the combination of adding more channels and a particular way of learning called dropouts.

3 What we wish this lecture also had to make things clearer?

- It would have been great if we could see the real-time demonstration of the effect of momentum on gradient descent as shown on this website. https://distill.pub/2017/momentum/
- Also, using the tools in https://alexlenail.me/NN-SVG/LeNet.html, we could have seen a clear demonstration of CNNs that covers the weight sharing, number of channels per layer, etc.

References

[Lev21] Sergey Levine. Lecture notes in designing, visualizing and understanding deep neural networks. https://cs182sp21.github.io/static/slides/lec-6.pdf, January 2021.

Lecture 8: 09/20 (Tuesday)

Lecturer: Prof. Anant Sahai

Scribes: Daisy Zhang

1 Agenda

Topics covered last time were Key ideas & Convnet manifestation. Today, we continue with Convnets, focusing on input standardization and activation normalization. The lecture included the following topics:

- Respect locality
- Respect "invariance" within data for problem domain
- Support hierarchical structure (Fine $\rightarrow$ Coarse)
- "Room to play"

which are related to the following realization techniques:

- Convolutional Structure
- Weight-sharing
- Data Augmentations
- Down Sampling
- Lifting to more channels as we get coarse
- Diversity to support learning
- Dropout
- More layers

2 Locality

We have the observation that many useful image features are local. To tell if a particular patch of an image contains a feature, It is enough to look at the local patch. We get a different output at each image location using the same filter. We assume locality exists in CNN, where inputs that are more close to each other are more correlated. In the context of image inputs, this assumption is qualitatively valid because of local patches of similar color, texture, or lighting. Figure 8.1 shows an example of locality.

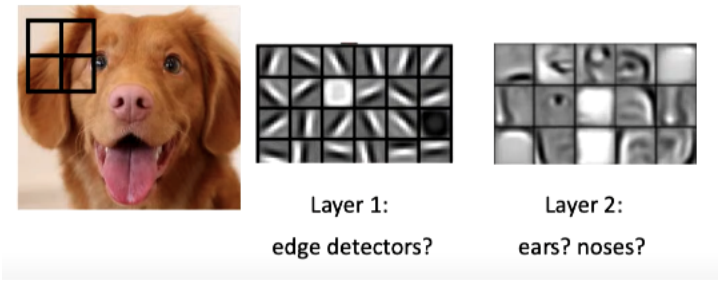


Figure 8.1: This picture of a dog shows that the nature of locality and hierarchy in vision inputs. Locality is exemplified by that most of the local patch in the leftmost picture has similar texture and lighting. Hierarchy is exemplified by that edge features in Layer1, the middle picture, can zoom out and form larger features such as ears and noses in Layer2, the rightmost picture. Image from Lecture8 slides.

3 Weight sharing

Weight sharing in CNN is that a convolutional kernel or filter scans across the whole image input and produces a feature map, so every neighborhood of pixels in the image is processed by the same kernel or filter. That is, the weights in the kernel or filter are shared. We don't assign weight to each individual pixel.

Example: Suppose we have a 5x5 gray-scale picture and a 3x3 kernel (Figure 8.2). We apply the kernel to one pixel of the input image and its surrounding local patch with the same size as the kernel. We get a number from the previous calculation, add the bias, apply non-linearity, and we finally get the new output of one depth of that one pixel of the image. When the image has multiple channels and each channel is convolved with a different kernel, this operation is called *depthwise convolution*.

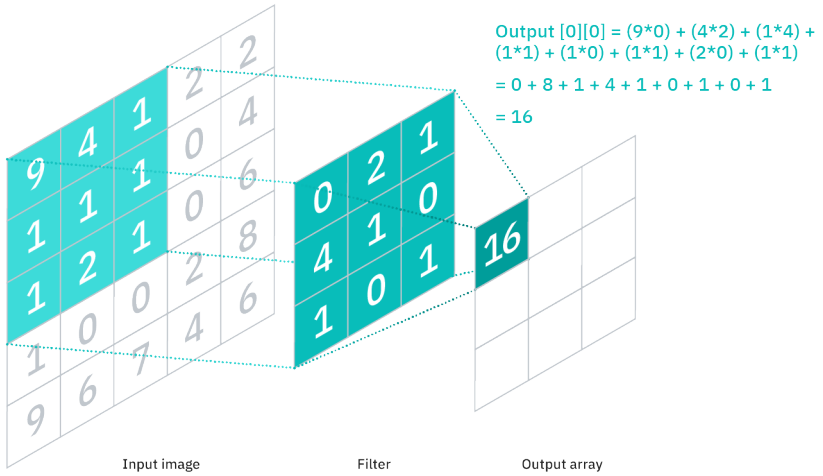


Figure 8.2: Example of a 3x3 convolutional kernel applied to a 5x5 input. The padding size is 0. Image from [2]

The weight-sharing structure provides translational equivalence. The resulting activation map remains the same under the translation of input feature map. Weight sharing also significantly reduces the number of weights of the network, which ensures higher computational speed.

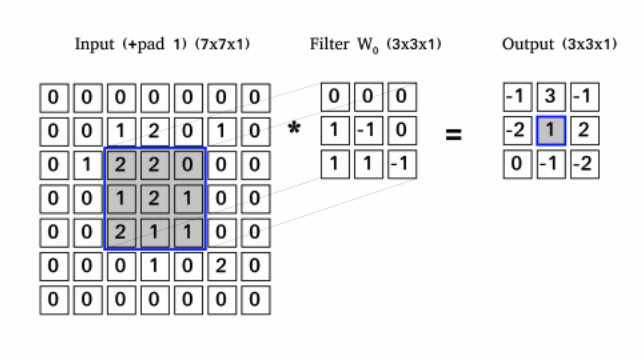


Figure 8.3: A 5x5 input with padding size 1 is applied with a 3x3 kernel, then generates a 3x3 output. Image from [1]

Sometimes, the *padding* operation is introduced. Padding is the addition of crafted pixels on the sides of the image so that the pixels on the borders are not lost from the output of convolution. Padding size is usually one less than kernel size. If the crafted pixels are all zeros, this operation is called *zero-padding*. If the crafted pixels are mirrored values from the original input, this operation is called *mirror-padding*. Deep learning application typically uses zero-padding for practice.

Figure 8.3 shows an example of convolution operation with padding.

Note: *depth* of one input/output layer in the CNN is a term for the number of channels of the layer. For example, the depth of an RGB image is 3.

4 Support hierarchy

Receptive Field is a term borrowed from biological vision. A particular pixel of the output generated by the neural network has a dependence on some, but not all pixels in the input. So this term defines what pixels in the original image this output depends on. Figure 8.4 is an example of receptive field.

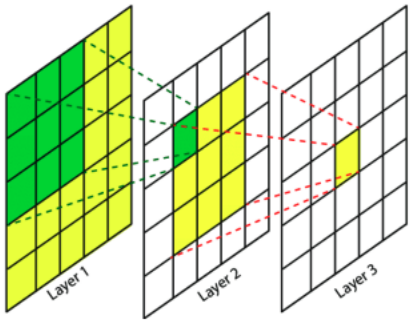


Figure 8.4: The yellow pixel in layer 3 has its receptive fields colored in yellow in layer 2 and layer 3. The green pixel in layer 2 has its receptive field in layer 1 colored in green

If we keep adding convolutions layers on the images, the receptive field will grow linearly. However, linear growth is slower than desired. Faster growth is desired because the neural network can easily learn the full hierarchy.

Therefore, we need a way to do dimensionality reduction on feature maps, i.e., *down sampling*.

For example, if we want to transform a 2x2x4 region of the output into a 1x1x4, we need to represent 2x2 elements with 1 pixel. This is also known as *pooling*.

There are several ways for pooling, including

- average
- max
- pick-one, i.e., pick the pixel at a certain location of the computed local patch, and discard the rest
- weighted average

Pick-one pooling seems a waste of computation because it discards most of the computed pixels. This method of pooling is implemented by *stride* instead, which only computes the pixel at the desired location to save time and computation.

The weighted average can be viewed as a convolutional layer and a stride.

Pooing layers make the receptive field grow exponentially. In addition to dimensionality reduction, pooling also provides local translational invariance, allowing the CNN to be more robust to features varying in their locations.

5 Structure summary

1. Convolutional layers
   (a) A way to avoid needing millions of parameters with images
   (b) Each layer is "local"
   (c) Each layer produces an "image" with roughly the same width and height, and number of channels = number of filters
2. Pooling: moving from fine to coarse but more abstract
   (a) If we ever want to get down to a single output, we must reduce resolution as we go
   (b) Max pooling: downsample the "image" at each layer, taking the max in each region. Max is differentiable. It stops gradient descent to the smaller value which it discards, so it ensures the gradient descent to the important feature.
   (c) This makes it robust to small translation changes.
3. Finishing it up
   (a) At the end, we get something small enough that we can "flatten" it (turn it into a vector), and feed it into a standard fully connected layer.
   (b) Suppose the input size is W, the kernel size K, the stride S, and the padding P. Input and kernel are both squared.
   Then the output size is
   $$\frac{W - K + 2P}{S} + 1$$

6 Data Augmentation

Regularity is needed for the neural network, which is the invariance to some insignificant changes to our knowledge. For example, it is expected that a pattern can still be recognized when it shifts horizontally by 2inches. However, these insignificant changes are not present in the training set. Data augmentation can apply these changes to the input images and feed the newly generated data to the neural network to train for robustness.

In practice, in order to save space, the newly generated images are not stored along with the original huge dataset. These images with data augmentation applied are generated on the fly. Modern deep learning applications typically never see the same image twice, because all data have been augmented during the training. During data augmentation, the label along with the input image normally does not change.

Data Augmentation represents some of the easiest ways that people can transfer the domain knowledge into the training process. We humans understand what are insignificant features by our evolved vision system. This knowledge can be encoded into the practice of data augmentation and drop it into the training. This practice can help the neural network know what is important based on our knowledge.

Basic augmentations include autocontrast, rotation, translation, posterization, etc. Some basic augmentations are exemplified in Figure 8.5. More aggressive augmentations include cutout, Mixup, CutMix, and PixMix [3].

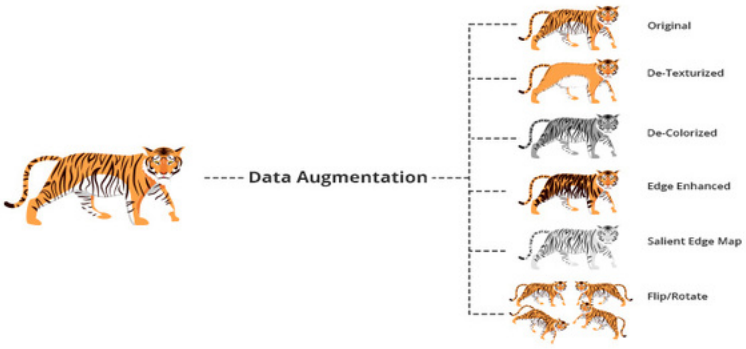


Figure 8.5: Examples of basic augmentations [4]

Figure 8.6: Examples of more aggressive augmentations [3]

7 Standardization and Normalization

Standardization is advised in almost all learning practices. Why do we need standardization? The reason behind it needs to be known. To understand it, it's necessary to know the answer to this question: what tempts a neural network to learn?

Suppose we want to learn $x * w$ and w is the parameter to learn. Then $\frac{d}{dw}xw = x$. Thus, larger $\|x\|$ moves w more in gradient descent. We want to move w more only when we are confident that it's in the right direction. The largeness of $\|x\|$ needs to relate to the essence of data. For example, if x is a weight measurement, then its pure value is larger when its unit is mg than when its unit is kg. However, in this case, we don't want to update more on w, the parameter to learn, because the largeness of data doesn't reflect that it is importance.

Accordingly, when the input size (magnitude) does not carry confident and important information, such as weight data in kg and in mg, we need to convert it to data with zero mean and unit variance.

The design choice with zero mean and unit variance is good because $0 + 0 = 0$ and $1 * 1 = 1$. We want to make sure that the neural network is learning with the greatest sensitivity when the dataset is essentially changing. For example, the elbow point of a ReLU network can capture where the predicted values vary. It will be hard for a ReLU network to learn if the elbow points are not aligned with the output action space. Standardization is a way to align inputs to where the function can learn more easily.

The expressive power of the network is not lost with standardization and normalization. The expressive power is embedded in the bias and the weight terms, which can shift the mean and variance.

We perform standardization on the training set by subtracting the mean and scaling the variance computed from the training set. We can use the same mean and variance from the training set to normalize the validation set.

References

[1] E. S. Agency. Machine learning group: About machine learning: Convolutional neural networks introduction.

[2] I. C. Education. Ibm cloud learn hub: What are convolutional neural networks?, 2020.

[3] D. Hendrycks, A. Zou, M. Mazeika, L. Tang, B. Li, D. Song, and J. Steinhardt. Pixmix: Dreamlike pictures comprehensively improve safety measures. *CVPR*, 2022.

[4] A. Oberoi. What is data augmentation in deep learning?, 2022.

CS 282 Deep Neural Networks **Fall 2022**

Lecture 9: ConvNets/CV

Lecturer: Anant Sahai *Scribe: Buyu Zhang, Michael Lam*

9.1 Input standardization, Normalization

A common maxim in machine learning is that input data should "always" be normalized, meaning the input data should somehow follow a distribution with zero mean and unit variance, i.e. a transformation of the form $x \to \frac{x - \text{“}E[X]\text{”}}{\text{“}\sigma[X]\text{”}}$ where subtraction by the expected value "E[X]" zeroes the mean, and division by the standard deviation "$\sigma[X]$" unitizes the spread. But this normalization raises several questions. Why are we even normalizing in the first place? Where do "E[X]" and, by extension, "$\sigma[X]$" come from?

9.1.1 Reasons for Normalization

Aside: Confidence in training points
Before we delve into ways to normalize our data, let's look at a toy example that demonstrates the motivation for normalization. Consider the following standard least squares problem:

$$X\overrightarrow{w} \approx \overrightarrow{y} \tag{9.1}$$

Suppose we know measurement $y[2]$ to a greater degree of confidence, i.e.

$$y[2] = y_{true}[2] + N(0, 0.01), \tag{9.2}$$

while other data points are described by

$$y[i] = y_{true}[i] + N(0, 1), i \neq 2 \tag{9.3}$$

How can we modify our ordinary least-squares algorithm to take advantage of this information? Intuitively, we'd like to somehow weight $y[2]$ more heavily since it carries more "information", in a sense, because we're more confident that it is close to the true value.

To do this, notice that the variance of y[2] is 0.01 times that of all the rest of the data points. If we multiply the second data point and its corresponding entries in X by 10, the variance of y[2] will now equal one (keep in mind that variance scales with the square of the data point's scaling factor). Not only that, but our second training point will now be weighted 10 times more than it was before.

$$10 \times y[2] = 10 \times y_{true}[2] + 10 \times N(0, 0.01) = 10 \times y_{true}[2] + N(0, 1) \tag{9.4}$$

Solution: Multiply second row of X by 10, second row of y by 10, do standard least squares.

Things to consider: What if we instead duplicate the second data point and its corresponding entries in X 10 times? How many times must we duplicate the point for the effect to be the same as scaling by 10? Is this effective when the feature space is large? (In general the effect of duplicating a datapoint will be dependent on the total number of datapoints. Duplicating a datapoint is a better choice for a non-linear model.)

We demonstrated in the above aside that it was possible to preprocess the data such that a specific data point which we know with high confidence can be weighted proportionally to its spread. Note that, in doing so, the data point now has a mean of zero and unit variance. *But that's exactly what normalization is!*

In the toy example above, normalization had the effect of correctly weighting points based on their importance (in this case, confidence). Normalizing our training data before inputting them into neural networks has a similar effect of scaling the parameters such that the raw *magnitudes* of the values don't necessarily impact the gradient calculation during the gradient descent backpropagating step (if we had massive, un-normalized values, the gradient would always be large). Moreover, the data points are generally centered at zero because the *ReLU* layers are often initialized at that point, making the neural net most sensitive and expressive when input values are around zero. Thus, normalization can additionally decrease the neural network's training time.

Food for thought: We often normalize our data beforehand because we don't expect the *magnitude* of the values to have any value in the classification process, but this isn't always true. Can you think of an example where normalization would preclude accurate classification?

9.1.2 Choices for "$E[X]$"

We are not generally provided with prior knowledge of the probability distribution from which our sample X data are drawn, so these values must be derived empirically, i.e. from the data set X itself during training time. Let us focus firstly on ways such an empirical average $E[X]$ can be derived.

Consider the following scenario where we have n 100 x 100 images in our data set, each with 3 channels: red, green and blue, with values between 0 and 255, inclusive. The following are ways we can compute an empirical expectation for any given point in any image:

1. **For any given position (x, y) in channel i, take the expected value corresponding to that point to be the average of the channel i values at (x, y) across all images**, i.e. the expected value for position (50, 50) in the red channel is the average of the red values at (50, 50) across all images. This can be done across our entire data set, with or without the addition of augmented images.
2. **Use the numeric midpoint of the range as the expected value for all positions** (e.g. choose 128 for range 0-255). This method is simple, but it assumes that the color values across all images are distributed evenly around the middle of the range. Consider what might happen if our data set consisted only of dark images (images with channel values between, say, 0 and 12). Would our normalization still be centered at 0?
3. **Take the expected value for position (x, y) at channel i to be the average of all the channel i values for that image.** For example, for any given image, the expected value for position (50, 50) of the red channel would be the average of the red channel value across all positions of that specific image.
4. **Set the expected value to be an average of the values within a local patch of pixel positions in the same channel within the same image**. For the expected value for position (50, 50) in the red channel, we may want to take the average of a 3×3 patch centered at that coordinate, i.e. the rectangle $[49, 51] \times [49, 51]$.

Food for thought: What are some of the advantages/disadvantages of some of the expected value definitions listed above? Especially for the definitions that average within a single image, what information may be lost in the normalization process?

Note: You may be wondering whether normalization may change the input in a way that reduces the information it contains and derails the optimization process. The good news is that our normalization, defined as $\frac{x - \text{“}E[X]\text{”}}{\text{“}\sigma[X]\text{”}}$, is composed of subtraction and division functions that can be undone by our biases and weights, respectively. Keep in mind that there are important exceptions given how our convolutional networks are structured. For example, due to the moving nature of the filter, there is no way for an expected value defined as the average of a particular position across all images (option 1 listed above) to be reversed. However, an average taken across all positions in all images would be reversible. Can you come up with any other examples?

These are just a few of the more common ways to compute an empirical expected value. Note that the empirical variance is calculated analogously, except, instead of averaging, we take the variance across a certain subset of points. There are myriad other definitions depending on the specific application, some of which are combinations/variations of the ones listed above. For example, for a much coarser average, we could define $E[X]$ to be the average of all values within the image across all channels (a variation of option 1). One could also imagine averaging across all channel positions across all images as an extension of this definition. When choosing what type of normalization to use for a specific problem, it is considered good practice to use whatever method has given good results for a similar problem in the past.

Each expectation calculation scheme can be seen as different ways to span 3 axes: the image positions, the image channels, and the different images within the data set (Figures 9.1-9.4). For example, we can visualize expectation (3), the average of the values at all positions in one channel for one image, as a prism that spans all positions, 1 channel, and 1 image (Figure 9.1). A few other examples are listed in Figures 9.1-9.4.

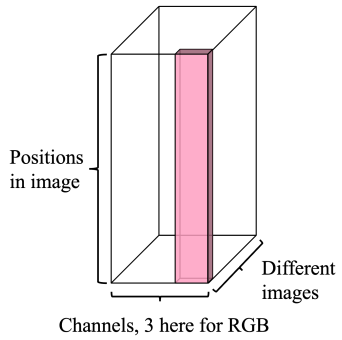


Figure 9.1: Demonstration for option 3.

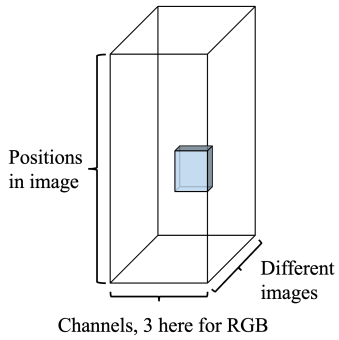


Figure 9.2: Demonstration for option 4.

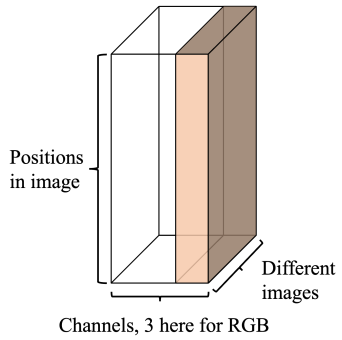


Figure 9.3: Variation of option 3.
(Averaging all images)

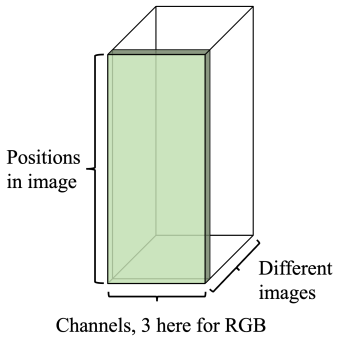


Figure 9.4: Variation of option 3.
(Averaging all channels)

9.1.3 Normalization methods

Some of the more common normalization methods have special names:

- **Batch normalization**
  Batch normalization normalizes the contributions to a layer for every mini-batch.
  At training time, the gradients are not calculated for all data at one time; instead, a batch of data is used. A batch normalization layer uses a mini-batch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the mini-batch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features. The batch normalization will become unstable when the batch size is too small.

- **Layer normalization**
  Layer normalization normalizes input across all channels in one image.

- **Instance normalization**
  Instance normalization normalizes across each channel in each training images. The problem instance normalization tries to address is that the network should be agnostic to the contrast of the original image.

- **Group normalization**
  Group Normalization normalizes over a group of channels for each training images.
  Group normalization is a medium between Instance normalization and layer normalization. When we put all the channels into a single group, group normalization becomes layer normalization. When we consider each individual channel a single group, it becomes instance normalization.

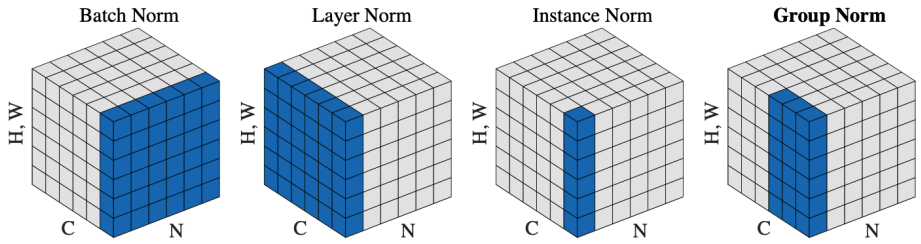


Figure 9.5: **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.[1]

9.1.4 Weight Standardization

What we talked before is input standardization. Another way to control the movement of gradient descent is weight standardization. Instead of applying the weights directly during the gradient calculation, the weights are normalized beforehand, preventing weights from getting too big.

9.1.5 Deep Net Structure for Convolutional Neural Nets

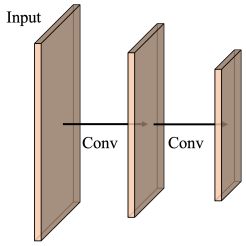


Figure 9.6: Deep net structure for Conv-nets

We've established that normalization in the input layer centers the data where the ReLU elbows are most active, but do we need to normalize each layer in the convolutional net (Figure 9.5)? After all, the output of each layer is the input to the next layer, and the repeated applications of the weighted convolutional layers can very quickly lead to an output that is, again, extremely small (leading to a problem known as the *vanishing* gradient) or very large (the *exploding* gradient).

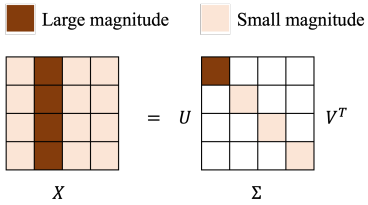
The solution is to not just normalize the initial training data, but to also normalize the outputs of intermediate convolutional layers. Intuitively, normalizing after every layer in the network should solve the problem, which was exactly what early researchers placed did. But since gradients don't tend to get extremely large or small over the course of the application of at least a few weighted convolutional layers, adding normalization at the very start and after every few layers is sufficient and is what is typically used in modern networks.

Aside: Large singular values
Consider the singular value decomposition of an arbitrary output matrix X:

$$X = U\Sigma V^T \tag{9.5}$$

The diagonal entries of Σ are the singular values of the matrix X, which is incidentally related the X's Frobenius norm, a measure of matrix's "mass", or the magnitude of its entries. There are two ways these singular values can get very large:

1. **A few columns of the X have extremely large magnitude.** The corresponding singular values for these columns would then be correspondingly large. This is the case where a few data points are much larger than all the others.



2. **All the columns are close to being collinear.** If many of the data points fall on the same line, the singular value corresponding to that direction will be large. This is the case when most data points follow a certain trend.

Normalization helps to ensure that large singular values come not from the former case, but from the latter. Gradient descent is thus more robust against large, outlying data points and more sensitive to strong trends.

9.2 Residual network

Another way to combat the effects of the vanishing/exploding gradients is the introduction of *skip connections*, bridges that allow the output in one layer to be the input to not just the layer immediately subsequent, but also to layers further down the net. In a network without skip connections, gradient updates must pass through all subsequent layers before reaching the weights of the current layer, which may lead to a gradient update that is minuscule. Skip connections make the weight layers more sensitive to gradient updates as the gradients have to backpropagate through much fewer layers of the network during the gradient calculation and are therefore much less likely to get extremely small from repeated applications of weight layers. Figure 9.7 contrasts the structure of the plain net with that of a residual net.

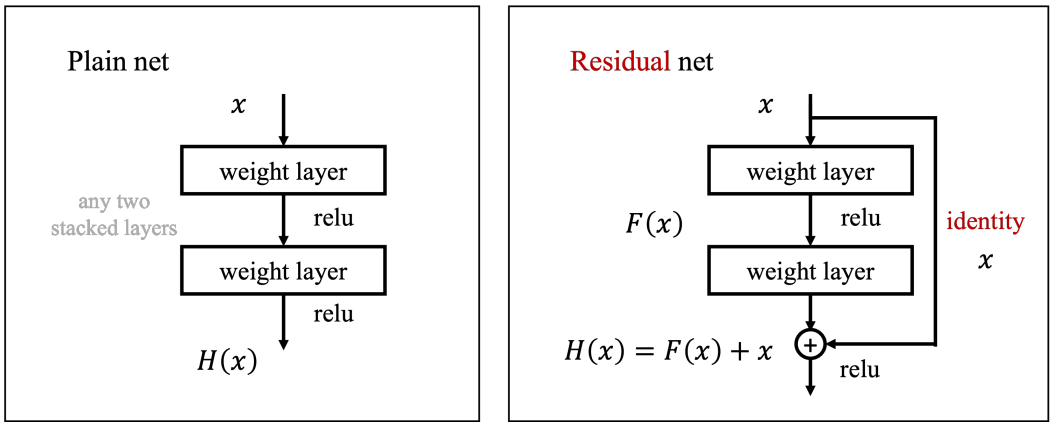


Figure 9.7: Plain net (left) and residual net (right).[2]

References

[1] Yuxing Wu, Kaiming He, Group Normalization, *Proceedings of the European conference on computer vision (ECCV)* (2018).

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren,Jian Sun, Deep Residual Learning for Image Recognition, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).

CS282A Lecture 10 Notes (09/27)

Matan Grinberg and Shruti Satrawada

September 2022

1 Residual Nets and their advantages

From last lecture, we recall the ResNet architecture.

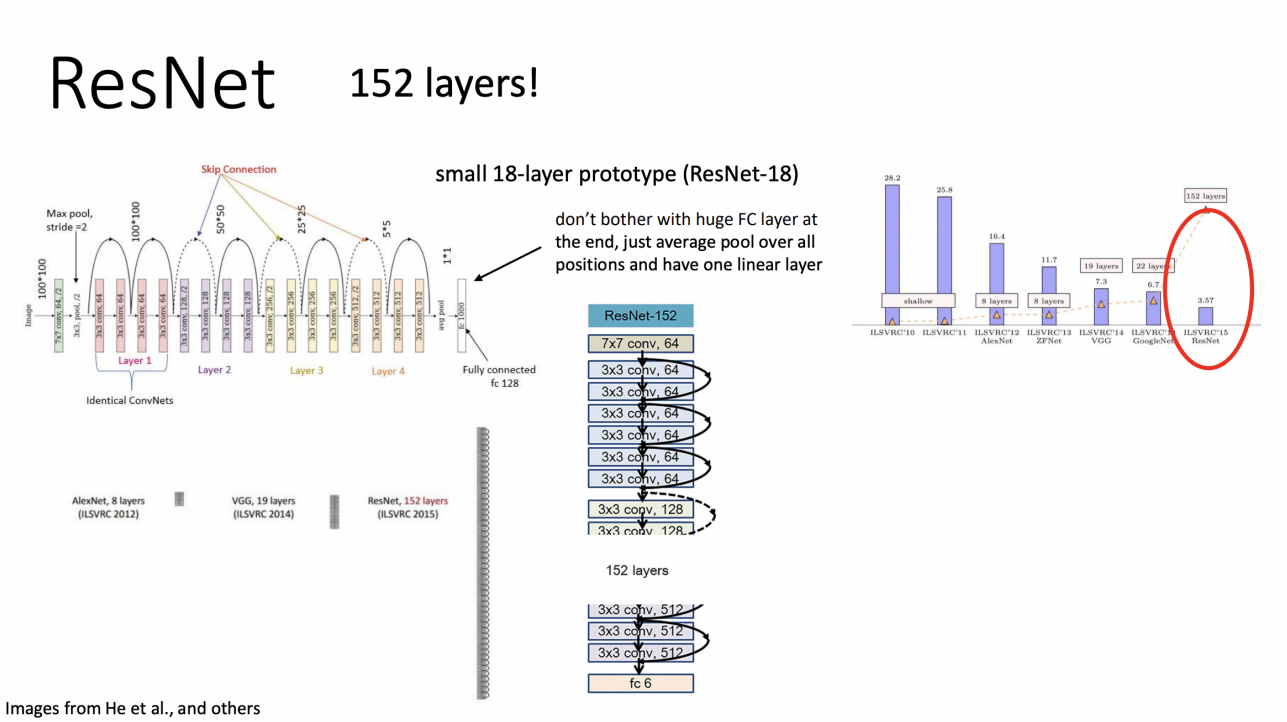


Figure 1: Overview of Residual Net architecture.

The modern convolutional neural network era was ushered in by the idea that each successive layer *modifies* the data in the pipeline, as opposed to completely transforming it.

We can also see how depth affects plain nets versus residual nets in Figure 2. An issue people found (that motivated this new residual architecture) was that in plain nets, introducing new layers created more error instead of creating more expressivity.

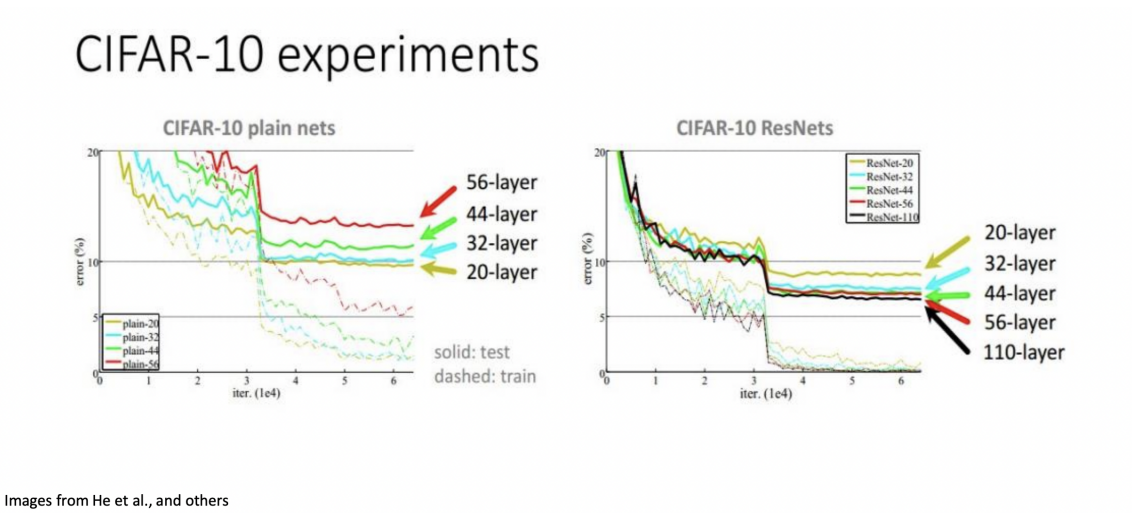


Figure 2: Effect of depth on error in CIFAR-10 experiments for plain nets and ResNets.

With this new residual net architecture (cf. Figure 3), each successive layer no longer completely transforms the data. Instead, through the use of the *skip-connection*, an identity operation is applied and added to the actual output of the layer. The effect of this is allowing the network to less dramatically modify the data in the pipeline, which in turn allows the depth of deep networks to more consistent learn salient features of the data.

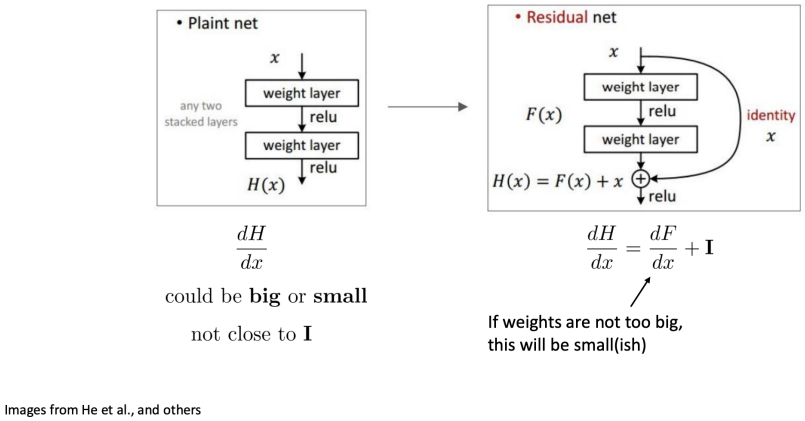


Figure 3: Difference between plain and residual architecture.

The derivative for a given layer of a residual net takes a form similar to that of Neural ODEs:

$$\frac{d}{ds}x(s) = \tilde{F}_s(x(s))\,, \tag{1}$$

where s is a fictitious "time" that signifies x going through the net and $\tilde{F}_s$ is learnable.

Now, as the number of these layers increases, do we eventually reach convergence? In other words, with enough layers, do things stop changing? We cannot in fact make general statements about the steady-state behavior of such deep networks. This is mainly due to the fact that we also cannot make any such guarantees about the behavior of ODEs of the form shown above. However, we can say if the residual blocks are small and get smaller in successive layers, then we would be able to make some guarantees about convergence.

2 ConvNeXt

2.1 Introducing ConvNeXt

A new architecture emerged that borrow from the success of transformers (without the actual transformer part). It differs in architecture from the residual net as shown in Figure 4.

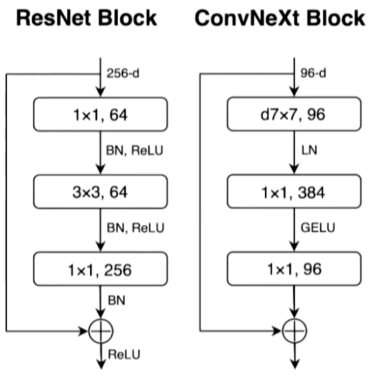


Figure 4: Residual net versus ConvNeXt. (Image from He et al.)

It is important to note that in ConvNeXt, instead of using the traditional batch normalization and ReLU activation, layer normalization and *Gaussian ReLU* (GeLU) are used. Furthermore, the 7×7 convolution is done within each channel, as opposed to across the channels (this is signified by the "d" in front of d7×7). Furthermore, ResNet has convolution operations taking place in the middle of the pipeline, whereas ConvNeXt has convolution in the beginning. This ultimately allows ConvNeXt to save a factor of 64 in parameters space, while also using and only one activation layer.

2.2 GeLU: Gaussian ReLU

We saw a new form of ReLU in the ConvNeXt structure called GeLU which stands for the Gaussian Error Linear Unit and is calculated by

$$x \cdot \Phi(x) \tag{2}$$

where $\Phi(x)$ represents the Gaussian CDF of x.

Theoretical Inspiration for GeLU: For ReLU we take the max(0, x), but why 0? Why not pick something else to act as our "gate" to decide whether the value x is passing through? Lets randomly draw a Gaussian, N. If x is greater than N, we pass the value through, and if not we return 0. GeLU is the expected value of this.

As can be seen in Figure 5, the GeLU curve is smoother than the ReLU curve, is non-convex and has a non-monotonic gradient.

This new formulation is discontinuous like ReLU.

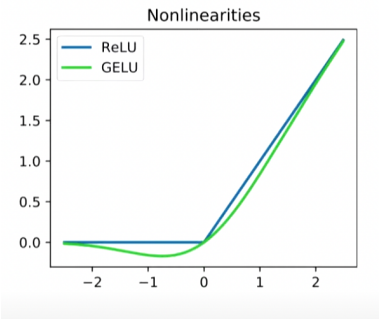


Figure 5: Graph comparing RELU versus GELU. Original Source: Wikipedia by Ringdongdang

2.3 Depthwise Convolution

In Depthwise Convolution, filters and inputs are broken into different channels, convoluted separately, and then concatenated.

In the ConvNeXt structure, Depthwise Convolution is used instead of typical convolution as it is less computationally demanding. In general nowadays, Depthwise Convolution is more commonly used than typical convolution. The difference between typical convolution and depthwise convolution is visualized in Figure 6

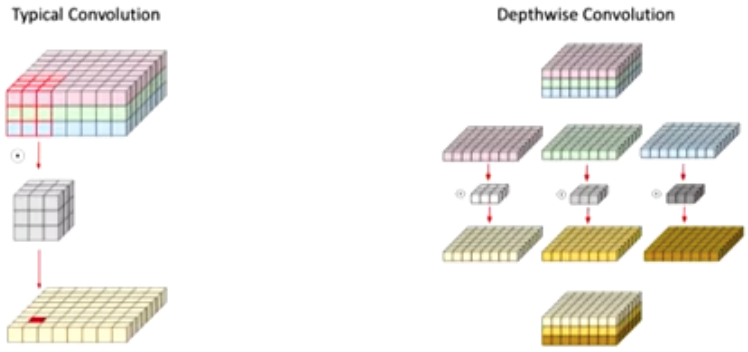


Figure 6: Regular Convolution versus Depthwise Convolution. Source: Lecture Slides.

2.4 Wrapping up ConvNeXt

Since ResNets in 2015, newer models like the 2022 ConvNeXt have improved performance as seen in Figure 7.

These newer models take bits and pieces from the old models. For example residual connections from ResNets are still common. However, there have been changes made as well, with depthwise (grouped) convolution being more common now than typical convolution.

ConvNeXt uses a cosine learning rate schedule, AdamW (Adam with weight decay), label smoothing, a type of dropout called Stochastic Depth Regularization, and aggressive data augmentation, all helping lead to higher accuracy.

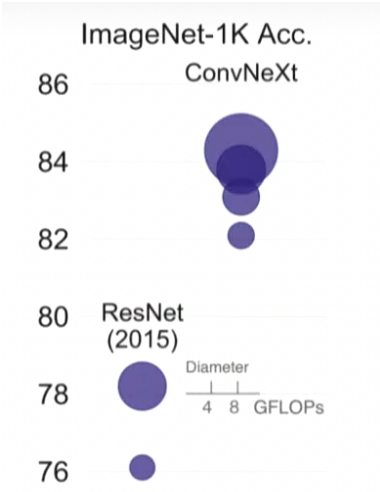


Figure 7: Performance of ResNet versus ConvNext on ImageNet-1K Acc. (Image from Liu et al, 2022)

3 Dropout

3.1 Dropout(basic)

Inspiration How can we simulate having an ensemble of diverse neural networks that we output the average of, like in Random Forest, in a method that's less costly? By utilizing dropout, there is essentially a slightly different neural network at each training step which provides an "ensemble-like" behavior.

Implementation

- During Training: While training on our minibatch, randomly "kill" certain units by setting them to zero before training. We will have a hyperparameter that provides the probability of "nulling" out a unit. When we "kill" a unit, all the attached weights do not get updated and we can think of the remaining weights "shaking" a bit to pick up the slack.
- Evaluation: The standard way evaluation is done after dropout is to use expected behavior.
  - Example: Lets say we're killing certain units with probability $\frac{1}{2}$. This means that our unit, x, is outputting 0 half the time and $ReLU(x)$ the other half of the time. We can calculate the expected behavior of our unit as $\frac{1}{2} \cdot ReLU(x) + \frac{1}{2} \cdot 0$. This essentially halves the output of our neuron during evaluation to account for us "nulling out" half the neurons while training.

Results of Dropout Dropout promotes diversity and redundancy in networks since at each training step we are essentially training a different neural network. It ensures that it isn't one unit's job to learn everything since that one unit can't always be relied on. It has a regularizing effect and is usually only used for MLPs (multi-layer perceptrons) on 1x1 blocks.

Does Dropout make training slower? No. Learning rate and dropout are trained together, so while dropout does reduce the size of our gradients, the learning rate can compensate for this.

Can Dropout hurt the performance? All regularization techniques have the ability to hurt performance since they shape the inductive bias. Depending on what we are modeling dropout can affect the performance differently. For a brief period of time, people thought that Normalization might negate the need for Dropout, but in practice having both performs better.

Mathematical Reasoning Behind Dropout In another lecture we discussed that we can think of our training algorithm seeing some version of the big singular values that are defined in the inductive bias. Generally for a matrix there are two ways you can have a large singular value. Either you have many different things pointing in the same direction, or you have one particular row or column that has a big value. Dropout drives us towards the direction of having lots of little things pointing in the right direction.

3.2 Stochastic Depth Regularization

Implementation During training randomly drop entire residual blocks. So during training some will be active and being trained while others are "gone". Very similar to Dropout but on a different scale.

3.3 Other Methods Similar to Dropout that Have Been Explored

- **Drop Connect**: Instead of "killing" certain units why not try "killing" weights? People have found that in practice this does not work as well.
- Another method that also adds the regularizing effect we see in Dropout is to multiply by random noise (in between 0 and 1) instead of multiplying by 0. Dropout tends to perform better in practice.
- What if we have logic behind what we turn off instead of randomly choosing? People have explored "killing" units more or less often based on what the size of their activation's tend to be as well as other similar ideas, but these have not been proven to be useful in practice so far.

4 Label Smoothing

4.1 Before Label Smoothing: One Hot Encoding

For Classic Cross-Entropy/Log-Loss Classification when we have a label "class 1", we represent this using one hot encoding. Since one hot encoding has an array of 0s with a 1 for the correct label as a goal instead of probabilities, it forces the model to try and be "super" confident. This means that the model will constantly be trying to get closer and closer and since with softmax we cannot actually reach a probability of 1 unless the input is infinity, the model will never be to reach the goal classification.

4.2 Label Smoothing

Label smoothing is an approach that fixes this by using probabilities in the goal array for our classification so our model can actually reach its goal.

We set our goal array y as follows:

$$y = [\frac{1 - \alpha \cdot k - 1}{k}, \frac{\alpha}{k}, \frac{\alpha}{k}, \frac{\alpha}{k}, ...]^T$$

where k is the number of classes and α is a hyperparameter

Reaching this goal y array is possible and in practice label smoothing does improve performance. We can think of the behavior as now being more similar to squared-error since they can both be satisfied.

Concern: Originally, people were concerned about this idea since in the real world some items are more similar than others. For example, a dog should be more similar to a cat than to the ocean. With that in mind, we can see that label smoothing tries to say that something is a dog but has an equal small probability of being either a cat or ocean when logically we would expect the probability for the cat to be higher.

CS 282 Deep Neural Networks **Fall 2022**

Lecture 11: GNN

Lecturer: Anant Sahai *Scribe: Jiashu Liang, Anirudh Rengarajan*

11.1 Key ideas of the Convolutional Neural Networks (CNNs)

Images have a structure with patterns that we want to learn and we need to create inductive biases to help us learn the structure. The key ideas including:

1. Convolutions with weight-sharing AND having an "image" at each layer: build the local convolutions and then use hierarchical depth to see the entire image
2. Residual Connections to fight dying gradients: every layer has an effect when parameters change on what happens at the end
3. Normalization to "adaptive speed bump" exploding gradients: brings down growing activation values
4. Pooling to downsample: allowing distant information to more quickly get used
   - Max Pooling $\left[\text{MaxPool}(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}) = \max(a,\, b,\, c,\, d,\, e,\, f,\, g,\, h,\, i) \right]$: "routes" gradients to specific parts of images
5. Data Augmentation, Dropout (including stochastic depth regularization), Label Smoothing

Food for thought: What is the conceptual similarity and difference between Residual Connections and Pooling?

Residual Connections and Pooling can both allow the gradients to depend on more things. You can also realize the effect of Residual Connections by Pooling (i.e., sum pooling all the layers before each layer). However, this will result in quadratic intrinsic growth of the gradients.

11.2 Graph Neural Networks (GNNs) as the "generalization" of CNNs

11.2.1 Basic GNN model

Instead of a 2D grid for an image, we have a graph with information in nodes. First, we can have a Simple Assumption as follows.

Simplifying Assumption: We can define a Single Graph topology. A graph can be defined by a tuple (V, E) where V is the set of all nodes and E are the set of all edges that connects the nodes in V. For GNN, we attempt to do a Graph-level classification task based on information in nodes and connections between said nodes.

An example of a GNN in a Single Graph topology can be shown in Figure 11.1. The nodes are connected with the black lines in the same layer. Then each node in the next layer is connected to its neighbors in the previous layer (dotted red lines) and itself in the previous layer (dotted black lines).

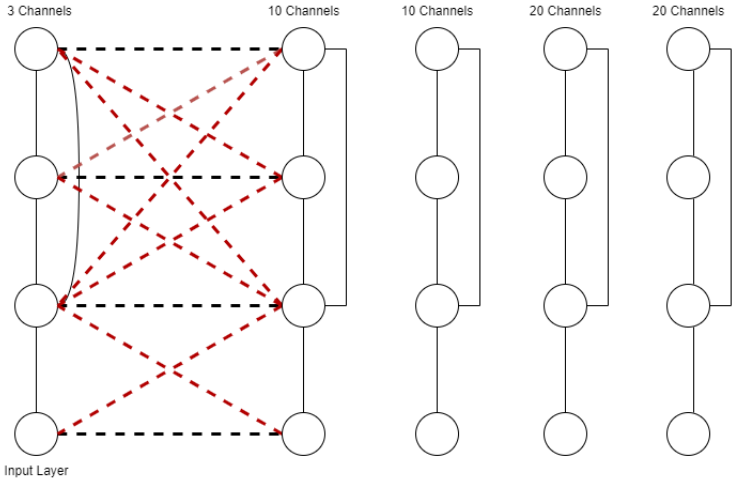
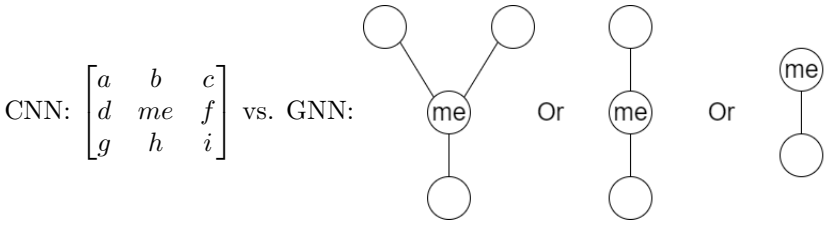


Figure 11.1: Example GNN showing connections inside a layer and between layers.

11.2.2 Differences between CNNs and GNNs from a Neighbor Perspective

What is different about "me" in these instances:



- We might have different numbers of neighbors to "me". In contrast, one pixel of an image usually has eight neighbor pixels in CNNs.
- We don't have separate names for neighbors of "me". This means that the neighbors do not have a particular order in GNNs. In contrast, the eight neighbor pixels of one pixel have their particular position in CNNs, like the pixel "b" is on the top of "me" and the pixel "f" is on the right.

11.3 Extension of key ideas of CNNs to GNNs

11.3.1 Weight sharing

The most important idea behind CNN is the convolution with weight-sharing. What kind of function can we have in place of convolution, respecting the properties of a graph?

1. First, this function should have some learnable parameters associated with it (like w_1 in Eqn 11.1a).
2. Second, it should take two arguments, the node itself and its neighbor nodes, because these are all the information we know related to this node.
3. Then, a method is needed to combine the information of these neighbor nodes regardless of the ordering or cardinality of neighbors. This method is not learnable and we need to choose from possible choices, including Sum (used in Eqn 11.1a), Maximum, Minimum, Product, Softmax, Variance, and so on.
4. Finally, neighbor nodes also have their learnable function, $g_{w_2}(\text{neighbor node})$.

These requirements lead us to Equation 11.1a, where "me" represents the node itself and "them" represents one particular neighbor node.

$$f_{w_1}[\text{me}, \sum_{\text{neighbors}} g_{w_2}(\text{them})] \tag{11.1a}$$

$$f_{w_1}[\text{me}, \sum_{\text{neighbors}} g_{w_2}(\text{me}, \text{them})] \tag{11.1b}$$

$$f_{w_1}[\text{me}, \sum_{\text{neighbors}} s_{w_2}(\text{me}, \text{them}) g_{w_3}(\text{them})] \tag{11.1c}$$

More generally, g_{w_2} can depend on "me" and "them" together to explain their connections, as shown in Equation 11.1b.

We can further factorize $g_{w_2}(\text{me}, \text{them})$ into $s_{w_2}(\text{me}, \text{them})$ and $g_{w_3}(\text{them})$ and get Equation 11.1c, where $g_{w_3}(\text{them})$ is regarded as the learnable function of "them" and $s_{w_2}(\text{me}, \text{them})$ is regarded as the connection (or similarity) of "me" and "them". In Transformer Architecture, $s_{w_2}(\text{me}, \text{them})$ is also known as "attention" because it is a learned amount of how much we pay attention to this neighbor.

We can also understand $\sum_{\text{neighbors}} s_{w_2}(\text{me}, \text{them}) g_{w_3}(\text{them})$ as a weighted average of $g_{w_3}(\text{them})$ with $s_{w_2}(\text{me}, \text{them})$ as the weights. We can even use these weights to do softmax.

Food for thought: If we view the GNN in an adjacency matrix formulation, is there a way to use our usual CNN weight operations more directly?

Answer: There is a way to leverage adjacency matrix formulation with use-cases in graph signal processing for signal reflection.

Researchers have generalized the idea of signal processing from one- or two-dimensional functions to general graph relationships. A convolution can be seen as an operation that respects the natural shift invariance on the infinite topology. The infinite topology means that the signal can keep going to the right or left. For a graph, there is no information about what the natural shift would be but there is indeed something we can do with the adjacency matrix. We can take a walk on the graph, or we can take products of the adjacency edge and itself. We can think of shifts on a graph as a kind of repeated product on the unit line. As an example, we can generalize the infinite line to the finite line by shifting on the circle because we can rotate on the circle.

There is also a beautiful connection between convolutions and the invocation of a different domain called frequency domain convolutions. In signal processing, convolution in the original domain corresponds to the multiplication in the frequency domain. But it turns out that the frequency domain is just the eigenbasis corresponding to a particular matrix associated with the structure because convolutions commute with each other and can share the eigenbasis. This allows people to consider what actions can commute with such a matrix formulation of a graph network, with one such formulation being the adjacency matrix. So you could imagine what operations (as matrices) can commute with the adjacency matrix. Those will be the counterpart of convolutions and will respect the entire graph in this abstract way. Graph signal processing considers those objects relevant in graph neural nets. This entire approach is sometimes encapsulated by the spectral methods.

However, there are also drawbacks to this representation. (*https://distill.pub/2021/gnn-intro/*) The number of nodes in a graph can sometimes be on the order of millions, and the number of edges per node can be highly variable. Often, this leads to very sparse adjacency matrices, which are space-inefficient. Another problem is that many adjacency matrices can encode the same connectivity, and there is no guarantee that these different matrices would produce the same result in a deep CNN (that is to say, they are not permutation invariant).

11.3.2 Pooling

Pooling (downsampling) groups similar nodes and coarsens the image in CNNs. Here, the similarity usually means how close the pixels are. In GNN, we just need a similar clustering method to "coarsen" the graph. Under the assumption that the graph topology is fixed, we can pre-compute a specific clustering of this graph based on its topological attribute. In this way, the four nodes in Figure 11.1 are shrunk to two nodes in Figure 11.2.

The clustering can also be learnable. We can use some similarity measures to cluster, like similar neighborhoods or similar values inside them. We can even use the learned similarity to simulate the effect of the clustering without doing a full clustering.

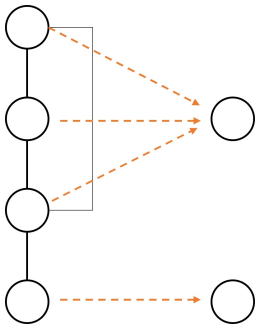


Figure 11.2: Example of pre-computed pooling in GNN

However, the use of pooling in GNN is not as useful as in CNN. Researchers have done experiments where they replaced well-defined pooling with random downsampling and found the network also performed very well. The reason is that the network can learn the appropriate information in other places, like $s_{w_2}(\text{me}, \text{them})$ in Equation 11.1c. The actual activation function, f_{w_1}, can determine certain patterns of "me" and "them" for the next layer, meaning it will learn to cluster information together in certain places if necessary. GNN has flexibility in the architecture itself.

11.3.3 What doesn't change?

In fact, almost everything else remains the same. For example, the residual connection only requires the structure of the object to be the same across layers, which GNN satisfies as well. GNN can also employ all normalization used in CNN, like layer norm, batch norm, instance norm, group norm, and so on. All the normalization needs are that the outputs have some sense of "likeness" to average over.

11.4 What if the graph topology is not fixed?

Let's generalize our assumption: the graphs can still give a single output but no longer have the same topology. What's the most naive thing we could do?

We could try to force the graphs into a similar shape. One way of doing so is to get rid of all topology, make all the nodes fully connected, and use edge labels to tell whether they were connected in the original or not. However, this does not work so well, especially from a computational point of view, because lots of graphs of interest are sparsely connected. Making the graph fully connected will result in a lot of unnecessary computation.

So are there any other solutions? In fact, we could ignore the topology mismatch and see whether it still works. To see whether something still works, two different parts need to be checked, whether it can still run and whether it can give good answers if it can still run.

Will the network still run if we have lots of different graphs as our data during training?

This question can also be broken down into two questions: does it still run as pseudo-code, and does it still run as code?

In terms of running pseudo-code, the answer is yes. Because everything is local, we are just iterating over local neighborhoods. We just have fewer local neighborhoods or more local neighborhoods for different graphs. The residual connections would definitely still work as long as the topology of each layer stays the

same. The normalization will still be meaningful, for example, if we just divide the nodes by the different numbers for different sizes of layers in the layer norm. It's only when the nodes are clustered ahead of time that the network may not be able to run. However, we seldom employ clustering because it does not make much of a difference (as mentioned before). Therefore, we can still run it as the pseudo-code.

In terms of running the actual code, we might have to worry about the size of the arrays allocated to make sure things fit. But the important thing is that the weights are not changed. The number of weights we have to learn does not change even if we have more different graphs.

Food for thought: Do we need to pad the graph with null nodes?

In CNN, we need to care about the pixels on the boundaries of images because they have less neighbor pixels. Usually we have zero padding for them or we just ignore these pixels. However, the nodes do not have the same number of neighbors generically. We don't have to pad the graph with null nodes because we never have to compute anything for the null nodes.

11.5 A little intro to RNN

After learning GNN, one question that pops up is whether we could have more weight sharing. In a ConvNet traditionally, the weights are shared within a layer but not across layers. We talked about the idea of the neural ODE, which was the perspective of a ConvNet as a ResNet solved as a differential equation. If the convergence of the neural ODE is wanted, we had to invoke some kind of weight sharing, at least hard or soft weight sharing across layers to ensure the limit existence since there were similar behaviors with respect to time. This requirement raises the question: should we share weights across layers?

We should do weight-sharing if the hierarchical structure has a self-similarity at different scales. This is one of the key design choices in the RNN family.

- Note that the word "layer" can be used in the same way we thought about the layer from a convolution point of view, which is what we backprop through. Once we implement weight sharing across layers, we can still backprop through it.

Let's give an example of an RNN now. Consider a task where you are required to identify a person's attributes based on their name. First, you represent the name by a sequence of characters, then represent these characters as a graph with internal labels. As such, we associate different names with different sizes of graphs, as shown in Figure 11.3.

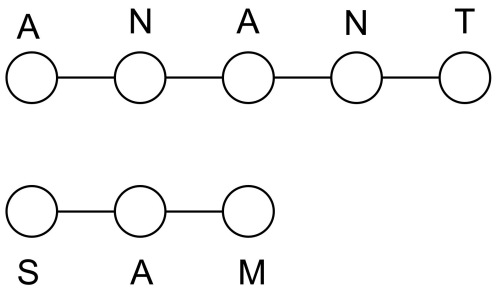


Figure 11.3: Graph interpretation of name stings

One possible solution is to just use a GNN. However, when people started working on these problems, they

were actually motivated by the connection with signal processing. In signal processing, you might have seen finite impulse response (FIR) filters and infinite impulse response (IIR) filters. The difference between them is that IIR filters have internal states, like momentum form in the momentum acceleration method. So for the things that are sequential in nature, we can employ the analogy of an IIR filter and think of a network that has internal states. We can treat these self-connections as a kind of internal state.

Another thing worth mentioning is that FIR filters act on the input but the IIR filter consumes the input, one at a time, like the momentum consuming the gradient in each cycle. The whole point of an RNN is to have this eating input behavior used in the network.

282 Scribing: Lecture 12

Jiayang Nie, Ophelia Wang

October 2022

1 Review: Skip Connections and ResNets

From the gradient perspective, residual connection blocks help address the issue of vanishing gradients. The key to residual block is in the gradient formula: $\frac{dH}{dx} = \frac{dF}{dx} + I$. Thus, the baseline gradient is 1 and can avoid gradient vanishing. The pesudo-code is in Figure 1. Through skip-connection, a deeper network yields better accuracy unlike what happens in vanilla convNet, as shown in Figure 1.

2 Speeding Up Training

- Use larger batch size, and linearly scale learning rate to speed up the training process.
- With larger batch size and possibly insufficient memory, use distributed data-parallel training by splitting one batch to different machines.
- To avoid over-fitting and obtain a better result in the test set, regularize by aggressive data augmentation.

3 Example-Difficulty of Samples in CNN

C-score is defined as

$$C_{p,n}(x, y) = E_D[P(f(x; D \setminus \{x, y\}) = y] \tag{1}$$

x, y are the design matrix representing pixels and the label respectively, and D are n i.i.d. samples from some population. We can think it as an analogy to leave-one-out validation. For example, as shown in Figure 2, the easier for the model to correctly identify an object, the higher the C-score would be. This is also can be thought of as consistency profile: the number of samples in the dataset with the same label that look like this image. For instance, the Game of Throne chair has a smaller consistency in compare to a regular chair in terms of identifying a chair. This gives us a proxy of how hard it is to identify one sample correctly.

In general, C-score increase as n increase, and C-score converge to 1 as n goes to infinity.

4 Depth-Complexity

Prediction depth is defined as the earliest layer in the model that classifies the sample correctly by KNN. Prediction depth is a good proxy for example difficulty. In general, examples with higher C-score (easier examples) have a small prediction depth and can be learnt in earlier epochs.

More formally, prediction depth is defined as:

$$argmin_{l \in L} f(x, \theta_l) = f(x, \theta_{>l}) \tag{2}$$

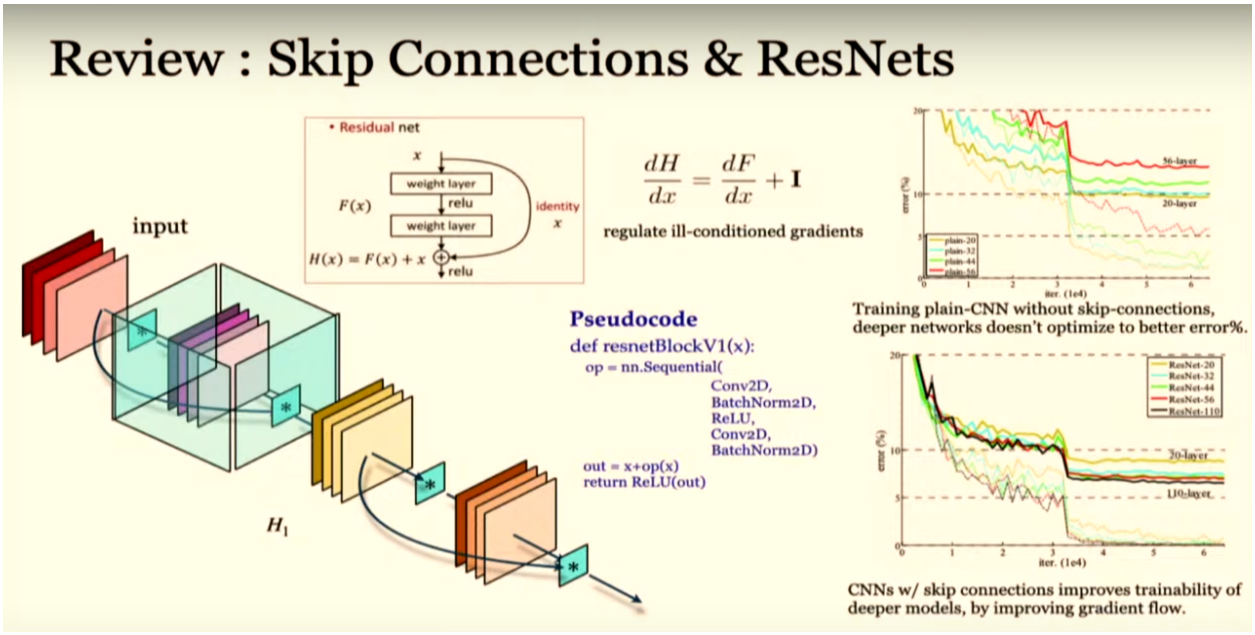


Figure 1: ResNet Layer Review [1]

5 Dense Prediction with Convolutions: Object Localization

In sparse prediction, the task is to predict the class of an image. In contrast, the task of object localization is to not only classify the image to a class but also detects which area of the object belonging to the class is in the image.

5.1 Problem Setup & Measurement Metrics

Previously we have $D = \{x_i, y_i\}$ where x_i represents the image and y_i represents the label. Now we have $D = \{x_i, y_i\}$ where x_i represents the image and y_i is a vector of (x_i, y_i, w_i, h_i) where (x_i, y_i) is the top left corner of the bounding box of the object, w_i represents the width and h_i represents the height. A common metrics for measuring performance is Intersection over Union(IoU). Assume we have a ground-truth box that locates the object, and the model predicts another box, IoU is the ratio of the intersection area(I) over the union area(U) of the ground-truth box and the predicted box. Usually, we declare that the model predicts correctly if IoU ≥ 0.5 and predicted class is right. Figure 3 gives a concrete example of what IoU is: the red area is the ground truth box, and purple area is the predicted box. IoU in this case is the area of the intersection of red and purple boxes over the area of the union of red and purple boxes.

5.2 Naive Approach

The most straightforward approach for solving the problem is to first train the classifier with cross-entropy loss, and then train a regression model on top of the convNets to learn the location of the box with Gaussian log-likelihood or MSE.

5.3 Sliding Windows Approach

A better approach is to classify every patch in the image. E.g. stretching and dividing an image into multiple sliding windows. Then then we could output the box with the highest class probability. In Figure 4, the original image of a cat is stretched vertically and horizontally so that we can find the optimal box that covers the cat object in this image. The key reason for stretching is to not limit the box to a fixed size box. For instance, for this example, the middle red box from the bottom image from Figure 4 is the best

Figure 2: C-Score Example [1]

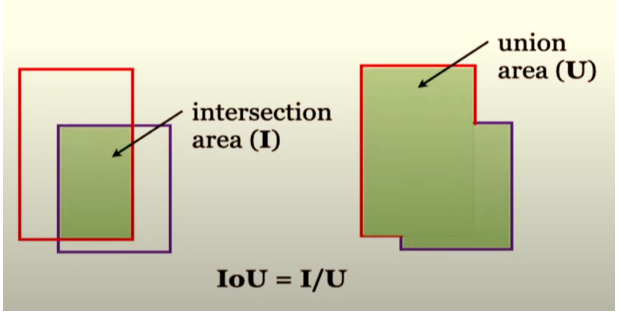


Figure 3: IOU Example [1]

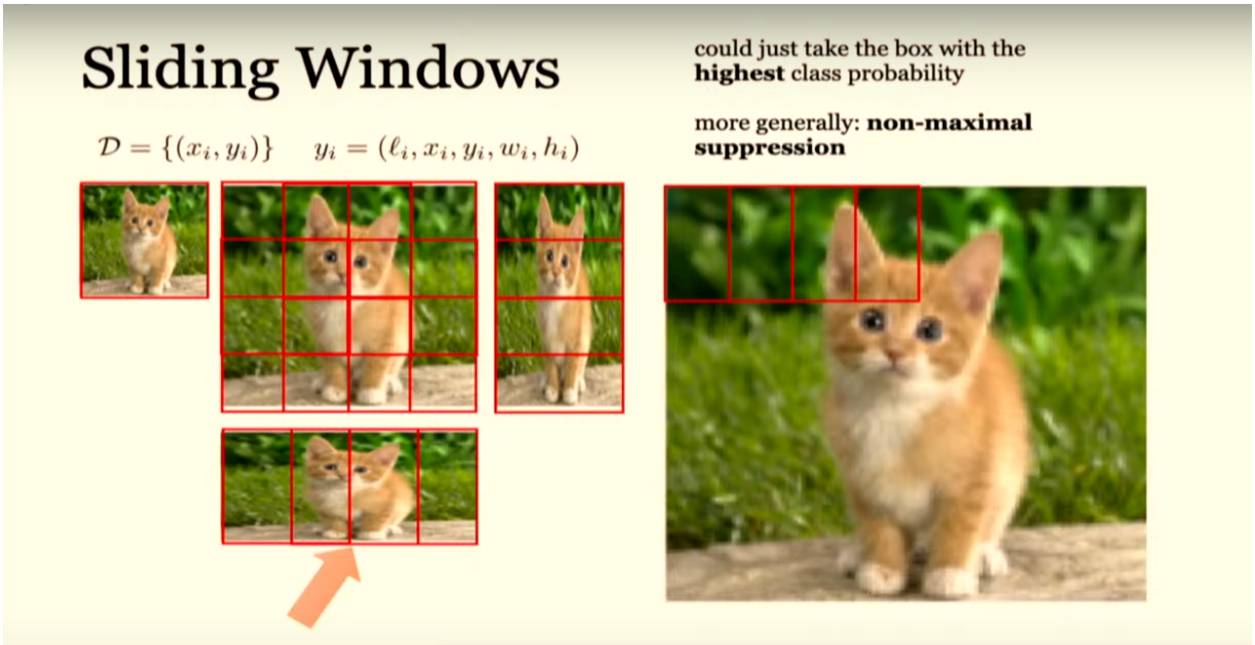


Figure 4: Sliding Windows [1]

bounding box. However, if we do not stretch it, then there does not exist a bounding box that is as good as this one in the original image. [Answer to audience's question]: In this example of find the cat, we should use tall and thin rectangles as the bounding boxes. Once we figured out which patch is the best patch, we can get the corresponding the corresponding bounding box in the original image by undoing projection. Then we can return the patch or the bounding box with the highest probability being a cat. In the case of localization, once we have all the predictions from the bounding boxes, the next question is that which one should we pick. In the case of localization when we know there is only one object in the image, then we can pick the one which has the highest probability. However if we are doing multi-object localization, then there are other algorithms such as non-maximal suppression. We basically say that we have some threshold and look particularly at a neighborhoods and pick objects corresponding to one particular class. [Answer to audience's question]: We look at different scales of the image and then run sliding window over that.

5.4 A Practical Approach - OverFeat

Combining the idea of regression and sliding window together, the approach of OverFeat provides a little "correction" to sliding window by adding small adjustments to the vector defining the bounding box. [3] We do the sliding window trick, but instead of predicting just the class, we also predict a bunch of coordinates, which can be think of as little corrections to the bounding box. First we can do a pre-train with the classifier, and then train the regression head on top of classification features. By passing over different regions at different scales, we can take an average of all the boxes as the final answer. Doing sliding window is expensive. Hence, the more practical way is to implementing convolutional layers to recuse calculations across windows.

Figure 5 shows an example adapted from the original paper of Overfeat. In combine of all the classification and regression heads from the sliding windows, the model is able to find the area in the image that it is most confident that it is a bear.

One of the downsides of using this approach is the increased computation complexity (36 windows = 36x the compute cost). To solve this, one reuse the calculations as shown in Figure 6. Fully connected layers are size-1 filter convolutional layers in disguise. Therefore, instead of using fully connected layers for classification task, we use convolutional layers. The benefit here is that when we scale the image, each convolutional layer will give more than 1 output, and we can reuse each of them for other windows. Assume

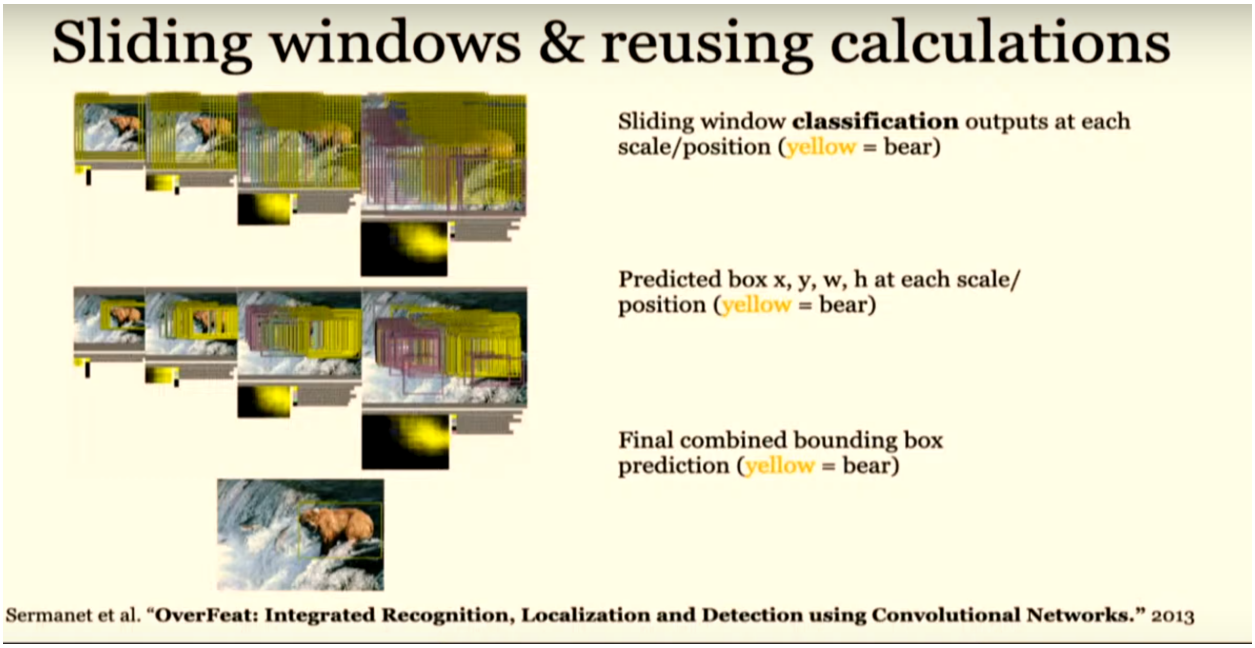


Figure 5: Overfeat Example [3]

we were running classifier on 14 x 14 image, by upsampling the image to be 16 x 16, the output dimension changes from 1 x 1 to 2 x 2. In this way, the computation time can be reduced to be about the same as convNets without sliding windows.

In summary, the building block is the convolutional network that outputs class and bounding box coordinates. Instead of looking at the combinatorial version of the problem, we look at different patches by upsampling and using a sliding window technique where we can still predict the probability of the class and the bounding box. To implement the sliding window effectively, what we do is instead of implementing the classifier with fully connected layers, we implement it with convolutions. Implementing the sliding window as just another convolution, with 1 by 1 convolutions for the classifier or regressor at the end to save on computation.

6 Object Detection

Then let's move on to object detection. Now the problem set up is slightly different. It is similar to the localization problem, but there is a added level of complexity where before we looking at the image and we have to predict details of one object. A more realistic setting is you want to identify all the objects in the image. The number of objects may be different for different images.

6.1 Dense-Prediction: Generating Multiple Outputs

One solution is: instead of making prediction for one class, we make predictions for all classes and predicting bounding boxes correspondingly. Each window can be a different object. Instead of selecting the window with the highest probability, just output an object in each window above some threshold.

6.2 Case Study

In Figure 7, this is a case study of the algorithm YOLO [2]. This algorithm provides a different take at the sliding window trick that we have seen in the lecture. The algorithm proposes that we only look at the image only once. Take this image and convert this into seven by seven grids in this example. For each of the grid, we predict what the bounding box is, what the class label is and additionally the confidence in our

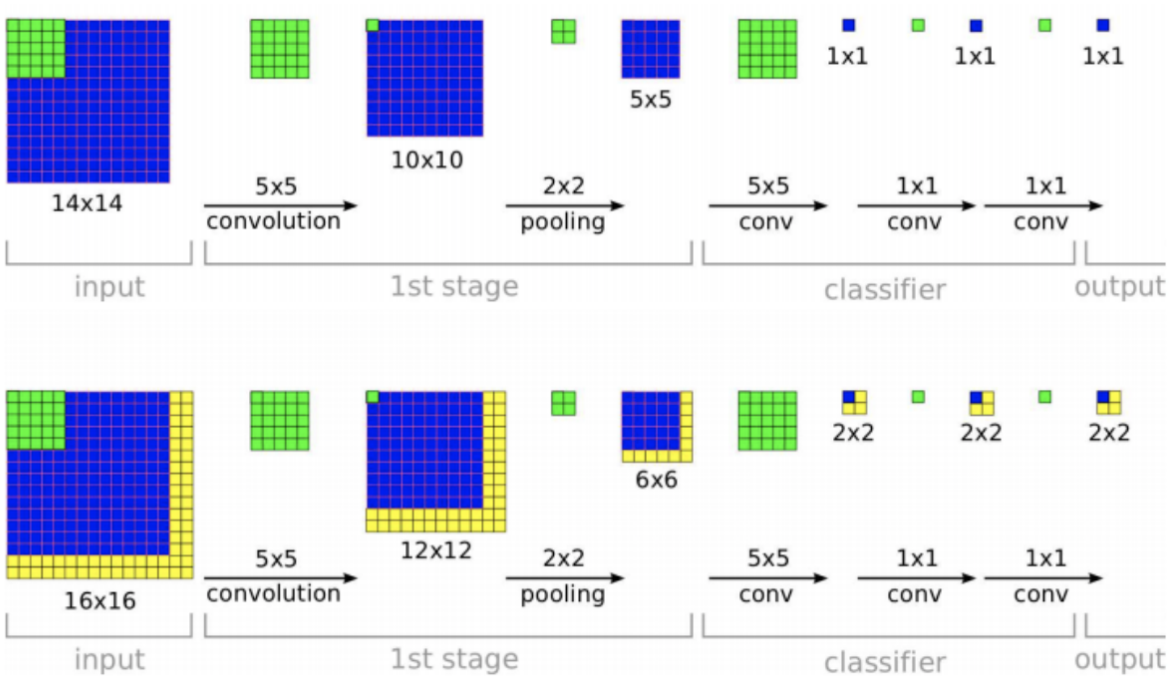


Figure 6: Convolutional Layers

own predictions such as IoU. For class label, we can use probability as a proxy. For the bounding box, we don't have any proxy of the confidence of the model.

6.3 CNNs + Region Proposal Networks

There is a different class of algorithms. Originally we pre-define which part of the image we should look at. Do we really need to do this before-hand or can my network learn to even predict where I should be looking at. The region proposal networks achieve that. The algorithm takes in an image as an input, then extract about 2 thousands region proposals. For each region proposal, the algorithm computes CNN features and then classify each region. This is a smarter sliding window technique. Instead of running sliding window on the region proposal on the input image, we first do the heavy lifting where we run the image through a coordinate for some layers. We get some activations and then run region proposal on top of these activations. How should we train the region of interest proposals? It's a very similar design to what we saw before such as OverFeat and Yolo, but now we predict if any object is present around that location. To build efficient detectors, we can use feature pyramids. We look at the image at different resolutions. We can think of it as am image pyramid where you have the lowest resolution version at the top and the highest resolution at the bottom. You can try model at each of the resolution and then generate predictions from each of these features and then pull them in some way such as looking at the max of those two to get your prediction. The key idea is to aggregate information across multiple scales.

6.4 Compute Efficient Detection

There are other different architectures people have run automated search for finding how we should connect these. Compared to mast RCNN and YOLO, the cost of training the Efficient Det models really pushes the boundary.

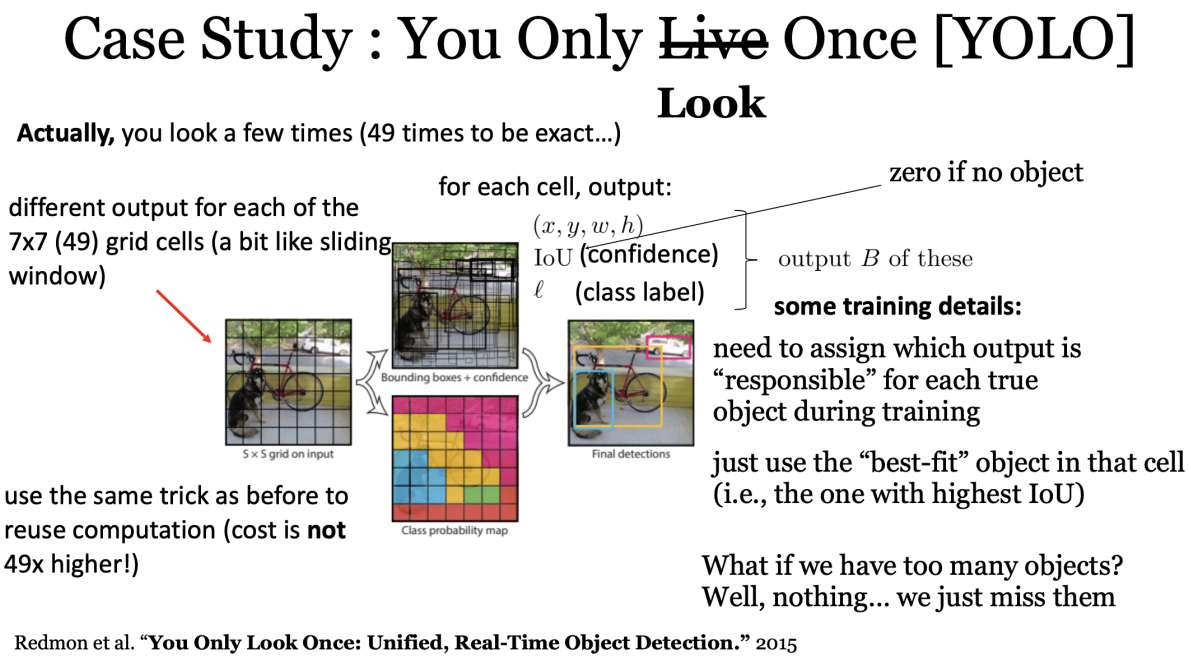


Figure 7: Case Study: You Only Look Once [2]

7 Semantic Segmentation

Previously we talked about how to locate one object inside an image. Now we would like to detect all objects in an image and label every single pixel with its object class.

7.1 Problem Setup

Assume there are K objects/classes inside an image, semantic segmentation is a K-class classification algorithm, predicting a label per pixel $y_i \in \{c_1, c_2, ..., c_K\}$. To make it computationally efficient, we can design a network a network architecture such as a fully convolutional network. We aim to have a set of operations that preserve the resolution at output. However, this is constrained by the fact that effective receptive field of convolution filters grows with depth. Only the early layers have the local view.

7.2 Conv. Operations: Down/UnSampling

Different convolutions operations can be used in the fully connected networks. There are normal convolutions, which reduce resolution with stride, padding, dilated convolutions, which increase receptive-field more rapidly, and transpose convolutions, which increase resolution with fractional stride. There are a lot of ways to design layers. One example is the Max Pooling, which remember the max element. In Max Unpooling, we set zeros at every other place and one only at the points which are used in the Max Pooling operations.

7.3 Bottleneck Architecture

As we get deeper in the network, we start to see a much bigger piece of the picture. Once we reach to low resolution, we can start up-sampling and turn low resolution feature vectors into high resolution per-pixel predictions. This is called a bottleneck architecture because we go from a high resolution and come to a low resolution with more depth and then go back to high resolution. Down and Up-sampling can be achieved with different kinds of convolution methods such as normal convolutions, dilated convolutions and transpose convolutions.

7.4 U-Net Architecture

When we down-sampling, we lose information, which cannot be recovered by up-sampling. The intuition behind U-Net architecture is to explicitly append filters from earlier down-sampling layer that preserve high-frequency details, concatnating them with filters along the channel dimension.

References

[1] Kumar Krishna Agrawal. "Guest Lecture: Kumar Krishna Agrawal". In: (2022). DOI: https://inst.eecs.berkeley.edu/~cs182/fa22/assets/slides/cs182lecture12vision.pdf.

[2] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: (2015). DOI: http://arxiv.org/abs/1506.02640.

[3] Pierre Sermanet et al. "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks". In: (2014). DOI: https://arxiv.org/abs/1312.6229.

EE 182 Scribe notes: RNN & LSTMs, Self-supervision

Zipeng Lin, Kuba Grudzien

December 8, 2022

1 Review for last lectures

So far, we have covered convolutional networks for images and generalized versions of convolutional network, graph neural nets.

We have the following are features for convolutional network for images.

Features of convolutional nets for images:

1. Weight-sharing across space
2. Residual Modules to support depth. Recall how we use the skip-connection to make sure the gradient is not vanishing.
3. Pooling to more quickly support long-range dependency. Pooling can process the convolutional result quickly.

Features of graph neural network (topology of image $\rightarrow$ topology of graphs) :

1. Weight-sharing across graph nodes. This is a more generalized version of CNN sharing since instead of grids, this shares weights across graph nodes.
2. modules in each layer can reference self, global state, and neighbor. This is because of the structure of graphs.
3. But need to aggregate symmetrically over neighbours. This is because we still need to maintain the invariance of the model.

2 Filters

We want to understand RNNs, and it turns out we want to first recall the filters in signal processing. They give the ideas of recurrent neural networks.

Definition 2.1 (FIR: Finite Impulse Response: generalized moving average)**.** ***Moving average*** *is taking the average while moving across different inputs. Recall in the previous lecture that momentum is just an exponentially weighted average.*

An example is output $y[t]$

$$y[t] = \sum_{i=-k}^{+k} x[t-i]h[i]$$

where x and h are the inputs and impulse response respectively.

*This is the definition of **convolution** (discrete). Notice that this is the same thing behind convolutional neural networks. Indeed, when we generalize this to 2D dimension we get convolutional neural networks.*

What if instead of having finite signals, we have infinite signals? This motivates us to look into IIR: infinite impulse response. Before going to the definition, we first realize that we can not really express the infinite operation by just writing them out. **We want a compact way** to write out infinite operations. The **key idea** to do this is to use hidden states.

Example 2.2 (Infinite impulse)

Consider the recurrence relationship

$$y[t] = a * y[t-1] + b * x[t]$$

we have both $x[t]$ and $y[t]$ are current state, while $y[t-1]$ is for past state. We get a sense that IIR is strongly related to time. Recall momentum discussion in lectures: results in exponentially weighted average.

The IIR filter has the key property that it processes the input sequentially. When there are inputs needed to be processed sequentially, IIR can be useful.

Similarly, we can process inputs sequentially:

$$\vec{y_t} = A\vec{y_{t-1}} + B\vec{x_t}$$

the vector version (still linear). The key idea is that the current state depends on the past state, and it is like momentum.

Let us consider a linear example: Kalman filter. It is a way to track the system.

Example 2.3 (Learned Kalman Filter)

Textbook definition of Kalman Filter (K.F): given known dynamic for some linear system driven by Gaussian Noise with known covariance. These dynamics have the state $\vec{h}$ hidden (think about this like the intermediate weights in MLP). Instead, we observe

$$\vec{x_t} = c\vec{h_t} + \vec{v_t}$$

where c and $\vec{v_t}$ are known, and $\vec{v_t}$ is known statistics.

K.F is a linear example. Therefore, we compute the K.F Dynamics (From known dynamics and observation structure).

$$\vec{h_{t+1}} = A\vec{h_t} + B\vec{x_t}$$

In a learned Kalman Filter, we do not know the dynamics. We want to learn A, B weights from data. Also, we know A, B matrices are constant across time. We want to solve the coefficients in recurrent relationships.

Let $W = A, B$ and h, x be the hidden state and input. W is the unknown weight to be learned. Assume we have traces of train data $\left(\vec{h_{t,j}}, \vec{x_{t,j}}\right)_{j=0}^{n_j}$ for the js being $1, 2, \ldots, m$. Notice that n_j is the length.

Setup of the RNN: see the figure 1, we input x from the bottom and input initial hidden state $\vec{o}$, the above are loss layer to calculate loss function and do gradient descent.

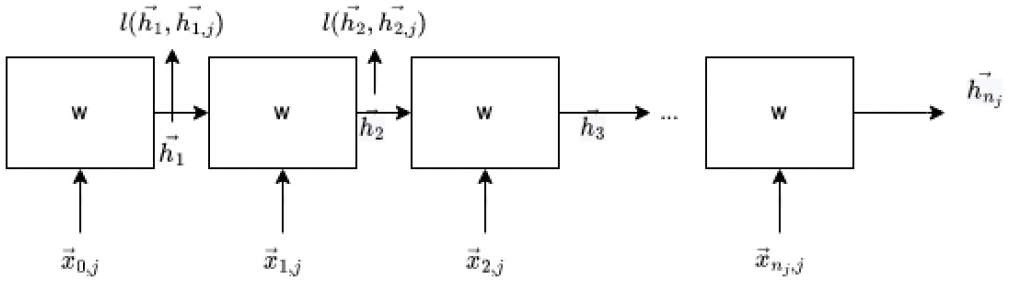


Figure 1: Simple-RNN

Now consider the inputs and outputs.

We have the inputs including the real world system, real-world states h_t, real random inputs u_t to this system, and real measurements/outputs x_t

We want to build a system (Kalman filters), a computation system specifically, to estimate the $\vec{h_t}$. In the textbook definition of Kalman filter, we just compute the system. The output of real-world-system is input for our computational system. The diagram from lecture below could improve your understanding:

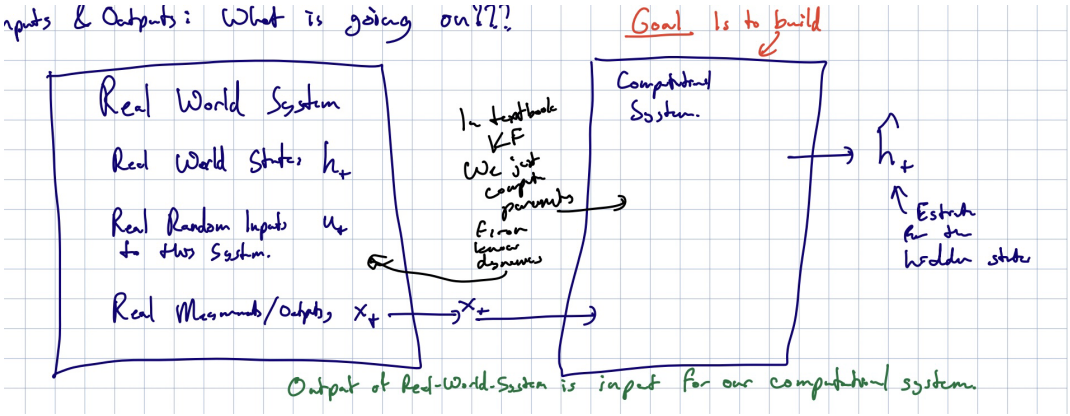


Figure 2: Kalman filter system

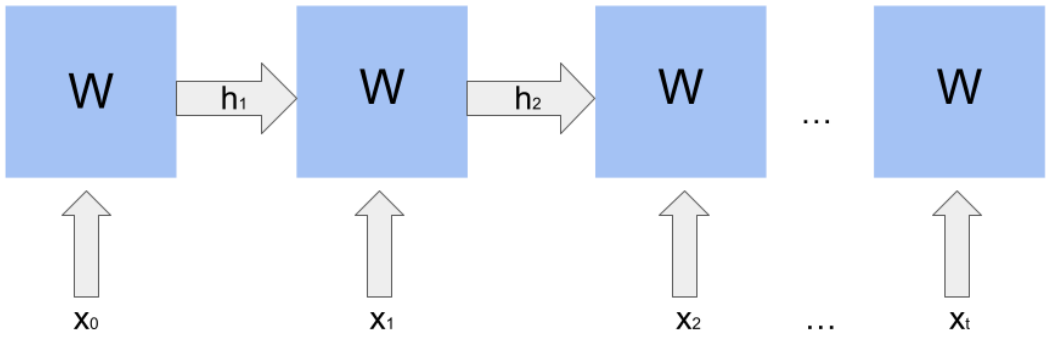


Figure 3: Simple RNN with input x and hidden state h

3 Recurrent neural network

Question: why do we take the loss like that? We compare our estimated value to the real value.

Question: why don't we compute h from x? This is because we want h to be dependent on time and want our system to reflect that. In the real world, we do not know what real data and we want to learn the filter dynamics.

We generalize the system described above to include non-linearities. (Aside, for the linear features, recall the neural network lectures, we can replace linear features with MLP).

In MLP, we can add expressiveness to the model by making the layers wider and the model deeper. Now the question is how to make complicated RNN models expand expressiveness. Recall the figure of RNN from above (figure 3). We can approach the problem from two kinds of perspectives: either we change the internal structure of RNN by a little bit, or like doing with other deep learning models, we stack over layers.

3.1 Choice one: stick to the picture but make things wider/deeper

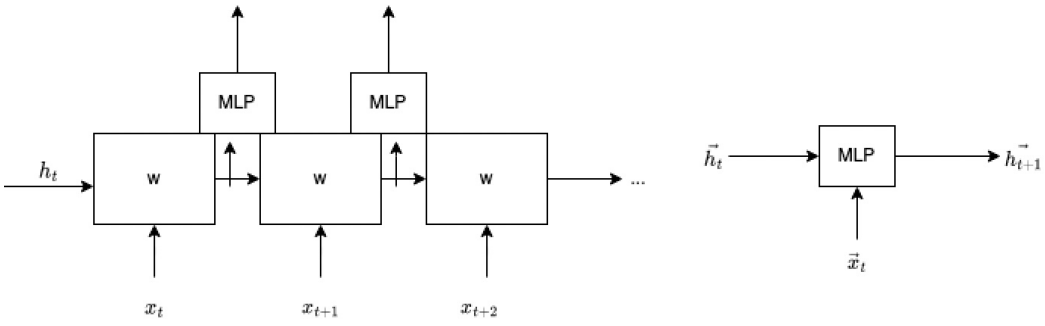


Figure 4: Option 1

We change two things here: the MLP and the h_t. We make both wider (MLP can be deeper too) to make it more expressive.

3.2 Choice two: Use layers of simpler RNNs

We use layers of simpler RNNs. Treat each RNN as a filter, we compose filters together.

It is the same way that convolution network gets deeper. We can put an output layer above, and the inputs are on the ground. The gradient could flow in two directions.

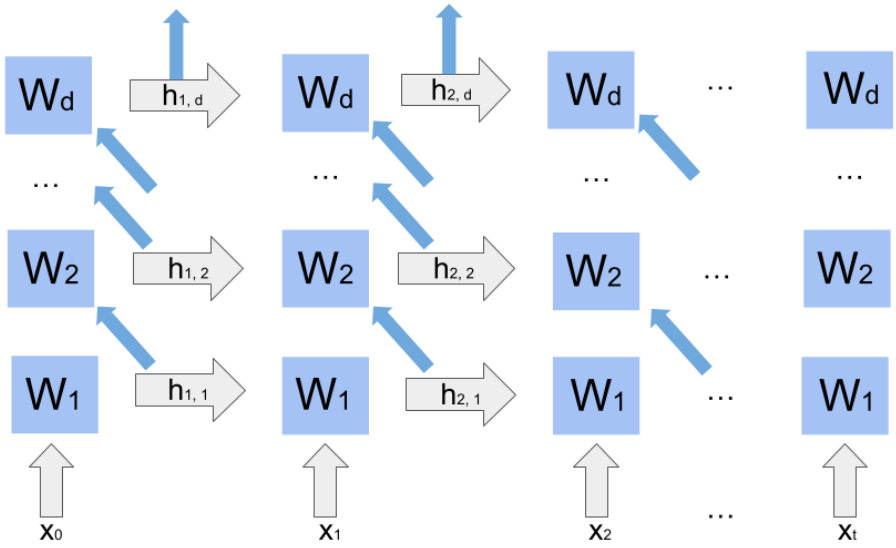


Figure 5: Option 2

3.3 RNN challenges

Similar to the challenges we face while using layers of CNN, we also need to deal with **dying gradients** and **exploding gradients**. In order to do that, we want to use a saturating non-linearity.

Definition 3.1 (non-saturating)**.** *A function f is non-saturating if and only if either its limit at ∞ is $+\infty$ or its limit at $-\infty$ is $+\infty$.*

Definition 3.2 (Saturating)**.** *A function is saturating if and only if f is not saturating*

Example 3.3 (ReLU is non-saturating)

The function ReLU reaches ∞ when $x \to \infty$ so it is non-saturating, which is why it is not often used in RNN. However, in the PyTorch version of RNN, you can still use ReLU to explore the details of computing.

Thinking 3.4 (Why not to use non-saturating nonlinearity?)
Since RNN are recurrent, the gradient would involve a lot of multiplication. Therefore, in order to prevent the gradient from exploding, we want to make sure the output **value** of the activation function would be smaller than one, otherwise, the result would diverge if we have too many RNN time steps.

Examples of saturating non-linearity, tanh and sigmoid.

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

and

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \in (0, 1)$$

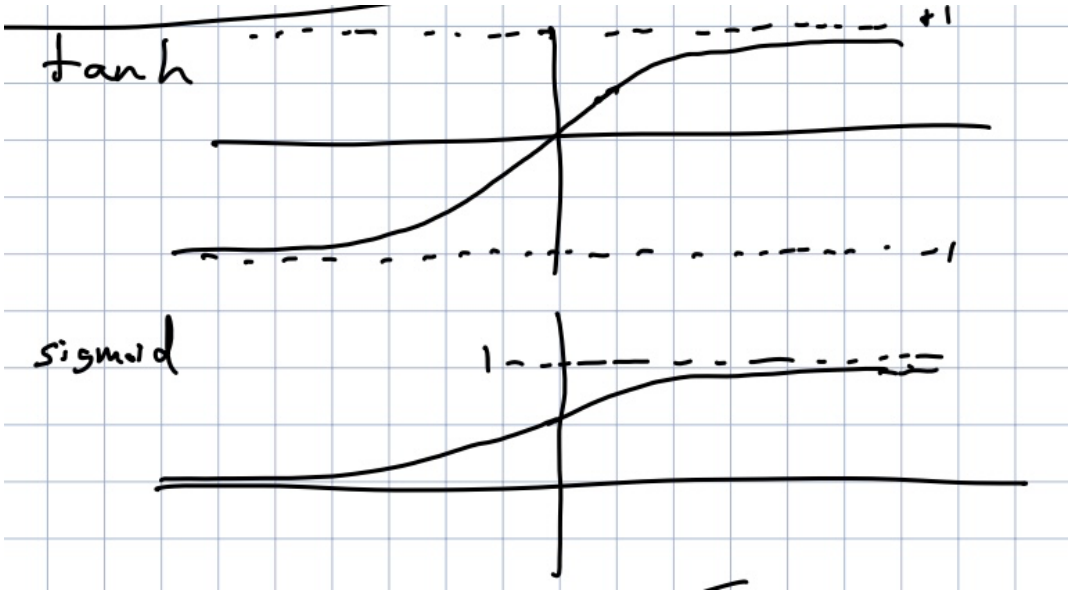


Figure 6: Graphs of two saturating non-linearities

From another perspective, we can use **Layer Normalization** on the data to prevent exploding gradient. We can apply layer normalization in two ways in the context of RNN: either we normalize the hs above, or, we can do the layer norm in the x direction. We can also put layer norm inside w_is. The reason why **we do not usually use Batch Normalization** is because it would not consider the recurrent part of the network, as in each recurrence calculation the statistics about the data would change. If you want to explore further, you can check paper `https://arxiv.org/abs/1603.09025` and see how reparametrization to get Batch Normalization work.

On the other hand, how do we combat dying gradients? Can we use "skip connection" like the one in ResNet? Yes, we can, but there should be discussions about which direction we should use

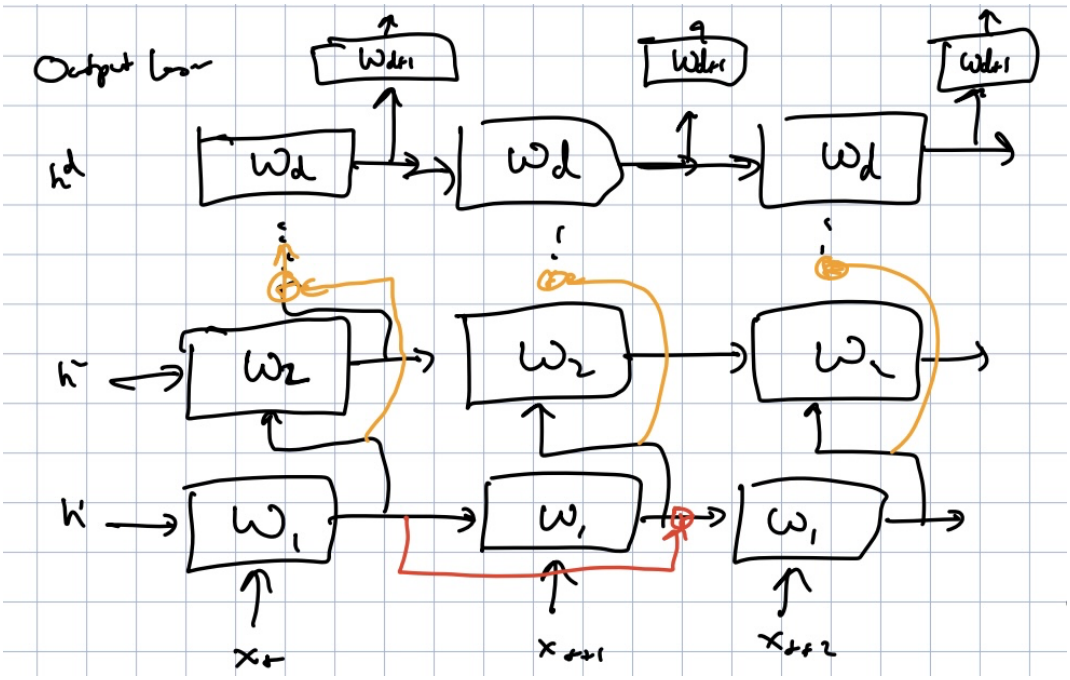


Figure 7: Several RNN layers

As we can see, the input is from the bottom, the hidden state are from the left. We have two options:

- Do ResNet skip connection in the vertical direction (orange lines)
- Do ResNet skip connection in the horizontal direction (red line)

We get the vertical skip connection could work, but the **horizontal one might not work**. The reason is that the horizontal direction is for the hidden state, so we ignore the current input slightly. In many applications, adding horizontal skips changes inductive bias in a bad way (since it can not go back without knowing the ignored inputs).

In order to address the horizontal direction of the skip connection, we introduce the following idea.

Key Idea: add a memory cell: make horizontal paths have a way for gradients to flow backward when those gradient values make sense, but which can learn to block gradients as well.

Context is represented by the symbol c_t and it should change but at a slower rate most of the time. However, when the input changes a lot, the context should change too. Most of the time, we want to multiply that with f_t. Usually, f_t is 1. However, sometimes f_t is not 1, we thus multiply the input by $1 - f_t$ and we have new context c_{t+1}. When $f_t = 1$, $(1 - f_t) * \text{input}$ has no affects on the context. f_t is the forget gate (1 being remembering). An example of f_t is

$$f_t = \text{sigmoid}(w_1 x_t + w_2 h + w_3 c_t + \text{ bias})$$

with $w_3 c_t$ sometimes being ignored. The input here would be

$$\text{input} = \tanh(w_1 x_t + w_2 h + w_3 c_t + \text{ bias})$$

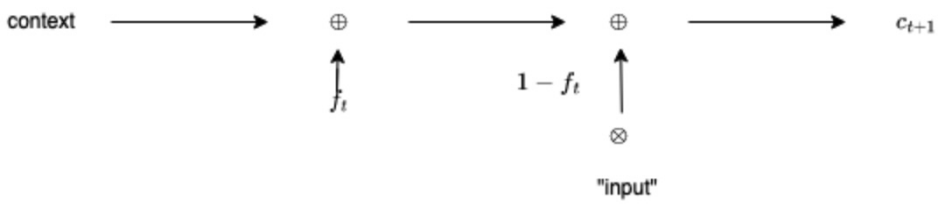


Figure 8: Flow in LSTM

The approach that follows the idea is adding memory to recurrent unit in addition to hidden state h_t.

Twist in practice (how it differs): usually, the current state needs to include the context, so we have the hidden state at time $t+1$, $\vec{h_{t+1}}$ is equal to

$$\vec{h_{t+1}} = \vec{o} \odot \tanh(c_{t+1}),$$
$$\odot : \text{element wise product},$$
$$\vec{o} : \text{computed with non-linearity from} x, h, \text{ and probably } c$$

this implies LSTM, which is in the discussion worksheet.

CS 282 Deep Neural Networks **Fall 2022**

Lecture 14: Attention/self-supervision

Lecturer: Anant Sahai *Scribe: Xin Chen, Yun Yeong Choi*

14.1 Attention

We have learned that the natural architecture design of deep learning models is useful for different tasks. For example, convolutional structure for computer vision tasks, sequential structure in RNN for language processing, forget gates in LSTM models, etc. Although the architecture of RNN model is useful for sequential data, it still has limitations in understanding human languages. One reason is that different human languages behave differently (e.g. different grammar, word orders, etc.) To handle this problem, we need to design some new architectures. First, let's look at the problem of machine translation as an example.

14.1.1 Motivation of Attention

When humans read a sentence, we read one word at a time. Therefore, we design the RNN encoder to do the same thing. As shown in Figure 14.1, each word is input into the encoder layers sequentially and each layer outputs a hidden state that is used as an input to the next layer. Each layer shares the same weights.

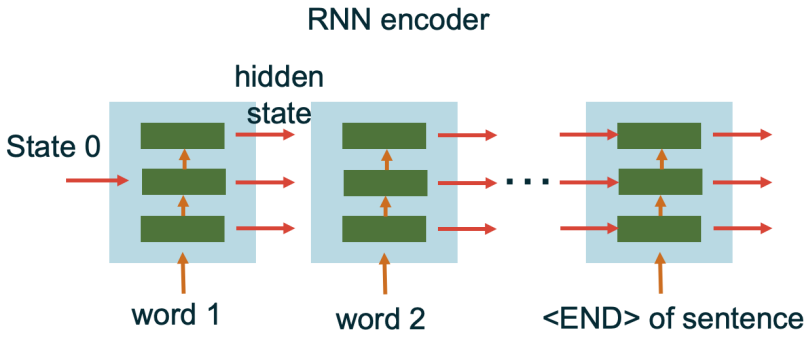


Figure 14.1: A RNN encoder used to process sequential data.

Aside: Why do we call the hidden output a state?

In a dynamic system, we define *"state"* as something that summarizes all the past that is relevant to the future. That is to say, if we know a state, we don't need to care about its past and that state will provide us with the information that we can use for the future. In the sentence example, the hidden state in the last layer should summarize all the previous words.

Up to this point, the normal RNN architecture works well. However, we want the model to translate the sentence into another language. Thus we need a decoder network after the encoder. The decoder will also use the sequential architecture and output the words in other languages, as shown in figure 14.2. The output in each layer will be used as the input into next layer.

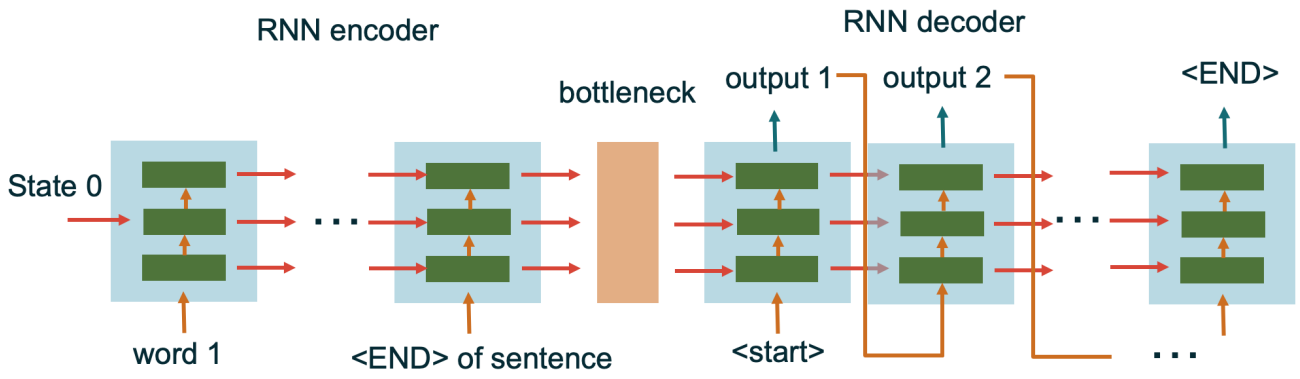


Figure 14.2: Encoder and Decoder in RNN.

Questions:

- Why can't we just map each word in the original sentence into the translated sentence?
  → Because different language has different grammatical structures. The first word in a sentence in language A may be the last word in language B.
- There are millions of words, do we need to have millions of output units for the last layer?
  → No. We can embed the word to appropriate high dimensional vectors.
- How do we translate output vectors to word?
  → Look for nearest neighbors of decoded vector.
- How do we propagate errors/loss?
  → 1) Just use output from decoder as a input. 2) Use local softmax for the rounding.
- How do we train the encoder and decoder?
  → Use supervised learning for decoder and self-supervision for encoder.

Note that there is a bottleneck between the encoder and decoder. We hope that this bottleneck can include everything about the input sentence (*recall "state"*). That is to say, we are squeezing all the previous information into this tiny bottleneck. This may cause a problem because there may be some details lost that are irrelevant to figuring out the structure of the output sentence but relevant to which specific word is in that sentence. For example, we input "My green cat is an alien". the workd "green" is used to decorate the "cat". In the output sentence in another language, we need to know which word ("green") is used to decorate the "cat". Then our model needs to look back into the input sentence. However, the architecture we just built cannot look back at things at different scales.

Recall that for image segmentation, the U-Net architecture concatenates features from the earlier layers to the upsampled features in the later layers to allow the model to look back into the encoder layers. Also,

images have a nice matrix structure for the model to process considering that we can clearly depict what the neighbors of pixels are. However, in the case of language, it is hard for the model to figure out the global idea of the neighborhood since it is not well defined in the language itself. **Therefore, we need to add something to our network, that allows the model to look back at the words that are originally embedded, and allows information to flow along the layers.**

Takeaway:

We want to add something in the language model, which allows the decoder layers to look back at what words are embedded. This is like a kind of memory where we are able to store something important in it and retrieve it later. We call this **"Attention."**

14.1.2 Attention structure: queries, keys and values

In the last section, we explained the motivation for "attention" and how it behaves similarly to "memory" in a computer. There are two methods of storing memory. One is like an "array," where we can store the value at an address and retrieve it later by looking into that address. The other method is like a "hash table," where we have a key that corresponds to a specific value and can retrieve the value by querying the key. Attention will use the hash table structure.

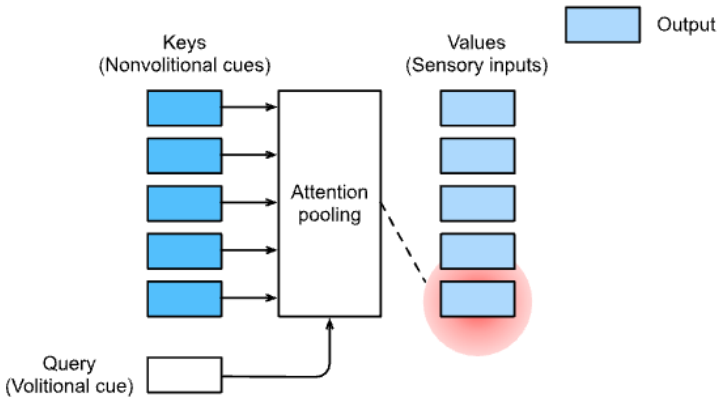


Figure 14.3: Scheme of attention. Query, Key, and Value [1].

Unlike the traditional hash table, now we can have a query vector that is not always exactly matched to keys in the hash table. So we need to design a "pooling" layer which not only can get an appropriate forward value, but also backpropagate gradients. Here are some ideas on how to design this hash table structure for attention.

1. Idea[-1]: A naive idea. Scan for an exact match to the key (like how a normal hash table functions). If found, return it.

   Problem: Really bad at initialization. When we initialize the model, we set a random key, and try to get the value with a random query. Then we are unable to get the value. Additionally, we are unable to calculate the gradient for backpropagation since even if we change the keys and queries by a small amount, there is still no matches, so there is no gradient.

2. Idea[0]: Scan for the closest match of query to key in the hash table, and return the value.

   Problem: Now we can retrieve a value and the gradient can reach the value. However, we still have the problem of calculating a gradient for the query and key. Note that we are using the approximation of query and key. This means that changing the query or the key by a small amount will result in the same value, so the gradients will be 0 and cannot update the weights.

   Whenever we make a hard decision, there's a gradient problem. We need to soften this hard decision.

3. Idea[1]: Scan for the closest matches of query to the keys. Then return the weighted average of the values.

   Since the weights now depend on both the query and the key, we have gradients on query, key and values. In this perspective, attention can be thought of as "queryable pooling", because keys and query values are also learnable parameters, but there are no learnable weights in the attention mechanism itself. As shown in figure 14.4, the hidden states in the encoder are used as keys and values, and the hidden state from a decoder layer is used as a query. The attention layer will output the weighted value which is used together with the hidden state from the decoder to generate the output. This output is then fed into the next decoder layer.

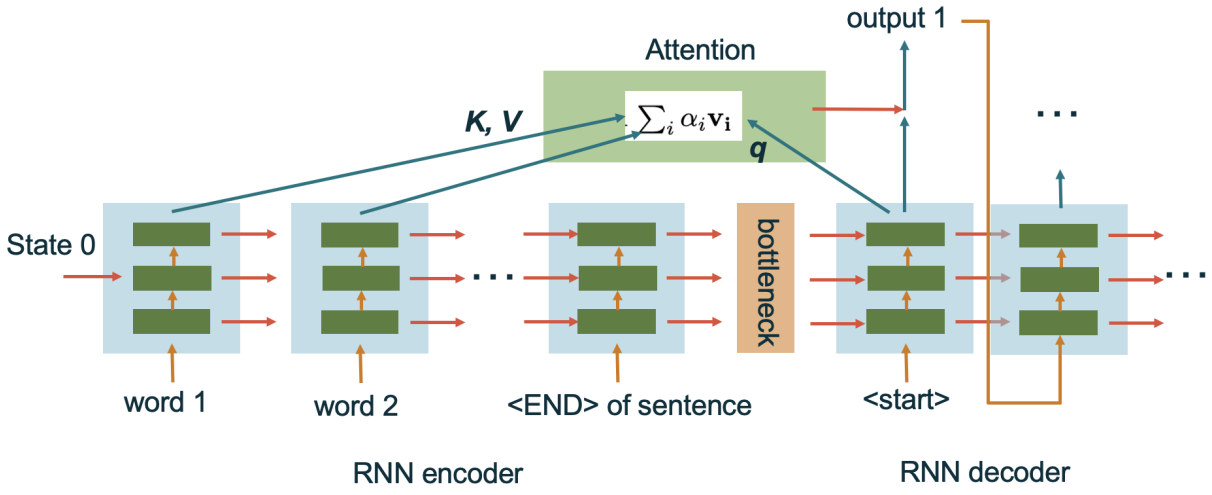


Figure 14.4: Scheme of attention in RNN layers.

Details:
With proper similarity function, weighted average will be

$$\textbf{Weighted Average} = \sum_{i=1}^{n} \text{Sim}(\textbf{query}, \textbf{key}_\textbf{i})\textbf{v}_\textbf{i} \tag{14.1}$$

where *Sim* is similarity function, **query** or **q** stands for query vector, $\textbf{key}_\textbf{i}$ or $\textbf{k}_\textbf{i}$ means ith key vector, and $\textbf{v}_\textbf{i}$ is a value corresponding to the ith value vector. Vectors/scalars are represented as bold/plain letters.

We have seen a similar ideas before - softmax and kernel-based methods. What should our similarity function be?

Inner product $\textbf{q}^\textbf{T}\textbf{k}$ or radial basis function (RBF) $\exp(-\gamma\|\textbf{q}-\textbf{k}\|^\textbf{2})$ are good choices. Note that the inner product should be normalized, otherwise if we have a long vector but low similarity, the inner product can likely still be large.

In "attention," we will use the normalized inner product:

$$e_i = \frac{\textbf{q}^\textbf{T}\textbf{k}_\textbf{i}}{\sqrt{d}} \tag{14.2}$$

where d is the dimensions of **q** and **k**. This scales the variance to 1 for random keys and queries, but still allows similarity to increase with $\mathcal{O}(\sqrt{d})$ when key and query are aligned.
After that, we use softmax to compute the weights:

$$\alpha_i = \frac{\exp(e_i)}{\sum_j \exp(e_j)} \tag{14.3}$$

Then "attention" will return $\sum_i \alpha_i \textbf{v}_\textbf{i}$ upon query.

Question:

- What can act as keys, values?
  → There are different components that can act as keys and values. For example, we can use a hidden state in the encoder as the key and another hidden state after that layer as the value. Also we can use the outputs in the decoder as the query.

14.1.3 Embed "order" information in Attention

Sometimes, we might need to take care about order information in the keys. For example, in natural language processing, "Prof. Sahai bites dog" and "Dog bites Prof. Sahai" makes the meaning totally different. One possible way to encode order/position/time is using complex vectors. If we let t be the position in sentence we want to encode, this can be differentiated from other sentence positions using complex expression.

$$e^{j\omega t} = \cos\omega t + j\sin\omega t \text{ or } \begin{bmatrix} \cos\omega_1 t \\ \sin\omega_1 t \\ \cos\omega_2 t \\ \vdots \\ \sin\omega_n t \end{bmatrix} \tag{14.4}$$

Note that we don't want to use the complex number in Neural Networks, so in practice, vector expression will be used. This positional encoding is useful since not only we can express relative shift using matrix multiplication ($e^{j\omega(t+\phi)} = e^{j\omega t} \times e^{j\omega\phi}$) but also we can differentiate vectors as on the unit circle in the complex plane. By concatenating the positional encoding to the key vector, we have both a regular key part and a positional encoding. Now we can query one or a combination of them.

Questions:

- Why do we need periodicity of time? We don't have that in our sentence.
  → The periodicity allows us to have an easier distinction between coarse and fine degrees of time. Additionally, we do not want to make our activations really big, and this periodicity confines the activations to a compact space.
- Is ω is learnable?
  → We can set it as learnable parameter. As far as we know, it has not proven beneficial to have learnable ω. People usually fix ω.

14.2 Self-supervision

In many contexts, we lack enough labeled data for supervising learning, so we may need to turn to unsupervised learning. There are two kinds of unsupervised learning: (1) Dimensionality reduction style (pre-regression) (2) Clustering style (classification).

Note that in these styles of unsupervised learning, there are no gradients or loss functions. But what if we can understand (1) and (2) in terms of loss function and gradient? That is to say, can we design these unsupervised learning techniques in terms of loss function, and then do gradient descent to update weights, which transforms this unsupervised learning problem into a "supervised learning", or **"self-supervision"** problem. Next lecture, we will see how we can turn the dimensionality reduction style of unsupervised learning into two different kinds of self-supervised learning problems – one we call the "autoencode style," and the other we call the "masked reconstruction style.

14.3 What we wish this lecture also had to make things clearer?

1. When talking about query, key and values, please use more clear examples to explain this concepts.
2. It would be better if the professor can give examples of word embedding.
3. We wish the professor can spend more time clarifying how position/time information is included in Attention. It would be better if the professor could present a trivial example.
4. It would be better if the professor talk more about the details of how the encoder/decoder is trained. For example, the professor mentioned we can just train the decoder alone without the encoder but we are not sure how. Currently it is very hand-waving.

References

[1] Aston Zhang, Zachary C. Lipton, Mu Li, Alexander J. Smola, Dive into Deep Learn-

ing, *arXiv preprint arXiv:2106.11342* (2021).

CS 182/282A Deep Neural Networks **Fall 2022**

Lecture 15: Self-Supervision and Autoencoders

Oct 13, 2022

Instructor: Anant Sahai *Scribes: Kiran Eiden, Lawrence Yunliang Chen*

15.1 Background: Unsupervised Learning

We start by recalling the main idea of unsupervised learning: given some data $\{\vec{x}_i\}$, discover an underlying pattern in the data. There are two basic types of unsupervised learning, which are:

1. Dimensionality reduction (e.g. principal component analysis). This is vaguely similar to regression, and the intent is to summarize or distill important information from the data. The algorithm we typically use for principal component analysis involves solving for a singular value decomposition, and thus is an eigenvalue computation-style algorithm.
2. Clustering (e.g. K-means). This is vaguely similar to classification, and the intent is to group related data points. K-means is typically solved using Lloyd's algorithm (Lloyd, 1982), which is an iterative alternating minimization-style algorithm.

We provide an overview of some of the possible uses of unsupervised learning in the two subsections.

15.1.1 Utilizing Unlabeled Data

The naive approach when doing supervised learning on datasets with unlabeled data is to simply discard the unlabeled data. Unsupervised learning allows one to take advantage of unlabeled data, and potentially improve upon the mapping found by only utilizing the supervised learning algorithm.

For example, imagine a large, high-dimensional dataset of dimension d with a small number of labeled points $n \ll d$. A pure supervised learning algorithm could not be applied to this dataset, as it would need to learn how to label data with d dimensions while only making use of the n labeled points. There are a couple of ways to resolve this using unsupervised learning:

- A dimensionality reduction algorithm can be applied to the entire dataset, including the unlabeled points. If the unsupervised learning algorithm is able to learn a mapping down to a low dimensional space with dimension $d' \leq n$, the supervised learning algorithm could potentially be applied to the low-dimensional representation of the dataset and successfully learn the proper labels.
- Similarly, clustering can be used to predict labels for unlabeled points in the dataset (assuming that points in the same cluster have similar labels). If the number of labeled points post-clustering $n' \geq d$, then the supervised learning algorithm can be applied to the new dataset with inferred labels.

For many problems in, for example, natural language processing and image processing, it is much easier to obtain unlabeled data than labeled data. The models often require large quantities of data to train, so it is

important to be able to utilize unlabeled data. In the context of deep learning and deep neural networks, this requires some extension of our unsupervised learning concepts like dimensionality reduction to work with gradient descent or another, similar approach.

15.1.2 Exploratory Data Analysis

Another possible use of unsupervised learning is in exploratory data analysis. High-dimensional data is difficult to understand and visualize, and sometimes one might want to use dimensionality reduction and clustering techniques to simplify the dataset for visualization purposes. This is also done in the context of deep learning, but will not be covered in this lecture.

15.2 Rethinking Dimensionality Reduction by PCA

Here we will only be considering principal component analysis (PCA) without removal of means. Let us compile all of our data into a data matrix

$$X = [\vec{x}_1, \vec{x}_2, ..., \vec{x}_n], \tag{15.1}$$

where each data point $\vec{x}_i \in \mathbb{R}^d$. Note that our data points are packed into X as columns rather than rows. We want to find a k-dimensional subspace S_k so that the average of the residual

$$||\vec{x}_i - P_{S_k}\vec{x}_i||^2 \tag{15.2}$$

is small, where $P_{S_k}\vec{x}_i$ is the projection of vector $\vec{x}_i$ onto the subspace S_k. We can write this in terms of the data matrix X and the Frobenius norm $||\cdot||_F$ as the minimization problem

$$\min ||X - P_{S_k}X||_F^2. \tag{15.3}$$

Classically, we would solve this problem by taking the singular value decomposition (SVD) of X. This can be written as

$$X = U\Sigma V^T \tag{15.4}$$

$$= \sum_{i=1}^{\min(d,n)} \sigma_i \vec{u}_i \vec{v}_i^T, \tag{15.5}$$

where the σ_i are our singular values and $\vec{u}_i$ and $\vec{v}_i$ are the rows of our unitary matrices U and V respectively. Our solution $\hat{S}_k$ is then given by

$$\hat{S}_k = \text{span}\,(\vec{u}_1, \vec{u}_2, ..., \vec{u}_k). \tag{15.6}$$

We can approximate X by truncating the sum over $\sigma_i \vec{u}_i \vec{v}_i^T$ at $i = k$.

This also gives us a simple definition for our projection. We can calculate $P_{\hat{S}_k}\vec{x}_i$ by defining a matrix

$$U_k = [\vec{u}_1, \vec{u}_2, ..., \vec{u}_k], \tag{15.7}$$

and then taking

$$P_{\hat{S}_k}\vec{x}_i = U_k U_k^T \vec{x}_i. \tag{15.8}$$

The k-dimensional representation $\vec{y}$ of $\vec{x}_i$ is just $U_k^T\vec{x}_i$, and $P_{\hat{S}_k}\vec{x}_i$ is a "reconstruction" of our original $\vec{x}_i$ in d-dimensional space. The transformation from $\vec{x}$ to its reconstruction is depicted in Figure 15.1.

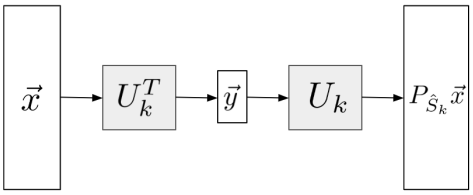


Figure 15.1: Dimensionality reduction and projection by PCA drawn as a block diagram.

We need to find a way to learn the mapping depicted in Figure 15.1 via gradient descent. This is discussed in the next section.

Note: If we replace X with X^T (i.e. if the data matrix contains row data instead of column data), we can see that U and V will be swapped in Equation 15.4. That means that for row data, our subspace in Equation 15.6 and matrix in Equation 15.7 would be defined by the rows of the V matrix from the SVD of X.

15.3 Autoencoders

We want gradient descent to be able to learn the map from $\vec{x}_i$ to $P_{\hat{S}_k}\vec{x}_i$. Matching the definition of U_k from PCA is not critical. There are multiple definitions of U_k that will produce the same $\hat{S}_k$ (consider permuting U_k and U_k^T, for example, or modifying U_k while preserving its column span). Furthermore, in the context of deep learning, we are not necessarily interested in learning an appropriate linear subspace, but some k-dimensional structure that might be defined by a non-linear mapping. We ultimately just need the k-dimensional representation to be useful for the purposes of our learning problem.

For further reference on autoencoders, see Goodfellow et al. (2016). Autoencoders are covered specifically in Chapter 14 of that book, which can be found online at `https://www.deeplearningbook.org/contents/autoencoders.html`.

15.3.1 Learning Problem Setup

Consider the setup shown in Figure 15.2. We take some input $\vec{x}$ and compress it down to a length k representation (bottleneck) called $\vec{y}$ via a linear map A. We then apply another linear map B to take it from the k-dimensional vector $\vec{y}$ to a reconstruction $\hat{\vec{x}}$.

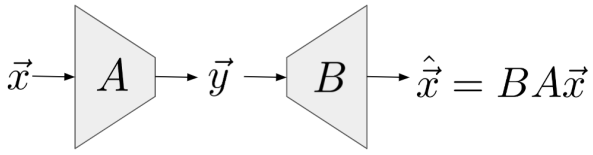


Figure 15.2: The basic form of an autoencoder.

We can complete the description of our learning problem by defining a loss function that is the mean-squared error of all $\vec{x}_i$ and $\hat{\vec{x}}_i$ in our dataset:

$$\mathcal{L}(\vec{x}, \hat{\vec{x}}) = \frac{1}{n}\sum_{i=1}^{n} \left|\left|\vec{x}_i - \hat{\vec{x}}_i\right|\right|^2 \tag{15.9}$$

$$= \frac{1}{n}\sum_{i=1}^{n} ||\vec{x}_i - BA\vec{x}_i||^2 . \tag{15.10}$$

This feels like a supervised learning problem. We are learning a mapping with parameters encapsulated in A and B that minimizes the difference between our reconstruction $\hat{\vec{x}}$ and our input $\vec{x}$. Specifically, it is a *self-supervised* learning problem, since our target is a function only of the input itself. Note that the identity is not expressible via this mapping since we pass through a bottleneck layer with dimension k, and the identity is a rank d linear map. The product BA can only produce a map of rank k, as A has dimensions $k \times d$ and B is $d \times k$. Thus we are learning an approximate reconstruction of our original data that minimizes the error for a given botteneck size k.

Note: The Eckart-Young-Mirsky theorem (Schmidt, 1907; Eckart and Young, 1936) tells us that the minimization problem described by Equation 15.3 for $\text{rank}(P_{S_k}X) \leq \text{rank}(X)$ always has a unique analytical solution in terms of the SVD of X. This motivates our approach to PCA. What we are doing here is effectively telling the computer that we know a solution exists and to go find our solution for us using gradient descent. In deep learning in general we often have less certainty – we tell the computer that we *hope* an answer exists to our minimization problem and that it has the representation we specified in our problem setup. We then ask the computer to go find it.

15.3.2 Non-Linearity and Autoencoder Approach

In general, we can replace A and B with non-linear encoder and decoder neural networks. This approach is called an *autoencoder* approach, since it encodes our input $\vec{x}$ in a low-dimensional representation $\vec{y}$ and learns how to reconstruct that input. Historically, autoencoder strategies were often used to first learn an initialization of the neural network weights, before training the network using the actual targets. This approach to neural network training was eventually replaced by end-to-end supervision, but has regained its footing in recent years as a pre-training approach for large models.

This pure form of autoencoder is not always an ideal way to solve a self-supervised learning problem, and many variations on the basic structure outlined here exist. Some of these are discussed in the subsequent section (Section 15.4).

For many practical problems, this basic form of autoencoder does not work so well because of the limited flexibility induced by the bottleneck. In particular, with this bottleneck, getting the training to work well for specific applications can be challenging, and in general, bigger networks will train more easily. So one may not want the data to go through the bottleneck but instead go through something bigger (i.e., allowing the dimension of $\vec{y}$ to be larger than that of $\vec{x}$). The issue, however, is that for a sufficiently large bottleneck layer the identity transformation will become a solution (i.e., the model learns the trivial matrices $BA = I$ and the latent representation $\vec{y}$ is not useful). We will discuss methods to address this issue in Section 15.5.

15.4 Parameterizations

In the previous section, we discussed the most basic form of an autoencoder, which we denote as **Parameterization 1**:

Parameterization 1:

The neural network contains two sets of learnable weights: matrices A and B, and they are independent.

From a classical point of view, one may argue that the only thing we need to learn is the subspace, which is the B matrix, and we do not need to learn the extra A matrix. Alternatively, one may argue that we only

need to learn the A matrix to compute the latent vector y and use the dimensionality reduction for other applications, while the B matrix is just there to set up the autoencoder. In fact, there are multiple ways to parameterize the learnable weights, which we discuss below.

15.4.1 Weight Sharing for A

We know that at optimality of Equation 15.10, $A = (B^TB)^{-1}B^T$ is the least square solution to achieve projection. This suggests we can do weight sharing between A and B.

Parameterization 2:

The neural network parameterizes the encoder weights as $A = (B^TB)^{-1}B^T$ and only learns the weight matrix B.

In this parameterization, the reconstructed $\hat{\vec{x}} = BA\vec{x} = B(B^TB)^{-1}B^T\vec{x}$. This is a differentiable function of the entries of B, and PyTorch can take gradients to learn B.

15.4.2 Partial Weight Sharing for A

In **Parameterization 2**, there is a complicated nonlinearity resulted from the matrix inverse $(B^TB)^{-1}$. Alternatively, one could replace the $(B^TB)^{-1}$ part by a learnable $k \times k$ matrix C. We thus get another parameterization:

Parameterization 3:

The neural network parameterizes the encoder weights as $A = CB^T$ and learns 2 weight matrices $B \in \mathbb{R}^{d\times k}$ and $C \in \mathbb{R}^{k\times k}$.

In this parameterization, the reconstructed $\hat{\vec{x}} = BA\vec{x} = BCB^T\vec{x}$, and the encoder and decoder share common weights B. As before, PyTorch can easily take gradients to learn both B and C.

15.4.3 Using the Inductive Bias of Gradient Descent

Apart from replacing $(B^TB)^{-1}$ with a learnable matrix C, we can also understand the inverse from another perspective. In general, taking the inverse of a matrix corresponds to solving some system of linear equations. From the perspective of deep learning, solving a system of linear equations is the same as minimizing a squared loss function. Here, the matrix $A = (B^TB)^{-1}B^T$ is the solution to the following least squares problem:

$$A\vec{x} = \arg\min_{\vec{y}} ||\vec{x} - B\vec{y}||^2 . \tag{15.11}$$

For deep learning, we can now solve A by gradient descent:

$$\vec{y}_0 = \vec{0}, \tag{15.12}$$

$$\vec{y}_{t+1} = \vec{y}_t + \eta B^T(\vec{x} - B\vec{y}_t). \tag{15.13}$$

In fact, we can draw out Equation 15.13 as a block diagram, as shown in Figure 15.3. $\vec{y}_t$ is first multiplied by $-B$, and then added to $\vec{x}$, before getting multiplied by ηB^T and added to $\vec{y}_t$.

Note that this block diagram looks like an recurrent neural network (RNN) block with a skip/residual connection. The blue dotted box in Figure 15.3 encompasses some computation where $\vec{y}_t$ is like a hidden

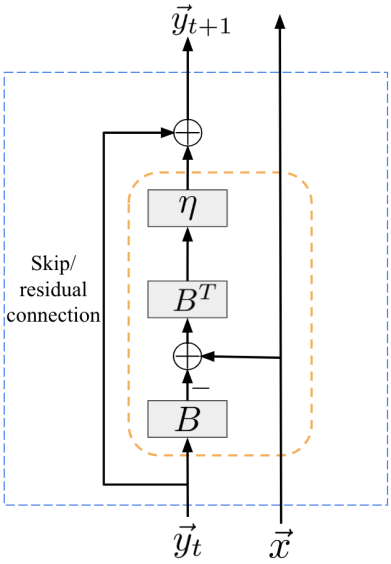


Figure 15.3: Equation 15.13 drawn as a block diagram.

state in an RNN that goes through the cell repeatedly at each time step, with an input $\vec{x}$ also fed into the cell. The orange dotted box encompasses some computation that learns the residual needed to add to $\vec{y}_t$.

With this observation, we can identify A as an infinite sequence of these computation blocks, where each pass through the RNN cell corresponds to a gradient step on $\vec{y}_t$, and after infinite gradient steps, $\vec{y}_t$ converges to the optimal solution of Equation 15.11, which is $A\vec{x}$. In this way, we unroll the gradient descent for learning the latent vector $\vec{y}$ as a deep neural network! Figure 15.4 depicts this deep neural network. We can summarize this as another parameterization method as follows.

Parameterization 4:

The neural network consists of a deep encoder in the form of an RNN with internal weights composed of B, and a linear decoder with weight B.

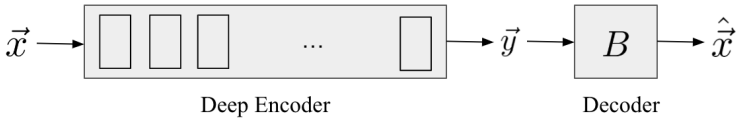


Figure 15.4: Autoencoder in the form of a deep RNN encoder and a linear decoder.

We note that the equivalence between $(B^TB)^{-1}B^T$ and an infinite sequence of the RNN cells is also related to the fact that a matrix inverse can be expressed as an infinite series. In practice, neural networks cannot be infinitely deep, so we just truncate it by setting a fixed depth (e.g., 10 layers).

We also note that **Parameterization 4** is just a variant of **Parameterization 2**. In fact, we can create other variants. For example, the RNN cell in Figure 15.4 does not need to look like Figure 15.3 — the weights B and B^T in Figure 15.3 can be some other learnable weights.

Comments: Dimensionality of $\vec{y}$ and motivation for data augmentation

- If $\dim(\vec{y}) = k < \dim(\vec{x}) = d$, none of **Parameterizations 1-4** will learn the identity transformation

if the model is linear. But if $\vec{x}$ has a low intrinsic dimension $d' < k < d$, it is possible for $\vec{y}$ to learn to contain enough information to fully recover $\vec{x}$, especially if there are bias terms or nonlinear layers. And that may be what we want, as $\vec{y}$ learns to capture the intrinsic dimension of $\vec{x}$.

- However, there are still some potential issues with the basic autoencoder approach (**Parameterizations 1-3**) of minimizing Equation 15.10. This is because there may be many choices for what $\vec{y}$ can be. In particular, $\vec{y}$ may contain all the information necessary to reconstruct $\vec{x}$, but it may also contain some other spurious information. And while the model may reconstruct $\vec{x}$ on the training data well, it may spuriously rely on the noises in $\vec{y}$ so that the reconstructions become incorrect and actually amplify the noise if $\vec{x}$ is noisy.

- In contrast, for **Parameterization 4**, usually B will learn to have some large singular values and some small singular values. Because of the implicit regularization effect of gradient descent, with early stopping, the latent vector $\vec{y}$ will adapt to move in the direction of large singular directions and not move much in the direction corresponding to small singular values. This has the interpretation that $\vec{y}$ does not move far in the badly-conditioned directions (since the network is not infinitely deep). In other words, it is more robust to noise. On the other hand, **Parameterizations 1-3** do not have such inductive biases built into the architecture.

- This motivates the approach of further introducing inductive biases through **data augmentation**, which we will discuss in the next section. In particular, data augmentation can encourage the model to learn $\vec{y}$ to preserve and strengthen the true signal in $\vec{x}$ and ignore (and ideally remove) false signals/noises in $\vec{x}$.

- If $\dim(\vec{y}) = k \geq \dim(\vec{x}) = d$, the 4 parameterization approaches need to be adjusted slightly:

  - **Parameterization 1:** No change needed. We still have $A \in \mathbb{R}^{k \times d}$ and $B \in \mathbb{R}^{d \times k}$.
  - **Parameterization 2:** The minimum norm solution to Equation 15.10 is now $A = B^T(BB^T)^{-1}$. So now $BA = BB^T(BB^T)^{-1} = I$!
  - **Parameterization 3:** Similar to before, we can replace the $(BB^T)^{-1}$ part by a learnable $d \times d$ matrix C. So now $A = B^TC$, and $BA = BB^TC$.
  - **Parameterization 4:** No change needed. We still solve Equation 15.11, and while the optimal solution changes to $A = B^T(BB^T)^{-1}$ when B becomes a wide matrix, the gradient descent formula Equation 15.13 does not change! Thus, the RNN remains the same, and this parameterization does not care whether B is tall or wide.

  ⋆ In this case, we see that **Parameterization 2** will always learn the identity transform, while **Parameterizations 1 and 3** *may* learn an identity transform (e.g., if it learns $C = (BB^T)^{-1}$). **Parameterization 4**, however, will likely not learn the identity transform because of early stopping of the gradient descent (recall that the neural network is not infinitely deep). This is another attractive advantage of **Parameterization 4** over the others.

15.5 "Excorcising" the Fear of Learning the Identity

As noted in the previous comment, when $\dim(\vec{y}) = k \geq \dim(\vec{x}) = d$, it is possible for the network to learn the identity transformation. One way to deal with this is data augmentation, where we change the input to be different from the target. With this change, the identity transform is no longer the optimal solution, and the hope is that the network will not learn the identity transform (except for **Parameterization 2**, which has no choice but to learn the identity).

15.5.1 Use Data Augmentation: Denoising Autoencoder

Recall data augmentation in computer vision, where we modify the images (e.g. rotate, crop, brighten) but keep the target labels the same. Here, the data augmentation we do is adding noise. We keep the target $\vec{x}$ the same but change the input to be $\vec{x} + \vec{n}$, where $\vec{n} \overset{\text{iid}}{\sim} N(0, \sigma^2)$.

Note that for the bottleneck architecture ($k < d$), the correct solution (Equation 15.6) has the property of averaging out noise. This is because the noise of the input, which is d-dimensional, has about $d\sigma^2$ energy. After projecting down to a k-dimensional subspace, the total energy of the noise is about $k\sigma^2$. This means that, the output $\hat{\vec{x}}$ only has about $k\sigma^2$ noise. Therefore, this denoising data augmentation creates an inductive bias to encourage the model to get rid of noise or average it out.

Also, as pointed out in the previous comments, for the $k \geq d$ case, **Parameterization 4** can achieve successful denoising with a deep (but not infinitely deep) neural network by learning a matrix B that has large and small singular values (so $\vec{y}$ learns to move in the well-conditioned direction and ignore the noises).

15.5.2 Masking/Inpainting: Kind of Data Augmentation

Another approach of data augmentation is masking/inpainting. Instead of adding noise, we remove entries from the input. For example, given a data point $\vec{x} = [x_1, x_2, x_3, x_4, x_5]^T$, we mask the input and feed into the neural network $[x_1, ?, x_3, x_4, ?]^T$, and ask the network to reconstruct $\vec{x}$. Again, the identity transformation is no longer the optimum. But suppose the underlying structure of $\vec{x}$ is 1-dimensional, it is possible for the model to learn to predict x_2 and x_5 from the other entries.

As we see, learning a low-dimensional subspace allows us to do many tasks, including denoising, straight autoencoding, as well as filling the blanks. Because the underlying structure supports many tasks, we can do any of these tasks for self-supervision, and it is useful to do so.

In practice, we need to think about how to implement masking. The simplest choice of putting in 0's does not work, as 0 is not a mask and will want to be reconstructed as 0. We will talk about this in the next lecture, and we will see that the architecture shown in Figure 15.4 is more suitable for masking than that in Figure 15.2.

15.6 What we wish this lecture also had to make things clearer?

1. In lecture, we used Figure 15.1 to describe PCA in terms of a linear map U_k derived from SVD. We then discussed how we might want to add some non-linearity (and indeed, how that is much of the point of moving to deep learning). However, when we went to describe the autoencoder, our autoencoder structure was effectively the same as Figure 15.1 and still utilized linear maps. We just replaced U_k and U_k^T with the matrices A and B with learnable parameters. It might make more sense to describe the autoencoder structure in terms of arbitrary non-linear maps A and B with learnable parameters, and then make any arguments we need to make about the learning problem (e.g. whether we can learn the identity) in terms of general non-linear maps. Alternatively, it might make more sense to discuss how we want to generalize to non-linear maps after writing down the basic autoencoder structure in terms of linear maps rather than before.

2. It would be clearer to summarize the contrast of the 4 parameterizations in each of the settings individually ($k < d$ and $k \geq d$ and with and without data augmentation) instead of talking about them at the end when multiple tweaks have been introduced. For example, currently, the $k \geq d$ case is discussed together with the denoising autoencoder, and when talking about the denoising effect of

Parameterization 4, it is not immediately clear which component is the main contributing factor, i.e., whether it comes from $k < d$ or the early stopping of gradient descent or the denoising data augmentation task. Similar for the disadvantages of the other parameterizations: under precisely what settings do they work or not work.

3. Some demos or plots showing different behaviors when the latent dimension (k) is smaller/larger compared to the input dimension (d) will be really helpful for interpreting the expected results.

References

Eckart, C. and Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1:211–218.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137.

Schmidt, E. (1907). Zur theorie der linearen und nichtlinearen integralgleichungen. *Mathematische Annalen*, 63:433–476.

CS 182/282A Deep Neural Networks **Fall 2022**

Lecture 16: Self-Supervision and Autoencoders (cont.)

Oct 18, 2022

Instructor: Anant Sahai *Scribes: Xingyi Yang, Sayan Seal*

16.1 Autoencoder Style Training

In Lecture 15, we learnt the structure of Autoencoder, which involves an Encoder, a Decoder, and a reconstruction loss for the training of parameters. Given a dataset of unlabeled data $\{\vec{x}_i\}$, we can train the Autoencoder, so that the Encoder transforms the input $\vec{x}_i$ into a new latent representation, denoted as $\vec{l}_i$, and the Decoder reconstructs the input $\vec{x}_i$ from the transformed representation $\vec{l}_i$. Then, the question is, how does an Autoencoder help practical machine learning tasks? When do we need an Autoencoder and how do we use it?

The key task is performed on the Encoder side. By training an Autoencoder, we get an Encoder which can transform a data point $\vec{x}_i$ into a transformed representation $\vec{l}_i$. Since we can recover much information of $\vec{x}_i$ from $\vec{l}_i$ through the Decoder, and $\vec{l}_i$ often is lower-dimensional, we assume that $\vec{l}_i$ keeps the essence of $\vec{x}_i$, while dropping some noisy information. The goal here is to learn some underlying pattern implicitly such that this will be helpful for other downstream tasks. The basic structure of an Autoencoder is shown in Figure 16.1. The part in the dotted box is important as it performs some encoding to appropriately distill the pattern. The remaining part serves as a *surrogate task*, which helps in the passage of gradients during training, but is trivial compared to the Encoder. In self-supervision, conceptually we create a surrogate task, where the target is a function of the input.

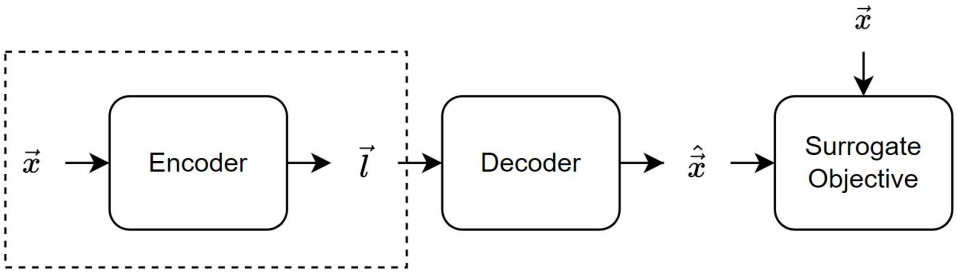


Figure 16.1: The basic form of an Autoencoder. We focus on the Encoder part in the following process.

Consider we are given a task with a set of labeled data $\{(x_i, y_i)\}$, and we are going to train a supervised model which provides a mapping from $\vec{x}_i$ to y_i. Normally, we train a deep neural network which maps input $\vec{x}_i$ to output $\hat{y}_i$ directly, and minimize the divergence between $\hat{y}_i$ and the ground-truth label y_i, as shown in Figure 16.2. However, if we are interested in classification, we do not optimize on the probability of misclassification, but on other forms of loss (surrogate loss) like squared error, one-hot encoding, cross-entropy loss or hinge loss. This pattern works for many tasks, but it may fail sometimes. The neural network may not train well if there are too little labeled data to support the training of a complex deep neural network of $\vec{x} \to y$.

Then comes the idea of using Autoencoder. Now that $\vec{l}_i$ provides a conciser representation of $\vec{x}_i$ itself, while

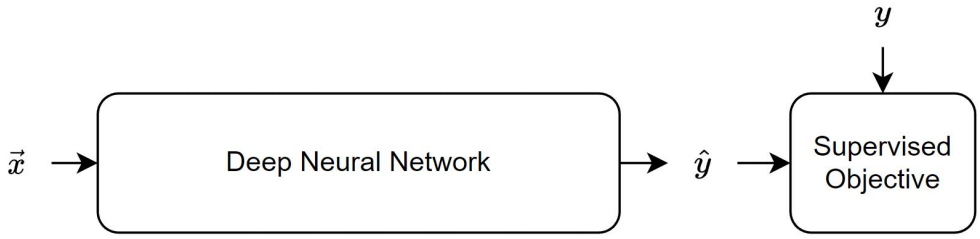


Figure 16.2: The common pattern for learning a supervised task using neural network.

keeping the most essential information, we can freeze the Encoder and learn a neural network which maps $\vec{l_i}$ to y_i, instead of $\vec{x}_i$. This way, the Encoder provides a refined representation of data, from which we can train a simpler neural network using gradient descent with less parameters. In this process, the Encoder takes a part of the representing work of neural networks, so that it is easier to train a neural network on top of it. The big picture of this idea in shown in Figure 16.3. Alternatively, the entire network can be learned in an end-to-end manner, with necessary fine-tuning of the encoder.

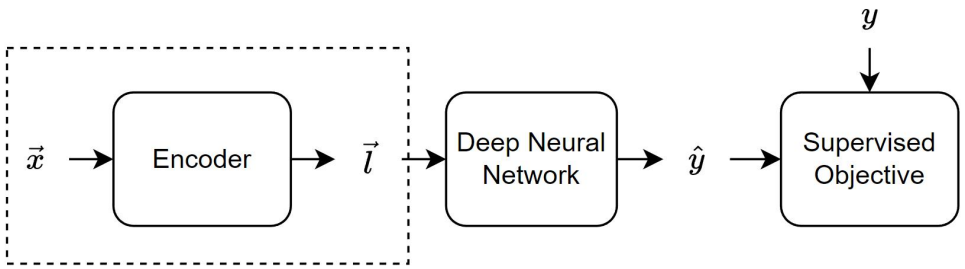


Figure 16.3: The big picture of using an Autoencoder to help a supervised task.

A big advantage of using Autoencoder is that we can use a larger set of unlabeled data for training of Autoencoder and leverage the knowledge for downstream tasks which come with a small set of labeled data. More specifically, consider a downstream task of mapping $\vec{x}_i$ to y_i, where there is a dataset $D = \{(x_i, y_i)\}$ of n pieces of data. And there is another unlabeled dataset $D_u = \{x_i\}$ of size N, where $N \gg n$. For the Autoencoder, we can use both D and D_u to train a better Encoder, because Autoencoder is an unsupervised model. This serves as a surrogate task. Then, for the downstream task, though we cannot use the dataset D_u directly, we manage to leverage the knowledge behind D_u by using the encoded representation $\vec{l_i}$ of Autoencoder. Figure 16.4 demonstrates this intuition.

This concept is very close to pre-trained models. However, the key difference between this type of approach and pre-trained models is that, here the surrogate task is not particularly interesting, but is designed to just learn the underlying pattern. In the case of pre-trained models, both the pre-training task to learn some pattern, and the actual task, which uses these patterns, are interesting. For example, pre-training a classifier on ImageNet, and then replacing the final classification layers to train on a smaller different dataset, such as classification of cancerous and non-cancerous tissues, are both interesting tasks.

16.2 Different Autoencoders based on Surrogate Tasks

There are multiple choices of surrogate tasks when we try to learn an Autoencoder. In terms of pre-process of input, there are three common tasks available: *vanilla autoencoding*, *denoising autoencoding*, and *masked*

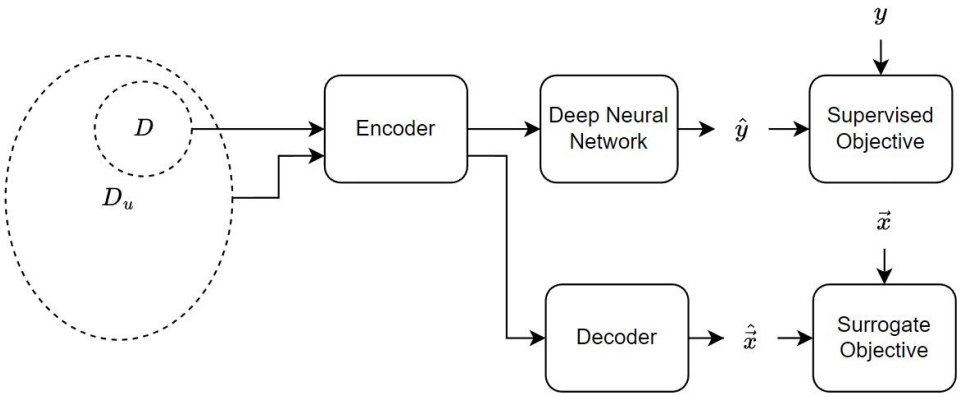


Figure 16.4: The demonstration of how an Autoencoder can help the downstream task by leveraging more training data in a self-supervised manner.

autoencoding. In the following parts, we denote $E(\cdot)$ as the Encoder function and $D(\cdot)$ as the Decoder function. The surrogate objective is denoted as $\mathcal{L}(\vec{x}, \hat{\vec{x}})$. The surrogate objective $\mathcal{L}$ is typically L2 distance between $\vec{x}$ and $\hat{\vec{x}}$.

16.2.1 Vanilla Autoencoder

The most basic idea is to directly use $\vec{x}_i$ as the input, and expect the Autoencoder to reconstruct the input $\vec{x}_i$ after decoding. Specifically, it minimizes $\mathcal{L}(\vec{x}_i, D(E(\vec{x}_i))$ directly.

It is the most naive approach to train an Autoencoder.

16.2.2 Denoising Autoencoder

We can also add some noise to the input $\vec{x}_i$ to train the model ability of denoising. That is, given an input vector $\vec{x}_i$, we add noise $\vec{n}$ to input $\vec{x}_i$ and give $\vec{x}_i + \vec{n}$ as input to the Encoder, where $\vec{n} \overset{\text{iid}}{\sim} N(0, \sigma^2)$. The surrogate target of the Decoder is still $\vec{x}_i$. Specifically, it minimizes $\mathcal{L}(\vec{x}_i, D(E(\vec{x}_i + \vec{n}))$. Then, to minimize the surrogate objective, the Autoencoder has to specify and diminish the noise part in $\vec{x}_i + \vec{n}$ when encoding. Thus the learnt Encoder is more robust to noise on the input. We can consider this approach as a kind of data augmentation. That is, by adding the noise term, more training data are generated to train the model. Moreover, this approach does not suffer from the problem of learning the identity mapping, while the Vanilla Autoencoder does. This is because, in this case, the input is different from the target, and the identity mapping is not the optimal solution.

16.2.3 Masked Autoencoder

Another approach of data augmentation is to mask some parts of each input vector $\vec{x}$. For example, given an input $\vec{x} = [x_1, x_2, x_3, x_4, x_5]^T$, we may randomly drop some of them, say, let the input to the Encoder be $\vec{x}' = [x_1, ?, x_3, x_4, ?]^T$. Specifically, it minimizes $\mathcal{L}(\vec{x}, D(E(\vec{x}'))$. The surrogate target of the Decoder is still $\vec{x}$. Then, to minimize the surrogate objective, the Autoencoder has to recover the missing information of the masked input $\vec{x}'$ according to the other parts. This approach can also be thought of as another kind of data augmentation, with the special symbol ?, but dealing with this symbol is a non-trivial task.

As there are several parameterization approaches introduced in Lecture 15, masked Autoencoder works differently with these approaches, as shown below.

1. The first parameterization approach is given by Equation 16.1

$$\begin{aligned} E(\vec{x}) &= A\vec{x} = \sum_{i=1}^{m} x[i]\vec{a}_i, \\ D(\vec{l}) &= B\vec{l}, \end{aligned} \tag{16.1}$$

where the dimension of $\vec{x}$ is m and $A = [\vec{a}_1\, \vec{a}_2 \cdots \vec{a}_m]$. The architecture for this is shown in Figure 16.5. It involves two parameter matrices, A and B. For this approach, we simply change the masked part of $\vec{x}$ into value 0. This approach is like adding a dropout layer before the Encoder layer. So, just like dropout layer, we need to compensate for the loss of size of $\vec{l}$ when we drop some inputs. That is, if we randomly drop inputs by probability $1-p$, we need to multiply the transformation matrix A by $\frac{1}{p}$. The limitation to this approach is that we always use the same transformation matrix A and B, whatever part we mask. However, we may want the model to have different parameters when different part of the input is masked, because intuitively we may want to use different strategies to recover the information of the input when we know which part of the input is missing.

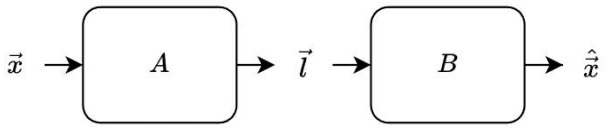


Figure 16.5: The demonstration of the first parameterization.

2. The second parameterization (Figure 16.6) fixes the limitation. It parameterizes the encoder as Equation 16.2

$$E(\vec{x}) = \begin{cases} (B^TB)^{-1}B^T\vec{x}, & \text{if } \dim(l) < \dim(x) \\ & \text{(reduction of parameters using least squares)} \\ B^T(BB^T)^{-1}\vec{x}, & \text{otherwise,} \\ & \text{(reduction of parameters using min norm)} \end{cases} \tag{16.2}$$

where $B = [\vec{b_1}^T\, \vec{b_2}^T \cdots \vec{b_m}^T]^T$ and $B^T = [\vec{b}_1\, \vec{b}_2 \cdots \vec{b}_m]$, and $D(\vec{l}) = B\vec{l}$. The second case of Equation 16.2 yields Equation 16.3

$$D(E(\vec{x})) = BB^T(BB^T)^{-1} = I, \tag{16.3}$$

which is not desirable, and hence this case is not used. This parameterization involves one parameter matrix, B. When masking the input for it, we drop the corresponding rows of B because these parts are not used by the input. Specifically, if there is an input vector $\vec{x} = [x_1, x_2, x_3, x_4, x_5]$ and we mask it as $\vec{x}' = [x_1, 0, x_3, x_4, 0]$, then we can remove the second and the last rows in B and get $\tilde{B}$. And then we have Equation 16.4.

$$E(\vec{x}) = (\tilde{B}^T\tilde{B})^{-1}B^T\vec{x}. \tag{16.4}$$

The advantage of this approach is that when different part of $\vec{x}$ is masked, the model changes accordingly, because $\tilde{B}$ is changed accordingly. For each possibility of masking, the resulting parameter $\tilde{B}$ is different, which intuitively provides different strategies for recovering information from the masked input. Also, in the decoder, $D(\vec{l}) = B\vec{l}$, where $\vec{l}$ is being learned, and we have an actual target $\vec{x}$. So, we get gradients during reconstruction, and not from the encoder, to set the rows of B which were previously dropped.

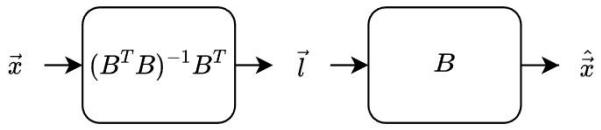


Figure 16.6: The demonstration of the second parameterization.

3. The third parameterization approach considers the encoder as an RNN style network with weight sharing (Figure 16.7). The gradient descent step is given by Equation 16.5.

$$\begin{aligned} \vec{l}_0 &= \vec{0}, \\ \vec{l}_{t+1} &= \vec{l}_t + \eta B^T(\vec{x} - B\vec{l}_t). \end{aligned} \tag{16.5}$$

   Running this block for d steps signifies gradient descent with early stopping, which can learn something meaningful. This is because, if B is such that most of its singular values are in the direction of the underlying pattern, and the other singular values are small, taking a few steps of gradient descent along that direction will project onto the pattern, with very little energy going in other directions. This is effectively denoising despite projecting onto a larger space. Hence, it is expected that deeper architectures for Autoencoders can learn interesting inductive biases.

   In this approach, we first compute $\vec{x} - B\vec{l}_t$. But some of the inputs are masked, and hence we need to first figure out what $? - *$ means ($*$ denotes some real number or vector). Two natural choices for this are either to use ? or 0, which give the same final result. This is because, in the next step, when we multiply ηB^T to $\vec{x} - B\vec{l}_t$, the multiplication of some quantity with 0 will yield 0, as will the multiplication with ? following the strategy used in the second parameterization. Hence $? - *$ is clamped as 0, which makes the residual 0. Treating ? directly as 0 is not a valid choice as $? - *$ will then be $-*$, denoting a residual. In this case, the model will try to update the parameters to get rid of the residual, which does not make sense since nothing should be present at the masked locations to drive the update at the encoder. Decoder is designed to deal with this issue, while the main task of the encoder is to find a good latent representation.

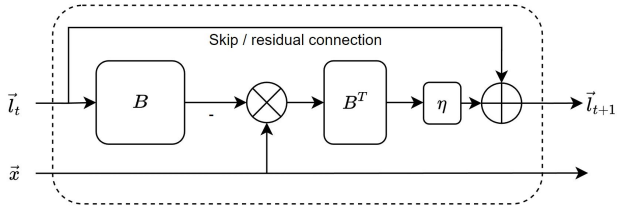


Figure 16.7: The demonstration of the parameterization of RNN style network with weight sharing.

The issue with the parameterizations discussed above is that the rows of the matrices corresponding to the masked entries in the encoder will have no update.

For each input, some probability metric is used to determine the position of the masks. The final goal here is to learn the underlying pattern, so that the model can learn to complete things. However, due to this masking, some of the important features may not get recovered, if those masked features cannot be extracted from the other unmasked features. We can expect that the different masks help in learning the implicit patterns on average.

Binary flags, instead of ?, can also be passed to the network to denote masking. The implementation depends on whether we want the model to learn how to use the binary flag or do some pre-processing with the flag.

Masked autoencoders can be seen as another way of learning low dimensional structure, apart from denoising autoencoders. The models tend to generalize well. For example, the task of adding noise in language models is difficult, and hence masking can be used. Even though blanks can be in a critical position, the model can learn something about the pattern so that some unlikely predictions can be discarded.

Even though with low dimensional latent space, there is no fear of learning the identity mapping, sometimes we want to have the freedom of learning larger l. In deep learning, we have seen many times that if we allow more parameters in the model, we tend to have better training, and achieve better performance. If l is big, for parameterization one, we have more rows of A with more parameters for initialization, and hence more room for a lucky guess to get a good row of A. Moreover, it is possible to have a low dimensional structure embedded in a high dimensional space that is still being purified in some way. The pure invertibility perspective of a matrix is not that useful since in matrix inversion, no information is lost. But the singular value perspective of a matrix helps to see the purification process in some way. It does not allow everything to go in the same way, but might shrink things in some direction and expand things in other directions.

16.3 Aside: Beam Search

A *language model* is a sequential model with access to some kind of memory. For RNN or LSTM, memory of the past is constrained to be the state, but attention mechanism is also used for language models. The general picture of a language model is shown in Figure 16.8.

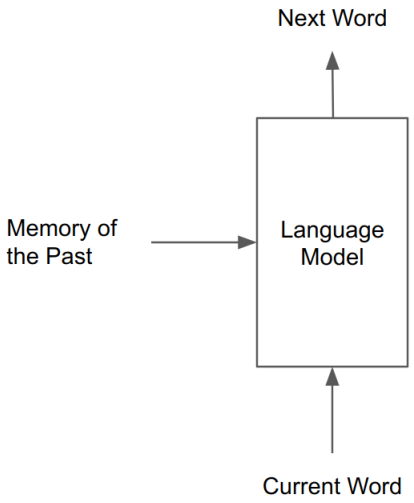


Figure 16.8: General Picture of a Language Model.

An example of self-supervised training of a language model is shown in Figure 16.9.

Words (represented using some vector embedding) can be considered as a discrete set of objects, and hence this self-supervision task is kind of a generalized classification problem. The output of the model at each step is not a single object, but a set of learned scores or probabilities corresponding to the different words. Self-supervised training over millions of sentences can make a model learn to put high scores on words that appear frequently, and lower the scores of words that do not appear. For example, if we have a sentence *"The man ate the ___"*, the probability of the next word being *carrot* should be much more compared to the word *of*.

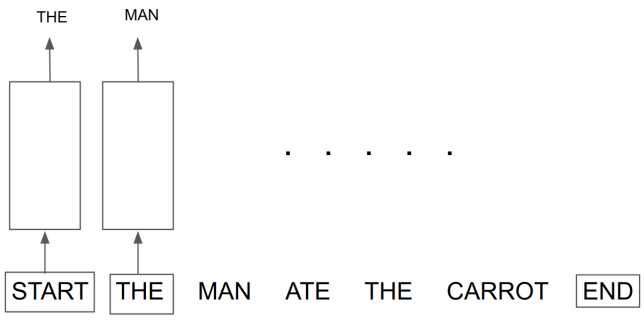


Figure 16.9: Self-supervised training of a Language Model where at each step, the model is predicting the next word conditioned on the previous words.

We can use the trained model on some task, for example, to generate a sentence. Filling in just one blank using the model is not a difficult task, as the model has already been trained on that. So, we can take the word corresponding to the highest output probability, or use random sampling. For the sentence *"The man ate the ___"*, if we want a single output, we can just get the output corresponding to the highest probability (suppose *carrot* in this case). But for a variable length output, different outcomes such as *"The man ate the banana."*, and *"The man ate the banana with the fork."* are possible. This task of completing a sentence is challenging. Exhaustively searching the tree of all $O(M^T)$ possible sequences, where M is the size of the vocabulary, and T is the maximum length of a sequence, to determine the true most likely sequence can be intractable.

The naive strategy is to pick the highest probability for the next word in a greedy manner, then add the word to the resulting sequence, and repeat the process using the updated sequence. But this may lead to a dead end, such as loops. Backtracking approach is not commonly used for language models, since it is difficult to interpret which situation is bad, so that the model can return to the last stable state. The data is always collected from a dataset containing good examples (i.e., from the pattern), and the model learns on that dataset. We do not have data that does not come from the pattern, or comes from almost the pattern.

Hence, a better strategy is not to commit to one particular thing, but instead keep a bag of current best possibilities in order to generate the most likely next sequence based on high likelihood. This is the main idea behind *Beam Search*. At each step, keep k best-so-far continuations based on the output probabilities. For each of the k continuations, predict the next output, which will generate k^2 possibilities, pick the k best ones, and repeat the process. The pseudocode of this strategy (taken from question 5 of hw6) is given in Algorithm 1:

Algorithm 1 Beam Search

for each time step t **do**
  for each hypothesis $y_{1:t-1,i}$ that are being tracked **do**
    find the top k tokens $y_{t,i,1}, \cdots, y_{t,i,k}$
  end for
  sort the resulting k^2 length t sequences based on the total log-probability
  store the top k sequences
  advance each hypothesis to time step t + 1
end for

A simple visualization of this strategy as a tree search problem is shown in Figure 16.10, where $k = 2$. The leftmost node can be considered as the START token, or one of the results so far. At each next step, based on the probabilities, 2 (red nodes) out of the 4 most probable nodes (red and blue nodes) are chosen for the

most likely paths. For a more concrete example, please refer to question 5 of hw6.

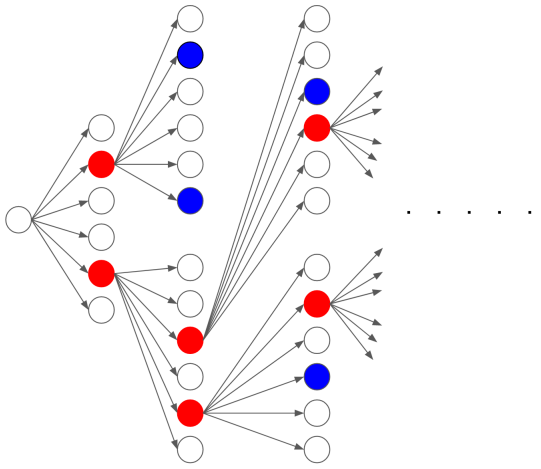


Figure 16.10: Simple visualization of the Beam Search strategy as a tree search problem ($k = 2$).

However, multiplication of the probabilities along the predicted sequence might create biases towards very short sentences. If appropriate normalization is used to tackle the issue, biases might be created for very long sentences. There are numerous tricks involved to get this strategy working properly.

16.4 What we wish this lecture also had to make things clearer?

1. It was briefly mentioned in the lecture that in the context of language models, adding noise is a difficult task, for which masking is used. Even though we believe that the topic will be covered in more details in the subsequent lectures, it would have been really helpful if some specific examples of such models using masking techniques were provided.

2. For all the parameterizations, the bottleneck case where $\dim(l) < \dim(x)$ does not have the issue of learning the trivial identity map. This issue might arise if $\dim(l) \geq \dim(x)$, and under this condition, learning the identity map is always the case for parameterization 2. However, it was mentioned in the lecture that sometimes, having larger latent space can be beneficial for learning better models (for the other parameterization cases). The manner in which the different relative latent space dimensions may be useful (or harmful), can be summarized for clear understanding. Also, in the case where $\dim(l) \geq \dim(x)$, the importance of dense and sparse representations can be mentioned to provide a complete picture. More figures might be included for explaining the different autoencoder ideas.

3. The overview of Beam Search was briefly covered in the lecture. But, some of the important concepts such as stopping criterion, or normalization techniques to deal with different lengths of the sequences, or multiplication of probabilities along the sequence were not covered. It would have been helpful to get some information on these concepts, along with a concrete example.

EECS 182/282A Lecture 17: Transformer Models

Instructor: Anant Sahai
Scribe: Chengyuan Li, Jackson Gao

November 13, 2022

1 Introduction

We have spent a lot of time understanding the idea of self-supervision where we only have access to unlabeled data (just like in unsupervised learning) and use data themselves to generate labels. One example that proved such learning scheme useful is doing **PCA-style dimensionality reduction "purification"** in an autoencoder style with self-supervision. Apart from PCA, **k-means style clustering** where we assign unlabeled data to groups is another kind of traditional unsupervised learning. In the first part of the lecture, we attempt to make k-means clustering compatible with gradient descent. Then, we will connect to the idea of attention and introduce transformer models.

2 Classic Lloyd's Algorithm for k-means

Given inputs $\{\vec{x_i}\}$, the following algorithm outputs cluster centers $\{\vec{r_j}\}$ such that $\vec{x_i}$ belongs to cluster j described by its center $\vec{r_j}$:

1. Initialize k cluster centers $\vec{r_1}, \vec{r_2}, \ldots, \vec{r_k}$ (e.g. randomly pick k input points)
2. Assign each data point $\vec{x_i}$ to a cluster j based on the closest $\vec{r_j}$
3. Update $\vec{r_j}$ to be the mean of cluster j
4. Repeat steps 2 & 3 until converged

When the algorithm sees a new point, it assigns the point to cluster j based on the closest $\vec{r_j}$. This is important because we want to use what we have learned on new data. Now we attempt to make the algorithm "trainable" with gradient descent.

2.1 SGD Approach (First Attempt)

Suppose we have initialized the cluster centers, we want to use our data points one by one to move the centers to be in a good place. This is different from the Lloyd's Algorithm in which we need to go through all data points in each update step.

1. Pick a point $\vec{x_i}$ (viewed as a mini-batch)
2. Find $\vec{r} = \text{argmin}_{\vec{r_j}} ||\vec{x_i} - \vec{r_j}||$
3. Minimize $||\vec{x_i} - \vec{r}||^2$ over $\vec{r}$ by taking a gradient step: $\vec{r} = \vec{r} + \eta \cdot 2(\vec{x_i} - \vec{r})$
4. Repeat steps 1 & 2 & 3 until converged

We can consider step 3 as an analog of step 3 in Lloyd's algorithm since setting $\vec{r_j}$ to be the mean of cluster j essentially minimizes the squared error between $\vec{r_j}$ and points in cluster j. However, step 2 is problematic because argmin is not differentiable. We want to replace argmin with an operation that is differentiable.

2.2 Softmax Idea

We first investigate the behavior of the following quantity:

$$e^{-\gamma||\vec{x_i}-\vec{r_j}||^2}. \tag{1}$$

Observe that when $\gamma > 0$, (1) is close to 0 if $\vec{r_j}$ is far from $\vec{x_i}$ and is close to 1 if $\vec{r_j}$ is close to $\vec{x_i}$. By scaling this quantity to be between 0 and 1, we can then apply normalization to turn them into probabilities and take expectations of interest.

Let $\alpha_j = \frac{e^{-\gamma||\vec{x_i}-\vec{r_j}||^2}}{\sum_l e^{-\gamma||\vec{x_i}-\vec{r_l}||^2}}$, we can view α_j as the amount we want $\vec{x_i}$ to pull $\vec{r_j}$. If $\vec{x_i}$ is far from $\vec{r_j}$, α_j is close to 0, and we do not want $\vec{x_i}$ to pull $\vec{r_j}$. If $\vec{x_i}$ is close to $\vec{r_j}$, α_j is close to 1, and we want $\vec{x_i}$ to pull $\vec{r_j}$. Therefore, minimizing the following quantity by taking gradient descent steps achieves our goal of getting $\vec{r_j}$ updated:

$$\sum_j \alpha_j ||\vec{x_i} - \vec{r_j}||^2. \tag{2}$$

Now we have everything ready for using gradient descent on k-means. Let's understand it by drawing the computational graph:

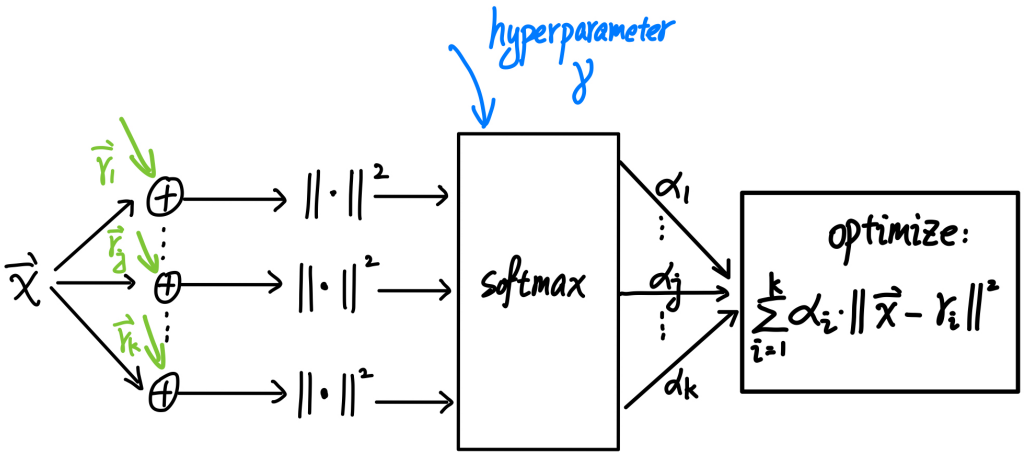


Figure 1: Computational graph (k-means with softmax idea)

Starting from the left, we first take the difference between input $\vec{x}$ and network parameters $\vec{r_1}, \vec{r_2}, \ldots, \vec{r_k}$, then take the squared norm of the differences and feed them into the softmax operator (with hyperparameter γ), which outputs α for the computation of our objective function (2). From there, we use gradient descent to backprop. (Note: In deep learning, the hyperparameter γ is traditionally referred to as "temperature" due to its analogy to statistical mechanics.)

Let's compare the softmax idea with our first attempt of using SGD. In our first attempt, we pick a point $\vec{x_i}$ and use it to update the closest $\vec{r_j}$ in each iteration, and the amount we move $\vec{r_j}$ is based on the learning rate η and the distance between $\vec{x_i}$ and $\vec{r_j}$. In the softmax idea, we do similar things in a batch, updating all $\vec{r_j}$ at once, but we control how much each $\vec{r_j}$ is updated by α_j (i.e. the softmax version of the distance between $\vec{x_i}$ and $\vec{r_j}$). Using the softmax idea, we successfully eliminated the problem we had in SGD while accomplishing our goal.

2.3 Goofy Alternative

Suppose we have the ideal case where clusters are widely separated as shown in Figure 2. $\vec{r_1}$, $\vec{r_2}$, $\vec{r_3}$ represent the center of each cluster. Let's consider a point $\vec{x_i}$ (circled, in the upper right cluster). With the softmax idea (1) where γ is relatively large, we have α_1, α_3 close to 0, and α_2 close to 1. This motivates us to think about an alternative approach for gradient descent.

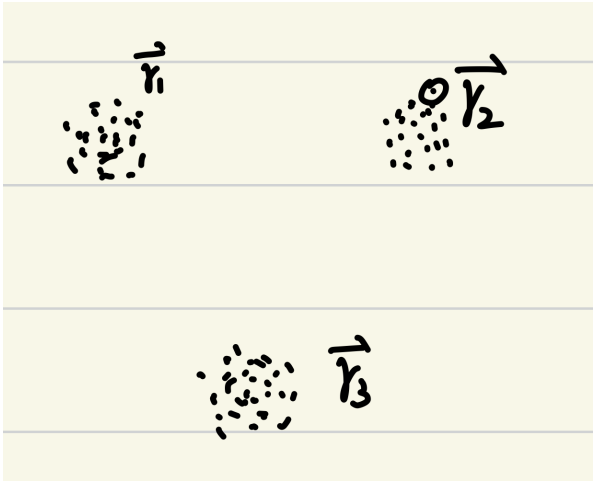


Figure 2: Motivation of goofy alternative

Instead of optimizing (2), we treat α_j as the probability that $\vec{x_i}$ is the closest to $\vec{r_j}$ and compute the expectation $\vec{\hat{r}} = \sum \alpha_j \vec{r_j}$ of where the closest cluster center is (to $\vec{x_i}$). We then minimize the distance $||\vec{\hat{r}} - \vec{x_i}||^2$ by gradient descent, which looks more like the standard gradient descent objective.

The goofy approach is interesting due to its connection with the attention mechanism. In attention, we have $\vec{k}$ (key), $\vec{q}$ (query), and $\vec{v}$ (value). We take the average of the values based on similarities to the keys that have gone through appropriate softmax. Here in $\vec{\hat{r}} = \sum \alpha_j \vec{r_j}$, we can treat $\vec{r_j}$ as the value $\vec{v}$ and α_j as the key $\vec{k}$ that have gone through softmax.

Again, let's understand it by drawing out the computational graph:

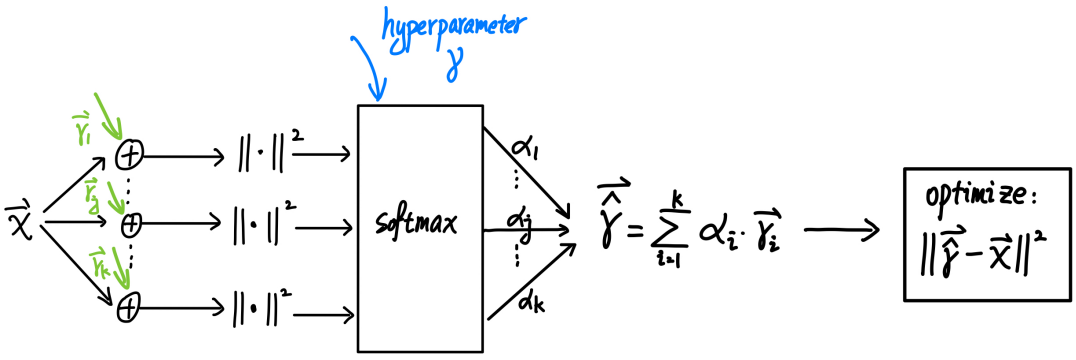


Figure 3: Computational graph (k-means with goofy alternative)

Comparing with the previous computational graph, we see that the only difference is the objective function to optimize at the end. Instead of directly optimize (2), we first compute the expectation $\vec{\hat{r}}$, then optimize the distance $||\vec{\hat{r}} - \vec{x}||^2$.

3 Transformer Models

Transformer models are first introduced in the paper *Attention is all you need*[1]. We will introduce two key ideas of transformers.

3.1 Key Idea 1

In the previous lecture, we talked about RNN which uses states to memorize knowledge. Attention mechanism can also be viewed as a kind of memory, which is a soft approximate hash table. Here we try to only use attention to "write things down" instead of using states as internal memory. Now, what we can do is to write memory and query memory. Then it comes to the first key idea of transformers: "I don't need internal memory if I can write notes to myself". What happens is we drop recurrence from the RNN approach, but keep weight sharing (across time).

Based on that, we can intuitively consider transformer models as GNN, which share weights between different nodes. Nodes in transformer models that pass across time can also be viewed as a fully connected graph. These basic ideas are similar.

This idea relies on the attention mechanism. We have introduced two ways of understanding attention:

1. Soft approximate hash table
2. Queryable softmax pooling

In the field of computer science, we can consider attention as a hash table; but in deep learning style, it is a pooling layer combined with softmax.

3.1.1 Recall attention

From the previous lecture, we have three vectors in attention: $\vec{k}, \vec{q}, \vec{v}$, and we have a hash table containing pairs of $\vec{k_i}$ and $\vec{v_i}$.

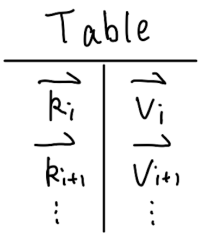


Figure 4: Attention hash table

When we apply a query $\vec{q_t}$, we can calculate the output as follows:

$$e_{i,t} = \frac{< \vec{q_t}, \vec{k_i} >}{\sqrt{d}} \tag{3}$$

$$\alpha_{i,t} = \frac{e^{e_{i,t}}}{\sum_j e^{e_{j,t}}} \tag{4}$$

$$output = \sum_i \alpha_{i,t} \vec{v_i} \tag{5}$$

In equation (3), d represents the dimension of the key. This process is what we called queryable softmax pooling, and equation (4) is doing softmax. Besides, if the process generating q is also populating the table with k, v, we call it self-attention. Basically, if you are writing to the hash table at the same time, it is self-attention; otherwise, if you only read the value from others, it is not. Just as Figure 5 shows, if x_{i+1}'s key also query its own key, then it is self-attention.

Aside: The product of $\vec{q_t}$ and $\vec{k_i}$ can be negative or positive, which represents the direction of the resulting vector. In transformer models, the negative value means they are quite different, but in other models it may have different meanings.

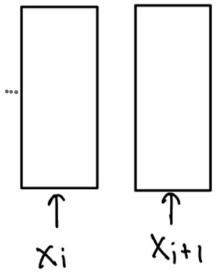


Figure 5: Self-attention

3.2 Key Idea 2

It is useful and important to use multiple channels in conv-nets, and we also want to use multiple channels in transformer models. We call it multi-headed attention in transformer models, and it has many parallel queryable softmax pooling.

If we think about max pooling in conv-nets, different channels may select different features. Now, when we generate different queries, it may comes up with different values.

Note: This lecture ended early as the scheduled fire alarm drill occurred.

References

[1] Ashish Vaswani et al. "Attention Is All You Need". In: (2017). DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

EECS 182/282A Deep Neural Networks Lecture 18 - 10/25/2022

Lecture 18: Transformers

Instructor: Anant Sahai *Scribe: Jianzhi Wang, Jason Yang*

1 Recap

1.1 Differences between RNN and Transformer

In both approaches, we have sequential inputs. In the RNN approach, we try to capture all the information from one state (the context token). In the transformer approach, we use the attention mechanism to learn the dependencies across the input sequence. This is because the attention mechanism allows us to query to multiple positions.

1.2 (Single-headed) Attention Block

It is helpful to think of the attention block as a "queryable softmax pooling" or soft approximation of a hash table (it contains a set of key-value pairs, where you can pass a query vector through it). You can also think of it as a differentiable black box.

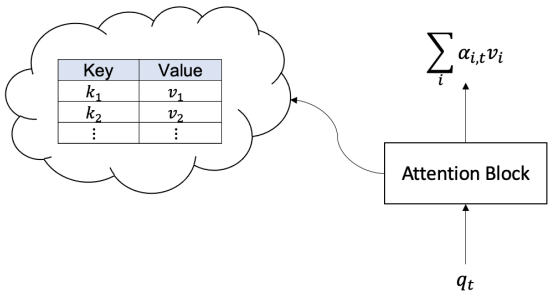


Figure 1: A single attention block

For a query, the output is approximately the value that corresponds to the nearest key, measured by inner product. Here, d is the common dimension of the key and query vector.

$$\text{sim}(q_t, k_i) = \frac{\langle q_t, k_i \rangle}{\sqrt{d}}$$

Therefore, the output is a linear combination of values: $\Sigma_i \alpha_{i,t} v_i$ where the $\alpha_{i,t} \in [0, 1]$ is obtained after a softmax operation on the similarity values. Recall that softmax allows gradient to flow to W_k, W_v and W_q. On the other hand, if argmax is used, generally there will be no gradient flows to W_k and W_q for keys that are not selected as the argmax.

1.3 Parallels with CNN

The attention block can be seen as the counterpart of a convolution layer in CNNs. In CNNs, the convolution layer plays the role of combining information from local neighbours. Here, the attention block is also aggregating information. The difference is that the attention block, unlike a convolution layer, does not have learnable parameters - they just respond to queries. We will bridge this gap in section 3.

The attention block is the counterpart to the convolution operation (aggregation of information). The table structure is similar to the concept of receptive field in CNN. However, in attention mechanism, an increase in receptive field refers to a larger number of positions in the sequence that the attention block attends to, rather than in terms of locality as in CNNs.

2 Multi-headed Attention

2.1 Introduction

Now that we know what the basic attention block looks like, we want to have the ability to attend to many things at once. We can do so with many attention heads, each equipped with its own key, value and query functions. This is the concept of multi-headed attention. Given an input token, we can query through multiple attention heads and concatenate the outputs together.

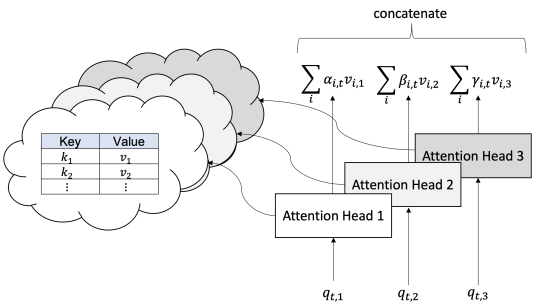


Figure 2: Multiheaded attention

Note: to ensure that the dimensions match with that of residual connection later on, we must make the dimensions of the queries smaller. This is so that when we concatenate the attention scores, we get back the same dimension as the input.

2.2 Parallels with CNN

Multi-headed attention is the counterpart of channels in CNNs. There can be many channels, but each attention head attends to its own channel. This is exactly like the depthwise convolution in ConvNext, where the convolution takes place per channel. The difference is that in CNNs, each channel is a real number, whereas here it is a vector.

Also, across different heads, the only thing that distinguishes them is the random initialization of their W_k, W_q, W_v. This is the same for different filters in CNNs, where the only thing that distinguishes them is

the random initialization the weights in each filter. We hope that this broken symmetry leads to different attention heads attending to different things.

3 Motivation of the Transformer Architecture

We now present the Transformer architecture with a step-by-step walkthrough.

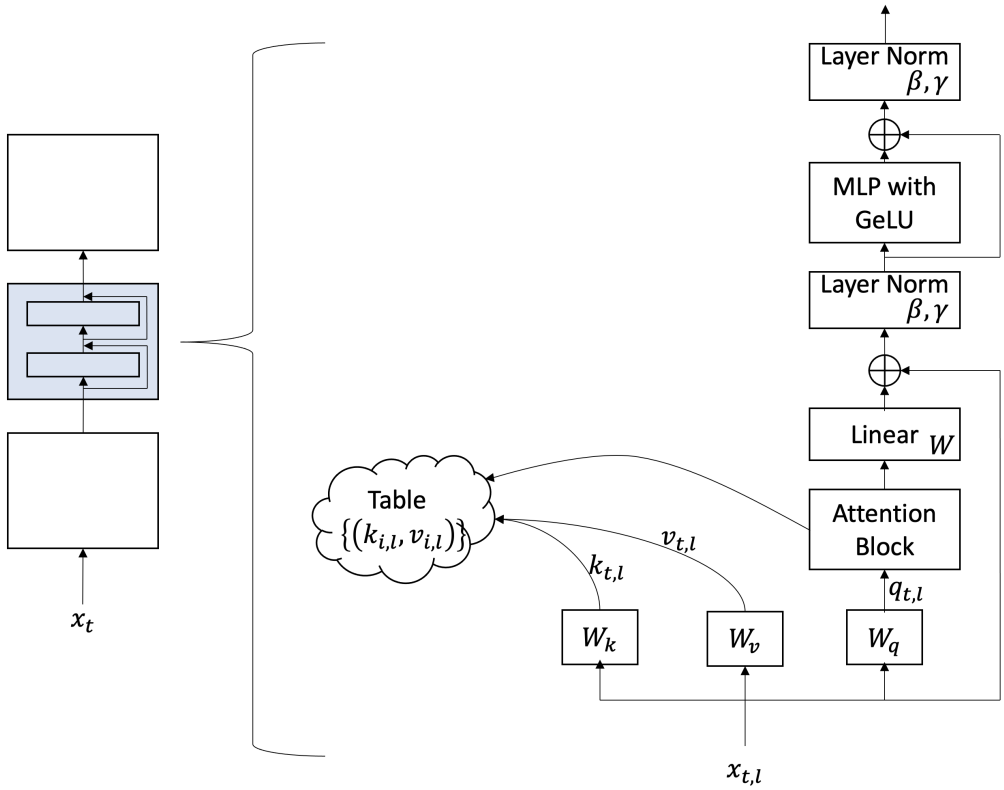


Figure 3: One layer of the Transformer architecture

1. Firstly, we receive input token $x_{t,l}$ from the previous layer. Here, t denotes time, or the position of this token in the sequence. l denotes the layer.

2. Using $x_{t,l}$, we create key vector $k_{t,l}$, query vector $q_{t,l}$ and value vector $v_{t,l}$ by applying linear layers with weights W_k, W_v and W_q respectively.

3. Store the key vector $k_{t,l}$ and value vector $v_{t,l}$ into the table. The table contains all key-value pairs generated by input sequence at layer l.

4. Pass the query vector $q_{t,l}$ as an input to the attention block (which has access to the table). Note: the attention block has a nonlinearity inside - the softmax that outputs the attention scores α.

5. The result of the attention block goes through a linear layer W. This is because the result is a linear combination of the values, and it would be more flexible to transform it.

6. This is now combined with $x_{t,l}$ via a skip connection, motivated by the ResNet architecture.
7. Now, it goes through a Layer Norm. Note that the skip connection in the previous step fixes the problem of vanishing gradients. Adding a LN now addresses the issue of exploding gradient, since of goal of Layer Norms is to control the behaviour of the output via standardization. Recall that LNs also have two learnable parameters β, γ to adjust the mean from 0 and variance from 1 so we will not lose expressive power.
8. We now add a MLP (with a nonlinearity such as GeLU) to increase the expressive power. Without it, the nonlinearity in the attention block is too weak and can be bypassed by the skip connection.
9. We use the same strategy to add a skip connection with a LN.

Previously, we have gathered a lot of ideas from the ConvNet and ResNet architectures. We now present both the ResNet and ConvNext architectures and match the ideas from these two architectures to their respective Transformer counterparts.

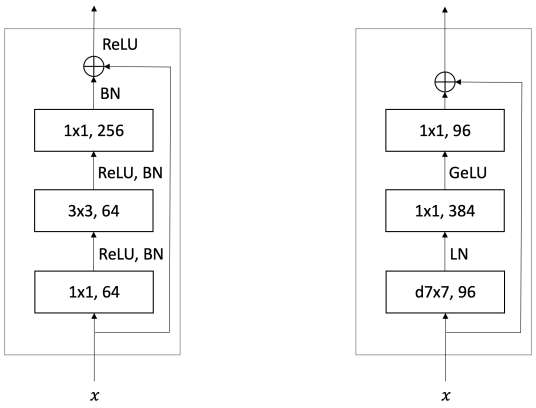


Figure 4: Classic ResNet (left) and ConvNext (right)

Similarities with ResNet:

- **Residual connections**: this was inspired by the ResNet architecture, where we want to increase complexity by adding depth and stacking multiple blocks while also ensuring a residual connection. This ensures good flow of gradients.
- **1x1 64**: Corresponds to W_k, W_q, W_v. W_k, W_v, W_q are like the filter weights and are shared in layer l across all time t, similar to how filter weights are shared in a ConvNet. All look locally at only that position t, which is similar to the 1x1 kernel.
- **3x3 64**: Corresponds to W_k, W_q, W_v and the attention block.
- **1x1 256**: Corresponds to W with the MLP + nonlinearity.

Similarities with ConvNext:

- **d7x7, 96**: Its counterpart is the entire part of the Transformer prior to the MLP, as noted in the section on Multi-headed Attention. We also see similarity in the applied LN.
- **GeLU, 1x1 96**: Exactly like the MLP + GeLU part of the Transformer.

4 Masking in Attention

Motivation: To process sequential input one token at a time, we can just populate the table one at a time in the transformer model. However, what if the entire input is available to us and we want to take advantage of the parallelism? If we naively populate the table with all key-value vectors, that would raise the issue of "peeking into the future", where we should not be able to attend to keys and values generated by future tokens.

The main idea is that we should not perform inner products with the key-value pairs from the future. This can be resolved in two ways. The first way is to surgically set the resulting value from inner products with future values to be $-\infty$ i.e. $\langle q_t, k_{t+k} \rangle = -\infty \; \forall k > 0$. This ensures that softmax assigns them a weight of 0, and their future values will never be used. The second way is to perform treatment on the αs instead. Let softmax run as per normal, then set $\alpha_{t+k} = 0 \; \forall k > 0$, then normalize the resulting αs.

5 Cross Attention

The only difference between cross attention and self attention is the tokens used to populate the table. Let's examine the encoder-decoder models.

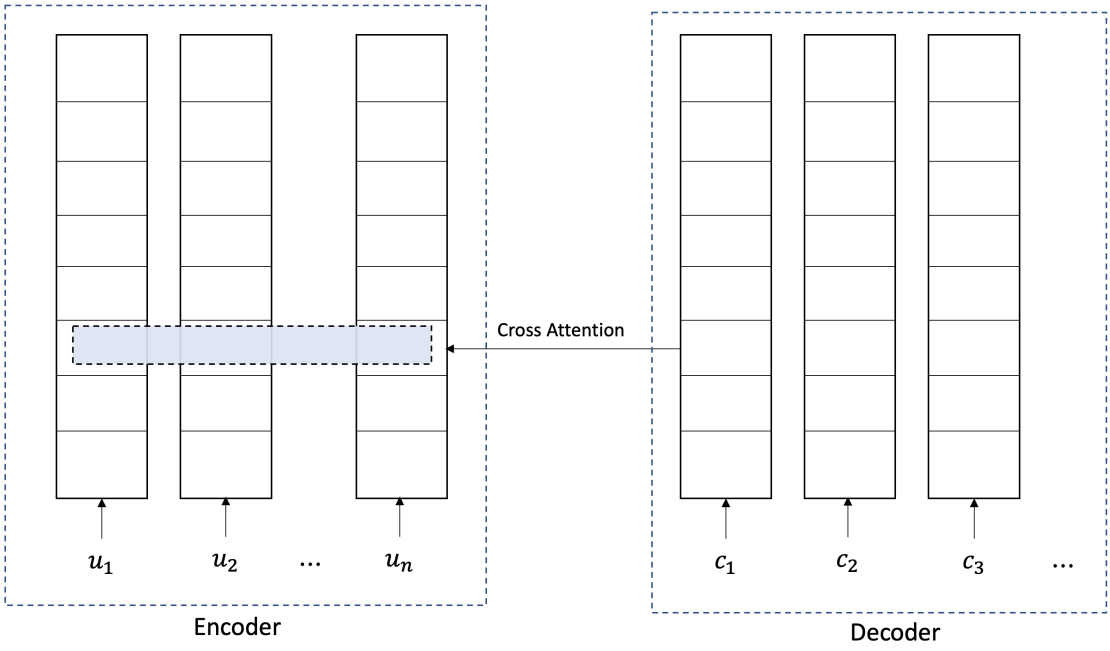


Figure 5: Decoder layers attend to key-value pairs generated by encoder layers via cross attention

The encoder part of the transformer takes in an input sequence $(u_i)_{i=1}^n$. Each layer will have its own table of key-value pairs. The decoder takes in a context sequence $(c_i)_{i=1}^m$. The attention blocks in decoder attends to the keys and values generated by the same layer in the encoder. In summary, the queries come from the decoder, and the key-value pairs are supplied by the encoder. It is also possible to use a combination of cross attention and self attention in the architecture.

6 Summary

The conception of the transformer model is not really motivated from a biological viewpoint, but rather from an engineering perspective. It is implementation friendly and good for parallelization.

However, one downside is the inherent quadratic time complexity of the attention blocks. Each attention block has to look through the entire set of key-value pairs, whose length is equal to the length of the entire input sequence. As an example, if there are N queries and M key-value pairs in table, the computation takes $O(MND)$, where D is the common dimension between the query and key/value vectors. This is because there are MN dot products, and each dot product takes $O(D)$ time. In self-attention, $M = N$, so the time complexity reduces to $O(N^2D)$, which is quadratic in N. This presents a challenge in scaling transformer models.

One way to circumvent this is via an approximate softmax attention through a kernel perspective (for example, we can approximate the softmax operation via $\text{sim}(q_i, k_j) = \phi(q_i)^T\phi(k_j)$). This approximation can effectively remove the quadratic factor and allow Transformers to scale.

Lastly, Transformers can be seen as a generalization of ConvNets. Therefore, it is a more flexible model. The weak inductive bias inherent to transformers implies that we need more training data to make learned patterns visible. This motivates the next topic on self-supervision and pre-training, both of which are absolutely critical for practical use of the Transformer model.

Lecture 19: 10/27 (Thursday)

Lecturer: Prof. Anant Sahai

Scribes: Nabeel Hingun

1 Agenda

1.1 Attention, ResNet Style Blocks with MLPs and LN

Last time, we talked about different ingredients that go into building a transformer [3]. The basic core ingredient was the attention mechanism, which is just a query-able softmax pooling. We also talked about the multi-headed variant where we can execute many different queries at once into many different tables.

There are two kinds of attention: self-attention and cross-attention. The difference lies in whether we are querying the same table we are sticking stuff into or a different table. Attention by itself has no learn-able parameters associated with it so we need to have learn-able parameters making the queries, keys and values. On top of that, we have ResNet-style blocks to combine something designed to use the attention mechanism with standard building blocks like multi-layer perceptrons and layer normalization to make something we can stack and make deep.

1.2 Position Encoding

The input to the transformer has no sense of ordering. To have some way of imposing order, we use position encoding to encode the positions of input tokens. The one we talked about last time was inspired by the hands of the clock, i.e, sines and cosines. This method is friendly to matrix multiplication and has the nice analogy to advancing the hands of the clock or moving back the hands of the clock. It gives us an absolute encoding of position which is friendly to relative position querying.

2 Variant of Position Encoding

There are many variations of position encoding. In particular, there is a distinction between the one we talked about in last lecture and the one that is used in practice. Last time, we talked about implementing position encoding as concatenating our input to the position encoding (fig. 19.1). Here, we can think of the input to the transformer as having two parts: (1) the data and (2) where that data was in the sequence.

While this 'concatenation' method can be done in practice and is very logical, a more common approach is to have the input and position added together (fig. 19.1). The question that arises is why this even works as it seems less natural.

First, recall that the vector sum is happening in a very high dimensional space (512, 1024, ... dimensions). In such high dimensional spaces, the position encodings are occupying a very limited set of the space which leaves a lot of space for the input data. Then, suppose we want to query for the position only. When we take the inner product, by linearity, we get (1) the inner product of the position oriented query and the input, added to (2) the inner product of the position oriented query

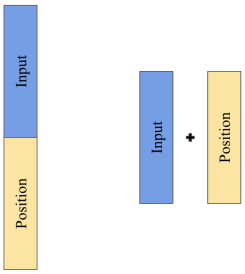


Figure 19.1: Position Embedding: Concatenation (Left) v/s Addition (Right)

with the position encoding. If the position encoding has a high correlation with the query, then (2) will be high whereas the position oriented query and the input will be something completely different. Therefore, the inner product has the ability to be able to pick up positions vs things in the input. Of course, there will be some interference with the query but not that much and so we expect that the sum can still make position encoding work. Thus, in high-dimensional space, adding is also fine for the inner product to be able to pick up positions versus things in the input. Addition is used in practice and usually has better performance.

Question: We want to use complex numbers because matrix multiplications are rotations (of the clock) but we are not actually using complex numbers. By using only sines and cosines, so do we still have some property of rotation?

Answer: The idea of the complex numbers rotating around is used as inspiration for the hands of the clock since we know complex multiplication can result in rotating these relative positions. Yet, if we represent complex numbers using a vector of two numbers, then we can think of complex multiplication as a linear operation represented by a matrix. This means that with matrix multiplication, it is easy to express a rotation, so the same property we had with the multiplication of complex numbers, we can basically inherit from the vector representation using sines and cosines.

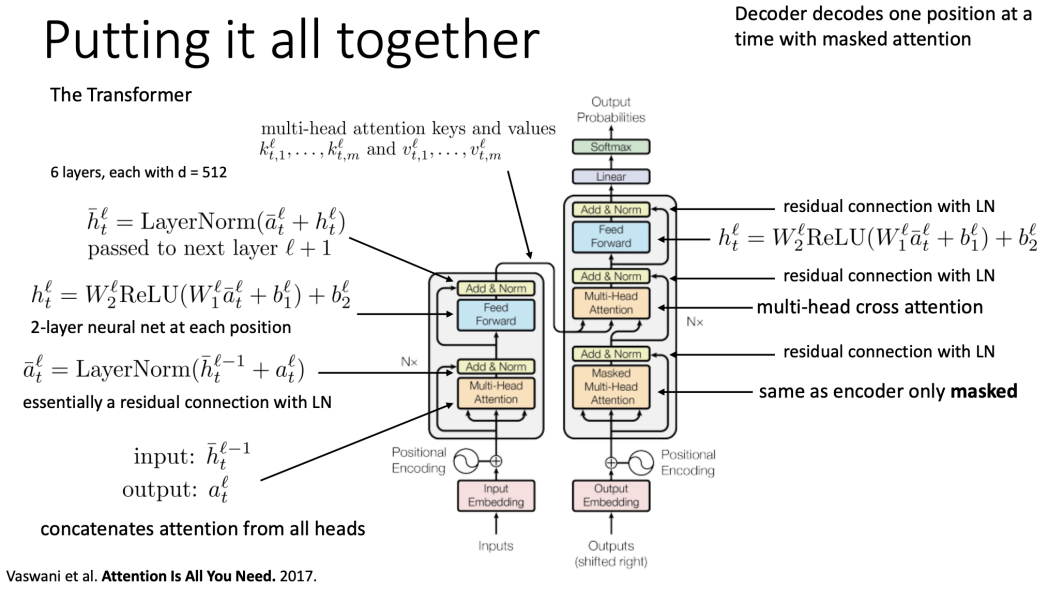


Figure 19.2: On the left, we have an encoder with N_x encoder blocks. The encoder is usually pre-trained with some specific task. On the right, we have a decoder made up of N_x decoder blocks. [1]

Question: If you want to rotate the hands of a clock by some angle, say 10 degrees, then we have to compute the sines and cosines of 10 degrees to populate the transformation matrix. Yet, if we look at the linear and non-linear operations in a transformer that define the entire path of the query or keys, we don't see a sine or cosine as a non-linearity. So how is it that we are actually able to move by 10 degrees?

Answer: The queries are the result of a learned transformations. We don't need to explicitly compute the sine and cosine computation at runtime. The transformation just has to be learn-able during training. Whatever the sines and cosines we actually need will come out during training. It is easy to learn a rotation as a transformation.

Follow Up Question: Does that require a lot of weights if we want to learn all the relevant angles?

Answer: Yes, there are potentially many different relevant angles we need to learn and so indeed, we would have many weights. The hope is that the model has enough expressive power to learn the transformations we need. In fact, transformers are very large models with high memory cost. For a 1000-length sequence, each with hidden dimension 1000, we can will need 1 million parameters. In addition, to compute softmax, we need $O(10^9)$ number of multiplications and additions.

3 Transformer Components

3.1 Attention

The transformer (fig. 19.2) uses a combination of self-attention, cross attention, layer norms and MLPs. But how is that all working together and why do we need these different components? Traditionally, in a decoder block, we start with a self-attention layer (bottom right orange block in fig. 19.2) which is looking at the input and other things in the sequence that might be relevant to understand what it should do next. Then we have cross-attention (center right orange block in fig. 19.2). The cross-attention accesses some table with key-value pairs. Typically, the key-values pairs come from the encoder. Since we have to interpret the embedding of the input token in some way, there is a weight key matrix that generates the keys and another weight matrix that generates the values. These matrices, W_k and W_v have learn-able weights. Gradients from decoder shape W_k and W_v because it is related to the task the decoder is doing (interpret the encoder embedding). For example, in fig. 19.2, W_k and W_v are multiplied by the last output of the encoder to produce $k^l_{t,1}, ..., k^l_{t,m}$ and $v^l_{t,1}, ..., v^l_{t,m}$.

3.2 MLP

Finally, the output is passed through some MLP (fig. 19.2). We need the latter because we need a non-linearity stronger than the non-linearities involved in attention.

3.3 Layer Normalization

There is also a question of why we need layer normalization and what is it doing. Recall that the attention mechanism is computing an inner product. From the query's point of view, the inner product is trying to say, of all the different keys, which is the one which is the most in this direction. Then we can see that layer norm is useful because it gives the query a particular direction.

Question: Do we need to normalize the keys too?

Answer: Not really. The same query is applied to a bunch of keys but for some keys you want them to be able to say, "i'm in this direction, yes, but i'm not that strong. If there is another direction that is stronger, then pick that on". Whereas the query is really picking a direction. If we change the norm of the query, we would still end up with the same ordered sequence of winners of the keys but all that would change is what happens with the softmax, in terms of how well it does the normalization. But that's not what we want so we don't want to impose a normalization constraint on the keys.

3.4 Query Standardization

Input standardization involves three parts. Take data,

- subtract mean
- divide by standard deviation
- possibly shift the mean and standard deviation by something that's learn-able.

These steps involve two sides: normalizing the size of everything and moving the means around. When it comes to preventing exploding gradients or vanishing gradients, the means are not relevant whereas sizes are. In the context of a transformer, when it comes to what we want from the query, we want it to have a norm that is something reasonable and doesn't change a lot between queries. Then, the query doesn't need biases and only needs scaling. Hence, one of the modifications to the Transformers that is often done is to remove the bias and stick to the scaling.

Question: When we are looking at a particular encoder layer, we have a self attention block inside which is looking into a table. Are the contents of the table the same for different input positions?

Answer: Yes. They are only different if we enforce some sort of sequentiality. On the decoder side, the sequentiality is often required.

We usually have two different design choices when it comes to the inputs of the attention layers. For instance, in an arbitrary decoder, which output of the encoder should the attention layer of block 7 (of the decoder) be looking at? One option would be to have every block look at the last output of the encoder. So if the encoder is 20 levels deep, then it is looking at the output of level 20. Every single layer of the decoder here is looking at the output of the last layer, layer 20. This follows the auto-encoder spirit where the output of the encoder is a distilled version of the output. Typically, this is what is done.

Another option is in the style of a U-net where we can look over at different levels of fineness of the encoder. So we can imagine a transformer decoder looking at the middle layers of the encoder for attention.

Question: When we say tables or weights are shared, are they shared horizontally or vertically?

Answer: The standard answer is horizontally. For the same layer, we have the same weights and if we have an attention block, it will have the same table. But at a different layer, there will be a different set of weights and a different table that it looks into. That said, it is often sometimes done to have weight sharing across layers as well.

3.5 Data

Transformers don't have a strong inductive bias so we need a lot of data to train. We get data scraped from the web and hopefully if we get all of this, we can get things to learn the underlying representation of what that language is. That's what we want our model to actually capture and then we can use it for different purposes.

4 Word Embeddings

For us to be able to learn from text data, we need to have a way of representing words such that their representations are meaningful. For example, in computer vision, the pixels of images mean something. They don't mean much, but they mean something. Thus, maybe if we had a more meaningful representation of words, then learning downstream tasks would be easier!

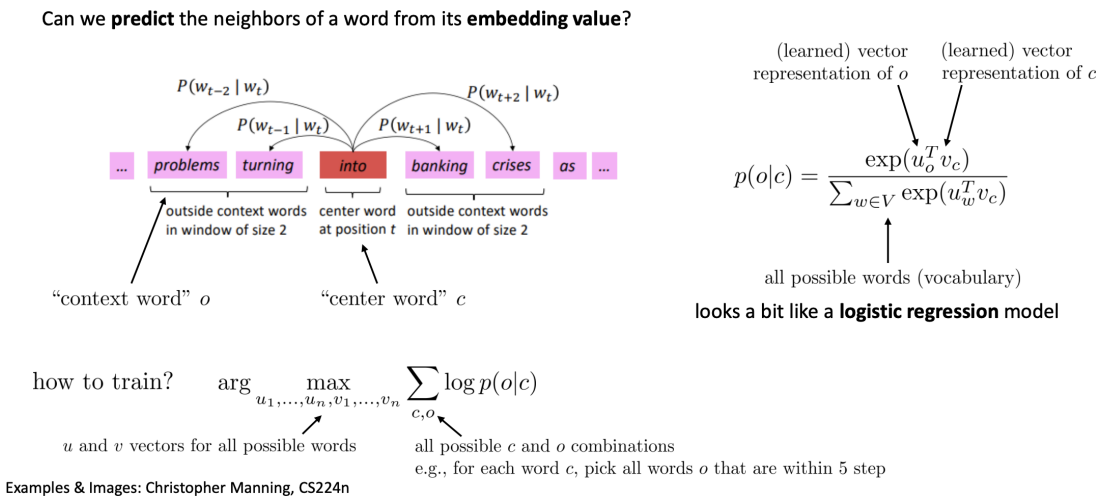


Figure 19.3: How we learn embeddings using word2vec [2]

We represent words with embeddings, but how do we learn these representations? The problem with words is that in the English language alone, you have something of the order of 100,000 words, with 8,000 common words. That's a lot of words. The default way of thinking of categorical variables using one-hot encoding is really annoying for words. We would have lots of words close to each other in meaning but which would not be reflected in this embedding. For example, 'actual' and 'actually' are going to have the same distance from each other as 'actual' and 'banana'. Therefore, this encoding doesn't tell us anything about the meaning of these words.

There also is a question of why we want vectors corresponding to similar words to be close to each other in the embedding space. All of our functions are continuous during training, so we expect to have the basic behaviour of things going in that are close will give things that are close going out. In particular, for losses, during training, we have we want a kind of continuity where words that are close get less words than things that are far. Then, if we have representation with vectors of similar words being close to each other, then hopefully if we make a small mistake during training, we will have something that's also close.

Now that we want our vectors to have this property, the question becomes, how do we learn them? The key idea in word2vec is that words occur in context with other words, i.e, the words that surround a word tells you what a word is like. Therefore, we want to find a representation that reflects this idea of having similar words have close embeddings.

Suppose, we want to learn the embedding for the center word c (fig. 19.3). We create a prediction task, such that, we try to predict the neighbors of the center word. To generate this probability, we use the standard go to way, i.e, we predict the words that are closest to the center word in terms of the inner product. Thus, we can take the embedding for a word o given the context and sum up over all other choices that could be there for the context word.

Question: Intuitively, what we want to do is to have an encoding of embedding of words so that given the neighbors, we can predict a particular word. Here, with word2vec, we are doing what feels like the opposite, i.e, we are predicting the neighbors given a particular word. Why do we do it this way?

Answer: The answer is simply because it is easier to do it this way. If there are multiple inputs and a single output, we need to figure how to combine all this information from the input. It is possible to do it, maybe through a weighted average of the neighbors, but at the end of the day, this is a surrogate task, so we can try the easiest thing first.

Note: There is a subtle distinction that two words should have the same representation if they are roughly interchangeable with each other in a sentence, not that they sit close to each other in a sentence.

References

[1] S. Levine. https://cs182sp21.github.io/static/slides/lec-12.pdf. 2021.

[2] S. Levine. https://cs182sp21.github.io/static/slides/lec-13.pdf. 2021.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. 2017.

EECS 182/282A Lecture 20: Pretraining and Fine-Tuning

Instructor: Anant Sahai
Scribes: Keaton Elvins, Raghav Ramanujam

1 Tokens

1. Tokenizer: In the typical approach for sequence models, the input is first passed to a tokenizer. This parses the input sequence into a series of discrete tokens (i.e. Token-1, Token-72, Token-985...). Once this sequence of discrete objects is generated, they are then passed to a learnable look-up table.

   > **Aside:** For NLP, the byte pair encoding scheme is typically used for this step, which entails repeatedly grouping the most common occurring pair of characters together until the desired "token budget" is reached (similar to the grouping in Huffman encoding but in reverse). This process addresses the issue of encountering out-of-vocabulary words since they will just be broken down into common character sub-groups (read more `here`).

2. Look-Up Table: At this step, each token gets mapped to a vector, which becomes the actual input that the transformer/sequence model sucks up. These vectors are all learnable, so this effectively adds (# of tokens × size of input vector) parameters.

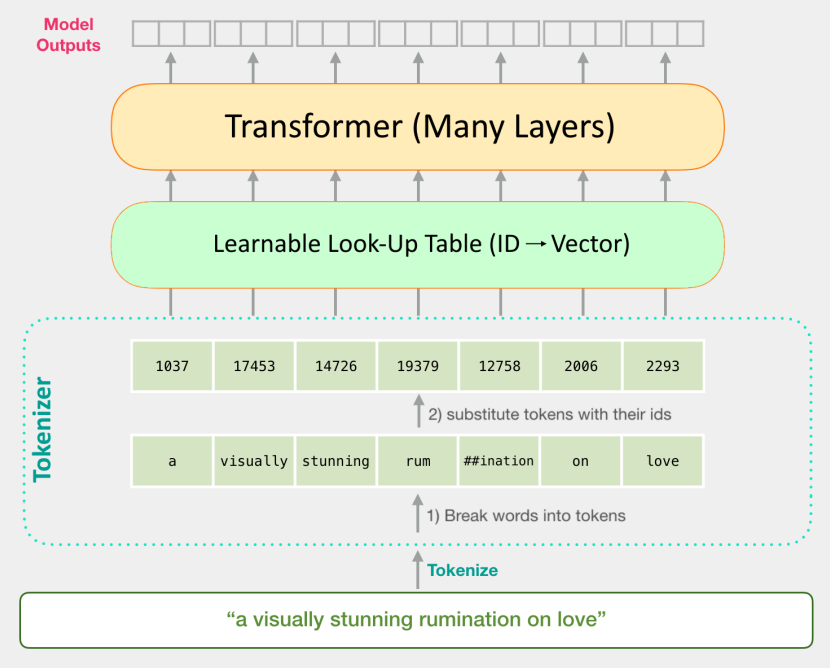


Figure 1: Example of input processing for transformer architecture [1]

2 Word2vec

2.1 Creating a tractable version of the problem

As we learned in the last lecture, the objective with word embeddings is to find vector representations of each word such that similar words $a, b \in V$ produce similar embeddings v_a, v_b. An initial approach could be to set up the optimization problem

$$\arg \max_{u_1,\cdots,u_n,v_1,\cdots,v_n} \sum_{c,o} \log p(o|c)$$

where

$$p(o|c) = \frac{\exp\left(u_o^\top v_c\right)}{\sum_{w \in V} \exp\left(u_w^\top v_c\right)}$$

The intuition for this approach stems from clustering: we want words that are roughly interchangeable in context to be placed near each other in the embedding. However, unlike K-means, our set of words is fixed, meaning we don't have to worry about fitting new words in (although we could always do this by recomputing as before).

One large problem with this approach is that when we attempt to run gradient descent, the denominator of the probability function is extremely costly to compute with a large vocabulary. To mitigate this, we can consider redefining the problem as something closer to binary classification. To do so, let's try a new probability function

$$p(\text{o is the right word}|c) = \sigma(u_o^\top v_c) = \frac{1}{1 + \exp\left(-u_o^\top v_c\right)}$$

The problem with this approach, however, is that it consists of only positive examples! Therefore, the optimization algorithm is incentivized to make all the embeddings line up to achieve really high probabilities. To address this, we can add

$$p(\text{w is the wrong word}|c) = \sigma(-u_w^\top v_c) = \frac{1}{1 + \exp\left(u_w^\top v_c\right)}$$

and optimize the function

$$\arg \max_{u_1,\cdots,u_n,v_1,\cdots,v_n} \sum_{c,o} \left(\log p(\text{o is right} \mid c) + \sum_w \log p(\text{w is wrong} \mid c) \right)$$

But this runs into the same problem as before! Now we're just summing over a huge number of negative examples, and the loss from these could just dominate. In order to balance this tug of war, we can instead just randomly sample a few words to be used for the "w is wrong" negative examples. This approach incorporates both an attractive force for words that occur together (push v_c towards u_o) and a repulsive force for those that do not (push v_c away from vectors u_w), and we are left with our Word2vec optimization problem

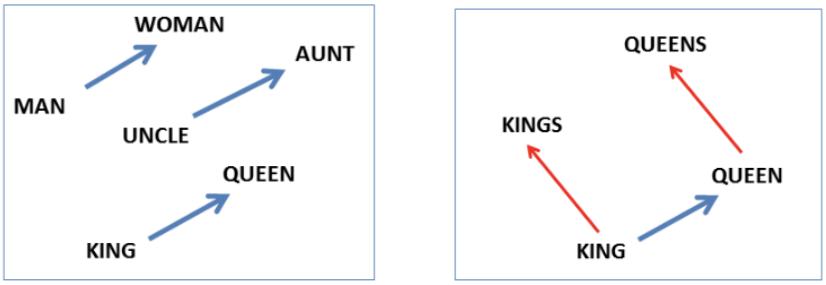
$$\arg \max_{u_1,\cdots,u_n,v_1,\cdots,v_n} \sum_{c,o} \left(\log \sigma(u_o^\top v_c) + \sum_w \log \sigma(-u_w^\top v_c) \right)$$

2.2 Interpreting the learned embeddings

After the development of Word2vec, researchers began to look at the actual embeddings to find if any interesting properties were present that might reflect the underlying structure of a language. After some exploration, they discovered that algebraic relations seemed to have some meaning with the embeddings. For example,

$$\text{vec}(\text{“woman”}) - \text{vec}(\text{“man”}) \simeq \text{vec}(\text{“aunt”}) - \text{vec}(\text{“uncle”})$$

$$\text{vec}(\text{“woman”}) - \text{vec}(\text{“man”}) \simeq \text{vec}(\text{“queen”}) - \text{vec}(\text{“king”})$$



(a) Gendered pairs have matching differences

(b) Plurality is captured similarly

Figure 2: Visualization of grammatical structure in the embedded space [2]

While these relationships are slightly idealized, the learned embeddings from Word2vec were able to capture a surprising amount of grammatical structure. Realizing this, researchers started to use the model to solve analogies (i.e. If women $\rightarrow$ man, aunt $\rightarrow$?). In practice, this returned a mix of nonsense responses and some actually interesting results.

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Figure 3: Examples of Word2vec's performance on more complex analogies [2]

Aside: Following this, concerns began to develop around whether all the stuff the embeddings were learning was desired (e.g. was it learning sexist/racist/homophobic ideas from the training data). To address this issue, researchers tried going through datasets to censor out nasty unwanted training points and applying data augmentation to make the embeddings invariant to these regularities. However, this is still a field of ongoing interest, as these approaches may not be enough.

3 Pretrained Language Models

3.1 Contextual representations

One big advantage of word embeddings over one-hot encodings is the incorporation of latent information about the language/word that might otherwise have to be learned by a downstream model. However, since the embeddings are constant in word2vec, the same word used in two different contexts will produce the same embedding.

Figure 4: Same word in two difference contexts [2]

This implies that we need some form of context-specific representation for our model. In order to address this problem, we can

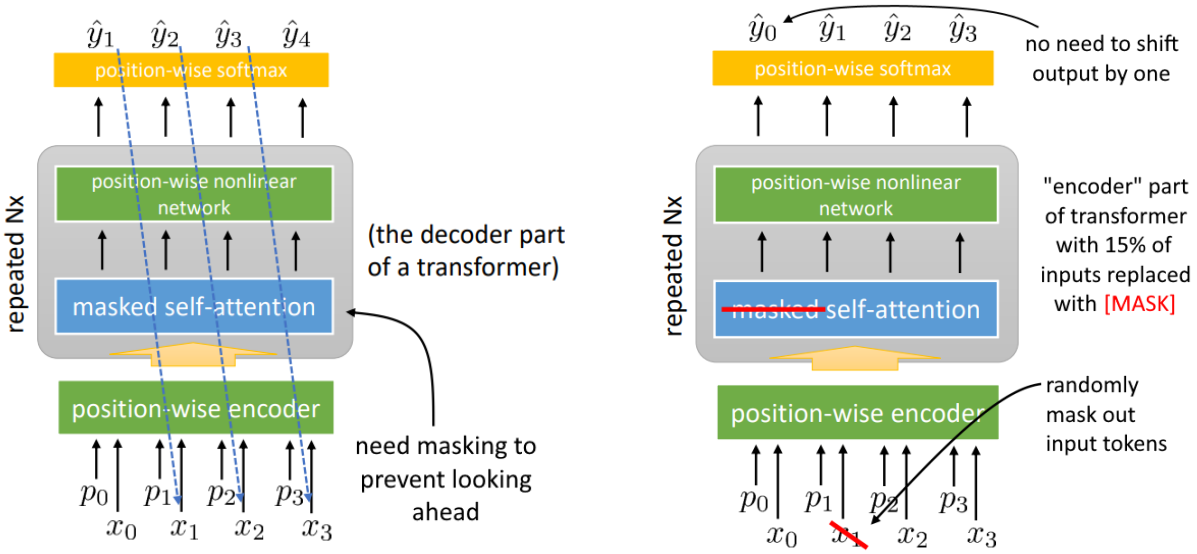
1. Train a language model on a surrogate task
2. Run it on a sentence
3. Take the hidden state from the model and treat it as the embedding

But this raises the complication: how do we train the best language model to get a high-quality embedding? What architecture should we use, and how does the surrogate task affect performance?

One approach is to train on the task of predicting the next word in a sentence while using a decoder-style transformer architecture (autoregressive idea used by GPT). However, since we don't want the attention mechanism to just be able to look ahead in the sentence and know exactly what to output, we have to implement **masked self-attention**. In practice, this is done by setting the relevant inner products to negative infinity, which reduces the contribution of their values to zero via the softmax.

Masked self-attention works for many tasks but introduces an interesting limitation: the model can only build context for a word by attending to the ones that came before it. In Word2vec, however, we were able to look at things on both sides of the center word to build context. How can we do something similar here?

3.2 Bidirectional Transformer Language Models



(a) One-directional transformer used in GPT

(b) Bidirectional transformer used in BERT

Figure 5: Differences between popular transformer-based language models [2]

We can modify our approach by changing the surrogate task! Instead of predicting the next word, we can mask out a small percentage of the input (replace with a [MASK] token) and train the model on predicting what those words would have been. This self-supervised task is very similar to what we did with masking in autoencoders, which we analyzed under the lens of low-rank approximation using PCA. Due to this change, we no longer need masked self-attention, and the model can build context for a given token using input that both precedes and follows it.

It is also important to distinguish that there are two losses we can use here: loss based on the predictions for the masked tokens, and loss for the rest of the sequence. It is critical to find a balance here, as letting the second option apply too much gradient pressure would result in the model just learning the identity function and giving up on the masked predictions. So in practice, the main goal is just to fill in the blanks, although BERT did modify this scheme slightly to find the right balance.

While BERT was training, not all of the 15% of tokens were actually replaced with the corresponding [MASK] token. While 80% of them still were, 10% were instead replaced with some random wrong word (similar to a denoising autoencoder), and the remaining 10% were just the correct word unchanged [3]. By leaving the token as is sometimes, we allow the second loss from above to apply a small amount of gradient pressure. This encourages the model to also take into account the current token, rather than being entirely context based, and strikes the appropriate balance between the two losses.

3.3 Training BERT

While training BERT (as seen in Figure 4a), the researchers took an additional step to try and force the model to learn sentence-level representations. They passed in two sentences at a time, separated by a [SEP] token, and randomly swapped the order of the sentences 50% of the time. They then added a surrogate binary classification task, denoted Next Sentence Prediction or NSP, and asked it to predict if the sentence order had been swapped or not (in addition to the previously stated task). In order to accomodate for this extra task, the researchers added a special [CLS] token at the start of each sentence pair, with its corresponding output from BERT being the output for the NSP task. This little tweak during pretraining turned out to be very beneficial for downstream tasks like question answering and natural language inference, as the model was forced to learn both context-dependent word-level and sentence-level representations.

Aside: Some people have tried using Word2vec as a starting point for the token-to-vector encoding for models like BERT, but it isn't an exact match since not all tokens are words. While they often are, these models have some token budget they must stay within, so sometimes complicated/rare words are broken up. Check out `https://beta.openai.com/tokenizer` for an example of how OpenAI's GPT family of models process text into tokens.

4 Fine-Tuning

4.1 Using BERT

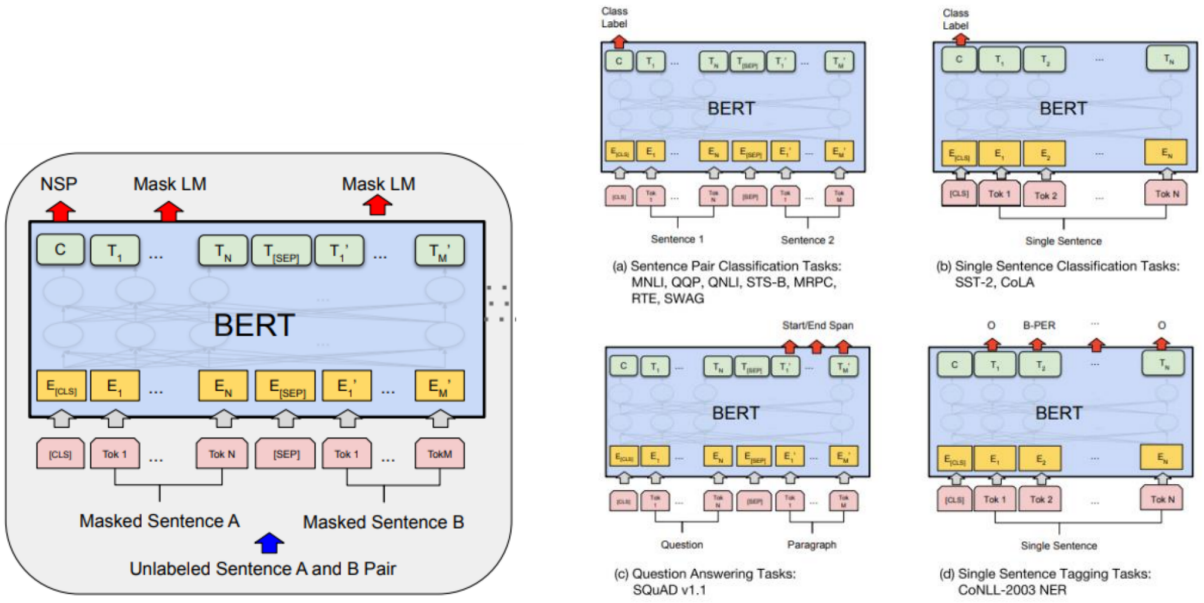
When it comes to actually trying to use BERT on some downstream task, we can think back to what we did with PCA: use an autoencoder approach, train the model, chop off the decoder part, and just use the embedding produced by the encoder on something new. The same works with BERT! One example would be entailment classification, or whether or not one sentence is logically a consequence of another. Since this is more related to the NSP task from training, we can cut off whatever classifier/linear-layer was used to make the NSP prediction, pass the embedded representation that BERT built to a new model, and train on our entailment task. However, now we have two options for this last fine-tuning step:

1. Freeze BERT and only train whatever classifier we've added at the end: This approach is similar to how we thought about the autoencoder in the PCA context. The entire encoding network is frozen and can be thought of as some input featurization for the final component to run inference on.

2. Train BERT end-to-end on the new task: Sometimes this approach of fine-tuning the entire model works better, but it also takes significantly more compute. In addition, the ratio of unlabeled to labeled data is often enormous, so trying to train such an over-parameterized network on a small set of labeled data may not be worthwhile.

4.2 Additional Tasks

Since the above procedure only really uses the first output from BERT, it is natural to wonder what uses the other outputs might have. It turns out that BERT is useful for many tasks, it simply depends on which components the user wants to use.



(a) BERT model during training

(b) BERT model during fine-tuning. Utilized outputs depend on use case

Figure 6: BERT model during training vs fine-tuning [2]

As seen in Figure 4b, we can build on top of the class label for classification tasks like above, or we can also select different outputs depending on the given task. For example, if we are trying to identify the span of contents in a paragraph that answer a given question, we can pass in the question followed by a separator token and the paragraph. We can then build on top of the outputs from the paragraph and fine-tune a model to choose the start/end of the span. The final example in the figure is entity-labeling, or identifying things like people's names, locations, and other categories.

4.3 Using BERT for feature generation

One question that may arise is how to use BERT to get features like we did with Word2vec. Since BERT has many layers, we have a choice of which hidden state to use. In practice, it is worth testing out the embedding from different layers, as well as the sum of different layers, to find the best possible contextualized representation. This idea is visualized below. More info can also be found at `http://jalammar.github.io/illustrated-bert/`.

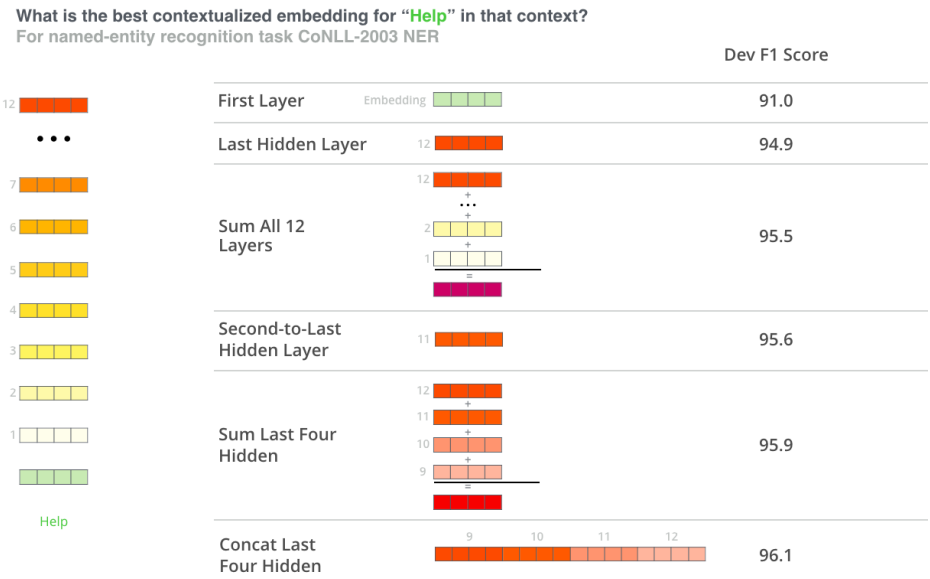


Figure 7: Representation scores of possible choices for embedding [4]

Aside: When exploring this, one may find the second-to-last layer works better than the last for feature generation. While the exact reason is up for debate, intuition for this could be that the last layer is more specific to the surrogate task while the second-to-last layer contains more general information.

5 Surprising Results With GPT

In the problem of text generation, we give the model some input for context and ask it to spit out a continuation (e.g. given the first paragraph, finish this article). Due to BERT being bidirectional and trained with context on both sides, it is not particularly suited or well-performing in this task. However, this problem is perfect for autoregressive models like the one-directional transformer architecture used in GPT (partially visualized in Figure 3a).

In fact, we can frame many tasks as text generation and see how GPT performs without any additional training (and therefore without performing further gradient descent). For the example of machine translation, one could input "The translation of 'she' to Spanish is 'ella'. The translation of 'ball' to Spanish is..." to GPT, and find that it would return the correct translation ('pelota') more times than just luck would suggest. The implications of such properties are still being figured out.

Food for thought: How would you frame the task of article summary to GPT?

References

[1] Alammar, J (2018). A Visual Guide to Using BERT for the First Time. https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

[2] Levine, Sergey. NLP Applications: CS 182 Lecture Slides. https://cs182sp21.github.io/static/slides/lec-13.pdf

[3] Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018. https://arxiv.org/abs/1810.04805

[4] Alammar, J (2018). The Illustrated Bert, ELMo, and co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-bert/

What we wish this lecture also had to make things clearer

The slides and the visuals were great, but we think the mechanics of byte pair encoding got somewhat brushed over. We would also have liked to see some visuals for BERT word embeddings similar to the ones we got for Word2vec, as well as a brief exploration of the main idea behind ELMo.

CS182 / CS282 - UC Berkeley

Lecture 21: Meta-learning, Fine-Tuning, Transfer Learning

3rd November, 2022

Lecturer: Prof. Anant Sahai *Scribes: Aman Saraf, Tianlun Zhang*

Reviewers: Kiran Eiden, Nikhil Prakash, Jiayang Nie

1 Lecture Overview

The context for this lecture is based on a exploration of transformer based language models which typically have very large parameter sets. This lecture will seek to explore strategies for fine-tuning and then come back to look at pre-training.

2 Fine Tuning

The overall approach to fine-tuning is as follows:

- **Step 1:** Pretrain a large model with self-supervision and lots of data. (***Classical ML Analog***: PCA on unsupervised data)
- **Step 2:** Fine-tune the model on task specific data set or objective. (***Classical ML Analog***: Train a classifier or regression model on data with reduced dimensionality)

Next we look at some strategies to fine-tune effectively, considering the benefits and limitations of each of them.

2.1 Strategies to Fine Tune

Let's start by considering the BERT-style model [1], which is pre-trained with two tasks:

- Sentence Order
- Fill in the blank (Masked Denoising Autoencoder)

2.1.1 Decapitate training "head" and *only* train a new "head"

Note: Here "head" refers to the part doing the surrogate task, and not a transformer "head".

In this strategy, we remove just the surrogate task layers from the end of the model and replace them with another newly initialized head. This head should be well suited to our task and usually comes from a similar domain to the surrogate tasks that the original model was trained on.
This new head now operates on features extracted from our task's dataset by the backbone for our

model. This is like lifting the feature space and treating the output before the surrogate head as features we want to run our task on.
For example we could:

1. Replace the final linear layer and softmax for the final sentence order task with a new linear + softmax layer for your own specific sentence-level task. We keep every other layer intact.
2. Train the new final layer, using Stochastic Gradient Descent (or any optimizer of your choice) for your specific task.

Note: In this approach we treat BERT as a feature map, where the last layers provide features.

Here we prompt the model using the token for sentence order classification. Remember that for BERT the input is as follows:

<classification token> sentence 1 <separation token> sentence 2

where sentence 1 and sentence 2 are masked. Masking is done by replacing a word by the special mask token, and therefore removal of prompt words requires no scaling like in dropout. This is also possible since tokens are discrete whereas in real number spaces having mask tokens is more difficult.

2.1.2 Variation: Use other BERT Embeddings

The choice of using features/embeddings from the last layer is not fixed. We can use other embeddings that BERT gives us. For example:

- Average together all the layers from the transformer.
- Concatenate the last four layers (arbitrary)
- Average together the last four layers (arbitrary)
- Use the last $N - 1$ layers
- and so on...

Similarly, if the task is "word-level" then we can use BERT as a word-level feature extractor as well.

Notice that here the language model almost behaves like a feature extractor, and exploring these strategies to combine features or choose the extraction backbone, is almost like a hyper-parameter search problem.

2.1.3 *Major* Variation: Train the entirety of the model after replacing head

Previously, we were only training the final layer and everything else was frozen. In this strategy, we will let **all** the weights adapt to our new task by training all layers at once.

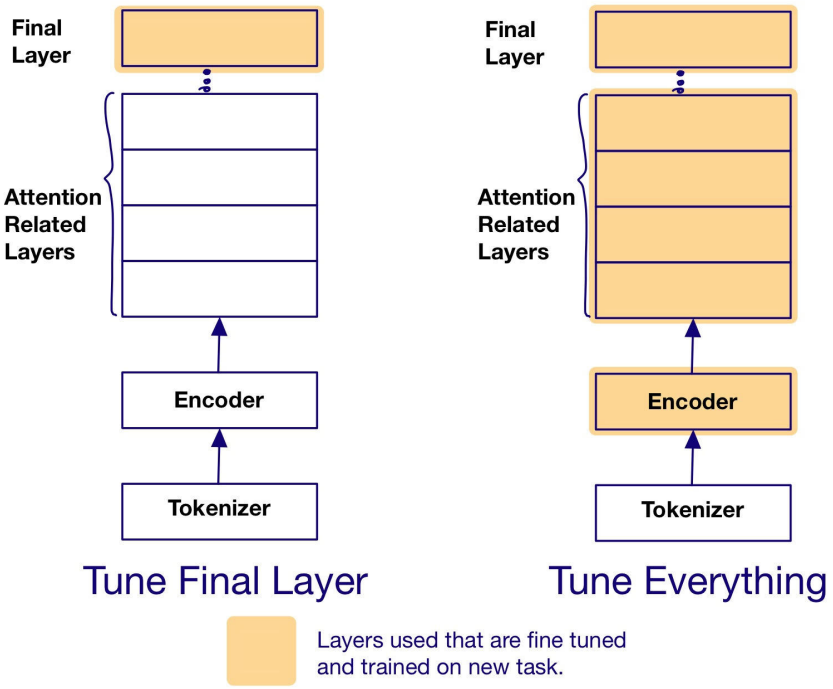


Figure 1: Illustration of different strategies of for fine tuning.

This strategy has a few advantages and many disadvantages, as follows:

- Pro:
  - **Increased Capacity:** We provide more capacity in the model, to fit our new tasks. This is obvious as we are training many more weights now in an over-parameterized model.
- Cons:
  - **Fear of Overfitting:** Excessive parameters leave us open to this risk.
  - **Harder to Scale:** to lots of different tasks - for example, if we only train the individual heads for each of our tasks, we only store each individual head, whereas if we allow the entire model to train we need to store a version of the entire model for each task.
  - **Divergent Backbone Gradients for different tasks:** If trained end to end on different tasks, the various gradients can start pushing the backbone in all sorts of directions, often divergent from one another.

2.2 Interesting Questions:

2.2.1 How is it possible, that "pre-training + fine-tuning" works, when just fine-tuning from random initialization does not work? Is not the model equally large in both cases and heavily over-parameterized?

There are two explanations for this behavior, although some questions still remain:

- **Explanation 1**: Somehow pre-training gets us to a "fortunate" initial condition. This initial condition is particularly suited for fine-tuning.
- **Explanation 2**: Because we have so many parameters, it's as though we are training a generalized linear model using "derivative" features.This works because the principal features of the generalized linear model are aligned to this family of tasks.

2.2.2 Is it necessary to replace the head with a linear + softmax layer or can we use other layers like Random Forests?

Intuitively should work similarly to how random forests work for other problems. Need to try it out.

2.2.3 Are there certain architectures for backbones and head combinations that when pre-trained adapt well to head replacement for other tasks, or that allow for training with multiple different heads without degrading performance for each other?

Seems like an open question.

3 Emergent Useful Behaviour in Large Language Models

Interesting new behaviors were observed in large language models like word2vec and Generative Pre-trained Transformer(GPT) [2]:

3.1 Word2Vec:

It turns out the geometry of learning word embeddings can capture regularities of the meanings. For example, using the difference between two words' embeddings learned by word2vec in solving the "analogies" problem, can get better results than guessing without any additional training:

Solving "analogies" example:
Input: ("Man", "Woman") ("King", "?")
Predict(output) : "?" — the right answer should be "Queen"

Here the strategy for extracting the analogy answer without training on this task was to use the difference vector between embeddings.

$$emb(\text{"}King\text{"}) + (emb(\text{"}Woman\text{"}) - emb(\text{"}Man\text{"})) = emb(\text{"?"})$$

By calculating the distance vector between the embeddings of the prompt analogy (in this case, "Man" and "Woman"), and then adding that to the incomplete analogy (here, "King"), we can predict the analogy output should be "Queen".

A more dramatic example of such learning, GPT-like models demonstrated an ability to answer many questions and complete sentences from just pretraining on predicting the next token tasks. **Complete the sentence:** Seed GPT with some text and see what it says next

Example:
Input : "Bob went to the zoo when he was frightened upon seeing a ferocious ....."
GPT continues: "...tiger. This tiger was huge. . . ."

People observed even more surprising behavior in that GPT could be used to answer questions. For example, when seeded with a prompt that provides analogies of states and their capitals, GPT was typically able to produce the capital for any state when asked to continue:

"Let's think of capitals of states. For example, the capital of California is Sacramento; the capital of Massachusetts is Boston; the capital of <QUERY STATE> is _____"

Here <QUERY STATE> could be any state and GPT would answer with the capital of the state correctly. This led to the area of **prompt engineering** - figuring out prompts that make the models do what we want. It turns out that we can get a language model to perform new tasks simply by exploring the generation capability of the model and picking appropriate texts, without any further training. We can simply treat the generative language model as a black box and encode our tasks as a call to that black box. There are no gradients and weights updating in this process. This is often called Zero-Shot or Few-Shot learning.

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Lecture 22 Scribe Notes

Ann-Katrine Christiansen & Jesper Hauch

November 8th, 2022

1 Continue Fine-tuning

In the last lecture, material regarding large language transformer based models was covered. More specifically, it was seen how to utilize models, such as BERT and GPT, for tasks different from their surrogate task. Fine-tuning models to a different task, ensured savings on computation and a better starting point for further training. Table 1 shows different methods of fine-tuning covered in this lecture and the last lecture.

	"Feature Extraction" Train new head	"Fine-tuning" Retrain everything	"Prompt Engineering" No gradient step, zero shot, few shot	"Prompt Tuning" In "computer-ese" w/ gradients
Number of parameters that need to be trained	Small to Medium	Huge	None	Small
Amount of task specific training data it can handle	Small to Huge	Small to Huge	Small	Small to Huge
Multi task scalability	Good	Bad	Good	Good
Performance on desired task	OK	Good	Bad to OK	Good

Table 1: Story of fine-tuning so far.

1.1 Feature Extraction

The first column of Table 1 describes a classical approach, where the pre-trained model is used to extract features as previously seen in PCA and k-means. This approach is conducted by removing the head of the pre-trained model and replacing it with a new task-specific head. During training, the weights of the pre-trained model are freezed and only the new head is trained to perform the task. The new task-specific head can choose to ignore or give precedence to certain parts of the pre-trained model depending on their relevance. This is why adding and training a new head works. There are multiple ways to extract the weights/features from the pre-trained model, for instance taking the weights of the last layer or the average of the top k layers as seen in the last lecture. This is known as the feature extraction paradigm.

In this paradigm, a small to medium number of parameters need to be trained since only the new head is trained for the different task. When training a new head, SGD based training is used and the approach can therefore handle any amount of data (the more, the better). Additionally, this paradigm is good in multitask scalability since a new head can be trained for each task. The performance on a desired task is viewed to be "OK", since only the head of the model is trained. Thus, the model only have a few parameters to adapt to the specific task.

1.2 Fine-tuning

Another approach, seen in column two of Table 1, is to not freeze the weights of the pre-trained model and use it as an advantageous starting point for training the model to perform a different task. This paradigm is called fine-tuning. Different strategies regarding the head of the pre-trained model have been used, where one keeps the head for the new task and another removes the head and replaces it with a new task-specific one.

Contrary to the feature extraction paradigm, retraining the entire model entails training a huge number of parameters. However, like the feature extraction paradigm, fine-tuning can handle any amount of data due to SGD based training. Unfortunately, fine-tuning scales badly for multiple tasks. This is due to the large size of the model which needs to be trained separately for each task and catastrophic forgetting/interference (explained more in depth in Section 2). Fine-tuning can still achieve good performance on specific tasks despite the poor multi-task scalability.

1.3 Prompt Engineering

A third approach, basic prompt engineering, treats the entire pre-trained model as a black box, where information from the model is retrieved with prompts. The prompts can be interpreted as questions framed in such a way, that makes the model do the task that you want. There are different ways to construct the prompt, where zero shot and few shot was covered in the last lecture. Examples of these can be found in Box 1. The internals of the model are not considered directly and no additional training is performed. Thus, training parameters using gradient steps is not possible in basic prompt engineering. In this way, you rely on the model's learned embeddings of a language to perform a different task.

Basic prompt engineering can only handle a limited amount of training data, since it must be included in the prompt given to the model for a specific task. Due to the transformer like architecture of the pre-trained model, there is a limit to how much training data can be considered at once due to the quadratic scaling of complexity. Different methods exist to

compress the training data before constructing the prompt. For example, utilizing concepts used in support vector machines, where some data points are considered more important than others. In this paradigm, you have one prompt per task which ensures good multitask scalability as no additional training is needed. For prompt engineering, it is surprising that it even works but performance is not great compared to the other paradigms.

Box 1: Examples of zero shot and few shot

Zero shot is when the model is asked a question directly without any training examples. A zero shot example is to ask the model *"What is the capital of California?"*

Few shot is when the model is provided with a few training examples included in the question or prompt. A few shot example is *"The capital of Massachusetts is Boston. The capital of Arizona is Phoenix. What is the capital of California?"*

In both cases the model is expected to answer *"Sacramento"*.

1.4 Prompt Tuning

Prompt tuning is the last paradigm covered in this lecture and arose from the wish to not be limited by the dataset size and lack of gradients in prompt engineering. The main idea is to create prompts in the computer's continuous spaced vector language ("computer-ese") instead of English or any other human language. In prompt tuning, you start out with an English language prompt in vector form, which is updated by gradient steps to make the prompt more understandable for the computer. Tuning the prompt greatly improves performance seen in basic prompt engineering while maintaining scalability. This paradigm will be covered further in the next lecture.

1.5 Note on Training

In essence, the specific parts of the model architecture, that are trained when learning a new task, is illustrated in Figure 1. In all the aforementioned paradigms, you execute batches at training and you can therefore also execute batches at use/test time. For instance, in prompt engineering you can group together different tasks in a batch and execute all at once. The same applies for feature extraction, where all features from the model can be extracted at once, and then used to run the specific head. When training a model to do multiple tasks at once, each task has its own loss function allowing it to adjust independently.

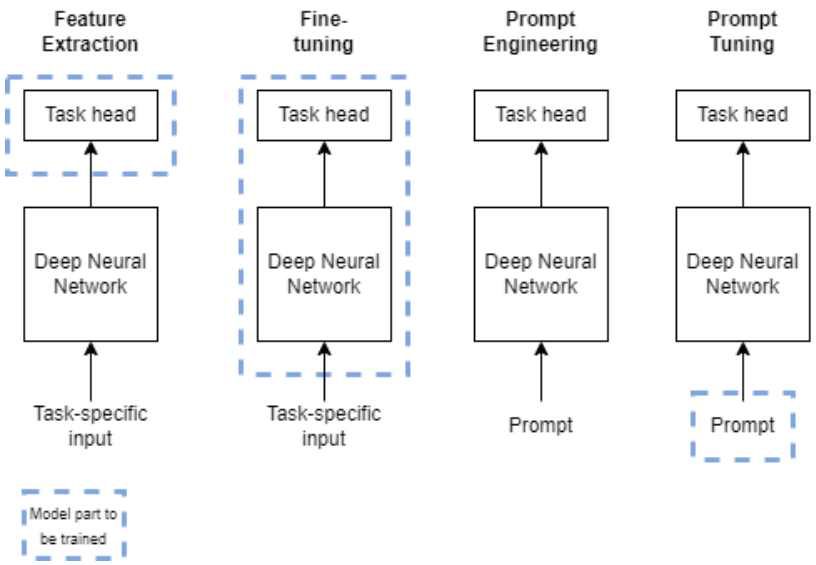


Figure 1: Illustration of model parts that are trained when learning new tasks.

2 Catastrophic Forgetting/Interference

Catastrophic Forgetting describes the phenomena of deep neural networks forgetting how to perform old tasks when trained to do new ones. The old task can be interpreted as the surrogate task performed during pre-training, whereas the new task can be the fine-tuning task at hand. A simple example of catastrophic forgetting in computer vision is found in Box 2.

Catastrophic forgetting is especially of interest in continual learning, where models learns a series of tasks sequentially. Traditionally, this has been largely important for people working in artificial intelligence, since they are trying to achieve models that can keep learning in a real environment. Counterintuitively, the phenomena of catastrophic forgetting can still occur even when continual learning is not trying to be performed.

Catastrophic forgetting does not align with our intuition of how deep learning models work. From a convolutional neural network perspective, our intuition is that early layers learn more basic low-level features, such as edges, local configurations of edges, component pieces, and textures, whereas the last layer learns task-specific features. In reality, this intuition is wrong, since earlier layers might already be somewhat task-specific. The somewhat task-specific means that there is distilling information that is generic to the problem domain. However, this distillation is favoring information that is relevant to the task and irrelevant information is favored less. This corresponds with what is seen in Table 1, where better performance on a desired task is found when the entire model is retrained in the fine-tuning paradigm as opposed to only training the head in feature extraction.

The fact that our intuition is different from reality gives rise to two questions.

1. How are task-specific features learned in early layers?
2. Why can learning these task-specific features early on break performance on previous tasks?

For the first question, it turns out, that when skip connections were introduced in neural network architectures, it became possible for early layers to learn task-specific features. This is a consequence of the weights being directly influenced by the loss from the final layer, which they can adjust to accordingly. The ability to learn task-specific features in early layers turns out to be beneficial in other contexts, such as being able to fit very large models on mobile devices as a result of early exiting. To answer the second question, if the earlier layers have adjusted and shifted out of alignment, the final head is unsuccessfully trying to pull all the layers back into alignment to perform the previous task. The information from the previous task might still be present in the model, which can be evident by the great performance achieved when retraining the previous task head.

Box 2: Example of catastrophic forgetting in image classification

Consider learning image classification with a convolutional neural network and dividing the training data to have each of the classes one at a time when training. All the data is utilized but the ordering in which classes occur is simply changed, as opposed to the random shuffle, that is normally done.

If the above procedure is followed, it can be observed that the model will become good at doing the first class. After the first class the model will do bad at the second class at first, but performance will increase as the model sees more examples of the class. This pattern will repeat itself throughout the rest of the training data. If the model is presented with an image seen from a previous class after a while, it will have forgotten how to correctly classify it even though it was able to do it earlier.

2.1 Solving Catastrophic Forgetting

A remedy for catastrophic forgetting can be to make early layers less task-specific but it is not considered as the main objective. The main objective is to ensure that the old task heads adapt while training new ones. As a result, early layers will have less task-specific information. Approaches that try to achieve this are provided below:

- The naive approach.

- Replay during training.
- Learning without Forgetting.

The naive approach is to do batch learning by having different tasks in the same batch instead of continual learning. In this way, the early layers learn low-level features that are good across all tasks and the heads learn to classify appropriately for all the tasks. This approach is not helpful as each time a new task is introduced, it is necessary to train on all tasks again.

A way to approximate the naive approach is the idea of **replay during training**. This idea is inspired by research in the neuroscience literature, which is described further in Box 3. The engineering approach to replay is to insert examples of the old tasks when training on the new task. As a result, when replaying old tasks to the model, it allows updating the heads of the old tasks so they are not forgotten when training on a new task. Replay is the golden standard of handling catastrophic forgetting today. However, replay incurs a minor problem related to storing examples of the old tasks in the replay buffer.

Another approach, described in the **Learning without Forgetting** paper is to only use new task data to train the network while preserving the original capabilities [1]. This is done by keeping a score for new task examples by creating pseudo-labels from predictions on the new task using the old heads. Afterwards the entire model is retrained on the new task using the pseudo-labels generated previously [2]. Therefore, in this approach, the old task heads are generating the target for the new task.

The learning without forgetting approach is related to *knowledge distillation*, where the general principle is that a learned neural network, that is reasonable good at doing a task, can be used to generate analog labels. This introduces the need for analog loss functions during training, such as mean squared error. The gradients calculated from the loss are used to update the new version of the old heads, and the lower layers. One variation of knowledge distillation is to treat the outputs of old tasks as probabilities, which can be modified by raising them to a (fractional) power, and use cross-entropy loss in retraining. A common choice is to take the square root of the probabilities, as this softens the distinction between large and small probabilities. Oppositely, raising probabilities to a non-fractional power, increases the distinction between large and small probabilities.

The performance of learning without forgetting is subpar compared to replay but is better than doing nothing. The problem with learning without forgetting compared to replay, is that distributional shifts are present, as only the new tasks distribution is observed and the old

task distributions are solely being updated by the new tasks. This causes decay of performing the old tasks.

Box 3: Connecting catastrophic forgetting to neuroscience

In the early development of deep neural networks, researchers found that catastrophic forgetting occurred when they tried to use a pre-trained network to learn a new task. From the perspective of viewing neural networks as some variation of a biological system, it was considered unrealistic that humans forget how to perform an old task when learning a new one.

In the neuroscience literature, it was found that when humans and animals dream, they replay experiences to build better memories. The replay of experiences was used to address the problem of catastrophic forgetting in artificial intelligence, when the concept of replaying training examples was introduced by using a replay buffer.

3 T5/BART

Recall talks about BERT and GPT from previous lectures. BERT is considered as an encoder-only transformer model and a masked auto-encoder, whereas GPT is considered as a decoder-only transformer model and training is predict-next-token. BERT is made for the feature extraction paradigm, since it is good at extracting context-specific features. For GPT, it is more targeted towards the prompt engineering paradigm, where generation of text is of importance.

Researchers found it odd that architectures only incorporated either an encoder or decoder strategy. As a result, Google and Facebook created T5 and BART respectively, which are both encoder-decoder transformer models and are basically the same idea. The idea was that researchers wanted to design a model that targeted the fine-tuning paradigm, which is the most widely used of the paradigms covered. T5 and BART are trained on massive corpora of text like BERT and GPT, except these architectures take advantage of their encoder-decoder architecture to provide greater flexibility for future/downstream tasks. The key idea of T5 and BART is a masked auto-encoder, where entire spans of tokens can be masked without the model knowing how many tokens are masked. This is unlike BERT, where spans of tokens cannot be masked without the model implicitly knowing how many tokens are masked given the positional encoding. An example of this can be found in Box 4.

Box 4: Example of masking in BERT, T5 and BART

Let us consider the following sentence:

"Thank you for inviting me to your party last week."

Imagine masking "for", "inviting", and "last" in the above sentence.

In BERT each masked word is replaced by a mask token. Therefore, masking this sentence would look like the following:

"Thank you <MASK1> <MASK2> me to your party <MASK3> week."

For T5 and BART, entire spans of tokens are masked without implicitly telling the model how many tokens are masked. Therefore, "for" and "inviting" are masked with the same mask token. The masked sentence looks like the following:

"Thank you <MASK1> me to your party <MASK2> week."

There is a subtle difference between how T5 and BART output their predictions to the masked tokens. T5 will fill in the masks by answering a prompt: <MASK1> is "for inviting" and <MASK2> is "last". BART will fill in the masked tokens by returning the unmasked sentence seen at the top of this box.

4 What we wish was explained more in detail

We found that the Learning without Forgetting approach could have been described better in the lecture by use of examples. Additionally, as indicated by one of the reviewers, it was unclear how the differences between BERT, T5, and BART end up affecting model performance.

References

[1] Zhizhong Li and Derek Hoiem. *Learning without Forgetting*. 2016. DOI: 10.48550/ARXIV.1606.09282. URL: https://arxiv.org/abs/1606.09282.

[2] La Tran. *Learning without forgetting simplified - Towards Data Science*. Nov. 2021. URL: https://towardsdatascience.com/learning-without-forgetting-simplified-33243bd0485a.

CS 282 Deep Neural Networks **Fall 2022**

Lecture 23: Nov 10, 2022

Lecturer: Anant Sahai *Scribe: Jing Xu, Kiran Ganeshan*

23.1 More Details on Prompt Tuning

Recall the prompt tuning setup, which is shown in Figure 1.

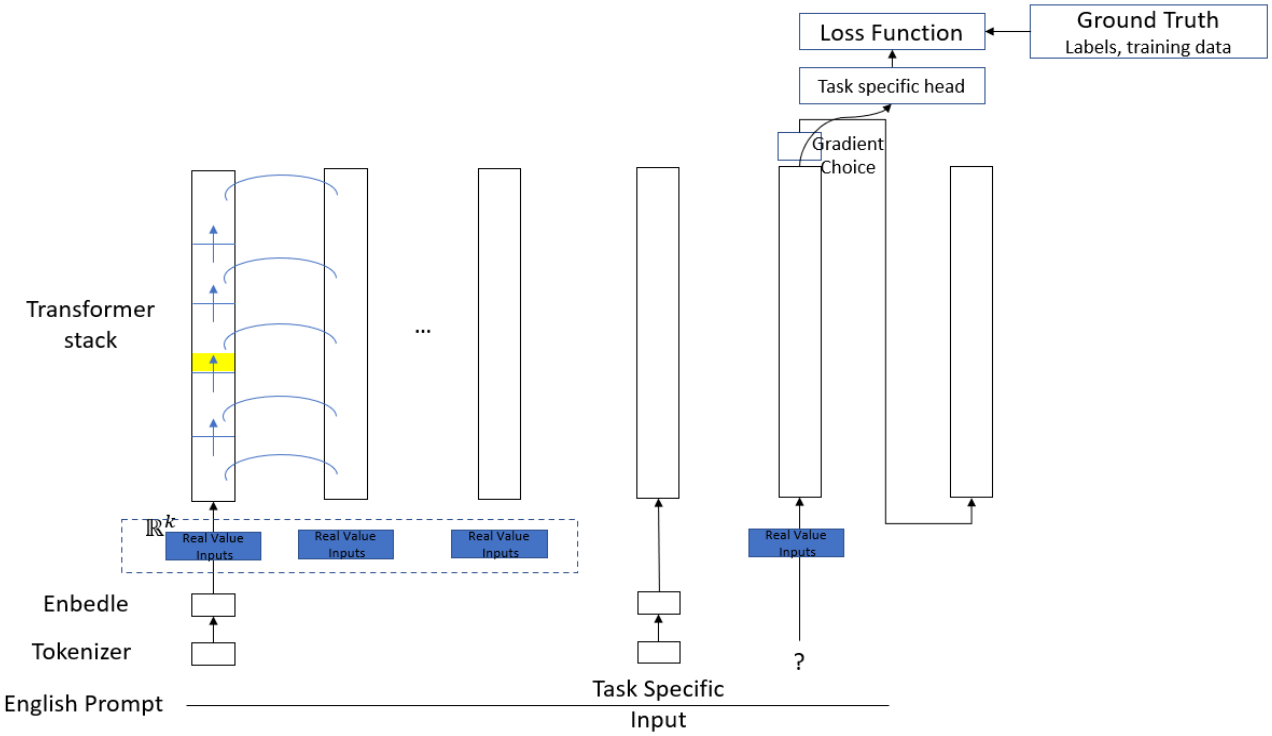


Figure 23.1: Framework of Prompt Tuning

For example, consider the prompt "What is the capital of California". We might consider this prompt to be part of a task that involves recalling capitals. However, the only token which specifies which specific subtask we want to solve is the "California" token. We refer to this as the Task Specific Input token in Figure 1. The remainder of the prompt could be changed without changing the semantics of the question.

With this in mind, note that the outputs of the Transformer depend only on the real vector-valued inputs that are produced by the embedder. This raises the following question: why do these real vector-valued inputs need to correspond to English sentences? (Why would it be that the output of the embedder is the

best choice?) While there are a finite number of choices within our embedding table, there are infinitely many choices for our inputs, so we can utilize gradient descent to find the best one. Fine tuning works with both small and large models, while prompt tuning seems to only work super well with larger models. The only way information is propagated to the next input is through the population of the attention tables. Why do I need consistency across layers? This is the inspiration for soft prompt in between layers. The supervision will be provided by the same loss function used in the feature-extraction view of task-specific finetuning. These are known as **soft prompts**. In Figure 1, there is a soft prompt in each transformer layer.

Note: this idea requires that we sample outputs in a differentiable way (with respect to scores) at the orange boxes to enable gradient flow to the prompt inputs. For example, argmax may not work.

Questions:

- How do we choose the length of our prompts?
  → Start with an English prompt that works OK, and stick with that length.
  → Search over prompt lengths for maximum accuracy, starting with the shorted prompt lengths. Which of these performs better is very sensitive to the underlying task.

Advantages:
1) Improves performance on underlying task
2) Maintains scalability by keeping the model weights the same across tasks
3) Allows us to use large training datasets
4) Lets us leverage ensembles, etc... (Standard Gradient Descent Approach)
5) Avoids the large performance differences between semantically similar prompts suffered by manual prompt engineering
6) Lets us use feature extraction as well, unlike manual prompt engineering

Disadvantages:
In order to reach full fine tuning level performance, we normally must use very large models. Example: If the transformer is a very large model (10billion+ params), then prompt will approach performance of fine tuning. If the transformer has 100 million params, for example, then prompt tuning will not get you there. This is not fully understood.

How can we improve this further?
A) Put soft prompts around the input, rather than having the input follow the soft prompt
B) Apply a learnable mapping to the input itself
A') Let the soft prompt depend on the input, rather than learning it independently
C) Add soft prompts to intermediate layers of the Transformer. Language models observe the initial condition and generate and continue the flow of the "differential equation". The prompt shapes the evolution of the system. We get a richer shape of how we set the initial conditions.

Questions:

- Would it help to add a trainable layer between the embedder and the soft prompts?
  → No, since this would just be another way to learn soft prompts. This time, soft prompts would be a linear transformation of embedder outputs, rather than parameter vectors. If we use a more complex model like an LSTM, this becomes identical to A'.
- If we add soft prompts to an intermediate layer in the Transformer at a token T, throwing away the output from previous layers at T, do we stop gradient flow to the soft prompts at the input?
  → No, because there is layer-wise self attention. Specifically, because the outputs of the first layer at later tokens depends on the keys and values of the first layer at T, there is "sideways" information flow along the sequence dimension at each layer, so the input soft prompt still receives gradient flow. The advantage of using soft prompts in intermediate layers is that performance benefits extend to much smaller models.
- What is the relationship between approach C and skip connections?
  → This approach is uniquely opposite to skip connections in that rather than encouraging gradient flow by adding inputs, we completely overwrite the intermediate values with a parameter vector, stopping gradient flow.

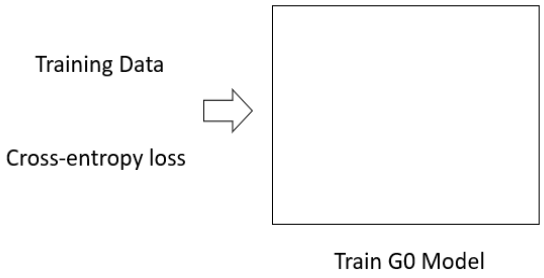


Figure 23.2: G0 Model

Aside

- Knowledge Distillation: use what's learned before as a source of labels
  → Take my training data and cross-entropy loss and train a model called generation zero model shown in Figure 2.

  → Consider Figure 3. Which is more true? The data points or the line? On one hand, the data points come from the actual world like experiments and have a truth value in that. However, the points are all noisy, and what you care about is the underlying pattern, and if you have successfully captured that, then the line is more true. The points are merely in the world and have all the hindrances of the real world. But, what you are truly looking for is the platonic reality of the model of the real world and the line is closer to that than the points. You ask the question, "maybe the outputs of my model are more true?". A good model is smoothing noise and other artifacts out of your data. Classification data is always corrupted. You know a cat is more dog than it is ice cream. But the labels did not reflect that at all. The labels only assign one class to each image. The scores of a classification model will presumably tell you that it is mostly a cat, but if I had to pick another class a good second choice would be a dog. What you learned from the G0 model is richer that your original training data.

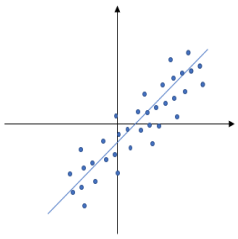


Figure 23.3: Which is more true?

→ Therefore, we can update the model with knowledge distillation loss like Figure 4.

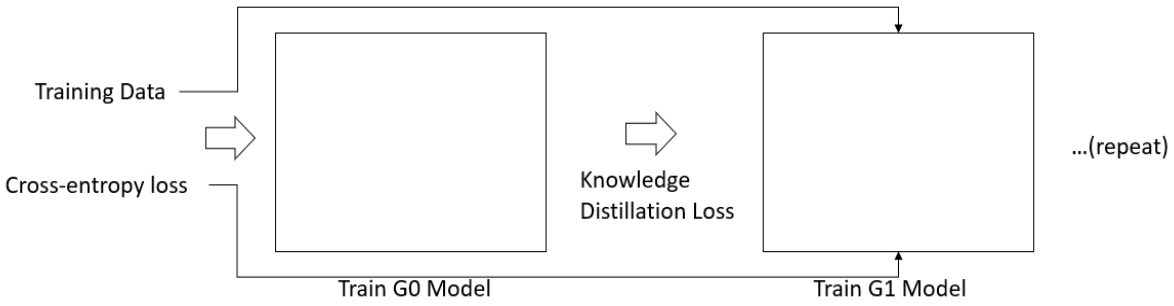


Figure 23.4: Iterations based on Knowledge Distillation

23.2 Meta-Learning

23.2.1 Multi-task Learning

Suppose we have a whole family of tasks, each with its own training data, and we want to find a system that can quickly/reliably learn new tasks. The new task is unseen.

Approaches to learn the new task:
1) Follow "feature extract" paradigm
2) Follow "fine tuning" paradigm

More details for "fine tuning" paradigm, what do we do on a new task?
a) Initialize our network with some parameters.
b) Do K-steps of SGD using our training data.
c) Evaluate on our hold-out set.
3) Follow nearest-neighbour paradigm
...

Consider the "fine tuning" paradigm, where we use MAML(Model-Agnostic Meta-Learning).

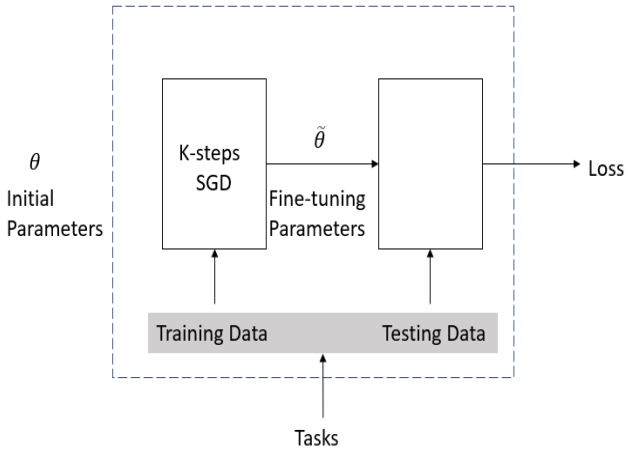


Figure 23.5: Framework of MAML for Learning the System

Let's just do SGD on the above problem in "fine tuning" paradigm shown in Figure 5. We split the tasks into training data and testing data. For this framework, the input is the initial parameters and the output is the loss. Therefore, we can do SGD for the system.

Questions:

- If we approach this according to Figure 5 with large K, could this lead to exploding or vanishing gradients?
  → Yes, it is a real concern. Therefore, we should keep K reasonable. We should sample each task many times and take different subsets of training data, since we can only afford to do so many steps of SGD.
- What is $\tilde{\theta}$?
  → These are the parameters fine tuned via SGD, which we can use on on testing data.

K-steps SGD itself in Equation 1 is differentiable, and we can view the equation like an RNN.

$$\theta_l[i] = \theta_{l-1}[i] + \eta * (-Gradient) \tag{23.1}$$

Questions:

- Comments
  → The goal of this algorithm is to find meta-parameter values such that optimal parameter values for each task are within 1 gradient step from the meta-parameter values. It seems unreasonable that we could find such meta-parameter values, as individual tasks likely have optimal parameter values that are far away from each other in parameter space. However, if we were to try taking multiple gradient steps and differentiating through them, this would create a more complex version of the algorithm that takes 1 gradient step. Therefore, we try taking 1 task-specific gradient step before exploring fancier algorithms. More generally, to create fancier algorithms, we should try what basic things first and get insights.
- What is the relationship between this approach and the second derivative, since second derivatives don't work on ReLU nonlinearities?
  → Yes, it looks like we do second derivative here. It turns out, however, since we use an approximation for the Hessian that only relies on taking multiple first derivatives, we never need to consider the second derivative of the ReLU function.

CS 282 Deep Neural Networks **Fall 2022**

Lecture 24: Meta-Learning

Lecturer: Anant Sahai *Scribe: Terrance Wang, Wyame Benslimane*

1 . Recap: MAML - Model Agnostic Meta-learning

1.1 What is the problem we are trying to solve:

In previous lecture, we discussed meta-learning. To put it simply, we are trying to learn algorithms that learn from other learning algorithms. Model-Agnostic Meta-Learning (MAML) extends this idea and tries to do multi-task learning by fine-tuning. Therefore we want to optimize for a pre-trained model that can adapt to a variety of tasks in a few gradient steps.

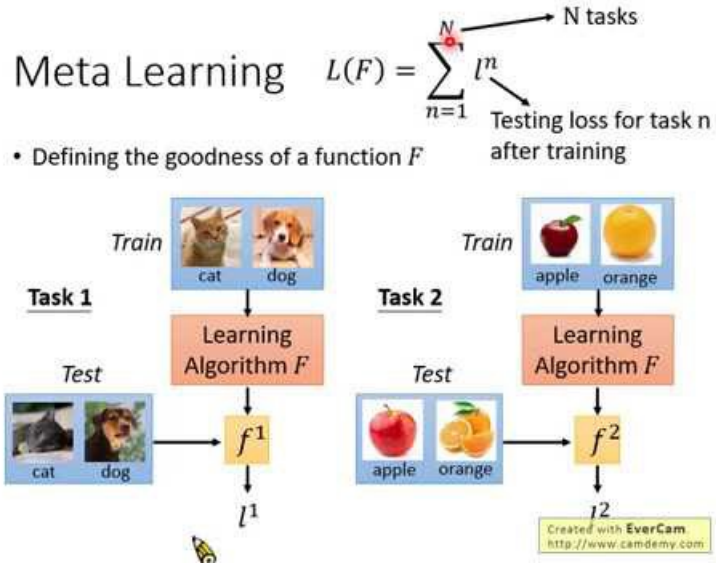


Figure 24.1: Meta Learning principle

1.2 What does the test-time look like?

The first step in this process is to start with a pre-trained model. Our choices are:

1. A model that already has a 'ready to modify' task head (see figure 24.2): In all deep models, the last block (linear layer) is responsible for converting the features learnt by the deep model to outputs for the problem the model is trying to solve. This block is what we call the task head. For example, in a classifier, this block is responsible for outputting the probability scores of different classes.
2. Start our pre-trained model with randomly initialized task specific head.

The next step in this process is to train the model using task specific training data, this can also be done in 2 different ways:

1. Fine tune the entire model.
2. Just train the task specific head.

Then we evaluate performance on task specific held out data. This step gives us a quantitative measure of how well we did.

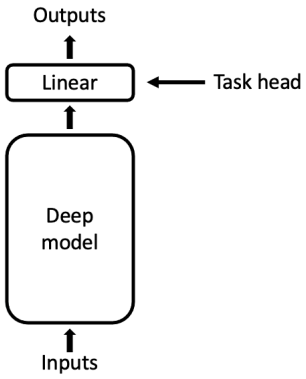


Figure 24.2: features generated by the deep model get fed into the task head to be converted to outputs.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Figure 24.3: MAML Pseudo-code Source

1.3 How do we optimize this problem?

In order to train any model, the first thing we need to have is **training data**. Therefore we need a lot of example tasks where each task has labeled training data. The next step is to use the standard approach for solving this problems in deep learning: SGD. This means that we are going to iterate the process below until it converges (or is stopped early):

1. Pick a mini-batch of tasks from training data.
2. Use the current parameters to evaluate the model on this batch (as though it was test time) and compute gradients using back-propagation.

3. Update the model's starting parameters with a step of size η_{outer} times the negative gradient.

This training procedure has a key difference with traditional model training. MAML requires us to evaluate the model on the held out set with a differentiable loss function. (e.g.: cross entropy loss instead of a binary right/wrong for a classification model) This is a necessary difference because instead of simply determining model performance on the held out set, we also want to update our initial weights. Furthermore, each task in the mini-batch is trained independently, and within a batch, each task starts with the same initial weights.

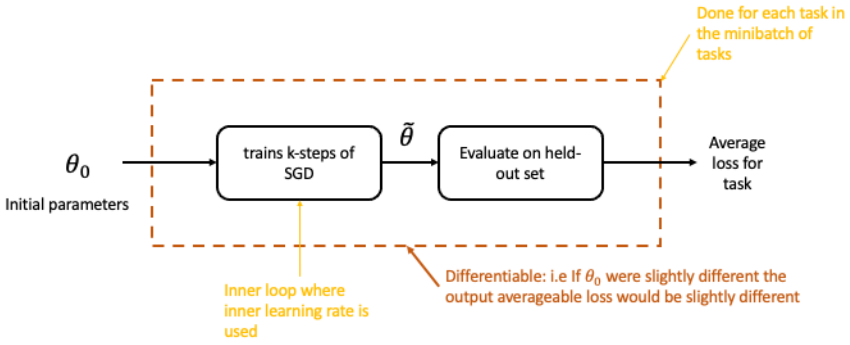


Figure 24.4: illustration of MAML training loop

For the previous figure 24.4, we can see that MAML trains 2 different loop. Remember that our training data points are different specific tasks. Therefore our 'inner loop' trains on task-specific labeled training data while the 'outer loop' trains on the different tasks.

Why do we need a differentiable loss function for MAML?

If we consider the block from the previous figure 24.4 where we train k-steps of SGD, the parameter $\theta_{t+1} = \theta_t + \eta_{inner}(-\Delta\mathcal{L}(\theta_t))$ is something that looks like an RNN because we are using k steps of SGD (See figure 24.7). Therefore we need the loss to be differentiable so that the gradient from the held out set can be passed back and used to update the initial weights.

It is important to differentiate between the two learning rates η_{outer} and η_{inner}, which correspond to the two training loops found in MAML. We can see that η_{outer} is used in the outer training loop, which tries to find a good initialization of the network and updates the model's starting parameters at the end of each mini-batch. η_{inner} is used in the inner loop, which updates the model for k steps on each mini-batch task before the gradient is calculated to update the initial parameters.

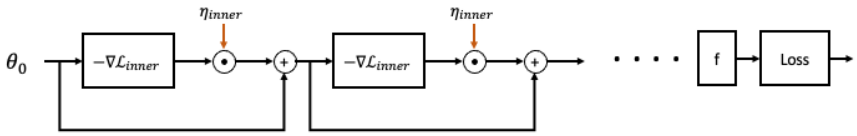


Figure 24.5: the unraveled inner loop resembles an RNN structure

1.4 What are the main challenges of MAML?

Exploding gradient during training: as k (the number of inner steps) increases, the network gets deeper, so exploding gradients become a problem.

Memory: large values of k require storing all the intermediate activations for back-propagation, and memory issues arise as a result.

In order to avoid these challenges, we can only used a limited k for our training tasks. Therefore at training time, k will small compared to the number of steps we will take to fine tune the model at test time.

Design choice:
If we are using randomly initialized task specific heads, when we resample a previously seen task, we can either use randomly initialize a new task specific head each time, or store and reuse the previously trained task specific head.

- the latter technique can be useful because the model will not be close to seeing all the data points in a given task within k inner loop steps.
- This can also be useful because the randomly initialized task head will not perform well at early iterations, and therefore will not be effective at correctly updating the model early on.

1.5 Step Back: Why does this work?

The first question to answer is: What are we learning?

- We are learning 'good features' for the deep network. Conventionally, we think of a deep neural network as an embedding that turns our input into a features. In this case the features are the outputs of the model that we are fed into the task specific head in order to get our predictions.
- We can also think of features as derivative features (tangent view). In this case, we consider the outputs of each layer of our deep network as features too. Therefore, when fine-tuning, we want these derivative features to let our model match the task quickly.
- Few shot fine tuning succeeds when the derivative features appropriately capture the task we're interested in. Because we are seriously overparameterized in few shot learning, we need the principal features of the tangent view do the work of capturing the task.

2 . Meta Learning Alternatives

Alternative options to MAML:

In practice, there are methods other than the MAML procedure described above that can be used for few shot learning. One option is to train the model on the union of all tasks. This means to simply run the standard supervised learning on a dataset that contains all data points from all the tasks, and train multiple task heads for each task at the same time. This alternative baseline method can do as well or better than MAML. One way to reason about why this works is that this is equivalent to MAML with k set to 0, or with the inner learning rate set to 0.

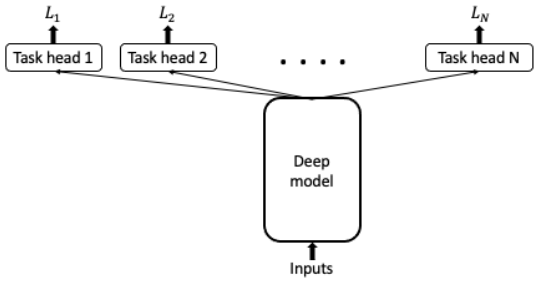


Figure 24.6: Task specific heads

Another alternative is to run MAML with a negative inner learning rate. This effectively makes the inner loop amplify the what's wrong in the model, and causes gradient descent in the outer loop to focus on parts of the model that are more wrong in the outer loop.

ANIL/Meta Opt Net/R2D2 approach

Motivation: The inner task head during training isn't very good when just initialized, so gradients aren't as helpful at improving the deep network as we might like.

Main idea: let's do a two phase approach

1. Freeze the "feature extractor network" and just optimize the linear task head.This is typically a convex problem with a closed form solution.
2. Now differentiate with respect to the parameters inside feature extractor.

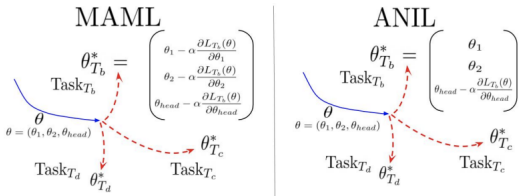


Figure 4: **Schematic of MAML and ANIL algorithms.** The difference between the MAML and ANIL algorithms: in MAML (left), the inner loop (task-specific) gradient updates are applied to all parameters θ, which are initialized with the meta-initialization from the outer loop. In ANIL (right), only the parameters corresponding to the network head θ_{head} are updated by the inner loop, during training **and** testing.

Figure 24.7: MAML vs ANIL Source

REPTILE - simpler than MAML

The inner loop of REPTILE is the same as MAML: take update from our initial parameters for k steps. But in the outer loop, don't take the derivative and just update parameters in the direction of the final weights from the inner loop.

CS 182/282A Deep Neural Networks **Fall 2022**

Lecture 25: 11/17/22

Lecturer: Anant Sahai *Scribe: Liam Tan*

25.1 Generative Tasks/Approaches

We will consider generation tasks, which come from the idea that if we can truly understand the underlying regularities of some data, the model should be able to generate a new unseen example of the data.

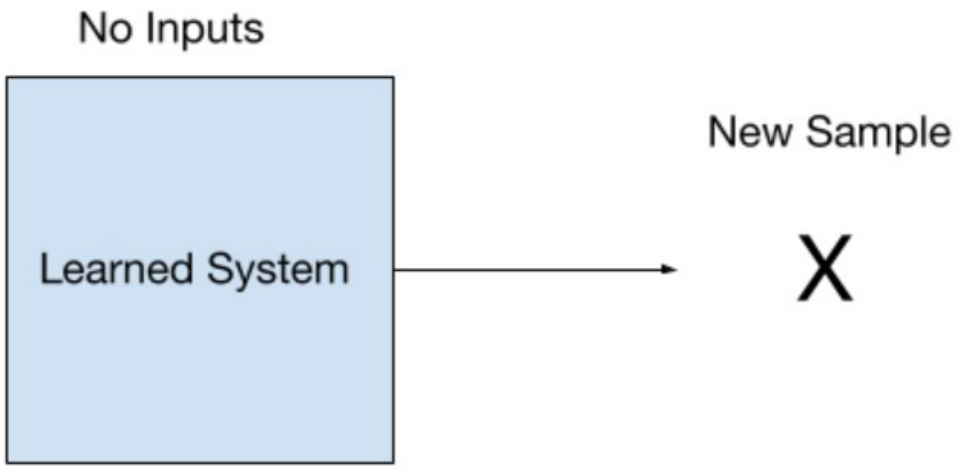


Figure 25.1: Basic Generation Model Framework

Take the system above. We have a learned generation system that takes in no inputs. Say it has learned to generate cats. When the model is called upon, we want it to generate a new cat. How can we ensure that a fresh image is generated each time, rather than a memorized image? We would like to have this system generate a different example each time. We do not want this to act as a deterministic circuit, one that cannot generate different things.

Thus, we will feed some sort of randomness to our system in order to ensure an identity map is not learned.

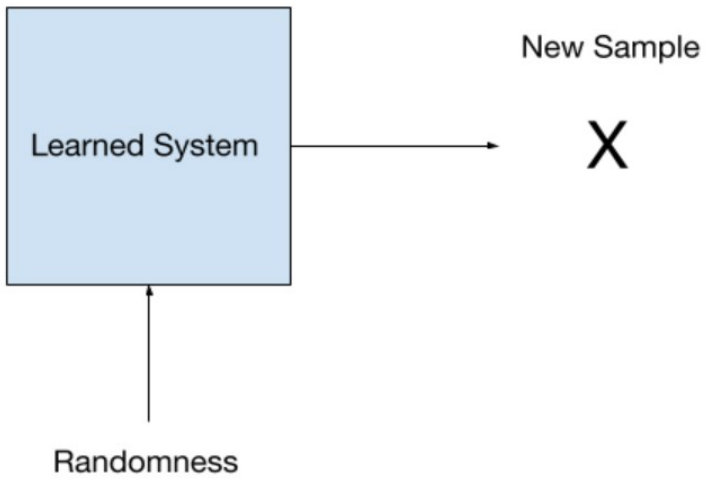


Figure 25.2: Generation Model Framework with Randomness

Given some randomness, we can generate new examples each time. Assume our learned system can generate fresh cat images each time it is run. What if we want to generate images of dogs, or images of planes?

Simple generation is not as useful as being able to control that generation. We turn to conditional generation.

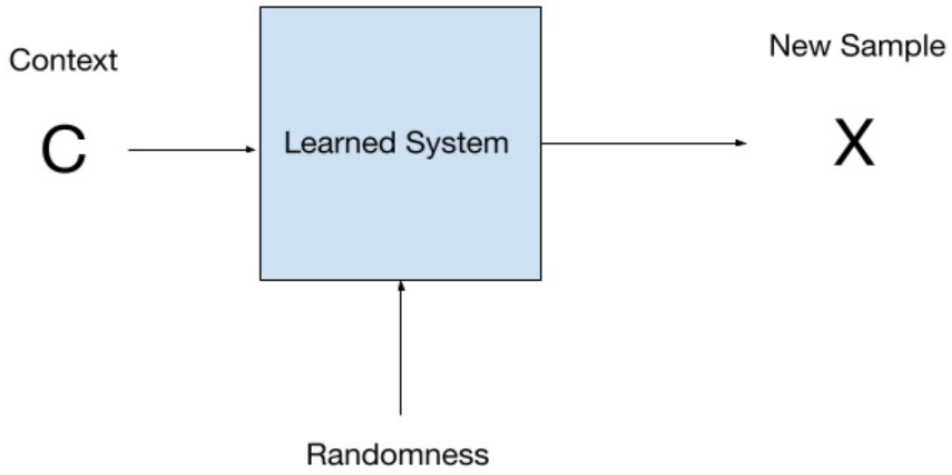


Figure 25.3: Conditional Generation Framework

Each time we run our generational model, we provide it some context (e.g tell the model to generate cats). Out comes a random sample that is correct in context. Say we tell it to generate an image of ice cream. It

will create a new image of ice cream.

Examples:

We give it an image of an object. The generational model outputs images of randomly sampled angles of that object.

Context can be some text. For example, we can feed it a label of an image. If we tell it to generate "cats playing poker" it will do so.

Questions:

- Why do we care about this? Isn't this just some novelty or a party trick?
  → There is a grand tradition of engineering and that is putting people out of work when machines can perform the task at hand
  → People are hired to construct 3D models for movies and video games and this is quite a laborious task. If generational models can do this, these designers are no longer needed.

  Generational models can aid in tasks where there is large amounts of human labor needed to create something.
  → Stock image generation. Photographers spend countless hours taking photos of simple objects for stock photos to be used in advertisements and such.

  With the rise of advanced generational models, photographers no longer need to spend that time taking and editing photos for that purpose.

- How can this be used for machine learning itself?
  We can use GANs for data augmentation.

  → We can use generational models to "create" more training data. These generated examples are viewed as a kind of data augmentation.

  They have learned a lot of regularity that you can then use to feed into your model.
  → Suppose we have very sensitive data that we do not want the model to memorize. We can use generational models for this.

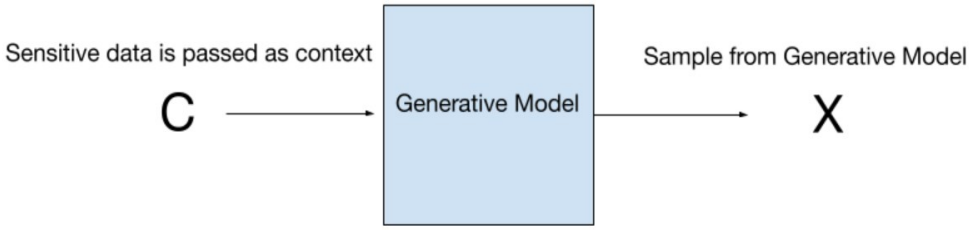


Figure 25.4: Data Sanitization/Privacy with Generational Models

We have a machine learning problem that requires us to learn from sensitive data. How can we make sure that our model does not memorize that data while still capturing the important underlying patterns within it? We can pass in our sensitive data into a generative model to create new data that can be passed into our machine learning model. If it is a good generative model and did not simply memorize things, we hope it does not output too much sensitive information. Thus we never have to give the sensitive data to someone that we may not trust.

25.2 Exploring and Designing Generational Models

25.2.1 Notes on Generation

Generation is a task unlike any previous task we've seen. It is very different than classification or regression. Currently, we've seen autocomplete for text generation and it can take a look at the previous word and try to guess the next one. More advanced autocompletes can take a look at all your words so far and try to predict the next words. Very advanced autcompletes can look at all the words so far and try to predict the next sentence. This is even being explored to see how these models can generate code. They take a look at what you are typing and suggest changes based on what it thinks you are trying to do. Given this rise in ability, researchers really want to understand generation tasks.

Generational Models is a phrase often used to capture the models that do the generation tasks. However, we must make the distinction between "model" and "task". Tasks refer to the different objective a system would like to achieve. This refers to regression, classification, and generation. There are a large scope of architectures that can do each task, so therefore when we refer to generation, we discuss the task, not the specific architecture that can do that task. For instance, transformers are useful in classification, as well as text generation(GPT). Generation is a task. The specific architectures that do these tasks are their own concepts.

25.2.2 Example with GPT

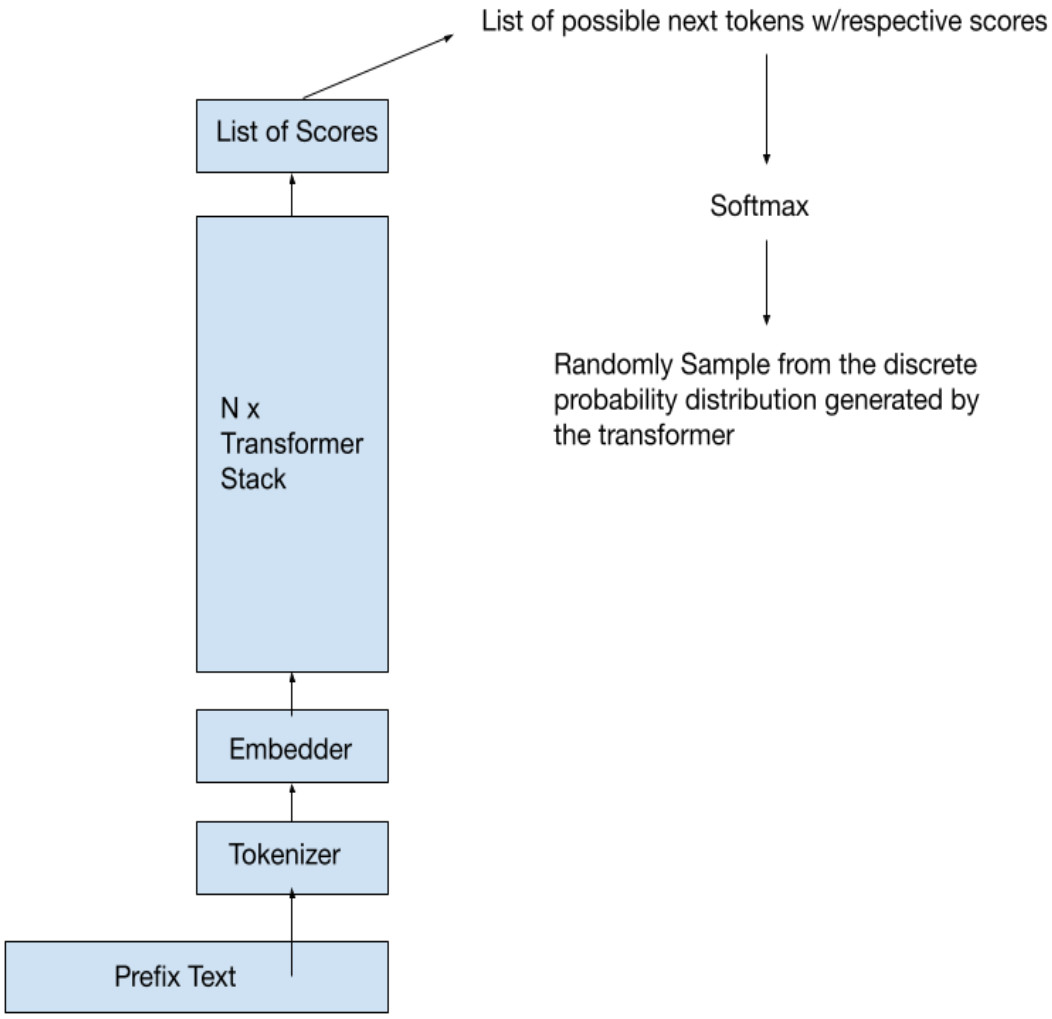


Figure 25.5: GPT for next token prediction

We use the GPT model for sentence generation. We provide a prefix text, which is the context in this scenario.

The prefix text will be accessed through the attention tables in each layer. As we pass input through the transformer stack, we observe a list of scores as output.

How can we use these scores to generate the next output?

We can take a softmax of all scores and according to this probability distribution, generate a sample for the next input. This sampling is the randomness in generation that allows us to create fresh examples each time.

Aside

- How do we actually sample from a discrete distribution?
  → A computer's random number generator will allow us to sample from $Unif[0,1]$, for instance. How can we use this to sample from a given discrete distribution?

Item	Probability	CDF
1	p_1	0
2	p_2	p_1
3	p_3	$p_1 + p_2$
4	p_4	$p_1 + p_2 + p_3$
...	...	...

Figure 25.6: Sampling from a discrete distribution

  → Let U be a random variable with distribution $Unif[0,1]$. Sample from U and look at what interval the realized value is in. Pick that item.

- How do we sample from a continuous distribution?

  → The main part we can take away from the discrete example is the CDF. Let's devise a strategy that utilizes the CDF.

  Let $F_X(x)$ be the CDF. $F_X(x) = P(X \leq x)$
  $P(X \leq x) = \int_{-\infty}^{x} p_X(x)dx$
  Sample U from $Unif[0,1]$. Solve $P(X \leq x) = u$ for x. $F(x) = u \implies x = F^{-1}(u)$
  Note that the CDF is monotonically increasing.

Aside Cont.

- How do we sample from a multidimensional $X \in R^d$ with density $f_X(x)$?
  $\rightarrow$ When we can do something for one dimension and want to try it for multiple dimensions, the first thing we must ask ourselves is "How can I turn this problem into a one dimensional problem?

  I know how to solve that, so if I can turn this into a one dimensional problem, I can solve this too". We can turn this into a repetition of one dimensional problems. We think, "maybe I can somehow factor the pdf into d different things".

  We can factor the pdf into a product of conditional probabilities. $f_X(\vec{x}) = f_{X_1}(\vec{x}[1]) * f_{X_2|X_1}(\vec{x}[2]|\vec{x}[1]) * f_{X_3|X_1,X_2}(\vec{x}[3]|\vec{x}[2],\vec{x}[1]) * ...$

  Start with $\vec{u}$ from i.i.d $Unif[0,1]$ d times. Use $\vec{u}[1]$ to get $\vec{x}[1]$ leveraging the marginal CDF $F_{X_1}(\vec{x}[1])$. Use $\vec{u}[2]$ and $\vec{x}[1]$ to find $\vec{x}[2]$, leveraging the conditional CDF $F_{X_2|X_1}(\vec{x}[2]|\vec{x}[1])$. Repeat to generate all d terms.

  The conditional generation, where we use the previous term to generate the next term is the Autoregressive generation used in GPT.

- How do we sample from any distribution when we are only provided a standard normal distribution?

  $\Phi(x)$ is the CDF of the Gaussian distribution.

  $\rightarrow$ Sample n from the standard normal $N \sim Normal(0,1)$. $\Phi(n) = u$. $\Phi(N) \sim Unif(0,1)$. This shows that if we have any continuous distribution, we can sample from any other continuous distribution. We take any arbitrary distribution and by applying the CDF, we can get a sample from a uniform distribution from 0 to 1. Now that we have a uniform distribution, we can sample any distribution we want with the methods from above. Any continuous function can simulate any discrete or mixed distribution.

  In general, X is a random variable with some arbitrary continuous distribution.

  Sample from X to realize the value x. Apply the CDF to x($F_X(x) = u$), and we now get a value u from 0 to 1. I claim that u is sampled uniformly from 0 to 1. Try to think about why this is true. Suppose I want to sample from a distribution with pdf $f_Y(y)$. Take $P(Y \leq y) = u$. Thus $F_Y(y) = u \implies y = F_Y^{-1}(u)$. y is the sample from our desired distribution.

- How can we use a discrete distribution to sample from a continuous distribution in spirit? We can also use more than just the discrete distribution.
  $\rightarrow$ We sample from the discrete distribution to decide which interval we are in. Then we use a uniform distribution to decide where in the interval we are.

- What are the interesting properties of the normal distribution?
  $\rightarrow$ If we add up many distributions, in limit the distribution becomes normal. This is the central limit theorem. In addition, adding up two independent gaussian distributions results in another gaussian distribution.

Say we want to sample images the same way GPT does. We want to use an autoregressive approach to mask out all future parts of the image and generate the image that way. Like GPT, we must do this in a one-by-one approach. This can either be one pixel at a time or one patch at a time.

The two main things we need in an autoregressive generation is randomness and an ordering of conditioning. The randomness comes from sampling our discrete distributions and the number of random variables is the

dimensionality of samples to generate. The ordering is used to compute the marginal distributions/CDF.

What is the main problem? With GPT next word prediction, we have some sort of ordering in which the words were passed into the transformer model, but with images, it is unclear as to how order the pixels/patches.

In comes raster scan ordering.

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	...	...	...	...	...	...	...	X

Figure 25.7: Raster Scan Ordering for Pixel Prediction

The pixels are ordered left to right top to bottom in raster scan ordering. Now we have an ordering and can train like GPT with a transformer model. This tends to work ok and is not bad. There are ways to train this with transformer models and even convnets. We can also do a similar approach patch by patch.

This raster scan approach is quite unintuitive because it does not capture the true topology of an image. Images are not exactly sequence data. This approach is not an intuitive method that comes from an idea of how images are structured. However, we use this because it tends to work somewhat.

Convnets and in turn weight sharing can also be utilized here but we must be careful about how to define the topology. What is near and what is far? In a convnet, the pixel right under the current pixel is classified as near. In raster scan, pixel 1 and 10 in figure 25.7 are 9 pixels away, which is quite far. Thus we need to include some sort of 2 dimensional positional encoding to encapsulate the properties of images.

With Convnets, we must think about how to bake in the inductive bias in the structure of the convolution. We have gradients and we need to mask out the gradients coming from the future data.

This method of masking and generating is not terrible, and there is some nice conditioning with the generation. The transformer allows context text with cross attention.

Images are large, and can often be millions of pixels. With the quadratic time complexity of attention, this is a problem. This style of generation is completely sequential and cannot be parallelized. Even with patching this method is considered to be quite slow.

25.2.3 Two Naive Approaches

25.2.3.1 Naive Approach #1

Due to the transformer approach being quite compute heavy, we look towards solutions that are faster.

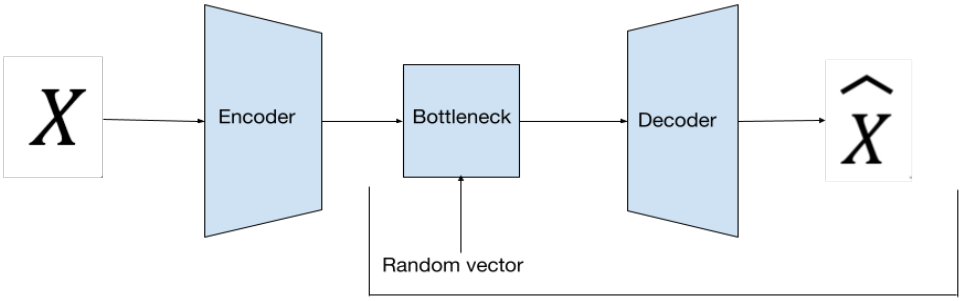


Figure 25.8: Utilizing an Autoencoder for Image Generation

We attempt to only use an autoencoder for image generation. We first train this autoencoder as a standard one. If it is a good autencoder, X and $\widehat{X}$ should be pretty similar.

We then take only the decoder side and feed in a random vector, hoping that it will generate a fresh image. This approach has a very fast runtime. However, the decoder outputs garbage. It turns out that there is still some structure in the bottleneck and that not all vectors in the bottleneck correspond to a real image when passed through the decoder.

Thus, we need to try to learn this structure, and therefore pass in gradients that guide us to learn that structure.

25.2.3.2 Naive Approach #2

Let's try using a classifier to help generate an image.

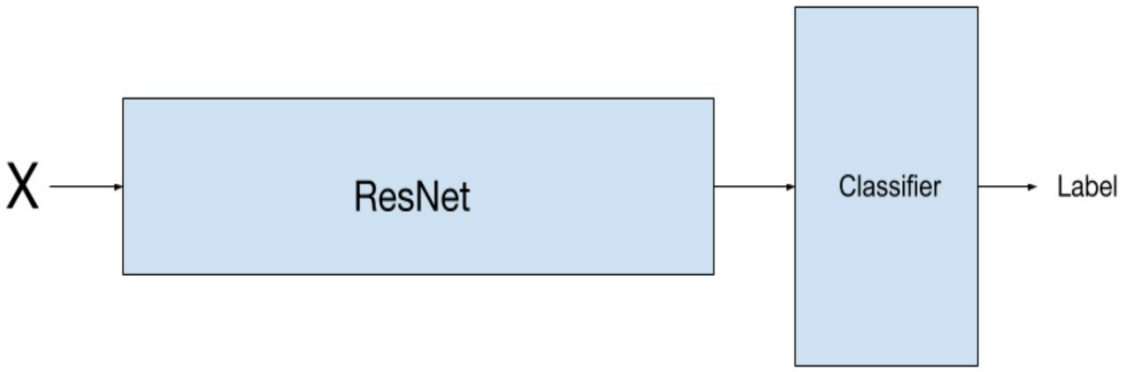


Figure 25.9: Utilizing Classifier for Image Generation

We start with random noise and pass that in. In each iteration, we want to make our image more catlike. Thus, we will do gradient ascent to make our score for the classification of a cat as large as possible

$X_t = X_{t-1} + \eta(\Delta Score_{cat}(X_{t-1}))$.

Thus we try to maximize the cat score and therefore we try to make our image look more like a cat each iteration. Every iteration we change our input image to make it have a higher cat score.

Suprisingly, our image output at the end of training looks like noise. We think, "Hrm, maybe we've ended up at a local minima and got unlucky". However, when we take a look at the classification scores/softmax scores, we see that the image is confidently classified as a cat! What happened here? "Hrm, perhaps it is because we started with random noise, which is so far from a real image that we could never possibly start with that image and get a real life image". Thus, we decide to pass in real images, say a picture of a table. We think, "maybe if we start with a real image like a table, the classifier can sort of work with the structure and go from there. We expect some modifications to the image and maybe a cat will appear on the table".

What was even more surprising was that after training this, the resultant X_{opt} looked like X and the scores said that this was a cat now! The image was somehow able to fool the classifier. This is called an adversarial example.

25.2.3.3 Generative Adversarial Networks

What if we combined these two approaches to train a classifier and a generator together during training?

Our generator would generate images, and our classifier would detect if the image was generated or if it was real. The classifier is now called the discriminator, because its job is to discriminate between real and fake images. The generator's job is to try to fool the discriminator into believing that the generated image is real. The loss will backpropagate and help train both parts of the model at the same time.

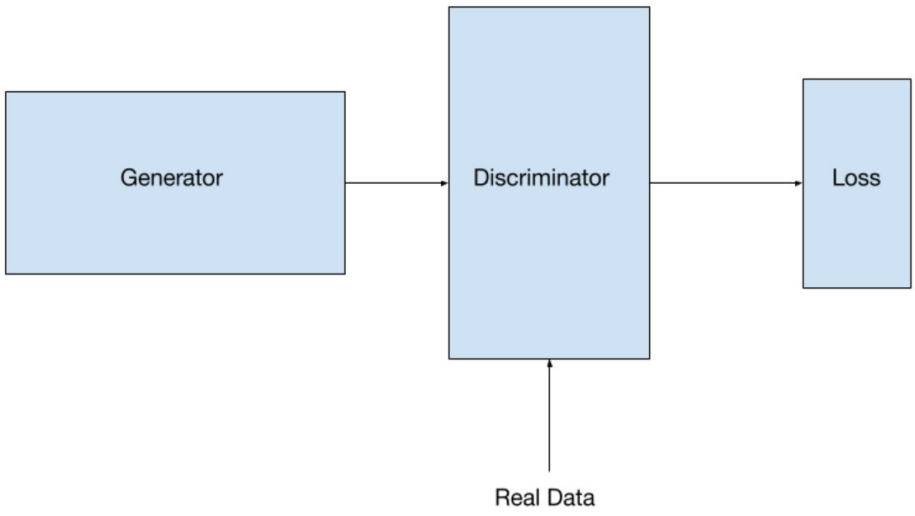


Figure 25.10: Basic GAN model

GANs are notoriously hard to train because they are so finicky and delicate. There has to be just the right alignment and balance for them to train well.

A common issue is called mode collapse. This is when a generator finds a decent output and continues to just produce that output. The discriminator soon learns to always reject that output, even if it could be real. A cat and mouse scenario happens and the generator is limited to producing a small subset of possible outputs. The next lecture goes into more detail about this.

See here for a visualization of GAN training: GAN Simulation

Figure 25.11: GAN Simulation

Live GAN training and even mode collapse can be observed.

25.3 References

References

[1] Minsuk Kahng, Nikhil Thorat, Polo Chau, Fernanda Viégas, Martin Wattenber. GAN Lab, "Play with Generative Adversarial Networks (GANs) in your browser!" In: (2019) DOI https://poloclub.github.io/ganlab/

CS 282A: Deep Neural Network **Fall 2022**

Lecture 26: November 22

Lecturer: Anant Sahai *Scribes: Zheyu Lu, Hyungki Im*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

26.1 Introduction

Generative models have not been around for a long time, but we have already seen waves of advances in this field, such as autoregressive (AR) models, generative adversarial network (GAN), and, more recently, the diffusion models. In this and the subsequent lectures, we will cover these frameworks with both high-level ideas and technical details.

In practice, we are usually interested in generating big objects, in which case we count on the fact that there is some kind of structure that is shared across the object. After all, what makes deep learning work is the idea of weight sharing, where something is replicated across different places in the objects of interest.

In the last lecture, we talked about different approaches for generation and one of them is the AR approach which takes the GPT style. Basically, it treats the object of interest as a sequence and samples one entry at a time conditioned on "past" information. Sampling consumes randomness, *i.e.,* randomness gets used every time for the generation of samples, and this is why we do not get the same result over and over again.

26.2 GAN Approach

26.2.1 GAN Architecture

Basically, the idea behind GAN approach is to leverage the fact that maybe we can tell real from fake and use this to help us generate real things. More concretely speaking, GAN approach uses a trained discriminator that can tell real examples of X from fake ones to train a generator that can produce realistic fakes. Figure 26.1 shows the GAN architecture and the architecture of the generator, and the discriminator that is used in GAN. The randomness comes into the generator as input and passes through a deep network structure, generating an example image X. The deep network used in the generator can contain any inductive biases that may help generate X. The discriminator takes this X (either real or fake) and classifies whether the X is real or fake. One of the advantages of being the discriminator being a deep architecture is that it actually learns the key discrimination between real and fake samples and tells this to the generator by backpropagation.

The core idea of the GAN approach is that the generator and the discriminator are trained together. However, it is very challenging to train the generator and the discriminator together, so we can split the training process into two steps. We discuss this strategy in the following two sections.

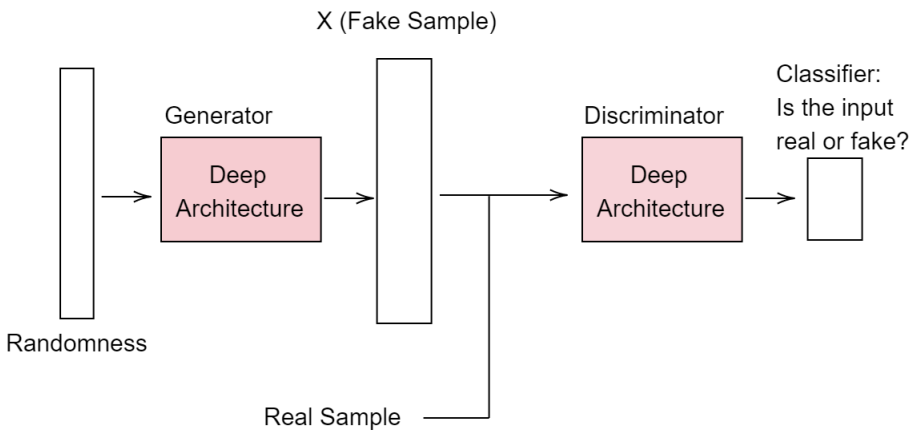


Figure 26.1: GAN Architecture

26.2.2 Step I: Train a Discriminator Given a Frozen Generator

In this step, we train the discriminator given a fixed generator. We can think of this as a classic binary classification problem where there are two categories: "real" and "fake". The frozen generator will generate some fake samples and we mix these fake samples with real samples and use these to train the discriminator.

26.2.3 Step II: Train a Generator Given a Frozen Discriminator

In this case, we can train the generator by inputting random samples (such as sampling from gaussian distribution) and obtain a "real" score or a "fake" score from the discriminator and take SGD steps. If the output of the discriminator is a "real" score, we should maximize it; otherwise, we should minimize it. Figure 26.2 illustrates this step.

Q. Why are we taking the gradient for the input, although it is random? The generator is not tuning to create a specific example that can fool the discriminator. Indeed, it tries to generate a general output that the discriminator can't easily distinguish from the real samples. That is the main reason why we are using the random input only once and throwing it away. This is different from the training procedure, which uses the same dataset again and again.

Now we can iterate between step I and step II to construct a generative model, and figure 26.3 illustrates a simplified version of this training process with a linear classifier as a discriminator.

However, there are several things that can go wrong in this setting. One possible flaw is if the discriminator is too good, then no matter what the generator does locally to its parameter, the discriminator would still say it is still too fake. Then this results in tiny gradients, which are not desirable. Also, if the target distribution is spread out instead of sticking together, it is highly possible for our generator to be oscillating between those target distributions. This is called the problem of mode collapse.

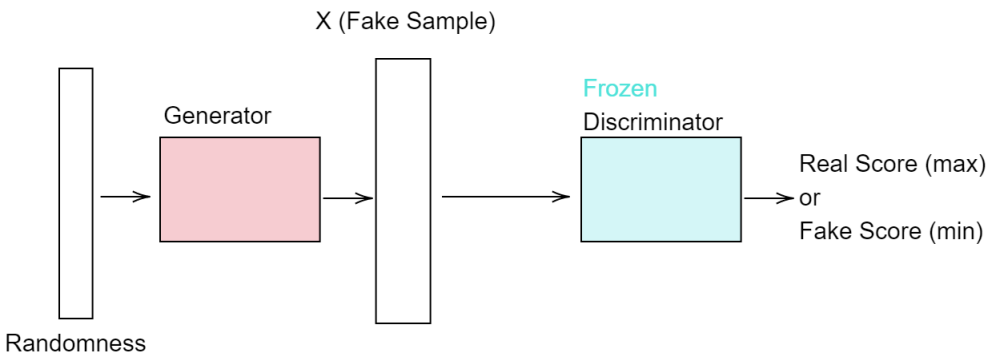


Figure 26.2: Train Generator with Frozen Discriminator

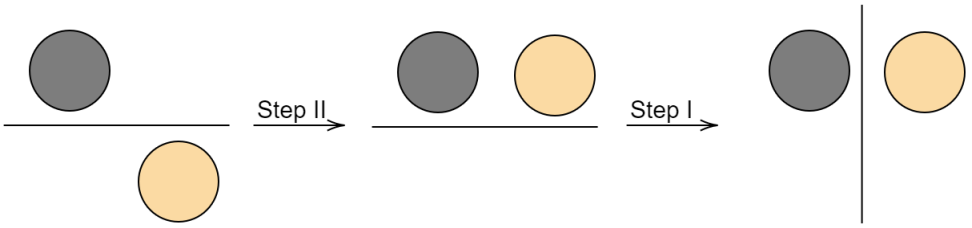


Figure 26.3: Iterating Step I and II: The black circle represents the distribution of real samples while the yellow circle represents the distribution of fake samples generated by the generator. We use a linear classifier as a discriminator and represented as a line in this figure.

26.2.4 Mode Collapse

GAN training is notoriously challenging to do because of the problem called mode collapse. This happens when the generator lacks sufficient diversity. Let's take a look at Figure 26.4.

As explained in the figure, the black dots represent the distribution of "real" data, and the orange cluster represents the distribution of the fake samples that are generated by the generator at some point. Our ultimate goal is to create orange clusters over all the real samples (black dots) in the figure. However, under most of the approaches, the generator will end up clustering at a single point of the real sample cluster just as the left figure in Figure 26.4. If this happens, the discriminator will then classify all the samples in the orange area (including the real samples) as fake. So, the generator will shift the orange area (fake samples) to other real sample clusters (In the Figure 26.4, we moved to counter clockwise.) Again, the discriminator will be trained to classify all the samples in the orange area at the right figure as fake samples, and this will continue on and on. This phenomenon is called mode collapse, and the reason for this naming is that some

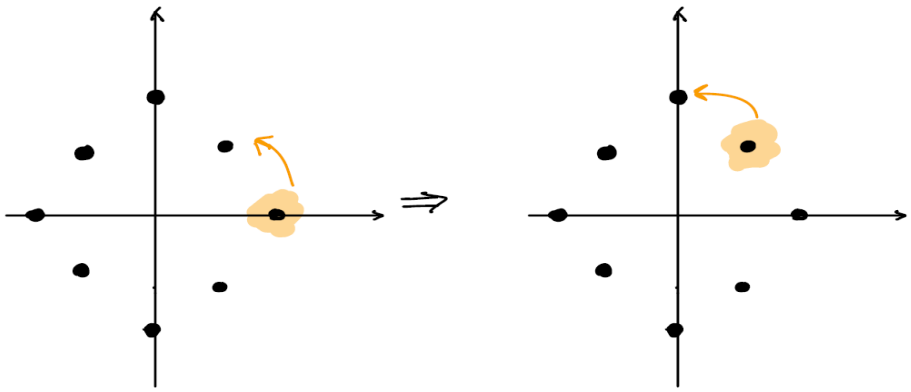


Figure 26.4: Black dots represent the distribution of real samples. The orange cluster represents the distribution of fake samples that are generated by the generator.

modes are not covered by the generator. Mode collapse is deeply rooted in this kind of adversarial learning setting, and this is an active area of research where many people are trying to come up with some methods to solve this issue.

26.3 Diffusion Approach

One of the interesting things about the revolution in deep learning is that sometimes people suddenly realize some specific math techniques that have existed for quite a long time are very important. In the context of diffusion models, these techniques are ordinary differential equations, stochastic differential equations, and thinking about things in continuous time. Before we formally dive into the details of diffusion models, in the following sections, we give two immature ideas that do not work in practice but serve as a good starting point.

26.3.1 First Attempt

Basically speaking, the idea is to use a denoising autoencoder to do generation as shown in Figure 26.5. The idea behind this is that denoising autoencoder has a true example X with noise, compresses X into the latent space and attempts to reconstruct X. However, this simple scheme will not work in practice because if you just add a huge noise to X, the input is dominated by the huge noise and effectively the whole framework becomes reconstructing X from pure noise where the shift is too large.

26.3.2 Second Attempt

Basically speaking, the idea is to do many stages of denoising, *i.e.,* add noise in stages rather than add a single stage of noise in the above first attempt. In this case, as is shown in Figure 26.6, we have a bunch of denoising autoencoders and correspondingly a bunch of targets to reconstruct. We have different noise at different stages.

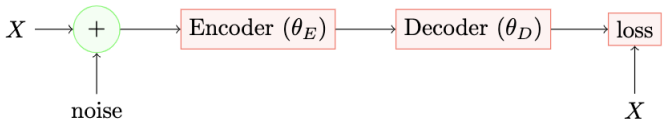


Figure 26.5: Schematics of the first idea where we have a single denoising autoencoder. We add noise to the input real value X and have one encoder with parameters θ_E, one decoder with parameters θ_D, and a loss layer trying to reconstruct X from the noisy data.

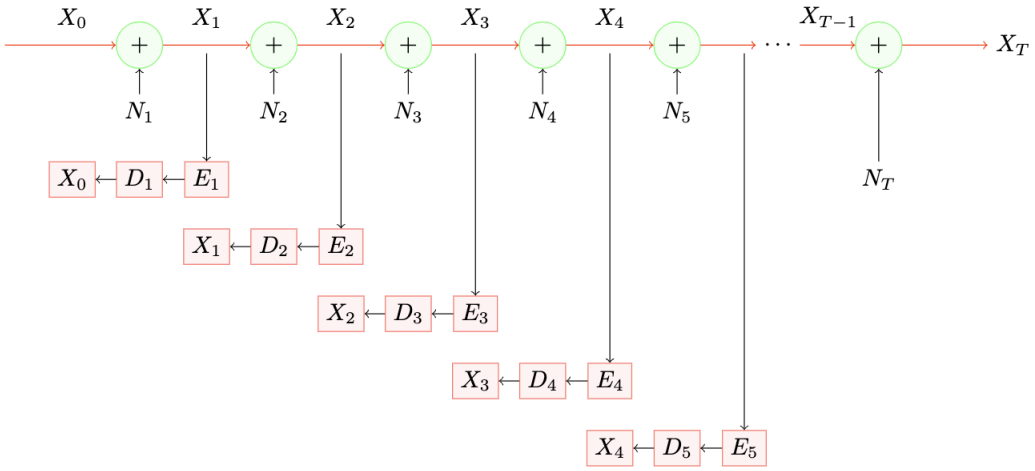


Figure 26.6: Schematics of the second idea where we have multiple denoising autoencoders. At each stage, we add noise $N_i \sim N(0,1)$ to the previous output X_{i-1} and have one encoder E_i and one decoder D_i trying to reconstruct the previous output X_{i-1} from the noisy data X_i.

Thinking about this abstractly is quite hard, so let us think of this using a concrete example on a two-dimensional plane as shown in Figure 26.7 where the true data X_0 are on a line. For the first denoising autoencoder block, we want to reconstruct X_0 from X_1 which means effectively the network will try to push points near the line onto the line, although we do not know what will happen for far away outliers since they may not exist in the training set. For the second denoising autoencoder block, it will also push X_2 towards the line on average although the points may be slightly far away from the line in some places. In general, in each of the denoising autoencoder block, the points will be pushed in the direction towards the line.

Notice that although we add noise with zero mean to X_0 and the resulting X_n has the same mean as X_0, the variance of X_n keeps growing. This is essentially the result of Brownian motion where $\langle x_t^2 \rangle \sim t$. And because the variance is growing over time, we want to standardize it which leads us to a modified design as shown in Figure 26.8 where at each step, we have

$$\begin{aligned} X_{i+1} &= \sqrt{1-\beta_i} X_i + \sqrt{\beta_i} N_i \\ \mathbb{E}[X_{i+1}] &= \sqrt{1-\beta_i}\mathbb{E}[X_i] + \sqrt{\beta_i}\mathbb{E}[N_i] = 0 \\ \mathrm{Var}[X_{i+1}] &= (1-\beta_i)\mathrm{Var}[X_i] + \beta_i \mathrm{Var}[N_i] = 1 \end{aligned} \tag{26.1}$$

Notice that in this design, the real value X_0 will keep shrinking towards 0 over time by a factor of $\prod_{i=1}^{n} \sqrt{1-\beta_i}$. As a result, we transfer whatever we have initially (X_0) to basically just noise at the end.

Also, it is worth noting here that although at each step we keep mean to be 0 and variance to be 1, the distribution is changing over time and we are not getting back to the start. Note that mean and variance alone could not uniquely define a distribution! This could also be illustrated by a two-dimensional example as shown in Figure 26.9. Initially, we have points staying on a circle, as we move forward by adding stages of noises, the circle becomes a cloud of points and the larger β is, the faster the blurring process is. As a result of the attenuation, the circle finally shrinks to a point and noises dominate everything.

Additionally, it is worth noting that standardization is just for controlling things from exploding and has nothing to do with the nature of the directionality of pushing towards the real value. For example, real images are not truly random and nearby pixels have closer values, but the noise will move them in totally different directions. Therefore, for a real image, any kind of divergences may be due to noise while commonalities are more likely due to real common features. Notice that even with these attenuation, the job of each denoising autoencoder block is still very clear, *i.e.,* remove the noise and reconstruct the input.

However, this approach actually still does not work in practice. We will see in the next lecture that how we could further modify the architecture and improve the training procedure to make it work.

26.4 What we wish this lecture also had to make things clearer?

1. It would be better if we provide some reasons why training the generator and the discriminator separately would not be efficient in an organized way.
2. It would be better to introduce some technical concepts to avoid mode collapse.
3. It would be better to introduce some knowledge of stochastic calculus and some technical details of the mathematical description about diffusion.
4. It would be better to add more technical details about the examples used in the lecture to illustrate the ideas behind diffusion models.
5. It would be better to explain more about why the two basic attempts do not work in practice.

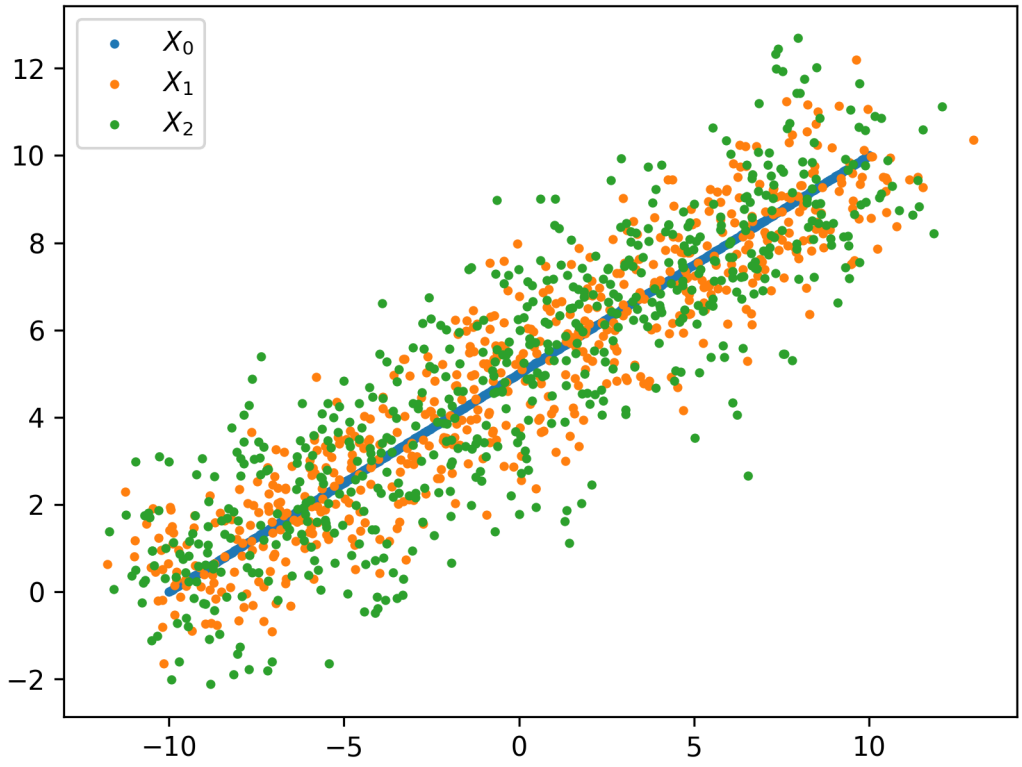


Figure 26.7: A two-dimensional example showing the effect of adding noise at different stages. X_0 are 600 points (x_0, y_0) on a line $y_0 = 0.5x_0 + 5$ ranging from -10 to 10. X_1 are the points (x_1, y_1) satisfying $x_1 = x_0 + n_x^{(1)}$ and $y_1 = y_0 + n_y^{(1)}$ where $n_x^{(1)} \sim N(0,1)$ and $n_y^{(1)} \sim N(0,1)$ are standard normal noise. X_2 are the points (x_2, y_2) satisfying $x_2 = x_1 + n_x^{(2)}$ and $y_2 = y_1 + n_y^{(2)}$ where $n_x^{(2)} \sim N(0,1)$ and $n_y^{(2)} \sim N(0,1)$ are standard normal noise. In this example, we have $\text{Var}(x_0) = 33.44, \text{Var}(y_0) = 8.36, \text{Var}(x_1) = 34.71, \text{Var}(y_1) = 8.95, \text{Var}(x_2) = 35.24, \text{Var}(y_2) = 10.22$. Notice that X_2 has larger variance than X_1 and X_1 has larger variance than X_0.

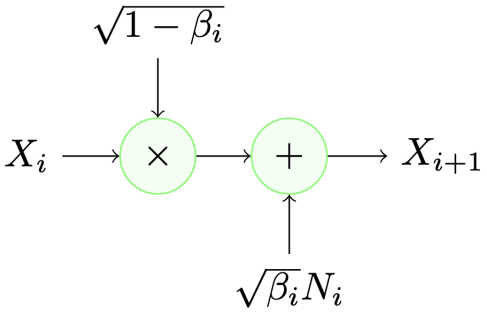


Figure 26.8: Schematics of the attenuation mechanism as a modification to each denosing autoencoder block in the second idea. For each stage, we standardize our result to make it have mean 0 and variance 1, *i.e.*, $X_{i+1} = \sqrt{1-\beta_i}X_i + \sqrt{\beta_i}N_i$ where $N_i \sim N(0,1)$, $\mathbb{E}[X_{i+1}] = 0$, and $\mathrm{Var}[X_{i+1}] = 1$.

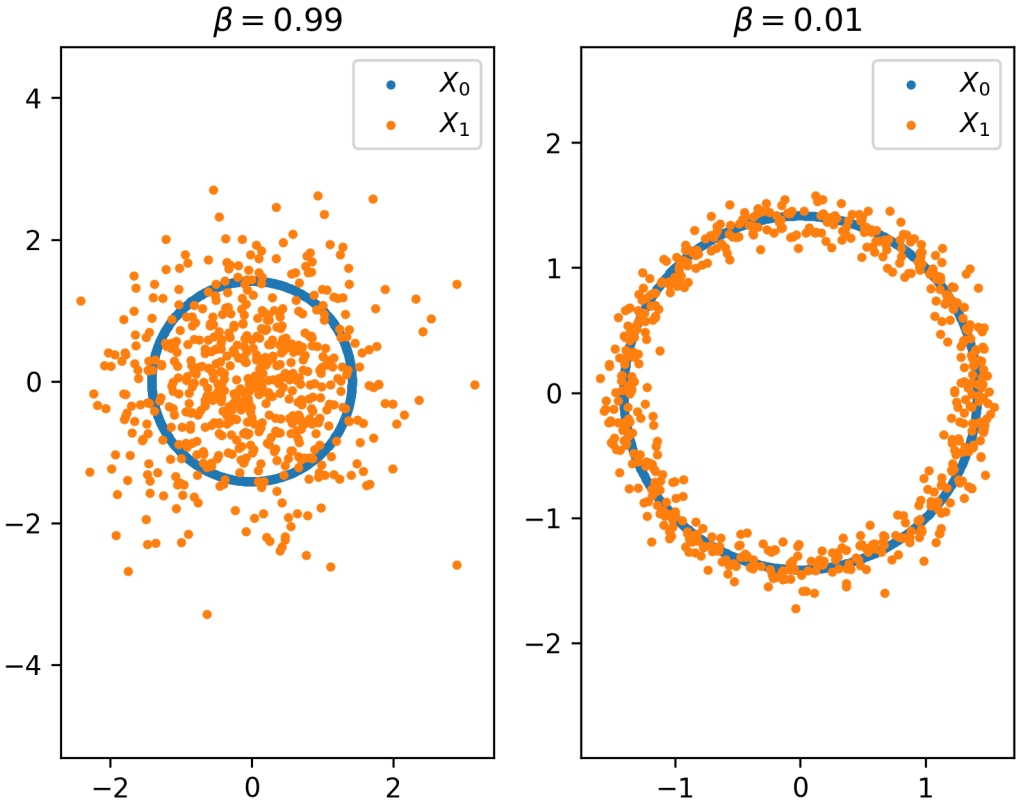


Figure 26.9: A two-dimensional example showing the effect of attenuation. X_0 are the points (x_0, y_0) on a circle centered at origin with radius $\sqrt{2}$. X_1 are the points (x_1, y_1) satisfying $x_1 = \sqrt{1-\beta}x_0 + \sqrt{\beta}n_x$ and $y_1 = \sqrt{1-\beta}y_0 + \sqrt{\beta}n_y$ where $n_x \sim N(0,1)$ and $n_y \sim N(0,1)$ are standard normal noise. On the left, we have $\beta = 0.99$. On the right, we have $\beta = 0.01$.

CS182 - Scribe Notes 11/29

Zipeng Lin, Numi Sveinsson

November 29th 2022

1 Introduction

Today the topic of the lecture are VAEs (variational autoencoders).

2 Recall from last time

Two ideas that don't quite work:

1. Plain autoencoder: we have an input X_i sent into the encoder and decoder to get $\hat{X}_i$ at training time. At testing time, we just use the encoder to generate the model. (bottleneck) The reason this might not work is because we need to decide from which distribution to sample the space $\vec{z}$ and we don't know that distribution.

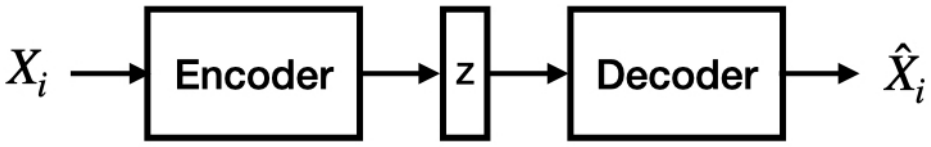


Figure 1: Autoencoder

   The distribution we input would be different from the output.

2. Plain repeated denoising generation. Multiply the data with $\sqrt{B_i}$, and then add a noise that is $N(0, 1 - B_i)$. To make the sequence long enough, we will get it would look Gaussian if T is large. Then we train a denoiser network that takes samples from this space X_T.
   Why does this fail: the samples from the space are very blurry after running through the denoisers. Each of the trained denoisers go a little outside the correct distribution and together it ends up being an accumulative error.

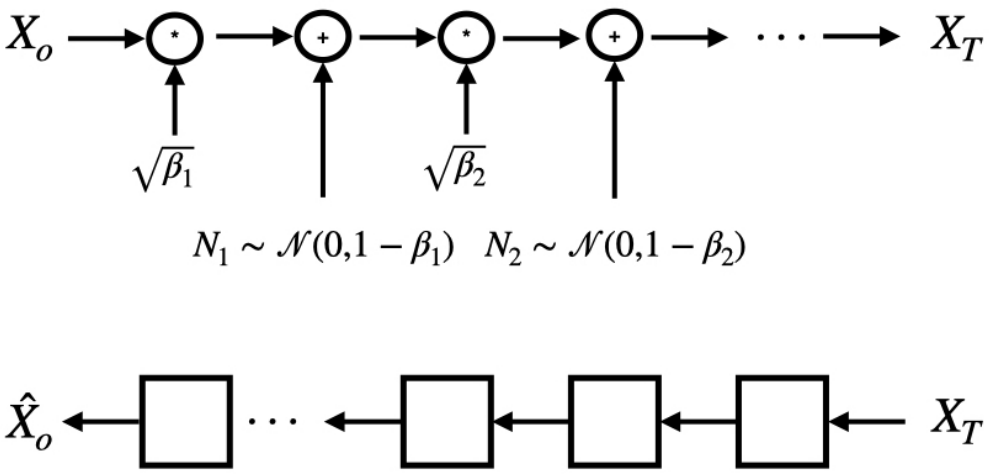


Figure 2: Denoise generation. The bottom row represents learned denoisers.

The distribution would be blurry too.

Now let's talk about how we fix the first approach above.

3 VAE: Variational Autoencoders

Intuition: decoder is just for training, encoder is for dimension reduction.
Question: from a hacking point of view, how to know the unknown distribution of $\hat{z}$?
Solution: force $\hat{z}$ to have a particular known distribution.

3 Ingredients:

1. Make the input to the "decoder" actually random during training.
2. Add a loss term on the distribution of $\hat{z}$. Make it look like what we want.
3. Parameterize so that we can do SGD.

For example, X will be sent to an encoder to a distribution of some kind and sent to loss and output a loss. Then, the sample from the distribution will be sent to a decoder and get $\hat{X}_i$, input the x_i to calculate loss and output $\mathbb{R}$.

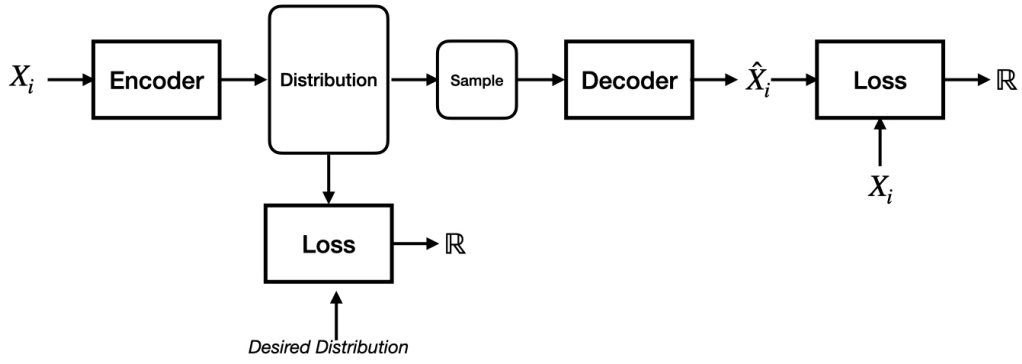


Figure 3: VAE

Questions: if we just take the input data, there are some amount of training data, we will get a collection of data point. If we treat the empirical data points as a distribution, we only get the reconstruction. We want to understand the distribution being generated. We can perturb those points at testing time. We want to have more diversity to emerge.

3.1 Desired Distribution

Desiderata for distribution:

1. Continuous
2. Easy to sample
3. Easy to compute the loss on distributions.

There are two candidates that we usually use: uniform distribution and Gaussian distribution. We choose normal distribution $N(\vec{0}, I)$.

Now we can just have the output from the encoder being a mean and covariance of that normal distribution. Now we have parameterized the distribution and can use SGD to change these variables.

Aside: we have the output being the squared root of the covariance to ensure positive definiteness later.

Sampling: is done by sampling a normal distribution $\hat{\epsilon}$, multiplying that with the covariance and adding to the mean.

3.2 Loss on Distribution

What loss should we use? How do we measure the distance between two probability distributions? There are lots of choices.

Choose KL divergence $KL(Q \mid P)$ "Relative entropy from P to Q", we have

$$KL(Q \mid P) = \int Q(z) \ln \frac{Q(z)}{P(z)} dz = E_{Z \sim Q}[\ln \frac{Q(z)}{P(z)}] \geq 0$$

this is nonnegative by Jensen's Inequality (proved in discussion) since ln is concave.

More importantly than being nonnegative, if two distributions are similar then it would be zero. It is also asymmetric. It means "it does not like it when Q puts lots of probabilities where P does not": this means if $P(z) \ll Q(z)$

the value would be really big. Follow the aside example to experience more about KL being asymmetric.

Aside: If we have iid draws from P, probability that it looks like they are drawn from Q is around $e^{-nD(Q|P)}$. If Q is a coin with only heads and P is half heads half tails, it is not going to happen. On the other hand, it might happen but not likely.

In particular, the KL divergence, $KL(Q \mid N(0, I_k)) = 1/2(\mathrm{Tr}(\Sigma_Q) + \vec{\mu_Q}^\intercal \vec{\mu_Q} - k - \log\det\Sigma_Q)$. It is important because PyTorch can take derivative for us. The gradient could flow back to the mean and covariance to update them. We try to regularize by imposing a loss term saying that please make the KL loss be normal.

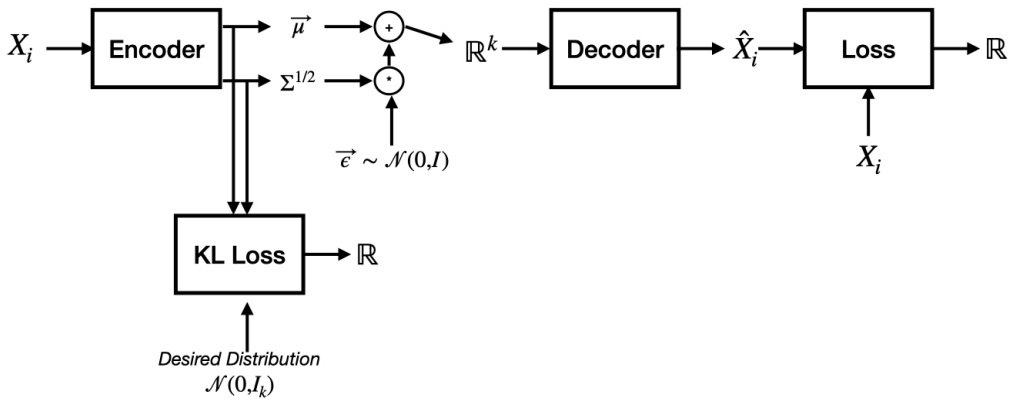


Figure 4: VAE - Enforcing Normal Distribution with KL-Divergence Loss

Question: can we do SGD? Does the loss affect the weights in the encoder? The spirit of SGD says we look at the entire distribution, while SGD says we take a sample and do it. We have x_i is random from training set. Gradient would hit the decoder, and would go towards $\hat{\mu}$ and $\Sigma^{1/2}$ because it would go through the multiplication. We can inject noise to the term D and add regularization term to make the system more robust.

3.3 Training

Challenges:

1. KL loss ends up dominating, we disconnect input. In this case the decoder carries no information, so it collapses. It would set "blurry average" outputs to D.

2. Alternatively, if the reconstruction loss dominates too much then we will get bad samples when we use it.

We need to have a balance between them.

3.4 Tricks

- Often reconstruction loss is allowed to dominate early in training and then distribution loss is used afterwards to tweak the sampling space without impacting reconstruction performance.

- During inference: the sample from $\vec{z}$ is run through decoder but then **rerun** through the encoder to refine the sample. That output is then used for generation. We only do this once or twice to prevent it from becoming 'mush'.

4 Diffusion Models with Denoising

Suppose we want x_t based on x_0

$$X_t \sim N(\sqrt{\alpha_t}X_0, (1-\alpha_t)I)$$

those are easy to understand and also for any s

$$X_t|X_s, s < t$$

What we want is to go backward for $X_{t-1}|X_t$, which is hard (this is intractable misbehavior).

Key idea 1: We approximate $X_{t-1}|X_t$ with $N(\mu_\theta(x_t,t), \Sigma_\theta(x_t,t))$. If we make those increments small enough, maybe the reverse thing might be the same. We want to do the same thing that we did VAE, we want to have a loss at the level of distribution that has what we want.

VAE-style idea: add a loss $KL(\dots \mid \dots)$ on this distribution.
We want to target something on the reverse path.

Key idea 2: $X_{t-1}|X_t, X_0$ is intractable by joint normality, then we will get a normal distribution with its means and variance. We can train the denoising autoencoder with using the same kind of regularization. Every time we do this we inject some noise. The added noise is what gives robustness.

Denoising is a form of feedback control.