CS182 Course Notes [Spring 2021]

Patrick Yin

Updated April 1, 2021

Contents

1 Note

The images used in this note are taken from Professor Sergey Levine's version of CS W182 / 282A: Designing, Visualizing and Understanding Deep Neural Networks. The course is linked here. These course notes are in progress. Also, (***) denotes that there is some material there that is not completely proven, which I may go back to later on to prove.

2 ML Basics

2.1 Risk

Definition 1 (Risk Function)**.** The risk function is the expected loss as a function of θ

$$R(\theta; f(\cdot)) = \mathbb{E}_{x\sim p(x), y\sim p(y|x)}[\mathcal{L}(x, y, \theta)]$$

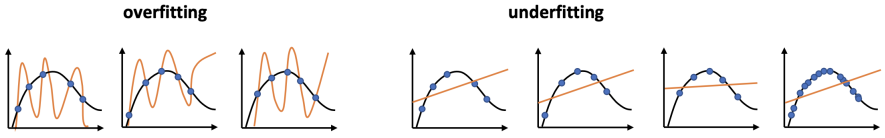
For instance, MSE is just risk with $\mathcal{L}(x, y, \theta) = (y - f_\theta(x))^2$. Since we don't know the true distribution in practice, we try to estimate this with empirical risk minimization.

Definition 2 (Empirical Risk)**.** The empirical risk is evaluated on samples from the true 2 distribution. It is given by

$$\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(x_i, y_i, \theta)$$

2.2 Bias-Variance Tradeoff

Overfitting occurs when empirical risk is low, but true risk is high. Underfitting occurs when empirical risk is high, and true risk is high. Below is a visual example. The black line is the true function, the dots are the datapoints, and the orange line is the learned function.



One thing to note is that the learned function looks different every time in the overfitted example but looks similar in the underfitted example. We can this having high and low "variance" respectively. This motivates what we call the bias-variance tradeoff.

Theorem 1 (Bias-Variance Tradeoff)**.** *The expected value of error w.r.t our data distribution* $\mathbb{E}_{\mathcal{D}\sim p(\mathcal{D})}[\|f_\mathcal{D}(x) - f(x)\|^2]$ *can be written as*

$$\mathbb{E}[\|f_\mathcal{D}(x) - f(x)\|^2] = \mathbb{E}[\|f_\mathcal{D}(x) - \mathbb{E}[f_\mathcal{D}(x)]\|^2] + \mathbb{E}[\|\mathbb{E}[f_\mathcal{D}(x)] - f(x)\|^2]$$

Proof.

$$\begin{aligned}\mathbb{E}[\|f_\mathcal{D}(x) - f(x)\|^2] &= \mathbb{E}[\|f_\mathcal{D}(x) - \mathbb{E}[\|f_\mathcal{D}(x)] + \mathbb{E}[\|f_\mathcal{D}(x)] - f(x)\|^2] \\ &= \mathbb{E}[\|f_\mathcal{D}(x) - \mathbb{E}[f_\mathcal{D}(x)]\|^2] + \mathbb{E}[\|\mathbb{E}[f_\mathcal{D}(x)] - f(x)\|^2]\end{aligned}$$

□

The left component is called variance, and the right component is the square of what we call bias. The variance, intuitively, is how much our prediction changes based on changing the dataset. The bias, intuitively, is the error that doesn't go away no matter how much data we have. What we have shown is that the expected error of model over all possible datasets can be broken down into Variance + Bias^2. If the variance is too high, we are overfitting. In the bias is too high, we are underfitting.

2.3 Regularization

Regularization is something we can add to the loss function to reduce variance (this will increase bias as well, so there is some tradeoff here).

2.3.1 Bayesian Interpretation

The bayesian interpretation of regularization is regarded as adding a prior on parameters. Intuitively, when we have high variance, our data doesn't give enough information about parameters. So we add in a prior into the loss function to help give it information to disambiguate between what, to model, seems to be equally good models. More formally, we want to maximize the probability of parameters, θ, given the data, $\mathcal{D}$. Using bayes rule, we know that

$$p(\theta|\mathcal{D}) = \frac{p(\theta, \mathcal{D})}{p(\mathcal{D})} \propto p(\theta, \mathcal{D}) = p(\mathcal{D}|\theta)p(\theta)$$

Since we trying to maximize this probability, this is equivalent to minimizing the negative log probability. More formally,

$$\begin{aligned}
\underset{\theta}{\text{argmax}}\, p(\theta|\mathcal{D}) &= \underset{\theta}{\text{argmax}}\, p(\mathcal{D}|\theta)p(\theta) \\
&= \underset{\theta}{\text{argmin}} -\log\, p(\mathcal{D}|\theta) - \log\, p(\theta) \\
&= \underset{\theta}{\text{argmin}}\, \mathcal{L}(\theta) \text{ s.t. } \mathcal{L}(\theta) = -(\sum_{i=1}^{N} \log\, p(y_i|x_i, \theta)) - \log\, p(\theta)
\end{aligned}$$

Here $p(\theta)$ is our prior. Is it possible to make a prior that makes the function overfit less (i.e. smoother)? A simple idea would be to let $p(\theta) = \mathcal{N}(0, \sigma^2)$ since the normal distribution assigns higher probabilities to small numbers. This would allow small parameter values. Intuitively, we don't want large parameter values since these typically lead to non-smooth and thus overfitted predictions. Going along the assumption that the parameters are normally distributed and uncorrelated, we see that

$$\begin{aligned}
\log\, p(\theta) &= \sum_{i=1}^{D} -\frac{1}{2}\frac{\theta_i^2}{\sigma^2} - \log\, \sigma - \frac{1}{2}\log\, 2\pi \\
&= -\lambda\|\theta\|^2 + \text{const s.t. } \lambda = \frac{1}{2\sigma^2}
\end{aligned}$$

Here, λ is our hyperparameter and our new loss function becomes

$$\mathcal{L}(\theta) = -(\sum_{i=1}^{N} \log\, p(y_i|x_i, \theta)) - \lambda\|\theta\|^2 \text{ if } \Sigma_\theta(x) = \mathbf{I}$$

3 Gradient Descent

3.1 Gradient Descent

To minimize a loss function, one idea is to go the direction of the gradient.

Algorithm 1 Gradient Descent

1: $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathcal{L}(\theta_k)$

The negative log likelihood of logistic regression is guaranteed to be convex. As a result, "all roads lead to Rome" in the sense that gradient descent will always converge to the global minimum given some small enough learning rate. However, the loss surface of a neural network is not as nice. There are local optima that gradient descent can get stuck in, plateaus (i.e. virtually flat lines) that trainers using a small learning rate can get stuck in, and saddle points that the neural network can get stuck in due to small gradients (in fact, in high dimensions most critical points are saddle points). So, the direction of steepest descent may not always be the best way to go.

3.2 Newton's Method

Newton's method gives a better descent direction than gradients, but Newton's method is too computationally expensive to practically use. Nonetheless, it is an ideal to strive for. The derivation for Newton's method comes from the fact that the multivariate Taylor expansion of the loss function is

$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta_0) + \nabla_\theta \mathcal{L}(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^T \nabla_\theta^2 \mathcal{L}(\theta_0)(\theta - \theta_0)$$

If we set the derivative to 0 and solve, we have a new descent algorithm:

Algorithm 2 Newton's Method Descent

1: $\theta_{k+1} \leftarrow \theta_k - \alpha(\nabla_\theta^2 \mathcal{L}(\theta_k))^{-1} \nabla_\theta \mathcal{L}(\theta_k)$

While gradient descent runtime is O(n), newton's method descent is O(n^3), making it not viable for neural network optimization.

3.3 Gradient Descent Optimization Algorithms

3.3.1 Momentum

Intuitively, if successive gradient steps point in different directions, we should cancel off the directions that disagree. Conversely, if the gradient steps point in the same direction, we should go faster in that direction. This motivates that idea of momentum. We are essentially blending in the previous directions into the current gradient step.
In practice, this method brings some benefits of Newton's method (which takes into account second derivative) at virtually no cost.

Algorithm 3 Momentum

1: $g_k = \nabla_\theta \mathcal{L}(\theta_k) + \mu g_{k-1}$
2: $\theta_{k+1} \leftarrow \theta_k - \alpha g_k$

3.3.2 RMSProp

Intuitively, the sign of the gradient tells us which way to go along each dimension, but the magnitude may be misleading. Even worse, the overall magnitude of the gradient can change drastically over the course of optimization, making learning rates hard to tune. What if we normalized out the magnitude of the gradient along each dimension? In RMSProp, we estimate the per-dimension magnitude of the gradient by taking a running average and normalizing the descent step by this value:

Algorithm 4 RMSProp

1: $s_k = \beta s_{k-1} + (1-\beta)(\nabla_\theta \mathcal{L}(\theta_k))^2$
2: $\theta_{k+1} \leftarrow \theta_k - \alpha \frac{\nabla_\theta \mathcal{L}(\theta_k)}{\sqrt{s_k}}$

Note that the square in line 1 and the division is line 2 is element-wise.

3.3.3 AdaGrad

AdaGrad is like RMSProp, but estimates per-dimension cumulative magnitude rather than per-dimension magnitude:

Algorithm 5 AdaGrad

1: $s_k = s_{k-1} + (\nabla_\theta \mathcal{L}(\theta_k))^2$
2: $\theta_{k+1} \leftarrow \theta_k - \alpha \frac{\nabla_\theta \mathcal{L}(\theta_k)}{\sqrt{s_k}}$

AdaGrad has some appealing guarantees for convex problems since the learning rate effectively decreases over time. But this only works if we find the optimum quickly before the rate decays too much. On the other hand, RMSProp tends to be much better for deep learning and most non-convex problems.

3.3.4 Adam

The basic idea of Adam is to combine momentum and RMSProp.
m_k, the first moment estimate, is a momentum-like estimate of the gradient.
v_k, the second moment estimate, estimates the magnitude of the gradients like in RMSprop. Since $m_0 = 0$ and $v_0 = 0$, the steps early on will be very small. To ameliorate this, we blow up the values of m_k and v_k early on on lines 3 and 4. Notice as that k goes to infinity, the denominator becomes 1, so the gradient blows up less and less. In the last line, we add an epsilon to prevent divison by

Algorithm 6 Adam

1: $m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_k)$
2: $v_k = \beta_2 v_{k-1} + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta_k))^2$
3: $\hat{m}_k = \frac{m_k}{1-\beta_1^k}$
4: $\hat{v}_k = \frac{v_k}{1-\beta_2^k}$
5: $\theta_{k+1} \leftarrow \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k}+\epsilon}$

zero. Some good default settings are $\epsilon = 10^{-8}$, $alpha = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$.

3.4 Gradient Descent Variants

3.4.1 Batch Gradient Descent

Normally, for gradient descent, every update we must compute the gradient of

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log\, p_\theta(y_i|x_i)$$

The issue with this is that computing and summing over all datapoints can be expensive during training if the dataset is large.

3.4.2 Stochastic Gradient Descent

Instead, we could just train on one datapoint at a time. If we sample randomly, this is still an unbiased estimator. The issue with stochastic gradient descent is that training on a single datapoint can have high variance, leading to gradient updates that can get unstable.

3.4.3 Mini-batch Gradient Descent

The compromise is to train on batches of data at a time. From our dataset, $\mathcal{D}$, we can sample a smaller batch $\mathcal{B} \subset \mathcal{D}$ and estimate $g_k \leftarrow -\frac{1}{B}\sum_{i=1}^{B} \log\, p_\theta(y_i|x_i)$ for gradient descent: $\theta_{k+1} \leftarrow \theta_k - \alpha g_k$. Each iteration samples from a different minibatch. In practice, instead of randomly sampling every iteration, we can shuffle the dataset in advance and construct batches out of the consecutive groups of $|\mathcal{B}|$ datapoints.

4 Neural Networks

4.1 Description of Neural Networks

I am largely going to glimpse over the description of vanilla neural networks. They are really just computational graphs that take in an input and output a prediction. They are also differentiable so we can take gradients and update the weights with gradient descent. For instance, a one-layer neural network would take in input x and output $\sigma(Wx+b)$ where σ is an activation function such as a ReLU and W and b are learned parameters through gradient descent.

4.2 Backpropagation

Backpropagation is an efficient way to find gradients for a neural network. The algorithm looks like this:

Algorithm 7 Backpropagation

1: Forward pass: calculate each $a^{(i)}$ and $z^{(i)}$ (i.e. run your input through the neural network to get the intermediate values and output)
2: Backward pass: Initialize $\delta = \dfrac{\mathrm{d}\mathcal{L}}{\mathrm{d}z^{(n)}}$
3: **for** each f with input x_f and parameters θ_f **do**
4: $\quad \dfrac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta_f} \leftarrow \dfrac{\mathrm{d}f}{\mathrm{d}\theta_f}\delta$
5: $\quad \delta \leftarrow \dfrac{\mathrm{d}f}{\mathrm{d}x_f}\delta$
6: **end for**

Then we just need to perform gradient descent on all the parameters to get their updated values.

4.3 Batch Normalization

We want all entries of our input to be roughly on the same scale or else gradients in the larger inputs will dominant the small ones. To solve this problem, we can standardize our inputs. One way is just to transform our inputs so they have $\mu = 0$, $\sigma = 1$. In an equation, this would be

$$\bar{x}_i = \frac{x_i - \mathbb{E}[x]}{\sqrt{\mathbb{E}[(x_i - \mathbb{E}[x])^2]}} \text{ s.t. } \mathbb{E}[x] \approx \frac{1}{N}\sum_{i=1}^{N} x_i$$

We're standardize the input, but how do we standardize the activations? This is where batch normalization comes in. For each activation layer $a^{(i)}$,

$$\mu^{(i)} \approx \frac{1}{B}\sum_{j=1}^{B} a_j^{(i)}$$

$$\sigma^{(i)} \approx \sqrt{\frac{1}{B}\sum_{j=1}^{B}(a_j^{(i)} - \mu^{(i)})^2}$$

$$\bar{a}_j^{(i)} = \frac{a_j^{(i)} - \mu^{(i)}}{\sigma^{(i)}}\gamma + \beta$$

where γ and β are learnable parameters. The reason why we do batch normalization over normalizing over all the data is because normalizing over all the data every activation layer is very expensive. By doing batch norm, we are able to standardize our activations cheaply. Our learnable parameters help scale our activations achieve more expressive behavior. For test time, we first take the mean and variance of our training data, freeze these values, and use them for testing.

4.4 Weight Initialization

We want to initialize our weights such that they are not too big or too small, so that the gradients propagate well.

4.4.1 Normal Initialization

A simple choice would to be initialize all weights according to $\mathcal{N}(0, .0001)$. The issue with this is that the magnitude of the activations exponentially decay with more layers. This is bad because if activations zero, then gradients are zero.

4.4.2 Xavier Initialization

Let's say we still initial our weights with $\mathcal{N}(0, \sigma_W^2)$ and our bias to around 0. Then $z_i = \sum_j W_{ij}a_j + b_i \approx \sum_j W_{ij}a_j$. Assuming that $a_j \sim \mathcal{N}(0, \sigma_a)$ (this is reasonable since $x \sim \mathcal{N}(0, 1)$). Then, $\mathbb{E}[z_i^2] = \sum_j \mathbb{E}[W_{ij}^2]\mathbb{E}[a_j^2] = D_a\sigma_W^2\sigma_a^2$ where D_a is dimensionality of a. If we select $\sigma_w^2 = \frac{1}{D_a}$, then $\mathbb{E}[z_i^2] = \sigma_a^2$. In other word, the variance of the next layer's activations are the same as the current layer's activations. This is good because the scale of activations doesn't increase or decrease as we increase the number of layers (based on the strong assumptions we've made) if we initialize the weights as $\mathcal{N}(0, \frac{1}{\sqrt{D_a}})$. This is called Xavier initialization.

4.4.2.1 Xavier Initializations for ReLUs For Xavier initializations on ReLUs, the negative half of 0-mean activations are removed so the variance is cut in half. In other words $\sigma_a'^2 = \frac{1}{2}\sigma_a^2$. So in order for $\mathbb{E}[z_i^2] = \sigma_a'^2$, we have to set $\sigma_w^2 = \frac{2}{D_a}$ (i.e. $\sigma_w = \frac{1}{\sqrt{\frac{1}{2}D_a}}$).

4.4.3 Bias Initialization

If we initialize biases at zero, with a ReLU, half our units will be dead on avaerage. So instead we can initialize our biases to some small positive constant such as 0.1.

4.4.4 Orthogonal Random Matrix Initialization

A more advanced weight initialization method tries to make the eigenvalues of the Jacobians be close to identity. Because this way, when we do backpropagation, our gradient isn't doesn't explode and vanish. To do this, we can generate a random weight matrix from a zero-mean normal distribution. Then we just do singular value decomposition on the matrix and use the left or right orthogonal matrix for our initialization depending on which one has the shape we need.

4.5 Gradient Clipping

With exploding gradients, where our trainer takes a gradient step too big, this can lead us very astray from the minimum we want to reach. This can happen when something gets divided by a small constant (e.g. in batch norm, softmax, etc.). One hack to ameliorate this issue is that use gradient clipping. Per-element clipping keeps each element of the gradient between a certain range (i.e. $\bar{g}_i \leftarrow \max(\min(g_i, c_i), -c_i)$). Another method, norm clipping, normalizes the gradient if it is above a certain threshold in magnitude (i.e. $\bar{g}_i \leftarrow g\frac{\min(\|g\|,c)}{\|g\|}$). To figure out c, we can train for a new epoch to see the magnitude of healthy gradients.

4.6 Ensemble Learning

From bias-variance tradeoff, our variance is $\mathbb{E}[\|f_{\mathcal{D}}(x) - \mathbb{E}[f_{\mathcal{D}}(x)]\|^2]$. If we estimate $\mathbb{E}[f_{\mathcal{D}}(x)] \approx \frac{1}{M}\sum_{i=1}^{M} f_{D_j}(x)$, maybe we find a good estimate for the expected value and reduce variance, thus decreasing our overall error.

4.6.1 Simple Ensemble Learning

A simple approach is to chop a big dataset into N overlapping but independently sampled parts, train N seperate models, and then use these models to make a prediction. A principled approach would be to average their predictions, so $p(y|x) = \frac{1}{M}\sum_{j=1}^{M} p_{\theta_j}(y|x)$. A simple approach would be just to take majority vote. In practice, is there is already a lot of randomness in our training already (random initialization, minibatch shuffling, stochastic gradient descent, etc.), we can just train N models on the same dataset and then either average their predictions or do a majority vote.

4.6.2 Faster Ensemble Learning

Faster ensemble learning would be for our multiple policies to learn from the same features. For sample, in an image task, we can get features out of an image from a convnet. And then from those features, we can do ensemble learning (versus doing ensemble learning end-to-end). In other words, we have N heads after the convnet.

4.6.3 Fasterer Ensemble Learning

A even faster and cheaper way to do ensemble learning is to save parameter snapshots over the course of SGD optimization and use each snapshot as a model in the ensemble. In this case, we can still average probabilities or vote. But we also have the third option of just averaging the parameter vectors together.

4.6.4 Dropout (aka Fastererer Ensemble Learning)

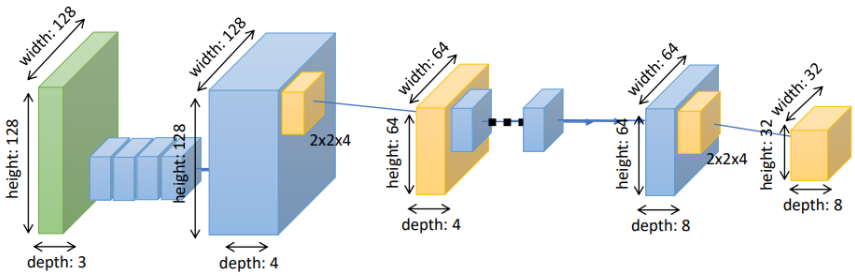
Making huge ensembles is pretty expensive, so can we make multiple models out of a single network? Dropout does it by randomly setting some activation to zero in the forward pass. Each activation has a p chance of not becoming 0 during the forward pass of training. At test time, we multiply our weights by $\frac{1}{p}$ since by forcing activations to zero, we are forcing the weights to be that much bigger. A smarter method of doing this is to actually divide our activations by p in the training step.

5 Computer Vision

5.1 Convolutional Neural Networks (CNNs)

5.1.1 Brief Introduction to CNNs

Convolutional neural networks have been very effective when dealing with image input. A standard CNN is made up of convolutional layers, pooling layers, and fully-connected layers. The motivation for CNNs comes from the fact that if we just use a fully-connected layer for an image, it would take millions of parameters and the relative positions of pixels in the image (which is important) would be lost. Instead, convolutional layers are used which use less parameters and retain the local information of pixels in an image. Pooling layers simply downsample an image at each layer. An example diagram of CNN is attached below, which a convolutional layer and pooling layer alternating.

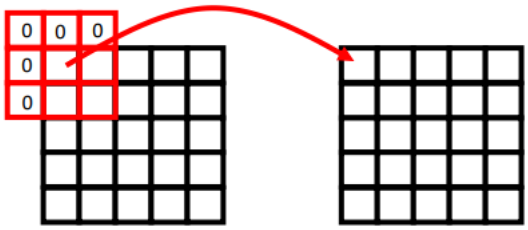


5.1.2 Convolutional Layer

When working with data for images, we usually deal with N-dimensional arrays, or tensors. The dimensions of the input layer is HEIGHT x WIDTH x CHANNELS (usually 3 channels for rgb). Next we will have filters: FILTER.HEIGHT x FILTER.WIDTH x OUTPUT CHANNEL x INPUT CHANNEL. This can be thought of as OUTPUT CHANNEL number of filters that are just FILTER.HEIGHT x FILTER.WIDTH block of weights. Each filters will convolve around the image to produce a OUTPUT.HEIGHT x OUTPUT.WIDTH "image". This image is then applied with a per-element activation function. There are OUTPUT CHANNEL of these filters, so we can concatenate these outputted images together to get a HEIGHT x WIDTH x OUTPUT CHANNEL tensor. For CNNs, the best way to learn is to watch an animation of it in action. One good animation is off of Stanford's CS231n class. The link to their CNN animation is here.

5.1.2.1 Padding With our current design, activations will shrink every layer because we are forced to cut off the edges. A workaround to this is to zero pad. Here, we pad the outside of our image with zeros, this way we don't have to cut of the edges when convolving through it, leading to the same dimension output as input. A small issue with zero padding is the these zeros will

be smaller than any pixel values in the image. To solve this we can average out the pixel intensities of the image and subtract it from each pixel. This way zero is the average of the pixel intensities of the image.



5.1.2.2 Pooling Pooling simply downsamples our image. A popular choice is max pooling, which takes every nxn piece in an image, takes the max value of these piece, and outputs just the max. So a 2x2 max pooling layer will make a 10x10x3 image into a 5x5x3 image.

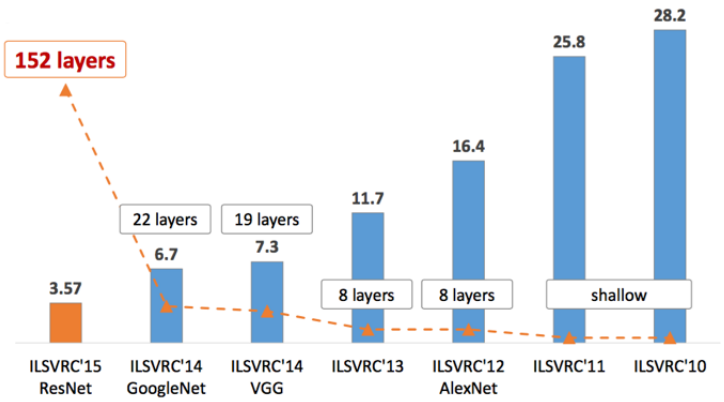
5.1.2.3 Putting it Together A standard conv net structure is to at each layer:

1. Apply conv, $H \times W \times C_{in} \to H \times W \times C_{out}$
2. Apply activation σ, $H \times W \times C_{out} \to H \times W \times C_{out}$
3. Apply pooling (width N), $H \times W \times C_{out} \to H/N \times W/n \times C_{out}$

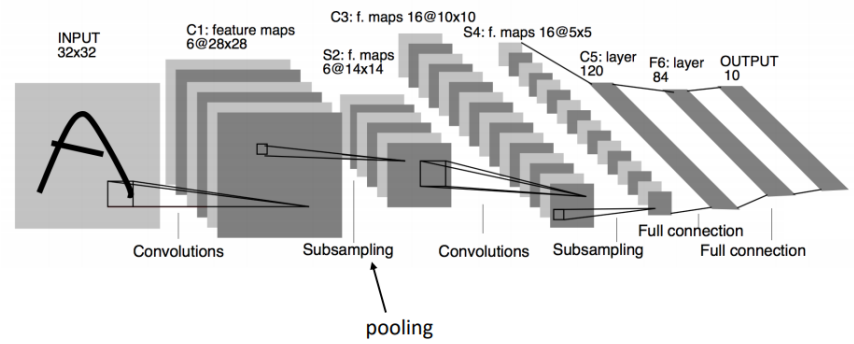
5.1.2.4 Strided Convolutions Applying a convolution can be computationally expensive. One idea is to skip over a few points. Strided convolutions skip over a few pixels every time a convolution happens. The amount of skipping is called a stride. Normally, our stride would be 1 because we don't skip over pixels.

5.1.3 Examples of CNNs

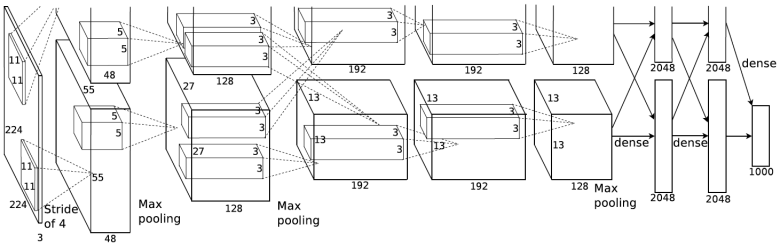
CNNs are progressed significantly over the last decade. We will cover the progress of CNNs over this decade as it has gotten better and better at classifying images on a image classificaiton bnechmark called ILSVRC (ImageNet), which contains 1.5 million images on 1000 categories. The error rate of these networks on ImageNet are attached below.



5.1.3.1 LeNet The LeNet network was one of the earliest CNNs and promoted the development of deep learning. It did handwritten digit classification on MNIST. It does convolutions, pooling, convolutions, pooling, and then goes through some fully-connected layers.



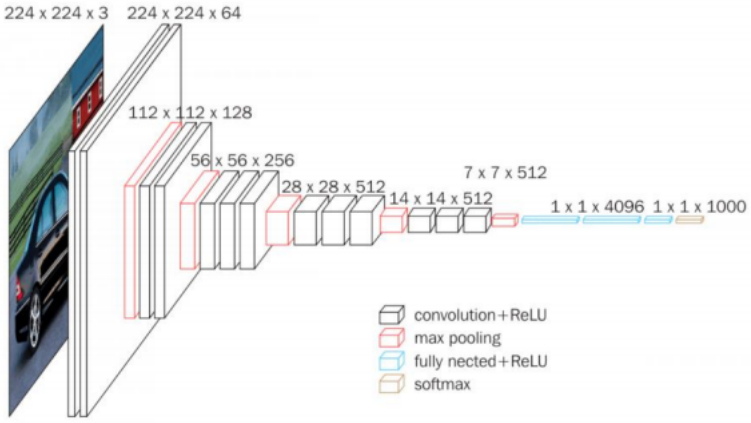
5.1.3.2 AlexNet AlexNet is the first real deep learning method to beat non-deep learning methods on ImageNet. The network looks a little strange because there are two different parts. This is because the model was trained on two GPUs so each section is trained on one GPU. Today, we don't really worry about this sort of things because GPUs have enough memory to keep the network on one GPU.



CONV1: 11x11x96, Stride 4, maps 224x224x3 -> 55x55x96 [without zero padding]
POOL1: 3x3x96, Stride 2, maps 55x55x96 -> 27x27x96
NORM1: Local normalization layer
CONV2: 5x5x256, Stride 1, maps 27x27x96 -> 27x27x256 [**with** zero padding]
POOL2: 3x3x256, Stride 2, maps 27x27x256 -> 13x13x256
NORM2: Local normalization layer
CONV3: 3x3x384, Stride 1, maps 13x13x256 -> 13x13x384 [**with** zero padding]
CONV4: 3x3x384, Stride 1, maps 13x13x384 -> 13x13x384 [**with** zero padding]
CONV5: 3x3x256, Stride 1, maps 13x13x256 -> 13x13x256 [**with** zero padding]
POOL3: 3x3x256, Stride 2, maps 13x13x256 -> 6x6x256
FC6: 6x6x256 -> 9,216 -> 4,096 [matrix is 4,096 x 9,216]
FC7: 4,096 -> 4,096
FC8: 4,096 -> 1,000
SOFTMAX

This network uses ReLU nonlinearities after each CONV and uses regularization techniques such as data augmentation and Dropout. It uses local normalization, which is not really used anymore (we use batch normalization now).

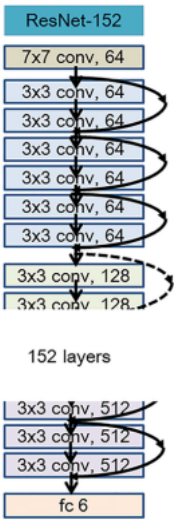
5.1.3.3 VGG VGG was able to get better accuracy by engineering much deeper networks. It follows a pattern of applying two convs and pooling, and then repeating.



CONV: 3x3x64, maps 224x224x3 -> 224x224x64
CONV: 3x3x64, maps 224x224x64 -> 224x224x64
POOL: 2x2, maps 224x224x64 -> 112x112x64
CONV: 3x3x128, maps 112x112x64 -> 112x112x128
CONV: 3x3x128, maps 112x112x128 -> 112x112x128
POOL: 2x2, maps 112x112x128 -> 56x56x128
CONV: 3x3x256, maps 56x56x128 -> 56x56x256
CONV: 3x3x256, maps 56x56x256 -> 56x56x256
CONV: 3x3x256, maps 56x56x256 -> 56x56x256
POOL: 2x2, maps 56x56x256 -> 28x28x256
CONV: 3x3x512, maps 28x28x256 -> 28x28x512
CONV: 3x3x512, maps 28x28x512 -> 28x28x512
CONV: 3x3x512, maps 28x28x512 -> 28x28x512
POOL: 2x2, maps 28x28x512 -> 14x14x512
CONV: 3x3x512, maps 14x14x512 -> 14x14x512
CONV: 3x3x512, maps 14x14x512 -> 14x14x512
CONV: 3x3x512, maps 14x14x512 -> 14x14x512
POOL: 2x2, maps 14x14x512 -> 7x7x512
FC: 7x7x512 -> 25,088 -> 4,096 ← almost all
FC: 4,096 -> 4,096
FC: 4,096 -> 1,000
SOFTMAX

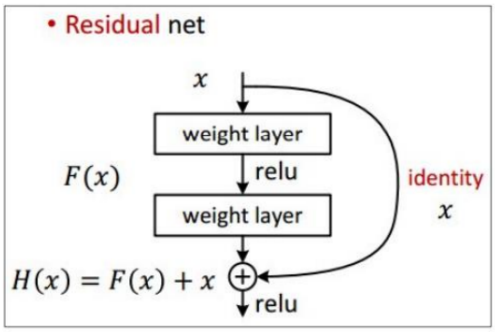
Most of the memory is used in the earlier layers due to the large image size. Most of the parameters are present in the first FC layer. This is not actually a good thing, and has been improved on in future networks.

5.1.3.4 ResNet ResNet is a very popular architecture today that is very deep. It is 152 layers. It has an error rate of 3.57% on ImageNet, which is better an a human.



The ResNet has repeated blocks of conv layers that gets pooled to a smaller

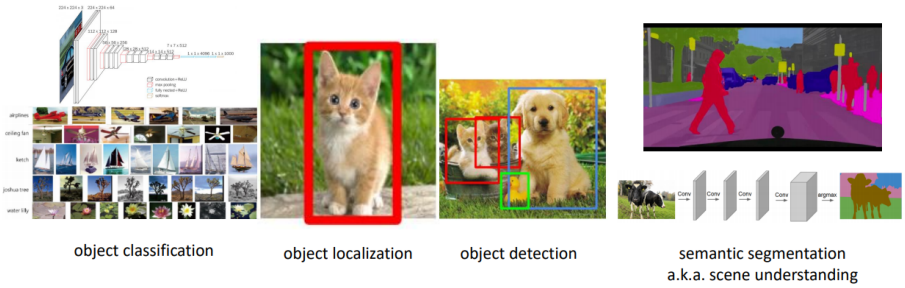
size, much like VGG. At the end, we average pool (like max pool but averaging instead) the convolutional response map and pass it directly through an FC layer into a softmax. For example, if our response map was $64 \times 64 \times 512$, then after average pooling it would just be a vector of length 512. We cut out the entire fully-connected block. It turns out the most of the work is done the convolutional layers so fully-connected blocks are not really needed. The reason why ResNet is able to be so deep is because it uses skip connections.



The residual layer takes every group of two convolutions, takes the input of the first input, and adds it to the output of the second layer. The reason why this helps networks train deeper networks is because during backpropagation, we want our gradients to be close to $\mathbf{I}$ to prevent exploding gradients or vanishing gradients. With a residual layer, $\frac{\mathrm{d}H}{\mathrm{d}x} = \frac{\mathrm{d}F}{\mathrm{d}x} + \mathbf{I}$, so if weights are not too big, the gradients will be close to $\mathbf{I}$.

5.2 Standard Computer Vision Problems

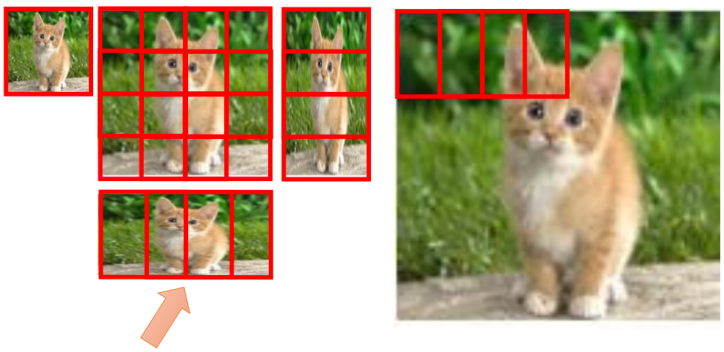
The four standard computer vision problems of increasing complexity is object classification, object localization, object detection, and semantic segmentation (a.k.a scene understanding). Object classification can be solved directly with the CNN architectures described above, so this section will go over tackling the other three problems.



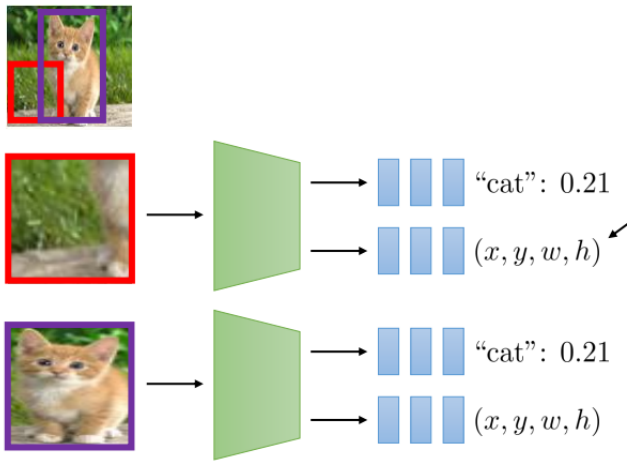
5.2.1 Object Localization

5.2.1.1 Intersection over Union (IoU) For object localization, our dataset predictions are now $(l_i, x_i, y_i, w_i, h_i)$ where (x_i, y_i) is the left corner of the box around the image, (w_i, h_i) are the height and width of the bounding box, and l_i is the classification of the image. An evaluation metric for accuracy of our bounding box is called intersection over union (IoU), which is just the intersection area of the predicted bounding box and true bounding box divided by the union area of the predicted bounding and true bounding box. One way to validate if a prediction is correct is to check if the class is correct and the IoU is greater than 0.5.

5.2.1.2 Sliding Windows Another idea for object localization is to classify every patch of an image to find which one has the highest probability. We can use a sliding window to determine the patches of an image. In addition to a sliding window, we can also vary the aspect ratio of the image as well as how big the sliding window is relative to the image. In the example below, we choose the entire image, then we do a sliding window over 1/4 of the image, then we change the aspect ratio and do sliding window, and then we do sliding window over 1/9 of the image, and so on.



5.2.1.3 OverFeat OverFeat uses both sliding windows and bounding box prediction to do object localization. It passes over different regions at different scales and roughly averages together boxes from different windows weighted by their probability.

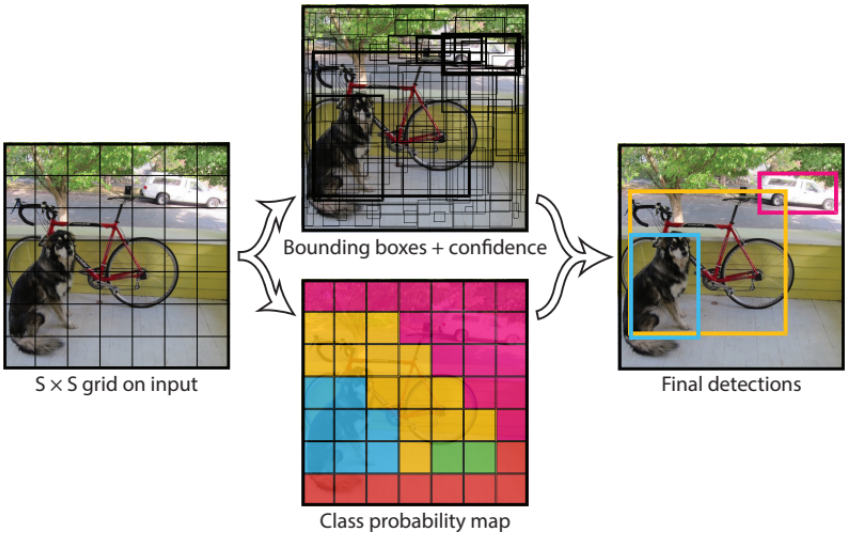


Doing sliding windows on the image can be very costly computationally. One idea to solve this is to use convolutional classification, where we slide over the convolutional layer rather than the image itself.

5.2.2 Object Detection

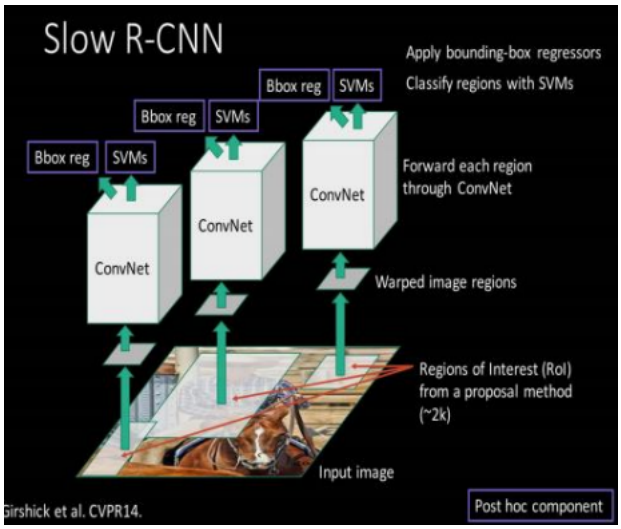
For object detection, naively we could just use the sliding window technique and just select the windows with highest probabilities. The issue with this is that high-scoring windows probably have other high-scoring windows nearby, so we can use non-maximal suppression, where we kill of any detections that have other higher-scoring detection of the same class nearby. We also need to output multiple things so we could pick a number for number of outputs beforehand. There are issues with each of these ideas individually, but after combining them we can get something that works.

5.2.2.1 You Only Look Once (YOLO) YOLO takes an input image and breaks it into a 7×7 grid. For each cell in the grid, the model outputs $B = 2$ sets of a bounding box location, a confidence score (an estimate over the IoU), and the class label. Note that the bounding boxes can go out of the cell. Lastly, we only output bounding boxes above a certain confidence score. During training, we need to assign responsibility of a cell to each true object. YOLO does this by using the cell with the highest IoU.



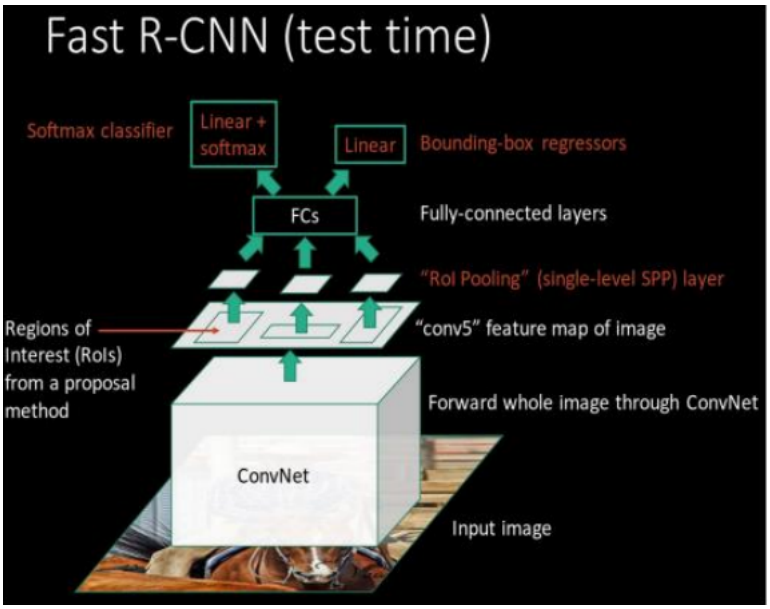
5.2.2.2 Region Proposals (R-CNN, Fast R-CNN, and Faster R-CNN)

R-CNN uses existing computer vision method to extract regions in image and feed these images into a CNN for classification. So essentially it is just a smarter sliding window.



However this is really slow. The solution to this is Fast R-CNN, where instead of getting Regions of Interest (RoIs) from the image itself, we geting the RoIs from a convolutional response map. We then take our region of interest, take all

the features inside that region, and pool them together (i.e. max pool). Then we just pass it into a FC layer to produce a class and bounding box.

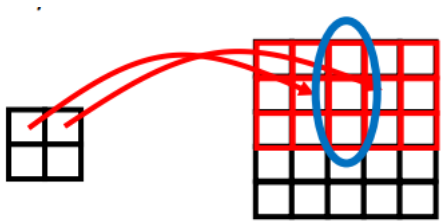


Currently R-CNN and Fast R-CNN uses selective search to find region proposals. This is slow, so instead we can train RoI proposals by taking the same convolutional response map, and at every position, predict if there is an object at that location and what its spatial extents are. In other words, this is essentially Overfeat but without predicting class. We are only looking for if an object might be present there. This method is called Faster R-CNN.

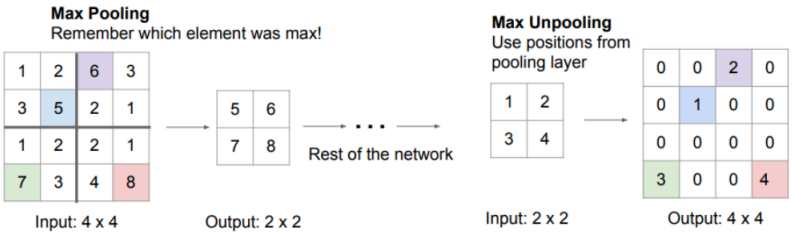
5.2.3 Semantic Segmentation

Semantic segmentation can be thought of as a per-pixel classifier. We want our output to have the same resolution as the input. We could solve this by using a CNN and never downsampling, but this is very expensive. So instead, we reduce the resolution like a regular CNN by downsampling and then increase the resolution again to output a label for every pixel by upsampling. To upsample, we will use a transpose convolution.

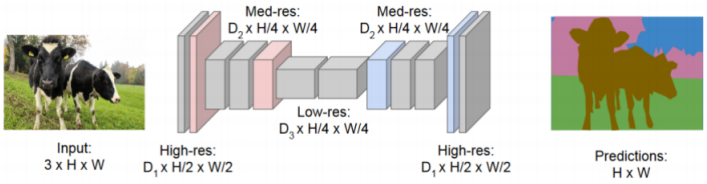
5.2.3.1 Transpose Convolution The transpose convolution is a convolution with a fractional stride. The overlapping region after multiplying by the filters can just be averaged.



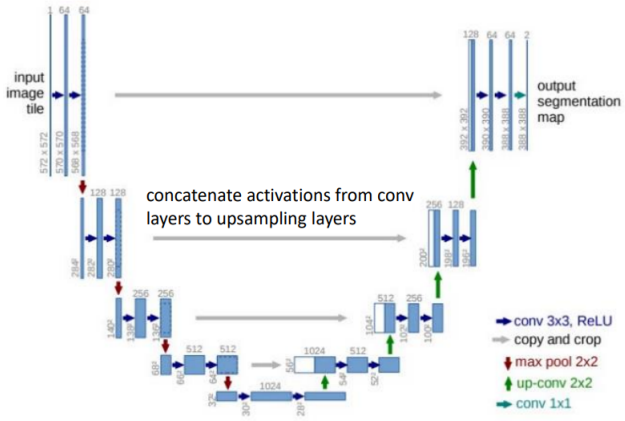
5.2.3.2 Un-pooling One trick for un-pooling is to, on the forward operation, save out which index had the max. Then later on in the network when we have the corresponding upsampling, we take that index and save the value in the low-resolution map to the corresponding index in the high-resolution map. This requires our network to be symmetric.



5.2.3.3 Bottleneck Architecture A simple kind of architecture for doing this is to use a conventional conv net such as a VGG or ResNet as the first part and then flip it around to get it back to the original resolution using transpose convolutions and un-pooling.



5.2.3.4 U-Net The issue with bottleneck architecture is that when you shove everything into the bottleneck, then some of the spatial information is actually lost. The solution to this is to take the low-resolution maps, upsample it, and then connect it with the high-resolution maps from before via skip connections. This is the idea before U-Net.

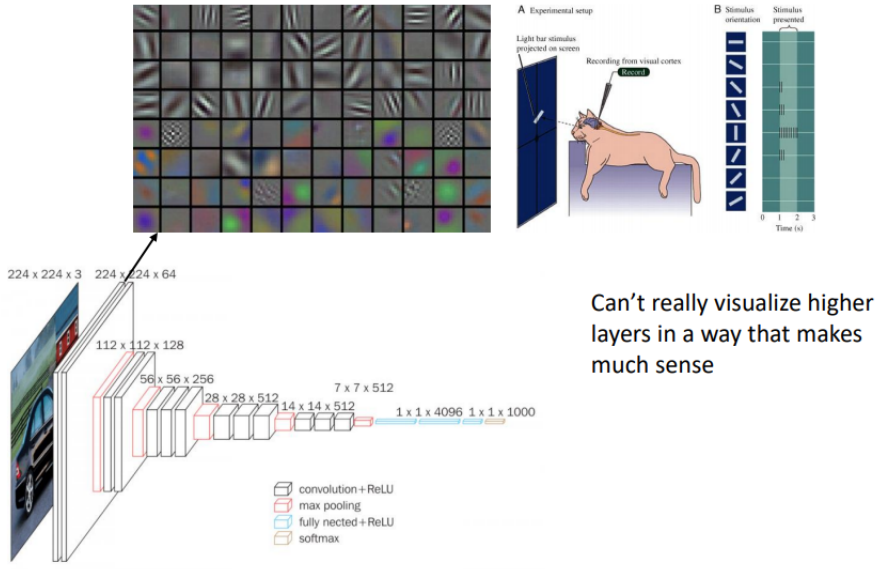


There are two convolutional layers, then a downsampling, and then repeated. Then when we start upsampling, we upsample the previous layer, take the layer from the original conv net that had the same resolution, and then we concatenate its activations.

5.3 Visualizing CNNs

5.3.1 Visualizing Filters

We can print out the numbers in the filters of the first layer to visually inspect the filters to see what they are looking for.



It is apparently here that some filters look for different types of edges, color patterns, etc. The edge detection is much like that of a mammalian brain because edges are the dominant features in natural images.

5.3.2 Visualizing Neuron Responses

We can also look for images that maximally excite specific units. So we run a filter at a specific layer across many images to see which portion of the images gives the highest values.

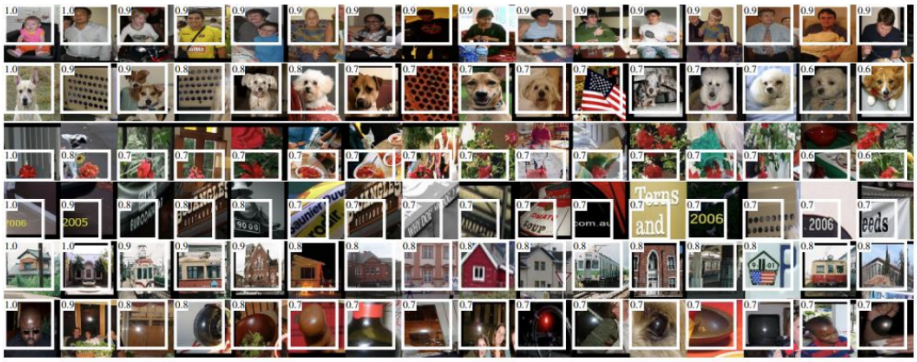


Figure 4: Top regions for six pool$_5$ units. Receptive fields and activation values are drawn in white. Some units are aligned to concepts, such as people (row 1) or text (4). Other units capture texture and material properties, such as dot arrays (2) and specular reflections (6).

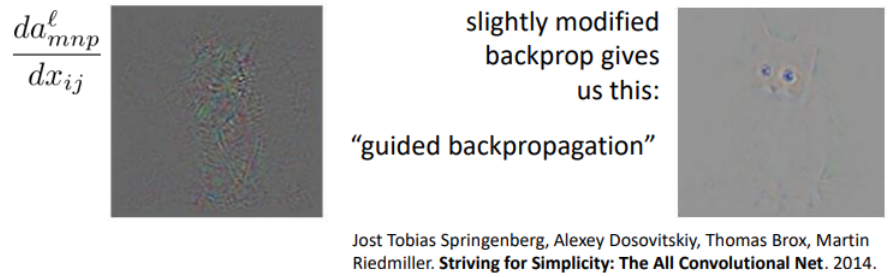
In the image above, in the bottom row, the filter seems to look for rounded shiny objects.

5.3.3 Using Gradients For Visualization

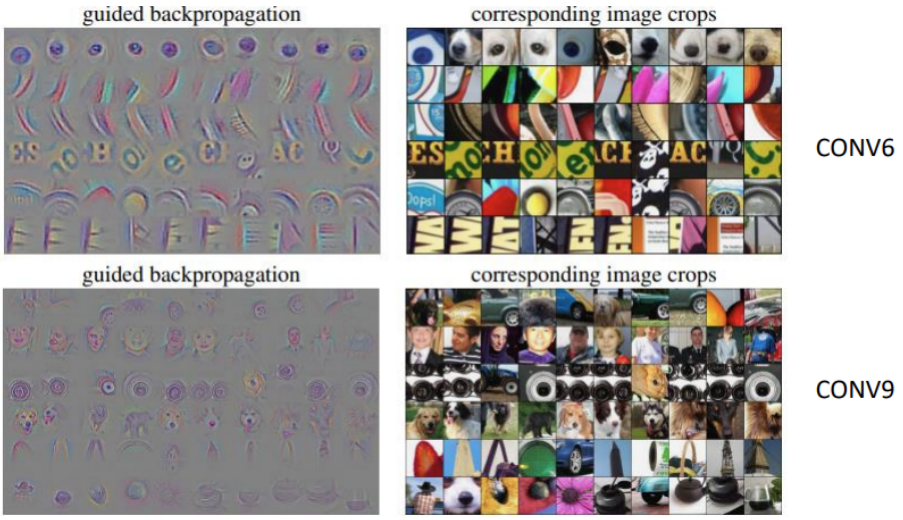
For any unit in the filter a^l_{mnp}, one way to see how much influence an image pixel has over that unit is that find $\dfrac{\mathrm{d}a^l_{mnp}}{\mathrm{d}x_{ij}}$. We can calculate it using backpropagation this way:

1. Set δ to $\dfrac{\mathrm{d}a^l_{mnp}}{\mathrm{d}a^l}$ (i.e. same size as a^l with $\delta_{mnp} = 1$, all other entries to 0).
2. Backprop from layer l to the image.
3. Last δ gives us $\dfrac{\mathrm{d}a^l_{mnp}}{\mathrm{d}x}$.

Using this process, we can calculate something like the image on the left at the bottom. We can kind of make out a cat but it's hard. With a slightly modified backprop, we can see that the filter unit looks for big blue eyes.



The trick (or hack) is called guided backpropagation. The idea is that backprop might not be very interpretable because many units in the network contribute positive or negative gradients. Maybe if we just keep the positive gradients, we'll avoid the compilated negative contributions, and get a cleaner signal. So for the backward step in ReLU, we also zero out negative gradients at each layer.



Using guided backprop, we can see that some filters have a preference for faces, noses, etc.

5.3.4 Optimizing the Input Image

What if we optimized the image to maximially activate a particular unit. In other words, we have $x \leftarrow x + \alpha \frac{\mathrm{d}a^{l}_{mnp}}{\mathrm{d}x}$. This is solving the optimization problem $x \leftarrow \mathrm{argmax}_x \, a^{l}_{mnp}(x)$. More generally, if we want to maximize the image based on filters in all positions of the image or if we want to maximize the probability of a particular class, we can solving the optimization problem $x \leftarrow \mathrm{argmax}_x \, S(x)$.

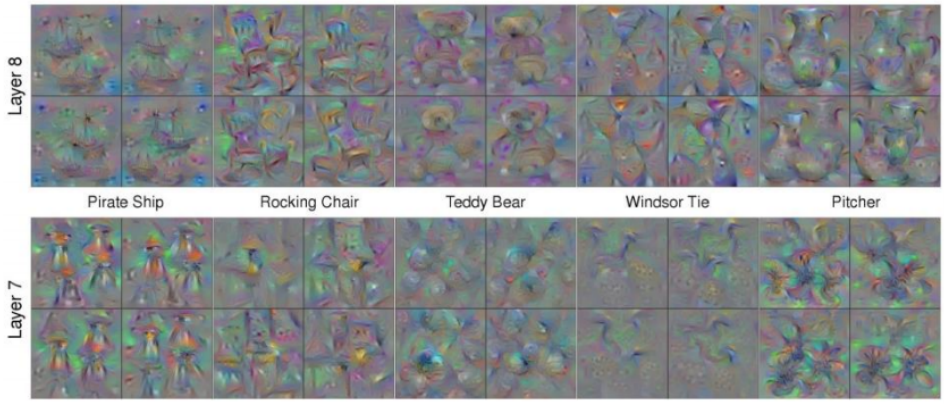
By itself, it is too easy to create crazy images without a regularizer. In the blue-eyed cat example, the blue channel will just go crazy to maximize the activations. So instead, our optimization problem because $x \leftarrow \text{argmax}_x S(x) + R(x)$. A simple choice is $R(x) = \lambda \|x\|_2^2$.

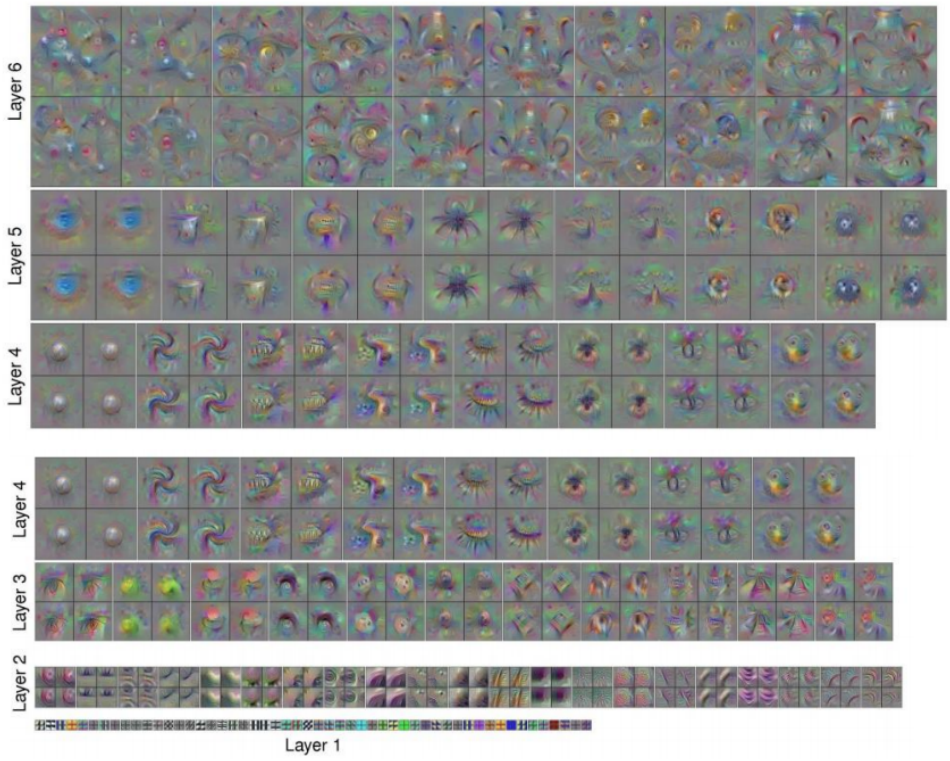
5.3.5 Visualizing Classes

With this in mind, let's try to visualize class labels. First we need to make sure to maximize activations before the softmax since the softmax normalizes the the different class probabilities together. As a result, there is a possibility backprop won't maximize the class label activations but rather minimize other class label activations. A more nuanced procedure that can lead to nicer results is to:

1. Update image with gradient.
2. Blur the image a little (so that optimizer doesn't produce crazy high-frequency details)
3. Zero out any pixel with small value (prevents minute noise from confusing the network)
4. Repeat

and then use a little more nuanced regularizer $R(x)$ in the optimization problem.



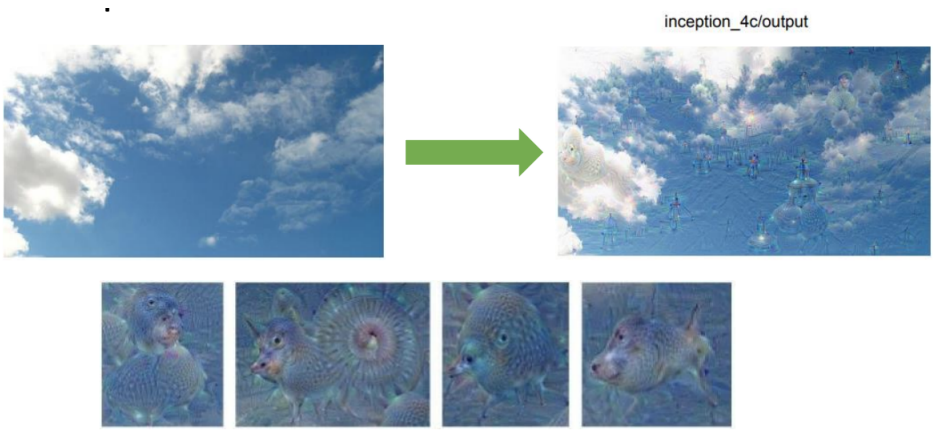


We can see that later layers visualize more complete outlines while earlier layers visualize more abstract components.

5.4 Deep Dream

Deep Dream essentially looks at which units are activated in a layer and activates it more. In pseudocode, it does this:

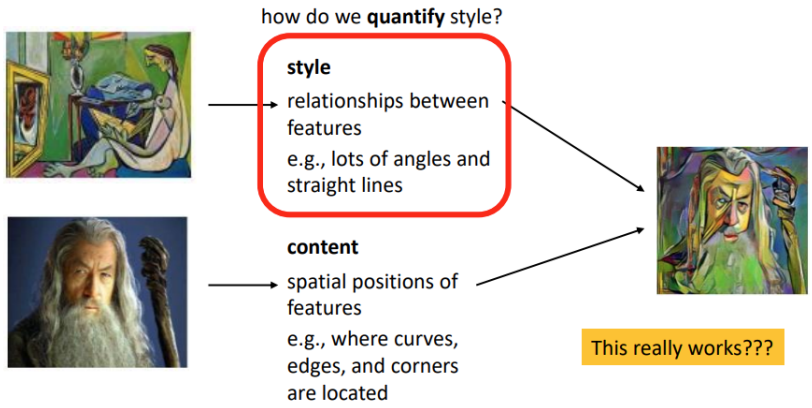
1. Pick a layer in conv net
2. Apply a little bit of jitter to the image
3. Run forward pass to compute activations of that layer
4. Set delta to be equal to the activations
5. Backprop and apply the gradient
6. Un-jitter the image
7. Repeat 2-6



Using Deep Dream, we start to see object pop up in the clouds. It essentially extenuates classes that it sees a little of bit already. We apply jitter as a regularizer to the image to prevent the optimizer to do crazy things to a single pixel. Some (possibly cursed) images generated by Deep Dream are shown below:

5.5 Style Transfer

So Deep Dream exaggerates the features in a single image. But, what if we could make features of one image look more like features in another. What if we could take the relationships between features of one image (i.e. the style) and the spatial positions of the features (i.e. the content) of another image and combine these to get a modified version of the content with the style from the other image.



How do we quantify style and content?

5.5.1 Style

For style, we want to discard spatial information. We instead only care about which units activate and how different units activate together. In other words, we ask, between filters, which features tend to co-occur? We can quantify this by building a feature covariance: $\text{Cov}_{km} = \mathbb{E}[f_k f_m]$. The expectation is taken over different positions in the image. So we will, over all positions of the image, average together the product of two features. If features occur together, their products will be very large. Otherwise, they will be small. We can build the Gram matrix where $G_{km} = \text{Cov}_{km}$. Let G^l be the source image Gram matrix at layer l and $A^l(x)$ be the new image Gram matrix at layer l. Our style loss will be

$$\mathcal{L}_{style}(x) = \sum_l \sum_{km} (G^l_{km} - A^l_{km}(x))^2 w_l$$

This weight w_l is a little delicate since we need to prioritize relative contribute to desired style of different levels of abstraction.

5.5.2 Content

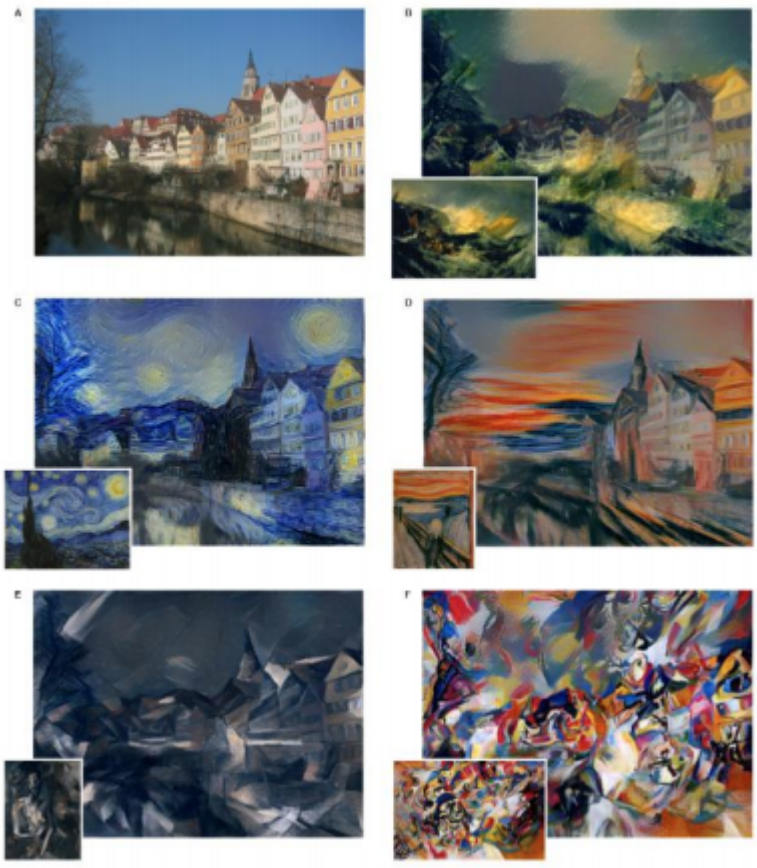
For content, we want to prevent the spatial positions instead of looking at relationships between features. So we can just directly match the features to the content image:

$$\mathcal{L}_{content}(x) = \sum_{ij} \sum_k (f^l_{ijk}(x_{content}) - f^l_{ijk}(x))^2$$

Notice that we choose a specific layer that we consider to be reflective of content, which is a delicate choice.

5.5.3 Putting it Together

Then our new image forced by running backpropagation with $x \leftarrow \text{argmin}_x \mathcal{L}_{style}(x) + \mathcal{L}_{content}(x)$



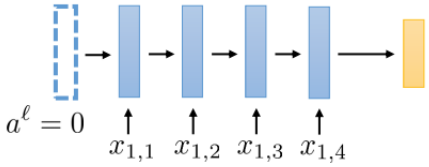
Using style transfer, we can above to transfer different styles to a base image. Very cool!

6 Natural Language Processing (NLP)

6.1 Recurrent Neural Networks (RNNs)

6.1.1 RNN Overview

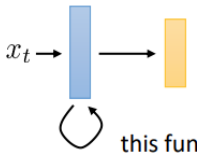
A standard neural network can't handle variable-length inputs (e.g. words in a sentence), motivating RNNs. RNNs are neural networks that share weights across multiple timesteps, take an input at each timestep, and have variable number of timesteps. In the graphic below, a^ℓ is the input "hidden" state and $x_{1,t}$ are the inputs into the network.



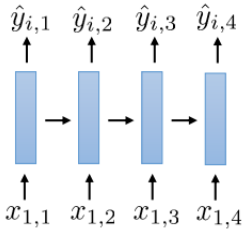
At each timestep of the RNN (i.e. each blue rectangle), we perform the following operation

$$a^\ell = \sigma(W^\ell \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix} + b^\ell)$$

Here, i stands for the ith input sequence. We are essentially just concatenating the hidden state from the previous timestep with the input at the current timestep, and then passing it through a feedforward layer. In concisely, RNNs can be shown this way as well:



The RNN shown above only takes in inputs at each step. But it could also have outputs as each step too. In the image below, $\hat{y}_\ell$ is the output of the tth time step.



At each timestep of the RNN (i.e. each blue rectangle), we now perform the following operation

$$a^\ell = \sigma(W^\ell \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix} + b^\ell)$$

$$\hat{y}_\ell = f(a^\ell)$$

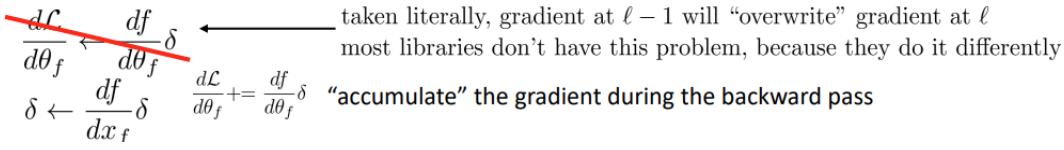
Here, $f(\cdot)$ is called a decoder. It could be something simple like a linear layer + softmax. For backpropogation, our new loss in this case would be

$$\mathcal{L}(\hat{y}_{1:T}) = \sum_l \mathcal{L}_l(\hat{y}_l)$$

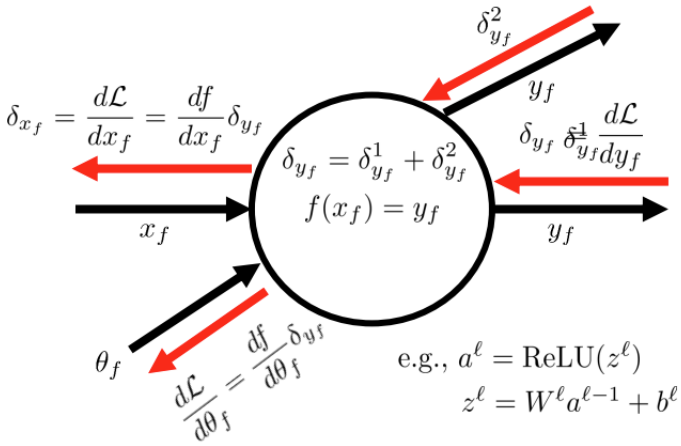
6.1.2 Backpropagation for RNNs

Using naive backpropagation for feedforward neural networks won't work for RNNs due to two issues:
1) RNN weights appear multiple times in backpropagation (once per timestep).
2) An RNN timestep has two heads (i.e. hidden state to pass into next timestep and hidden state to pass to decoder).

To solve these two problems, we need to
1) Accumulate the the gradients of the parameters at each timestep as we do backpropagation instead of setting them.



2) Perform reverse-mode automation differentiation, which is a fancy word for adding up delta vectors coming from all the descendants of a neuron to calculate gradients.

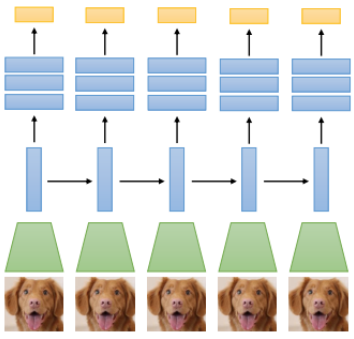


6.1.3 Issues with RNNs

Since RNNs can get very deep when an input sequence is long, our gradients for weights in the earlier timesteps tend to be either 0 (if many jacobians along the chain rule are less than 1) or infinity (if many jacobians along the chain rule are greater than 1). The exploding gradient problem, where gradients tend towards infinity, can be solved with hacks like gradient clipping. However, the vanishing gradient problem, where gradients tend toward 0, cannot be easily solved as the signal from later steps is essentially lost. For best gradient flow, we want our intermediate jacobians to be the identity matrix. This vanishing gradient problem motivates LSTMs.

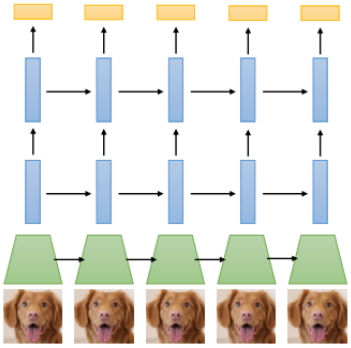
6.1.4 Small Extensions to RNNs

6.1.4.1 RNN Encoders and Decoders We've already seen an RNN "decoder," which is a function that takes in a hidden state of the RNN and outputs something we want. Similarly, there are RNN "encoders," which take in an input and embed it for the network.



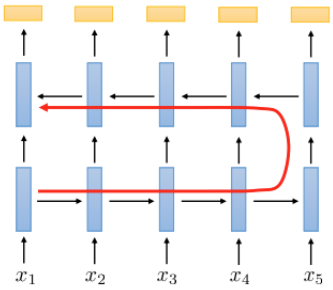
In this example, the images are "encoded" by a convolutional neural network (i.e. the "encoder") for the RNN input. The outputted hidden states, as we've seen before, as "decoded" by a fully connected neural network (i.e. the "decoder") to get the output we want.

6.1.4.2 Multi-layer RNNs We can also stack multiple layers together for an RNN like so:



For the first layer of RNNs, instead of passing the hidden layer to a decoder, we pass it as input to the second layer of RNNs. Then the second layer functions essentially the same as our single-layer RNN.

6.1.4.3 Bidirectional RNNs Bidirectional RNNs is just a multi-layer RNN with the direction of the second layer reversed like so:

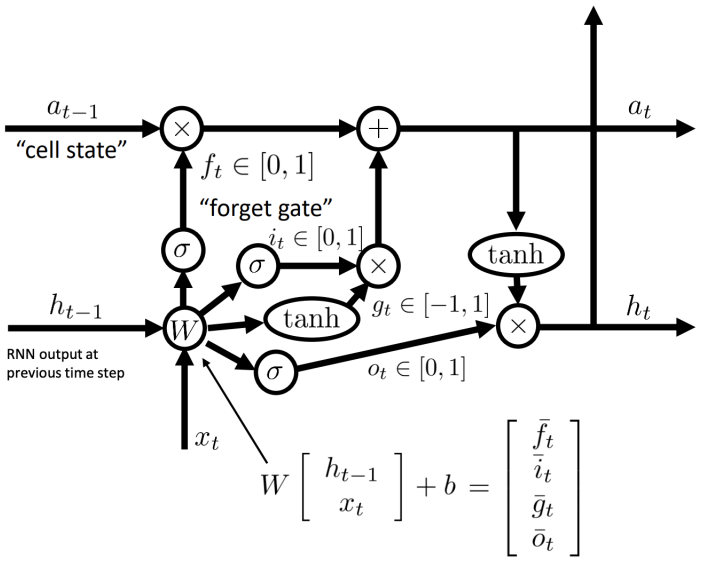


The motivation behind this is that for some tasks, we might need to look at inputs in the future to determine a prediction in the past. For instance, in speech recognition, we may want to know the entire sentence to give context for the meaning a word in the middle of the sentence.

6.2 Long Short-Term Memory (LSTM)

6.2.1 LSTM Overview

To solve the vanishing gradient problem, we want the intermediate jacobians in backpropagation to be I when we want to "remember" the signal (otherwise, we want it to be 0 when we want to "forget" the signal). This motivates the design of an LSTM cell, which is shown in the image below:



An LSTM cell takes in a cell state (a_{t-1}), a hidden state (h_{t-1}), and an input (x_t). Like with RNNs, h_{t-1} is concatenated with x_t and multiplied by weights. Unlike RNNs, the weight matrix, W, is four times as long as h_{t-1}. As a result, $W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b$ is four times as long as h_{t-1}. This output is split into four equal-length subvectors: $\bar{f}_t, \bar{i}_t, \bar{g}_t, \bar{o}_t$. Through the diagram, we see that the following operations occur on feedforward:

$$f_t = \sigma(\bar{f}_t), i_t = \sigma(\bar{i}_t), g_t = tanh(\bar{g}_t), o_t = \sigma(\bar{o}_t)$$

$$a_t = f_t \cdot a_{t-1} + i_t \cdot g_t \cdot \mathbb{1}$$

$$h_t = a_t \cdot o_t$$

$$\hat{y}_t = f(h_t)$$

Here, a_t and h_t are the cell states and hidden states to be passed in to the next timestep. $f(\cdot)$ is some decoder function, and $\hat{y}_t$ is the output of the model at timestep t, which goes into the loss function.

6.2.2 Intuition behind LSTM

We call f_t the forget gate because when it is 0, a_{t-1} is completed overwritten by the current input and when it is 1, a_{t-1} is copied for the current timestep. In a sense, the value of f_t chooses how much we want to "forget" the previous information. i_t, the input gate, determines if we want to have a modification to the cell state (i_t is 1 if we can the cell state to be modified, 0 if not). g_t determines what that modification will be. This allows the model to separately choose whether to modify or not and how to modify. o_t, the output gate, controls the next hidden state.

Notice that all the operations applied to a_t are linear and that it doesn't really change between timesteps. This leads to simple and well-behaved gradients. Yet, it still has the benefits of nonlinearities through the nonlinear modification of g_t and nonlinear readout (i.e. decoding) through $f(\cdot)$. While f_t isn't actually I as we wanted, this ends up working well in practice. The cell state is called long-term memory because it doesn't really change over time and retains information over long periods of time, while the more messy hidden state is called short-term memory because it is always changing (i.e. at each step, the hidden state is basically overwritten by W and x_t).

6.2.3 Gated Recurrent Units (GRUs)

f_t and g_t are the most important gates. This is because g_t can incorporate everything i_t has to do. Similarly, the tanh input to h_t can, in theory, incorporate everything o_t has to do. So maybe we can simply the LSTM network more. In fact, GRUs do just that. GRUs are LSTMs without an output gate. In practice, this tends to work just as well.

6.3 Autoregressive Models

6.3.1 Definition of Autoregressive Model

Autoregressive model is a just a big word for a model that predict future states based on past states. For example, in text generation, an autoregressive model can take in the first word of the sentence as a past state and predict the future state (i.e. the next word). This is sometimes referred to as structured prediction, because multiple outputs have strong dependencies between these outputs. For instance, beyond predicting a sentence correctly or not, we also require that sentences make grammatical sense.

6.3.2 Distributional Shift

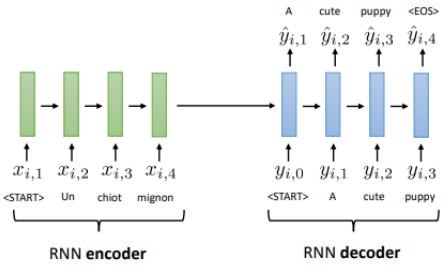
Autoregressive models run into an issue where the network only always sees true sequences as inputs during training, but a test-time it gets as input its own (potentially incorrect) predictions. This is called distributional shift, because the input distribution shifts from true string in training to synthetic strings at

test time. One solution to this problem is scheduled sampling: feed in mostly ground truth tokens as input at the beginning of training and feed in mostly the model's own predictions at the end of training. The schedule for probability of using ground truth input tokens is a hyperparameter that can be tuned.

6.4 Seq2Seq

6.4.1 Definition of Sequence-to-Sequence (Seq2Seq) Models

Sequence-to-sequence learning is about training models to convert sequences from one domain to sequences in another. An example is machine translation. This is done with an encoder-decoder framework. In the RNN example below, we have an RNN encoder to encode the French and an RNN decoder to decode it to English.



6.4.2 Beam Search

6.4.2.1 Motivation Say we are training and evaluating a seq2seq model with a softmax at each decoding timestep, which assigns probability to a list of words in a vocab list. During evaluation, each decoding timestep outputs a word, which is used as input to the next decoding timestep (i.e. text generation). During evaluation of a seq2seq model, at first glance, we might think that we would want to greedily choose the word with the highest probability at each timestep in the RNN decoder and feed that to the next decoding timestep. However, greedily choosing highest probability words may inadvertently lead us to paths where we are forced to choose incorrect words down the line. Thinking about this in terms of probability, given inputs $x_{1:T}$, we want to maximize $p(y_{1:T_y}|x_{1:T})$ where $y_{1:T_y}$ are our predictions (i.e. the output of the RNN decoder). However, greedily choosing the highest probability at each step doesn't necessarily maximize this joint probability. Notice that the joint probability can be rewritten as

$$p(y_{1:T_y}|x_{1:T}) = \prod_{t=1}^{T_y} p(y_t|x_{1:T}, y_{0:t-1})$$

$$\log p(y_{1:T_y}|x_{1:T}) = \sum_{t=1}^{T_y} \log p(y_t|x_{1:T}, y_{0:t-1})$$

So we actually want to maximize the sum of the log probability of the decoded output of each decoding timestep. Doing this by brute force is computationally expensive since it takes exponential time: for M words and sentences of length T, a brute force solution would have to look through all possible M^T sequences. Maybe we can come up with a slightly less accurate, but faster solution?

6.4.2.2 Overview of Beam Search Based on our intuition of sequence decoding, let us assume that while choosing a highest-probability word on any step may not be optimal, choosing a very low-probability word is very unlikely to lead to a good result. In other words, while we can't be greedy, we can be somewhat greedy. Intuitively, beam search is just this: we store the k best sequences so far, and update each of them. Note that $k = 1$ is just greedy decoding. The pseudocode of the algorithm roughly looks like this:

Algorithm 8 Beam Search

1: **for** $t \in \{1, \ldots, T\}$ **do**
2: For each hypothesis $y_{1:t-1}$ that we are tracking (there are k of these), find the top k tokens $y_{t,1}, ..., y_{t,k}$ (i.e. using softmax log probabilities)
3: Sort the resulting k^2 length t sequences by their total log probability
4: Save any sequences that end in EOS (i.e. end of sentence token)
5: Keep the top k sequences
6: Advance each hypothesis to time $t + 1$ if we have not reached our stop condition
7: **end for**
8: **return** Saved sequence with highest score where

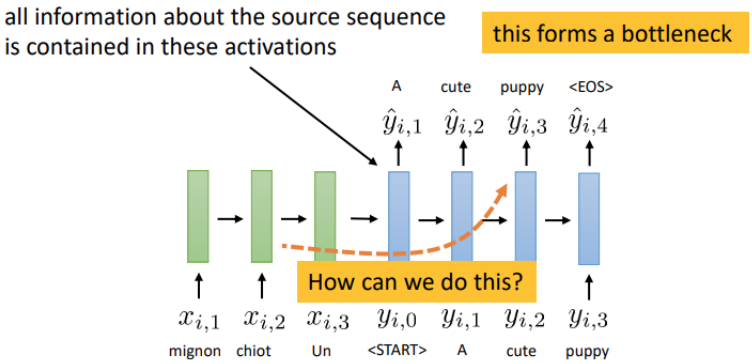
$$score(y_{1:t}|x_{1:T}) = \frac{1}{T}\sum_{t=1}^{T} \log p(y_t|x_{1:T}, y_{0:t-1})$$

Our stop condition can either be some cutoff length T or until we have N hypotheses that end in <EOS> (i.e. end of sentence token). To calculate highest score at the end, we divide the total log probability of the decoded sequence by its length as to normalize against its length. We do this because longer sequences will naturally have lower log probability since log p < 0 always.

6.5 Attention

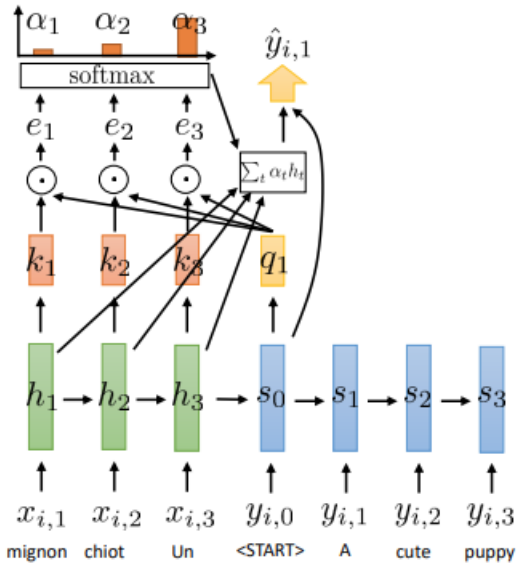
6.5.1 Motivation for Attention

In seq2seq with an RNN encoder and decoder, we have a bottleneck problem. For decoder timesteps that come later, the encoder signal they receive don't come from the encoder directly, but rather come from the activations of decoder timesteps before it. To solve this, we want to find someway for decoding timesteps to peek at the source sentence. This is where attention comes in.



6.5.2 Attention Overview

A standard attention model is shown below.



Crudely speaking, for each decoding timestep which takes in a word as input, we want the model to choose which word in the encoder is important to it and use that information. On a more intuitive level, to do this, for each hidden state in the decoder, s_l, we will put it through a query function $q(\cdot)$ to get the query vector $q_l = q(s_l)$. This will represent what we are looking for at this step. Similarly, for each hidden state in the encoder, h_t, we will get the key vector $k_t = k(h_t)$. This vector will represent what type of information is present at this step. To measure similarity between the key and query vectors, we will assign an attention score, $e_{t,l}$, to each (key, query) pair: $e_{t,l} = k_t \cdot q_l$. To find the encoding hidden state that matters the most, we then just select the h_t with

the highest attention score. In reality, taking an argmax is nondifferentiable so instead we take softmax of $e_{t,l}$ for all t starting from $t = 0$ (i.e. the first decoding timestep) and weight the encoding hidden states with these softmaxed scores instead. More rigorously, we will be doing these operations:

$$\text{Encoder side:}$$

$$k_t = k(h_t)$$

$$\text{Decoder side:}$$

$$q_l = q(s_l)$$

$$e_{t,l} = k_t \cdot q_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_{t,l} h_t$$

Now that we have a_l, which is the information that we what from the encoder for s_l, we have a few options for how we want to incorporate this information into our network. One way would be to concatenate a_{l-1} to the hidden state: $\begin{bmatrix} s_{l-1} \\ a_{l-1} \\ x_l \end{bmatrix}$. Another way could be using a_l for the readout (i.e. $\hat{y}_l = f(s_t, a_l)$). One other way would be to concatenate a_l as input into the next RNN layer if we are using a stacked RNN.

6.5.3 Attention Variants

There are a number of variants in attention. For choices of k and q, we could just make them identity functions. But maybe we want more expressivity. A common way to do this is just to use linear multiplicative attention, where $k_t = W_k h_t$ and $q_l = W_q s_l$. This is also convenient because $e_{t,l} = h_t^T W_k^T W_q s_l = h_t^T W_e s_l$, so we just have to learn one matrix instead of two. One other separate variant is to replace

$$a_l = \sum_t \alpha_{t,l} h_t$$

with

$$a_l = \sum_t \alpha_{t,l} v(h_t)$$

with some value function. This can give a little added flexibility.

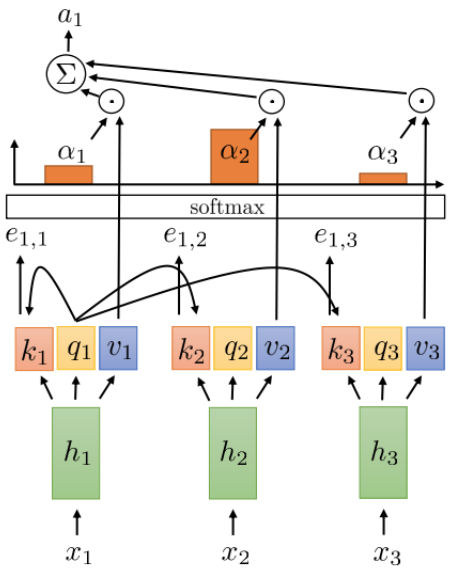
6.5.4 Why Attention Works

Attention is very powerful, because now all decoder steps are connected to all encoder steps. So with shorter paths between gradients (O(1) rather than O(n)), now these gradients are much better behaved.

6.6 Self-Attention

6.6.1 Overview of Self-Attention

Now that we know attention, the question becomes do we even need a recurrent structure? What if we just took attention and deleted connections between timesteps? It turns out that we can transform our RNN into a purely attention-based model, where attention can access every time step rather than just the decoder timesteps. This is called self-attention.



So what's so difference about this self-attention model compared to attention? There are two major differences. First, we have deleted the connections between timesteps as well as between the encoder and decoder. However, this raises the issue where the decoding timesteps don't have access to previous decoding timesteps. So our second change is to give key, value, and query vectors to each timestep and perform the same attention calculations. The self-attention equations thus are:

$$k_t = k(h_t)$$

$$v_t = v(h_t)$$

$$q_t = q(s_t)$$

$$e_{l,t} = q_l \cdot k_t$$

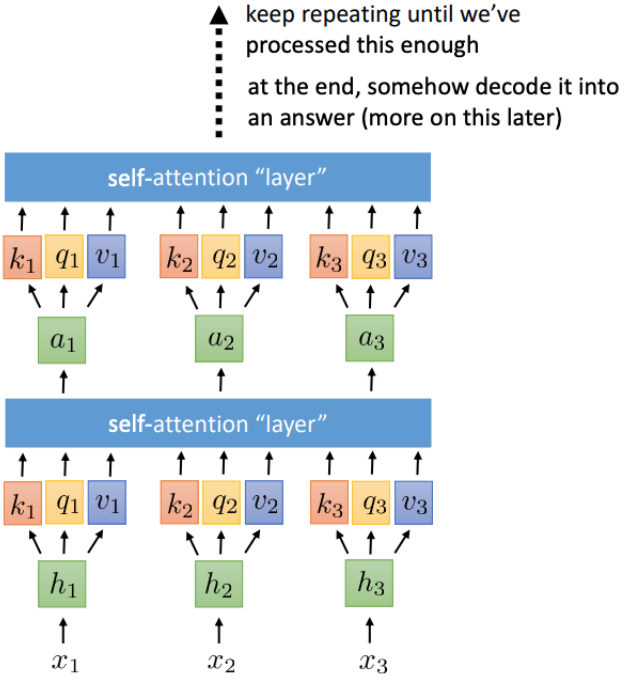
$$\alpha_{l,t} = \frac{\exp(e_{l,t})}{\sum_{t'} \exp(e_{l,t'})}$$

$$a_l = \sum_t \alpha_{t,l} v_t$$

Note that even though this is no longer a recurrent model, it is still weight sharing. For instance, if $h_t = \sigma(Wx_t + b)$. For each timestep, t, h_t all share the same weights.

6.6.2 Multi-Layer Self-Attention

The self-attention layer can be boiled down into the bottom 3 equations in the self-attention equations above. With this definition, we can actually stack self-attention networks on top of each other. In the image below, we have a feedforward network to get key, query, and values, a self-attention layer, and then a feedforward network again, and then a self-attention layer, and so on.



6.7 Transformers

6.7.1 Introduction to Transformers

Now that we know self-attention, we can develop a really powerful type of sequence model called a transformer. These types of models are currently very popular and the state-of-the-art for language problems. Examples include BERT and GPT-3. Before we can go into transformers, we first need to describe 4 changes we need to add to self-attention in order to understand the components of transformers. These 6 changes are:

1. Positional encoding: addresses lack sequence information, so we need to add this in somehow

2. Multi-headed attention: our current attention model doesn't have the ability to select multiple timesteps that it thinks are important

3. Adding nonlinearities: if our value function is linear, we lose a lot of expressive power without nonlinearities between self-attention layers.

4. Masked decoding: decoders can't look at key, value pairs in the future since it hasn't generated the words yet so we need to deal with this somehow as well.

5. Cross-attention: If we use masked attention for our decoder (from change #4), then we need some way to connect the decoder and encoder using attention. This is called cross-attention.

6. Layer normalization: Batch normalization is hard for sequence models due to small batch size and sequences being different lengths, so we will use layer normalization instead.

6.7.2 Components of Transformers

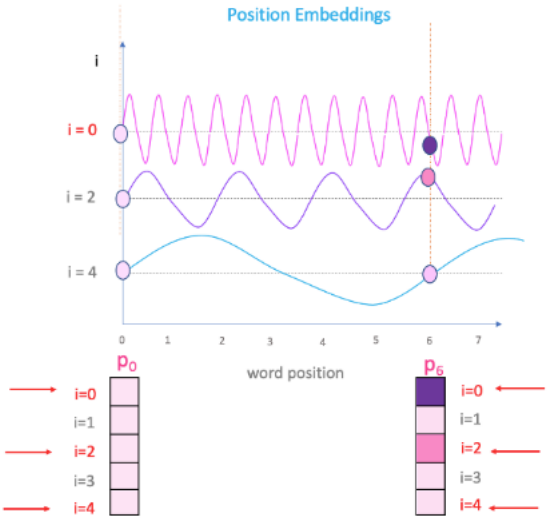
6.7.2.1 Positional Encoding First, we'll try to add in positional encodings. The issue we have right now is that for self-attention, our sequence might as well be a bag of words since we have lost the recurrent connections between timesteps. However, positions of words in sentences carries information. So we need to add some information for position. The most naive idea is to just append the timestep t to the input (i.e. $\bar{x}_t = \begin{bmatrix} x_t \\ t \end{bmatrix}$). However, in language, absolute position seems to be less important than relative position. This motivates frequency-based representations, where

$$p_t = \begin{bmatrix} \sin(t/10000^{2*1/d}) \\ \cos(t/10000^{2*1/d}) \\ \sin(t/10000^{2*2/d}) \\ \cos(t/10000^{2*2/d}) \\ \dots \\ \sin(t/10000^{2*\frac{d}{2}/d}) \\ \cos(t/10000^{2*\frac{d}{2}/d}) \end{bmatrix}$$

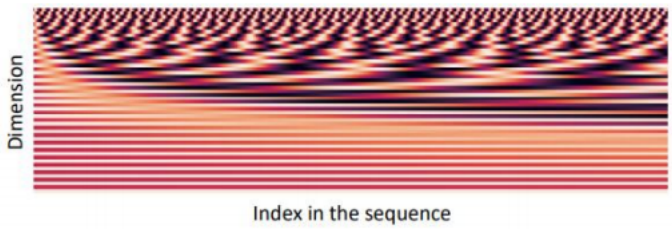
Here, d is the dimensionality positional encoding, and 10000 is an arbitrary choice. In the graphic below, the x axis is t and we are varying i to see how the values of the positional encoding changes.

"i"

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

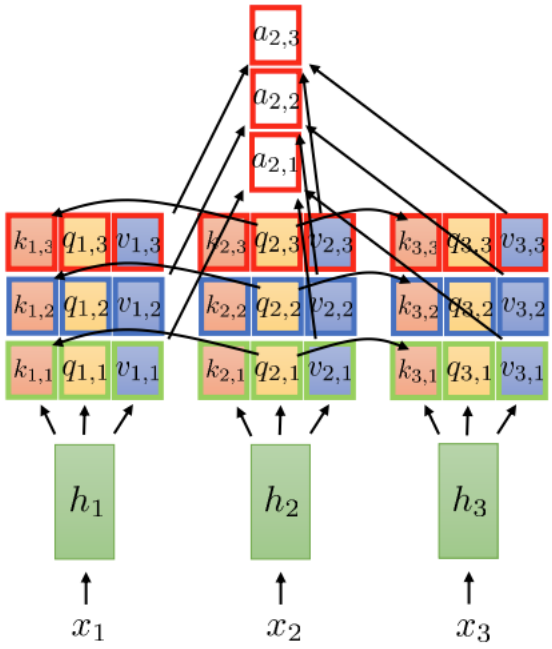


By ranging between different frequencies, we can encode the relative position of t. In the chart below, we see that the first dimension does something like an even-odd indicator of t, and the steps start to take longer as we go down the positional encoding vector.



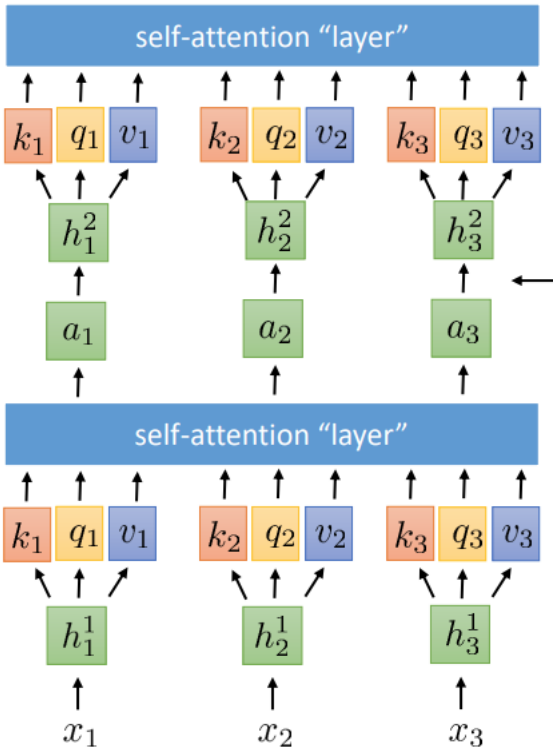
One last method to do positional encoding is to learn it. We could learn a weight vector $P = [p_1, p_2, ..., p_T] \in \mathbb{R}^{d \times T}$ where T is the max sequence length and d is dimensionality of the positional encoding. Every sequence would have these same learned valued, but the learned values would be difference between timesteps. To incorporate positional encoding, we can either concatenate them ($\bar{x}_t = \begin{bmatrix} x_t \\ p_t \end{bmatrix}$) or add them after embedding the input ($\text{emb}(x_t) + p_t$).

6.7.2.2 Multi-headed Attention Since we are relying entirely on attention now, we might want to incorporate more than one time step. With the current design, the softmax will be dominated by one value, and it is hard to specify that you want two different things. To solve this problem, we will just have multiple self-attention heads and then concatenate the result.



In the example above, there are 3 heads. So we will compute weights independently for each head and, in this case, $a_2 = \begin{bmatrix} a_{2,1} \\ a_{2,2} \\ a_{2,3} \end{bmatrix}$.

6.7.2.3 Adding Nonlinearities In the last step of self-attention, $a_l = \sum_t \alpha_{1,t} v_t$. If v_t is just a linear transformation of the hidden state, then every self-attention layer is just a linear transformation of the previous layer (with non-linear weights since α_t is calculated with a softmax). This is not very expressive. So we will apply nonlinearities each step after each self-attention layer. This is sometimes referred to as position-wise feedforward network.



Here the left-arrow points to where the nonlinearity occurs. Rather than passing a_t directly into the next layer, we will pass a neural network through it first.

6.7.2.4 Masked Decoding In our decoder, we require the output from a previous timestep to pass into the current timestep during test time. However, according to our self-attention model, at the previous timestep, we need the key and values of all future timesteps. To solve this issue, we cannot allow self-attention into the future. So we will create an attention variant for the decoder called masked attention where

$$e_{l,t} = \begin{cases} q_l \cdot k_t & l \geq t \\ -\infty & \text{o/w} \end{cases}$$

Essentially, we will be ignoring the key, value pairs for future timesteps. In practice, to do this, $\exp(e_{l,t})$ will be replaced with 0 inside the softmax if $l < t$.

6.7.2.5 Cross-Attention Cross-attention is just an attention network except the queries come from the decoder and the key, value pairs come from the encoder. Note that cross-attention can also be multi-headed.

6.7.2.6 Layer Normalization Batch normalization empirically seems to enable faster and more stable training. However, batch norm is hard to do for sequence models since sequences are different lengths and we often are forced to have small batches due to long sequences. The solution is to use layer norm instead of batch norm, where we normalize across the dimension of the activation rather than the batch size. For an activation, a, the layer norm equations looks like this:

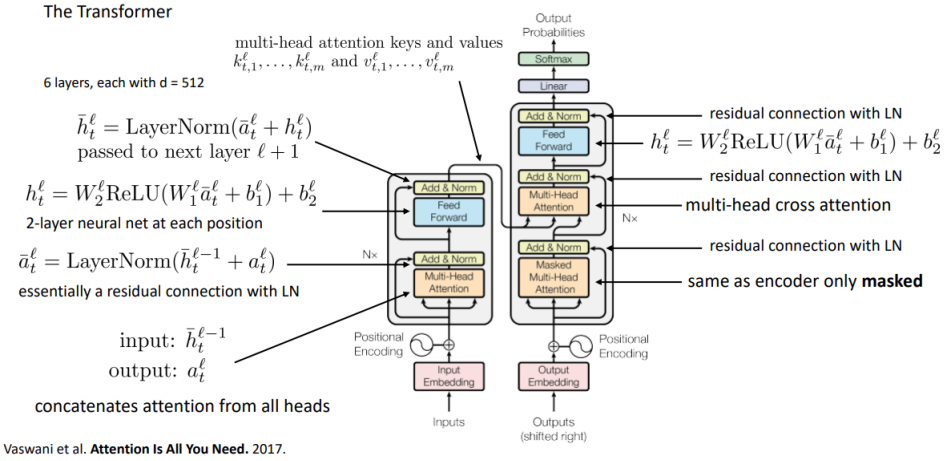
$$\mu = \frac{1}{d}\sum_{i=1}^{d} a_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(a_i - \mu)^2}$$

$$\bar{a} = \frac{a - \mu}{\sigma}\gamma + \beta$$

6.7.3 The Transformer

Now that we have all the components of the transformer, we can finally put it all together to build out the transformer.



Vaswani et al. **Attention Is All You Need.** 2017.

We'll start from the bottom left and then work our way up.

6.7.3.1 The Encoder The left "network" is the encoder. It takes in the inputs, embeds them, and then either concatenates and adds it to the positional encoding (created with whichever scheme we want). With these hidden states calculated, we input this hidden state sequence into the multi-headed attention network. We then get that output, layer norm it, and add it to the hidden state (i.e. a skip connection). This is then passed through a point-wise nonlinear

network, layer normalized again, and then added to the original output of the multi-headed attention network (i.e. another skip connection). The output of this goes into both the decoder as well as back into the start of the encoder as hidden states. For the encoder, we do this N times and thus store N layers of hidden states to input into the decoder.

6.7.3.2 The Decoder The right "network" is the decoder. It takes in either the ground-truth sequence labels during training or just start of sentence token at the first timestep during testing. Once again, this sequence is embedded and positional encodings are added to it. Rather than putting the hidden states into a multi-head attention network, we must put it through a masked multi-headed attention network since future timesteps may not exist yet. We add and norm (i.e. layernorm and skip connection) like with the encoder. Now the outputs here will be put into multi-headed cross-attention layer where the query comes from the decoder and the key, value pairs come from the encoder. We add and norm the output of this, put it through a point-wise nonlinear network, and then add and norm again. We do this N times as well, and then push the output through a linear network and softmax to get word probabilities.

6.7.3.3 Other Details In the paper that proposed this, Attention Is All You Need, they set $N = 6$, $d = 512$, and had 8 heads. So for the multi-headed attention network, since the inputs would be size 512, the key, value, and query vectors would have $512/8 = 64$ since there are 8 heads. Then the output of the attention network would be 8 heads of dimension 64, so after concatenating them together we get back to dimension 512.

6.7.4 Why Transformers?

6.7.4.1 Downsides One downside to transformers are that attention computations are technically $O(n^2)$. For the first issue, the $O(n^2)$ isn't really a huge issue because most of the network is not spent computing the dot products. In other words, most of the computation is done in $O(n)$ and the $O(n^2)$ section is comparatively much smaller. Another downside is that the network is somewhat complicated to implement. So this make it a bit tricky to work with as well as require some hyperparameter tuning to get it working well.

6.7.4.2 Upsides Transformers have really good long-range connections, since every timestep is connected with a jump of length one. They are also easier to parallelize since the entire encoder is entirely parallelizable. In practice, transformers can be quite a bit deeper than RNNs. As a result, transformers seem to work much better than RNNs and LSTMs in many real cases.

6.8 Word Embeddings

6.8.1 Word2Vec

We want a good way to represent words as vectors to be inputted into our language models. We want these vector representations to be meaningful in the sense that vectors corresponding to similar words should be close together. The basic principle to word2vec is that for each word, its neighbors are close under this representation. So for each word in a dataset (i.e. the "center word" c), we will attempt to maximize the probability of its neighboring words (i.e. the "context words" o) given the center word. More formally, we want

$$\underset{u_1,\ldots,u_n,v_1,\ldots,v_n}{\text{argmax}} p(o|c) \text{ s.t. } p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Note that for the ith word, it has two representations: u_i and v_i, for context word representation and center word representation respectively. Having two vectors instead of one makes optimization easier for the softmax. At the end, we can average u_i and v_i to get our representation of word i. Also, the radius for context words can be set to something like 5.

We have one small issue in that the denominator of $p(o|c)$ is really costly to compute since it requires summing over the entire dataset. Instead of summing over the entire dataset, we can instead make the optimization problem into a binary classification problem. So instead, we have

$$p(o|c) = \sigma(u_o^T v_c) = \frac{1}{1 + \exp(-u_o^T v_c)}$$

We also need to add in examples of o that not the right word for c otherwise the optimum for the objective function would involve huge u_i and v_i such that all the binary classifications return true:

$$p(\neg o|c) = \sigma(-u_o^T v_c)$$

Our new optimization problem is now:

$$\underset{u_1,\ldots,u_n,v_1,\ldots,v_n}{\text{argmax}} \sum_{c,o} \Big(\log p(o|c) + \sum_w \log p(\neg w|c) \Big)$$

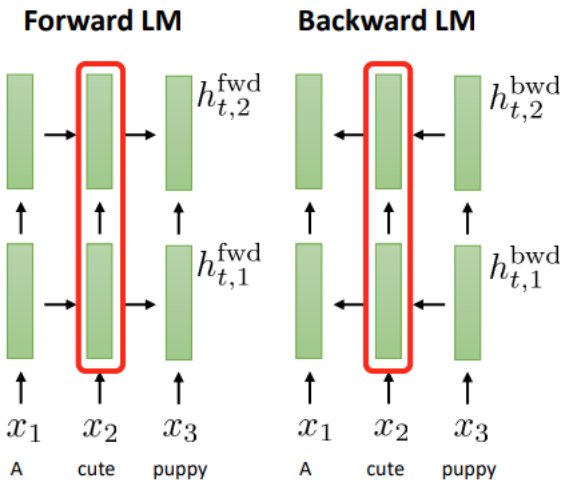
where w are random words that are not context words. Condensing everything together, our optimization problem is:

$$\underset{u_1,\ldots,u_n,v_1,\ldots,v_n}{\text{argmax}} \sum_{c,o} \Big(\log \sigma(u_o^T v_c) + \sum_w \log \sigma(-u_w^T v_c) \Big)$$

6.9 Pretrained Language Models

6.9.1 ELMo

ELMo is a bidirectional LSTM model used for context-dependent embeddings.



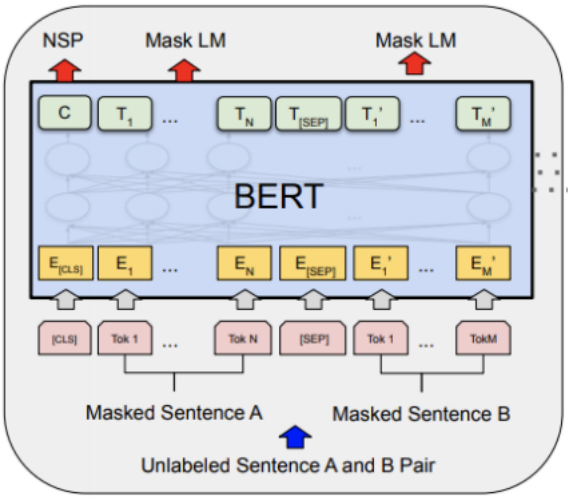
It consists of a forward and backward LSTM trained on a large corpus of unlabeled text data, and

$$\text{ELMO}_t = \gamma \sum_{i=1}^{L} w_i [h_{t,i}^{\text{fwd}}, h_{t,1}^{\text{bwd}}]$$

The ELMo representation of the word embedding is concatenated as input into the downstream task. This provides a context specific and semantically meaningful representation of each token. Note that we have both a forward and backward language model since context both before and after the word may be important for contextual representations of the word. A small additional note is that the weight γ and w_i are typically made to be learnable in the downstream task as well.

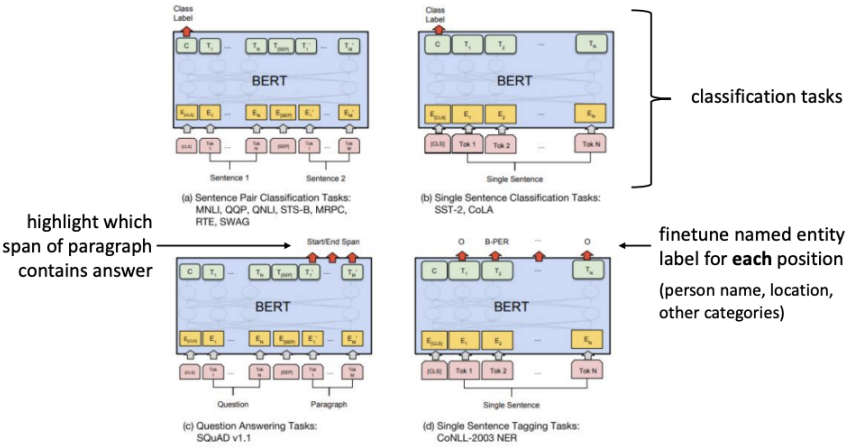
6.9.2 BERT

6.9.2.1 Training BERT BERT is a transformer language model used for context-dependent embeddings (i.e. ELMo but with transformers). It is what is used for most language tasks today. BERT is essentially the "encoder" part of a transformer with 15% of inputs replaced with [MASK]. The issue with just using the encoder part of the transformer (i.e. using self-attention) is that the language model can trivially learn to access the right answer at time t from the input at time $t+1$. So by replacing words with a mask token, we are asking BERT to fill in the missing words and in doing so, create a good representation. The self-attention layer makes BERT bidirectional. A diagram of BERT is attached below:



BERT also has two sentences as input since many downstream tasks require processing two sentences such as question answering. From the diagram we can see that, Masked Sentence A has a Mask LM output and Masked Sentence B has a MASK LM output. Reconstructing all tokens at each time step forces learning context-dependent word-level representations. During training, we also randomly swap the order of the sentences 50% of the time. The output from the first [CLS] token, NSP, is a binary classifier output that predicts whether the first sentences follows the second sentence or precedes it. This forces learning sentence-level representations.

6.9.2.2 Applying BERT to Downstream Tasks For downstream tasks, we now take this pretrained BERT model, put a cross-entropy loss on the first output, and finetune the whole model end-to-end on the new task. Below, we see four example downstream tasks:

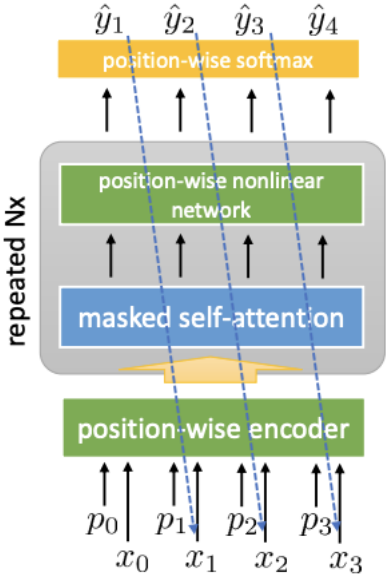


The top two examples are classification tasks. For these tasks, we simply replace our first output loss to be a class label classification loss (no losses on any other outputs). The bottom left example is a question answering task, where we want to highlight where in the paragraph the answer to a question is. So here, we put losses on the paragraph output. For the bottom right example, we want to tag each word of a single sentence with some category (e.g. verb, noun). To do this, we put a loss the sentence outputs.

6.9.2.3 Getting Features from BERT Similar to ELMo, we can also get features from the encoding layers of BERT and pass that in with the input. Since BERT has 12 layers, we have a choice in selecting which layers we want to use.

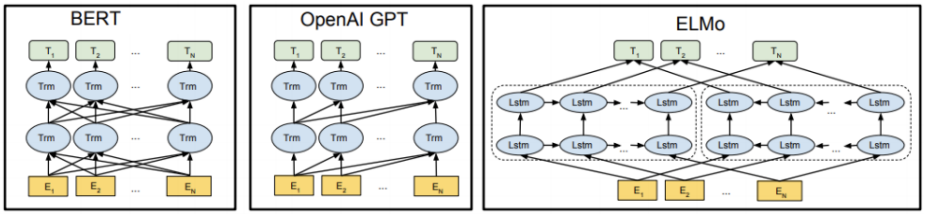
6.9.3 GPT

Text generation isn't really that great with BERT since it uses self-attention to fill in missing words. Here, we can use something like GPT, which is the decoder part of the a transformer (i.e. it uses masked self-attention, so it is a one-directional forward transformer).



As a side note, OpenAI GPT-3 is a newer version of OpenAI GPT-2 with more layers.

6.9.4 Summary



bidirectional transformer
+ great representations
- can't generate text

one-directional transformer
+ can generate text
- OK representations

bidirectional LSTM
- OK representations
(largely supplanted by BERT)

7 Reinforcement Learning

7.1 Imitation Learning

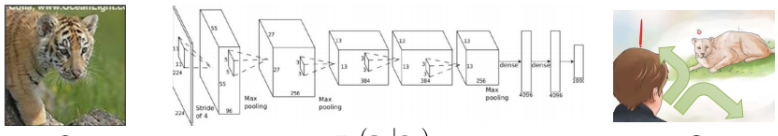
Note that imitation learning isn't part of RL, but provides good context for it.

7.1.1 Prediction to Control: Challenges

Prediction (e.g. image classification) assumes that the data is distributed i.i.d and the objective is to the predict the right label. On the other hand, control (e.g. autonomous driving) may not be i.i.d. Instead, each decision can change future inputs and the objective can be more abstract: to accomplish a task. Unlike prediction problems where there is ground truth supervision, control problems may have high-level supervision (e.g. navigate to location).

7.1.2 Terminology

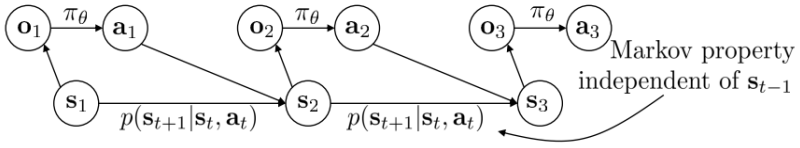
In RL, we have states, observations, and actions. States are everything an agent needs to know about the system. If an agent at a given state takes an action, the environment will output probability distribution of next states given that (state, action) pair. Observations are observations of the state. For this reason, states are thought of as fully observed while observations are thought of as partially observed. The goal of RL is to find the optimal policy, which is a mapping from states to actions, given an agent moving in an environment. As an example, states could be a car's position, velocity, acceleration, etc. while observations could be images of the car moving. Actions would then be the steer, throttle, and brake of the car.



$\mathbf{s}_t$ – state
$\mathbf{o}_t$ – observation
$\mathbf{a}_t$ – action

$\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ – policy
$\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ – policy (fully observed)

In RL, we assume states follow the markov property. In other words, $s_i|s_{i-1}$ is independent of s_j where $j < i - 1$. In other words, if you know the current state, previous states won't help predict future states. Observations don't obey the markov property.



7.1.3 Behavioral Cloning and Distributional Shift

Behaviorial cloning, or imitation learning, is just using supervised learning for control problems. For instance, an observation is the car's camera and the action is turning left or right, and we are given ground truth supervision. In theory, this doesn't work because we have a distributional shift problem, where the input distribution shifts from true strings from training to synthetic strings at test time. In other words, our policy will venture into states that it has never seen before in the training data and make increasingly poor decisions as the error compounds.

7.1.4 Mitigating Distributional Shift

So the problem we have right now is that $p_{\text{data}}(o_t) \neq p_{\pi_\theta}(o_t)$. But maybe we can make our policy $\pi_\theta(a_t|o_t)$ very accurate such that $p_{\text{data}}(o_t) \approx p_{\pi_\theta}(o_t)$.

7.1.4.1 Non-Markovian Behavior One possible reason for error is that the behavior at some timestep depends not only on the current observation. So conditioning on all past observations can help reduce error. In other words, we want $\pi_\theta(a_t|o_1, ..., o_t)$ instead of $\pi_\theta(a_t|o_t)$. One idea for doing this is to use a RNN, LSTM, or Transformer process the sequence of observations.

7.1.4.2 Multimodal Behavior Another reason for error is that the agent can have many valid action sequences given an initial state. But upon averaging these valid action sequences, which learned policies may learn to predict, the action becomes invalid. For instance, a car can swerve left or right around an obstacle, but the average, going straight, is not a valid action. There are three ways to solve this issue:

1. **Output Mixture of Gaussians**: we can have our policy have n heads, each policy learning a u_i, Σ_i, and w_i so that $\pi(a|o) = \sum_i w_i N(u_i, \Sigma_i)$. This can be trained by optimizing the log likelihood of these Gaussians. The issue with this is that in high dimensional spaces, we have need an exponential number of gaussian elements to approximate the higher dimensional distribution. So if we don't have enough heads, this solution may fall flat.

2. **Latent Variable Models**: Given a state, the reason we have different actions is due to a piece of information not present in the observation. These unknown variables are called latent variables, which will be passed

in with the observation to the policy. During test time, the latents will be sampled randomly. During training time, the latent variables can be determined using things like conditional variational autoencoders, normalizing flow/realNVP, and stein variational gradient descent. The former two topics will be covered later.

3. **Autoregressive Discretization**: The third way to represent multimodal distributions is to discretize the actions because a softmax can represent multimodal distributions. The issue is that the number of bins is exponential with respect to the action space size. Autoregressive discretization discretizes one dimension at a time. For instance, our model first predicts a distribution over the binned first element of the action vector. This discrete sample is then passed into the next network, which predicts the distribution of the second element of the action vector, and so on. At test time, we sample each action element according to its predicted softmax distribution.

7.1.4.3 Dataset Aggregation (DAgger) The goal to DAgger is collect training data from $p_{\pi_\theta}(o_t)$ instead of $p_{\text{data}}(o_t)$. The algorithm is:

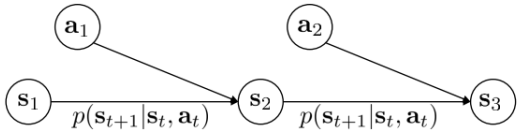
Algorithm 9 DAgger

1: **while** some stopping condition is not satisfied **do**
2: Train $\pi_\theta(a_t|o_t)$ from human data $\mathcal{D} = \{o_1, a_1, ..., o_N, a_N\}$.
3: Run $\pi_\theta(a_t|o_t)$ to get dataset $\mathcal{D} = \{o_1, ..., o_M\}$.
4: Ask human to label $\mathcal{D}_\pi$ with actions a_t.
5: Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$
6: **end while**

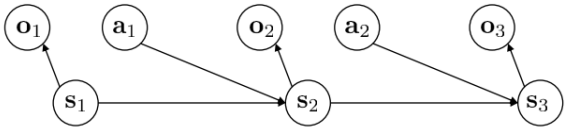
The intuition for why this works is that the more we aggregate the datasets, the closer $p_{\text{data}}(o_t)$ gets to $p_{\pi_\theta}(o_t)$.

7.2 Introduction to Reinforcement Learning

Reinforcement learning is built around a Markov decision process (MDP), where we have $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$. Our reward function, r, maps states, $\mathcal{S}$, and actions, $\mathcal{A}$, to reward values. Our transition operator (a.k.a transition probability, a.k.a dynamics) gives the probability of a next state given a state and action. So our MDP looks like this:



A partially observed MDP is similar to the MDP above but with $\mathcal{E}$, our emission probabilities $p(o_t|s_t)$ and our observation space $\mathcal{O}$. Our POMDP looks like this:



The goal of RL is not to maximize rewards greedily, but to maximize rewards over time. In other words, our optimal policy is

$$\theta^* = \underset{\theta}{\text{argmax}}\, \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)]$$

where

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

7.3 REINFORCE [Introduction to Policy Gradients]

REINFORCE is a policy gradient algorithm, which essentially is gradient ascent on $J(\theta)$. Below we will detail the algorithm and the derivation.

7.3.1 Objective Function

Looking at our optimization problem,

$$\theta^* = \underset{\theta}{\text{argmax}}\, \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)]$$

, we can approximate $J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)]$ with

$$J(\theta) \approx \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

In other words, we run the policy N times, get the total reward, and average them together. For shorthand, we will say that $r(\tau) = \sum_t r(s_t, a_t)$ so we have $J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[r(\tau)]$.

7.3.2 Derivation

We know that

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau) r(\tau) d\tau$$

so

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau$$

A convenient identity we can use to break down this form is:

$$p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau)$$

Applying this identity in reverse, we see that

$$\nabla_\theta J(\theta) = \int p_\theta(\tau)\nabla_\theta \log(p_\theta(\tau))r(\tau)d\tau = \mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log(p_\theta(\tau))r(\tau)]$$

To calculate $\nabla_\theta \log(p_\theta(\tau))$, note that

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1)\prod_{t=1}^{T} \pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

$$\log(p_\theta(\tau)) = \log(p(s_1) + \sum_{t=1}^{T} \log(\pi_\theta(a_t|s_t)) + \log(p(s_{t+1}|s_t, a_t))$$

$$\nabla_\theta \log(p_\theta(\tau)) = \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_t|s_t))$$

so

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_t|s_t))\right)\left(\sum_{t=1}^{T} r(s_t|a_t)\right)\right]$$

We can approximate $\nabla_\theta J(\theta)$ with

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t}))\right)\left(\sum_{t=1}^{T} r(s_{i,t}|a_{i,t})\right)$$

Once we've computed this gradient, we can just do gradient ascent:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$$

We have just described the REINFORCE algorithm, which can be summed up as:

Algorithm 10 REINFORCE

1: **while** some stop condition is not satisfied **do**
2: sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (i.e. run the policy)
3: Calculate $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})))(\sum_{t=1}^{T} r(s_{i,t}|a_{i,t}))$
4: $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
5: **end while**

7.3.3 Intuition

Note that for MLE supervised learning,

$$\nabla_\theta J_{ML}(\theta) \approx \sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t}))\right)$$

So policy gradient is essentially, in terms of implementation, just MLE but with a $\sum_{t=1}^{T} r(s_t|a_t)$ weight for each rollout. In other words, positive reward is made more likely and negative reward is made less likely. So, in practice, REINFORCE is basically learning by trial and error. Also note that REINFORCE works for partially observability as well, so we can also have

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|o_{i,t})) \right) \left(\sum_{t=1}^{T} r(s_{i,t}|a_{i,t}) \right)$$

7.3.4 Making REINFORCE Work

The current implementation of REINFORCE never works, but we can make some additional improvements to get it to sometimes work.

7.3.4.1 Causality Currently, we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) \right) \left(\sum_{t=1}^{T} r(s_{i,t}|a_{i,t}) \right)$$

If we distribute our reward sum in, we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) \left(\sum_{t'=1}^{T} r(s_{i,t'}|a_{i,t'}) \right)$$

It turns out that in expectation, this is the same as

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) \left(\sum_{\boxed{\text{t'=t}}}^{T} r(s_{i,t'}|a_{i,t'}) \right)$$

(***) This is because for $t < t'$, the policy at time t' cannot affect reward at time t. This estimator is now actually better because, even though in expectation the rewards taken out always integrate to zero, for a finite number of samples, our sampling error might influence our gradient. By zeroing out this noise, we have decreased the variance of our estimator. This estimator is always better. As a small note, we call $\hat{Q}_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}|a_{i,t'})$ the "reward to go", so

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) \hat{Q}_{i,t}$$

7.3.4.2 Baselines In our intuition subsection, we noted that REINFORCE makes positive reward trajectories more likely and negative reward trajectories less likely. However, if all our rewards are positive, we run into a clear problem.

Instead, we want to make trajectories more likely not if their reward is positive, but if their rewards are better than average. So instead of

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log(\pi_\theta(\tau)) r(\tau)$$

we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log(\pi_\theta(\tau))[r(\tau) - b] \text{ s.t. } b = \frac{1}{N}\sum_{i=1}^{N} r(\tau)$$

Also note that $\nabla_\theta \log(\pi_\theta(\tau))$ is shorthand for $\sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t}))$ and $r(\tau)$ is shorthand for $\sum_{t=1}^{T} r(s_{i,t}|a_{i,t})$.

It turns out that subtracting the baseline is unbiased in expectation. This is because

$$\begin{aligned}
\mathbb{E}[\nabla_\theta \log(\pi_\theta(\tau) b] &= \int \pi_\theta(\tau) \nabla_\theta \log(\pi_\theta(\tau)) b d\tau \\
&= \int \nabla_\theta \pi_\theta(\tau) b d\tau \\
&= b \nabla_\theta \int \pi_\theta(\tau) d\tau \\
&= b \nabla_\theta 1 \\
&= 0
\end{aligned}$$

By subtracting the mean, we are then reducing the variance of the estimator and making the estimator more accurate. As a side note, the average reward is not actually the best baseline to use, but it is pretty good.

7.3.5 Off-Policy Learning

REINFORCE is an on-policy learning algorithm because at each iteration, we need to rerun the new updated policy in the environment. This can be extremely inefficient. Instead, what if we want to do is off-policy learning, where we don't have samples from $\pi_\theta(\tau)$ but some $\bar{\pi}(\tau)$. To do this, we must first understand importance sampling.

7.3.5.1 Importance Sampling (IS) We can rewrite the expectation over one distribution as the expectation over another as such:

$$\begin{aligned}
\mathbb{E}_{x\sim p(x)}[f(x)] &= \int p(x) f(x) dx \\
&= \int q(x) \frac{p(x)}{q(x)} f(x) dx \\
&= \mathbb{E}_{x\sim q(x)}[\frac{p(x)}{q(x)} f(x)]
\end{aligned}$$

7.3.5.2 Policy Gradient with IS With this in mind, we can rewrite $J(\theta)$ in distribution of $\bar{\pi}(\tau)$.

$$\begin{aligned}
J(\theta) &= \mathbb{E}_{\tau\sim\bar{\pi}(\tau)}\left[\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)}r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\bar{\pi}(\tau)}\left[\frac{p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)}{p(s_1)\prod_{t=1}^{T}\bar{\pi}(a_t|s_t)p(s_{t+1}|s_t,a_t)}r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\bar{\pi}(\tau)}\left[\frac{\prod_{t=1}^{T}\pi_\theta(a_t|s_t)}{\prod_{t=1}^{T}\bar{\pi}(a_t|s_t)}r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\bar{\pi}(\tau)}\left[\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)}r(\tau)\right]
\end{aligned}$$

Now that we have rewritten $J(\theta)$, let us estimate $J(\theta')$ given some rollout of policy $\pi_\theta(\tau)$.

$$\begin{aligned}
J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)}r(\tau)\right] \\
\nabla_{\theta'}J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(\tau)}{\pi_\theta(\tau)}r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)\nabla_{\theta'}\log(\pi_{\theta'}(\tau))}{\pi_\theta(\tau)}r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log(\pi_{\theta'}(a_t|s_t))\right)\left(\sum_{t=1}^{T}r(s_t,a_t)\right)\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log(\pi_{\theta'}(a_t|s_t))\sum_{t'=t}^{T}r(s_{t'},a_{t'})\right)\right]\text{(by causality)} \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log(\pi_{\theta'}(a_t|s_t))\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right)\left(\sum_{t'=t}^{T}r(s_{t'},a_{t'})\left(\prod_{t''=t}^{t'}\frac{\pi_{\theta'}(a_{t''}|s_{t''})}{\pi_\theta(a_{t''}|s_{t''})}\right)\right)\right]
\end{aligned}$$

It turns out that after deleting the importance weights for future trajectories, we still have a valid policy gradient:

$$J(\theta') = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log(\pi_{\theta'}(a_t|s_t))\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right)\left(\sum_{t'=t}^{T}r(s_{t'},a_{t'})\right)\right)\right]$$

The reason why this is valid was not explained in class, so I will leave a mark here for if I decide to come back to prove this later (***).

7.3.5.3 First-order Approximation for IS One issue is that our importance weights from timestep 1 to t become degenerate exponentially fast. So

instead, we can write out objective differently. Instead of writing it as an expectation over trajectories, we can write it as a expectation over (state, action) marginals. Then we can compute an importance weight over these marginals (***)

$$\begin{aligned}\nabla_{\theta'} J(\theta') &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t},a_{i,t})}{\pi_{\theta}(s_{i,t},a_{i,t})}\nabla_{\theta'}\log(\pi_{\theta'}(a_{i,t}|s_{i,t}))\hat{Q}_{i,t} \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t})}{\pi_{\theta}(s_{i,t})}\frac{\pi_{\theta'}(a_{i,t}|s_{i,t})}{\pi_{\theta}(a_{i,t}|s_{i,t})}\nabla_{\theta'}\log(\pi_{\theta'}(a_{i,t}|s_{i,t}))\hat{Q}_{i,t} \\ &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(a_{i,t}|s_{i,t})}{\pi_{\theta}(a_{i,t}|s_{i,t})}\nabla_{\theta'}\log(\pi_{\theta'}(a_{i,t}|s_{i,t}))\hat{Q}_{i,t}\end{aligned}$$

For the last line, notice that our estimator is no longer unbiased. But if θ and θ' are very close together, the error can be bounded (***).

7.3.6 Policy Gradient in Practice

Policy gradients are high variance, so more samples and larger batches will be needed in practice due to gradients being really noisy. Tweaking learning rates will also be harder, and adaptive step-size such as ADAM become much more important. Policy gradient (REINFORCE) can be used in any setting where we have to differentiate through a stochastic but non-differentiable operation.

7.4 Actor-Critic [Policy Gradients Formalized]

7.4.1 Improving Policy Gradient

Without accounting for baseline, our policy gradient is:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_{\theta}\log(\pi_{\theta}(a_{i,t}|s_{i,t}))\hat{Q}^{\pi}_{i,t}$$

Our reward-to-go, $\hat{Q}_{i,t}$ is a one-sample estimator $\sum_{t'=1}^{T} r(s_{i,t'},a_{i,t'})$. What we really want is $\hat{Q}_{i,t} \approx \sum_{t'=t}^{T}\mathbb{E}_{\pi_\theta}[r(s_{t'},a_{t'})|s_t,a_t]$, which is the true expected reward-to-go. Then our policy gradient becomes:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_{\theta}\log(\pi_{\theta}(a_{i,t}|s_{i,t}))Q(s_{i,t},a_{i,t})$$

where $Q(s_t,a_t) = \sum_{t'=t}^{T}\mathbb{E}_{\pi_\theta}[r(s_{t'},a_{t'})|s_t,a_t]$. We now need to account for the baseline, which is $b_t = \frac{1}{N}\sum_i Q(s_{i,t},a_{i,t})$. In expectation, we define this as $V(s_t) = \mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}[Q(s_t,a_t)]$, so then

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_{\theta}\log(\pi_{\theta}(a_{i,t}|s_{i,t}))(Q(s_{i,t},a_{i,t}) - V(s_{i,t}))$$

We call $A(s_{i,t}, a_{i,t}) = Q(s_{i,t}, a_{i,t}) - V(s_{i,t})$ the advantage, which is how much better the Q-value is than the average value at state s. So then,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) A(s_{i,t}, a_{i,t})$$

7.4.2 Terminology

- $Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t]$ is the Q-function, which is the total reward from taking a_t in s_t in expectation.
- $V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q^\pi(s_t, a_t)]$ is the value function, which is the expected value of the Q-function where the actions are distributed according to the policy π.
- $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ is the advantage function, which is how much better a_t is than the average action of state s_t.
- $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) A^\pi(s_{i,t}, a_{i,t})$ is the policy gradient

Note that our old policy gradient has $A^\pi(s_{i,t}, a_{i,t}) = \sum_{t'=1}^{T} r(s_{i,t'}, a_{i,t'}) - b$, which is unbiased but is a high variance single-sample estimate.

7.4.3 Policy Evaluation (i.e. Value Function Fitting)

Note that

$$\begin{aligned} Q^\pi(s_t, a_t) &= r(s_t, a_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] \\ &= r(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)}[V^\pi(s_{t+1})] \\ &\approx r(s_t, a_t) + V^\pi(s_{t+1}) \end{aligned}$$

The last approximation is a single-sample estimate of s_{t+1}, but it is not a single-sample estimate of the rest of the trajectory because we are using V^π. So we have a little approximation error when we do this. But this is very convenient because now

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t)$$

So one idea is that we only have to fit the value function, and then we can calculate the advantage function directly from the value function. So we will have one network with parameters θ be the policy and one network with parameters ϕ try to approximate the value function.

7.4.3.1 Monte Carlo Policy Evaluation Monte Carlo policy evaluation is just what we did with REINFORCE where we just sample trajectories and use these to update our policy. In this case, we can approximate

$$V^{\pi}(s_t) \approx \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$$

If we have multiple samples, we have

$$V^{\pi}(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$$

but this is typically not going to happen since normally, we can only initialize our agent at the initial states and roll it out. This means we can't initialize our agents at intermediate states.

Thus, we will train our value function network with training data:

$$\{(s_{i,t}, y_{i,t})\}$$
$$\text{s.t. } y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$$

Our supervised regression loss will be

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_{i} \|\hat{V}_{\phi}^{\pi}(s_i) - y_i\|^2$$

The neural network will average together values of nearby states, so we will get better value function estimates then directly plugging in the values into policy gradient.

7.4.3.2 MC Policy Evaluation with Bootstrapping Our current Monte Carlo target is $y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$, but our ideal target is

$$\begin{aligned} y_{i,t} &= \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'}) | s_{i,t}] \\ &\approx r(s_{i,t}, a_{i,t}) + V^{\pi}(s_{i,t+1}) \\ &\approx r(s_{i,t}, a_{i,t}) + V_{\phi}^{\pi}(s_{i,t+1}) \end{aligned}$$

The last line is called bootstrapping, where we directly use our previous fitted value function in our label. So our new training data is:

$$\{(s_{i,t}, y_{i,t})\}$$
$$\text{s.t. } y_{i,t} = r(s_{i,t}, a_{i,t}) + \hat{V}_{\phi}^{\pi}(s_{i,t+1})$$

Our supervised regression loss will be

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_{i} \|\hat{V}_{\phi}^{\pi}(s_i) - y_i\|^2$$

7.4.4 Actor-Critic

The term actor-critic refers to the fact that we have two neural networks now: the policy which is the the actor and the critic which is the value function, which criticizes the policy and tries to estimate its value. Actor-critic is really just REINFORCE but with another network estimating the value function rather than directly fixing the value to be the single-sample estimators.

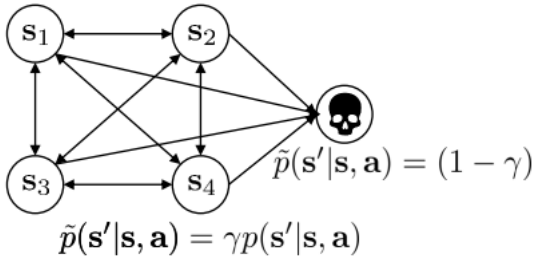
7.4.4.1 Batch Actor-Critic We can sample trajectories in batches, update the value function network, calculate the advantage, use it compute the policy gradient, do gradient descent, and then repeat.

Algorithm 11 Batch Actor-Critic

1: **while** some stop condition is not satisfied **do**
2: Sample $\{s_i, a_i\}$ from $\pi_\theta(a_t|s_t)$ (i.e. run the policy)
3: Fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums
4: Evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$
5: Calculate $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log(\pi_\theta(a_i|s_i))\hat{A}^\pi(s_i, a_i)$
6: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
7: **end while**

In step 2, we fit $\hat{V}^\pi_\phi(s)$ with bootstrap (i.e. Monte Carlo policy evaluation with bootstrapping).

7.4.4.2 Discount Factor Also note that we have discount factors, γ, since without them, $\hat{V}^\pi_\phi$ can become infinitely large as episode length gets longer. So by discounting, we are really saying that rewards sooner are better than rewards later. Discount factors are between 0 and 1. Something like 0.99 works well. With the discount factors, we still have a valid MDP because we can think about at every timestep, we have a $1 - \gamma$ probably of entering the death state.



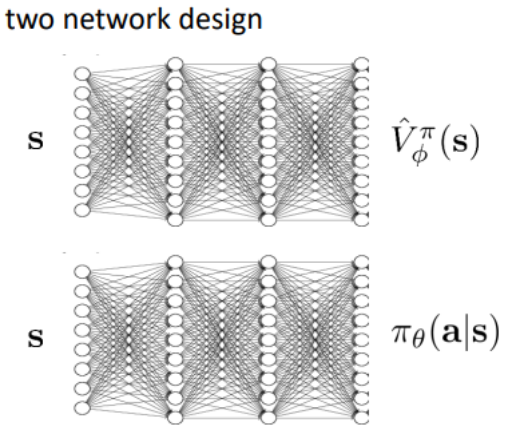
7.4.4.3 Online Actor-Critic A fully online actor critic can actually just update the actor and critic on each action of the policy.

Algorithm 12 Online Actor-Critic

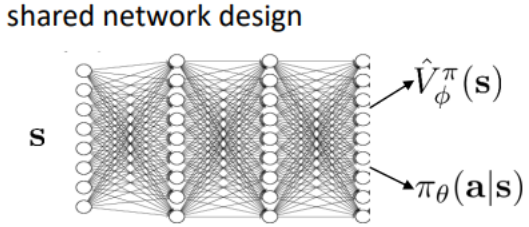
while some stop condition is not satisfied **do**
  Take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
  Update $\hat{V}_\phi^\pi$ using target $r + \gamma \hat{V}_\phi^\pi(s')$
  Evaluate $\hat{A}^\pi(s,a) = r(s,a) + \gamma \hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$
  Calculate $\nabla_\theta J(\theta) \approx \nabla_\theta \log(\pi_\theta(a|s)) \hat{A}^\pi(s,a)$
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end while

In reality, SGD with a single sample is not very stable, so we actually could use a multithreading process where different threads collect a sample and we average over those samples to get a minibatch. This is called asynchronous-advantage actor critic.

7.4.4.4 Architecture Design A simple architecture choice is to have two separate neural networks: one maps states to scalar values (i.e. value function), and one maps states to distributions of actions (i.e. policy). This design is simple and stable, but it can a little inefficient since there are no shared features between actor and critic.



A more advanced design is one in which we have a single neural network with two heads, one with the scalar value output and one with the output being a distribution of actions.



7.5 Policy Iteration, Value Iteration, and Q-Iteration

7.5.1 Policy Iteration

$\text{argmax}_{a_t} A^\pi(s_t, a_t)$ is the best action from s_t if we follow π, which is at least as good as any $a_t \sim \pi(a_t|s_t)$. So, what if we don't have a policy network. Instead, we'll fix

$$\pi'(a_t|s_t) = \begin{cases} 1 & a_t = \text{argmax}_{a_t} A^\pi(s_t, a_t) \\ 0 & \text{otherwise} \end{cases}$$

This new policy will be as good as π (probably better). On a high level, policy iteration would then just be:

Algorithm 13 Policy Iteration (High Level)

1: **while** some stop condition is not satisfied **do**
2: evaluate $A^\pi(s, a)$
3: set $\pi \leftarrow \pi'$
4: **end while**

7.5.2 Policy Iteration with Dynamic Programming

But how do we evaluate the advantage function? Let's start simple. Let's assume we know $p(s'|s, a)$ and both s and a are both discrete and small. We can perform the bootstrapped update:

$$V^\pi(s) \leftarrow \mathbb{E}_{a\sim\pi(a|s)}[r(s, a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)}[V^\pi(s')]]$$

Since π is deterministic, we have

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma \mathbb{E}_{s'\sim p(s'|s,\pi(s))}[V^\pi(s')]$$

So our policy iteration algorithm would look like:

Algorithm 14 Policy Iteration with Dynamic Programming

1: **while** our value function has not converged **do**
2: evaluate $V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma \mathbb{E}_{s'\sim p(s'|s,\pi(s))}[V^\pi(s')]$
3: set $\pi \leftarrow \pi'$
4: **end while**

7.5.3 Value Iteration with Dynamic Programming

Recall that $A^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \mathbb{E}[V^\pi(s'_i)] - V^\pi(s_i)$ and

$$\pi'(a_t|s_t) = \begin{cases} 1 & a_t = \text{argmax}_{a_t} A^\pi(s_t, a_t) \\ 0 & \text{otherwise} \end{cases}$$

We see that the the $V^\pi(s)$ term doesn't depend on a, so

$$\underset{a_t}{\text{argmax}}\, A^\pi(s_t, a_t) = \underset{a_t}{\text{argmax}}\, Q^\pi(s_t, a_t)$$

So we can actually skip the policy and compute values directly.

Algorithm 15 Value Iteration

while our value function has not converged **do**
  set $Q(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}[V(s')]$
  set $V(s) \leftarrow \max_a Q(s,a)$
end while

7.5.4 Fitted Value Iteration

In most problems, our state and action space will be too big to store in a tabular form. So we will have a neural network that maps states to values.

Algorithm 16 Fitted Value Iteration

while some stop condition is not satisfied **do**
  set $y_i \leftarrow \max_{a_i}(r(s_i, a_i) + \gamma \mathbb{E}[V_\phi(s_i')])$
  set $\phi \leftarrow \text{argmin}_\phi \frac{1}{2} \sum_i \|V_\phi(s_i) - y_i\|^2$
end while

7.5.5 Fitted Q-Iteration

However, we have an issue here. There is no way to evaluate the max without knowing the transition probabilities. To solve this, instead of fitting $V^\pi(s)$:

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma \mathbb{E}_{s' \sim p(s'|s,\pi(s))}[V^\pi(s')]$$

we will fit $Q^\pi(s,a)$:

$$Q^\pi(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}_{s' \sim p(s'|s,a)}[Q^\pi(s', \pi(s')]$$

In doing so we can approximate $\mathbb{E}_{s' \sim p(s'|s,\pi(s))}[V_\phi(s_i')] \approx \max_{a'} Q_\phi(s_i', a_i')$.

Algorithm 17 Fitted Q-Iteration

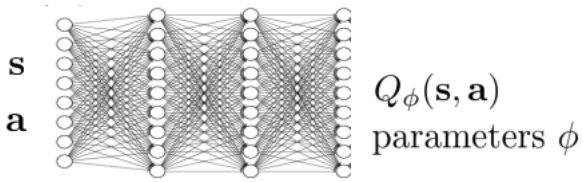
while some stop condition is not satisfied **do**
  set $y_i \leftarrow \max_{a_i}(r(s_i, a_i) + \gamma \mathbb{E}[V_\phi(s_i')])$ s.t. $\mathbb{E}[V_\phi(s_i')] \approx \max_{a'} Q_\phi(s_i', a_i')$
  set $\phi \leftarrow \text{argmin}_\phi \frac{1}{2} \sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$
end while

This turns out to work for off-policy samples (unlike actor-critic), because the Q-function is conditioned on action. We want only have one network, so there is no need for a high-variance policy gradient. The downside is that this algorithm loses the convergence guarantees for non-linear function approximation that policy gradients have (***).

7.5.6 Fully Fitted Q-Iteration

Algorithm 18 Fully-Fitted Q-Iteration

collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy
for K iterations **do**
  set $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$
  set $\phi \leftarrow \text{argmin}_\phi \frac{1}{2} \sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$
end for



7.6 Q-Learning

7.6.1 Online Q-Iteration

Now that we know fully-fitted Q-iteration, we can also implement the online version (a.k.a Watkin's Q-learning), which takes one step in the environment, observes one transition, computes one target value, and then takes one gradient step of SGD.

Algorithm 19 Online Q-Iteration

while some stop condition is not satisfied **do**
  take some action a_t and observe (s_i, a_i, s'_i, r_i)
  set $y_i = r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$
  set $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$
end while

This forms the foundation of deep Q-learning, but we need to add some fixes to get this working for deep learning.

7.6.2 Exploration with Q-Learning

One issue we have is that we only choose actions with respect to the argmax policy. This is an issue because our initial Q-function is really bad, it is possible we choose the same bad action and never seen the rest of the environment. So instead, we can have some exploration strategy. A simple one is the epsilon-greedy strategy where

$$\pi(a_t|s_t) = \begin{cases} 1 - \epsilon & a_t = \text{argmax}_{a_t} Q_\phi(s_t, a_t) \\ \frac{\epsilon}{|\mathcal{A}|-1} & \text{otherwise} \end{cases}$$

In other words, we choose the argmax most of the time. But sometimes we will choose another random action for exploration. A more sophisticated strategy is Boltzmann exploration where:

$$\pi(a_t|s_t) \propto \exp(Q_\phi(s_t, a_t))$$

This strategy allows for the agent to rarely go to actions it knows is bad and most commonly go to action it knows is good.

7.6.3 Q-Learning with Replay Buffer

Another issue we have is that in our online Q-Iteration, our samples are correlated. This is different from SGD where we assume the data is chosen i.i.d. Since sequential states are strongly correlated, it is possible our Q-value network will overfit to different chunks along a training trajectory. To solve this, we introduce replay buffers, which stores a dataset of the agent's most recent trajectories (old trajectories are thrown away when the replay buffer hits a threshold limit). Then we will do gradient descent on a batch of the replay buffer rather than a sample:

Algorithm 20 Full Q-Learning with Replay Buffer

1: **while** some stop condition is not satisfied **do**
2: collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
3: **for** K iterations **do**
4: sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$
5: $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)])$
6: **end for**
7: **end while**

7.6.4 Q-Learning with Target Networks

We have one more issue in that our Q network changes every gradient step, so our target changes every gradient step. As a result, it is possible that this type of "gradient descent" won't converge, since our network is sort of "chasing its own tail". The solution to this is to save an old version of the model for gradient descent and take multiple gradient steps before updating a newer version of the model for gradient descent. We call this old version of the model to be used in the loss function for gradient descent the target network.

Algorithm 21 Q-Learning with Replay Buffer and Target Network

while some stop condition is not satisfied **do**
  save target network parameters: $\phi' \leftarrow \phi$
  for N iterations **do**
    collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
    for K iterations **do**
      sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$
      $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_{\phi'}(s'_i, a'_i)])$
    end for
  end for
end while

7.6.5 Classic Deep Q-Learning (DQN)

Using target networks and replay buffers, we can construct the classic DQN:

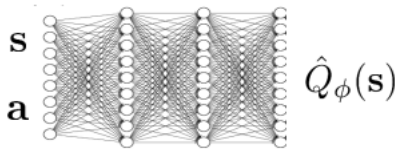
Algorithm 22 Classic Deep Q-Learning (DQN)

while some stop condition is not satisfied **do**
  take some action a_i and observe (s_i, a_i, s'_i, r_i), add it to $\mathcal{B}$
  sample mini-batch (s_j, a_j, s'_j, r_j) from $\mathcal{B}$ uniformly
  compute $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using target network $Q_{\phi'}$
  $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_j)$
  update ϕ': copy ϕ every N steps
end while

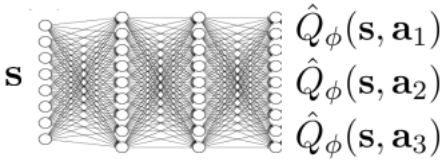
This is essentially Q-Learning with Replay Buffer and Target Network with $K = 1$.

7.6.6 Q-Function Network Representation

We have a choice how we want to represent the Q-function network. One option would be to input state and actions and output a scalar value that represents the Q-value. This is more common for continuous actions.



Another option would be to input just the state and have multiple heads as output for each action. This is more common for discrete actions.



7.6.7 Off-Policy Actor Critic

Taking the max over continuous actions can be very nontrivial. One solution is to use off-policy actor-critic. Note that normal actor-critic is only an on-policy algorithm, but we can use Q-functions to get around this:

Algorithm 23 Off-Policy Actor-Critic

while some stop condition is not satisfied **do**
  take some action a_i and observe (s_i, a_i, s'_i, r_i), add it to $\mathcal{B}$
  sample mini-batch (s_j, a_j, s'_j, r_j) from $\mathcal{B}$ uniformly
  compute $y_j = r_j + \gamma \mathbb{E}_{a'_j \sim \pi'_\theta(a'_j|s'_j)}[Q_{\phi'}(s'_j, a'_j)]$ using target ϕ' and θ'
  $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_j)$
  $\theta \leftarrow \theta + \beta \sum_j \nabla_\theta \mathbb{E}_{a \sim \pi_\theta(a|s_j)}[Q_\phi(s_j, a)]$
  update ϕ' and θ' every N steps
end while

7.6.8 Q-Learning in Practice

Q-Learning takes some care to stabilize, so the recommendation is to first test on easy, reliable tasks first. Large replay buffers tend to help improve stability. Q-Learning also takes time due to its exploration challenge, so it might be no better than random for a while before rapid improvement. For this reason, it helps to start with high exploration (epsilon) and gradually reduce.

8 Unsupervised Learning

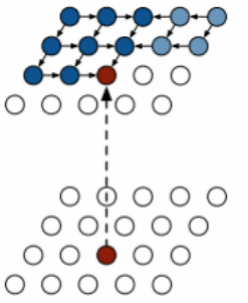
8.1 Autoregressive Generative Models

Autoregressive generative models are "language models" for other types of data, although the more accurate way is to say that language models are a special type of autoregressive generate models. We can representive autoregressive models with RNNs/LSTMs, local context models like PixelCNNs, or transformers. They provide full distribution with probabilities, are conceptually very simple, but are very slow for large datapoints such as images so they are generally limited in image resolution. The main principle behind training autoregressive generative models is for the unlabeled dataset made up of datapoints x,

1. Divide up x into dimensions $x_1, ..., x_n$
2. Discretize each x_i into k values
3. Model p(x) via the chain rule $p(x) = p(x_1)p(x_2|x_1)p(x_3|x_{1:2})...$
4. Use your favorite sequence model to model p(x)

8.1.1 PixelRNN

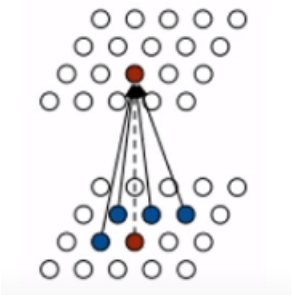
In a PixelRNN, pixels are generated one at a time, left-to-right, top-to-bottom, one color channel at a time. In other words, we train on LSTM on the pixels of an image that is flattened so that it predicts the next pixel (i.e. 256-way softmax) given what it currently has. Some issues with PixelRNN is that it is really slow due to the sheer number of pixels in an image. Also, a row-by-row LSTM might struggle to capture the spatial context (i.e. pixels right above it are far away). Some idea proposed to get over this is the diagonal BiLSTM, where we short the connections between hidden states of pixels right above to the hidden state of the current pixel.



Diagonal BiLSTM

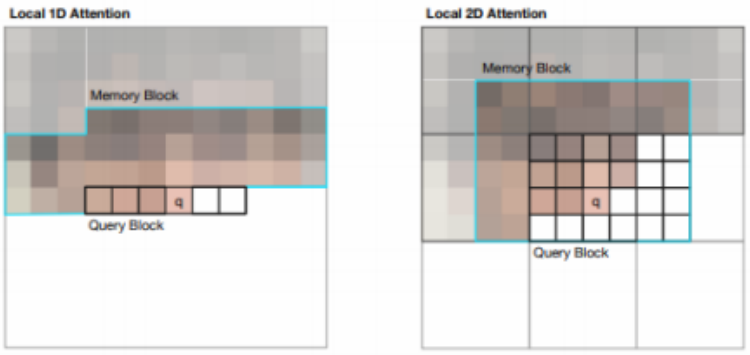
8.1.2 PixelCNN

A PixelCNN is a much faster version of PixelRNN by not building a full RNN over all pixels, but just using a convolution to determine the value of a pixel based on its neighborhood. To do this, we will use a CNN but at each pixel, we mask out that pixel, pixels below it, and pixels to the right. Note that we must use zero-padding so that we keep the resolution of our output (i.e. the 256-way softmax) to be the same as the resolution of our input image. We can parallel training now, but we can't parallelize generation because we need previous pixels to determine future pixels. Also, it may seem that given a pixel, we are no longer conditioning on all the past pixels anymore but rather just the pixels that aren't masked in its neighborhood. However, since pixels are generated one at a time, even pixels outside of the neighborhood will influence our prediction indirectly.



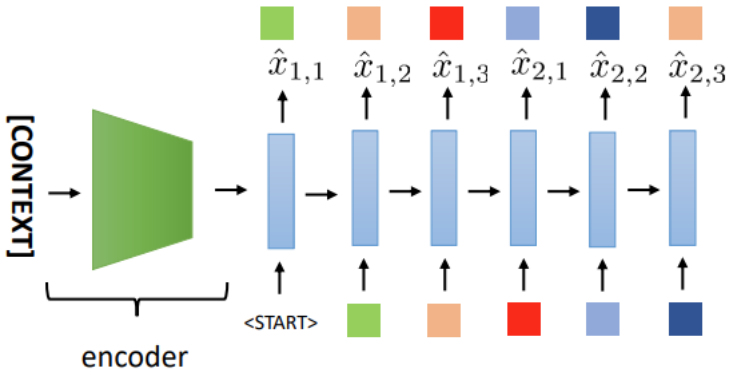
8.1.3 Pixel Transformer

We can use a transformer decoder-style architecture to build a model over images. Note that in self-attention, all pixels are equally "close" so we don't have the spatial context issue as in PixelRNN. However, the number of pixels can be huge and attention models can become prohibitively expensive. One idea is to only compute attention for pixels that are not too far away. So a pixel transformer uses a modified masked self-attention where it masks out pixels too far away in addition to future pixels. There are two ways to mask out pixels: Local 1D Attention is much like PixelRNN but with self-attention while Local 2D Attention is much PixelCNN but with self-attention.



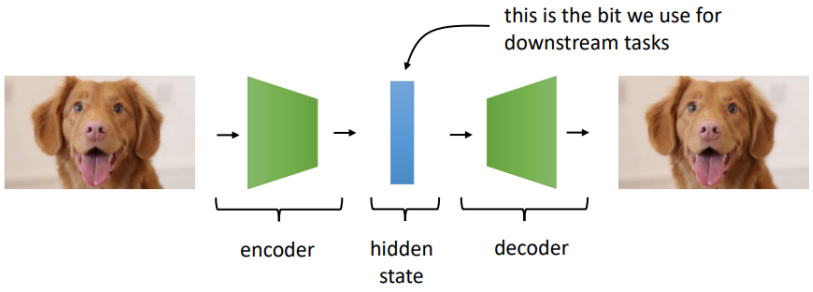
8.1.4 Conditional Autoregressive Models

We can also have autoregressive generative models that are conditioned on another piece of information. For instance, we can generate images of specific types of objects or generate distributions over actions for imitation learning conditioned on the observation. To do this, we encode our context information and pass it into the first timestep of our autoregressive generative model.



8.2 Autoencoders

An autoencoder is a network that encodes an image into some hidden state, and then decodes that image as accurately as possible from that hidden state. Something about the design of the model, or in the dat processing or regularization, must force the autoencoder to learn a structured representation.



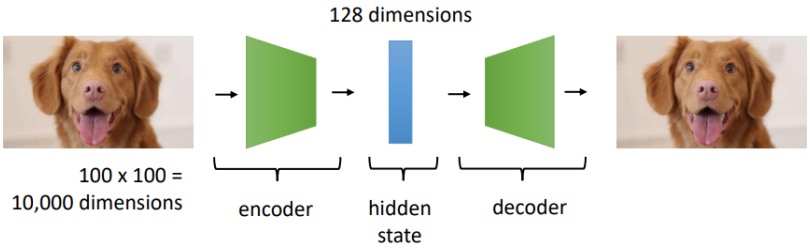
8.2.1 Types of Autoencoders

We can force structure in a few different ways:

- Dimensionality: make the hidden state smaller than the input/output, so the network must compress it
  - Very simple to implement
  - Simply reducing dimensionality often doesn't provide the structure we want. It may actually lump together dissimilar things.
- Sparsity: Force the hidden state to be sparse (most entries are zero), so that the network must compress the input
  - Principled approach that can provide a "disentangled" representation
  - Harder in practice, requires choosing the regularizer and adjusting hyperparameters
- Denoising: Corrupt the input with noise, forcing the autoencoder to learn to distinguish noise from signal
  - Very simple to implement
  - Not clear which layer to choose for the bottleneck, many ad-hoc choices (e.g. how much noise to add)
- Probabilistic Modeling: Force the hidden state to agree with a prior distribution

8.2.2 Bottleneck Autoencoder

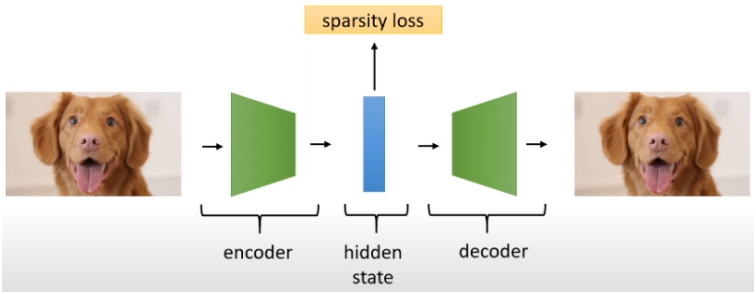
The classic bottleneck autoencoder does exactly dimensionality reduction.



If the encoder and decoder are linear, this exactly recovers PCA. So bottleneck autoencoders can be seen as "non-linear dimensionality reduction". This could be useful because dimensionality is lower and we can use various algorithms that are only tractable in low-dimensional spaces (e.g. discretization). This design is antiquated, but good to know about historically.

8.2.3 Sparse Autoencoder

In a sparse autoencoder, in addition to the reconstruction loss, we have an additional loss to encourages our hidden state to be sparse (i.e. most values are zero). This motivation behind this is that sparse representations are more structured since there are many attributes in the world, and images don't have most attributes.

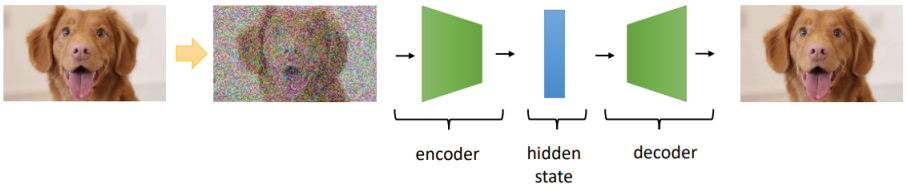


Note that the dimensionality of the hidden state can be larger than the input. This is called an overcomplete representation. We may want this because there may be because a very large number of attributes. There are a variety of sparsity losses we can employ. A simple sparsity loss is to regularize the hidden states by the L1 norm. There are other sparsity losses like lifetime sparsity (i.e. force every dimension of the hidden state to be non-zero in a small fraction of the images) or spike and slab models.

8.2.4 Denoising Autoencoder

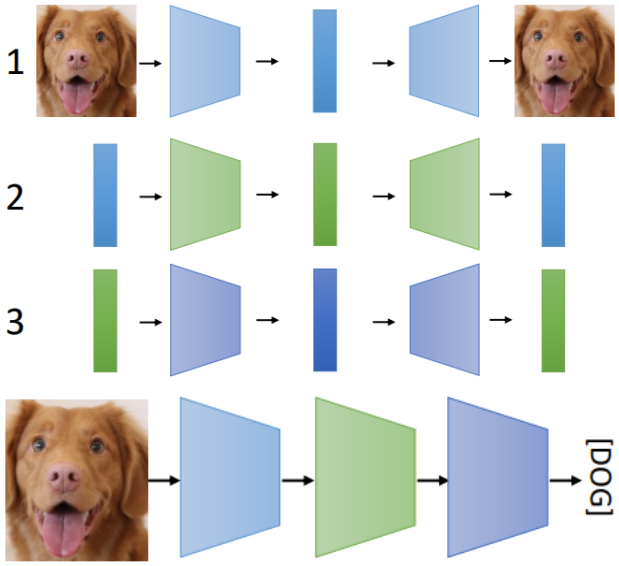
The idea behind denoising autoencoders is that a good model that has learned meaningful structure should fill in the blanks. There are many variants of this basic idea, and it is one of the most widely used simple autoencoder designs.

For instance, BERT is an example of a denoising autoencoder since it masks words.



8.2.5 Layerwise Pretraining

Historically (from 2006-2009), one of the dominant ways to train deep networks was to use layerwise pretraining. Here, we would train an autoencoder, get the hidden state representation, train another autoencoder on the hidden state, and so on. The idea was that each time we train the autoencoder, we make the representation a little more abstract, disentangled, and lossy. Then the combination of all these autoencoder encoders will encode the original image into a much more abstract representation. Then when we want to use these models to solve some downstream task, we compose the encoders all of these autoencoders to get a very deep network and then fine-tune end-to-end.



After 2009 or so, we got a lot better at training deep networks end-to-end due to ReLU, batchnorm, better hyperparameter tunings, weight initializations like Xavier, optimizers like ADAM, etc. As a result, autoencoders became less important.

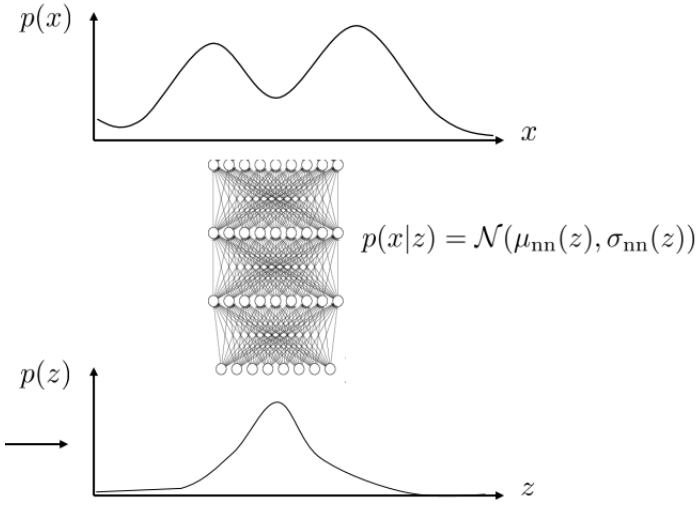
8.2.6 Autoencoders Today

Today, autoencoders can less widely used these days because there are better alternatives. For representation, VAEs and contrastive learning are used. For generation, GANs, VAEs, and autoregressive models are used. A big problem with autoencoders is that sampling (generation) from an autoencoder is hard, which limits its uses. The variational autoencoder (VAE) addresses this, which is discussed about later.

8.3 Latent Variable Models

8.3.1 Latent Variable Models

A latent variable deep generative model is usually a model that turns random numbers into valid samples (e.g. images). The idea is that we have some very complicated distribution $p(x)$ that we are trying to model. Instead of directly trying to model this distribution, we can try to model it as a composition of two very simple distributions $p(z)$ and $p(x|z)$. We can fix $p(z)$ to be a very simple distribution such as a zero-mean normal, and then we get $p(x|z)$ to be a normal distribution where the mean and variance is some neural network function of z. Then, $p(x) = \int p(x|z)p(z)dz$. We are essentially offloading most of the complexity into the mapping from z to the mean and variance of x given z, which is a complex deterministic function modeled by a neural network.



8.3.2 Estimating Log-Likelihood

We have our model $p_\theta(x)$ and data $\mathcal{D} = \{x_1, .., x_N\}$. We then do a maximum likelihood fit:

$$\theta \leftarrow \underset{\theta}{\text{argmax}} \frac{1}{N} \sum_i \log\, p_\theta(x_i)$$

, which is

$$\theta \leftarrow \underset{\theta}{\text{argmax}} \frac{1}{N} \sum_i \log \left(\int p_\theta(x_i|z) p(z) dz \right)$$

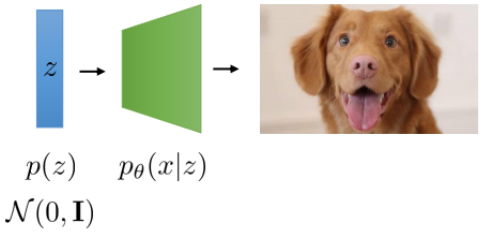
This objective is intractable, so we use the expected log-likelihood.

$$\theta \leftarrow \underset{\theta}{\text{argmax}} \frac{1}{N} \sum_i \mathbb{E}_{z \sim p(z|x_i)} [\log\, p_\theta(x_i, z)]$$

Since $p(z)$ can be fixed, we need to learn a $p(z|x_i)$ and a $p(x_i|z)$. The intuition for this optimization problem is that we "guess" the most likely z given x_i from $p(z|x_i)$ and pretend that it is the right one. Calculating $p(z|x_i)$ is called probabilistic inference.

8.3.3 Latent Variable Models in Deep Learning

We can choose something simple for z such as $\mathcal{N}(0, I)$. We can then use a neural network decoder to map z to x: $p_\theta(x|z)$. For generation, we first generate a vector of random ngumbers ($z \sim p(z)$) and then turn it into an image ($x \sim p(x|z)$).



How do we represent $p_\theta(x|z)$? Option 1 is to assume pixels are continuous-valued so $p_\theta(x|z) = \mathcal{N}(\mu_\theta(z); \sigma_\theta(z))$. An easy choice is to let σ be a constant, either a learned constant independent of z or chosen manually (e.g. $\sigma = 1$ reduces to an MSE loss). Option 2 is to know that the pixels are discrete-valued, so we could just use a 256-way softmax. This works well, but is slow. Other choices are discretized logistic or binary cross-entropy.

In terms of architecture choice for the decoder, we could just use a fully connected network, which works well for tiny images or non-image data. A better choice is to use transpose convolutions.

For training, we have three basic choices:

1. VAEs: Perform inference to figure out $p(z|x_i)$ for each training image x_i and minimize expected NLL $\mathbb{E}_{p(z|x_i)}[-\log p(x_i|z)]$
   (a) sample $z \sim p(z|x_i)$
   (b) reduce -log $p(x_i|z)$ with SGD
2. Normalizing Flows: Use an invertible mapping z to x
3. GANs: Match the distribution $\mathbb{E}_{z\sim p(z)}[p_\theta(x|z)]$

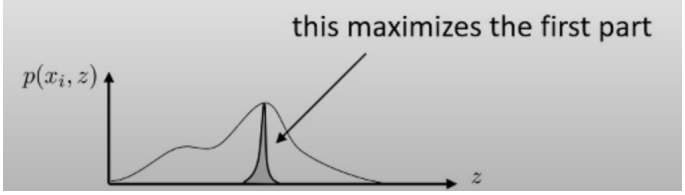
8.3.4 Variational Inference

8.3.4.1 Variational Approximation Let's approximate $p(z|x_i)$ by $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$. Note that the real $p(z|x_i)$ can be very complicated, so this can be a fairly crude approximation. For any choice of $q_i(z)$, we can construct a lower bound for $\log(p(x_i))$ as such

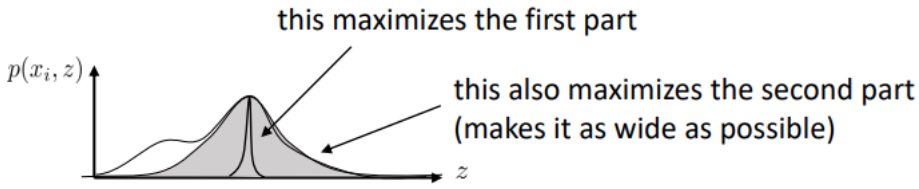
$$\begin{aligned}
\log(p(x_i)) &= \log \int_z p(x_i|z)p(z)dz \\
&= \log \int_z p(x_i|z)p(z)\frac{q_i(z)}{q_i(z)}dz \\
&= \log \mathbb{E}_{z\sim q_i(z)}\left[\frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&\geq \mathbb{E}_{z\sim q_i(z)}\left[\log \frac{p(x_i|z)p(z)}{q_i(z)}\right] \text{ (by Jensen's Inequality)} \\
&= \mathbb{E}_{z\sim q_i(z)}[\log p(x_i|z) + \log p(z)] - \mathbb{E}_{z\sim q_i(z)}[q_i(z)] \\
&= \mathbb{E}_{z\sim q_i(z)}[\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i)
\end{aligned}$$

Here $\mathcal{H}$ is entropy. Before we go on, we will first introduce the ideas of entropy and KL-divergence.

8.3.4.2 Entropy $\mathcal{H}(p) = -\mathbb{E}_{x\sim p(x)}[\log p(x)] = -\int_x p(x)\log p(x)dx$. Intuitively, entropy measures how random a random variable is, or how large the log probability in expectation under itself is. For instance, a higher variance (i.e. wide) normal distribution will have a higher entropy than a lower variance (i.e. narrow) normal distribution. When we look at maximizing $\mathbb{E}_{z\sim q_i(z)}[\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i)$, if we only maximize the first part, we will find a $q_i(z)$ that is narrowly peaked at the maximum point probability of $p(x_i, z)$ like below



By also maximizing the second part, we also make the $q_i(z)$ distribution as wide as possible like below



8.3.4.3 KL-Divergence

$$\begin{aligned} D_{KL}(q\|p) &= \mathbb{E}_{x\sim q(x)}\left[\log \frac{q(x)}{p(x)}\right] \\ &= \mathbb{E}_{x\sim q(x)}[\log\, q(x)] - \mathbb{E}_{x\sim q(x)}[\log\, p(x)] \\ &= -\mathbb{E}_{x\sim q(x)}[\log\, p(x)] - \mathcal{H}(q) \end{aligned}$$

Intuitively KL-Divergence measures how different two distributions are, or how small the expected log probability of one distribution is under another, minus entropy. Notice that KL-Divergence is always non-negative, and it is zero when p and q are the same distribution. If we view KL-divergence as the last expression above, if we were to minimize only the first part of KL-divergence (without entropy), we end up getting a $q(x)$ that is narrow and peaked at the maximum probability point of p. When we include the entropy term in the minimization problem, we see that we are also additionally forcing the $q(x)$ distribution to be as wide as possible.

8.3.4.4 Variational Approximation Continued

We currently have

$$\log(p(x_i)) \geq \mathcal{L}_i(p, q_i) = \mathbb{E}_{z\sim q_i(z)}[\log\, p(x_i|z) + \log\, p(z)] + \mathcal{H}(q_i)$$

We call $\mathcal{L}_i(p, q_i)$ the evidence lower bound (ELBO). A good $q_i(z)$ is one that makes the right hand side of the inequality as close to the left hand side as possible. Intuitively, this occurs if $q_i(z)$ approximates $p(z|x_i)$ well (i.e.

$D_{KL}(q_i(z)\|p(z|x))$ is low). We can show this rigorously by seeing that

$$\begin{aligned}
D_{KL}(q_i(z)\|p(z|x_i)) &= \mathbb{E}_{z\sim q_i(z)}\left[\log \frac{q_i(z)}{p(z|x_i)}\right] \\
&= \mathbb{E}_{z\sim q_i(z)}\left[\log \frac{q_i(z)p(x_i)}{p(x_i,z)}\right] \\
&= -\mathbb{E}_{z\sim q_i(z)}[\log p(x_i|z) + \log p(z)] + \mathbb{E}_{z\sim q_i(z)}[\log q_i(z)] + \mathbb{E}_{z\sim q_i(z)}[\log p(x_i)] \\
&= -\mathbb{E}_{z\sim q_i(z)}[\log p(x_i|z) + \log p(z)] - \mathcal{H}(q_i) + \log p(x_i) \\
&= -\mathcal{L}_i(p,q_i) + \log p(x_i) \\
\log p(x_i) &= D_{KL}(q_i(z)\|p(z|x_i)) + \mathcal{L}_i(p,q_i) \\
&\geq \mathcal{L}_i(p,q_i)
\end{aligned}$$

So the smaller the KL-Divergence is between $q_i(z)$ and $p(z|x_i)$, the better $\mathcal{L}_i(p,q_i)$ approximates $\log p(x_i)$. That means maximizing $\mathcal{L}_i(p,q_i)$ will push up on $p(x_i)$, and the smaller the KL-divergence is, the closer $\mathcal{L}_i(p,q_i)$ is to our objective. So maximizing $\mathcal{L}_i(p,q_i)$ w.r.t q_i minimizes KL-divergence, and maximizing $\mathcal{L}_i(p,q_i)$ w.r.t our model $p(x|z)$ trains our model.

8.3.4.5 Classic Variational Inference

Knowing this, we can now sketch our a classic variational inference algorithm:

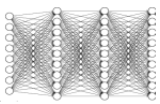
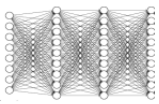
Algorithm 24 Classic Variational Inference

1: **for** each x_i (or mini-batch) **do**
2: calculate $\nabla_\theta \mathcal{L}_i(p,q_i)$ by sampling $z \sim q_i(z)$ and calculating $\nabla_\theta \mathcal{L}_i(p,q_i) \approx \nabla_\theta \log p_\theta(x_i|z)$
3: $\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}_i(p,q_i)$
4: update q_i to maximize $\mathcal{L}_i(p,q_i)$
5: **end for**

For updating q_i, a simple case would be to assume that $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$ so then we can perform gradient ascent on μ_i and σ_i by calculating $\nabla_{\mu_i}\mathcal{L}_i(p,q_i)$ and $\nabla_{\sigma_i}\mathcal{L}_i(p,q_i)$. However, the issue with this could be that we have $|\theta| + N \times (|\mu_i| + |\sigma_i|)$ parameters where N is the number of datapoints. This could be enormous.

8.3.5 Amortized Variational Inference

Remember that $q_i(z)$ should approximate $p(z|x_i)$ so we can instead learn a separate network $q_i(z) = q(z|x_i) \approx p(z|x_i)$.

z  $p_\theta(x|z)$ $\qquad$ x  $q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \sigma_\phi(x))$

In other words, amortized variational inference amortizes the challenges of inference over all the datapoints by using a neural network to perform inference for us. So now, we have

$$\log(p(x_i)) \geq \mathcal{L}(p_\theta(x_i|z), q_\phi(z|x_i)) = \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log\, p(x_i|z) + \log\, p(z)] + \mathcal{H}(q_\phi(z|x_i))$$

Algorithm 25 Amortized Variational Inference

1: **for** each x_i (or mini-batch) **do**
2: calculate $\nabla_\theta \mathcal{L}(p_\theta(x_i|z), q_\phi(z|x_i))$ by sampling $z \sim q_\phi(z|x_i)$ and calculating $\nabla_\theta \mathcal{L} \approx \nabla_\theta \log\, p_\theta(x_i|z)$
3: $\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}$
4: $\phi \leftarrow \phi + \alpha \nabla_\phi \mathcal{L}$
5: **end for**

Remember that $\mathcal{L}_i = \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log\, p(x_i|z) + \log\, p(z)] + \mathcal{H}(q_\phi(z|x_i))$. So how do we calculate $\nabla_\phi \mathcal{L}$?

8.3.5.1 Policy Gradient Approach L_i breaks down into the expected value term and the entropy term. The entropy of a Gaussian is closed-form and its gradient is closed form, so we have no problem there. For the expected value term, we can let $r(x_i, z) = \log\, p_\theta(x_i|z) + \log\, p(z)$, so the expected value can be rewritten as $J(\phi) = \mathbb{E}_{z\sim q_\phi(z|x_i)}[r(x_i, z)]$. Thus, we can just do policy gradient by computing $\nabla J(\phi) \approx \frac{1}{M} \sum_j \nabla_\phi \log\, q_\phi(z_j|x_i) r(x_i, z_j)$. The advantage to using policy gradients is that it can handle both discrete and continuous latent variables. However, it has an issue in that policy gradient is a high variance estimator, so it requires multiple samples and small learning rates.

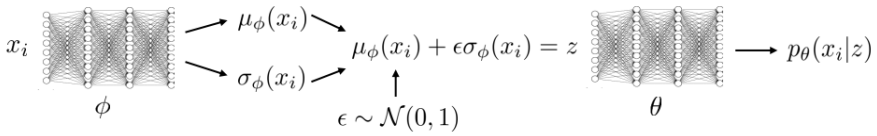
8.3.5.2 Reparameterization Trick Approach An alternative is instead to use the reparameterization trick. Notice that

$$\begin{aligned} J(\phi) &= \mathbb{E}_{z\sim q_\phi(z|x_i)}[r(x_i, z)] \\ &= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}[r(x_i, \mu_\phi(x_i) + \epsilon\sigma_\phi(x_i))] \end{aligned}$$

So to estimate $\nabla_\phi J(\phi)$, we can sample $\epsilon_1, ..., \epsilon_M$ from $\mathcal{N}(0, 1)$ (although a single sample also works well) and calculate $\nabla_\phi J(\phi) \approx \frac{1}{M} \sum_j \nabla_\phi r(x_i, \mu_\phi(x_i) + \epsilon_j \sigma_\phi(x_i))$. Another way to look at it is that

$$\begin{aligned} J(\phi) &= \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log\, p_\theta(x_i|z) + \log\, p(z)] + \mathcal{H}(q_\phi(z|x_i)) \\ &= \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log\, p_\theta(x_i|z)] + \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log\, p(z)] + \mathcal{H}(q_\phi(z|x_i)) \\ &= \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log\, p_\theta(x_i|z)] - D_{KL}(q_\phi(z|x_i)\|p(z)) \\ &= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}[\log\, p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i))] - D_{KL}(q_\phi(z|x_i)\|p(z)) \\ &\approx \log\, p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - D_{KL}(q_\phi(z|x_i)\|p(z)) \end{aligned}$$

The KL-divergence of Gaussians has a convenient analytical form, so taking the gradient of it is fine. Computing the gradient of the first log term is also just a matter of backpropagation. For visually, we can see our two networks as such

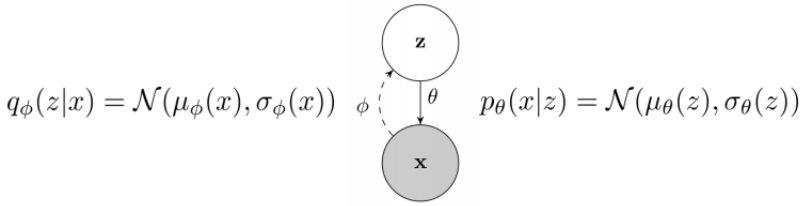


All we have to do now is do autodiff on our ELBO to update weights. The advantage to using the reparameterization trick is that it is very simple to implement and has low variance. However, it only works for continuous latent variables.

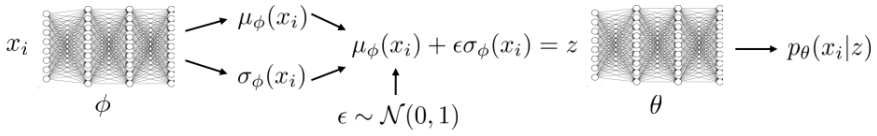
8.4 Variational Autoencoder (VAE)

8.4.1 VAE

A VAE is exactly the diagram above. Put another way we have our latent prior z and our input x. We have two networks: an encoder that performs inference and a decoder that produces x given z.



VAEs can do crazy things like generate new images of people's faces (i.e. those faces don't really exist). Our VAE has the following architecture that we've seen above



with the following objective function:

$$\max_{\theta,\phi} \sum_i \log p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - D_{KL}(q_\phi(z|x_i)\|p(z))$$

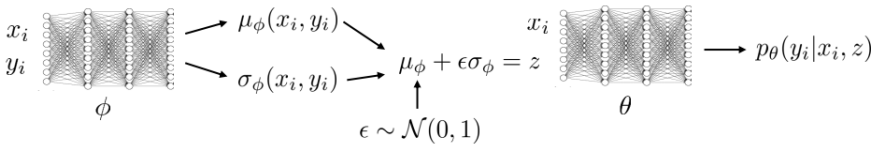
Notice that the first part of this objective looks like an autoencoder objective. The second part looks like a penalty on how much $q_\phi(z|x_i)$ deviates from our prior $p(z)$. Intuitively, this allows for at test time, when we sample from $p(z)$, the decoder will know what to do which these zs. In test time, we simply sample $z \sim p(z)$ and $x \sim p(x|z)$.

8.4.2 Conditional VAE

A conditional VAE is a VAE but we replace the x_i term with $y_i|x_i$ (i.e. we're conditioning on x):

$$\mathcal{L}_i = \mathbb{E}_{z\sim q_\phi(z|x_i,y_i)}[\log p(y_i|x_i,z) + \log p(z|x_i)] + \mathcal{H}(q_\phi(z|x_i,y_i))$$

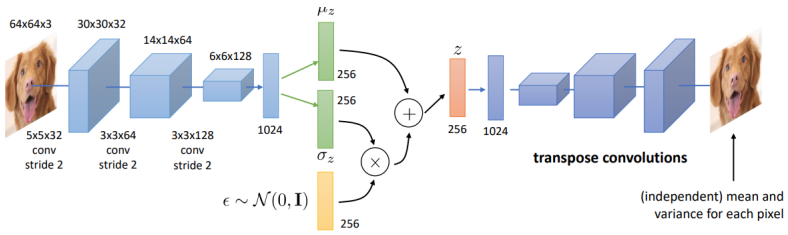
Our network now looks like this:



This is the same as a VAE network except we now pass in x_i into our θ network as well as our ϕ network if we choose to do so. At test time, we either sample $z \sim p(z)$ or $z \sim p(z|x_i)$ depending on our design choice. Then we sample $y \sim p(y|x_i,z)$.

8.4.3 VAEs with Convolutions

For images, simply need to a CNN encoder and FC network to produce the mean and variance of the latent. We then perform transpose convolutions to decode our latent and produce a mean and variance for each pixel of our output image.



8.4.4 β-VAE

Especially for conditional VAEs, our policy may be tempted to ignore the latent codes, or generate poor samples. Initially z just looks like noise, so it may be hard to figure out how to use it especially when part of our objective is to make z look more like our prior distribution. So we have two possible problems:

1. Our latent code is ignored (i.e. $p_\theta(x|z) \to p(x)$) and our reconstructed image will look like a blurry average image
2. Our latent code is not compressed (i.e. $q_\phi(z|x)$ is very far from $p(z)$) so our reconstructed image will be great in training, but upon sampling we will get garbage. This is because we have essentially learned the identity function in training.

For our first problem, $D_{KL}(q_\phi(z|x)\|p(z))$ is too low because $q_\phi(z|x) \approx p(z)$. For our second problem, our KL-divergence is too high because we have too much info in z and it has deviated from the prior, leading to us just learning an identity function. What this means we need to control this KL-Divergence carefully to get good results. So we add a multiplier to adjust regularizer strength:

$$\max_{\theta,\phi} \frac{1}{N} \sum_i \log\, p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - \beta D_{KL}(q_\phi(z|x_i)\|p(z))$$

In problem 1, we want to decrease β and in problem 2, we want to increase β. Oftentimes, we can adjust β manually to get good reconstructions and good samples. Sometimes, this is hard because earlier on in training, the VAE needs to learn to pay attention to z and later on in training, it needs to do a better job of compression so it can sample. So it can be useful to schedule β by starting with a low β to get the VAE to use z to reconstruct and later on, raise β so the samples look good.

8.5 Normalizing Flows

8.5.1 Invertible Mappings and Normalixing Flows

Instead of having $p(x|z) = \mathcal{N}(\mu_{nn}(z), \sigma_{nn}(z))$, what if we used a deterministic function instead: $x = f(z)$? Now we have

$$p(x) = p(z)\left|\det\, \left(\frac{df(z)}{dz}\right)\right|^{-1}$$

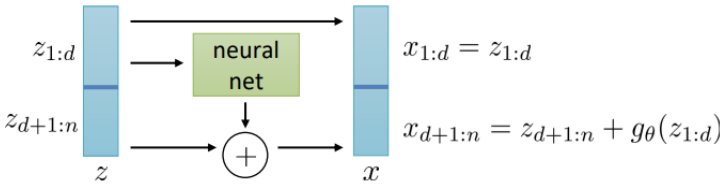
by the change of variables formula. If we can somehow learn invertible mappings from z to x, this makes the determinant easy to compute. There is no more need for lower bounds, and we can get exact probabilities/likelihoods. A normalizing flower model consists of multiple layers of invertible transformations. The training objective is still MLE:

$$\max_\theta \frac{1}{N} \sum_{i=1}^N \log\, p(x_i) = \max_\theta \frac{1}{N} \sum_{i=1}^N \log\, p(f^{-1}(x_i)) - \log\, \left|\det\, \left(\frac{df(z)}{dz}\right)\right|$$

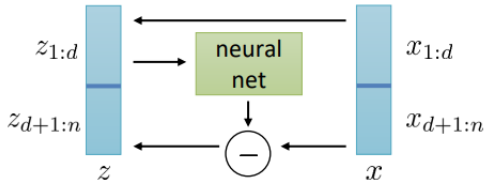
where $z = f^{-1}(x)$. Here, $f(z) = f_4(f_3(f_2(f_1(z))))$. In other words, $f(z)$ is a composition of many invertible functions. If each layer is invertible, the whole thing is invertible. Oftentimes, invertible layers also have very convenient determinants. The goal now is to design an invertible layer, and then compose many of them to create a fully invertible neural network.

8.5.2 Nonlinear Independent Components Estimation (NICE)

One idea if what if we force part of the layer to keep all the information so we can then recover anythings that was changed by the nonlinear transformation. We create the following architecture where g_θ is some neural network.



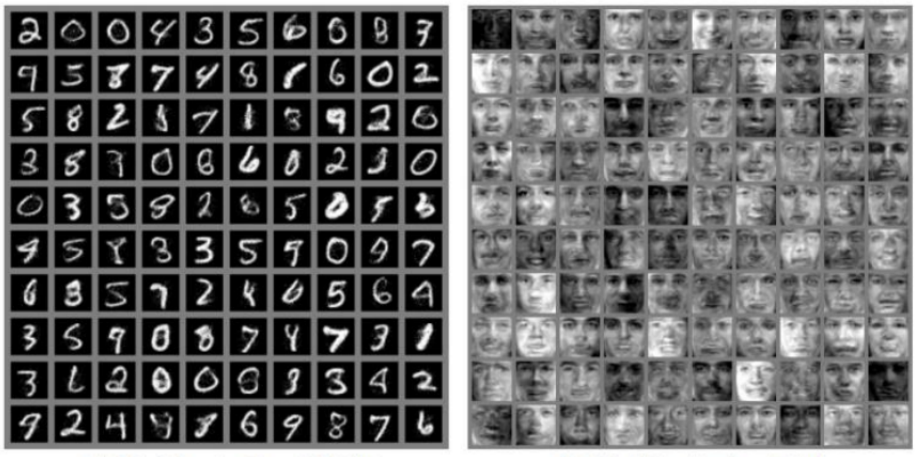
Notice that with this architecture, we can actually invert the layer as such



We recover $z_{1:d} = z_{1:d}$, recover $g_\theta(z_{1:d})$, and then recover $z_{d+1:n} = x_{d+1:n} - g_\theta(z_{1:d})$. The Jacobian of this layer is:

$$\frac{df(z)}{dz} = \begin{bmatrix} \frac{dx_{1:d}}{dz_{1:d}} & \frac{dx_{1:d}}{dz_{d+1:n}} \\ \frac{dx_{d+1:n}}{dz_{1:d}} & \frac{dx_{d+1:n}}{dz_{d+1:n}} \end{bmatrix} = \begin{bmatrix} I & 0 \\ \frac{dg_\theta}{dz_{1:d}} & I \end{bmatrix}$$

so then $\left|\det\left(\frac{df(z)}{dz}\right)\right| = 1$. This is really simple and convenient, but its representationally a bit limiting since we can't change the scale. NICE stacks multiple of these invertible layers and can get reasonable results on MNIST and faces, but on more complex images, the model starts to struggle.

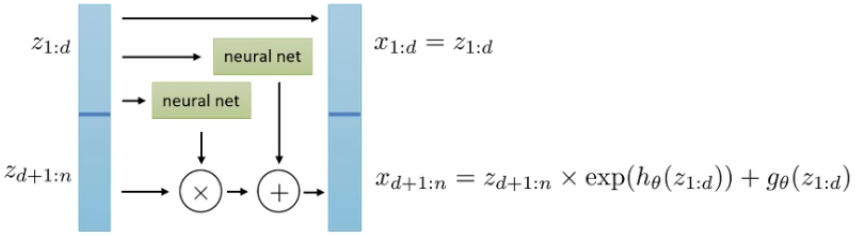


(a) Model trained on MNIST (b) Model trained on TFD

(c) Model trained on SVHN (d) Model trained on CIFAR-10

8.5.3 Non-Volume Preserving Transform (Real-NVP)

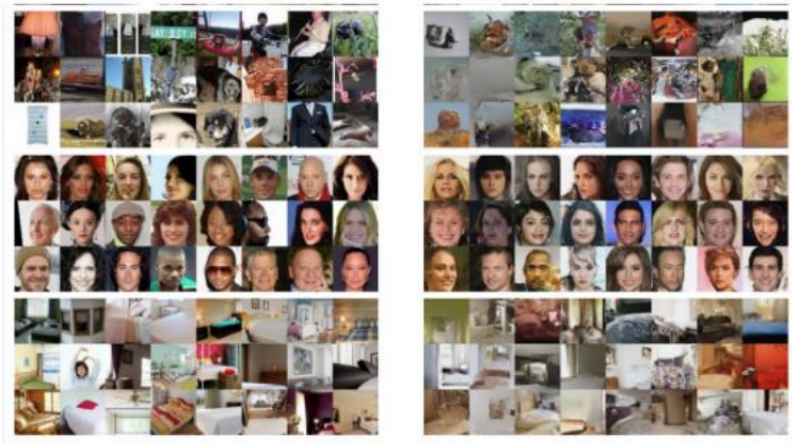
Real-NVP makes the invertible layer more expressive with a small change of adding a scaling factor.



The purpose of the exponential transform is just to make it positive. It's not enough to use ReLUs since we don't want zeros. The inverse is derived by recovering $z_{1:d} = x_{1:d}$, recovering $g_\theta(z_{1:d})$ and $h_\theta(z_{1:d})$, and then recovering $z_{d+1:n} = (x_{d+1:n} - g_\theta(z_{1:d}))/\exp(h_\theta(z_{1:d}))$. Now our Jacobian becomes

$$\frac{df(z)}{dz} = \begin{bmatrix} I & 0 \\ \frac{dg_\theta}{dz_{1:d}} & \text{diag}(\exp(h_\theta(z_{1:d}))) \end{bmatrix}$$

so we have $\left|\det\left(\frac{df(z)}{dz}\right)\right| = \prod_{i=d+1}^{n} \exp(h_\theta(z_{1:d})_i)$. This is a lot more expressive and leads to much better generated samples:



8.5.4 Concluding Remarks

An advantage of normalizing flows is that we can get exact probabilities and likelihoods. There is no need for lower bounds and conceptually simpler. However, it requires a special architecture. Z must also have the same dimensionality as X. This is a really big deal with high-resolution images.