CS182 Course Notes [Spring 2021]

Patrick Yin

Updated March 7, 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.

2 ML Basics

2.1 Risk

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

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

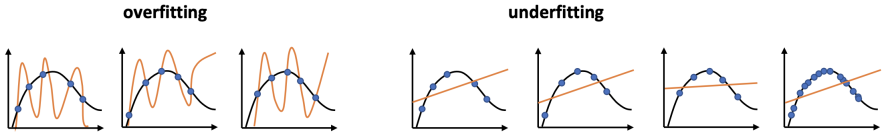
For instance, MSE is just risk with $\mathcal{L}(x, y, \theta) = (y - f_\theta(x))^2$. Since we don't know the true distribution in practice, we try to estimate this with empirical risk minimization.

Definition 2 (Empirical Risk)**.** The empirical risk is evaluated on samples from the true 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,

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

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

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

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: $\theta_{k+1} \leftarrow \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k}+\epsilon}$

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

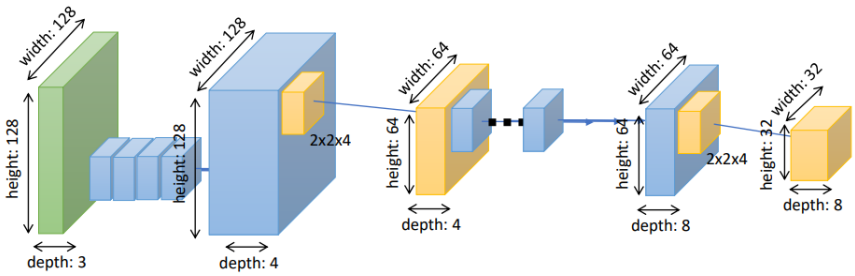
Making huge ensembles is pretty expensive, so can we make multiple models out of a single network? Dropout does it by randomly setting some activation to zero in the forward pass. Each activation has a p chance of not becoming 0 during the forward pass of training. At test time, we multiply our weights 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.

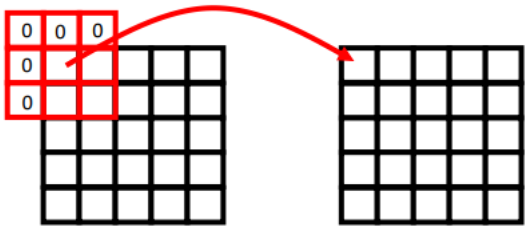


5.1.2 Convolutional Layer

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.

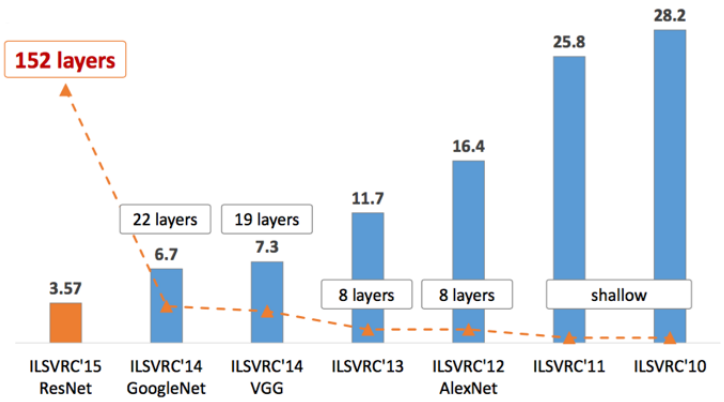
5.1.2.3 Putting it Together A standard conv net structure is to at each layer:

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), $H \times W \times C_{out} \to H/N \times W/n \times C_{out}$

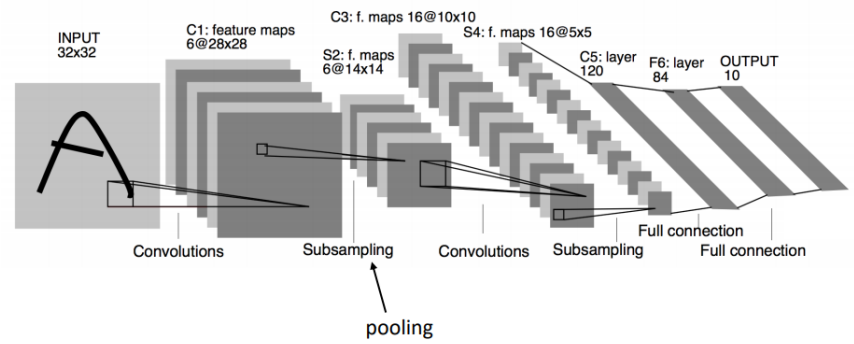
5.1.2.4 Strided Convolutions Applying a convolution can be computationally expensive. One idea is to skip over a few points. Strided convolutions skip over a few pixels every time a convolution happens. The amount of skipping is called a stride. Normally, our stride would be 1 because we don't skip over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

5.1.3 Examples of CNNs

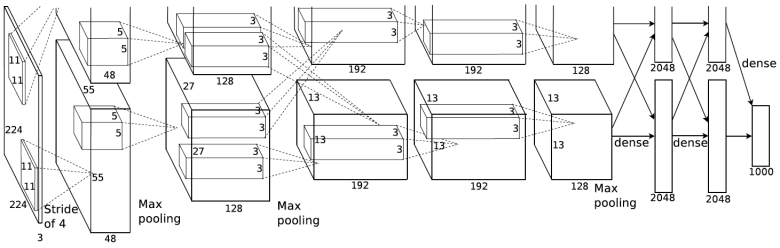
CNNs are progressed significantly over the last decade. We will cover the progress of CNNs over this decade as it has gotten better and better at classifying images on a image classificaiton bnechmark called ILSVRC (ImageNet), which contains 1.5 million images on 1000 categories. The error rate of these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



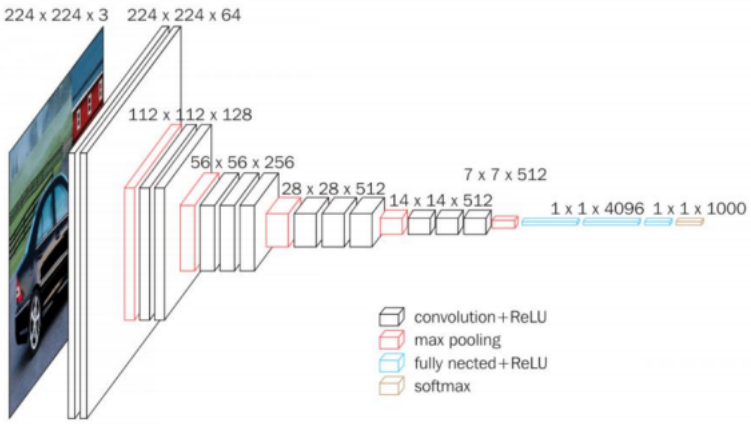
5.1.3.2 AlexNet AlexNet is the first real deep learning method to beat non-deep learning methods on ImageNet. The network looks a little strange because there are two different parts. This is because the model was trained on two GPUs so each section is trained on one GPU. Today, we don't really worry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

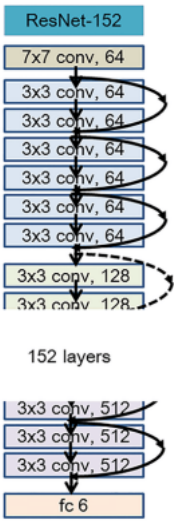
5.1.3.3 VGG VGG was able to get better accuracy by engineering much deeper networks. It follows a pattern of applying two convs and pooling, and then repeating.



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

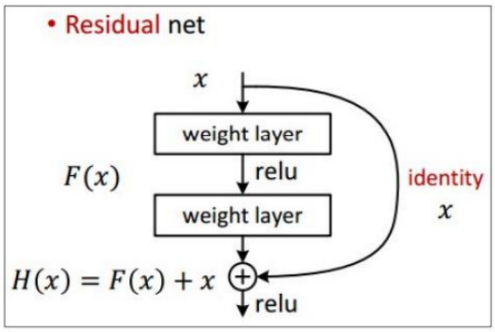
Most of the memory is used in the earlier layers due to the large image size. Most of the parameters are present in the first FC layer. This is not actually a good thing, and has been improved on in future networks.

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

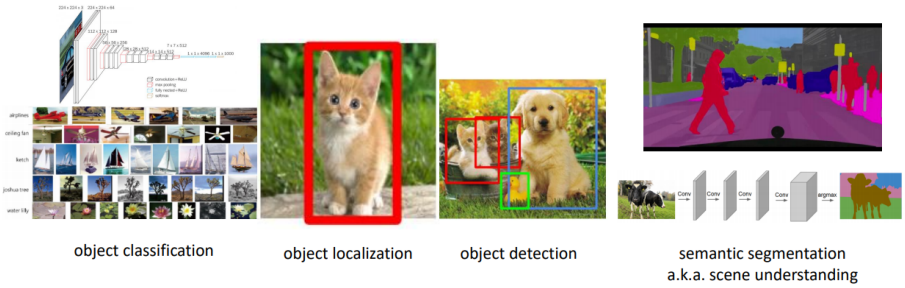
size, much like VGG. At the end, we average pool (like max pool but averaging instead) the convolutional response map and pass it directly through an FC layer into a softmax. For example, if our response map was $64 \times 64 \times 512$, then after average pooling it would just be a vector of length 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

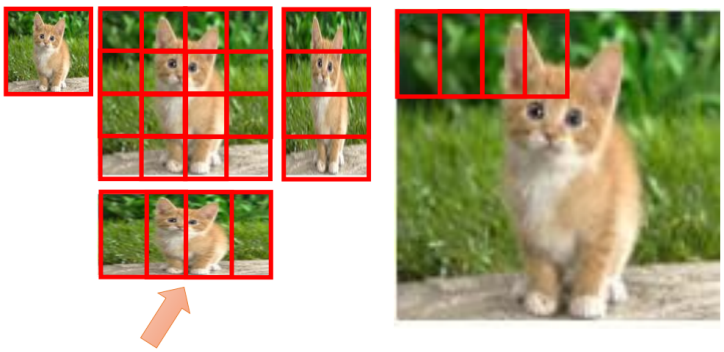
The four standard computer vision problems of increasing complexity is object classification, object localization, object detection, and semantic segmentation (a.k.a scene understanding). Object classification can be solved directly with the CNN architectures described above, so this section will go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



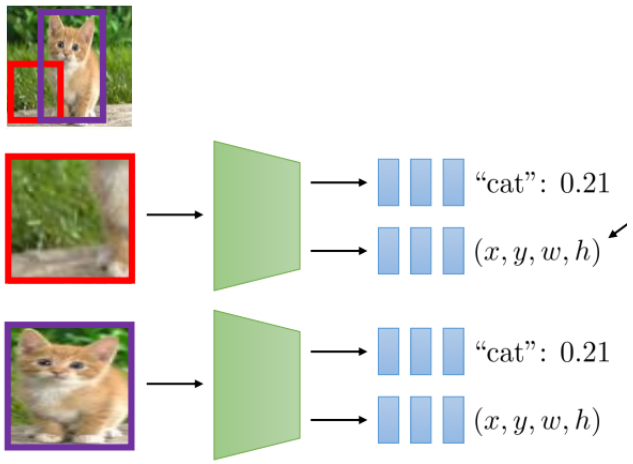
5.2.1 Object Localization

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.

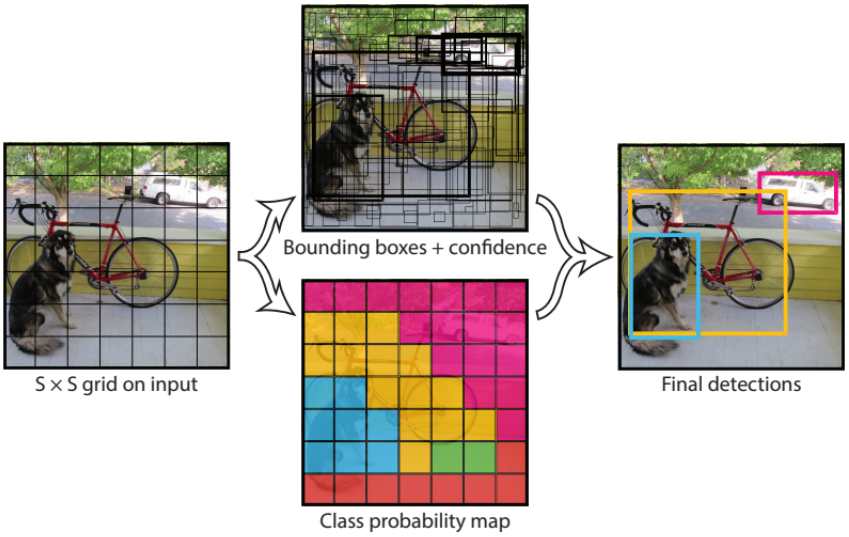


Doing sliding windows on the image can be very costly computationally. One idea to solve this is to use convolutional classification, where we slide over the convolutional layer rather than the image itself.

5.2.2 Object Detection

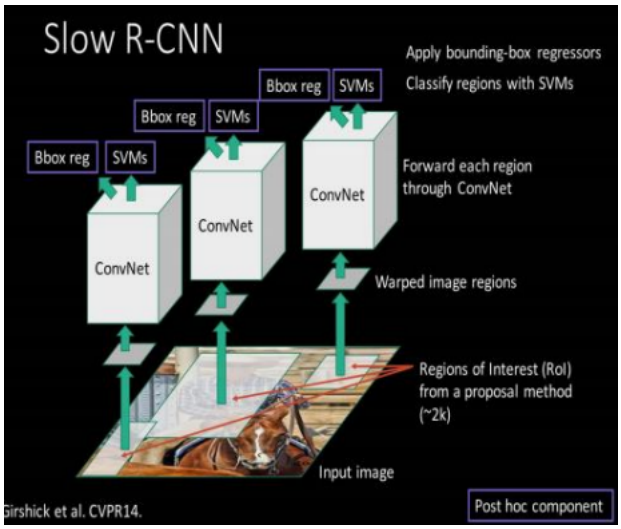
For object detection, naively we could just use the sliding window technique and just select the windows with highest probabilities. The issue with this is that high-scoring windows probably have other high-scoring windows nearby, so we can use non-maximal suppression, where we kill of any detections 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.



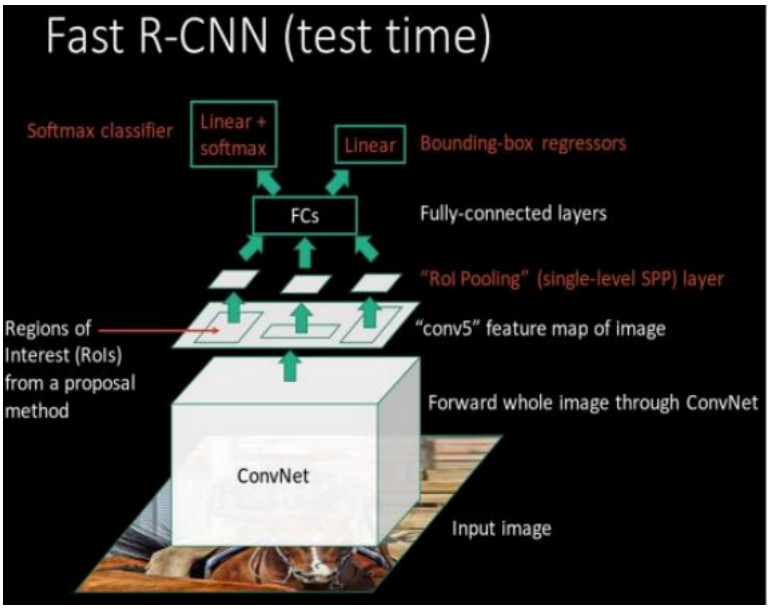
5.2.2.2 Region Proposals (R-CNN, Fast R-CNN, and Faster R-CNN)

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.

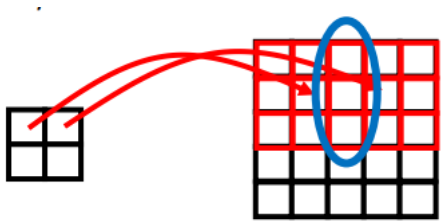


Currently R-CNN and Fast R-CNN uses selective search to find region proposals. This is slow, so instead we can train RoI proposals by taking the same convolutional response map, and at every position, predict if there is an object at that location and what its spatial extents are. In other words, this 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.

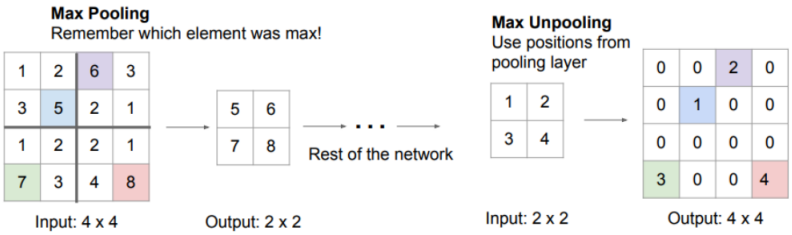
5.2.3 Semantic Segmentation

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.

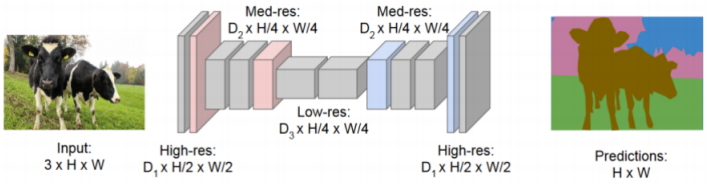
5.2.3.1 Transpose Convolution The transpose convolution is a convolution with a fractional stride. The overlapping region after multiplying by the filters can just be averaged.



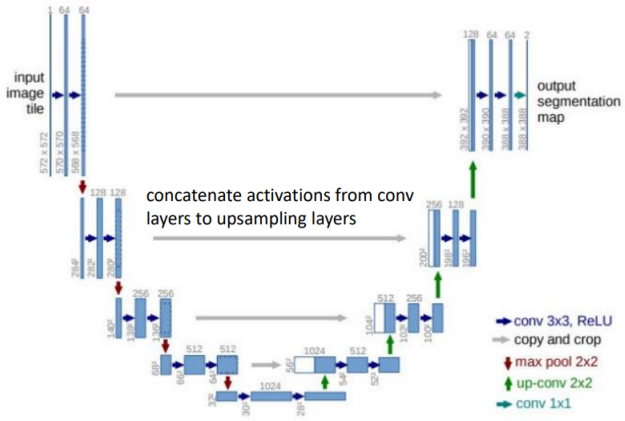
5.2.3.2 Un-pooling One trick for un-pooling is to, on the forward operation, save out which index had the max. Then later on in the network when we have the corresponding upsampling, we take that index and save the value in the low-resolution map to the corresponding index in the high-resolution map.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

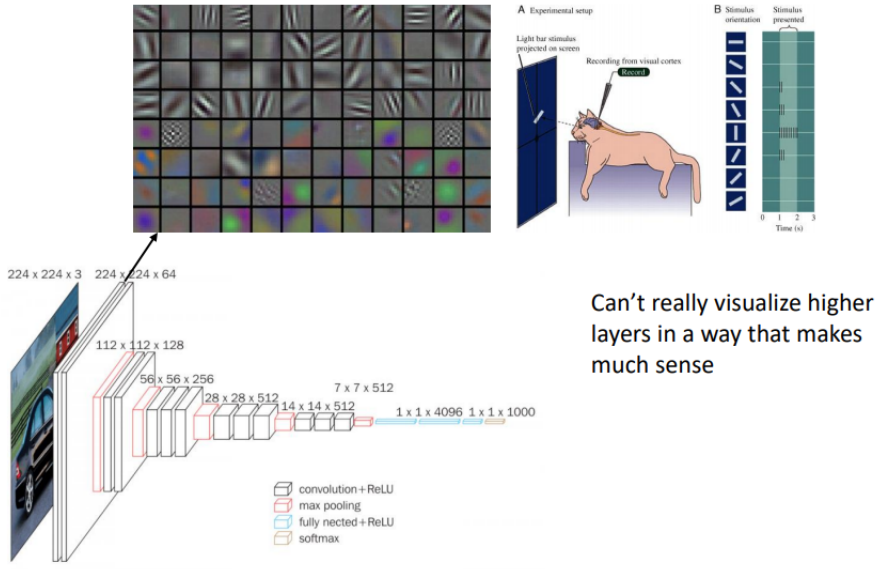


There are two convolutional layers, then a downsampling, and then repeated. Then when we start upsampling, we upsample the previous layer, take the layer from the original conv net that had the same resolution, and then we concatenate its activations.

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.

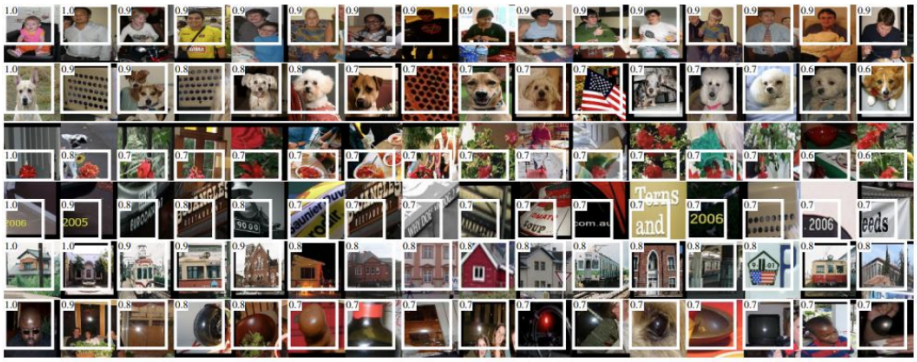


Figure 4: Top regions for six pool$_5$ units. Receptive fields and activation values are drawn in white. Some units are aligned to concepts, such as people (row 1) or text (4). Other units capture texture and material properties, such as dot arrays (2) and specular reflections (6).

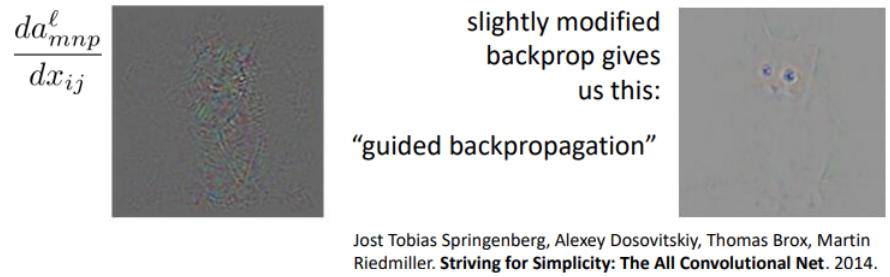
In the image above, in the bottom row, the filter seems to look for rounded shiny objects.

5.3.3 Using Gradients For Visualization

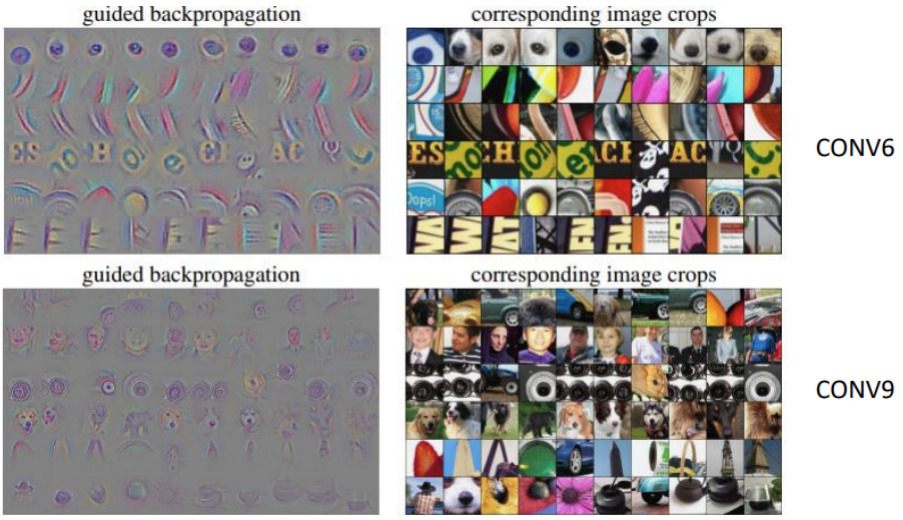
For any unit in the filter a^l_{mnp}, one way to see how much influence an image pixel has over that unit is that find $\frac{\mathrm{d}a^l_{mnp}}{\mathrm{d}x_{ij}}$. We can calculate it using backpropagation this way:

1. Set δ to $\frac{\mathrm{d}a^l_{mnp}}{\mathrm{d}a^l}$ (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 $\frac{\mathrm{d}a^l_{mnp}}{\mathrm{d}x}$.

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 $x \leftarrow \mathrm{argmax}_x \, S(x)$.

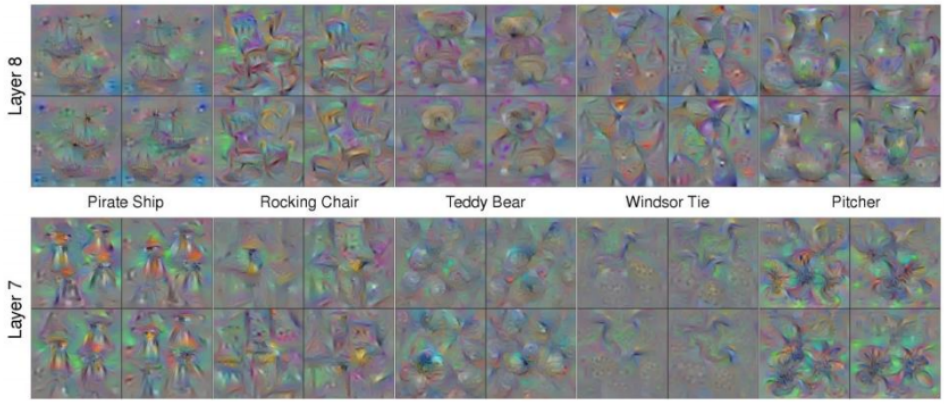
By itself, it is too easy to create crazy images without a regularizer. In the blue-eyed cat example, the blue channel will just go crazy to maximize the activations. So instead, our optimization problem because $x \leftarrow \text{argmax}_x S(x) + R(x)$. A simple choice is $R(x) = \lambda \|x\|_2^2$.

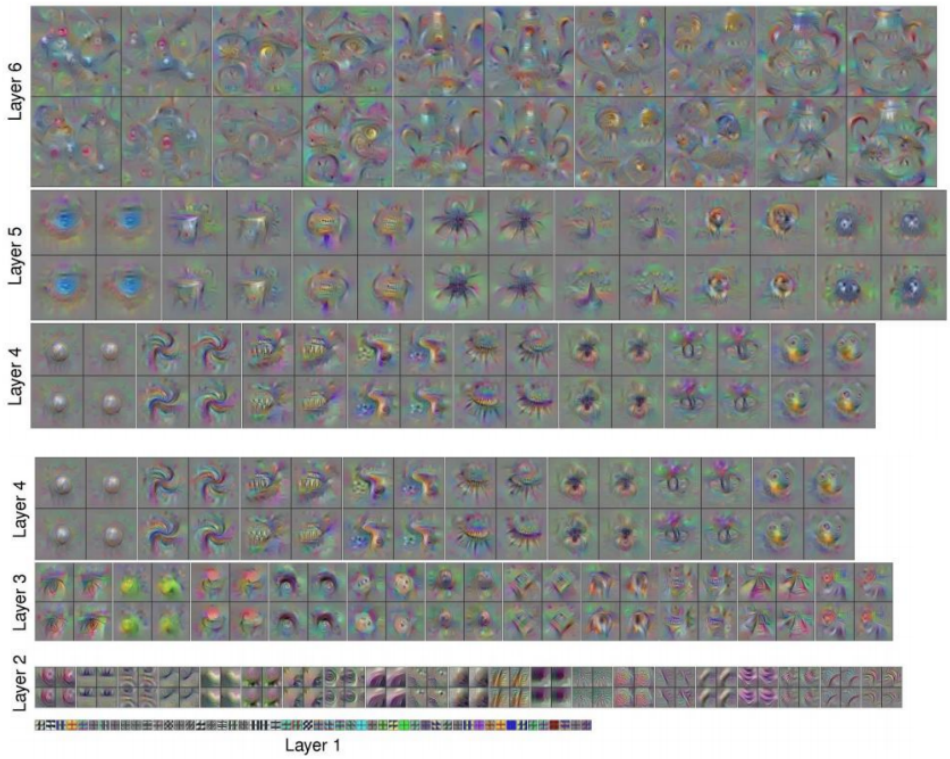
5.3.5 Visualizing Classes

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.



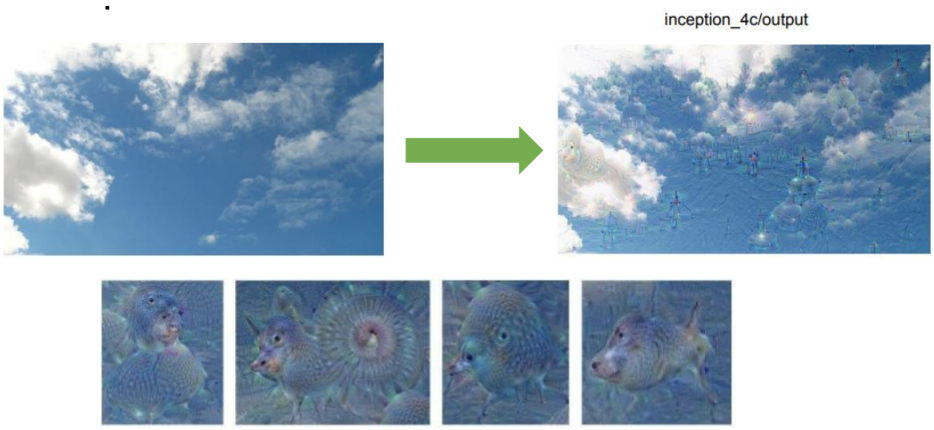


We can see that later layers visualize more complete outlines while earlier layers visualize more abstract components.

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:

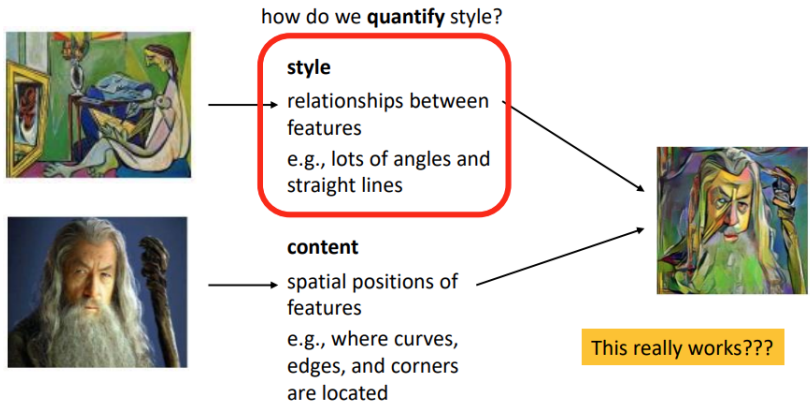
1. Pick a layer in conv net
2. Apply a little bit of jitter to the image
3. Run forward pass to compute activations of that layer
4. Set delta to be equal to the activations
5. Backprop and apply the gradient
6. Un-jitter the image
7. Repeat 2-6



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

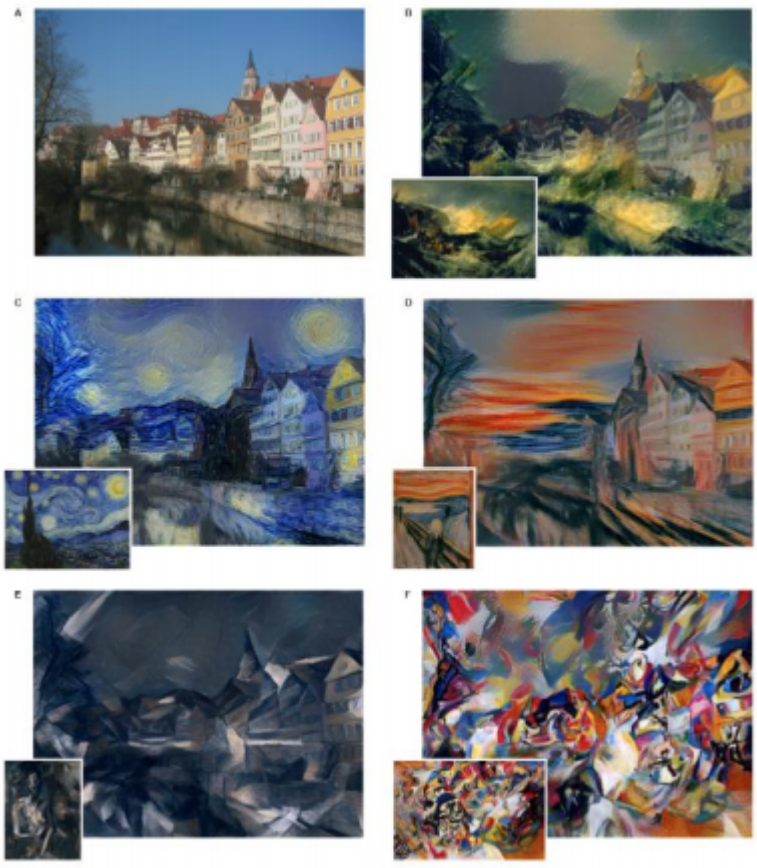
For content, we want to prevent the spatial positions instead of looking at relationships between features. So we can just directly match the features to the content image:

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

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 $x \leftarrow \text{argmin}_x \mathcal{L}_{style}(x) + \mathcal{L}_{content}(x)$



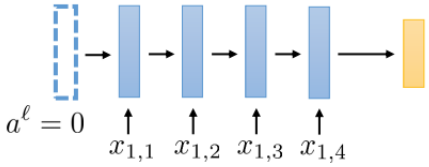
Using style transfer, we can above to transfer different styles to a base image. Very cool!

6 Natural Language Processing (NLP)

6.1 Recurrent Neural Networks (RNNs)

6.1.1 RNN Overview

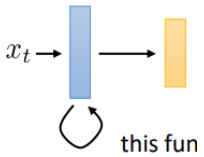
A standard neural network can't handle variable-length inputs (e.g. words in a sentence), motivating RNNs. RNNs are neural networks that share weights across multiple timesteps, take an input at each timestep, and have variable number of timesteps. In the graphic below, a^ℓ is the input "hidden" state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



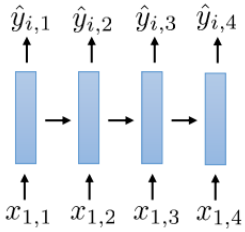
At each timestep of the RNN (i.e. each blue rectangle), we perform the following operation

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

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

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

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

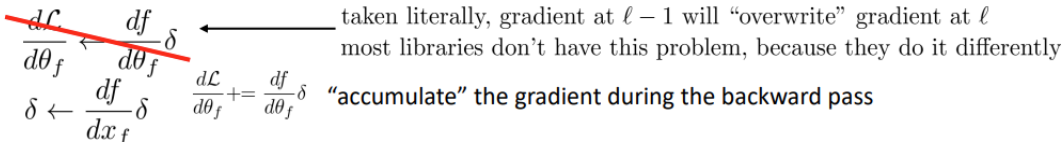
Here, $f(\cdot)$ is called a decoder. It could be something simple like a linear layer + softmax. For backpropogation, our new loss in this case would be

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

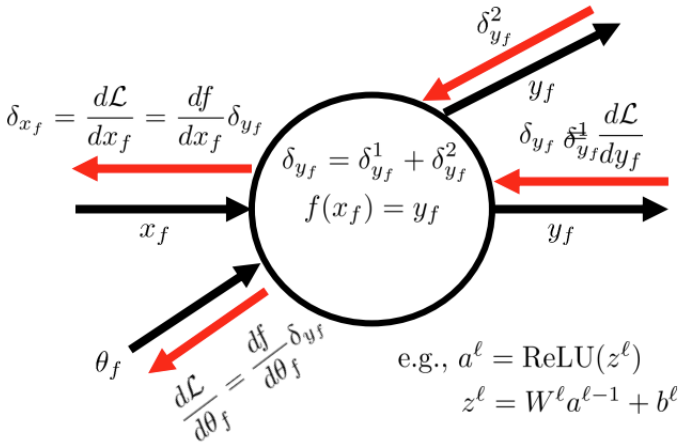
6.1.2 Backpropagation for RNNs

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.

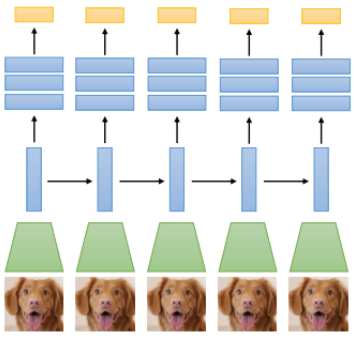


6.1.3 Issues with RNNs

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.

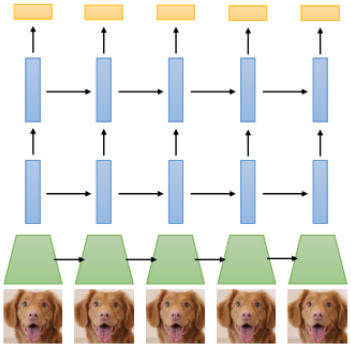
6.1.4 Small Extensions to RNNs

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.



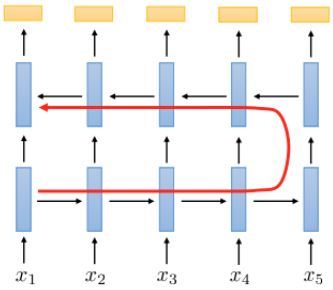
In this example, the images are "encoded" by a convolutional neural network (i.e. the "encoder") for the RNN input. The outputted hidden states, as we've seen before, as "decoded" by a fully connected neural network (i.e. the "decoder") to get the output we want.

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:

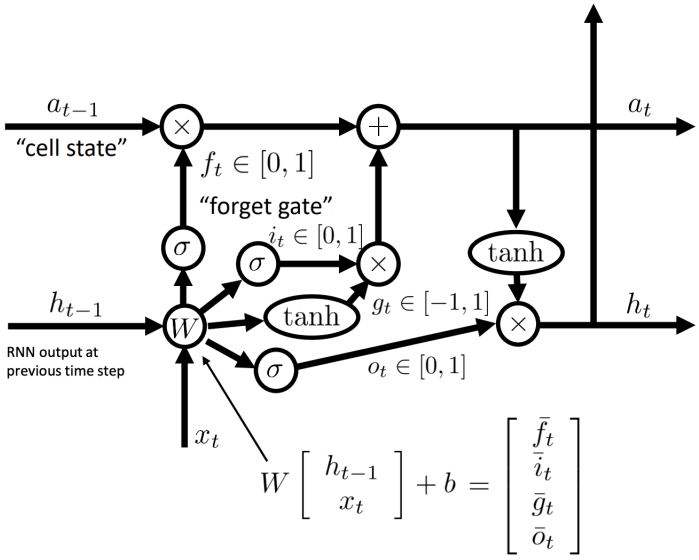


The motivation behind this is that for some tasks, we might need to look at inputs in the future to determine a prediction in the past. For instance, in speech recognition, we may want to know the entire sentence to give context for the meaning a word in the middle of the sentence.

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

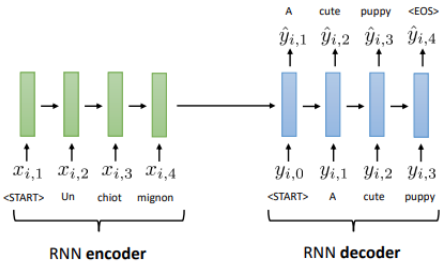
Autoregressive models run into an issue where the network only always sees true sequences as inputs during training, but a test-time it gets as input its own (potentially incorrect) predictions. This is called distributional shift, because the input distribution shifts from true string in training to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

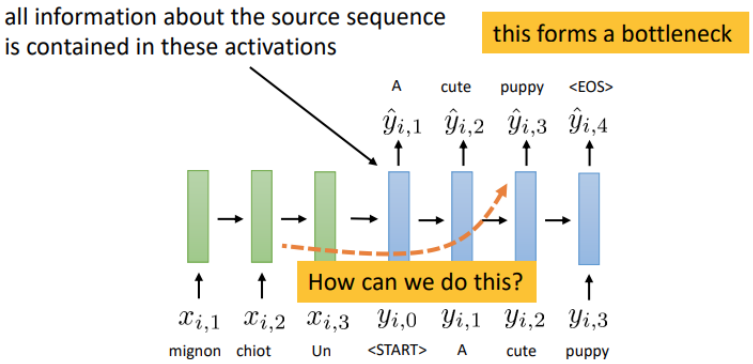
$$score(y_{1:t}|x_{1:T}) = \frac{1}{T}\sum_{t=1}^{T} \log p(y_t|x_{1:T}, y_{0:t-1})$$

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

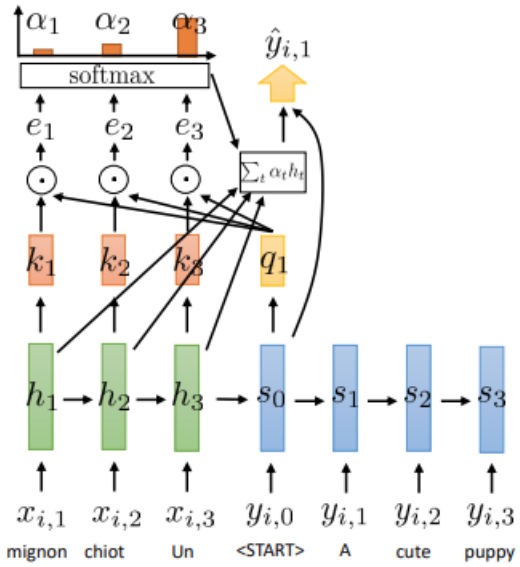
6.5.1 Motivation for Attention

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:

Encoder side:

$$k_t = k(h_t)$$

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.

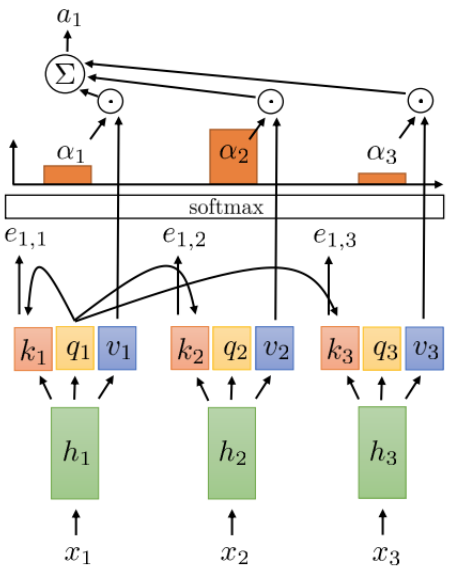
6.5.4 Why Attention Works

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:

$$k_t = k(h_t)$$

$$v_t = v(h_t)$$

$$q_t = q(s_t)$$

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

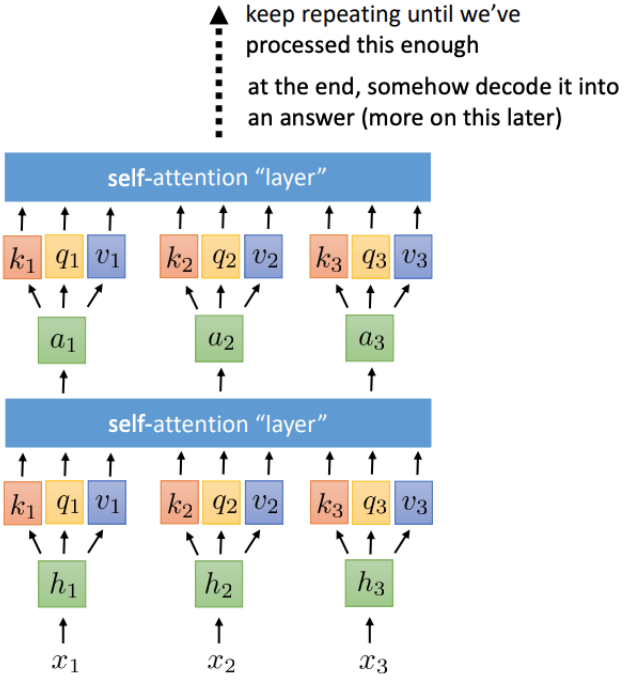
$$\alpha_{l,t} = \frac{\exp(e_{l,t})}{\sum_{t'} \exp(e_{l,t'})}$$

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

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

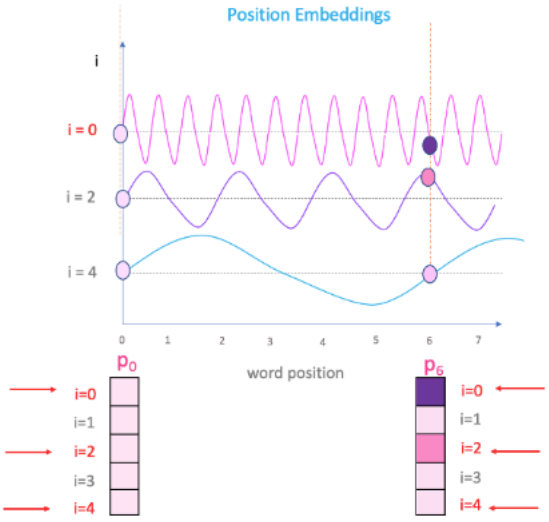
6.7.2.1 Positional Encoding First, we'll try to add in positional encodings. The issue we have right now is that for self-attention, our sequence might as well be a bag of words since we have lost the recurrent connections between timesteps. However, positions of words in sentences carries information. 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

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

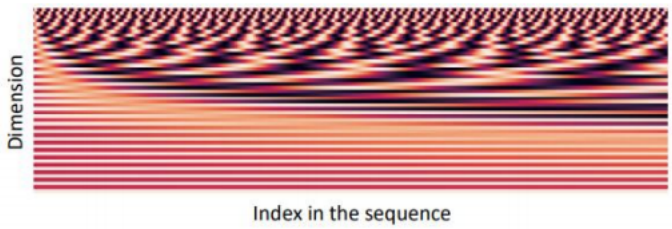
Here, d is the dimensionality positional encoding, and 10000 is an arbitrary choice. In the graphic below, the x axis is t and we are varying i to see how the values of the positional encoding changes.

"i"

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

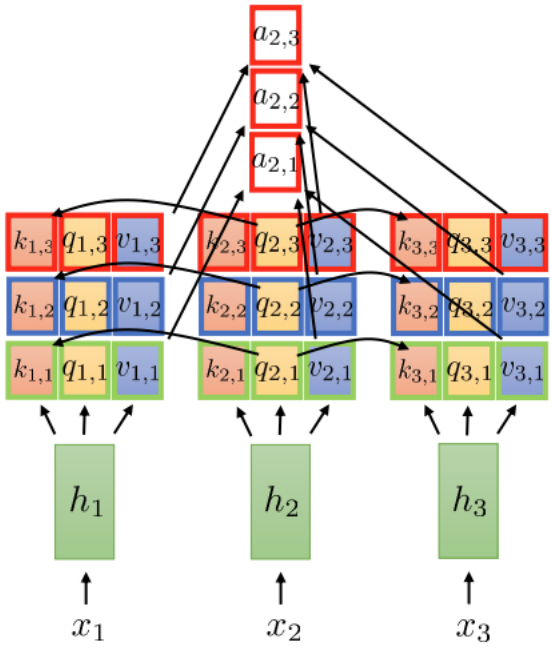


By ranging between different frequencies, we can encode the relative position of t. In the chart below, we see that the first dimension does something like an even-odd indicator of t, and the steps start to take longer as we go down the positional encoding vector.



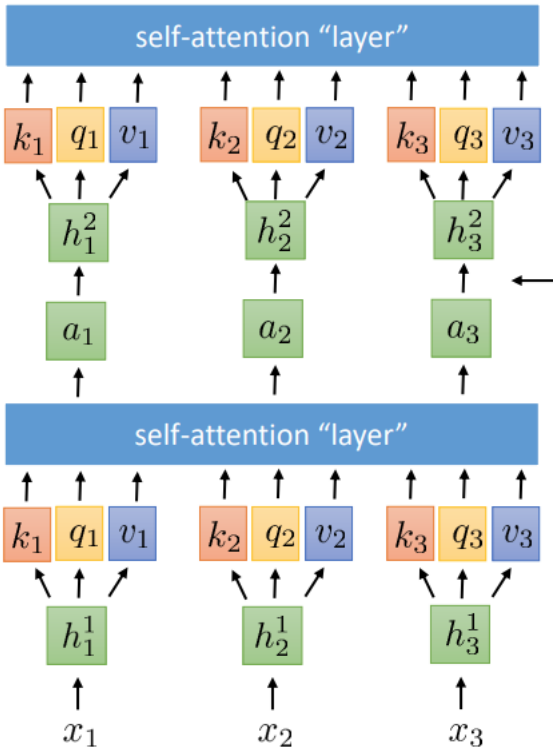
One last method to do positional encoding is to learn it. We could learn a weight vector $P = [p_1, p_2, ..., p_T] \in \mathbb{R}^{d \times T}$ where T is the max sequence length and d is dimensionality of the positional encoding. Every sequence would have these same learned valued, but the learned values 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:

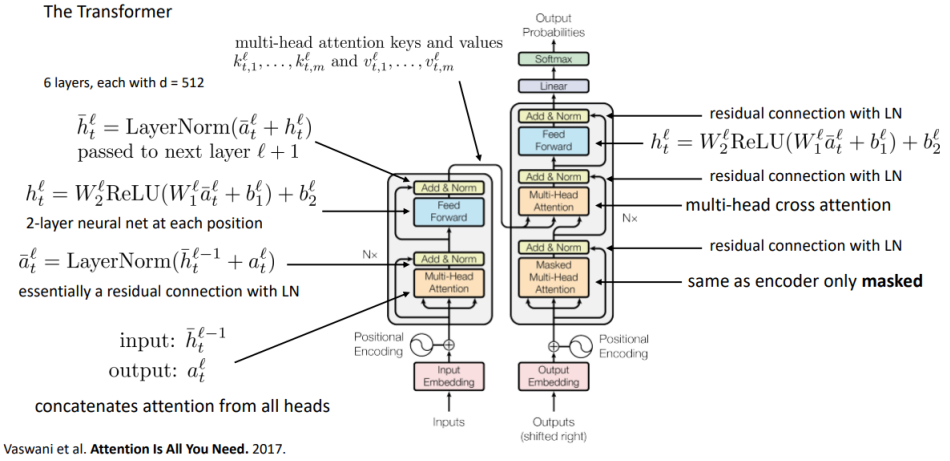
$$\mu = \frac{1}{d}\sum_{i=1}^{d} a_i$$

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

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

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.