Machine Learning Basics

Saket Choudhary

October 21, 2018

This report summarizes various techniques that are part of almost all learning algorithms, with a focus on neural networks.

1 Gradient Descent

Given a dataset $\mathbf{X}$ we want to develop a model $g(\theta)$ that 'best' explains the data. In order to gauge how 'well' this model is doing we use a cost function, $\mathcal{C}(\mathbf{X}, g(\theta))$. Since we want $g(\theta)$ to be as close to $\mathbf{X}$, we would like to minimize this cost function. In principle, we are looking for an estimate $\hat{\theta}$ such that $\hat{\theta} = \text{argmin}_\theta \, \mathcal{C}(\mathbb{X}, g(\theta))$

Basic Idea: In order for the parameter θ to move towards a local minima (where $\mathcal{C}(\mathbb{X}, g(\theta))$ is minimized), we should be able to iteratively adjust it such that it moves in the direction where the gradient of this cost function is large and negative , to ensure we move downhill the 'valley'.

Smarter Way of iterating: In iterating over values of θ to arrive at the optimum θ_{opt} where the error is minimized. We can keep on moving at a fixed rate. But, a faster way to do this would be to move in large steps where the curvature of the cost function is relatively flat and in slower steps when the curvature is relatively steep.

More formally, the function to be minimized is given by $f(\theta) = \mathcal{C}(\mathbf{X}, g(\theta))$. If $f_i(x_i, \theta)$ represents the error for the i^{th} data point.

$$f(\theta) = \sum_{i=1}^{n} f_i(x_i, \theta)$$

Examples: f_i will generally be mean-square loss $f_i = (x_i - \theta)^2$ for linear regression, and cross entropy loss $f_i = x_i \log(\theta_i) + (1 - x_i) \log(1 - \theta)$

1.1 Simplest Gradient Descent

Define η_t to be learning rate. The gradient descent recipe is then as follows:

- Step 1: Initialize θ to θ_0
- Step 2: $\nu_t = \eta_t \Delta_\theta f(\theta_t)$
- Step 3: $\theta_{t+1} = \theta_t - \nu_t$

1.2 Gradient Descent vs Newton-Raphson method

Newton-Raphson method uses first order taylor approximation to estimate the value at $f(\theta + \nu)$. Let $H(\theta)$ be the hessian matrix and ν be the step size:

$$f(\theta + \nu) \approx f(\theta) + \Delta_\theta f(\theta)\nu + \frac{1}{2}\nu^T H(\theta)\nu$$

For an optimum step size ν_{opt}, $\Delta_\theta f(\theta + \nu_{\text{opt}}) = 0$,

$$\begin{aligned} 0 &= \Delta_\theta f(\theta) + H(\theta)\nu_{\text{opt}} \\ \nu_{\text{opt}} &= -(H(\theta))^{-1}\Delta_\theta f(\theta) \end{aligned}$$

This rule can in turn be summarized as:

$$\begin{aligned} \nu_t &= (H(\theta))^{-1}\Delta_\theta f(\theta) \\ \theta_{t+1} &= \theta_t - \nu_t \end{aligned}$$

And hence drawing analogy with the Gradient descent rule, the learning rate for Newton's method is $\nu_t = (H(\theta))^{-1}$ which is adaptive. How?

For single dimension, this translates to $\nu_t = \left(\frac{\partial^2 f(\theta)}{\partial \theta^2}\right)^{-1}$ and hence at large curvature, is smaller and at small curvatures moves very fast.

While Newton's method is more adaptive, it is not useful:

1. Calculating hessian is expensive!
2. Even if we rely on Quasi-newton methods that do not calculate the Hessian explicittly and use an approximation, they are memory intensive since each hessian calculation will be required to store n^2 entries.

1.3 Limitations of vanilla Gradient Descent (GD)

1. GD finds *local* minima
2. It is sensitive to initial conditions
3. Gradient computations are often expensive
4. Very sensitive to learning rates
5. Treats all directions in parameter space uniformly and unlike Newton's method does not take large steps in flat directions or smaller steps in steeper directions
6. Can take exponential time to escape saddle points even with random initialization

1.4 Stochastic Gradient Descent (SGD)

SGD incorporates stochasticity by incorporating approximating the gradient on a subset of the whole data called a *minibatch*. Given n data points and a minibatch size of M, there will be n/M minibatches.

For $k = 1, 2 \dots n/M$, define $B_1, B_2, \dots, B_k$ to be minibatches and for the k^{th} minibatch, define the mini batch approximation to the gradient by cost function to:

$$\Delta_\theta f^{MB}(\theta) = \sum_{i \in B_k}^{M} \Delta_\theta f_i(x_i, \theta)$$

And the SGD algorithm is given by:

$$\begin{aligned} \nu_t &= \eta_t \Delta_\theta f^{MB}(\theta) \\ \theta_{t+1} &= \theta_t - \nu_t \end{aligned}$$

So what is stochastic about a SGD is the choice of dataset B_k for which the gradient is calculated. This is randomly chosen at **each** step.

1.5 SGD with Momentum

SGD is mostly used in conjunction with the momentum term:

$$\begin{aligned} \nu_t &= \gamma \nu_{t-1} + \eta_t \Delta_\theta f(\theta_t) \\ \theta_{t+1} &= \theta_t - \nu_t \end{aligned}$$

where $0 \leq \gamma \leq 1$. ν_t is a running average of its previous observation ν_{t-1} weighed by γ. Typical time scale of the "memory" where gradient is considered is given by $(1-\gamma)^{-1}$. Essentially, this past "memory" term helps the parameter move towards the optimum faster by making it move faster along the direction where gradients point in the same directions and prevents it from moving in directions where gradients are continuously changing directions, avoiding the typical zig-zag motion in case of SGDs.

1.5.1 Nesterov Accelerated Gradient (NAG)

$$\begin{aligned} \nu_t &= \gamma \nu_{t-1} + \eta_t \Delta_\theta f(\theta_t + \gamma \nu_{t-1}) \\ \theta_{t+1} &= \theta_t - \nu_t \end{aligned}$$

As pointed out earlier, we want to be able to move with larger step size along regions of small curvature and have smaller steps size for regions with relatively large curvature. NAG allows for larger learning rates, by asking the parameter to jump in the direction of previous gradient and then performing a correction. In momentum we perform a correction first and then make a jump in the direction of velocity.

1.5.2 RMSprop

RMSprop reduces the learning rate η_t in the direction where the norm of the gradient is consistently large, by additional keeping track of the second moment of the gradient $s_t = \mathbb{E}[g_t^2]$:

$$\begin{aligned} g_t &= \Delta_\theta f(\theta) \\ s_t &= \beta s_{t-1} + (1-\beta) g_t^2 \\ \theta_{t+1} &= \theta_t - \eta_t \frac{g_t}{\sqrt{s_t + \epsilon}} \end{aligned}$$

and ϵ is a small regularization constant.

1.5.3 ADAM

ADAM goes one step beyond by keeping track of the running average of both first and second moments of the gradient. Additionally, it also performs a bias correction step to account for the fact that the moments are being estimated from running averages, rather than through a closed form solution.

$$\begin{aligned} g_t &= \Delta_\theta f(\theta) \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ s_t &= \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1} \\ \hat{s}_t &= \frac{s_t}{1-\beta_2} \\ \theta_{t+1} &= \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{s_t} + \epsilon} \end{aligned}$$

2 Feedforward Neural Networks

The basic unit of a neural net is a neuron that takes a vector of inputs $\mathbf{x} = (x_1, x_2 \dots x_n)$ and produces a scalar output $a_i(x)$. Many such stacked units form a neural net. $a_i(x)$ is a non-linear transformation of the linear combination of the weights.

Universality theorem: A neural net with one hidden layer can approximate any continuous function. The proof is a bit non-trivial. But a visual proof appears in Chapter 4 of Neilsen's book.

The activation function $a_i(x)$ is often modeled as a non-linear sigmoid. $a_i(x) = \sigma(z_i)$ where $z_i = w_i x_i + b_i$.

2.1 Why non-linearity matters?

Why have a non-linear function as a choice of activation function at all? Why not just any function?

The purpose of the activation function is to introduce non-linearity into the network. If the activation is linear, the entire network will act like just one single perceptron with no gradients. A linear function of a linear function is linear.

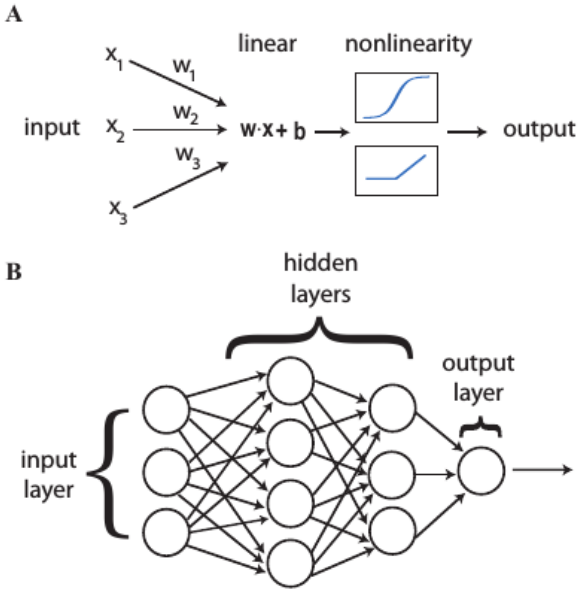


Figure 1: Neural Network Architecure. Figure borrowed from [1]

2.2 Basic Neural Networks

2.2.1 Types of activation

2.3 Backpropogation

2.4 Regularizing NNs

2.4.1 SGD and early stopping

SGD provides implicit regularization by providing stochasticity.

The idea of early stopping is straightforward: stop training when the validation error starts to increase. The validation error will tend to increase when the model begins to overfit.

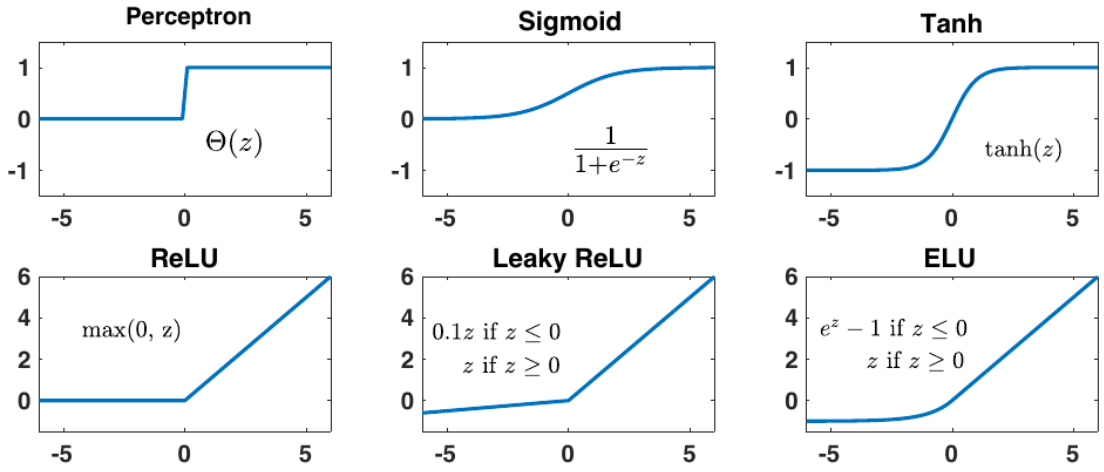
2.4.2 Dropout

Dropouts prevent overfitting by dropping off neurons randomly in order to reduce spurious correlations between neurons.

More specifically, for each minibatch in the gradient descent, a neuron is dropped with probability p, the gradient descent is now performed only on the remaining neurons. Thus, on average, the weights of each neuron are only present a fraction p of the time, so this needs to be accounted for in the testing stage too. Thus, the testing weights are rescaled $w_{test} = p\dot{w}_{train}$

2.4.3 Batch Normalization

Neural networks are easier to train when the inputs are centered around zero with respect to the bias, because this prevents neurons from saturating (?)



Batch-normalization ensures that the inputs remain bias centered over each minibatch.

2.5 Bias Variance tradeoff

Let y be generated from a noisy model:

$$y_i = x_i + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$

We approximate $f(x)$ using $\hat{f}(x)$.

$f(x_i)$ here is deterministic and hence the following two relations hold:

$$E[f] = f$$
$$Var[f] = 0$$

Our cost function is given by:

$$\mathcal{C}(x, \hat{f}(x)) = \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2$$

The expectation E below is taken over datasets D_j

$$\begin{aligned}
\mathbb{E}[\mathcal{C}(x, \hat{f}(x))] &= \mathbb{E}[\sum_{i=1}^{n}(y_i - f_i + f_i - \hat{f})^2] \\
&= \mathbb{E}[\sum_{i=1}^{n}(y_i - f_i)^2 + (f_i - \hat{f})^2 + 2(y_i - f_i)(f_i - \hat{f})] \\
&= \sum_{i=1}^{n}\mathbb{E}[(y_i - f_i)^2] + \sum_{i=1}^{n}\mathbb{E}[(f_i - \hat{f})^2] + 2\mathbb{E}[(y_i - f_i)(f_i - \hat{f})] \\
&= \sum_{i=1}^{n}\mathbb{E}[(y_i - f_i)^2] + \sum_{i=1}^{n}\mathbb{E}[(f_i - \hat{f})^2]
\end{aligned}$$

where the last equality holds because $\mathbb{E}[(y_i - f_i)(f_i - \hat{f})] = \mathbb{E}[(y_i - f_i)]\mathbb{E}[(f_i - \hat{f})] = 0$

Now,

$$\begin{aligned}
\mathbb{E}[(f_i - \hat{f}_i)^2] &= \mathbb{E}[(f_i - \mathbb{E}[\hat{f}_i] + \mathbb{E}[\hat{f}_i] - \hat{f}_i)^2] \\
&= \mathbb{E}[(f_i - \mathbb{E}[\hat{f}_i])^2] + \mathbb{E}[(\mathbb{E}[\hat{f}_i] - \hat{f}_i)^2] + 2\mathbb{E}[(f_i - \mathbb{E}[\hat{f}_i])(\mathbb{E}[\hat{f}_i] - \hat{f}_i))] \\
&= \mathbb{E}[(f_i - \mathbb{E}[\hat{f}_i])^2] + \mathbb{E}[(\mathbb{E}[\hat{f}_i] - \hat{f}_i)^2]
\end{aligned}$$

where the last equality again holds because the last term is zero since:

$$\begin{aligned}
\mathbb{E}[(f_i - \mathbb{E}[\hat{f}_i])(\mathbb{E}[\hat{f}_i] - \hat{f}_i))] &= \mathbb{E}[f_i - \mathbb{E}[\hat{f}_i]]\mathbb{E}[\mathbb{E}[\hat{f}_i] - \hat{f}_i))] \\
&= 0
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathbb{E}[\mathcal{C}(x, \hat{f}(x))] &= \sum_{i=1}^{n}\mathbb{E}[(y_i - f_i)^2] + \mathbb{E}[(f_i - \mathbb{E}[\hat{f}_i])^2] + \mathbb{E}[(\mathbb{E}[\hat{f}_i] - \hat{f}_i)^2] \\
&= \sum_{i=1}^{n}\sigma_\epsilon^2 + \text{Bias}^2 + \text{Variance}
\end{aligned}$$

where

$$\begin{aligned}
\text{Bias} &= \mathbb{E}[f - \mathbb{E}[\hat{f}]] \\
\text{Variance} &= \mathbb{E}[(\mathbb{E}[\hat{f}_i] - \hat{f}_i)^2]
\end{aligned}$$

3 Autoencoders

Autoencoders are a part of the "Generative modeling" paradigm in Machine learning research, where the aim is to model the distribution $P(X)$ given datapoints X in some high dimensional space $\mathbb{R}^m$. They can be thought of neural networks that essentially try to copy the input to output.

More formally, autoencoders consist of an encoding and a decoding layer. Given an input $\mathbf{x}$, the encoding layer represents the input into its coded form $h = f(\mathbf{x})$ while the decoding layer provides a reconstruction of the original input $r = g(h) = g(f(x)) \approx x$ such that a loss $L(g(f(x)), x)$ is minimized, where L is a loss function that penalizes $g(f(x))$ for being dissimilar to x. Autoencoders with dimensions less than the original dimension of input $\mathbf{x}$ are called undercomplete autoencoders.

3.1 PCA

When the decoding function is linear and the loss is mean squared loss, the undercomplete autoencoders is known to span the same subspace as PCA.

3.2 Sparse Autoencoders

Sparse autoencoders are autoencoders with a sparsity penalty $\Omega(h)$ on the encoding layer $h = f(x)$.

$$L(\mathbf{x}, g(f(\mathbf{x})) + \Omega(h)$$

The sparsity constraint ensures that the autoencoder learns unique statistical properties of the data, rather than acting just as a copy-paste function.

3.3 A more intuitive explanation

The overall aim of autoencoders is to be able to model the true probability distribution $P_{true}(X)$ of a given dataset X. Since $P_{true}(X)$ is unknown, our aim is to learn an alternate distribution $P(X)$m such that $P(X)$ is "close" to $P_{true}(X)$. We want to avoid putting restrictions on the structure of data and we also want the solution to be tractable (in a computationally expensive context) even for large datasets.

We will focus on a special category of autoencoders called the "Variational Autoencoders" (VAE) for the rest of our sections.

3.4 Sparse Autoencoders and Latent Variables

The sparsity constraint in Sparse Autoencoders can be understood as an approximation of the maximum likelihood training of a generative model that has latent variables.

3.4.1 Latent Variables

Consider a setting where our dataset includes objects like trees, house, and cat and dog pictures. We want to be able to generate images which looks the ones in our dataset, but not exactly the same. If we are able to generate the left half of a tree, then the right half of it will be very different from the right half of a dog. Before we generate any image, it is useful (and essential) to think about the kind of image we are going to generate to ensure that the pixels are coherent with the image we are going to generate.

Before our model generates these images, it will randomly pick up an object z from $\{\text{tree}, \text{house}, \text{cat}, \text{dog}\}$ and make sure that all pixels in that image are "coherent" in some sense: you do not expect green patches to appear on the right. This z is a latent variable. It is latent because given a new generated image, we do not know which settings of the latent variable generated the final image. z for example could be the patten of edges in our images.

3.4.2 Model

For the model to be representative of the data X, there should be at least on configuration of the latent variable z such that the data generated resembles all data points in X.

More formally, the distribution $P(X)$ is modeled using a generative process conditioned on the latent variable Z and we want to maximize $P(X)$. We have a way of sampling Z using some distribution $P(z)$ and this a deterministic function $f(z;\theta)$ parameterized by θ. We wish to optimize θ, such that $f(z;\theta)$ generates samples that look like X

We want to maximize $P(X)$ for each X in the training set, to be able to hope that this model will be able to generate "similar" samples.

$$P(X) = \int P(z)P(X|z;\theta)dz$$

where $P(X|z;\theta) = f(z;\theta)$.

In VAEs, the choice of this function $P(X|z;\theta) = \mathcal{N}(X|f(z;\theta), \sigma^2 I)$, but this is not a hard requirement. If X is binary, $P(X|z;\theta)$ could be bernoulli. It can be any distribution other than a dirac-delta, as long as it is computable.

VAEs solve this problem of "generative modeling" by telling us what our choice of latent variable z should be and how to overcome the integral involved over all such values of z.

Before our model starts generating any images, it needs to make some decisions. It needs to first decide what image it will be generating, than decide the angles, stroke width etc. We want to avoid deciding such properties by hand and we also want to avoid describing any sort of dependencies this latent variable should capture (the house images for example, always has more edges). VAEs take a very unintuitve approach of dealing with this by imposing a gaussian distribution on $z \sim \mathcal{N}(0, \sigma^2 I)$. How does that work?

Any distrbution in d dimensions can be generated by taking a set of d gaussian random variables and passing them through some non-linear transformation. For example we can generate a 2D ring starting with a 2D multivariate gaussian using this transformation: $g(z) = z/\alpha + z/||z||$ where $z \sim N(0, \sigma^2 I)$ and α is a shrinkage parameter.

An example is given below:

3.5 Objective function

For most choices of z $P(X|z)$ will be close to zero. So VAEs tend to sample only those values of z which are more probably of generating X. So we are interested in a function $Q(z|X)$ that takes the data X and gives us a distribution over z that are likely to produce X. Ideally, the space spanned by $Q(z|X)$ is much smaller than the "prior" distribution $P(z)$.

$$\begin{aligned}
KL(Q(z)||P(z|X)) &= E_{z\sim Q}[\log Q(z) - \log P(z|X)] \\
P(z|X) &= \frac{P(X|z)P(z)}{P(X)} \\
KL(Q(z)||P(z|X)) &= E_{z\sim Q}[\log Q(z) + \log P(X) - \log P(X|z) - \log P(z)] \\
KL(Q(z)||P(z|X)) &= E_{z\sim Q}[\log Q(z) - \log P(z)] + \log P(X) - E_{z\sim Q}[\log P(X|z)] \\
KL(Q(z)||P(z|X)) &= KL(Q(z)||P(z)) + \log P(X) - E_{z\sim Q}[\log P(X|z)] \\
\log P(X) - KL(Q(z)||P(z|X)) &= E_{z\sim Q}[\log P(X|z)] - KL(Q(z)||P(z))
\end{aligned}$$

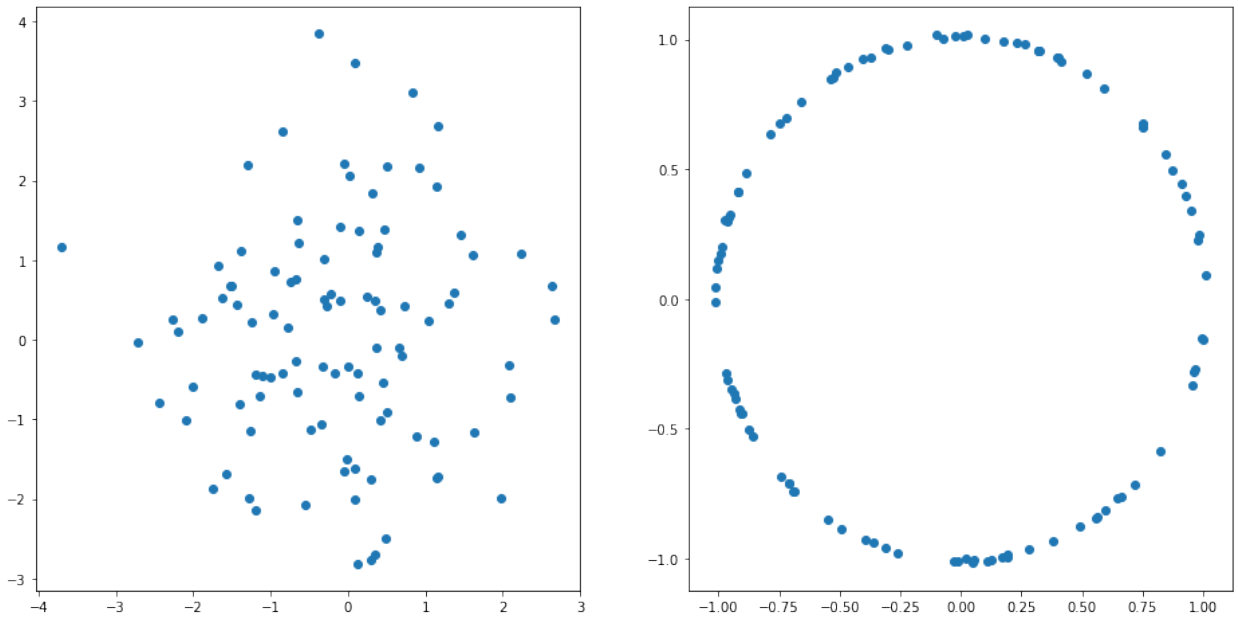


Figure 2: It is possible to transform a gaussian (left) to look like a ring (right)

We want to maximize the quantity on left hand side because we want to maximize $\log(P(x))$, however we alos want to penalize $Q(z)$ if it does not resemble the "true" unknown distribution $P(z|X)$ Alternatively, our loss function for performing gradient descent is given by:

$$KL(Q(z)||P(z)) - E[\log P(X|z)]$$

where the first term denotes the information lost in representing $P(z)$ as $Q(z)$ and the second term denotes the reconstruction loss

So we want $KL(Q(z)||P(z|X))$ to be minimized. If we find a magic function $Q(z)$ which can approximate $P(z|X)$ perfectly, $KL(Q(z)||P(z|X)) = 0$.

3.6 What should be ideal Q(z—X)?

Answer: $Q(z|X) = \mathcal{N}(z|X, \sum)$ just works.

To perform gradient descent on the right hand side, we take one sample of z and treat one sample of that as an approximation of $E[\log P(X|z)]$, this is performed over all datapoints, say D

$$E_{X\sim D}[\log P(X) - KL(Q(z)||P(z|X))] = E_{X\sim D}[E_{z\sim Q}[\log P(X|z)] - KL(Q(z)||P(z))]$$

We can just take the gradient of this equation and everything looks okay. However, during back-propogation, we will encounter a layer that is sampling z from $Q(z|X)$ which is itself a stochastic unit. In order to be able to sample from $Q(z|X) \sim \mathcal{N}(\mu(X), \sum(X))$ we first sample $\epsilon \sim \mathcal{N}(0, I)$ and then compute $z = \mu(X) + \sum^{1/2}(X)\epsilon$. This is called the reparameterization trick.

3.7 KL divergence between two gaussians (Used above for calculating losses)

Assume $q \sim \mathcal{N}(\mu_0, \sigma_0^2)$ and $p \sim \mathcal{N}(\mu_1, \sigma_1^2)$:

$$
\begin{aligned}
KL(q||p) &= \int q(x)(\log(q(x) - \log(p(x))dx \\
q(x) &= (2\pi\sigma_0^2)^{-\frac{1}{2}} \exp \frac{-(x-\mu)^2}{2\sigma_0^2} \\
\log q(x) &= -\frac{1}{2}\log 2\pi - \log(\sigma_0) - \frac{-(x-\mu)^2}{2\sigma_0^2} \\
KL(q||p) &= \int q(\log \frac{\sigma_1}{\sigma_0} + \frac{(x-\mu_1)^2}{2\sigma_1^2} - \frac{(x-\mu_0)^2}{2\sigma_0^2})dx \\
\int q\frac{(x-\mu_0)^2}{2\sigma_0^2}dx &= \frac{1}{2\sigma_0^2} \\
\int q\frac{(x-\mu_1)^2}{2\sigma_1^2}dx &= \frac{1}{2\sigma_1^2}(\int x^2 q dx + \int q\mu_1^2 - \int 2\mu_1 x q dx) \\
&= \frac{EX^2 + \mu_1^2 - 2\mu_1 EX}{2\sigma_1^2} \\
&= \frac{\sigma_0^2 + \mu_0^2 + \mu_1^2 - 2\mu_1\mu_0}{2\sigma_1^2} \\
&= \frac{\sigma_0^2 + (\mu_1 - \mu_0)^2}{2\sigma_1^2} \\
KL(q||p) &= \log \frac{\sigma_1}{\sigma_0} - \frac{1}{2\sigma_0^2} + \frac{\sigma_0^2 + (\mu_1 - \mu_0)^2}{2\sigma_1^2}
\end{aligned}
$$

This section was written with the help of material from the following papers: [2] and [3].

4 Neural Networks and CNNs

A neural net receives input at the input layer, transforms it through the hidden layer through a non-linear transformation before outputting the final outcomes at the output layer. The hidden layer is made of neurons that are "fully connected" to the previous layer.

Fully Connected: This type of connection implies neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer are not.

4.1 Activation functions

4.1.1 sigmoid

$\sigma(x) = \frac{1}{1+\exp -x}$ squeezes the values in $(-\infty, \infty)$ to $[0, 1]$ (negative numbers become 0 and positive numbers become 1).

sigmoid saturation: Sigmoid outputs saturate near zero or one. And hence the gradients will be zero. This will often lead to limited learning.

sigmoid outputs are not zero centered: This causes the neurons in later layers (following a sigmoid output) to receive non-zero centered input. This can create issues as if the data coming in is always positive, the last few layers will have only positive or negative gradients. This gives rise to a zig-zag dynamics

tanh activation: $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} = 2\sigma(2x) - 1$ It also has the same saturated gradient problem, but its output is zero-centered and hence is preferred over sigmoid

ReLu: $max(0, x)$ helps get rid of the vanishing gradients problem, speeds up stochastic gradient descent. However ReLUs are suspectible to death because of say a large negative bias term which will always output 0. Leaky ReLUs exist to get rid of this problem ($y = 0.01x$ for $x < 0$) so that the gradients have a chance to recover in the long run because of the positive gradient at $x < 0$.

Choosing Kernel Sizes

This is arbitrary, mostly. The only thumb rule is larger kernel sizes are slower to train, give more granularity on the number of features that can be learned.

5 Why CNN for images?

Regular neural nets will not scale for images. Images are three dimensional entities. An image of size $32x32x3$ (where 3 = number of channels) will have $32x32x3 = 3078$ weights to learn. This will blowup for a deeper layer and/or a larger image size.

CNNs take the advantage of images having a third dimension. So ConvNets also have three dimensions: width height and depth. The "depth" here is not the depth of the network, but the depth of the channels.

Each convolution layer consists of a set of learnable filters. The "filter" is a 2D matrix whose cell matrices represent the weights we want to learn. It extends all the way along the depth of the input volume (the number of channels). A filter of size $5x5x3$ will have a height and width of 5 and has 3 channels, same as the number of channels in input image. Convolution involves sliding this filter over the image and taking the dot products between the entries of the filter and the input. If we start with 10 filters, we will have a stack of 10 2D activation maps. These can then be stacked to produce the output.

The spatial extent(height and width) of the filter is called receptive field.

Example: If the input image is size $32x32x3$ and the filter size is $5x5x3$, we have $5x5x3 = 75$ + 1 (bias) = 76 parameters to learn.

5.1 Parameter Sharing

Consider input image of size $227x227x3$. Consider the filter size to be $F = 11$, stride to be $S = 4$ and padding to be $P = 0$. Assume $K = 96$ is the depth of this convnet. So the resulting volume will be $\frac{227-11}{4} + 1 = 55$. So we have a total of $55x55x96 = 290,400$. Each of these neurons are connected to a image window of size $11x11x3$ and neuron along the depth of 96 covers one single region of size $11x11x3$.

Each neuron has $11x11x3 = 363 + 1(\text{bias}) = 364$ weights that are unknown. But if learn these weights for $55x55x96 = 290,400$ neurons, it blows up to $290,400x364$ which is too large.

However, if we think about what's being learned through these filters is features. If a feature is important to be learned at some spatial location $(x1, y1)$ it should also be important at $(x2, y2)$

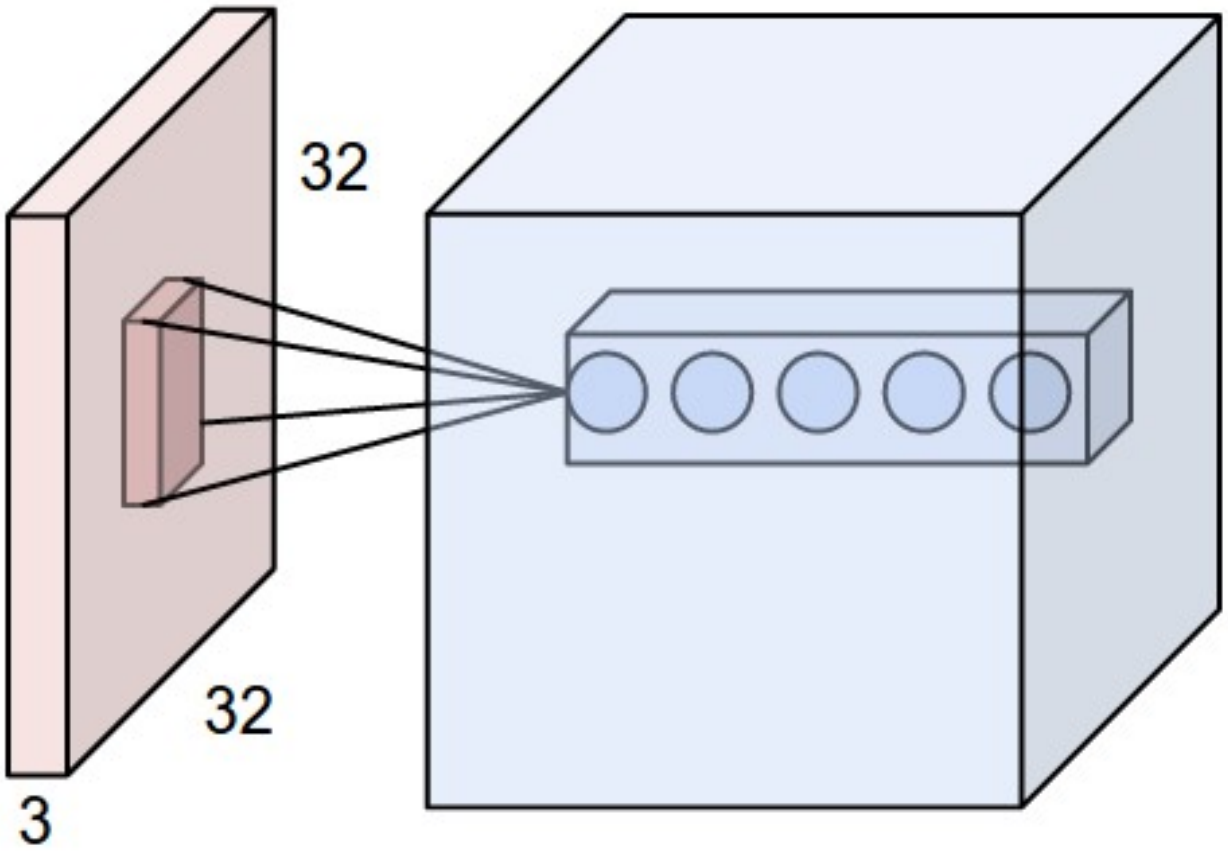


Figure 3: All 5 neurons along the depth capture the same spatial region and have different set of parameters.

and hence we don't need to learn separate weights for all neurons that are in the same level. More intuitively, if detecting edge at one location is important, it should be important to do so at other location too and thatcan be done using the same filter. So all $55x55$ neurons in each depth slice (out of 96) share the same set of parameters (weights and biases), so we end up learning only $96x11x11x3 = 34848$ parameters.

5.2 Summary of a Conv Layer

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Hyper parameters:
  - Number (Depth) of filters: K
  - Spatial extent (width=height): F
  - Stride: S
  - Zero Padding: P
- Results in an output volume of $W_2 \times H_2 \times D_2$

- $D_2 = K$
- $W_2 = H_2 = \frac{W_1 - F + 2P}{S} + 1$

- With parameter sharing, there are only $(F \times F \times D_1) \times K$ weights and K biases learned where the weights are shared over each $W_2 \times H_2$ sized area slice for depth D_1
- In the output volume the values at the d^{Th} slice of area $W_2 \times H_2$ is obtained by performing convolution using the d^{th} kernel
-

5.3 Pooling layer

Pooling reduces the spatial size of the representation thus reducing the amount of parameters and making the entire operation translation indepedent of the input. Larger size of pooling kernel is destructive. most common is $F = 2, S = 2$ or $F = 3, S = 2$.

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Two hyperparameters:
  - Stride S
  - Spatial extent F
- Produces a volume of size $W_2 \times H_2 \times D_2$
  - $W_1 = H_2 = \frac{W_1 - F}{S} + 1$
  - $D_2 = D_1$

6 AlexNet and related papers

Used dropouts for preventing overfitting. TODO.

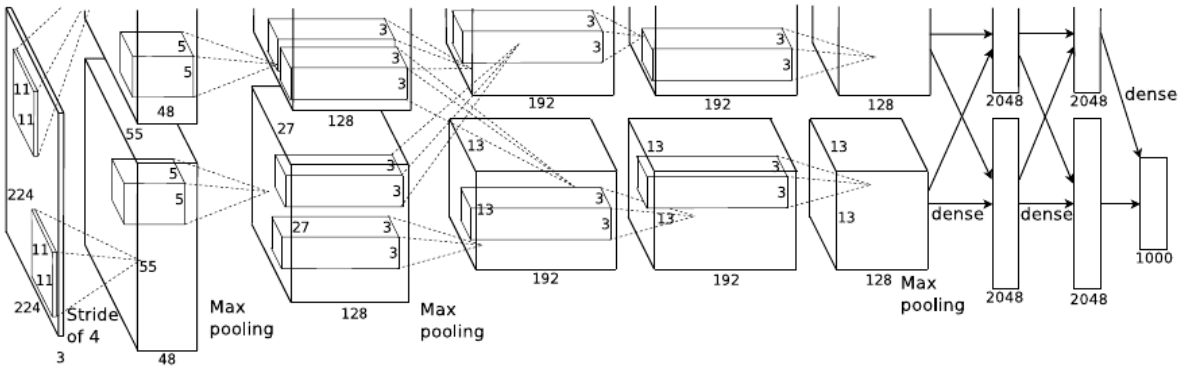


Figure 4: AlexNet

7 Variational Inference

A common problem in modern statistics is to be able to approximate difficult to compute probability distributions. The most common use of this in bayesian inference where the aim is to infer the unknown quantities using a posterior distribution.

Variational Inference converts the problem of inferring this probability distribution to an approximation problem. It can be thought of as an alternate to MCMC, but is faster and scales easily to larger datasets.

Consider we have a probability distribution $p(\mathbf{x})$. $\mathbf{z} = z_1, z_2, z_3 \dots z_m$ are m latent variables and $\mathbf{x} = x_1, x_2 \dots x_n$ are n observed random variables. Assume we have large number of datapoints.

$$p(x) = \int p(z)p(x|z)dz$$

7.1 MCMC and Why not MCMC?

Our aim is to infer $p(z|x)$. In an MCMC setting, we would create an ergodic markov chain on z, whose stationary distribution is $p(z|x)$. We would then sample from this chain to collect samples from the stationary distribution and the posterior will be estimated empirically from a subset of these collected samples.

We need a faster way than MCMC to be able to do this, especially given our assumption of large dataset. So we will replace the slow step of sampling with optimization.

MCMC and Variational Inference (VI) essentially solve the same problem. MCMC samples a markov chain while VI solves an optimization problem. MCMC approximates the posterior of the chain while VI approximates through optimization.

7.2 Variational Inference

Let $\mathbb{Q}$ represent a family of approximate densities over the latent variables z. We want to find a member $q^*\mathbf{z}$ such that the KL divergence between this family and the posterior $p(z|x)$ is minimized:

$$q^*(\mathbf{z}) = \text{argmin}_{q(z)\in Q}\ \text{KL}(q(z)||p(z|x))$$

Hence our original inference problem has been converted to an optimization problem. The complexity of this optimization problem depends on how complex is our choice of family of distribution $\mathbb{Q}$. We want $\mathbb{Q}$ to be flexible enough so that $p(z|x)$ can be approximated closely but simple enough so that the problem is still tractable.

Our goal is to be able to approximate a conditional density of the latent variables $p(z|x)$ given observed variables $\mathbf{x}$.

Our strategy would be to use a family of densities over the latent variables, parametrizing them "free" variational parameters. Optimization will find the member of this family with the setting of these parameters that is closest in KL divergence to $p(z|x)$

7.3 Evidence Lower Bound

is carried ou Consider the KL divergence equation:

$$\begin{aligned}
\mathrm{KL}(q(z)||p(z|x)) &= \int q(z)\log(\frac{q(z)}{p(z|x)}) \\
&= \mathbb{E}_{||}[\log(q(z))] - \mathbb{E}_{||}[\log(p(z|x))] \\
&= \mathbb{E}_{||}[\log(q(z))] - \mathbb{E}_{||}[\log(\frac{p(x,z)}{p(x)})] \\
&= \mathbb{E}_{||}[\log(q(z))] - \mathbb{E}_{||}[\log(p(x,z))] + \mathbb{E}_{||}[\log p(x)] \\
&= \mathbb{E}_{||}[\log(q(z))] - \mathbb{E}_{||}[\log(p(x,z))] + \log p(x)
\end{aligned}$$

However the above equation is intractable because $\log(p(x))$ is intractable. So we optimize an alternate function: $ELBO(q)$

$$\begin{aligned}
ELBO(q) &= \mathbb{E}_q[\log(p(x,z))] - \mathbb{E}_q[\log q(z)] \\
&= \mathbb{E}_q[\log(p(z|x)p(x))] - \mathbb{E}_q[\log q(z)] \\
&= \mathbb{E}_q[\log(p(z|x))] + \mathbb{E}_q[\log(p(z))] - \mathbb{E}_q[\log q(z)] \\
&= \mathbb{E}_q[\log(p(x|z))] + \mathbb{E}_q[\log(p(z))] - \mathbb{E}_q[\log q(z)] \\
&= -KL(q(z)||p(z)) + \mathbb{E}_q[\log p(x|z)]
\end{aligned}$$

Why the name ELBO? It Lower BOunds the (log) Evidence. Assuming we have proven that KL divergence is non-negative $KL(q(z)||p(z)) \geq 0$, we get $\log(p(x)) \geq ELBO(q)$

So instead we maximize this $ELBO(q)$ function. The first term of the last equation implies we are penalizing $q(z)$ to be different from $p(z)$ and at the second term implies we are trying to maximize the expected log likelihood of observing x through $p(x|z)$.

The first term in the original $ELBO(q)$ equation is trying to maximize the expected complete log likelihood, which would generally involve an EM solution. When $q(z) = p(z|x)$ then the ELBO is simply $log(p(x))$. So in a traditional EM setup, the E step would involve computing $p(z|x)$ assuming it is tractable unlike in variational inference.

8 Updating mean and variance

Given a large collection of vectors, we want to calculate its mean and variance repeatedly. In particular, we don't want to store things as a matrix for later calculation but want to do these two calculations

$$\begin{aligned}\mu_n &= \frac{1}{n}\sum_{i=1}^{n} x_i \\ &= \frac{1}{n}(x_n + \sum_{1=1}^{n-1} x_i) \\ &= \frac{1}{n}(x_n + (n-1)\mu_{n-1}) \\ &= \mu_{n-1} + \frac{1}{n}(x_n - \mu_{n-1})\end{aligned}$$

8.1 Identities

$$\begin{aligned}x_n - \mu_{n-1} &= n(\mu_n - \mu_{n-1}) \\ x_n - \mu_n &= n(\mu_n - \mu_{n-1}) + \mu_{n-1} - \mu_n \\ &= (n-1)(\mu_n - \mu_{n-1})\end{aligned}$$

8.2 Updating Varinace

Consider

$$\begin{aligned}\sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_n)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(x_i^2 + \mu_n^2 - 2x_i\mu_n) \\ &= \frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2n\mu_n^2 + n\mu_n^2) \\ &= \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \mu_n^2\end{aligned}$$

Define: $S_n = n\sigma_n^2 = \sum_{i=1}^{n} x_i^2 - n\mu_n^2$
Then,

$$
\begin{aligned}
S_n - S_{n-1} &= \sum_{i=1}^{n} x_i^2 - n\mu_n^2 - \sum_{i=1}^{n} x_i^2 + (n-1)\mu_{n-1}^2 \\
&= x_n^2 - n\mu_n^2 + (n-1)\mu_{n-1}^2 \\
&= x_n^2 - \mu_{n-1}^2 - n(\mu_n^2 - \mu_{n-1}^2) \\
&= x_n^2 - \mu_{n-1}^2 - n(\mu_n - \mu_{n-1})(\mu_n + \mu_{n-1}) \\
&= x_n^2 - \mu_{n-1}^2 - (x_n - \mu_{n-1})(\mu_n + \mu_{n-1}) \\
&= x_n^2 - \mu_{n-1}^2 - (x_n\mu_n + x_n\mu_{n-1} - \mu_{n-1}\mu_n - \mu_{n-1}^2) \\
&= x_n^2 - x_n\mu_n - x_n\mu_{n-1} + \mu_{n-1}\mu_n \\
&= (x_n - \mu_n)(x_n - \mu_{n-1}) \\
S_n &= S_{n-1} + (x_n - \mu_n)(x_n - \mu_{n-1})
\end{aligned}
$$

8.2.1 Aliter

$$
\begin{aligned}
S_n &= \sum_{i=1}^{n} (x_i - \mu_n)^2 \\
&= \sum_{i=1}^{n} ((x_i - \mu_{n-1}) - (\mu_n - \mu_{n-1}))^2 \\
&= \sum_{i=1}^{n} (x_i - \mu_{n-1})^2 + \sum_{i=1}^{n} (\mu_n - \mu_{n-1})^2 - 2\sum_{i=1}^{n} (x_i - \mu_{n-1})(\mu_n - \mu_{n-1})
\end{aligned}
$$

Consider individual terms,

$$
\begin{aligned}
\sum_{i=1}^{n} (x_i - \mu_{n-1})^2 &= (x_n - \mu_{n-1})^2 + \sum_{i=1}^{n-1} (x_i - \mu_{n-1})^2 \\
&= (x_n - \mu_{n-1})^2 + S_{n-1} \\
&= n^2(\mu_n - \mu_{n-1})^2 + S_{n-1}
\end{aligned}
$$

$$
\sum_{i=1}^{n} (x_i - \mu_{n-1})^2 = n(\mu_n - \mu_{n-1})^2
$$

$$\begin{aligned}\sum_{i=1}^{n}(x_i-\mu_{n-1})(\mu_n-\mu_{n-1}) &= (\mu_n-\mu_{n-1})\sum_{i=1}^{n}(x_i-\mu_{n-1})\\ &= (\mu_n-\mu_{n-1})(x_n-\mu_{n-1}+\sum_{i=1}^{n-1}(x_i-\mu_{n-1}))\\ &= (\mu_n-\mu_{n-1})(x_n-\mu_{n-1}-(n-1)\mu_{n-1}+\sum_{i=1}^{n-1}x_i)\\ &= n(\mu_n-\mu_{n-1})^2\end{aligned}$$

Thus,

$$S_n = S_{n-1} + n(n-1)(\mu_n-\mu_{n-1})^2$$

9 PCA/ICA/CA

Given data points $\{x_1, x_2, \ldots, x_N\}$ in and $\mathbb{R}^D$ space, where $N > D$. We want to represent these datapoints in a reduced dimension $M < D$ such that the variance represented in this domain is maximized.

Let's take the simplest case when $M = 1$. We are hence looking for a direction u_1, such that $u_1 u_1^T = 1$'

So we have the following information:

Original point: $x_{D\times 1}$

Projected point: $u_1^T x$ where u_1 is $D \times 1$

Define $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$

$$\begin{aligned}\text{Variance} &= \frac{1}{N}\sum_{i=1}^{N}(u_1^T x_i - u_1^T\bar{x})^2\\ &= \frac{1}{N}\sum_{i=1}^{N}(u_1^T x_i - u_1^T\bar{x})(u_1^T x_i - u_1^T\bar{x})^T\\ &= u_1^T S u_1\\ S &= \frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})(x_i-\bar{x})^T\end{aligned}$$

Since we want to maximize variance, it is equivalent to maximizing TODO

10 UMAP

UMAP [4] is another dimensionaloty reduction technique. In gist, the topological structure of the pictured data is an approximately homotopy to the manifold from where the data weas sampled. In topology two continuous functions are homotopic if one can be continuously deformed into the

other, For example, a donut can be deformed into a cup. In a more non-prceise manner, this means the connected componets are connected on the manifold too and the disconnected components are disconnected on the manifold.

On KL divergence and its asymmetricness

References

[1] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to Machine Learning for physicists,"

[2] C. Doersch, "Tutorial on Variational Autoencoders,"

[3] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes,"

[4] L. McInnes and J. Healy, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," 00004.