DATA SCIENCE II: Machine Learning MTH 9899 Baruch College

Lecture 5: Genetic Algorithms and Optimization

Adrian Sisser

May 9, 2018

Outline

1. Optimization
   - Genetic Algorithms
2. Practical ML
3. Neural Networks
   - Activation Functions
4. Training Tricks
   - Dropout
   - Batch Normalization
5. Neural Network Topologies
   - RNNs

Optimization

Optimization is an important part of ML.

- Many of the techniques we've talked involve iterative algorithms to solve: k-Means, Regression Trees, Forward/Backward Selection, Neural Networks, etc.
- Regression Trees (at least the recursive partitioning approach that we've talked about) take a greedy approach to solving the problem of splitting - this isn't globally optimal.
- k-Means and EM are both sensitive to starting points.

Are there any ways to try to find globally optimal solutions?

Outline

1. Optimization
   - Genetic Algorithms
2. Practical ML
3. Neural Networks
   - Activation Functions
4. Training Tricks
   - Dropout
   - Batch Normalization
5. Neural Network Topologies
   - RNNs

GAs - An Overview

Genetic Algorithms (GAs), are an attempt to mimic a biological system. They borrow heavily from biology for terminology and concepts.

Chromosome A representation of a potential solution.

Gene A particular parameter value on a chromosome.

Genotype A candidate solution's underlying chromosomes.

Phenotype A candidate solution's 'appearance'. Often the same as the genotype.

Mutation A 'random' change to a candidate solution that might or might not improve it.

Fitness A measure of a candidate solution's performance, ie R^2 for regression.

Generation A set of candidate solutions that exist and 'compete' together.

High-level Algorithm

So what do we do? A very rough overview:

- Create an initial generation of n, possible solutions.
- Test the fitness of these candidate solutions, and pick the best j.
- Take these j solutions, and create many more variants, by 'mutating' them.
- Go back to the 'fitness test', and repeat this process until you have a 'good' solution.

Mutations

So how do we mutate things?

Crossover 'Swap' parts of two chromosomes by switching adjoining sections of 2 chromosomes.

Point Mutation Change a given gene's value randomly, in hopes it will produce a better solution.

A Practical Application

One simple example of a GA in practice is variable selection in regression. We've looked at a few techniques for solving this:

- Forward Selection - Start with an empty set and add in the most relevant variables
- Backward Selection - Start with a full set and remove the least relevant variables
- Lasso - Penalize the norm of β to reduce the complexity.

In practice, these all have significant issues. Can we use a GA to try and solve this?

Modeling Variable Selection

What's a reasonable way to model this problem? We can create a simple bit-string, representing whether or not each variable is included in the regression.

$$\begin{aligned} C &= b_0, b_1, ..., b_F \\ \text{where} \quad & b_i \in 0, 1 \\ & \text{and } b_i = 1 \text{ means feature } i \text{ is turned on} \end{aligned}$$

The Parameters

There are a lot of ’knobs’ to turn in a Genetic Algorithm:

- Mutation Probabilities - the probability of a point mutation or a cross-over
- Population Sizes - A larger population allows weaker members to stay around, and might improve overall results
- Mutation Types - The ’geometry’ of cross-over mutations
- Generation Selection - In the example, we went with a simple top n selection. More advanced techniques do things like limit the number of children that can survive.

Applications

There are a lot of novel GA based applications out there:

- **NEAT** - NeuroEvolution of Augmenting Topologies - A technique to try and evolve NN topologies
- Neural Networks - Can use GA for training rather than backpropagation
- KMeans - We can try crossing over various set memberships to see if we can get more stable long-term solutions.
- Regression Trees - The evtree package in R, implements GA evolution of a regression tree

Feature Importance

Feature importance is the study of what features in a ’Black Box’ ML model are driving the results. Marcos describes it’s importance in his “First Law of Backtesting”

> *Backtesting is not a research tool. Feature importance is.*
>
> *– Marcos Lopez de Prado, Advances in Financial Machine Learning*

MDA

Lopez de Prado discusses several techniques, one is Mean Decrease Accuracy, MDA. The procedure is simple.

- For each feature, *shuffle* the inputs for feature i.
- For the shuffled input set, generate predictions and measure the accuracy.
- Repeat as many times as needed for each feature.

You can now compare the prediction accuracy with feature i before and after shuffling.

MDA

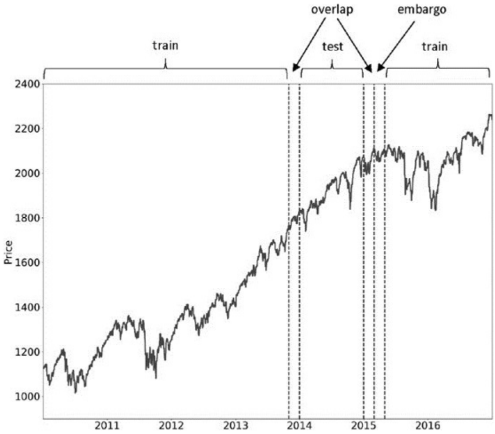
This technique has a few strengths vs others:

- You don't have to refit a model.
- It can work on any ML technique.
- It can capture the interaction of 2 features.
- It's intuitive.

Cross-Validation

CV has particular challenges in finance. Primary, almost all financial modelling involves a timeseries. This leads to 2 particular phenomena:

- In a timeseries where you are modelling multiple periods forward, the observations overlap each other. This means that your test set contains points overlapping with the training set and you don't have independent samples.
- Autocorrelation means that points that aren't directly overlapping from observable data may have overlap in a latent variable that you aren't measuring.



[1] Source: Advances in Financial Machine Learning, Marcos Lopez de Prado

Outline

1. Optimization
   - Genetic Algorithms
2. Practical ML
3. Neural Networks
   - Activation Functions
4. Training Tricks
   - Dropout
   - Batch Normalization
5. Neural Network Topologies
   - RNNs

So far, we've mostly spoken about the sigmoid activation function, as it's the 'original' and has nice derivative properties:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x))$$

Output Activation Functions For Classification

For classification outputs, we need a special activation function. The most popular choice is the Softmax activation:

$$\mathcal{A}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

This converts the output to a PDF across the different class of outputs.

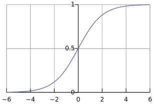
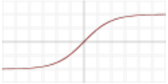
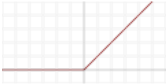
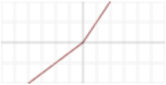
What are the requirements for a good activation function:

- Non-Linear - otherwise we're just fitting a linear model.
- Differentiable? - Not exactly - it just needs to be subdifferentiable
- Not easily saturated..

Does a step function work?

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$

No! It is subdifferentiable, but the gradient is 0 for most of it, so we can't make any progress with Gradient Descent.

Name	$\mathcal{A}(x)$	$\frac{d\mathcal{A}}{dx}$	
Sigmoid	$\sigma(x)$	$\sigma(x)(1-\sigma(x))$	
Tanh	$\tanh(x)$	$1-\tanh^2(x)$	
Rectified Linear Unit (ReLU)	$\max(x,0)$	$\begin{cases} 0 & \text{for} \quad x<0 \\ 1 & \text{for} \quad x\geq 0 \end{cases}$	
Parameterized ReLU	$\begin{cases} \alpha x & \text{for} \quad x<0 \\ x & \text{for} \quad x\geq 0 \end{cases}$	$\begin{cases} \alpha & \text{for} \quad x<0 \\ 1 & \text{for} \quad x\geq 0 \end{cases}$	

So why do we have 2 versions of ReLU?

ReLU

ReLU (and it's variants) have become extremely popular:

- PReLu helps the fact that we have a large area with 0 gradient in 'vanilla' ReLU - this means that we can get 'stuck' in this region and the neuron becomes useless.
- ReLu has a gradient of 1 in it's 'active' area. This means that if we build very deep networks, we don't have to worry about the gradient vanishing.

Vanishing Gradient

The “Vanishing Gradient Problem” is a well known one in training deep neural networks. The basic idea is that when you use a sigmoid activation function, for a very large area, the gradient is nearly 0. When we have a deep network, with many layers and we apply back propagation, by the time we get back to the early layers, the gradient is extremely small.

ReLU fixes this because the derivative when you are in the ’active’ region of the neuron is 1.

This has been a fundamental change in NN and enabled much deeper networks.

Vanishing Gradient

There are other ways to fix the Vanishing Gradient Problem.

- Other activation functions that don't become saturated.
- Training methods - we can train a network layer by layer in an unsupervised manner.
- Residual Network (ResNet) - We feed the input from previous layer(s) plus the output of the previous hidden layer to each layer.

Outline

1. Optimization
   - Genetic Algorithms
2. Practical ML
3. Neural Networks
   - Activation Functions
4. Training Tricks
   - Dropout
   - Batch Normalization
5. Neural Network Topologies
   - RNNs

Dropout

Dropout is a very effective technique to avoid overfitting in deep NNs. The idea is to randomly 'dropout' certain units from an NN during training. By randomly selecting which units to drop out, we avoid overfitting and learning spurious patterns from the input data.

$$r \sim \text{Bernoulli}(p)$$

$$h_{\text{out},i} = \begin{cases} \mathcal{A}(XW_i + B_i) & \text{for} \quad r = 0 \\ 0 & \text{for} \quad r = 1 \end{cases}$$

- We pick probability p that a given unit is ’dropped’ from a given. Usually 0.5 is a good choice.
- We can think of all of this as us training random subset of features of the original dataset.
- When we go to do a prediction, we use all nodes, but scale them by the dropout probability, p.

Outline

1. Optimization
   - Genetic Algorithms
2. Practical ML
3. Neural Networks
   - Activation Functions
4. Training Tricks
   - Dropout
   - Batch Normalization
5. Neural Network Topologies
   - RNNs

Batch Normalization is another technique to speed up convergence of NN training. It is based on the idea that while we normalize overall input values to be $N \sim (0, 1)$, each minibatch might not have the same distribution, and $N \sim (0, 1)$ might not be ideal. Let X be the input matrix for a layer, l and μ_i and σ_i represent the mean and sd of the ith column. We will transform every input as:

$$
\begin{aligned}
X' &= XW + B \\
X_i'' &= \gamma_i \frac{(X_i' - \mu_i)}{\sigma_i} + \beta_i \\
h_{\text{out}} &= \mathcal{A}(X'')
\end{aligned}
$$

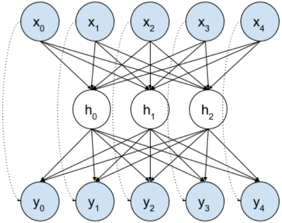
Here, we have transformed each column of X'' to be $N \sim (\beta, \gamma)$.

It is important to note that γ and β are learned parameters here - they help us figure out the optimal scale for training each layer. A few other notes:

- We can trivially extend backpropagation to allow us to learn β and γ
- This is a fairly new technique published in 2015 by Google.
- The initial paper shows training networks more than 10x faster for the same quality
- This improvement combined with existing techniques allowed a significantly lower error rates on ImageNet Classification.

Autoencoders

Autoencoders are a simple type of feedforward network. The goal is to train a simple 1 layer hidden network with the same data for x and y, and so learn to 'recreate' the input. The hope is to learn high-level features that describe the input.



The dashed lines indicate that the values are equal

What would an autoencoder with a small h and a linear activation function be doing?

PCA!!

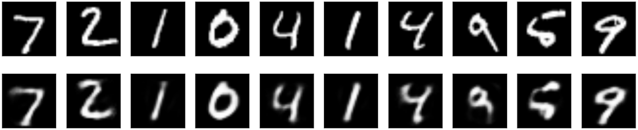
By choosing more interesting functions other than linear, we can get more interested compressed representations.

We have a few ways to constrain our Autoencoder.

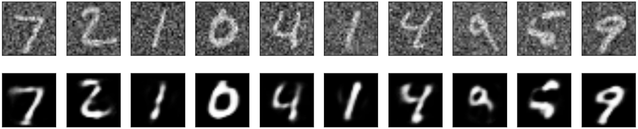
- We can control the size of the hidden layer. By making h smaller than the input, we force the autoencoder to only learn essential features of the data.
- We can force the autoencoder to be *small*, ie $\mathcal{L}' = \mathcal{L} + \|W\|_2$.
- We can force the autoencoder to be *sparse*, ie $\mathcal{L}' = \mathcal{L} + \|W\|_1$.

Let's look at an example. We'll train an autoencoder on MNIST data. A few notes:

- We'll be looking at 28 x 28 grayscale images.
- We are shrinking these 784 pixels down to only 32 hidden units
- We trained for 50 epochs, using cross-entropy loss on a pixel-by-pixel basis.



Autoencoders are also very good at getting rid of noise:



There are a few twists on Autoencoders:

- Denoising Autoencoders - We deliberately 'corrupt' our input data and train on that.
- Deep Autoencoders - We can train our autoencoder using more than one hidden layer.
- Stacked Autoencoders - We can train a deep network by layering autoencoders together, where we use the hidden vector representation of each autoencoder as the input(/output) of the next layer.

Outline

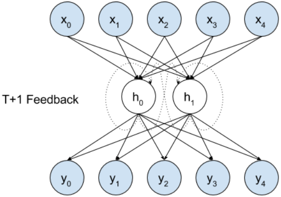
1. Optimization
   - Genetic Algorithms
2. Practical ML
3. Neural Networks
   - Activation Functions
4. Training Tricks
   - Dropout
   - Batch Normalization
5. Neural Network Topologies
   - RNNs

Recurrent NNs

So far, the networks we have talked about have been straightforward “Feed-Forward Networks”. That is to say, they have been an directed acyclic graph, where information only flows in one direction. It’s natural to wonder how we can use NNs to learn about time-based patterns. A few examples would be:

- Financial Time Series - We are trying to fit the future return of a stock to the past n days return.
- Video - We want to identify what’s happening in a frame of video - it’s natural to want to know what prior frames looked like.
- Text - If we want to predict the next word in a paragraph, we need to know the prior words.
- Speech - To recognize a sentence, it’s useful to know the words that come before ... and after.

Now, we will talk about Recurrent NNs (RNNs). The basic idea is that values from our hidden layers are fed back in at the next time cycle. This allows us to model past inputs.



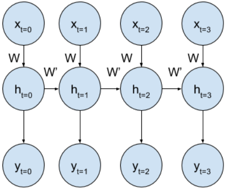
Simple RNN

So one obvious question is, how do we train this? The network has a loop in it!

The idea here is to *unroll* our network across time. We will keep the same W matrix at each time.

For simplicity, we have compressed each layer of each time step into a single ‘node’. The subscripts represent a time index.

NOTE: W MATRIX IS THE SAME IN EACH STEP

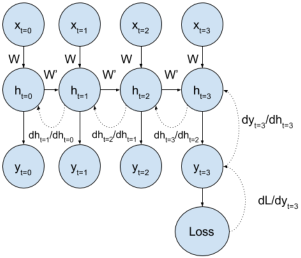


BPTT

Now that we've unrolled our network, we can again do a simple backpropagation algorithm. We call this Backpropagation Through Time or BPTT.

For simplicity, we have compressed each layer of each time step into a single 'node'. The subscripts represent a time index.

NOTE: W MATRIX IS THE SAME IN EACH STEP



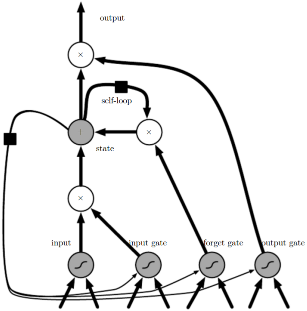
There is one big problem with the simple RNN model that we've discussed so far ... **it doesn't work**!

In practice, this model of RNNs suffers from the same Vanishing/Exploding Gradient problem that we spoke about. If we think of a simple model that is trying to learn the next word in a paragraph, we could easily need a memory going back 20+ words. As we already know, this is extremely hard.

$$\frac{\partial L}{\partial h_{t=0}} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h_{t=3}} \frac{\partial h_{t=3}}{\partial h_{t=2}} \frac{\partial h_{t=2}}{\partial h_{t=1}} \frac{\partial h_{t=1}}{\partial h_{t=0}}$$

LSTM

One solution that was developed to solve these issues with simple RNNs is the confusingly named LSTM - Long Short Term Memory model. LSTMs look incredibly complicated, but can be understood with a few minutes of thinking about it.



[1] Source: http://www.deeplearningbook.org/contents/rnn.html

LSTM

Below, we will use c to represent the 'state' of the cell, and H to represent hidden layer values.

Forget Gate The forget gate controls whether or not we discard the prior memory state, c:

$$f_t = \sigma(X_t W_{f,x} + H_{t-1} W_{f,H} + b_f)$$

Input Gate The input gate controls how much the new incoming information gets incorporated into the memory state:

$$i_t = \sigma(X_t W_{i,x} + H_{t-1} W_{i,H} + b_i)$$

LSTM

Output Gate The output gate controls whether or not the current memory state is output:

$$o_t = \sigma(X_t W_{o,x} + H_{t-1} W_{o,H} + b_o)$$

Memory Update The memory cell is updated based on how much we ’forget’ and how much we accept new ’input’:

$$c_t = f_t \circ c_{t-1} + i_t \circ \mathcal{A}_{\text{tanh}}(X_t W_{c,x} + H_{t-1} W_{c,H} + b_c)$$

Hidden Layer Output The hidden layer output is based on the ’output’ gate and the new state:

$$h_t = o_t \circ \mathcal{A}_{\text{tanh}}(c_t)$$

So how do LSTMs help? Well, the hidden state going out is a function of the current memory cell, c. As long as the forget gate, f, stays near one, we keep our memory state around. This is much easier than the vanishing gradients of a 'vanilla' RNN. We also add in the new changes to c, rather than multiply, so they propagate backwards better.