MACHINE LEARNING

Created: February 15, 2021
Last modified: January 27, 2023

Tony Menzo

1 Introduction

1.1 The landscape

2 Training models

2.1 Gradient Descent

The Gradient Descent (GD) training model is an iterative optimization that gradually tweaks the model parameters to minimize the **cost function** over the training set, eventually converging to the same set of parameters as the first method. GD is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. GD measures the local gradient of the error function with regards to the parameter vector $\boldsymbol{\theta}$, and it goes in the direction of the descending gradient. In practice we begin by filling the parameter vector $\boldsymbol{\theta}$ with random values (random initialization) and perform the algorithm iteratively until it converges to a minimum of the cost function.

Of critical importance in the GD algorithm is the step size h. The size of the step is determined by the learning rate parameter. If the learning rate is too small, the algorithm will take many iterations to converge whereas if the learning rate is too fast, the optimum $\boldsymbol{\theta}$ i.e. the minimum of the cost function may be missed entirely. On top of that, not all cost functions have no local minima. We could work with a cost function which as local minima as well as saddle points. This complicates the use of gradient descent because we are interested in the **global** minimum of the cost function.

Training a model using gradient descent means searching for a combination of model parameters that minimizes a cost function. It's a search in the model's *parameter* space: the more parameters a model has, the more dimensions the space has, and the harder the search is.

In order to implement GD, we need to compute the gradient of a cost function with respect to each model parameter θ_i. Given the Mean Square Error (MSE) cost function for a model hypothesis $h_{\boldsymbol{\theta}}$ on a training dataset $\mathbf{X}$

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)} \right)^2 \tag{1}$$

The partial derivative with respect to the jth parameter is given by

$$\frac{\partial}{\partial \theta_j}\text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)} \tag{2}$$

This can be represented as a matrix

$$\nabla_\theta \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1}\text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) \tag{3}$$

Note that this method involves calculations over the full training set $\mathbf{X}$, at each GD step. Thus, this algorithm is terms *Batch Gradient Descent*. This makes it especially slow on large training sets. But it does scale well with the number of features.

Once we have the gradient vector, which points uphill, just go in the opposite direction to go downhill i.e. subtracting the gradient vector $\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta})$ from $\boldsymbol{\theta}$. This is precisely where the learning rate hyperparameter η comes into play: multiply the gradient vector by η to determine the size of the downhill step.

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta}) \tag{4}$$

Stochastic Gradient Descent:

On the flip-side, stochastic gradient descent picks a random instance in the training set at every step and computes the gradients based only on the single instance. This makes it much faster than the BGD due to the smaller amount of data to manipulate at each step and it also makes it possible to train on huge datasets (out-of-core). Do to its random nature, the SGD is much less regular than BGD: the cost function will bounce up and down, decreasing only on average. Over time it will end up near the minimum but will continue to bounce around. Thus, the final parameters are good, but no optimal. This training technique is also suited for handling irregular cost functions, given that it is possible for the algorithm to jump out of local minima, giving it a better chance of finding the global minima.

As discussed above, randomness is good for finding the global minima but bad for converging on the absolute minimum. One solution is to gradually reduce the learning

rate, starting with large steps (to make quick progress and jump out of local minima) and gradually reducing allowing the algorithm to settle at the global minimum. This process is termed *simulated annealing*, and algorithm inspired from the process of annealing in metallurgy. The function which determines the learning rate at each iteration is called the *learning schedule*.

Mini-batch Gradient Descent:

The last GD algorithm we look at is called *Mini-batch Gradient Descent* (MbGD). As you may guess, instead of computing the gradients based on the full taining set or based on just one instance, MbGD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of MbGDs over SGD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

3 Testing and Validating

To see how your model generalizes to new data you of course need to test it out on new data outside of your training dataset.

Best practice is to split your data into two sets: the training dataset and the test dataset. The first is used to train your model and the second is used to verify or nullify the effectiveness of your model. The error rate obtained from the test data set is called the generalization error (or out-of-sample error).

If the training error is low but the generalization error is high, this indicates that your model is overfitting the training data.

Suppose your have two models model A and model B. How to we choose between the two models? The obvious, and unavoidable answer would be to train both models on a training dataset and choose the model which produces a lower generalization error on the test dataset. Say model A produces a lower generalization error but you want to apply some regularization to avoid overfitting. Now we must answer the question of how to choose the value of the regularization hyperparameter. One option would be to train 100 different models using 100 different values for the hyperparameter on the training dataset. Say you follow this methodology and find the model which produces the lowest generalization error of say 5%. You launch this model into production and find that it doesn't perform as well as expected and produces a generalization error of 15%. What happened?

The problem is that you measured the generalization error multiple times on the test

set, and you adapted your model and hyperparameters to produce the best model *for that particular set*. This means that your model is unlikely to perform well on new datasets.

A common solution to the problem stated above is called *holdout validation*: you simply hold out part of the training set to evaluate several candidate models and select the best one. The held out set is called the *validation* set (or *development* set, or *dev* set).Specifically, you train multiple models with various hyperparameters on the reduced training set and you select the model that performs the best on the validation set. After the holdout validation step you retrain the model on the full dataset (reduced training set + validation set). Then, of course, you evaluate the final model on the test dataset.

This works quite well unless the validation set is too small. This will cause the model evaluations to be imprecise and may cause you to select a sub-optimal model. On the flip-side, if the validation set is too large, then the remaining training set will be much smaller than the full training set. This can be bad because it's not ideal to compare candidate models trained on much smaller training sets (compared to the full dataset). A nice example would be like choosing the fastest sprinter to participate in a marathon.

We can fix this by performing repeated *cross-validation*, using many small validation sets. Each model is evaluated once per validation set and then the evaluations are averaged over all validation sets. This does however increase the training time significantly depending on how many validation sets are used.

4 The Perceptron

The *Perceptron* is one of the simplest *artificial neural network* (ANN) architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The inputs are numbers and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = \mathbf{x}^T\mathbf{w}$), then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$, where again $z = \mathbf{x}^T\mathbf{w}$. Of course the most common step function used in Perceptrons is the Heaviside step function. In some cases the sign function is used instead.

$$\text{heaviside}(z) \equiv \Theta(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ if } z \geq 0 \end{cases} \tag{5}$$

$$\text{sgn}(z) = \begin{cases} -1 \text{ if } z < 0 \\ 0 \text{ if } z = 0 \\ +1 \text{ if } z > 0 \end{cases} \tag{6}$$

A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class. For example you could use a single TLU to classify specific items such as cars or iris flowers based on attributes of the item (petal length, and width for the iris flower). Training the TLU in this case means finding the right values for the weights ($w_0, w_1,$ and w_2). These weights must be a function of there inputs in this case $w_i(x_i)$.

A Perceptron is composed of a single layer of TLUs, with each TLU connected to all the inputs. When all neurons in a layer are connected to every neuron in the previous layer (its input neurons), it is called a *fully connected layer* or a *dense layer*. A Perceptron with two inputs and three outputs can classify instances (a set of attributes) simultaneously into three different binary classes, which makes it a multi-output classifier[1]. It is possible to efficiently compute the outputs of a layers of artificial neurons for several instances at once using

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b}) \tag{7}$$

Where $\mathbf{X}$ represents the matrix of input features (one row per instance, one column per feature), $\mathbf{W}$ is the weight matrix containing all of the connection weights except the ones from the bias neuron (one row per input neuron, one column per artificial neuron in the layer), $\mathbf{b}$ is the bias vector and contains all the connection weights between the bias neuron and the artificial neurons (one bias term per artificial neuron), and ϕ is called the activation function: when the artificial neurons are TLUs, it is a step function.

Training the Perceptron:

The original algorithm for training Perceptrons was inspired by *Hebb's rule* (or *Hebbian learning*): when a biological neuron triggers another neuron often, the connection between those two neurons grows stronger. This is summarized in the following phrase, "Cells that fire together, wire together": that is, the connection weight between two neurons is increased whenever they have the same output. Perceptrons are trained using a variant of this rule which takes into account the error made by the network; it reinforces connections which help reduce the error. Specifically, the Perceptron is fed

[1] unclear to me at this moment why this would be beneficial.

one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection wights for the inputs that would have contributed to the correct prediction. This is encapsulated quantitatively in the following rule

$$w_{i,j^{(\text{next step})}} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \tag{8}$$

Where $w_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron, x_i is the ith input value of the current training instance, $\hat{y}_j$ is the output of the jth output neuron for the current training instance. y_j is the target output of the jth output neuron for the current training instance, and η is the learning rate.

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns. However, if the training instances are linearly separable, this algorithm will converge to a solution (Perceptron convergence theorem).

Single-layer Perceptrons gained notoriety as a useless construct because it failed to solve some trivial problems (for example the exclusive OR or XOR problem). This notoriety abated upon the realization that a ***multi-layer*** Perceptron (MLP) could be used to solve these problems.

4.1 Multi-layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the output layer. Layers close to the input layer are called the *lower layers* and the layers close to the outputs are termed the *upper layers*. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

When an artificial neural network contains a deep stack of hidden layers, it is called a ***deep neural network*** (DNN).

How to train MLPs?:

For years after their introduction, researchers struggled to find a way to train this architecture. In 1986, Rumelhart, Hinton, and Williams published a groundbreaking paper introducing *backpropagation*. In essence, backpropagation is simply Gradient Descent using an efficient technique for computing the gradients automatically: in two passes through the network, the backpropagation algorithm computes the gradients of the network's error with respect to every model parameter and updates the weights in a usual GD step. In other words, it can find how each connection weight and each

bias should be changed in order to reduce the error. This process is repeated until the network converges to a solution. The algorithm is important so let's step through it in more detail:

- The algorithm handles one mini-batch at a time (for example 32 instances), and it goes through the full training set multiple times. Each pass is called an *epoch*.
- Each mini-batch is passed from the input layer to the first hidden layer. The algorithm computes the output of all of the neurons in the layer (for every instance in the mini-batch). This continues until we reach the final output layer. All intermediate results are saved in memory as they will be needed in the backwards pass.
- Next, the algorithm measures the network's output error (uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- It computes how much error each output connection contributed to the error. This is done analytically by applying the chain rule. This is repeated until it reaches the initial input layer (hence the name backpropagation).
- Finally, the algorithm performs a GD step to change all the connection weights in the network using the error gradients it just computed.

It is important to initialize all of the hidden layers' connection weights randomly, or else the training will fail.

In order for the algorithm to work properly we must replace the original step-function with a function which allows the computation of a gradient. These are termed ***activation functions*** for example:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \qquad \text{Logistic function} \tag{9}$$

$$\tanh(z) = 2\sigma(2z) - 1, \qquad \text{Hyperbolic tangent function} \tag{10}$$

$$ReLU(z) = \max(0, z), \qquad \text{Rectified Linear Unit function} \tag{11}$$

4.2 Uses

What can we use MLPs for? Firstly, MLPs can be used for regression tasks. If you want to predict a value, then you just need a single output neuron, its output is the

predicted value. For multivariate regression (i.e. to predict multiple values at once), you need one output neuron per output dimension. In general, when building an MLP for regression, you do not want to use any activation functions for the output neurons, so they are free to output any range of values. However, if you want to guarantee that the output will be positive, then you can use the ReLU activation function, or the *softplus* activation function in the output layer. If you want to guarantee that the predictions will fall within a given range of values, the you can use the logistic function or the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you might prefer to use the mean absolute error instead.

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the logistic activation function: the number will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. MLPs can also easily handle multi-label binary classification tasks.

5 Training Deep Neural Networks

5.1 Vanishing/Exploding Gradient Problems

As we have seen, the backpropagation algorithm works by working from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost funciton for each parameter in the network, it uses the gradients to update each parameter with a GD step.

There is one problem which may arise when using this algorithm. As the algorithm progresses, the gradients often get smaller and smaller as the algorithm progresses down the layers and as a result the algorithm leaves the lower layer connections virtually unchanged and the training never converges to a good solution. This is called the *vanishing gradient* problem. The opposite can also occur in which the gradients grow bigger and bigger causing the algorithm to diverge. This is called the *exploding gradient* problem. More generally, DNNs suffer from unstable gradients; different layers may learn at widely different rates.

Glorot and He Initialization:

In order to avoid exploding and vanishing gradients in the backpropagation step

we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction. It is actually not possible to guarantee both unless the layer has an equal number of input and neurons but we can compromise for a solution which works well in practice: the connection weights of each layer must be initialized randomly as a normal distribution with mean 0 and variance

$$\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}} \tag{12}$$

or a uniform distribution between $-r$ and r, with

$$r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}} \tag{13}$$

Where $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}})/2$. This is known as Glorot initialization and using it can speed up training significantly. The above pertains to the hyperbolic tangent activation function. For the ReLU activation function we should use the *He* initialization and likewise for the SELU (to be described later) activation function we should use the *LeCun* initialization.

5.2 Non-saturating Activation Functions

The problem of vanishing and exploding gradients is in part due to a poor choice in activation function. Until around 2010, when Glorot and Bengio published their paper on initialization techniques, most people assumed that because biological neurons used roughly sigmoid activation functions, these must also be an excellent choice for DNNs. It turns out that this is not necessarily the case and other activation functions behave much better in DNNs. Specifically the ReLU activation, mostly because it does not saturate for positive values and because it is fast to compute.

The ReLU function is not perfect, however. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively die, meaning they stop outputting anything other than 0. In some cases you may find that half of your network's neurons are dead, especially if you use a large learning rate. A neuron dies when its weights get tweaked in such a way that the weighted sum of it inputs are negative for all instances in the training set. When this happens, it outputs 0 and the gradient descent does not affect it because the gradient of the ReLU function is 0 when its input is negative.

To solve this, we may use a variant ReLU activation function such the *leaky ReLU*:

$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z) \tag{14}$$

The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$, and is typically set to 0.01. This ensures that the leaky ReLU never dies. A paper in 2015 compared several variants of the ReLU activation function and concluded that the leaky variants always outperformed the strict ReLU activation function. Interestingly, setting $\alpha = 0.2$ (huge leak) seemed to result in better performance than $\alpha = 0.01$ (small leak). They also evaluated the *random leaky ReLU* (RReLU), where α is picked randomly in a given range during training, and it is fixed to an average value during testing. It performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set). Finally, the *parametric leaky ReLU* (PReLU) was also tested, where α is authorized to be learned during the training (instead of being a hyperparameter, it becomes a parameter that can be modified by back propagation like any other parameter). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

A new activation function was proposed in 2015 called the *exponential linear unit* (ELU) that outperformed all the ReLU variants: training time was reduced and the neural network performed better on the test set.

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(e^z - 1) \text{ if } z < 0 \\ z \text{ if } z \geq 0 \end{cases} \tag{15}$$

The main drawback of the ELU activation function is that it is slower to compute than the ReLU and its variants (due to the exponential function), but during training this is compensated by the faster convergence rate. At test time, anELU network will be slower than a ReLU network.

5.3 Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that the won't come back during training.

In 2015 a technique called *Batch Normalization* (BN) was introduced to address the vanishing/exploding gradients problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden

layer, zero-centering and normalizing each input, then scaling and shifting the result using to new parameter vectors per layer: one for scaling, one for shifting. In other words, this operation lets the model learn the optimal scale and mean of each of the layer's inputs/ In many cases if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set: the BN layer will do it for you (approximately).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and standard deviation of each input over the current mini-batch (hence the name Batch Normalization). It takes all of the values for one input in a mini-batch and normalizes the whole mini-batch. It shifts the inputs in such a way to produce a mean of zero and normalizing such that the sum of all inputs is equal to one.

6 Generative machine learning

6.1 Autoencoders

6.2 Invertible neural networks a.k.a normalizing flows

7 Physics Informed Neural Networks (PINNs)

8 Symbolic regression

Typically when we think about performing a regression analysis on data y_i it coincides with a situation in which we have data and a theory motivated model containing unknown parameters. The regression analysis is then used to make an estimate of these unknown parameters given the data via least squares, maximum log likelihood, etc. While these types of regression analyses perform searches in the space of *parameters*, symbolic regression methods perform searches in the space of *mathematical expressions*. This type of regression analysis coincides with a situation where we have data but not a theory motivated model. It is an attempt to to discover both the model structure as well as it's parameters.

The problem: given a dataset $D = \{\vec{x}_i, y_i\}_{i=1}^n$ where $\vec{x} \in \mathbb{R}^d$ is an input vector and $y_i \in \mathbb{R}$ is a scalar output alongside a class of functions F which consist of mappings $f : \mathbb{R}^d \to \mathbb{R}$ we would like to find the map f^* in F which minimizes the loss function

ℓ

$$\ell(f) \equiv \sum_{i=1}^{n} \ell\left(f(x_i), y_i\right) \tag{16}$$

where, for example, the loss may be something like the mean-squared-error (MSE)

$$\ell(f) \approx \sum_{i} \left(y_i - f(x_i)\right)^2. \tag{17}$$

The major differentiator between the methods used to solve this problem is related to how the function class F is characterized and represented. To define F we specify a library of elementary operations, functions, and variables and consider all function compositions within the library.

There are many algorithmic techniques to solve the symbolic regression problem stated above. At its core it's a problem involving optimized searching over the expression library F. Naturally, because of this, it is a very rapidly progressing field of research in the machine learning literature.

The basic idea behind symbolic regression is to take data as input i.e. $\vec{x}, \vec{v}, m, \rho$, etc. typically as a function of time or some evolution parameter of the system and return a symbolic expression describing the dynamics of the system. For example, take two massive bodies interacting gravitationally: into the black box neural network (our symbolic regressor) we input different time series data over many time steps for many different initial conditions. This would be our training dataset, and as a result we would like the neural network to output something like the force between the two objects which governs their motion i.e.

$$|\vec{F}| = G\frac{m_1 m_2}{r^2}. \tag{18}$$

First attempts at doing something like this essentially developed a lexicon of constants, symbols, functions, and operators

$$\{m_1, m_2, x, y, a, b, +, -, \times, \div, \exp, \log, \sin, \cdots\} \tag{19}$$

randomly select items from the list to form an expression

$$a^2 + \log(x) \tag{20}$$

and compare the output of the expression with all of the time series data given in the training dataset. At the end of a training step you come up with a scheme for altering the expression

$$a^2 - \log(x), \tag{21}$$

rinse and repeat over some number of epochs and hope for the best. As the lexicon grows this approach becomes more and more unfeasible, especially when the attempting to obtain unknown symbolic expressions with no great way of iterating through each configuration of the lexicon. However, there have been other methods developed to tackle this problem (with much greater success) using graph neural networks.

9 Methods

9.1 A gallant first approach: Linear symbolic regression

At first glance, a good starting point would involve defining a function class or 'library' of elementary operations that can then be fit to the data according to a linear combination of linear and nonlinear functions. This *predefines* the model structure and reduces the problem down to solving for the roots of a coupled set of differential equations. That is given a function class $F \supseteq [x, x^2, \cos(x), \sin(x), \exp(x)]$ we make the ansatz

$$f(x) = \sum_j w_j f_j(x) = w_1 x + w_2 x^2 + w_3 \cos(x) + w_4 \sin(x) + w_5 \exp(x) \tag{22}$$

where w_i are the weight coefficients. We can then minimize the squared error loss function

$$\ell(f(x_i)) = \sum \left(y_i - w_1 x_i + w_2 x_i^2 + w_3 \cos(x_i) + w_4 \sin(x_i) + w_5 \exp(x_i)\right)^2 \tag{23}$$

by computing the derivatives with respect to the weight coefficients and setting them equal to zero. In this way we are left with solving for the a system of linear equations which are easily solvable in the ordinary manner i.e. pick your favorite root finding method. Simple extensions allow for this to be done for any $n-$dimensional dataset i.e. above we have: 1 input return 1 output, but we could also easily have a situation in which we have: multiple inputs $x, y, z, \cdots$ returning 1 output $y_i = f(x_i, y_i, z_i, \cdots)$. For one, by design, this analysis hindered by it's inability to capture the multiplication of two functions or situations in which the variable of interest has a multiplicative or additive factor. Not to mention, the size of the library grows exponentially with the number of dependent variables. This limits the generalizability and is ultimately why I have term it a gallant first effort: it works some of the time when the model is sufficiently simple, but most of the time it doesn't.

9.2 Nonlinear symbolic regression

Nonlinear is the 'next-best' iteration after our first attempt and the most straightforward application of machine learning in the space. In summary this technique epitomizes the phrase 'throw the kitchen sink at it'. Here we define a multi-layer feed-forward neural network with a single output node. The activation function layers are replaced by a layer containing our predefined class functions. This effectively reduces the problem to learning the weights of the neural net such that the loss function is minimized. An important caveat in this case is that because a single output can, in principle, be obtained from an infinite number of models we need to introduce a way to incorporate 'brevity' as a better solution. This is typically called enforcing sparsity and can be implemented by an additional term to the loss function:

$$\ell(y_i, f(x_i)) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{n} W^n \cdot W^n \tag{24}$$

where N denotes the total number of data points in the training dataset, λ is a tunable hyperparameter, and n denotes the number of layers in the network with weight layers. This approach is fully differentiable and scales well to higher dimensional problems, back propagation presents as a numerical issue for many elementary functions ($\log, \div$, etc.)

By initializing the weights c_I and computing the loss ℓ between the output and the input data $\vec{x}$ and looking at the derivative $d\ell/c_i$ and adjust parameters by some small value ϵ with sign chosen based on the sign of the gradient. After a certain number of iterations after the loss plateaus you can make an inference as to the analytic form based on the final values of c_I. What I have just described also describe a fully-connected supervised neural net perceptron.

9.3 Genetic Algorithms

Genetic programming is a long standing approach in computer programming in which the program starts with a 'population' of 'individuals' (set of syntax trees) that is generated stochastically. The genetic program then *evolves* the initial population using a set of 'transition rules' (operations) that is defined over the 'individual' or tree space. Commonly used evolutionary operations include: mutation, crossover, and selection. These programs also implement the idea of 'fitness' into the selection operation. This involves selecting k trees from the population randomly and selecting the sample which has the lowest loss i.e. highest fitness. The program goes schematically as: 1.

transition rules are applied to the selected population of trees built randomly from the set of predefined functions 2. the loss function is evaluated on each tree in the population and 3. an elite subset of trees (possessing the highest fitness) are selected for the next iteration.

These types of symbolic regression techniques are sensitive to hyperparameters and do not scale well to higher dimensional datasets.

9.4 Transformers

Transformers are an auto-regressive (each new sample is conditioned by the previously sampled sequence as well as the latent sequence) neural network typically employed in natural language processing models to process sequence data. The basic idea is to use a mechanism known as 'self attention' which allows the network to capture long-range correlations within a sequence. The model itself consists of an encoder and decoder.

While processing a word or symbol, *attention* enables the model to focus on other words in the input that are closely related to that word. The architecture uses this attention mechanism to relate each word in the sequence to every other word in the sequence.

The transformer encoder works approximately as follows:

1. The input data is tokenized and embedded, then fed to the encoder
2. The embedded matrix is then fed into three different 'functions' which contain a linear layer (for backprop) and output the query, key, and value matrices.
3. The attention score matrix can be computed via

$$a = \text{SoftMax}\left(\frac{Q \cdot K}{\sqrt{n_{\text{embedding}}}}\right) V \tag{25}$$

Within a transformer encoder decoder pair (there are usually multiple encoder-decoder pairs within a single transformer) there are three self-attention steps. One self attention in the encoder where it applies self-attention between the input with itself, two in the decoder: the first pay self attention to the target and the second does self attention between the target sequence and the input sequence.

Each attention layer consists of a **value**, **key**, and **query**. There are usually many attention layers processing at the same time in parallel hence the name multi-headed

self attention. By weighting the value by the dot product of the query and key we $\sim$ obtain the attention score for a given sequence represented as a relationship score between each word in the sequence.

10 Applications

What are the actually algorithms which can implement the above methods to solve the symbolic regression problem stated in Sect.(8). Broadly, the approaches can be split into two classes:

1. One-step approach: use the input data directly to learn a symbolic model
2. Two-step approach: Encode the data into a different representation and then try to learn a symbolic model

The first approach can be implemented with limited success. The two step approach has been met with more success. Some possibilities which have been considered in the literature:

1. Learn a model using a regular NN and apply a model-of-a-model symbolic regression method
2. Learn a model using a graph neural network (GNN) and apply symbolic regression to the learned graph model (see Sect.(11)).
3. Learn a reduced representation of the original dataset using a neural net (e.g. an encoder) and apply symbolic regression simultaneously
4. Learn intrinsic properties of data and recursively use them to simplify global symbolic regression problem

11 Discovering Symbolic Models From Deep Learning with Inductive Bias–2006.11287

They propose a general framework to leverage advantages of both deep learning and symbolic regression. The set up can be understood as follows:

1. Create a deep learning model with a separable internal structure that **mimics** or provides an inductive bias (assumption) that is well matched to the nature of the data.

2. Train the deep learning model
3. Implement symbolic regression on the distinct functions learned by the model internally
4. Replace the neural nets in the deep model with the symbolic expressions.

11.1 Graph neural networks

The first thing to know about graph neural networks is that they are built from two components: vertices and edges. The vertices can have properties or 'labels' (such as mass, position, velocity, etc) and the edges contain tuneable weights which allow for the adjustment of 'interaction strengths' between vertices. Additionally, one can make the graph fully connected or sparse (i.e. not fully connected). For symbolic regression applications there has been success using something called a *graph network* (which is a type of graph neural network). The idea is to try and make your graph network mimic the physical system you are trying to model as much as possible. For example, in the n–body problem we may choose n vertices. There are three models in the graph neural network: 1. the edge model, 2. the node model, and 3. the global model.

1. Compute the edge messages. We have the edge function

$$\phi^e(v_1, v_2) = e_k \tag{26}$$

   Where e_k is termed an edge message which will just be a vector of scalars. The edge function ϕ^e, which should represent the force between the two connecting vertices, is approximated by a neural network. The output of the edge model gets passed to the node model.

2. Compute the vertex messages i.e. vertex outputs. We have the vertex function

$$\vec{a} = \nu(e_1, \cdots, e_{n-1}) \approx \nu\left(\sum_i^{n-1} e_i\right) \tag{27}$$

   where we have approximated the edge dependence as a sum over all edges. The output is an updated node: a vector representing some property or dynamical update (to the kinematics). Now, we also approximate the edge function ν with a neural network.

3. Compute the loss function. The loss function compares the output of each vertex i.e. the acceleration to the actual acceleration at each time step. Each operation up to now was differentiable.

4. Approximate with symbolic regression. Each neural network represents a symbolic expression. Because we have two neural networks within the graph we get two symbolic expressions. The symbolic regression is literally an iterated construct-compare. What is simpler now is that we only need to find symbolic representations of nodes and edges rather than the whole system.

At the end of the day, the improved performance of the symbolic regression can be attributed to two facts: 1. the graph neural net model provides as an easier 'crutch' for the symbolic regression tools to focus on 2. the inductive biases (assumptions) within the graph neural networks significantly constrain the search space of the regression model.

This leaves a caveat: how do you implement useful inductive biases on models for which you don't know the basic setup of the problem i.e. the type of force? Perhaps these situations don't arise because we aren't interested in these types of 'unsolvable' problems.

We choose a set of closed–form analytic expressions stochastically and weight each expression based on its simplicity and accuracy. The symbolic expression is chosen, the real numerical constants are fit to the data, this is repeated and the symbolic expressions are chosen based on the best performance. The way in which this iterates depends on the model.

12 Common ML lingo

Batch learning: System is incapable of learning incrementally, the system must be trained using all of the data at one time. If you want the system to know about new data, it must be trained from scratch. Training a full set of data can take many hours.

Online learning (incremental learning): Train the system incrementally by feeding it data instances sequentially either individually or in small groups called *mini-batches*. This type of system is good for problems which receive data in a continuous flow such as stock market models. Once the system has learned from a set of data, this data can be discarded. This type of learning is also useful for models which need to train on datasets that are too large to fit on one machines main memory (out-of-core learning).

An important parameter to consider in online learning is the rate at which you incorporate new data into your system i.e. the *learning rate*. High learning rates

adapt to new data quickly but forget about old data. Lower learning rates will learn more slowly but be less sensitive to noise in the new data/outliers in the data. The downside to online learning is that the system is susceptible to performance sinks given bad data.

Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. Choosing appropriate features is termed *feature engineering* and involves:

1. Feature selection: selecting the most useful features to train on among existing features.
2. Feature extraction: combining existing feature to produce a more useful one (dimensional reduction i.e. merging features)
3. Creating new features by collecting data.

Overfitting: Model performs well on the training data but does not generalize well to new data. This can occur if the training dataset is noisy or too small (sampling noise). More simply overfitting occurs when the model is too complex relative to the amount and noisiness of the training data. To fix this, one can simplify the model with fewer training parameters, gather more training data, or reduce the noise in the training data.

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*. The amount of regularization to apply during learning can be controlled by a hyper-parameter (parameter of learning algorithm not of the model).

Just as we can overfit our training data, we can also underfit the training data indicating that our model is too simple to learn the underlying structure of the data. To fix underfitting we can

1. Select a more powerful model with more parameters
2. Feed better features to the learning algorithm (feature engineering)
3. Reduce the constraints on the model

Unsupervised learning: In unsupervised learning, the training data is unlabeled and the system must learn without a teacher i.e. we have the input features $\mathbf{X}$, but we do not have the labels $\mathbf{y}$. This is especially nice if your dataset very large and frequently changing (changing labels). Relabeling the training data each time would be a tedious and costly task. This is where unsupervised learning comes into play.

Clustering: Clustering is the task of identifying similar instances and assigning them to *clusters*, i.e. groups of similar instances. There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a *centroid*. Other algorithms look for continuous regions of densely packed instances which can take any shape. Some algorithms are hierarchical and search for clusters of clusters. The list goes on. We'll look at two popular clustering algorithms: K-Means and DBSCAN.

K-Means: In short, given a dataset, the K-Means algorithm will try to find the centers of any densely packed region in the dataset and then will assign each instance of the dataset to the closest blob. Once we have the centroids, breaking the dataset into clusters by assigning each instance to its closest centroid is not very hard. However, if we aren't given the positions of the centroids how do we proceed? We could proceed by placing the centroids randomly (choosing k instances and using these as the centroids), labeling all instances (i.e. which cluster it has been assigned to), updating the centroids to a new instance within the newly formed clusters, rinse and repeat over some number of iterations and the algorithm will converge. Unfortunately, although the algorithm is guaranteed to converge, it may not converge to the right solution: this depends on the centroid initialization. One solution is to run the above clustering algorithm multiple times and keep the best solution. This can be controlled by a hyperparameter n denoting the number of times to run the algorithm. The best solution is determined via a performance metric called the models *inertia* corresponding to the mean squared distance between each instance and its closest centroid. The model with the *lowest* inertia is kept.

The K-Means++ algorithm is an improved K-Means algorithm which utilizes a smarter initialization step that tends to select centroids that are distant from one another, and thus makes it much less likely that the K-Means algorithm will converge to a non-optimal solution. This makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution. The initalization algorithm goes something like:

1. Choose one centroid $\boldsymbol{c}^{(1)}$, chosen uniformily at random from the dataset.
2. Take a new centroid $\boldsymbol{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability: $D(\mathbf{x}^{(i)})^2 \sum_{j=1}^{m} D(\mathbf{x}^{(j)})^2$ where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen. The probability distribution ensures that instances further away from the already chosen centroids are more probable.

3. Repeat the previous step until all k centroids have been chosen.

Two other ways to speed up the algorithm can be achieved by utilizing the triangle inequality and by keeping track of lower and upper bounds for distances between instances and centroids thereby avoiding many unnecessary distance calculations. We can also use mini-batches, moving the centroids just slightly at each iteration (this speeds up the algorithm by a factor of 3 or 4).

We can also go on to methods for choosing the optimal number of clusters using the *silhouette coefficient* but I will leave that for another time.

Limits of K-Means:

The K-Means algorithm has many merits however as we have seen there are many inheriant disadvantages. One being that we must run the algorithm multiple times to converge on the best clustering solution. Moreover, K-Means does not behave well when the clusters have varying sizes, densities, or non-spherical shapes.

Clustering can be useful in applications of image segmentation (partitioning a image into multiple segments), data preprocessing and in semi-supervised learning.

References