Solving Differential Equations Using Neural Networks via Random Gradient Free Methods

Patrick Mellady

Motivation

While analytic solutions exist for differential equations of certain kinds, many differential equations must be solved through the implementation of numerical methods. Classically, these numerical methods are systematic and exploit the geometry of the problem and solve the equation using calculus based methods. However, with the advent of modern computing, it is possible to solve differential equations using the framework of neural networks. This current work will explore the ideas presented in [1] and expand upon those ideas using random gradient free minimization techniques, as presented in [2], in order to solve both ordinary and partial differential equations.

The Neural Network

We will use neural networks to find unique solutions to differential equations using a problems associated initial/boundary conditions. In order to implement neural network methods, we will first explore the output of a neural network and see how we can use the output along with the initial/boundary conditions to estimate the solution. Using this estimated solution, we can easily create a loss function which we will minimize via a gradient descent like approach.

Suppose we have some differential equation, $f(y, y', y'', \cdots, y^{(n)}) = h(x)$, with some boundary conditions. We will solve this problem with a single hidden layer neural network. Our network will take as input a point from the domain, and output the corresponding value of y. The neural network can be thought of as a composition of functions that takes a point from $\mathbb{R}^n$ to $\mathbb{R}$, thus, we will denote the neural network as a function $N : \mathbb{R}^n \to \mathbb{R}$. We will define our network with the following function:

$$N(x) = v^T \sigma(u + Wx), \quad u, v \in \mathbb{R}^q, W \in \mathbb{R}^{q \times d}$$

where q is the number of nodes in our hidden layer and d is the dimension of the problem.

With the above function defined, we can begin to use the statement of the problem to create a loss function, which we can minimize to fit the model. Let our differential equation be as above with boundary conditions $y^{(k)}(0) = h_k(x), \quad k = 0, 1, 2, \cdots, n-1$ where the functions h_k are known. We can then create the fitted values of the model by performing some linear combination of the initial/boundary conditions with $N(X)$, call this $\hat{Y}$. We can differentiate the $\hat{Y}$ to find the necessary fitted derivatives to approximate the differential equation we wish to solve.

Using the fitted values for the function and its derivatives to estimate the differential equation, we can create the loss function $E = \sum_{i=1}^{n}(f(\hat{y}_i, \hat{y}_i', \hat{y}_i'', \cdots, \hat{y}_i^{(n)}) - h(x_i))^2$, where $\hat{y}_i^{(k)}$ is the k^{th} estimated derivative evaluated at x_i. The problem now simply becomes finding the values of the parameters of the network that minimize E.

Random Gradient-Free Methods

The fundamental problem now becomes taking derivatives of the output of the neural network. Techniques such as backpropogation and stochastic gradient descent are popular methods for quickly and accurately calculating these derivatives, however, this work will explore the implementation of random gradient free methods as an interesting exercise. Since the problem of minimizing the squared loss function is convex, we may implement random gradient free methods for minimization as described in [2]. Random gradient free methods make use of the directional derivative, which we define below.

Definition Directional Derivative

Given some scalar function $f : \mathbb{R}^n \to \mathbb{R}$ and a vector $v \in \mathbb{R}^n$, we define the directional derivative of f in the direction v to be

$$f'(x, v) = \lim_{h \to 0} \frac{f(x+hv) - f(x)}{h}$$

With the foundation of the directional derivative, we introduce random gradient free methods, which are built upon the implementation of a random oracle of the form

$$g_{\mu_k}(x_k) = \frac{f(x_k + \mu_k u) - f(x_k)}{\mu_k} \cdot u$$

where $\mu_k \in \mathbb{R}$ and $u \in \mathbb{R}^n$ is a random vector. Note that this preliminary definition of the random oracle closely mimics a numerical directional derivative of the objective function. Three versions of this oracle are presented in [2]:

1. $g_\mu(x_k) = \frac{f(x_k+\mu_k u) - f(x_k)}{\mu_k} \cdot u$
2. $\hat{g}_\mu(x_k) = \frac{f(x_k+\mu_k u) - f(x_k - \mu_k u)}{2\mu_k} \cdot u$
3. $g_0(x_k) = \lim_{\mu_k \to 0} \frac{f(x_k+\mu_k u) - f(x_k)}{\mu_k} \cdot u = f'(x_k, u) \cdot u$

The main difference between each of these methods and the true definition as presented above is that the true directional derivative is a scalar, while these oracles are in $\mathbb{R}^n$. This modification tells the updating algorithm, described in the next section, how far in each random direction to travel as a multiple of the directional derivative.

Solving the Neural Network With Random Gradient Free Methods

In a single layer neural network, we can easily run the chain rule when taking the derivative of $N(x)$ with respect to the input. However, the implementation of the derivative of the loss function becomes much more difficult, so this is where we will use random gradient free methods. Recall from the definition of our network that $N(x) = v^T\sigma(z) = v^T\sigma(u + Wx)$, thus we must optimize with respect to three parameters: v, u, W. This implies that the loss function is a function of four inputs: x, u, v, W, so we will write the loss function as $E(x, u, v, W) = \sum_{i=1}^{n}(f(\hat{y}_i, \hat{y}_i', \hat{y}_i'', \cdots, \hat{y}_i^{(n)}) - h(x_i))^2$. Now, using the random oracle above, we compute

$$g^1_{\mu_k} = \frac{E(x,u_k+\mu_k u,v_k,W_k)-E(x,u_k,v_k,W_k)}{\mu_k}u$$
$$g^2_{\mu_k} = \frac{E(x,u_k,u_k+\mu_k v,W_k)-E(x,u_k,v_k,W_k)}{\mu_k}v$$
$$g^3_{\mu_k} = \frac{E(x,u_k,v_k,W_k+\mu_k)-E(x,u_k,v_k,W_k)}{\mu_k}W$$

Using the three oracles above, we have the following algorithm:

- Initialize u_0, v_0, W_0
- While$(E(x,u_k,v_k,W_k) > \epsilon)$:
  - Calculate $g^1_{\mu_k}$, $g^2_{\mu_k}$, $g^3_{\mu_k}$
  - $u_k = u_{k-1} + \alpha g^1_{\mu_k}$
  - $v_k = v_{k-1} + \alpha g^2_{\mu_k}$
  - $W_k = W_{k-1} + \alpha g^3_{\mu_k}$

Choice of Hyperparameters μ_k and α

The above methods require certain tuning parameters, so arises the question of intelligently picking their values. Choosing the the updating step step-size is simple: since the squared error loss function is convex, we can use the backtracking line search method to find the optimal value of α. The optimal choice of μ_k is more tricky, and the following convergence result, presented in [2], doubles as a method to choose the step size parameter, μ_k.

Theorem

Let $f : \mathbb{R}^n \to \mathbb{R}$ be convex and have Lipschitz continuous gradient with Lipschitz constant L_1 then for $\mu_k = \frac{1}{4(n+4)L_1}$ and $N \geq 0$, we have:

$$\frac{1}{N+1}\sum_{k=0}^{N}(\phi_k - f^*) \leq \frac{4(n+4)L_1||x_0-x^*||^2}{N+1} + \frac{9\mu^2(n+4)^2L_1}{25}$$

Where ϕ_k is the our estimate from the random gradient method and f^* is the true minimized value.

The above theorem tells us a bound on the mean squared error. In practice, we need to choose μ_k sufficiently small in order to generate a solution within an acceptable predetermined accuracy of the true solution. By choosing $\mu_k \leq \frac{5}{3(n+4)}\sqrt{\frac{\epsilon}{2L_1}}$, the error bound described above gives:

$$\frac{9\mu_k^2(n+4)^2L_1}{25} \leq \frac{9(\frac{5}{3(n+4)}\sqrt{\frac{\epsilon}{2L_1}})^2(n+4)^2L_1}{25} = \frac{9\frac{25}{9(n+4)^2}\frac{\epsilon}{2L_1}(n+4)^2L_1}{25} = \frac{\epsilon}{2}$$

So, as $N \to \infty$, we can select the value of μ_k as described above and the prediction error will be bounded by some pre-specified value. Thus, we have theoretical justification for using the random gradient free method as a minimization technique in our neural network problem, as it does converge to the appropriate minimum.

Although this result is theoretically helpful and shows the convergence of the algorithm if certain values are known, we do not necessarily know all the quantities needed for choosing the optimal value of μ_k. For this reason, in the implementation that follows, we will use arbitrarily chosen values for μ_k that seem to produce relatively good results, and use the theorem above as justification for the method as a whole.

Implementation and Numerical Results

To explore the neural network via random gradient free methods for solving differential equations, we will solve three problems: $^{1)}$ $y' + y = e^x$, $^{2)}$ $y'' + 2y' + y = 2xe^{-x}$, and $^{3)}$ $y_x + yy_t = 0$. In general, we will have the following network structure:

- The activation function, $\sigma(\cdot)$, will be the sigmoid function $\frac{1}{1+e^{-x}}$.
- For univariate problems: $N^{(k)}(x) = \sum_{i=1}^{q} v_i w_i^k \sigma^{(k)}(u_i + w_i x)$
- For multivariate problems: $\frac{\partial^K N}{\partial x_1^{k_1} \cdots \partial x_d^{k_d}} = \sum_{i=1}^{q} v_i w_{i1}^{k_1} \cdots w_{id}^{k_d} \sigma^{(K)}(u_i + \sum_{j=1}^{d} w_{ij} x_j)$

Lastly, the algorithms will be implemented with a warm starting technique. The weights of the model are all initialized to have a value of 1, and the first 150-300 iterations (depending on the problem) of the descent will be calculated with a random search. The parameters will move along any direction that decreases the loss function in pre-specified increments until the loss function stops decreasing. After the warm start iterations, we minimize with the methods presented in [2].

It is important to note that using a purely random search would not work for finding quality solutions in any feasible amount of time. The dimension of the problem is too large for a purely random search to be effective, in fact, the random search proved to be useless beyond the warm start iterations as early as iteration 100. Thus, the methods presented in [2] are necessary.

Problem 1

Consider the problem $y' + y = e^x$ with boundary condition $y = 0$ at $x = 0$. This gives us $\hat{y} = xN(x)$ which implies $\hat{y}' = N(x) + xN'(x)$ and our loss function becomes $\sum_{i=1}^{n}((x_i+1)N(x_i) + x_i N'(x_i) - e^{x_i})^2$. We solve the differential equation by the method developed above and compare the solution with the analytic one to verify the method. The analytic solution to the problem is $y = \frac{-1}{2}e^{-x} + \frac{1}{2}e^x$.

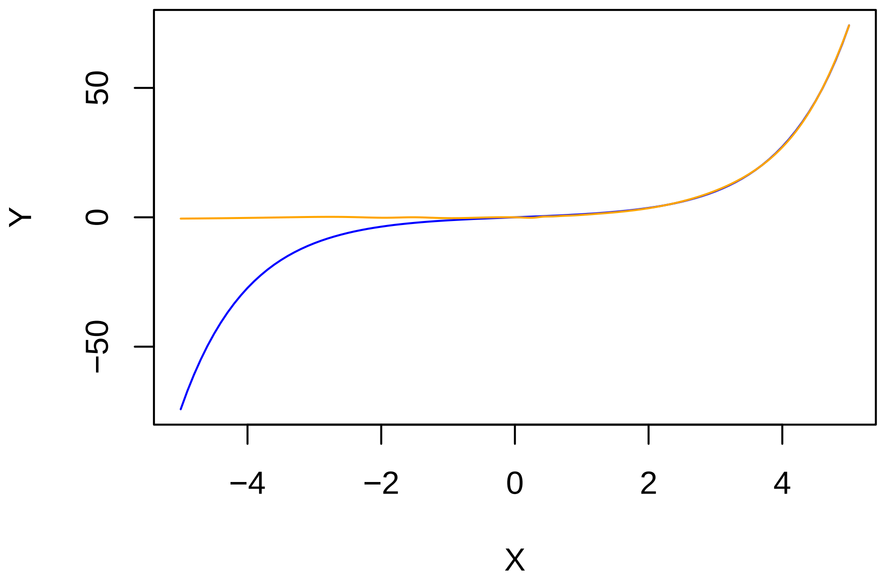
Using the method described above, we obtain weights of:

```
## U -9.6 -6.2 -2.9 -2.8 1 -5 7.5 4.7 -1.5 -9.4

## V 23 0.82 -5.9 8 -5.4 5.2 -4 7.3 0.11 1.6

## W 1.8 7.1 7.2 0.84 1.3 15 4.6 3.1 -6.8 15
```

These weights correspond to an error of 491.2705. The plot of the estimated solution is given below in orange while the analytic solution is in blue.



Problem 2

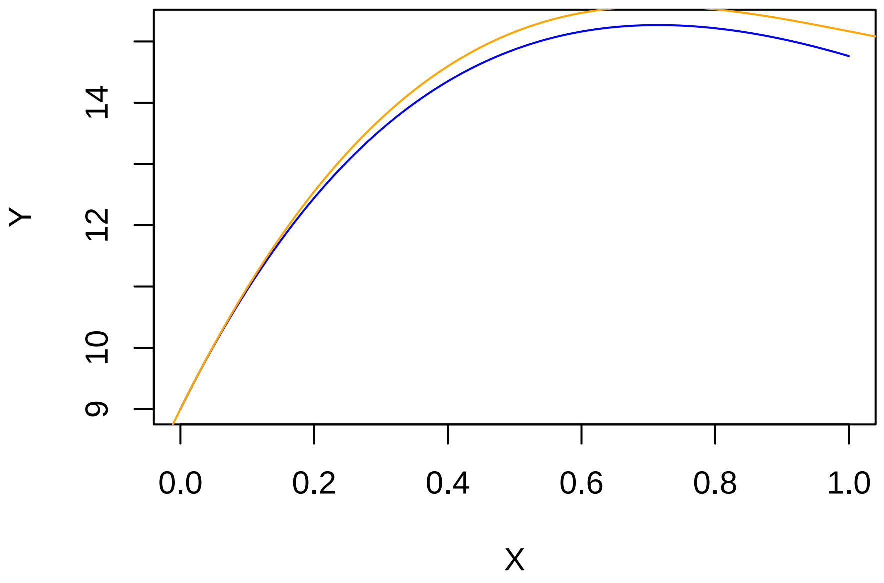
Consider the problem $y'' + 2y' + y = 2xe^{-x}$ with boundary conditions $y = 9$ at $x = 0$ and $y' = 22$ at $x = 0$. This gives us $\hat{y} = x^2N(x) + 22x + 9$, $\hat{y}' = 2xN(x) + x^2N'(x) + 22$, and $\hat{y}'' = 2N(x) + 4xN'(x) + x^2N''(x)$. Our loss function becomes $\sum_{i=1}^{n}((x^2 + 4x + 2)N(x) + (2x^2 + 4x)N'(x) + x^2N''(x) + 22x - 2xe^{-x} + 53)^2$. We again solve the differential equation by the method developed above and compare the solutions. The analytic solution to the problem is $y = 9e^{-x} + 31xe^{x} + \frac{x^3}{8}e^{-x}$. The weights found via the above method are

```
## U 28 4.7 86 -120 16 -68 -150 190 19 -1.8
```

```
## V 36 140 -180 -140 50 68 130 -210 130 110
```

```
## W 150 66 -64 10 32 -100 -110 -35 210 0.41
```

These weights correspond to an error of 10005.78. The plot of the estimated solution is given below in orange while the analytic solution is in blue.



Problem 3

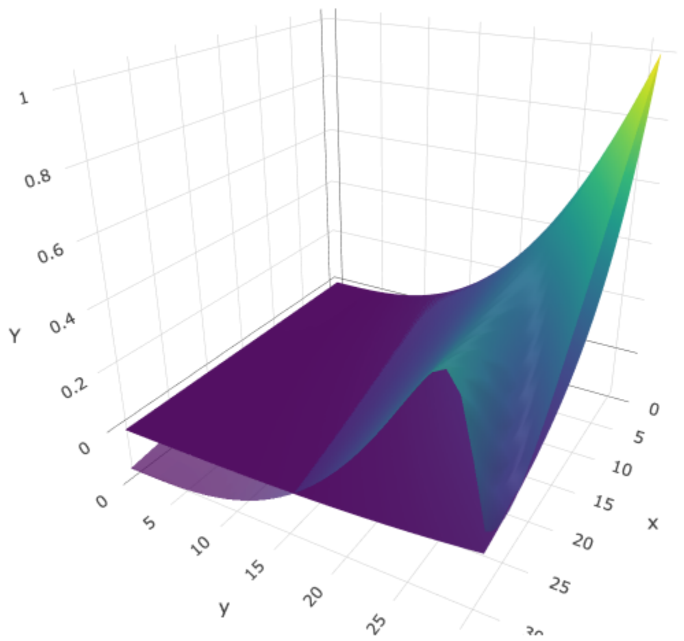
Consider the problem $y_x + yy_t = 0$ with the auxiliary condition $y(0,t) = t^3$. This gives us $\hat{y} = xN(x,t) + t^3$, $\hat{y}_x = N_x(x,t) + xN_x(x,t)$, $\hat{y}_t = xN_t(x,t) + 3t^2$. Our loss function becomes $\sum_{i=1}^{n}\sum_{j=1}^{n}(N_x(x,t) + xN_x(x,t) + (xN(x,t) + t^3)(xN_t(x,t) + 3t^2))^2$. We solve the differential equation by the method developed above and compare the solutions one last time. The analytic solution to the problem is $y(x,t) = e^{-3x}y^3$. The weights found via the above method are

```
## U 13 -94 45 5.6 -25 -28 78 -20 -3.9 43
```

```
## V 0.22 -96 -11 5 6.3 48 14 -25 9.5 -8.8
```

```
## W 47 -21 91 7.7 -41 -21 10 -37 -18 -11 -66 13 -38 18 -8.1 -55 45 -10 -29 3.5
```

These correspond to an error of 72.72. The plot of the estimated and analytic solutions are given below.



Discussion

The methods used above proved to be effective at solving both ordinary and partial differential equations. However, the solutions were not as good as they could be. One way to make the solutions more accurate would be longer computing time. When the algorithms terminated, they were still actively decreasing the loss function, but time constraints on the project required them to stop where they did. Additionally, the effects more complex network structure could impact the rate at which the loss function can be decreased, which would make for an interesting future project.

Further, it was revealed that, by varying the choice of the tuning parameters within the problem, the convergence rate could be adjusted based on the current state of the solution. This suggests that there may be some adaptive selection for the tuning parameters when the theoretically optimal parameters are unknown. Comparing the convergence rate of the above methods to methods that employ stochastic gradient descent and backpropogation would be quite interesting and is currently a work in progress. The code for the problems above (excluding plotting) are attached following the references.

References

1. Rostamian, Rouben. "Neural Networks for Solving ODES." Programming Projects in C for Students of Engineering, Science, and Mathematics , SIAM, Soc. for Indust. and Appl. Math., 2014, pp. 375–390.

2. Nesterov, Y., Spokoiny, V. *Random Gradient-Free Minimization of Convex Functions. Found Comput Math 17, 527–566 (2017).* https://doi.org/10.1007/s10208-015-9296-2

Code

Problem 1

```
################################################################################
########## Neural Network ODE Solver for the equation y'+y=exp(x) ##############
################################################################################

# Setting Parameters
n<-1000                                      # the number of mesh points in the interval
q<-10                                       # number of neurons (one layer)
X<-matrix(seq(-5,5,length.out=n), ncol=1)    # making the grid of mesh points

# Initialize weights on the nodes
U<-matrix(rep(1, q), ncol=1)
V<-matrix(rep(1, q), ncol=1)
W<-matrix(rep(1,q), ncol=1)

# Components of the network ####################################################
# Sigmoid function for the activation of the neurons
s<-function(X){
  n<-length(X)
  res<-c()
  for(i in 1:n){
    temp<-1/(1+exp(-X[i]))
    res<-c(res, temp)
  }
  return(res)
}

# Derivative of the sigmoid function
# (used in derivatives of the estimated solution)
s_prime<-function(X){
  n<-length(X)
  res<-c()
  for(i in 1:n){
    temp<-exp(-X[i])/(1+exp(-X[i]))^2
    res<-c(res,temp)
  }
  return(matrix(res, ncol=1))
}

# Non-unique portion of the estimated solution ##################################
# Defining the non-unique portion of the estimated solution
# (does not include the boundary conditions)
N<-function(x, U, V, W){
  S<-s(U+W*x)
  N_x<-t(V)%*%S
  return(N_x)

}

# Defining the derivative of the non-unique portion of the estimated solution
# (does not include the boundary conditions)
N_prime<-function(x, U, V, W){
  N_prime_x<-t(W)%*%(V*s_prime(U+W*x))
  return(N_prime_x)
}

# Changing the problem to an optimization problem ##############################
# Defining the error function that we want to minimize (includes the boundary conditions)
E<-function(X, U, V, W){
  n<-length(X)
  temp<-c()
  for(i in 1:n){
    x<-X[i]
    temp<-c(temp, ((x+1)*N(x, U, V, W)+x*N_prime(x, U, V, W)-exp(x))^2)
  }
  return(sum(temp))
}


# Components of the gradient-free method ########################################
# Calculating the numerical directional derivative for the optimization method
g_mu<-function(x, U_k, V_k, W_k, du, dv, dw, mu){
  gmu_u<-(1/mu)*(E(x, U_k+mu*du, V_k, W_k)-E(x, U_k, V_k, W_k))[[1]]*du
  gmu_v<-(1/mu)*(E(x, U_k, V_k+mu*dv, W_k)-E(x, U_k, V_k, W_k))[[1]]*dv
  gmu_w<-(1/mu)*(E(x, U_k, V_k, W_k+mu*dw)-E(x, U_k, V_k, W_k))[[1]]*dw
  return(list(U_k=gmu_u, V_k=gmu_v, W_k=gmu_w))
}

# Back tracking line search to find optimal step size
back_track<-function(X, U_k, V_k, W_k, g_k, rho=0.3, c=0.005, h=4.05e-6){
  t<-0.005
  while(E(X, U_k-h*g_k[[1]], V_k-h*g_k[[2]], W_k-h*g_k[[3]])-E(X, U_k, V_k, W_k)>=c*t){
    h<-h*rho
```

```r
  }
  return(h)
}


# Implementation of the neural network #########################################
sol_warm_start<-function(X, U0, V0, W0, K=1000, epsilon=0.8, mu=4.05e-8,
                         h=4.05e-8, warm=T, warm_iter=150){
  n<-length(X)
  i<-0
  if(warm){
    u<-U0
    v<-V0
    w<-W0
    for (i in 1:warm_iter) {
      p<-0.1
      variance <- .2
      stepu<-p*matrix(rnorm(q, 0, variance), ncol=1)
      stepv<-p*matrix(rnorm(q, 0, variance), ncol=1)
      stepw<-p*matrix(rnorm(q, 0, variance), ncol=1)

      unew <- u + stepu
      wnew <- w + stepw
      vnew <- v + stepv
      counter <- 0

      unew1 <- unew
      wnew1 <- wnew
      vnew1 <- vnew

      while(E(X, unew1, vnew1, wnew1) < E(X, u, v, w)) {
        unew1 <- unew1 + stepu
        wnew1 <- wnew1 + stepw
        vnew1 <- vnew1 + stepv
        counter <- counter + 1
      }
      unew <- unew1 - stepu
      wnew <- wnew1 - stepw
      vnew <- vnew1 - stepv
      if (E(X, unew, vnew, wnew) < E(X, u, v, w)) {
        u <- unew
        v <- vnew
        w <- wnew
      }
      cat("Level ", counter, " finna garkage!!", E(X, u, v, w), " \n")
      print(i)
    }
    U0<-u
    V0<-v
    W0<-w
  }
  error<-E(X, U0, V0, W0)
  k<-0
  while(error>epsilon){
    du<-matrix(rnorm(q, 0, 1), ncol=1)
    dv<-matrix(rnorm(q, 0, 1), ncol=1)
    dw<-matrix(rnorm(q, 0, 1), ncol=1)
    g_k<-g_mu(X, U0, V0, W0, du, dv, dw, mu)
    h<-back_track(X, U0, V0, W0, g_k)
    U_k<-U0-h*g_k[[1]]
    V_k<-V0-h*g_k[[2]]
    W_k<-W0-h*g_k[[3]]
    U0<-U_k
    V0<-V_k
    W0<-W_k
    error<-E(X, U_k, V_k, W_k)
    k<-k+1
    if(k%%25==0){
      cat(k, " Nesterov garkages with a finna garkage of ", error,
          " and a stepping gark of ", h ,"\n")
    }
    if(k>5000){
      break
    }
  }
  return(list(U=U_k, V=V_k, W=W_k))
}
```

Problem 2

```
################################################################################
########## Neural Network ODE Solver for the equation y''+2y'+y=2e^(-x) ########
################################################################################

# Setting Parameters
n<-1000                                   # the number of mesh points in the interval
q<-10                                     # number of neurons (one layer)
X<-matrix(seq(0,1,length.out=n), ncol=1)    # making the grid of mesh points

# Initialize weights on the nodes
U<-matrix(rep(1, q), ncol=1)
V<-matrix(rep(1, q), ncol=1)
W<-matrix(rep(1,q), ncol=1)
```

```r
# Components of the network ####################################################
# Sigmoid function for the activation of the neurons
s<-function(X){
  n<-length(X)
  res<-c()
  for(i in 1:n){
    temp<-1/(1+exp(-X[i]))
    res<-c(res, temp)
  }
  return(res)
}

# Derivative of the sigmoid function
# (used in derivatives of the estimated solution)
s_prime<-function(X){
  n<-length(X)
  res<-c()
  for(i in 1:n){
    temp<-exp(-X[i])/(1+exp(-X[i]))^2
    res<-c(res,temp)
  }
  return(matrix(res, ncol=1))
}

s_dprime<-function(X){
  n<-length(X)
  res<-c()
  for(i in 1:n){
    temp1<--exp(-X[i])/(1+exp(-X[i]))^2
    temp2<-2*exp(-2*X[i])/(1+exp(-X[i]))^3
    temp<-temp1+temp2
    res<-c(res,temp)
  }
  return(matrix(res, ncol=1))
}

# Non-unique portion of the estimated solution ##################################
# Defining the non-unique portion of the estimated solution
# (does not include the boundary conditions)
N<-function(x, U, V, W){
  S<-s(U+W*x)
  N_x<-t(V)%*%S
  return(N_x)

}

# Defining the derivative of the non-unique portion of the estimated solution
# (does not include the boundary conditions)
N_prime<-function(x, U, V, W){
  N_prime_x<-t(W)%*%(V*s_prime(U+W*x))
  return(N_prime_x)
}

N_dprime<-function(x, U, V, W){
  N_dprime_x<-sum(V*W^2*s_dprime(U+W*x))
  return(N_dprime_x)
}

# Changing the problem to an optimization problem ##############################
# Defining the error function that we want to minimize (includes the boundary conditions)
E<-function(X, U, V, W){
  n<-length(X)
  temp<-c()
  for(i in 1:n){
    x<-X[i]
    temp<-c(temp, ((x^2+4*x+2)*N(x, U, V, W)+(2*x^2+4*x)*N_prime(x, U, V, W)+
                     x^2*N_dprime(x, U, V, W)-2*x*exp(x)+22*x+53)^2)
  }
  return(sum(temp))
}


# Components of the gradient-free method ########################################
# Calculating the numerical directional derivative for the optimization method
g_mu<-function(x, U_k, V_k, W_k, du, dv, dw, mu){
  gmu_u<-(1/mu)*(E(x, U_k+mu*du, V_k, W_k)-E(x, U_k, V_k, W_k))[[1]]*du
  gmu_v<-(1/mu)*(E(x, U_k, V_k+mu*dv, W_k)-E(x, U_k, V_k, W_k))[[1]]*dv
  gmu_w<-(1/mu)*(E(x, U_k, V_k, W_k+mu*dw)-E(x, U_k, V_k, W_k))[[1]]*dw
  return(list(U_k=gmu_u, V_k=gmu_v, W_k=gmu_w))
}

# Back tracking line search to find optimal step size
back_track<-function(X, U_k, V_k, W_k, g_k, rho=0.3, c=0.005, h=4.05e-6){
  t<-0.005
  while(E(X, U_k-h*g_k[[1]], V_k-h*g_k[[2]], W_k-h*g_k[[3]])-E(X, U_k, V_k, W_k)>=c*t){
    h<-h*rho
  }
  return(h)
}

sol_warm_start<-function(X, U0, V0, W0, K=1000, epsilon=0.8, mu=4.05e-8,
                         h=4.05e-8, warm=T, warm_iter=150){
  n<-length(X)
  i<-0
```

```r
if(warm){
  u<-U0
  v<-V0
  w<-W0
  for (i in 1:warm_iter) {
    p<-0.1
    variance <- .2
    stepu<-p*matrix(rnorm(q, 0, variance), ncol=1)
    stepv<-p*matrix(rnorm(q, 0, variance), ncol=1)
    stepw<-p*matrix(rnorm(q, 0, variance), ncol=1)

    unew <- u + stepu
    wnew <- w + stepw
    vnew <- v + stepv
    counter <- 0

    unew1 <- unew
    wnew1 <- wnew
    vnew1 <- vnew

    while(E(X, unew1, vnew1, wnew1) < E(X, u, v, w)) {
      unew1 <- unew1 + stepu
      wnew1 <- wnew1 + stepw
      vnew1 <- vnew1 + stepv
      counter <- counter + 1
    }
    unew <- unew1 - stepu
    wnew <- wnew1 - stepw
    vnew <- vnew1 - stepv
    if (E(X, unew, vnew, wnew) < E(X, u, v, w)) {
      u <- unew
      v <- vnew
      w <- wnew
    }
    cat("Level ", counter, " finna garkage!!", E(X, u, v, w), " \n")
    print(i)
  }
  U0<-u
  V0<-v
  W0<-w
}
error<-E(X, U0, V0, W0)
k<-0
while(error>epsilon){
  du<-matrix(rnorm(q, 0, 1), ncol=1)
  dv<-matrix(rnorm(q, 0, 1), ncol=1)
  dw<-matrix(rnorm(q, 0, 1), ncol=1)
```

```
    g_k<-g_mu(X, U0, V0, W0, du, dv, dw, mu)
    h<-back_track(X, U0, V0, W0, g_k)
    U_k<-U0-h*g_k[[1]]
    V_k<-V0-h*g_k[[2]]
    W_k<-W0-h*g_k[[3]]
    U0<-U_k
    V0<-V_k
    W0<-W_k
    error<-E(X, U_k, V_k, W_k)
    k<-k+1
    if(k%%25==0){
      cat(k, " Nesterov garkages with a finna garkage of ", error,
          " and a stepping gark of ", h ,"\n")
    }
    if(k>5000){
      break
    }
  }
  return(list(U=U_k, V=V_k, W=W_k))
}
```

Problem 3

```
################################################################################
########## Neural Network PDE Solver for Burgers' Equation #####################
################################################################################

# Setting Parameters
n<-32                                       # n^2 is the number of meshpoints in the region
d<-2                                         # the number of dimensions of the input
q<-10                                         # the number of neurons in the hidden layer

# Initialize weights on the nodes
U<-matrix(rep(1, q), ncol=1)
V<-matrix(rep(1, q), ncol=1)
W<-matrix(rep(1,q*d), ncol=d)

# Components of the network ####################################################
# Sigmoid function for the activation of the neurons
s<-function(x){
  res<-1/(1+exp(-x))
  return(res)
}

# Derivative of the sigmoid function
# (used in derivatives of the estimated solution)
s_prime<-function(x){
  res<-exp(-x)/(1+exp(-x))^2
  return(res)
}

# Second derivative of the sigmoid function
# (used in derivatives of the estimated solution)
s_dprime<-function(X){
  n<-length(X)
  res<-c()
  for(i in 1:n){
    temp1<--exp(-X[i])/(1+exp(-X[i]))^2
    temp2<-2*exp(-2*X[i])/(1+exp(-X[i]))^3
    temp<-temp1+temp2
    res<-c(res,temp)
  }
  return(matrix(res, ncol=1))
}

N<-function(X, U, V, W){
  N_x<-t(V)%*%s(U+W%*%X)
  return(N_x)

}

# Partials of the non-unique portion of the estimated solution
N_t<-function(X, U, V, W){
  N_t_xt<-sum(V*W[,2]*s_prime(U+W%*%X))
  return(N_t_xt)
}

N_x<-function(X, U, V, W){
  N_x_xt<-sum(V*W[,1]*s_prime(U+W%*%X))
  return(N_x_xt)
}

E<-function(U, V, W){
  grid<-seq(0,1,length.out=n)
  temp<-c()
  for(x in grid){
    for(t in grid){
      X<-matrix(c(x,t))
      temp<-c(temp, (N(X, U, V, W)+x*N_x(X, U, V, W)+
                    (x*N(X, U, V, W)+t^3)*(x*N_t(X, U, V, W)+3*t^2))^2)
    }
  }
  return(sum(temp))
}
```

```r
g_mu<-function(U, V, W, du, dv, dw, mu){
  gmu_u<-(1/mu)*(E(U+mu*du, V, W)-E(U, V, W))[[1]]*du
  gmu_v<-(1/mu)*(E(U, V+mu*dv, W)-E(U, V, W))[[1]]*dv
  gmu_w<-(1/mu)*(E(U, V, W+mu*dw)-E(U, V, W))[[1]]*dw
  return(list(U_k=gmu_u, V_k=gmu_v, W_k=gmu_w))
}

# Back tracking line search to find optimal step size
back_track<-function(U, V, W, g_k, rho=0.3, c=0.005, h=4.05e-6){
  t<-0.005
  while(E(U-h*g_k[[1]], V-h*g_k[[2]], W-h*g_k[[3]])-E(U, V, W)>=c*t){
    h<-h*rho
  }
  return(h)
}

sol_warm_start<-function(U0, V0, W0, K=1000, epsilon=0.8, mu=4.05e-8,
                         h0=4.05e-8, warm=T, warm_iter=300){
  i<-0
  if(warm){
    cat("Warm finnage","\n")
    u<-U0
    v<-V0
    w<-W0
    for (i in 1:warm_iter) {
      p<-.4
      variance <- .2
      stepu<-p*matrix(rnorm(q, 0, variance), ncol=1)
      stepv<-p*matrix(rnorm(q, 0, variance), ncol=1)
      stepw<-p*matrix(rnorm(d*q, 0, variance), ncol=d)

      unew <- u + stepu
      wnew <- w + stepw
      vnew <- v + stepv
      counter <- 0

      unew1 <- unew
      wnew1 <- wnew
      vnew1 <- vnew

      while(E(unew1, vnew1, wnew1) < E(u, v, w)) {
        unew1 <- unew1 + stepu
        wnew1 <- wnew1 + stepw
        vnew1 <- vnew1 + stepv
        counter <- counter + 1
      }
      unew <- unew1 - stepu
      wnew <- wnew1 - stepw
      vnew <- vnew1 - stepv
      if (E(unew, vnew, wnew) < E(u, v, w)) {
        u <- unew
        v <- vnew
        w <- wnew
      }
      cat("Level ", counter, " finna garkage!!", E(u, v, w), " \n")
      print(i)
    }
    U0<-u
    V0<-v
    W0<-w
  }
  k<-0
  error<-E(U0, V0, W0)
  cat("0 garkages with a finna garkage of ", error,"\n")
  while(error>epsilon){
    du<-matrix(rnorm(q, 0, 1), ncol=1)
    dv<-matrix(rnorm(q, 0, 1), ncol=1)
    dw<-matrix(rnorm(d*q, 0, 1), ncol=d)
    g_k<-g_mu(U0, V0, W0, du, dv, dw, mu)
    h<-back_track(U0, V0, W0, g_k, h=h0)
    U_k<-U0-h*g_k[[1]]
    V_k<-V0-h*g_k[[2]]
    W_k<-W0-h*g_k[[3]]
    U0<-U_k
    V0<-V_k
    W0<-W_k
    error<-E(U_k, V_k, W_k)
    k<-k+1
    if(k%%25==0){
      cat(k, " Nesterov garkages with a finna garkage of ", error,
          " and a stepping gark of ", h ,"\n")
    }
    if(k>K){
      break
    }
  }
  return(list(U=U0, V=V0, W=W0))
}
```