Resolver problemas de control óptimo con diferenciacion automatica

Jocelyn Diaz Perez
Tutor: Dr. Kernel Moreno Prieto

4 de agosto de 2019

Índice general

Capítulo 1

Introducción

Durante los últimos años la inteligencia artificial, el aprendizaje de máquina y el aprendizaje profundo se han dado a conocer en la comunidad cientifica a traves de articulos, llevando a los investigadores a buscar más sobre estos temas y las aplicaciones que estos tienen. El aprendizaje de máquina es un campo dentro de la inteligencia artificial y el aprendizaje profundo es un subcampo dentro del aprendizaje de máquina [3]. El aprendizaje profundo busca automatizar la inteligencia y poco a poco se a logrado gracias al uso de redes neuronales artificiales (ANN, por sus siglas en inglés) las cuales son un modelo específico del aprendizaje profundo.

Las ANN son capaces de imitar y predecir el comportamiento de sistemas dinámicos, por lo cual pueden reconocer patrones y aprender de ellos, una ANN conoce el error que comete al hacer este proceso. Debido a esto las ANN se han usado principalmente para el reconocimiento de imagenes, minería de texto y procesamiento de voz. Pero no debemos olvidar que las ANN pre alimentadas son funciones de aproximación universales, así que se pueden usar como herramientas de análisis numérico.

La solución de ecuaciones diferenciales ordinarias (ODE) y ecuaciones diferenciales parciales (PDE) es esencial para varios campos en la ingeniería, donde los métodos tradicionales como elementos finitos, diferencias finitas, entre otros dependen de discretizar el dominio resolver débilmente. Por lo general estos métodos son efectivos pero se limitan a soluciones discretas y su diferenciabilidad es limitada, para evitar esta situación en soluciones numéricas se implementan métodos con ANN [6]; pues una característica de las ANN es que son una forma de diferenciar fácil de usar en cualquier cálculo subsecuente, dando modelos compactos de solución con menor demanda de espacio de memoria, así, una de las aplicaciones importantes de las ANN

como herramienta de análisis numérico es que se pueden utilizar para resolver ecuaciones diferenciales, tanto parciales como ordinarias. De los diversos métodos que utilizan ANN para derivar el más exacto es el de diferenciación automática [2].

1.1. Objetivo

El objetivo de este reporte es mostrar que un método que utilice ANN pre alimentadas para resolver ecuaciones diferenciales, particularmente las ecuaciones que modelan problemas de control óptimo.

1.2. Hipótesis

La hipótesis de este trabajo consiste en probar que el método descrito en el artículo [5], que se utiliza para resolver ecuaciones diferenciales ordinarias y parciales usando redes neuronales, se puede aplicar a problemas de control óptimo.

Capítulo 2

Metodología a emplear

2.1. Aprendizaje profundo

El aprendizaje de máquina se encarga de descubrir reglas para ejecutar tareas de procesamiento de datos y para ello necesitamos: datos de entrada, ejemplos o salidas esperadas y una forma de medir si el algoritmo está haciendo un buen trabajo o no. Un modelo de aprendizaje de máquina transforma los datos de entrada en salidas significativas (proceso de aprendizaje), es decir, el aprendizaje de máquina busca las representaciones útiles de los datos de entrada, dentro de un espacio definido de posibilidades, usando la ayuda de una señal pre-alimentada.

El aprendizaje profundo es un subcampo de métodos que pertenece a el aprendizaje de máquina, que se dedica a estudiar y desarrollar máquinas que pueden aprender [5]. El aprendizaje profundo se caracteriza por tener capas sucesivas de representaciones de datos, donde las capas aprenden automáticamente debido a la exposición que tienen con los datos de entrenamiento. Lo que sucede específicamente es que, los datos de entrada son parametrizados por los pesos de las capas. Se buscan los pesos para las capas en la red, de tal forma que la red hace un mapeo para cada entrada con su objetivo asociado.

Debido a la estructura del aprendizaje profundo, por lo general se representan con ANN. Una ANN puede tener millones de parametros, asi que encontrar los valores correctos de cada peso se vuelve una tarea tediosa, además de que cada valor se ve afectado por su valor anterior. Para que esto no ocurra, se usa el error obtenido de la función de pérdida como una función prealimentada para ajustar poco a poco los pesos.

2.2. Redes neuronales artificiales

Una ANN copia el proceso de aprendizaje del cerebro humano con el fin de extraer patrones de datos históricos, durante varios años esta tecnología ha sido aplicada exitosamente en diferentes áreas. Una neurona biológica puede tener 10^4 entradas diferentes y puede enviar la salida a cualquier otra neurona, pero una ANN solo puede usar técnicas computacionales tradicionales. Por ello se utilizan las ANN prea alimentadas para formar un modelo a partir del entrenamiento de los datos (las entradas) y tienen la ventaja de ser funciones de aproximación.

Figura 2.1: Estructura de un perceptrón.

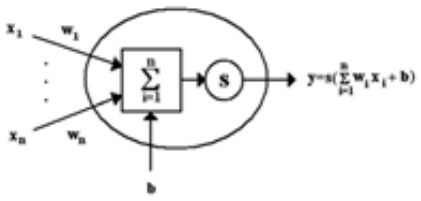
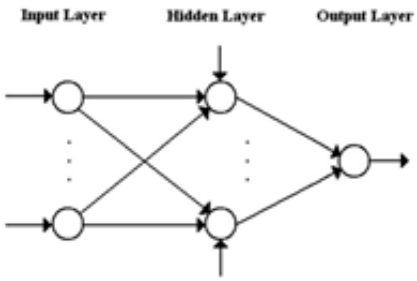


Figura 2.2: Perceptrón multi-capa



Un ejemplo serían los perceptrones, los cuales son ANN supervisadas, es decir la ANN se adapta dependiendo de las entradas, la figura 2.1 nos muestra la estructura de un perceptrón. Los perceptrones multicapa aplica un mapeo no lineal específico para realizar una correspondencia entre las unidades de entrada y las unidades de salida [1], en la figura 2.2 podemos observar como

luce un perceptrón milti-capa.

2.3. Diferenciación automática

Los métodos para diferenciar computacionalmente se pueden clasificar en 4 categorías: 1) Codificar manualmente las derivadas, pero es propenso a errores y consume tiempo. 2)Diferenciación numérica usando aproximadores de diferencia finita, esta categoría es fácil de implementar, pero puede tener errores de redondeo o errores de truncamiento. 3)Diferenciación simbólica usando manipulaciones de expresiones en sistemas algebraicos computacionales. La diferenciación simbólica tiene los problemas de las primeras categorías, además de que los resultados usualmente son complejos o son expresiones encriptadas. 4) Diferenciación automática (AD por sus siglas en inglés). Esta categoría es la mejor de las 4 categorías mencionadas.

AD es una familia de técnicas que hace derivadas computacionales a través de la acumulacion de valores, durante la ejecucion de código para generar evaluaciones de derivadas numéricas [2]. AD está compuesto por dos modalidades hacia adelante y hacia atrás; para explicar cómo funciona AD resolveremos la función $f(x_1, y_1) = ln(x_1) + x_1x_2 - sin(x_2)$, con ambas modalidades.

Figura 2.3: Grafica computacional de la función f(x_1, y_1) = $ln(x_1) + x_1x_2 - sin(x_2)$

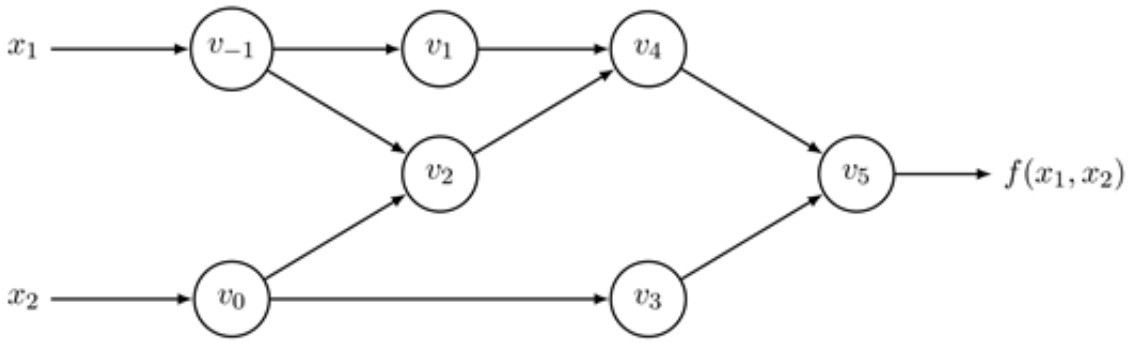
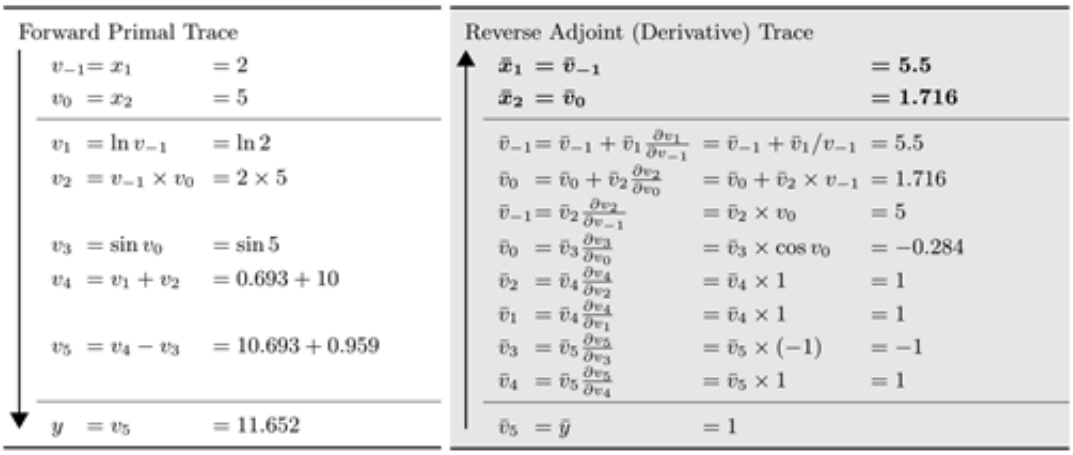


Figura 2.4: Modalidad hacia adelante de la función f(x_1, y_1) = $ln(x_1)+x_1x_2-sin(x_2)$



Forward Primal Trace		
$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$	$= 1$	

2.3.1. Modalidad hacia adelante

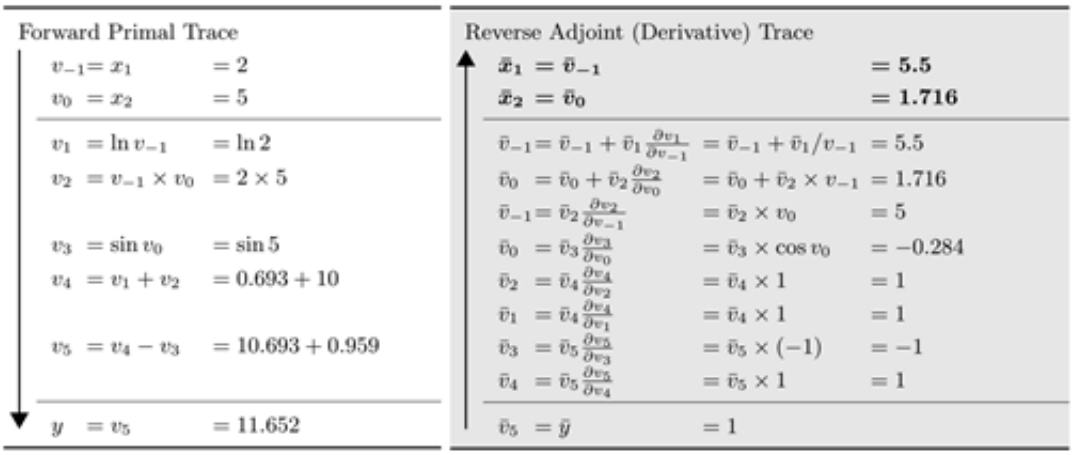
Consideramos la evaluación de la función $f(x_1, y_1) = ln(x_1) + x_1x_2 - sin(x_2)$, dada en el lado izquierdo de la figura 2.4. Para calcular la derivada de f con respecto de x, se empieza asociando que cada variable intermedia v_i con la derivada $\dot{v} = \frac{\partial v_i}{\partial x_1}$. Aplicando regla de la cadena a cada operación elemental en el rastro original hacia adelante, generando el rastro tangente(derivada) correspondiente, dado en el lado derecho de la tabla 2.4. Evaluando las v_i originales en la cerradura con sus correspondientes tangentes v_i, dando la derivada final requerida en la variable final $\dot{v}_5 = \frac{\partial y}{\partial x_1}$.

Esto naturalmente se generaliza para obtener el jacobiano de una función $f : \mathbb{R}^n \to \mathbb{R}^m$, con n variables independientes (entradas) x_i y m variables dependientes (salidas) y_j. En este caso, cada paso hacia adelante del AD es inicializado con el solo ajuste de una de las variables $\dot{x}_i = 1$ y ajustando las demás a 0, es decir, ajustarlas a $\dot{x} = e_i$, cuando e_i es la unidad i-ésima del vector.

2.3.2. Modalidad hacia atrás

Esta modalidad pertenece a la generalización del algoritmo de propagación hacia atrás, en el cual se propagan las derivadas hacia atrás desde una salida dada. Esto se hace al complementar cada variable intermedia v_i con el adjunto $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$, esto representa la sensibilidad de una salida considerada y_j con respecto a los cambios en v_i. AD hacia atrás se compone de dos fases.

Figura 2.5: Modalidad hacia atrás de la función f(x_1, y_1) = $ln(x_1) + x_1x_2 - sin(x_2)$



Forward Primal Trace			Reverse Adjoint (Derivative) Trace		
$v_{-1} = x_1$	$= 2$		$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$v_0 = x_2$	$= 5$		$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$v_1 = \ln v_{-1}$	$= \ln 2$		$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$		$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
			$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$v_3 = \sin v_0$	$= \sin 5$		$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$v_4 = v_1 + v_2$	$= 0.693 + 10$		$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
			$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$		$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
			$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$y = v_5$	$= 11.652$		$\bar{v}_5 = \bar{y}$	$= 1$	

En la primera fase, el código de la función original se corre hacia adelante, llenando las variables intermedias v_i y registrando las dependencias en la gráfica computacional a través de un proceso de teneduría. En la segunda fase, las derivadas se calculan por propagar $\bar{v_i}$ adjuntos hacia atrás desde las salidas hasta las entradas. En la figura 2.5 del lado izquierdo se puede ver las declaraciones adjuntas, correspondiente a cada operación elemental original en el lado derecho. Nos concentramos en calcular la contribución de $\bar{v_i} = \frac{\partial y}{\partial v_i}$ en el cambio de cada variable v_i a el cambio de la salida y.

Por ejemplo si tomamos a v_0, tomando en cuenta la figura 2.3, podemos ver que y se ve afectado por v_2 y v_3, así que el cambio en y está dado por $\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}$, por la notación dada anteriormente esta expresión se puede reescribir como $\bar{v_0} = \bar{v_2}\frac{\partial v_2}{\partial v_0} + \bar{v_3}\frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}$. Esta contribución es realizada en dos pasos incrementales $\bar{v_0} = \bar{v_3}\frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}$ y $\bar{v_0} = \bar{v_0} + \bar{v_2}\frac{\partial v_2}{\partial v_0}$.

Una de las ventajas importantes de la modalidad hacia atrás es que es significativamente menos costosa al evaluar qué modo hacia adelante, debido a que esta tiene numerosas entradas. El entrenamiento de una ANN es un problema de optimización con respecto a su conjunto de pesos, la cual puede ser solucionada con AD hacia atrás ya que aplicando esta modalidad a una función objetivo, evaluando el error de la ANN como una función de sus pesos, se puede realizar el cálculo de las derivadas parciales necesitadas para realizar las actualizaciones de los pesos.

2.4. Resolver ODE y PDE con AD

En el artículo [5] se muestra un método general para resolver PDE y ODE con condiciones iniciales o condiciones de frontera. Se usa una ANN pre alimentada como el elemento de aproximación base y sus parámetros (los pesos y las tendencias) son ajustados para minimizar una función de error y para entrenar la ANN se utilizan tecnicas de optimizacion. A partir de esto se propone la siguiente solución:

$$\Psi_t = A(\vec{x}) + F(\vec{x}, N(\vec{x}, \vec{p})) \tag{2.1}$$

El primer término satisface las condiciones iniciales o de frontera y contiene parámetros no ajustables. El segundo término es una ANN pre alimentada que se encarga de satisfacer la ecuación diferencial, donde $\vec{x}$ es un vector de entrada; $\vec{p}$ son los parámetros ajustables; $N(\vec{x}, \vec{p})$ es una ANN pre alimentada que contiene una sola salida. Para mostrar el método consideramos las siguientes ecuaciones diferenciales:

ODE de primer orden:

$$\frac{d\Psi(x)}{dx} = f(x, \Psi) \tag{2.2}$$

con $x \in [0, 1]$ y condición inicial de $\Psi(0) = A$. Por lo cual la solución de prueba se escribe como:

$$\Psi_t(x) = A + xN(x, \vec{p}) \tag{2.3}$$

Donde $N(\vec{x}, \vec{p})$ es la salida de una ANN pre alimentada con una sola entrada x y los pesos $\vec{p}$. La función a minimizar es:

$$E[\vec{p}] = \sum_i [\frac{d\Psi_t(x_i)}{dx} - f(x_i, \Psi_t(x_i))]^2 \tag{2.4}$$

Para minimizar el error necesitamos derivar nuestra solución de prueba, para ello se tiene el siguiente código del método para una ODE de primer orden:

$def neural\_network(W, x) :$
$a1 = sigmoid(np.dot(x, W[0]))$
$returnnp.dot(a1, W[1])$

$defd\_neural\_network_dx(W, x, k = 1) :$
$returnnp.dot(np.dot(W[1].T, W[0].T * *k), sigmoid_grad(x))$

$defloss\_function(W, x):$
$loss\_sum = 0.$
$forxiinx:$
$net\_out = neural\_network(W, xi)[0][0]$
$psy\_t = 1. + xi * net_out$
$d\_net_out = d\_neural\_network\_dx(W, xi)[0][0]$
$d\_psy_t = net_out + xi * d_net_out$
$func = f(xi, psy_t)$
$err\_sqr = (d_psy_t - func) * *2$
$loss\_sum+ = err\_sqr$
$returnloss\_sum$

Para el proceso de optimización se usa el descenso de gradiente, por lo cual necesitamos derivar la solución con respecto a los pesos, para ello se usara AD, especificamente la función de python Autograd. Esta función hace las derivadas de cualquier orden de funciones particulares y no se mete con la epsilon en el enfoque de diferenciación finita.

$importautograd.numpyasnp$
$fromautogradimportgrad$
$importautograd.numpy.randomasnpr$
$fromautograd.coreimportprimitive$
$W = [npr.randn(1, 10), npr.randn(10, 1)]$
$lmb = 0.001$
$foriinrange(1000):$
$loss_grad = grad(loss_function)(W, x_space)$
$W[0] = W[0] - lmb * loss_grad[0]$
$W[1] = W[1] - lmb * loss_grad[1]$
ODE de segundo orden:

$$\frac{d^2\Psi(x)}{dx^2} = f(x, \Psi, \frac{d\Psi(x)}{dx}) \tag{2.5}$$

con las siguientes condiciones iniciales $\Psi(0) = A$, $\frac{d}{dx}\Psi(0) = A'$. Se propone la siguiente solución de prueba:

$$\Psi_t(x) = A + xA' + x^2 N(x, \vec{p}) \tag{2.6}$$

con las condiciones de Dirichlet de dos puntos $\Psi(0) = A$, $\Psi(1) = B$. Se propone la siguiente solución de prueba:

$$\Psi_t(x) = (1 - x)A + xB + x(1 - x)N(x, \vec{p}) \tag{2.7}$$

Donde la función a minimizar es la misma que para las ODE de primer orden. El código del método es el siguiente:

$def psy\_trial(xi, net_out) :$
$return xi + xi * *2 * net_out$
$psy\_grad = grad(psy\_trial)$
$psy\_grad2 = grad(psy\_grad)$
$defloss\_function(W, x) :$
$loss\_sum = 0.$
$forxiinx :$
$net\_out = neural\_network(W, xi)[0][0]$
$net\_out\_d = grad(neural\_network\_x)(xi)$
$psy\_t = psy\_trial(xi, net\_out)$
$gradient\_of\_trial = psy\_grad(xi, net\_out)$
$second\_gradient\_of\_trial = psy\_grad2(xi, net\_out)$
$func = f(xi, psy\_t, gradient\_of\_trial)$
$err\_sqr = (second\_gradient\_of\_trial - func) * *2$
$loss\_sum+ = err\_sqr$
$returnloss\_sum$

Se toman las PDE con la siguiente forma:

$$\frac{d^2}{dx^2}\Psi(x,y) + \frac{d^2}{dy^2}\Psi(x,y) = f(x,y) \tag{2.8}$$

Donde $x \in [0,1]$, $y \in [0,1]$, con las siguientes condiciones de Dirichlet: $\Psi(0,y) = f_0(y)$, $\Psi(1,y) = f_1(y)$, $\Psi(x,0) = g_0(x)$ y $\Psi(x,1) = f_1(x)$. La solución de prueba propuesta es:

$$\Psi_t(x,y) = A(x,y) + x(1-x)y(1-y)N(x,y,\vec{p}) \tag{2.9}$$

Se escoge un $A(x,y)$ para satisfacer las condiciones de Dirichlet y para encontrarla se usa lo siguiente:

$$A(x,y) = (1-x)f_0(y)+xf_1(y)+(1-y)g_0(x) - [(1-x)g_0(0) + xg_0(1)]+yg_1(x) - [(1-x)g_1(0) + xg \tag{2.10}$$

La función a minimizar es la siguiente:

$$E[\vec{p}] = \sum_i [\frac{d^2}{dx^2}\Psi_t(x_i,y_i) + \frac{d^2}{dy^2}\Psi_t(x_i,y_i) - f(x_i,y_i)]^2 \tag{2.11}$$

El programa para esta PDE es:

$defloss\_function(W, x, y):$
$loss\_sum = 0.$
$forxiinx:$
$foryiiny:$
$input\_point = np.array([xi, yi])$
$net\_out = neural\_network(W, input\_point)[0]$
$net\_out\_jacobian = jacobian(neural\_network\_x)(input\_point)$
$net\_out\_hessian = jacobian(jacobian(neural\_network\_x))(input\_point)$
$psy\_t = psy\_trial(input\_point, net\_out)$
$psy\_t\_jacobian = jacobian(psy\_trial)(input\_point, net_out)$
$psy\_t\_hessian = jacobian(jacobian(psy\_trial))(input\_point, net\_out)$
$gradient\_of\_trial\_d2x = psy\_t\_hessian[0][0]$
$gradient\_of\_trial\_d2y = psy\_t\_hessian[1][1]$
$func = f(input\_point)$
$err\_sqr = ((gradient\_of\_trial\_d2x + gradient\_of\_trial\_d2y) - func) * *2$
$loss\_sum+ = err\_sqr$
$returnloss\_sum$

Finalmente este método tiene las siguientes ventajas sobre los demás métodos: 1) La solución es diferenciable, cercana a una forma analitica y facil de usar en cualquier cálculo subsecuente. 2) El número requerido de parámetros del modelo es menor que en otras técnicas, por lo cual la solución obtenida es compacta y la demanda de memoria es menor. 3)Se puede aplicar para ODE, PDE y sistemas de ODE.

Capítulo 3

Resultados

Se debe mencionar que para estos problemas consideramos las soluciones u como valores ya dados y no nos adentraremos en cómo encontrar estos valores. Por lo cual al conocer las soluciones, las sustituimos y llegamos a ecuaciones diferenciales que se pueden resolver con el método antes descrito.

3.1. Crecimiento de una planta

Supongamos que un jardinero tiene un número de plantas que quiere hacer crecer hasta una determinada altura para una fecha dada. La velocidad natural de crecimiento puede ser acelerada por luz artificial para reducir las horas de obscuridad, ya que en estas horas la planta no crece. Podemos modelar este proceso con una ecuación diferencial para la altura $x(t)$ de la planta al tiempo t, de la siguiente forma:

$$\frac{dx}{dt} = 1 + u \tag{3.1}$$

Donde $u(t)$ mide el exceso de velocidad de crecimiento producido por la luz artificial. Supones que la altura es cero al principio, pero se requiere que la altura sea de 2 unidades al final, por lo cual tenemos las siguientes condiciones:

$$x(0) = 0x(1) = 2 \tag{3.2}$$

Para este problema considerando u(t)=1 [4].

3.1.1. Solucion analítica

$importautograd.numpyasnp$
$fromautogradimportgrad$
$importautograd.numpy.randomasnpr$
$fromautograd.coreimportprimitive$
$frommatplotlibimportpyplotasplt$

$nx = 10$
$dx = 1./nx$

$defA(x) :''' Parteizquierdadelaecuaciónoriginal'''return0$

$defB(x) :''' Partederechadelaecuaciónoriginal'''return2$

$deff(x, psy) :''' d(psy)/dx = f(x, psy)'''returnB(x) - psy * A(x)$

$defpsy_analytic(x) :''' Soluciónanalíticadelproblema'''return2 * xx\_space = np.linspace(0, 1, nx)y$

3.1.2. Solución con AD

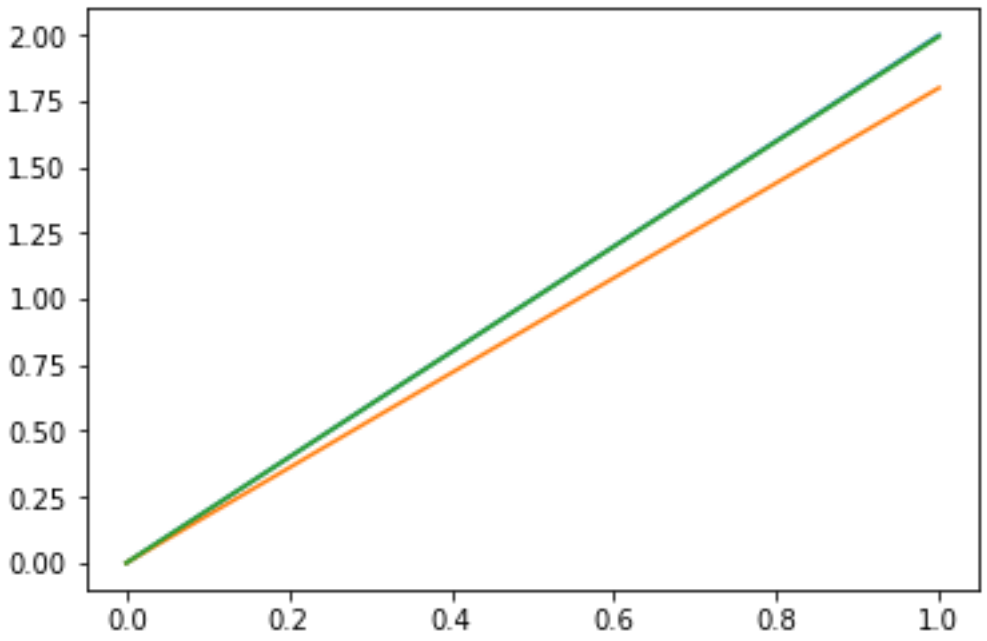
$defsigmoid(x) :$
$return1/(1 + np.exp(-x))$
$defsigmoid\_grad(x) :$
$returnsigmoid(x) * (1 - sigmoid(x))$
$defneural\_network(W, x) :$
$a1 = sigmoid(np.dot(x, W[0]))$
$returnnp.dot(a1, W[1])$
$defd\_neural\_network\_dx(W, x, k = 1) :$
$returnnp.dot(np.dot(W[1].T, W[0].T * *k), sigmoid_grad(x))$
$defloss\_function(W, x) :$
$loss\_sum = 0.$
$forxiinx :$
$net\_out = neural\_network(W, xi)[0][0]$
$psy\_t = 1. + xi * net\_out$
$d\_net\_out = d\_neural\_network\_dx(W, xi)[0][0]$
$d\_psy\_t = net\_out + xi * d\_net\_out$
$func = f(xi, psy\_t)$
$err\_sqr = (d\_psy\_t - func) * *2$
$loss\_sum+ = err\_sqr$
$returnloss\_sum$

$W = [npr.randn(1, 10), npr.randn(10, 1)]$
$lmb = 0.001$
$x = np.array(1)$
$printneural\_network(W, x)$
$printd\_neural\_network\_dx(W, x)$
$foriinrange(1000) :$
$loss\_grad = grad(loss\_function)(W, x\_space)$
$printloss\_grad[0].shape, W[0].shape$
$printloss\_grad[1].shape, W[1].shape$
$W[0] = W[0] - lmb * loss\_grad[0]$
$W[1] = W[1] - lmb * loss\_grad[1]$
$printloss\_function(W, x\_space)$
$r = loss\_function(W, x\_space)$
$print("MSE = " + str(r))$
$res = [0 + xi * neural\_network(W, xi)[0][0]forxiinx\_space]$
$print(W)$
$lineaazul : solucionanalitica$
$lineaverde : solucionneuronal$
$lineaamarilla : diferenciasfinitas$
$plt.figure()$
$plt.plot(x\_space, y\_space)$
$plt.plot(x\_space, psy\_fd)$
$plt.plot(x\_space, res)$
$plt.show()$

3.2. Conclusión

Para el problema de creciemiento de una planta se obtuvieron dos soluciones para compararlas entre ellas, en la figura 3.1 se puede observar que la solución con AD se ajusta mejor que la solución con diferencias finitas. En este caso el problema es uno de los mas sencillos en control optimo ademas de que ya conociamos la solución para optimizar u, pero el hecho de que se da una solucón aceptable nos da a pensar que el método propuesto en el articulo [5] o por lo menos el uso de AD puede usarse para resolver problemas mas complejos de control optimo.

Figura 3.1: La línea azul representa la solución analítica, la línea amarilla la solución con diferencias finitas y la línea verde representa la solución con AD.



Bibliografía

[1] R. Shekari Beidokhti A. Malek. *Numerical solution for high order differential equations using hybrid neural network - Optimization method.* ELSEVIER.

[2] Alexey Andreyevich Radul Jeffrey Mark Siskind Atilim Gunes Baydin, Brak A. Pearlmutter. *Automatic Differentiation in Machine Learning: a survey.* Journal of Machine Learning Research, 2018.

[3] Francois Chollet. *Deep learning with Python.* Manning Publication, 2018.

[4] Leslie M. Hocking. *Optimal Control, an introduction to the teory with applications.* Oxford University Press, 1991.

[5] D. I Fotiadis I. E. Lagaris, A. Likas. *Artificial Neural Networks for solving Ordinary and Partial Differential Ecuation.* IEEE, 1997.

[6] M. Kiener M. M. Chiaramonte. *Solving differential ecuation using neural networks.*