Portafolio de Análisis:

Implementación de técnicas de aprendizaje máquina.

Diego Isaac Fuentes Juvera

A01705506

Introducción

En este proyecto usé el Dataset de Student Stress Monitoring Datasets de Kaggle para generar modelos de machine learning que pudieran predecir el nivel de estrés en estudiantes universitarios de 18 a 21 años tomando en cuenta parámetros psicológicos, fisiológicos, ambientales, académicos, y sociales, permitiendo de esta manera conocer cuáles son las variables que podemos mejorar en nuestro estilo de vida para reducir nuestro nivel de estrés y mejorar nuestra calidad de vida. Se aplicó un proceso de ETL donde se limpiaron los datos, se aplicaron transformaciones, y se redujo la dimensionalidad del Dataset para poder simplificar el proceso de entrenamiento, el cuál fue llevado a cabo en una red neuronal densa multicapa programada sin el uso de librerías.

Relevancia

Hoy en día la salud mental necesita más atención que nunca puesto que vivimos en lo que algunos llaman una 'Epidemia silenciosa'. La combinación de las redes sociales, la competitividad académica y laboral a nivel mundial, y la reciente pandemia, entre muchas otras causas, han aumentado el nivel de estrés, depresión, y ansiedad en los jóvenes de manera considerable. Durante la pandemia se estimó que aproximadamente 1 de cada 4 jóvenes presentaba síntomas de depresión, el doble de los niveles anteriores (JAMA Pediatrics, 2021). Estos números alarmantes hacen que nos debamos preguntar qué podemos hacer para mejorar nuestra salud mental y para ayudar a otras personas, así como preguntarnos de qué manera podemos detectar a quienes necesitan ayuda antes de que el estrés se convierta en un problema grave.

Análisis del Dataset

El Dataset fue publicado en Kaggle por Sultanul Ovi, un estudiante de doctorado en ciencias computacionales en la universidad George Mason, en Estados Unidos. Los datos fueron recolectados a través de encuestas aplicadas a 843 estudiantes universitarios de entre 18 a 21 años de todo Estados Unidos, donde se incluyeron preguntas para conocer el demográfico de los encuestados, indicadores emocionales y de estrés (cualitativo), indicadores físicos y de salud, fuentes de estrés académicas y sociales, datos sobre sus relaciones, y finalmente el estrés experimentado. Todos los datos fueron convertidos a parámetros

numéricos (enteros positivos) según las respuestas obtenidas. Estos son los parámetros y su descripción:

Parámetro	Rango	Descripción
anxiety_level	0-21	Nivel de ansiedad medido.
self_esteem	0-30	Autoestima percibida por el estudiante.
mental_health_history	0-1	Antecedente personal de problemas de salud mental (0 no, 1 si).
depression	0-27	Nivel de depresión medido.
headache	0-5	Frecuencia de dolores de cabeza.
blood_pressure	0-3	Estado de la presión arterial.
sleep_quality	0-5	Calidad de sueno percibida
breathing_problem	0-5	Frecuencia de problemas respiratorios.
noise_level	0-5	Nivel de ruido en el entorno.
living_conditions	0-5	Calidad de las condiciones de vida.
safety	0-5	Percepción de seguridad en el entorno.
basic_needs	0-5	Grado de satisfacción de necesidades básicas.
academic_performance	0-5	Desempeño académico.
study_load	0-5	Carga de estudios percibida.
teacher_student_relationship	0-5	Calidad de la relación con profesores
future_career_concerns	0-5	Preocupacion respecto al futuro profesional
social_support	0-3	Percepción de apoyo de otras personas.
peer_pressure	0-5	Nivel de presión social percibida.
extracurricular_activities	0-5	Nivel de participación en actividades extracurriculares.
bullying	0-5	Frecuencia de experiencias de acoso escolar.
stress_level	0-2	Nivel de estres experimentado (Sin estrés, eustrés, angustia/Distress)

Tratamiento de los datos

Utilicé Python 3.11.6 como entorno de ejecución. Me apoyé de varios notebooks de Jupyter para llevar a cabo pruebas y al final pasé todos los scripts generados a un archivo .py.

Utilicé las librerías **math** y **random** la creación de la red neuronal; utilicé **pandas** y **numpy** para el tratamiento y limpieza de los datos; utilicé **matplotlib** y **seaborn** para la elaboración de las gráficas con los resultados.

Removí las filas con valores faltantes, así como las filas con valores duplicados. Debido que los datos fueron obtenidos a partir de variables cualitativas el dataset no cuenta con outliers fuera los rangos definidos para cada respuesta, por lo que no fue necesario remover los valores extremos.

Para poder hacer la reducción dimensional tomé en cuenta 2 cosas: El nivel de correlación o significancia del parámetro respecto al nivel de estrés, y mi interés personal en estudiar el parámetro, ya sea

por que considero que ese parámetro tiene o ha tenido un impacto en mi vida, o porque creo que estudiándolo seré capaz de ayudar más a otros.

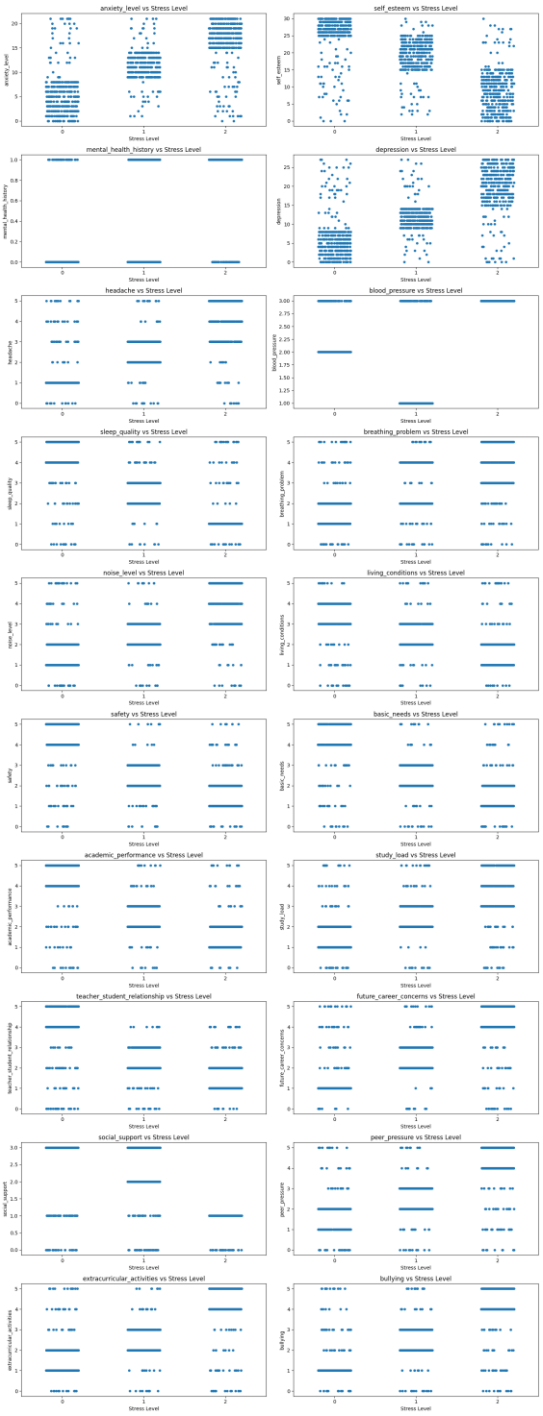
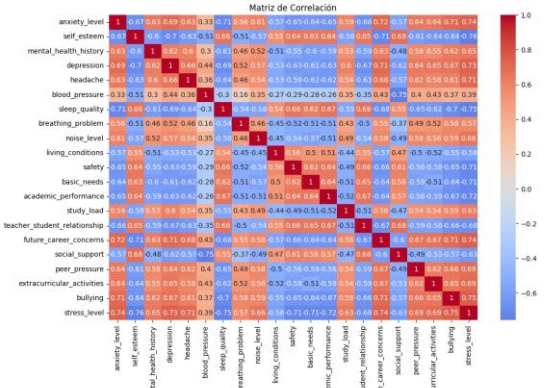


Figura 1.1

La Figura 1.1 muestra las gráficas de cada uno de los registros de cada parámetro contrapuesto al nivel de estés. Los ordene por nivel de estrés para que los registros que reportaron 0, 1, y 2 salieran a la izquierda, en medio, y a la derecha respectivamente para facilitar la comprensión de la correlación. Para casi todas las variables se observa una relación entre su valor y el número de respuestas de cada nivel de estrés. Esto es especialmente notorio en las variables de ansiedad, autoestima, y depresión debido a que utilizan un rango mayor de respuestas posibles, creando bloques notorios de respuestas. Para el resto de variables también se puede ver el aumento/descenso en la densidad de respuestas de cada nivel de estrés según el cambio en la variable, y la única variable donde es menos notorio son el histórico de salud mental y la presión arterial.

Después de este análisis, generé una matriz de correlación para poder conocer de forma numérica cómo se estaban comportando las variables:



10 de las 20 variables tuvieron una correlación mayor a 0.7, por lo que para mi modelo de predicción automáticamente descarté el resto de las variables que tenían una correlación de entre 0.39 y 0.69.

Finalmente me quedé con los parámetros de autoestima, calidad de sueño, depresión, nivel de ansiedad, y dolores de cabeza para enfocarme en variables de salud física y mental.

Para la normalización de los datos limité los resultados de salida a un rango de -1 a 1 para que coincidan con el output de la función TanH. También llevé a cabo varios experimentos con la arquitectura que diseñé (la cual describo en otro punto del reporte), llegando a la conclusión que los mejores resultados se dieron al normalizar en un rango de 0 a 5 usando ReLU como función de activación, sin normalizar, y normalizando de -2 a 2 usando TanH como función de activación.

Arquitectura matemática

Para la construcción de la red neuronal utilicé una arquitectura matemática que permite representar operaciones matemáticas como un grafo. Para esto me inspiré en el video 'The spelled-out intro to neural networks and backpropagation: building micrograd' de Andrej Karpathy.

La clase nodo es la base de todo, fundamentalmente solo almacena un valor y describe de dónde salió. En la figura 2.1 se observa el constructor de la clase, donde se guarda el valor numérico del nodo, el operador que se usó para construir ese valor numérico (+, -, /, *, etc), una etiqueta opcional, el gradiente del nodo (calculado de forma automática con regla de la cadena durante el backward pass), y un conjunto de nodos que fueron operados para crear el nodo actual.

```python
class Node:

    def __init__(self, data, _children=(), _op='', label=''):

        # Valor numérico del nodo
        self.data= data
        # Operador usado para crear este nodo
        self._op = _op
        # Nombre del nodo (opcional)
        self.label = label
        # Gradiente de la función respecto a este nodo
        self.grad = 0.0
        # Nodos que fueron operados para crear el nodo actual
        self._prev = _children
```

Figura 2.1

Para poder aplicarles operaciones a los nodos sobrecargué todos los operadores necesarios para llevar a cabo mi modelo. Por ejemplo al sumar un nodo con otro se crea un nuevo nodo cuyo valor es la suma de ambos nodos, sus nodos previos corresponden a ambos nodos que operamos, y la operación que se llevó a cabo para llegar a este resultado se define como suma o '+'. Esto se repite con los demás operadores como se muestra en la figura 2.2:

```python
    # Sobrecarga de operadores para poder hacer operaciones entre nodos.

    def __repr__(self):
        return f"Node(data={self.data})" #, children={self._prev})"

    def __add__(self, other):
        out = Node(self.data + other.data, (self, other), '+')
        return out

    def __mul__(self, other):
        out = Node(self.data * other.data, (self, other), '*')
        return out

    def __sub__(self, other):
        out = Node(self.data - other.data, (self, other), '-')
        return out

    def __pow__(self, other):
        out = Node(self.data**other.data, (self, other), '**')
        return out

    def __iadd__(self, other):
        total = self.data + other.data
        out = Node(total, (self, other), '+')
        return out

    def __truediv__(self, other):
        out = Node(self.data / other.data, (self, other), '/')
        return out
```

Figura 2.2

También implementé 2 funciones de activación como operaciones posibles para los nodos: ReLU y Tangente Hiperbólica, las cuales normalizan la salida y aplican transformaciones no lineares, lo que permite aumentar muchísimo las

capacidades de abstracción de la futura red neuronal donde se implementen.

```python
# Funciones de activación que pueden aplicarse a un nodo.

def tanh(self):
    x = self.data
    if x > 100:
        x = 100
    elif x < -100:
        x = -100
    t = (math.exp(2*x) - 1) / (math.exp(2*x) + 1)
    out = Node(t, (self, ), 'TanH')
    return out

def RelU(self):
    x = self.data
    r = max(0.01*x, x)
    out = Node(r, (self, ), 'RelU')
    return out
```

Figura 2.3

Un ejemplo rápido de uso es la figura 2.4, donde se ve que el Nodo resultante de multiplicar 3 * 5 tiene valor 15, 2 nodos hijos de valor 3 y 5, y un operador de origen "*".

```
Node(5) * Node(3)
✓ 0.0s
Node(data=15, children=(Node(data=5, children=(), operator=),
Node(data=3, children=(), operator=)), operator=*)
```

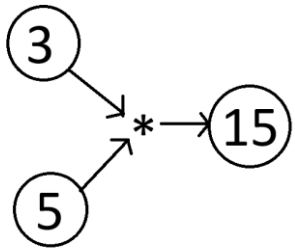


Figura 2.4, 2.5, y 2.6

Finalmente, lo que hace que este modelo matemático sea especialmente poderoso para implementar redes neuronales es su capacidad de obtener el gradiente en cada nodo de forma automática respecto a cualquier nodo de salida.

Para obtener el gradiente necesito hacer una derivada parcial de la salida de mis operaciones respecto a cada otra variable y operación que la conforma. Debido a que tenemos muchas operaciones las cuales se afectan entre sí para llegar al resultado final, debemos aplicar la regla de la cadena, permitiéndonos saber cuanto aporta cada variable al resultado que tenemos y hacia donde las tenemos que mover para obtener el resultado que queremos.

La regla de la cadena dice que "una variable Z depende de una variable Y y a su vez esta depende de X (esto es Y y Z son variables dependientes) entonces Z también depende de X , en tal caso, la regla de la cadena enuncia que:" (Wikipedia, 2025)

$$\frac{dz}{dx} = \frac{dz}{dy} * \frac{dy}{dx}$$

Lo que significa que podemos evaluar cada derivada de forma independiente (la derivada de Z respecto a Y, y de Y respecto a Z) y al final calcular cual es la derivada de Z respecto a Y.

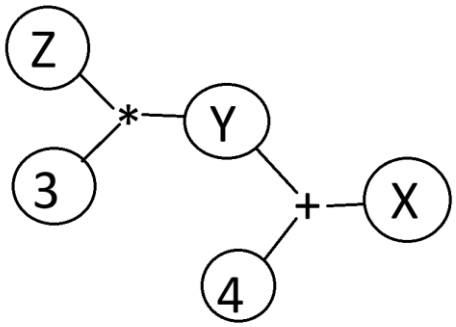


Figura 2.7

Intentémoslo con la figura 2.7 Para obtener la derivada de Z respecto a X primero debemos calcular la derivada de Y respecto a X y a su vez la derivada de X

respecto a X. Debido a que X es el nodo de salida, su derivada siempre será 1 ya que si X cambia en una unidad, la salida también lo hace (ya que X es la salida)

Para obtener la derivada de Y respecto a X aplicamos la regla de la cadena ahora derivando una suma. La derivada de una suma es 1, demostrado por la ecuación de la derivada:

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

Donde si reemplazamos f(x) por Y + Z (la suma de los nodos hijos) nos queda:

$$f'(x) = \lim_{h \to 0} \frac{Y + Z + h - (Y + Z)}{h}$$

Simplificando:

$$f'(x) = \lim_{h \to 0} \frac{Y + Z + h - Y - Z)}{h}$$

Cancelando Y con -Y y Z con -Z:

$$f'(x) = \lim_{h \to 0} \frac{h}{h}$$

Dividiendo h sobre h:

$$f'(x) = 1$$

Con ambas derivadas, aplicamos regla de la cadena multiplicando la derivada del nodo X por 1, ósea 1*1 = 1, significando que por cada unidad que cambie el nodo Y, la salida también cambiará en 1. Lo mismo aplica para el otro lado de la suma, dando que la derivada del 4 también es 1. La suma sirve para 'distribuir' el gradiente en la ecuación.

Continuando la regla de la cadena ahora para calcular la derivada de Z respecto a Y:

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

Reemplazamos f(x) por 3 * Z:

$$f'(x) = \lim_{h \to 0} \frac{3 * (Z + h) - 3 * Z}{h}$$

Multiplicando por 3:

$$f'(x) = \lim_{h \to 0} \frac{3Z + 3h - 3Z}{h}$$

Simplificando:

$$f'(x) = \lim_{h \to 0} \frac{3h}{h} = 3$$

Nos podemos dar cuenta que la derivada de Z respecto a Y es igual al valor que tenga el nodo del otro lado de la multiplicación.

Para la regla de la cadena tenemos que multiplicar la derivada de Y respecto a X por 3 (la derivada de Z respecto a Y) para obtener la derivada de Z respecto a X, dando como resultado 3*1 = 3, significando que si Z cambia en 1, la salida cambia en 3.

Esta misma lógica se aplica a cualquier otra operación, donde conociendo su derivada simple podemos encadenarla a través de los operadores de suma y obtener su derivada respecto a la salida.

Dentro del código, el calculo del gradiente o backward pass está programando en 2 fases: Un pequeño ciclo que itera por todos los nodos que conforman el grafo para anotarlos (figura 2.8), y un ciclo que recorre el grafo completo de enfrente hacia atrás para calcular el gradiente de cada uno de los nodos respecto al nodo de salida donde se llamó la función (figura 2.9).

```python
def backward(self):
    self.grad = 1

        # Crear lista topológicamente ordenada
    topo = []
    visited = set()

    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:
                build_topo(child)
            topo.append(v)

    # Construir orden topológico
    build_topo(self)
```

Figura 2.8

```python
for node in reversed(topo):
    if node._op == '+':
        for n in node._prev:
            n.grad += 1.0 * node.grad

    elif node._op == '-':
        node._prev[0].grad += 1.0 * node.grad
        node._prev[1].grad += -1.0 * node.grad

    elif node._op == '*':
        node._prev[0].grad += node._prev[1].data * node.grad
        node._prev[1].grad += node._prev[0].data * node.grad

    elif node._op == '**':
        exp = node._prev[1].data
        node._prev[0].grad += exp * (node._prev[0].data ** (exp - 1)) * node.grad
        # recurse(node._prev[1])

    elif node._op == 'TanH':
        node._prev[0].grad += (1 - node.data**2) * node.grad
        # self._prev[0].recurse()

    elif node._op == 'ReLU':
        if node.data > 0:
            node._prev[0].grad += 1 * node.grad
        else:
            node._prev[0].grad += 0.01*node.grad
        # node._prev[0].recurse()

    elif node._op == '/':
        node._prev[0].grad += (1/ node._prev[1].data) * node.grad
        node._prev[1].grad += ((-node._prev[0].data) / (node._prev[1].data**2))* node.grad
```

Figura 2.9

La figura 2.9 muestra el código para calcular el gradiente según el operador. En el caso de la suma toma el gradiente del nodo de salida y lo añade a su propio gradiente. En la multiplicación toma el gradiente del nodo de salida por el gradiente del otro nodo hijo. Para la función ReLU toma el gradiente del nodo de salida multiplicado por 1 si el nodo actual es mayor a 0 o 0.01 si es menor a 0.

Para añadir nuevas funciones de activación u operadores en el futuro, además de definir su método para operar, también se debe definir su derivada en esta parte del código y listo.

Red Neuronal

Hasta ahora nuestra arquitectura matemática no conoce nada sobre aprendizaje automático, solo sabe hacer operaciones y calcular el gradiente, por lo que debemos definir mecanismos para que pueda aprender y generar un modelo matemático por si sola.

Una neurona no es más que un conjunto de parámetros multiplicados por pesos y sumados entre sí. Esto los vuelve perfectos para ser representados como operaciones de nodos. En nuestro modelo definimos nuestra neurona como una función de activación, una lista de valores de entrada, una lista de pesos (que al principio son definidos como valores aleatorios entre -1 y 1), y un sesgo:

```python
class Neuron:
    def __init__(self, inputs, activation):
        # Lista de pesos (uno por cada entrada)
        self.activation = activation
        self.w = []
        for _ in range(inputs):
            self.w.append(Node(random.uniform(-1, 1)))
        # Bias
        self.b = Node(random.uniform(-1,1))
```

Figura 3.1

Estos valores se operarán entre sí para obtener un resultado al llamar a la neurona con la función sobrecargada de __call__, que permite evaluar la salida de la neurona al recibir una serie de entradas ‘x’, como se muestra en la figura 3.2:

```python
# Al llamar a la neurona con una lista de e
def __call__(self, x):
    # Inicializar el total de la activación
    total = Node(self.b.data)

    # Crea pares de peso y entrada.
    for wi, xi in zip(self.w, x):
        # Si la entrada no es un nodo, lo c
        if type(xi) != Node:
            xi = Node(xi)
        # print('weight/input: ', wi, xi)
        # Multiplica el peso por la entrada
        activation = wi * xi
        total = total + activation
    # Normaliza la activación de la neurona
    if self.activation == 'TanH':
        output = total.tanh()
    elif self.activation == 'ReLU':
        output = total.RelU()
    else:
        output = total

    # out = activation.tanh()
    return output
```

Figura 3.2

Al llamar a la neurona, se añade el bias a la activación total, luego con la función zip se emparejan los pesos de la neurona con su entrada correspondiente, convirtiendo la entrada en un nodo para que pueda ser operado dentro de nuestra arquitectura matemática. Al obtener todos los resultados de multiplicar las entradas con los pesos, aplicamos la función de activación que le corresponde a esa neurona y regresamos el nodo con el valor de activación final.

En la figura 3.3 se define un método para obtener los parámetros de la neurona, lo que servirá para ajustarlos durante el entrenamiento.

```python
# Regresa los parámetros de la
def parameters(self):
    return self.w + [self.b]
```

Figura 3.3

Cada red neuronal además de contar con muchas neuronas se conforma de capas. Las capas son conjuntos de neuronas que obtienen las mismas entradas y operan en paralelo sin interactuar entre sí (al menos en redes neuronales básicas) para devolver una salida en conjunto.

Esto se define en la clase Layer de la figura 3.4, la cual recibe como parámetros el número de entradas y salidas esperadas de la capa, así como la función de activación que tendrá cada neurona. Al llamarla se evalúa cada neurona de la capa con las entradas dadas (x) y se regresan los nodos con la salida de cada neurona.

```python
class Layer:
    def __init__(self, inputs, outputs, activation):
        # Crea una lista de neuronas correspondiendo con el
        self.neurons = []
        for _ in range(outputs):
            self.neurons.append(Neuron(inputs, activation))

    # Al llamar la capa con una lista de entradas regresa la
    def __call__(self, x):
        output = []
        for neuron in self.neurons:
            # print('neuron: ', neuron.w, neuron.b)
            output.append(neuron(x))
        return output[0] if len(output) == 1 else output

    # Regresa los parámetros de cada neurona de la capa como
    def parameters(self):
        parameters = []
        for neuron in self.neurons:
            ps = neuron.parameters()
            parameters.extend(ps)
        return parameters
```

Figura 3.4

De igual manera tiene un método definido para obtener todos los parámetros de todas las neuronas, que permitirá ajustarlos en el entrenamiento.

Finalmente, el nivel de abstracción más alto en el que operan los nodos es una red neuronal. Para esto creamos varias capas conectadas entre sí, las cuales cuentan con un número de entradas igual al número de salidas de la capa anterior, y un número de neuronas o salidas y función de activación

para cada capa definido por el usuario (Figura 3.5)

```
class NN:
    def __init__(self, inputs, layers, activations):
        # Crear una lista con el número de valores por capa incluyendo la entr
        inout = [inputs] + layers
        self.layers = []

        # Crea capas que tienen el número de entradas = al número de salidas d
        for i in range(len(layers)):
            self.layers.append(Layer(inout[i], inout[i+1], activations[i]))
```

Figura 3.5

Cuenta también con una función para obtener el resultado final (iterando en las capas y dando como entrada la salida de la capa anterior hasta llegar a la última) y otra función para obtener los parámetros.

```
    # Al llamar la red neuronal con una list
    def __call__(self, x):
        for layer in self.layers:
            # print('layer: ', layer)
            x = layer(x)
        return x

    # Regresa lois parámetros de cada neuron
    def parameters(self):
        parameters = []
        for layer in self.layers:
            ps = layer.parameters()
            parameters.extend(ps)
        return parameters
```

Figura 3.6

Para construir un modelo se crea un objeto de la clase NN con el número de entradas, la descripción de las capas, y las funciones de activación para cada capa

```
stress_predictor = NN(5, [8, 4, 1], ['ReLU', 'ReLU', 'TanH'])
```

Aprendizaje automático

El fordward pass consiste en un ciclo donde se itera en cada uno de los conjuntos de entradas para obtener su salida correspondiente evaluando al modelo.

```
def predict(network, xs):
    prediction = []
    for x in xs:
        prediction.append(network(x))
    return prediction
```

Figura 4.1

El ciclo de entrenamiento (figura 4.3) consta de evaluar el modelo con el conjunto de datos de entrenamiento, y calcular la perdida utilizando la fórmula del error cuadrático. Esta se define dentro de un nuevo nodo conocido como perdida, donde se operan los nodos con la salida del modelo y las salidas esperadas. El hecho de que la perdida esté definida como un Nodo, con operaciones cuya derivada programamos, nos permite que podamos calcular el gradiente de toda la red respecto a la pérdida de forma automática.

En la figura 4.2 usamos el método loss.backward() para calcular el gradiente de todos los nodos que conforman la red neuronal y la función de perdida.

Finalmente, obtenemos todos los parámetros de la red neuronal (el conjunto de pesos y bias de cada neurona), y multiplicamos su gradiente por el learning rate y por -1 para minimizar la función de pérdida. Repetimos el ciclo hasta finalizar los ciclos de entrenamiento que definimos.

```python
def train(network, xs, ys, epochs=1000, learning_rate=0.01, printability=500):
    for i in range(epochs):

        # Fordward
        ypred = predict(network, xs)
        # Calculo de la pérdida con el error cuadrático.
        loss = Node(0.0)
        for yt, yout in zip(ys, ypred):

            # Convert single values to list
            if isinstance(yt, (int, float)):
                yt = [yt]

            # Ensure prediction is in correct format
            if not isinstance(yout, list):
                yout = [yout]

            for target, pred in zip(yt, yout):
                loss += (pred - Node(target))**Node(2)

        loss = loss * Node(1/len(ys))
        # Backward
        # Backpropagation calculando el gradiente de la función de pérdida resp
        loss.backward()

        # Update
        for parameter in network.parameters():
            parameter.data += -learning_rate * parameter.grad
            parameter.grad = 0.0

        if i % printability == 0:
            print(f"Epoch {i}, loss: {loss.data}")
```

Figura 4.2

Resultados

```python
train(stress_predictor, X_train_list, Y_train_list, epochs=200, learning_rate=0.01, printability=20)
```

En mi primera iteración solo usé TanH como función de activación, y di al modelo 5 features de entrada, 3 capaz ocultas de 8, 6 y 4 neuronas, y 1 neurona de salida. Entrené al modelo durante 200 iteraciones y con 0.01 de learning rate, obteniendo un error cuadrático medio de 0.2074 en el conjunto de entrenamiento y de 0.3898 en el de pruebas. Los resultados de la figura 5.1 están ordenados por nivel de estrés real para facilitar su visualización:

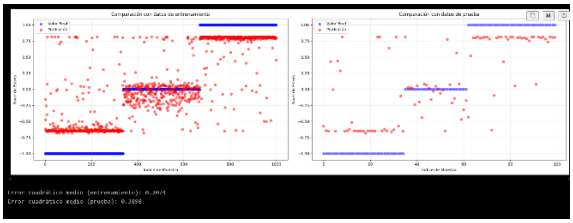


Figura 5.1

Aunque estos resultados me pusieron muy contento ya que mostraba que mi modelo era capaz de aprender, aún tenía muchas áreas de mejora. Los principales problemas es que se tardaba más de media hora en entrenar, lo que adjudico principalmente al uso desmedido de la función TanH. También el modelo es muy sensible a los resultados de 0 (tienen mayor variación que los extremos, y esto también se lo adjudico a que alrededor del 0 es donde TanH tiene el mayor gradiente. Y finalmente al modelo le cuesta generar valores mayores a 0.75 o menores a -0.7

Mejoras

Lo primero que hice para mejorar el modelo fue implementar la función de activación ReLU. Esta función al ser muy simple tanto en el fordward como el backward pass hace que el modelo corra más rápido y me permite poner más capaz o features. El modelo paso de entrenarse en media hora a 15 minutos, lo que me permitió subir mis iteraciones a 400 para los entrenamientos y obtener mejores resultados

Además normalicé todos los features en un rango de 0 a 5, permitiendo a todos los parámetros estar en la misma escala y haciendo que no se perdiera información por el corte de los datos que aplica ReLU.

Tras aplicar estas mejoras y entrenar el modelo obtuve un error cuadrático medio de 0.1494 en el entrenamiento (respecto a 0.2074 en el modelo anterior) y de 0.3673 en los datos de prueba (respecto a 0.3898)

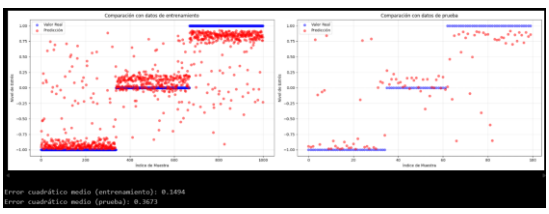


Figura 5.2

Fuera del entrenamiento y para la evaluación del modelo apliqué una normalización a los datos de salida para que solo dieran 0, 1, o 2 si eran menor a -0.5, de -0.5 a 0.5, o mayor a 0.5 respectivamente, permitiéndome mejorar los resultados y generar una matriz de confusión.

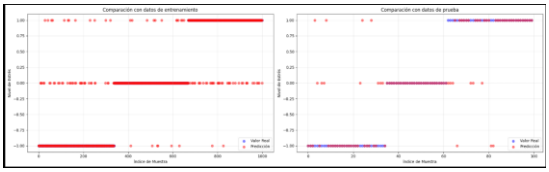


Figura 5.3

Finalmente calculé la precisión de mi modelo y mi matriz de confusión y obtuve una precisión de 0.891 en los datos de entrenamiento y de 0.80 en los datos de prueba.

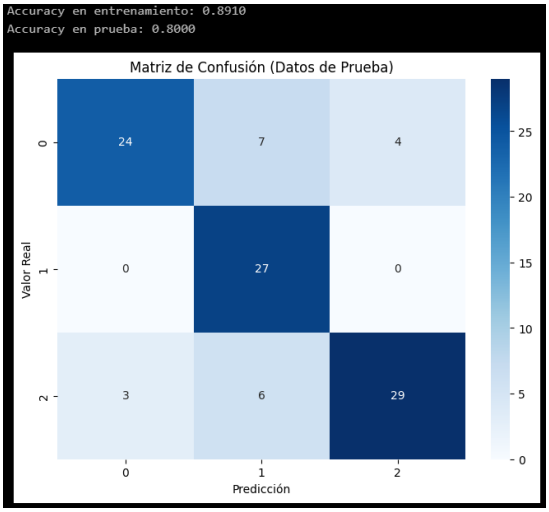


Figura 5.4

Conclusiones

Estoy muy contento con los resultados de esta actividad y sobre todo con mi aprendizaje, ya que el intentar hacer algo que parecía tan complicado desde 0 me permitió entenderlo de forma integral y afrontar cada problema que se fue presentando de forma más manejable, ya que cada nivel de abstracción (Nodo, Neurona, Capa, y Red) trabaja de forma independiente, por lo que depurar e implementar nuevas funcionalidades es fácil.

Pidiéndole a la red neuronal que prediga mi nivel de estrés con los parámetros de autoestima, calidad de sueño, depresión, ansiedad, y dolor de cabeza en 3, 3, 2, 1 y 2 respectivamente este se encuentra en un nivel medio con un valor de 0.12, pero si cambio mi calidad de sueno de 3 a 5 mi nivel de estrés podría bajar hasta -0.76, ósea casi nulo. Esto me indica que yo, además del modelo, también puedo mejorar mi calidad de vida con acciones que parecieran sencillas pero que tienen un gran impacto.

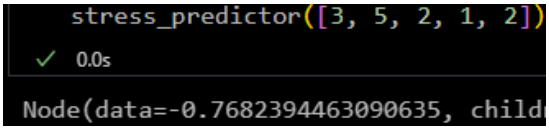


Figura 6.1

Finalmente me emociona seguir aprendiendo y trabajando en mi modelo, ya que aún tiene mucho margen de mejora, sobre todo en la precisión de las mediciones alrededor del 0 donde veo potencial a cambiar las funciones de activación y añadir más capas ocultas que permitan a la red hacer mejores abstracciones.