Universidad Nacional Autónoma de México
Facultad de Ciencias

Cómputo Evolutivo

Proyecto Final

Carlos Emilio Castañon Maldonado & Dana Berenice Hérnandez Norberto

1. Introducción

Este proyecto tiene como objetivo la optimización de una red neuronal por medio de algoritmos evolutivos, en particular por medio del conocido algoritmo genético. Para lograrlo nos basaremos en el articulo el articulo "Training Feedforward Neural Networks Using Genetic Algorithms"de David J. Montana y Lawrence Davis, donde se plantea la relación del uso de redes Neuronales con Computo Evolutivo mediante algoritmos genéticos para su optimización. Para entender mejor el problema planteado es necesario preguntarnos ¿qué es una red neuronal? ¿cómo funciona? y principalmente ¿por qué puede resultar mejor optimizarla con algoritmos evolutivos?

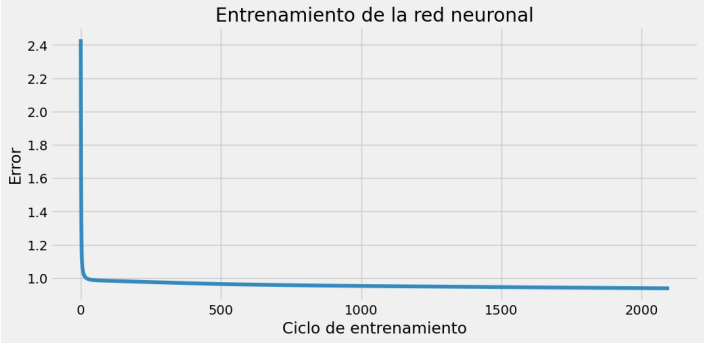
- **¿Qué es una red neuronal? ¿cómo funciona?**
  Una red neuronal es un algoritmo bioinspirado en una neurona biologica, perteneciente a los algoritmos de aprendizaje supervisado, es decir, a los algoritmos de apredizaje en donde nuestro modelo tiene las respuestas correctas y con base en ellas va aprendiendo a dar mejores resultados.
  Una red neuronal multicapa como la que utilizaremos en el proyecto se encuentra compuesta por capas de neuronas interconectadas, la primera capa llamada capa de entrada, la última llamada capa de salida y las capas intermedias llamadas capas ocultas. A cada conexión entre neuronas se le es asignado un peso, normalmente inicializado de forma aleatoria con una distribución uniforme, los cuales le permite a la red asignarle un valor de salida a cada una de las entradas con las que se está alimentando.
  Para ello, es necesario que la red vaya ajustando los pesos asignados a cada nodo por cada ciclo en el que se alimenta, debido a esto, existe el algoritmo de entrenamiento. Normalmente, el ajuste de pesos y sesgos se realiza por medio de la optimización de gradiente descendente, calculando el gradiente de los datos y permitiendonos disminuir el error de una red neuronal de forma estocastica.

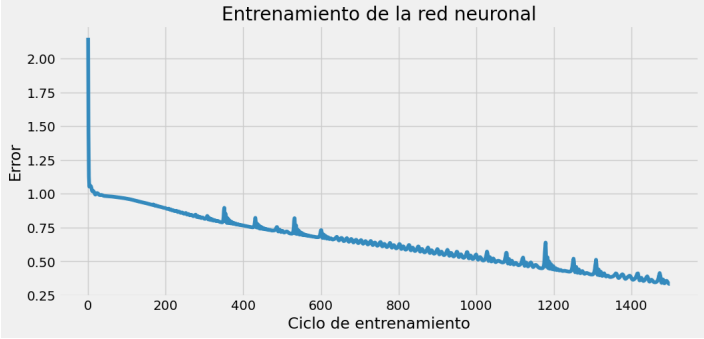
- **¿Por qué puede resultar mejor optimizarla con un algoritmo evolutivo?**
  Existen escenarios donde el error de una red neuronal se estanca y no baja de un umbral establecido en cierto punto del entrenamiento, esto puede pasar por diversos factores, como el uso de hiperparametros incorrectos, como el número de neuronas utilizadas, número de capas, el uso de una tasa de aprendizaje muy grande, etc. O bien, por la inicialización o ajuste de pesos cerca de óptimos locales, por ello es de nuestro interés aplicar este tipo de algoritmos a herramientas muy utilizadas actualmente, como son las redes neuronales, además de su experimentación con ellas.

- **Algoritmo**
  Recrearemos los experimentos realizados en el articulo, modificando la base. en el articulo abordan una base de calidad del vino, en nuestro caso utilizaremos una base datos perteneciente a todas las posibles jugadas para ganar o perder en el famoso juego de tic tac toe. Consideramos que es una buena base de datos debido a que tic tac toe puede llegar a ser considerado un juego de estrategia, y por lo tanto, a determinar su siguiente paso para resultar ganador.

2. Motivaciones

A lo largo del curso de alguna forma siempre hemos optimizado algún problema con alguna de las técnicas de computo evolutivo (como el problema de la n-coloración). Al descubrir este tipo de métodos nos dimos cuenta de la gran cantidad de aplicaciones que podemos llegar a tener, en este caso, sabemos que una red neuronal puede no ser optimizada correctamente si es inicializada en un óptimo local, este tipo de problemas pueden ser solucionados con algoritmos evolutivos. El uso de redes neuronales en el mundo de la Inteligencia Artificial es fundamental y el hecho de que obtengamos mejores resultados y reduzcamos el error obtenido. Observemos el siguiente ejemplo, graficando el error contra el numero de iteraciones de una red neuronal mal optimizada, comparada con los resultados de una red neuronal optimizada con un algoritmo evolutivo.





Y de esa forma tener un modelo de red neuronal entrenado y optimizado para que pueda jugar en terminal o en una interfaz gráfica el juego de tres en raya contra un oponente humano.

3. Desarrollo

Realizaremos el algoritmo genetico con base en los operadores ya trabajados en el curso, no obstante, es necesario conocer cómo los adaptamos a una estructura ya establecida, como es una red neuronal.

- **Codificación de cromosomas:** Utilizaremos una codificación de cromosomas, comenzando con los sesgos de la capa de salida, continuando con sus pesos, lo aplicamos de igual forma a las capas ocultas de afuera hacia dentro, siguiendo la arquitectura de la red neuronal con la que estamos trabajando. Por lo tanto, la representación de soluciones que utilizaremos será un vector de pesos de números reales.
- **Función de evaluación:** Optimizaremos el error obtenido en el entrenamiento de la red, utilizando el suma de errores al cuadrado para ello.
- **Inicialización:** Inicializaremos la población aleatoriamente de pesos con una distribución t, en comparación a la inicialización común de pesos en una red neuronal, basada comúnmente en una distribución uniforme.
- **Operadores:** Se utilizarán los mismos operadores que en un algoritmo genético normal, como son la cruza y mutación, adaptándolos a la arquitectura de la red.

- **Mutación:** Se definen dos principales metodologías para la mutación, mutación sesgada o insesgada.
  - **Mutación insesgada:** Para cada entrada en el cromosoma se tendrá una probabilidad fija p = 0.1 de reemplazarlo con un valor aleatorio elegido de la distribución de probabilidad de inicialización.
  - **Mutación sesgada:** Para cada entrada del cromosoma tendrá una probabilidad fija p = 0.1 de agregarle un valor aleatorio elegido de la distribución de probabilidad de inicialización.

  Es decir, en la primera opción reemplazaremos el valor de la entrada del cromosoma, para la segunda opción agregaremos ruido.
- **Cruza:** Al igual que con el operador de mutación, proponen 2 tipos de operadores de cruza principales:
  - **Cruza de pesos:** Este operador pone un valor en cada posición del cromosoma del hijo seleccionando aleatoriamente uno de los dos padres y usando el valor en el mismo posición en el cromosoma de ese padre.
  - **Cruza de nodos:** Este operador selecciona uno de las dos redes de los padres y encuentra el nodo correspondiente en esta red. Luego pone el peso de cada entrada.

Usaremos la biblioteca de pytorch para la implementación de la red neuronal, además de que todo el código será en el lenguaje de programación de python. Comenzamos con definir la red neuronal sobre la cual evaluaremos el algoritmo genético. Así mismo, definimos funciones auxiliares para poder utilizarla.

```
class Gen_net(nn.Module):
    '''
    Definimos un modelo de red neuronal sobre el cual podamos
    evaluar el algoritmo genetico.
    '''
    def __init__(self, in_features, hidden_1, hidden_2, out_features):
        '''
        La estructura de la red como se define en el articulo.
        Dado que no usamos el mismo conjunto de datos
        los valores de in_features y out_features se veran afectados.
        '''
        super(Gen_net, self).__init__()
        self.layer1 = nn.Linear(in_features, hidden_1)
        self.layer2 = nn.Linear(hidden_1, hidden_2)
        self.layer3 = nn.Linear(hidden_2, out_features)
        self.sigmoid = nn.Sigmoid()

        # Calculamos el numero total de pesos en la red
        self.total_weights = self.count_weights()

    def count_weights(self):
        '''
        Calcula el numero total de pesos en la red.
        '''
        total = 0
        for param in self.parameters():
            total += param.numel()
        return total

    def forward(self, X):
        '''
        Definimos la pasada hacia adelante de la red (forward).
```

```
        '''
        X = X.to(self.layer1.weight.device)  # Con esto aseguramos que X este en el mismo
        ↪  dispositivo que los pesos de la primera capa
        X = self.layer1(X)
        X = self.sigmoid(X)
        X = self.layer2(X)
        X = self.sigmoid(X)
        X = self.layer3(X.float())
        return X


    def build_from_chromosome(self, chromosome):
        '''
        Definimos una funcion para que a partir de un cromosoma modifique los
        pesos de la red.
        '''
        with torch.no_grad():
            # Extraemos pesos
            w1 = torch.tensor(chromosome[:self.layer1.weight.numel()])
            .reshape(self.layer1.weight.shape).float()
            w2 = torch.tensor(chromosome[self.layer1.weight.numel():
            self.layer1.weight.numel() + self.layer2.weight.numel()]).
            reshape(self.layer2.weight.shape).float()
            w3 = torch.tensor(chromosome[self.layer3.weight.numel():]).
            reshape(self.layer3.weight.shape).float()

            # Actualizamos pesos
            self.layer1.weight.data = w1
            self.layer2.weight.data = w2
            self.layer3.weight.data = w3


    def training(self, X, Y, optimizer):
        '''
        Definimos una funcion de entrenamiento utilizando la misma
        funcion de error que se usa para obtener el fitness de la red.
        '''
        criterion = nn.MSELoss()
        optimizer.zero_grad()
        output = self(X)
        loss = criterion(output, Y)
        loss.backward()
        optimizer.step()
        return loss.item()

# De la misma manera, definimos una función con el fin de calcular
# la matriz de confusión, esta matriz nos permite evaluar los
# resultados y la calidad de la red que programamos.

\begin{minted}[bgcolor=bg,frame=lines,framesep=2mm]{python}
def confusion(model, X, Y):
    '''
    Definimos una funcion de confusion para evaluar el modelo.
```

```
    '''
    # Evaluar el modelo en el conjunto de datos de entrada
    Y_pred = model.predict(X)

    # Calcular la matriz de confusion
    confusion_matrix = pd.crosstab(pd.Series(Y, name='Actual'), pd.Series(Y_pred,
    ↪  name='Predicted'))

    return confusion_matrix
```

Ahora definimos la estructura del algoritmo genético, con la inicialización de la población con una distribución t, la función a optimizar, como mencionamos anteriormente, como la suma de errores al cuadrado y una representación de soluciones en forma de un vector de números reales.

- **Mutación:** Aplicamos el operador para como una mutación insesgada, es decir, reemplazamos el valor de la entrada del cromosoma, en lugar de agregar ruido.
- **Cruza:** En este caso, aplicamos una cruza de pesos en lugar de una cruza de nodos.

```
class Gen_train():
    '''
    Definimos la estructura del algortimo genetico
    '''
    def __init__(self, model, parent_scalar, operator_probabilities, population_size):
        '''
        Definimos las propiedades del algortimo como se describe en el articulo.
        En la presente usamos el parametro model para tener un ejemplar de la red
        sobre la cual podamos realizar las operaciones necesarias.
        Incializamos la poblacion utilizando una distribucion t
        '''
        self.model = model
        self.parent_scalar = parent_scalar
        self.operator_probabilities = operator_probabilities
        self.population_size = population_size
        self.population = []
        self.initialize_population()
        self.losses = []

    def initialize_population(self):
        '''
        Inicializamos la poblacion utilizando una distribucion t
        '''
        # Poblacion inicial para una distribucion t
        for _ in range(self.population_size):
            chromosome = np.random.standard_t(df=1, size=self.model.total_weights)
            self.population.append(chromosome)

    def get_weights(self, element):
        '''
        Funcion para obtener las matrices de pesos a partir de un elemento de la
        ↪  poblacion.
        Debe tener las mismas dimenciones que los pesos para la red.
```

```
        '''
        return element


    def set_chromosome(self, w1, w2, w3, element):
        '''
        Funcion para actualizar un elemento de la poblacion a partir de los pesos que
        ↪ determinan
        a una red. Se actualizara el ejemplar de la poblacion que se encuentre en la
        ↪ posicion
        element.
        '''
        self.population[element] = np.concatenate((w1.flatten(), w2.flatten(),
        ↪ w3.flatten()))


    def fitness(self, X, Y, element):
        '''
        Funcion para determinar la adecuacion de un elemento de la poblacion.
        En este caso, la adecuacion nos servira para determinar los elementos
        mas aptos. Para esta implementacion se considerara que una adecuacion
        menor sera de un menor individuo, por lo que tendra mayores
        probabilidades de reproducirse.
        '''
        chromosome = self.population[element]
        self.model.build_from_chromosome(chromosome)

        X_tensor = torch.from_numpy(X.values.astype(np.float32))
        Y_tensor = torch.from_numpy(Y.values.astype(np.float32))

        # Evaluamos el fitness usando el error cuadratico medio
        outputs = self.model(X_tensor)
        loss = nn.MSELoss()(outputs, Y_tensor)
        return loss.item()



    ## Definimos las operaciones que se usan en el experimento numero 5 del articulo.
    ## Todas las definiciones de los operadores se encuentran en la seccion 5 del
    ↪ articulo.

    def train(self, steps):
        '''
        Definimos la funcion de entrenamiento, la cual realizara el numero de pasos
        seleccionados. Para ello usamos la variable de parent_scalar para determinar la
        ↪ probabilidad
        de que un individuo de la poblacion sea remplazado en cada iteracion.
        Esta funcion muestra una grafica del error al finalizar el entrenamiento.
        Y regresa una red con los pesos del individuo con mejor fitness al finalizar el
        ↪ entrenamiento.
        '''
        # Definimos el conjunto de datos
        X_train, X_val, Y_train, Y_val = read_data(path)
```

```
        # Inicializamos el mejor modelo y su fitness
        best_model = None
        best_fitness = np.inf
        losses = []
        for step in range(steps):
            # Evaluamos la adecuacion de cada individuo en la poblacion
            fitness_scores = [self.fitness(X_train, Y_train, i) for i in
            ↪  range(self.population_size)]
            best_individual_index = np.argmin(fitness_scores)
            current_best_fitness = fitness_scores[best_individual_index]

            # Almacenamos el mejor modelo hasta el momento
            if current_best_fitness < best_fitness:
                best_fitness = current_best_fitness
                best_model = self.model

            # Mostramos el progreso cada 100 pasos en pantalla
            if step % 100 == 0:
                print(f"Step: {step}, Best fitness: {best_fitness}")

            # Seleccionar los padres (los mejores individuos de la poblacion)
            parents_indices = np.argsort(fitness_scores)[:int(self.parent_scalar *
            ↪  self.population_size)]

            # Generamos la nueva generacion utilizando los operadores geneticos
            new_population = []
            for _ in range(self.population_size):
                parent1, parent2 = np.random.choice(parents_indices, size=2,
                ↪  replace=False)
                child = self.crossover(self.population[parent1], self.population[parent2])
                child = self.mutate(child)
                new_population.append(child)

            # Actualizamos la poblacion
            self.population = new_population

            # Registramos la perdida
            losses.append(best_fitness)

        # Mostramos la curva de aprendizaje
        plt.plot(losses)
        plt.xlabel('Generation')
        plt.ylabel('Best Fitness')
        plt.title('Training Curve (Genetic Algorithm)')
        plt.show()

        return best_model

    def crossover(self, parent1, parent2):
        '''
        La operacion de cruce como se describe en el articulo.
        '''
```

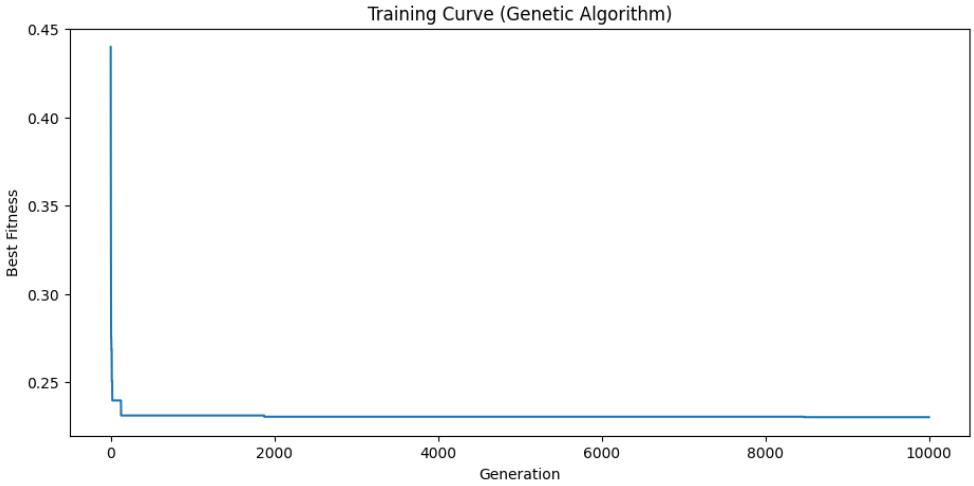
```
        child = np.zeros_like(parent1)
        for i in range(len(parent1)):
            if np.random.rand() < 0.5:
                child[i] = parent1[i]
            else:
                child[i] = parent2[i]
        return child

    def mutate(self, chromosome):
        '''
        La operacion de mutacion como se describe en el articulo.
        '''
        for i in range(len(chromosome)):
            if np.random.rand() < self.operator_probabilities['mutation']:
                chromosome[i] = np.random.standard_t(df=1)
        return chromosome

    def confusion(self, X, Y):
        '''
        La funcion de confusion para evaluar el modelo.
        '''
        # Convertir los datos a tensores
        X_tensor = torch.from_numpy(X.values.astype(np.float32))
        Y_tensor = torch.from_numpy(Y.values.astype(np.float32))

        # Predecir las etiquetas de los datos de entrada
        outputs = self.model(X_tensor)
        _, predicted = torch.max(outputs.data, 1)

        # Calcular la matriz de confusión
        cm = confusion_matrix(Y_tensor.numpy(), predicted.numpy())

        return cm
```

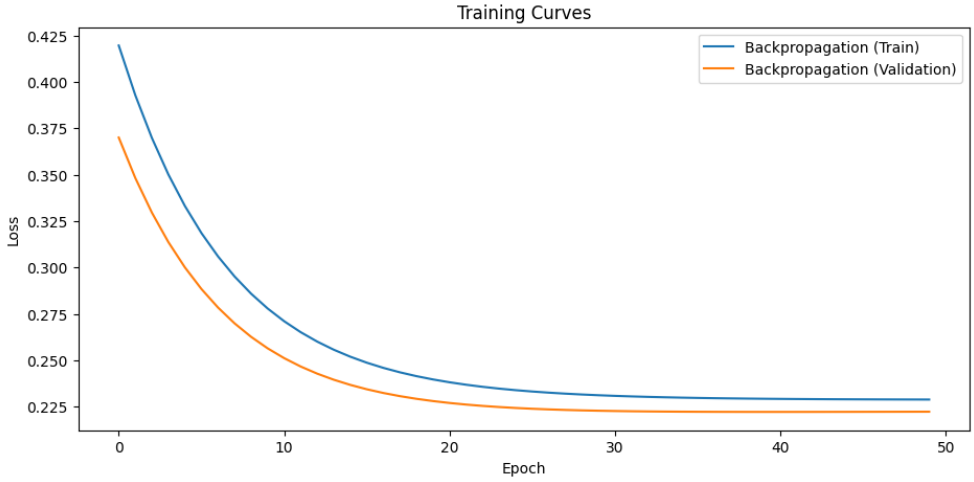
Ahora que tenemos la red neuronal a la que aplicaremos el algoritmo genético, implementamos una red neuronal con un entrenamiento común por el algoritmo de backpropagation, con el fin de comparar ambos algoritmos de entrenamiento.

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(NeuralNet, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size1)
        self.layer2 = nn.Linear(hidden_size1, hidden_size2)
        self.layer3 = nn.Linear(hidden_size2, output_size)
        self.sigmoid = nn.Sigmoid()


    def forward(self, X):
        X = self.sigmoid(self.layer1(X))
        X = self.sigmoid(self.layer2(X))
        X = self.layer3(X)
        return X

    def predict(self, X):
        with torch.no_grad():
            pred = self.forward(torch.tensor(X))  # Convierte X a un tensor de PyTorch
        return pred.numpy().flatten().round().astype(int)

# Entrenamiento con backpropagation
def train_with_backpropagation(X_train, Y_train, X_val, Y_val, input_size, hidden_size1,
↪  hidden_size2, output_size, epochs):
    model = NeuralNet(input_size, hidden_size1, hidden_size2, output_size)
    criterion = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(torch.tensor(X_train.values.astype(np.float32)))  # Convertimos el
        ↪  DataFrame a un numpy array y luego a un tensor de PyTorch
        loss = criterion(outputs.view(-1),
        ↪  torch.tensor(Y_train.values.astype(np.float32)))  # Aplanamos los tensores
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

        # Calculamos la perdida en el conjunto de validacion
        with torch.no_grad():
            val_outputs = model(torch.tensor(X_val.values.astype(np.float32)))  #
            ↪  Convertimos el DataFrame a un numpy array y luego a un tensor de PyTorch
            val_loss = criterion(val_outputs.view(-1),
            ↪  torch.tensor(Y_val.values.astype(np.float32)))  # Aplanamos los tensores
            val_losses.append(val_loss.item())

    return model, train_losses, val_losses
```

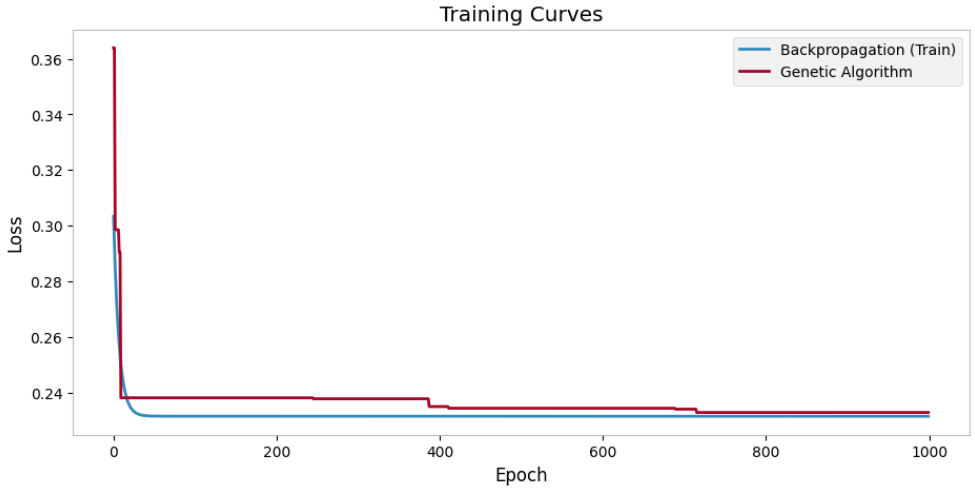
4. Resultados:

Se realizaron una serie de experimentos con el fin de caracterizar el comportamiento de ambos algoritmos, en el primer experimento optimizamos una red con algoritmos genéticos con 10000 ciclos, obteniendo así un error de 0.222, a diferencia del algoritmo de backpropagation, con un error de 0.225. Es interesante debido a que la red neuronal optimizada por medio de backpropagation permaneció en ese error desde la iteración número 50, por ello podemos inferir que el algoritmo de descenso por el gradiente llega rápidamente a disminuir un error, sin embargo, ya que llega a él, se queda estancado y es difícil que disminuya. En comparación con el algoritmo genético, usa en promedio 2 minutos con 30 segundos por cada 100 pasos del entrenamiento y llega de una forma más lenta a disminuir el error, pero obtiene mejores resultados conforme avanzan las iteraciones.

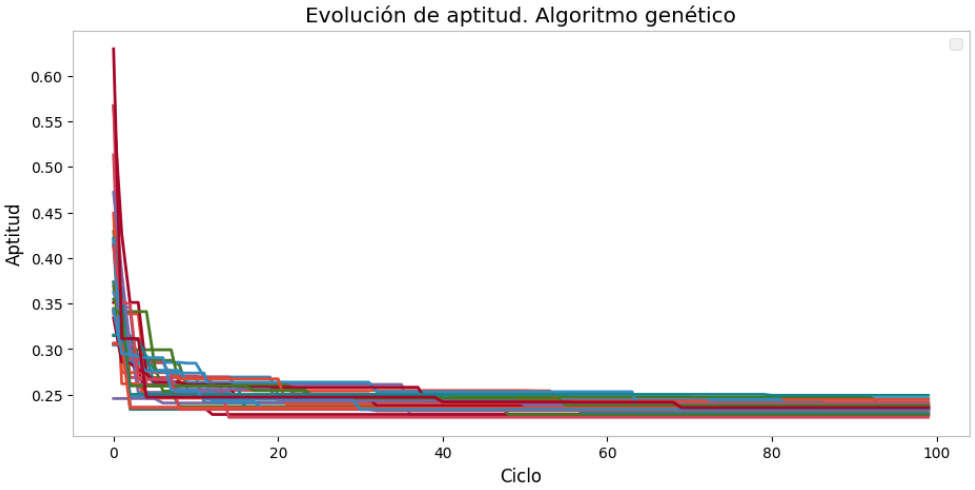


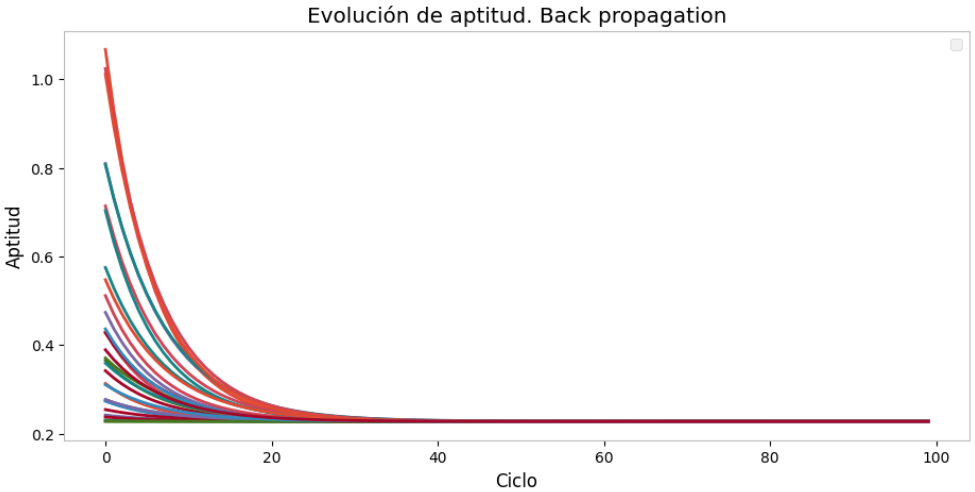


Ahora bien, realizando otro experimento pero ahora con 1000 ciclos de entrenamiento obtuvimos que el error final de la red entrenada por medio de algoritmos genéticos era de 0.2328 mientras que para backpropagation obtuvimos un error de 0.2314. La hipótesis que tenemos al respecto está basada en la cantidad que ciclos que ingresamos al entrenamiento, inferimos que si aumentamos la cantidad de ciclos, los resultados en el error mejoran para el algoritmo genético, mientras que en con gradiente descendente se mantienen.



Como tercer experimento, realizamos 30 entrenamientos con 100 ciclos cada una, para redes neuronales inicializadas desde cero, para no continuar con el entrenamiento que ya tenían, si no para entrenarlas desde un inicio, obteniendo así resultados más acertados para ambos algoritmos. Como primera observación de las gráficas obtenidas, podemos señalar la variabilidad de los resultados del algoritmo genético comparado con la tendencia del algoritmo por gradiente.





Para analizar mejor estos resultados calculamos las estadísticas de los entrenamientos generados. El mejor valor que obtuvimos del algoritmo genético es menor al de backpropagation, sin embargo, en el valor promedio y peor valor, obtuvimos peores resultados.

Algoritmo	Mejor valor	Valor promedio	Peor valor
Algoritmo genético	0.225	0.235	0.249
Backpropagation	0.227	0.227	0.228

Algunos resultados que nos parecieron interesantes fueron las estadísticas de los errores iniciales que obtenemos para cada uno de los algoritmos. Al contrario que con los errores finales, obtenemos que los errores iniciales del algoritmo genético se encuentran más concentrados entre 0 y 1, creemos que esto se debe a la inicialización de pesos con una distribución t.

Algoritmo	Mejor valor	Valor promedio	Peor valor
Algoritmo genético	0.245	0.391	0.629
Backpropagation	0.227	0.474	1.067

Veamos que la llegada a una mejor optimización del entrenamiento no depende directamente de la cantidad de iteraciones, debido a que el resultado mínimo de los entrenamientos de la red por medio de algoritmos genéticos es menor al resultado comparado. Por ello también podemos inferir que es debido a la aleatoriedad a la que se ven expuestos ambos algoritmos.

5. Conclusiones:

El uso de algoritmos genéticos para la optimización de redes neuronales es una buena opción que podemos seguir investigando, por lo encontrado en los experimentos realizados podemos inferir, como mencionamos anteriormente, que este funciona de una manera más óptima para un modelo que queramos entrenar con más iteraciones, posiblemente para redes neuronales más profesionales.
La optimización del error por medio algoritmos genéticos obtienen resultados comparables y hasta mejores a comparación de los algoritmos de gradiente, sin embargo, llegan a ser costos en cuanto a recursos computacionales referentes a los tiempos de iteración. De la misma manera, logramos nuestro objetivo de optimizar una red que nos permitiera programar un juego en terminal que pudiera jugar un ser humano de manera óptima.
Una opción interesante que igualmente plantea el articulo es la mezcla de ambos algoritmos, es decir, en una parte del estancamiento del algoritmo de gradiente descendente entra el algoritmo genético con el fin de evitar óptimos locales, en caso de haber caído en uno. De esta manera obtenemos una red neuronal más optimizada.

Referencias

[1] Articulo sobre la optimizacion de una RN con Computo Evolutivo (Algs Geneticos)

[2] Data Set Tic-Tac-Toe

[3] Pytorch