Inteligencia Artificial 2025

Primer Proyecto

27.febrero.2025

A lo largo del curso trabajaremos con los proyectos desarrollados por el grupo de inteligencia artificial de la Universidad de Berkeley. La idea es aplicar un conjunto de técnicas de IA para resolver diferentes problemas.
Ingresar al sitio de materiales de IA de la Universidad de Berkeley: `http://ai.berkeley.edu/project_overview.html`
`https://inst.eecs.berkeley.edu/~cs188/sp25/projects/`

Antes de comenzar el desarrollo del proyecto conviene revisar la documentación general del proyecto, así como la guía y recomendaciones: `http://ai.berkeley.edu/project_instructions.html`.

Primer Proyecto: Machine Learning.
En este primera entrega, vamos a implementar algoritmos de clasificación como el KNN, el naïve Bayes, perceptrones y redes neuronales, y aplicaremos estos modelos para diversas tareas de clasificación e identificación de lenguajes.

Ingresar al sitio del proyecto **Machine Learning**, y asegurarse de leer las especificaciones
`https://inst.eecs.berkeley.edu/~cs188/sp25/projects/proj5/#question-3-6-points-digit-classification`.

1. **Problema 1**: Perceptrón binario.

   Antes de comenzar esta parte, asegúrese de tener instalados pytorch, numpy y matplotlib.

   En esta parte, implementará un perceptrón binario. Su tarea será completar la implementación de la clase `PerceptronModel` en **models.py**. Para el perceptrón, las etiquetas de salida serán 1 o -1, lo que significa que los puntos de datos $(\mathbf{x}, y)$ del conjunto de datos tendrán y como un `Torch.Tensor` que contiene 1 o -1 como sus entradas.

   Sus tareas son:

   - Completar la función `init(self, dimensions)`. Esto debería inicializar el parámetro de peso en `PerceptronModel`. Tenga en cuenta que aquí, debe asegurarse de que su variable de peso se guarde como un objeto `Parameter()` de dimensión 1 por dimensions. Esto es para que el *autograder*, así como pytorch, reconozcan su peso como un parámetro de su modelo.
   - Implementar el método `run(self, x)`. Esto debería calcular el producto escalar del vector de peso almacenado y la entrada dada, devolviendo un objeto Tensor.
   - Implementar `get_prediction(self, x)`, que debería devolver 1 si el producto escalar no es negativo o -1 en caso contrario.
   - Escribir el método `train(self)`. Esto debería recorrer repetidamente el conjunto de datos y realizar actualizaciones en los ejemplos que están mal clasificados. Cuando se completa un recorrido completo por el conjunto de datos sin cometer ningún error, se ha logrado una precisión de entrenamiento del 100% y el entrenamiento puede finalizar.

   Por suerte, Pytorch facilita la ejecución de operaciones en tensores. Si desea actualizar su peso según alguna dirección de tensor y una magnitud constante, puede hacerlo de la siguiente manera: `self.w += direction * magnitude`. Para esta pregunta, así como para todas las restantes, cada lote devuelto por `DataLoader` será un diccionario en el formato: {`'x':features, 'label':label`} donde la etiqueta es el valor o los valores que queremos predecir en función de las características.

   Para probar su implementación, ejecute el *autograder*: `python autograder.py -q q1`

2. **Problema 2**: Regresión no-lineal.

   Para esta pregunta, se entrenará una red neuronal para aproximar la función $\sin(x)$ en el intervalo $[-2\pi, 2\pi]$. Deberán completar la implementación de la clase `RegressionModel` en **models.py**. Para este problema, una arquitectura relativamente simple debería ser suficiente (consultar los consejos de arquitectura en Consejos de redes neuronales). Usar `mse_loss` como función de pérdida.

   Sus tareas son:

   - Implementar `RegressionModel.__init__` con cualquier inicialización necesaria.
   - Implementar `RegressionModel.run` (`RegressionModel.forward` en Pytorch) para devolver un tamaño de lote por 1 nodo que represente la predicción de tu modelo.
   - Implementar `RegressionModel.get_loss` para devolver una pérdida para las entradas y salidas de destino dadas. Tenga en cuenta que `get_loss` toma un tensor de entrada, no una predicción, por lo que deberá pasar `x` a través de su red antes de usar una función de pérdida.
   - Implementar `RegressionModel.train`, que debería entrenar su modelo utilizando actualizaciones basadas en gradientes.

   Solo hay una única división del conjunto de datos para esta tarea (es decir, solo hay datos de entrenamiento y no hay datos de validación ni conjunto de prueba). Se espera que su modelo tenga una pérdida o error de 0.02 o mejor, promediada en todos los ejemplos del conjunto de datos. Puede usar la pérdida de entrenamiento para determinar cuándo detener el entrenamiento. Tenga en cuenta que el modelo debería tardar unos minutos en entrenarse.

   Para probar su implementación, ejecute el *autograder*: `python autograder.py -q q2`

3. **Problema 3**: Red neuronal.

   Implementar una red neuronal para resolver alguna de las dos opciones a continuación:

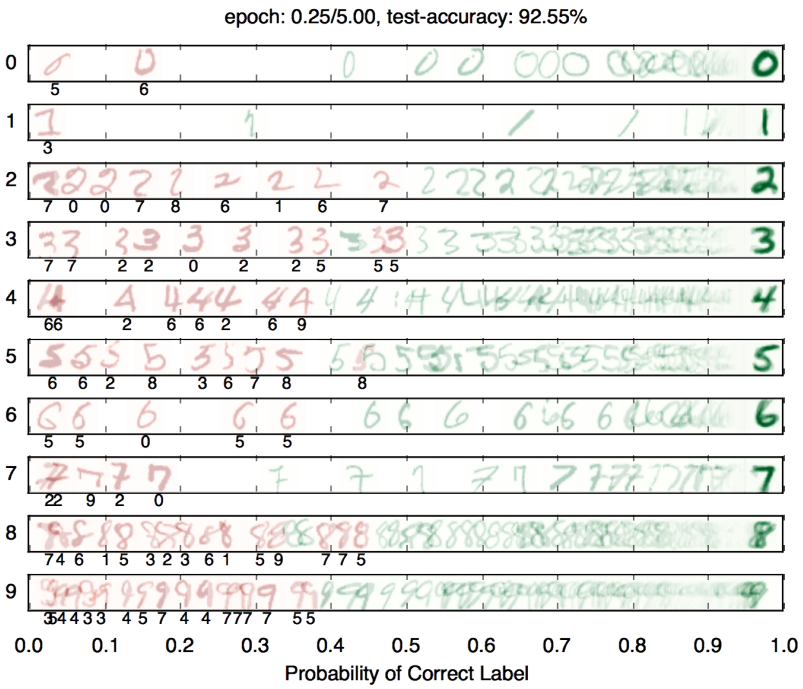
   Opción A (Clasificación de Dígitos):

   En este proyecto, entrenará una red neuronal para clasificar dígitos escritos a mano del conjunto de datos MNIST. Cada dígito tiene un tamaño de 28×28 píxeles, cuyos valores se almacenan en un vector de 784 dimensiones de números de punto flotante. Cada salida que proporcionamos es un vector de 10 dimensiones que tiene ceros en todas las posiciones, excepto un uno en la posición correspondiente a la clase correcta del dígito.

   Complete la implementación de la clase `DigitClassificationModel` en **models.py**. El valor de retorno de `DigitClassificationMod` debe ser un nodo de tamaño `batch_size` $\times 10$ que contenga puntajes, donde los puntajes más altos indican una mayor probabilidad de que un dígito pertenezca a una clase particular (0-9). Debe usar `cross_entropy` como su pérdida. No coloque una activación ReLU en la última capa lineal de la red.

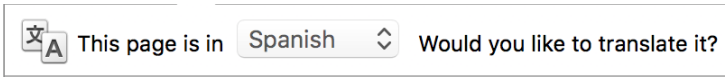
   Para esta pregunta, además de los datos de entrenamiento, también hay datos de validación y un conjunto de prueba. Puede utilizar `dataset.get_validation_accuracy()` para calcular la precisión de validación de su modelo, lo que puede resultar útil al decidir si detener el entrenamiento. El conjunto de prueba será utilizado por el calificador automático. Se espera que su modelo alcance una precisión de al menos el 97% en el conjunto de prueba. en los datos de validación después del entrenamiento durante aproximadamente 5 épocas. Tenga en cuenta que la prueba lo califica en función de la precisión de la prueba, mientras que usted solo tiene acceso a la precisión de la validación; por lo tanto, si su precisión de validación alcanza el umbral del 97%, aún puede fallar la prueba si su precisión de la prueba no alcanza el umbral. Por lo tanto, puede ser útil establecer un umbral de detención ligeramente más alto en la precisión de la validación, como 97.5% o 98%.

Para probar su implementación, ejecute el *autograder*: `python autograder.py -q q3`.



Opción B (Identificación de Lenguajes):

La identificación de idioma es la tarea de averiguar, dado un fragmento de texto, en qué idioma está escrito el texto. Por ejemplo, su navegador podría detectar si ha visitado una página en un idioma extranjero y ofrecerse a traducirla para usted. Aquí hay un ejemplo de Chrome (que utiliza una red neuronal para implementar esta función):



En este proyecto, vamos a construir un modelo de red neuronal pequeño que identifica el idioma de una palabra a la vez. El conjunto de datos consta de palabras en cinco idiomas, como la tabla a continuación:

Palabra	Idioma
discussed	Inglés
eternidad	Español
itseänne	Finlandés
paleis	Holandés
mieszkać	Polaco

Distintas palabras constan de diferentes cantidades de letras, por lo que nuestro modelo necesita tener una arquitectura que pueda manejar entradas de longitud variable. En lugar de una única entrada $\mathbf{x}$ (como en las preguntas anteriores), tendremos una entrada separada para cada carácter de la palabra: $x_0, x_1, \ldots, x_{L-1}$, donde L es la longitud de la palabra. Comenzaremos aplicando una red $f_{inicial}$ que es igual a las redes de los problemas anteriores. Esta acepta una entrada x_0 y calcula un vector de salida h_1 de dimensión d:

$$h_1 = f_{inicial}(x_0).$$

A continuación, combinaremos la salida del paso anterior con la siguiente letra de la palabra, generando un resumen vectorial de las dos primeras letras de la palabra. Para ello, aplicaremos una subred que acepta una letra y genera un estado oculto,

pero que ahora también depende del estado oculto anterior h_1. Denotamos esta subred como f:

$$h_2 = f(h_1, x_1).$$

Este patrón continúa para todas las letras de la palabra de entrada, donde el estado oculto en cada paso resume todas las letras que la red ha procesado hasta el momento:

$$\begin{aligned} h_3 &= f_(h_2, x_2), \\ &\vdots \\ h_L &= f_(h_{L-1}, x_{L-1}). \end{aligned}$$

A lo largo de estos cálculos, la función $f(\cdot, \cdot)$ es la misma parte de la red neuronal y utiliza los mismos parámetros entrenables; $f_{inicial}$ también compartirá algunos de los mismos parámetros que f. De esta manera, los parámetros utilizados al procesar palabras de diferente longitud se comparten. Puede implementar esto utilizando un bucle for sobre las entradas proporcionadas `xs`, donde cada iteración del bucle calcula $f_{inicial}$ ó f.

La técnica descrita anteriormente se denomina red neuronal recurrente (RNN). A continuación, se muestra un diagrama esquemático de la RNN:

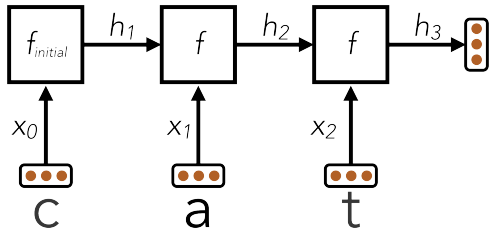


Figure 1: Uso de la RNN para codificar la palabra "cat" en un vector h_3.

Su tarea: Completar la implementación de la clase `LanguageIDModel`. Se espera que su arquitectura logre una precisión de al menos el 81% en el conjunto de pruebas. Se recomienda leer la sección de Batching y Design Tips en la página del proyecto.

Para probar su implementación, ejecute el *autograder*: `python autograder.py -q q4`.

Evaluación : Preguntas 1 y 2: 6 puntos; Pregunta 3: 8 puntos; para un total de 20.
Fecha de Entrega: lunes 17 de marzo.