Intro

Existen muchas decisiones que tomar a la hora de diseñar y configurar un modelo en Machine Learning. Para la mayoría de los casos lo conveniente es resolverlo empíricamente por prueba y error evaluando los resultados.

En este trabajo se implementa una red neuronal para poder identificar números escritos a mano provistos por un dataset denominado MNIST. Estos datos se procesan y formatean para poder ser interpretados por Keras. El modelo se diseña en base a los parámetros, se entrena y se evalúa para obtener como resultado una tasa de error que refleja el éxito del modelo.

Dataset

The MNIST dataset provee un método para la evaluación de machine learning en el problema de clasificación de dígitos escritos a mano. El dataset fue construido con un gran numero de documentos escaneados disponibles por el NIST de donde proviene su denominación Modified NIST or MNIST dataset.

Los dígitos están normalizados y centrados, brindando una entrada excelente para el uso de problemas de machine learning. Cada imagen tiene 28x28 pixeles, 784 en total. Una base estándar de 60000 imágenes es usadas para entrenar el modelo y separadamente 10000 imágenes son usadas para evaluar los resultados.

El reconocimiento de dígitos consta de evaluar la imagen y clasificar estas en uno de los 10 dígitos posibles (0 a 9), el resultado es reportado por la función de evaluación como el Error. Un excelente resultado es aquel que provee una predicción menor al 1%, y los mejores resultados se encuentran cerca del 0.2% con “Large Convolutional Neural Networks”.

Dentro del repositorio de Git de este proyecto la carpeta "1-ImportData” contiene un ejemplo de como importar a Python el dataset del MNIST y muestra alguno de los números importados.

Evaluación

Para este proyecto se creo un repositorio en git en el cual se presentan 5 carpetas que constituyen el proyecto.

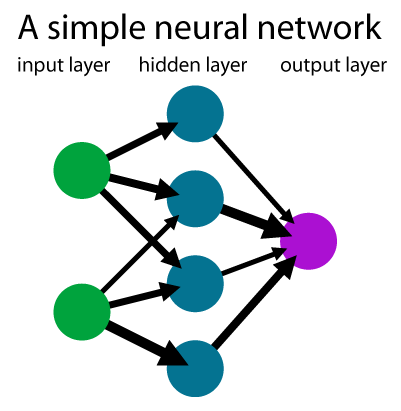
1. ImportData: Describe como importar el dataset del MNIST.
2. MetNN: En base al MNIST se modela y evalua una Simple NN.
3. SimpleCNN: Una evolución del caso anterior usando una red convolucional.
4. LargeCNN: Una mejora agregando mas layers a la red convolucional.
5. NN-Research: El código completo para evaluar todas las redes propuestas y realizar los test que este proyecto plantea.

Dado que el entrenamiento de redes neuronales requiere mucho poder de cómputo y tiempo, se optó por utilizar un servidor remoto y aprovechar para realizar los cálculos en paralelo dado que Keras lo hace de forma automática. Lamentablemente no pudimos utilizar la placa de video para acelerar aún más los cálculos debido a una incompatibilidad de drivers. El equipo utilizado fue un servidor remoto prestado por la Universidad Tecnológica Nacional Facultad Regional Bahía Blanca y consta de las siguientes características:

* CPU: AMD FX-4100 Quad-Core Processor
* RAM: 8Gb DDR3

Simple NN Model

La implementación mas sencilla de una red neuronal es aquella que tiene pocas capaz, esto requiere mucho menos potencia y tiempo de computo. Para ello se consta con solamente una hidden layer con el mismo numero de neuronas que entradas (784). Una primera de función de activación es usada en la hidden layer y otra función de activación de salida es utilizada en la output layer. La bibliografía recomienda utilizar Relu para la activación de hidden layer y softmax para la activación de output layer y transformar la salida en una probabilidad que clasifique entre los 10 dígitos posibles.



Test 1

Basado en las funciones de activación existentes, listadas a continuación, se buscó la mejor combinación de estas.

* relu
* softmax
* tanh
* sigmoid
* hard\_sigoid
* linear
* selu
* softplus
* softsign
* elu

La combinación de todas las funciones de activación dio como resultado 100 combinaciones posibles las cuales se presentan en la tabla del anexo. La función utilizada para replicar este test y los demás se encuentra en el código principal del proyecto.

Los demás parámetros de este test fueron:

* Optimization Function: adam
* Epochs : 5
* Batch Size: 200

Test 1 – Resultados

|  |  |  |
| --- | --- | --- |
| Activation 1 | Activation 2 | Error % |
| selu | softmax | **8,1** |
| selu | sigmoid | **8,24** |
| tanh | softmax | **8,4** |
| softsign | softmax | **8,45** |
| linear | softmax | **8,46** |

Test 2 – Optimization Function

En un segundo test se probaron todas las funciones de optimización disponibles en base a los mejores resultados del primer test dando como resultado 91 combinaciones.

* SGD
* RMSprop
* Adagrad
* Adadelta
* Adam
* Adamax
* Nadam

|  |  |  |  |
| --- | --- | --- | --- |
| Activation 1 | Activation 2 | Optimizer | Error % |
| relu | softmax | nadam | **7,28** |
| selu | softmax | nadam | **7,52** |
| tanh | softmax | nadam | **7,68** |
| linear | softmax | nadam | **7,7** |
| elu | softmax | nadam | **7,72** |

Viendo la tabla completa de resultados se concluye que el mejor optimizador para este problema es Nadam. A partir de esta etapa nos quedamos con las 5 mejor función de activación para la hidden layer (relu, selu, tanh, linear, elu) y las siguientes pruebas se evaluarán siempre en base a estas 5 funciones.

Test 3 – Loss Function

La función Loss es un parámetro requerido para compilar el modelo y optimizar el puntaje del modelo. De las combinaciones posibles resutlaron 60 Tests

* mean\_squared\_error
* mean\_absolute\_error
* mean\_absolute\_percentage\_error
* mean\_squared\_logarithmic\_error
* squared\_hinge
* hinge
* categorical\_hinge
* logcosh
* categorical\_crossentropy
* binary\_crossentropy
* kullback\_leibler\_divergence
* poisson
* cosine\_proximity

|  |  |  |  |
| --- | --- | --- | --- |
| Activation 1 | Activation 2 | Loss Function | Error % |
| relu | softmax | binary\_crossentropy | **1,39%** |
| selu | softmax | binary\_crossentropy | **1,43%** |
| tanh | softmax | binary\_crossentropy | **1,46%** |
| linear | softmax | binary\_crossentropy | **1,46%** |
| elu | softmax | binary\_crossentropy | **1,50%** |

Con los resultados obtenidos se resuelve que la función binary crossentropy es la mejor opción con un margen de más de 5.5% por debajo del error de otras funciones.

Test 4 – Epoch number

El numero de epochs permite …

95 Tests

1 – 19

|  |  |  |  |
| --- | --- | --- | --- |
| N° Epochs | Activation 1 | Activation 2 | Error % |
| 19 | relu | softmax | **0,67%** |
| 18 | relu | softmax | **0,69%** |
| 17 | relu | softmax | **0,73%** |
| 16 | relu | softmax | **0,77%** |
| 14 | relu | softmax | **0,78%** |

Del grafico anterior se ve que usando relu se obtiene una más gran variación al variar el número de epochs, dando como resultado que para 1 epochs relu es la peor métrica y para 19 es la mejor. La cantidad de epochs tiene un impacto directo en el tiempo de computo, el cual se refleja en el siguiente gráfico.

Test 5 – Batch Size

25 – 500

|  |  |  |  |
| --- | --- | --- | --- |
| bath size | Activation 1 | Activation 2 | Error % |
| 25 | relu | softmax | **0,82%** |
| 50 | relu | softmax | **1,00%** |
| 75 | relu | softmax | **1,13%** |
| 25 | selu | softmax | **1,18%** |
| 100 | relu | softmax | **1,19%** |

Optimal Settings

CUDA\_VISIBLE\_DEVICES=1 python main.py simple\_NN nadam binary\_crossentropy 15 15 relu softmax

('Training Time : ', 1073.427568912506)

10000/10000 [==============================] - 1s 74us/step

('Evaluation Time : ', 0.7415130138397217)

Baseline Error: 0.42%

CUDA\_VISIBLE\_DEVICES=1 python main.py simple\_NN nadam binary\_crossentropy 15 10 relu softmax

('Training Time : ', 1589.2177197933197)

10000/10000 [==============================] - 1s 74us/step

('Evaluation Time : ', 0.7362439632415771)

Baseline Error: 0.43%

Acknowledgments

The computer to develop this project was provided by Universidad Tecnologica Nacional (UTN) of Argentina. ́

who is the best in MNIST ? -http://rodrigob.github.io/are\_we\_there\_yet/build/classification\_datasets\_results.html#4d4e495354