Facultad de Ciencias, UNAM

Sistema de clasificación automática de documentos

Reconocimiento de Patrones y Aprendizaje Automatizado

Oscar Andrés Rosas Hernandez

27 de abril de 2020

Índice general

Parte I

Marco Teórico

Capítulo 1

Aprendizaje Supervisado

Definimos al machine Learning como el área que estudia como hacer que las computadoras puedan aprender de manera automática usando experiencias (data) del pasado para predecir el futuro.

1.1. Definición

La mayoría del aprendizaje automático práctico utiliza aprendizaje supervisado. El aprendizaje supervisado es donde se tiene variable(s) de entrada (x) y una variable de salida (Y) y se utiliza un algoritmo para "aprender" la función que mapea la entrada a la salida.

$$Y = f(X) \tag{1.1}$$

El objetivo es aproximar la función tan bien que cuando tenga nuevos datos de entrada (x) pueda predecir las variables de salida (Y) para esos datos.

Se llama aprendizaje supervisado porque el proceso de un algoritmo que aprende del conjunto de datos de capacitación puede verse como un profesor que supervisa el proceso de aprendizaje.

Conocemos las respuestas correctas, el algoritmo realiza predicciones de forma iterativa sobre los datos de entrenamiento y es corregido por el profesor.

Los problemas de aprendizaje supervisados pueden agruparse en problemas de regresión y clasificación.

- Clasificación: Un problema de clasificación es cuando la variable de salida es una categoría, como $rojo$ o $azul$ o $enfermedad$ y $sin_enfermedad$.
- Regresión: Un problema de regresión es cuando la variable de salida es un valor contiuo, como $dolares$ o $peso$ o $probabilidad$.

Durante el entrenamiento, un algoritmo de clasificación recibirá una serie de datos con una categoría asignada. El trabajo de un algoritmo de clasificación es tomar un valor de entrada y asignarle una clase o categoría que se ajuste según los datos de training proporcionados. [1]

1.1.1. Accuracy

Esta dada por:

" De todos nuestros datos, que tanto % clasificamos correctamente. "

En otra palabras:

$$accuracy := \frac{TrueNegative + TruePositive}{All}$$

Ahora, "accuracy" no es siempre la mejor métrica, sobretodo cuando nuestros datos no estan balanceados, es decir cuando en cada una de nuestras clases tenemos una cantidad diferente de datos.

Capítulo 2

Redes Neuronales

2.1. General

Es una práctica común conceptualizar una red neuronal artificial como una neurona biológica. Técnicamente hablando, una red neuronal artificial, como lo expresó Ada Lovelace, es el "cálculo del sistema nervioso".

Una versión muy, muy simplificada del sistema nervioso. Tanto es así que algunos neurocientíficos probablemente han perdido las esperanzas tan pronto como vieron lo sencilla y alejada de la realidad que es.

Las redes neuronales consisten en una capa de entrada, una o varias capas ocultas y una capa de salida. Hay que tener en cuenta que las capas se cuentan desde la primera capa oculta. Cuantas más capas ocultas haya, más compleja es la red.

En general decimos que cuatro o más capas ocultas constituyen una red profunda.

Echemos un vistazo a los componentes comunes compartidos por todas las capas: sinapsis y neuronas.

Las sinapsis toman el valor de la entrada almacenada de la capa anterior y la multiplican por un peso (w). Entonces podemos decir que la tarea principal dentro del aprendizaje profundo es calibrar (o encontrar) los pesos a un valor específico para obtener un resultado preciso.

Es como cuando te estas bañando y hay que mover las dos manijas para encontrar el punto perfecto donde esta la temperetura ideal. Posteriormente, las sinapsis hacia adelante propagan sus resultados a las neuronas.

En general se pueden ver como:

$$Y = b + \sum w_i x_i \tag{2.1}$$

Finalmente tras realizar este calculo y para lograr la no linearlidad que generalmente

buscamos se aplican funciones de activación, entre las mas famosas estan la ReLU y la sigmoide.

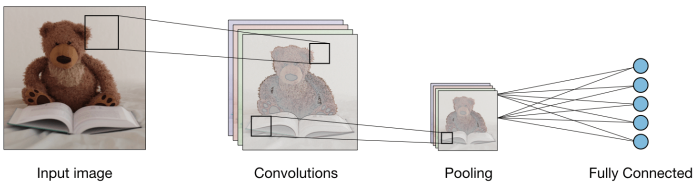
2.2. Softmax

La función de activación sigmoide esta muy bien para la clásica clasificación binaria (2 posibles clases) pero hay que usar algo más si es que buscamos hacer clasificación que tiene mas de dos clases, es aquí donde entra la softmax, la idea básica es distribuir la probabilidad de diferentes clases para que en total sumen 1.

Es decir que toma como entrada n clases y tenemos n salidas pero con la característica esencial de que la suma de todas es igual a una y no son negativas, por lo que ya podemos verlas / interpretarlas como probabilidades.

2.3. Redes Convolucionales

Las redes neuronales convolutionales o CNN estan formadas por 3 partes generalmente:



- Las capas convolucionales usa filtros que realizan operaciones de convolución mientras se escanea / pasa la "imagen" entrada, sus hiperparámetros incluyen el tamaño del filtro y el salto que se da, así como que es lo que pasa en las orillas de la imagen de entrada, la salida se llama mapa de características o mapa de activación.
- Agrupación (pooling): Es una operación que busca la disminución de la resolución, generalmente se aplica después de una capa de convolución; produce cierta invariancia espacial. En particular, el max y mean pooling son tipos especiales de pooling donde se toman el valor máximo y promedio, respectivamente.
- Lineares: Totalmente conectada: Funciona en una entrada aplanada donde cada entrada está conectada a todas las neuronas, para esto tenemos que "espagetificar" nuestra entrada, estas si están presentes, se encuentran hacia el final de la arquitectura y se pueden usar para optimizar objetivos como las probabilidades de una clase.

2.4. Dropout

En pocas palabras, el dropout se refiere a ignorar unidades (es decir, neuronas) durante la fase de entrenamiento de cierto conjunto de neuronas que se elige al azar.

Por "ignorar", quiero decir que estas unidades no se consideran durante un pase particular hacia adelante o hacia atrás.

Más técnicamente, en cada etapa de entrenamiento, los nodos individuales se eliminan de la red con probabilidad 1-p o se mantienen con probabilidad p, de modo que queda una red reducida; Los bordes entrantes y salientes a un nodo abandonado también se eliminan.

2.4.1. ¿Por qué lo necesitamos?

¿Por qué necesitamos literalmente apagar partes de una red neuronal? La respuesta a estas preguntas es para evitar el overfitting.

Una capa totalmente conectada ocupa la mayoría de los parámetros y, por lo tanto, las neuronas desarrollan una codependencia entre ellas durante el entrenamiento, lo que frena el poder individual de cada neurona, lo que lleva a un ajuste excesivo de los datos de entrenamiento.

Para esto es que el dropout cae como anillo al dedo.

Parte II

Sistema de reconocimiento de objetos en imágenes

Capítulo 3

El problema

El problema que elegí fue el predecir, dada una fotografía de una mano haciendo algo algún símbolo (letra) en lenguaje de signos, ser capaz de obtener la letra o símbolo descrito.

3.1. Importancia de resolverlo

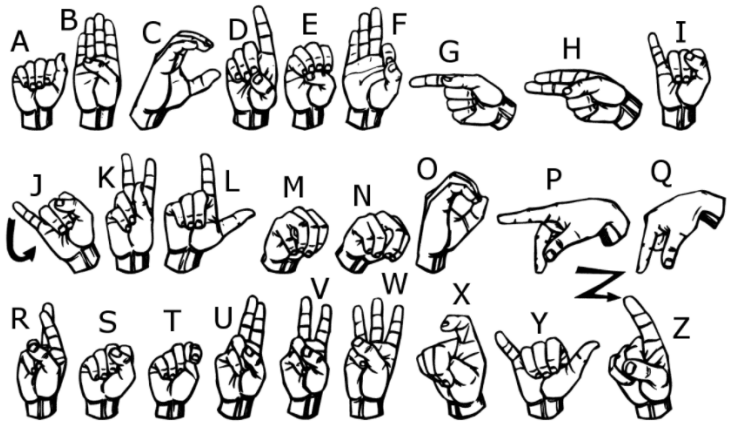
La técnica del reconocimiento de objetos de manera automática en imágenes es muy usada en la industria, por ejemplo en los sistemas de navegación automonos de los autos siendo capaces de encontrar la linea en medio del camino o de encontrar y localizar a las personas dentro de una foto o poder leer los números de tu tarjeta con una foto de la misma.

Este problema en particular creo que muy obvio la importancia: no todos nacemos o tenemos las mismas oportunidades por lo que buscar crear mediante la tecnología sistemas que nos permitan comunicarnos mejor es de vital importancia, sistemas como el que propongo aquí (ayudados de otro que nos permita encontrar manos dentro de una imagen) nos podrían ayudar a crear un sistema de apoyo para traducir de manera totalmente automática una conversación o un video en el que se use el lenguaje de signos. Permitiendo que sin mayor problema para el usuario se pueda crear un "transcript" que nos permita entender de manera escrita lo que una persona esta diciendo, o visto de otra manera podría ser el equivalente que tenemos de nuestros sistemas de dictado automática en el que presionamos un botón y empezamos a hablar, viendo como las palabras aparecen en tiempo real en la pantalla, podríamos usar este sistema que propongo como una pieza clave para poder crear un equivalente en el que alguien hable en lenguaje de señas y se pueda ver una transcripción en tiempo real del contenido, esto por ejemplo también seria muy útil para conferencias y videoconferencias / reuniones online.

Es decir, un buen algoritmo que resuelva este problema podría ayudar de manera pragmática a las personas sordas y con dificultades auditivas a comunicarse mejor utilizando aplicaciones de visión por computadora.

El Instituto Nacional de Sordera y otros Trastornos de las Comunicaciones (NIDCD) indica que el lenguaje de señas americano de 200 años es un lenguaje completo y complejo (del cual los gestos con letras son solo una parte), pero es el idioma principal para muchos norteamericanos sordos.

El ASL es el idioma minoritario líder en los Estados Unidos después de los "cuatro grandes": español, italiano, alemán y francés. Se podría implementar la visión por computadora en una computadora de tablero económica como Raspberry Pi con OpenCV, y algunas Text-to-Speech para permitir aplicaciones de traducción mejoradas y automatizadas.



Capítulo 4

El dataset / Business Understanding

En mi caso el dataset que balanceaba más unos datos limpios con una gran cantidad de información fue uno bastante famoso: MNIST Sign Language:

El conjunto de datos de imagen original MNIST, ya saben el que todos conocemos de los dígitos escritos a mano es un punto de referencia bastante popular para los métodos de aprendizaje automático, pero los investigadores han renovado sus esfuerzos para actualizarlo y desarrollar reemplazos directos que son más desafiantes para la visión por computadora y originales para aplicaciones del mundo real.

El MNIST que se presenta aquí sigue el mismo formato CSV con etiquetas y valores de píxeles en filas individuales. La base de datos de gestos con las manos en lenguaje de señas americano representa un problema de varias clases con 24 clases de letras (excluyendo J y Z que requieren movimiento). Por eso mismo solo vamos a usar 25 clases (podríamos hacer 24, pero dado que la clase que falta es la 9 pensé más coherente dejarlas en 25).

El formato del conjunto de datos está diseñado para coincidir estrechamente con el clásico MNIST. Cada caso de entrenamiento y prueba representa una etiqueta (0-25) como un mapa uno a uno para cada letra alfabética A-Z (y no hay casos para 9 = J o 25 = Z debido a movimientos gestuales). Los datos de entrenamiento (27,455 casos) y los datos de prueba (7172 casos) son aproximadamente la mitad del tamaño del MNIST estándar, pero por lo demás son similares con una fila de encabezado de etiqueta, pixel1, pixel2 ... .pixel784 que representa una sola imagen de 28x28 píxeles con valores de escala de grises entre 0-255.

Los datos originales de la imagen del gesto de la mano representaban a múltiples usuarios que repetían el gesto en diferentes fondos. Los datos del lenguaje de señas MNIST provienen de una gran extensión del pequeño número (1704) de las imágenes en color incluidas como no recortadas alrededor de la región de interés de la mano. Para crear nuevos datos, se usó una tubería de imagen basada en ImageMagick e incluyó el recorte a solo manos, escala de grises, cambio de tamaño y luego crear al menos más

de 50 variaciones para aumentar la cantidad.

La estrategia de modificación y expansión fue filtros ('Mitchell', 'Robidoux', 'Catrom', 'Spline', 'Hermite'), junto con un 5 % de pixelación aleatoria, +/- 15 % de brillo / contraste, y finalmente una rotación de 3 grados. Debido al pequeño tamaño de las imágenes, estas modificaciones alteran efectivamente la resolución y la separación de clases de formas interesantes y controlables.

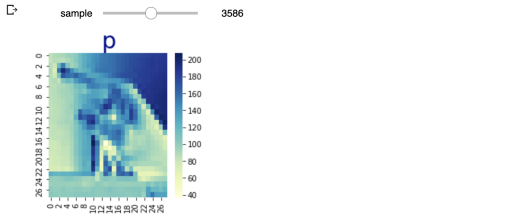
Este conjunto de datos se inspiró en el Fashion-MNIST 2 y la línea de aprendizaje automático para gestos de Sreehari 4.

En mi caso lo descarge de Kaggle y lo pueden encontrar en el siguiente link: `https://www.kaggle.com/datamunge/sign-language-mnist`

El problema es que por la velocidad de mi computadora el entrenamiento lo hice en colab y dado que con mi internet actualmente me hubiera tomado un par de horas descargar y subir a drive el dataset preferí subirlo a github y linkearlo de manera directa en Google Colab.

Podemos explorar el dataset con el siguiente código que hice:

```
[ ] from ipywidgets import interact

    @interact(sample = (0, len(data)))
    def show_image_dataset(sample):
        info, label = data[sample], labels[sample]
        pixels = info.reshape(IMAGE_SIZE, IMAGE_SIZE)

        _, ax = plt.subplots(figsize=(3.5, 3.5))
        sns.heatmap(data=pixels, cmap="YlGnBu", ax=ax)
        plt.title(num_to_letter(label), fontsize=30, color="#00008b")

        plt.show()
```



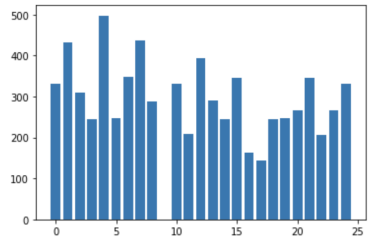
Esto lo hago con ayuda de seaborn y uso un map de color azul, para no ver el clásico blanco y negro, los invito a probarlo en mi código.

Ademas podemos de manera casi inmediata ver como están distribuidos las clases dentro del entrenamiento para ver si es que tenemos un dataset balanceado o no, a lo que yo concluyo que si bien no esta perfectamente balanceado creo que las variaciones son lo suficientemente pequeñas para no preocuparnos mucho de eso.

Podemos explorar el dataset con el siguiente código que hice:

```
[ ] import collections
    counter = collections.Counter(labels)
    print(counter)
    fig, ax = plt.subplots()
    ax.bar(list(counter.keys()), list(counter.values()))
    plt.show()
```

```
Counter({4: 498, 7: 436, 1: 432, 12: 394, 6: 348, 15: 347
```



Capítulo 5

La propuesta

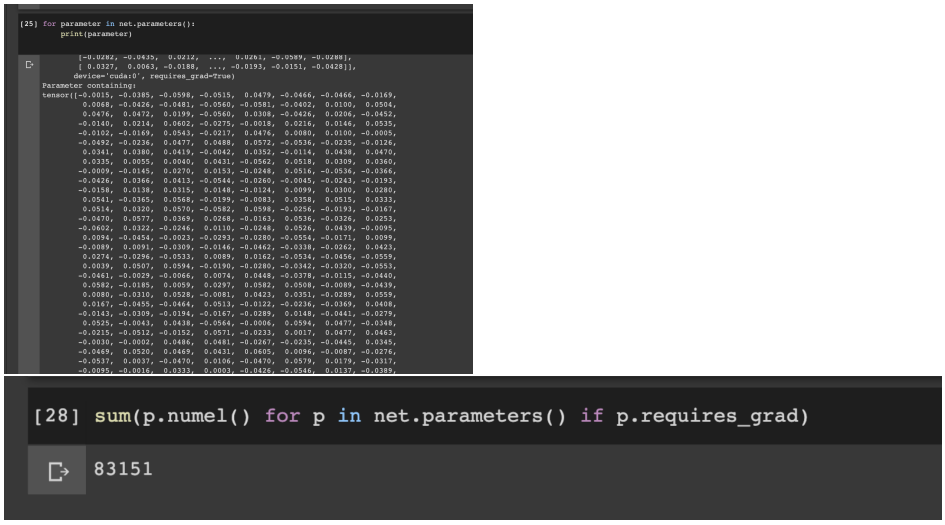
Mi hipótesis sería que usando este dataset y mediante aprendizaje supervisado (redes neuronales, en especial convolucionales y usando algo de dropout) se podría crear un sistema que fuera capaz de clasificar con un gran alto nivel de exactitud (accuracy) (alrededor de un 85 %).

Las redes neuronales convolucionales fueron elegidas por los puntos anteriormente dichos: sobre todo por la gran capacidad que tiene a la hora de clasificar (con la ayuda de capaz lineales) objetos dentro de una imagen, ademas que gracias al uso de Colab puedo tener acceso a una buena GPU con la que podemos hacer muchos más experimentos e iterar bastante rápido, ademas decidí usar dropout después de las redes convolucionales para eliminar algo de la dependencia de las neuronas y evitar con ello el overfitting.

La principal desventaja de esta propuesta, es decir de usar redes neuronales convolucionales es que encontrar el valor ideales para los hiper parámetros es algo que no trivial, ademas de necesitar una gran dataset para tener resultados decentes y finalmente que es al final de cuentas una caja negra que cuesta bastante interpretar, por lo que incluso si se logra el objetivo sera bastante difícil entender que es lo que está haciendo nuestra red neuronal y porque esta eligiendo esta clase en vez de otra.

Ademas de que el cálculo foward incluso ya entrenada no es tan rápido por lo cual toma un equipo con unas buenas características para poder hacer una detección en tiempo real.

Algo también importante es hablar sobre el espacio de hipótesis de esta idea, este depende del número de variables y los valores que puedan tomar esas variables. En nuestro caso veamos todos los parametros o pesos de la propuesta final que codifique para que nos demos una idea de la cantidad de parametros que hay que hacerles "fine tunning":



Como podemos ver, no son pocos, de hecho llegamos cerca de los 100, 000 parametros que vamos a modificar, esto nos da una escala de cuan difícil de resolver es tu problema.

Capítulo 6

La implementación

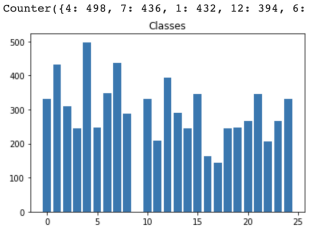
6.1. Preparación de los datos

Lo primero que tuvimos que hacer fue preparar los datos, es decir descargarlos, y poder separar nuestra información entre training, test y validation, vimos el tamaño de cada una:

```
[26]
    base_url = "https://raw.githubusercontent.com/SoyOscarRH/Learning/master/UNAM/MachineLearning/Sign/dataset/"

    data_url = base_url + "sign_mnist_test.csv"
    data_raw = pd.read_csv(data_url, sep=",")
    data, labels = dataframe_to_data(data_raw)
    print("data:", len(data), "elements")

    test_url = base_url + "sign_mnist_test.csv"
    test_validation_data_raw = pd.read_csv(test_url, sep=",")

    n = len(test_validation_data_raw)
    test_data_raw       = test_validation_data_raw.loc[:n//2, :].copy()
    validation_data_raw = test_validation_data_raw.loc[n//2:, :].copy()

    test_data, labels_test = dataframe_to_data(test_data_raw)
    print("test:", len(test_data), "elements")

    validation_data, labels_validation = dataframe_to_data(validation_data_raw)
    print("validation:", len(validation_data), "elements")

    data: 7172 elements
    test: 3587 elements
    validation: 3586 elements
```

Now we are going to look at the number of elements each class has (in training)

```
[27] import collections
     counter = collections.Counter(labels)
     print(counter)
     fig, ax = plt.subplots()
     ax.bar(list(counter.keys()), list(counter.values()))
     plt.title("Classes")
     plt.show()

     Counter({4: 498, 7: 436, 1: 432, 12: 394, 6: 348, 15: 347, 21: 346, 24: 332, 10
```

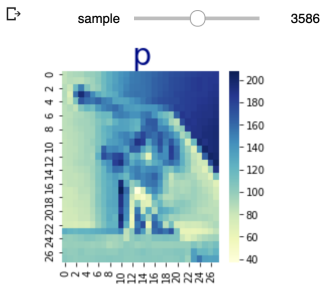



```
[28] IMAGE_SIZE = 28
```

Con esto listo como dije antes podemos ver que tan balanceadas estaban las clases y cree un pequeño widget que te permite explorar fácilmente el dataset.

▾ Interact with the dataset

```
[30] from ipywidgets import interact

     @interact(sample = (0, len(data)))
     def show_image_dataset(sample):
         """ Given an index we show in a heatmap the i'nth image in the dataset"""
         info, label = data[sample], labels[sample]
         pixels = info.reshape(IMAGE_SIZE, IMAGE_SIZE)

         _, ax = plt.subplots(figsize=(3.5, 3.5))
         sns.heatmap(data=pixels, cmap="YlGnBu", ax=ax)
         plt.title(num_to_letter(label), fontsize=30, color="#00008b")

         plt.show()
```



6.2. Creación del clasificador

Esto fue mucho más sencillo de que esperaba, nuestro primer paso semi previo fue el de tomar nuestros datos y volverlos tensores para que pytorch los pueda entender.

We transform our data into tensors and reshape each element so it is actually a 2D matrix

```
[32] data = [pic.reshape(1, IMAGE_SIZE, IMAGE_SIZE) for pic in data]
     validation_data = [pic.reshape(1, IMAGE_SIZE, IMAGE_SIZE) for pic in validation_data]
     test_data = [pic.reshape(1, IMAGE_SIZE, IMAGE_SIZE) for pic in test_data]

     x                       = torch.FloatTensor(data)
     y                       = torch.LongTensor(labels.tolist())

     validation_data_formated = torch.FloatTensor(validation_data)
     validation_labels        = torch.LongTensor(labels_validation.tolist())

     test_data_formated       = torch.FloatTensor(test_data)
     test_labels              = torch.LongTensor(labels_test.tolist())
```

Después declaramos nuestra red neuronal, para ser exactos usando pytorch y para trabajar con esta librería tenemos que implementar dos métodos, un constructor que nos declarara que elementos tiene nuestra arquitectura y una función llamada foward que nos permite expresar el flujo de nuestros datos.

Así que vamos a explicar mas a detalle: Como entrada a nuestros sistema no tendremos una simple imagen en 2d, es decir una matriz, sino que como trabajamos como batch tendremos un arreglo de matrices, nuestro primer paso sera pasar por unas capas convolucionales, como es en blanco y negro usamos solo un canal de entrada y usando un kernel de 3x3 tendremos 10 filtros de salida, estos serán reducidos usando max pooling de 2x2, con esto nuestra información entra a la siguiente capa, dentro de ella repetiremos el mismo proceso ahora incrementando en 10 nuestros filtros de salida.

De tal manera que al final hemos pasado por 3 capas convolucionales y ahora usaremos dropout para mejorar el entrenamiento y evitar el overfitting con una probabilidad bastante alta de un 0.4.

Finalmente hay que tomar las salidas y espagetificarlas, en este caso deje el cálculo tal cual para que sea más obvio porque esa capa conexa requiere esas conexiones, finalmente esta capa tiene 256 neuronas ocultas que son tomadas por la siguiente capa teniendo las 25 clases de salida.

Entre las capaz lineales usamos una simple relu y nuestro trabajo aquí esta listo.

```
[34] class Network(nn.Module):

        def __init__(self):
            super(Network, self).__init__()

            # convolutional layer (sees 28x28x1 image tensor)
            self.conv1    = nn.Conv2d(in_channels=1, out_channels=10, kernel_size=3)
            self.pool1    = nn.MaxPool2d(2)

            # convolutional layer (sees 10x tensor)
            self.conv2    = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3)
            self.pool2    = nn.MaxPool2d(2)

            # convolutional layer (sees 20x tensor)
            self.conv3    = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3)
            self.dropout1 = nn.Dropout2d(0.4)

            self.fc3      = nn.Linear(30 * 3 * 3, 256)
            self.fc4      = nn.Linear(256, 25)

            self.softmax  = nn.LogSoftmax(dim=1)

        def forward(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = self.pool1(x)

            x = self.conv2(x)
            x = F.relu(x)
            x = self.pool2(x)

            x = self.conv3(x)
            x = F.relu(x)
            x = self.dropout1(x)
```

Tambien les damos valor a unos cuantos hiperparametros:

Basic hyperparams

```
[33] epochs = 25
     batch_size = 64
     learning_rate = 0.0015
```

Finalmente y para hacer reproducible a nuestro modelo usaremos una semilla fija para la generación de números aleatorios y ademas usaremos cuda de ser posible, con esto podemos ver de manera más compacta todo lo importante de nuestra red:

Instanciate a network, make it reproducible

```
[35] torch.manual_seed(0)
     np.random.seed(0)
     net = Network()
     if torch.cuda.is_available(): net.cuda()
     print(net)
```

```
Network(
  (conv1): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(20, 30, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout2d(p=0.4, inplace=False)
  (fc3): Linear(in_features=270, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=25, bias=True)
  (softmax): LogSoftmax()
)
```

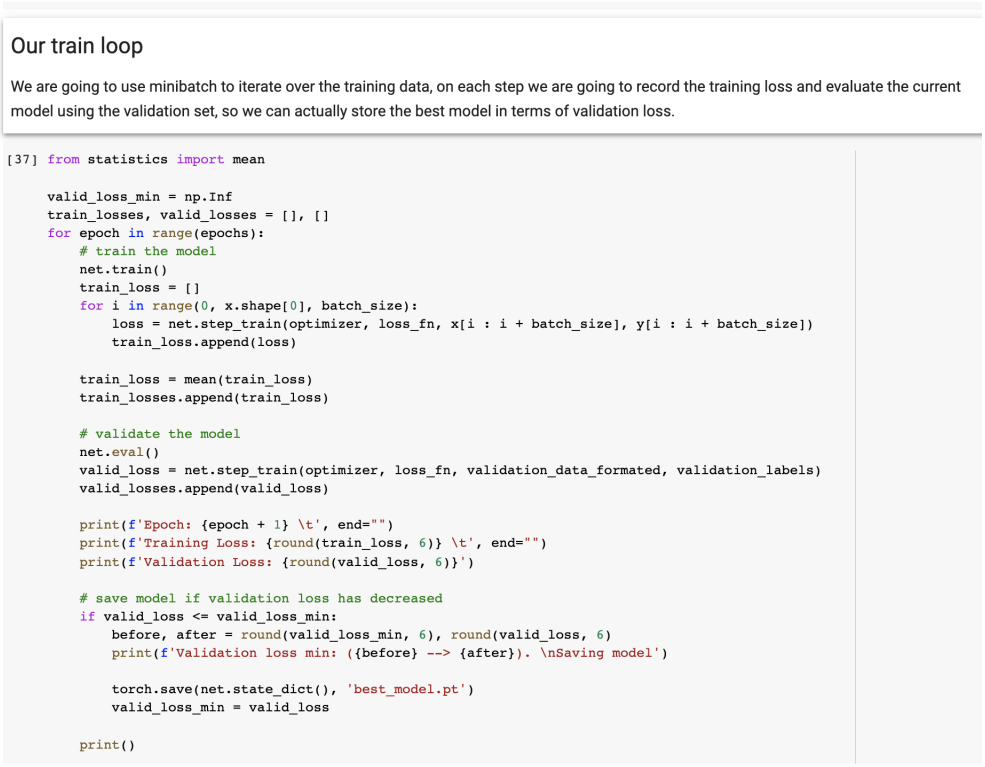
Finalmente y no menos importante usamos tanto una función clásica de optimización como es SGD o stocastic gradient descent con un momentum del 0.7 y como estamos haciendo clasificación con más de dos clases el función de perdida estándar es entropía cruzada.

We are going to use stocastic gradient descent with 0.7 as our default momementum

```
[36] optimizer = optim.SGD(net.parameters(), learning_rate, momentum=0.7)
     loss_fn = nn.CrossEntropyLoss()
```

6.3. Training Loop

Ahora con la arquitectura ya definida vamos a hacer el training loop, donde iremos alimentando a nuestra red neuronal con batches de datos y viendo como es que va la perdida, en este caso use dos perdidas, tanto la de entrenamiento, que espero que siempre baje como la de validación que empezara a subir si es que tenemos un modelo que hace overfitting, para evitarlo cada que alcancemos un mínimo en nuestro error de validación vamos a guardar los pesos del modelo, de tan manera que tras las epochs que pusimos nos podamos quedar con el mejor modelo, no solo el ultimo.



Our train loop

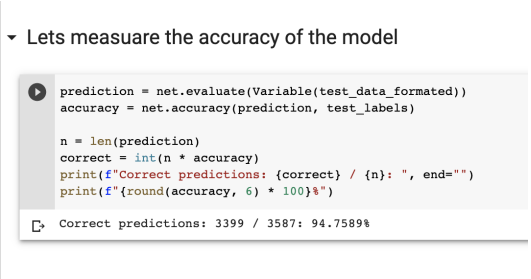
We are going to use minibatch to iterate over the training data, on each step we are going to record the training loss and evaluate the current model using the validation set, so we can actually store the best model in terms of validation loss.

```
[37] from statistics import mean

     valid_loss_min = np.Inf
     train_losses, valid_losses = [], []
     for epoch in range(epochs):
         # train the model
         net.train()
         train_loss = []
         for i in range(0, x.shape[0], batch_size):
             loss = net.step_train(optimizer, loss_fn, x[i : i + batch_size], y[i : i + batch_size])
             train_loss.append(loss)

         train_loss = mean(train_loss)
         train_losses.append(train_loss)

         # validate the model
         net.eval()
         valid_loss = net.step_train(optimizer, loss_fn, validation_data_formated, validation_labels)
         valid_losses.append(valid_loss)

         print(f'Epoch: {epoch + 1} \t', end="")
         print(f'Training Loss: {round(train_loss, 6)} \t', end="")
         print(f'Validation Loss: {round(valid_loss, 6)}')

         # save model if validation loss has decreased
         if valid_loss <= valid_loss_min:
             before, after = round(valid_loss_min, 6), round(valid_loss, 6)
             print(f'Validation loss min: ({before} --> {after}). \nSaving model')

             torch.save(net.state_dict(), 'best_model.pt')
             valid_loss_min = valid_loss

         print()
```

6.4. Evaluación

Lo que hicimos para evaluar al sistema fue:

- Crear otro widget que nos permita explorar el dataset de prueba pero añadir la predicción del modelo en datos que nunca había visto para que podamos entender de manera intuitiva como va
- Creamos también una función auxiliar que toma todo el conjunto de test y nos regresa el accuracy para que podamos compararlo, que en mi caso me siento muy impresionado por el mismo.



En este podemos ver inmediatamente que nuestro sistema cumplio nuestro objetivo, logran un mas de 85 % de accuracy en el conjunto de prueba.

6.4.1. Codigo

Codigo principal

Se encuentra el Google Colab añado un pdf con la version mas actual pero recomiendo de gran medida correrlo por ustedes mismos haciendo click en el link: `https://colab.research.google.com/drive/1ms7-m-3skncTgqDRMo8lTeLjwZne4hAy#offline=true&sandboxMode=true`

SignLanguage

April 27, 2020

0.0.1 Before nothing please check that we are running python3.6+ and have a GPU

[0]:
```
!python3 --version
```

```
Python 3.6.9
```

0.1 Lets import some libraries so we can see the dataset

[0]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```

[0]:
```
def dataframe_to_data(raw):
    '''Simple function that takes a datafram and
       split it into 2 parts, the labels col and everything else'''
    labels = raw['label']
    raw.drop('label', axis=1, inplace=True)

    return raw.values, labels.values
```

We are going to download the dataset, and use pandas to read it and parse it, then we are going to use our costume function to split it into labels and data.

Finally we are going to split the test dataset into validation and actual test

With the split been made we show how many elements each section has

[0]:
```
base_url = "https://raw.githubusercontent.com/SoyOscarRH/Learning/master/UNAM/
↪MachineLearning/Sign/dataset/"

data_url = base_url + "sign_mnist_test.csv"
data_raw = pd.read_csv(data_url, sep=",")
```

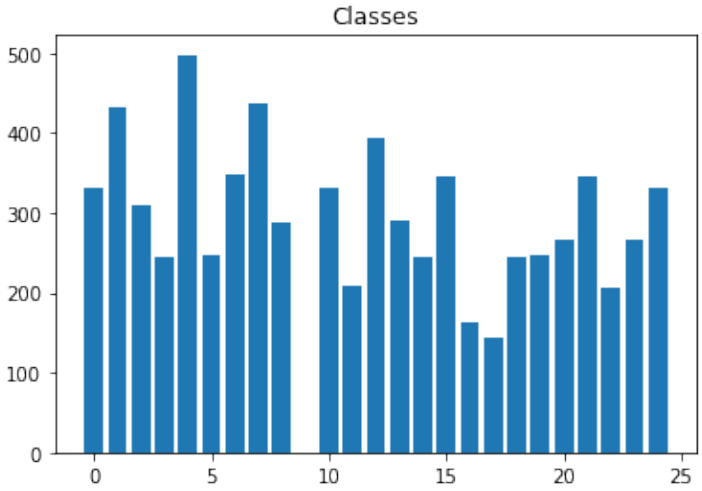
```
data, labels = dataframe_to_data(data_raw)
print("data:", len(data), "elements")

test_url = base_url + "sign_mnist_test.csv"
test_validation_data_raw = pd.read_csv(test_url, sep=",")

n = len(test_validation_data_raw)
test_data_raw       = test_validation_data_raw.loc[:n//2, :].copy()
validation_data_raw = test_validation_data_raw.loc[n//2:, :].copy()

test_data, labels_test = dataframe_to_data(test_data_raw)
print("test:", len(test_data), "elements")

validation_data, labels_validation = dataframe_to_data(validation_data_raw)
print("validation:", len(validation_data), "elements")
```

```
data: 7172 elements
test: 3587 elements
validation: 3586 elements
```

Now we are going to look at the number of elements each class has (in training)

[0]:
```
import collections
counter = collections.Counter(labels)
print(counter)
fig, ax = plt.subplots()
ax.bar(list(counter.keys()), list(counter.values()))
plt.title("Classes")
plt.show()
```

```
Counter({4: 498, 7: 436, 1: 432, 12: 394, 6: 348, 15: 347, 21: 346, 24: 332, 10:
331, 0: 331, 2: 310, 13: 291, 8: 288, 23: 267, 20: 266, 19: 248, 5: 247, 14:
246, 18: 246, 3: 245, 11: 209, 22: 206, 16: 164, 17: 144})
```



[0]:
```
IMAGE_SIZE = 28
```

[0]:
```
def num_to_letter(num: int) -> str:
    """ siple helper function to take an int and transform it into a letter so we can show it to you"""
    start = ord('a')
    return chr(num + start)

examples = [num_to_letter(i) for i in range(26)]
print(examples)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

0.2 Interact with the dataset

[0]:
```
from ipywidgets import interact

@interact(sample = (0, len(data)))
def show_image_dataset(sample):
    """ Given an index we show in a heatmap the i'nth image in the dataset"""
    info, label = data[sample], labels[sample]
    pixels = info.reshape(IMAGE_SIZE, IMAGE_SIZE)

    _, ax = plt.subplots(figsize=(3.5, 3.5))
    sns.heatmap(data=pixels, cmap="YlGnBu", ax=ax)
```

```
    plt.title(num_to_letter(label), fontsize=30, color="#00008b")

    plt.show()
```

```
interactive(children=(IntSlider(value=3586, description='sample', max=7172), Output()), _dom_c
```

1 Declare the arquitecture of the neuronal network

[0]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
```

We transform our data into tensors and reshape each element so it is actually a 2D matrix

[0]:
```
data = [pic.reshape(1, IMAGE_SIZE, IMAGE_SIZE) for pic in data]
validation_data = [pic.reshape(1, IMAGE_SIZE, IMAGE_SIZE) for pic in␣
↪validation_data]
test_data = [pic.reshape(1, IMAGE_SIZE, IMAGE_SIZE) for pic in test_data]

x                       = torch.FloatTensor(data)
y                       = torch.LongTensor(labels.tolist())

validation_data_formated = torch.FloatTensor(validation_data)
validation_labels        = torch.LongTensor(labels_validation.tolist())

test_data_formated       = torch.FloatTensor(test_data)
test_labels              = torch.LongTensor(labels_test.tolist())
```

Basic hyperparams

[0]:
```
epochs = 25
batch_size = 64
learning_rate = 0.0015
```

[0]:
```
class Network(nn.Module):

    def __init__(self):
        super(Network, self).__init__()

        # convolutional layer (sees 28x28x1 image tensor)
        self.conv1    = nn.Conv2d(in_channels=1, out_channels=10, kernel_size=3)
        self.pool1    = nn.MaxPool2d(2)

        # convolutional layer (sees 10x tensor)
        self.conv2    = nn.Conv2d(in_channels=10, out_channels=20,␣
↪kernel_size=3)
```

```
        self.pool2    = nn.MaxPool2d(2)

        # convolutional layer (sees 20x tensor)
        self.conv3    = nn.Conv2d(in_channels=20, out_channels=30,␣
↪kernel_size=3)
        self.dropout1 = nn.Dropout2d(0.4)

        self.fc3      = nn.Linear(30 * 3 * 3, 256)
        self.fc4      = nn.Linear(256, 25)

        self.softmax  = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = F.relu(x)
        x = self.dropout1(x)

        x = x.view(-1, self.fc3.in_features)

        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))

        return self.softmax(x)


    def evaluate(self, x):
      """Take a element through the nn and out the best candidate class"""
      if torch.cuda.is_available(): x = x.cuda()
      output = self(x)
      return torch.max(output.data, 1)[1]


    def step_train(self, optimizer, loss_fn, x, y):
        """ Do one step in training, return the loss from back prop"""
        x = Variable(x)
        y = Variable(y)
        if torch.cuda.is_available(): x, y = x.cuda(), y.cuda()

        optimizer.zero_grad()
```

5

```
        loss = loss_fn(self(x), y)
        loss.backward()
        optimizer.step()

        return loss.item()


    def accuracy(self, predictions, labels) -> float:
        """ given 2 iterables representing the actual labels and the prediction␣
↪of our system we output
            our estimated accuracy"""
        correct = 0
        for prediction, label in zip(predictions, labels):
            if prediction == label: correct += 1

        return correct / len(predictions)
```

Instanciate a network, make it reproducible

[0]:
```
torch.manual_seed(0)
np.random.seed(0)
net = Network()
if torch.cuda.is_available(): net.cuda()
print(net)
```

```
Network(
  (conv1): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv2d(20, 30, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout2d(p=0.4, inplace=False)
  (fc3): Linear(in_features=270, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=25, bias=True)
  (softmax): LogSoftmax()
)
```

We are going to use stocastic gradient descent with 0.7 as our default momementum

[0]:
```
optimizer = optim.SGD(net.parameters(), learning_rate, momentum=0.7)
loss_fn = nn.CrossEntropyLoss()
```

1.1 Our train loop

We are going to use minibatch to iterate over the training data, on each step we are going to record the training loss and evaluate the current model using the validation set, so we can actually store the best model in terms of validation loss.

[0]:
```
from statistics import mean

valid_loss_min = np.Inf
train_losses, valid_losses = [], []
for epoch in range(epochs):
    # train the model
    net.train()
    train_loss = []
    for i in range(0, x.shape[0], batch_size):
        loss = net.step_train(optimizer, loss_fn, x[i : i + batch_size], y[i :␣
↪i + batch_size])
        train_loss.append(loss)

    train_loss = mean(train_loss)
    train_losses.append(train_loss)

    # validate the model
    net.eval()
    valid_loss = net.step_train(optimizer, loss_fn, validation_data_formated,␣
↪validation_labels)
    valid_losses.append(valid_loss)

    print(f'Epoch: {epoch + 1} \t', end="")
    print(f'Training Loss: {round(train_loss, 6)} \t Validation Loss:␣
↪{round(valid_loss, 6)}')

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        before, after = round(valid_loss_min, 6), round(valid_loss, 6)
        print(f'Validation loss min: ({before} --> {after}). \nSaving model')

        torch.save(net.state_dict(), 'best_model.pt')
        valid_loss_min = valid_loss

    print()
```

```
Epoch: 1        Training Loss: 3.12588   Validation Loss: 2.8588
Validation loss min: (inf --> 2.8588).
Saving model

Epoch: 2        Training Loss: 2.708383          Validation Loss: 2.434011
Validation loss min: (2.8588 --> 2.434011).
Saving model

Epoch: 3        Training Loss: 2.260573          Validation Loss: 1.87094
Validation loss min: (2.434011 --> 1.87094).
Saving model
```

```
Epoch: 4        Training Loss: 1.803474         Validation Loss: 1.292004
Validation loss min: (1.87094 --> 1.292004).
Saving model

Epoch: 5        Training Loss: 1.408076         Validation Loss: 0.924011
Validation loss min: (1.292004 --> 0.924011).
Saving model

Epoch: 6        Training Loss: 1.121076         Validation Loss: 0.714124
Validation loss min: (0.924011 --> 0.714124).
Saving model

Epoch: 7        Training Loss: 0.829007         Validation Loss: 0.504813
Validation loss min: (0.714124 --> 0.504813).
Saving model

Epoch: 8        Training Loss: 0.67426   Validation Loss: 0.39776
Validation loss min: (0.504813 --> 0.39776).
Saving model

Epoch: 9        Training Loss: 0.581228         Validation Loss: 0.367483
Validation loss min: (0.39776 --> 0.367483).
Saving model

Epoch: 10       Training Loss: 0.511361         Validation Loss: 0.351019
Validation loss min: (0.367483 --> 0.351019).
Saving model

Epoch: 11       Training Loss: 0.487887         Validation Loss: 0.331776
Validation loss min: (0.351019 --> 0.331776).
Saving model

Epoch: 12       Training Loss: 0.451158         Validation Loss: 0.324673
Validation loss min: (0.331776 --> 0.324673).
Saving model

Epoch: 13       Training Loss: 0.416918         Validation Loss: 0.324102
Validation loss min: (0.324673 --> 0.324102).
Saving model

Epoch: 14       Training Loss: 0.414308         Validation Loss: 0.322166
Validation loss min: (0.324102 --> 0.322166).
Saving model

Epoch: 15       Training Loss: 0.400752         Validation Loss: 0.320976
Validation loss min: (0.322166 --> 0.320976).
Saving model
```

8

```
Epoch: 16       Training Loss: 0.393943         Validation Loss: 0.319565
Validation loss min: (0.320976 --> 0.319565).
Saving model

Epoch: 17       Training Loss: 0.379305         Validation Loss: 0.321035

Epoch: 18       Training Loss: 0.37971   Validation Loss: 0.318718
Validation loss min: (0.319565 --> 0.318718).
Saving model

Epoch: 19       Training Loss: 0.367212         Validation Loss: 0.318221
Validation loss min: (0.318718 --> 0.318221).
Saving model

Epoch: 20       Training Loss: 0.360242         Validation Loss: 0.319724

Epoch: 21       Training Loss: 0.365743         Validation Loss: 0.318356

Epoch: 22       Training Loss: 0.359045         Validation Loss: 0.318892

Epoch: 23       Training Loss: 0.354851         Validation Loss: 0.317742
Validation loss min: (0.318221 --> 0.317742).
Saving model

Epoch: 24       Training Loss: 0.35308   Validation Loss: 0.318673

Epoch: 25       Training Loss: 0.317728         Validation Loss: 0.216968
Validation loss min: (0.317742 --> 0.216968).
Saving model
```

Recover the "best" model

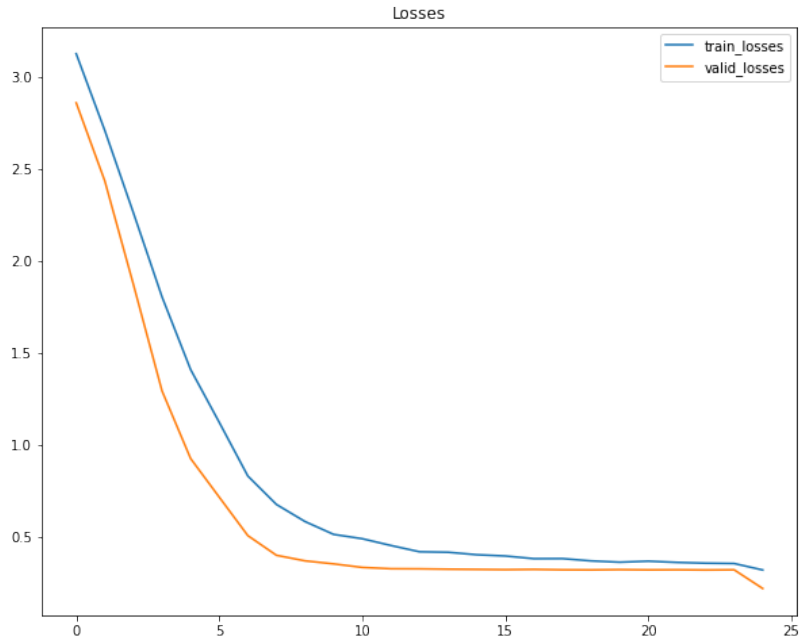
[0]:
```
net.load_state_dict(torch.load('best_model.pt'))
```

[0]:
```
<All keys matched successfully>
```

Show more visually the change in loss

[0]:
```
%matplotlib inline

plt.figure(figsize=(10,8))
plt.plot(train_losses, label="train_losses")
plt.plot(valid_losses, label="valid_losses")
plt.legend()
plt.title("Losses")
```

[0]:
```
Text(0.5, 1.0, 'Losses')
```



[0]:
```
## Switch to evaluate mode, no more backprop and no dropout
net.eval()
```

[0]:
```
Network(
  (conv1): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv2d(20, 30, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout2d(p=0.4, inplace=False)
  (fc3): Linear(in_features=270, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=25, bias=True)
  (softmax): LogSoftmax()
)
```

[0]:
```
@interact(sample = (0, len(test_data)))
def show_image_dataset(sample):
```

```
    """Same helper iteractive tool but now we evaluate our model and get the␣
↪actual prediction"""
    info, label  = test_data[sample], labels_test[sample]
    pixels       = info.reshape(IMAGE_SIZE, IMAGE_SIZE)

    input_pixels    = info.reshape(1, IMAGE_SIZE, IMAGE_SIZE).tolist()
    predicted_label = net.evaluate(torch.FloatTensor([input_pixels]))

    _, ax = plt.subplots(figsize=(3.5, 3.5))
    sns.heatmap(data=pixels, cmap="YlGnBu", ax=ax)

    actual    = f"{num_to_letter(label)}  actual"
    predicted = f"{num_to_letter(predicted_label)}  predicted"
    plt.title(actual + "\n" + predicted, fontsize=20, color="#00008b",␣
↪loc="left")

    plt.show()
```

```
interactive(children=(IntSlider(value=1793, description='sample', max=3587), Output()), _dom_c
```

1.2 Lets measuare the accuracy of the model

[0]:
```
prediction = net.evaluate(Variable(test_data_formated))
accuracy = net.accuracy(prediction, test_labels)

n = len(prediction)
correct = int(n * accuracy)
print(f"Correct predictions: {correct} / {n}: ", end="")
print(f"{round(accuracy, 6) * 100}%")
```

```
Correct predictions: 3367 / 3587: 93.86670000000001%
```

6.5. Experimentos

En general, esta idea para la implementación la base en implementaciones pasadas que había hecho para clasificar el dataset de CIFRA-10 a este le retire una capa convolucional y pase de un kernel de 5x5 a uno de 3x3 pensando en que con esta baja resolución y en blanco y negro no quería perder detalles importantes muy rápidamente, todo esto ayudo a mejorar notablemente el desempeño.

El siguiente paso fue colocar potencias de dos de tal manera de que optimizara el cache de la mejor manera, es decir usando batch de 64 y teniendo 256 neuronas ocultas. Esto termino sin afectar de una manera significa el resultado.

Finalmente y lo que si afecto en gran medida fue la cantidad de iteraciones que daba y el learning_rate, al principio al entrenarla en mi laptop lo máximo que podía esperar eran 5 iteraciones pero bajando el learning rate y dando espacio a más iteraciones fue la clave para pasar el 83 % al 95 %. :D

Otro experimento que intente era cambiar la probabilidad del dropout, pero tampoco obtuve mejoras significativas por lo que lo deje en el que daba el mejor resultado que es alrededor del 0.5.

6.6. Posibles mejoras a futuro

- Podríamos usar menos capas profundas para poder estudiar de mejor manera los filtros resultantes, o en general creo que seria una buena idea usar mapas de activación para ver que están haciendo los filtros
- Probar este tipo de arquitectura en un dataset con más definición y al final con color de tan manera que podemos entender si el color de la piel juega un papel aquí
- Podríamos probar con pequeños gifs, es decir secuencias de fotogramas para poder representar las últimas dos letras faltantes. (seria interesante probar redes recurrente + convolucionales).

6.7. Conclusión

Gracias a los resultados vimos que es posible crear un algoritmo con la ayuda de las redes neuronales que nos permite clasificar con un gran desempeño la letra dada una imagen de una mano realizando ese símbolo en lenguaje americano de señas.

Bibliografía

[1] *Aidan Wilson*, Sep 29, 2019
https://towardsdatascience.com/a-brief-introduction-to-supervised-learning

[2] Apr 27, 2020
https://stanford.edu/~shervine/teaching/cs-230/
cheatsheet-convolutional-neural-networks