Barcelona School of Economics

Computational Machine Learning II - Final Project

American Sign Language Detection - Iván Aguilar & Juan Picciotti

Introduction

American Sign Language (ASL) is a natural language that serves as the predominant sign language of Deaf communities in the United States and most of Anglophone Canada. ASL is a complete and organized visual language that is expressed by facial expression as well as movements and motions with the hands.

In this project we will focus on two approaches to the ASL detection:

1. In the first part, we will follow the YOLO proposed project with all its requirements, which includes reading reference papers, applying YOLO to a custom dataset and discussing results.
2. In the second part we demonstrate how to build from scratch a couple of basic neural networks and compare basic architectures, differences and results for each of them.

We believe that understanding YOLO and basic architectures fully will provide the most coverage in the learning of the inner workings of convolutional neural networks and will allow us to sketch a timeline on the high speed of progress in this area of deep learning.

All the code relevant to complete this exercise was completed in python and is available in two notebooks, corresponding to each part of the project.

YOLOv5 to detect ASL:

YOLO: how does it work?

Brief clarification: this section is based on the papers describing YOLO versions 1 and 2. Note that the actual model we used is YOLOv5, which builds from those. YOLOv5 was developed entirely on pytorch and has its associated paper still to be published. There is some controversy related to the name of the model since the original authors have stepped away from the project, but that is not within the scope of this work.

You Only Look Once: Unified, Real-Time Object Detection

YOLOv1 considers object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. One convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes.

This unified model allows for fast and globally reasoned predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance. This results in highly generalizable models.

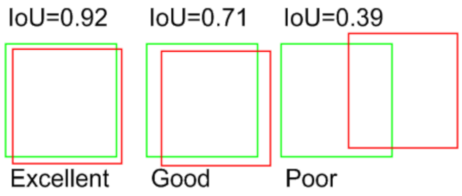
YOLO divides the input image into an S × S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts:

- *B* bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts.

  Formally, the confidence is defined as:

  $$P(object)\ *\ IOU_{truth\ vs\ pred}$$

  With IOU being the intersection over the union of the true box vs the predicted one:



- *C* conditional class probabilities:

  $$P(Class_i|\ Object)$$

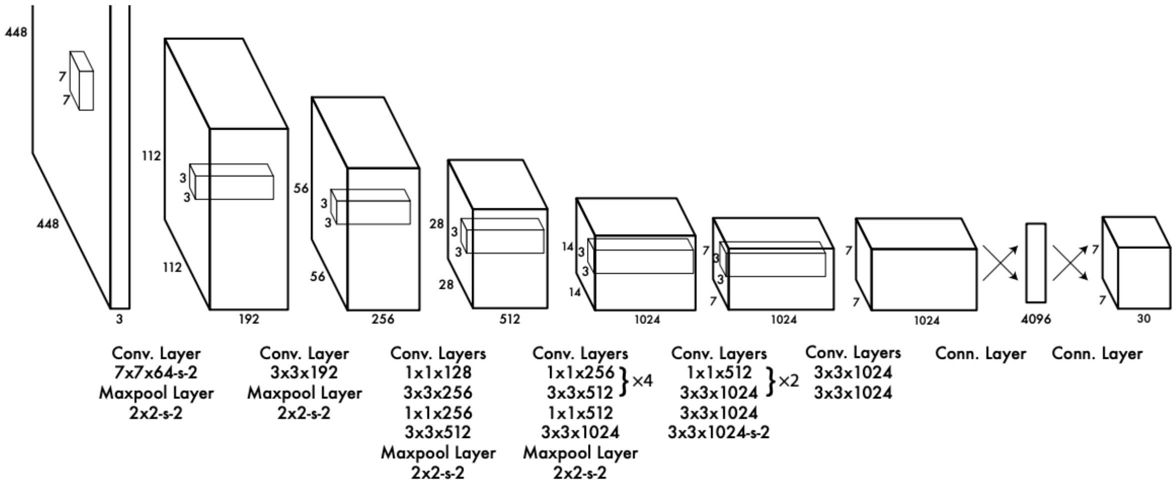
  Only one set of class probabilities is predicted per grid cell, regardless of the number of boxes B. At test time, the conditional class probabilities and the individual box confidence predictions are multiplied, which defines the defines class-specific confidence scores for each box:

  $$P(Class_i|\ Object)\ \ *\ \ P(object)\ \ *\ \ IOU_{truth\ vs\ pred} = P(Class_i)\ *\ \ IOU_{truth\ vs\ pred}$$

  These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.

The initial convolutional layers of the network extract features from the image while the fully connected layers predict the output probabilities and coordinates. The network architecture is inspired by the GoogLeNet model for image classification, having:

- 24 convolutional layers followed by
- 2 fully connected layers.
- Instead of the inception modules used by GoogLeNet, YOLOv1 uses 1 × 1 reduction layers followed by
- 3 × 3 convolutional layers.



The convolutional layers are pretrained on the ImageNet classification task at half the resolution (224 × 224 input image). For detection, the resolution is 448 x 448, since this task requires finer-grained data. The final output of the network is the 7 × 7 × 30 tensor of predictions for both class probabilities and bounding box coordinates. The bounding box width and height are normalized by the image width and height so that they fall between 0 and 1. A linear activation function is employed for the final layer and all other layers use the following leaky rectified linear activation.

As for loss function, YOLOv1, unlike classifier-based approaches, is trained on a loss function that directly corresponds to detection performance and the entire model is trained jointly:

$$\begin{aligned}
&\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

Note that:

- It uses a sum-squared error loss, which although does not perfectly align with the goal of maximizing average precision, it is easy to optimize.

- Many grid cells in images do not contain any object. This pushes the “confidence” scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, which is why λ_{coord} = 5 (penalizes more the loss associated with the bounding box) and λ_{noobj} = 0.5 (reduces the loss from confidence prediction for boxes that do not contain objects) are introduced.

- A square root is introduced in the bounding box width and height instead of the width and height directly, so that errors in large boxes and small boxes are weighted differently (small deviations in large boxes matter less than in small boxes).

To train the network, in the original paper they used:

- 135 epochs on the training and validation data sets from PASCAL VOC 2007 and 2012.

- Batch size of 64, a momentum of 0.9 and a decay of 0.0005.

- Learning rate scheduled as follows:

  - They start with a smaller learning rate for the first epochs, which they gradually increase. This avoids divergence due to unstable gradients.
  - From epoch 75 onwards, they reduce in discrete steps the learning rate for fine tuning.

- To avoid overfitting, dropout and extensive data augmentation is employed.

YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class, this helps mitigate multiple detections of the same object, but limits the number of nearby objects that the model can predict.

Compared to other models that were state of the art at the moment the paper was released, which were based either on a sliding window approach or region proposals, YOLO’s main source of error is incorrect localizations.

Localization errors account for more of YOLO’s errors than all other sources combined. Fast R-CNN makes much fewer localization errors but far more background errors.

Combining the models, since they make different types of errors, results in an effective boosting of Fast R-CNN’s performance.

YOLO9000: Better, Faster, Stronger

In this paper, the authors build upon the original YOLO model. In this iteration, it uses a hierarchical view of object classification that allows to combine distinct datasets together. With this approach, they generated a model that was able to detect over 9000 object categories.

The aim here was to reduce the number of localization errors, thus they focused mainly on improving recall and localization while maintaining classification accuracy. To do so, they considered the following factors:

- Batch Normalization: by adding batch normalization on all of the convolutional layers in YOLO, they got more than 2% improvement in mAP. Batch normalization also helps regularize the model, which allows the authors to remove dropout from the model without overfitting.

- High Resolution Classifier: as commented in the previous section, which summarizes the first YOLO paper, the original model trains the classifier network at 224 × 224 and increases the resolution to 448 for detection. This means the network has to simultaneously switch to learning object detection and adjust to the new input resolution. For YOLOv2, the authors first fine tune the classification network at the full 448 × 448 resolution for 10 epochs on ImageNet. This gives the network time to adjust its filters to work better on higher resolution input. Then they fine tune the resulting network on detection. This high resolution classification network gives us an increase of almost 4% mAP

- Convolutional With Anchor Boxes: the authors remove the fully connected layers from YOLO and use anchor boxes to predict bounding boxes. They eliminate one pooling layer to make the output of the network's convolutional layers higher resolution.

  Additionally, they shrink the network to operate on 416 input images instead of 448×448. By doing this, they get an odd number of locations in our feature map (13 x13 after YOLO's convolutional layers downsample the image by a factor of 32), so there is a single center cell. Objects, especially large objects, tend to occupy the center of the image so it is good to have a single location right at the center to predict these objects instead of four.

  By using Anchor Boxes, the authors face two issues with YOLO:

  - Dimension Clusters: the box dimensions are hand picked. The network can learn to adjust the boxes appropriately but better priors for the network to start with make it easier for the network to learn to predict good detections. Instead of choosing priors by hand, the authors propose running k-means clustering on the training set bounding boxes to automatically find good priors.

    The distance metric for k-means was: $d(box,\ centroid)\ =\ 1\ -\ IOU(box,\ centroid)$

    They ran k-means for various values of k and plot the average IOU with the closest centroid. They then chose k considering the tradeoff between model complexity and high recall. The cluster centroids are significantly different from hand-picked anchor boxes. The result was fewer short, wide boxes and more tall, thin boxes

  - Direct location prediction: model instability, especially during early iterations. Most of the instability comes from predicting the (x, y) locations for the box.

    To solve this, instead of predicting absolute offsets they predict location coordinates relative to the location of the grid cell. This bounds the ground truth to fall between 0 and 1.

- Multi-Scale Training: since the model only uses convolutional and pooling layers it can be resized at will. To make YOLOv2 robust to running on images of different sizes the authors trained the model using different input image sizes. Every 10 batches our network randomly chooses a new image dimension size. This regime forces the network to learn to predict well across a variety of input dimensions. This means the same network can predict detections at different resolutions.

- Training for classification: the network was trained on the standard ImageNet 1000 class classification dataset for 160 epochs using stochastic gradient descent with a starting learning rate of 0.1, polynomial rate decay with a power of 4, weight decay of 0.0005 and momentum of 0.9 using the Darknet neural network framework (in YOLOv5 replaced by pytorch). During training standard data augmentation tricks including random crops, rotations, and hue, saturation, and exposure shifts were used.

- Training for detection: the authors modify the network for detection by removing the last convolutional layer and instead adding on three 3 × 3 convolutional layers with 1024 filters each followed by a final 1 × 1 convolutional layer with the number of outputs we need for detection. They also add a passthrough layer from the final 3 × 3 × 512 layer to the second to last convolutional layer so that the model can use fine grain features.

  The network is then trained for 160 epochs with a starting learning rate of 10^{-3}, which is later on reduced in posterior epochs, like in the first paper, for finer tuning. In this case dividing it by 10 at 60 and 90 epochs.

- Merging datasets:

  - The authors mix images from both detection and classification datasets. When the network sees an image labeled for detection it backpropagates based on the full YOLOv2 loss function. When it sees a classification image it only backpropagates loss from the classification-specific parts of the architecture.

  - Detection datasets have only common objects and general labels, like “dog” or “boat”. Classification datasets have a much wider and deeper range of labels, for instance: “Norfolk terrier”, “Yorkshire terrier”, and “Bedlington terrier”. Considering this, using a softmax for classification is not ideal since it assumes the classes are mutually exclusive, but “dog” and “Norfolk terrier” are not.

    To solve the merging of the labels of the datasets, the authors propose a hierarchical classification which implies that during training the propagation of ground truth labels is done up the tree so that if an image is labeled as a “Norfolk terrier” it also gets labeled as a “dog” and a “mammal”, etc.

    If the network sees a picture of a dog but is uncertain what type of dog it is, it will still predict “dog” with high confidence but have lower confidences spread out among the hyponyms.

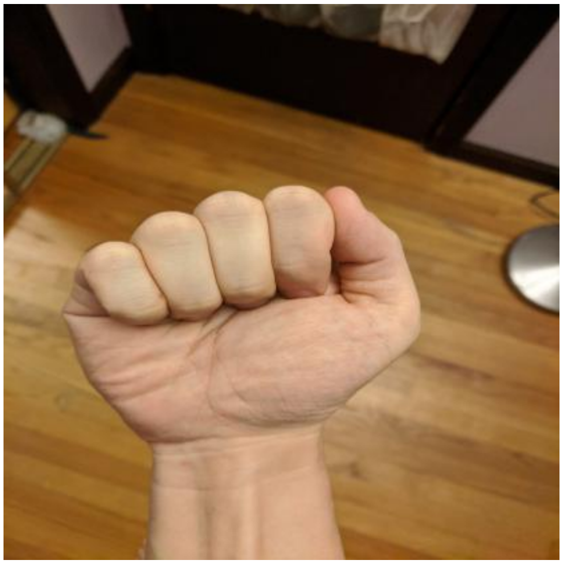
    This formulation also works for detection. In this case, instead of assuming every image has an object, YOLOv2’s objectness predictor is used to compute $P(physical\ object)$. The detector predicts a bounding box and the tree of probabilities. Then the model takes the highest confidence path at every split until it reaches some threshold and predicts that object class.

Using YOLOv5 for American Sign Language Detection

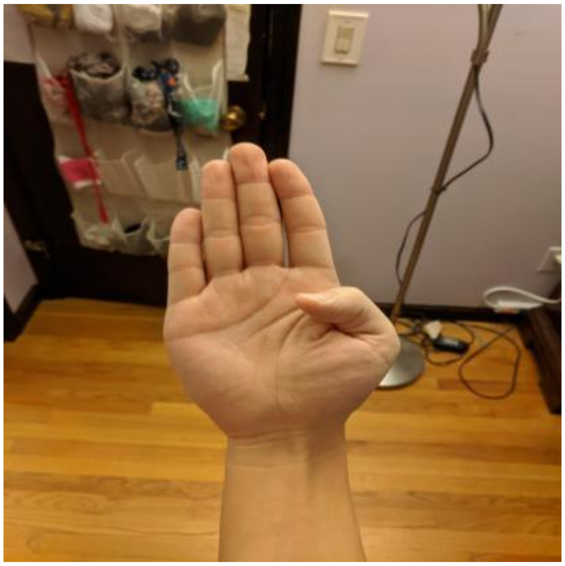
Compared to the previous approaches, here the idea is to use YOLO to find sign language within the images, that is, the images are not the signs themselves but potentially contain signs in them.

The Data

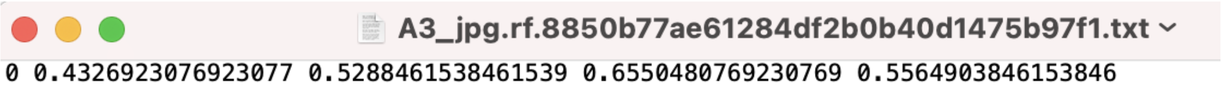
We employed a public dataset available online which contained the images and labels in the format the algorithm needs as inputs:



Sign for “A”



Sign for “B” - Note the complicated background



A3_jpg.rf.8850b77ae61284df2b0b40d1475b97f1.txt

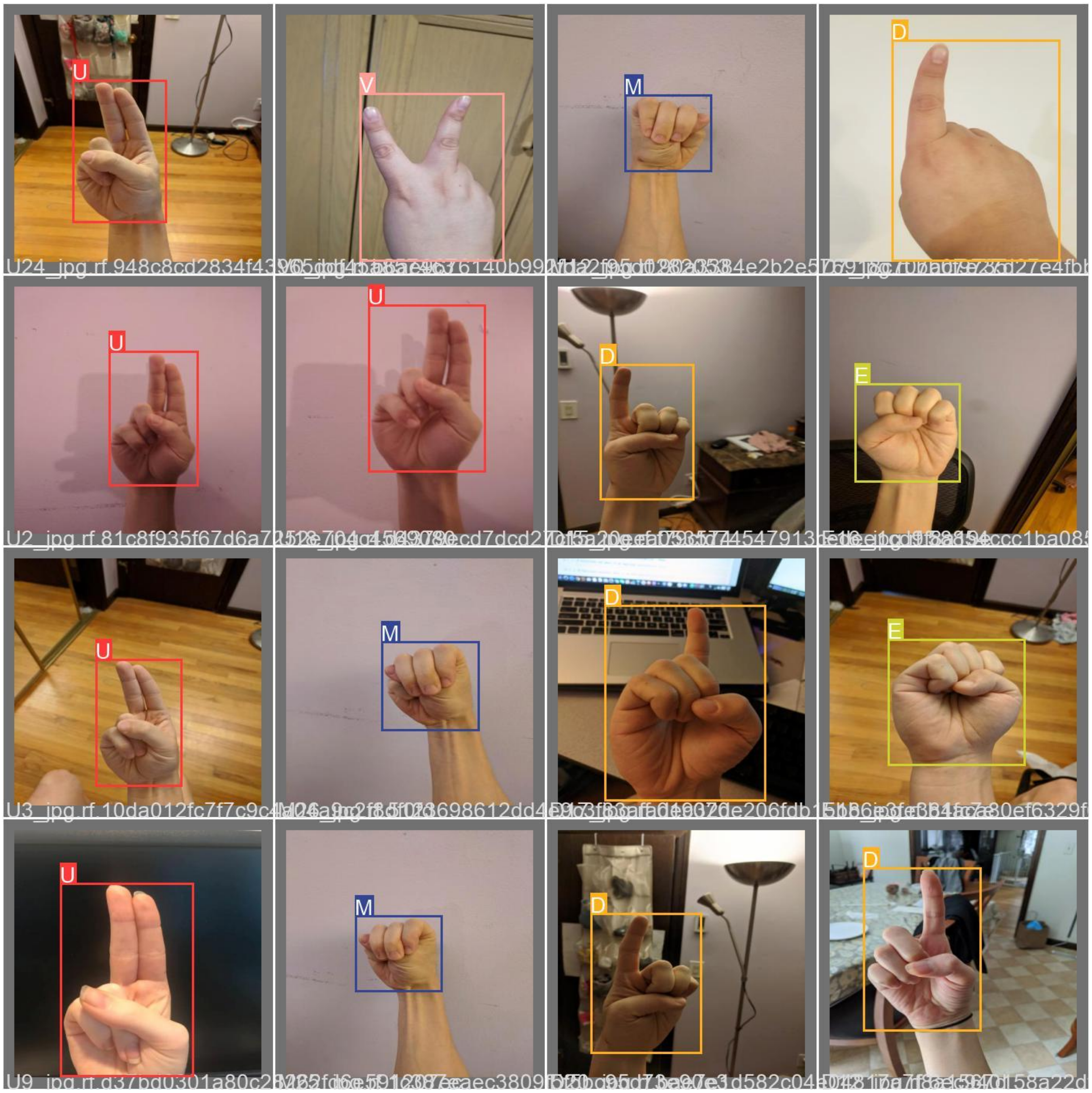
```
0 0.4326923076923077 0.5288461538461539 0.6550480769230769 0.5564903846153846
```

Label file example for “A”

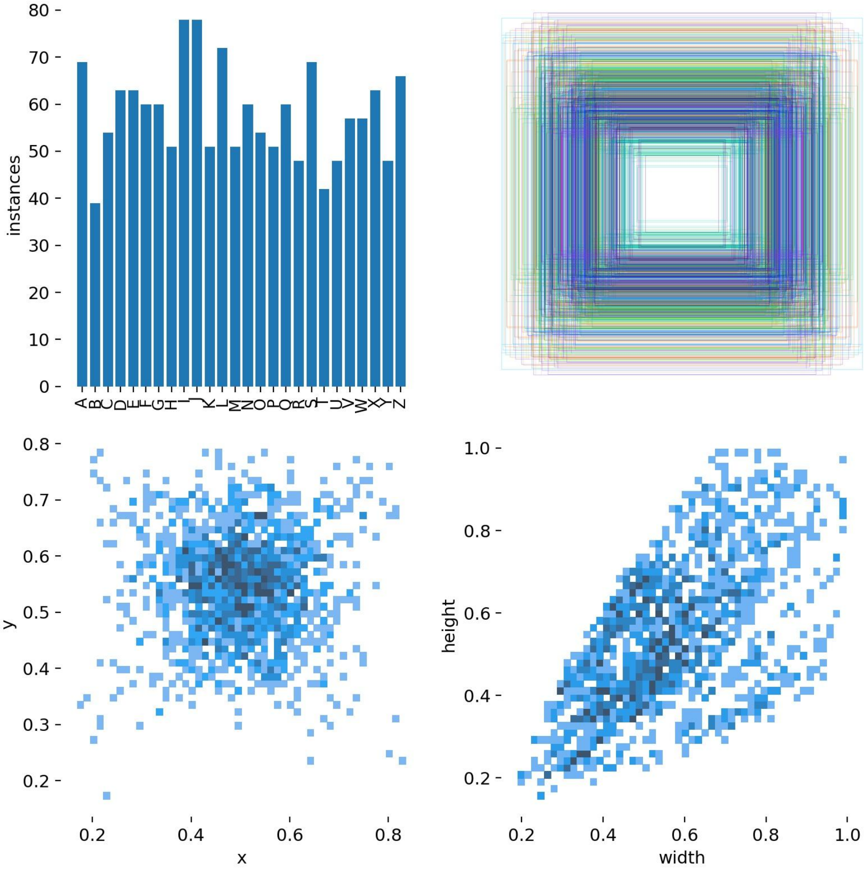
The label-file is a .txt that consists of 5 pieces of data, separated by spaces: the first data point is the label_id and the remaining 4 define the bounding box associated with that label. Note they are scaled in terms of the size of the image. Label files can have multiple lines, each of them referring to one object within the image.

The whole dataset consists in the following quantities of images and labels:

- Training set: 1512.
- Validating set: 144.
- Testing set: 72.


U
V
M
D
U24_jpg.rf.948c8cd2834f43
U
U
D
E
U2_jpg.rf.81c8f935f67d6a7
U
M
D
E
U3_jpg.rf.10da012fc7f7c9c4
U
M
D
D
U9_jpg.rf.d37bd0301a80c2

The dataset aims to have high variability in terms of backgrounds, sizes, contrasts, etc.



The Training Data

Top left
Number of objects in the images. “B” and “T” are underrepresented.

Top right:
Sizes and shapes of all the labels

Bottom left:
Relative position of the objects in the images.

Bottom right:
Relative heights and widths of the labels in the images.

The Model

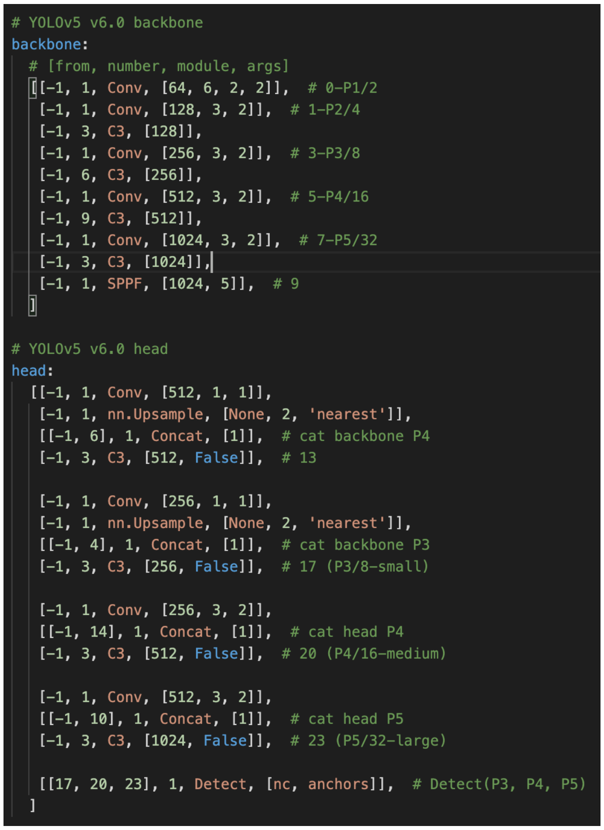
The training was done using Google Colab’s GPU, taking 1.42 hrs for the 150 epochs to complete. Additionally, for testing purposes, we ran the same code on a local runtime using only CPU and it required 7.50 hrs. To do so we run the following command:

```
!python train.py --img 320 --cfg yolov5_26classes.yaml --hyp hyp.scratch-med.yaml --batch 32 --epochs 150 --data colab.yaml --weights yolov5s.pt --workers 24 --name colab_asl
```

Here, we are able to pass a number of arguments:

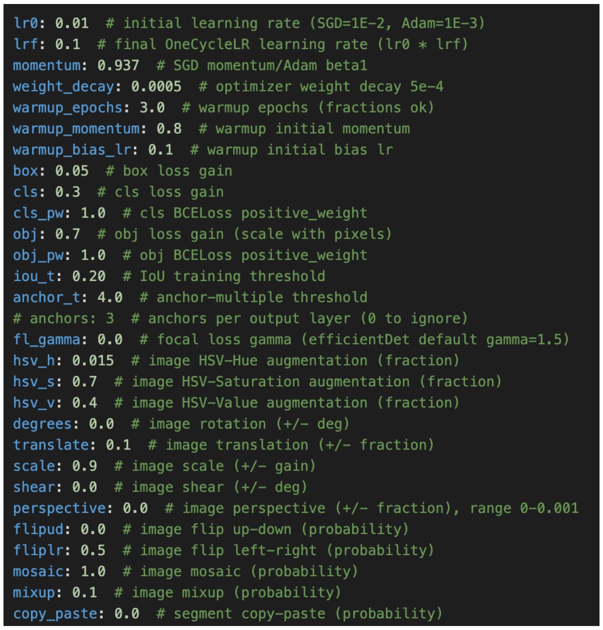
- img: image size, set at 320.
- cfg: reads a .yaml where the architecture of the model is defined.
- hyp: defines hyperparameters for training such as learning rates, momentum, etc.
- batch: batch size. We chose 32
- epochs: number of training epochs. We used 150.

- data: indicates YOLOv5 where the train-validate-test data are. Specifies the number of labels and their names.
- weights: specify a path to weights to start transfer learning from. Pretrained models reduce training times and enhance performance significantly. Here we choose the generic COCO pretrained checkpoint.
- cache: cache images for faster training



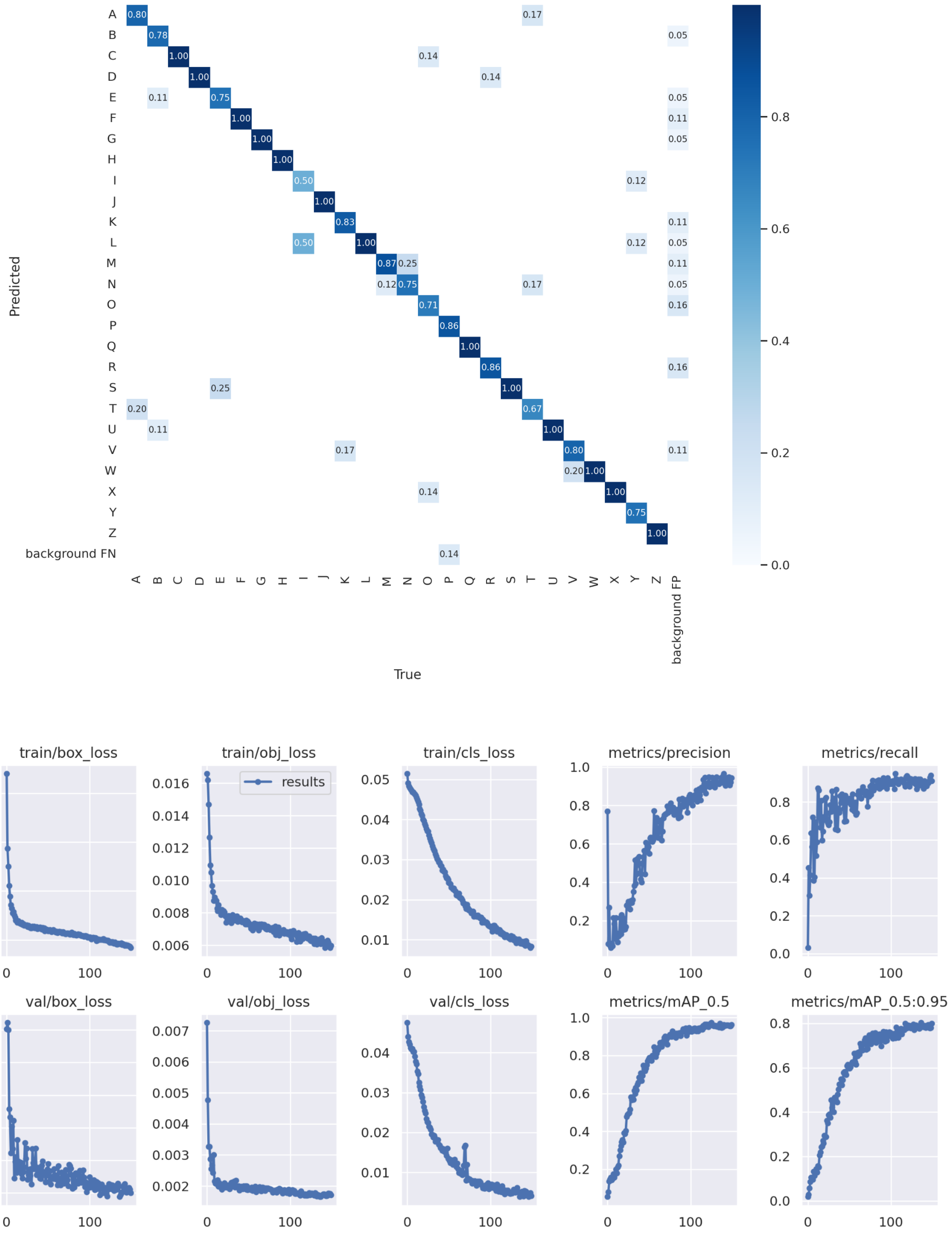
```
# YOLOv5 v6.0 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [64, 6, 2, 2]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, C3, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 6, C3, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, C3, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 3, C3, [1024]],
   [-1, 1, SPPF, [1024, 5]],  # 9
  ]

# YOLOv5 v6.0 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, C3, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, C3, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, C3, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],
   [[-1, 10], 1, Concat, [1]],  # cat head P5
   [-1, 3, C3, [1024, False]],  # 23 (P5/32-large)

   [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
  ]
```

Architecture employed

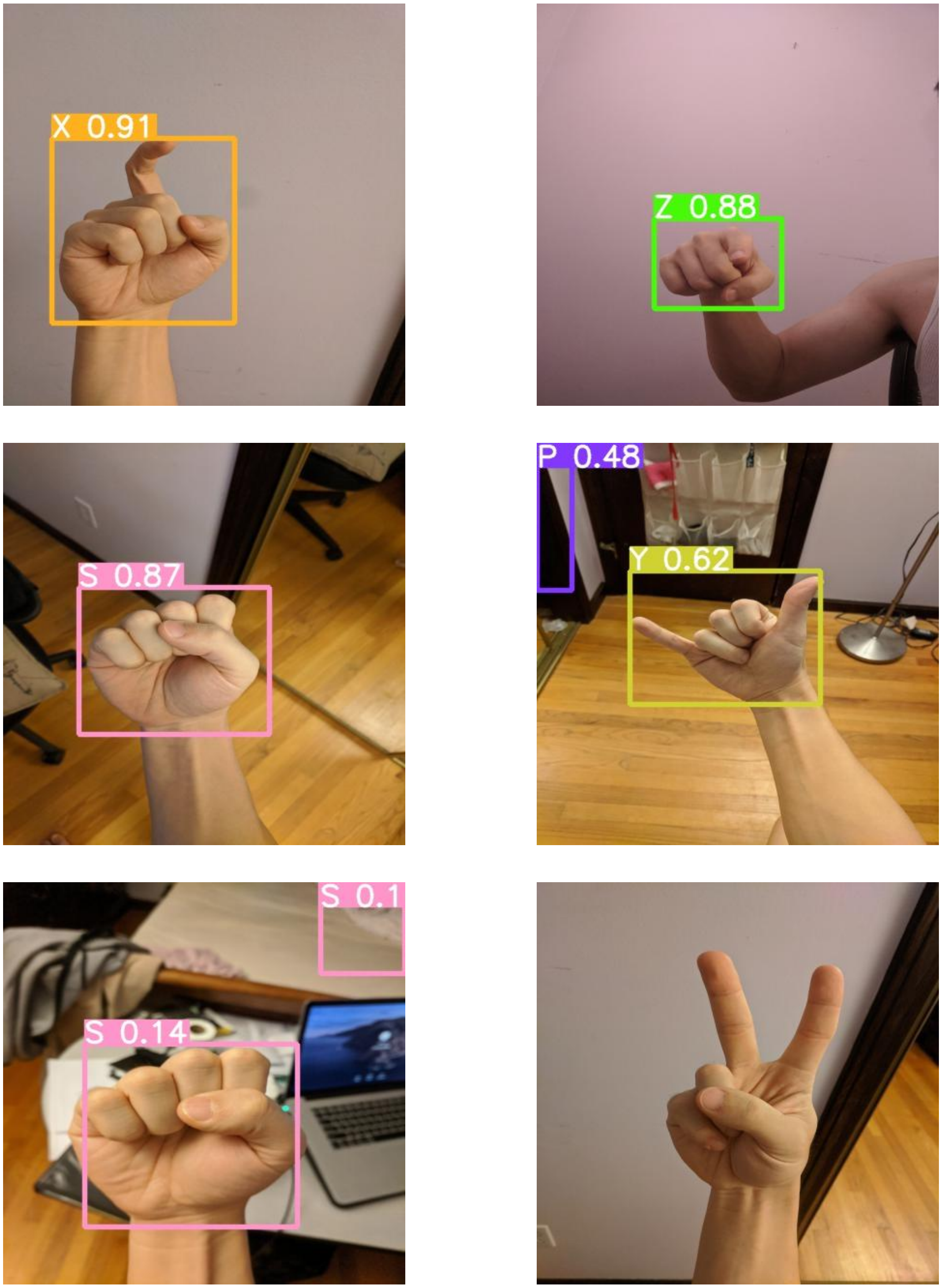


```
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.1  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.3  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 0.7  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
# anchors: 3  # anchors per output layer (0 to ignore)
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 0.0  # image rotation (+/- deg)
translate: 0.1  # image translation (+/- fraction)
scale: 0.9  # image scale (+/- gain)
shear: 0.0  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.5  # image flip left-right (probability)
mosaic: 1.0  # image mosaic (probability)
mixup: 0.1  # image mixup (probability)
copy_paste: 0.0  # segment copy-paste (probability)
```

Hyperparameters



Metrics offered at the training stage - From the confusion matrix, we see the model seems to have trouble distinguishing I from L. We see how the model reduces box, object and class losses with every new epoch.



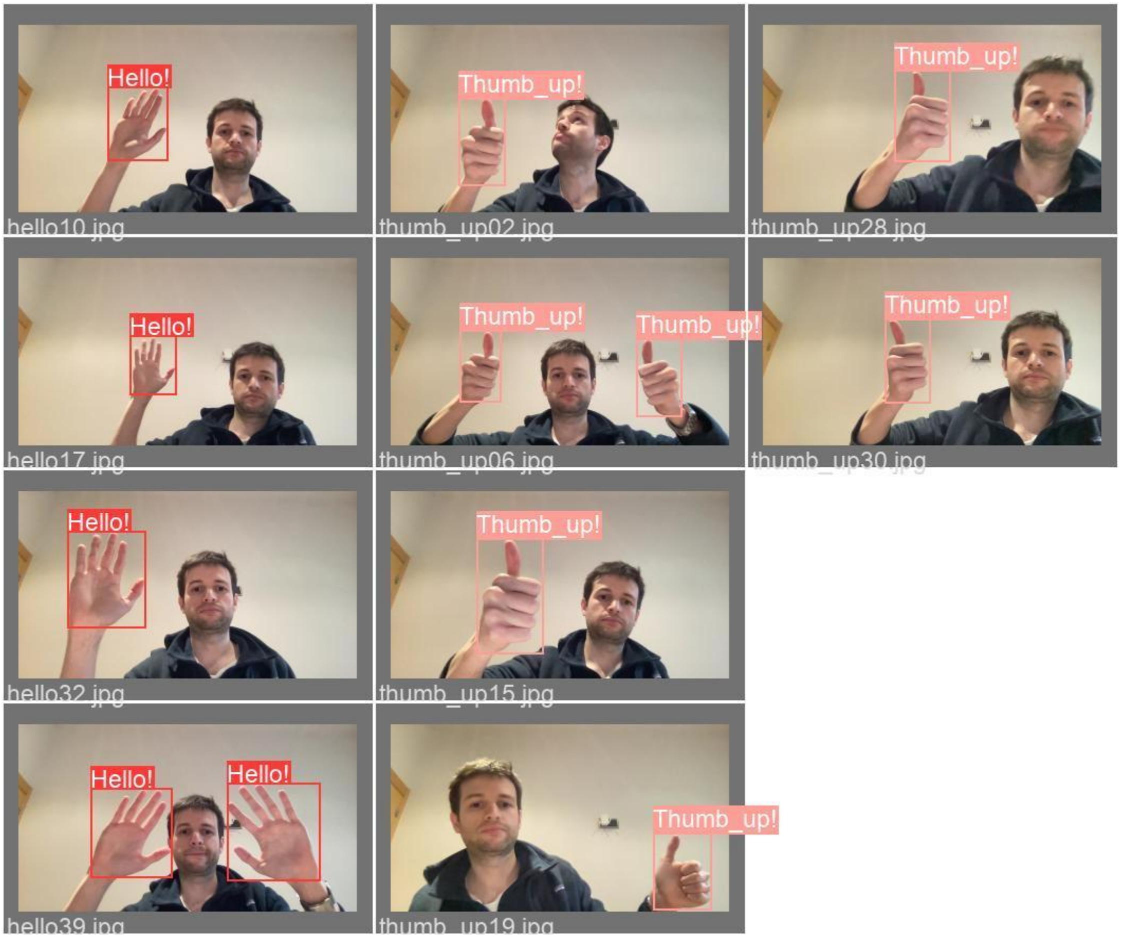
Examples of detection with our trained model - Some objects are detected with high confidence (top left and right, middle left), some background errors appear (middle right and bottom left) and in some cases, no objects are detected (bottom right)

YOLOv5 with a dataset made by ourselves

Additionally, we generated 100 images of ourselves signing to our webcams one of two “objects”:

- 50 of us signaling "hello".
- 50 giving "thumb up".

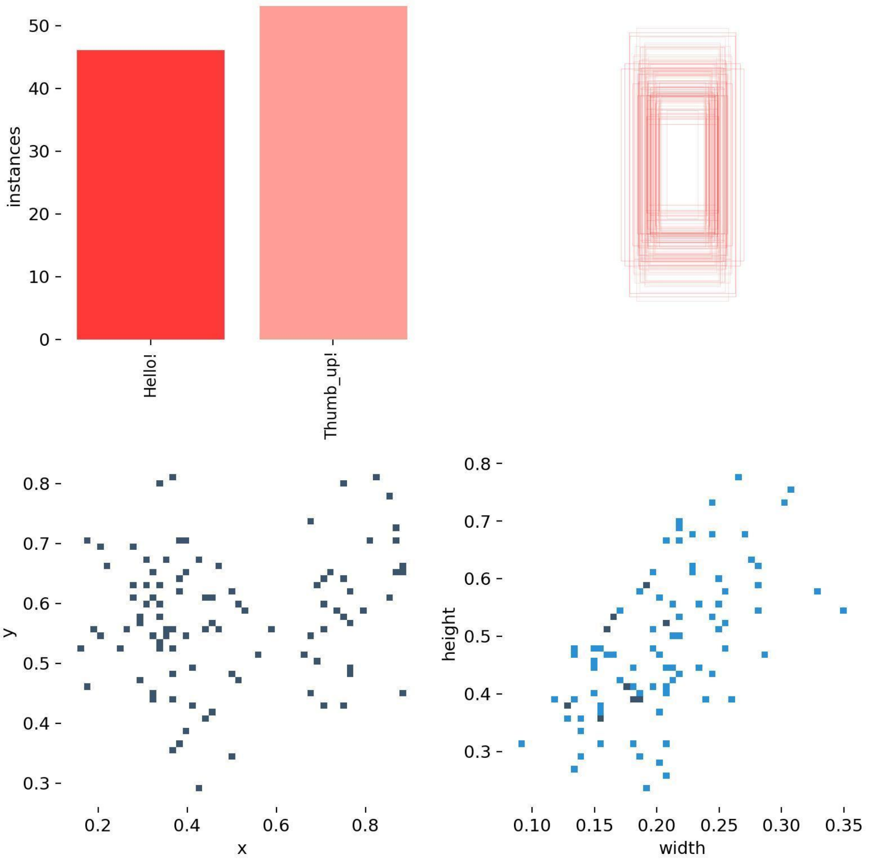
We labeled each of the images using this application.



In order to train, validate and test the model, we split those into:

- 80% for training purposes.
- 10% for validating purposes.
- 10% for testing purposes.

And moved them to new folders as YOLOv5 requires.



The Training Data

Top left
Number of objects in the images.

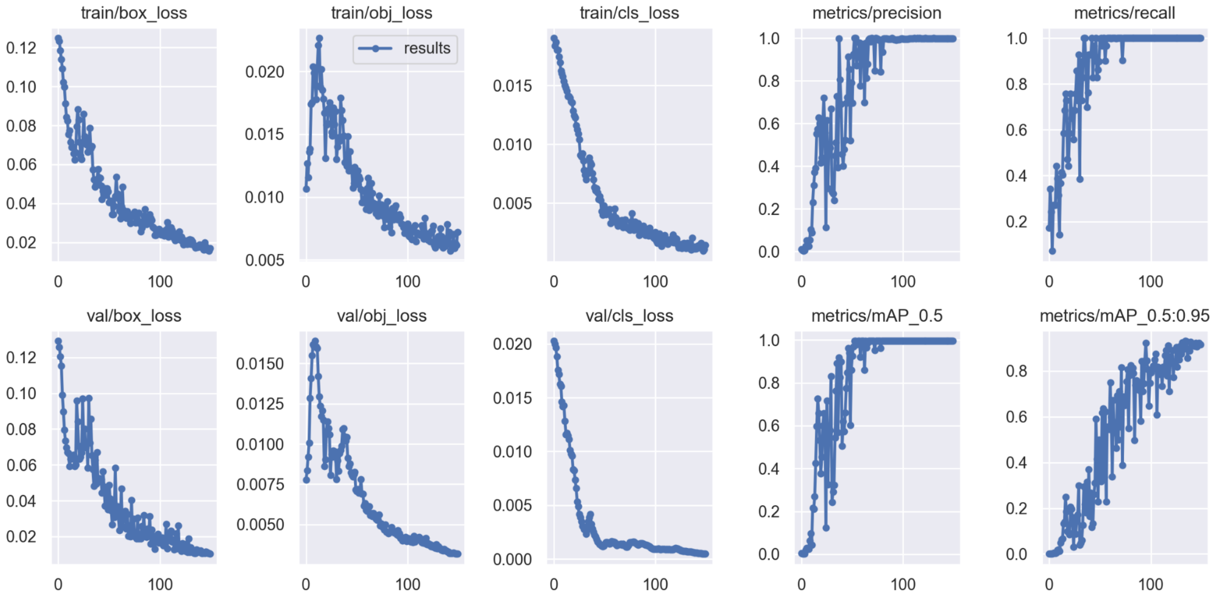
Top right:
Sizes and shapes of all the labels

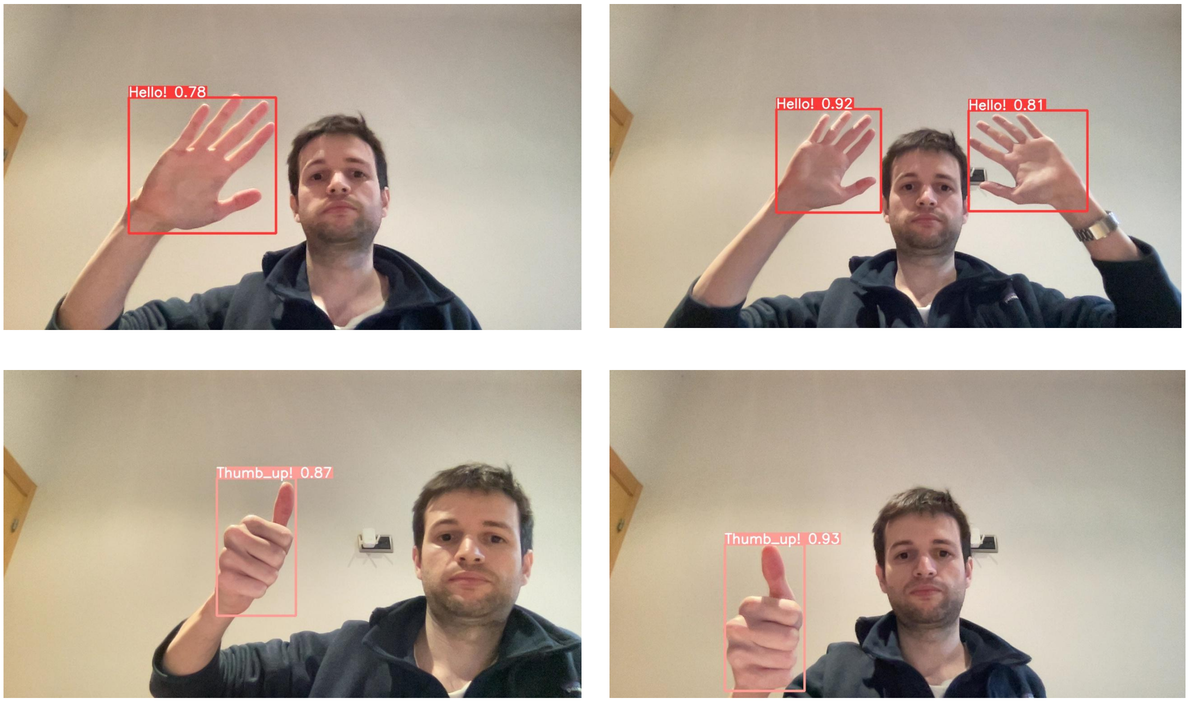
Bottom left:
Relative position of the objects in the images.

Bottom right:
Relative heights and widths of the labels in the images.

We trained our model starting from the same weights and using the same hyperparameters we described in the previous section. It is interesting to see how, despite being a small dataset, the results are quite good: the model generally finds the objects and does so with high confidence.

In this case, we did the training on a local runtime with CPUs, in order to evaluate changes in the training time. It took 0.346 hours (20 minutes) to complete 150 epochs.





Top: metrics generated during the training period, using train and validation data. The results are similar to those in the first case. The confusion matrix is very simple here, the model achieves perfect scores at detecting and labeling.

Bottom: Examples of detection with our trained model - Detects the objects with high confidence.

Creating and training custom neural networks from scratch

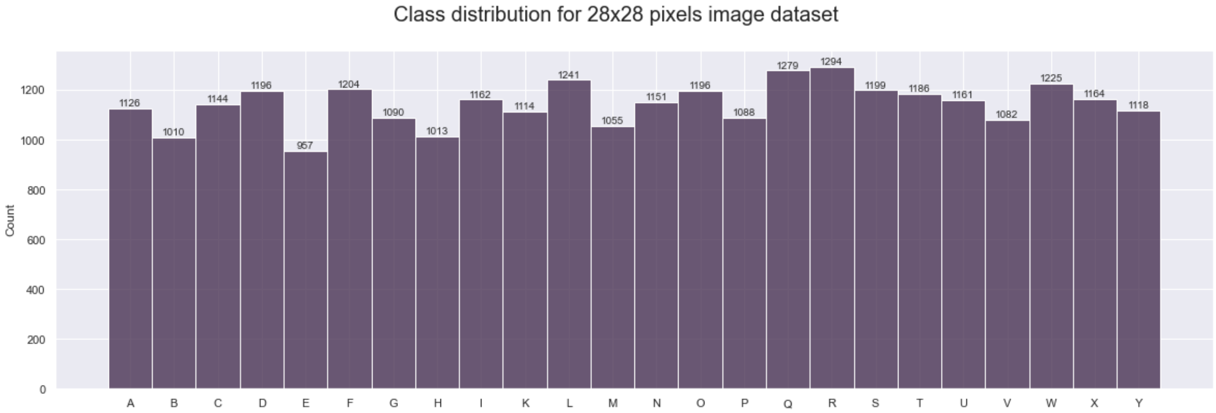
The Data

For the first part of the exercise, we will be using an image dataset with 26 classes, each class corresponding to a letter of the ASL alphabet. The images are 28x28 pixels with 1 channel. The dataset contains around 35k images.

The images are provided in a csv file and we will use a custom extended dataset class to convert the csv file into a pytorch dataset and dataloader, this will allow us to iterate through the dataset with our specified batch size and use the native coding of our x and y variables through the training and testing tasks.

In the image below, we can see a preview of 30 random images from the dataset:

And we can also see below that the classes are well represented



The Model: Simple neural network

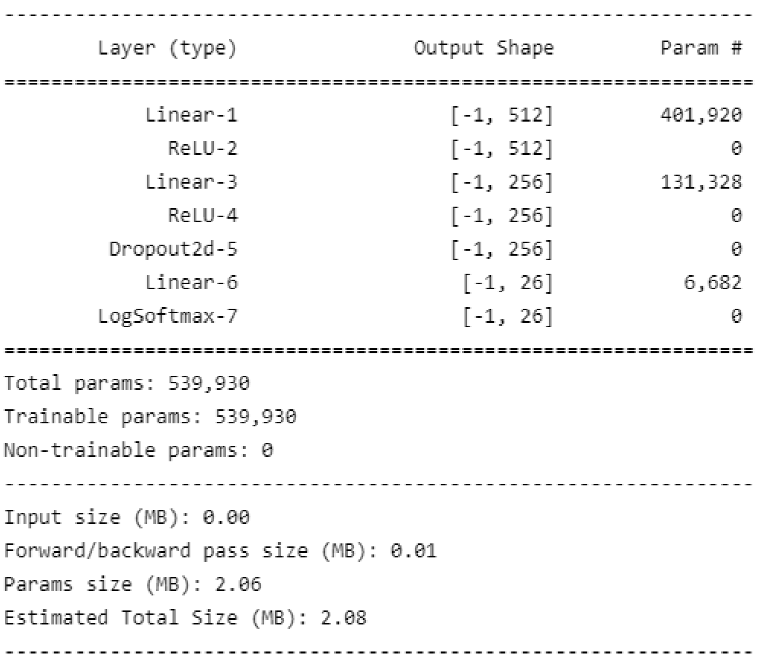
On a high-level perspective, artificial neural networks can be divided into Layers. We can distinguish three types of layers: the Input Layer and the Output Layer, which are representations of the input and output respectively, as well as optional Hidden Layers. Hidden layers are simply all the other layers in between, which are performing complementary computations, resulting in intermediate feature representation of the input.

When the neurons of a given layer are always connected to all neurons of the previous and subsequent layers. This is the most basic kind of layer, called Fully Connected Layer, which can be used for many different tasks across many different domains. As we will see later, there also exist other types of layers, which are better suited for certain challenges

Architecture

The simple network architecture scenario will use only 3 fully connected or linear layers together with their corresponding non-linear activation functions (ReLU) and a final softmax layer. Softmax is used as the activation function for multi-class classification problems where class membership is required on more than two class labels. On the second linear layer we have incorporated a dropout stage with a dropout rate of 0.5, this will help us prevent overfitting.

The layer output size of the architecture can be seen in the below summary print out:



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 512]         401,920
              ReLU-2                  [-1, 512]               0
            Linear-3                  [-1, 256]         131,328
              ReLU-4                  [-1, 256]               0
         Dropout2d-5                  [-1, 256]               0
            Linear-6                   [-1, 26]           6,682
        LogSoftmax-7                   [-1, 26]               0
================================================================
Total params: 539,930
Trainable params: 539,930
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 2.06
Estimated Total Size (MB): 2.08
----------------------------------------------------------------
```

Parameters and optimizer

For the simple network scenario, we will apply the following main parameters:

Hidden units

We will set the initial size of our linear to 512 hidden units, and we will decrease it to 256 in the second linear layer. This setup will help us improve the prediction power of the algorithm, previous tests with lower hidden units showed less predictive power. This might be explained by the fact that such a simple network depends highly on the data and number of parameters provided.

Learning rate

The learning rate parameter will be set initially to 0.0001. The training process we have builts is equipped with the capability to use decreasing learning rate using pytorch exponential learning rate scheduler, but the parameter that sets the decreasing rate (gamma) has been set to 1 to not decrease it at all. It would be possible to fine tune it further to also help with the overfitting issue.

Batch size

We choose a batch size = 16, which was defined at the moment the pytorch data loader was created. This provides good training stability and generalization performance. Larger batch sizes tend to increase the overfitting by quite a significant amount..

Epochs

We will be running only 10 epochs, even though some of the models show that results could be improved by increasing the number of epochs, but in order to be able to try different things and still computationally manageable in terms of time we have chosen this short number of epochs.

Optimizer

We will be using the SGD optimization algorithm, which is a simple yet very efficient approach to fitting linear classifiers and it is known to generalize better.

Stochastic gradient descent (SGD) algorithm is described as follows:

1. Choose an initial vector of parameters w and learning rate eta.

2. Repeat until an approximate minimum is obtained:
    a. Randomly shuffle samples in the training set.
    b. For i=1,2,...,n do:

$$w := w - \eta \nabla Q_i(w).$$

SGD objective function:

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w).$$

where the parameter {\displaystyle w}w that minimizes {\displaystyle Q(w)}Q(w) is to be estimated

SGD update step:

$$w := w - \eta \nabla Q(w) = w - \frac{\eta}{n}\sum_{i=1}^{n} \nabla Q_i(w).$$

where eta is a step size, sometimes called learning rate

Results

Before reporting our results some additional considerations need to be made. The entire dataset has been divided into train/test with a ratio of 0.8/0.2.

The loss function in our test and train tasks is the cross entropy loss, which is very useful when training a classification problem. This loss will be used to calculate the accuracy as well (number of correctly classified images / total dataset). We will be comparing accuracy and loss between our train and test executions.

The cross-entropy between two probability distributions, such as Q from P, can be stated formally as: *H(P, Q)*

Where H() is the cross-entropy function, P may be the target distribution and Q is the approximation of the target distribution.

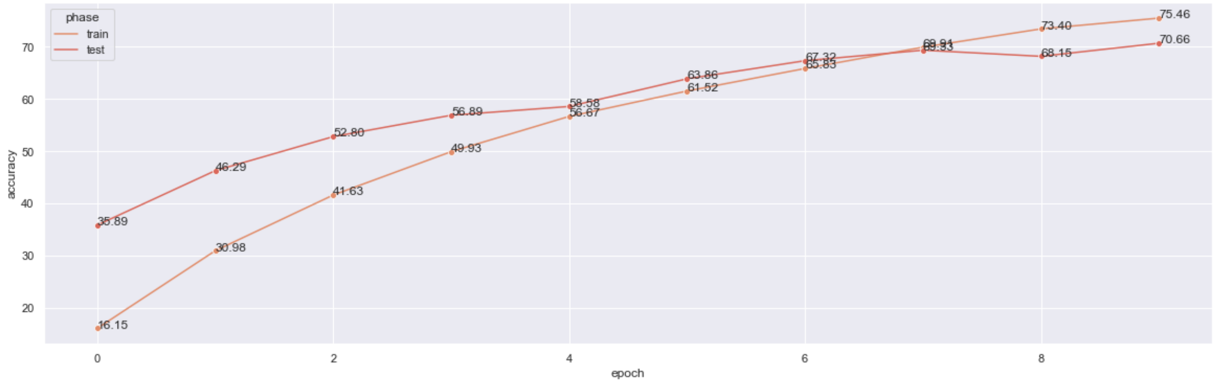
Cross-entropy can be calculated using the probabilities of the events from P and Q, as follows:

*H(P, Q) = – sum x in X P(x) * log(Q(x))*

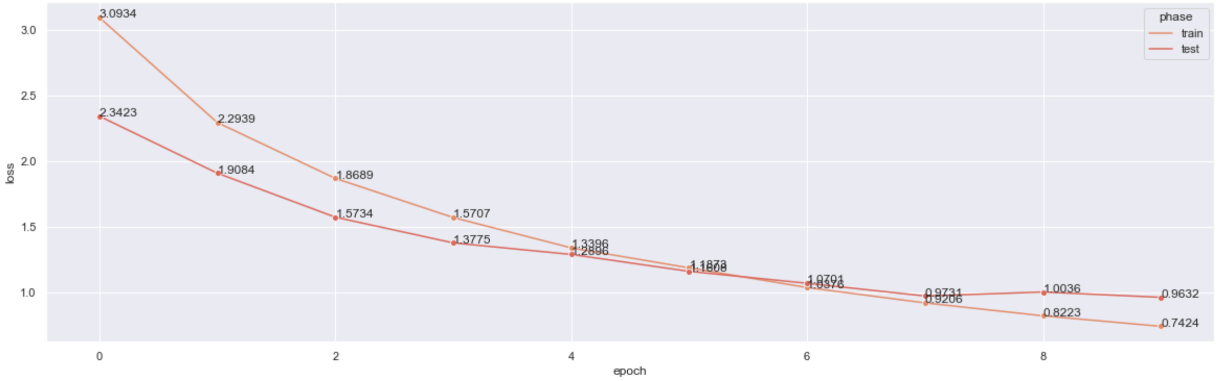
Where P(x) is the probability of the event x in P, Q(x) is the probability of event x in Q and log is the base-2 logarithm. We will be comparing accuracy and loss between our train and test executions.

The results for the simple network are very encouraging, it reaches a relatively high accuracy and low loss but most importantly it seems to be increasing linearly and could probably be improved by extending the number of epochs. On the negative side, the model overfits a bit after a certain number of epochs which would require further fine-tuning.

Train vs test accuracy for simple network



Train vs test loss for simple network



The Model: Convolutional Neural Network

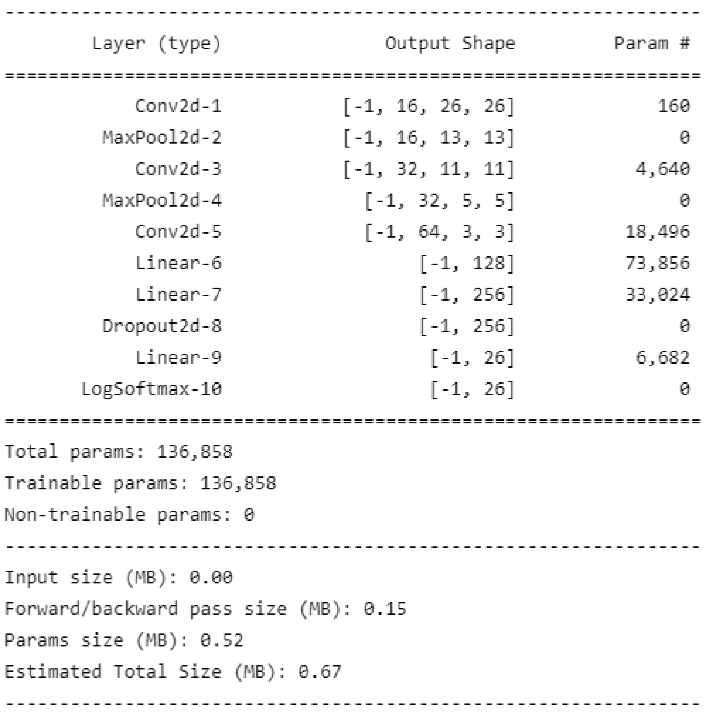
Now on this part we will built a more robust neural network oriented to processing of images which are the Convolutional neural networks.
Being designed to work with image, the neurons of a CNN layer are organized across the three dimensions, height, width and depth, just like the pixels in an image where the depth dimension would differentiate the different color values.

Architecture

For the convolutional network the architecture gets a bit more complex. We will include three convolutional layers, all of them with their corresponding non-linear activation function (ReLU) and in the two initial convolutional layers we will follow them by a max pool layer. Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map.

After the featurizer part of the network we three fully connected layers with an intermediate dropout layer set to 0.5, to help us control overfitting issues. At the end we normalize our output with a softmax layer. Softmax is used as the activation function for multi-class classification problems where class membership is required on more than two class labels.

In the image below we can see a full description of the neural network constructed and the output sizes of each of the convolutional layers, which have been set increasingly to 16, 32 and 64.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 26, 26]             160
         MaxPool2d-2           [-1, 16, 13, 13]               0
            Conv2d-3           [-1, 32, 11, 11]           4,640
         MaxPool2d-4             [-1, 32, 5, 5]               0
            Conv2d-5             [-1, 64, 3, 3]          18,496
            Linear-6                  [-1, 128]          73,856
            Linear-7                  [-1, 256]          33,024
         Dropout2d-8                  [-1, 256]               0
            Linear-9                   [-1, 26]           6,682
       LogSoftmax-10                   [-1, 26]               0
================================================================
Total params: 136,858
Trainable params: 136,858
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.15
Params size (MB): 0.52
Estimated Total Size (MB): 0.67
----------------------------------------------------------------
```

Parameters and optimizer

In terms of parameters we have left most of the parameters similar to what we did for the simple network, but there are some important differences which are highlighted below:

Hidden units

We have reduced the number of hidden units for the initial linear layer to 128. This helps with computational time and reduces the number of total parameters produced. For this most specialized network, the additional depth is not needed.

Learning rate

We have set a decreasing learning rate multiplier (gamma) to 0.8. This means that for each epoch the initial learning rate will be reduced by 0.8 times. This prevents the training to overfit as well.

Momentum

This parameter has been set to a value of 0.9. Momentum is a moving average of our gradients and it helps us 'denoise' the data and bring it close to the original function.

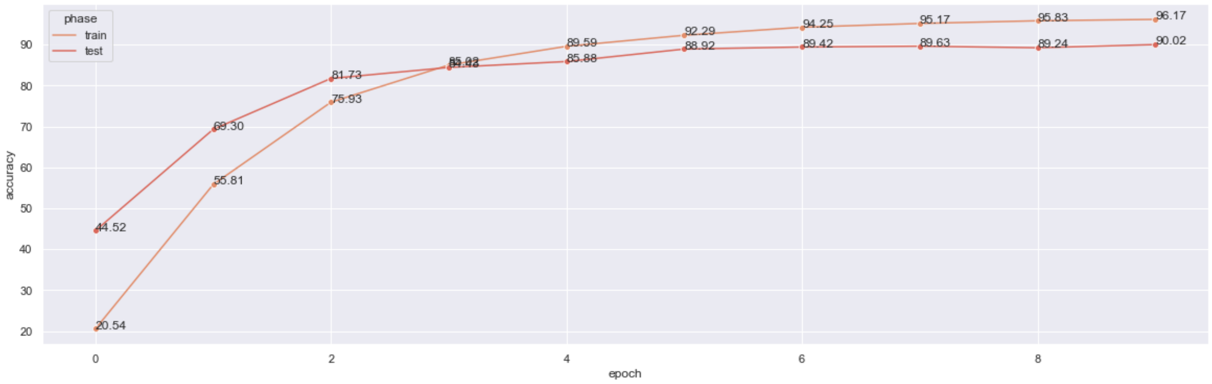
For the rest of parameters, including batch size, optimizer and loss function we have kept the same setup as in the simple network approach.

Results

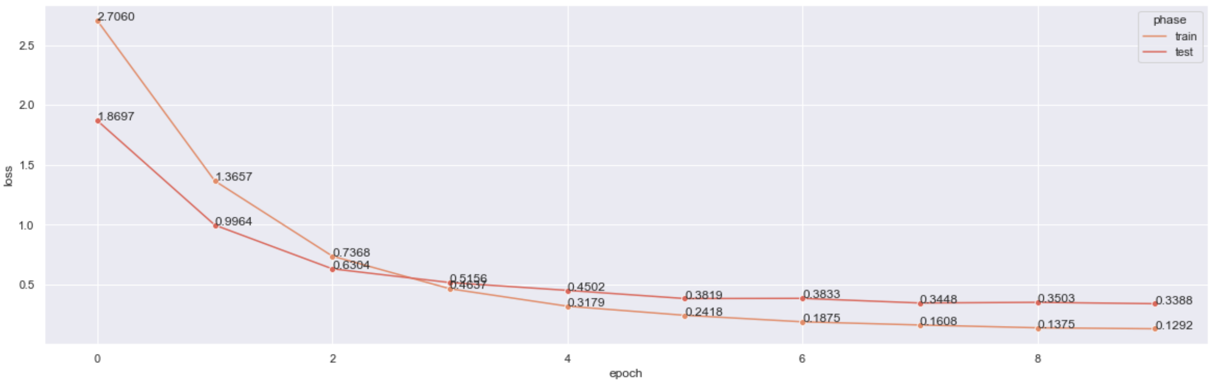
The CNN performs quite well, reaching 90% accuracy in 10 epochs. However we can see that the increasing rate for the accuracy is not very pronounced, which means that it probably won't reach a higher score with this setup or it will do so slowly that is technically impossible to improve. Same case for the loss, after 10 epochs it seems to improve very little.

Also we can appreciate some considerable overfitting, despite our efforts to prevent it with the dropout, the decreasing learning rate and even the small batch size. All of the above means that there is still room for improvement on this custom CNN architecture.

Train vs test accuracy for CNN network



Train vs test loss for CNN network



Final remarks

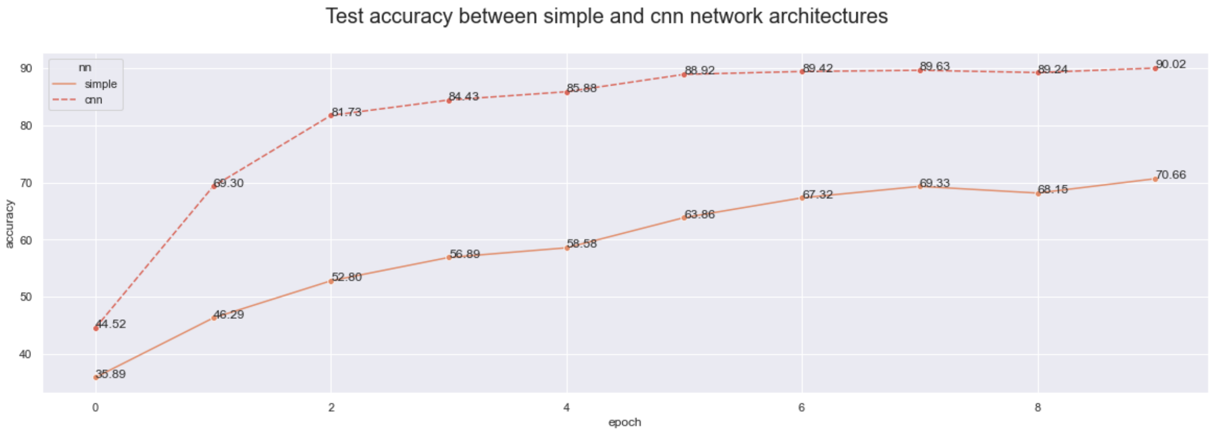
When looking at the accuracy and loss functions of the two neural network setups we have used on this first part of the exercise we can clearly see that the CNN outperformed the simple setup by a significant amount. This comes to no surprise as the CNN purpose is exactly the processing and classification of images.

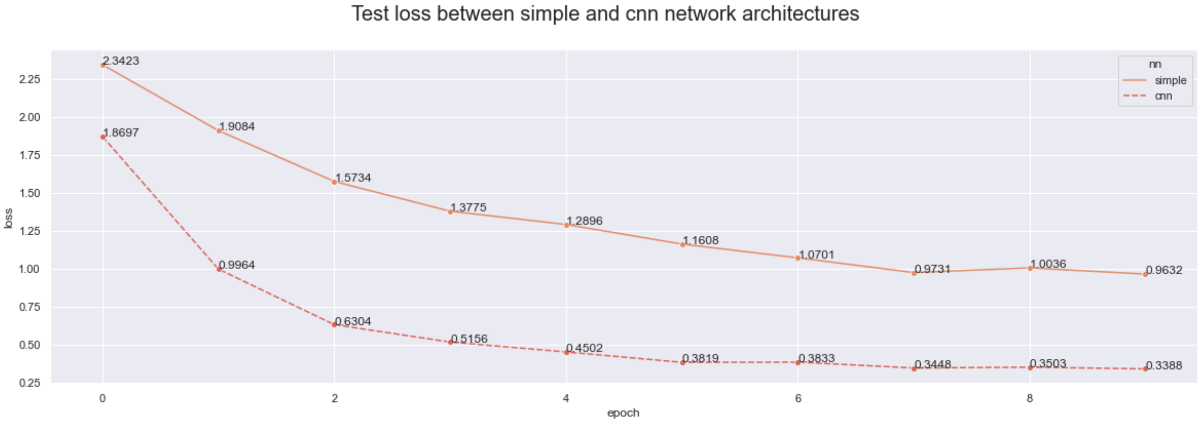
However we must note that the simple network seems to have more potential to grow as epochs are increased, which means that it could potentially reach similar accuracy and loss scores if epochs are increased, while the CNN seems to have reached a plateau.

There is also room for improvement on the overfitting area, as both networks show signs of this with as little as 10 epochs.

Also it is important to note that we kept the architecture simple enough to make them computationally feasible, but the implementation of already established CNN architectures could improve the CNN scores considerably.

Another option that was only partially studied during this exercise was to run the same setup on more robust dataset, for example a 32x32, 3 channels image dataset which we only partially processed with the simple network and it did quite a significant improvement, however the processing times for the CNN made the dataset unfeasible for this exercise.





Considering all the above results, we believe there is a good amount of success in our classification exercise with ASL images from scratch. However these are static images produced in a controlled environment. Our next challenge would be to be able to implement a more modern setup that will be able to not only classify but detect and locate the hand gestures on an image and finally to be able to train that setup with our own images. This will be explored in the next section in full detail.

Conclusions

With the execution of this project we were able to explore the development of image classification using neural networks, from a very basic approach to the most current and advanced pre-trained models such as YOLO. This combined approach has provided us with very useful experience in order to apply the right tool for the right purpose. It was clear that for the american sign language alphabet prediction YOLO technique is overkill and acceptable results can be obtained with simpler custom developed models. However YOLO brings object detection to the picture and also it is very efficient in terms of adding new learning signs to the model based on images produced by ourselves, which opens up a whole new range of possibilities on applying these pre-trained models.

For future experiments it would be interesting to try with multiple dataset and experiment what are the limits of the neural networks learning potential while still keeping the computational time affordable, as this was one of the main challenges we faced when trying to train our models, both in the custom CNN and the YOLO approach.

Works Cited

Altenberger, Felix, and Claus Lenz. “A Non-Technical Survey on Deep Convolutional Neural Network Architectures.” *A Non-Technical Survey on Deep Convolutional Neural Network Architectures*, https://arxiv.org/pdf/1803.02129.pdf.

Redmon, J., and Others. “[1506.02640] You Only Look Once: Unified, Real-Time Object Detection.” *arXiv*, 9 May 2016, https://arxiv.org/abs/1506.02640. Accessed 25 March 2022.

Redmon, Joseph, and Ali Farhadi. “[1612.08242] YOLO9000: Better, Faster, Stronger.” *arXiv*, 25 December 2016, https://arxiv.org/abs/1612.08242. Accessed 25 March 2022.