18786 HW3

Daren Yao

darenyao@andrew.cmu.edu

I've uploaded a zip file of the entire code work, where `Deliverable12.ipynb`, `Deliverable3.ipynb`, and `Deliverable4.ipynb` correspond to the code requirements in `HW3.pdf`.

Deliverable 1 → `Deliverable12.ipynb`

Deliverable 2 → `Deliverable12.ipynb`

Deliverable 3 → `Deliverable3.ipynb`

Deliverable 4 → `Deliverable4.ipynb`

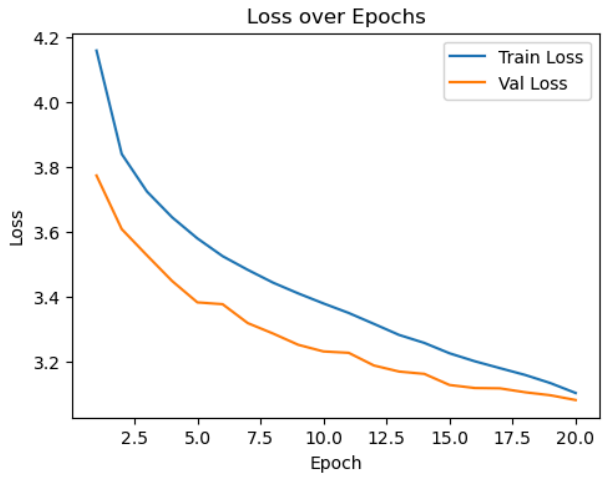
If the notebook files were saved correctly, the corresponding answers and deliverables should be expected in the outputs.

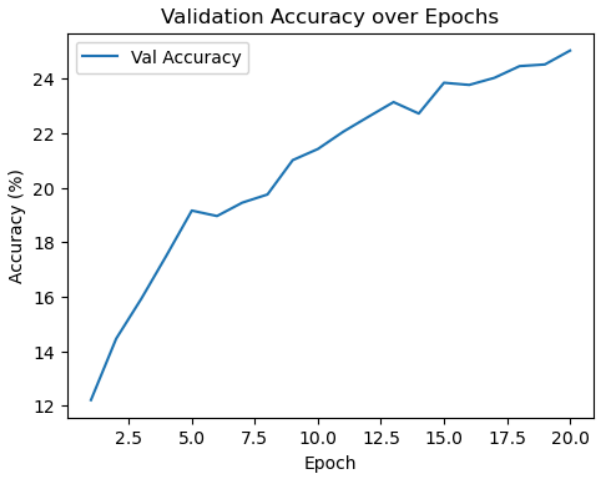
Note: `Deliverable4.ipynb` uses a lot of functions defined in `utils_coco.py` (to handle importing annotations, calculating mAP, etc.).

Deliverable 1

Fully Connected Neural Network (FCNN)

First, I split the training data into an 80:20 ratio for training and validation. This approach helps monitor potential overfitting by observing performance on the held-out validation set.





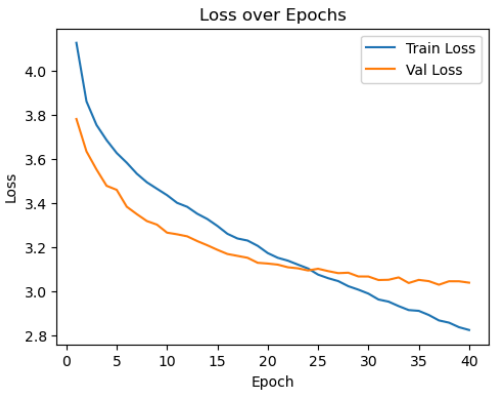
First Round of Experiments

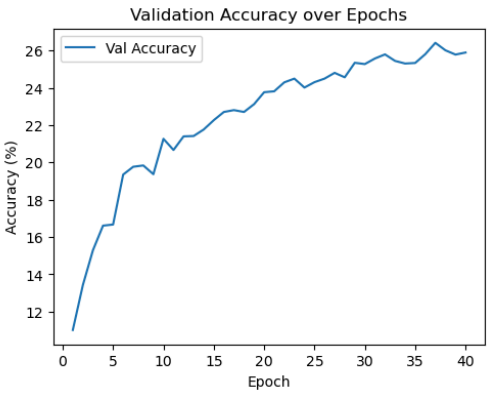
- Learning Rate: 0.001 | Epochs: 20

The model still showed room for improvement, so I increased the learning rate and extended the number of epochs in the next round.

Second Round of Experiments

- Learning Rate: 0.005 | Epochs: 50



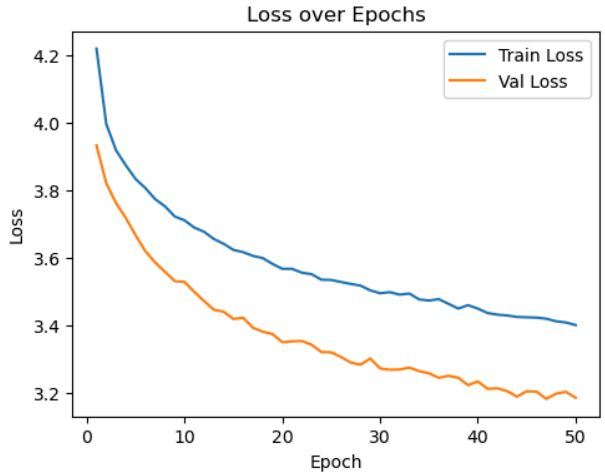


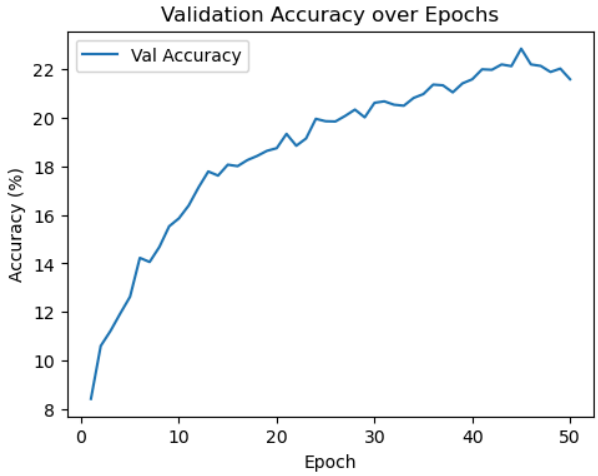
By around Epoch [40/50], the training loss reached 2.8261, while validation loss was 3.0404 and validation accuracy 25.90%. Shortly afterward, early stopping was triggered.

Epoch [40/50] Train Loss: 2.8261, Val Loss: 3.0404, Val Acc: 25.90%

Early stopping triggered!

After this, I applied certain data transformations (e.g., random crops or flips), but the results actually deteriorated. Generally, FCNNs appear not to require extensive data augmentation, largely because they rely heavily on raw pixel correlations. In contrast to convolutional networks, spatial transformations can disrupt FCNNs' learned patterns instead of providing beneficial variation.



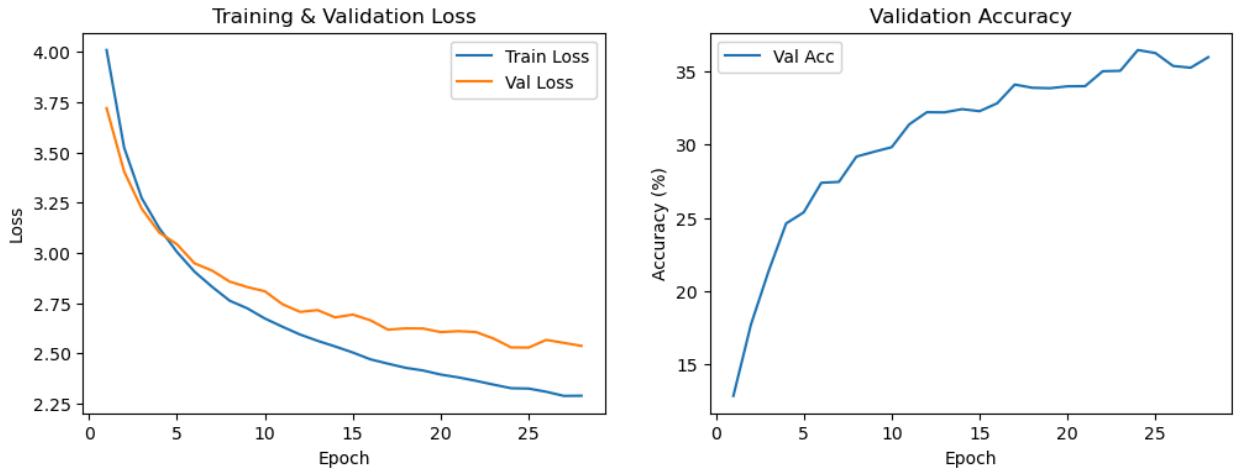


Therefore, without data augmentation, the final performance essentially represents the best outcome achievable with this FCNN structure. Below is the prediction visualization for the FCNN.

CNN

At first, I implemented Conv2D by nested loops, but this would result in slow training.So I used matrix multiplication (unfold's method).This method makes reasonable use of the GPU's matrix arithmetic acceleration, which greatly speeds up the training process.

In the initial simple CNN structure, I obtained 40% test accuracy.



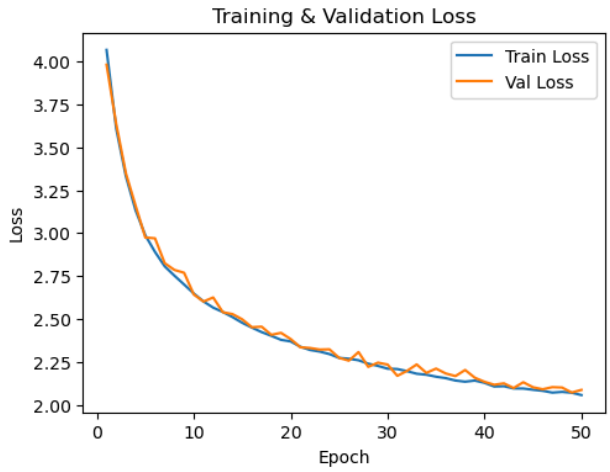
I thought I could get better performance using a more complex CNN structure, so I added batchnorm to each layer on top of adding two extra convolutional pooling layers, and also increased the depth of the fully connected layer appropriately.This really improved the performance of the model significantly, and the model has come to 48% of the test acc.

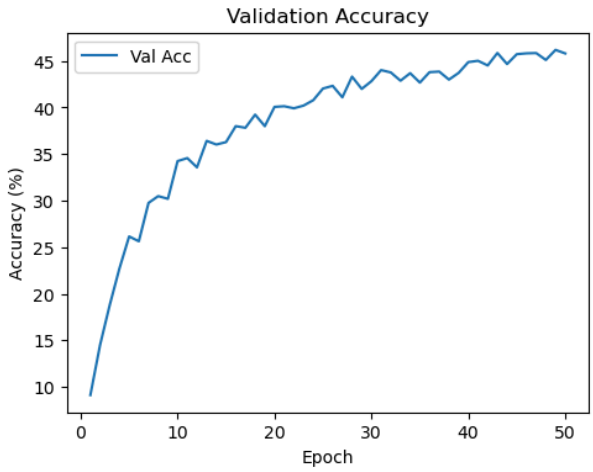
Here is the structure of this CNN:

Python

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=100):
        super(SimpleCNN, self).__init__()
```

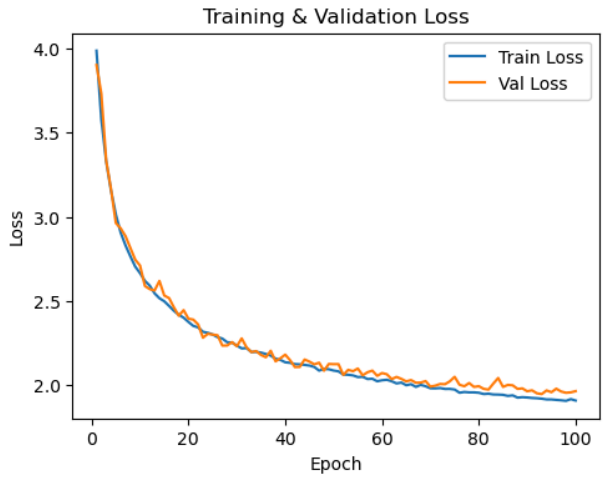
```
        self.conv1 = MyConv2D(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.bn1   = nn.BatchNorm2d(16)
        self.pool1 = MyMaxPool2D(kernel_size=2, stride=2)

        self.conv2 = MyConv2D(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.bn2   = nn.BatchNorm2d(32)
        self.pool2 = MyMaxPool2D(kernel_size=2, stride=2)

        self.conv3 = MyConv2D(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.bn3   = nn.BatchNorm2d(64)
        self.pool3 = MyMaxPool2D(kernel_size=2, stride=2)


        self.conv4 = MyConv2D(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.bn4   = nn.BatchNorm2d(128)
        self.pool4 = MyMaxPool2D(kernel_size=2, stride=2)


        self.fc1 = nn.Linear(128*2*2, 256)
        self.bn5 = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256,128)
        self.bn6 = nn.BatchNorm1d(128)
        self.fc3 = nn.Linear(128, num_classes)

        self.relu = nn.ReLU()
        self.Dropout = nn.Dropout(0.3)

    def forward(self, x):
        # conv1 → ReLU → pool1
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.pool1(x)
        # conv1 → ReLU → pool1
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu(x)
        x = self.Dropout(x)
        x = self.pool3(x)


        x = self.conv4(x)
                                            x = self.bn4(x)
        x = self.relu(x)
        x = self.Dropout(x)
        x = self.pool4(x)

        x = x.view(x.size(0), -1)

        x = self.fc1(x)
        x = self.bn5(x)
        x = self.relu(x)
        x = self.Dropout(x)
        x = self.fc2(x)
        x = self.bn6(x)
        x = self.relu(x)
        x = self.Dropout(x)
        x = self.fc3(x)
        return x
```

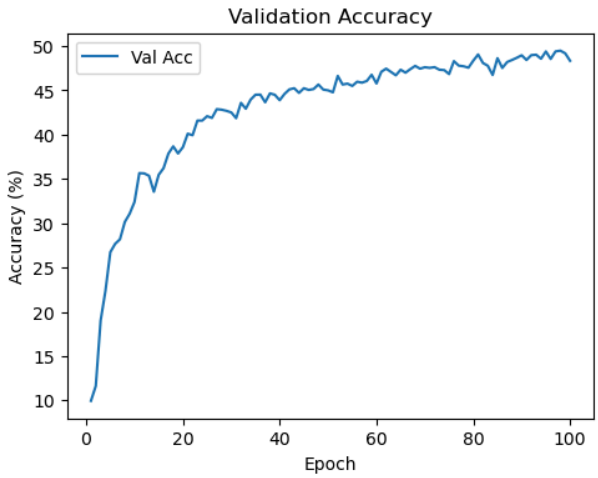




The performance of this model hasn't fully converged and it doesn't show over fitting, maybe the learning rate is too small? I decided to increase the lr and epoch appropriately to see how far this model will converge.

Lr: 0.001 → 0.002 | Epoch: 50 → 100





Extending the training time does improve the model's performance somewhat, but this improvement is limited.(A test acc of 50 percent was eventually achieved)

Visualization:

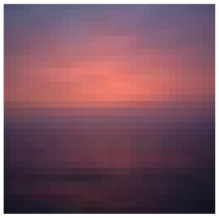
Pred: dolphin
True: mountain

Pred: squirrel
True: forest

Pred: dolphin
True: seal

Pred: lamp
True: mushroom

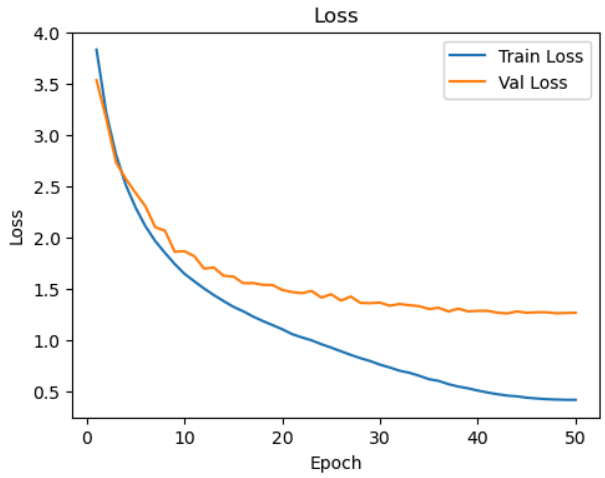
Pred: sea
True: sea

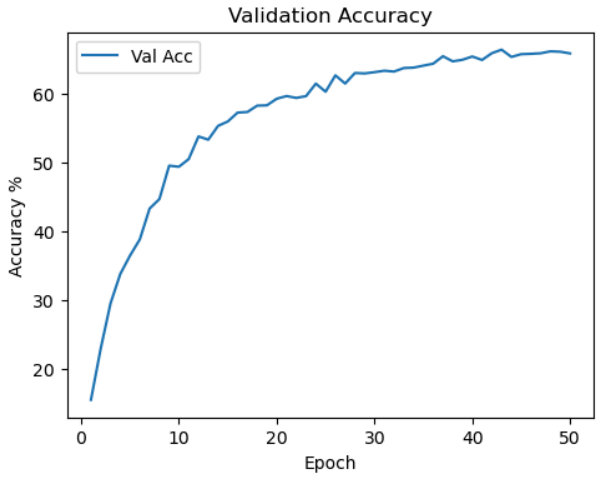


Deliverable 2

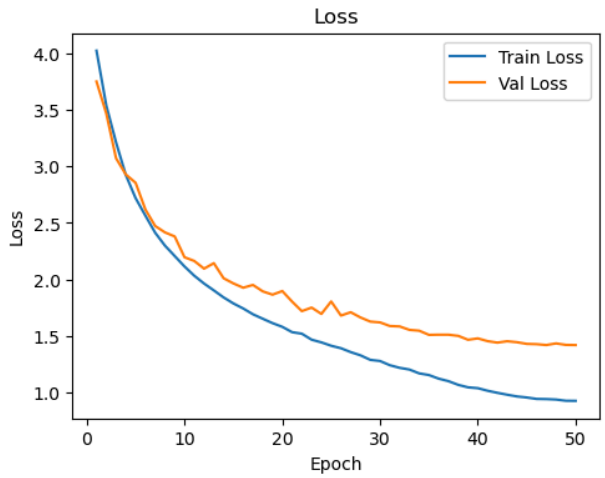
Mini-Resnet

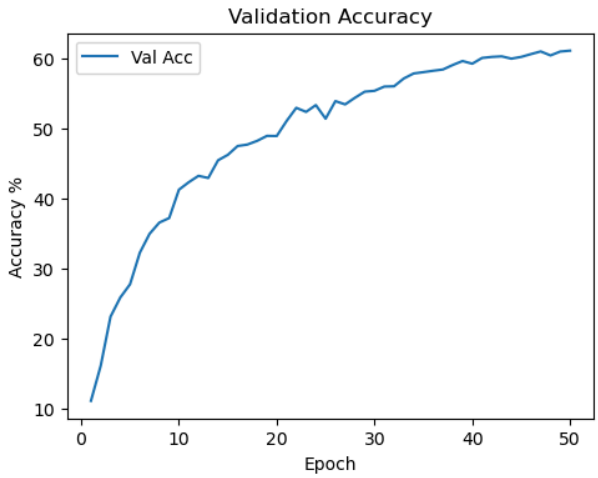
By designing a relatively shallow residual network, the model has achieved extremely good performance.During training, I used cosine annealing and SGD.





As you can see, the model has shown some overfitting.On the one hand, I am going to try a deeper resnet.On the other hand, I added more regularization to avoid overfitting (adding dropout, and doing more TRANSFORMATION on the original data).





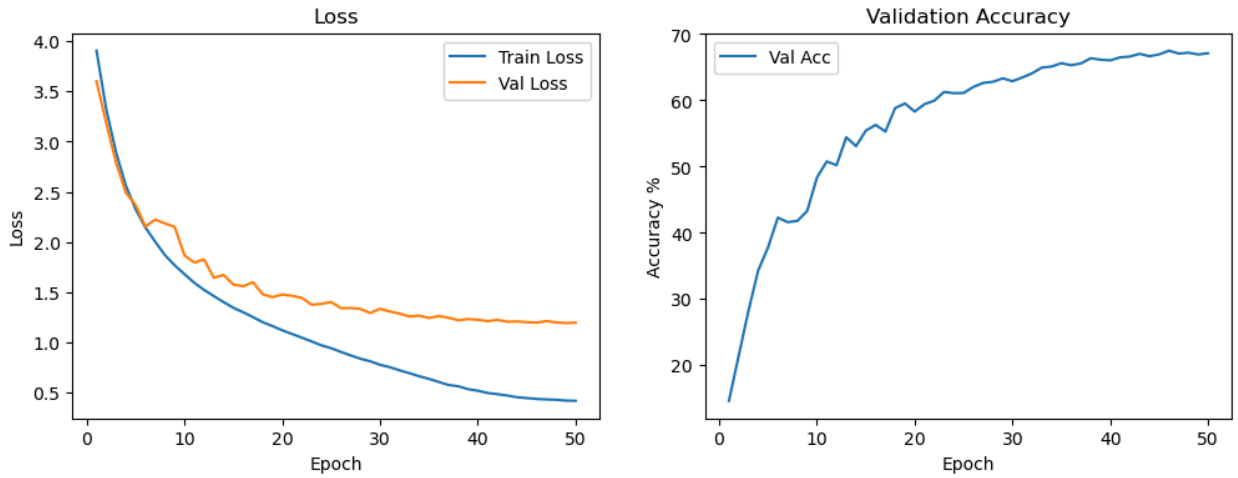
However, the depth of the model did not improve its performance, it just added a lot of training time in vain. After that, I tried many more structures along these lines, none of which achieved much improvement.

Returning to the original one, I noticed that the original model also had worse performance compared to the one that trained on data that didn't use aggressive transformation.

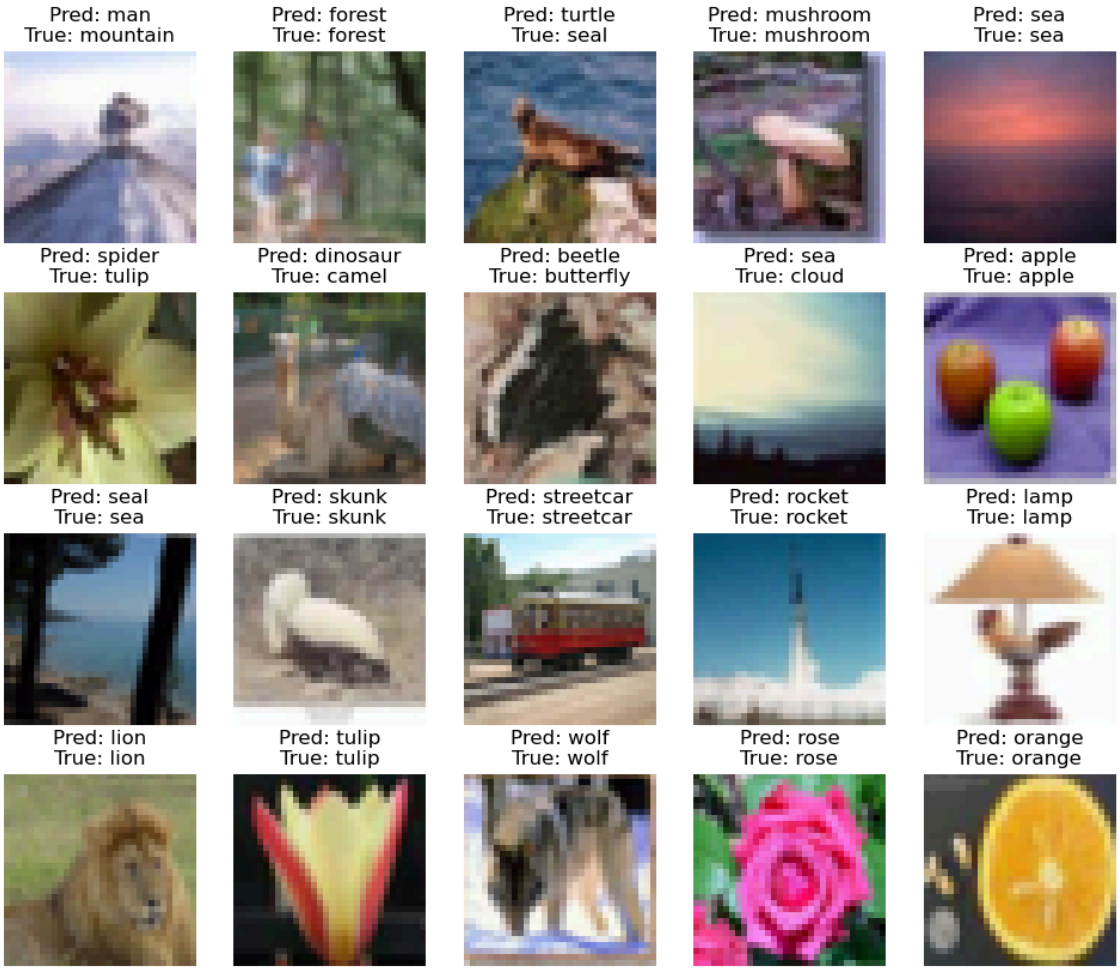
Therefore, I tried to train two model only using a little transformation and still using dropout. The training speed increased significantly, but the model didn't have any improvement.

It seems that sometimes the rotation or colour perturbation on some of the samples introduces too much noise for the model to accommodate. In conclusion, My Mini ResNet's best performance is around 67% test acc.

The training result of `model_mini_res_2 = MiniResNet(block=BasicBlock, num_blocks=(2,2,1,1), num_classes=100)` is:



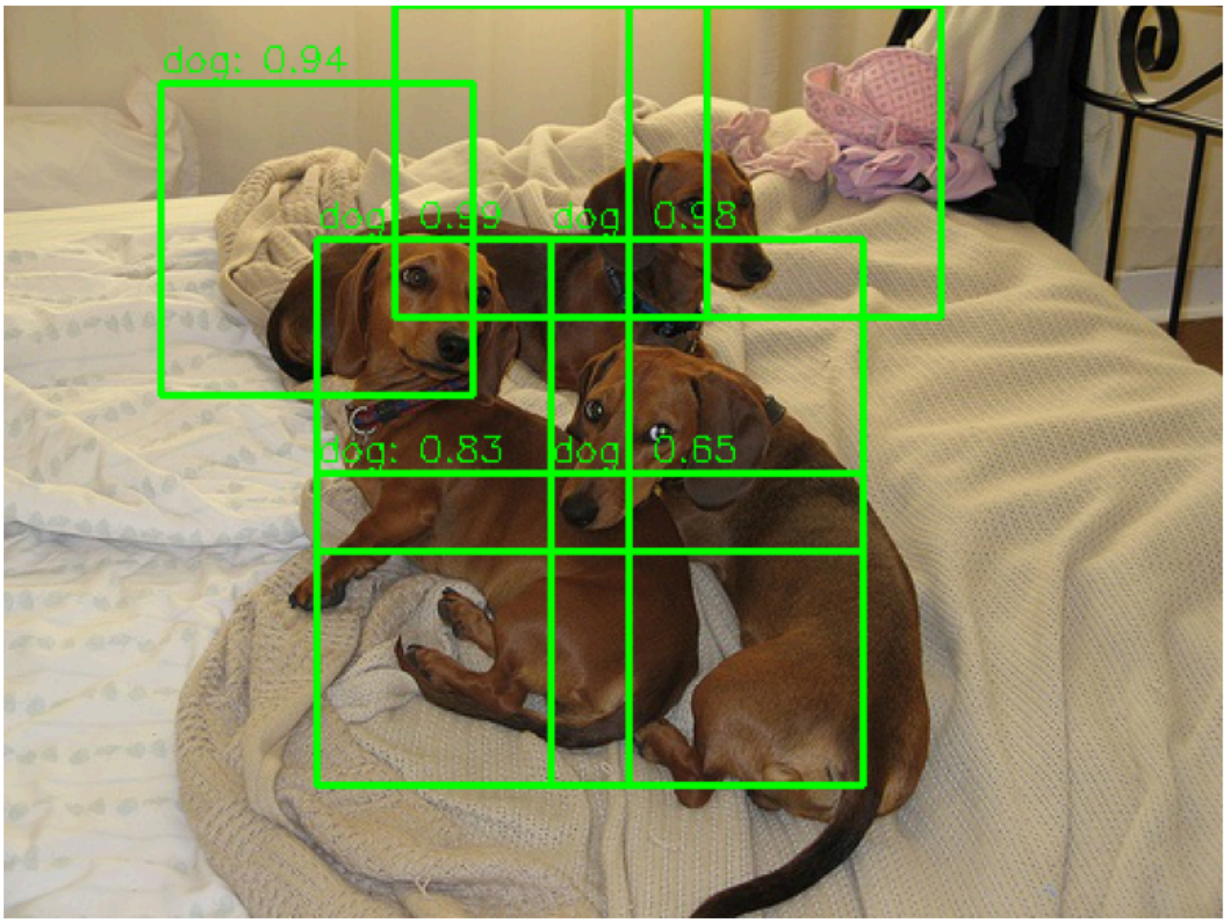
Visualization:


Pred: man
True: mountain
Pred: forest
True: forest
Pred: turtle
True: seal
Pred: mushroom
True: mushroom
Pred: sea
True: sea
Pred: spider
True: tulip
Pred: dinosaur
True: camel
Pred: beetle
True: butterfly
Pred: sea
True: cloud
Pred: apple
True: apple
Pred: seal
True: sea
Pred: skunk
True: skunk
Pred: streetcar
True: streetcar
Pred: rocket
True: rocket
Pred: lamp
True: lamp
Pred: lion
True: lion
Pred: tulip
True: tulip
Pred: wolf
True: wolf
Pred: rose
True: rose
Pred: orange
True: orange

Deliverable 3

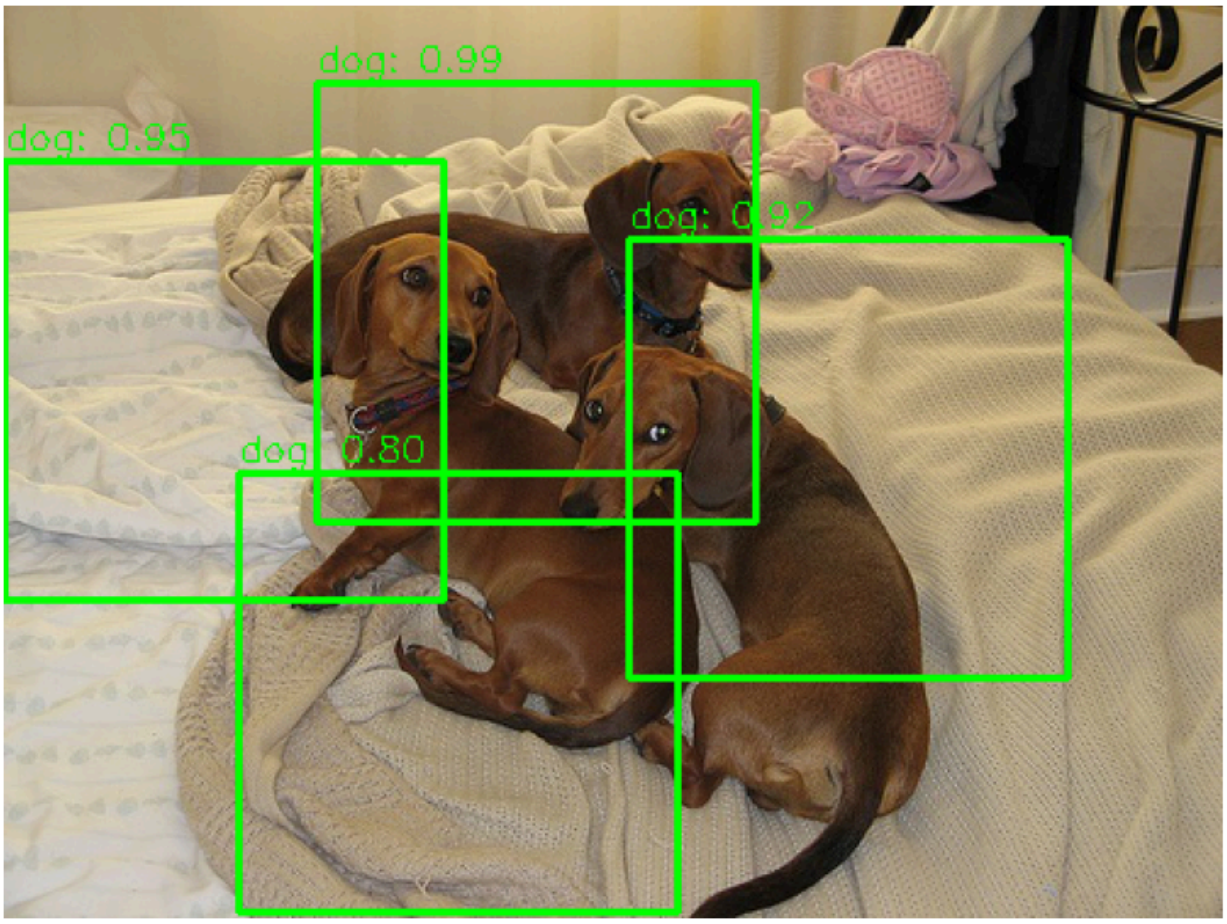
Approach: Making a Detector Using a Classifier

- Generate region proposals (sliding window)
    - Traverse the input image with a fixed-size window .
    - Slide this window across the image at a chosen step size.
    - Each window is treated as a “candidate region” that potentially contains an object.
- Use a pre-trained classification network
    - For each candidate region (sub-image), pass it to a pre-trained ImageNet classifier (e.g., a ResNet or VGG).
    - Take the max of the probabilities of all dog-related (or cat-related) classes to get an overall “dog” (or “cat”) score.
    - If the score exceeds a threshold (e.g., 0.5), label that window as containing a dog/cat.
- Filter out background and apply Non-Maximum Suppression (NMS)
    - Many sub-image windows will likely overlap, and some will be pure background.
    - Discard any window whose object score is below the threshold.
    - Apply **NMS** to merge or suppress highly overlapping windows (based on IoU) so that only the highest-confidence boxes remain.
- Draw bounding boxes
    - Each window that survived the NMS step is drawn as a bounding box.
    - Annotate it with the predicted class (dog/cat) and the confidence score.

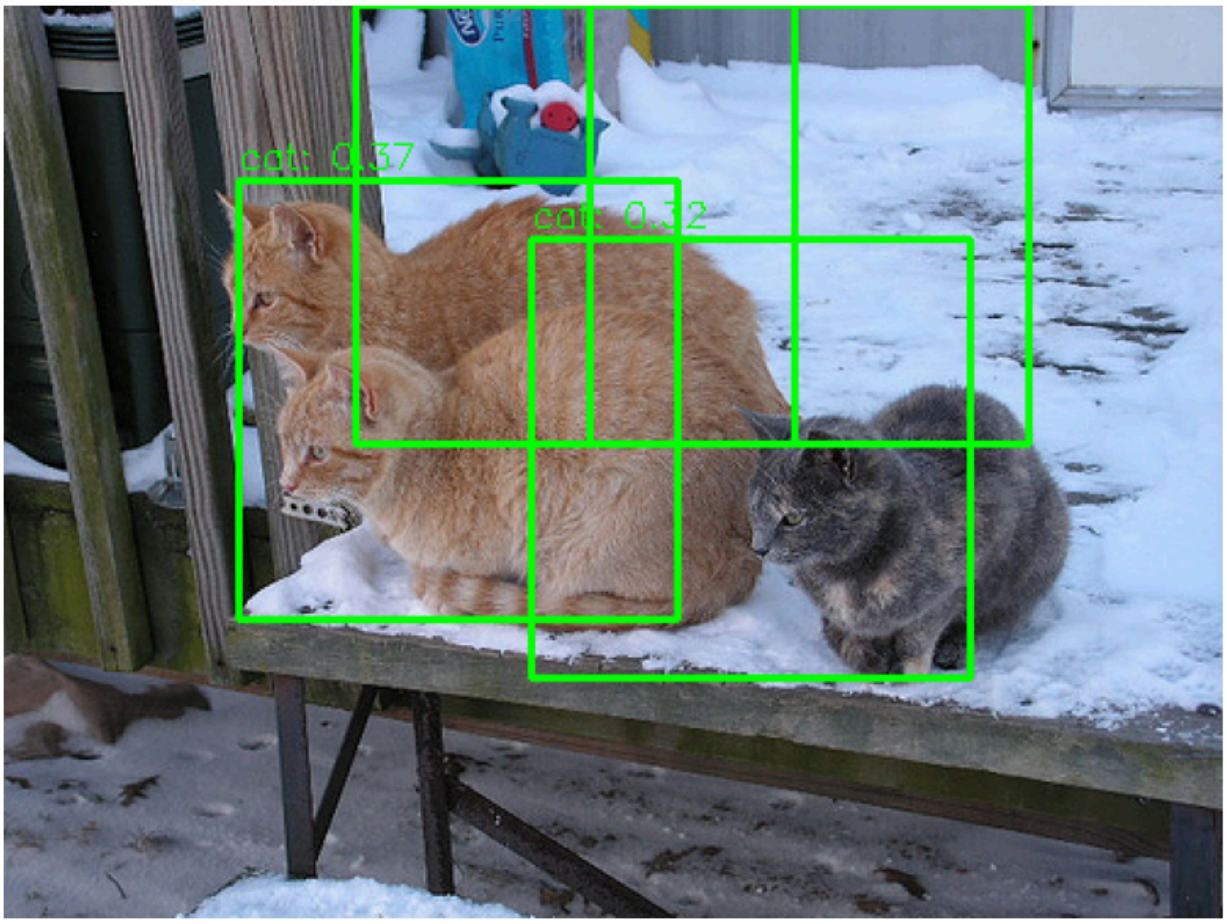


Why the Performance Is Not Great (Fixed-Range Sliding Windows)

- **Rigid window size**
  - A fixed-size window (e.g., 128*128) may not align well with objects that are smaller or bigger than the window, leading either to missed detections (if the object is too large) or lots of background in the window (if the object is smaller than the window).
  - After changing the window size to an appropriate value (e.g. 180*180), the detection performance seems way more better.



- **Single-scale assumption**
    - Real images can contain objects of various scales. Our method's performance strongly depends on the chosen window size.



Constantly modifying the sliding window and various thresholds for a particular image does give relatively acceptable results. However, such results cannot be generalised to other images.

Deliverable 4

Measuring Average Detection Latency

Loading two pretrained models—Ultralytics YOLO and TorchVision's Faster R-CNN—and measuring their per-image inference times on the 5,000 images in the COCO validation set was straightforward: load each model, run inference on each image (or in chunks), record the time, and then compute the average.

```
YOLO Average Detection Latency: 0.016159525108337403

Faster R-CNN Average Detection Latency) : 0.027700710868835448
```

Saving All Prediction Results

- Initially, I tried to store all detections from 5,000 images directly into a single Python list during inference, which caused excessive memory usage and slowdown. To address this, I adopted a **chunk-based** approach:
- For each chunk (e.g., 500 images), run inference and write the results immediately to a JSON file.
- Clear the in-memory list, then move on to the next chunk.
  This way, the overall process no longer caused severe RAM usage, and each partial result could be merged afterward.

Loading COCO Annotations Without the Official COCO API

The idea was to parse `instances_val2017.json` using Python's `json` module. However, a significant challenge arose: **COCO category IDs do not necessarily align with YOLO's class IDs.**

- For YOLO, the official 80 classes are often assigned IDs 0~79, but COCO's `category_id` are 1~90 (with some IDs missing or rearranged).

- To fix this mismatch, I spent considerable time constructing a **mapping function** that reconciles COCO's `category_id` with YOLO's or FRCNN's class numbering.
- **Note**: The category mismatch problem turned out to be complex. Here, I used GPT/Copilot to assist in building a proper "cat_id → class_id" mapping, which ultimately resolved issues like mixing up giraffes and backpacks, etc.

Computing mAP

I implemented a single IoU threshold (0.5) mAP computation using the 11-point interpolation. The procedure:

- Group predictions by class and sort them by confidence (descending).
- Match predicted boxes to ground-truth boxes via IoU ≥ 0.5.
- Calculate precision/recall curves and do 11-point interpolation to get AP per class.
- Average over all classes to obtain mAP.

With everything correctly aligned—i.e., chunked inference, proper category mapping, and a custom mAP evaluator—I obtained consistent, reasonable results for both YOLO and Faster R-CNN on COCO val2017.

Appendix

I've uploaded a zip file of the entire code work, where `Deliverable12.ipynb`, `Deliverable3.ipynb`, and `Deliverable4.ipynb` correspond to the code requirements in `HW3.pdf`.

Deliverable 1 → `Deliverable12.ipynb`

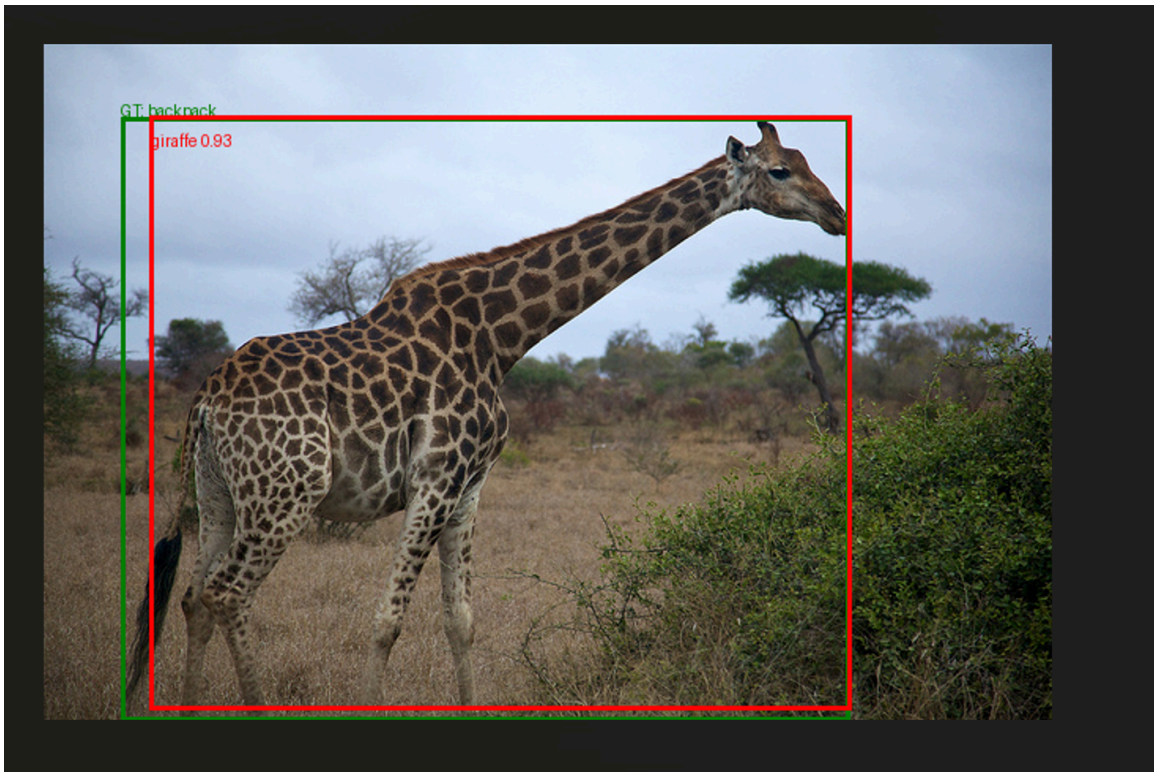
Deliverable 2 → `Deliverable12.ipynb`

Deliverable 3 → `Deliverable3.ipynb`

Deliverable 4 → `Deliverable4.ipynb`

Note: `Deliverable4.ipynb` uses a lot of functions defined in `utils_coco.py` (to handle importing annotations, calculating mAP, etc.).

Some of the outputs of `visualize_detections` function (I used this function to check whether the mapping function is working or not):



This is the image where I found that the model would wrongly map the categories and classes. (The ground truth was wrong because of my mapping function)