

# Homework 3: Supervised Learning

JUN-KAI CHEN, 111550001

April 14, 2025

## CNN

### load\_train\_dataset()

The content of `load_train_dataset()` function is shown in Figure 1.

This function traverses the given director, processing the child directories if its name is contained in the classes defined in `label_map` variable. The process of a child directory is to find all the files with jpg format, and add the files themselves to `images` while the names are added to `labels`.



```
1 def load_train_dataset(path: str='data/train/')->Tuple[List, List]:
2     images = []
3     labels = []
4     label_map = {
5         "elephant": 0,
6         "jaguar": 1,
7         "lion": 2,
8         "parrot": 3,
9         "penguin": 4
10    }
11
12    images = []
13    labels = []
14
15    for animal in os.listdir(path):
16        animal_path = os.path.join(path, animal)
17        if os.path.isdir(animal_path) and animal in label_map:
18            for img_file in os.listdir(animal_path):
19                if img_file.lower().endswith('.jpg'):
20                    img_path = os.path.join(animal_path, img_file)
21                    images.append(img_path)
22                    labels.append(label_map[animal])
23
24    return images, labels
```

Figure 1: The content of `load_train_dataset()` function.

### load\_test\_dataset()

The content of `load_test_dataset()` function is shown in Figure 2.

This function is quite similar to the previous one. There is only a difference that the test dataset has no label and is placed in a directory. Thus, the code of loading test dataset needn't traverse the child directory or process the label.

```

1 def load_test_dataset(path: str='data/test/')→List:
2     images = []
3     for img_file in os.listdir(path):
4         if img_file.lower().endswith('.jpg'):
5             img_path = os.path.join(path, img_file)
6             images.append(img_path)
7
8     return images

```

Figure 2: The content of `load_test_dataset()` function.

## `__init__()`

The content of `CNN().__init__()` function is shown in Figure 3.

This `CNN` class is inherited from `torch.nn.Module`, which is a base class in PyTorch model. The constructor receives a parameter to determine the classes of output. At here, the number of classes is 5, which means there are five types of animals.

The constructor creates three layers. Each layer contains convolution, batchnorm, ReLU, and max pooling. These four steps compose a typical CNN structure. At the end of the third layer, a fully connected layer is added to output results.

```

1 def __init__(self, num_classes=5):
2     super().__init__()
3     self.conv1 = nn.Sequential(
4         nn.Conv2d(3, 16, kernel_size=3, padding=1), # Output: (16, 224, 224)
5         nn.BatchNorm2d(16),
6         nn.ReLU(),
7         nn.MaxPool2d(2, 2) # Output: (16, 112, 112)
8     )
9     self.conv2 = nn.Sequential(
10        nn.Conv2d(16, 32, kernel_size=3, padding=1), # Output: (32, 112, 112)
11        nn.BatchNorm2d(32),
12        nn.ReLU(),
13        nn.MaxPool2d(2, 2) # Output: (32, 56, 56)
14    )
15    self.conv3 = nn.Sequential(
16        nn.Conv2d(32, 64, kernel_size=3, padding=1), # Output: (64, 56, 56)
17        nn.BatchNorm2d(64),
18        nn.ReLU(),
19        nn.MaxPool2d(2, 2) # Output: (64, 28, 28)
20    )
21    self.fc = nn.Sequential(
22        nn.Flatten(), # Output: 64*28*28 = 50176
23        nn.Linear(64 * 28 * 28, 512),
24        nn.ReLU(),
25        nn.Dropout(0.5),
26        nn.Linear(512, num_classes)
27    )

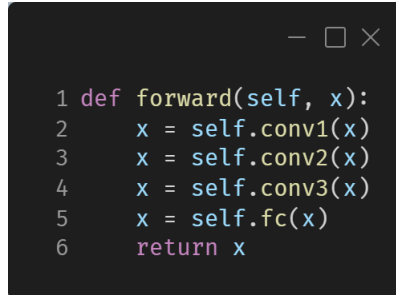
```

Figure 3: The content of `CNN().__init__()` function.

## forward()

The content of `CNN().forward()` function is shown in Figure 4.

This function is the most important part in a CNN model, but the operation is quite simple. I input the received vector `x` into the CNN and return the result. Notice that the `convX` and `fc` variables are declared as objects. These objects are callable since the `__call__()` function is implemented in the base class.



```
1 def forward(self, x):
2     x = self.conv1(x)
3     x = self.conv2(x)
4     x = self.conv3(x)
5     x = self.fc(x)
6     return x
```

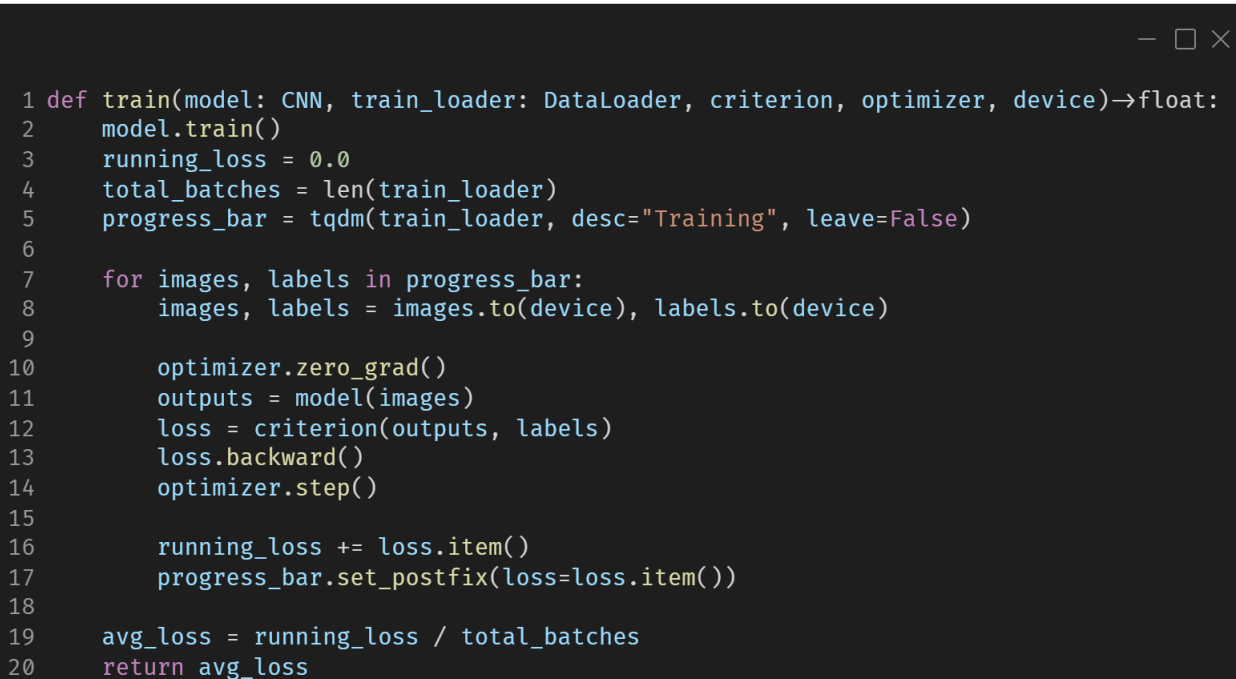
Figure 4: The content of `CNN().forward()` function.

## train()

The content of `CNN().train()` function is shown in Figure 5.

The `train()` function receives a CNN model, a loader, a criterion, and other parameters. At first, it set the model to train mode. Each batch of data, including images and labels, is then moved to a designated device. The next training step will first clear the accumulation of the previous training. The data is then fed into the model for forward propagation, and the difference is calculated based on the result to update the model parameters.

In order to better display the progress of training, a progress bar will be created during the training process through the `tqdm` module. After all data have been trained, that is, after one epoch, the average loss is calculated and returned.



```
1 def train(model: CNN, train_loader: DataLoader, criterion, optimizer, device) -> float:
2     model.train()
3     running_loss = 0.0
4     total_batches = len(train_loader)
5     progress_bar = tqdm(train_loader, desc='Training', leave=False)
6
7     for images, labels in progress_bar:
8         images, labels = images.to(device), labels.to(device)
9
10        optimizer.zero_grad()
11        outputs = model(images)
12        loss = criterion(outputs, labels)
13        loss.backward()
14        optimizer.step()
15
16        running_loss += loss.item()
17        progress_bar.set_postfix(loss=loss.item())
18
19    avg_loss = running_loss / total_batches
20    return avg_loss
```

Figure 5: The content of `CNN().train()` function.

## validate()

The content of `CNN().validate()` function is shown in Figure 6.

The `validate()` function is similar to the `train` function. The difference is that validation doesn't need to update the model or calculate parameters, so it sets model to evaluation mode and use the `no_grad()` function. It compares the data labels with the results obtained from the evaluation to calculate the accuracy of the model. The loss value and accuracy will be returned to the caller at the end of function.

```
1 def validate(model: CNN, val_loader: DataLoader, criterion, device)→Tuple[float, float]:
2     model.eval()
3     running_loss = 0.0
4     correct = 0
5     total = 0
6     total_batches = len(val_loader)
7
8     with torch.no_grad():
9         progress_bar = tqdm(val_loader, desc="Validating", leave=False)
10        for images, labels in progress_bar:
11            images, labels = images.to(device), labels.to(device)
12            outputs = model(images)
13            loss = criterion(outputs, labels)
14            running_loss += loss.item()
15            _, predicted = torch.max(outputs.data, 1)
16            total += labels.size(0)
17            correct += (predicted == labels).sum().item()
18            progress_bar.set_postfix(loss=loss.item())
19
20    avg_loss = running_loss / total_batches
21    accuracy = correct / total
22    return avg_loss, accuracy
```

Figure 6: The content of `CNN().validate()` function.

## test()

The content of `CNN().test()` function is shown in Figure 8.

The `test()` function mainly accepts a trained CNN model and the data set to be tested. Its work is somewhat similar to `validate`. The data will be fed into the model in sequence, obtain the predicted results, and finally write them into a csv file. Unlike the previous two functions, `test` itself does not know the labels of the data it is passed.

```

1 def test(model: CNN, test_loader: DataLoader, device):
2     model.eval()
3     predictions = []
4     with torch.no_grad():
5         for images, image_ids in tqdm(test_loader, desc="Testing", leave=False):
6             images = images.to(device)
7             outputs = model(images)
8             _, predicted = torch.max(outputs, 1)
9             predictions.extend(zip(image_ids, predicted.cpu().numpy()))
10    predictions.sort(key=lambda x: int(x[0]))
11    with open('CNN.csv', 'w') as f:
12        f.write('id,prediction\n')
13        for image_id, pred in predictions:
14            f.write(f'{image_id},{pred}\n')
15    print(f"Predictions saved to 'CNN.csv'")
16    return

```

Figure 7: The content of `CNN().test()` function.

## plot()

The content of `plot()` function is shown in Figure 8.

The `plot` function uses the matplotlib package. First, it sets up a fixed-size canvas and plots the received training and validation errors.

Figure 9 is the loss picture painted by the `plot`

```

1 def plot(train_losses: List, val_losses: List):
2     plt.figure(figsize=(10, 6))
3     epochs = range(1, len(train_losses) + 1)
4
5     plt.plot(epochs, train_losses, label='Train Loss', marker='o')
6     plt.plot(epochs, val_losses, label='Validation Loss', marker='o')
7
8     plt.xlabel('Epoch')
9     plt.ylabel('Loss')
10    plt.title('Training and Validation Loss')
11    plt.legend()
12    plt.grid(True)
13    plt.savefig('loss.png')
14    plt.close()
15
16    print("Save the plot to 'loss.png'")
17    return

```

Figure 8: The content of `plot()` function.

## Experiments

As can be seen in Figure 9, at epoch 4, the train loss decreases, indicating that the model has memorized the data used for training. However, the validation loss increases significantly, which means that the model has over-memorized the training data, resulting in poor performance on unseen data. That is, overfitting occurs. Also in the last epoch, there is a slight overfitting phenomenon.

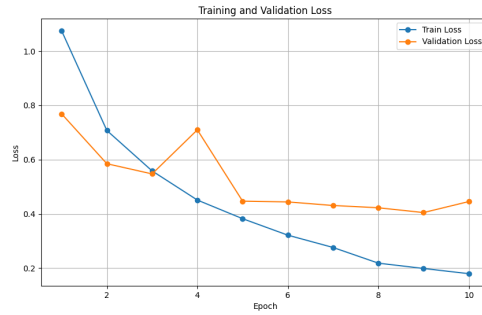


Figure 9: The loss of each epoch.

Two strategies are used here to prevent the model from overfitting. The first is to add Dropout to the third convolution layer and modified the parameter in the fully connected layer, which can effectively reduce the probability of neurons relying on specific paths and improve the ability to respond to different situations. The second is to add weight decay, which will penalize model parameters that are too large and prevent the model from overfitting the training data.

The code for the two strategies is posted in Figures 10 and 11, and the loss results are shown in Figures 12 and 13. The original CNN used Dropout in the fully connected layer. After adding the effect of Dropout, the training accuracy has been improved. In addition, it can be seen from Figure 10 that the overfitting situation has been improved. The strategy of using weight decay can also improve accuracy and reduce overfitting, but the effect is not as good as the former.

```

○○○

1 self.conv3 = nn.Sequential(
2     nn.Conv2d(32, 64, kernel_size=3, padding=1), # Output: (64, 56, 56)
3     nn.BatchNorm2d(64),
4     nn.ReLU(),
5     nn.MaxPool2d(2, 2), # Output: (64, 28, 28)
6     nn.Dropout2d(0.3)
7 )
8 self.fc = nn.Sequential(
9     nn.Flatten(), # Output: 64*28*28 = 50176
10    nn.Linear(64 * 28 * 28, 512),
11    nn.ReLU(),
12    nn.Dropout(0.6),
13    nn.Linear(512, num_classes)
14 )

```

Figure 10: The modification of CNN with dropout method.

```

○○○

1 optimizer = optim.Adam(base_params, lr=1e-4, weight_decay=1e-4)

```

Figure 11: The modification of CNN with weight decay method.

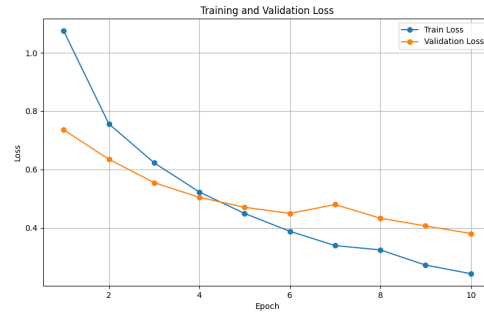


Figure 12: The loss of each epoch with dropout method.

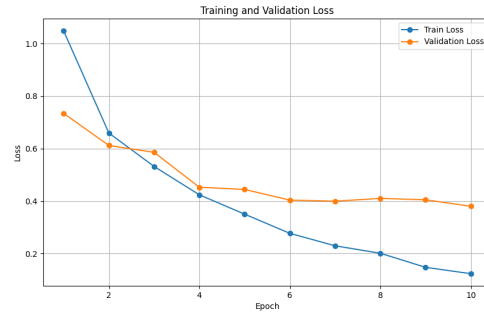


Figure 13: The loss of each epoch with weight decay method.

## Decision Tree

### get\_features\_and\_labels()

The content of `get_features_and_label()` function is shown in Figure 14.

This function receives a pre-trained CNN model, in this case a `mobilenetv3_small_100` model. It uses the model to extract the features of given data, which is another parameter. The features will be casted to `pd.DataFrame` type to further operations.

```

1 def get_features_and_labels(model: ConvNet, dataloader: DataLoader, device)→Tuple[List, List]:
2     model.eval()
3     features = []
4     labels = []
5     with torch.no_grad():
6         for images, lbls in tqdm(dataloader, desc="Extracting features"):
7             images = images.to(device)
8             outputs = model(images)
9             features.extend(outputs.cpu().numpy())
10            labels.extend(lbls.numpy())
11    features_df = pd.DataFrame(features)
12    labels_np = np.array(labels)
13    return features_df, labels_np

```

Figure 14: The content of `get_features_and_label()` function.

### get\_features\_and\_paths()

The content of `get_features_and_paths()` function is shown in Figure 15.

This function is similar to the previous function. It extracts the features and maps to the path of the train data. An important thing is that the program sorts the path list since the original list is in lexicographic order and what the main function needs is in integer order.

```
1 def get_features_and_paths(model: ConvNet, dataloader: DataLoader, device)→Tuple[List, List]:
2     model.eval()
3     features = []
4     paths = []
5     with torch.no_grad():
6         for images, image_names in tqdm(dataloader, desc="Extracting features"):
7             images = images.to(device)
8             outputs = model(images)
9             features.extend(outputs.cpu().numpy())
10            paths.extend(image_names)
11    sorted_indices = sorted(range(len(paths)), key=lambda i: int(paths[i]))
12    sorted_features = [features[i] for i in sorted_indices]
13    sorted_paths = [paths[i] for i in sorted_indices]
14    features_df = pd.DataFrame(sorted_features)
15    return features_df, sorted_paths
```

Figure 15: The content of `get_features_and_paths()` function.

## `_build_tree()`

The content of `_build_tree()` function is shown in Figure 16.

This function builds a decision tree based on the input features and uses the `_best_split()` function to split the data. If a valid split is found, the data is divided into left and right subsets using `_split_data()` function. Each internal node and its children store the features and thresholds used for splitting. This process continues until the entire tree is built, where nodes are represented as nested dictionaries containing either class predictions or further splits.

```
1 def _build_tree(self, X: pd.DataFrame, y: np.ndarray, depth : int = 0):
2     min_samples_split = 10
3     num_samples_per_class = np.bincount(y)
4     predicted_class = np.argmax(num_samples_per_class)
5     if depth >= self.max_depth or len(set(y)) == 1 or X.empty or len(y) < min_samples_split:
6         return {'type': 'leaf', 'class': predicted_class}
7     if depth >= self.max_depth or len(set(y)) == 1 or X.empty:
8         return {'type': 'leaf', 'class': predicted_class}
9     feature_index, threshold = self._best_split(X, y)
10    if feature_index is None:
11        return {'type': 'leaf', 'class': predicted_class}
12    left_X, left_y, right_X, right_y = self._split_data(X, y, feature_index, threshold)
13    self.progress.update(1)
14    return {
15        'type': 'node',
16        'feature_index': feature_index,
17        'threshold': threshold,
18        'left': self._build_tree(left_X, left_y, depth + 1),
19        'right': self._build_tree(right_X, right_y, depth + 1)
20    }
```

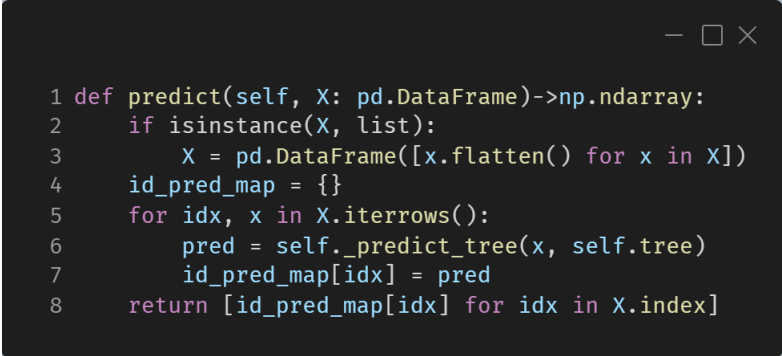
Figure 16: The content of `_build_tree()` function.



## predict()

The content of `predict()` function is shown in Figure 17.

This function generates class predictions for input samples using a trained decision tree. If the input is a list, it is first converted into a DataFrame. Then, for each sample, the `_predict_tree` method is called to traverse the tree and determine its class. The results are returned as a list, preserving the original sample order.



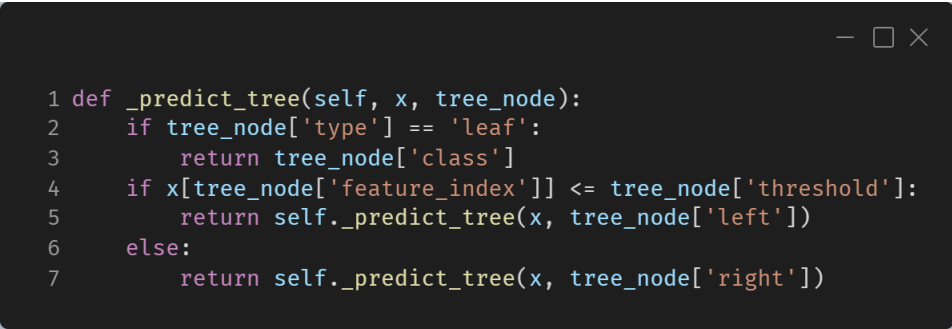
```
1 def predict(self, X: pd.DataFrame)->np.ndarray:
2     if isinstance(X, list):
3         X = pd.DataFrame([x.flatten() for x in X])
4     id_pred_map = {}
5     for idx, x in X.iterrows():
6         pred = self._predict_tree(x, self.tree)
7         id_pred_map[idx] = pred
8     return [id_pred_map[idx] for idx in X.index]
```

Figure 17: The content of `predict()` function.

## \_predict\_tree()

The content of `_predict_tree()` is shown in Figure 18.

This function recursively traverses the decision tree to predict the class of a single input sample `x`. At each node, it checks whether the sample's feature value is below the threshold to decide whether to follow the left or right subtree. When a leaf node is reached, it returns the predicted class.



```
1 def _predict_tree(self, x, tree_node):
2     if tree_node['type'] == 'leaf':
3         return tree_node['class']
4     if x[tree_node['feature_index']] <= tree_node['threshold']:
5         return self._predict_tree(x, tree_node['left'])
6     else:
7         return self._predict_tree(x, tree_node['right'])
```

Figure 18: The content of `_predict_tree` function.

## \_split\_data()

The content of `_split_data` function is shown in Figure 19.

This function splits the dataset into two subsets based on a specified feature index and threshold. Samples with values less than or equal to the threshold go to the left subset, while the rest go to the right. It returns the corresponding features (`X`) and labels (`y`) for both subsets.

## \_best\_split()

The content of `_best_split` function is shown in Figure 19.

```

1 def _split_data(self, X: pd.DataFrame, y: np.ndarray, feature_index: int, threshold: float):
2     feature = X.iloc[:, feature_index]
3     left_mask = feature <= threshold
4     right_mask = ~left_mask
5     left_dataset_X = X[left_mask]
6     left_dataset_y = y[left_mask]
7     right_dataset_X = X[right_mask]
8     right_dataset_y = y[right_mask]
9     return left_dataset_X, left_dataset_y, right_dataset_X, right_dataset_y

```

Figure 19: The caption of `_split_data` function.

The `_best_split` function finds the optimal feature and threshold to split the dataset in order to maximize information gain. For each feature, it tests the 25th, 50th, and 75th percentiles as potential thresholds. It calculates the entropy for the left and right subsets and selects the split with the highest information gain. It returns the index of the best feature and the corresponding threshold.

```

1 def _best_split(self, X: pd.DataFrame, y: np.ndarray):
2     best_gain = -1
3     best_feature = None
4     best_thresh = None
5     current_entropy = self._entropy(y)
6     for feature_index in range(X.shape[1]):
7         values = X.iloc[:, feature_index].values
8         thresholds = np.percentile(values, [25, 50, 75])
9         for threshold in thresholds:
10             left_y = y[values <= threshold]
11             right_y = y[values > threshold]
12             if len(left_y) == 0 or len(right_y) == 0:
13                 continue
14             left_entropy = self._entropy(left_y)
15             right_entropy = self._entropy(right_y)
16             weighted_entropy = (len(left_y) * left_entropy + len(right_y) * right_entropy) /
17 len(y)
18             info_gain = current_entropy - weighted_entropy
19             if info_gain > best_gain:
20                 best_gain = info_gain
21                 best_feature = feature_index
22                 best_thresh = threshold
23     return best_feature, best_thresh

```

Figure 20: The content of `_best_split` function.

## `_entropy()`

The content of `_best_split` function is shown in Figure 19.

The `_entropy` function calculates the entropy of a label array `y`, which measures the impurity or randomness of the class distribution. It computes the probability of each unique class and then applies the entropy formula:

$$\text{Entropy} = - \sum p(x) \log_2 p(x)$$

The function returns this entropy value as a float.

```
1 def _entropy(self, y: np.ndarray)->float:
2     unique_classes, class_counts = np.unique(y, return_counts=True)
3     probabilities = class_counts / len(y)
4     entropy_value = -np.sum(probabilities * np.log2(probabilities))
5     return entropy_value
```

Figure 21: The content of `_entropy` function.

## Experiment

When the depth is 5, the validate accuracy is 0.7498. When the depth is 7 or 9, the values are 0.7754 and 0.8098 respectively. Since the accuracy increases with the depth, we can know that up to this depth, the model is still in the correct learning stage and has not yet experienced overfitting. Higher depth allows the tree to consider more features and therefore make better decisions.

## Kaggle Scoring

### CNN

The score of CNN on Kaggle is shown in Figure 20.


7	111550001		0.874	1	1d
---	-----------	---	-------	---	----

Figure 22: The score of CNN on Kaggle.

### Decision Tree

The score of decision tree is shown in Figure 21.


8	111550001		0.774	1	2h
---	-----------	---	-------	---	----

Figure 23: The score of decision tree on Kaggle.