Image Classification for CIFAR-10

Elian Iluk, Ariel Levovich, Tomer Shor

The Problem

The goal of our project is to classify a given image into one of 10 classes: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship,* and *truck*. To achieve this, we implemented four models of increasing complexity:

- **Baseline Model:** Random guessing.
- **SoftMax Model:** A single linear layer with SoftMax activation.
- **Basic Neural Network:** Fully-connected layers with ReLU activation.
- **Convolutional Neural Network (CNN):** Complex architecture using convolutional, batch normalization, dropout, and max-pooling layers.

The Dataset

The CIFAR-10 dataset is a widely-used benchmark dataset for image classification tasks. It consists of 60,000 32x32 color images, evenly distributed across 10 classes. There are 50,000 images for training and 10,000 for testing. The dataset is balanced and diverse, making it suitable for entry-level testing of deep learning models.

Models and Results

Baseline Model

This model is the simplest one as it only guesses a number between 1 to 10 and then classify the image to the class which its index is the number we guessed. As expected we got 0.1 accuracy (as there is only 10 classes and the probability to choose each class is 10).

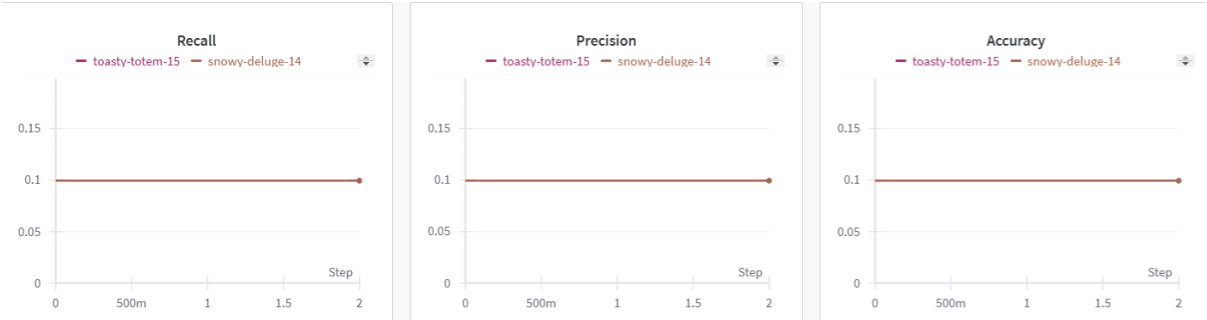

Figure 1: Baseline Model Results: Accuracy, Precision, and Recall.

Observations:

- Accuracy, precision, and recall are uniformly low at 9.94% due to random guessing.
- This model requires negligible training time.

Baseline Code

```
np.random.seed(42)
num_classes = 10

random_predictions = np.random.randint(0, num_classes,
↪  size=len(test_dataset))

true_labels = [label for _, label in test_dataset]

accuracy = accuracy_score(true_labels, random_predictions)
precision = precision_score(true_labels, random_predictions,
↪  average='macro', zero_division=0)
recall = recall_score(true_labels, random_predictions, average='macro',
↪  zero_division=0)

```

SoftMax Model

This model uses the SoftMax function to calculate the probability to the image to be in the each class and then classify the image to the class which got the highest probability. This model process the image through 2 layers, the first is a Linear layer which just multiply the image matrix and another weight matrix and add a bias, the results being process to the second layer which is a SoftMax which result one 1*10 size vector with the probabilities. The model is trained for a fixed number of epochs using the SGD+momentum optimizer and several different augmentations to the image, after each epoch there is a validation part where we can see the development of the model and afterwards we test our model.

0.1 results

Our model gave a test accuracy of 0.4, which is a massive improvement from the baseline model, but still not good enough as there is still plenty of room for improvement. We can see in the results that we tried different things to improve the accuracy- different optimizers (Adam, SGD...), different number of epochs(30,50...), different learning rates(0.1, 0.01, 0.001....) and more. The best result was using SGD+momentum optimizer for 50 epochs with 0.001 learning rate.

Figure 2: SoftMax Model different runs.

Observations:

- Test accuracy improves to 40.42%, marking a substantial improvement over the baseline.
- Precision and recall align closely with the test accuracy, indicating balanced performance.
- Training time is relatively short, making this model efficient for simple tasks.

SoftMax Code

```
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        return self.softmax(self.linear(x))

```

Basic Neural Network

This model is one of two models which uses a neural network type architecture to answer our problem. This model uses a several fully connected layers to process the image through which the dimension of the output matrix get smaller each time to process good the image and in the end get a matrix with only 10 slots which represent the probabilities for each class and then classify the image to the class with the highest probability. This model also uses the RELU function as an activation function after each Linear layer to turn each element to be in the range of [0-infinite].

0.2 results

This model gave much better results from the models before as it provided a 0.6 test accuracy(and similar precision and recall) which is 0.2 better than the SoftMax model

and 0.5 better than the baseline model. As before, we tried different things to improve the accuracy of the test, and we found out that the best were to use 100 epochs with 6 fully connected layers using SGD+momentum optimizer and Cross Entropy as the loss function alongside 0.01 learning rate.

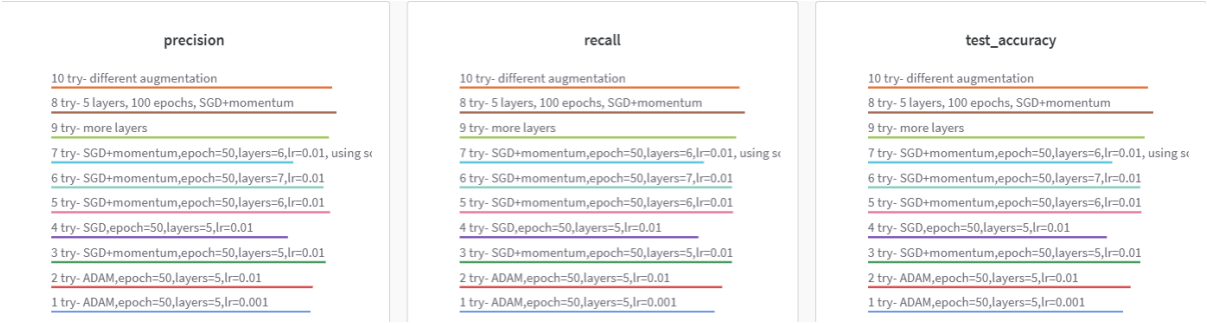

Figure 3: Basic Neural Network different runs.

- Accuracy improves significantly to 59.95%, with precision and recall closely following.
- This model demonstrates the impact of added complexity with fully-connected layers and ReLU activation.
- Training time increases but remains manageable for typical hardware setups.

Basic Neural Network Code

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.relu = nn.ReLU()
        self.fc6 = nn.Linear(32*32*3, 1024)
        self.fc1 = nn.Linear(1024, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc6(x))
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.relu(self.fc4(x))
        x = self.fc5(x)
        return x

```

Convolutional Neural Network

This model is the most complex one. This uses different kinds of layers to classify the image correctly:

1. **Convolution Layers:** These layers use sliding windows to identify different patterns in the image. The initial layers detect simple patterns (such as shapes and lines), while deeper layers identify more complex patterns (such as faces or specific objects). In our problem, we experimented with various numbers of convolutional layers and selected the configuration that gave the best results alongside different parameters.

2. **Batch Normalization Layers:** These layers normalize the input to each layer, ensuring a mean of 0 and a variance of 1 within a mini-batch. This helps maintain a stable input distribution to the layers, reducing the likelihood of the model getting stuck in poor parameter regions and resulting in better and faster training. We applied batch normalization after each convolutional layer to maintain stable data, which was a significant factor in achieving good accuracy.

3. **Dropout Layers:** Dropout is a regularization technique used in deep learning to prevent overfitting by introducing noise into the training process. It works by randomly "dropping out" (setting to zero) a subset of neurons during each training iteration. We added dropout layers after each fully connected layer, as skipping essential layers such as convolution or batch normalization would not be beneficial. This layer was crucial in preventing overfitting.

4. **Max-Pooling Layers:** These layers downsample the image, decreasing its size to help find important features. We applied max-pooling after every two convolutional layers. This allowed the model to learn features from the "big" image and progressively downsample it to focus on essential features, ultimately improving classification accuracy.

0.3 results

This model outperforms all the other models as it reaches test accuracy of 0.9 which is 0.3 higher than the simple fully connected model, 0.5 higher than the SoftMax model and massive 0.8 higher than the baseline model (the precision and recall similar). This model is by far the best model to use when trying to face our problem.

In this model we also tried several things to try to improve the test accuracy, as we tried different number of layers (conc, batch norm, maxpool and more...) and different settings for each. And the best settings for this model was to use 400 epochs with learning rate of 0.01, the SGD+momentum optimizer, 4 conv layers and 4 fully-connected layers, using the Cross Entropy as the function loss (the exact settings for each layer provided in the code itself).

To achieve these results we tried several different settings to get the lowest loss and the best test accuracy (precision and recall follows) while preventing overfitting and underfitting.

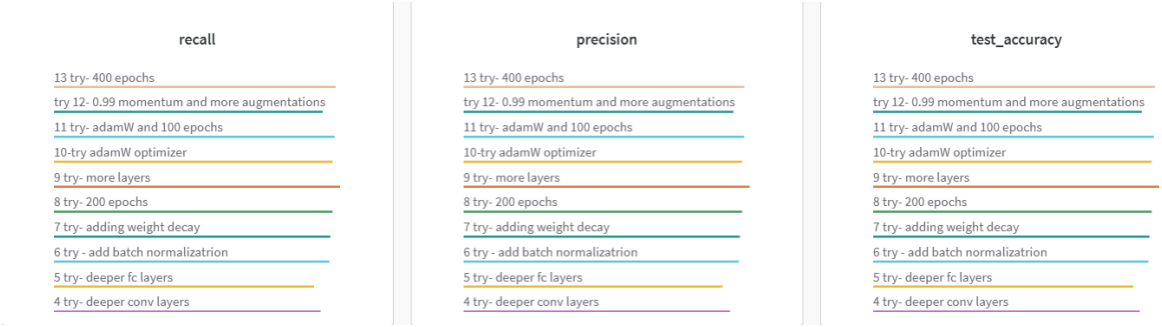

Figure 4: Convolutional Neural Network different runs.

Observations:

- The CNN achieves an impressive test accuracy of 90.56%, significantly outperforming all other models.
- Precision and recall metrics are also the highest, demonstrating the robustness of the model.
- Training time is notably longer due to the computational cost of convolutional and normalization layers.
- This model is highly suitable for complex image classification tasks requiring high accuracy.

Convolution Neural Network Code

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.batchNorm1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.batchNorm2 = nn.BatchNorm2d(64)

        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.batchNorm3 = nn.BatchNorm2d(128)

        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
        self.batchNorm4 = nn.BatchNorm2d(256)

        self.maxpool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(256 * 8 * 8, 512)
        self.batchNorm5 = nn.BatchNorm1d(512)

        self.fc2 = nn.Linear(512, 256)
        self.batchNorm6 = nn.BatchNorm1d(256)
```

```
        self.fc3 = nn.Linear(256, 128)
        self.batchNorm7 = nn.BatchNorm1d(128)

        self.fc4 = nn.Linear(128, 10)
        self.batchNorm8 = nn.BatchNorm1d(64)

        self.dropout = nn.Dropout(0.5)
        self.relu = nn.ReLU()


    def forward(self, x):
        x = self.relu(self.batchNorm1(self.conv1(x)))
        x = self.maxpool(self.relu(self.batchNorm2(self.conv2(x))))
        x = self.relu(self.batchNorm3(self.conv3(x)))
        x = self.maxpool(self.relu(self.batchNorm4(self.conv4(x))))

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

        x = self.dropout(self.relu(self.batchNorm5(self.fc1(x))))
        x = self.dropout(self.relu(self.batchNorm6(self.fc2(x))))
        x = self.dropout(self.relu(self.batchNorm7(self.fc3(x))))

        x = self.fc4(x)
        return x
```

Comparison Between the Models

Model Name	Test Accuracy (%)	Precision (%)	Recall (%)
Baseline	9.94	9.94	9.94
SoftMax	40.42	39.93	40.42
Fully Connected Model	59.95	59.66	60.23
CNN	90.56	89.46	89.28

Table 1: Comparison of Test Accuracy, Precision, and Recall Across Models.

Model Name	train time (seconds)
Baseline	5
SoftMax	120
Fully Connected Model	300
CNN	1200

Table 2: Comparison of training time Across Models.

Observations:

- The CNN significantly outperforms other models in terms of accuracy, precision, and recall but requires much more training time.
- Simpler models like the SoftMax model are computationally efficient and could be suitable for less complex tasks.
- Fully connected models strike a balance between performance and training complexity.

Shared Code

Data Augmentation and Loaders

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(train_data,
↪   batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = DataLoader(test_set, batch_size=128, shuffle=False)

```

Training and Validation phase

```
for epoch in range(1, n_epochs + 1):
    train_loss = 0.0
    valid_loss = 0.0
    correct_train = 0
    total_train = 0

    model.train()
    for data, target in train_loader:
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
```

```
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)
        _, preds = torch.max(output, 1)
        correct_train += (preds == target).sum().item()
        total_train += target.size(0)

    model.eval()
    correct_valid = 0
    total_valid = 0
    for data, target in valid_loader:
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        output = model(data)
        loss = criterion(output, target)
        valid_loss += loss.item() * data.size(0)
        _, preds = torch.max(output, 1)
        correct_valid += (preds == target).sum().item()
        total_valid += target.size(0)

    train_loss /= len(train_loader.sampler)
    valid_loss /= len(valid_loader.sampler)
    train_accuracy = correct_train / total_train
    valid_accuracy = correct_valid / total_valid

```

Test phase

```
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

model.eval()
y_true = []
y_pred = []

for data, target in test_loader:
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    output = model(data)
    loss = criterion(output, target)
    test_loss += loss.item() * data.size(0)
    _, pred = torch.max(output, 1)
    y_true.extend(target.cpu().numpy())
    y_pred.extend(pred.cpu().numpy())
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.cpu().numpy())
    for i in range(batch_size):
```

```
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

test_loss /= len(test_loader.dataset)
test_accuracy = 100. * np.sum(class_correct) / np.sum(class_total)

precision = precision_score(y_true, y_pred, average='macro')
recall = recall_score(y_true, y_pred, average='macro')

```

Conclusion

The results of this project demonstrate the significant impact of model complexity on performance. While the baseline model provided minimal accuracy, each subsequent model improved upon the previous one, with the CNN achieving an impressive 90% test accuracy. This highlights the importance of employing advanced architectures like convolutional layers, batch normalization, and dropout to handle complex image classification tasks. Future research could explore additional techniques such as transfer learning, hyperparameter optimization, or experimenting with deeper architectures to further enhance model performance. The insights gained from this project underline the effectiveness of deep learning methods in tackling challenging problems like image classification.