Neural Network Models for Object Recognition

Christopher Final

Learning objectives

- Develop a Neural Network using the CIFAR-10 image dataset for object recognition.
- CIFAR-10 is derived from the "80 million tiny images" dataset and comprises 60,000 coloured images, and was curated by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton (Krizhevsky et al 2009).
- Primary objective is to design the neural networks and assess the model's efficiency. It's imperative to partition a validation set from the provided training dataset.

What is CIFAR 10 and its purpose in Machine Learning

CIFAR 10 is a dataset made up of 60,000 images that are 32x32 pixels.

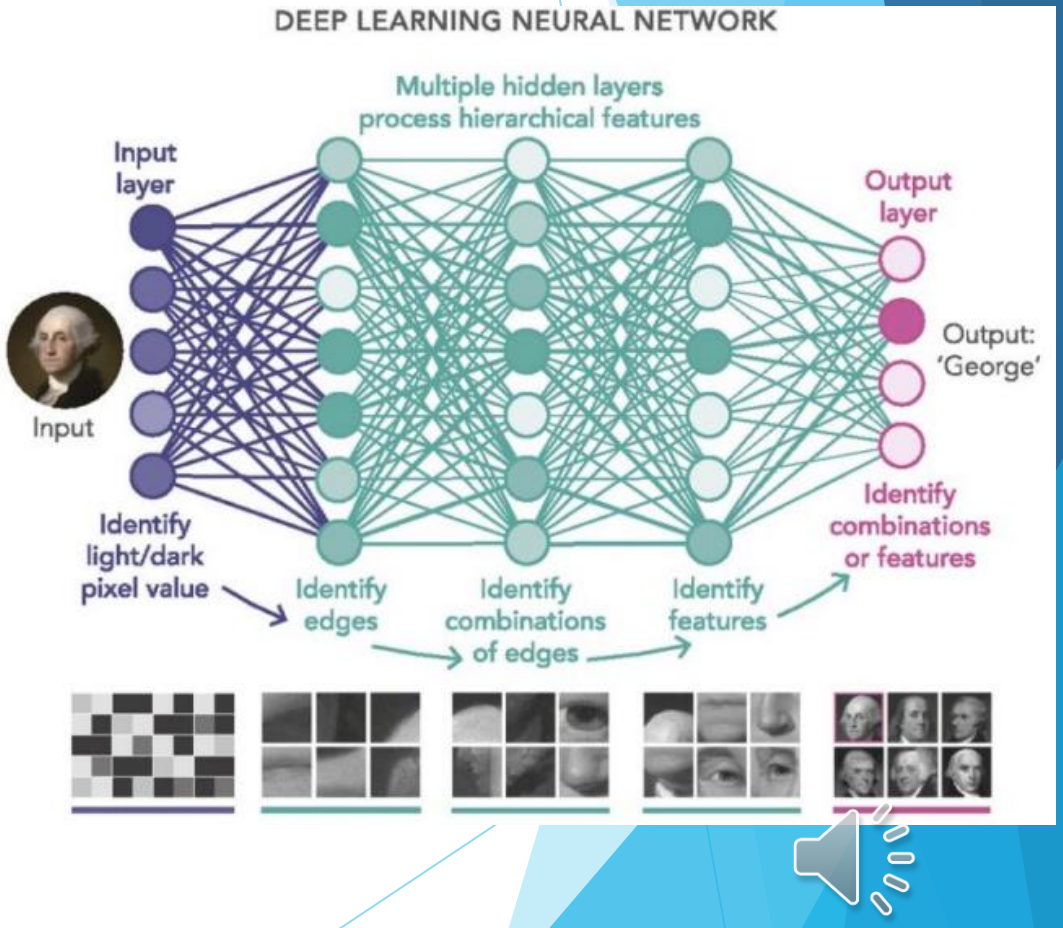
50,000 images are in the **Training Set**, whilst 10,000 are in the **Test Set**.

These images are unclear, so the aim is to develop a system capable of accurately categorising new, unseen images (Test Set) into one of the **ten predefined categories**, based off the learning from **training set**.

Deer or plane?

How will we achieve a Training model?

1. **Data Preprocessing** - Prepare the dataset for training.
2. **Model Definition** - Define a neural network architecture.
3. **Training the model** – Creating an algorithm to calculate/minimise the loss function.
4. **Evaluation** - Evaluate the trained model on the test set to determine its accuracy and ability to generalize to new images.
5. **Prediction** - Use the trained model to predict the category of new, unseen images.



Data Preprocessing

- To start, preparing for training by normalising the image data and converting it into a suitable format for the neural network.
- We have to convert pixel values range of 0-255, into 0-1. This makes the data distribution more standard, which helps the **neural network learn more effectively**.
- The values 0.4914, 0.4822, 0.4465 represent the mean values of Red, Green, and Blue.
- The other values, 0.2023, 0.1994, 0.2010, are the standard deviation of Red, Green, and Blue.

```
# Define transformations for the training and testing sets
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
```

Splitting into training set and validation set

- To train our Neural Network I have defined a training set, ‘trainset’.
- The validation set or ‘testloader’ is the variable I use later to see how well the Neural Network can evaluate an image.

```
# Load the CIFAR-10 dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=5,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=5,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

How a CNN Trains

- Creating a neural network architecture, for handling image data was next. I used a Convolutional Neural Network (CNN).
- As shown by Figure 1 by Guo et al (2017), a CNN involves three layers; **Convolutional**, **pooling**, and **fully connected**.
- By stacking convolutional layers, introducing non-linearity, reducing complexity, and extracting features, I can now produce a **Classification Output**.

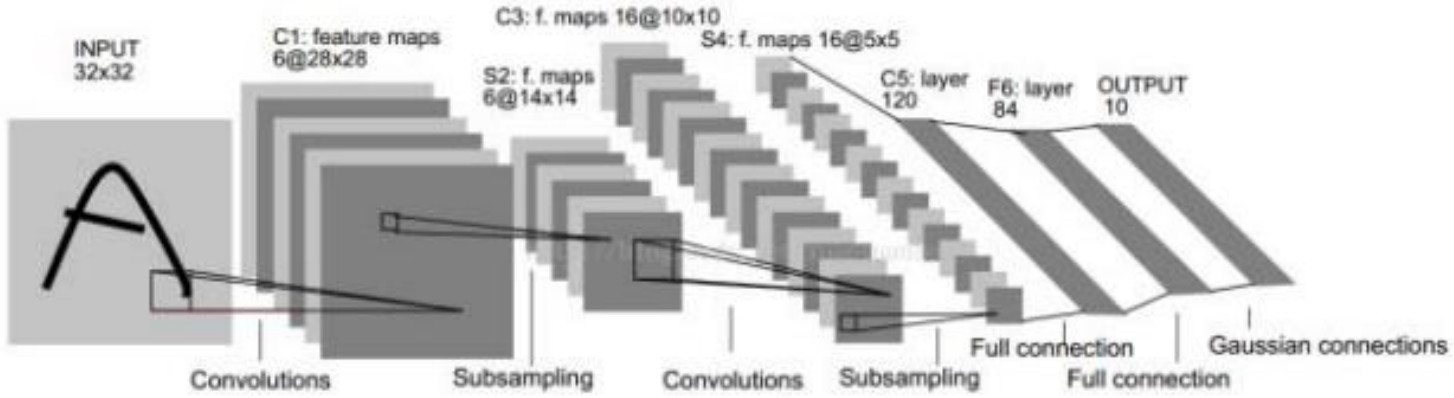


Figure 1. The architecture of LeNet-5[1] network(adapted from [1])

Model Definition

- The code below goes through the steps on the previous slide that were; parse into convolutional layer 1, insert into pooling layer, parse into convolutional layer 2, parse into fully connected layers 1,2 and 3.
- Then a **Forward Pass** to allow for a prediction of each image.

```
# Defining a simple Convolutional Neural Network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5) #First convolutional layer - 3 input channels RGB,6 output channels,
        self.pool = nn.MaxPool2d(2, 2) #Pooling - taking 2x2 pixel patch of each image
        self.conv2 = nn.Conv2d(6, 16, 5) #Second convolutional layer - take the 6 channel output from conv1
        self.fc1 = nn.Linear(16 * 5 * 5, 120) #First fully connected layer
        self.fc2 = nn.Linear(120, 84) #Second fully connected layer - output of 120 is feeded back into fc2
        self.fc3 = nn.Linear(84, 10) #Output Layer - has to fit into 10 classes w/probability

    def forward(self, x): #declare forward pass of function
        x = self.pool(F.relu(self.conv1(x))) #activation function 1
        x = self.pool(F.relu(self.conv2(x))) #activation function 2
        x = x.view(-1, 16 * 5 * 5) #Flatten
        x = F.relu(self.fc1(x)) #Fully connected layer 1 passed back into activation function
        x = F.relu(self.fc2(x)) #FCL2 passed back into activation function
        x = self.fc3(x)
        return x

net = Net()
```

Training to adjust the error/weights

- Training the model involves adjusting the model's weights based on the training data. This phase includes the actual learning process where the model parameters are optimised to minimise the loss function.
- As stated by Kubat (2017), “The task for machine learning is to provide algorithms capable of finding weights that result in good classification behaviour”.
- Using backpropagation and gradient descent for optimisation, we find the ‘loss’ by measuring the distance between the predicted output and the actual label of the training set images.
- Over multiple iterations (epochs), this process allows the CNN to fine-tune its weights, gradually learning to extract relevant features and accurately classify the images.

Training the model

- In my code, I have set parameters up for the training model.
- For example; **Learning rate** of 0.001, 10 **epoch's** (number of cycles), and the **running loss**.

```
# Define a Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the network
for epoch in range(10):  # loop over the dataset multiple times. Can Alter Epoch e.g. in range(XXX)
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        optimizer.zero_grad() # zero the parameter gradients
        outputs = net(inputs) #Forward pass
        loss = criterion(outputs, labels) #Compute the loss
        loss.backward() #Backward pass (compute gradients)
        optimizer.step() #Update weights
        # print statistics
        running_loss += loss.item()
        if i % 200 == 199:    # print every 200 mini-batches
            print(f'[Epoch {epoch + 1}, Batch {i + 1}] loss: {running_loss / 200:.3f}')
            running_loss = 0.0

print('Finished Training')
```

```
[Epoch 1, Batch 200] loss: 2.295
[Epoch 1, Batch 400] loss: 2.250
[Epoch 1, Batch 600] loss: 2.148
[Epoch 1, Batch 800] loss: 2.042
[Epoch 1, Batch 1000] loss: 1.920
[Epoch 1, Batch 1200] loss: 1.943
[Epoch 1, Batch 1400] loss: 1.860
[Epoch 1, Batch 1600] loss: 1.814
[Epoch 1, Batch 1800] loss: 1.785
[Epoch 1, Batch 2000] loss: 1.742
[Epoch 1, Batch 2200] loss: 1.748
[Epoch 1, Batch 2400] loss: 1.756
[Epoch 1, Batch 2600] loss: 1.676
[Epoch 1, Batch 2800] loss: 1.723
[Epoch 1, Batch 3000] loss: 1.732
[Epoch 1, Batch 3200] loss: 1.703
[Epoch 1, Batch 3400] loss: 1.665
[Epoch 1, Batch 3600] loss: 1.661
[Epoch 1, Batch 3800] loss: 1.596
[Epoch 1, Batch 4000] loss: 1.649
[Epoch 1, Batch 4200] loss: 1.618
[Epoch 1, Batch 4400] loss: 1.602
```

Evaluation

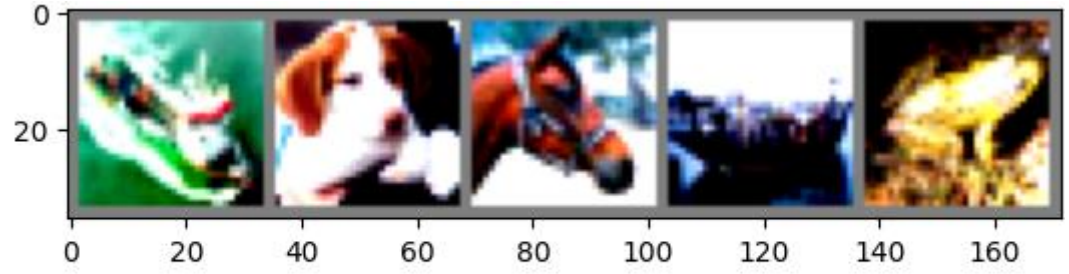
- Able to now evaluate the trained model on the test set. Determining its accuracy and ability to generalise to unseen images.
- The images were passed through variable ‘net’ for network which was defined in the previous step ‘training the model’.
- The accuracy from my run through was 61.44% on the 10,000 test images.

```
# Save the trained model
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

# Test the network on the test data
net.load_state_dict(torch.load(PATH))

correct = 0
total = 0
with torch.no_grad():
    for data in testloader: #loading in test data
        images, labels = data
        outputs = net(images) #Creating forward pass
        _, predicted = torch.max(outputs.data, 1) #converts output to predicted label class (1-10)
        total += labels.size(0) #x+=y same as x=x+y creates accumulated figures
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct / total:.2f}%')
```

```
Accuracy of the network on the 10000 test images: 61.44%
```

Prediction

- Use the trained model to predict the category of new, unseen images. The model outputs probabilities for each of the ten categories, and the category with the highest probability is selected as the predicted label.
- Image below shows a batch of 5 images with their true labels and their predicted labels. 3 out of the 5 are correct to give a rough 60% accuracy.




```
GroundTruth:  ship  dog   horse ship  frog
Predicted:  ship  cat   truck ship  frog
```

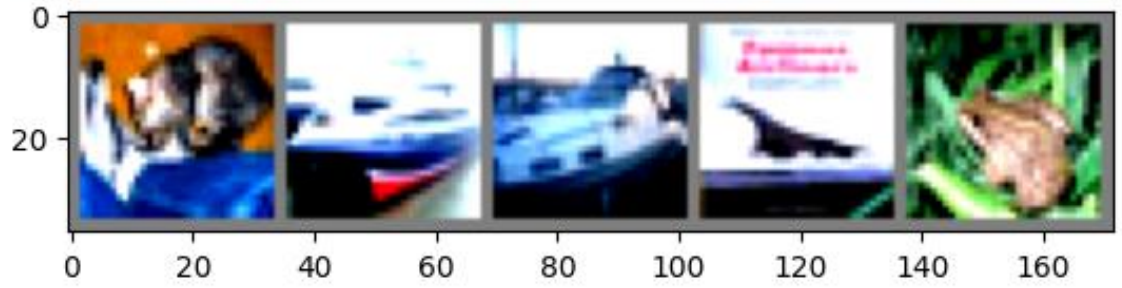
How to achieve perfection?

- In our initial run, we had an accuracy of 61.44% for our network. But how can we make it more efficient/accurate?
- Playing around with certain parameters will yield results, but it's not always the case of 'one size fits all'.
- Here are some examples;
  - Increase number of **epochs** (passes through network) – allow it to learn more from the extra passes. BUT we need to monitor for overfitting which impacts accuracy.
  - Create new training samples – greater diversity to help reduce overfitting.
  - **Adjust Learning Rate** (LR) – Higher **LR** can speed up training but could cause a suboptimal solution. Lower **LR** may require more epochs also. Finding the right balance is important.
  - Hyperparameter tuning – Techniques like **Bayesian optimisation** will allow to find the ideal learning rate, batch size, number of layers for efficiency.

Re-running the code

- In my notebook file, I re-ran the code from steps 3-5, and adjusted the learning rate from **0.001 to 0.00095**, and increased the **epochs from 10 to 20**.
- Our efficiency only increased by 0.03% to 61.47%. The number of correct guesses on 5 random images was 4 out of 5.

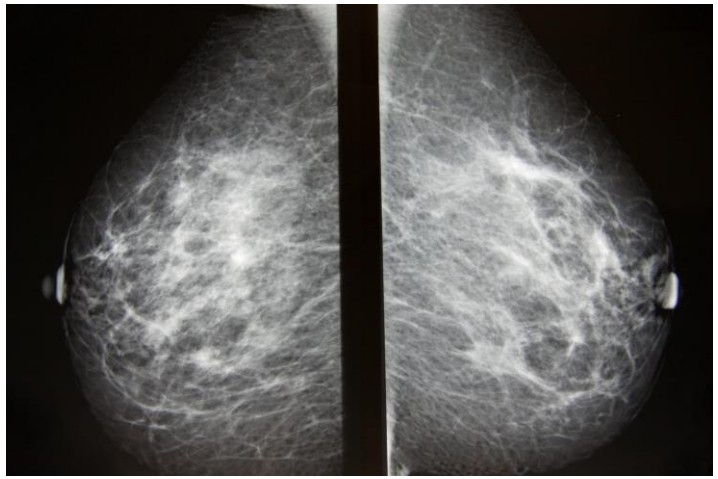
```
[Epoch 20, Batch 9800] loss: 0.707
[Epoch 20, Batch 10000] loss: 0.755
Finished Training
Accuracy of the network on the 10000 test images: 61.47%
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



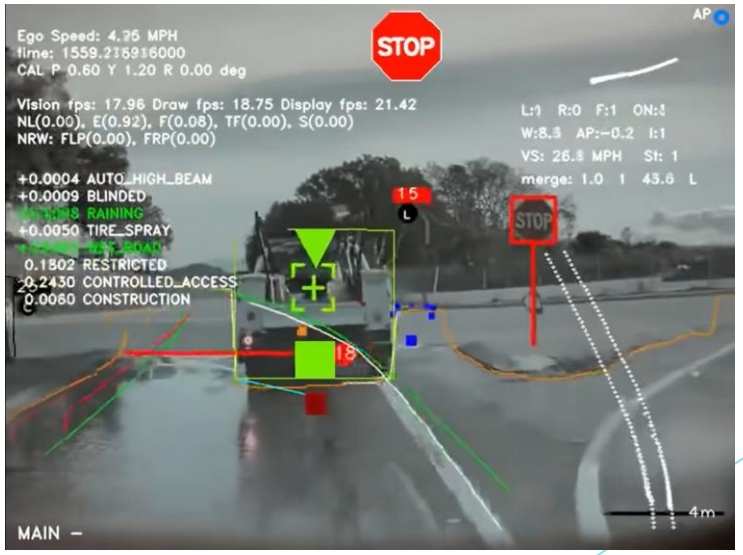

```
GroundTruth:  cat   ship  ship  plane frog
Predicted:  cat   ship  plane plane frog
```

Image classification and machine learning examples

- Breast Cancer Diagnosis – Nahid & Kong (2017), show the importance of image classification and machine learning in diagnosing breast cancer and increasing female mortality.



- Self-driving cars – Garg et al (2022) show how image classification is used in cars to assess obstacles (e.g. other cars, road signs). With explanation on the benefit to providing accessibility to users such as the blind/disabled.



Conclusion

- Image classification is a powerful tool in machine learning with diverse applications.
- By understanding the process of training a model, from data preprocessing to model evaluation, and leveraging CNNs, I know the knowledge to develop accurate and efficient image classification systems.
- It’s helped me to appreciate the complexities and potentials of image classification in various fields such as cancer diagnosis.

References

- Krizhevsky, A., Nair, V. and Hinton, G. (2009) CIFAR-10 (Canadian Institute for Advanced Research). Available at: http://www.cs.toronto.edu/~kriz/cifar.html (Accessed: 13 July 2024).
- Kubát, M. (2017) An introduction to machine learning /. Cham, Switzerland : Springer Nature,.
- Guo, T., Dong, J., Li, H. and Gao, Y., 2017. Simple convolutional neural network on image classification. In 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA) (pp. 721-724). IEEE.
- Nahid, A.A. and Kong, Y., 2017. Involvement of machine learning for breast cancer image classification: a survey. *Computational and mathematical methods in medicine*, *2017*(1), https://doi.org/10.1155/2017/3781951.
- Garg, N., Ashrith, K.S., Parveen, G.S., Sai, K.G., Chintamaneni, A. and Hasan, F., 2022. Self-driving car to drive autonomously using image processing and deep learning. *International Journal of Research in Engineering, Science and Management*, *5*(1), pp.125-132.