Deep Learning HW3

Baktash Ansari

99521082

Q1)

a)

- A high learning rate can cause the model's parameters to update too aggressively, leading to overshooting the optimal values and causing the optimization process to diverge.
- Rapid and large updates to the model's parameters can cause oscillations or bouncing around the optimal values, making it difficult for the optimizer to converge.
- A high learning rate may cause the model to focus too much on the training data, leading to poor generalization to new, unseen data.
- Extremely large updates can result in numerical instability, leading to NaN values in the loss function. (this problem made a huge effort to fix for myself in the CI course recent assignment
- A learning rate that is too high can cause the optimization process to oscillate or overshoot, preventing the model from converging to the optimal solution

To detect these problems:

- Use learning rate schedules or adaptive learning rate algorithms to dynamically adjust the learning rate during training
- Monitor the performance on a validation set to ensure that the model is not overfitting to the training data.
- Plotting training and validation loss curves over time can provide insights into the behavior of the optimization process.
- Check the norm of the gradients during training. Exploding gradients (very large gradient values) may indicate a high learning rate. (this option is so important in my opinion)

b)

- Training may progress very slowly, requiring a large number of epochs to reach an acceptable solution.
- A very low learning rate increases the risk of getting stuck in local minima,
- A low learning rate makes the optimization process sensitive to noisy gradients, potentially leading to slow or inefficient convergence.

- When encountering plateaus(زینی) or flat regions in the loss landscape, a very low learning rate may struggle to move the model parameters away from these regions.

For detection:

- Monitoring the training loss over time and observing whether it plateaus or decreases very slowly.
- If the model consistently converges to suboptimal solutions, it may indicate that the learning rate is too low.
- Monitoring the training loss for erratic behavior or lack of improvement over time.
- If the model appears to stall or make minimal progress during training, it could be an indication of difficulty in escaping plateaus.
- Plotting the training loss over time and inspecting the behavior of the optimization process can provide insights into whether the learning rate is too low. Look for signs of slow convergence.

c)

A saddle point in the graph of the loss function refers to a point where the gradient is zero but the point is not a minimum or maximum. At a saddle point, the surface of the loss function curves up in some dimensions and down in others, resembling the shape of a saddle. Saddle points can pose challenges for optimization algorithms because the gradient is zero, making it difficult for traditional gradient-based optimization methods to escape from them.

SGD:

- SGD is a simple optimization algorithm that is easy to implement and computationally efficient.
- It requires less memory compared to more complex optimization algorithms.
- SGD can have slow convergence, especially in the presence of saddle points or areas with a flat gradient, as it relies solely on the gradient information to update parameters.
- In shallow saddle points, where the gradient is zero or small, SGD may get stuck or converge very slowly due to its reliance on gradient information.

ADAM:

- Adam's adaptive learning rate can be advantageous near saddle points. It allows the algorithm to adjust the learning rate for each parameter individually, helping it navigate through flat regions more effectively.
- Adam incorporates momentum, which helps the optimizer accumulate information about the direction of the gradient, making it more resilient to being trapped in saddle points.
- Adam introduces additional computational complexity compared to SGD. This overhead may become a disadvantage in scenarios where computational resources are limited.

d)

In the context of optimizers and the movement towards the minimum of the function in different iterations, the larger the size of our batch in each iteration, the more accurately our loss function represents the original loss function, and our movement towards the minimum will be smoother. The smaller the batch, the less accurate the approximation of the function will be, and there will be more oscillation in the loss. Therefore, the left figure corresponds to a larger-sized batch or even a batch with the entire dataset, and the right figure is for a mini-batch.

Q2)

First I write the forward passing formulas:

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

$$O_{21} = X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22}$$

$$O_{22} = X_{22}F_{11} + X_{23}F_{12} + X_{32}F_{21} + X_{33}F_{22}$$

O11 = 6 + 0 -6 + 1 = 1
O12 = 8 + 0 - 3 - 3 = 2
O21 = 4 + 0 - 12 - 2 = -10
O22 = 2 + 0 + 6 + 0 = 8

1	2
-10	8

Now we use GAP for the output:
G = (O11 + O12 + O21 + O22) / 4 = (1 + 2 - 10 + 8) / 4 = 0.25

We know that dL/dG = 1

For backpropagation we should calculate:

dL/dX = ?
dL/dF = ?

For calculating dL/dX we have:

dL/dX = dL/dG * dG/dO * dO/dX
dL/dG = 1

dG/dO:
dG/dO11 = ¼
dG/dO12 = ¼
dG/dO21 = ¼
dG/dO22 = ¼

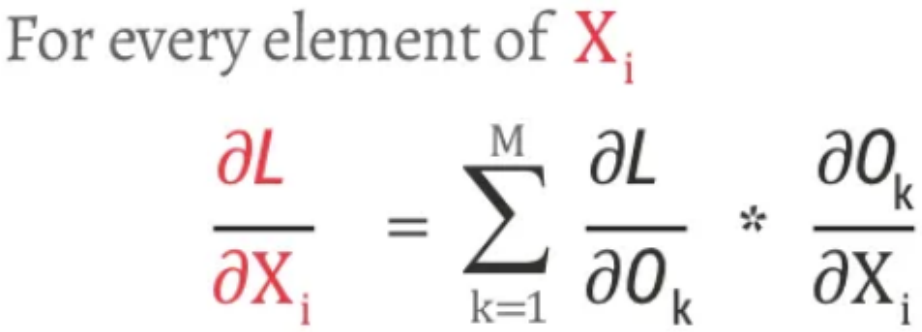
dO/dX:
We know that:
O11 = X11F11 + X12F12 + X21F21 + X22F22
O12 = …
O21 = …
O22 = …
So we have:
dO11/dX11 = F11 = 2
dO11/dX12 = F12 = 0
dO11/dX21 = F21 = -3
dO11/dX22 = F22 = 1

dO12/dX12 = F11 = 2
dO12/dX13 = F12 = 0
dO12/dX22 = F21 = -3
dO12/dX23 = F22 = 1

dO21/dX21 = F11 = 2
dO21/dX22 = F12 = 0
dO21/dX31 = F21 = -3
dO21/dX32 = F22 = 1

dO22/dX22 = F11 = 2
dO22/dX23 = F12 = 0
dO22/dX32 = F21 = -3
dO22/dX33 = F22 = 1

Now we should calculate dL/dX
We have: dL/dX = 1 * dG/dO * dO/dX
By this formula: (we put G instead of L in this formula)



We have:

dL/dX11 = ¼ * F11 = ½
dL/dX12 = ¼ * F11 + ¼ * F12 = ½ + 0 = ½
dL/dX13 = ¼ * F12 = 0
dL/dX21 = ¼ * (F21 + F11) = -¼
dL/dX22 = ¼ * (F22 + F21 + F12 + F11) = 0
dL/dX23 = ¼ * (F22 + F12) = ¼
dL/dX31 = ¼ * F21 = -¾
dL/dX32 = ¼ * (F22 + F21) = -½
dL/dX33 = ¼ * F22 = ¼

So dL/dX =

1/2	1/2	0
-1/4	0	1/4
-3/4	-1/2	1/4

For calculating dL/dF we have:

dL/dF = dL/dG * dG/dO * dO/dF
dL/dF = dG/dO * dO/dF

dO/dF:

We know that:
O11 = X11F11 + X12F12 + X21F21 + X22F22
O12 = …
O21 = …
O22 = …
So we have:
dO11/dF11 = X11 = 3
dO11/dF12 = X12 = 4
dO11/dF21 = X21 = 2
dO11/dF22 = X22 = 1
….
Same as above

By this formula: (put G instead of L for our case)

$$\text{For every element of } F$$

$$\frac{\partial L}{\partial F_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial F_i}$$

We have:
dL/dF11 = ¼ (X11 + X12 + X21 + X22) = 5/2
dL/dF12 = ¼ (X12 + X13 + X22 + X23) = 7/4
dL/dF21 = ¼ (X21 + X22 + X31 + X32) = 5/4
dL/dF22 = ¼ (X22 + X23 + X32 + X33) = -1

So we have:

dL/dF:

5/2	7/4
5/4	-1

Q3)

a)

Layer	Output shape	# of params
Input	(500, 7)	0
Conv1D	(498, 16)	(7*3+1)*16 = 352
MaxPool1D	(249, 16)	0
Conv1D	(245, 32)	(16*5+1)*32 = 2592
MaxPool1D	(122, 32)	0
Conv1D	(118, 64)	(32*5+1)*64 = 10304
MaxPool1D	(59, 64)	0
Flatten	(,3776)	0
Dense	(,128)	128 * (3776 + 1)
Dense	(,5)	(128 + 1) * 5

Total parameters = 352 + 2592 + 10304 + 483456 + 645 = 497349

b)

The main difference between Conv2D (2-dimensional convolution) and Conv3D (3-dimensional convolution) lies in the dimensionality of the data they operate on. Just like Conv1D is specialized for 1-dimensional data, Conv2D is for 2-dimensional data (like images), and Conv3D is for 3-dimensional data.

Conv2D:
Used for processing 2D grid data, such as images.
Convolves over two dimensions (e.g., height and width for images).

Conv3D:
Used for processing 3D grid data, such as video data (which is essentially a sequence of 2D frames) or volumetric data (e.g., medical imaging).
Convolves over three spatial dimensions (e.g., depth, height, and width).

Usage of Conv3D:
Video Analysis: In video data, each frame can be considered as a 2D grid, and Conv3D can learn spatiotemporal features across multiple frames.
Medical Imaging: In volumetric medical imaging data (e.g., CT scans), where the data has three spatial dimensions.
3D Object Recognition: For tasks where the spatial arrangement of features in three dimensions is crucial.

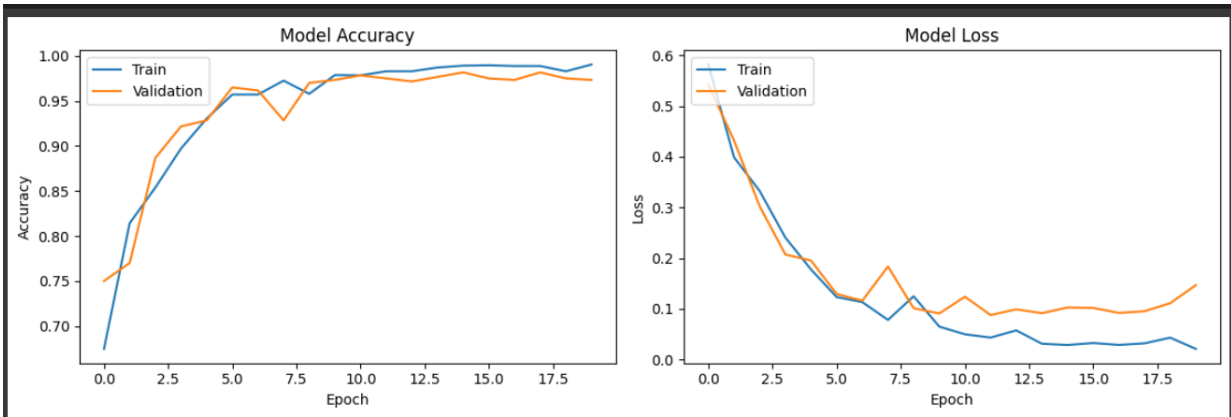
Q4)

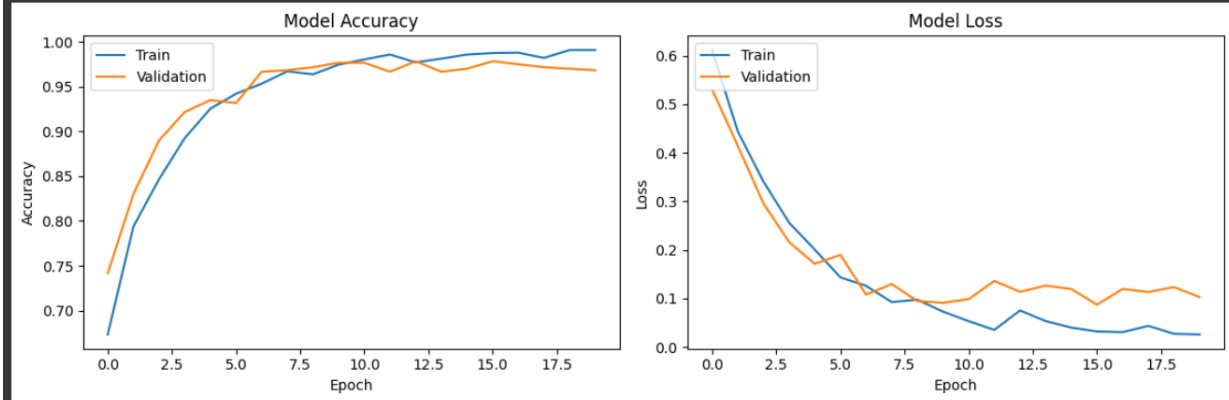
Everything is obvious in the notebook.
The model's structure:

```
seq_model = Sequential()
seq_model.add(Input(shape = (256,256, 1)))
seq_model.add(Rescaling(1./255))
seq_model.add(Conv2D(filters = 16, kernel_size = (5, 5), activation = 'relu'))
seq_model.add(MaxPool2D())
seq_model.add(Dropout(0.5))
seq_model.add(Conv2D(filters = 32, kernel_size = (5, 5), activation = 'relu'))
seq_model.add(MaxPool2D())
seq_model.add(Dropout(0.4))
seq_model.add(Conv2D(filters = 64, kernel_size = (3, 3), activation = 'relu'))
seq_model.add(MaxPool2D())
seq_model.add(Dropout(0.3))
seq_model.add(Conv2D(filters = 128, kernel_size = (3, 3), activation = 'relu'))
seq_model.add(MaxPool2D())
seq_model.add(Dropout(0.2))
seq_model.add(Flatten())
seq_model.add(Dense(64, activation = "relu"))
seq_model.add(Dropout(0.2))
seq_model.add(Dense(32, activation = "relu"))
seq_model.add(Dense(1, activation = "sigmoid"))
seq_model.summary()
```

Epoch = 20

```
Epoch 1/20
75/75 [==============================] - 7s 71ms/step - loss: 0.6115 - accuracy: 0.6733 - val_loss: 0.5304 - val_accuracy: 0.7417
Epoch 2/20
75/75 [==============================] - 6s 79ms/step - loss: 0.4441 - accuracy: 0.7937 - val_loss: 0.4143 - val_accuracy: 0.8300
Epoch 3/20
75/75 [==============================] - 5s 61ms/step - loss: 0.3408 - accuracy: 0.8467 - val_loss: 0.2957 - val_accuracy: 0.8900
Epoch 4/20
75/75 [==============================] - 6s 71ms/step - loss: 0.2560 - accuracy: 0.8925 - val_loss: 0.2161 - val_accuracy: 0.9217
Epoch 5/20
75/75 [==============================] - 7s 82ms/step - loss: 0.2006 - accuracy: 0.9254 - val_loss: 0.1712 - val_accuracy: 0.9350
Epoch 6/20
75/75 [==============================] - 5s 62ms/step - loss: 0.1434 - accuracy: 0.9421 - val_loss: 0.1896 - val_accuracy: 0.9317
Epoch 7/20
75/75 [==============================] - 6s 72ms/step - loss: 0.1259 - accuracy: 0.9533 - val_loss: 0.1079 - val_accuracy: 0.9667
Epoch 8/20
75/75 [==============================] - 5s 63ms/step - loss: 0.0925 - accuracy: 0.9671 - val_loss: 0.1296 - val_accuracy: 0.9683
Epoch 9/20
75/75 [==============================] - 6s 74ms/step - loss: 0.0974 - accuracy: 0.9638 - val_loss: 0.0947 - val_accuracy: 0.9717
Epoch 10/20
75/75 [==============================] - 6s 81ms/step - loss: 0.0728 - accuracy: 0.9746 - val_loss: 0.0910 - val_accuracy: 0.9767
Epoch 11/20
75/75 [==============================] - 5s 62ms/step - loss: 0.0532 - accuracy: 0.9804 - val_loss: 0.0986 - val_accuracy: 0.9767
Epoch 12/20
75/75 [==============================] - 6s 71ms/step - loss: 0.0351 - accuracy: 0.9858 - val_loss: 0.1360 - val_accuracy: 0.9667
Epoch 13/20
75/75 [==============================] - 5s 63ms/step - loss: 0.0752 - accuracy: 0.9771 - val_loss: 0.1136 - val_accuracy: 0.9783
Epoch 14/20
75/75 [==============================] - 6s 77ms/step - loss: 0.0533 - accuracy: 0.9812 - val_loss: 0.1264 - val_accuracy: 0.9667
Epoch 15/20
75/75 [==============================] - 5s 61ms/step - loss: 0.0399 - accuracy: 0.9858 - val_loss: 0.1194 - val_accuracy: 0.9700
Epoch 16/20
75/75 [==============================] - 5s 71ms/step - loss: 0.0319 - accuracy: 0.9875 - val_loss: 0.0871 - val_accuracy: 0.9783
Epoch 17/20
75/75 [==============================] - 5s 63ms/step - loss: 0.0305 - accuracy: 0.9879 - val_loss: 0.1196 - val_accuracy: 0.9750
Epoch 18/20
75/75 [==============================] - 5s 70ms/step - loss: 0.0434 - accuracy: 0.9821 - val_loss: 0.1132 - val_accuracy: 0.9717
Epoch 19/20
75/75 [==============================] - 5s 63ms/step - loss: 0.0272 - accuracy: 0.9908 - val_loss: 0.1233 - val_accuracy: 0.9700
Epoch 20/20
75/75 [==============================] - 6s 72ms/step - loss: 0.0258 - accuracy: 0.9908 - val_loss: 0.1026 - val_accuracy: 0.9683
```





Q5)

Convolutional layers are commonly used in neural networks for image classification tasks due to their ability to capture local patterns and spatial hierarchies in images. However, there are situations where convolutional layers may not be the most suitable choice or might introduce challenges. Let's explore when to use convolutional layers and when not to:

When to Use Convolutional Layers for Image Classification:

Spatial Hierarchies and Local Patterns: Convolutional layers are effective at capturing spatial hierarchies and local patterns in images. They can recognize features like edges, textures, and shapes, which are crucial for image classification.

Translation Invariance: Convolutional layers provide translation invariance, meaning they can recognize patterns regardless of their position in the image. This is beneficial for tasks where the exact location of features is not crucial.

Parameter Sharing: Convolutional layers use parameter sharing, reducing the number of parameters in the network compared to fully connected layers. This is especially important for large images, as it helps in managing computational complexity.

Reduced Sensitivity to Image Variations: Convolutional layers are less sensitive to small variations in input images, making them robust to changes in lighting, orientation, and other factors.

When Not to Use Convolutional Layers or Potential Challenges:

Non-Image Data: If the input data is not in the form of a grid (like images), convolutional layers may not be appropriate. For example, sequential data like time series or text data might benefit more from recurrent layers or attention mechanisms.

Limited Receptive Field: Convolutional layers may struggle with capturing global dependencies in very large images if the receptive field of the convolutional filters is limited. In such cases, larger receptive fields or alternative architectures might be necessary.

Computational Cost: For small datasets or simple tasks, using convolutional layers may introduce unnecessary computational overhead. In such cases, a simpler architecture without convolutional layers might be more efficient.

Overfitting: Convolutional layers can potentially lead to overfitting, especially if the dataset is small or lacks diversity. Regularization techniques, data augmentation, or the use of pre-trained models can help mitigate this issue.

Limited Data: If the dataset is insufficient, using deep convolutional architectures might lead to poor generalization. Transfer learning, where pre-trained models on larger datasets are fine-tuned, can be a helpful strategy in such scenarios.

Q6)

a)

The primary purpose of using 1x1 filters in convolutional layers is to perform dimensionality reduction. They are often referred to as pointwise convolution or network-in-network structures. By using 1x1 filters, you reduce the number of channels in the feature maps, effectively compressing the information.
Additionally, the 1x1 convolutional layer introduces non-linearity through the application of activation functions, allowing the network to learn more complex relationships within the data.

In 1X1 Convolution simply means the filter is of size 1X1. This 1X1 filter will convolve over the ENTIRE input image pixel by pixel.

For example with the input of 64X64X3, if we choose a 1X1 filter (which would be 1X1X3), then the output will have the same Height and Weight as the input but only one channel — 64X64X1

Now consider inputs with a large number of channels — 192 for example. If we want to reduce the depth and but keep the Height X Width of the feature maps (Receptive field) the same, then we can choose 1X1 filters (remember Number of filters = Output Channels) to achieve this effect. This effect of cross-channel down-sampling is called Dimensionality reduction.

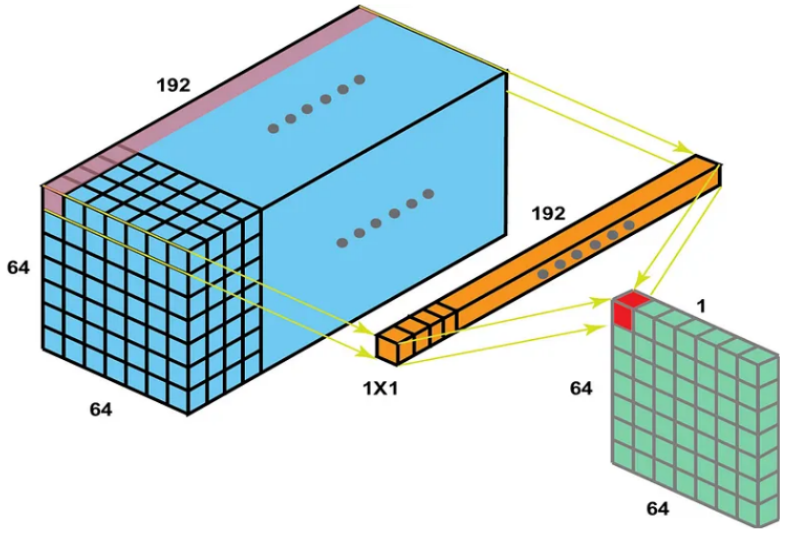


Image is adopted from this Link

One usecase of 1*1 filter is on GoogleNet for reducing costs.

b)

After applying 1x1 filters, you gain a transformed version of the input feature map with reduced dimensionality. Each pixel in the output corresponds to a weighted sum of the input pixels in its receptive field, and this process helps maintain important features while discarding less relevant ones.

c)

- **Inception Modules (GoogLeNet):** One notable model using 1x1 filters is GoogLeNet, which incorporates them in its inception modules to capture both local and global features efficiently.

- **ResNet Bottleneck Blocks:** Residual Networks (ResNets) often use 1x1 filters in the bottleneck blocks to reduce dimensionality before applying larger filters.

d)

- **In Small Networks**: In smaller networks or shallow architectures, the overhead of using 1x1 filters might not provide significant advantages, and the additional computational cost may outweigh the benefits.

- **Limited Channel Redundancy:** If the input data has very limited channel redundancy, meaning each channel contains highly unique information, the dimensionality reduction offered by 1x1 filters may not be as impactful.

e)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Input

# Assuming grayscale images with shape (height, width, channels)
input_shape = (28, 28, 64)

model = Sequential()
model.add(Input(shape=input_shape))
model.add(Conv2D(filters=32, kernel_size=(1, 1), activation='relu'))
# Add more layers as needed for your specific task

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Generate a random example input
import numpy as np
example_input = np.random.rand(1, *input_shape)
print("Input shape: ", example_input.shape)

# Get the model prediction
result = model.predict(example_input)
print("Output Shape", result.shape)
```

```
Input shape:  (1, 28, 28, 64)
1/1 [==============================] - 0s 188ms/step
Output Shape (1, 28, 28, 32)
```

As we can see the channel dimension reduce to the number of filters and the first and second dimension don't change.

Q7)

Code is provided in the zip file

Inception Module: The inception module is a type of convolutional layer that applies multiple different convolutional operations to the same input and then concatenates the results. This allows the network to learn different types of features at each layer. In this network, the inception module consists of four parallel paths:

- A 1x1 convolution.
- A 3x3 convolution following a 1x1 convolution.
- A 5x5 convolution following a 1x1 convolution.
- A 3x3 max pooling followed by a 1x1 convolution.

Each path uses different filter sizes (1x1, 3x3, and 5x5), allowing the network to capture spatial information at different scales. The 1x1 convolutions before the 3x3 and 5x5 convolutions are used to reduce the dimensionality of the input, which can help to reduce the computational complexity of the network.

Stride in Convolutional Layers: The stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as

we slide them around. This will produce smaller output volumes spatially. In general, we always use stride 1 in the first CONV layer of the network, and stride 2 in one or more of the CONV layers deeper in the network.

Convolutional Layers: The model starts with two convolutional layers with 32 and 64 filters respectively, each with a 3x3 kernel size. These layers are used to extract low-level features from the input images, such as edges and textures. After a max pooling layer, the inception module is applied to extract higher-level features at different scales. After the inception module, two more convolutional layers with 128 and 256 filters respectively are applied. These layers can extract even more complex features from the input. The use of multiple convolutional layers of increasing depth allows the network to learn a hierarchy of features, which can be beneficial for complex tasks like image classification. The final feature maps are flattened and passed to a dense layer for classification. The dense layer uses a softmax activation function to output a probability distribution over the 10 classes.

At last, the accuracy of the model on training becomes **95%**.

```
Epoch 1/10
782/782 [==============================] - 21s 21ms/step - loss: 1.4049 - accuracy: 0.4942 - val_loss: 1.0835 - val_accuracy: 0.6203
Epoch 2/10
782/782 [==============================] - 15s 20ms/step - loss: 0.9385 - accuracy: 0.6730 - val_loss: 0.8429 - val_accuracy: 0.7100
Epoch 3/10
782/782 [==============================] - 15s 19ms/step - loss: 0.7462 - accuracy: 0.7417 - val_loss: 0.7941 - val_accuracy: 0.7257
Epoch 4/10
782/782 [==============================] - 16s 20ms/step - loss: 0.6190 - accuracy: 0.7852 - val_loss: 0.7467 - val_accuracy: 0.7384
Epoch 5/10
782/782 [==============================] - 16s 20ms/step - loss: 0.5025 - accuracy: 0.8243 - val_loss: 0.7238 - val_accuracy: 0.7519
Epoch 6/10
782/782 [==============================] - 15s 20ms/step - loss: 0.3944 - accuracy: 0.8629 - val_loss: 0.7950 - val_accuracy: 0.7503
Epoch 7/10
782/782 [==============================] - 16s 21ms/step - loss: 0.3041 - accuracy: 0.8925 - val_loss: 0.8978 - val_accuracy: 0.7387
Epoch 8/10
782/782 [==============================] - 15s 20ms/step - loss: 0.2224 - accuracy: 0.9215 - val_loss: 1.0347 - val_accuracy: 0.7332
Epoch 9/10
782/782 [==============================] - 15s 20ms/step - loss: 0.1698 - accuracy: 0.9391 - val_loss: 1.2532 - val_accuracy: 0.7279
Epoch 10/10
782/782 [==============================] - 15s 19ms/step - loss: 0.1386 - accuracy: 0.9501 - val_loss: 1.2925 - val_accuracy: 0.7351
```