Week 8 - ConvNet
Machine Learning

David Esteso Calatrava

October 23, 2024

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

TensorFlow

School of Computer Scinece and Statics

Question 1: Convolution Implementation

(a) Implement a convolution function

This Python code defines a function to perform 2D convolutions:

```python
def convolve2d(input_matrix, kernel_matrix, padding='valid', stride=(1, 1)):
    kernel_height, kernel_width = kernel_matrix.shape

    # Apply padding if 'same' is specified
    if padding == 'same':
        padding_height = (kernel_height - 1) // 2
        padding_width = (kernel_width - 1) // 2
        padded_input_matrix = np.pad(input_matrix,
                                    ((padding_height, padding_height), (padding_width,
                                        padding_width)),
                                    mode='constant')
    elif padding == 'valid':
        padded_input_matrix = input_matrix # No padding for 'valid'
    else:
        raise ValueError("Invalid␣padding␣type.␣Use␣'same'␣or␣'valid'.")

    padded_input_height, padded_input_width = padded_input_matrix.shape

    stride_vertical, stride_horizontal = stride

    output_height = (padded_input_height - kernel_height) // stride_vertical + 1
    output_width = (padded_input_width - kernel_width) // stride_horizontal + 1

    output_matrix = np.zeros((output_height, output_width))

    # Perform convolution
    for output_row_index in range(output_height):
        for output_column_index in range(output_width):
            # Extract region of the input corresponding to the kernel
            row_start = output_row_index * stride_vertical
            row_end = row_start + kernel_height
            col_start = output_column_index * stride_horizontal
            col_end = col_start + kernel_width
            input_region = padded_input_matrix[row_start:row_end, col_start:col_end]

            # Convolve (element-wise multiplication and sum)
            output_matrix[output_row_index, output_column_index] = np.sum(input_region *
                kernel_matrix)

    return output_matrix
```

The function starts by obtaining the dimensions of the kernel and input matrix using `.shape`. If the padding type is `same`, zeros are added around the input matrix to maintain the output size equal to the input size. The padding is calculated as `(kernel size - 1) / 2`. If `valid` padding is used, no padding is added, and the convolution is performed directly on the input matrix. The stride defines how many pixels the kernel moves vertically and horizontally.

Mathematically, convolution involves sliding the kernel (a smaller matrix) over the input matrix, multiplying corresponding elements, and summing these products at each position. This value is stored in the output matrix. For each step, the kernel is shifted according to the stride, and the process repeats. The function calculates the size of the output matrix based on the input dimensions, kernel, padding, and stride, initializes the output matrix with zeros, and performs this element-wise multiplication and summation across the input matrix. This results in the final convolved matrix.

(b) Load and process an image with different kernels

This snippet defines the function that we are going to use:

```python
def process_image(image_path, kernel, id):
    im = Image.open(image_path)
    rgb = np.array(im.convert('RGB'))

    # Extract the red channel (R) from the RGB image
    r_channel = rgb[:, :, 0]

    # Save the red channel as a separate image
    if id == "kernel2":
        plt.imshow(r_channel, cmap='gray')
        plt.axis('off')

        plt.savefig(f'../output/RED_{id}.jpg', bbox_inches='tight', pad_inches=0)
        plt.close()

    # Apply the 2D convolution on the red channel
    output_array = convolve2d(r_channel, kernel)

    output_image = Image.fromarray(np.uint8(np.clip(output_array, 0, 255)))

    output_image.save(f'../output/RGB_{id}.jpg')
```

The `process_image` function begins by loading the image from the specified path and converting it into an RGB array using `PIL` to handle the image data. It then selects the red channel (`R`) from the RGB image for processing. Isolating a single channel allows for focused analysis on specific color information, which can be useful for certain image processing tasks, such as edge detection or feature extraction, where variations in a particular color channel are more pronounced. Once the red channel is isolated, the function applies a 2D convolution using the provided kernel. The convolution operation enhances certain features in the image, such as edges or textures, based on the kernel applied. The result is an output matrix which is clipped to ensure values stay within the valid range [0, 255] for image representation. Clipping is necessary because the convolution process can produce pixel values that exceed the typical range for image data, leading to visual artifacts if not constrained. Finally, the processed image is saved in the output folder with an identifier.

$$\texttt{kernel1} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad \texttt{kernel2} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

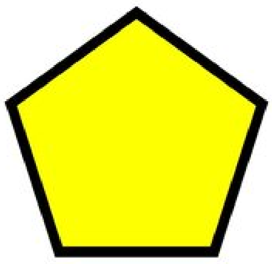
Figure 1: Yellow pentagon

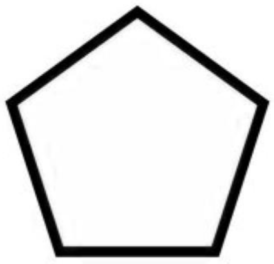
Figure 2: Red channel of Figure 1

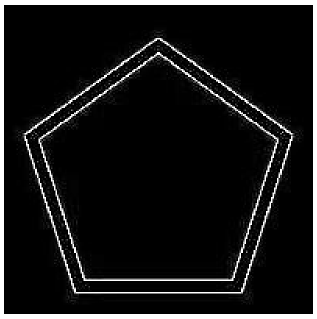
Figure 3: Result of applying kernel 1

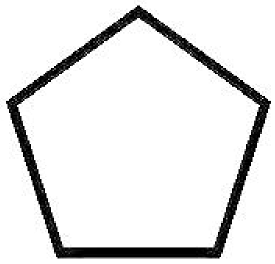
Figure 4: Result of applying kernel 2

When applying the two kernels to the image of a pentagon with a pronounced black border and a yellow interior, we observe distinct effects due to the specific properties of each kernel.

When applying `kernel 1` to a color n, we encounter two distinct situations. The most possible values for n are either 0 (black) or 255 (white) (Figure 2). In the first scenario, if a pixel has the same color n as its neighbors, the output is calculated as $8n - 8n = 0$, resulting in black in RGB. This explains why the interior and exterior of the pentagon, as well as the black border, appear black.

In the second situation, we consider a pixel of color n (which could be white, 255) that is surrounded by neighboring pixels of a different color, specifically black (0). If we examine a white pixel located on the edge, it has black neighbors inside the border. When these black neighbors are multiplied by -1, they contribute 0 in the convolution. As a result, the kernel effectively highlights these edge pixels in white, creating a striking contrast against their neighboring black pixels (Figure 3).

In contrast, `kernel 2` only considers four of its eight neighboring pixels, specifically those directly above, below, to the left, or to the right. As a result, black pixels will remain black, and white pixels will also remain white since there is no substantial subtraction that can cause them to turn black. This explains the minimal differences between the red channel image and the result of applying this kernel (Figure 4).

Question 2: Building a Convolutional Network

(a) ConvNet architecture

This is the architecture of the ConvNet used in the provided code:.

```
model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:], activation='relu'
    ))
```

The architecture begins with a Conv2D layer featuring 16 filters, designed to capture simple, low-level features like edges and textures. Utilizing fewer filters initially helps manage the model's complexity: the denser the layer—that is, the greater the number of neurons and their connections—the higher the total number of parameters that need to be learned. This reduction in initial complexity decreases the risk of overfitting, allowing the network to concentrate on the most basic patterns.

```
model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
```

The next layer remains a Conv2D with 16 filters and incorporates a stride of (2, 2) to diminish spatial resolution, enabling the network to focus on higher-level features. This configuration introduces translation invariance, meaning the model becomes less sensitive to the exact positioning of an object within an image.

```
model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
```

In the third layer, the number of filters is increased to 32. As the network deepens, it learns more complex and abstract features, necessitating more filters due to the increased complexity of the patterns being learned. The reduced resolution from the previous layer permits the use of more filters without significantly raising the computational load.

```
model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
```

Similarly, the fourth layer, which also operates as a convolutional layer for high-level features, maintains 32 filters and applies a stride of (2, 2). This further reduces the size of the feature map, enabling the model to cluster similar features while minimizing noise. We have always used a 3×3 kernel.

```
model.add(Dropout(0.5))
```

The fifth layer incorporates a Dropout layer, which is crucial for preventing overfitting. By randomly deactivating 50% of the neurons during training, the model encourages better generalization and avoids becoming overly reliant on specific neurons, thereby preventing the learning of overly specific patterns that may not generalize well to unseen data.

```
model.add(Flatten())
```

The sixth layer is a Flatten layer that transforms the multidimensional feature map into a one-dimensional vector, as the output needs to be presented to the final dense layer in a flat format.

```
model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1
    (0.0001)))
```

Finally, the seventh layer is a dense layer with a softmax activation function, well-suited for multiclass classification tasks involving 10 classes. Therefore, we have 10 output channels, as the number of classes is set to 10. This layer aggregates the deep features learned in the previous layers to make the final prediction. Additionally, L1 regularization is applied to penalize larger weights, promoting a simpler and more interpretable model that enhances generalization.

In other words, every decision about designing this ConvNet is based on some deep learning theory and neural network architecture principles. Gradual decrease in resolution, an increase in the number of filters in deeper layers, applying of Dropout to avoid overfitting, and final structuring by means of Flatten and Dense layers, all were aimed at enhancing the ability of a network to learn patterns relevant to a task at hand, with minimal overfitting and optimal computational efficiency.

(b) Model training results

(i) Parameters and performance

The Keras model reports a total of 37,146 parameters. Among these, the layer with the most parameters is the dense layer, which contains 20,490 parameters. This significant number is primarily due to the fully connected nature of dense layers, where each neuron is connected to every neuron in the previous layer, leading to a larger parameter count. Regarding model performance (Table 1), the test accuracy is approximately 50%, while the training accuracy is around 64%. This indicates a discrepancy between the two metrics, suggesting that the model performs better on the training data compared to the test data. Such a difference may imply that the model is overfitting to the training dataset, learning specific patterns rather than generalizable features. It is noteworthy that the average F1 score across all classes is approximately 0.5. When comparing this model to the baseline predictor, which has an accuracy of only around 10%, it's clear that the Keras model far exceeds this performance. Given that the task involves differentiating between 10 classes, a baseline predictor will generally perform poorly as the number of classes increases. Thus, the larger the number of classes, the greater the disparity between a competent model and the baseline predictor. We have also trained a logistic regression model using the same number of samples and achieved an accuracy of approximately 25%. While a 50% accuracy might not seem impressive at first glance, it is important to note that other linear models do not come close to this level of precision. This indicates that our logistic ConvNet is performing relatively well compared to its peers in the linear modeling space.

(ii) Diagnostics from training history

In the previous section, we noted that the training accuracy was around 64%, while the test accuracy stood at approximately 50%. This disparity raises concerns about the model's ability to generalize to unseen data. When we refer to the accuracy ans loss graphs, we are analyzing the model's performance over epochs. An epoch represents one complete iteration through the training dataset. During each epoch, the model adjusts its parameters based on the errors it made, allowing it to improve its prediction performance. By plotting accuracy and loss against epochs, we can observe how the model adapts and learns from the data over time. Upon reviewing the accuracy plot (Figure 5), there seems to be a slight indication of overfitting, as the training accuracy continues to rise while the validation accuracy plateaus. This discrepancy suggests that the model is learning the training data well but may struggle to generalize effectively. Additionally, looking at the loss plot reinforces this suspicion: while the training loss decreases steadily, the validation loss begins to flatten, indicating that the model may be fitting the training data too closely. While these observations don't definitely prove overfitting, they indicate that we should carefully consider the model's complexity and explore regularization techniques, such as increasing the amount of data or trying different regularization values, to enhance its ability to generalize.

(iii) Effect of increasing training data

Here we examine how increasing the training data size affects accuracy and training time.

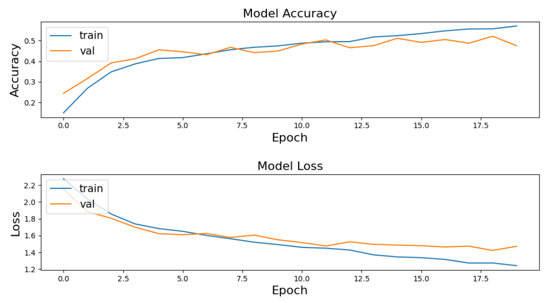

Figure 5: 5k samples model

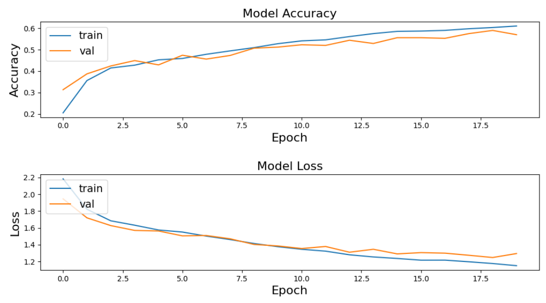

Figure 6: 10k samples model

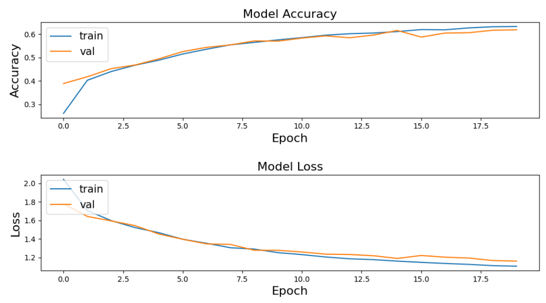

Figure 7: 20k samples model

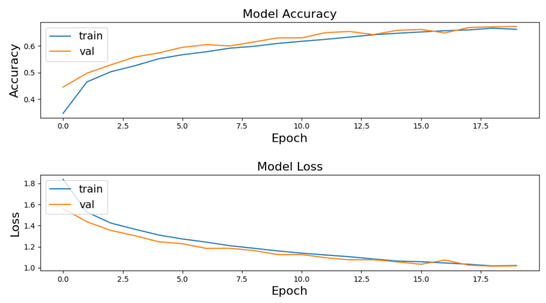

Figure 8: 40k samples model

As the amount of training data increases, the time taken to train the network shows a clear upward trend. For instance, with around 5,000 samples, the training time was roughly 7.5 seconds, which escalated to about 53 seconds for approximately 40,000 samples. This increase in training time is expected since larger datasets require more computations to update the model weights, reflecting the additional complexity introduced by handling more data.

Samples	Exec. Time (s)	Loss (Train)	Acc. (Train)	Loss (Test)	Acc. (Test)
5,000	7.51	1.0918	64.90%	1.4518	50.58%
10,000	13.33	1.0362	66.64%	1.2969	56.17%
20,000	25.15	0.9933	68.33%	1.1479	61.83%
40,000	52.83	0.8867	71.96%	1.0166	66.84%

Table 1: Experiment Results with Different Amounts of Data

When examining prediction accuracy on both training and test datasets, a notable improvement is observed with the increase in training samples. Training accuracy increased from about 65% with 5,000 samples to around 72% with 40,000 samples, indicating that the model learns more effectively with more data. Similarly, test accuracy also improved, starting at roughly 51% and reaching around 67%. Furthermore, the gap between test and training accuracies narrowed significantly, reducing from around 14% with 5,000 samples to about 5% with 40,000 samples. This trend suggests that a larger training set enhances the model's ability to generalize to unseen data, reducing the risk of overfitting.

In terms of loss, training loss improved from approximately 1.09 to 0.89 as the data increased from 5,000 to 40,000 samples, indicating that the model not only fits the training data better but also minimizes errors. Likewise, test loss decreased from around 1.45 to about 1.02, showing a substantial reduction that further confirms the model's enhanced performance with more data. This significant decrease in loss aligns with our earlier discussion about how increasing data helps reduce overfitting, reinforcing the benefits of larger training datasets.

The analysis of the plots of the history variable for each run (Figures 5 - 8) indicates a tremendous trend, with an increase in the amount of training data, both test accuracy and training accuracy curves lie close to each other, reflecting much consistency between their results. Another good reassurance is that the training and test loss values go down together. By this alignment, one could understand that the model indeed benefits with a larger training set for its generalization to new, unseen data, thereby reducing overfitting. On the whole, we may safely conclude here that the larger the dataset, the smaller the difference will be between train and validation both in terms of accuracy and loss.

(iv) Effect of L1 regularization

Now, we will explore the effect of varying the L1 regularization weight and its influence on training/test accuracy, comparing with the effect of increasing training data.

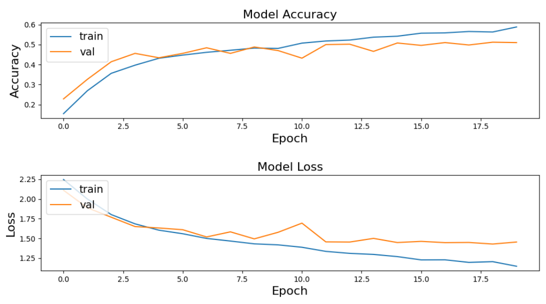

Figure 9: 5k model with weight 0

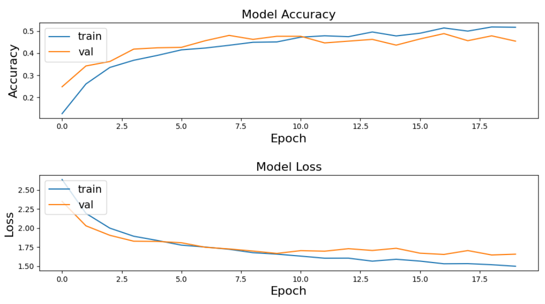

Figure 10: 5k model with weight 0.001

We notice that, as the weight parameter is varied, there is an interesting trend in the performance metrics. In particular, we find that loss has a tendency to flatten out with increasing values of the weight parameter (Figures 9 - 12); this would indicate a version of the model

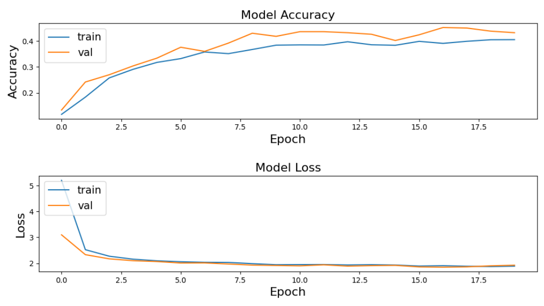

Figure 11: 5k model with weight 0.01

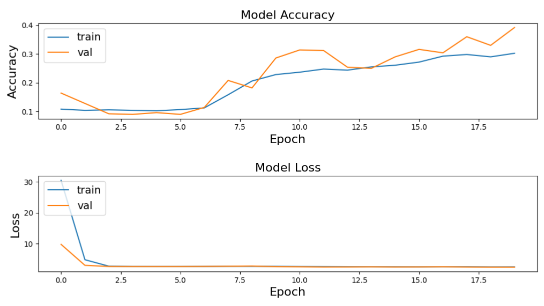

Figure 12: 5k model with weight 0.1

Regularization	Exec. Time (s)	Loss (Train)	Acc. (Train)	Loss (Test)	Acc. (Test)
0.0	7.44	1.0607	63.70%	1.4133	49.39%
0.0001	7.45	1.2324	59.64%	1.5257	47.72%
0.001	7.69	1.3966	58.10%	1.5816	49.84%
0.01	7.40	1.7122	47.24%	1.7964	43.10%
0.1	7.45	2.3529	33.86%	2.3841	32.52%

Table 2: Performance Metrics with Different Regularization Parameters

which is insensitive to training examples. But again, the less the value of the weight parameter is, the lower the loss values are; hence, the regularization is weaker and the model fits better with the training data.

On the other hand, increasing the value of the weight parameter deteriorates the accuracy curves. We observe that precision does not continue to rise and, instead results in poorer metrics of performance. That hints that higher values of the weight parameter may result in underfitting where a model is too rigid to capture the underlying pattern in the training data.

These observations therefore hint at the fact that the weight parameter, in general, strikes a delicate balance between how much complexity can be allowed within the model and just enough to allow an improvement in the accuracy and a reduction in loss. Too high a value of this may prevent the model from learning well from the data.

Comparison of Strategies to Combat Overfitting

First, let's review the conclusions drawn from our analyses. We have observed that increasing the amount of training data yields beneficial results, despite the trade-off of longer training times. Conversely, finding the optimal weight parameter involves a search for the most effective value. While we identified an optimal value for the weight parameter, increasing it beyond this point proved inefficient in terms of results. This indicates that we are comparing two distinct practices that are not mutually exclusive. Moreover, tuning the weight parameter does not contribute to longer execution times. So, which approach is better for combating overfitting?

Our analysis shows that, apart from execution time, increasing the amount of training data consistently leads to better outcomes. This is evident in the diminishing difference in accuracy between the test and validation sets as we increased the training data. However, selecting the correct weight parameter remains critical; performance declines when the magnitude of the weight parameter exceeds the optimal choice.

It's important to highlight that having a sufficient amount of training data is essential, even with a perfectly tuned weight parameter. As demonstrated with the dataset of 5,000 samples, the best accuracy we achieved was only around 50%. This underscores the point that having more data is crucial for a model to be truly useful. In fact, one of the main objectives of large companies today is to gather millions of data points to train their models effectively.

In the scenario of choosing between these strategies, it's worth noting that a model trained on 1,000 data points—even with an optimal weight parameter—would not outperform a model trained on 50,000 data points with a suboptimal weight parameter. This further illustrates the importance of having a substantial amount of data for effective model training.

In conclusion, this comparison does not lead to a definitive choice between one strategy over the other, as they serve as complementary approaches. Whenever you have enough data and a sufficiently powerful machine, the evidence suggests that you should leverage the maximum amount of data possible. While increasing training data generally improves performance and mitigates overfitting, fine-tuning the weight parameter is also essential to achieving the best results and maximize the potential of that data. But if you have a dataset large enough such that the training time does not extend just to seconds or minutes, but more extreme measures of duration, then obviously experimenting with different regularization parameters can multiply this training time. Therefore, while this analysis remains meaningful, it points out an important consideration of trade-offs among data size and computational feasibility in choosing regularization strategies. A balanced approach that incorporates both strategies may yield the most effective outcomes in practice.

Question 3: Modifying the ConvNet

(c)(i)(ii) Max-pooling implementation & performance comparison

To incorporate MaxPooling, we remove the stride option from the second convolutional layer and add the MaxPooling layer in between the convolutional layers where the feature density changes:

```
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
```

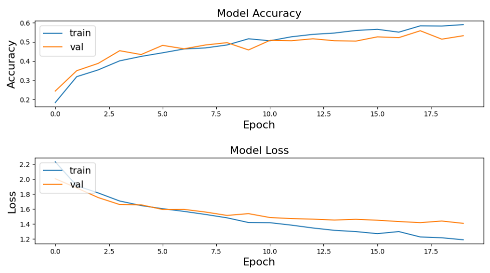

Table 3: 5k samples model Max-pooling

Metric	Value
Execution Time (s)	13.04
Loss (Train)	1.0509
Accuracy (Train)	66.02%
Loss (Test)	1.3731
Accuracy (Test)	53.85%

Table 4: Model Performance with Max-pooling

After running the ConvNet, Keras reports that the number of parameters has remained the same as in the original network. For 5,000 training data points, the prediction accuracy on both the training and test data has improved, though not significantly. The average F1 score for the max pooling model is about 0.52, while the previous model has an F1 score of around 0.50. This shows that the performance differences between the two models are not significant.

The number of parameters has stayed the same because the architecture of the network, specifically the layers and the size of the filters, has not changed. The stride and pooling operations only modify how the convolutional layers process the input by reducing the spatial dimensions of the feature maps. These operations don't alter the number of trainable parameters (such as weights and biases) within the convolutional filters themselves.

What stands out the most is the significant increase in time, which was expected. Both methods aim to achieve a similar goal: downsampling. The difference lies in how they do it. Stride directly impacts the number of convolutions, so using a stride reduces the number of convolutions performed. Pooling, on the other hand, happens after the convolutions, so it doesn't affect the number of convolutions.

Let's consider an input image of size 8×8 and a convolution filter of size 3×3. Without stride (stride $= 1$), the output size will be 6×6, resulting in $6 \times 6 = 36$ convolutions. Using a stride of 2, the output size becomes 3×3, with a total of $3 \times 3 = 9$ convolutions. As you can see, stride 2 reduces the number of convolutions from 36 to 9, a factor of 4 reduction.

If we apply max pooling with a 2×2 filter after the convolutions (without stride), the output size is reduced to 3×3, but the total number of convolutions remains 36, as pooling does not affect the number of convolutions, only the output size.

In summary, stride directly reduces the number of convolutions, while pooling reduces the output size after convolutions without changing their number.

Question 4: Optional - Exploring a Deeper Network

(d) Performance of a thinner, deeper ConvNet

To make the network thinner and deeper, we have added this code at the beginning of it:

```
model.add(Conv2D(8, (3, 3), padding='same', input_shape=input_shape, activation='relu'))
model.add(Conv2D(8, (3, 3), strides=(2, 2), padding='same', activation='relu'))
```

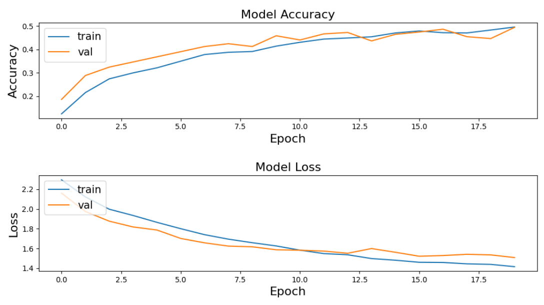

Figure 13: 5k samples deeper model

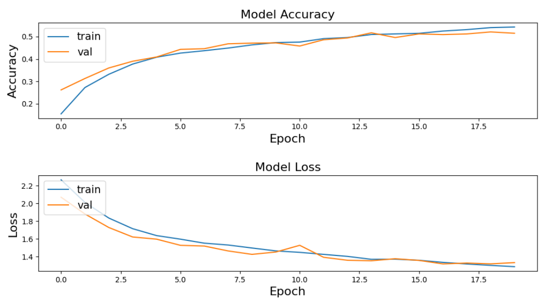

Figure 14: 10k samples deeper model

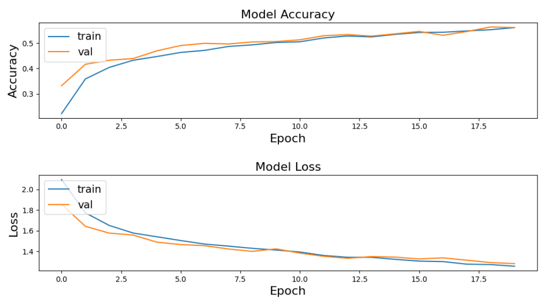

Figure 15: 20k samples deeper model

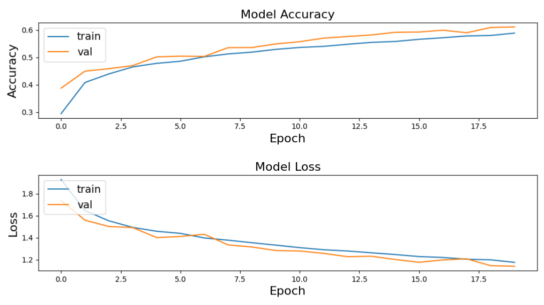

Figure 16: 40k samples deeper model

The comparison between the deeper, finer network and the less deep network reveals distinct differences in their performance. The deeper network generally exhibits a faster execution

Samples	**Exec. Time (s)**	**Loss (Train)**	**Acc. (Train)**	**Loss (Test)**	**Acc. (Test)**
5,000	7.56	1.3725	52.74%	1.5119	46.74%
10,000	12.39	1.2365	56.97%	1.3681	51.33%
20,000	22.12	1.1086	61.44%	1.2266	56.94%
40,000	41.17	0.9831	67.31%	1.0786	62.77%

Table 5: Experiment Results on a deeper and thinner model

time, particularly as the dataset size increases, indicating a more efficient training process. In contrast, the less deep network, while slower, consistently achieves higher accuracy and lower loss on both the training and test sets across all sample sizes. The shallow model has an F1 score around 0.62, while the deep model achieves an F1 score around 0.68. Although the difference is not extreme, it is noteworthy and suggests that increasing the model's depth may lead to some improvement in performance. At the level of overfitting, both networks perform similarly, with comparable differences between training and test accuracy. This indicates that neither network demonstrates a clear advantage over the other in terms of overfitting the training data.

The decrease in execution time can be attributed to the addition of a less dense layer at the front of the neural network, which reduced the total number of parameters from 37,146 to 23,314. In a neural network, each neuron connects to many others, and each connection has a weight that the model learns during training. Fewer weights in the less dense layer mean fewer calculations during the forward pass, where input data is processed. Similarly, in backpropagation, the model adjusts weights based on prediction errors, and with fewer weights to update, this step also takes less time.

Additionally, this slight reduction in perfomance is maybe due to the well-known problem of vanishing and exploding gradients during the process of backpropagation. In deep models, the gradients responsible for the update of the weights could get very small-something known as the vanishing gradient-or become too large, so-called exploding gradient. While vanishing gradients make it difficult to train early layers, exploding gradients might result in uncontrolled updates of the weights at every step of training, hence causing divergence.

Intuitively, one may realize that to some extent, the use of ReLU can alleviate this problem, since the gradients can causally flow more easily along the network, hence improving the learning of the earlier layers. However, ReLU itself does not naturally avoid the problem of exploding gradients. Note that in cases where the ReLU outputs some big positive value, its contribution to large gradients is indeed big during backpropagation through deeper layers. This further increases the problem of exploding gradients that brings instability during training. Therefore, even though the ReLU activation function may improve gradient propagation, observing both vanishing and exploding gradients during training in deep neural networks becomes very important. However, we cannot definitively state that either of these phenomena has occurred without conducting a more thorough investigation.

In conclusion, although it may seem that the new network has not improved upon the previous one, we must consider that with 40,000 samples, we trained a model that took 20% less time while producing results that are only 6% worse. This trade-off suggests that neither network is clearly superior. If faster training is a priority, the new network may be the better choice. However, if you can sacrifice some speed for a slight improvement in accuracy, the previous network remains a solid option.

Appendix: Code

main.py

```
import part1 as p1
import pandas as pd
import numpy as np
import part2 as p2


if __name__ == "__main__":

    kernel1 = np.array([
        [-1, -1, -1],
        [-1,  8, -1],
        [-1, -1, -1]
    ])

    kernel2 = np.array([
        [ 0, -1, 0],
        [-1,  8, -1],
        [ 0, -1, 0]
    ])

    p1.process_image('../data/shape.jpg', kernel1, "kernel1")
    p1.process_image('../data/shape.jpg', kernel2, "kernel2")

    # Train model with 5, 10, 20 and 40K samples
    p2.train_and_save_model("a_5k", n_samples=5000)
    p2.train_and_save_model("a_10k", n_samples=10000)
    p2.train_and_save_model("a_20k", n_samples=20000)
    p2.train_and_save_model("a_40k", n_samples=40000)

    # Train a model with pooling
    p2.train_and_save_model("b_pooling", n_samples=5000, pooling=True)

    # Train model with 5k samples and weight 0, 0.0001 ... 0.1
    p2.train_and_save_model("c_weight_0", n_samples=5000, weight=0.0)
    p2.train_and_save_model("c_weight_00001", n_samples=5000, weight=0.0001)
    p2.train_and_save_model("c_weight_001", n_samples=5000, weight=0.001)
    p2.train_and_save_model("c_weight_01", n_samples=5000, weight=0.01)
    p2.train_and_save_model("c_weight_1", n_samples=5000, weight=0.1)

    # Train a deeper model with 5, 10, 20 and 40K samples
    p2.train_and_save_model("d_5k", n_samples=5000, deeper=True)
    p2.train_and_save_model("d_10k", n_samples=10000, deeper=True)
    p2.train_and_save_model("d_20k", n_samples=20000, deeper=True)
    p2.train_and_save_model("d_40k", n_samples=40000, deeper=True)

    p2.train_logistic_regression_with_cv()
```

part1.py

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def convolve2d(input_matrix, kernel_matrix, padding='valid', stride=(1, 1)):
   # Get kernel dimensions
   kernel_height, kernel_width = kernel_matrix.shape

   # Apply padding if 'same' is specified
   if padding == 'same':
       padding_height = (kernel_height - 1) // 2
       padding_width = (kernel_width - 1) // 2
       padded_input_matrix = np.pad(input_matrix,
                                    ((padding_height, padding_height), (padding_width,
                                        padding_width)),
                                    mode='constant')
   elif padding == 'valid':
       padded_input_matrix = input_matrix # No padding for 'valid'
   else:
       raise ValueError("Invalid␣padding␣type.␣Use␣'same'␣or␣'valid'.")

   # Get padded input matrix dimensions
   padded_input_height, padded_input_width = padded_input_matrix.shape

   # Get stride values
   stride_vertical, stride_horizontal = stride

   # Calculate output matrix dimensions
   output_height = (padded_input_height - kernel_height) // stride_vertical + 1
   output_width = (padded_input_width - kernel_width) // stride_horizontal + 1

   # Initialize output matrix
   output_matrix = np.zeros((output_height, output_width))

   # Perform convolution
   for output_row_index in range(output_height):
       for output_column_index in range(output_width):
           # Extract region of the input corresponding to the kernel
           row_start = output_row_index * stride_vertical
           row_end = row_start + kernel_height
           col_start = output_column_index * stride_horizontal
           col_end = col_start + kernel_width
           input_region = padded_input_matrix[row_start:row_end, col_start:col_end]

           # Convolve (element-wise multiplication and sum)
           output_matrix[output_row_index, output_column_index] = np.sum(input_region *
               kernel_matrix)

   return output_matrix

def process_image(image_path, kernel, id):
   # Load the image and convert it to an RGB array
   im = Image.open(image_path)
   rgb = np.array(im.convert('RGB'))

   # Extract the red channel (R) from the RGB image
   r_channel = rgb[:, :, 0]

   # Save the red channel as a separate image
   if id == "kernel2":
       plt.imshow(r_channel, cmap='gray')
       plt.axis('off')

       plt.savefig(f'../output/RED_{id}.jpg', bbox_inches='tight', pad_inches=0)
       plt.close()
   # Apply the 2D convolution on the red channel
   output_array = convolve2d(r_channel, kernel)

   # Clip values to the range [0, 255] and convert the result back to an image
   output_image = Image.fromarray(np.uint8(np.clip(output_array, 0, 255)))

   # Save the processed image to the output folder with an identifier
   output_image.save(f'../output/RGB_{id}.jpg')
```

part2.py

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Flatten, Conv2D
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import os
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
import time
from sklearn.preprocessing import StandardScaler

def define_model(input_shape, num_classes, deeper=False, pooling=False, weight=0.0001):
    model = keras.Sequential()

    if deeper:
        model.add(Conv2D(8, (3, 3), padding='same', input_shape=input_shape, activation='
            relu'))
        model.add(Conv2D(8, (3, 3), strides=(2, 2), padding='same', activation='relu'))
        model.add(Conv2D(16, (3, 3), padding='same', activation='relu'))
        model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
        model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
        model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    else:
        model.add(Conv2D(16, (3, 3), padding='same', input_shape=input_shape, activation='
            relu'))
        if pooling:
            model.add(Conv2D(16, (3, 3), padding='same', activation='relu'))
            model.add(layers.MaxPooling2D(pool_size=(2, 2))) # Capa de max-pooling
        else:
            model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))

        model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
        if pooling:
            model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
            model.add(layers.MaxPooling2D(pool_size=(2, 2)))
        else:
            model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))

    # Add dropout, flatten and dense layer
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(
        weight)))

    return model
```

```python
def train_and_save_model(id, num_classes=10, input_shape=(32, 32, 3), n_samples=5000,
    batch_size=128, epochs=20, weight=0.0001, pooling=False, deeper=False):
    output_dir = f"../output/{id}"
    os.makedirs(output_dir, exist_ok=True)

    # Load data
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    x_train = x_train[:n_samples]
    y_train = y_train[:n_samples]

    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    print("Original␣x_train␣shape:", x_train.shape)

    # Convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    # Define the model
    model = define_model(input_shape, num_classes, deeper=deeper, pooling=pooling, weight=
        weight)

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

    # Save the model summary
    with open(os.path.join(output_dir, "model_summary.txt"), "w", encoding='utf-8') as f:
        model.summary(print_fn=lambda x: f.write(x + '\n'))

    # Measure the execution time
    start_time = time.time()

    # Train the model
    history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
        validation_split=0.1)
    model.save(os.path.join(output_dir, "cifar_model.h5"))

    # Stop time after training
    execution_time = time.time() - start_time

    # Plot accuracy and loss
    plt.figure(figsize=(12, 6))

    # Plot accuracy
    plt.subplot(211)
    plt.plot(history.history['accuracy'], label='train')
    plt.plot(history.history['val_accuracy'], label='val')
    plt.title('Model␣Accuracy', fontsize=16)
    plt.ylabel('Accuracy', fontsize=16)
    plt.xlabel('Epoch', fontsize=16)
    plt.legend(loc='upper␣left', fontsize=14)

    # Plot loss
    plt.subplot(212)
    plt.plot(history.history['loss'], label='train')
    plt.plot(history.history['val_loss'], label='val')
    plt.title('Model␣Loss', fontsize=16)
    plt.ylabel('Loss', fontsize=16)
    plt.xlabel('Epoch', fontsize=16)
    plt.legend(loc='upper␣left', fontsize=14)

    # Adjust spacing between the plots
    plt.subplots_adjust(hspace=0.6)

    plt.savefig(os.path.join(output_dir, "model_performance.png"))
    plt.close()
    # Predictions and metrics
    evaluate_model(model, x_train, y_train, x_test, y_test, output_dir, execution_time)


def evaluate_model(model, x_train, y_train, x_test, y_test, output_dir, execution_time):
    # Predictions and metrics for training data
    preds = model.predict(x_train)
    y_pred = np.argmax(preds, axis=1)
    y_train1 = np.argmax(y_train, axis=1)

    # Save classification report and confusion matrix for training data
    with open(os.path.join(output_dir, "classification_report_train.txt"), "w", encoding='
        utf-8') as f:
        f.write(classification_report(y_train1, y_pred))

    cm_train = confusion_matrix(y_train1, y_pred)
    cm_train_rounded = np.round(cm_train, 3)

    with open(os.path.join(output_dir, "confusion_matrix_train.txt"), "w", encoding='utf-8')
         as f:
        np.savetxt(f, cm_train_rounded, fmt='%.3f', delimiter=',', header="Predicted␣Class
            ,0,1,2,3,4,5,6,7,8,9", comments="")

    # Predictions and metrics for test data
    preds = model.predict(x_test)
    y_pred = np.argmax(preds, axis=1)
    y_test1 = np.argmax(y_test, axis=1)

    # Save classification report and confusion matrix for test data
    with open(os.path.join(output_dir, "classification_report_test.txt"), "w", encoding='utf
        -8') as f:
        f.write(classification_report(y_test1, y_pred))

    cm_test = confusion_matrix(y_test1, y_pred)
    cm_test_rounded = np.round(cm_test, 3)

    with open(os.path.join(output_dir, "confusion_matrix_test.txt"), "w", encoding='utf-8')
        as f:
        np.savetxt(f, cm_test_rounded, fmt='%.3f', delimiter=',', header="Predicted␣Class
            ,0,1,2,3,4,5,6,7,8,9", comments="")

    # Analyze model parameters
    total_params = model.count_params()
    layer_params = {layer.name: layer.count_params() for layer in model.layers}
    max_layer = max(layer_params, key=layer_params.get)
    max_params = layer_params[max_layer]
    test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
    train_loss, train_accuracy = model.evaluate(x_train, y_train, verbose=0)

    common_label = np.argmax(np.bincount(y_test1))
    baseline_accuracy = np.sum(y_test1 == common_label) / len(y_test1)
    results = (
        f"Total␣parameters␣in␣the␣model:␣{total_params}\n"
        f"Layer␣with␣most␣parameters:␣{max_layer}␣({max_params}␣parameters)\n"
        f"Test␣accuracy:␣{test_accuracy␣*␣100:.2f}%\n"
        f"Test␣loss:␣{test_loss:.4f}\n"
        f"Train␣accuracy:␣{train_accuracy␣*␣100:.2f}%\n"
        f"Train␣loss:␣{train_loss:.4f}\n"
        f"Execution␣time:␣{execution_time:.2f}␣seconds\n"
        f"Baseline␣accuracy␣(most␣common␣label):␣{baseline_accuracy␣*␣100:.2f}%\n"
    )

    # Save the analysis results to a text file
    with open(os.path.join(output_dir, "analysis.txt"), "w", encoding='utf-8') as f:
        f.write(results)

def train_logistic_regression_with_cv(n_samples=5000):
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

    # Use only the first n_samples
    x_train = x_train[:n_samples]
    y_train = y_train[:n_samples]

    # Flatten the images
    x_train = x_train.reshape(x_train.shape[0], -1)
    x_test = x_test.reshape(x_test.shape[0], -1)

    # Scale the features
    scaler = StandardScaler()
    x_train = scaler.fit_transform(x_train)
    x_test = scaler.transform(x_test)

    # Create a logistic regression model
    model = LogisticRegression(max_iter=1000, multi_class='multinomial', solver='lbfgs')

    # Perform cross-validation
    scores = cross_val_score(model, x_train, y_train, cv=5, scoring='accuracy')

    # Print the mean and standard deviation of the cross-validation scores
    with open("../output/cross_validation_results.txt", "w", encoding='utf-8') as f:
        f.write(f'Cross-Validation␣Accuracy:␣{scores.mean():.4f}␣␣{scores.std():.4f}\n')

    model.fit(x_train, y_train)
    test_accuracy = model.score(x_test, y_test)
    with open("../output/cross_validation_results.txt", "a", encoding='utf-8') as f:
        f.write(f'Test␣Accuracy:␣{test_accuracy:.4f}\n')
```